Sistemi operativi 20/02/17

Che cosa sono i sistemi operativi? Un sistema operativo non è niente altro che un pezzo di software in realtà è un grande pezzo di software più complesso che l'uomo abbia mai organizzato. Si pone tra due mondi completamente diversi e per certi versi incompatibili tra loro. Da un lato si trova l'hardware (componenti che hanno problematiche per quanto riguarda la gestione, al loro sviluppo efficiente) dall'altra parte abbiamo gli utenti che sono i programmatori. Da una parte il sistema operativo deve far fronte alla parte hardware e gestirlo di conseguenza dall'altra parte si interfaccia con il programmatore e lo deve gestire di conseguenza quindi si trova a cavallo tra due mondi. Al programmatore deve garantire un utilizzo efficiente delle risorse hardware senza che il programmatore si debba preoccupare di come questo utilizzo debba essere svolto. Quindi il sistema operativo lo possiamo vedere composto da 2 livelli: al livello più basso si interfaccia con l'hardware ed ha una prima astrazione dell'hardware, quindi il compito dell'"hardware-specific software and device driver" è quello di trasformare tutti l'hardware e tutti i dispositivi in dispositivi a livello superiore differenti più facili da trattare in modo da offrire un'interfaccia a livello superiore di sistema operativo in modo che sia semplificato rispetto l'hardware. E poi abbiamo un livello superiore, livello che implementa tutte le funzionalità principali ai sistemi operativi legate alle politiche di gestione (da come gestire le comunicazioni sulla rete a come gestire la memorizzazione delle informazioni su disco a come gestire l'esecuzione di più programmi contemporaneamente sul processore e via discorrendo). In altri termini nel livello superiore si implementano le politiche di gestione. Nel livello inferiore si implementano i (? 34.56) di gestione. Sotto questo profilo il sistema operativo ha 3 ruoli: il ruolo di arbitro, di illusionista e di colla. Arbitro perché sopra il sistema operativo ci sono i programmatori tipicamente più di uno anche nel caso in cui il computer è ad uso personale utilizzo diversi programmi sviluppati da diversi programmatori e li utilizzo contemporaneamente. Tali programmi utilizzano le risorse di un singolo computer fisico tutti contemporaneamente. D'altra parte esiste un'unica tastiera, un unico schermo, un unico processore ma utilizzo il computer in modo multitasking. Il problema sorge quando tutte le applicazioni vogliono utilizzare le stesse risorse ed è qui che entra in gioco il sistema operativo nel ruolo di arbitro infatti decide quando un singolo programma può e deve utilizzare una singola risorsa. Il suo secondo ruolo è quello di illusionista perché in questa ottica il sistema operativo da l'illusione ad ogni singola applicazione di aver a disposizione un intero computer tutto per sé. È proprio su questa base che si legge il sistema operativo e si legge il multitasking. Ogni programma può essere scritto immaginando di avere un computer interamente a propria disposizione ignorando che ci sono tanti altri programmi scritti da programmatori che possono essere eseguiti contemporaneamente. Quando abbiamo scritto il primo programma c non abbiamo mai pensato che più programmi potessero essere eseguiti contemporaneamente questo perché il sistema operativo ha giocato bene il ruolo da illusionista. Il lavoro di colla è il lavoro più "infame" che il sistema operativo può fare ed è il lavoro che comporta il 90 % del codice sviluppato all'interno del sistema operativo. Che cosa è questa colla? Nel caso in cui noi volessimo installare windows 10 avremo giga e giga di roba da installare di cui all'interno troveremo codice per l'agevolazione del programmatore questo codice riguarda: la posizione, codice per la versione cinese etc… se vedessimo il computer finalizzato soltanto allo svolgimento di un compito per cui il computer e il sistema operativo deve mettere in condizione di eseguire un programma in maniera efficiente e corretta, la maggior parte del codice che c'è in quel sistema operativo in realtà non serve. Se noi pensassimo che tale programma deve essere svolto su più computer in possesso da più utenti in diverse condizioni allora in quel caso possiamo notare che all'interno del codice del sistema operativo vi è un supporto proprio per questo. Questo codice di

Fabio Catinella – Niccolò Cardelli – Andrea Tosti – Emanuele Torrisi - 20 Febbraio 2017

colla che sostanzialmente serve per tenere tutto assieme in realtà è un codice semplice da capire e quindi non lo tratteremo ma ci concentreremo sugli altri due compiti del sistema operativo ovvero il ruolo di illusionista e quello di arbitro che sono gli aspetti critici (ovvero quelli che rendono un sistema operativo effettivamente un sistema operativo affidabile ed utilizzabile) che però comportano il 10% del codice e invece ignoriamo tutto il resto. La colla tuttavia rappresenta un ruolo importante perché è grazie a questo aspetto che poi il sistema operativo viene venduto(?) e poi utilizzato. Questi ruoli tuttavia possono essere riscontrati in molti altri aspetti dell'informatica come il cloud computing (parola per indicare tutta una serie di sistemi nei quali la memorizzazione e l'elaborazione può avvenire su dei server di rete) (44.53). Anche nei servizi web abbiamo problematiche simili noi possiamo aprire con un browser diversi tab per collegarci a diversi siti contemporaneamente. Più utenti si possono connettere contemporaneamente allo stesso server web e quindi il server web riceverà tantissime richieste da tantissimi utenti sparpagliati per il mondo. Nessuno ha conoscenza di cosa stanno chiedendo gli altri nessun utente sa che ci sono altri utenti che fanno richieste, il server web deve mettere a disposizione risorse per gestire le richieste di tutti quanti (quindi deve fare da referee e allo stesso modo deve fare da illusionista perché il server web in realtà risponde ad un unico indirizzo ma non è mica vero che dietro ci sia un unico server con un unico indirizzo IP. Potrebbero esserci server sparpagliati per il mondo. Infatti quando noi digitiamo www.google.com in realtà non sappiamo a quale server ci stiamo collegando. Può essere un server in Gran Bretagna o in California etc… . Quindi anche qui c'è da fare un lavoro di illusionismo ovvero mascherare più server che svolgono quell'unico compito sotto un unico punto di accesso. Un altro esempio è un data base multi utente oppure internet. Un esempio più concreto del web services è il server web. Il server web è una cosa concettualmente semplice. Come funziona un server web? Abbiamo un cliente quindi un broswer e questo utente fa un'operazione di "get" che chiede al server web di andare a leggere un file da qualche parte in un suo disco. Questa get arriva al server, il server va a cercare in un suo disco il file, lo trova, lo legge e poi lo spedisce indietro. Però possiamo avere tantissimi utenti che contemporaneamente fanno la get e dobbiamo gestirli tutti, alcune richieste possono comportare anche calcoli (come ad esempio la ricerca di google che di per se rappresenta un calcolo di quello che noi stiamo cercando). Ci sono server web che all'interno nascondono database oppure nascondono veri e propri motori di calcolo. Ci sono altre complicazioni alla gestione all'efficienza del server web; spesso per questioni di efficienza molti server web fanno uso pesantissimo di cache ma il bello è che queste cache possono essere presenti non solo nel web ma anche in punti intermedi e sparpagliati su internet, per cui se qualcuno sta leggendo lo stesso giornale che sto leggendo io non è detto che la mia richiesta debba arrivare al server ma la risposta potrebbe ritornarmi indietro molto prima perché potrebbe esserci un server intermedio o un proxy che ha le stesse informazioni che ha memorizzato per un utente in precedenza. Un'ulteriore complicazione è dovuta al fatto che i server potrebbero inviare non soltanto dati o risultati di computazione indietro ai client ma potrebbero inviare anche script per personalizzare le pagine. Abbiamo altre complicazioni ovvero il server web potrebbe essere aggiornato oppure cliente e server potrebbero avere velocità differenti quindi non è detto che le richieste che arrivano dai clienti possano corrispondere alla velocità del server e d'altra parte il server non può sovraccaricare il cliente di informazioni, dovranno in qualche modo sincronizzarsi. Quindi in una cosa apparentemente facile come il server web ci sono una miriade di complicazioni legate a questioni di efficienza di praticità di generalità… e tutto questo ha fatto si che i server root siano diventati oggetti complessissimi. Tutte queste complicazioni ci sono anche nei sistemi operativi nella stessa forma. Ci sono quindi "problemi sfida" legati all'affidabilità, vorrei far si che il sistema sia affidabile cioè che

funzioni per quanto più tempo possibile e che non faccia reboot frequenti per esempio. Ci sono problemi legati alla sicurezza cioè voglio che le informazioni siano protette da un utilizzo scorretto o malizioso. Il problema della portabilità è un altro problema enorme. Che cos'è la portabilità? Il sistema operativo si pone a cavallo tra l'hardware e le applicazioni il problema è che di hardware ne esiste tantissimo abbiamo processori completamente differente, schede di rete, dispositivi vari, etc, in realtà ogni computer ha una scelta infatti ci sarà un tipo di processore un particolare tipo di scheda grafica, un particolare disco… d'altra parte ci sono molti altri computer simili a questo che possono cambiare processore o scheda di rete etc… ora se progetto un sistema operativo direttamente per una macchina lo vendo solo a quelle persone che hanno quella macchina specifica in realtà ho un grosso problema di portabilità nel sistema operativo stesso. Devo creare un sistema operativo che sia portabile su diverse macchine. Quindi abbiamo un primo problema di portabilità legato all'hardware e poi abbiamo anche un secondo problema di portabilità dovuto al programmatore. Se un programmatore sviluppa un programma per un particolare sistema operativo devo assicurare che tale programma valga anche per gli altri sistemi operativi. Poi abbiamo sfide legate alle prestazioni di diversa natura e spesso contrastanti (56.43) come il problema di latenza: vorrei che su ogni singola richiesta le risposte arrivino in tempi rapidi e non voglio che l'utilizzo da più utenti della stessa macchina introduca più ritardo di quanto non sia necessario, vorrei che il sistema operativo abbia un vasto overhead quindi introduca lui stesso poco carico sulle risorse hardware se le risorse le impiego per il sistema operativo vorrei che le stesse non venissero usate da altre parti. Vorrei garantire la predicibilità perché vorrei che ogni operazione di complessità paragonabile impieghino lo stesso tempo. Tutti i requisiti visti fino ad ora sono requisiti funzionali e non funzionali e in qualche modo sono sotto il controllo del programmatore. Ci sono tuttavia degli elementi che sono al di fuori del controllo del progettatore di sistemi operativi come il problema di sapere quanta gente, dopo che avrò sviluppato il mio sistema operativo (spesa non indifferente) lo comprerà, tale elemento non è nelle mani del progettatore di sistemi operativi ma risiede nelle mani di chi fa marketing.

Ci sono due filosofie che stanno prendendo piede ed è un pò l'approccio di avere un sistema proprietario rispetto ad un sistema aperto con un hardware proprietario rispetto ad uno aperto. Ad esempio il mondo apple è sviluppato su un hardware molto controllato, il sistema operativo è chiuso rispetto l'hardware e quindi si eliminano molti problemi che altrimenti si avrebbero, il sistema operativo è più sicuro e il sistema operativo è più facile da sviluppare d'altra parte l'hardware veniva a costare anche di più perché eri costretto ad immedesimarti in quello stile di vita. L'hardware aperto invece è estremo in questo senso infatti vi è una compatibilità hardware molto maggiore tuttavia questo comporta avere più problemi nel sviluppare il sistema operativo. Stabilire quale dei due è migliore non è facile stabilirlo in realtà sono due filosofie completamente diverse e non si può dire quale è migliore. Questo per fare capire che in realtà i problemi non sono dovuti sono al comparto tecnico ma sono dovuti anche a molti altri fattori che non vanno sottovalutati.

Storia dei sistemi operativi

I Sistemi operativi moderni nascono negli anni 60 sulle ceneri di multics (progetto fallito) e msv (progetto diffuso all'epoca) in realtà le basi degli attuali sistemi operativi sono proprio di quel periodo. Prima degli anni 60 vivevano in varie forme ma davvero troppo elementari per essere considerati i progenitori degli attuali sistemi operativi invece nei sistemi operativi di multics e msv c'erano molti concetti che ritroviamo oggi come il multitasking. Negli anni 70 da msv abbiamo

un'evoluzione verso vms mentre dall'esperienza di multics è nato unix. Verso il finire degli anni 70 vi è stata un'altra grossa rivoluzione perché prima i sistemi operativi erano per grossi computer mainframe, computer che costavano tantissimo e ve ne erano pochi per centri di calcoli sul fine degli anni 70 è nato il concetto di personal computer su idea dell'ibm, cioè volevano che ognuno avesse un computer per uso personale. Quest'idea era ritenuta "pazza" ma alla fine fu messa in pratica e si sono sviluppati tantissimi computer con sistemi operativi diversi . Il sistema operativo più diffuso all'epoca era l'MS-DOS sviluppato dall'ibm per i personal computer, parallelamente nello stesso periodo abbiamo il sistema operativo per apple. Qual era l'elemento di innovazione ma in realtà di conservazione/arretramento dei personal computer? Mentre i mainframe erano macchine che dovevano gestire diversi utenti che avevano centinaia di programmi in funzione contemporaneamente l'idea dei progettisti dei personal computer era quella che un personal computer per una persona dovesse svolgere una operazione alla volta non c'è nessun motivo del perché una persona su un personal computer voglia avviare due programmi contemporaneamente. Quindi in un unico balzo si è tornato indietro di 20 anni. Quindi alla fine degli anni 70 c'erano due approcci: computer di fascia alta: mainframe (computer multi utente e multi applicazione) e computer di fascia personale mono utente e mono applicazione. Quindi dall' MS-DOS si è passati, negli anni 80, alla versione a finestre copiando l'approccio della apple. Ma cosa era questo sistema a finestre? Non era niente altro che un'interfaccia grafica nei confronti di un sistema operativo vecchio stile quindi quello che in realtà faceva aprendo più finestre era offrire la possibilità di eseguire più applicazioni contemporaneamente. Ma il sistema operativo sottostante non aveva la capacità di eseguire più programmi contemporaneamente quindi tale operazione non era possibile. Microsoft quindi di conseguenza voleva progettare un sistema operativo nuovo che permettesse di usare il multitasking ma i tentativi fatti soprattutto negli anni 90 non hanno avuto successo per via dell'adozione. Sul sistema operativo Windows funzionavano bene i giochi infatti la maggior parte degli utenti lo usava per giocare, anche se vi erano persone che usassero anche applicazioni professionali. Questo comportò la nascita di due sistemi operativi uno moderno ed uno antico completamente differenti ed incompatibili. Soltanto negli 2000 sono riusciti ad unificare questi due sistemi nel senso che una branca si estinta e oggi Windows 10 ne rappresenta l'esistenza. Nel creare questo sistema operativo Windows si era resa conto che doveva partire da 0 per questo motivo ha assunto in blocco gli sviluppatori di vms e gli ha dato praticamente quasi carta bianca chiedendo di sviluppare un sistema operativo prendendo il meglio che c'era senza problemi di compatibilità con le altre versioni precedenti o importandosene in modo marginale. In parallelo si sono sviluppate altre storie ed in particolare quelle di multics. Multics era un grosso progetto che con l'hardware dell'epoca volevano sviluppare un sistema operativo ambizioso che avesse una capacità di gestire un'intera città come numero utenti. E doveva venire incontro a diverse esigenze che non riuscendo a soddisfarle tutte il progetto è fallito. Un gruppo di queste persone che lavorava per multics dall'oggi al domani si ritrovò senza lavoro avendo computer avanzatissimi che non potevano utilizzare perché il progetto multics era fallito. Decisero così di creare un nuovo sistema operativo derivante e simile a multics e di eliminare tutti quegli ostacoli che ne hanno impedito lo sviluppo finale. Da qui nacque Unix. Il nome per lo appunto deriva da multics (che stava ad individuare un sistema operativo che doveva fare molte cose) perché il sistema operativo che volevano creare doveva essere fatto bene e svolgere una sola cosa (per questo Unix). Hanno avuto un'idea geniale ovvero svilupparlo in un linguaggio ad alto livello. Era un'idea innovativa perché al tempo i sistemi operativi venivano sviluppati con il linguaggio macchina a bassissimo livello. Per sviluppare Unix hanno dovuto inventare un nuovo linguaggio ovvero il C. Sono passati 47 anni e dentro uno

smartphone vi è ancora una versione di Unix a dispetto del fatto che diciamo che il mondo vada velocissimo. Se viene ancora usato voleva dire che il sistema operativo che era stato creato era geniale e ben fatto tuttavia se potessimo vedere il codice ci accorgeremo che è completamente diverso da quello di Unix di 47 anni fa ma le idee di implementazione sono le medesime. Unix ha avuto un problema di adozione perché è nato come "figlio di nessuno" quindi il gruppo di sviluppo decise di dare questo sistema operativo all'università dalla quale poi sono nati diversi gruppi di studenti che erano diventati esperti di quel sistema visto che era gratuito ed accessibile da chiunque. Questo ha fatto si che tutte le principali aziende richiedessero di Unix perché c'erano persone che sapevano utilizzarlo. Tutto ciò ha generato una guerra tra le varie aziende perchè ognuna voleva una versione personalizzata di Unix ed incompatibile con le altre ma tutte le aziende non ne chiedevano una qualsiasi ma volevano la vera ed unica versione originale di Unix. Tutto questo è sfociato in una guerra con tanto di tribunale e si è concluso con l'interesse degli utenti finali che avevano problemi ad avere versione di Unix tutte differenti ed incompatibili l'una con l'altra. Non c'era più portabilità tra vari sistemi e questo generava problemi. Finalmente poi si ebbe uno standard per le varie versioni di Unix e si decise di accomunare le varie versione con alcune specifiche comuni a tutti i sistemi Unix, si sono concentrati sull'interfaccia offerta ai programmatori ed hanno definito lo standard Posix. Lo standard Posix definisce non come debba essere scritto un sistema operativo ma definisce come il sistema operativo deve offrire la propria interfaccia ai programmatori lasciando poi libertà alle singole aziende produttive di sviluppare Unix a proprio piacimento e secondo le proprie esigenze. Lo standard Posix è tutt'ora valido ed è quello che usiamo a laboratorio. Da un certo punto di vista questo ha permesso la nascita di Linux. Linux è un nucleo di un sistema operativo che è stato sviluppato in Unix sviluppato da uno studente finlandese per gioco. Successivamente questo nucleo è entrato in contatto con il sistema gnu e le due cose si sono fusi dando vita a gnu-linux che attualmente conosciuto come Linux ed è alla base di tutte le versioni Linux standard attuali ed è la base sulla quale poi è stato sviluppato android. Infine un'altra storia altrettanto interessante è la storia del mac osx che per certi veris è comune a windows e ms-dos. Con le stesse problematiche e caratteristiche era nato come sistema operativo per uso personale e anche all'interno di quel mondo è stata necessaria la sua evoluzione. Nel caso di mac os questo è avvenuto con mac osx che è stato la fusione del mondo di unix con l'interfaccia grafica di mac os e da questo è nato mac osx e poi in seguito ios. Un grande vantaggio di ios è che dentro vi è Unix e non Linux quindi abbiamo a disposizione tutti gli strumenti di Unix per poter lavorare. Nel caso di windows non è compatibile con lo standard posix per cui non ci sono gli stessi strumenti. Un'altra cosa per mettere in evidenza la problematica dei sistemi operativi riguarda l'hardware la cosa deve far pensare perché se da un lato Unix nelle sue idee e nel suo progetto di massima è rimasto inalterato fino ad oggi, l'hardware non è rimasto inalterato ed ha avuto uno sviluppo enorme. Si è passati dall'inizio degli anni 80 con una macchina tipica con le caratteristiche che vedete nella prima colonna (vedi slide) le caratteristiche di questa macchina sono di un server di discreta potenza quindi con una velocità del processore di un mips (capacità di eseguire un milione di operazioni al secondo) il costo dei mips era nell'ordine di 100 mila dollari quindi nel caso delle macchine da 2 mips dovevamo spendere 200 mila dollari con caratteristiche del genere: memoria ram 128 k il disco 10 mb la connessione ad internet 9,8 al secondo, connessione di rete 3 mbit al secondo ma condivisi. La connessione internet prima era un unico cavo a cui si attaccavano più server e questo cavo aveva la capacità di trasportare 3 mbit al secondo ma erano condivisi a tutti e soli i server collegati a questo cavo e il numero medio degli utenti collegati era un centinaio queste erano le caratteristiche di un server di buona fascia, intorno agli anni 85-86 erano diventate le caratteristiche di un personal

computer e quindi nel giro di 5 anni avevamo speso 100 mila dollari per una macchina del genere ma con un paio di milioni delle vecchie lire si poteva acquistare la stessa macchina qualche anno più tardi. Se ci spostiamo una quindicina di anni verso la metà degli anni 90 un personal computer di buone caratteristiche andava sui 300 mips con un costo di 30 dollari per mips, 128 mega di ram, 4 gb di disco connessione di rete di 256 kbit al secondo connessione di rete di 10 mbit al secondo ad uso singolo invece che condivisi e con un utente per macchina. Se ci spostiamo al 2011 una macchina equivalente va 10 mila mips il costo medio è di 50 centesimi per mips la memoria fisica è dell'ordine di 10 gb il disco di 1 tera connessione ad internet dell'ordine di 5 mbit al secondo connessione di rete di 1 mbit al secondo utenti per macchina: minore di 1 ovvero che un singolo utente di macchine del genere ne ha 4 o 5 . perché abbiamo visto questo passaggio? Abbiamo visto che l'hardware è cambiato in maniera drammatica in realtà i tempi dell'hardware sono estremamente veloci. D'altra parte quando si sviluppa un sistema operativo lo si può sviluppare per 5 anni perché i tempi di progettazione e di sviluppo e quindi i costi sono estremamente più alti dei costi dell'hardware. Forse negli anni 50 e 60 non era così ma ora si. Quindi chi sviluppa un sistema operativo ora non può pensare di cambiarlo tra 3 anni e neanche tra 5. Un sistema operativo per avere qualche profitto deve vivere 10 o 20 anni e questo era già vero negli anni 70 80 e 90. Quindi in realtà chi progettava quei sistemi operativi sapeva di questo ed era ben conscio che dovrà tenere conto di una evoluzione tecnologica dell'hardware che è difficilmente prevedibile ma che è molto forte per cui bisogna essere davvero molto provvidenti. E non ci si può limitare bisogna prevedere quali saranno le necessità da qui a 10 20 30 anni che gli utenti potrebbero avere. Immaginate di dover pensare ad un sistema operativo che soddisfi le esigenze delle persone tra 20 o 30 anni questa è una sfida ulteriore che si pone nei confronti di chi deve progettare e fino ora questa sfida è stata raccolta. se voi prendete windows che progetta un sistema operativo ogni 3 anni in realtà se andate a vedere bene non produce un sistema operativo nuovo. Nel suo nucleo, nella sua parte fondamentale è rimasto lo stesso negli ultimi 15 20 anni quello che cambia è la colla. Cambia l'interfaccia, gli elementi grafici, può cambiare qualche dettaglio ma il nucleo è rimasto lo stesso quello che usiamo noi adesso è rimasto lo stesso dagli anni 90. Stesso discorso per unix o linux. Le innovazioni vengono introdotte ma quando sono effettivamente molto solide e a quel punto si introducono gradualmente.

Rivediamo ora degli aspetti tecnici dei sistemi operativi e della loro evoluzione. Per capire perché i sistemi operativi si sono sviluppati in una certa maniera bisogna capire le esigenze dell'epoca nella quale si sono sviluppati e poi delle epoche successive. I primi computer erano estremamente costosi per cui l'obiettivo principale di chi doveva sviluppare un sistema operativo e lo doveva utilizzare era di massimizzarne l'uso. Se io spendo 10 milioni di euro per comprare un computer e so che tra 3 o 4 anni quel computer sarà da buttar via perché uscirà qualcosa di più potente e non sarà più adatto alle mie esigenze il mio obiettivo primario è di usarlo il più possibile in questi 3 anni perché soltanto così saranno stati ammortizzati i costi cioè li ripagherò in base all'uso. Quindi l'obiettivo dei sistemi operativi dell'epoca era quello di garantire la massima utilizzazione del sistema. Con il tempo il mondo è cambiato radicalmente infatti se guardo all'utilizzo del computer medio nel particolare il processore è meno dell'1%. Avere avuto un tasso di utilizzo per un computer di quell'epoca sarebbe stato un disastro, era come buttare i soldi fuori dalla finestra. Una seconda caratteristica è che si eseguiva un programma alla volta. Un solo programma veniva messo in esecuzione e restava in funzione fino a quando non terminava e quando terminava arrivava un altro programmatore che faceva girare il proprio programma e così via. Quindi l'esecuzione dei programmi era seriale. In

queste condizioni il sistema operativo non aveva gran che da fare, doveva fornire soltanto gli elementi necessari per: 1) caricare un programma 2) metterlo in esecuzione 3) aspettare la terminazione. Eventualmente se si riteneva che il programma fosse andato in un loop infinito esisteva la possibilità di interromperlo via hardware e quindi il sistema operativo tornava operativo e dava la possibilità di caricare il prossimo programma. Questa era la funzione principale del sistema operativo e l'altra funzione era quella di automatizzare l'utilizzo di alcuni dispositivi hardware quindi in queste condizioni il sistema operativo era molto diverso da quello che conosciamo oggi era sostanzialmente un insieme di librerie, di funzioni che permettevano il facile utilizzo dell'hardware e permettevano il caricamento di un programma. Gli utenti all'epoca non avevano la possibilità di interagire con il programma mentre era in funzione. L'utente caricava il programma e dati di cui aveva bisogno aspettava a quel punto che il programma terminasse. A quel punto quando il programma terminava stampava i risultati in qualche forma generalmente su carta e l'utente poteva avere i risultati. Durante l'esecuzione non c'era modo di interagire in nessun modo e quindi bisognava aspettare la terminazione del programma. In questo mondo si sono sviluppati i sistemi detti batch che avevano proprio queste caratteristiche. Quindi in un sistema batch i programmi vengono portati all'inizio vengono ordinati e vengono messi in esecuzione uno alla volta, ogni programma batch si chiama "job", un job accomuna codice da eseguire con dati su cui il codice deve essere eseguito. Tutti i dati di cui il codice ha bisogno devono essere caricati quando il programma viene caricato. Come funzionavano queste macchine all'epoca? Il programmatore tipicamente un ricercatore, scriveva il proprio programma su delle schede perforate, aveva una macchina da scrivere all'interno della quale digitava spesso in codice binario poi successivamente in fortran, incideva sulle schede perforate il proprio codice dopo di che decideva su quali dati il codice doveva essere eseguito e tutto questo produceva un pacco di schede perforate di una dimensione abbastanza consistente. Non aveva modo di testarlo, si caricava le schede perforate in spalla ed andava al centro di calcolo. A Pisa c'era un centro di calcolo dell'IBM, chi ci ha lavorato scriveva programmi in linguistica computazionale, dopo che portavano queste schede perforate al centro di calcolo c'era un tecnico che controllava se le schede erano state scritte correttamente e nel caso in cui fossero state scritte bene ovvero non presentavano errori elementari (perché non controllava direttamente il codice) stabiliva lui in che ordine dovevano essere eseguite. In un periodo successivo con macchine più evolute caricava queste schede perforate, venivano caricate sul disco e poi da li c'era uno schedulatore che decideva quando mandare il codice in esecuzione. Quindi il programmatore portava il pacco di schede, poteva sapere quando venivano messe in esecuzione e poi doveva aspettare il termine dell'esecuzione. Ma che succedeva se il programma non terminava? Visto che non potevano sprecare tempo prezioso il programmatore doveva fare anche una stima del tempo di esecuzione del programma: contava le istruzioni di linguaggio macchina, per ogni istruzione contava quanto tempo ci voleva su quel determinato processore, faceva una stima accurata dei tempi di esecuzione e se per caso quel programma girava più tempo del previsto veniva bloccato. Bloccandolo veniva stampato un "vamp" della memoria cioè per ogni unità della memoria veniva stampato il contenuto, questo veniva dato al programmatore che tornato a casa faceva un debug su questo vamp. Quindi cercava di capire l'errore e quando lo trovava, riscriveva il pacco di schede perforate e le riportava al centro di calcolo. Il modello di esecuzione di questo sistema si chiama "single task" che significa che veniva messo in esecuzione il primo job e seguiva fino a terminazione. A quel punto veniva caricato un secondo job e seguiva fino a terminazione e via dicendo. Ora però che succedeva? Che il primo job doveva svolgere un po di calcoli sul processore poi doveva leggere dei dati da un pacco di schede perforate o dal disco, per cui doveva impegnarsi

in un'operazione di input output. Durante questa operazione di input-output per il processore non era possibile fare altro finché non ha letto tutti i dati che gli servono da dispositivo sul processore non si svolge alcuna attività solo fino a quando non abbiamo completato la lettura dei dati a quel punto il processore svolge dell'altro calcolo, poi deve fare una stampa e quindi il processore rimane inutilizzato per un certo tempo fintanto che l'operazione sul dispositivo non sia completata a questo punto torna in esecuzione il job e quando ha finito l'esecuzione termina. Era uno spreco di processore enorme. Spendevamo 700 mila dollari solo per fare un input/output. Posso fare qualcosa per recuperare quei soldi? Nei primi sistemi batch lo schema era quello delle schede perforate ma già dagli anni 50 il modello diventava questo: il pacco di schede perforate poteva essere letto in anticipo, passando dalla memoria veniva messo nel disco quando il job passava l'esecuzione interagiva con il disco per prendere i dati o per scriverli e quando aveva finito stampava i risultati dalla stampante. In questo modello potevamo andare a caricare diversi job all'interno del disco in anticipo. Il fatto di avere più job a disposizione ha introdotto una soluzione al problema precedente chiaramente in questo contesto il job + 1 non può fare niente se è bloccato in un'operazione di input/output ma il processore che è libero lo potrei utilizzare per iniziare il compito di p2 che non ha bisogno di dispositivi. Per fare questo però p1 e p2 devono essere entrambi in memoria principale altrimenti non posso eseguire p2 quando p1 si interrompe. Quindi dai sistemi batch mono programmati si è passato ai sistemi batch multi programmati all'interno dei quali (il sistema operativo doveva essere scritto interamente rispetto a prima) si prevedeva la gestione della memoria in modo più complessa. Una parte della memoria era riservata al sistema operativo ma il resto della memoria era ripartita tra i vari programmi con i rispettivi dati. In questa maniera quando veniva messo in esecuzione il programma p1, quando il programma p1 si doveva interrompere per eseguire un'operazione di IO, aveva a disposizione il programma p2 da poter far eseguire su processore. Quindi si è passati da un modello single task ad un modello multi task dove eseguo p1 quando p1 si interrompe eseguo p2 quando p2 si interrompe e p1 è ancora impegnato con Input/output posso eseguire p3 a questo punto quando anche p3 si interrompe sui dispositivi ammesso che in questo caso tutti e 3 i job richiedano usi di dispositivi differenti, in questo caso non ho altri programmi da mandare in esecuzione e quindi sul processore non posso farci niente resta inutilizzato nel momento nel quale p1 completa la sua operazione su dispositivo posso rimettere in esecuzione p1 e via dicendo. Allora in questo caso ho il vantaggio di usare maggiormente il processore e tutto questo si materializza in tempi di latenza molto più brevi perché tutti i job si completano all'instante 12 invece che terminare l'ultimo all'istante 27. Quindi con una soluzione che riguarda il progetto del sistema operativo con un minimo di supporto hardware sono passato da avere un sistema lento ad avere un tempo molto più veloce che è in grado di fare lo stesso lavoro in meno della metà del tempo

Sistemi operativi 23/02/17

La scorsa lezione abbiamo visto il passaggio da singletask a multitask. Questo passaggio è stato necessario per rendere più efficace l'utilizzo dell'hardware sostanzialmente all'epoca si era in una situazione nella quale i computer erano estremamente costosi, l'organizzazione dei sistemi operativi era a task singolo quindi esecuzione dei task in maniera strettamente sequenziale abbiamo visto che non permetteva un uso efficiente del processore che era la parte più costosa del sistema e per questo motivo è stato introdotto un meccanismo multitasking per cui teniamo in memoria più programmi contemporaneamente e non appena il processore si rende disponibile perché il programma che era nell'esecuzione precedente si deve interrompere per eseguire un'operazione di IO, riassegniamo il processore ad un altro programma in questo modo il processore rimane sempre in utilizzo, rimane sempre qualcosa da fare e in questa maniera riusciamo a ridurre i tempi di esecuzione dei programmi perché tagliamo i tempi morti, ogni singolo programma impiega sempre lo stesso tempo ad eseguire dal momento in cui viene caricato sul processore al momento in cui termina ma la differenza è che a causa di questa maggiore efficienza per unità di tempo siamo in grado di terminare e completare un numero maggiore di programmi. Se le condizioni non cambiano ovvero i computer rimangono così costosi e il nostro obiettivo è quello di ottimizzare il tempo del processore questo tipo di soluzione è pienamente soddisfacente non c'è bisogno di inventarsi altro se non che il mondo è cambiato perché sono state introdotte tutta una serie di innovazioni oltre che sulla velocità del processore e sui suoi costi sono state introdotte innovazioni relativamente all'interfaccia verso gli utenti, sono stati introdotti i video terminali quindi la possibilità per gli utenti di inserire informazioni tramite tastiera e ricevere informazioni tramite il monitor quindi progressivamente e contemporaneamente ad un calo di costi dell'hardware si è aperta una possibilità per l'utente di interagire con i propri programmi durante l'esecuzione quindi si è passati in una nuova fase dove in realtà non c'era da ottimizzare più soltanto il tempo del processore ma c'era in qualche modo da ottimizzare anche il tempo utente. Quindi passiamo da un modello di più programmi dove l'obiettivo era quello di ottimizzare il processore ad un modello nel quale abbiamo più programmi in esecuzione sul sistema ma dobbiamo cercare di dare una risposta ad ognuno più velocemente possibile e via via che questa tendenza quindi abbassamento di costi del processore e aumento dell'interattività diventava sempre più forte diventava sempre più importante ottimizzare il tempo dell'utente quindi garantire all'utente non soltanto una terminazione prima possibile ma anche una interazione comoda. Per questo motivo si è passati ad un modello mutitasking di questo genere ad un modello time sharing. Qual è la differenza? Nel modello time sharing il processore può essere riassegnato ad un altro programma anche se il programma in esecuzione non si è bloccato per un processo di IO quindi il processore viene riassegnato a prescindere con un principio di rotazione. per cui ogni programma viene assegnato un quanto di tempo di esecuzione sul processore questo quanto è fisso uguale per tutti, quando il processore ha eseguito quel programma per quel quanto di tempo quindi quando il quanto di tempo è scaduto il processore viene assegnato ad un altro programma e poi ad un altro ed ad un altro quando li abbiamo completati tutti si ricomincia dal primo. Cosa cambia con questo tipo di organizzazione? Ogni programma verrà comunque eseguito come se in realtà gli altri non ci fossero quindi dal punto di vista del singolo programma cambia poco cambiano sicuramente i tempi di esecuzione perché nelle esecuzione del programma 1 ogni tanto dovrò interrompermi per eseguire gli altri. Quello che cambiano sono 2 aspetti 1 è che tutti i programmi avanzano quindi evolvono in modo unico nel tempo come se il programma avesse a disposizione avesse a disposizione un proprio computer con un proprio processore però un po' più lento di quello originale. Quindi il sistema operativo sta agendo da illusionista dà l'illusione ad ogni programma di avere il proprio processore. questa illusione viene fatta avendo un solo processore fisico che in realtà esegue materialmente i compiti ma ripartendo il tempo di questo processore fisico tra i processori fittizi tra i processori virtuali. In questo modo ogni utente ha l'impressione che il suo programma evolva con una sua velocità ma che evolva in maniera piuttosto costante perché queste interruzioni sono relativamente piccole quindi non c'è la percezione di una vera e propria interruzione come poteva esserci nel caso precedente di multitasking. Il secondo aspetto è che con questo tipo di organizzazione in realtà il sistema operativo diventa più lento perché dovrò gestire queste interruzioni e per farlo, togliere un programma dall'esecuzione, metterne un altro sul processore fisico

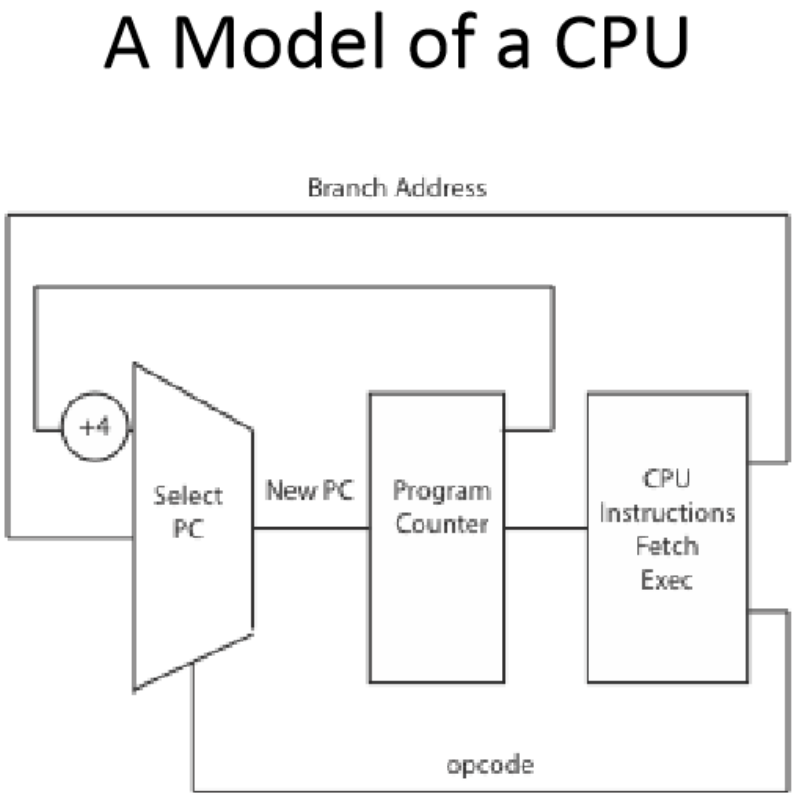
comporta del lavoro aggiuntivo che deve svolgere il sistema operativo e che non è in gratis. Quindi in realtà sto perdendo tempo di calcolo e quindi sto perdendo soldi perché il tempo del processore stavolta è impiegato anche per operazioni di servizio quindi sto barattando un costo del processore in cambio di un sistema che è in grado di essere più iterativo nei confronti degli utenti perché un utente non dovrà come ad esempio in questo caso aspettare molto a lungo la terminazione del programma p3 o l'interruzione del programma p3 per poter dare una risposta su utente in questo caso l'interazione ci può essere perché seppur un po più lentamente ma in maniera continuativa tutti i programmi vanno avanti e quindi ogni programma avrà la possibilità di interagire con il proprio utente entro tempi più brevi. Tutto questo ha un senso perché il costo del processore si è abbassato quindi non è un problema spendere una parte del tempo del processore per fare questa rotazione e d'altro canto ho un grosso vantaggio perché perdo meno tempo degli utenti quindi gli utenti risparmiano più tempo. Questa evoluzione è quella che si avuta verso la metà degli anni 60, fine anni 60 quando è nato Unix. I sistemi che utilizziamo attualmente non hanno esattamente questo meccanismo ma hanno un'evoluzione di questo meccanismo dove l'elemento base quindi la rotazione tra tutti i programmi in esecuzione resta pari pari. Ci sono alcuni meccanismi in più per rendere il sistema più efficiente che vedremo in seguito. Ora qual è la situazione attuale? I sistemi time sharing sono stati introdotti in un momento nel quale i costi si stavano abbassando, la situazione attuale è quella che ormai la risorsa processore non costa quasi più niente un processore anche molto potente si può comprare anche per pochi dollari. Quindi la situazione attuale è che possiamo mettere processori ovunque e in effetti siamo circondati da computer dappertutto (smathphone, tablet, laptop) che significa questo da un punto di vista del sistema operativo? Significa che mentre negli anni settanta dovevamo progettare un sistema operativo per macchine più o meno simili macchine ragionanevolmente potenti che dovevano gestire più utenti, stavolta invece vi trovate a sviluppare un sistema operativo che può finire in un cellulare, in un laptop, server sul cloud, datacenter e via discorrendo. Ora un sistema operativo che deve andare in un cellulare da quelli di un datacenter hanno dei requisiti un po differenti perché girano su computer con caratteristiche e potenzialità completamente differenti quindi si progetta un sistema operativo per ogni computer? Si, esistono proposte di avere sistemi operativi specifici per certi domini applicativi ma in realtà tendenza più forte è quella di uniformare i sistemi operativi quindi in un microonde potrebbe esserci lo stesso sistema operativo che google opera per eseguire i calcolo per le nostre ricerche. Questo richiede molto sforzo nella progettazione per venire incontro le esigenze di tutti. Questa è la situzione attuale ma quella futura come sarà? Chi progetta un sistema operativo non deve soltanto catturare soltanto la situazione attuale ma dovrà andare incontro anche a quello che succederà negli anni a venire. Negli anni a venire i datacenter diventano sempre più grossi e potenti, i numeri di processori per computer tende ad aumentare, tende ad incrementare il numero di computer per utente questo perché potendo mettere computer su più oggetti questo vuol dire che ogni utente avrà molti computer con i quali rapportarsi che dovranno ordinarsi tra loro, dovranno interagire e svolgere compiti comuni e c'è la tendenza ad avere sistemi di memorizzazione su scala sempre più grande, queste sono delle tendenze che possiamo osservare già ora e che ci possiamo aspettare ragionevolmente per il futuro e queste sono quelle che determineranno le prossime evoluzioni dei sistemi operativi. Rifacendo un passo indietro in ogni caso quando progettiamo un sistema operativo dobbiamo tenere conto che potrà essere utilizzato da un unico utente o potrà essere utilizzato da più utenti sicuramente in entrambi i casi su quel sistema ci saranno tanti programmi in esecuzione piò o meno contemporaneamente che saranno tra loro in competizione per l'utilizzo delle risorse. I programmi hanno bisogno della memoria principale, avranno bisogno di utilizzare il disco hanno bisogno di usare il processore e poi dovranno usare lo schermo il mouse la tastiera e via dicendo. Tutte queste risorse sono gestite dal sistema operativo che dovrà assegnare di volta in una volta un programma o altro a seconda delle sue necessità. Qual è il modo giusto di assegnare le risorse? Qual è il modo di stabilire che ad un certo programma dò una certa quantità di memoria, dò un certo tempo al processore, gli attribuisco un certo spazio nel disco? Tenendo conto che i programmi possono avere caratteristiche, necessità, situazione contestuali completamente differenti quindi devo rapportare tutto questo. In realtà non c'è una sola risposta sintetica che si possa dare infatti la risposta è tutto il corso che affronteremo. Quindi nel corso di tutte le lezioni che vedremo vedremo la risposta a tutte queste domande.

I sistemi operativi sostanzialmente si costruisce sopra un corso di architettura perché studiamo il modo in cui possiamo gestire in modo efficiente l'architettura, l'hardware per fornire dei servizi a chi sta sopra quindi al programmatore o all'utente. In ogni caso alcuni aspetti vanno ripresi per vederli nella nostra prospettiva, molte cose viste dal lato architetture adesso a sistemi operativi è il momento di acquisire un altro punto di vista dell'hardware e guardandolo dall'alto ovvero guardandolo dal punto di vista di ciò che ci offre e che ci può offrire. Quindi questa prima di questa lezione serve per mettere quei contenuti sotto una prospettiva diversa.

Come è fatto un computer? Abbiamo a disposizione uno o più processori, abbiamo a disposizione una memoria principale, un certo numero di dispositivi e poi dei meccanismi che sono i meccanismi delle interruzioni ed il meccanismo di accesso diretto alla memoria. Il processore è il cuore, è l'elemento che esegue le istruzioni quindi quando scriviamo un programma in alto livello questo viene tradotto in istruzioni elementari che materialmente vengono svolte ed eseguite dal processore. Per poter svolgere queste operazioni ha bisogno di memoria. La memoria serve per contenere le istruzioni, il processore le deve poter leggere per poi eseguirle e deve utilizzare la memoria per conservare i risultati intermedi dei suoi calcoli. Poi i dispositivi che servono per ricevere informazioni dal mondo esterno o per inviare informazioni al mondo esterno quindi rendono il computer un elemento integrato con quello che c'è al di fuori. I dispositivi nei computer moderni sono computer essi stessi hanno un alto grado di indipendenza hanno infatti un loro processore una loro memoria. I gestori dei dispositivi da un lato sono dei computer che gestiscono i dispositivi dall'altro lato si devono rapportare con il processore perché tutte le informazioni lette tramite dispositivo, ad esempio il mouse, alla fine devono arrivare ad un processore che sta eseguendo il nostro programma per poter essere interpretate. I dati digitati sulla tastiera non servono al gestore della tastiera in quanto tale il gestore della tastiera prende le informazioni digitate sulla tastiera e le deve passare ad un processore quindi è evidente che mi serve un meccanismo per permettere al processore di interagire correttamente con il gestore di dispositivi. Su tutto questo si sviluppa un piccolo problema di sincronia, il processore ha dei suoi (?) (23.17) ed esegue in maniera sincrona il programma. Il gestore del dispositivo invece è legato ai tempi del dispositivo. Quindi il gestore del dispositivo legge il tasto quando io essere umano premo sulla tastiera non quando un programma ha deciso che io devo premere sulla tastiera. Quindi il gestore del dispositivo riceve le informazioni dall'esterno in maniera totalmente asincrona con l'esecuzione dei programmi. Quindi avremo un bisogno di gestire questa differenza tra un processore che in maniera sincrona con un suo ritmo esegue un programma e un gestore dei dispositivi che in modo totalmente asincrono con dei ritmi e tempi completamente diversi riceve input dall'esterno e deve passare questo input al processore. Questo meccanismo di sincronizzazione tra dispositivi si sviluppa tramite questi due meccanismi: meccanismo delle interruzioni e il meccanismo delle memory access. Ad esempio tra adesso tra un paio di ore andrò a casa e dovrò cucinare la pasta quindi metterò l'acqua a bollire e sarò impegnato in una serie di attività legate a questo particolare compito, in maniera completamente indipendente e asincrona potrebbe arrivare una telefonata, la telefonata arriva sotto forma di interrupt, il telefono squilla e mi avvisa che c'è informazione che proviene da qualche altra parte del resto del mondo per me. D'altra parte io sono impegnato in un'attività che ha i suoi tempi ed è indipendente dalla telefonata, per cui mi trovo in una situazione dovrò gestire questa interruzione. Posso mollare tutto li e andare a rispondere al telefono quindi in qualche maniera gestire questa interruzione e prenderne atto, ricevere delle informazioni quando tutti questo si è interrotto chiudo la telefonata e ritorno al mio compito precedente come se non fosse successo niente dal punto di vista dell'azione cucinare. Oppure sono in un momento particolarmente delicato e non posso correre al telefono buttando tutto via e allora decido di post-porre il minimo indispensabile la risposta alla chiamata fintanto che non ho lasciato il compito che sto svolgendo in situazioni di sicurezza in maniera da poterlo riprendere in un instante successivo senza danno, senza averne alterato il senso complessivo. Questo tipo di interazione è la stessa che si sviluppa tra dispositivi e processore nel momento nel quale arriva un interrupt. Il gestore del dispositivo riceve un dato da tastiera lo riconosce e mette un interrupt al processore, il processore se può interrompere ciò che sta facendo, riceve l'interrupt e lo gestisce oppure post-pone leggermente il

riconoscimento dell'interruzione e appena può riconosce l'interruzione, la gestisce e prende il dato. Questo è il meccanismo delle interruzione, per quanto riguardo il meccanismo degli accessi diretti alla memoria ci sono tutta una serie di cose che i dispositivi possono fare in maniera indipendente. Per esempio se sto scolando la pasta e arriva una telefonata dico a mio figlio di rispondere e lui gestisce la telefonata e poi mi riporta a parole il contenuto. Questo possibile approccio succede con le memory access in realtà il gestore del dispositivo è in grado di svolgere un compito ancora più sofisticato e prendere le informazioni e memorizzarle direttamente in memoria senza che io materialmente debba andare a prenderle. Quindi in questo modo quando ho una telefonata c'è qualcuno che me la porta direttamente in memoria mentre sto svolgendo in realtà un altro compito. Le interruzioni ci serviranno tantissimo però in pratica questo viene gestito dal sistema operativo. Per quanto riguarda il processore ci sono alcuni elementi che ci interessano e sono: i registri generali e i registri stato. I registri generali non sono altro che variabili che il processore utilizza per conservare risultati temporali delle proprie elaborazioni, variabili molto semplici, elementari. I registri di stato sono invece registri speciali che hanno un significato particolare per il processore e sono quelli che gli permettano di svolgere correttamente il suo compito. In particolare ce ne sono 3 particolarmente importati per noi: c'è un registro che contiene il contatore di programma, instraction pointer, un valore che chiamare. Il contatore di programma contiene l'indirizzo della prossima istruzione da eseguire. Il processore è una fabbrica che non si ferma mai, appena completata l'esecuzione di una istruzione va subito ad eseguire l'istruzione successiva ma la deve leggere dalla memoria quindi deve sapere dove si trova. Per legge per sapere dove si trova utilizza il contatore di programma. Il contatore di programma in ogni istante punta alla prossima istruzione da eseguire. Un altro registro molto importante per noi è lo stack pointer. Lo stack è una struttura dati utilizzatissima in informatica, utilizzata nei linguaggi di programmazione e la utilizzano tantissimo i sistemi operativi e le architetture. Lo stack pointer che ne sono presenti diversi in qualunque sistema operativo è un'area di memorizzazione temporanea sulla quale l'accesso è particolarmente efficiente e veloce che è caratterizzata da un utilizzo a stack per lo appunto. Lo stack si utilizza durante l'invocazione di funzione/procedura. Per cui quando invochiamo una procedura in cima allo stack viene messo il blocco di archivazione della procedura, se poi questa procedura ne invoca un'altra, in cima allo stack viene aggiunto un altro blocco di archiviazione della funzione e poi un'altra ancora e così via. È uno stack perché quando la funzione più interna che è stata invocata a termine si toglie quel blocco di attivazione e si passa al blocco di attivazione precendente. Quindi in realtà in realtà lo stack si utilizza in maniera incrementale e poi decrementale. Si riempie e si svuota seguendo sempre lo stesso ordine. Quando arriva un'interruzione, è una situazione molto simile per certi versi ad un'invocazione di funzione. Interrompo quello che sto facendo e passo all'esecuzione di un'altra funzione. Questa cosa richiede la memorizzazione di un po' di informazioni in cima allo stack. Quindi nella gestione delle interruzioni (e poi in altre circostanze) lo stack è fondamentale. Quindi il processore deve mantenere il puntatore allo stack attivo in maniera tale da poterlo utilizzare quando arriva un'interruzione o succedono altri eventi e per questo motivo lo stack pointer è un registro speciale. Poi abbiamo un altro registro speciale che è il "program status register" e prende tantissimi nomi a seconda del processore / hardware etc… questo registro di stato non è altro che una maschera di bit cioè un registro per cui ogni singolo bit ha un proprio significato indipendente dagli altri bit. Un certo numero di bit rappresentano dei codici di condizione quindi ci dicono i risultati nell'ultima operazione eseguita, se c'è stata una divisione per zero etc… poi ci sono un paio di bit per lo meno che sono particolarmente critici e dei quali parleremo oggi più avanti. Un bit rappresenta la modalità del processore se il processore opera in stato utente o supervisore e un altro bit che mi dice se le interruzioni sono abilitate o disabilitate. Questo bit è utile perché nel caso della cottura della pasta se sto scolando la pasta non posso ricevere una telefonata in quell'istante e quindi in questo caso disabilito il bit di interruzione per permettere di completare un'operazione delicata, quando ho finito l'operazione delicata posso riattivare il bit di interruzione e posso ricevere e rispondere alle interruzioni che mi arrivano. Chiaramente se io disabilito il bit delle interruzioni troppo a lungo poi dopo rischio di perdere telefonate o altri eventi che accadono quindi il processore rischia di non poter servire le richieste da parte dei dispositivi quindi normalmente questo bit deve essere attivo e può essere disabilitato occasionalmente altrimenti abbiamo un problema. Come funziona il processore? il

processore funziona con un ciclo di estrazione ed esecuzione. Il processore terminata l'esecuzione di un'istruzione ne esegue subito un'altra. In effetti il processore stesso esegue delle istruzioni che sono scritte nel suo firmware. Queste istruzioni definiscono il ciclo di estrazione ed esecuzione che è formata in questa maniera: per prima cosa il processore controlla se ci sono interruzioni pendenti nel caso in cui ci fossero e le interruzioni sono abilitate allora gestisce l'interruzione in una qualche maniera. Se non ci sono inturrezioni pendenti prende il program counter lo spedisce alla memoria e si fa restituire dalla memoria il contenuto di quelle celle corrispondenti al program counter all'interno di queste celle trova l' istruzione da eseguire quindi carica questa l'istruzione all'interno di un suo registro, la esegue dopo di che o contemporaneamente all'esecuzione incrementa il program counter per poter andare a puntare all'istruzione successiva. Di default le istruzioni sono messe tutte di fila una dietro l'altra esattamente come le istruzioni del nostro codice C sono tutte sequenziali. Quindi in seguito ad un'istruzione la successiva si trova nell'indirizzo successivo. Quindi il program counter viene automaticamente incremenetato dal processore. perché viene incrementato di 4? perché stiamo ipotizzando che un'istruzione occupi 4 byte. Fatto questo si completa il ciclo di estrazione ed esecuzione e si ricomincia d'accapo. Quindi il processore nuovamente riprende a testare l'interruzioni se non ci sono interruzioni carica l'istruzione, l'esegue e continua così. Quindi è vitale che in ogni istante il program counter punti sempre ad un'istruzione sensata, se per caso il program counter punta ad un'istruzione insensata il processore rischia di andare in uno stato inconsistente. Possiamo rappresentare il processore con un diagramma a blocchi.

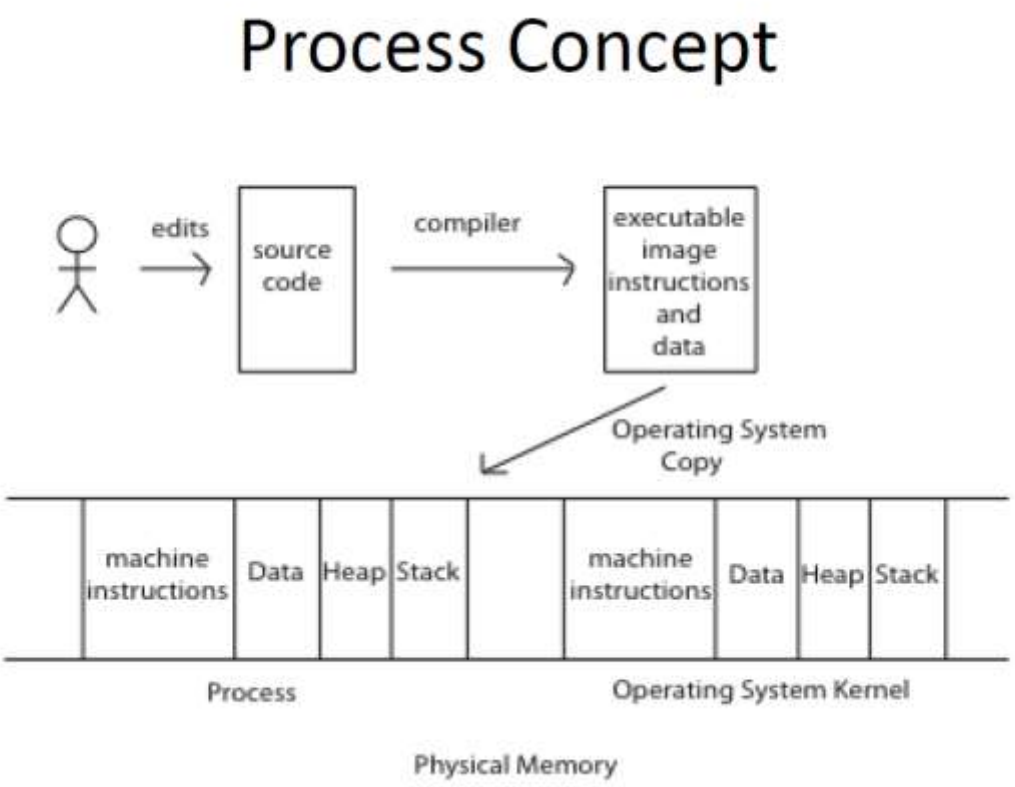


Per cui questo è il blocco che carica ed esegue l'istruzione (CPU instructions fetch exec). Questo blocco prende il valore del program counter esegue l'instruzione contenuta in memoria all'indirizzo corrispondente al program counter, quando ho completato l'esecuzione che cosa fa? L'istruzione può essere di tipi differenti: può essere un'istruzione di salto quindi che modifica che modifica il program counter andando a spostarlo sull'indirizzo che non è il successivo oppure può essere un altro tipo di istruzione che quindi non ha l'effetto di alterare il program counter. Se l'instruzione è di salto il risultato di questa istruzione è un nuovo valore da attribuire al program counter. Se invece l'istruzione non è di salto allora il program counter viene aumentato in maniera automatica di 4 e il valore del program counter va all'ingresso di questo selettore dopo di che si stabilisce se utilizzare il nuovo valore del program counter come calcolato dall'istruzione di salto oppure se

utilizzare come nuovo valore del program counter quello incrementato di 4. Se l'istruzione non è di salto allora si mette il nuovo valore all'interno del registro program counter. C'è da fare una considerazione, in questo modello di processore questo modulo va ad eseguire va ad eseguire un'istruzione puntata dal program counter e fa qualsiasi cosa quest'istruzione gli dica. Se l'istruzione gli dice vai in memoria a quest'indirizzo, modificala, scrivila quel modulo di esecuzione dell'istruzione fa esattamente quello che c'è scritto nell'istruzione né più né meno. Immaginate di avere un modello di processori di questo tipo, riportato in un sistema operativo che fa time sharing dove abbiamo un sistema operativo multi tasking dove abbiamo tanti programmi caricati in memoria tutti indipendenti l'uno dall'altro e sono tutti differenti allora questo potenzialmente apre un grosso problema perché se quest'istruzione che se sto eseguendo è un'istruzione di un certo programma A ma in questa istruzione c'è scritto "vai a modificare una zona di memoria che è di proprietà che è stata assegnata ad un programma B" voi "fate le scarpe" al programma B, lo possiamo danneggiare, possiamo perderne il contenuto, modificare il risultato in maniera scorretta. Quindi si pone un grosso problema di protezione che in realtà non si pone necessariamente così gratuitamente all'architettura. Quel processore che abbiamo visto prima deve eseguire istruzioni ma calato all'interno di un sistema operativo che deve gestire più programmi contemporaneamente quel modello di processore apre un problema relativo alla protezione. Perché nasce il problema? Potrebbe capitare che un codice sia bacato, ad esempio va in segmentation fault ma cosa succede quando abbiamo un segmentation fault? Accedeva alla memoria che non doveva accedere, generalmente la segmentation fault era dovuto ad un errore dovuto con i puntatori perché mettevamo un indirizzo completamente rotto che va in una locazione arbitraria dove c'è niente dentro un puntatore. In realtà nel modello di processore che abbiamo visto la segmentation fault non può esistere perché qualsiasi operazione facciamo è sempre corretta. Quindi il segmentation fault lo prendiamo perché nel computer non abbiamo il processore visto poco fa, abbiamo processori che si sono evoluti per venire incontro alle esigenze dei consumatori che richiedevano caratteristiche differenti. Ci serve qualche meccanismo di protezione perché il codice che eseguiamo potrebbe avere dei bug, ci potrebbero essere degli errori fatti dal programmatore che possono causare problemi sia al programma stesso sia agli altri pogrammi sia al sistema operativo in maniera totalmente involontaria. Oppure ci possono essere dei programmi scritti apposta per fare danno agli altri. Questo non è soltanto un problema che nasce in un contesto specifico di programmi scritti da noi ma ormai nasce in una valanga di contesti, possiamo scaricare programmi da internet mandarli in esecuzione sul nostro sistema operativo etc. quindi la protezione è vitale, ci serve assolutamente, in realtà nei primissimi computer che erano sigletask dove si eseguiva un programma alla volta, la protezione non mi interessava realmente perché se per caso quel programma aveva un errore, faceva danno a se stesso e comunque sia l'operatore che gestiva il sistema, quando passava il tempo previsto di esecuzione per quel programma lo terminava e ne caricava un altro. Quindi in un contesto mono programmato la protezione non è un problema, non esiste dopo, potrei completamente fregarmene. Non è un caso che i primi pc (personal computer) che erano tornati indietro di 20 anni per quanto riguarda i sistemi operativi e utilizzavo un sistema operativo (l'MS-DOS che era mono programmato) avevano dei processori che non facevano protezione e il sistema operativo non faceva protezione perché si eseguiva un programma alla volta e se andava male l'utente spegneva il personal computer, lo riavviava e ripartiva nuovamente. Ora ci serve la protezione quindi ci serve imporre i vincoli con i quali le istruzioni sono eseguite dal ciclo "fetch and execute" del processore. come li impongo questi vincoli? Un modo semplice è quello di adottare lo schema di Java. Nello schema di Java quando mando in esecuzione un programma in realtà quel programma viene eseguito in una macchina virtuale cioè ho un programma che non è quello che sto mandando in esecuzione ed ho l'interprete. ogni programma scaricato, preso dagli utenti e riprogrammato… in realtà viene eseguito all'interno dell'interprete. Quest'interprete carica l'istruzione e l'esegue e la esegue solo se quest'istruzione dal punto di vista della protezione è sicura, non crea problemi. Se per caso quell'istruzione che vuole eseguire non è permessa o cerca di fare qualcosa che non è permesso allora blocco semplicemente il programma. Questo è il modello utilizzato da Javascript ed è un modello usatissimo nella sua semplicità in tanti contesti. Cosa deve fare il sistema operativo? Deve creare un processore virtuale sopra un processore fisico e su questo processore virtuale implementare l'interprete software di ogni istruzione scritta dai

programmatori. Questa è una possibilità ma introduce un overhead non da poco. È possibile invece eseguire del codice direttamente sul processore ma avendo la garanzia della protezione? Ovviamente se il processore, se l'hardware non è stato progettato per aiutarci in questo non si può fare, quindi c'è bisogno che il processore sia organizzato diversamente. Questa esigenza è nata con i primi sistemi operativi multitasking ed è stata percepita dai progettisti dell'hardware che hanno dovuto comunque sottostare a dei requisiti troppo forti per essere rifiutati quindi hanno iniziato a riprogettare il loro processori in maniera tale che facessero protezione in maniera nativa con dei meccanismi inglobati direttamente nel processore. Però la protezione non è una cosa che può essere affidata soltanto al processore, è una cosa che riguarda il processore e il sistema operativo. L'intero sistema è protetto grazie ad una serie di meccanismi e politiche che sono implementate in parte nell'hardware dal processore in parte nel software dal sistema operativo. Per vedere come sono fatti questi meccanismi di protezione bisogna introdurre 3 concetti che sono: il concetto di processo, il concetto di dual mode del processore (modalità kernel e modalità supervisore) ed il concetto di commutazione sicura da modalità utente a modalità supervisore. Il concetto di processo è una astrazione introdotta a software dal sistema operativo quindi i processi non sono elementi concreti, sono elementi astratti realizzati dai sistemi operativi per permettere lo sviluppo di politiche di protezione e per usare correttamente i meccanismi di protezione che invece ci mette a disposizione il processore. Questi meccanismi di protezione sono: la modalità dual mode user/kernel e la modalità di commutazione automatica tra questi due nodi. Nella modalità kernel mode (o modalità supervisore) il processore opera in modalità privilegiata quindi nella modalità kernel mode tutto è possibile, qualsiasi istruzione può essere eseguita. Mentre in modalità utente (user mode) ci sono dei limiti, quindi non tutte le istruzioni possono essere eseguite e ci sono anche dei limiti allo spazio di azione di queste istruzioni. Quindi nella user mode si eseguono istruzioni con possibilità ridotte, in kernel mode il processore esegue istruzioni con privilegi completi. Questo meccanismo deve essere realizzato a firmware. L'altra cosa che ci serve è avere un meccanismo corretto per poter passare dall'uno all'altro, se il processore è in user mode, può eseguire tutte le istruzioni che rientrano all'interno dello spazio di azione della user mode quindi che non violino la protezione, però ogni tanto sarà necessario eseguire delle istruzioni più libere al di fuori di questo spazio vincolato, per fare questo bisogna passare in kernel mode. Come faccio a passare in kernel mode da user mode? Non posso realizzare un'istruzione assembler utilizzabile dall'utente dove l'utente dice "ok ora passo in kernel mode" perché altrimenti avendo un programma qualsiasi invece di eseguirlo in user mode lo eseguo in kernel mode e quello che succede succede. Quindi questo passaggio da user mode a kernel mode non è un passaggio che può essere richiesto dal programmatore e quindi dal programma. Deve avvenire soltanto in certe condizioni, c'è da capire quali sono queste condizioni e c'è da capire come si fa a sviluppare questo passaggio. Stesso discorso al contrario, se sono in modalità kernel mode come faccio a passare in user mode? Quindi mi serve un meccanismo per poter passare liberamente in queste due modalità. La modalità kernel mode o user mode alla fine della storia è un bit all'interno del processore, quindi per dire al processore che è in kernel mode basta settare un bit, per dire che il processore è in user mode basta resettare quel bit, questo attiva e disattiva funzioni di protezioni all'interno del processore.

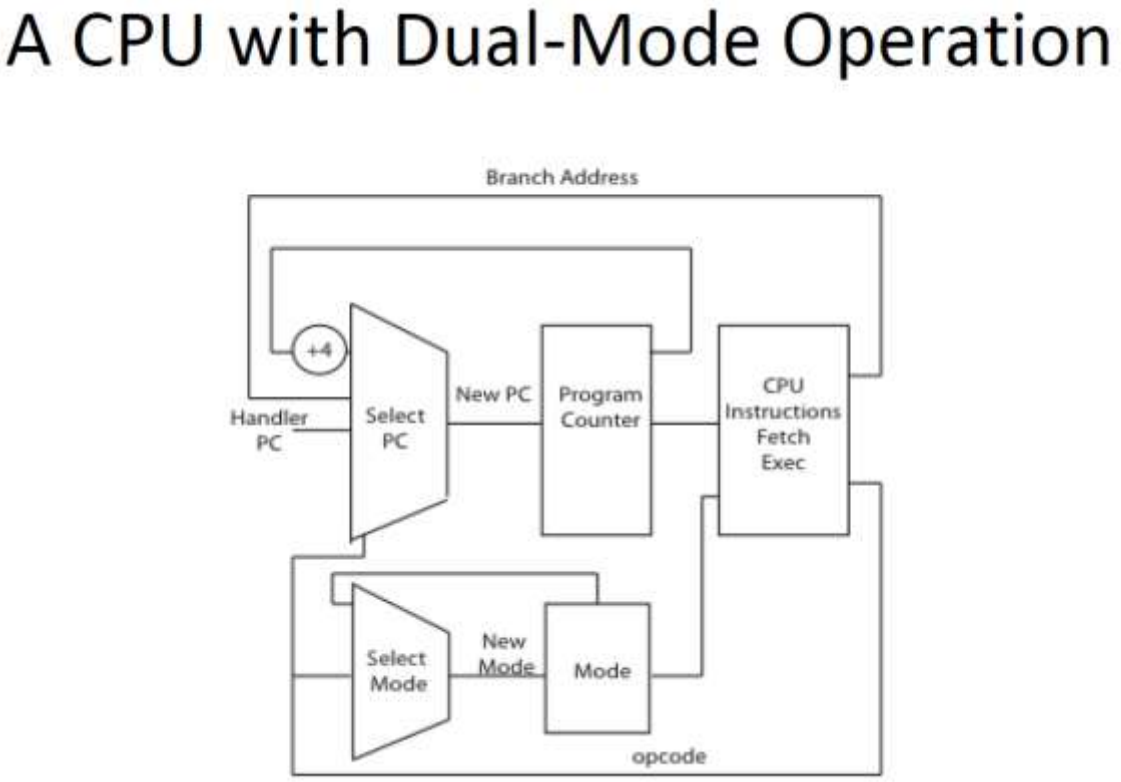
Processo: scriviamo un programma in un linguaggio ad alto livello, lo compiliamo quindi traduciamo questo programma in un programma scritto in linguaggio macchina, a questo punto questo programma viene preso e caricato in memoria principale per essere eseguito. Quindi siamo in una situazione nella quale il nostro programma viene preso, messo in memoria principale, all'interno della memoria mettiamo le istruzioni e poi riserviamo dello spazio per contenere i dati, lo heap (zona di memoria allocata dinamicamente) e poi uno stack. Il problema è che la memoria è unica, da qualche parte sarà presente il nucleo sistema operativo che conterrà il codice del sistema operativo, i dati del sistema operativo, anche lui avrà il suo heap, i suoi stack (sono più di uno) e via discorrendo. Quello che facciamo a questo punto una volta caricato il nostro programma in memoria principale, lo mettiamo in esecuzione ma dobbiamo garantire che tutte le istruzioni che lui esegue, restino confinate all'interno di questo spazio e non vadano fuori.



Per questo motivo è stato definito il concetto di processo. Che cos'è un processo? È una sequenza di attività corrispondenti ad un programma, quindi specificate in un programma, che viene eseguito su un processore con privilegi limitati. Ci sono diversi elementi del processo in particolare questa è una astrazione non è un elemento concreto. nei sistemi operativi ogni volta che viene fatta un'estrazione in realtà bisogna fare due cose: bisogna creare delle strutture dati che rappresentano questa astrazione e bisogna sviluppare delle funzioni che permettono di maniporarla. Nel caso dei processi in particolare servono diversi elementi tra cui un "process control block (PCB)" che è una struttura dati che descrive il processo e che contiene tutte le informazioni associate al processo. Il process control block viene conservato all'interno una struttura dati del nucleo del sistema operativo all'interno della quella vi sono tutti i PCB di tutti i processi presenti nel sistema ed ogni processo ha poi al suo interno due elementi: un trherad ed un spazio di indirizzamento. Il thread sarebbe l'esecuzione di una sequenza di istruzioni quindi una sottoattività. Nel caso più semplice il processo è single thread quindi l'attività svolta dal processo è in realtà è la stessa attività svolta dal thread. Nei casi un po più sofisticati il processo può essere formato da più thread quindi l'attività svolita dal processo in realtà è l'unione delle attivitò svolte dai singoli thread. Ogni thread svolge la sua attività la sua attività andando ad eseguire un pezzo di programma che è il programma che è stato caricato in memoria quando abbiamo definito il processo. Lo spazio di indirizzamento invece è un insieme di diritti di quel processo ed in particolare questo spazio di indirizzamento stabilisce quali sono i limiti allo spazio di memoria che il processo può utilizzare. In qualche maniera tramite il processo noi possiamo delimitare uno spazio di esecuzione di un pezzo di codice. Senza il processo, ci mancherebbe lo spazio di esecuzione quindi ci mancherebbero dei confini all'esecuzione di un programma e ricadremmo ai sistemi delle origini dove appunto questi confini non esistevano e qualsiasi programma poteva fare qualsiasi cosa. Che differenza c'è tra programma e processo? Un programma è un'entità totalmente statica, un programma è una serie di istruzioni che stanno ferme fisse ed immobili. Un programma non produce nessun risultato, posso scrivere un programma per scrivere la radice quadrata di un numero ma il programma non mi da la radice quadrata di 4. Il processo viceversa è la sequenza di azioni che si sviluppano nell'esecuzione di un programma quindi se voglio calcolare la radice quadrata di 4, devo avere un processo che esegue il programma radice quadrata che prende in ingresso il

numero 4 e mi restituisce 2. Quindi il processo è un'entità dinamica mentre il programma è un'entità statica. Sullo stesso programma posso attivare più processi, ogni processo esegue lo stesso programma su dati differenti e produce risultati differenti per questo motivo lo stato interno di questi processi sarà anche differente. Che cosa è il process control block? È una struttura dati che serve per descrivere il processo, il processo è un'entità astratta realizzata dal sistema operativo ed è realizzato tramite una struttura dati che è il process control block ed una serie di funzioni per manipolare i processi. Il process control block è fondamentale senza il quale non avremo processo. Che cosa contiene questa struttura dati? Contiene informazioni legate ai processi. Quali? Il nome del processo (numero nel caso di UNIX), dei puntatori ai thread del processo (quindi agli esecutori delle sotto attività, quelle normalmente svolgono le attività), riferimenti alla zona di memoria associata al processo e può contenere altre informazioni di altre risorse acquisite dal processo. Il process control block contiene una descrizione del processo dal punto di vista delle risorse che quel processo ha acquisito ed ha in uso e dal punto di vista delle attività, quindi thread, che quel processo sta eseguendo. Tramite le informazioni che abbiamo tramite il process control block legato al processo possiamo attivare dei meccanismi di protezione che ne limitano lo spazio di azione. Questa astrazione dei processi è il primo elemento che ci serve per sviluppare le politiche di protezione, gli altri due elementi che ci servono sono la doppia modalità del processore che è un meccanismo hardware e poi un meccanismo di passaggio dalla modalità utente a supervisore. Per quanto riguarda questa modalità duale, abbiamo detto il processore deve operare in una di queste due modalità. Che cosa significano concretamente? In modalità supervisore il processore può eseguire delle istruzioni con privilegi pieni, questo vuol dire che queste istruzioni possono andare ad accedere liberamente alla memoria senza limiti, possono utilizzare i dispositivi liberamente quindi possono andare a leggere e scrivere informazioni sui dispositivi e di conseguenza possono anche gestire le interruzioni dei dispositivi, viceversa quando siamo in modalità utente operiamo con privilegi limitati che sono quelli garantiti dal nucleo del sistema operativo. Quali sono effettivamente questi diritti? Questo lo importa il sistema operativo e il sistema operativo e il processore rispetta questi limiti. In qualche maniera quando il sistema operativo decide che deve essere eseguito il processo A e quel processo può utilizzare può utilizzare la memoria a partire dalla locazione 100 fino alla locazione 1000, passa questa informazione all'hardware e l'hardware si assicura che quell'istruzione eseguite durante questo periodo si riferiscano soltanto agli indirizzi che avevamo ovvero da 100 a 1000. L'informazione sul fatto che il processore operi in modo supervisore o utente è conservata nel "program status registrer" quel registro speciale che contiene informazioni sullo stato del processore. Nel caso delle architetture x86 è conservato all'interno di un registro che si chiama EFLAGS, nel nostro corso è nella program statu register ed è un bit. Un bit sarà modalità utente/supervisore poi ci sarà un bit per interruzioni abilitate o disabilitate, poi ci saranno una serie di bit per impostare lo stato del processore dal punto di vista dell'ultima operazione eseguita. Come deve essere fatto un processore per questa doppia modalità? Il processore si complica un pochettino perché abbiamo aggiunto dei requisiti ulteriori. La parte superiore è rimasta uguale quello che cambia è la parte più

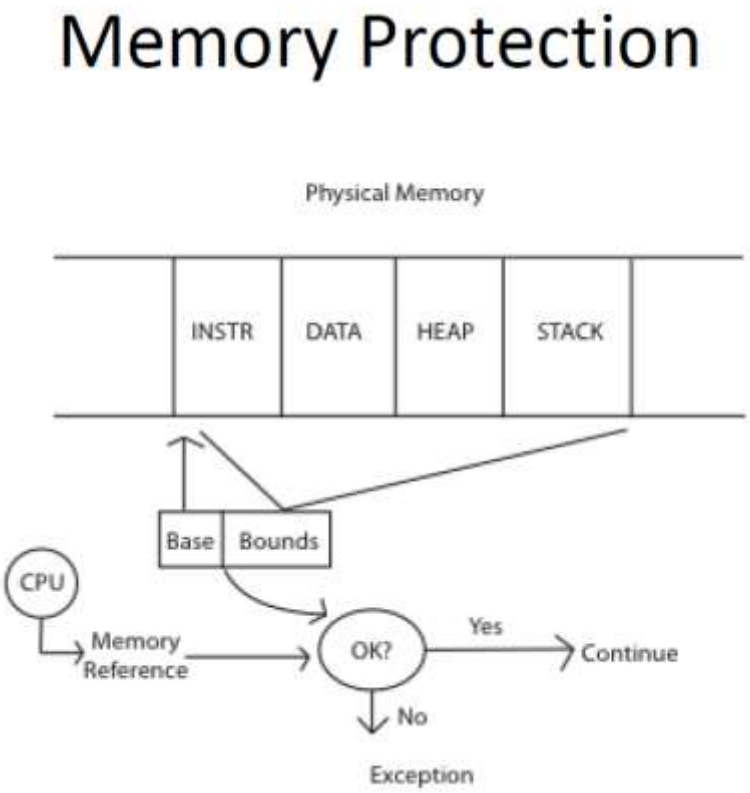
in basso che serve per gestire la modalità utente/supervisore e serve per gestire la commutazione dall'una all'altra. Quindi la modalità utente/supervisore è conservata in un registro ed è un bit. Questo bit viene



passato al blocco che fa la fetch and execute per verificare che l'istruzione rispetti i limiti imposti. Terminata l'istruzione bisogna aggiornare il valore di questo bit che può essere o il valore precedente se non abbiamo commutato modo oppure può essere il valore nuovo se abbiamo commutato modo. Quindi la commutazione avviene in concomitanza con l'esecuzione di alcune particolari istruzioni all'interno del processore o in concomitanza di alcune particolari situazioni per cui se queste situazioni si sono verificate o no, questo viene specificato dal ciclo di estrazione ed esecuzione della linea che va in input al selettore e dice se commutare la modalità o mantenere la modalità precedente. Queste informazioni sono molteplici e riguardano lo stato utente/supervisore altre invece riguardano le istruzioni di salto. C'è quest'altro elemento che si chiama "handler PC" che serve per gestire le interruzioni. Quindi per vedere le caratteristiche di questa doppia modalità dobbiamo vedere a questo punto una serie di elementi che si devono combinare per poterle implementare correttamente. La prima questione è quali sono le istruzioni privilegiate, quali sono le istruzioni che si possono eseguire in modalità kernel e che possono essere eseguite dal sistema operativo e invece quali sono le istruzioni non privilegiate che sono le uniche che possono essere eseguite dai programmi in modalità utente. L'altro aspetto è una volta che ho impostato in modo utente come posso imporre dei vincoli sullo spazio di memoria utilizzabile dalle istruzioni? Mi serve un meccanismo per questo. Tramite questi due aspetti (quali sono le istruzioni privilegiate e come si fa ad imporre dei limiti all'accesso in memoria) posso capire quali sono i supporti che mi dà l'hardware una volta che è in stato utente per imporre i limiti all'esecuzione delle istruzioni. Però tutto questo da solo non basta mi serve un altro meccanismo che è quello del timer perché occasionalmente avrò bisogno di riportare il sistema in modalità supervisore. L'idea è sostanzialmente è questa: se io mando in esecuzione un programma di un utente, quindi creo un processo che esegue un programma di un utente e lo mando in esecuzione di fatto sto assegnando il processore a questo processo, sto impostando il processore in modalità utente quindi sono sicuro che quel processo lavorerà con privilegi limitati e non potrà uscire dal proprio spazio assegnato. Il problema è che se mi limito a questo, il processo potrebbe rimanere in esecuzione in eterno e impedire l'esecuzione di altri processi. Quindi mi servirà per garantirmi mi servirà un meccanismo che mi permette di riacquisire il controllo del

sistema operativo, ritornare in modalità supervisore e interrompere quel processo e metterne in esecuzione un altro. Questo Timer lo sfrutterò abbondantemente per implementare quel meccanismo di time-sharing che abbiamo visto prima ma è anche una garanzia per evitare che un processo possa prendere il controllo del processore. In effetti è un meccanismo di protezione perché devo proteggere non soltanto lo spazio ma anche il tempo quindi la protezione, i privilegi riguardano sia lo spazio in tempo di memoria e di istruzioni sia il tempo cioè il tempo che ho a disposizione per poter utilizzare il processore anche quello deve avere un limite e per imporre una limitazione sul tempo mi serve un timer. Infine il meccanismo di commutazione da stato utente a supervisore deve essere sempre lì e verrà presentato alla fine. Dobbiamo vedere quali sono le istruzioni privilegiate ovvero cosa non posso far fare ad un processo utente e che posso fare soltanto dentro il sistema operativo? Ad esempio modificare liberamente lo spazio di memoria dove il sistema operativo è stato scritto e all'interno del quale vi sono i suoi dati rientra nella protezione della memoria e non nell'azione di un'istruzione privilegiata e quindi non è un esempio valido. La modifica esplicita dell'extraction pointer non è neanche un esempio valido infatti se io faccio un goto il parametro è un numero scritto dal programma scelto dal programmatore. Il goto è una modifica deliberata dell'extraction pointer o del program counter quindi in modalità utente posso modificare l'extraction pointer. Scrivere nel registro di una periferica non rappresenta un esempio valido: il registro della periferica sta all'interno del controller della periferica in un computer separato di fatto quindi tecnicamente difficile in realtà i registri di una periferica sono mappati in memoria e quindi vado a scrivere in una zona di memoria e di conseguenza accedo alla periferica per cui non c'è un'istruzione che agisce sulla periferica ma c'è un'istruzione che agisce in memoria ma le istruzioni che agiscono in memoria non sono privilegiate le posso fare liberamente di nuovo proteggere una periferica significa proteggere la memoria. l'allocazione di memoria è un'operazione molto più astratta. È un'operazione che facciamo quando vogliamo caricare un programma, creo un processo per contenere questo programma, lo metto in questa zona di memoria e qui faccio l'allocazione. E quindi non è un'istruzione di linguaggio macchina che allora la memoria è una operazione complessa del sistema operativo che trova uno spazio di memoria libera, decide che in quella zona di memoria ci carica il programma e si segna all'interno del descrittore del processo che la memoria allocata a quel processo è quello, non è un'istruzione di linguaggio macchina, non può essere un'istruzione privilegiata. L'abilitazione dell'interruzione non è neanche un esempio valido ma **disabilitare un'interruzione rappresenta un esempio valido** è critico perché devo imporre dei limiti sia nel tempo che nello spazio. Per imporre dei limiti nel tempo imposto un timer, quando il timer scade eseguo il nucleo del sistema operativo e poi lui metterà in esecuzione un altro processo ma questo timer è un dispositivo che genera un'interruzione quindi se disabilito l'interruzione, blocco il timer, acquisisco perennemente la proprietà del processore e poi non me lo toglie più nessuno. In aula alla fine della lezione scatta la campanella, so che è finita l'ora, deve entrare il professore successivo ma se la campanella non c'è il professore continua a tenere la porta chiusa e fa lezione fino a quando non moriamo tutti. Questo è un meccanismo di protezione con il timer e io non posso alterare la campanella, perché se altero la campanella non finisco mai la lezione. Non posso disabilitare un'interruzione, non posso dare questo strumento ai programmatori che scrivono programmi "spazio utente". Altre istruzioni privilegiate? In program status register è un registro alla mercé del programmatore. È un registro del processore che viene modificato dal ciclo di estrazione ed esecuzione ma quel registro non lo posso modificare liberamente. Quel registro contiene il bit di disabilitazione e riabilitazione delle istruzioni e contiene il bit di stato utente e supervisore, quel bit non lo posso modificare. Quindi non posso avere un'istruzione che mi modifica in stato utente quel bit. In alcune architetture sono previste per agire sull'IO per andare a leggere e scrivere sui dispositivi, queste istruzioni sono presenti allora sono privilegiate, se invece i dispositivi sono mappati in memoria allora non c'è bisogno di rendere privilegiate perché vale il meccanismo di protezione della memoria. Se un processo esegue un'istruzione privilegiata in user mode cosa deve succedere? Una possibilità è che posso in user mode ridurre il set di istruzioni per cui quelle istruzioni non sono proprio rappresentabili. Per quanto riguarda il blocco non è una operazione fattibile per l'impossibilità di bloccare il processore. Il processore deve continuamente eseguire istruzioni. Quello che si fa nella pratica è: se un processo in stato utente esegue un'istruzione privilegiata, il processore se ne accorge, solleva un'interruzione che si chiama

eccezione in questo caso e questa eccezione causa l'esecuzione del sistema operativo, il sistema operativo a questo punto analizza la situazione, verifica cosa è successo e tipicamente stampa a schermo un segmentation fault e abortisce il processo. Questo riguarda le istruzioni privilegiate e cosa bisogna fare quando si esegue un'istruzione privilegiata quando non ne abbiamo il diritto e questo è il primo aspetto. Il secondo aspetto riguarda la protezione della memoria. il modello che vedremo è un modello vecchio perché la seconda parte del corso approfondirà la protezione della memoria. lo schema è il seguente: questo programma viene eseguito da un processo e questo programma sarà caricato in memoria a partire da un



indirizzo iniziale fino ad un certo massimo finale. Nel momento nel quale questo processo viene eseguito, un'istante prima di mandarlo in esecuzione, il sistema operativo inizializza all'interno del processore due registri speciale che sono "base" e "limite". Questi registri sono inizializzati per contenere l'indirizzo iniziale e l'indirizzo finale dello spazio di memoria assegnato al processo che ora sto mettendo in esecuzione. Quindi il processore deve prevedere questa possibilità quindi ci devono essere questi due registri nel processore dopo di che ogni qual volta il processore invia un indirizzo alla memoria quindi ogni qual volta esegue un ciclo di fetch and execute spedisce un indirizzo alla memoria per carica l'istruzione e poi spedisce un indirizzo di memoria per caricare e scrivere un dato, questi indirizzi devono passare tramite un controllo per cui si verifica che siamo all'interno di questi due limiti e se stanno all'interno di questi due limiti possono andare in memoria altrimenti si solleva un'eccezione. Questo meccanismo presenta tantissimi limiti era tipico dei sistemi anni 60 primi anni 70 quando c'erano diversi limiti di ciò che si poteva fare dal punto di vista dell'hardware. Questo controllo qui è un controllo che deve fare l'hardware. Questo è un meccanismo molto rigido perché una volta che ho allocato memoria per quel processo, quel processo può allocare ulteriore memoria quindi potrebbe avere bisogno di allocare dinamicamente la memoria d'altra parte con questo meccanismo non è semplice, è possibile ma è molto costoso per questo motivo i sistemi che adoperavano questo meccanismo in realtà allocavano lo stack in fondo allo spazio di memoria associato, i dati all'inizio erano fatti di memoria associata subito dopo il codice e in mezzo lasciavano uno spazio di memoria vuoto (lo heap) che serviva come camera di compensazione per fare le malloc o per aumentare la dimensione dello stack. Ora il grosso problema è come faccio a stimare la quantità di memoria della quale il processo ha bisogno? Spesso quello che succede è che si sovradimensiona e quindi si spreca memoria oppure si

sottodimensiona e allora il processo può esaurire lo heap quando ha esaurito lo heap tipicamente fa danno a sé stesso. Questo limite può essere superato utilizzando dei meccanismi di indirizzamento virtuale quindi usando non dei semplici sistemi di controllo dei limiti ma utilizzando dei meccanismi di interruzione dinamica degli indirizzi che ci permettono una gestione più flessibile della memoria fisica mappando in memoria virtuale la memoria fisica. Cosa fa il programma? Dichiara una variabile statica, all'interno del main dichiara una variabile locale, assegna un valore a queste variabili e poi stampa l'indirizzo dalla variabile e il contenuto dalla variabile. Se lo mettiamo in esecuzione otteniamo i risultati nella slide. Se noi eseguiamo questo

Example: Corrected (What Does this Do?)

```
int staticVar = 0;     // a static variable
main() {
   int localVar = 0;   // a procedure local variable

   staticVar += 1; localVar += 1;

   sleep(10);  // sleep causes the program to wait for x seconds
   printf ("static address: %x, value: %d\n", &staticVar, staticVar);
   printf ("procedure local address: %x, value: %d\n", &localVar, localVar);
}

Produces:
 static address: 5328, value: 1
 procedure local address: fffffffe2, value: 1
```

programma su 10 processi indipendenti cosa ci aspettiamo di ottenere? Ci sono due risposte: 1) Risultati uguali, il programma è lo stesso e in qualunque posto venga caricato comunque dal suo punto di vista verrà sempre caricato dall'indirizzo 0 e siccome il programma è sempre uguale, il programma occuperà sempre lo stesso spazio. Quindi i risultati saranno sempre uguali perché ogni processo gestirà sempre la sua memoria al suo interno alla stessa maniera e quindi quelle variabili saranno sempre allo stesso posto quindi i valori saranno sempre identici. 2) Nel caso in cui lo mandassimo davvero in esecuzione otterremo risultati sempre diversi perché qualche anno fa sono stati inventati dei meccanismi di attacco ai sistemi con l'attacco del buffer overflow. Per cui era possibile violare i sistemi inviando pacchetti sulla rete malformati molto lunghi e i sistemi operativi scritti male non se ne accorgevano, facevano un grande pasticcio con gli stack e poi da lì uno poteva eseguire il codice di qualsiasi sulle macchine degli altri. Per evitare questo tipo di attacco i compilatori moderni fanno un meccanismo di randomizzazione dello spazio degli indirizzi quindi spostano lo stack in punti arbitrali, la memoria viene allocata in maniera arbitrale ad ogni esecuzione. Quindi in teoria seppure dovrebbero essere identici in pratica per questo ulteriore meccanismo di protezione dei compilatori e dei sistemi diventano casuali. Che relazione ha con quello che diceva prima? Per effetto di questi meccanismi di protezione noi carichiamo i programmi in un loro spazio gli diciamo che questo spazio parte dall'indirizzo zero ed arriva ad un indirizzo massimo, loro gestiscono il loro spazio sempre alla stessa maniera anche se sono processi diversi eseguiti in tempi diversi, il programma è lo stesso e quindi lo spazio di memoria verrà gestito alla stessa maniera e quindi gli indirizzi restano gli stessi.

Sistemi operativi 27/02/17

Per quanto riguarda il processo, sostanzialmente è una sequenza di azioni che il processore esegue sulla base di un programma. Quindi il processo è un'entità dinamica, è un'astrazione, realizzata dal nucleo, e permette l'esecuzione di un programma con diritti limitati. Quindi mettendo in esecuzione un processo, specifichiamo che da quel momento in poi il processore deve eseguire le istruzioni applicando dei limiti a ciò che può fare, dove questi limiti sono di diversa natura, sono intermedi di accesso in memoria, sono istruzioni privilegiate e sono anche intermedi di tempo, quindi dopo un certo tempo il sistema operativo deve riacquisire il controllo per riportargli il valore in stato supervisore. Quindi la prima astrazione è quella di processo, il secondo elemento è la modalità utente-supervisore. In particolare la modalità utente-supervisore da sola non basta, deve essere combinata ad altri meccanismi, in particolare dobbiamo decidere quali sono le istruzioni privilegiate, stabilire che eseguiamo qualcosa in modalità supervisore o modalità utente ha poco senso se non stabiliamo che cosa si può fare e cosa non si può fare in ognuna delle due modalità. Abbiamo visto alcuni esempi di istruzioni privilegiate, poi abbiamo visto come si possono imporre i limiti all'accesso in memoria utilizzando la coppia dei registri base limite e infine restava da vedere il meccanismo del timer. Il timer è importante perché una volta che noi commutiamo il processore in modalità utente, il processore resta in modalità utente, e da questa modalità non abbiamo modo di ritornare in stato supervisore tramite un'istruzione, quindi tutte le istruzioni che il processore esegue sono istruzioni modalità utente che non hanno nessun modo di poter tornare in stato supervisore(stato kernel), perché il passaggio da stato utente a stato supervisore è un passaggio che richiede dei diritti privilegiati, quindi dobbiamo già essere in modalità kernel per poterci passare. Una volta che siamo passati in modalità utente, a quel punto ci restiamo. Chi resta in esecuzione è il processo che abbiamo mandato in questa modalità. Se non facciamo niente per porre dei limiti temporali, questo processo può restare in esecuzione in eterno, sempre in stato utente, però nessun altro processo potrà acquisire i diritti di esecuzione, a meno che lui spontaneamente in qualche modo non li ceda. Per questo motivo ci serve il meccanismo del timer. Il timer è un dispositivo che viene impostato dal nucleo per rimettere un'interruzione entro un certo tempo, quindi, quando il nucleo decide di mettere in esecuzione un processo, prima di metterlo in esecuzione, imposta il timer in maniera tale che scatti ad esempio al termine del quanto di tempo. A questo punto passa in modalità utente, contemporaneamente cede il controllo del processore al processo utente, il processo utente è libero di fare tutto ciò che vuole all'interno dei vincoli della modalità utente del processore. Quindi può eseguire tutte le istruzioni non privilegiate, può accedere alla sua memoria liberamente. Quando ad un certo punto però, il dispositivo timer scatta, genera un'interruzione, quest'interruzione viene riconosciuta dal processore, e in virtu' del meccanismo di interruzione, il sistema operativo riacquisisce il controllo, quindi può interrompere il processo che era in esecuzione e riassegnare il processore ad un altro processo. Ci sono tre cose importanti da ricordare: che quando arriva un'interruzione, il controllo viene restituito al sistema operativo, ed in particolare una funzione del sistema operativo che gestisce le interruzioni, che si chiama Interrupt Handler. La seconda cosa importante è che la frequenza delle interruzioni del timer, quindi l'istante nel quale il timer dovrà scattare, non è impostato in modalità utente e non può essere impostato in modalità utente, ma può essere impostato soltanto dal nucleo in stato supervisore. Se il processo utente potesse modificare l'istante di interrupt, quindi il tempo del timer, potrebbe spostarlo avanti nel tempo e potrebbe mantenere il controllo del processore al di là dei limiti temporali che gli vogliamo imporre. In alcuni casi le interruzioni possono essere rinviate di poco se si sono verificate delle situazioni per cui c'è bisogno rinviarle, però questo rinvio non può essere di nuovo fatto dal codice utente ma dev'essere una cosa che può fare solo il sistema operativo in modalità supervisore. Il motivo del perché si verifica in queste situazioni e di come si fa a ritardare un'interruzione lo vediamo quando vediamo il meccanismo di gestione delle interruzioni e quando si parlerà più avanti della sincronizzazione tra processi. Questi sono i meccanismi che ci permettono di rendere efficaci i meccanismi di protezione, sostanzialmente nella commutazione di processo il sistema operativo stabilisce quali sono i limiti all'interno del quale può essere eseguito del codice in modalità utente, dopodiché, stabilendo quali sono le istruzioni privilegiate, utilizzando il timer per limitare il tempo di esecuzione, il codice utente,

stabilendo dei limiti alla memoria, abbiamo i meccanismi che ci permettono di rafforzare i meccanismi di protezione, di rendere effettivi i meccanismi di protezione. L'ultimo elemento che manca è il passaggio da stato utente a stato supervisore. Fatto questo il meccanismo di protezione è completo. Il passaggio da stato utente a stato supervisore in realtà è duplice: abbiamo il caso in cui siamo in stato utente e dobbiamo passare in stato supervisore, oppure siamo in modo supervisore e dobbiamo passare in stato utente. Questi due passaggi sono molto differenti. Vediamo il primo, da stato utente a stato supervisore. Questo passaggio è forzato all'arrivo di un'interruzione. Quando un timer genera un'interruzione, questa interruzione è riconosciuta dal processore all'interno del ciclo fetch-execute, il processore quindi passa ad eseguire una funzione di gestione delle interruzioni che è del sistema operativo, e nel fare questo passaggio in maniera automatica, con un suo meccanismo, commuta da stato utente a stato supervisore, quindi questa commutazione non è ottenuta come un'istruzione, ma è un meccanismo del processore che quando riconosce un'interruzione, applica questo passaggio. Questo vale per tutte le interruzioni, non soltanto l'interruzione generata dal timer, ma anche per le interruzioni generate da un qualsiasi dispositivo. Questo perché in modalità utente non si può interagire direttamente con i dispositivi, l'interruzione con i dispositivi è riservata al sistema operativo in stato supervisore. Un'interruzione generata da qualsiasi dispositivo, in realtà, è parte della comunicazione tra il dispositivo e il sistema operativo, di conseguenza tutte le interruzioni devono essere intercettate dal sistema operativo, devono mettere in esecuzione codice del sistema operativo in stato supervisore. Tutte le interruzioni provocano questo passaggio. Non è l'unico caso, può capitare che il processo, quando è in esecuzione in stato utente, commetta degli errori, per esempio violi i limiti di protezione della memoria, tenti di eseguire istruzioni privilegiate, oppure banalmente commetta degli errori (divisione per zero, altre operazioni di questo genere). Queste situazioni non sono interruzioni provenienti dai dispositivi, ma sono situazioni generate nel processore nell'esecuzione di codice utente, quindi non sono propriamente interruzioni, vengono chiamate eccezioni, a tutti gli effetti si comportano come interruzioni, l'unica differenza è che sono sincrone con l'esecuzione del codice. Per distinguere dalle interruzioni, che sono eventi esterni, le chiamiamo eccezioni, ma la loro gestione è perfettamente identica alle interruzioni. Quando un processo nella sua esecuzione genera un'eccezione, per esempio una violazione della protezione della memoria, viene settato un bit di eccezione che è l'equivalente di un bit di interruzione, che viene riconosciuto dal processore nel ciclo fetch-execute, così come vengono riconosciute le interruzioni, e anche l'eccezione causa l'esecuzione nel nucleo del sistema operativo, di un gestore dell'eccezione, di conseguenza anche questo passaggio comporta un passaggio da modalità utente a modalità supervisore. L'ultimo caso di passaggio da modalità utente a modalità supervisore è nel caso delle chiamate di sistema. Abbiamo detto che un processo quando lavora in modo utente non può utilizzare i dispositivi perché questi sono gestiti dal sistema operativo in modalità supervisore, quindi abbiamo una protezione dei dispositivi per evitare che gli utenti li possano usare liberamente. Il motivo per impedire gli utenti di usare liberamente i dispositivi è che queste sono risorse condivise e quindi dobbiamo impedire che un utente possa iniziare una stampa se è in corso la stampa di un altro utente per esempio. D'altra parte però, se impediamo ai processi utente di accedere ai dispositivi, significa che stiamo impedendo ai processi utente di usarli, di stampare qualcosa a schermo, di leggere dal mouse, di leggere dal disco. Questo non è quello che vogliamo fare in realtà, noi vogliamo permettere ai processi utente di usare le risorse hardware, quindi dispositivi, ma vogliamo che questo uso sia mediato dal sistema operativo in forma controllata. Quindi dobbiamo permettere ai processi in stato utente di utilizzare i dispositivi, allo stesso tempo dobbiamo impedirgli di utilizzarli liberamente. Come si fa? In realtà i processi utente non utilizzeranno mai i dispositivi in maniera diretta, ma fanno una richiesta al sistema operativo di utilizzarli per conto loro. Piuttosto che andare a leggere direttamente dal disco, un processo utente chiederà al sistema operativo di leggere per suo conto quella porzione del disco, il sistema operativo in questo modo potrà controllare se quel processo utente effettivamente è autorizzato a leggere da quella porzione del disco. Pure noi quindi potremo eseguire l'operazione per conto suo. Quindi ci serve un meccanismo che permetta ai processi in stato utente di chiedere l'esecuzione di un'azione, di una procedura nel sistema operativo. Questo è complicato perché non possiamo saltare arbitrariamente ad

un'istruzione del sistema operativo. Questo potrebbe creare dei problemi: immaginiamo di prendere un programma qualsiasi scritto da noi, immaginiamo che anziché venga eseguito dalla prima riga, come normalmente uno si aspetta, venga eseguito da una riga a caso. Difficilmente possiamo aspettarci un risultato coerente da quel programma, se iniziamo ad eseguirlo da una riga a caso. Tutti i programmi sono fatti per essere eseguiti dalla prima riga seguendo una sequenza di istruzioni ben precisa. Se permettessimo ai processi in stato utente di saltare liberamente dentro al nucleo, quindi dentro le funzioni del sistema operativo, questi potrebbero saltare a un'istruzione arbitraria e potrebbero creare problemi all'interno del nucleo, sicuramente potrebbero alterarne il funzionamento normale. Quindi dobbiamo permettere da un lato ai processi utente di invocare delle funzioni del sistema operativo, ma questa invocazione delle funzioni del sistema operativo, dev'essere controllata, dobbiamo fare in modo che questa invocazione avvenga soltanto per alcune funzioni ben definite e non specificando un indirizzo arbitrario ma partendo sempre dall'inizio di ognuna di queste funzioni. Le funzioni che il sistema operativo mette a disposizione dei processi utente, vengono anche dette chiamate di sistema, sono un numero limitato rispetto a tutte le funzioni, al codice che il sistema operativo implementa, ma permettono l'utilizzo di tutte le risorse hardware del sistema e il processore deve mettere a disposizione un meccanismo per invocare queste funzioni e soltanto queste. Il meccanismo si chiama meccanismo delle chiamate di sistema, è un meccanismo parallelo all'invocazione di funzione di procedura. La differenza è che questa invocazione di funzione di procedura avviene in modo protetto. Il processore deve prevedere all'interno del suo set di istruzioni un'istruzione specifica per eseguire le chiamate di sistema. Quest'istruzione specifica prende vari nomi (svc, int, a seconda dei processori prende nomi differenti), ma fondamentalmente quello che fa è di generare un'interruzione. Di nuovo ci si appoggia al meccanismo delle interruzioni per avere un passaggio dalla modalità utente alla modalità supervisore in modo protetto. Abbiamo anche il passaggio da modalità supervisore a modalità utente. Perché ci serve questo passaggio? Quando creiamo un nuovo processo e dobbiamo metterlo in esecuzione, la creazione del processo avviene nel nucleo del sistema operativo in stato supervisore (in stato kernel), però quando la creazione è completa dobbiamo mettere il processo in esecuzione e questo deve essere messo in esecuzione in stato utente, quindi ci serve un meccanismo per fare questo, per poter passare il controllo al processo appena creato e contemporaneamente far passare il processo in stato utente. Il passaggio da stato supervisore a stato utente deve avvenire anche quando abbiamo completato la gestione di un'interruzione: se era in esecuzione un processo in stato utente, arriva un'interruzione da un dispositivo, quest'interruzione provoca il passaggio in stato supervisore l'invocazione di una funzione di gestione dell'interruzione nel nucleo del sistema operativo che dialoga col dispositivo, faccia quel che deve fare, quando ha terminato deve restituire il controllo al processo in stato utente. Quindi quando la funzione di gestione delle interruzioni termina, quando l'interrupt Handler ha completato il suo compito, deve restituire il controllo al processo in stato utente, e nel restituire il controllo deve ripassare da stato supervisore a stato utente. Il passaggio da stato supervisore a stato utente ci serve anche quando applichiamo la commutazione di contesto: basti ricordare che nel time sharing assegniamo un quanto di tempo ad ogni processo e li mettiamo in esecuzione ciclicamente. Quando termina il quanto di tempo, arriva l'interruzione dal timer, mette in esecuzione il nucleo che toglie dall'esecuzione un processo, assegna il processore ad un altro processo, a questo punto questo processo al quale è stato assegnato il processore deve tornare in esecuzione in modalità utente, quindi anche nella commutazione di contesto ci serve il passaggio da stato supervisore a stato utente, ed infine ci serve anche per il meccanismo delle APP CALL. Il meccanismo delle APP CALL è un meccanismo che permette al sistema operativo, quindi al nucleo, di invocare funzioni nei processi (thread, trend, ???), per inviare delle notifiche, dei segnali, quindi quando il sistema operativo vuole inviare un segnale a un processo utente, quindi vuole mettere in esecuzione una certa funzione, il sistema operativo predispone il segnale in stato supervisore e poi passa il controllo al processo in stato utente per eseguire la funzione che il sistema operativo vuole invocare. Se mettiamo in fila tutte queste cose, abbiamo diversi casi di passaggio da stato utente a stato supervisore, diversi casi di passaggio da stato supervisore a stato utente, quindi potenzialmente sono tanti meccanismi diversi che ci servono. In realtà i meccanismi alla fine sono soltanto due: il passaggio da stato supervisore a stato utente

viene fatto appoggiandosi sul meccanismo delle interruzioni del processore, quindi che siano interruzioni esterne, che siano eccezioni, che siano chiamate di sistema, si sviluppa sempre lo stesso meccanismo. Il passaggio da stato supervisore a stato utente, quale che sia il caso, viene sempre sviluppato tramite un'istruzione privilegiata che si chiama Return From Interrupt. Quindi è evidente che per capire questa modalità di passaggio, da stato utente a stato supervisore, bisogna capire bene come funzionano gli interrupt e come questi vengono gestiti dal sistema operativo. Nella gestione degli interrupt ci sono due elementi: la gestione dell'interrupt che arriva e la terminazione dell'interrupt, quindi abbiamo terminato di servirlo, come ritorniamo. Per la gestione dell'interrupt, ci serve un vettore delle interruzioni, che altro non è che una tabella che contiene indirizzi delle funzioni Handler, quindi per ogni interruzione che arriva, il vettore delle interruzioni ci dice qual è la funzione Handler del nucleo del sistema operativo che la deve gestire. Questo vettore delle interruzioni contiene l'indirizzo iniziale degli Handler. Utilizzando l'interrupt vector, siamo sicuri che qualsiasi interruzione non possa invocare il sistema operativo in un indirizzo arbitrario, ma invoca il sistema operativo in alcuni ben precisi punti di accesso che sono gli Handler. Associato all'indirizzo dell'Handler nell'interrupt vector c'è anche la parola di stato, un registro di stato, che serve per essere copiato all'interno del registro di stato del processore e serve quindi per impostate modalità supervisore e fare altre cose che vedremo. Il secondo elemento che ci serve è uno stack per gestire le interruzioni. Quindi il nucleo in realtà, per la gestione delle interruzioni, ha un proprio stack (ne ha più d'uno in realtà), all'interno del quale vengono messi i record di attivazione delle funzioni che seguono poi l'interruzione. Quindi nello stack del nucleo viene messo il record di attivazione dell'interrupt Handler, e poi se l'interrupt Handler invoca delle altre funzioni del nucleo, queste creano dei record di attivazione di queste funzioni all'interno dello stack del nucleo. Il fatto di utilizzare uno stack specifico per la gestione delle interruzioni, quindi diverso dallo stack che i processi utilizzano in spazio utente, permette di rendere il nucleo indipendente dallo stack utente, quindi evitiamo che ci possano essere problemi nell'utilizzo dello stack utente. Un altro elemento importante è il masking delle interruzioni: quando riceviamo un'interruzione da un dispositivo (o che sia anche un'eccezione generata dal processo, o che sia una chiamata del sistema), dobbiamo passare all'esecuzione dell'handler nel nucleo, se però, mentre stiamo eseguendo l'handler, arriva un'altra interruzione, che succede? Il sistema operativo potrebbe entrare in confusione, quindi per evitare questo problema, l'esecuzione dell'handler viene fatta a interruzioni disabilitate. Nel passaggio all'handler, quando arriva l'interruzione, si passa in modalità supervisore e si disabilitano le interruzioni, contemporaneamente si salta alla funzione di gestione delle interruzioni, quindi all'interrupt handler. L'handler viene eseguito ad interruzioni disabilitate, per impedire che ulteriori interruzioni che arrivano in questa fase, possano provocare uno stato inconsistente nel processore e nel sistema operativo. Il punto è che quando arriva un'interruzione, il processore passa in modo supervisore, disabilita le interruzioni e inizia ad eseguire una funzione del nucleo che si chiama handler, quindi abbiamo realizzato un passaggio da del codice che era eseguito in stato utente sul processo ad un codice che viene eseguito nel sistema operativo in stato supervisore ad interruzioni disabilitate. Perché disabilitiamo le interruzioni nell'esecuzione dell'handler? Perché se in questa fase, mentre eseguiamo l'handler, arriva un'ulteriore interruzione, questa potrebbe causare un'inconsistenza nello stato del processore del sistema operativo e poi tutto il sistema potrebbe andare in crash. Quindi quando arriva un'interruzione dobbiamo fare tante cose: passare in modo supervisore, disabilitare le interruzioni, saltare ad un indirizzo ben preciso contenuto all'interno del vettore delle interruzioni, e poi in virtù del fatto che passiamo in modalità supervisore dobbiamo modificare i diritti di accesso alla memoria, dobbiamo permettere di utilizzare memoria del nucleo e via discorrendo. Quindi dobbiamo abolire i limiti dati dalla protezione della memoria fin tanto che il processo era in stato utente. Tutte queste cose devono essere fatte in un ordine ben preciso? E se si, con quale ordine? Vanno fatte tutte contemporaneamente. In maniera atomica, indivisibile, contemporanea, devo modificare il Program counter (saltare), impostare lo stack pointer per puntare lo stack del nucleo per utilizzare lo stack del nucleo come stack di riferimento per l'handler, devo disattivare i meccanismi di protezione della memoria e devo commutare da stato utente a stato supervisore e devo anche disabilitare le interruzioni. Tutte queste cose non le possiamo fare senza avere un

aiuto da parte dell'hardware, dev'essere il processore che ci mette a disposizione un meccanismo per fare tutte queste cose contemporaneamente. Tutto ciò deve essere fatto senza che il processo utente si renda conto di niente. Il passaggio all'interrupt Handler deve avvenire senza modificare lo stato del processo utente. Immaginiamo un programma in esecuzione, quindi abbiamo il processo, arriva l'interruzione dal disco per una richiesta fatta da qualcun altro, non vogliamo che a causa di questa interruzione il processo restituisca il risultato sbagliato o vada in crash, etc. La gestione delle interruzioni deve essere trasparente in questo senso: quando arriva l'interruzione la gestisco e interrompo il processo che era in esecuzione, ma quando ho terminato la gestione dell'interruzione, devo ripristinare tutto come prima, in maniera tale che il processo possa riprendere la sua esecuzione come se non fosse successo niente. L'unica cosa che causa un interruzione dal punto di vista del processo che era in esecuzione è soltanto un po' di ritardo, ma non certamente una modifica dei propri dati, del proprio program counter, etc. Questo passaggio diventa piu' critico, perché se devo modificare il program counter per saltare all'indirizzo dell'handler, sto modificando un program counter che contiene, nel momento in cui arriva l'interruzione, l'indirizzo che attualmente viene eseguita dal processo in stato utente, quindi il valore contenuto all'interno del program counter non posso cancellarlo/sovrascriverlo, ma lo devo conservare, perché quando ho finito di eseguire l'interruzione dovrò riprendere esattamente da quel punto, l'esecuzione del processo utente. Contemporaneamente a tutte queste modifiche, attuate per rispondere all'interruzione dovrò preoccuparmi di salvare lo stato del processo che era in esecuzione. Rivediamo tutti questi concetti uno per volta partendo dal vettore delle interruzioni. Il vettore delle interruzioni è una struttura dati che sta in memoria principale nello spazio protetto dal nucleo, che contiene l'indirizzo di ogni handler per ogni possibile interruzione. L'handler è una funzione scritta dal sistema operativo, propria del sistema operativo. Se sul nostro hardware mettessimo linux/ma cos/windows, scopriremmo di avere handler completamente differenti fra loro, infatti ogni sistema operativo gestisce le interruzioni a modo suo. Questa tabella deve essere presente e deve avere la stessa forma per tutti i sistemi operativi perché in realtà è utilizzata dal processore. Per ogni possibile codice di interruzione questa tabella deve avere una riga che contiene l'indirizzo di quell'handler. La tabella è inizializzata dal sistema operativo, quando il sistema operativo si accende, viene messa in funzione, una delle prime cose che fa è impostare dei valori per tutti questi indirizzi, quindi legare le interruzioni ai propri handler. In realtà l'interrupt handler oltre ad avere gli indirizzi delle varie funzioni di gestione, contiene anche, associate ad un indirizzo, la parola di stato, per impostare i diritti di stato supervisore nell'esecuzione dell'handler. Di handler ce ne sono diversi, alcuni riguardano le interruzioni esterne, altri riguardano le eccezioni, altri le chiamate di sistema. Hanno tutti la stessa gestione. Inizializzare il vettore delle interruzioni è un'operazione delicata, che succede se mentre sto inizializzando il vettore delle interruzioni, mi arriva un'interruzione? Il processore salta ad un indirizzo che non è l'indirizzo iniziale dell'handler, è un altro indirizzo, ed inizia ad eseguire istruzioni a partire da quel punto. A queste condizioni è molto facile che il sistema operativo vada in crash. Quindi quando si inizializza il vettore delle interruzioni, il sistema operativo disabilita le interruzioni e le riabilita quando ha terminato. Per quanto riguarda lo stack del nucleo (usato per gestire le interruzioni), abbiam detto che il nucleo ha uno stack apposta per gestire le interruzioni che è diverso dallo stack che i processi utilizzano in spazio utente. Perché non possiamo utilizzare lo stack utente concretamente? Che tipo di problemi comporta? Immaginiamo di essere in un sistema multiprocessore, con processi multithread, lo stack utente in realtà appartiene allo spazio di memoria del processo utente sul quale potenzialmente possono agire tutti i thread di quel processo per cui se un thread del processo invoca il nucleo o arriva un'interruzione, quel thread viene interrotto e su quel processore parte l'esecuzione dell'handler di gestione dell'interruzione, ma su un altro processore potrebbe ancora essere in esecuzione un thread dello stesso processo che potrebbe andare a modificare lo stack del nucleo tranquillamente perché è memoria sua. Quindi questo gli permetterebbe di modificare il nucleo e quindi danneggiare il nucleo e farlo fallire, oppure potrebbe permettergli di andare a scrivere dei dati all'interno dello stack per cui quando il nucleo ritorna da interruzione, restituisce il controllo al processo utente lasciandogli i diritti di supervisore. Ogni processo/ogni thread ha un doppio stack: stack che utilizza in stato utente e stack che viene utilizzato in stato supervisore quando vengono eseguite

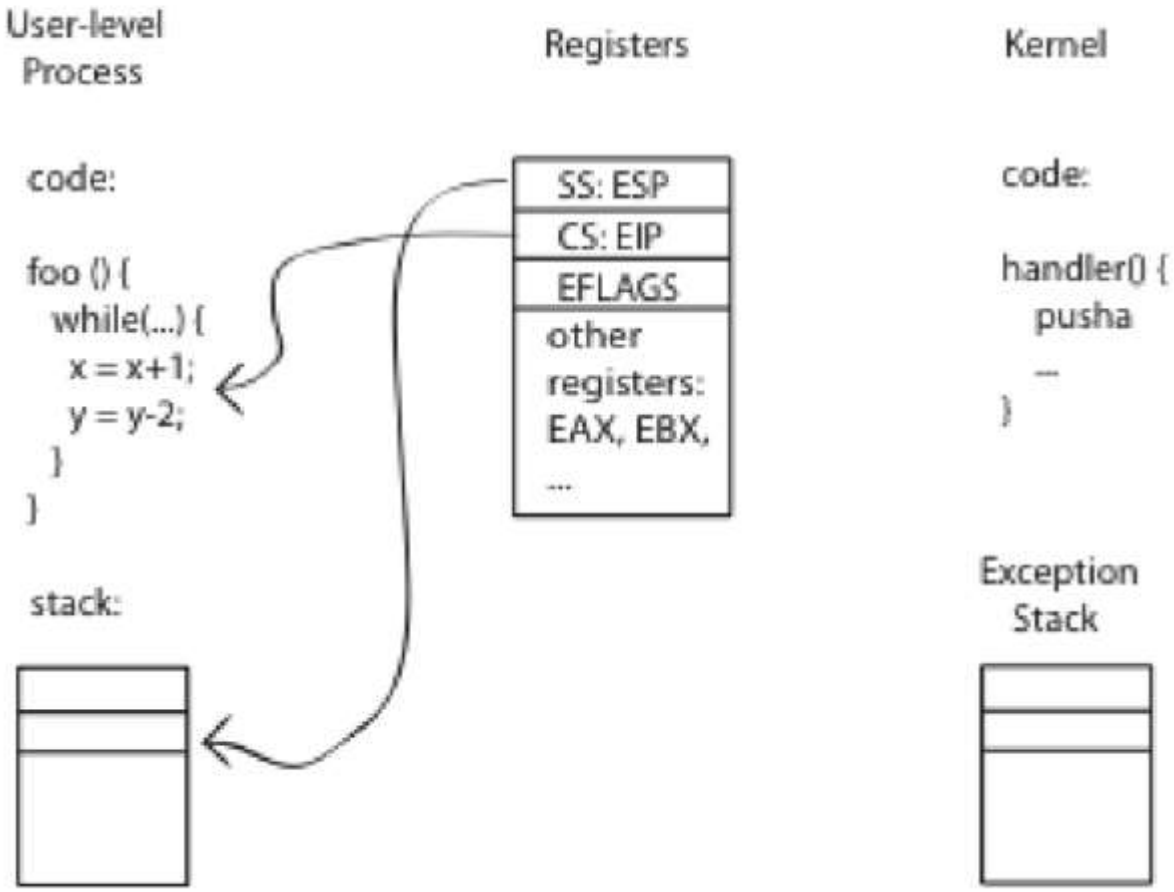
chiamate di sistema o rimanda all'interruzione. Quando il processo è in esecuzione utilizza il proprio stack, e all'interno dello stack in stato utente ci sarà il record di attivazione della funzione main, il record di attivazione delle varie funzioni vuotate(?) dal processo. Lo stack del nucleo, abbinato a questo processo, è vuoto. Quando invece il processo è pronto ad andare in esecuzione ma non possiede il processore, c'è un altro processo in esecuzione ma non lui, allora lo stack utente contiene le stesse informazioni, nello stack del nucleo invece viene conservato lo stato del processo pronto per essere ripristinato, pronto per rimettere in esecuzione il processo stesso. Se invece il processo utente invoca una chiamata di sistema, in spazio utente ci sono tutti i record di attivazione delle varie funzioni invocate dal processo in stato utente più c'è un record di attivazione per la chiamata di sistema, dopodichè nello stack del nucleo troveremo lo stato salvato di quel processo e poi i record di attivazione per la chiamata di sistema e i vari record di attivazione legati al driver, quindi al componente del sistema operativo che gestisce il dispositivo.

PAUSA PRIMA ORA.

Per gestire correttamente le interruzioni, ci serve il vettore delle interruzioni, ci serve uno stack nel nucleo per la gestione delle interruzioni, poi ci serve disabilitare le interruzioni. Quando l'handler dell'interruzione viene messo in esecuzione, viene messo in esecuzione ad interruzioni disabilitate e quando poi ha terminato, riabilita le interruzioni e restituisce il controllo in stato utente al processo che era in esecuzione. In realtà però la disabilitazione delle interruzioni è un meccanismo molto utile che viene usato anche in altre circostanze, non solo nella gestione degli handler, è un meccanismo privilegiato quindi i processi in stato utente non possono disabilitare le interruzioni, c'è un'istruzione specifica che disabilita le interruzioni ed è una istruzione privilegiata. Il sistema operativo la può però utilizzare in diverse circostanze, per esempio quando fa una commutazione di contesto quindi quando cambia il processo in esecuzione oppure la utilizza per ritardare una interruzione se si trova in una fase critica: ad esempio sta modificando il vettore delle interruzioni, disabilita le interruzioni e poi quando ha completato lo riabilita. In generale disabilitare le interruzioni è una cosa utile, risolvere una serie di problemi, d'altra parte se lo utilizza in maniera eccessiva può creare problemi gravi perché quando disabilitiamo le interruzioni, effettivamente le interruzioni non le riconosciamo e quindi non le possiamo servire. Se disabilitiamo le interruzioni troppo a lungo, può capitare che le interruzioni si accumulino e alla fine ci perdiamo degli eventi. In generale le interruzioni si possono disabilitare per periodo di tempo molto brevi, il minimo indispensabile, generalmente per eseguire una manciata di istruzioni o poco più. Nell'architettura x86 ci sono due istruzioni (clear, sti) che disabilitano e riabilitano le interruzioni. Questo meccanismo di abilitazione e disabilitazione delle interruzioni si applica ad una sola CPU, non si applica a tutte le CPU, ogni CPU ha il proprio e questo ha poi un po' di implicazioni. In ogni caso sull'utilizzo delle interruzioni per gestire queste situazioni critiche avremo modo di riparlarne quando faremo la sincronizzazione. Per effetto della disabilitazione delle interruzioni, l'interrupt handler può eseguire il suo codice senza bloccarsi. L'interrupt handler serve per gestire la comunicazione con un dispositivo, quindi: un dispositivo completa un'operazione, lancia un interrupt, il sistema operativo in risposta esegue l'handler, l'handler comunica col dispositivo, scambia informazioni, etc. Il problema è che se questa comunicazione col dispositivo porta via molto tempo, noi dobbiamo tenere le interruzioni disabilitate molto a lungo e questa non è una buona idea. Per questo motivo in realtà la comunicazione con i dispositivi, ma in generale tutta la gestione delle interruzioni viene fatta in due fasi: c'è l'handler che serve a riconoscere soltanto l'interruzione e capire qual è l'azione da compiere, una volta che l'ha capito passa ad eseguire un'altra funzione (nel caso della comunicazioni con i dispositivi è il driver la parte che effettivamente scambia le informazioni col dispositivo). Questa seconda parte di gestione delle interruzioni può essere fatta ad interruzioni abilitate e spesso viene demandata ad un thread del nucleo (per lo meno nei sistemi moderni). In generale il meccanismo funziona in questa maniera: arriva un'interruzione, il processore la riconosce e mette in esecuzione il nucleo del sistema operativo che esegue l'handler a interruzioni disabilitate in stato supervisore. L'interrupt handler si preoccupa di essere eseguito in maniera tale da non alterare i dati dei processi utente, capisce qual è l'interruzione, capisce qual è l'azione da compiere, quindi il sistema operativo in qualche modo ne prende atto, e mette in esecuzione un thread del

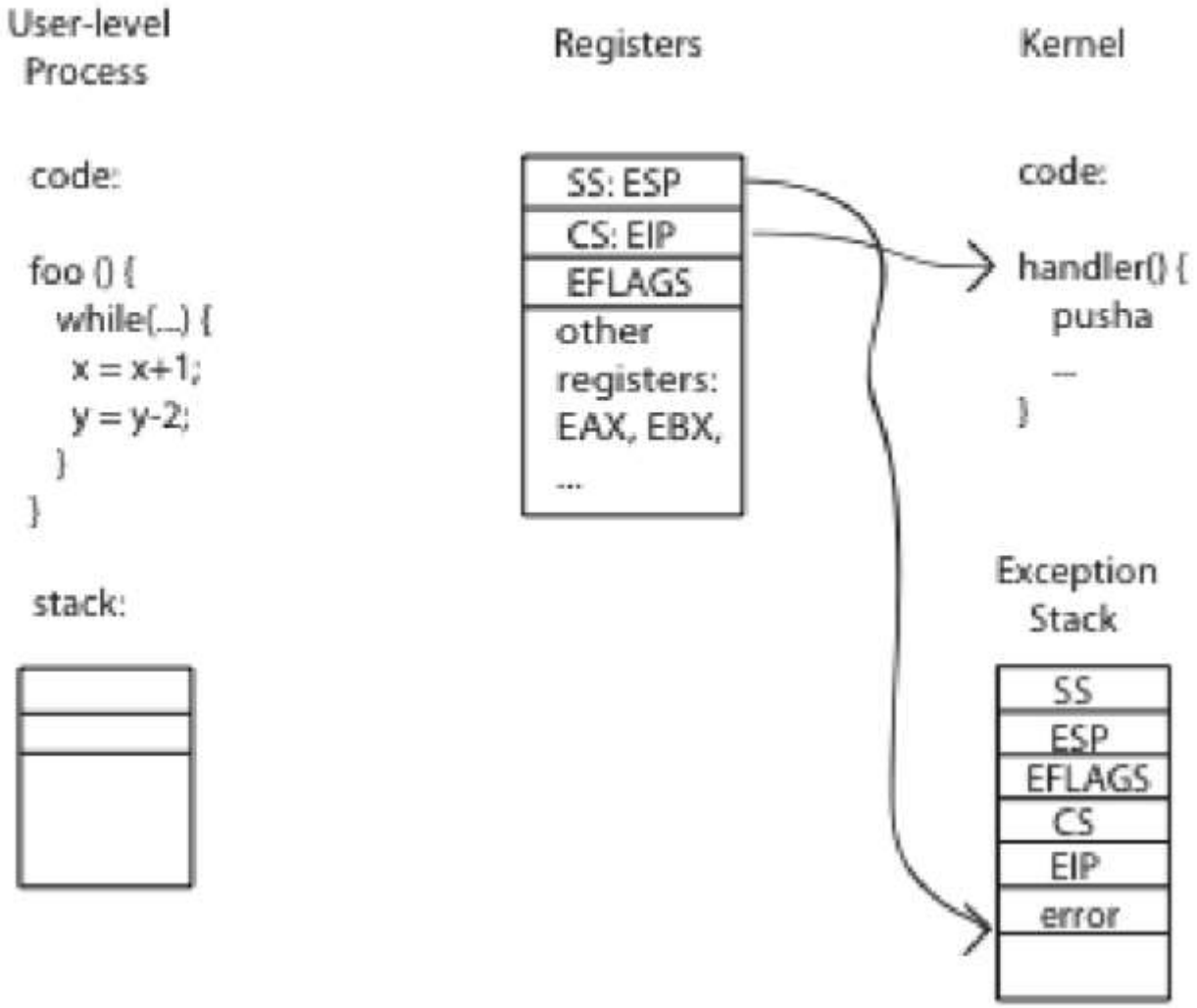
nucleo che completa l'operazione, quindi fa quello che effettivamente deve fare. A questo punto l'interrupt handler però termina perché tanto c'è un altro thread del nucleo che si prende carico di completare la gestione dell'interruzione, quindi l'interrupt handler termina e restituisce il controllo al processo che era in esecuzione prima. In questo modo l'handler dura poco tempo, può essere svolto ad interruzioni disabilitate e non crea troppi problemi. La gestione dell'interruzione avviene in maniera indipendente tramite un thread del nucleo che può essere eseguito ad interruzioni abilitate, quindi non crea problemi al gestore delle interruzioni. Ciò vale sia se l'interruzione arriva dell'esterno, quindi arriva da un dispositivo, sia se c'è stata un'eccezione o se c'è stata una chiamata di sistema, per cui l'interrupt handler può materialmente eseguire una chiamata di sistema e metterla da parte oppure può materialmente eseguire l'eccezione che ha causato problemi al processo in un thread a parte, e a questo punto svolgere questi compiti ad interruzioni abilitate. I sistemi moderni, le interruzioni le disabilitano soltanto per l'handler e poi utilizzano thread del nucleo per svolgere i compiti di gestione vera dell'interruzione. Quando arriva l'interruzione, abbiamo visto che ci sono diverse cose da fare: bisogna cambiare il program counter, passare in modalità supervisore, disabilitare le interruzioni, commutare lo stack pointer per usare lo stack pointer del nucleo e poi bisogna farlo senza perdere i dati del processo che era in esecuzione. Tutte queste operazioni non possono essere fatte in tempi separati, ma vanno fatte tutte contemporaneamente. Questa operazione è detta modo di trasferimento atomico (Atomical transfer). Sostanzialmente è un'unica operazione atomica, indivisibile che viene fatta con l'aiuto del processore. Nell'architettura x86, il program counter, la program status word, vengono copiate all'interno di alcuni registri temporanei per evitare di distruggere il program counter, la program status word, in uso dal processo utente. Dopodichè si modifica la program status word passando alla modalità supervisore, si modifica lo stack pointer per passare lo stack del nucleo (anche per questo lo stack pointer deve essere salvato) e a questo punto in cima allo stack del nucleo vengono memorizzati lo stack pointer, il program counter, la program status word. Fatto questo in cima allo stack del nucleo abbiamo i valori dei registri speciali del processore che sono al sicuro. A questo punto il processore guarda il vettore delle interruzioni, trova un nuovo program counter, una nuova program status word, li carica e da questo momento in poi passa ad eseguire codice del nucleo. L'interrupt handler viene eseguito subito dopo, quindi la prima istruzione eseguita dopo l'interruzione, è già un'istruzione dell'handler, è la prima istruzione dell'handler. Ricordiamo che il processo che era in esecuzione stava usando i registri del processore, sia quelli speciali che quelli generali. I registri speciali li ha salvati il processore con questa operazione atomica e li ha messi in cima allo stack del nucleo. Ma i registri generali, quelli sui quali il processo aveva i dati temporanei della sua elaborazione, contengono ancora i valori in uso dal processo. D'altra parte, l'interrupt handler per poter lavorare avrò bisogno di usare i registri generali. Prima di poter fare il suo lavoro, l'interrupt handler si deve preoccupare di salvare i registri generali del processore in cima allo stack. Il nucleo ha bisogno di utilizzare i registri del processore che sono presenti in un'unica copia e per poter lavorare deve usare quelli, ma se li utilizza liberamente, li modifica. Tutto questo meccanismo serve per salvare lo stato del processo, in maniera tale che il processo non si accorga che è arrivata l'interruzione. Quando l'interrupt handler ha terminato, dovrà eseguire esattamente tutto il percorso inverso, quindi ripristinare i registri generali così com'erano, e poi ripristinare program counter, stack pointer, program status word così com'erano. Vediamo degli esempi.

Before



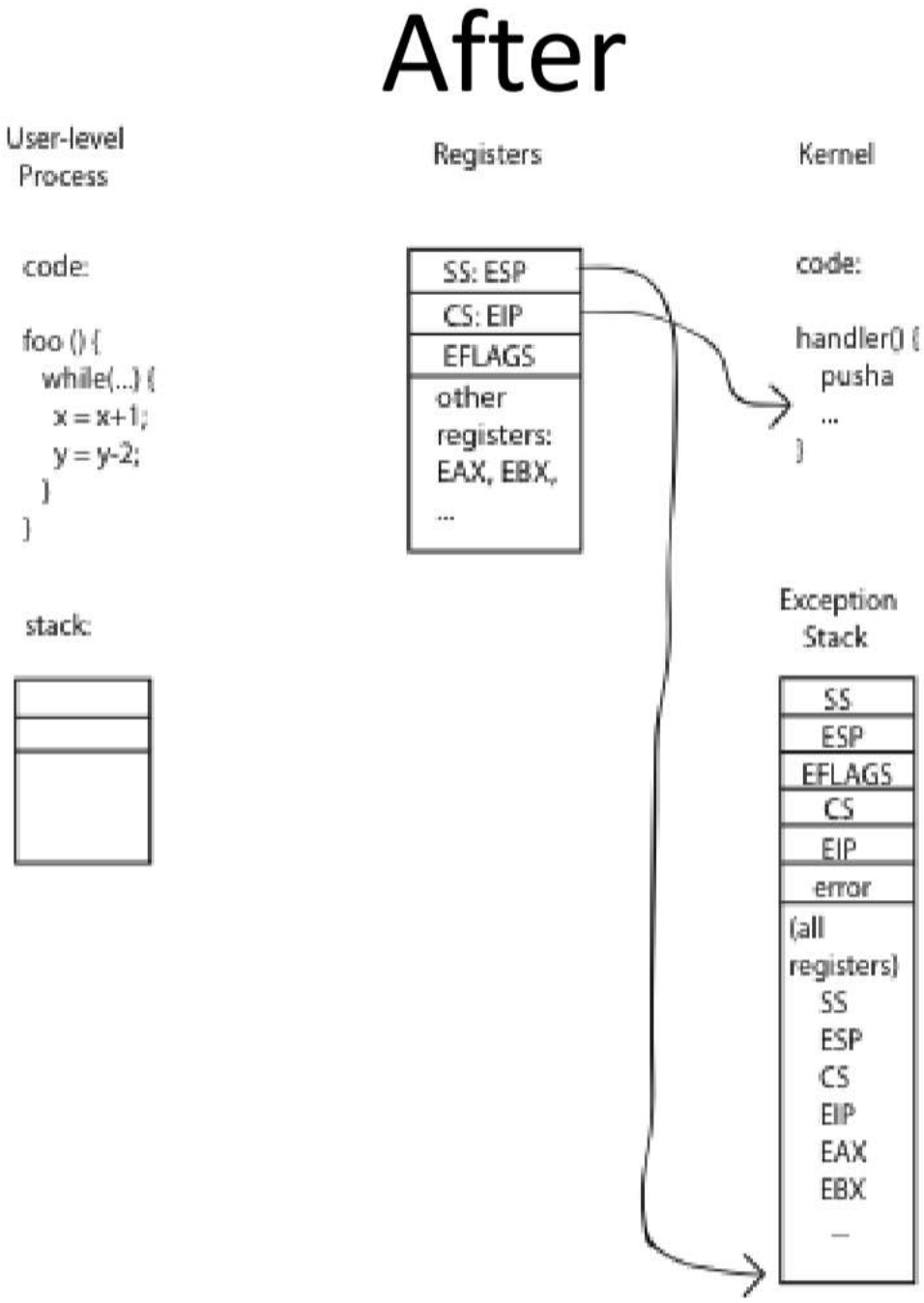
Abbiamo un processo che sta eseguendo codice a livello utente, nel riquadro al centro ci sono i registri del processore, c'è lo stack pointer (SS: ESP) che punta in cima allo stack del nucleo, poi c'è un secondo registro del processore che è CS: EIP che sarebbe il program counter, che punta alla prossima istruzione da eseguire. C'è poi il registro EFLAGS che sarebbe la program status word (la parola di stato), poi nel processore ci sono tutti gli altri registri generali. Stiamo eseguendo codice utente di un processo, la prossima istruzione che dovremmo eseguire è y = y – 2 ; ma prima di eseguire questa istruzione arriva un'interruzione.

During



Lo stack pointer SS: ESP è cambiato perché punta in cima allo stack del nucleo. In cima allo stack del nucleo troviamo già inseriti dall'hardware lo stack pointer, il program counter e il registro di stato EFLAGS, quindi all'interno del nucleo il processore in maniera automatica ha memorizzato tutte queste cose. Dopodiché, sempre il processore in maniera automatica, ha modificato il valore del program counter CS: EIP che adesso punta all'handler. Tutto questo è successo in un attimo, al riconoscimento dell'interruzione, nessuna

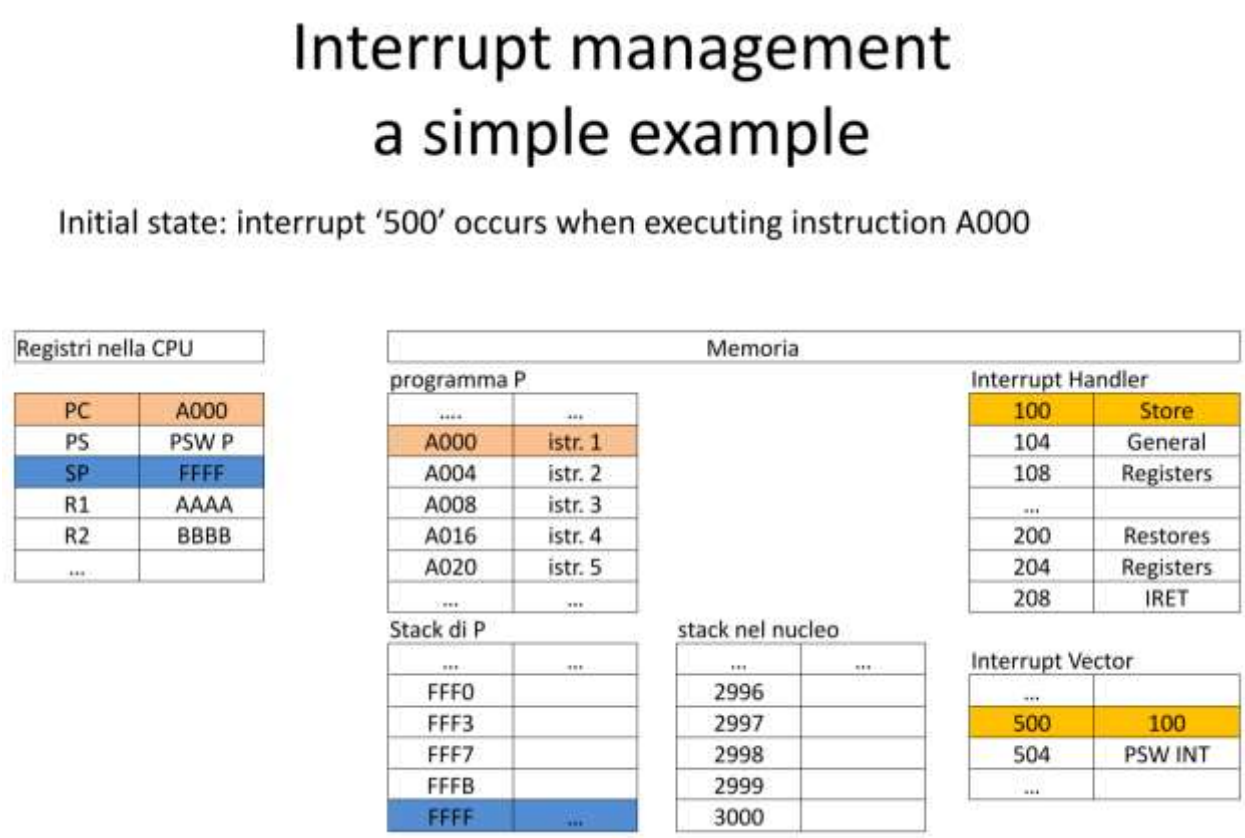
istruzione del linguaggio macchina è stata eseguita per far questo, quel passaggio dallo stato precedente a questo stato, da BEFORE a DURING, è avvenuto nel tempo di riconoscimento delle interruzioni da parte del processore, che prima di eseguire qualsiasi altra istruzione ha applicato questa trasformazione.
Nell'applicare questa trasformazione ha anche caricato nel registro EFLAGS (program status word) un nuovo valore prelevato dal vettore delle interruzioni. In questa parola di stato (EFLAGS) c'è codificato il fatto che il processore è in stato supervisore e che le interruzioni sono disabilitate. Fatto questo il processore inizia ad eseguire l'handler istruzione per istruzione. I registri generali non sono stati salvati da nessuno, quindi questi registri generali contengono ancora i valori del processo in stato utente, quindi per prima cosa l'handler deve prendere questi registri e salvarli in cima allo stack, non a caso la prima istruzione è "pusha", quindi copia il registro "a" in cima allo stack. Quando l'handler ha terminato il salvataggio dei registri generali e lo deve fare usando istruzioni del linguaggio macchina, la situazione è questa



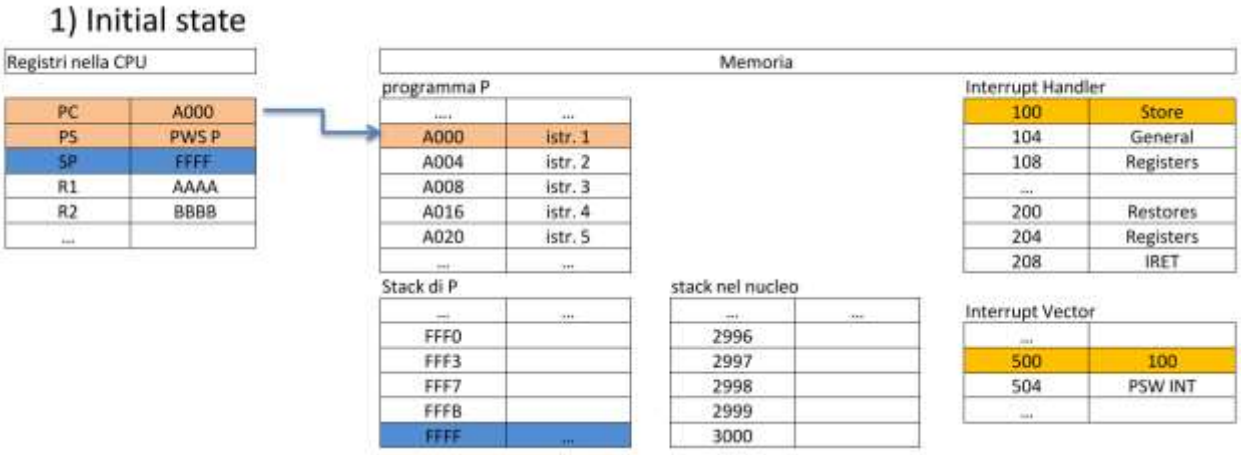
Qui ci sono i registri salvati dal processore e qui ci sono i registri salvati dall'handler. A questo punto l'handler ha salvato i registri generali ed è libero di svolgere il suo compito, quindi può capire perché è stata

generata un'interruzione, che cosa deve fare di conseguenza, che tipo di interruzione si tratta e metterla in esecuzione la parte alta del driver, quindi il thread che materialmente andrà a gestire quest'interruzione. A questo punto l'handler ha finito il suo compito. Che cosa deve fare ora? Deve fare tutto il contrario di quello che ha fatto fino ad ora quindi ripristinare tutti i registri, quindi prima di tutto salva i registri generali e poi di nuovo in un'unica operazione atomica ripristina lo stack pointer per puntare allo stack pointer utente, ripristina il program counter per puntare al program counter utente, ripristina la program status word perché prende il valore che aveva quando c'era in esecuzione il processo utente e nel far questo ripristina la modalità utente e abilita le interruzioni. Tutte queste operazioni, così come dovevano essere fatte con un'unica operazione atomica al riconoscimento delle interruzioni, devono essere fatte adesso con un'unica operazione atomica quando l'handler ha terminato il suo compito. Stavolta non c'è un meccanismo del processore che lo fa, è un'istruzione specifica che si chiama spesso iRet (ritorno da interruzione), è un'unica istruzione del linguaggio macchina che fa tutto questo. A questo punto vediamo un esempio di come avviene.

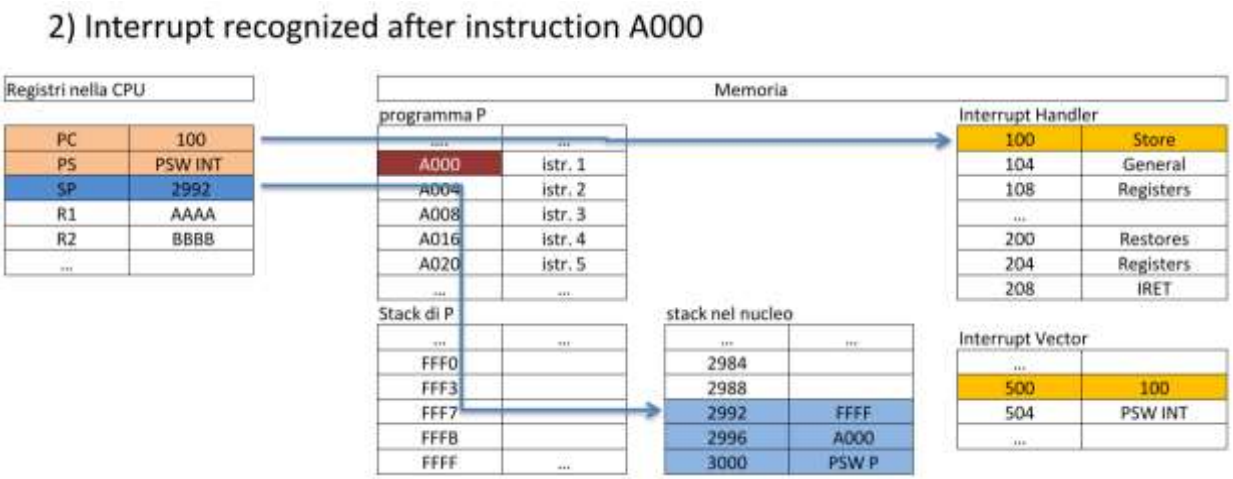
Esempio simil-compito



A lato ci sono i registri del processore, in particolare il processore ha i tre registri speciali PC(program counter), PS(program status), SP(stack pointer) e un certo numero di registri generali R1, R2, … . Accanto troviamo la memoria. In memoria ci sono diverse strutture: c'è il programma P che stiamo attualmente eseguendo, c'è lo stack del processo P che stiamo attualmente eseguendo (sono le due colonne "programma P" e "Stack di P"). Questa è una parte di memoria in stato utente, riservata al processo in esecuzione, poi da qualche altra parte in memoria c'è lo stack del nucleo che attualmente è vuoto, sempre da qualche altra parte in memoria nel nucleo, c'è il vettore di interruzione che è una tabella che associa ad ogni codice di interruzione un indirizzo e una program status word (PSW) da utilizzare e poi c'è da qualche parte in memoria nel nucleo l'interrupt handler, quindi la funzione del nucleo che deve essere eseguita quando arriva una certa interruzione.

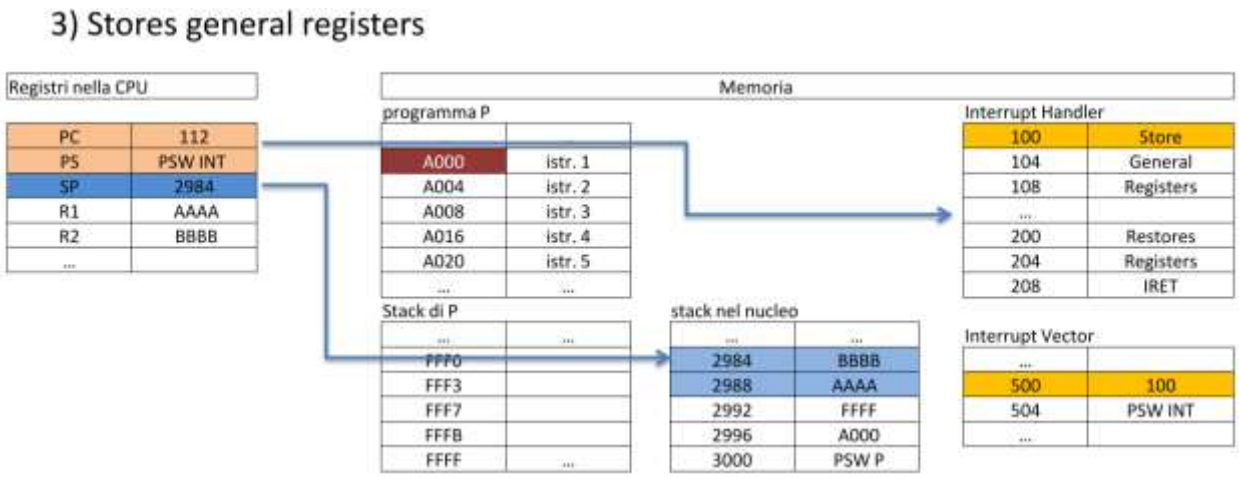


In questo momento il processo sta eseguendo l'istruzione A000, quindi sta eseguendo l'istruzione 1. Lo stack pointer del processo punta all'indirizzo FFFF. In questo istante, terminata, o durante l'esecuzione dell'istruzione A000 arriva un'interruzione, l'interruzione viene riconosciuta. L'interruzione viene riconosciuta quando terminiamo il ciclo di estrazione-esecuzione dell'attiva istruzione A000 ed iniziamo il ciclo successivo, quindi quando il program counter è diventato A004, quindi punta all'istruzione successiva, ma prima di andare a caricare quest'istruzione riconosciamo l'interruzione (è arrivata l'interruzione 500). Cosa succede? In alto c'è lo stato iniziale. Nel punto 2) c'è tutto ciò che ha fatto il processore quando ha riconosciuto l'interruzione:

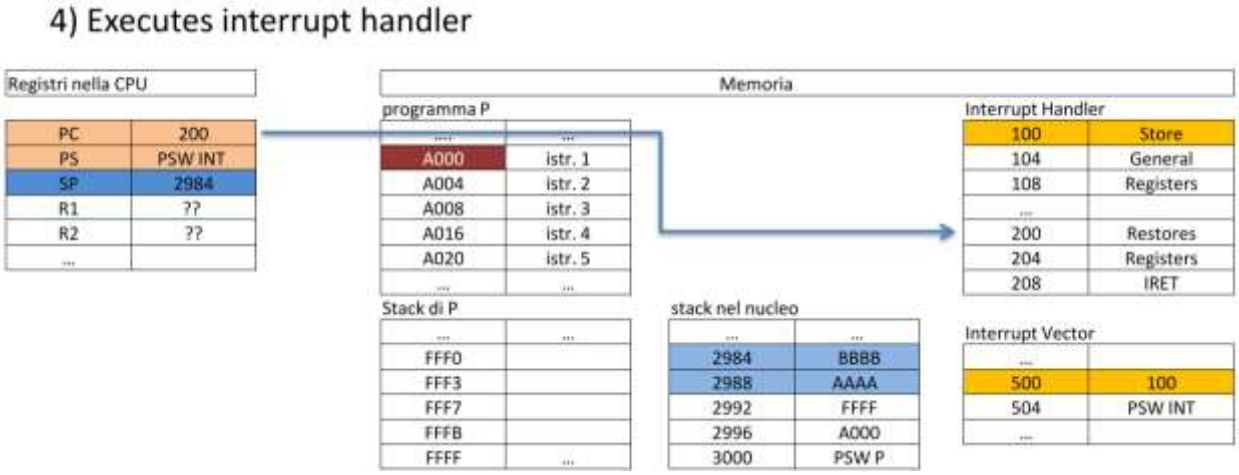


Quindi che ha fatto il processore? Ha copiato PC, PS, SP in cima allo stack del nucleo, PC non è più A000 ma è A004 perché il processore si stava apprestando ad eseguire l'istruzione 2. Quindi salvati i registri speciali in cima allo stack del nucleo, sempre il processore, ha modificato PC, PS, SP. SP l'ha già modificata perché deve puntare alla cima dello stack del nucleo. PC e PS invece li ha prelevati dal vettore delle interruzioni in corrispondenza con l'interruzione 500. Per effetto di questa modifica stiamo adesso lavorando dove? Il program counter punta alla prima istruzione dell'interrupt handler, la program status word(PSW) codifica il fatto che siamo in modalità supervisore a interruzioni disabilitate e lo stack pointer punta allo stack del nucleo. Va tutto bene perché il processo che era in esecuzione non ha subito danni nel suo spazio di memoria e i suoi registri, quelli che ho attualmente modificato, sono stati salvati.

In questa fase i registri generali R1, R2 continuano ad avere i valori AAAA e BBBB che erano i valori che stava utilizzando il processo quando era in esecuzione. Quindi se in questa fase l'interrupt handler inizia a modificare, ad utilizzare R1, R2 per scrivere, o per fare i suoi lavori, distrugge delle informazioni del processo. A partire da questo stato l'interrupt handler come prima istruzione salva i registri generali.

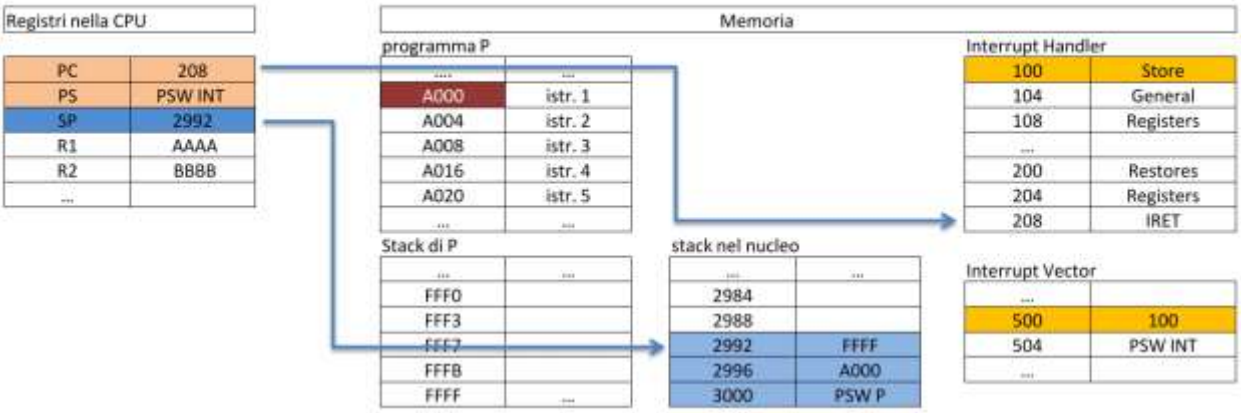


In alto c'è lo schema precedente al punto 2, in basso c'è lo stato che abbiamo quando l'interrupt handler ha eseguito tutte le istruzioni che salvano i registri generali. Quindi l'interrupt handler ha salvato R1, R2 in cima allo stack del nucleo, a questo punto i registri R1, R2 possono essere usati liberamente. Nel far questo ho eseguito delle istruzioni nel linguaggio macchina per cui il program counter è cambiato e punta all'istruzione successiva. Questa è la prima istruzione dove l'handler inizia a svolgere effettivamente il suo lavoro perché ha liberato il processore, ha liberato il registro del processore, ora l'interrupt handler può iniziare effettivamente a gestire l'interruzione 500.



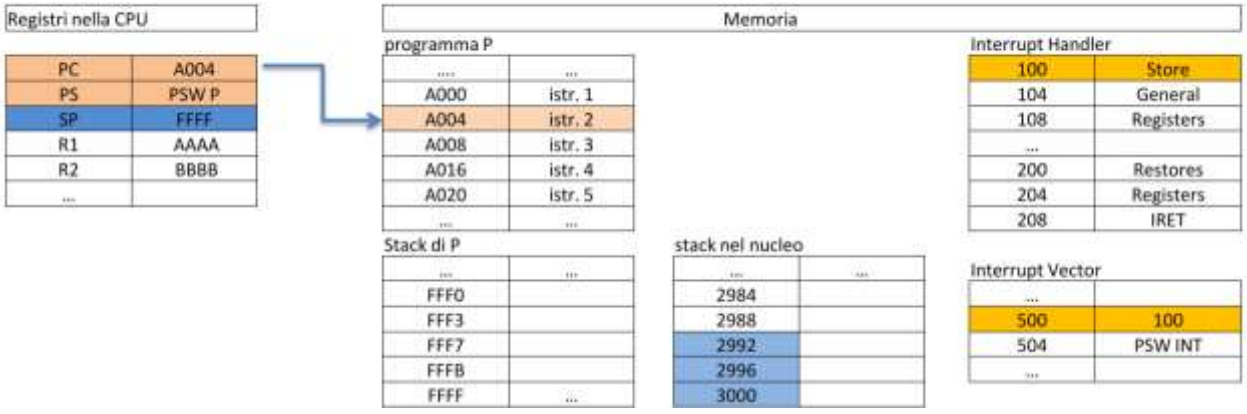
Per un po' di tempo il processore va avanti eseguendo l'interrupt handler, non sappiamo cosa fa materialmente durante l'interrupt handler perché non sappiamo che interruzione è questa: se è una interruzione che viene dal disco dovrò fare un lavoro, se è un'interruzione che viene da una stampa un altro lavoro, se è un'eccezione un altro, se una system call qualcos'altro. Non sappiamo cosa sta facendo l'handler in questa fase, sappiamo però che nell'eseguire queste operazioni il suo program counter verrà incrementato e ad un certo punto l'handler arriverà a completare il suo lavoro. Quindi arriva l'istruzione 200 dove l'handler ha completato il suo lavoro ma a questo punto deve predisporre tutto per riportare in esecuzione il processo che c'era prima, il processo p. Non può mandare in esecuzione subito il processo p perché i registri generali a questo punto hanno un significato che non si sa più qual è, hanno dei valori che non conosciamo e non hanno senso per il processo p. I registri speciali PC, PS e SP sono anche questi modificati, non hanno senso per il processo p, quindi ora come ora non possiamo subito mandare p in esecuzione. La prima fa l'handler a questo punto è ripristinare i registri generali, quindi dall'istruzione 200 alla 208 prende R1, R2 dalla cima dello stack dove erano memorizzati e li ricopia nei registri generali del processore R1, R2.

5) Restores general registers



Fatto questo resta da utilizzare l'istruzione iRet che in un'unica azione atomica ripristina PC, PS, SP. iRet prende PC, PS, SP dalla cima del nucleo e li copia all'interno del processore. L'ultima parte che resta da fare è eseguire l'istruzione iRet.

6) Executes IRET



Un istante dopo aver eseguito l'istruzione iRet, la situazione è: ha preso PC che era A004 che l'ha ricopiato nel program counter, ha preso PS che era PSW P e l'ha ricopiata nella program status word, ha preso lo stack pointer che era FFFF e l'ha ricopiato nello stack pointer. A questo punto abbiamo ripristinato esattamente la stessa situazione che aveva ilprocesso quando era arrivato l'interruzione quindi dal punto di vista del processo è come se non fosse avvenuto nulla, la prossima istruzione che il processore eseguirà sarà l'istruzione A004 che è l'istruzione 2 e viene eseguita con lo stesso identico stato del processore e della memoria prima dell'interrupt handler.

Sistemi operativi 02/03/17

Un altro meccanismo importante è quello delle chiamate di sistema, allora, in qualche modo se vi ricordate abbiamo un problema di protezione, quindi dobbiamo avere il modo di poter eseguire dei processi con dei diritti limitati per questo abbiamo un doppio stato nel processore: utente e supervisore (o user mode e kernel mode). In user mode il processore ha diritti limitati quindi non può eseguire istruzioni privilegiate, non può accedere a certi spazi di memoria, in modalità supervisore invece questi limiti non ci sono, il sistema operativo normalmente opera in modo supervisore, i processi utente operano invece in stato utente. Ovviamente questo ci pone un problema, ovvero che nel momento in cui mettiamo in esecuzione un processo in stato utente gli stiamo impedendo di interagire con gli altri processi o con il sistema operativo. Ovviamente quando siamo nel nucleo, e quindi in modalità supervisore, abbiamo il problema del dover garantire un passaggio sicuro verso la modalità utente per mettere in esecuzione i processi. Per questo secondo problema quindi per come passare da nucleo ai processi, quindi da supervisore a stato utente, abbiamo visto che c'è il meccanismo delle interruzioni, abbiamo visto che le interruzioni provocano questo passaggio. Resta però un'altra questione che è quella legata alle chiamate di sistema. Come fanno i processi in stato utente a richiedere i servizi al sistema operativo per utilizzare le risorse. Un processo operando in stato utente può generare indirizzi che stanno nel suo spazio di memoria, quindi lui non può fisicamente invocare una funzione del sistema operativo, se lui tentasse di chiamare il sistema operativo, dovrebbe saltare ad un indirizzo del nucleo e a questo punto si causerebbe subito un'eccezione di violazione della protezione. Questo è il primo problema, il secondo problema è che non possiamo permettere ai processi di invocare liberamente qualsiasi indirizzo del sistema operativo, perché un processo per sbaglio oppure per malizia potrebbe saltare in un punto arbitrario del sistema operativo e questo potrebbe causare dei disastri. Immaginatevi un vostro programma che anzi che iniziare dalla prima riga della funzione main inizia da una riga arbitraria in mezzo al codice, probabilmente produrrebbe dei risultati senza senso, e nella stessa situazione si potrebbe ritrovare il sistema operativo. Non posso invocare una funzione del sistema operativo in un punto arbitrario, ma devo partire esattamente dall'inizio di alcune precise funzioni che sono quelle che offrono i servizi per i processi. Quindi abbiam bisogno di queste due cose: ***Poter accedere alle funzioni del sistema operativo da dei punti di accesso predefiniti e ben definiti*** in modo che il processo non possa invocare in maniera totalmente arbitraria il sistema operativo, e ***dobbiamo contemporaneamente garantire il passaggio da stato utente a stato supervisore***. Il meccanismo che implementa tutto questo è il meccanismo delle chiamate di sistema. In generale qual è la situazione; immaginate di essere all'interno di un vostro programma utente, nella funzione main volete invocare certe funzioni che presuppongono l'utilizzo, per esempio, dell'hardware. Quante volte avete utilizzato delle printf? Tante immagino. La printf è una funzione che utilizza l'hardware, fa stampare qualcosa sul video, sullo schermo. Sappiamo che però allo schermo è associata una porzione di memoria protetta che può modificare solo il sistema operativo, quindi in realtà dietro a quella printf ci deve essere una chiamata di sistema al nucleo. Ora però voi in realtà scrivete printf con una serie di parametri di notazione C, è una invocazione di funzione non è un'invocazione di una chiamata di sistema; Che sta succedendo? Quella che voi utilizzate è un'invocazione di funzione ad una libreria C che sarà la stdio o stdlib piuttosto che una qualsiasi altra funzione di libreria del C. Quindi voi con printf invocate in realtà una funzione della libreria C che però in questo caso al suo interno contiene una chiamata di sistema. Una funzione di libreria che al suo interno viene contiene una chiamata di sistema in gergo viene chiamata STUB, sono funzioni che non hanno un ruolo particolare se non quello di predisporre l'invocazione di altre funzioni. In questo caso invocare il nucleo del sistema operativo è una cosa molto dipendente dal sistema, dall'hardware e via discorrendo… Per questo motivo, la chiamata di sistema viene mascherata all'interno delle funzioni di libreria così che il programmatore C non si deve preoccupare di farlo. In particolare questa funzione fa due cose: predispone il passaggio dei parametri che devono essere passati al sistema operativo, invoca la chiamata di sistema e poi fa la terza cosa che è ricevere il risultato dal sistema operativo e lo predispone per essere restituito al chiamante; quando fate una scanf avete qualcosa che ritorna in realtà, anche se implicitamente. Quindi voi vi limitate in realtà non ad utilizzare la system call ma ad utilizzare le funzioni di libreria. Come sono fatte

queste funzioni; prendono i parametri che passate, li predispongono in qualche maniera e poi chiamano / eseguono un'istruzione in linguaggio macchina che prende vari nomi a seconda del tipo di processore che stiamo usando. Che cos'è questa istruzione del linguaggio macchina (trap), intanto è un'istruzione non privilegiata perché può essere eseguita dai processi che vengono eseguiti in stato utente, quindi chiunque la può eseguire liberamente, anche voi nel vostro codice potreste scrivere questa istruzione assembler e causereste un'invocazione di una chiamata di sistema, l'unico motivo per cui non viene fatto nella programmazione è che è scomodo, però la potete usare liberamente. Allora questa istruzione del linguaggio macchina che fa? Setta un bit di interruzione, non uno come quello delle interruzioni che vengono dai dispositivi ma in realtà c'è una maschera che ci dice che tipo di interruzione è arrivata, segna un'interruzione proveniente da un processo quindi una chiamata di sistema sostanzialmente. Per effetto del settaggio di questo bit che succede; il processore fa il suo ciclo di fetch exexute e prima di eseguire l'istruzione successiva testa se sono arrivate delle interruzioni, scopre che è stato settato il bit di interruzione associato alle chiamate di sistema e quindi attiva il meccanismo delle interruzioni, esattamente lo stesso meccanismo che abbiam visto la lezione scorsa, uguale identico; quindi in base al quel meccanismo in modo protetto e sicuro, provoca il passaggio a stato supervisore, il salvataggio dei registri speciali, la disabilitazione delle interruzioni, e l'esecuzione dell'handler di quell'interruzione. Questo handler che cosa è? È una funzione del nucleo che gestisce le chiamate di sistema. Deve capire qual è la chiamata di sistema che è stata invocata (c'è ne sono tante; una per la printf, una per la scanf ecc. ma tutte quante vengono invocate settando lo stesso bit di interruzione) poi deve leggere i parametri associati a questa chiamata di sistema e a questo punto invoca la vera e propria chiamata di sistema che è un'altra funzione che sta nel nucleo e che svolge effettivamente quel compito, quindi questo handler è il punto di accesso a tutte le chiamate di sistema, che smista poi la chiamata verso la funzione corretta.

Domanda: Una volta che la trap setta l bit interruzione com'è che il processore capisce che è associata una chiamata di sistema e non un'interruzione dai dispositivi? Nel processore esiste un registro delle interruzioni; mettiamo che questo sia un registro a 8 bit, il registro è fatto così:

- Un certo numero di bit sono associati ad un certo numero di dispositivi, per esempio: i primi 6 bit sono associati alle interruzioni dei dispositivi
- 1 bit particolare per esempio il settimo è riservato alle chiamate di sistema
- 1 bit particolare per esempio, l'ottavo è riservato alle eccezioni, agli errori di protezione o alla divisione per 0, qualsiasi cosa rilevata dal processore

I primi 6 bit vengono settate da delle linee esterne provenienti dai dispositivi, il bit di interruzione associato alle system call, viene settato dall'istruzione trap (istruzione del processore), l'ottavo bit delle interruzioni è settato dal processore stesso quando rileva un errore. In realtà al processore non interessa minimante qual è la causa dell'interruzione, perché non la deve gestire lui. Lui guarda qual è il bit di interruzione che si è attivato, va nel vettore delle interruzioni e mette in esecuzione la funzione corrispondente, siccome abbiamo 8 bit, abbiamo un vettore di interruzione con 8 record, ogni record contiene l'indirizzo dell'handler associato a quella serie di interruzioni. Quindi il risultato è che siccome è stato settato il bit del numero, per esempio 7, associato alle chiamate di sistema, nell'interrupt vector si accede alla riga 7 della tabella e si salta a quel indirizzo che è dove risiede l'handler. Il processore non sa cosa sta eseguendo, perché il vettore delle interruzioni è impostato dal sistema operativo. Il meccanismo che il processore attiva è uguale identico a tutte le classi di interruzione. Che sia un'interruzione del dispositivo o una generata dalla trap, il meccanismo è quello che abbiamo visto la volta scorsa ed è sempre lo stesso. Appoggiandosi su quel meccanismo a questo punto abbiamo messo in esecuzione l'handler, l'handler come vi dicevo è la funzione del sistema operativo che deve gestire le chiamate di sistema e quindi che cosa fa; deve capire qual è la chiamata di sistema che il processo utente vuole chiamare e quali sono i parametri. E qui abbiamo un problema: come faccio a capire che l'utente voleva invocare una scanf, printf? Come faccio a leggere i parametri di queste funzioni? Bisogna che in qualche modo il sistema operativo ce li abbia, ci sono diversi

modi di passaggio delle informazioni dal processo utente alla funzione handler, alcuni sistemi mettono questi parametri nei registri generali del processore, per esempio convenzionalmente, il registro R1 del processore contiene un codice che rappresenta il codice della chiamata di sistema che voglio invocare; se nel registro R1 c'è scritto: 15 allora invoco la printf, se c'è 19 invoco la scanf e i parametri sono messi in altri registri R2 R3 ecc. In altri sistemi, questi parametri vengono messi dallo STUB in cima allo stack utente, lo STUB non ha accesso allo stack del nucleo, ha accesso solo allo stack utente, quindi mette queste informazioni in cima allo stack utente. Per cui l'handler o prende gli argomenti nel codice della chiamata di sistema dai re3gistri generali del processore oppure se li va a prelevare dallo stack utente. L'handler lo può fare perché essendo nel nucleo, sa qual è l'indirizzo dello stack utente, banalmente lo va a leggere da dove il meccanismo delle interruzioni lo ha salvato, e poi va ad accedere alla memoria utente. Quindi quale che sia il meccanismo a questo punto l'handler prende il codice delle interruzioni, prende i parametri e invoca la funzione che implementa quella chiamata di sistema, questa è un'invocazione normale di funzione. Quando quella funzione è terminata, (non è detto che venga svolta tutta ora) fa una return, ritorna il controllo all'handler che a questo punto prende i valori di ritorno da quella funzione e li predispone per essere passati al processo utente. Com'è che li passa al sistema utente, di nuovo, o li salva nei registri generali oppure li carica in cima allo stack utente. Ora, il vostro collega ha chiesto: Sì, ma quando c'è la trap si disabilitano le interruzioni quindi l'handler, le system call vengono eseguite a interruzioni disabilitate e poi si riabilitano con il return oppure fanno altro? Nei sistemi operativi fine anni '70, tutto il nucleo veniva eseguito a interruzioni disabilitate, questo però era molto inefficiente perché la chiamata del sistema poteva portare via del tempo, durante questo tempo non si potevano gestire le interruzioni provenienti dai dispositivi, che venivano ritardate e qualche volta perse. Nei sistemi moderni invece, le interruzioni vengono abilitate il prima possibile, per cui una volta che questo handler ha completato il prelievo dei parametri, può riabilitare le interruzioni e a quel punto può invocare la funzione. La funzione viene eseguita a interruzioni abilitate, quando la funzione termina, le interruzioni sono ancora abilitate l'handler prende questi parametri, inizia a ripristinare tutto quanto in maniera tale da predisporre l'esecuzione, ma quando fa questa predisposizione di nuovo dovrà disabilitare le interruzioni, perché deve iniziare a ripristinare i registri generali, dovrà iniziare a toccare tutta una serie di elementi critici, per cui nella fase finale disabilita le interruzioni e si predispone a fare la ret. Alla ret le interruzioni vengono abilitate nuovamente. Badate bene che quando riabilitate le interruzioni potrebbe scattare un'interruzione del timer, mettere in esecuzione un altro processo, invoca una chiamata di sistema che rinvoca questo handler, per cui voi avete un'istanza dell'handler arrivata ad un certo punto e una nuova invocazione dell'handler che sta ripartendo da capo e l'esecuzione di questo di sovrappone, potreste avere l'esecuzione di due, tre, quattro, dieci handler in contemporaneo, in concorrenza fra loro. Scrivere il sistema in modo che tutto questo mantenga un senso e funzioni correttamente non è banale. Per questo motivo i vecchi sistemi operativi tenevano le interruzioni disabilitate e non ci pensavano più.

Quindi riassumendo, che cosa fa l'handler?

- Individua gli argomenti
- li registra nello stack
- copia gli argomenti, se sono nello stack, sono nello stack utente, quindi deve copiare dalla memoria utente alla memoria del nucleo, ma così facendo si protegge il nucleo stesso (evitiamo che l'utente possa fare dei pasticci andando a scrivere qualcosa nel nucleo) e nel farlo controlla che i parametri siano ben formati (altrimenti eccezione e il processo viene terminato).
- Li valida
- Invoca la chiamata di sistema vera e propria
- Copia i risultati indietro

Che differenza c'è tra usare i registri generali o usare lo stack? Avete usato la scanf e forse vi siete resi conto che la scanf non ha un numero precisato di parametri. Quindi se voi usate i registri generali per

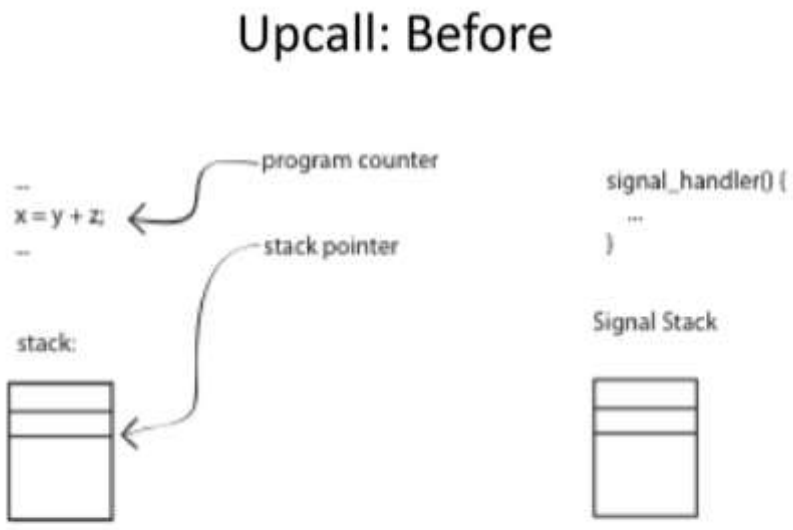
passare i parametri della scanf, i registri generali sono limitati, quindi, in teoria, questo pone un limite al numero di parametri che potete passare con la scanf, che però questo limite in realtà non ha. Invece infilare queste cose nello stack ci permette di gestire più comodamente funzioni e chiamate di sistema che hanno un numero variabile di parametri.

Vediamo come tutto questo viene utilizzato; le chiamate di sistema, sonio usatissime, i processi le usano molto pesantemente. Prendiamo come esempio, un web server, che cosa succede? Il web server viene attivato sulle richieste che arrivano dai client sparpagliati su internet i cui browser inviano una richiesta diretta. Quando il server viene attivato, la prima cosa che fa è quella di mettersi in attesa dei messaggi che arrivano dai clienti, come fa a mettersi in attesa? Invoco una chiamata di sistema, perché deve vedere se ci sono messaggi pendenti nella scheda di rete, questa è un dispositivo, deve essere usata dal nucleo quindi la prima cosa che fa il web server è una read su socket (struttura dati che rappresenta la scheda di rete), fa una richiesta di lettura e a quel punto, resta lì in attesa. Questa è una chiamata di sistema. A questo punto dall'interfaccia di rete arriva un messaggio, questo messaggio viene copiato dalla scheda di rete (viene quindi lanciata un'interruzione) e lo deposita in un buffer dei messaggi che vengono dalla rete e a questo punto si rende conto che è un messaggio destinato al web server. Quindi che cosa fa il nucleo a questo punto? Copia il messaggio nello spazio di memoria utente del web server e fa una iret, per restituire il controllo al web server, che analizza la richiesta (lo fa in modo utente), si rende conto che questa richiesta comporta una lettura di un file da disco, quindi invoca una chiamata di sistema per leggere il file. Questa chiamata di sistema, comporta da parte del nucleo una azione sul dispositivo "disco", il sistema operativo invia la richiesta al dispositivo disco, l'interfaccia del disco la esegue, quando ha completato l'operazione lancia un interrupt, il nucleo del sistema operativo prende questo interrupt, copia i dati letti dal disco in memoria utente del web server, perché ha capito che questo interrupt è associato al web server, quando ha terminato questa copia fa un iret, a questo punto il web server ha l'informazione da restituire al client internet, quindi (punto 9) predispone il messaggio di risposta ed infine lo spedisce (fa una write sul socket, una chiamata di sistema). Quindi un'operazione semplice come rispondere ad una richiesta e mandare la risposta, comporta l'uso di una certa quantità di chiamate di sistema e di interruzioni. Tutto questo voi l'avete fatto innumerevoli volte; avete usato le printf, avete scritto su file (fopen, fclose).

Un altro meccanismo che ancora non avete visto a laboratorio, ma che vedrete, è il meccanismo delle UPCALL. Nei sistemi operativi oltre a predisporre del meccanismo delle interruzioni che spesso serve per fare le chiamate di sistema, e quindi per chiamare da STATO UTENTE funzioni del sistema operativo in modalità Kernel, il sistema operativo offre anche la funzionalità inversa, quindi c'è anche la possibilità da parte del nucleo di invocare funzioni in spazio utente di un determinato processo. Questa sembra una cosa strana perché il sistema operativo in realtà offre dei servizi, il processo in spazio utente usa i servizi quindi per usare i servizi fa le chiamate di sistema. Questo invece è un meccanismo contrario, il nucleo invoca una funzione handler del processo utente quindi nell'invocarla passa da stato supervisore a stato utente e invochiamo un servizio che sta offrendo il processo utente. A cosa serve questo meccanismo? Intanto è usato molto (in UNIX si chiama meccanismo dei segnali) perché in realtà i processi hanno bisogna di cooperare tra di loro e questo è uno dei meccanismi che possono usare. Tramite il meccanismo dei segnali un processo può inviare una notifica asincrona ad un altro processo, provoca l'esecuzione asincrona di una funzione dell'altro processo. Questo è un uso che viene fatto dei segnali per permettere ai processi di coordinarsi, di sincronizzarsi fra di loro, ma c'è un altro uso importante delle upcall perché servono per implementare le macchine virtuali. Realmente; come fa un sistema operativo a girare su un altro sistema operativo? Uno dei meccanismi che vengono usati, è proprio questo delle upcall. Teniamo da parte momentaneamente la questione delle macchine virtuali, vediamo come funziona il meccanismo delle upcall prima. Le upcall sono intuitivamente delle interruzioni, ma funzionano al contrario, non sono interruzioni che arrivano al nucleo, ma sono interruzioni che arrivano ai processi. È strano perché un processo può essere impegnato a svolgere il suo compito scritto nel suo programma (un compito strettamente sequenziale), gli arriva una upcall che comporta l'esecuzione di un pezzo di codice totalmente scollegato

con quello che lui sta eseguendo, ma previsto nel codice del processo ovviamente (l'esecuzione asincrona di un handler) quando l'handler è terminato, il processo continua ad eseguire il suo codice come se nulla fosse successo. Normalmente come si usa l'handler? Quando viene eseguito, io capisco che è avvenuta una certa cosa, scrivo questa informazione in una struttura dati mia, in questo modo, tutto il resto del mio codice lo viene a sapere (uso tipico). Per cui il mio codice fa una certa cosa, quando arriva il segnale da un altro processo, dice "ok, ho completato questa certa operazione quindi tu puoi fare altro" , questa informazione viene in qualche modo modificata all'interno di una struttura dati del mio processo e termina lì, quindi il processo tiene traccia che quella upcall è stata ricevuta e ha avuto un effetto modificando lo stato del processo, a questo punto quando il processo ritorna alla sua esecuzione, non può sapere che l'upcall è stata eseguita, perché eseguita in maniera totalmente asincrona, il codice del processo che è stato eseguito prima e come se non si accorgesse dell'upcall, ma siccome è stato modificato lo stato del processo, prima o poi quando il processo andrà a leggere qualche struttura dati modificata, prenderà atto che quella upcall è arrivata e quindi si comporterà di conseguenza.

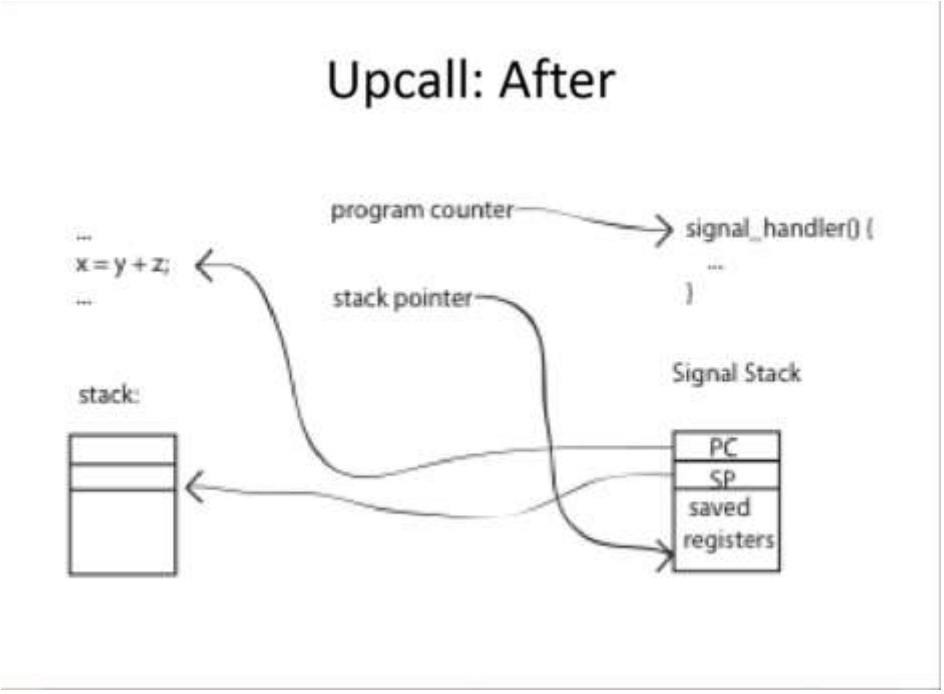
È un meccanismo che io uso tantissimo in giardino. Voglio predisporre l'orto. Prendo il concime e mi avvio, però mentre vado verso l'orto, mi rendo conto che sta crescendo un'erbaccia lì da qualche parte, allora cosa faccio? Io non voglio distrarmi a pensare all'erbaccia, altrimenti non faccio più nulla quindi, prendo uno strumento che serve ad estirpare l'erbaccia e lo butto lì vicino, e poi continuo a fare l'orto in giardino. Quindi la mia attività principale (quella di fare l'orto) in realtà è come se non fosse disturbata. Ho visto quella upcall(l'erbaccia) non l'ho servita subito ma ho modificato lo stato (lo strumento gettato) per ricordarmelo, vado a fare quello che dovevo fare. Quando poi ho finito il mio compito e sto pensando ad altro, ritorno lì e vedo che c'è lo strumento fuori posto, mi ricordo di estirpare l'erbaccia non ho altro da fare e lo faccio. Questo è il mio meccanismo di gestire le upcall, se usata in maniera esagerata, quello che succede èp che cominciate ad accumulare cosa da fare su cose da fare e vi dimenticate di quello che volevate fare in origine., nella realtà i processi non se lo dimenticano.



Come funziona?

Il nostro processo sta eseguendo il suo codice, tranquillo, è arrivato all'istruzione x=y+z, il suo stack punta al suo stack pointer (stato utente), e quello che lui farà un istante dopo sarà quello di eseguire la prossima istruzione.

Per un qualche motivo (fine del quanto di tempo) il processo viene interrotto qui. Lui non si rende conto di questa interruzione, perché come sappiamo, il meccanismo del time sharing garantisce che i processi vadano avanti senza rendersi conto di essere occasionalmente interrotti, quando poi verrà richiamato, riprenderà la sua esecuzione e lui sa che ripartirà dall'istruzione successiva a questa. Ora però che cosa succede? Passano in esecuzione altri processi sul nucleo. Uno di questi processi, o il nucleo stesso, decide di inviargli una upcall. Questo comporta l'esecuzione di questo handler, che cos'è questo handler? È una funzione dello stesso processo, (il programmatore che ha scritto questo codice, ha scritto anche l'handler). Il programmatore era conscio che questo programma avrebbe prima o poi ricevuto una system call e per questo ha predisposto l'hanlder . Il sistema operativo in questa situazione genera, l'upcall e realizza tutto questo meccanismo (vedi immagine). Allora, il program counter e lo stack pointer del processo utente li conserva nello stack delle gestione delle upcall(o stack dei gestione dei segnali). Quindi in qualche maniera, sta salvando lo stato del processo e lo sta salvando in uno stack che si trova in stato utente, non ha bisogna di salvarlo in stack del nucleo o altro, perché qui il nucleo non è interessato, l'interruzione viene verso il processo, è lui che la deve gestire. Il program counter e lo stack pointer vengono salvati in cima allo stack dei segnali e vengono modificati dal nucleo in maniera tale da puntare all'inizio del signal handler e alla fine dello stack; dopo il PC (program counter) e lo SP (stack pointer) vengono anche messi i registri salvati. Si sta creando un meccanismo simile a quello delle interruzioni, ma anzi che essere l'hardware a farlo è il sistema operativo che lo sta facendo. Quindi il sistema operativo prende PC ,SP, Program status word, registri generali del processo che erano in cima allo stack del nucleo; li copia nello stack utente riservato ai segnali. Nello stack del nucleo stavolta ci mette un PC e un Program status word, diverse, quelli associati all'handler e al signal stack, e fa una iret. Facendo una iret in queste condizioni, il programma del processo viene riattivato, ma l'istruzione che viene eseguita non è quella che segue x=x+z bensì è la prima istruzione del signal handler. Il signal handler prende atto del fatto che è arrivato questo segnale, modifica una propri struttura dai per tenere traccia che questo segnale è arrivato, a questo punto, quando ha terminato, ripristina i registri generali e poi fa un return. Il return comporta il ripristino di PC SP, che tornano ad avere i valori vecchi, per cui al return del signal handler, in realtà torna in esecuzione il codice originario così come se non fosse successo niente. Questo meccanismo permette di eseguire una funzione handler del vostro codice in maniera asincrona rispetto al resto del codice. L'handler può essere eseguito n qualsiasi momento e senza danni.



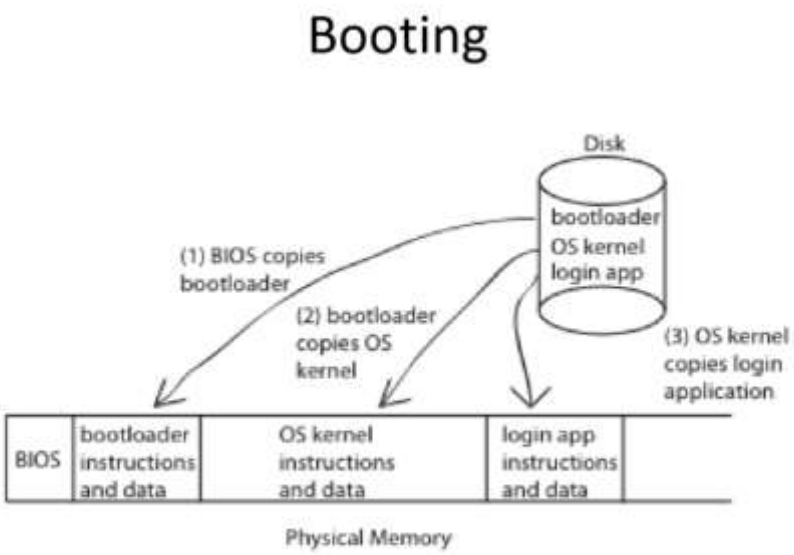
E nel caso fosse un processo a fare la upcall?

In realtà un processo non fa la upcall, il processo invia un segnale al nostro processo. Utilizzerà una chiamata di sistema per invocare un segnale al nostro processo e di conseguenza questa chiamata di

sistema sa che vogliamo invocare un segnale, sarà lei a fare la upcall. La upcall la esegue sempre il nucleo, o per conto del nucleo stesso, o per conto di un altro processo.

Restano due questioni di cui vi volevo parlare: uno è il **Booting** e l'altra sono le **Macchine Virtuali.**

Tutti i meccanismi che abbiamo visto finora, sono simili a prevedono un sistema che già funziona a regime; devo aver inizializzato il vettore delle interruzioni, devo aver predisposto gli handler ecc. Tenete presente che quando accendete la macchina, la memoria è vuota. Come si passa da uno stato di memoria vuota ad uno stato di sistema operativo a regime in modo da poter gestire le interruzioni ecc.? Tutto questo è affidato al boot(operazione che viene eseguita all'accensione).



Probabilmente avrete sentito parlare del termine BIOS. Il BIOS è una ROM, quindi una memoria a sola lettura presente in tutti i computer, piuttosto piccola e neanche particolarmente lenta, che contiene del codice molto semplice. Quando un computer viene acceso, il processore inizia automaticamente ad eseguire il codice del BIOS. Il codice del BIOS fa pochissime cose: va a guardare qual è il disco attivo (configurato da dei parametri all'interno dei dischi stessi), guarda nel disco qual è la partizione attiva (quella che contiene il sistema operativo) e di quel disco va a caricare il BOOTLOADER, che altro non è che un settore del disco che contiene una funzione ben precisa che contiene le istruzioni per caricare il sistema operativo. Il sistema operativo non viene caricato dal BIOS ma dal BOOTLOADER perché sul quel computer vorreste poterci mettere MacOS o Linux ecc. ed ognuno avrà le proprie specifiche metodologie per essere caricato, e chiaramente non volete metterle nel BIOS. Il BIOS ha un meccanismo di base per andare a caricare il BOOTLOADER, questo è specifico per il sistema operativo ma questo deve trovarsi sempre nel solito punto del disco. Il bootloader contiene istruzioni e dati per caricare il resto del sistema operativo. Una volta che il sistema è stato caricato in memoria, il bootloader passa la palla al main del sistema operativo, che avvierà tutta un'altra serie di funzioni, tra le quali farà anche partire il processo di login. E quel processo farà poi partire il processo di shell sul quale poi potrete dare i comandi. Una volta che siete arrivati a questo punto, tutto il resto funziona, perche il bootloader ha caricato il sistema operativo, questo predispone il vettore delle interruzioni e tutto il resto. Naturalmente alcune di queste operazioni devono essere eseguite ad interruzioni disabilitiate.

L'ultima questione di questi lucidi riguarda le macchine virtuali. Come vi dicevo prima le upcall sono necessarie per le macchine virtuali.

Nel mio caso ho Windows, lui lavora in modo supervisore mentre tutti i processi lavorano in stato utente. Ad un certo punto installo una macchina virtuale e su questa installo Linux. La macchina virtuale però non è il nucleo del sistema, in realtà è un processo che opera in stato utente e la macchina virtuale è un processore fasullo sul quale installo un sistema operativo nuovo. Questo sistema operativo ospite però,

affinché le cose funzionino correttamente, non si deve accorgere del fatto che è ospite; lui deve pensare di essere il vero sistema operativo. Quindi il sistema operativo ospite è convinto di operare in modalità supervisore e creerà dei processi che operano in stato utente. Mai per nessun motivo deve sospettare minimamente di essere eseguito in stato utente. Avete visto Matrix? Quando passate nel mondo virtuale dovete pensare di essere nel mondo reale. E qui non sono ammessi déjà-vu. La macchina virtuale ospitata, per funzionare correttamente, dovrà intercettare le chiamate di sistema dai processi utente della macchina virtuale ospitata, questi potranno fare delle chiamate di sistema per invocare questa macchina, che dovrà gestire le interruzioni dai dispositivi. *Questo codice è scritto per usare istruzioni privilegiate*. Come funziona in realtà il meccanismo e su che principi si basa? Supponiamo che un processo utente della macchina ospitata invochi una chiamata di sistema, sappiamo che viene invocata tramite l'istruzione trap che setta un bit di interruzione, il processore che di tutto questo non vuol sapere niente, nel ciclo di estrazione esecuzione capisce che è arrivata un'interruzione e salta all'handler in modalità supervisore a interruzioni disabilitate, la disgrazia è che l'handler è nella macchina virtuale. Primo problema. Supponiamo di aver risolto questo problema, in qualche maniera. Abbiamo messo in esecuzione l'handler, ma l'handler eseguirà istruzioni privilegiate per usare i dispositivi o magari quando avrà terminato il suo compito eseguirà un iret , che è un'altra istruzione privilegiata, ma è eseguita in stato utente quindi causa un eccezione che scatena l'intervento di questo sistema operativo che per quanto ne sappiamo dovrebbe abortire la macchina ospitata. Come funzione realmente il meccanismo della virtualizzazione, giocando pesantemente con il meccanismo delle interruzioni e delle upcall. Ma molto pesantemente. La macchina virtuale ha diritti utente, non può catturare le chiamate di sistema né le interruzioni, ma noi dobbiamo, al livello di macchina ospitante, creare le condizioni affinché anche le interruzioni e le chiamate di sistema vengano virtualizzate. Ovviamente la macchina ospitante deve sapere che quel processo appartiene ad una macchina virtuale, perché altrimenti non funzionerebbe.

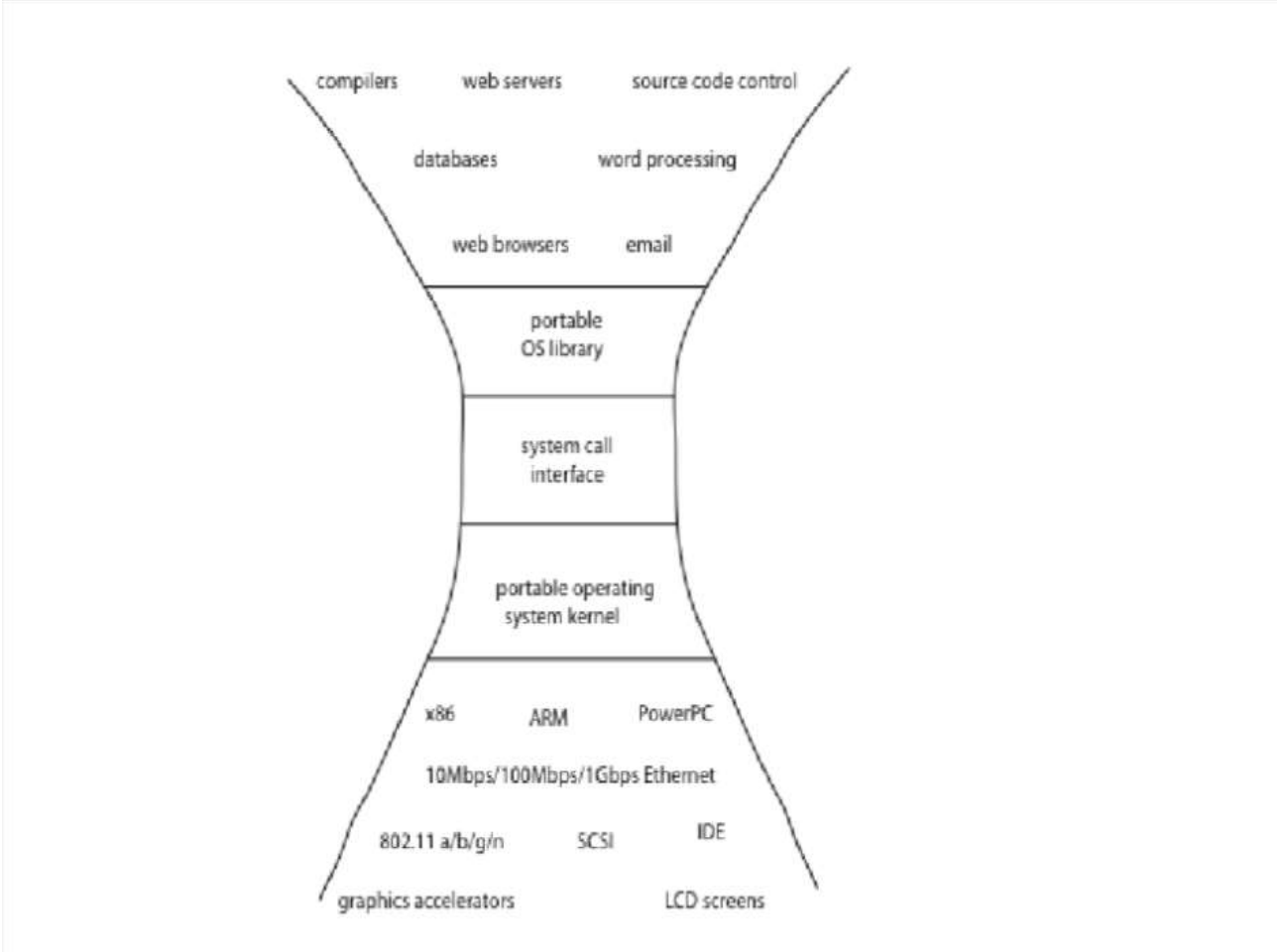
Immaginate che il processo in spazio utente invochi una chiamata di sistema, che genera un'interruzione che viene intercettata dall'interrupt handler di questo nucleo che però è impostato per la virtualizzazione sa che il processo che ha generato quella system call, in realtà, è un processo che sta girando su una macchina virtuale e sa che quella chiamata di sistema, non la deve gestire lui ma deve essere gestita dalla macchina virtuale , allora trasforma la gestione di questa interruzione in una upcall. Quindi dal suo punto di vista la gestione di questa chiamata di sistema non deve svolgersi con un compito che deve fare lui direttamente sui dispositivi, ma deve fare una upcall nei confronti dell'handler del sistema operativo nella macchina virtuale ospitata.

In questa maniera, questo sistema operativo ospitante può mettere in esecuzione l'handler della macchina ospitata. Ovviamente per farlo, deve sapere qual è l'handler da invocare, quindi la macchina virtuale deve avere configurato, non il sistema operativo ma la macchina visuale, nei confronti di questo nucleo in maniera appropriata i parametri e le upcall, in maniera tale da legarle alle upcall di questo sistema. Per sapere come legare le upcall, prima questo sistema operativo ha predisposto il suo vettore delle interruzioni nella sua memoria, quindi in realtà le upcall corrispondono al vettore delle interruzioni di questo sistema. A questo punto su questo sistema iniziamo ad eseguire l'handler, che dovrà eseguire delle istruzioni privilegiate, per esempio fare la iret. Allora però l'esecuzione dell'iret causa un'interruzione nel processore perché questo handler viene eseguito in stato utente, quindi nuovamente si creano un'eccezione che provoca l'esecuzione dell'interrupt handler all'interno del nucleo ospitante, capisce che questa eccezione è stata causata dall'esecuzione dell'iret, quindi si rende conto che è lecita e che non è un errore e esegue l'iret per conto di questo sistema. Fa lui l'iret predisponendo però nello stack del nucleo i parametri che mettono in esecuzione il processo al quale va mandata la iret, che è un processo della macchina virtuale. Stessa cosa se arriva un'interruzione dai dispositivi, per un operazione che è stata chiesta dalla macchina virtuale, il sistema operativo prende le informazioni, fa finta di essere l'hardware, quindi prende queste informazioni le copia in qualche buffer ma con upcall metterà in esecuzione il gestore delle interruzioni qua dentro, che prenderà l'informazione dove si aspetta di prelevarla, nel buffer

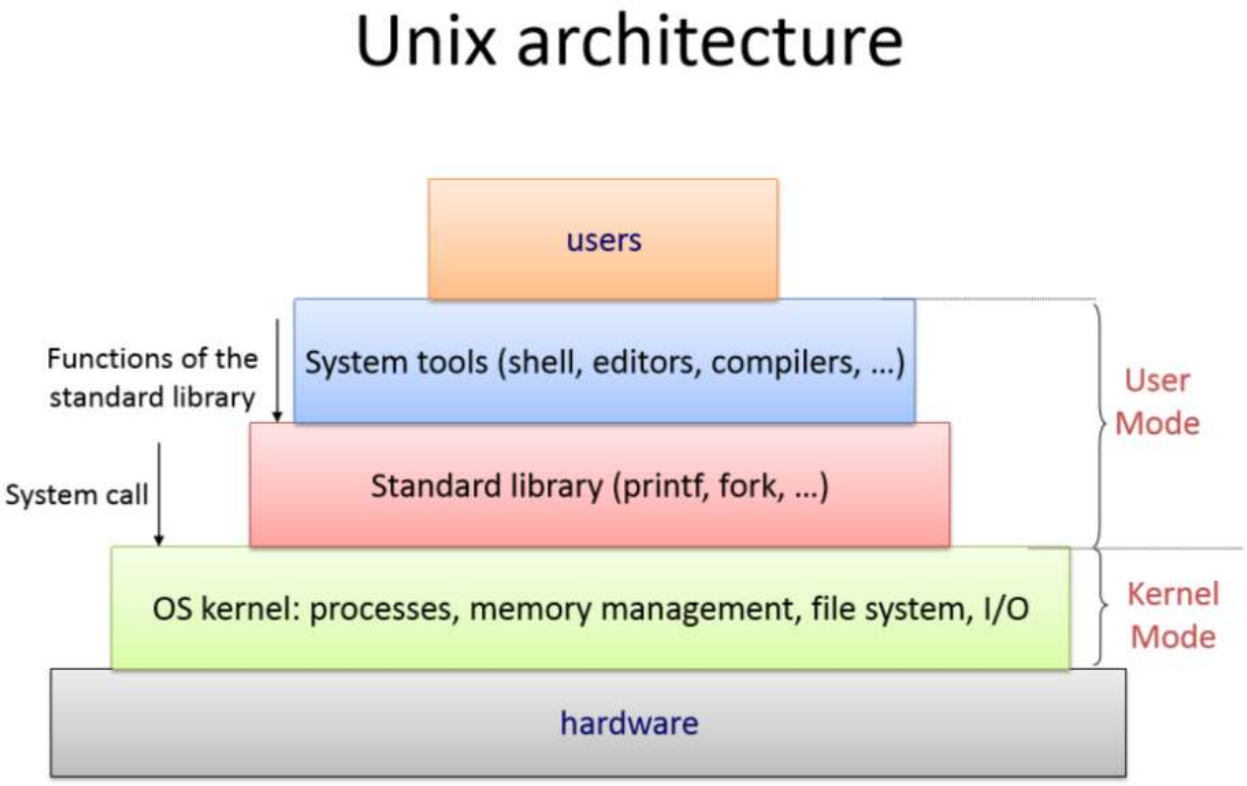
predisposto dal nucleo qua sotto e a questo punto gestisce queste interruzioni a livello utente. Tutto questo viene fatto così perché permette di fare una cosa molto utile; permette l'esecuzione di tutto questo sistema e dei suoi processi direttamente sul processore in maniera attiva, non devo metter tutto in una macchina virtuale simil java per eseguire tutte le esecuzioni in modo simulato, i processi fanno il loro lavoro alla piena velocità del processore, quasi non c'è penalizzazione. Gli unici ritardi intervengono quando ci sono chiamate di sistema, interruzioni o in generale ci sono da gestire operazioni che riguardano il nucleo e i processi in un unico hardware perché tutte queste vanno mediate dal nucleo vero.

Pausa

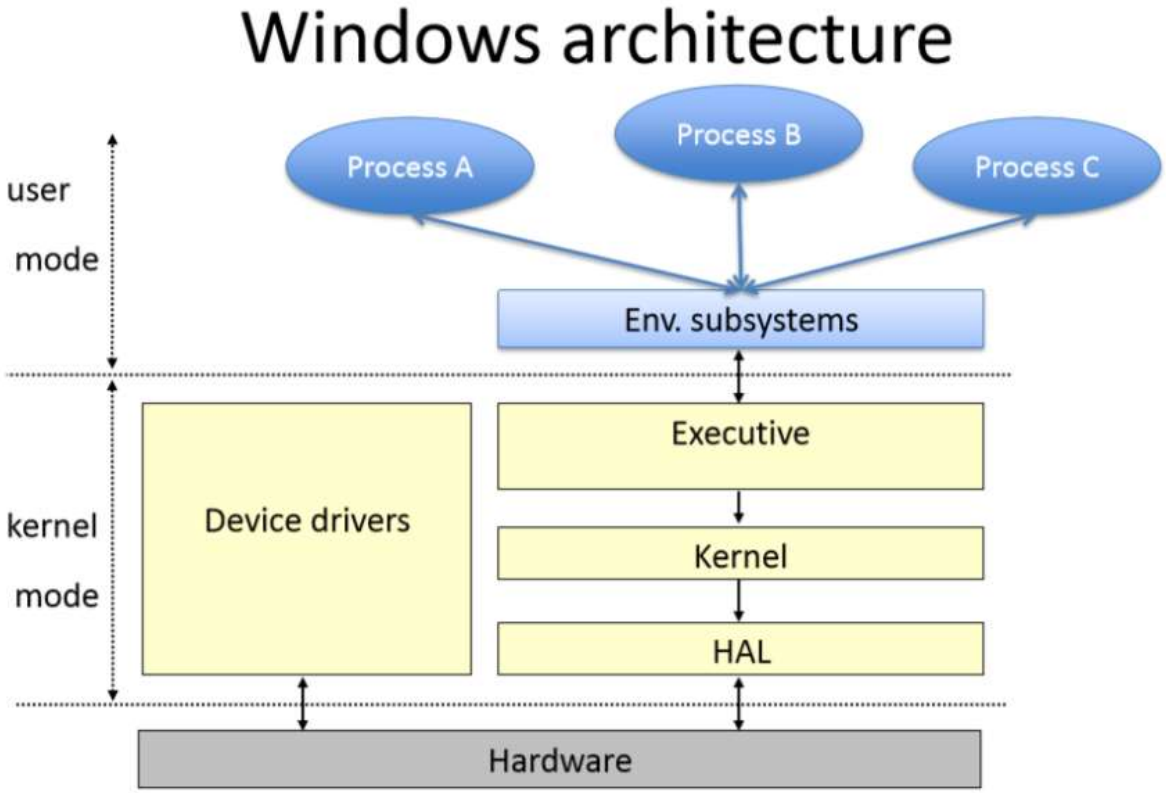
Vediamo un po' di prerequisiti che ci servono per affrontare poi meglio la parte che riguarda la concorrenza. Le cose che vi dico ora sono un'anticipazione di alcune cose che poi vedrete meglio e più in dettaglio a laboratorio, però alcuni concetti vi servono.



Noi sappiamo che il sistema operativo è sostanzialmente un condimento che situato tra due parti del panino una con le applicazioni utente in alto e l'hardware in basso. Qui abbiamo il nucleo del sistema operativo che deve essere anche portabile con tutta una sua struttura interna complessa, abbiamo un'interfaccia delle chiamate di sistema e sopra questa abbiamo una libreria, per esempio in C ne abbiamo a bizzeffe, che servono per contenere i famosi stub per semplificare l'accesso alle funzioni del sistema operativo. Generalmente l'architettura di UNIX viene mostrata con un disegno di questo tipo dove in basso abbiamo l'hardware poi abbiamo il nucleo del sistema operativo che è la parte più corposa che gira in modalità supervisore, e che contiene i meccanismi principali.



Sopra questo, in modalità utente, abbiamo la libreria standard, sopra di essa abbiamo gli strumenti (compilatori, editor, shell ecc.) e sopra di questo abbiamo i processi utente. Il livello delle chiamate di sistema è questa piccola interfaccia che sta tra le librerie e il nucleo. L'interfaccia tra gli altri livelli è data dal linguaggio C, dalle invocazioni di funzione. Se noi passiamo da Windows invece :



La struttura è un pochino differente, intanto Windows ha una notazione tutta sua per cui ciò che in Linux si chiama kernel in Windows identifica altra roba. Quello che in Windows funziona in modalità kernel è tutta questa che contiene: i dispositivi, una parte che è hardware dependance che serve a rendere il sistema portabile su diversi processori, poi quello che Windows chiama il kernel che contiene le funzioni del sistema operativo per la gestione dei processi della memoria della i/o e poi sopra c'è l'executive che serve a implementare le politiche di gestione. Sopra a tutto questo, questa linea tratteggiata in alto è la linea delle chiamate di sistema, sopra a questa linea ci sono degli oggetti che funzionano in modo utente e sono i sottosistemi d'ambiente che implementano vari sistemi e offrono librerie e via discorrendo.

Ovviamente per parlare di sistemi operativi è importante anche capirsi su quali sono le interfacce. Le interfacce e le loro funzionalità determinano come poi, in maniera molto netta, il sistema operativo offra certe funzionalità. È importante vedere tutto ciò che riguarda le interfacce e i processi. Partiamo da quello che fanno gli utenti quando utilizzano il sistema operativo, per esempio voi su Linux vi siete interfacciati con la shell, la shell è un processo che lavora in modalità utente e gira sopra il nucleo di Linux. Tramite questa shell voi potete fare delle cose molto potenti, potete invocare delle funzioni e dei comandi oppure compilare ecc. quando voi date un comando alla shell, per esempio il comando cc

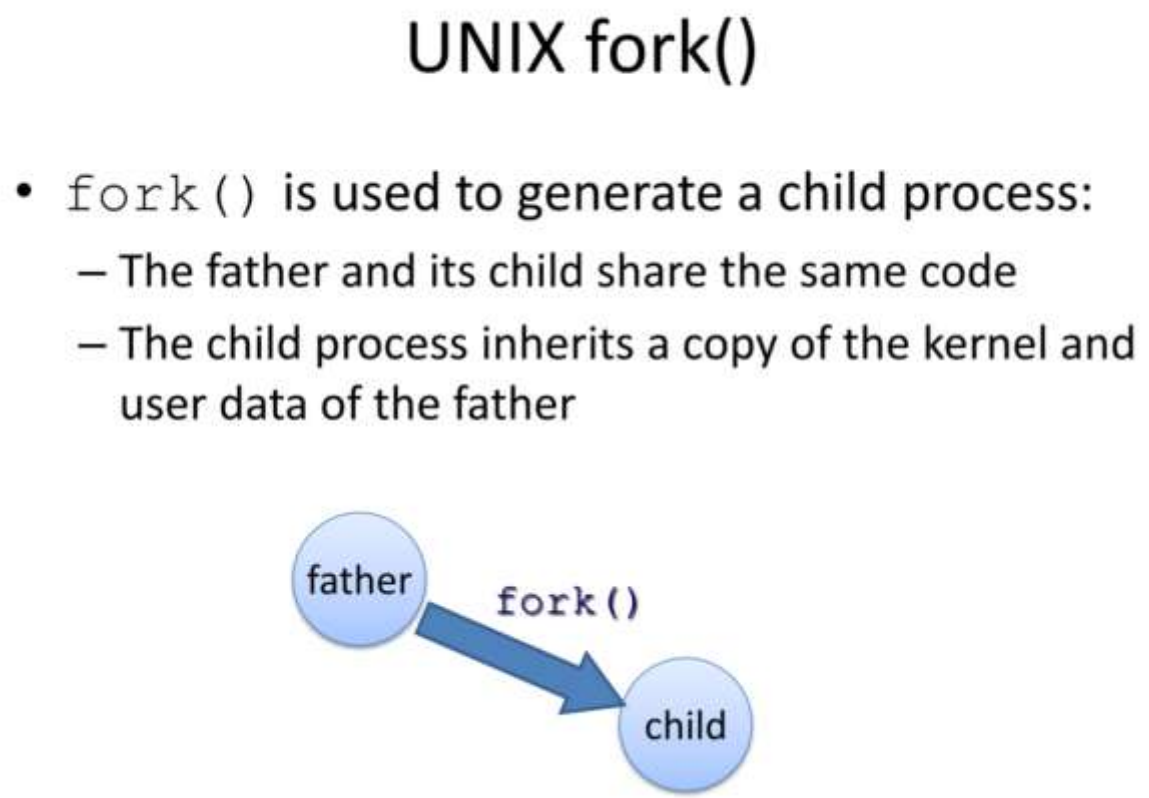
Shell

- A shell is a job control system
  - Allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells
- Example: to compile a C program

```
cc –c sourcefile1.c
cc –c sourcefile2.c
ln –o program sourcefile1.o sourcefile2.o
```

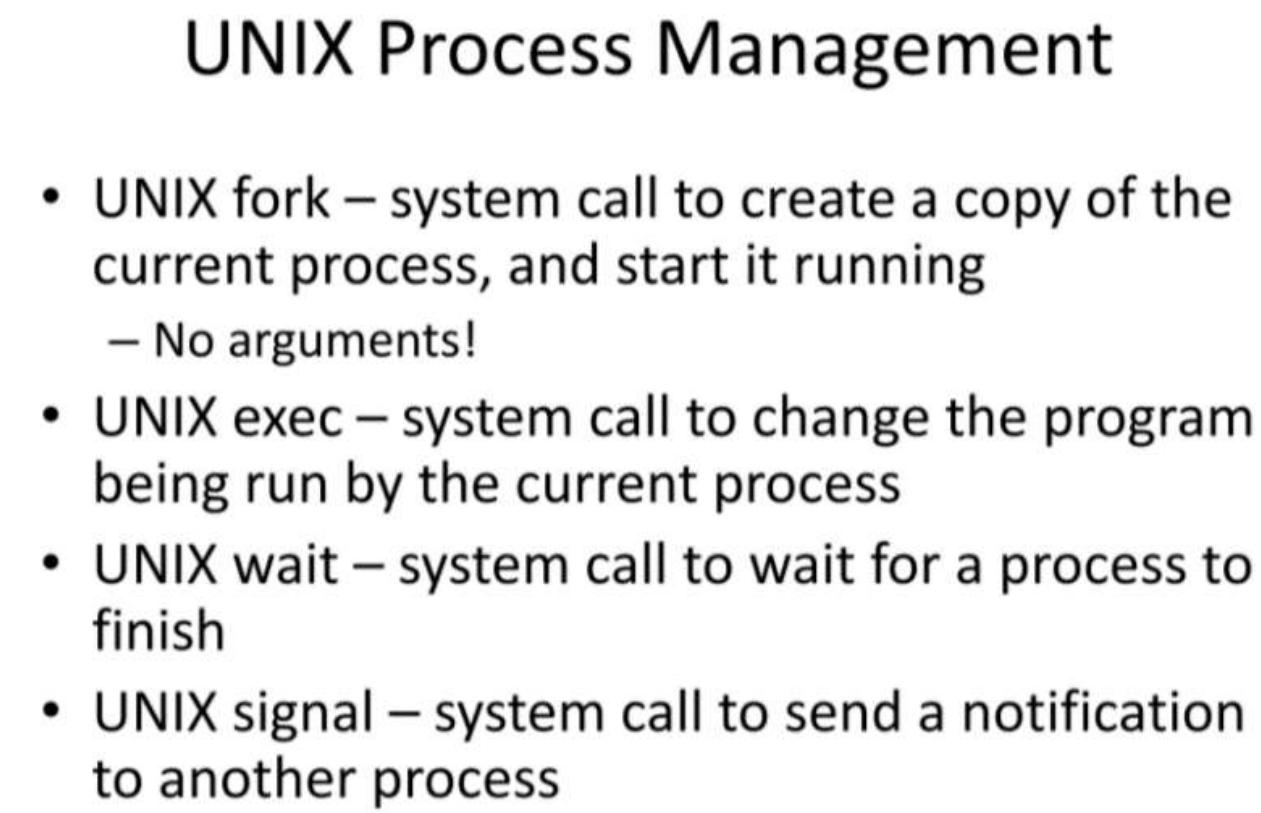
State chiedendo alla shell di mettere in esecuzione il compilatore, e il compilatore viene eseguito come un nuovo processo che compila il programma sourcefile1.c. Ora però il problema è che la shell è lei u processo che opera in stato utente e che si ritrova a dover invocare un altro processo. D'altra parte i processi sono delle astrazioni create dal nucleo, creare un processo comporta la generazione e l'inizializzazione di strutture presenti all'interno del nucleo, queste sono una serie di operazioni che solo il nucleo può fare. Come fa la shell a mettere in esecuzione un altro processo lo deve fare andando a chiedere al nucleo, per questo motivo il nucleo deve prendere delle chiamate di sistema che permettono di creare i processi e di gestirli. Nel caso di Windows, la chiamata di sistema che crea i processi si chiama CREATEPROCESS. Cosa comporta creare un processo? Comporta la creazione e l'allocazione di una struttura dati nel nucleo per poter rappresentare il processo, comporta l'allocazione in memoria dello spazio per il processo e dobbiamo proteggerlo, e dobbiamo copiare all'interno di questo spazio il codice del processo, dobbiamo informare lo schedulatore che questo processo esiste, però tutto questo non basta. Guardate l'esempio prima. Qui abbiamo messo in esecuzione il processo cc ma gli abbiamo passato degli argomenti. Quindi non basta soltanto creare il processo, ma bisogna passargli anche gli argomenti. E non è finita. Il comando cc che cosa fa, restituisce errore se non trova il file. I processi non hanno bisogno soltanto della memoria, della struttura dati e degli argomenti, ma hanno bisogno di tutta una serie di impostazioni nell'ambiente; devono sapere qual è l'utente che li sta eseguendo, che diritti ha di esecuzione per evitare di far danno al sistema operativo. Devono sapere qual è la directory sulla quale devono operare, qual è quella di default. Devono sapere talmente tante cose che io non le so tutte, però il risultato è che se voi andate a vedere la createprocess Windows ha una quantità di parametri incredibile. Perché quando create un processo dovete dargli tanti parametri. Il risultato è che creare processi con questo tipo di interfaccia risulta molto complesso.

L'approccio invece di UNIX è molto più semplice. È radicale nel senso opposto, per evitare di dover definire tutte le proprietà al processo, nell'atto della creazione dei processi in unix si dice i prendere tutti i parametri del processo creante. Il meccanismo di unix per creare processi che si chiama FORK, crea dei cloni. Quando un processo invoca una fork, crea un processo figlio identico in tutto e per tutto al processo padre. Prima di addentrarci nel meccanismo vediamo quali sono i suoi grossi vantaggi.



La chiamata di sistema fork di unix non ha parametri, non ne ha bisogno, perché tutti i parametri di configurazione del figlio sono già noti, basta che il sistema operativo vada a prendere tutte le informazioni dal padre. D'altra parte, creare un clone di un processo esistente non ha molta utilità. Normalmente io creo un processo perché voglio fagli svolgere un compito. Nel caso precedente quando noi in shell digitiamo *cc -o nomefile vorremmo* creare un processo che esegua un compilatore, non un'altra shell. Il meccanismo di fork se si limitasse a questo sarebbe inutile, infatti il problema è risolto abbinando alla fork una seconda chiamata di sistema che si chiama EXEC. Il modello di creazione dei processi unix prevede un doppio passaggio : prima creo un processo clone identico al padre , poi il processo clone evolve tramite exec andando a modificare il suo codice. In questo modo semplifico perché tutti i parametri legati all'ambiente restano gli stessi, il processo modificato con l'exec continua a mantenere tutte le proprietà legate all'ambiente del processo precedente. Stessa directory di lavoro, tutti i parametri ecc. L'unica cosa che cambia è il codice, la exec deve specificare il nome del file da eseguire e i suoi argomenti e di conseguenza il processo viene riscritto, la zona di memoria viene ridimensionata e re inizializzata con il nuovo codice. In

più ci sono almeno altri due meccanismi legati alla terminazione. Sono WAIT e EXIT.

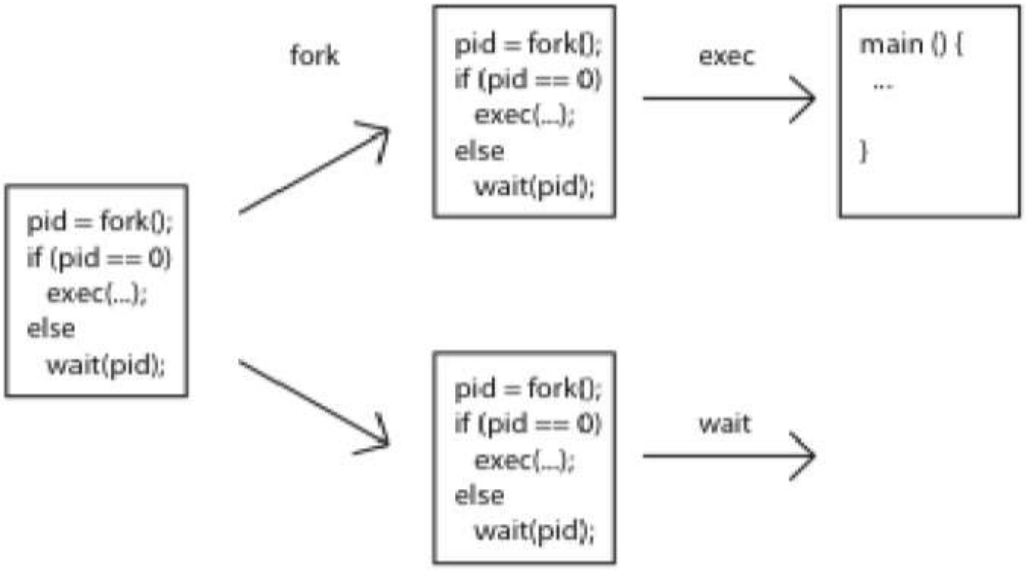


La fork clona i processi. Il processo figlio condivide: stesso codice, stesso stack, stessi parametri, stesso program counter, stesso tutto. La fork non prende parametri ma restituisce un numero. Nel momento in cui faccio la fork, quando termina restituisce il controllo al padre che avrà bisogno di sapere qual è il codice del figlio, infatti la fork al padre restituisce l'identificatore del processo figlio però una volta che la fork è terminata nel sistema è presente un processo figlio che farebbe le stesse cose del padre da dopo la fork.

Quando il processo figlio viene messo in esecuzione, viene messo in esecuzione dalla stesso in cui il padre era arrivato ad eseguire il codice, quindi dal codice della fork per cui quando il codice della fork del figlio termina il processo figlio si ritrova ad eseguire l'istruzione successiva alla fork. A questo punto ci serve un meccanismo per capire qual è il processo padre e qual è il processo figlio, quindi al padre viene restituito l'identificatore del figlio, al figlio invece, viene restituito 0. Analizzando il risultato della fork subito dopo il figlio si accorge che è figlio e il padre di essere padre, il figlio può eseguire l'exec e quindi cambiare il suo codice, il padre può fare altro. Normalmente questo meccanismo si usa in questo modo:

- Il processo padre prende il risultato della fork e lo mette in un parametro PID
- Guarda il parametro
- Se PID è 0, allora è figlio e può chiamare la exec
- Se PID > 0, allora è il padre e può continuare
- L'ordine in cui i processi proseguono è sconosciuto e dipende dal sistema operativo

UNIX Process Management



La fork può restituire errore ?

- Per esempio quando non c'è più memoria per allocare processi

E la exec ?

- Se il programma non esiste oppure non è eseguibile o non avete i diritti per eseguirlo.
- Non c'è memoria perché il processo che andate ad eseguire potrebbe essere più grande del codice del figlio.

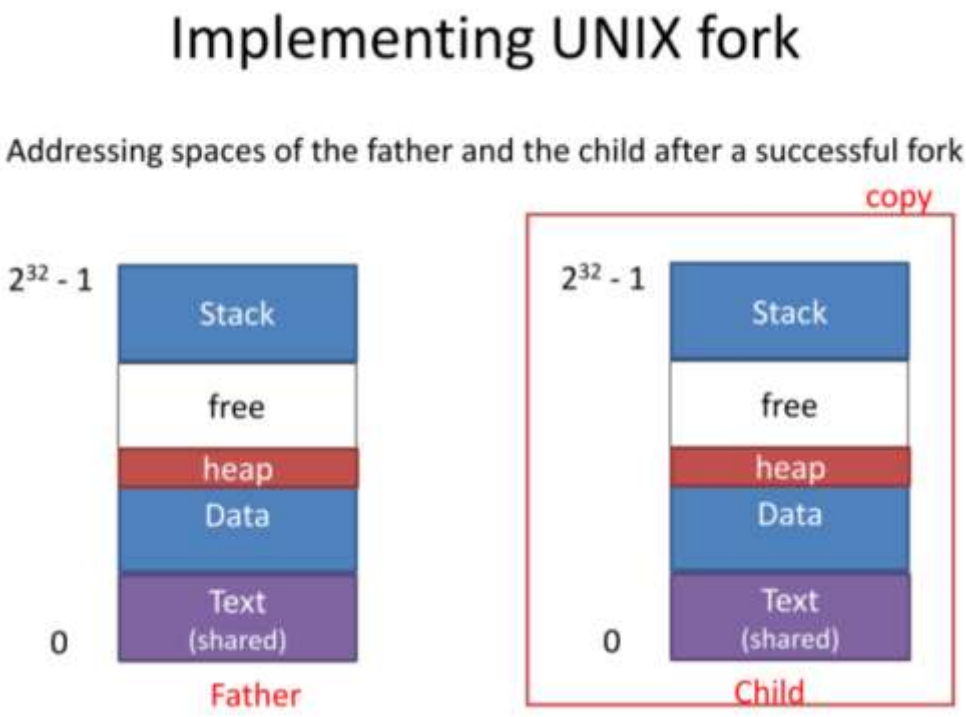
Se volete mandare in palla un sistema unix scrivete questo codice :

while (true){ fork(); }

Per questo motivo le ultime versioni di unix mettono un limite al numero di fork che un processo può fare.

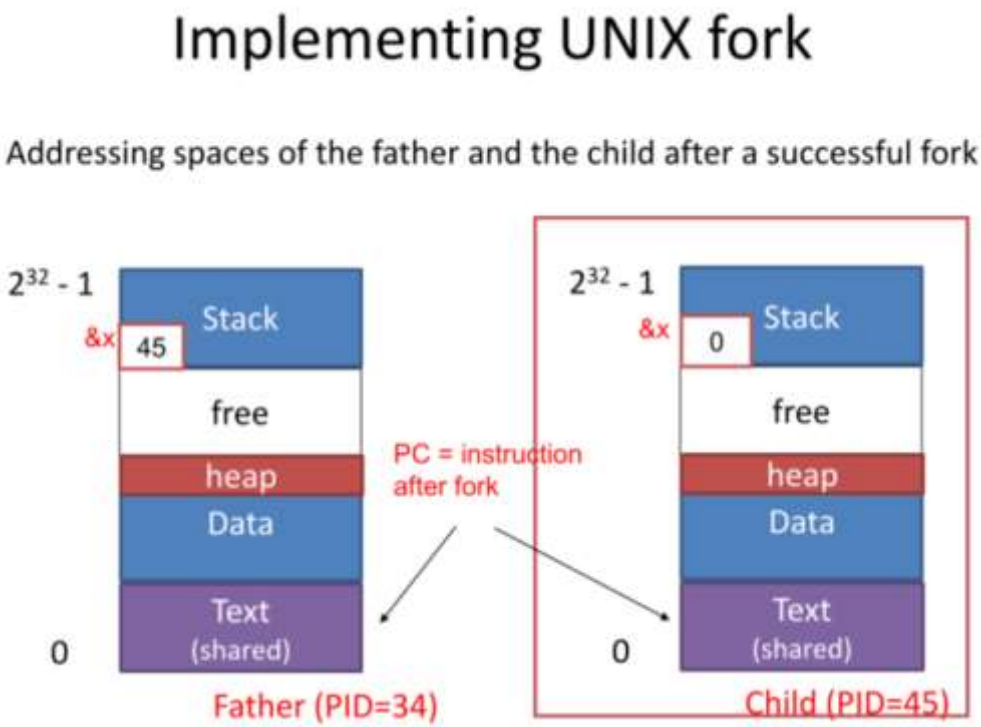
Come si implementa la fork? Dal lato della fork, la prima cosa da fare è allocare una struttura dati che è il descrittore del processo (PCB) per contenere tutte le informazioni del processo utente del figlio, in unix il PCB non è l'unica struttura ma usa due strutture : Process structure e user structure. Una volta allocate queste strutture nel nucleo, vengono inizializzate copiando pari pari il contenuto delle stesse strutture del padre. Qualsiasi fossero le informazioni associate al padre verranno copiate nel figlio, ad eccezione del PID. Dopo di che la fork alloca nuovo spazio in memoria per il processo figlio e prende tutta la memoria del padre e la ricopia nel figlio (byte a byte). Tutto questo ha l'effetto, di fatto, di creare una clonazione perfetta, manca un ultima cosa : dobbiamo restituire il PID del figlio al padre e viceversa. Quindi una volta che ha fatto questa clonazione, in cima allo stack del padre viene scritto il codice indentificato del figlio , in cima alla stack del figlio viene scritto il valore 0. Fatto questo abbiamo terminato e possiamo informare lo

scheduler che c'è anche il processo figlio e che deve metterlo in esecuzione. Per cui quello che succede è



questo:

Questa è la memoria del padre con codice, dati spazio libero e stack, viene interamente duplicata nel figlio e poi viene inizializzato diversamente la cima dello stack.



Siccome il program counter del padre e del figlio è lo stesso, entrambi vengono eseguiti in uno spazio di memoria virtuale tutti e riprenderanno ad eseguire il codice esattamente dallo stesso punto (alla terminazione della fork). Per quanto riguarda l'exec, deve prendere un parametro, un mone di un programma da eseguire, e deve riscrivere il codice del figlio inizializzando i dati. È importante che l'exec non

crei un nuovo processo ma riutilizza il processo esistente, ne cambia solo il suo codice. Eredita anche il PCB, ma però cambia lo spazio di indirizzamento che corrisponde allo spazio di indirizzamento del programma che stiamo eseguendo. La exec se ha successo non ritorna niente, perché non ha senso che restituisca un valore perché dopo la exec il processo inizierà ad eseguire la funzione “main” del programma che abbiamo messo in esecuzione e il main non si aspetta nessun valore. Se invece fallisce restituisce un codice di errore e in questo caso ovviamente il processo può gestire. Mantiene anche le risorse del padre, se il padre aveva aperto un file il figlio si ritrov a lo stesso file aperto anche dopo l’exec.

Sistemi operativi 06/03/17

Nella scorsa lezione abbiamo visto la fork. La fork è un meccanismo di unix per creare un processo, la fork non prende parametri perché il processo viene generato identico al processo generante, quindi rispetto al processo padre con delle piccole differenze legate al fatto che al processo figlio viene restituito l'id 0 mentre invece al processo padre la fork restituisce l'indentificatore dei figli. Per ora tutto questo semplifica la definizione, la catena dei processi, perché permette di evitare al programmatore di dover esprimere tutta una serie di informazioni legate ai parametri di funzionamento dei processi, quindi non bisogna definire niente perché il figlio eredita tutto questo dal padre. Ovviamente non ha molta utilità creare un processo identico ad uno già esistente, perché lui cosa fa: fa le stesse cose del padre e non è particolarmente utile. In realtà c'è il secondo meccanismo che è il medesimo della exec tramite il quale un processo può sostituire il proprio codice. L'exec prende come parametri il nome di un file eseguibile (*path_name), una serie di argomenti terminati dall'argomento uno e restituisce un intero che è un codice di errore. Però in realtà se la exec ha successo ovviamente non restituisce niente al processo invocante perché un istante dopo aver invocato la exec il processo ripartirà ad eseguire un codice completamente nuovo, a partire dal main, quindi ovviamente il valore di ritorno della exec non ha nessun siginificato. Il valore di ritorno della exec sta nel significato soltanto se ci sono degli errori, soltanto se ci sono dei problemi. In questo caso il codice non viene riscritto, il processo continua l'esecuzione sul vecchio codice che dovrà ricevere l'errore della exec e gestirlo. È importante che dopo la exec il processo resta lo stesso, quindi mantiene lo stesso PID, lo stesso codice e quindi di conseguenza resta nella stessa gerarchia padre-figlio rispetto al padre, mantiene lo stesso Process Control Block, quindi nel caso di UNIX, stesse strutture processo e utente. Quindi non bisogna riallocare niente all'interno del nucleo per questo processo però vengono azzerati i segnali pendenti, quindi se ci sono delle UPCALL che qualcun altro ha inviato a questo processo nel frattempo, queste vengono azzerate, lo stack all'interno del nucleo viene lasciato, quindi il processo continuerò ad utilizzare lo stesso stack all'interno del nucleo, lo stack al livello utente verrà azzerato e re-inizializzato, magari anche riallocato perché stiamo partendo ad eseguire un nuovo programma, un nuovo main e l'altra cosa importante è che mantiene le risorse assegnate, i file aperti. Per effetto del fatto che mantiene allocati gli stessi file aperti in precedenza, anche dopo l'exec in realtà il processo figlio mantiene la capacità di comunicare col padre tramite dei canali di comunicazione (lo vedremo in laboratorio quando faremo le Pipe).

La fork e la exec riguardano i meccanismi di creazione, ci sono poi i meccanismi di terminazione. In particolare ci sono diverse cause di terminazione: un processo può terminare perché commette degli errori, ad esempio viola i meccanismi di protezione, oppure può terminare per altri tipi di errore (divisione per zero), oppure può terminare esplicitamente invocando la chiamata di sistema exit. Quando il processo termina, che termini per un errore o che termini per la exit, in (MI?) UNIX, deve restituire un codice di terminazione al processo padre. Ovviamente non può imporre il processo padre di riceverlo questo codice di terminazione, però ogni processo che termina deve restituire questo codice di terminazione. Nel caso della exit il codice di terminazione (un codice scelto dall'utente), tramite questo codice il processo può comunicare qualcosa al padre (se è terminato con successo o se c'è stato qualche tipo di problema), ma il codice di terminazione lo sceglie il programmatore. Se invece il processo termina a causa di un errore, per una violazione di protezione, in questo il codice di terminazione viene fissato dal sistema. Questo codice di terminazione viene restituito all'atto della exit e il padre lo può ricevere tramite la chiamata di sistema Wait. Ci sono diverse situazioni che si possono verificare, legate alle tempistiche di esecuzione della Wait e della exit. Può capitare che il figlio invochi la exit ma il padre non ha ancora invocato la Wait, può capitare che il padre abbia invocato la Wait ma nessun figlio abbia fatto ancora la exit, può capitare che un figlio faccia la exit però nel frattempo il padre ha già terminato per cui non può fare la Wait, oppure può capitare che il padre non faccia mai la Wait per niente. Tutto questo in unix viene gestito in diverse maniere, siccome unix dal sistema operativo deve garantire che il codice di terminazione raggiunga il padre quando il padre lo richiederà, il meccanismo funziona in questa maniera: nel momento nel quale il processo fa la exit, il sistema operativo va a verificare se il padre ha già fatto la wait: se il padre non ha fatto la wait allora la

exit resta sospesa, quindi il processo non può ancora terminare definitivamente perché deve conservare all'interno del suo process control block, all'interno delle sue struttre del nucleo, deve continuare a conservare il codice di terminazione. Nel momento nel quale il padre farà la Wait questo codice potrà essere passato, il processo potrà terminare a tutti gli effetti. D'altra parte il processo che è in questo stato (che si chiama stato di Zombie, quindi sospeso tra la vita e la morte), il processo in questo stato in realtà non ha più niente da fare, il codice è terminato, quindi non ha più nessuna utilità mantenere allocata la memoria per il codice, la memoria per i dati. Quindi quando si fa la exit, del processo si può de-allocare tutta la zona di memoria riservata in spazio utente in maniera tale da renderla disponibile per altri processi, non si può ancora de-allocare le strutture dati del nucleo che descrivono il processo, quindi il processo continua ad esistere, non può più essere eseguito. Quando poi il padre farà la Wait, a quel punto il sistema operativo si renderà conto che il processo aveva già fatto la exit in stato di zombie quindi preleva dal descrittore del processo il codice di terminazione, lo restituisce al padre e a questo punto il processo figlio può essere de-allocato completamente e smette di esistere. Viceversa, se è il padre che fa prima la Wait ma il figlio la Exit ancora non l'ha fatta, il padre resta sospeso, quindi il padre non può proseguire la sua elaborazione, si interrompe, si sospende, quando poi il figlio farà la exit, a quel punto al padre verrà restituito il codice di terminazione, verrà riattivato e potrà continuare la sua elaborazione. Il formato della Wait e della exit prende queste forme, quindi la exit non restituisce niente, quindi (qualcosa?) il tipo void, perché non deve restituire niente, una volta che il processo fa la exit termina e basta , non c'0è una istruzione successiva alla exit. E prende invece in int, il codice di terminazione. Per quanto riguarda la Wait invece restituisce l'intero status che è il codice di terminazione del processo. In realtà un processo padre può avere tanti processi figli, può aver fatto diverse fork, per cui quando il processo padre fa Wait, in realtà, riceverà il codice di terminazione del figlio che è terminato, ma deve sapere qual è il figlio che è terminato quindi gli viene restituito anche il bit del figlio(?). Poi c'è un altro caso: che succede se il processo padre è terminato prima del figlio, per cui non può fare la wait, non può ricevere il codice di terminazione, il figlio fa la exit e a questo punto resta in stato di zombie per sempre continuando ad occupare risorse all'interno del nucleo. Questo ovviamente non è ammissibile, per questo motivo quando il processo padre termina, tutti i figli di questo processo vengono adottati dal processo Init, che è un processo di sistema sempre presente. Per cui da questo momento in poi, se questi processi sono orfani, fanno la exit, il loro codice di terminazione viene prelevato dall'Init, non se ne fa niente, ma questo permette la terminazione di tutti questi processi in forma definitiva. Vista la fork, la Wait, la exit abbiamo tutti gli elementi che ci servono per poter implementare una Shell che prende una forma di questo tipo. La shell che noi normalmente utilizziamo per dare i nostri comandi, per compilare, mandare in esecuzione. La shell in realtà esegue un ciclo infinito all'interno del quale legge una riga da tastiera, la interpreta, esegue il comando specificato e poi aspetta la sua terminazione. Cosa fa la Wait materialmente? Fin tanto che legge e fa il parsing di una riga di comando, dalla quale estrae il nome del programma che voleva mandare in esecuzione quindi il nome del comando e i suoi argomenti. Una volta estratto il comando da mandare in esecuzione(gli argomenti), la shell fa una fork, da questo momento in poi abbiamo due rami: il processo shell che si trova nel ramo else perché il PID restituito dalla fork è maggiore di 0 oppure il ramo figlio, quello che eseguirà il figlio della shell al quale la fork restituirà il valore 0. Il figlio dovrà eseguire il programma specificato passandogli gli argomenti. Il processo padre che invece è la shell si sospenderà aspettando la terminazione del figlio. Una volta che il figlio è terminato, la Wait riattiva la shell che potrà ritornare e poi rieseguire nuovamente il prossimo ciclo. Vediamo ora gli aspetti di concorrenza del sistema.

Il sistema operativo ha bisogno di fare tante cose contemporaneamente perché dovrà gestir le interruzioni, dovrà gestire le interazioni con diversi utenti, per poter gestire le interazioni con diversi utenti dovrà eseguire diversi comandi, anche su dispositivi, poi dovrà fare altri lavori, ad esempio il sistema operativo controlla se ci sono aggiornamenti, in più ci sono tutti quei processi che lavorano in background, quindi in maniera parallela per svolgere gli altri compiti e così via.

In generale all'interno del sistema operativo ci sono tante cose da fare, pensare di fare tutto quanto con un unico codice sequenziale è troppo complesso, per questo motivo si usano i processi, quindi ad ogni processo gli si associa un ruolo, un compito e porta avanti quello. Alcuni processi sono degli utenti, altri sono del sistema operativo. Questa necessità di svolgere tanti compiti contemporaneamente, non è un problema soltanto del sistema operativo, ma lo ritroviamo in tanti altri contesti, per esempio i server che devono gestire tante connessioni internet contemporaneamente verso i client, i programmi paralleli che gestiscono tante cose contemporaneamente per aumentare le prestazioni, per poter sfruttare una disponibilità maggiore di processori. Gli stessi programmi che hanno interfacce utente debbono gestire più cose contemporaneamente perché tramite l'interfaccia utente, l'utente può inserire diversi comandi, in punti diversi, con diversi significati e ognuno di questi può attivare delle funzioni del programma come pure dispositivi, programmi che utilizzano la rete o che son legati ai dispositivi, fanno più cose contemporaneamente per un problema di prestazioni (per nascondere la latenza). La necessità di svolgere più cose contemporaneamente ce l'hanno anche i nostri processi utente, non soltanto il sistema operativo, quindi da un lato il sistema operativo deve poter fare più cose mettendo in esecuzione più processi di più utenti. Dall'altro lato noi stessi, nello scrivere il nostro programma, potremmo aver bisogno di fare più cose contemporaneamente. Ad esempio un word processor, concettualmente è un programma molto semplice perché noi scriviamo e il word processor lo visualizza e lo formatta. In realtà, nel fare quest'operazione, deve fare tante cose: da un lato deve gestire l'input della tastiera, quindi ogni volta che premiamo un tasto, il tasto premuto deve essere letto, interpretato e visualizzato a schermo, ed anche inserito nella struttura dati che rappresenta il testo. Può capitare che il tasto premuto comporti un ritorno a capo che fa passare alla pagina successiva, in questo caso il word processor è costretto a riformattare l'intero documento. L'operazione di riformattazione del documento può portar via anche un tempo consistente, non solo i word processor hanno delle funzioni che lavorano in background, per esempio periodicamente vanno a salvare il contenuto del documento per evitare che se c'è un guasto, un'interruzione, una chiusura accidentale, si perda tutto quanto quello che è stato inserito. Immaginiamo dal punto di vista dell'utente se tutte queste azioni fossero fatte sequenzialmente: premiamo un tasto sulla tastiera, scatta l'interrupt, mette in esecuzione il processo, il processo legge questo tasto, lo va a visualizzare sullo schermo e a questo punto lo va a inserire nella struttura dati. Si rende conto che la struttura dati ha bisogno di essere riformattata e fa partire la riformattazione che è un calcolo abbastanza complesso. Durante tutto questo periodo, però, se l'utente digita nuovamente un altro tasto non può succedere niente: il processo è impegnato nel fare la riformattazione del documento, ma siccome sta facendo questo non può leggere dati da tastiera, soltanto quando ha finito leggerà il carattere dalla tastiera e potrà visualizzarlo e inserirlo nel testo ed eventualmente eseguire una riformattazione. Questo tipo di funzionamento causa dei problemi: l'utente medio dopo che ha premuto il secondo tasto e vede che non succede nulla, che non compare nulla, pensa di essersi sbagliato e lo ripreme, continua a non succedere niente perché il processo è impegnato, allora inizia a premere nervosamente sulla tastiera. Tutto ciò causa che quando il processo ha finito l'elaborazione, va a leggere il prossimo carattere che è lo stesso e lo inserisce e a un certo punto inizierà a leggere tutti i caratteri associati agli input sulla tastiera prima che l'utente possa rendersi conto che sta funzionando, ma è molto lento, e inizia a premere DEL per annullare tutto ciò che ha fatto. Tutto questo non è gestibile. Il concetto di programmazione sequenziale non è adatto a risolvere questo problema, siamo abituati a scrivere programmi sequenziali che iniziano dalla prima riga e vengono eseguiti istruzione per istruzione secondo un ordine ben preciso. Se dobbiamo eseguire più cose contemporaneamente questo modello non è più sufficiente, abbiamo bisogno di strutturare il nostro processo come se fosse formato da più processi. I processi sono associati alla protezione, in effetti sono due cose che stanno assieme: un flusso di esecuzione e un dominio di protezione associato. Il risultato è che questi processi, sebbene lavorino con la stessa applicazione e quindi devono avere la stessa visione degli stessi dati, in realtà sono isolati fra loro, non possono condividere niente, potranno mandarsi dei messaggi ma non possono in realtà condividere memoria per effetto dei meccanismi di protezione della memoria, quindi quando un processo ha letto l'input da tastiera ha il problema di comunicarlo al processo che deve fare la riformattazione ma non lo può

fare in maniera efficace. In realtà questo modello è stato abbandonato perché non va sempre bene per quanto riguarda l'astrazione sui processi. Per questo motivo tra i sistemi è stato introdotto il concetto di Thread. Inizialmente i sistemi operativi, quando qualcuno ha iniziato a pensare all'esigenza di avere dei thread, i sistemi operativi non hanno subito introdotto questo concetto, prima è stato introdotto a livello applicativo e poi col tempo è diventato una componente essenziale del nucleo dei sistemi operativi, quindi attualmente tutti i sistemi operativi sono multithread, sebbene c'è voluto del tempo per fare questo passaggio. L'Unix classico anni '70 non prevedeva i thread, questi vennero introdotti a fine anni '80. Cos'è un Thread?

Un thread è una sequenza di esecuzione sequenziale, quindi un'attività sequenziale che rappresenta un compito schedulabile, nel momento nel quale introduciamo i thread, i thread ereditano dai processi il concetto di attività e diventano loro gli elementi schedulabili. In un sistema multithread non si schedulano i processi, non si mandano in esecuzione i processi, ma si mandano in esecuzione i thread. Il concetto di thread è ortogonale a quello di processo, esattamente come il concetto di protezione è ortogonale al concetto di esecuzione. Quindi il processo mantiene la definizione di protezione, quindi al processo associamo un dominio di protezione, ciò significa che associamo a zona di memoria, associamo a serie di risorse, quindi una serie di azioni che il processo nel suo complesso può svolgere ma l'attività del processo viene svolta dai suoi thread. I thread, essendo parte di un processo, possono svolgere la loro attività all'interno del dominio di protezione specificato dal processo. Possiamo avere uno o più thread per processo, quindi per dominio di protezione, possiamo però anche avere un modello convenzionale con un singolo thread per programma, in questo caso abbiamo un processo con un singolo thread e questo rientra nel modello classico di unix. Oppure possiamo avere programmi multithread, quindi un processo con più thread che condividono strutture dati, questi thread sono isolati dagli altri processi. E poi possiamo avere thread del nucleo. Quindi in generale anche il nucleo, avendo l'esigenza di svolgere più compiti contemporaneamente può avere al suo interno più thread. In questo caso i thread del nucleo condividono le strutture dati del nucleo e sono in grado di eseguire compiti con istruzioni privilegiate, per esempio possono disabilitare le interruzioni quando serve, possono interagire con i dispositivi e così via. Quindi i thread, una volta introdotti, sono utili a diversi livelli, sono utili nel sistema operativo, sono utili nei processi utente. Qual è l'idea dei thread? È quella di avere a disposizione un numero infinito di processori sui quali possiamo mandare in esecuzione infinite sequenze di elaborazione indipendenti. Dal punto di vista del programmatore, non c'è in realtà limite al numero di thread, quindi al numero di processori che lui ha a disposizione. Se ha bisogno di svolgere un'altra attività indipendente dalle altre, può creare un altro thread, a questo thread il sistema operativo assegna un processore virtuale (non una macchina virtuale!) e lo mette in esecuzione. Nella realtà le cose sono differenti, perché in realtà non abbiamo tanti processori fisici quanti sono i thread, il numero dei processori fisici è limitato. Nella realtà quello che sta succedendo, come in questo caso, è che abbiamo solo due processori fisici, su questi due possiamo mettere in esecuzione due thread e gli altri thread in realtà sono pronti a passare in esecuzione e il sistema operativo gestisce la transizione tra thread per mandarli in esecuzione. Quindi in qualche modo il sistema operativo ripartisce il tempo dei processori fra tutti e cinque i thread (in questo esempio), scegliendo di volta in volta quello che deve andare in esecuzione. Dal punto di vista del programmatore, in realtà, quello che succede al livello sottostante non è importante, la visione è questa: lui scrive un programma sequenziale, lo associa ad un thread e lui sa che quel thread eseguirà sequenzialmente queste azioni, con un ordine rigoroso specificato dal programma che lui ha asserito. Nella realtà, quando questo thread viene mandato in esecuzione, per effetto del fatto che ci sono tanti thread presenti nel sistema e che questi thread competono per utilizzare i processori, in realtà non è detto che il flusso di esecuzione sia esattamente come lo aveva immaginato il programmatore. Quindi può capitare che effettivamente tutte le istruzioni del thread vengano eseguite sequenzialmente una dopo l'altra come previsto dal programmatore, ma può anche capitare che il thread ad un certo punto esegua la prima istruzione x = x + 1; a questo punto il sistema operativo decida di interrompere temporaneamente l'esecuzione di questo thread per utilizzare il processore fisico per

eseguire un altro thread, quindi in questo intervallo il thread deve rimanere sospeso, non può continuare a lavorare. Soltanto dopo verrà rimesso in esecuzione ed eseguirà le altre istruzioni. Questa interruzione può non capitare affatto o può capitare in un punto arbitrario del thread, non predicibile. Quindi il thread potrebbe essere interrotto qui, o qui o potrebbe essere interrotto più volte in un punto qualsiasi del codice. Il risultato è che il sistema operativo garantisce, in caso di interruzione, che il thread comunque continui a funzionare correttamente conservando il proprio stato, quindi internamente il thread avrà piena consistenza dei dati, indipendentemente dal numero di volte dagli istanti in cui è stato interrotto dal sistema operativo. D'altra parte non possiamo fare nessuna previsione sui tempi di esecuzione del thread, perché se il thread non viene interrotto andrà avanti con la massima velocità che il processore gli può garantire. Se il thread viene interrotto, resta interrotto per un tempo arbitrario, non sappiamo quando tornerà in esecuzione, e quindi non sappiamo quando completerà effettivamente il suo lavoro. In generale abbiamo diversi modi con i quali il sistema operativo può eseguire questi thread, che si dicono concorrenti, perché concorrono nell'utilizzo di una risorsa comune che è il processore, quindi potremmo avere un'esecuzione dove sono in effetti serializzati nel caso in cui abbiamo un unico processore e il modello di scheduling utilizzato è uno scheduling senza pre-rilascio (quello tipico dei primi sistemi), oppure potremmo avere una esecuzione parallela nel caso in cui il sistema operativo disponga di un numero di processori sufficiente per mandarli in esecuzione parallelamente, oppure potremmo avere più in generale un'esecuzione di questi thread molto mista e frammentata, fasi nelle quali vengono eseguiti sequenzialmente, fasi nelle quali vengono eseguiti parzialmente in modo parallelo in funzione di quelle che sono di volta in volta le risorse disponibili.

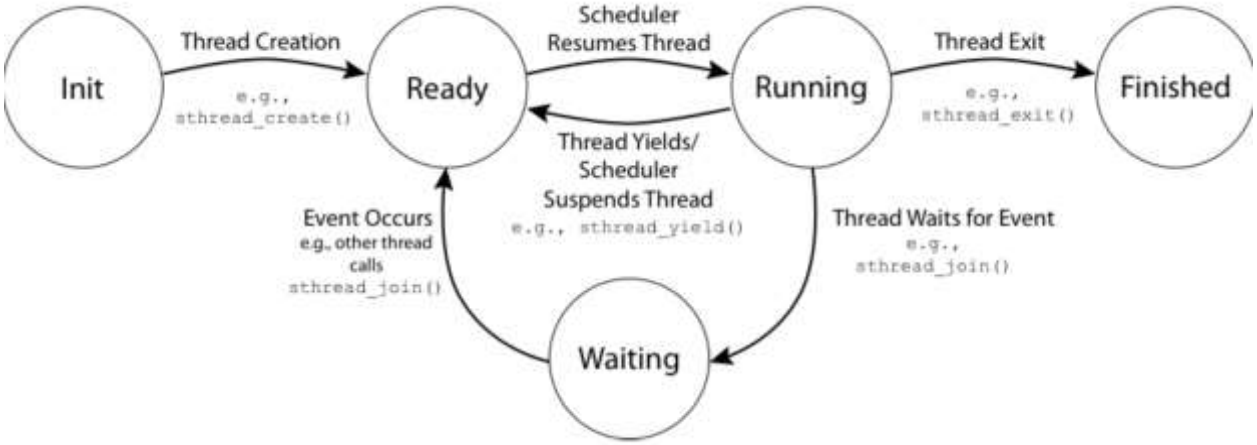
Prendiamo questo esempio:

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```

supponiamo di scrivere un processo che fa 10 fork (e 10 thread), e fatte le fork, ogni thread va a stampare un'informazione associata al suo codice. Se facciamo una cosa di questo genere, vedremo che in effetti i thread vengono eseguiti secondo degli ordini arbitrari. Per esempio inizialmente passa in esecuzione il thread 0 che saluta, il thread 0 che termina, questo vuol dire che l'esecuzione del thread 0 è stata interrotta e nella sua interruzione è stato messo in esecuzione il Thread 1 che ha salutato, poi è tornato in esecuzione

il thread 0 che ha terminato. Dopodiché passano in esecuzione i thread 3 e 4, questo vuol dire che l'esecuzione del thread 1 è stata interrotta per far completare il thread 0 e poi per iniziare l'esecuzione del thread 3 e 4, poi thread 1 termina, poi abbiamo una lunga sequenza di tutti thread (dal 5 al 9) che vengono messi in esecuzione con ordine arbitrari sebbene il thread 2 sia stato creato prima del thread 5, in realtà per qualche motivo il thread 5 si trova ad essere eseguito per prima. In realtà anche in un esperimento così semplice vediamo che non è realmente possibile fare previsioni sull'ordine col quale questi thread verranno messi in esecuzione, creati e ordinati, andranno in esecuzione in modi arbitrari e termineranno in tempi arbitrari. Qual è il numero massimo di thread che possono essere eseguiti contemporaneamente? Questo è pari al numero di processori. Qual è il minimo di thread che possono essere eseguiti contemporaneamente? Zero. Infatti il sistema operativo potrebbe decidere che in un dato istante di tempo nessun thread può essere eseguito. Quindi in un istante di tempo potrebbe darsi che non ci sia nessun thread in esecuzione, in un istante di tempo differente può darsi che il numero di thread in esecuzione sia pari al massimo possibile, oppure sia pari a un qualsiasi valore intermedio compreso tra zero e il massimo. Il sistema è troppo complesso per poter fare una previsione. I tempi di avanzamento relativi, di conseguenza, non sono predicibili. Come si implementano i thread? Si utilizzano delle strutture dati e delle operazioni. La struttura dati che serve per rappresentare i thread è il Thread Control Block che è una struttura che contiene i metadati associati al thread. In particolare conterrà informazioni sul contesto, parametri di scheduling, identificatore del thread, identificatore del processo al quale il thread appartiene e qualsiasi altro metadato che ci venga in mente e poi un certo numero di operazioni sui thread. Il thread non è una entità concreta, è un'entità astratta creata dal sistema operativo, e per creare questa entità astratta il sistema operativo definisce strutture dati e operazioni. Nel caso dei thread, la struttura dati è il PCB e le operazioni sono operazioni del tipo: ci sarà un'operazione di Fork che permette di creare un nuovo thread e di mettere in esecuzione una certa funzione. Quando creiamo il thread dobbiamo dire quale funzione vogliamo eseguire, quindi il codice associato a questo thread. Nel caso dei thread, è differente rispetto al caso dei processi perché i thread hanno molti meno parametri per essere eseguiti mentre i processi devono specificare, all'atto della creazione di un processo, bisogna stabilire una quantità enorme di parametri. Nel caso del thread, molti di questi parametri non sono necessari perché il thread esiste all'interno di un processo, per cui c'è già la memoria, c'è già il (?) di protezione, c'è già un ambiente predefinito. Quindi in realtà per mettere in esecuzione il thread basta soltanto specificare il nome della funzione da eseguire, il codice che vogliamo eseguire e passargli i rispettivi argomenti. Come nel caso dei processi unix, esistono operazioni per fare la exit, quindi per terminare un thread che serve perché, quando un thread termina, bisogna liberare le strutture dati associate al thread. La Join che è l'equivalente della Wait in unix per aspettare la terminazione di un thread e poi per esempio la Yield tramite la quale un thread può lasciare spontaneamente il processore, quindi tramite la Yield un thread che è attualmente in esecuzione, può dire al sistema operativo che interrompe spontaneamente l'esecuzione e dà il processore a qualcun altro. In certo casi la Yield è utile, per quanto possa esser strano. In laboratorio vedremo le operazioni definite dalla libreria Pthread che prende una forma leggermente differente da questa. Una volta che abbiamo definito una struttura dati e abbiamo creato il thread tramite la fork o il meccanismo previsto dal sistema operativo, a questo punto il thread inizia ad esistere ed entra in un ciclo di vita che viene gestito dal sistema operativo. In particolare durante la fase di creazione, quando dobbiamo inizializzare le strutture dati associate al thread, il thread inizia ad esistere in stato di Init, quindi sostanzialmente: facciamo una fork di un thread, il sistema operativo alloca un thread control block (una struttura dati per descrivere un nuovo thread), da questo istante il thread inizia ad esistere ma non posso ancora metterlo in esecuzione, farlo lavorare, perché in realtà devo prima inizializzare la sua struttura dati, il suo PCB. Quindi in questo periodo il thread resta in stato di Init. Quando ho completato la sua inizializzazione, a quel punto il thread cambia stato e viene portato in stato di pronto, quindi quando la creazione è completata, il thread passa in stato di pronto. Che cosa vuol dire pronto? Vuol dire che il thread è pronto ad essere eseguito, ha tutto quello che gli serve per poter essere eseguito: ha un PCB inizializzato, ha un suo stack che è stato inizializzato, sappiamo già qual è la prima istruzione da eseguire, è l'istruzione associata alla prima funzione, quella che devo mandare

in esecuzione. Quindi è tutto pronto per mandarlo in esecuzione, l'unica cosa che gli manca è il processore. All'atto della creazione non possiamo mandare direttamente in esecuzione tutto il thread sul processore perché il processore è utilizzato dal sistema operativo e sul processore il sistema operativo alloca, avrà già allocato altri thread, magari sta svolgendo lui un compito. Quindi l'attribuzione del processore ai thread non può essere fatta in maniera arbitraria, ma ci deve essere qualcuno che decide chi deve utilizzare, quando deve utilizzare, per quanto tempo può utilizzare il processore. È un componente specifico del sistema operativo che si chiama Schedulatore (Scheduler). Lo scheduler è il componente del sistema operativo che decide l'attribuzione del processore ai vari thread e per poter fare questo lavoro, lo scheduler deve sapere quali sono i thread che stanno aspettando di passare in esecuzione e quali sono i thread che sono attualmente in esecuzione, quindi ci sono delle strutture dati che lo schedulatore utilizza. Quando il thread viene creato, viene si portato nello stato di pronto, ma nel fare questa operazione, il descrittore del thread viene inserito all'interno di una struttura dati dello schedulatore che contiene tutti i thread pronti (generalmente questa è una coda). Fin tanto che è in stato di pronto, il thread non muove un'unghia, il suo stato è lì immobile. Quando lo schedulatore, per qualsiasi motivo, decide che è giunto il momento di portarlo in esecuzione perché si è liberato un processore, allora lo toglie dalla coda "pronti" e lo porta materialmente in esecuzione.



La freccia da "pronto" a in "esecuzione" è una decisione dello schedulatore. Come fa a portarlo in esecuzione? Per fare questa operazione, deve fare la Commutazione di Contesto: sposta il contesto del thread dallo stato pronto all'interno del processore e a questo punto il thread è in esecuzione. Quando è in esecuzione, il thread esegue le istruzioni del suo programma. Tra le istruzioni del suo programma potrebbe esserci una exit (thread_exit) e allora in questo caso, il thread esce dallo stato "in esecuzione", viene portato nello stato "terminato" in cui viene deallocato il descrittore del thread e poi a quel punto il thread smette di esistere. Più in generale il thread svolgerà il suo compito e avrà bisogno di parecchio tempo per svolgerlo. Possono capitare in particolare due cose se il thread non termina prima: la prima cosa è che in un sistema operativo Time Sharing, quando lo scheduler ha messo in esecuzione il thread, ha impostato un timer, quindi scatta il timer, ritorna in esecuzione lo scheduler che a questo punto decide di rimuovere il thread in esecuzione perché c'è già stato abbastanza, il suo quanto di tempo è già stato consumato, quindi lo toglie dall'esecuzione e lo riporta in stato di pronto. Nel fare quest'operazione il processore si libera, quindi lo (?) sceglierà un altro thread pronto e lo metterà in esecuzione. Quindi il thread può abbandonare lo stato di esecuzione a causa di una decisione dello scheduler, attivata dall'interruzione del timer e in questo caso si attiva di nuovo il meccanismo di commutazione di contesto per cui si toglie il thread dal processore, il suo contesto si sposta al sicuro da qualche altra parte, si prende un thread pronto, si prende il suo contesto e lo si mette sul processore. Oppure può capitare che il thread abbia bisogno di utilizzare un dispositivo e per far questo utilizzi una chiamata di sistema. Quindi per esempio il thread vuole leggere un blocco dal disco, invoca la chiamata di sistema Read per leggere questa informazione. Però, ricordando quanto detto nella primissima lezione, quando un job invocava un'azione sui dispositivi, in realtà lui deve aspettare il termine

di quest'azione sul dispositivo, e per tutto questo periodo di tempo lui in realtà non può svolgere un compito perché sta aspettando il risultato. La stessa cosa succede nel caso del thread: il thread invoca una chiamata di sistema per fare un'operazione su disco, il disco è mille volte più lento del processore, quindi passa un sacco di tempo prima che il disco possa dargli la risposta. Quando scriviamo un codice del genere per leggere dal disco, cosa scriviamo nel codice: read qualcosa, e poi c'è già l'istruzione successiva pronta per manipolare i dati appena letti, non aspettiamo da nessuna parte, da nessuna parte c'è scritto che quell'operazione sul disco porta via del tempo per cui quel thread che sta eseguendo il nostro codice dovrà aspettare. Nel nostro codice, subito dopo la read, immediatamente iniziamo ad utilizzare i dati letti dal disco, però tra la read e l'operazione successiva nel nostro codice, in realtà passa un sacco di tempo. In questo tempo, se il thread restasse in esecuzione, sarebbe tempo buttato perché dovremmo continuare ad impegnare il processore soltanto nell'attesa di un'operazione dal disco. Abbiamo visto nella prima lezione che questo non è conveniente e in questo caso conviene riassegnare il processore a qualcun altro. C'è un problema: immaginiamo di assegnare il processore a qualche altro thread, ma al nostro thread cosa succede? Questo thread non è pronto perché i dati dal disco non sono ancora arrivati, quindi ci serve uno stato differente per poter rappresentare i thread che stanno aspettando qualcosa, può essere un dato dal disco, può essere un dato dalla tastiera, qualsiasi altra operazione che porta via del tempo e non può essere svolta sul processore. Questo stato è lo stato di Wait. Questa transizione da stato di "esecuzione" a stato di "wait" in realtà è causata dal thread stesso che volontariamente esegue una chiamata di sistema per eseguire tipicamente un'operazione sui dispositivi, o una comunicazione con dei processi o con altri thread. Questa operazione non può essere svolta immediatamente, porta via del tempo, e quindi il processo viene bloccato dal sistema operativo, viene congelato in questo stato. Il processore si libera, di conseguenza lo scheduler può essere riattivato per scegliere un altro thread pronto da mandare in esecuzione. Quando successivamente l'operazione che il nostro thread aveva richiesto si è completata, quindi quando il disco ha completato l'operazione che era stata richiesta, manda un'interruzione, il sistema operativo la intercetta, si rende conto che era un'operazione richiesta da un particolare thread che è in stato di attesa, a questo punto quel thread può essere riattivato. Potenzialmente può tornare in esecuzione. Il problema è che, nuovamente, la scelta dei thread che possono stare in esecuzione è una scelta dello scheduler, non è una scelta arbitraria, quindi il thread che è stato riattivato, che ha ricevuto quei dati, viene riportato in stato di pronto, questa operazione la fa il sistema operativo, ma viene attivata da un'interruzione dei dispositivi, da una ricezione di un segnale, dalla ricezione di un messaggio da parte di un altro processo, a seconda delle cause di sospensione. Una volta che il thread è pronto, a questo punto il sistema operativo chiama di nuovo lo scheduler, mette di nuovo in esecuzione lo scheduler che può fare due cose: o lascia tutto com'è, quindi il thread che era in esecuzione lo lascia in esecuzione, oppure prende atto del fatto che c'è un nuovo thread pronto e allora lo mette in esecuzione e quindi provoca una commutazione di contesto. In ogni caso questa decisione è una decisione dello scheduler, per questo motivo il thread fa questo giro eventualmente. Per implementare i thread correttamente abbiamo bisogno di mettere in piedi questo ciclo di vita: abbiamo bisogno di attribuire ai thread uno stato che può essere Init, pronto, esecuzione, sospeso o terminato, questa codifica dello stato del thread è conservata nel descrittore del thread e ogni volta che il thread è in uno di questi stati, il suo descrittore si troverà in una struttura dati dello schedulatore (la struttura dati dei tre tronchi) o in una struttura dello schedulatore che contiene i thread in esecuzione o in un'altra struttura dati ancora che rappresenta i thread che sono sospesi. Ora però abbiamo un problema perché: sappiamo che i thread per essere eseguiti hanno bisogno non soltanto di queste informazioni generali sullo stato, ma in realtà l'esecuzione comporta attribuire valori al program counter, allo stack pointer, alla program status word, ai registri generali .. Quindi quando un thread viene interrotto, questo insieme di registri che formano il suo contesto, dev'essere conservato da qualche parte. Quando il thread è in esecuzione sul processore, il suo contesto starà nel processore, i registri del processore saranno inizializzati con i suoi valori, ma quando il thread si trova invece in stato di pronto o in stato di attesa, dove sta il suo contesto? Quando il thread è in stato di Init, la posizione del suo descrittore del thread è in qualche struttura del sistema operativo che contiene i thread che sono in fase di creazione, quindi non è

ancora unita agli altri descrittori del thread e la posizione del contenuto dei suoi registri è all'interno del PCB, quindi in particolare il PCB deve contenere abbastanza campi per contenere una coppia dei registri del processore. Quando il thread va in stato di pronto, il suo PCB è nella lista "pronti" e il suo contesto, quindi la coppia dei registri del processore, sono nel TCB. Questo perché il thread non è in esecuzione, non può svolgere nessun compito, quindi lì dobbiamo conservare il contenuto dei suoi registri. Quando però il thread passa in stato di esecuzione, il suo descrittore passa nella lista dei thread in esecuzione e stavolta il suo contesto viene spostato all'interno del processore. Questo non vuol dire che nel TCB abbiamo cancellato quello che c'era, nel TCB quei campi ci sono e restano, il problema è che i valori dei registri del processore contenuti nel TCB, quando siamo in stato di esecuzione, non sono più aggiornati perché i valori corretti sono all'interno del processore. Quando poi il thread, per qualche motivo, passa in stato di Wait, allora il suo descrittore è in una struttura dati che rappresenta i thread in stato di Wait, in realtà di queste ce ne sono tante di strutture dati, una per ogni causa di sospensione e la posizione dei suoi registri va nel TCB. Quando è finito va nella lista dei processi che sono in fase di terminazione e quindi viene cancellato. I suoi registri non hanno più nessun interesse a questo punto. A questo punto che cosa bisogna fare quando si crea un thread (quindi quando viene fatta la fork di un thread)? La prima operazione è quella di allocare un descrittore per i thread, quindi viene allocato il Thread Control Block all'interno del nucleo del sistema operativo, poi dev'essere allocato uno stack per il thread, uno stack all'interno del nucleo per il thread per gestire correttamente le chiamate di sistema e le interruzioni e poi uno stack a livello utente. Dello stack a livello utente se n'è già occupato il supporto a tempo di esecuzione del linguaggio, dello stack all'interno del nucleo se ne deve occupare il sistema operativo. Quindi alla creazione del thread il sistema operativo alloca lo stack nel nucleo, lo stack in stato utente del thread è già allocato dal supporto del linguaggio. In cima allo stack del nucleo ci si mette il record di attivazione per una funzione STUB che incapsula il thread, a questo punto sempre in cima allo stack del nucleo si mette l'indirizzo di partenza della funzione e gli argomenti sullo stack e a questo punto il thread può essere messo nella lista pronti e fatto questo prima o poi passerà in esecuzione. Perché ci server uno STUB quando definiamo il thread? In effetti lo STUB per il thread fa veramente molto poco, invoca la funzione e poi si assicura di chiamare la exit per fare in modo che la exit ci sia sempre, quindi se il programmatore si dimentica di metterla, utilizzare uno STUB come invocatore del thread ci garantisce che la exit sia sempre presente e questo garantisce che il thread termini sempre correttamente (non che non faccia errori, ma che se non ha fatto errori e non ci sono stati altri problemi, termina con una dialogazione(?) delle sue strutture dati). Il thread ha uno stato che sarebbe pronto/esecuzione/attesa/etc, dopodiché però il thread avrà anche una sua memoria, all'interno della sua memoria ci saranno le sue variabili e tutte le informazioni che utilizza per andare avanti con la sua elaborazione. Tutte queste informazioni in realtà sono conservate all'interno della memoria associata al processo. Il thread è un componente del processo, il processo è quello che definisce il dominio di protezione, quindi alloca la memoria e i thread vengono eseguiti all'interno di questo spazio. Se il processo è multithread, più thread utilizzano questo spazio. Se diamo un'occhiata alla struttura della memoria del processo, questa struttura conterrà del codice, delle variabili globali, uno heap e abbiamo visto nelle lezioni passate che c'è uno stack, ma quello vale se il processo ha un unico thread. In realtà nel caso in cui il processo sia multithread, in memoria devono essere presenti più stack, uno per ogni thread, ogni thread ha il proprio. Mentre per quanto riguarda codice, variabili globali e heap queste informazioni che sono in memoria principale, sono associate al processo e quindi sono utilizzabili da tutti i thread di questo processo, quindi sono condivisi, all'interno del codice ci saranno le varie funzioni eseguite dai thread, ogni thread a sua volta, sempre in memoria principale, ha un suo stato proprio specifico, che include il suo stack e il Thread Control Block (TCB). Di stack in realtà ce ne son due, comunque avrà il suo stack in memoria utente e il suo stack all'interno del nucleo, e poi avrà il Thread Control Block (TCB) all'interno del nucleo. Il dominio di protezione lo definisce il processo, non i thread, questo vuol dire che codice, variabile e heap sono utilizzabili liberamente da tutti i thread, gli stack dei thread si trovano in memoria utente e sono utilizzabili in teoria da tutti i thread, però se noi permettessimo un thread di andare ad alterare lo stack di un altro thread faremmo dei danni al nostro stesso codice. Sebbene non ci siano dei meccanismi di

protezione per gli stack dei thread, di fatto ogni thread utilizza il proprio stack e non utilizza lo stack degli altri. Possiamo fare questa ipotesi perché i thread devono collaborare fra loro, non hanno nessun interesse a danneggiarsi. Viceversa, il Thread Control Block (TCB) è protetto all'interno del nucleo, quindi i thread possono usare il proprio e lo stack del nucleo per ogni thread sta all'interno del nucleo. Se ritorniamo all'esempio originale del Word Processor, perché è conveniente utilizzare i thread per un programma di quel genere? Perché i vari thread, che svolgono le varie funzioni del word processor, stavolta ognuno svolge il proprio compito ma possono condividere memoria: condividono le variabili globali e lo heap. Questo vuol dire che il thread che gestisce l'input da tastiera, quando legge un tasto da tastiera, può andare a modificare direttamente la struttura dati che descrive il documento che sarà nello heap e questa modifica è direttamente accessibile/utilizzabile dal thread che invece farà riformattazione del documento, quindi non abbiamo di passare informazioni in maniera complicata tra due processi in deroga ai meccanismi di protezione, stavolta è naturale per i thread condividere memoria e quindi condividere le stesse strutture dati. L'operazione fatta su un thread sulla struttura dati condivisa è immediatamente visibile a tutti gli altri thread, il passaggio di informazioni è molto rapido ed efficiente, d'altra parte ogni thread ha il suo flusso di esecuzione indipendente dagli altri per cui il thread che legge i tasti dalla tastiera, in realtà, può subito essere pronto a ricevere questi dati, perché non deve aspettare la terminazione degli altri thread. A questo punto di modi di implementare i thread se ne presentano diversi: i primi sistemi unix non prevedevano thread al loro interno, quando qualcuno ha iniziato a pensare all'utilità di strutturare i processi in più thread, si è trovato in un mondo nel quale i sistemi operativi offrivano in realtà come unica astrazione quello del processo monothread. L'approccio scelto dai programmatori dell'epoca è stato di dire: implemento i thread tramite delle funzioni di libreria in spazio utente. Un modo per implementare i thread è questo e questo stesso modo è quello che è stato utilizzato per esempio nelle prime versioni di java. Possiamo definire più thread a livello utente in un mondo nel quale il sistema operativo vede solo processi monothread. In realtà questo tipo di implementazione dei thread al livello utente è totalmente irrilevante per il sistema operativo perché è una scelta totalmente autonoma del linguaggio, quindi è il linguaggio che schedula i thread, è il linguaggio che deve definire il TCB, è il linguaggio che deve definire lo stack dei thread, è il linguaggio che li deve schedulare. Quindi se abbiamo dei thread a livello utente, dal punto di vista del sistema operativo, questi thread non esistono, il thread schedula il processo monothread ed è questo processo monothread che poi al suo interno ripartisce il tempo che gli è stato attribuito per ripartire i thread(?) al livello utente. Oppure possiamo avere il modello di unix classico con processi single thread oppure possiamo avere un meccanismo in easty/ISTI(??) con processi monothread e processi multithread e poi anche il thread del nucleo, che è la situazione dei sistemi operativi moderni. In questo caso si pone il problema di come far convivere fra loro thread del nucleo, thread a livello utente, thread implementati nel nucleo ma dei processi. Poi esiste una via di mezzo, tra thread a livello utente e thread a livello del nucleo che sono le Scheduler Activation di windows che sono una via di mezzo per cercare di avere i vantaggi dell'uno e dell'altro modo. Vediamo a questo punto il mondo che contiene processi, thread, implementati a diversi livelli e vediamo che sono informazioni che stanno nel descrittore del processo e nel descrittore del thread. Se siamo in un sistema operativo multithread, con processi multithread, deve continuare ad esistere il concetto di processo perché questo è necessario per definire i domini di protezione e deve esistere il concetto di thread perché è necessario per permettere ai processi di svolgere più attività contemporaneamente. Il risultato è che dobbiamo avere sia il Process Control Block (PCB) associato ai processi, sia il Thread Control Block (TCB) uno per ogni thread. I PCB stanno nella tabella dei processi, i TCB stanno nella tabella dei thread. I thread implementati a livello utente, quindi da funzioni di libreria del linguaggio, hanno una tabella dei thread al loro interno implementata in stato utente nella memoria utente del processo. C'è una tabella dei thread in spazio utente. Dopodiché se i thread sono implementati anche al livello del nucleo, per i thread che funzionano con implementazione a livello del nucleo, questi hanno un altro TCB memorizzato all'interno di tabella dei thread che è conservata dentro il nucleo. Cosa c'è nel PCB e cosa c'è nel TCB? Nel PCB ci sono tutte le informazioni legate al dominio di protezione, quindi a parte il nome del processo, nel PCB dobbiamo conservare informazioni legate alla memoria in uso dal processo alle

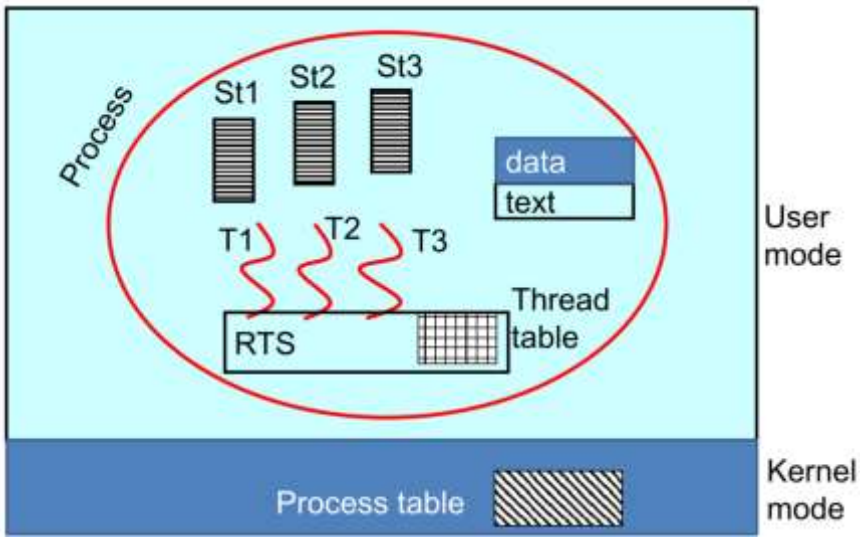
eventuali risorse che il processo ha allocato o meglio, che i thread di questo processo hanno allocato in passato e poi contiene un riferimento a tutti i thread associati al processo. D'altra parte, nel PCB, nel momento nel quale introduciamo i thread, sono scomparse informazioni legate al contesto (la copia dei registri del processore), sono scomparse informazioni legate allo scheduling, etc, perché il concetto di esecuzione è passato ai thread, per cui i thread dovranno mantenere lo stato, il contesto del thread, i parametri di scheduling, il riferimento agli stack e via discorrendo. Tutte queste informazioni, nel caso classico di unix sono contenute in un unico descrittore, sono tutte nel PCB, nei sistemi moderni sono separate. I descrittori dei thread contengono l'informazione alla destra, i descrittori dei processi contengono l'informazione alla sinistra.

- PCB:
  - Process name (PID)
  - Assigned memory
  - Other resources
    - Devices, open files, …
  - Handlers to the process' threads
  - …
- TCB:
  - Thread ID
  - State
  - Context of the thread
  - Scheduling parameters
  - Reference to the stack
  - …

I thread possono essere implementati a due livelli, li possiamo implementare direttamente nel linguaggio e qui per il sistema operativo sono irrilevanti, oppure li possiamo implementare all'interno del nucleo. Che cosa cambia se implementiamo nelle due maniere: I thread a livello utente possono essere implementati da chiunque senza problemi (basti prendere un qualsiasi sistema operativo e su quello creiamo un programma che contiene uno scheduler, le funzioni per creare e inizializzare i thread e le strutture dati per rappresentare i thread), quindi sono interamente implementate generalmente all'interno di una libreria e il sistema operativo non ha visione di questi thread, di conseguenza il sistema operativo non li può schedulare. Tutti questi thread, a livello utente, in realtà vengono eseguiti all'interno di un unico thread al livello del nucleo, che ripartisce il suo tempo fra tutti questi thread a livello utente. Il risultato è che lo scheduling del thread al livello utente deve essere svolto dalla libreria, non è svolto dal sistema operativo. Ora c'è un problema, perché se vogliamo attribuire dei quanti di tempo all'esecuzione dei thread e vogliamo ripartire il tempo di esecuzione in maniera uguale fra tutti questi thread a livello utente, non lo possiamo fare in realtà perché il meccanismo per implementare il time sharing con i quanti di tempo si basa sulle interruzioni del processore, ma le interruzioni del processore vengono intercettate dal sistema operativo sulle interruzioni del timer, ma le interruzioni del timer vengono intercettate dal sistema operativo. Quindi lo schedulatore dei thread a livello utente non può ricevere interruzioni dal timer, di conseguenza non può implementare un meccanismo di time sharing. Il risultato è che i thread a livello utente normalmente utilizzano un meccanismo di scheduling cooperativo. Che cosa vuol dire cooperativo? Vuol dire loro cooperano per cedere il processore l'un l'altro utilizzando le Yield per rilasciare il processore. Se facciamo un salto indietro nel tempo, andiamo a prendere per esempio le prime versioni di windows (95, 98) o apple(anni '90), all'epoca erano costruiti su sistemi che non prevedevano meccanismi concorrenti già al loro interno, non prevedevano una definizione di processi posti al loro interno. Quando successivamente hanno introdotto le interfacce (con windows 95) in realtà queste erano interfacce appiccicate sopra un sistema operativo classico Single Task come era MS-DOS e su questo hanno cercato di definire un concetto di processo/thread, però il problema era che non c'erano reali meccanismi sotto per poterlo supportare. Il

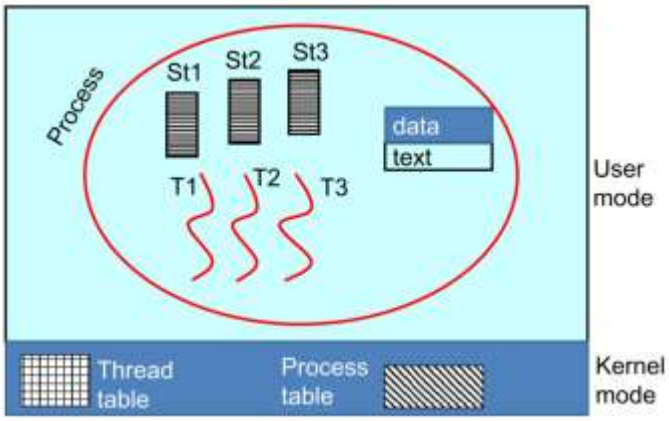
risultato è che se eravamo programmatori dell'epoca, e scrivevamo un programma, all'interno di questo programma dovevamo mettere ogni tanto delle istruzioni Yield per cedere il controllo agli altri processi. Per la sua tesi di laurea il professore doveva scrivere/utilizzare un simulatore. Questo simulatore quando veniva lanciato in esecuzione, portava via parecchio tempo, doveva stare lì ore, giorni per fare quel che doveva fare. Il problema era che se il processore era impegnato a fare tutti questi calcoli, non poteva rispondere all'interfaccia grafica per cui una volta che lanciava una simulazione, il computer si bloccava, non si poteva più interagire fin tanto che non aveva terminato la simulazione. La soluzione era quella di mettere delle Yield nel codice, perché mettendo delle Yield nel codice, soprattutto all'interno dei cicli di calcolo più impegnativi, il suo codice cedeva il controllo al sistema operativo che poteva mettere in esecuzione un altro processo, per esempio il processo legato all'interfaccia grafica, per cui in questa maniera c'era la possibilità di utilizzare anche gli altri programmi. Il sistema non si bloccava più ma poteva essere utilizzabile anche se non era molto comodo. Questa stessa situazione si ripropone se utilizziamo i thread a livello utente. Se usiamo i thread a livello utente, quando lo scheduler di livello utente mette in esecuzione un thread, questo thread usa tutto il tempo del processore che gli è stato assegnato, il sistema operativo liberamente può mandare in esecuzione gli altri processi, ma ogni volta che rimetterà in esecuzione questo processo, ritornerà in esecuzione sempre il solito thread, quindi se anche io ho definito a livello utente più thread, non ho modo di mandarli in esecuzione, a meno che quel thread che è in esecuzione faccia una Yield, invochi lo scheduler a livello utente che può mettere in esecuzione un altro thread a livello utente del suo stesso Pool. La situazione è di questo tipo: abbiamo il sistema operativo, ipotizziamo di avere un sistema operativo che definisce processi monothread (caso di unix classico).

In spazio utente è definito un processo che è questa ellisse rossa,

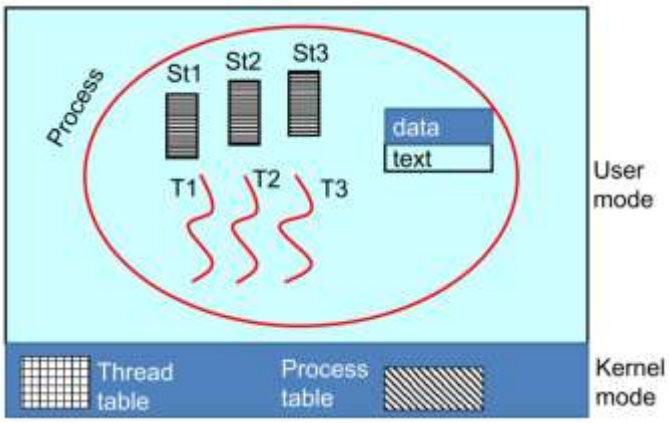


Questo processo definisce un dominio di protezione per cui all'interno di questo spazio nessun altro processo può entrare (zona protetta), quando il sistema operativo mette in esecuzione questo processo, il processo si organizza per gestire il suo tempo dividendolo fra tre task T1, T2, T3. Ma questi tre task sono definiti in realtà dal supporto a tempo di esecuzione che definisce la tabella dei thread, questi tre task condividono codice e dati, ognuno di questi tre task ha il suo stack a livello utente, se il supporto a tempo di esecuzione ha deciso di mettere in esecuzione il Task 1, a questo punto sta eseguendo le istruzioni del Task 1. Se il processo viene deschedulato e viene messo in esecuzione un altro processo, va bene, in realtà è tutto il processo che resta sospeso (resta in stato di pronto). Quando questo processo ritorna in esecuzione però, riparte da dove si era interrotto, quindi ritorna in esecuzione il Task 1. Non c'è nessun modo per il

supporto a tempo di esecuzione di riacquisire il controllo del tempo di questo processore per fare una commutazione di contesto fra T1 e T2. Per questo motivo è necessario che T1, T2, T3 facciano delle Yield, facendo le Yield invocano il supporto a tempo di esecuzione che mette in esecuzione lo schedulatore e attua una commutazione di contesto a livello utente fra T1, T2, T3. Quindi il tipo di scheduling è di tipo cooperativo perché T1, T2, T3 cooperano per ripartirsi il tempo assegnato a questo processore. Che succede se T2 invoca una chiamata di sistema bloccante (se T2 vuole utilizzare il disco ad esempio)? Se il thread T1 invoca una chiamata di sistema, la chiamata di sistema mette in esecuzione il nucleo del sistema operativo, si rende conto che la chiamata è bloccante e quindi deve mettere il processo in stato di attesa, perché il grosso punto di vista del sistema operativo è soltanto la tabella dei processi e quindi lo stato (esecuzione, pronto, etc) è legato all'intero processo. Quindi sono tutti i thread che vengono bloccati. Questo è un limite dei thread a livello utente. Con i thread a livello utente, la creazione, la terminazione, la commutazione di contesto, sono operazioni efficienti perché la creazione dei thread in realtà viene fatta dal supporto a tempo di esecuzione, da una funzione di libreria, non devo invocare una chiamata di sistema per creare un thread. La creerò all'interno della libreria. Le invocazioni della libreria sono invocazioni di funzione che sono molto più efficienti delle chiamate di sistema. La commutazione di contesto non passa per un'interruzione del timer che mette in esecuzione lo scheduler, che copia il contesto all'interno del nucleo del processore, etc. Ma in realtà il contesto viene salvato direttamente all'interno del TCB (se sono con i thread a livello utente) quindi la commutazione di contesto fra un thread e l'altro che è attivata dalla Yield è molto efficiente. L'altro grosso vantaggio è che possono essere implementati su sistemi operativi che non supportano il multithreading (esempio: le prime versioni di unix o le prime versioni di java). D'altra parte ci sono dei grossi limiti dei thread a livello utente perché le chiamate di sistema bloccanti bloccano tutti i thread a livello utente di un processo, oltretutto i thread a livello utente non mi danno un vantaggio concreto nelle architetture multiprocessore. Il processore attribuisce in questo modello processi single thread nel nucleo, ogni processo è multithread a livello utente. Nel nucleo ad ogni processo è associato uno stato (pronto, esecuzione, bloccato). Lo schedulatore prende ogni singolo processo nella sua interezza e lo mette o in stato di esecuzione o in pronto o in bloccato. Quando lo schedulatore prende un processo e lo mette in stato di esecuzione, non può sapere che questo processo è multithread, quindi gli assegna un processore e lo mette in esecuzione su un processore. Quel processo a livello utente utilizza il suo tempo per ripartirlo fra più thread, ma questa è una decisione sua, siccome gli è stato assegnato un unico processore fisico, dovrà ripartire il tempo in quell'unico processore fisico. Per nessun motivo il sistema operativo ci segnerà due processori se ho un unico processo e un unico thread. I thread a livello utente presentano diverse limitazioni, molto più forti degli svantaggi, in realtà è stato fatto tanto sforzo per cercare di trovare delle soluzioni intermedie e per esempio le scheduling activation sono uno di questi esempi. Come si fa ad ovviare ai limiti dei thread a livello utente? Ci sono due risposte: una è di non usare i thread a livello utente ma di usare i thread implementati dal nucleo, però se i vantaggi a livello thread sono comunque buoni, positivi (lo scheduling costa poco, la creazione costa poco, tutte le operazioni sui thread costano poco) e questo è un aspetto che desidero, allora mi serve una soluzione intermedia, ma questa soluzione intermedia non può essere messa in piedi soltanto con la libreria del linguaggio, bisogna che abbia un supporto da parte del nucleo, quindi ho bisogno di una soluzione ibrida, queste sono appunto le scheduling activation. Prima di vedere questo schema, vediamo invece come sono i thread implementati dal nucleo.



Con i thread implementati dal nucleo, il modello che abbiamo è quello di processi multithread, quindi il nucleo del sistema operativo fornisce un supporto per creare, manipolare, gestire i thread, di conseguenza per poterlo fare, la tabella dei thread del nucleo sta dentro il nucleo. Tutte le operazioni sui thread (creazione, terminazione, scheduling, join, Yield, etc) sono tutte operazioni che richiedono l'invocazione di una chiamata di sistema. Non possiamo creare un thread senza chiamata di sistema perché il suo descrittore è nel nucleo, stiamo parlando di thread a livello del nucleo. Quindi la creazione di un thread comporta la creazione e inizializzazione del descrittore dei thread che sta nel nucleo, per poterlo fare bisogna invocare una chiamata di sistema. Il risultato è che tutte queste operazioni sono più costose. A questo punto però, siccome il nucleo vede i thread e schedula i thread (non schedula più i processi), se un processo ha al suo interno due, tre, quattro thread, questi thread possono andare in esecuzione contemporaneamente su processori differenti perché tutti i thread di questo processo vanno a finire nella lista dei thread pronti e lo scheduler attinge a questa lista per mandarli in esecuzione e quindi può scegliere quello che gli pare, anche mandare in esecuzione tutti i thread dello stesso processo contemporaneamente. In effetti scrivere codice a multithread è un vantaggio coi sistemi operativi attuali perché un processo che utilizza due thread ha più possibilità di andare in esecuzione rispetto ad un processo single thread (per una mera questione di probabilità, son due contro uno) per cui in realtà viene eseguito più velocemente. Com'è l'implementazione stavolta al livello del nucleo? Come al solito il processo che definisce un dominio di protezione, quindi un'area protetta inaccessibile agli altri processi, all'interno di quest'area abbiamo la memoria dei dati, il codice e abbiamo gli stack a livello utente dei thread, e abbiamo un certo numero di thread, in questo caso T1, T2, T3. All'interno del nucleo abbiamo la tabella dei thread e la tabella dei processi.



Quindi T1, T2, T3 in realtà sono definiti qui dentro, il processo è definito qua dentro, sempre nel nucleo ci saranno tre stack al livello del nucleo, uno per ogni thread, più tutti gli altri stack del nucleo per gli altri thread e processi. Se il thread T1 invoca una chiamata di sistema per una operazione su disco, il sistema operativo sa che questa chiamata di sistema è stata invocata da T1 perché T1 che è in esecuzione, non tutto

il processo, quindi se T1 invoca una chiamata di sistema, il sistema operativo blocca solo T1 ma può continuare a schedulare T2, T3, perché T2, T3 hanno il loro descrittore nella lista dei thread pronti e quindi possono essere utilizzati liberamente. Nel caso di thread a livello utente, le operazioni sui thread sono effettuate tramite chiamate di sistema (creazione , Yield, etc), questo significa che c'è un Overhead maggiore nella gestione dei thread perché per fare queste operazioni bisogna invocare una chiamata di sistema che comporta tutto il meccanismo di gestione delle interruzioni, salvataggio dei registri, operazioni, poi ripristino di registri IRET, che è un'operazione molto più costosa rispetto a invocare la funzione Yield del supporto a tempo di esecuzione, che è una semplice invocazione di funzione. I thread sono schedulati direttamente dal sistema operativo, quindi il risultato è che i thread possono invocare i System Call bloccanti perché questi hanno effetto soltanto su di loro, non hanno effetto sugli altri. Quindi se vogliamo implementare il codice del nostro Word processor ci conviene di gran lunga utilizzare i thread a livello del nucleo. A questo punto per quanto riguarda i thread nei processi, abbiamo visto che i thread ci servono, sono utili a livello utente, non implementazioni tra livello utente, in generale l'utilizzo dei thread è utile all'interno dei processi, quindi per gli utenti l'astrazione dei thread è fondamentale perché questo permette di ripartire più attività e svolgerle concorrentemente, quindi da concorrenza ai processi utente. Ci sono diverse opzioni, l'opzione A è quella utilizzata nelle prime versioni di java, per cui i thread erano implementati a livello utente da librerie a livello utente con tutti gli svantaggi che abbiamo visto (svantaggi e vantaggi) oppure l'opzione B che è quella utilizzata dai sistemi moderni, è quella di implementare i thread al livello del nucleo, con tutti i vantaggi/svantaggi che abbiamo visto. In effetti nei sistemi moderni queste due modalità convivono: possiamo avere processi multithread che utilizzano thread a livello del nucleo e ogni thread a livello del nucleo può ulteriormente ripartire il suo tempo su più thread a livello utente, nel caso di windows questi thread a livello utente si chiamano "fibre". I thread a livello utente presentano dei limiti però hanno dei vantaggi per quanto riguarda l'overhead. I thread a livello nucleo superano questi limiti ma hanno costi maggiori per l'elaborazione. Non si può ovviare ai thread livello utente operando soltanto a livello utente, quindi il problema non è risolvibile. D'altra parte è utile avere una soluzione ibrida. Quindi una soluzione che permette di avere thread a livello utente che hanno scheduling con prerilascio e che seguono system call non bloccanti, però che siano implementati in maniera più simile ai thread livello utente. Queste cose non è possibile farle senza l'aiuto del nucleo, quindi bisogna avere un supporto ulteriore del nucleo per poter far questo. Questo è quello che si è fatto ad esempio in windows con l'astrazione delle scheduler activations. Con le scheduler activations si riesce ad avere uno scheduling con prerilascio e si riesce ad avere system call non bloccanti. Per quanto riguarda lo scheduling con prerilascio, come si fa? Lo scheduling con prerilascio è quello che funziona sulla base del timer, quindi quello che permette allo scheduler di riacquisire il controllo quando è terminato il quanto di tempo quando ci sono esigenze di altra natura. A dispetto del fatto che il processo che sta in esecuzione possa continuare la sua esecuzione, abbiamo la possibilità di eseguire lo scheduler per interromperlo (questo vuol dire scheduling con prerilascio). Come fare lo scheduling con prerilascio con thread a livello utente quindi? Se ne parla nella prossima lezione.

Threads in a Process

- Threads are useful at user-level
  - Parallelism, hide I/O latency, interactivity
- Option A (early Java): user-level library, within a single-threaded process
  - Library does thread context switch
  - Kernel time slices between processes, e.g., on system call I/O
- Option B (Linux, MacOS, Windows): use kernel threads
  - System calls for thread fork, join, exit (and lock, unlock,…)
  - Kernel does context switching
  - Simple, but a lot of transitions between user and kernel mode
- Option C (Windows): scheduler activations
  - Kernel allocates processors to user-level library
  - Thread library implements context switch
  - System call I/O that blocks triggers upcall
- Option D: Asynchronous I/O

Nell'ultima lezione abbiamo visto l'utilità dei thread, abbiamo visto che aiutano i processi a strutturare la propria attività in più sotto attività concorrenti. La cosa non è altrettanto comoda farla strutturando un'applicazione in più processi, perché i processi possono sì, comunicare tra loro, ma la loro comunicazione non è efficiente come quanto uno potrebbe desiderare. Per questo motivo hanno inventato i thread, che sono delle sotto attività, svolgono delle attività concorrenti all'interno del singolo processo, e hanno il grosso vantaggio che condividono memoria all'interno del processo, quindi lo scambio tra informazione tra i thread è estremamente efficiente. Ci sono diversi modi di implementare i thread, abbiamo visto quelli a livello utente (sotto forma di funzione di libreria),e questa è l'assunzione che faceva java nelle prime versioni di UNIX, questo tipo di implementazione è efficiente perché semplifica tutte le operazioni di creazione e gestione dei thread, d'altra parte, ha degli svantaggi significativi in particolare non permette di sfruttare bene l'architettura multiprocessor perché i thread a livello utente vengono mandati tutti sullo stesso processore e prevede in caso un thread, a livello utente, faccia una chiamata di sistema, il blocco di tutti thread del suo processo una cosa che non è auspicabile. L'altra possibilità è quella utilizzata dai sistemi moderni, ed è quella di offrire thread a livello del nucleo. Quindi thread stavolta sono implementati dal nucleo del sistema operativo che esso stesso ormai è multithread, questo significa che tutte le operazioni sui thread devono avvenire tramite chiamate di sistema (più costose), però abbiamo diversi vantaggi, in particolare i thread possono essere schedulati direttamente dal nucleo con politiche di scheduling con pre-rilascio ad esempio Time-sharing, e poi le chiamate di sistema non sono bloccanti. Più recentemente ci si è posti il problema di trovare una soluzione che permettesse di ottenere entrambi i vantaggi, sia quelli dei thread a livello utente che quelli dei thread a livello del nucleo. Una possibile soluzione, in Windows si

chiama **Scheduler activations** che vi citavo la volta scorsa. Come vengono realizzati i thread in questo caso? Ovviamente se vogliamo uno scheduling con pre-rilascio e vogliamo evitare tutti gli altri problemi, non lo possiamo fare soltanto scrivendo del codice a livello di libreria, ma abbiamo bisogno di un sistema operativo che ci aiuti. Quindi se il sistema operativo è stato modificato per supportare certe funzionalità dei thread a livello utente le cose migliorano. In particolare, per realizzarle si sfrutta, in maniera sostanziale, un supporto del nucleo che avviene in termini di upcall. Considerate un problema come quello dello scheduling per i thread a livello utente. Allora il thread a livello utente non vengono schedulati dal nucleo, vengono schedulati dal supporto a tempo di esecuzione che decide quando l'intero processo viene messo in esecuzione ed è lui che decide quando ripartire il tempo assegnato a questo processo. Che cosa succede quindi nel livello utente? Lo scheduler assegna il tempo di questo processore ad un thread, se questo thread non cede il controllo, continua ad utilizzare il processore e lo scheduler non può tornare in esecuzione, quindi non abbiamo la possibilità di mandare in esecuzione gli altri scheduler a livello utente. Come si può forzare il meccanismo? All'interno del sistema operativo il meccanismo funziona perché c'è un timer hardware che periodicamente manda delle interruzioni, in questo modo riattiva il sistema operativo che può decidere, in questo modo, di alternare tra loro i processi. Quello che ci servirebbe per i thread a livello utente, un timer che possa mettere in esecuzione il supporto a tempo di esecuzione del linguaggio, in particolare lo schedulatore che quindi può alternare tra loro i vari processi. Il problema è che il timer, è un componente hardware gestito dal sistema operativo e non può altro che essere così, quindi ci serve in realtà un timer virtuale, un qualche cosa che riattivi periodicamente il supporto a tempo di esecuzione, e questo tipo di supporto può esser dato al sistema operativo in termini di upcall. Quindi il sistema operativo può creare dei timer virtuali che quando scattano inviano delle upcall ai processi che ne hanno fatto richiesta , quindi in questo modo lo scheduler a livello utente si registra presso il sistema operativo in modo da avvisarlo che lui è uno scheduler che gestirà i thread a livello utente del suo processo, imposta un timer virtuale, dopo di che assegna il suo tempo di processore ad uno dei thread dei suoi thread a livello utente, nel frattempo il tempo passa, quando scatta il timer virtuale e il nucleo del sistema operativo se ne rende conto, segnala questo fatto con una upcall al processo, questa upcall mette in esecuzione lo scheduler del RTS (Run time support) che a questo punto può commutare ad un altro thread a livello utente.

Domanda: Può ripetere qual è il problema iniziale?

Allora, il problema iniziale è questo: immaginiamo di essere in un mondo dove i processi sono single thread, il mondo che c'era negli anni '70, quindi un processo definisce un dominio di protezione e una attività sequenziale perché scritto per farlo. Allora è comodo anche per i processi gestire più cose contemporaneamente, abbiamo visto la volta scorsa l'esempio del WordProcessor in cui ci sono tante cosa da fare ed è bene che queste cose vengano fatte concorrentemente. Col modello a processo con single thread, all'interno del processo non è possibile strutturare più attività concorrenti, quindi se vogliamo sviluppare un'applicazione tipo quella del WordProcessor, la soluzione sarebbe quella di definire più processi collegati fra loro, ogni processo gestiste una diversa attività, e quindi in questo modo abbiamo parallelismo, sostanzialmente nelle attività dell'applicazione. Il problema è che i processi definiscono un dominio di protezione che rende difficoltoso la comunicazione tra processi. Non è impossibile far comunicare due processi, è soltanto più costoso. Allora, per cercare di ovviare a questo problema visto che strutturare un processo di più attività concorrenti è molto utile, si è cercato di trovare una soluzione che semplificasse lo scambio di informazioni fra queste attività concorrenti. La soluzione è stata di introdurre un ulteriore astrazione, che è quella dei thread. Per far questo bisogna fare diversi interventi, in particolare quello più grosso è che, il processo a questo punto non è più associato al concetto di esecuzione. Il processo, in quanto tale, resta soltanto un contenitore che definisce un dominio di protezione. Al suo interno il processo contiene più thread ognuno di questi thread svolge una sua attività indipendente dagli altri thread. Perché risolviamo il problema? Perché questi thread nello stesso processo lavorano all'interno dello stesso dominio di processore quindi possono condividere la memoria e lo scambio di informazioni fra

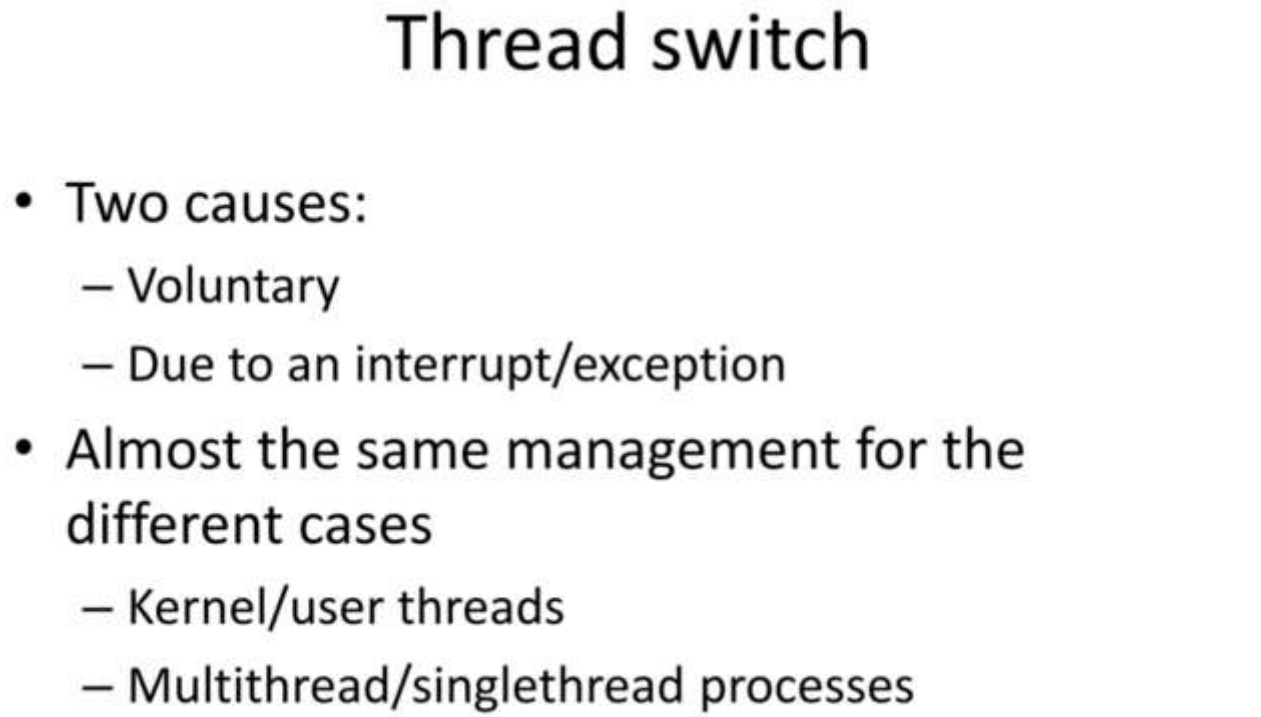
thread è immediato. È come quando al compito, guardate il foglio del vostro vicino di banco, non c'è bisogna di un passaggio di informazioni esplicito, perché vedete la soluzione direttamente scritta lì affianco.

Resta un'altra questione, i thread come li realizziamo? Abbiamo visto diverse alternative, possiamo realizzarli senza dir niente al sistema operativo. Io programmatore, scrivo il mio codice in modo da prevedere più attività concorrenti, però la gestione di queste attività, quindi come passare da l'uno all'altra, me lo implemento io. Questo però è molto pesante per il programmatore. Il passo successivo è stato quello di dire "va bene, mettiamolo pure nel linguaggio. Tanto la gestione dei thread è uguale per tutti i programmatori". Per dare un supporto nel linguaggio, si sono realizzati linguaggi che prevedevano thread a livello utente, implementati tramite un RTS del linguaggio. Facendo questo sono nati, (come al solito l'appetito vien mangiando), introducendo i thread ci siamo resi conto che ci sono delle limitazioni, perché; io ho le mie attività concorrenti a livello utente, immagino che queste attività possano svilupparsi in maniera indipendente, in realtà, se solo una queste invoca una chiamata di sistema, mi si bloccano tutte. Questo non mi piace, vorrei che se un thread sta leggendo da tastiera, gli altri thread possano continuare nel loro lavoro (uno formatta il documento, uno salva una copia, ecc.). Però con i thread implementati a livello utente questo non lo posso fare, quindi perdo tanti vantaggi. Il passo successivo allora è stato quello di dire "benissimo, allora i thread li implementiamo nel nucleo". Se implementiamo i thread nel nucleo del sistema operativo, i thread a questo punto sono totalmente indipendenti nella loro esecuzione, perché è il sistema operativo che può decidere quali di questi thread mettere in esecuzione. E se uno si blocca, perché lui ha chiesto di fare un'operazione di I/O, il nucleo sa che ci sono gli altri thread che possono andare avanti. Questo ha sostanzialmente risolto tutti i problemi. A questo punto ne nasce uno nuovo; uno dice "va bene, con i thread a livello nucleo sono soddisfatto. Riesco a fare tutto quello che voglio, ma c'è ancora un piccolissimo neo. Le operazioni sui thread, di creazione, di commutazione di contesto, mi piacevano di più quando c'erano i thread a livello utente perché costavano meno" "e ma non sei mai soddisfatto" "No, cerchiamo di far meglio.". La soluzione è quella di trovare una via "ibrida" per cui alcune cose le posso fare in spazio utente in maniera efficiente, quello che non posso fare in spazio utente lo faccio con l'aiuto del nucleo. Una delle cose importanti è quella di poter schedulare i thread a livello utente in maniera non cooperativa, quindi, facendo in modo che vengano schedulati indipendentemente dalla loro volontà. Facendo in modo che ai thread io possa sempre dare un quanto di tempo, alla fine di quel quanto di tempo, metto in esecuzione un altro thread, e questo senza che i singoli thread stiano lì a contare per decidere quando loro devono cedere il processore. Come faccio a fare questo interamente in spazio utente? Non lo posso fare perché, per dire "ti metto in esecuzione per un quanto di tempo" dovrei avere un contatore del tempo che mi sveglia quando è finito il tempo. Questo contatore del tempo, esiste ma lo sta usando il sistema operativo e non lo può dare ad un processo. Ma il sistema operativo può creare dei contatori del tempo virtuali, dei timer virtuali, e offrirli ai processi. Quindi un processo può. Lo scheduler del RTS del linguaggio all'interno di un processo si registra presso il sistema operativo, gli chiede di inizializzare un timer virtuale, a questo punto può mettere in esecuzione un thread e dimenticarselo. Quando scade il tempo virtuale, il sistema operativo lo può gestire, perché è gestito con le interruzioni, a questo punto il sistema operativo si rende conto che deve far scattare il timer virtuale per quel processo, e lo fa scattare inviando una upcall. Avvisa il RTS di quel processo che è scattato il timer, il RTS mette in esecuzione lo scheduler che, indipendentemente dalla volontà del thread che era in esecuzione, fa la commutazione.

Se lo scheduler di quel processo a livello utente si è registrato presso il sistema operativo, quando un thread invoca una chiamata di sistema bloccante, il sistema operativo può accorgersi di questo fatto, che sì, è una chiamata di sistema proveniente da un processo, ma in realtà quel processo ha dei thread a livello utente. Quindi anziché bloccare l'intero processo, può inviare una upcall al RTS, in questo modo l'RTS può richiedere di continuare ad usare il processore, perché ha un altro thread da mandare in esecuzione. Attua la commutazione di contesto e il sistema operativo però, deve continuare a schedulare quel processo. Il sistema operativo deve dare una grossa mano per far tutto questo.

Queste sono le Scheduler activations. Esiste in realtà un'altra possibilità per gestire più attività contemporaneamente senza introdurre i thread, che è quello di gestire l'I/O in modo asincrono. Anche questo ve lo cito, perché in alcuni sistemi, soprattutto sistemi ad uso speciale, sistemi enbedded, questo tipo di soluzione è ancora in uso, ci sta che il vostro smartwatch (se ha Android no) funzioni ancora così. Come funziona l'I/O asincrono? Ipotizziamo io voglia leggere ad un certo punto un dato dalla tastiera, allora do il comando alla tastiera e guardo se per caso l'utente ha premuto un tasto. Però questa richiesta non è bloccante per cui, se il sistema operativo si rende conto che non è stato premuto nessun tasto, mi restituisce il controllo. Il problema qual è? Per il sistema operativo è facilissima la cosa, perché a questo punto la chiamata di sistema; entra dentro il nucleo, controlla se c'è un dato da prendere, se c'è lo restituisce, altrimenti dice "non c'è niente". Il problema grosso è per il programmatore. Immaginate di scrivere del codice nel quale la scanf potrebbe dirvi "no, l'utente non ha digitato niente". Come dovreste strutturarlo il codice? Intanto avete un vantaggio, perché se la scanf non ha fatto niente, allora potete fare un'altra cosa, per esempio la formattazione del testo, però d'altra parte, dovete stare attenti perché se l'utente preme un tasto sulla tastiera e voi non ve ne accorgete per tempo, questo è un problema. Quindi dovete prevedere il codice fatto a cicli, all'interno dei quali periodicamente controllate la lettura del dato della tastiera, anche se state facendo un'altra attività completamente differente. La cosa non è impossibile da fare, perché si fa di continuo in certi sistemi e in certe condizioni. È un tipo di programmazione simile a quella che fa chi programma con interfacce grafiche, perché tutte le funzioni associate dall'interfaccia si attivano soltanto quando voi agite su di essa.

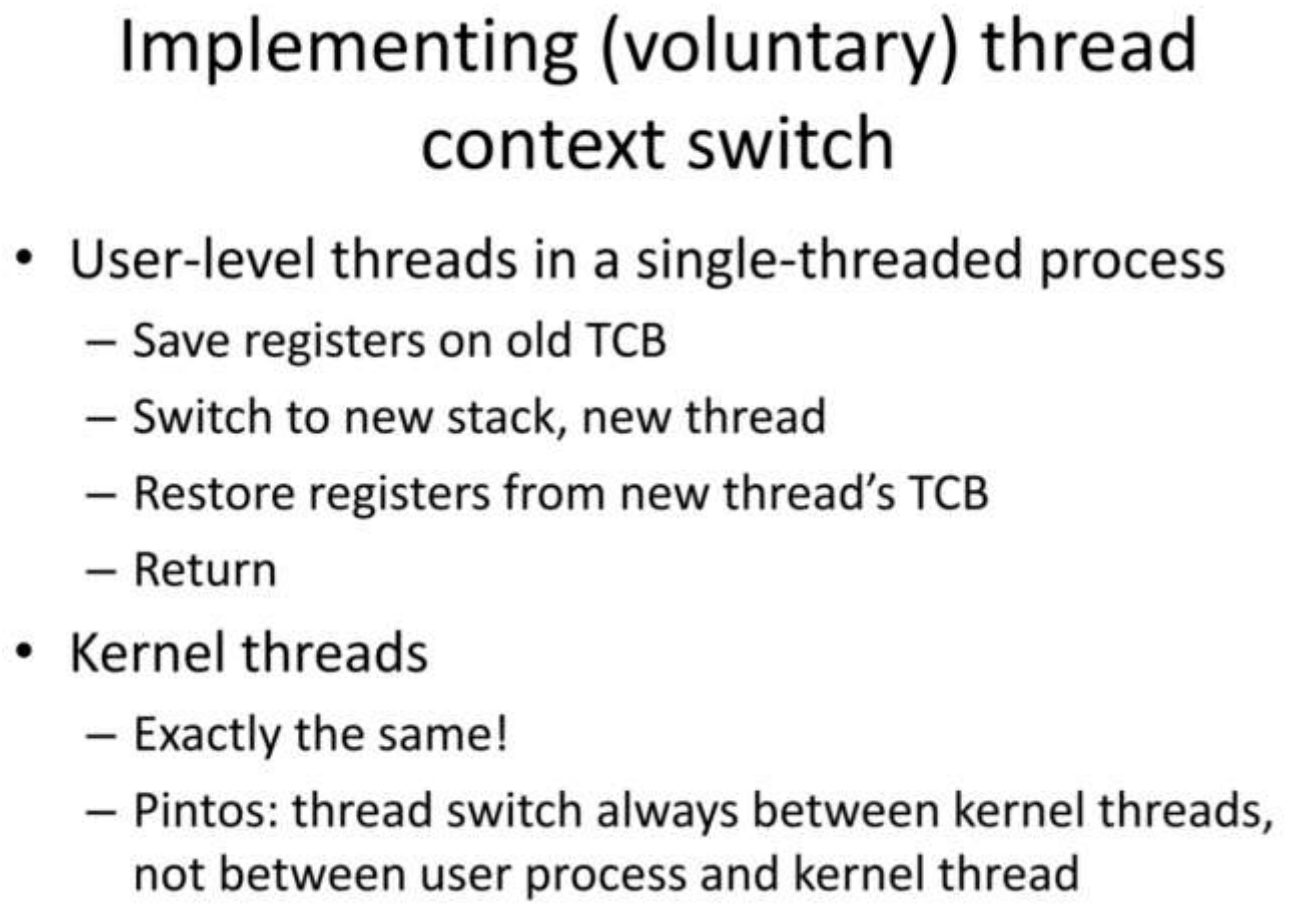
A questo punto vediamo i meccanismi per far funzionare i thread. Noi abbiamo sempre detto fino ad ora, "benissimo, c'è lo schedulatore che è un componente del sistema operativo, che ad un certo punto va messo in esecuzione per togliere un processo dall'esecuzione, per mettercene un altro". Questa operazione di togliere un thread dall'esecuzione e mettercene un altro, è detta **Commutazione di contesto o Thread switch.**



Può avvenire per tanti motivi, in particolare può essere: volontaria, se un thread decide spontaneamente di cedere il controllo. Oppure può essere dovuta a certe cause; interruzioni, eccezioni ecc. La causa più tipica,

è l'interruzione che arriva dal timer. Al nucleo arriva il timer, mette in esecuzione lo scheduler, che attua la commutazione di contesto per alternare i thread in esecuzione secondo il principio del time sharing. La cosa è complicata, perché abbiamo tanti casi; potremmo avere una commutazione di contesto per thread che stanno a livello utente, potremmo avere una commutazione di contesto per thread che stanno al livello del nucleo, la commutazione di contesto può essere causata per diversi motivi. In pratica il meccanismo di commutazione di contesto è pressoché simile in tutti i casi. Ci sono sì, delle situazioni speciali se stiamo utilizzando i thread a livello utente piuttosto thread a livello del nucleo, però alla fine della storia, è sempre quello.

Immaginiamo di essere in un sistema con processi a singolo thread. Quando dobbiamo fare la commutazione di contesto, cosa sta succedendo?



C'è un thread che è in esecuzione, su un processore. Su questo processore, il thread sta utilizzando i registri di questo processore, quindi i valori delle sue variabili intermedie sono memorizzate nei registri, l'istruzione che il thread sta eseguendo è conservata nel PC, lo stato dell'esecuzione è nella Program Status word (PS) e via discorrendo. Non si può semplicemente mettere un altro thread in esecuzione sul processore, perché per il thread che era in esecuzione rischiamo di perdere tutte le informazioni, esattamente come quando abbiamo visto il caso delle interrupt. La prima cosa da fare è quella di salvare tutti i registri del processore, ma dove li salviamo? Nel caso delle interrupt le salvavamo sullo stack e in prima battuta, in realtà, anche in questo caso tutti i registri verranno salvati nello stack. Però in realtà, siccome questo thread che era in esecuzione non sappiamo quando potrà tornare in esecuzione, potrebbe passare tantissimo tempo, potrebbe succedere di tutto, è opportuno salvare questi registri in una posizione più sicura. Questo posto è il Thread Control Block (TCB). Una volta salvati tutti i registri del processore associati al thread che era in esecuzione nel TCB, il processore è pulito; possiamo scegliere un altro thread, prendere dal suo TCB i valori dei registri, ricopiarli nel processore, ovviamente facendo questo impostiamo lo stack corretto per l'altro thread, impostiamo tutto quello che serve (PC, PS) e questo punto possiamo restituire il controllo al thread

nuovo, quello che abbiamo messo finalmente nel processore. Questa operazione, come principio, è la stessa, che stiamo parlando di thread a livello del nucleo o che stiamo parlando di thread a livello utente. Quello che cambia è che alcune di queste operazioni di ritorno che restituisce il controllo al thread, avviene in maniera molto differente, se siamo su thread a livello utente è un return tipo quello del C, se siamo su thread a livello del nucleo, questa è una iret. Se siamo sui thread a livello del nucleo, parte di questo salvataggio viene fatto tramite il meccanismo delle interruzioni e ci si appoggia temporaneamente allo stack del nucleo. Se siamo su thread a livello utente invece si appoggia allo stack del livello utente. La sostanza non cambia.

Two threads call yield

Thread 1's instructions	Thread 2's instructions	Processor's instructions
call thread_yield		call thread_yield
save state to stack		save state to stack
save state to TCB		save state to TCB
choose another thread		choose another thread
load other thread state		load other thread state
	call thread_yield	call thread_yield
	save state to stack	save state to stack
	save state to TCB	save state to TCB
	choose another thread	choose another thread
	load other thread state	load other thread state
return thread_yield		return thread_yield
call thread_yield		call thread_yield
save state to stack		save state to stack
save state to TCB		save state to TCB
choose another thread		choose another thread
load other thread state		load other thread state
	return thread_yield	return thread_yield
	call thread_yield	call thread_yield
	save state to stack	save state to stack
	save state to TCB	save state to TCB
	choose another thread	choose another thread
	load other thread state	load other thread state
return thread_yield		return thread_yield
...	...	...

Per esempio, immaginando di avere due thread che operano a livello utente, primo thread ad un certo punto, manderà una Yield, questo provocherà il salvataggio dello stato nello stack per poter eseguire lo scheduler che salverà lo stato dallo stack al TCB e sceglierà un altro thread, quindi caricherà lo stato di questo thread sul processore e lo metterà in esecuzione. A questo punto, passa in esecuzione l'altro thread che ad un certo punto farà una Yield (ha l'effetto di salvare una parte dello stato del processore in cima allo stack), invoca lo schedulatore che completa il salvataggio dei registri dallo stack e dal processore verso il TCB, a questo punto sceglie un altro thread, ripristina lo stato di quel thread nel processore e si continua.

Thread switch on an interrupt

- Thread switch can occur due to timer or I/O interrupt
  - Tells OS some other thread should run
- Simple version (Pintos)
  - End of interrupt handler calls switch_threads()
  - When resumed, return from handler resumes kernel thread or user process
- Faster version (textbook)
  - Interrupt handler returns to saved state in TCB
  - Could be kernel thread or user process

Che cosa succede quando siamo con thread a livello del nucleo e la commutazione di contesto avviene a causa di una interruzione? Tipicamente l'interruzione del timer. Se arriva l'interruzione del timer, questa informazione dirà al sistema operativo che bisogna cambiare il thread in esecuzione. Quindi l'interruzione del timer viene gestita direttamente dallo scheduler, l'interruzione mette in esecuzione un handler di interruzione che a sua volta attiva lo scheduler. Ci sono diverse versioni, a seconda dei vari casi che si possono presentare o a seconda di quale ottimizzazione possiamo fare. Quella completa è questa.
L'handler salva lo stato nel TCB, a questo punto invoca lo schedulatore che sceglie un nuovo thread e a quel punto viene fatto il ripristino. In alternativa, l'handler si limita a smistare il compito allo schedulatore che poi salva, con un'altra funzione del nucleo, i dati dallo stack al TCB, fa la sua scelta e via discorrendo.

Il nucleo può avere una struttura arbitrariamente articolata al suo interno e sono tante le funzioni che possono essere invocate a catena, oppure facendo fare tante cose all'handler risparmiando invocazioni.

Thread switch

- Save registers (context) of the old thread in the TCB
- Move old thread's TCB to Ready List or to a Waiting List
- Select a new thread from the Ready List
- Restores new thread's registers from TCB to processor
- Put new thread's TCB in the Running List
- return control to the new thread (IRET)

Questo è sempre il solito discorso. Quindi, è arrivata l'interruzione del timer, i registri del processore li salvo nel TCB del processo che era in esecuzione. Il descrittore del thread che era in esecuzione, non sta più nella lista dei thread in esecuzione, ma lo devo spostare nella lista dei thread pronti. Questa è una decisione dello scheduler. Il primo punto può essere un meccanismo svolto parzialmente dall'handler, ma il secondo è completamente a carico dello scheduler. A questo punto lo scheduler sceglie un nuovo thread dalla lista pronti, potrebbe essere lo stesso thread se non ce ne fossero altri, in generale sarà un altro thread e a questo punto si fa il ripristino del thread scelto. Si accede al TCB, si prendono i registri di sistema e si copiano nel processore, si sposta il TCB nella lista dei thread in esecuzione (queste due cose possono essere fatte in un ordine arbitrario) ed infine si restituisce il controllo al nuovo thread con l'iret.

Ora questa operazione ha un certo overhead; una parte di questo overhead (di calcolo aggiuntivo che dobbiamo fare), è esplicito e invece un'altra parte è implicito ed è anche difficile da stimare.

Thread switch - overhead

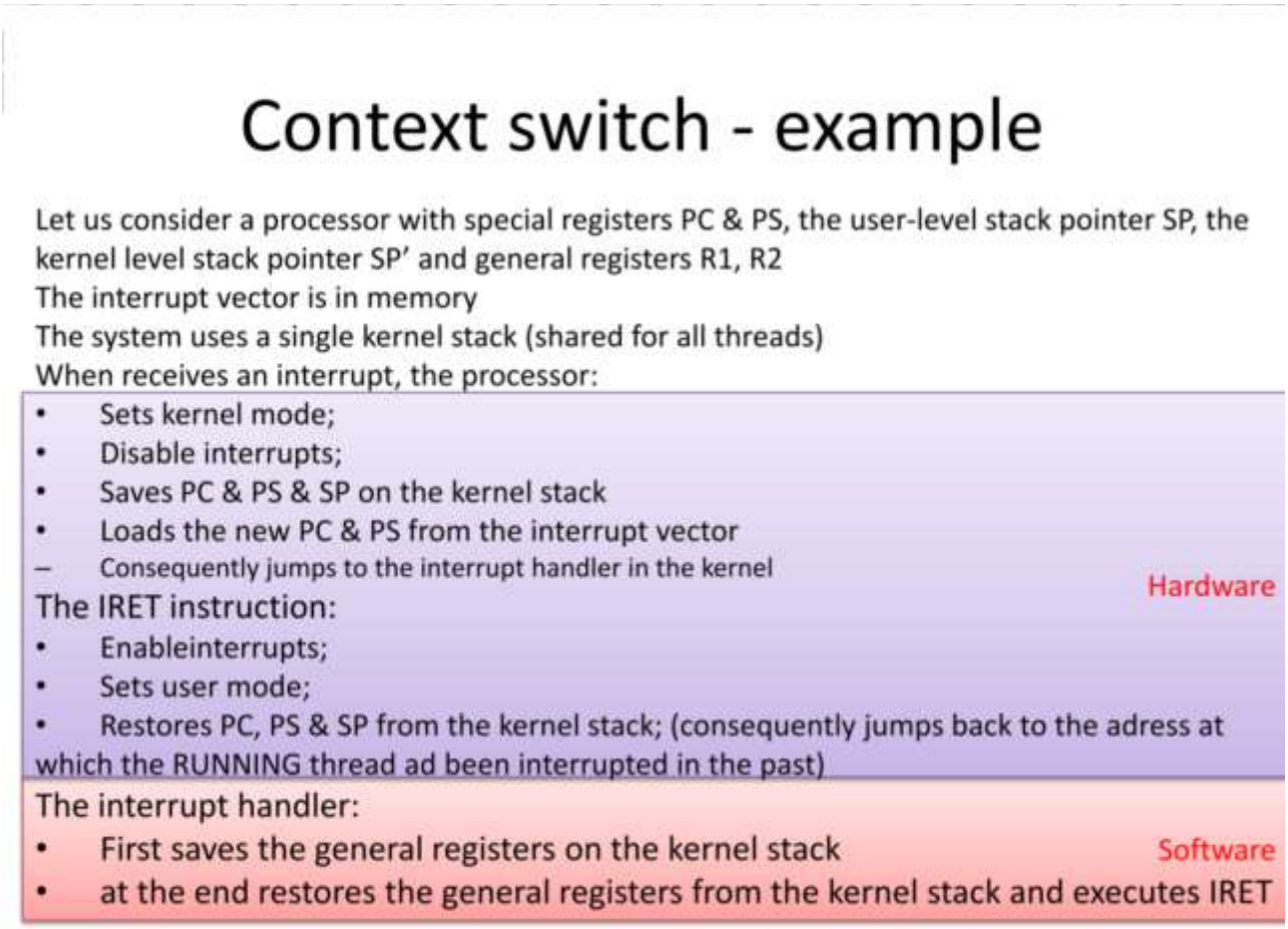
- Due to registers save and restore
- Due to TCB queues management
- Memory cache invalidation
  - link to the computer architecture class
- Induced operations on the memory manager
  - Address exceptions
  - Page faults
  - MMU invalidation
    - Will be discussed later on

Certamente una parte del corso è legata alla copia dei registri, se il processore ha un banco di 100 registri generali, noi dobbiamo fare 100 copie di queste informazioni dal processore alla memoria e questo ha un costo non indifferente. Poi dobbiamo gestire le code dei TCB, dobbiamo andare ad accedere alla lista dei thread pronti, alla lista dei thread in esecuzione e dobbiamo spostare i descrittori da una parte all'altra. Quest'altro punto in rosso, è un punto delicato. Perché? Perché è un costo nascosto. Voi sapete che i processori per accelerare le loro operazioni, usano una CACHE. Questa cache dati permette di risparmiare tempo negli accessi in memoria, il problema è che, se noi facciamo una commutazione di contesto, il thread che mettiamo in esecuzione è un altro, quest'altro thread accederà a delle altre locazioni di memoria che sono le sue, sono diverse rispetto alle locazioni di memoria che erano in uso dal thread precedente. Questo fa sì che la cache del processore diventi, di fatto in larga misura, inutile. Forse non tutta la cache diventa inutile perché dipende cosa farà questo thread, magari utilizzerà alcune chiamate di sistema e ci sono delle cose che sono in comune con il vecchio thread. Siccome questa cache sarà inutile, il nuovo thread, una volta messo in esecuzione, creerà molti fault di cache. Tutte le volte che proverà ad accedere alla memoria, l'MMU si renderà conto che non potrà soddisfare quello richieste andando a cercare nella cache e che bisognerà andare a cercare nella memoria principale, provocando un ritardo. Nella fase iniziale dovrò usare più la memoria e meno la cache.

Un altro overhead, che ora è difficile da spiegare perché ci mancano tanti elementi, può essere prodotto dal gestore della memoria. Il gestore della memoria è un componente del sistema operativo che si preoccupa di segnalare la memoria libera ai processi e nel farlo, deve fare tutta una serie di cose che vedremo nella seconda parte del corso. Il fatto di passare da un thread all'altro ha un impatto sulla gestione della memoria. Come vedete, una commutazione di contesto non è "gratis", porta via del tempo, perché dobbiamo fare più operazioni e perché induce un rallentamento del thread che viene messo in esecuzione, che non si trova a lavorare a regime. Quindi cosa ci dice questo? Ci dice che non dobbiamo esagerare con il time sharing. Da un certo punto di vista, sarebbe utile avere i quanti di tempo più piccoli possibile, perché

in questo modo i thread si alternano più velocemente l'uno con l'altro e danno maggior impressione di avanzare in modo uniforme. Se vice versa i quanti di tempo sono molto lunghi, vediamo i thread che avanzano un po' a scatti e questo non è carino. Quindi sarebbe bene, avere i quanti di tempo piccoli, ma d'latra parte se li riduciamo troppo, questo costo può diventare significativo.

Vediamo a questo punto, un esempio di come funziona la commutazione di contesto:



La commutazione di contesto presenta casi, sia per quello che vi dicevo prima, se stiamo parlando di thread a livello utente o thread a livello del nucleo, il motivo che li ha scatenata e poi dipende molto anche da come è fatto il processore e anche da alcuni aspetti del sistema operativo (quanti stack ci sono, se ne è presente uno nel nucleo, come è fatto il TCB e via discorrendo.). Le variabili sono tante.

Supponiamo di avere un processore, con i registri speciali PC e PS, uno stack a livello utente indirizzato dallo stack pointer SP, uno stack a livello del nucleo indirizzato dal registro SP' (il processore ha due registri per indirizzare lo stack, questo semplifica notevolmente le cose nella commutazione di contesto), e supponiamo che abbia 2 registri generali R1 e R2. Non ha senso metterne tanti, tanto tutti i registri generali hanno lo stesso trattamento. Sappiamo che il vettore delle interruzioni sta in memoria. Vi ricordo che il vettore delle interruzioni è una struttura dati gestita del nucleo che associa, ad ogni interruzione ricevuta dal processore, l'indirizzo di un handler da mettere in esecuzione. Supponiamo anche che il sistema operativo utilizzi anche un unico stack al livello del nucleo per tutti i thread (in alcuni sistemi questa ipotesi non è vera, ogni thread ha il suo stack nel nucleo). Quando il processore, riceve un'interruzione viene messo in esecuzione un meccanismo hardware che abbiamo visto le lezioni scorse; il processore imposta la modalità supervisore, disabilita le interruzioni, salva PC PS SP sullo stack del nucleo, accede al vettore delle interruzioni, estrae PS PC dal thread e li mette nei registri PC PS del processore. Questo è esattamente il comportamento che abbiamo visto qualche lezione fa. Supponiamo che, sempre ad hardware, ci sia il

meccanismo delle iret che fa l'esatto opposto, per cui; riabilita le interruzioni, imposta la modalità utente, prede PC PS SP dalla cima delle stack del nucleo e li mette nel processore e facendo questo restituisce il controllo al thread a livello utente. Mentre invece a livello software abbiamo l'interrupt handler che salva i registri generali nello stack utente e quando la gestione dell'interruzione è stata completata, fa l'operazione inversa; ripristina i registri generali dalla cima dello stack nel processore. Quindi l'interrupt handler è uno STUB per il gestore dell'interruzione, per cui l'interrupt handler si preoccupa soltanto di completare il salvataggio dei registri dal processore allo stack e poi dallo stack al processore. Qui in mezzo invoca la funzione di gestione dell'interruzione. Queste sono le premesse, per questo particolare esercizio che vediamo ora.

Context switch – example 1

Hyp. A): thread T1 invokes a system call. At the end it remains in RUNNING state

1) Initial situation during the execution of SVC instruction (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
1000	
1001	
1002	
1003	
1004	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

kernel SP	0FFF

Tutto questo lo rappresento con un po' di tabelle, in particolare, nel nucleo da qualche parte ci sarà il descrittore del thread T1, ci sarà il descrittore del thread T2, ci sarà lo stack del nucleo poi sempre in memoria, c'è il vettore delle interruzioni, da qualche parte in memoria il nucleo si ricorsa qual è lo stack pointer del nucleo e poi abbiamo il processore. C'è una piccola discrepanza con il lucido precedente. In questo caso il processore ha un unico stack e non due. Cosa c'è da dire di queste strutture che abbiam visto? Allora, il descrittore del thread che cosa contiene? Contiene lo stato del thread, se è pronto, in esecuzione, in attesa, poi contiene una copia dei registri del processore e altre informazioni che ai fini fi questo esercizio non ci servono. Ora che cosa capiamo guardando questi due descrittori dei thread? Capiamo che T2 è pronto, quindi non è in esecuzione e i suoi registri sono quelli conservati nel suo descrittore del thread. Quando verrà messo in esecuzione, questi registri dovranno andare nel processore. Del thread T1 invece capiamo che è il thread in esecuzione e di conseguenza i valori dei registri conservati nel suo TCB non hanno senso perché è in esecuzione sul processore, quindi i valori dei registri sono quelli che stanno nel processore e sono quelli aggiornati. Nel TCB ci sarà una copia vecchia, che non ci interessa. Questo vale per tutti, anche per la PS. Ora in questo momento il processore sta lavorando in modo utente perché c'è scritto là sopra, quindi i registri sono associati al thread 1 e nell'istruzione puntata dal PC c'è l'istruzione SVC che è l'istruzione con la quale il thread T1 sta invocando una chiamata di sistema. SVC è

l'acronimo di SuperVisorCall (la trap della lezione scorsa), l'istruzione a linguaggio macchina che permette al thread di invocare una chiamata di sistema. Se vi ricordate l'SVC setta un interrupt all'interno del processore.

FINE PT1

Context switch - example

1) Initial situation during the execution of SVC instruction (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
1000	
1001	
1002	
1003	
1004	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF

2) After interrupt (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5000
PS	AA45
SP	1002
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF

Siamo nello stato 1), per cui nel momento nel quale il thread invoca l'SVC la situazione è questa. L'SVC setta un interrupt e questo interrupt viene riconosciuto nel ciclo di estrazione-esecuzione successivo dal processore, che mette in atto un meccanismo automatico all'interno dello stesso processore che produce questo stato, quello del punto 2) è quello che risulta un instante dopo aver riconosciuto l'interrupt. Che cosa è successo? Automaticamente il processore ha preso i valori di PC PS SP e li ha copiati in cima allo stack del nucleo, contestualmente ha preso PC PS dal vettore delle interruzioni della riga associata alla interruzione ricevuta e le ha copiate all'interno dei registri PC PS del processore. Sempre contemporaneamente ha preso lo SP dal nucleo e l'ha copiato nel registro SP, ovviamente aggiornato perché l'SP ci ha già inserito i valori. Evidentemente per fare questo passaggio dallo SP, questa è un'informazione che c'è scritta da quale parte nel nucleo, qual è lo SP da utilizzare quando siamo in modalità supervisore. Per fare questo passaggio, per copiare questo valore qui dentro presumibilmente il processore dispone di registri di appoggio per gestire correttamente questo scambio. Fatto questo, che cosa sta succedendo? Sta succedendo che adesso noi ci troviamo, ad esempio, ad eseguire l'handler dell'interruzione a partire dall'indirizzo 5000 in modalità supervisore a interruzioni disabilitate. I valori di PC PS SP del thread che era in esecuzione, sono salvi in cima allo stack del nucleo, però i registri generali sono ancora quelli in uso dal thread precedente, mentre invece noi dobbiamo gestire l'interruzione e per far questo dovremmo far del lavoro sui registri generali del nuovo thread. Quindi il prossimo passo, fatto dall'handler, sempre ad interruzioni disabilitate, è partire dallo stato 2) prendere i registri generali e copiarli

in cima allo stack del nucleo.

Context switch - example

2) After interrupt (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5000
PS	AA45
SP	1002
R1	4500
R2	CD31

3) after temporary storage of registers (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	4500
1003	CD31
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

A questo punto, che cosa sta succedendo? Lo SP ovviamente, si modifica perché tiene conto dell'inserimento di questi valori in cima allo stack. I valori di R1 e R2, da questo momento in poi, non hanno più significato per noi perché, prenderanno i valori che gli attribuirà l'handler e poi la funzione di gestione delle interruzioni, quindi prenderà dei valori che serviranno al nucleo per fare il suo lavoro. Non possiamo sapere sulla base del testo dell'esercizio cosa diventeranno quei registri. Questo completa un primo salvataggio. Il thread T1 in questa fase è ancora formalmente in esecuzione, il suo TCB non contiene ancora una copia dei registri del processore perché i suoi registri, in questo momento, si trovano nello stack. Ora possono capitare diverse cose; può darsi che questa SVC non sia bloccante, magari il thread T1 deve tornare subito in esecuzione. Se questa è la situazione, potrei anche evitare di copiare questi registri dallo stack al TCB e rimettere subito T1 in esecuzione. Se invece la chiamata di sistema è bloccante, quindi T1 si deve bloccare, allora è opportuno, visto che c'è un unico stack nel nucleo, prendere questi valori e copiarli nel TCB.

Vediamo questo primo caso:

Context switch - example

4) at the end of the primitive (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	4500
1003	CD31
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

5) during extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5100
PS	AA45
SP	1002
R1	4500
R2	CD31

Quindi, siamo nel punto precedente in alto, nel punto 5) siamo alla fine della iret. Durante la chiamata di sistema, che cosa è cambiato? Abbiamo eseguito la funzione che gesiste la chiamata di sistema, che ha manipolato R1 e R2 in una maniera che non sappiamo, però ha deciso che il thread che era in esecuzione può restare in esecuzione, quindi ha deciso di non salvare i registri nel TCB, ma invece, li lascia nello stack perché da lì poi, li pyò rimettere direttamente T1 in esecuzione, senza perdere tempo. Per farlo dobbiamo, quindi, attingere dalla cima dello stack, ma prima di fare l'iret, che cosa dobbiamo fare? Dobbiamo aver ripristinato anche i registri generali, perché l'iret ripristina i registri speciali dalla cima dello stack, ma non ripristina i registri generali. Quindi questo stato, deve già aver ripristinato R1 e R2 nel processore prima di procedere con l'iret. A questo punto, il PC che punta alla locazione 5100 punta all'iret, da questo stato la prossima istruzione da eseguire sarà l'iret.

Context switch - example

5) during extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5100
PS	AA45
SP	1002
R1	4500
R2	CD31

6) at the end of IRET USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

L'iret prende i registri speciali, dalla cima del nucleo e li copia nel processore. Questo passo è sufficiente per ripristinare il thread T1 in esecuzione. Perché T1 si ritrova il suo PC, il suo SP, i valori che aveva quando è stato interrotto e si ritrova anche la sua PS, di conseguenza si ritrova in stato utente ad interruzioni abilitate. Lo stack del nucleo è pulito, il TCB di T2 non è stato toccato, il TCB di T1 non è stato toccato perché in questo caso qui particolare, T1 ha invocato una chiamata di sistema non bloccante, che gli ha restituito subito il controllo, per cui non c'era bisogno di salvare i registri nel TCB.

Context switch – example 2

Hyp. B): thread T1 invokes a system call that switches T2 in RUNNING state

1) Initial situation during the execution of SVC instruction (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
1000	
1001	
1002	
1003	
1004	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

kernel SP	0FFF

Vediamo l'altro caso. Supponiamo che T1 esegua una chiamata di sistema bloccante che comporta la commutazione di contesto con T2. Di nuovo, questo è lo stato iniziale, esattamente con le stesse informazioni di prima. Il thread T1 esegue la SVC.

Context switch - example

1) Initial situation during the execution of SVC instruction (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
1000	
1001	
1002	
1003	
1004	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF

2) After interrupt (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5000
PS	AA45
SP	1002
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF

Nel momento nel quale, l'SVC viene eseguita nel ciclo di estrazione – esecuzione successivo, viene riconosciuta l'interruzione, di nuovo il processore, in maniera automatica prende lo SP del nucleo, lo porta nello SP e contemporeneamente prende PC PS SP e li copia in cima allo stack del nucleo. Questo l'ha fatto il processore. Lo SP punta alla prima posizione libera dello stack.

Context switch - example

2) After interrupt (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5000
PS	AA45
SP	1002
R1	4500
R2	CD31

3) After temporary storage of registers (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	4500
1003	CD31
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

Dopo di che, a questo punto siamo in modalità supervisore a interruzioni disabilitate, l'handler salva il registri generali, quindi prende R1 R2 e li copia in cima allo stack del nucleo.

Context switch - example

3) After temporary storage of registers (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	4500
1003	CD31
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

4) After storage of registers of T1 and restore of registers of T2 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	A12C
1000	16F2
1001	A275
1002	25CC
1003	F012
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

A questo punto, viene invocata la funzione di gestione della chiamata di sistema, che poi però invocherà l'esecuzione dello scheduler, perché questa qui corrisponde ad una Yield. Quindi da questo stato, dal momento in cui l'handler invoca la funzione che attua la commutazione di contesto, al momento in cui è stata eseguita. Che cosa cambia? Dobbiamo salvare i registri di T1 all'interno del suo TCB in modo sicuro, quindi tutto ciò che stava in cima allo stack del nucleo è stato copiato dentro il TCB di T1. T1 è stato messo in stato di attesa, non era una Yield, era una chiamata di sistema bloccante e quindi T1 si sospende, si invoca lo scheduler, che scopre che c'è T2 che può essere messo in esecuzione, commuta lo stato di T2 da Ready a Running, e poi copia i registri dal TCB di T2 in cima allo stack del nucleo. A questo punto, la commutazione di contesto è quasi finita, perché T1 è salvo nel suo TCB ed è in attesa, T2 è già stato messo in stato di esecuzione formalmente, ma in pratica ancora non lo è, perché dobbiamo concludere la chiamata di sistema. Per concludere la chiamata di sistema dobbiamo copiare i registri dalla cima del nucleo nel processore.

Context switch - example

4) After storage of registers of T1 and restore of registers of T2 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	A12C
1000	16F2
1001	A275
1002	25CC
1003	F012
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

5) During extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	A12C
1000	16F2
1001	A275
1002	
1003	
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1002
R1	25CC
R2	F012

Quindi che si fa? Prima di tutto copiamo i registri generali e questo lo fa di nuovo l'handler tramite lo STUB, quindi dalla cima dello stack copia R1 e R2 nel processore. E a questo punto siamo pronti per eseguire l'iret.

Context switch - example

5) During extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	A12C
1000	16F2
1001	A275
1002	
1003	
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1002
R1	25CC
R2	F012

5) During extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
1000	
1001	
1002	
1003	
1004	

registers	
PC	A12C
PS	16F2
SP	A275
R1	4500
R2	CD31

Completato il ripristino dei registri generali, siamo in questo stato, eseguiamo la iret, che nuovamente fa il solito lavoro; prende dalla cima dello stack del nucleo PC PS SP e li copia nel processor. In questo modo abbiamo che il processore commuta in stato utente, abilita le interruzioni ed inizia ad eseguire le istruzioni dall'indirizzo A12C. L'indirizzo A12C è l'istruzione del processo T2, che quindi si ritrova in esecuzione, con il suo stack e con i suoi registri.

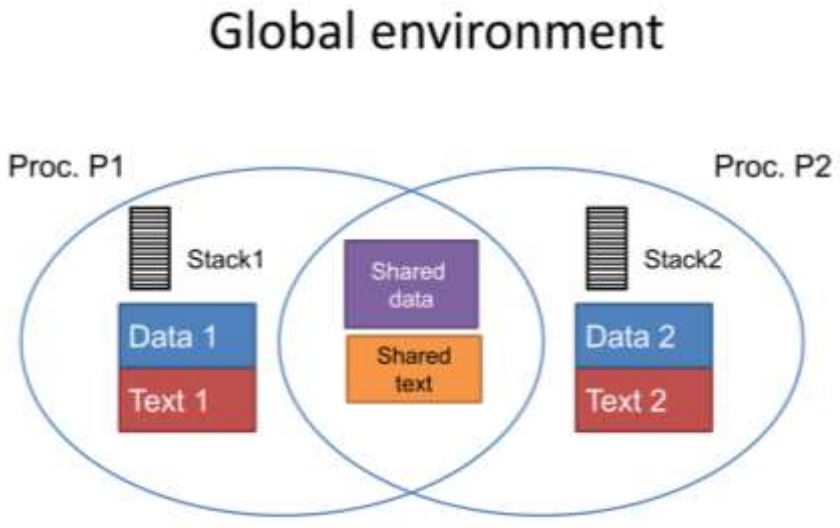
Sistemi operativi 13/03/17

Guardiamo brevemente la correzione dell'esercizio dell'altra volta. Vi ricordo che la situazione era questa: abbiamo due thread Ti e Tj con Ti in esecuzione e Tj in stato pronto quindi lo stato del processore dispone i registri speciali, i registri a stato utente e i registri a stato supervisore è quello mostrato qui ed ha senso per il Thread Ti. Per cui in particolare i registri R1 R2 R3 R4 sono quelli aggiornati in Ti, PC PC e SP sono (?) con Ti. Ti contiene una copia dei registri nel suo descrittore ma le copie contenute nel descrittore non sono più valide perché nel frattempo nell'ultimo salvataggio Ti è in stato di esecuzione quindi i valori dei registri sono cambiati. Dobbiamo far vedere lo stato dei descrittori dei registri durante la fase di estrazione della funzione di servizio quindi cosa vuol dire? Ti ha generato una SVC, l'interruzione settata dall'SVC è stata riconosciuta dal processore, il processore ha attivato il meccanismo di riconoscimento dell'interruzione e a questo punto si sta apprestando ad eseguire la prima istruzione dell'handler. Quello che fa il processore per via del meccanismo di interruzioni in maniera totalmente automatica prende i valori di PC e PS e li copia in cima allo stack del nucleo, non ha bisogno di toccare lo Stack Pointer perché dispone di un doppio banco di registri quindi lo Stack Pointer del processo utente non ha bisogno di toccarlo, va ad agire soltanto sullo Stack Pointer riservato allo stack del nucleo. Questo è quello che il processore deve fare, in alcuni casi il processore potrebbe salvare anche qualcosa in più o qualcosa in meno, questo lo vedremo di volta in volta. Contemporaneamente dopo aver salvato PC e PS, prende dal vettore delle interruzioni il valore PC e PS da caricare all'interno del processore. Quindi fatto questo, l'interruzione ha completato il suo lavoro, il meccanismo di riconoscimento dell'interruzione ha completato il suo lavoro, il processore inizia il prossimo ciclo di estrazione ed esecuzione all'interno del quale andrà ad estrarre l'istruzione posizionata alla locazione 0425 che stavolta è un'istruzione dell'handler. Vi ricordo che quando andrà ad eseguire la prima istruzione dell'handler il valore di PS è cambiato (275E) e al suo interno tra le altre cose codificherà che il processore è in stato supervisore e che le interruzioni sono disabilitate. Quindi fatto questo, viene eseguita la prima istruzione dell'handler e cosa farà? Avendo un doppio banco non ha bisogno per lavorare di manipolare i registri generali stato utente, lo può fare direttamente andando a lavorare con i registri a stato supervisore. Cosa farà questo hander? Capisce che è stata invocata una SVC, capisce qual è la funzione che è stata richiesta dal processo, invoca la funzione del sistema operativo corrispondente, questa funzione del sistema operativo decide, per qualche motivo, di deschedulare Ti e di mettere in esecuzione Tj. Nel far questo deve salvare lo stato di Ti nel suo descrittore e deve caricare lo stato di Tj in modo tale da predisporre tutto per l'esecuzione di Tj. Noi dobbiamo mostrare quello che succede quando arriviamo all'esecuzione iRet, un istante prima di eseguire l'iRet. Per sapere quello che dobbiamo fare prima di eseguire l'iRet, dobbiamo ricordarci cosa fa l'iRet. L'iRet prende dei registri dalla cima dello Stack e li carica nel processore. In cima allo stack cosa troverà l'iRet? L'iRet si comporta esattamente al contrario del meccanismo di interruzione quindi in questo caso il meccanismo di interruzione ha salvato PC e PS, stavolta quando verrà invocata l'iRet, ripristinerà PC e PS dalla cima dello stack. Questo vuol dire che tutto il lavoro che l'iRet non fa per mettere Tj in esecuzione deve essere fatta dal sistema operativo, quindi prima dell'iRet. La funzione di gestione della SystemCall prima dell'iRet deve aver preso i registri generali in stato utente che contengono valori associati a Ti e li deve aver copiati all'interno del descrittore di Ti. non basta perché i registri speciali PS e PC, conservati in cima dello stack del nucleo sono valori che hanno un senso per Ti e quindi devono anche essi essere memorizzati all'interno del descrittore di Ti. Fatto questo lo stato di Ti è al sicuro nel suo descrittore, può procedere a mettere Tj in esecuzione. Siccome Ti non andrà più in esecuzione il suo stato transita in "pronto", il suo descrittore di conseguenza verrà messo nella lista dei descrittori pronti mentre estrarrà il descrittore di Tj e dal descrittore di Tj preleverà i valori "gialli" (che sono i registri generali) e li ricopierà all'interno dei registri del banco utente. Prenderà i valori PC e PS del descrittore Tj e li copierà in cima allo stack del nucleo ed infine bisognerà mettere il descrittore Tj in stato "esecuzione" all'interno della lista dei thread in esecuzione. A questo punto il codice del sistema operativo si deve fermare, non può fare altro perché l'unica cosa che manca in realtà è quella di caricare i valori di PC e PS di Tj ma questo viene fatto dall'iRet ad hardware per garantire che il ripristino del Program Counter e della Program Status Word venga fatto contestualmente in una sola operazione. Quando restituiamo il controllo a Tj non basta saltare all'istruzione di Tj, bisogna tornare in stato utente e bisogna

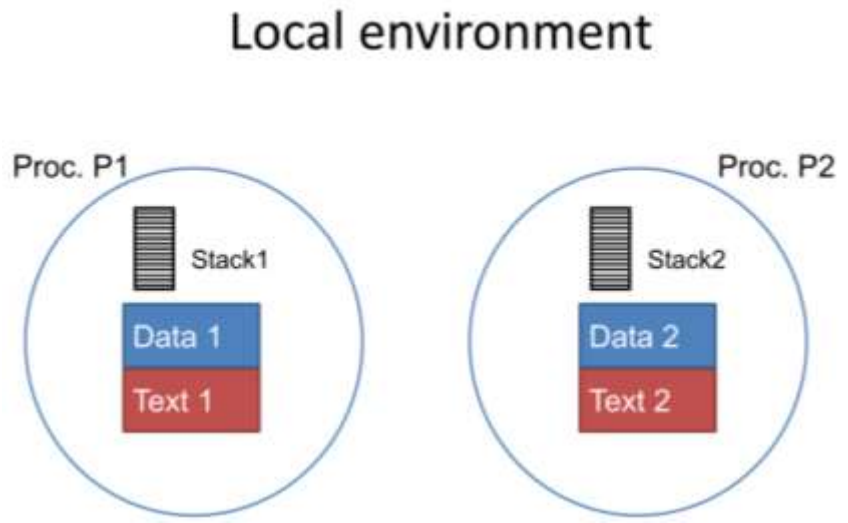
riabilitare nuovamente le interruzioni. Quindi nel fare tutto questo la funzione di gestione dell'SVC ha lavorato usando i registri dello stato supervisore e ha modificato i valori di PC del processore andando a puntare di volta in volta altre istruzioni di questa funzione all'interno del nucleo. Per cui in realtà i valori contenuti in questi registri non li conosciamo, sono però riassociati all'esecuzione di una funzione del nucleo. La cosa importante è che nello stato utente, i registri a stato utente siano fra quelli di Tj, in cima allo stack del nucleo ci siano i valori di Tj, lo stack del nucleo punti correttamente alla cima. a questo punto si può eseguire l'iRet. L'iRet prende i valori di PC e PS dalla cima del nucleo li copia nel processore di conseguenza aggiorna lo Stack Pointer e da questo istante in poi Tj è in esecuzione in stato utente a interruzioni abilitate.

Fine esercizio

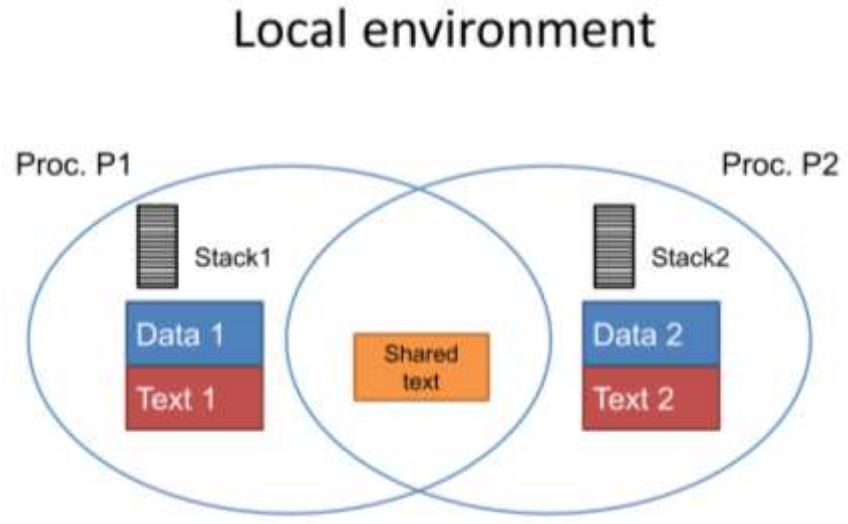
Abbiamo definito i processi, abbiamo definito i thread, possiamo iniziare a farli lavorare. Se vi ricordate l'obiettivo di introdurre i thread era quello di avere la possibilità di far lavorare più attività concorrentemente. Abbiamo visto che far collaborare i processi tra loro era un po scomodo perché ci di mezzo i meccanismi di protezione e per questo motivo sono stati introdotti i thread. Il modello di cooperazione che abbiamo con i thread e il modello di cooperazione che abbiamo con i processi sono diversi al punto che hanno anche un nome differente. Quando ci si riferisce a modelli di cooperazione tra entità che possono utilizzare una memoria condivisa allora si parla di modello globale dove i processi (oppure thread) condividono dati e questo viene fatto tramite la condivisione della memoria. Viceversa il modello nel quale i processi non condividono memoria viene detto ambiente locale. In questo caso ogni singola entità può utilizzare solo la propria memoria, non c'è una condivisione quindi la condivisione deve avvenire esplicitamente con uno scambio di messaggi. Nell'ambiente locale non abbiamo memoria condivisa. Nell'ambiente globale abbiamo una situazione di questo genere (11.55)



dove le entità possono avere dei loro dati ma hanno anche una zona condivisa, tipicamente si condivide il codice (se c'è codice condiviso tra i due) e si condividono i dati. Nel modello ambiente globale le entità che cooperano vanno ad agire direttamente sugli oggetti condivisi. Gli oggetti condivisi (che sono i cerchi) possono essere condivisi fra diverse entità in diverse modalità; ci possono anche essere oggetti proprietari, in questo caso P3 ha un accesso esclusivo all'oggetto 4 viceversa O3 è condiviso da P1 e P3, O2 è condiviso tra P1 e P2. Quindi in un contesto di condivisione di memoria, a livello globale può anche essere molto articolato. Questo è il modello che si sviluppa con i thread. I thread effettivamente condividono tutta la memoria del processo del quale fanno parte e poi sta al programmatore gestire l'organizzazione per cui lui decide cosa i singoli thread possono condividere tra loro sebbene non siano meccanismi di protezione forzati dal sistema operativo. Nell'ambiente locale invece abbiamo una situazione di questo tipo (13.20)



dove ogni processo possiede i propri dati, il proprio codice, il proprio stack ma per effetto dei meccanismi di protezione, non può condividere niente con gli altri processi. In effetti anche nei sistemi con un modello ambiente locale (quindi anche in UNIX ad esempio) un po di condivisione nascosta la si sfrutta lo stesso in particolare per quanto riguarda il codice condiviso. Se più processi caricano la stessa libreria, in un modello di questo tipo, ovviamente ci costringerebbe a caricare due volte la stessa libreria per tutti e due i processi e la cosa è abbastanza scomoda. Stesso discorso per la fork, quando facciamo la fork siamo costretti a copiare tutto il codice. D'altra parte il codice è un elemento passivo nel senso che non deve essere modificato/alterato, resta quello quindi è un qualcosa che si potrebbe condividere se ci fosse la possibilità ed è per questo motivo che nello UNIX attuale pur tenendo un modello di ambiente locale tra i processi, per cui i processi non condividono memoria ufficialmente, in pratica almeno il codice, quando possibile, viene condiviso. Se ci restringiamo ai dati invece, il modello all'ambiente locale è di questo tipo (14.50)



per cui ogni processo utilizza soltanto i dati privati e non ha modo di andare a leggere o scrivere o alterare i dati che appartengono ad altri processi. Quindi la comunicazione deve avvenire con meccanismi espliciti per esempio con i canali di comunicazione. Quando un processo vuole passare informazioni ad un altro processo lo deve fare aprendo un canale verso quel processo e utilizzando delle chiamate di sistema che permettono di spedire informazioni esattamente come se questi processi si trovassero in una rete, in un sistema distribuito per cui non soltanto non condividono memoria logicamente ma non condividono neanche memoria fisica. Il modello ambiente locale è il modello che abbiamo tipicamente in internet nel quale i processi possono risiedere su macchine fisicamente differenti per cui neanche volendo possiamo condividere memoria e l'unica possibilità è quella di condividere dei canali di comunicazione e usare una comunicazione

esplicita. Il modello attuale quindi lo troviamo in due contesti allo stato attuale, tra i processi UNIX o tra i processi distribuiti tipo internet. Il modello ambiente globale lo troviamo se realizziamo il nostro processo con un processo multithread per cui tutti i thread di quel processo operano in un ambiente globale. Quando si opera in un ambiente globale i thread collaborano andando a leggere e scrivere le stesse locazioni di memoria, possiamo quindi allocare una variabile, questa variabile è accessibile da entrambi i thread, una modifica fatta da un thread è automaticamente visibile dall'altro thread. Questo modello di cooperazione comporta la necessità di sincronizzare i thread. Quindi la cooperazione in ambiente globale è molto semplice però comporta delle problematiche aggiuntive in particolare legate alla sincronizzazione. Vediamo questo esempio(17.26):

Synchronization Motivation

Thread 1	Thread 2
p = someFn();	while (! isInitialized) ;
isInitialized = true;	q = aFn(p);
	if q != aFn(someFn())
	panic

supponiamo di avere due thread, di aver scritto un processo con due thread, un primo thread si preoccupa di inizializzare una certa struttura e il secondo thread la ottimizza, quindi chiaramente la struttura non può essere utilizzata prima di essere inizializzata. Come risolviamo questo problema? Siamo in ambiente globale, dopo aver inizializzato la struttura il thread che la inizializza setta la variabile IS_INITIALIZED a TRUE. In questo caso la struttura si chiama P(una variabile di qualche tipo) quindi dopo aver inizializzato P in qualche maniera impostiamo la variabile inizializzata a TRUE, a questo punto l'altro thread, che deve utilizzare P, inizierà con un codice di quel tipo (18.19) finchè la variabile IS_INITIALIZED non è TRUE cicla, quindi aspetta, nel momento in cui si esce da questo while siamo sicuri che questa variabile è stata inizializzata per cui possiamo leggere un valore. Ora in questo caso facendo un giochino strano prendiamo il valore di P applichiamo la funzione di A e quindi Q in qualche modo sarebbe la funzione A applicata a qualche funzione dopo di che controlliamo che Q ha effettivamente il valore che ci aspettiamo e se per caso non torna allora "panico". Questo codice è corretto, questo modo di sincronizzare questi due thread funziona?

Risposta collega 1: Il vostro collega dice che non funziona perché non possiamo fare assunzioni sull'ordine in cui i due thread verranno eseguiti. Ma questo non basta perché dovremmo (per dimostrare che effettivamente non funziona) provare questi thread, supponendo che vengano interrotti in punti arbitrari, e far vedere che c'è almeno un caso in cui i due vanno in panico. Se non siete in grado di farlo allora il codice funziona.

Risposta collega 2: il vostro collega dice che abbiamo una risorsa condivisa P di cui T1 la scrive, T2 la utilizza ma non so in che ordine avverrà. In effetti proprio per forzare un ordine in questo codice utilizzo un'altra risorsa condivisa che è la variabile "inizializzato"(IS_INITIALIZED) per cui se IS_INITIALIZED è FALSE (supponiamo che sia così) allora quando T2 viene messo in esecuzione, se questa variabile è false, resto intrappolato all'interno di questo loop e ne esco solo quando questa variabile diventa TRUE ma l'unico che può fare questo è il thread T1 e lo farà solo dopo aver dato un valore a P. Quindi è possibile che T2 utilizzi P prima che T1 l'abbia inizializzata? No.

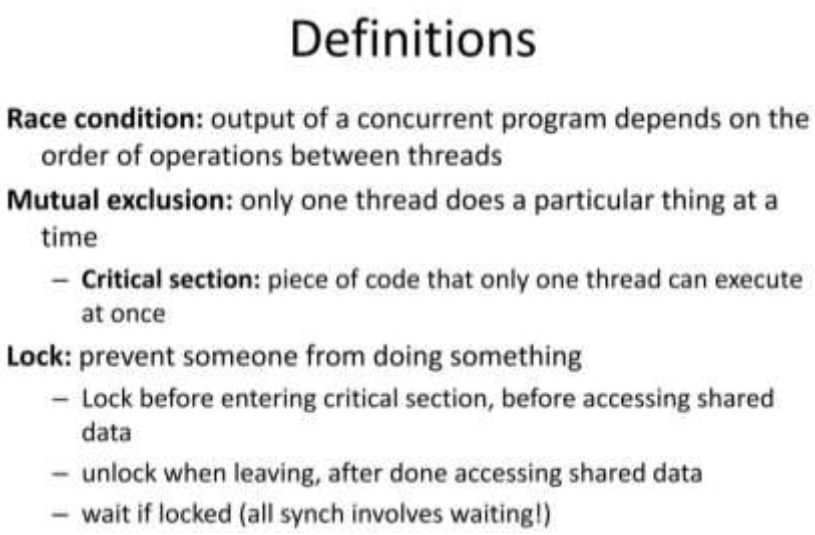
Questo è un punto chiave, la sincronizzazione è l'argomento più importante del corso. In teoria si, questo codice funziona e T2 non va mai in panico, disgraziatamente nei sistemi moderni possono capitare diverse cose. Il compilatore potrebbe rendersi conto che la variabile IS_INIZIALIZED è sempre TRUE cioè non viene mai utilizzata, non le viene mai data un altro valore, oppure potrebbe rendersi conto, quando va a compilare T2 che non c'è nessun problema nell'eseguire prima l'uno e poi l'altro perché dal suo punto di vista quando compila T2 nelle inizializzazioni di P la variabile IS_INIZIALIZED non è utilizzata. Qui il compilatore non può sapere cosa abbiamo in mente quindi non sa che stiamo utilizzando questa variabile (24.21) per sincronizzare due thread. Noi abbiamo scritto un codice C sequenziale che è quello di T1, lo compiliamo e il compilatore per questioni di ottimizzazione, può stabilire che l'ordine delle istruzioni deve essere invertito perché in questo modo il processore è più veloce. Si può rendere conto che sebbene IS_INIZIALIZED abbia come valore iniziale FALSE in realtà non è mai utilizzata quando è FALSE nel codice di T1 quindi la mette direttamente a TRUE, la definisce già come costante vera per risparmiare un'operazione. Soltanto al programmatore, che scrive il codice di T1 e T2 che sa che questo codice ha un problema di sincronia e che utilizza questa variabile per risolvere quel problema di sincronia, soltanto per questo programmatore è chiaro che IS_INIZIALIZED deve essere eseguita dopo l'assegnamento di P = SOMEFUNCTION ma questo potrebbe non essere chiaro per il compilatore perché nessuno gliel'ha detto e perché il codice può essere compilato in tempi differenti e non ha proprio modo di saperlo. Quindi uno potrebbe compilare questo codice, impostando dei parametri per il compilatore in modo tale che il compilatore non faccia pasticci. Però se noi abbiamo scritto questo codice oggi, se un domani questo codice va da un'altra persona che si rende conto che l'abbiamo sempre compilato nel Makefile disabilitando una certa ottimizzazione, quest'altra persone potrebbe riattivare questa ottimizzazione, per 100 volte questo codice funziona, però quando viene dato al committente va il crash e fa danni. C'è anche un altro problema, non è soltanto il compilatore che può ordinare le istruzioni ma è anche l'hardware. Nei processori moderni, per andare più veloci, il processore può avere degli stati all'interno dei quali viene ordinata la sequenza di alcune istruzioni per andare più rapidamente, che succede se il processore se per motivi suoi decide di spostare quest'istruzione prima o magari durante l'inizializzazione per qualsiasi motivo? Succede che l'altro thread potrebbe trovare la variabile inizializzata a TRUE, esce, inizia ad utilizzare la variabile per calcolare Q e calcola un valore sbagliato. Cosa si capisce da qui? (27.28) si capisce che un problema di sincronizzazione esiste e nasce nel momento nel quale io ho un codice multithread che condivide memoria, se decido di dividere il lavoro tra più thread può capitare benissimo che un thread debba fare una cosa prima ed uno la debba fare dopo però un meccanismo artigianale per fare questo non ci aiuta, ci serve un meccanismo più strutturato. Vediamo un esempio più articolato (27.55).

Too Much Milk Example

	Person A	Person B
12:30	Look in fridge. Out of milk.	
12:35	Leave for store.	
12:40	Arrive at store.	Look in fridge. Out of milk.
12:45	Buy milk.	Leave for store.
12:50	Arrive home, put milk away.	Arrive at store.
12:55		Buy milk.
1:00		Arrive home, put milk away. Oh no!

Supponiamo di avere due coinquilini A e B (presumibilmente sono due DJ perché lavorano la notte e si svegliano tardi) il primo si alza alle 12:30, va a fare colazione, apre il frigorifero, si rende conto che non c'è il latte quindi va a comprare il latte. Alle 12:35 esce di casa, alle 12:40 arriva al negozio, contemporaneamente

alle 12:40 si sveglia il coinquilino. Anche lui va a fare colazione, apre il frigorifero, si rende conto che manca il latte, si veste di corsa e alle 12:45 esce per il negozio mentre alle 12:45 l'altro sta comprando il latte, a questo punto alle 12:50 il primo torna a casa, farà colazione e metterà il latte nel frigo. B alle 12:50 arriva al negozio, alle 12:55 compra il latte alle 13:00 arriva a casa, mette il latte in frigorifero e si rende conto che c'era già. Cosa sta succedendo in questo caso? Abbiamo due attività parallele, perché le due persone non fanno time-sharing con il cervello, che sviluppano una "corsa critica". Cos'è una corsa critica? Tutti e due stanno svolgendo una certa attività ma complessivamente stanno collaborando, la collaborazione sta nel comprare il latte se manca e o lo fa uno o lo fa l'altro. Il problema è che questa attività avrà successo, quindi sarà eseguita correttamente, se il coinquilino A riesce ad arrivare a casa prima che il coinquilino B vada ad aprire il frigorifero, quindi c'è una corsa critica sotto questo punto di vista tra le due attività. Se queste due attività vengono eseguite in modo separato quindi un'attività si completa prima che inizi l'altra, il problema non si pone. Se il coinquilino B invece che svegliarsi alle 12:40 si fosse svegliato alle 12:55 non ci sarebbe stato nessun problema. Abbiamo due attività parallele, in questo caso, che sviluppano una corsa critica. In particolare di tutte le cose che fanno queste due persone nella loro vita questa particolare fase (quindi nel caso del B dalle 12:40 alle 13:00 e nel caso di A dalle 12:30 alle 12:50) è una parte particolare di attività che svolgono, di cooperazione, che sarebbe svolta correttamente se avessimo potuta farla in "mutua esclusione". Cioè il fatto che la faccia uno esclude il fatto che la faccia l'altro e viceversa. Ci saranno altre cose nella loro vita che non dovranno essere fatte in mutua esclusione, che potrebbero essere fatte contemporaneamente (se B decide di andare a trovare i genitore mentre A va a trovare la fidanzata sono due attività che possono essere svolte contemporaneamente perché non sono attività che richiedono collaborazione) ma nelle attività che richiedono collaborazione si possono sviluppare corse critiche e di conseguenza abbiamo bisogno di mutua esclusione. Le attività che comportano collaborazione come in questo caso (32.30)



sono dette "sezioni critiche". Quindi il race_condition (corsa critica) avviene quando il risultato di un'azione concorrente tra più attività dipende dall'ordine con il quale queste due attività sono state eseguite. Per garantire o per forzare l'ordine di queste due attività è necessario avere dei vincoli in particolare i vincoli di mutua esclusione che dicono sostanzialmente che se un'attività sta eseguendo una certa azione allora nessun altro la può svolgere contemporaneamente, possono fare altro ma non devono svolgere quell'operazione. Per intenderci, mentre la persona A sta andando a comprare il latte, la persona B continuerà a dormire che è un'altra azione, ma non è mutualmente esclusiva con l'andare a comprare il latte o se A sta andando a comprare il latte B può andare a comprare il pane, nuovamente sono due azioni ma non devono essere eseguite in mutua esclusione tra loro. Le azioni in mutua esclusione sono quelle che in qualche modo sono della stessa classe: se tutti e due comprano il latte, se tutti e due comprano il pane. La sezione critica dal nostro punto di vista è una porzione di codice di un thread che utilizza una struttura dati condivisa. Quindi con questa definizione andare ad individuare le sezioni critiche è abbastanza semplice, quando andiamo a

scrivere i thread se c'è una struttura dati condivisa, individuiamo le porzioni di codice che la utilizzano, quelle sono sezioni critiche. Quelle sezioni critiche vanno protette con un'esecuzione in mutua esclusione. Come si fa a garantire la mutua esclusione? Ci sono dei meccanismi, non è una cosa che viene gratis e dobbiamo avere un meccanismo che permetta ad un thread di dire "sto iniziando ad eseguire una sezione critica quindi acquisisco la mutua esclusione e poi quando ho terminato cedo la mutua esclusione perché ho terminato la sezione critica." Questo meccanismo è il meccanismo di "lock" ed è un meccanismo che impedisce a qualcuno di fare qualcosa quindi un qualcosa che un thread acquisisce, a questo punto una volta acquisito nessun altro lo può ottenere e una volta che un thread ha acquisito un lock può eseguire la sezione critica perché nessun altro può ottenere il lock fino a quando lui lo tiene, gli altri thread che non possono avere il lock devono attendere in qualche forma che chi ha il lock lo ceda. Una volta terminata la sezione critica il thread che l'acquisita lo può cedere, cede il lock e può essere acquisito da qualcun altro che vuole eseguire a sua volta la sezione critica. È importante ricordarsi che ogni volta che abbiamo un meccanismo di lock o qualsiasi cosa di equivalente avremmo anche delle attese. Quindi ogni qual volta voglio fare qualcosa ma non la posso fare perché c'è una sezione critica in mutua esclusione quindi qualcun altro ha acquisito la lock io dovrò in qualche forma aspettare. Abbiamo visto delle sezioni critiche nelle volte scorse? Non esplicitamente però in qualche forma sono state citate. Quando arriva un'interruzione, dobbiamo andare a salvare i registri del processore in cima allo stack ma questa operazione è critica perché se durante questa operazione arriva un'altra interruzione potremmo pasticciare con i registri e pasticciare con lo stack. Quindi l'operazione di gestire l'interruzione, di salvataggio dei registri è una sezione critica e deve essere eseguita in mutua esclusione perché un'altra interruzione attiverebbe l'esecuzione di un'altra sezione critica in concomitanza. Come si fa nel caso del processore? Si disabilitano le interruzioni. Quindi il motivo per il quale quando c'è una chiamata di sistema o un'interruzione le interruzioni vengono disabilitate è proprio questo per permettere l'esecuzione in mutua esclusione di una sezione critica che è quella che salva i registri del processore nello stack e poi per contro disabilitare le interruzioni viene fatta quando si ripristinano i registri. Torniamo all'esempio del troppo latte, quali sono i problemi che si possono sviluppare? I problemi sono due: liveness e safety (37.54)

Too Much Milk, Try #1

- Correctness property
  - Someone buys if needed (liveness)
  - At most one person buys (safety)
- Try #1: leave a note

```
if !note
  if !milk {
     leave note
     buy milk
     remove note
  }
```

Il problema di liveness è che qualsiasi soluzione io metta in piedi deve garantire che se non c'è latte qualcuno lo compri. Non posso mettere in piedi una soluzione usando dei meccanismi di lock, sezione critica o altro che non garantisca questa proprietà. È necessaria per la correttezza del mio codice. È necessario che uno solo lo faccio però almeno uno lo deve fare. "almeno uno" è la proprietà di liveness "al più uno" (ovvero che venga fatta una sola volta) è una proprietà di safety. La prima proprietà è di liveness che se manca il latte qualcuno lo compri quindi in qualche modo il sistema deve restare vivo, qualcosa deve succedere quindi almeno uno compra il latte ma questa non basta perché come abbiamo visto dall'esempio precedente, l'esempio era live

ma non era safe perché si acquistava troppo latte quindi quello che devo garantire è che ciò che avvenga avvenga nella misura giusta non oltre quindi nell'esempio precedente che il latte venga comprato una sola volta, non più di una e questa è una proprietà di safety. In generale per avere un codice corretto dobbiamo garantire entrambe cioè dobbiamo garantire che quello che deve avvenire avvenga ed inoltre avvenga nella misura giusta quindi "che avvenga" è una proprietà di liveness e "che avvenga nella misura giusta" è una proprietà di safety. Detto questo possiamo provare a modificare il codice per vedere se riusciamo a mettere queste due persone in condizione di acquistare il latte rispettando le proprietà di safety e di liveness. Un'idea potrebbe essere che se per caso non c'è il latte lascio una nota(40:56) e vado a comprare il latte.

Too Much Milk, Try #2

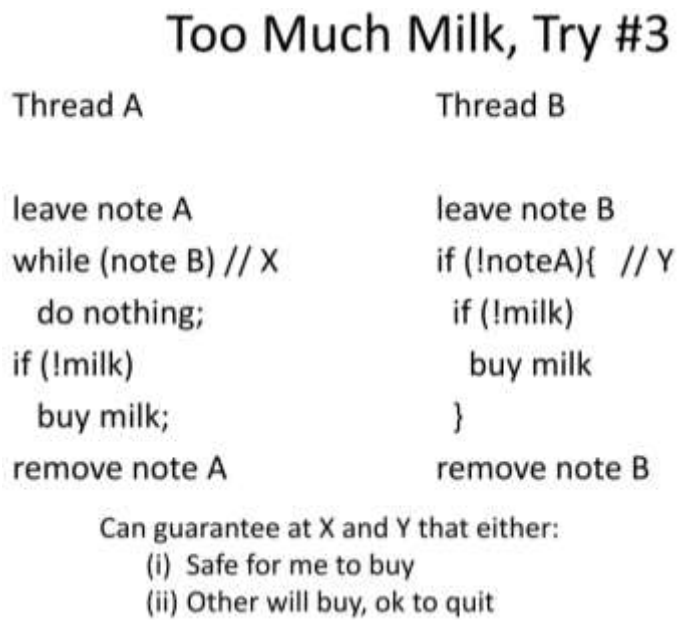
Thread A	Thread B
leave note A	leave note B
if (!note B) {	if (!noteA){
if (!milk)	if (!milk)
buy milk	buy milk
}	}
remove note A	remove note B

Questa nota serve a quell'altro in modo tale che se trova la nota, anche se manca il latte, non lo va a comprare. Quindi potrei scrivere una soluzione di questo genere. Il thread A quando si sveglia prima di far qualsiasi cosa, quando arriva al frigorifero controlla se non c'è una nota perché la presenza della nota vuol dire che l'altro thread è già uscito per comprare il latte. Quindi se non c'è la nota allora controllo se non c'è il latte lascio la nota, vado a comprare il latte e quando torno la rimuovo, stessa cosa fa il thread B. quindi sto usando il meccanismo della nota come meccanismo di sincronizzazione. Funziona? Ipotizziamo che il compilatore non sia ottimizzato e che l'hardware non rovini le istruzioni, ci concentriamo sulla natura del problema. Il vostro collega dice che se sono eseguiti contemporaneamente abbiamo un problema perché l'esecuzione in contemporanea comporta che tutte e due vanno a controllare che non ci sia la nota, nessuno dei due la trova, vanno tutti e due se c'è il latte, nessuno dei due lo trova, vanno entrambi a comprare il latte. L'esecuzione in contemporanea richiede un multiprocessore, supponiamo di avere un unico processore. Anche se non c'è parallelismo (c'è time-sharing dunque) può capitare che il thread A vada a controllare se non c'è la nota, venga deschedulato, passa in esecuzione il thread B che vede se c'è la nota, non la trova, controlla se non c'è il latte, lascia la nota, viene deschedulato, ritorna in esecuzione il thread A, controlla se c'è latte e vede che non ce n'è, lascia anche lui una nota e a questo punto in qualsiasi ordine vengano eseguiti tutti e due andranno a comprare il latte. Il parallelismo spesso può aumentare i problema ma se ci sono problemi con il parallelismo ci sono anche con la concorrenza. Dove sta il problema di questa soluzione in realtà? Il problema sta nel fatto che la nota è stata messa troppo tardi perché potrei adoperare una soluzione di questo tipo (45.38) anziché lasciare la nota dopo la lascio prima, i thread possono distinguere la propria nota da quella dell'altro quindi il thread A lascia la sua nota (nota A), se non c'è la nota di B va a comprare il latte e B fa la stessa cosa.

PAUSA

Dov'è ora il problema? Il vostro collega dice che ora nessuno compra il latte, se vi ricordate le due proprietà che dobbiamo garantire sono le proprietà di liveness e di safety. Quale proprietà viene ora violata? Quella di liveness. Può capitare che il thread A lasci la nota, venga deschedulato, il thread B lasci la nota, vede che c'è

già la nota di A ma viene deschedulato, il thread A vede che c'è già la nota di B a questo punto toglie la sua nota, il thread B a questo punto toglierà anche lui la sua nota e a questo punto nessuno compra il latte. Quindi non abbiamo una soluzione al problema. Dove sta in realtà il problema? Che per quanto ci sforziamo, se noi realizziamo del codice perfettamente simmetrico, non ne usciamo da questo problema. Quello che dobbiamo fare è di cercare di rompere la simmetria tra il thread A e B. guardiamo l'esempio (2.30)

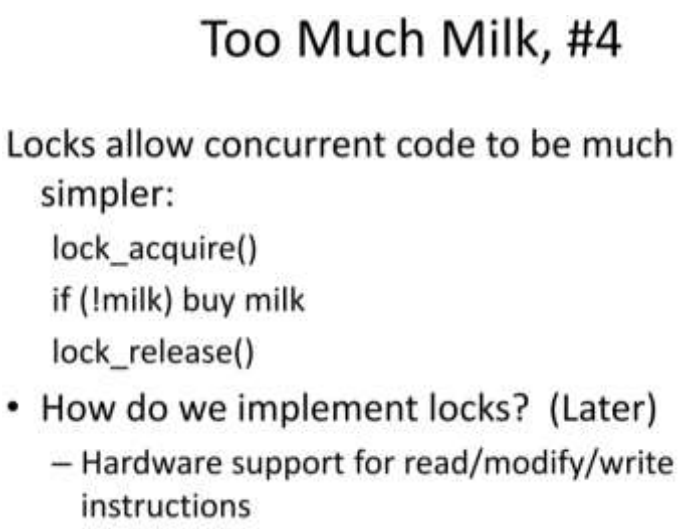


ci sono diverse cose da notare in questo codice; la prima cosa è che non è più simmetrico e che anche se è molto piccolo il codice è molto complesso. Cosa fa il thread A? lascia la nota, poi fintanto che c'è la nota di B aspetta, quindi ci sono due possibilità: la nota c'è o non c'è. Se c'è la nota aspetta che B la rimuova, se non c'è la nota e non c'è il latte lo va a comprare e in questo modo rimuove la sua nota. Viceversa B lascia la sua nota, se non c'è la nota di A e se non c'è il latte lo va a comprare e poi rimuove la nota. Perché abbiamo questa asimmetria? Perché in qualche modo uno dei due codici garantisce la safety e l'altro la liveness (anche se non è propriamente così). Come si fa a vedere che questo codice funziona? Questo codice può garantire in particolare che nei punti del codice X e Y si verificano due possibilità: nel punto X, A verifica se per lui è sicuro comprare e nel punto Y invece B verifica che l'altro comprerà e quindi è inutile per lui uscire. Quindi in qualche modo nel punto Y stiamo garantendo la safety e al punto X stiamo garantendo la liveness. Che cosa succede quando eseguo questo codice(5.21)? In generale quando trovate del codice concorrente, per farvi un'idea di cosa potrebbe succedere, provo ad eseguire quel codice mentalmente o carta e penna, provando a deschedulare i thread nel momento nel quale potrebbe sembrare più svantaggioso per loro essere deschedulati quindi mettendoli in difficoltà, provando quindi diverse combinazioni cercando di farmi un'idea di cosa succede e del perché certe proprietà vengono garantite ed altre no. Questo non è una dimostrazione ma permette di capire il problema e il problema del codice. Quando uno ha capito il comportamento del codice può abbozzare una dimostrazione di correttezza. Vediamo cosa succede con A e con B in questo caso (6.21). Supponiamo arrivi prima A e lui lascia la nota, se B a questo punto non è proprio arrivato, A si rende conto che nel while la nota B non c'è e quindi se serve va a comprare il latte con sicurezza. Quindi in realtà se serve il latte e non c'è la nota di B, A sicuramente compra il latte. Supponiamo invece che A abbia lasciato la sua nota e trova la nota di B, il problema è che se c'è la nota di B, A e B stanno facendo contemporaneamente queste sezioni critiche, quindi il comportamento di A e di B è dipendente l'uno dall'altro. Può darsi, se B ha lasciato la nota, che B compri il latte come può darsi che non lo compri, allora A in ogni caso, aspetta fintanto che B non ha rimosso la nota perché se B ha comprato il latte a quel punto A non lo comprerà perché lo troverà già. Se B non ha comprato il latte allora A lo comprerà. Siccome A in questo caso aspetta, io non posso avere un comportamento simmetrico per B altrimenti rischierei di far aspettare anche B e l'uno e l'altro si aspetterebbero all'infinito. Il comportamento di B deve essere differente per

funzionare. Se entrambi hanno messo la nota, A aspetta quindi tocca a B se comprare o meno il latte, una volta che B avrà deciso se comprare o meno il latte, avrà fatto ciò che deve fare e quindi avrà rimosso la nota, a quel punto A potrà andare avanti e decidere se comprare il latte oppure no dopo che B avrà preso la sua decisione quindi si sta cercando di serializzare questi due momenti. Fintanto che A aspetta, non avrà deciso se comprarlo o no, sta solo aspettando che B abbia preso la sua decisone. Come si comporta B? B prima di tutto lascia la nota, se non c'è la nota di A vuol dire che B è arrivato prima e se non c'è il latte rimuove la nota e lo compra. Se invece B, dopo aver lasciato la nota, trova che c'è anche la nota di A, allora vuol dire che sono arrivati concorrentemente, stavolta però B può sfruttare la sua conoscenza del comportamento di A, sa come si comporterà, nel particolare sa che A aspetterà B e poi dopo procederà a comprare il latte. Quindi se c'è già la nota di A, B non fa niente e rimuove la sua nota. Perché? Se c'è la nota di A potrebbero succedere due cose: potrebbe succedere che A abbia lasciato la nota ma non abbia visto la nota di B e quindi procederà a comprare il latte, oppure può darsi che A avrà lasciato la nota, ma abbia visto anche la nota di B quindi aspetterà. Quindi B in questa fase non può sapere se A comprerà il latte oppure no però sa che A o sta comprando il latte o sta aspettando. Se B non compra il latte e rimuove la sua nota che succede? Se A sta comprando il latte, lo compra e va bene se A sta aspettando, come B rimuove la nota A andrà a comprare il latte e va ancora bene. La storia di questa cosa qui è interessante perché parliamo degli anni 60 quindi i ricercatori stavano iniziando a scontrarsi con questa forma, la prima soluzione software a questo problema è stata presentata da Dijkstra a Pisa dove lui nel suo lavoro ha presentato il problema, ha presentato la soluzione (di quella che abbiamo visto è una derivazione) e poi ha fatto una riflessione cioè ha detto che certamente possiamo trovare una soluzione software al problema e questo è un esempio però non è una cosa raccomandabile perché la soluzione è complicata, non è una cosa immediata. Al tempo non si parlava ancora di compilatori ottimizzati che riordinano le istruzioni o di processori che riordinano le istruzioni. Lui diceva che comunque la soluzione software è complicata non è immediato per chi legge il codice capire cosa sta succedendo. Questa struttura quindi messa in codici complessi, rende ancora più complessi i codici da capire e da gestire. Il secondo problema è che non è neanche efficiente perché? Il vostro collega dice che quello che vediamo è una soluzione che funziona con due thread ma con n thread questa soluzione cosa diventa? Non va bene e andrebbe modificata ha ragione ma per questo problema c'è già una soluzione che è l'algoritmo di Peterson che risolve il problema con n thread via software ed è un concentrato di complicazioni però per l'algoritmo di Peterson come quelli di Dijkstra valgono esattamente le stesse considerazioni quindi anche questo non è efficiente. La risposta è che quando il thread A trovaa che c'è la nota di B, entra nel ciclo while nel quale aspetta ma il tipo di attesa che fa A che tipo di attesa è? Immaginate in un sistema time-sharing A trova la nota di B ed è all'inizio del suo quanto di tempo (quindi B non viene messo in esecuzione fino a quando A non ha esaurito il suo quanto di tempo) e per tutto il quanto di tempo A tiene impegnato il processore in un loop inutile. Quindi il vero problema è che il thread A che non può entrare nella sezione critica per vedere se c'è il latte oppure se non c'è, sta sprecando tempo del processore per fare un'azione inutile che non è utile né a se stessa né a B. sarebbe stato più utile in questa fase bloccare A, cedere il controllo a B e fare andare avanti B. più in generale A sta facendo attesa attiva, sta sia aspettando ma sta aspettando eseguendo altre istruzioni e quindi impegnando il processore. in generale le soluzioni che prevedono attesa attiva non sono delle buone soluzioni a meno di casi particolari. Quindi quello che serve è una soluzione senza attesa attiva. Questo è il nocciolo della questione che aveva tirato fuori all'epoca Dijkstra per cui oltre a questa soluzione ha presentato una soluzione alternativa che prevede però un aiuto da parte dell'hardware. Lui diceva "se i processori sono quelli che ci sono ora e i sistemi operativi rimangono quelli di ora non possiamo farci nulla quindi l'unica soluzione è una soluzione software di questo tipo però è svantaggiosa, c'è bisogno di avere un supporto dall'hardware e dal sistema operativo (poi vedremo in che forma) per realizzare dei meccanismi che garantiscono la mutua esclusione in maniera più efficiente". Questi meccanismi appena proposti sono meccanismi che prevedono una combinazione di hardware e lavoro del sistema operativo. Quindi le lezioni da trarre sono (19.42) che la soluzione è complicata, con l'algoritmo di Peterson con n nodi è ancora più complicata quindi difficile da utilizzare e da mantenere, dopo di che abbiamo un ulteriore problema perché sappiamo che i compilatori moderni e i processori moderni riordinano

le istruzioni (lo possono fare per lo meno) per cui una soluzione totalmente software di questo tipo rischia di fallire perché il processore ha invertito l'ordine delle istruzioni o magari per qualsiasi altra ottimizzazione non fa fare ad A certe cose perché il compilatore ritiene che sia inutile farle, elimina automaticamente alcune righe di codice, può succedere di tutto. Quindi in realtà ci serve un meccanismo che funziona e che sia anche a prova di ottimizzazioni e poi queste soluzioni mancano di generalità è difficile generalizzare più processori se non introducendo ulteriore complessità. A questo punto facciamo un altro esempio: (20.56) Fare un esempio di situazione nella quale del codice concorrente va in confusione, va in interferenza, le strutture dati vengono modificate in maniera scorretta e via discorrendo… (disegno). Immaginate di avere due processi che interagiscono tramite uno stack che è una delle strutture più semplici da immaginare. (qui processo o thread vengono usati in maniera indifferente come se fossero sinonimi ipotizzando che possano cooperare in ambiente globale) il comportamento atteso qual è? Quando P1 vuole inserire qualcosa in cima allo stack, incrementa la cima del puntatore e la cima dello stack e poi nello stack, nella posizione top, deposita il valore che vuole inserire. Viceversa il processo che vuole estrarre dallo stack cosa farà? Siccome è un processo che vuole inserire prima incrementa il puntatore e poi li deposita, il processo che estrae deve prima leggere dalla cima dello stack e poi decrementare il puntatore. Se questo codice semplice, che è chiaramente una sezione critica perché sto utilizzando la stessa struttura dati condivisa in due processi differenti. Quindi P1 sta accedendo alla struttura stack che è condivisa con il processo P2. Immaginate cosa può capitare se i due processi eseguono questa sezione critica in maniera concorrente. Se viene eseguito prima P1 e poi P2 non abbiamo mai problemi, lo stack è usato correttamente ma se invece usiamo concorrentemente per cui le istruzioni di P1 si possono mescolare a quelle di P2, può succedere questo: prima viene messo in esecuzione P1, incrementa lo stack, viene deschedulato, passa in esecuzione P2, estrae dalla cima dello stack, decrementa il puntatore alla cima, viene deschedulato, a questo punto P1 va a depositare il valore in cima allo stack e questa è chiaramente un'interferenza. Far vedere con un disegno come P1 e P2, essendo deschedulati in questo modo, vanno a leggere e scrivere dei valori sbagliati e lasciano la struttura inconsistente. (disegno) supponiamo che la situazione dello stack sia questa, come funziona lo stack? Devo inserire un nuovo valore, devo incrementare top e poi metto il valore quindi se top punta a questo elemento(cima dello stack), il valore che bisogna estrarre dallo stack in questo momento è c. Quindi chi vuole estrarre qualcosa deve estrarre c, se io voglio inserire qualcosa devo creare un nuovo elemento in cima allo stack e quindi y andrà inserito qua dentro. Quindi se viene eseguito prima P1 devo inserire y qui e top deve puntare qui. Se eseguo prima P2 devo leggere c e al posto di c deposito y. Se io eseguo prima P2 in questa situazione, z prende c e top punta ad a. Se io eseguo prima P1, top viene incrementato e y viene messo sopra c. Quindi se prima eseguo P1, P2 leggerà y, se io eseguo prima P2, P2 leggerà c e nello stack ci sarà scritto a,b,a. Questo è il comportamento normale che mi posso aspettare. Cosa succede se viene eseguito concorrentemente senza nessuna protezione riguardo alla mutua esclusione? Può capitare qualsiasi cosa, posso eseguire pirma P1 come posso eseguire prima P2 il problema sarà che leggerò dei dati sbagliati. Il dato corretto è che o P2 legge y se arriva prima P1 o P2 legge c se arriva prima P2 ma non può capitare che P2 legga qualcosa diversa da c o da y. Supponiamo che venga eseguito prima P1 ed esegua top++, a questo punto P1 viene deschedulato, passa in esecuzione P2, P2 legge z dalla cima dello stack, nella cima dello stack cosa c'è ora? Boh, sarà un valore a caso che non avrà niente a che fare con quello che volevamo. Decrementiamo top che quindi torna a puntare c, a questo punto passa all'esecuzione P1 che scrive in cima allo stack il valore di y quindi cancella c e ci scrive y. Abbiamo due problemi: 1) è che non sappiamo cosa ha letto P2, può essere qualsiasi cosa, 2) problema è che abbiamo perso dalla cima dello stack un valore che non è stato letto da nessuno, abbiamo sovrascritto. In generale quello che sta succedendo è un'interferenza. Questa interferenza è causata dal fatto che quel codice di P1 è una sezione critica, questo codice di P2 è una sezione critica e posso essere eseguiti concorrentemente ma in mutua esclusione quindi o l'uno o l'altro. La sezione critica è una porzione di codice che utilizza una struttura dati condivisa, qual è la struttura dati condivisa? È lo stack. Quali sono le variabili che descrivono e fanno parte di questa struttura? Non è soltanto l'array stack, la struttura si compone di due elementi, stack e la variabile top e sono tutte e due parte della stessa struttura. Quindi quando dovete andare ad individuare una sezione concorrente nel vostro codice, guardate bene.

Pensare che una struttura condivisa sia soltanto un array è sbagliato se quell'array viene utilizzato con delle variabili a supporto che servono per gestirlo e questo è esattamente il caso. Quindi di questa struttura fanno parte la variabile top e la variabile stack, top serve per accedere correttamente a stack, quindi la sezione critica è unica, non sono due sezioni critiche differenti, non c'è una sezione critica per top e una sezione critica per stack, c'è un'unica sezione critica che si riferisce a tutta la porzione di codice che accede al suo complesso. È evidente che per risolvere il problema ci serve un meccanismo che chiameremo meccanismo di lock, che permette di acquisire la mutua esclusione. Quali sono le proprietà di questi lock? Questi lock devono avere due meccanismi: un meccanismo per acquisirlo e un meccanismo per rilasciarlo. Il meccanismo per acquisire il lock verifica "se il lock è libero lo acquisisco", il meccanismo per rilasciarlo lo cede e se per caso qualche altro thread stava aspettando di ottenere il lock, quindi era bloccato nella lock acquired, la lock release deve riattivarlo, deve risvegliarlo.(36.00) Deve avere poi 3 proprietà: 1) deve avere al massimo un thread che possiede un particolare lock in un certo istante di tempo, questo per garantire la sicurezza (garantisce che non possono esserci due thread che contemporaneamente eseguono la stessa sezione critica della stessa classe, sulla stessa struttura). 2) Se il lock è libero quindi nessuno lo possiede, allora chi lo chiede lo deve ottenere (questo garantisce che ci sia un progresso quindi liveness). 3) Quando chi possiede il lock termina e non ci sono thread che hanno priorità più alta rispetto al nostro thread che sta aspettando di avere il lock allora questo nostro thread che sta aspettando, ottiene il lock entro un certo tempo finito. Ora se noi disponessimo di un meccanismo di questo tipo, riscrivere il codice del troppo latte diventa una cosa molto semplice (37.37)



perché scriverei: lock acquired, se non c'è il latte vai a comprarlo e quando è terminato lock release. La nostra sezione critica è che se non c'è il latte lo vado a comprare. Questo schema è standard, se ci dovesse capitare un esercizio simile si inizia sempre con una lock acquired e si termina con una lock release intorno alle sezioni critiche. L'implementazione di queste lock richiederà un supporto dell'hardware e del sistema operativo quindi in realtà stiamo spostando tutta la complessità della sincronizzazione all'interno del lock, quindi non la facciamo sparire ma la spostiamo solo in un altro contesto dove in realtà diventa più facile da gestire perché possiamo contare su un aiuto da parte dell'hardware. Le lock sono utilizzatissime e ci sono diversi linguaggi che le forniscono in maniera nativa. Noi anche le utilizziamo quando facciamo le chiamate di sistema o utilizziamo le funzioni di libreria per svolgere dei compiti (ad esempio quando facciamo la malloc, è un pezzo di codice che deve essere eseguito in una sezione critica e quindi normalmente deve essere racchiusa tra una lock acquired e una lock release e per lo stesso motivo la free è racchiusa tra una lock acquired e una lock release perché si va ad accedere a strutture che conservano la configurazione della memoria e sono sezioni critiche). Quali sono le regole per usare correttamente le lock? Inizialmente la lock deve essere libera quindi se c'è un'inizializzazione, la lock deve essere inizializzata a "libero". Prima di accedere ad una struttura dati condivisa bisogna acquisire la lock. Per convenienza gli accessi alle strutture dati condivise si scrivono

all'interno di funzioni / procedure. Per cui in realtà quello che succede è che mettiamo il lock all'inizio della procedura e lo rilasciamo alla fine della procedura. Questo ad esempio è il modello usato da Java. Quando iniziamo ad accedere ai dati condivisi facciamo la lock, appena abbiamo terminato di usare i dati condivisi (la struttura dati condivisa) dobbiamo ricordarci di fare subito la release. Mantenere una lock quando non serve può penalizzare le prestazioni del sistema perché dei thread che devono svolgere del lavoro si trovano bloccati in attesa che si liberi la lock. Quindi la lock va usata ma il minimo indispensabile dove serve. Quando non ci serve più dobbiamo ricordarci di rilasciarla sempre la lock. Sebbene sia possibile, non è buona prassi lasciare un lock aperto in ereditarietà a qualcuno altro quindi per esempio il mio thread acquisisce una lock, acquisisce una struttura dati, poi so che c'è un altro thread che deve lavorare, non faccio la release ma faccio lavorare l'altro thread e poi lui farà la release. Questo tipo di schema è possibile ma non è consigliabile per tutta una serie di problemi dovuti alla gestione del codice, di correttezza etc… quindi questo tipo di pratiche non sono buone e soprattutto mai e poi mai accedere a strutture dati condivise senza fare la lock perché questo in 99 casi su 100 non comporta niente, non fa niente , possiamo eseguire 100 volte il nostro codice e va sempre bene, lo eseguiamo la 101esima volta ed esplode. Il problema che si verifica dal mancato uso delle lock è un problema delicatissimo legato agli instanti in cui un thread viene messo in esecuzione. Il più delle volte capita che i thread riescono a completare il lavoro su una struttura dati condivisa prima che qualcun altro cerchi di andarci sopra per cui l'assenza di lock è molto difficile da verificare. Non basta vedere come facciamo di solito mandare in esecuzione 100 volte il codice, con 100 input diversi e verificare che per questi 100 input gli output siano tutti corretti. Questo nella programmazione concorrente non è una garanzia di correttezza e non rende la programmazione concorrente. In un codice concorrente su un dato di input che abbiamo già verificato, può darsi che il codice su un dato input funzioni 100 volte e la 101esima volta non funzioni. Quindi rendersi conto di problemi di concorrenza è estremamente difficile e il debug di un codice concorrente va affrontato soprattutto capendolo e analizzando bene il codice più che testandolo. La concorrenza è usatissima in qualsiasi sistema di controllo, se il sistema di controllo controlla qualcosa che può mettere in pericolo la vita umana, scrivere codice che si sincronizza male può causare danni irreparabili

Sistemi operativi 16/03/17

Nell'ultima lezione abbiamo visto i problemi che possono sorgere nel momento nel quale più thread cooperano in un contesto ad ambiente globale, hanno bisogno di sincronizzarsi all'accesso, all'utilizzo delle risorse condivise, alle strutture dati condivise, se non lo fanno, questo accesso può causare facilmente interferenze. Abbiamo visto il meccanismo di Lock e che va utilizzato in questo modo: ogni qualvolta dobbiamo accedere a una struttura dati condivisa, acquisiamo prima il lock e poi quando abbiamo terminato rilasciamo il Lock. Questo meccanismo ci garantisce il fatto che nell'accesso alla struttura dati condivisa il nostro thread è l'unico a lavorare, tutti gli thread che devo prima acquisire il Lock si trovano bloccati (se il lock è già stato acquisito). Ci sono delle regole per usare questi Lock, inizialmente i Lock sono sempre liberi, bisogni sempre utilizzare una lock_acquire () prima di accedere ad una struttura dati condivisa all'inizio della procedura, bisogna sempre rilasciare, quando abbiamo finito di usare quella struttura dati condivisa, alla fine della procedura e non è una buona prassi, sebbene sia potenzialmente possibile, però è fortemente sconsigliato, quello di lasciare il Lock per qualche altro thread, quindi lasciare la struttura dati vincolata per qualche altro thread che poi a sua volta dovrà fare la lock_release (). La regola buona è che: chi fa il Lock deve anche poi fare la lock_release () quando ha terminato di utilizzare la struttura. Se si tenta di accedere ad una struttura senza il Lock questo è un potenziale pericolo, quindi questo va evitato. Immaginiamo di avere questo codice

```
if (p == NULL) {
  lock_acquire(lock);
  if (p == NULL) {
    p = newP();
  }
  release_lock(lock);
}
use p->field1
```

```
newP() {
  p = malloc(sizeof(p));
  p->field1 = ...
  p->field2 = ...
  return p;
}
```

Immaginiamo che questo codice venga eseguito da più thread concorrentemente. La domanda è: funziona? Idealmente quello che ci aspettiamo che faccia è: il primo thread che arriva alloca la variabile p e la inizializza e poi la può utilizzare, se arriva un thread e trova la variabile p allocata e già inizializzata allora a quel punto la può usare direttamente e non ha bisogno di allocarla. Se immaginiamo di avere due thread che eseguono questo codice concorrentemente, va tutto bene? Ci sono problemi? Mi aspetto che quando un thread arriva ad utilizzare la variabile p, questa sia stata allocata ed inizializzata. Avviene sempre ciò? Abbiamo due thread A e B, che iniziano ad eseguire questo codice, potenzialmente concorrentemente. Vogliamo che la variabile p sia allocata una sola volta e sia inizializzata una sola volta. Quindi o la alloca e inizializza A o la alloca e inizializza B. Comunque sia uno dei due certamente la alloca e la inizializza perché se nessuno dei due la fa, si rischia di arrivare al punto di utilizzare una variabile che non è allocata oppure che non è inizializzata correttamente. Se ho questi due thread A e B, sono certo che uno dei due certamente la allochi e la inizializzi prima che l'altro la utilizzi? Questo codice non funziona perché: immaginiamo di avere due thread A e B, il primo thread, il thread A, arriva per primo, vede che p è NULL, quindi entra nel ramo, acquisisce la Lock, p è ancora NULL quindi fa la newP (), fa la malloc, a questo punto il thread A viene deschedulato, quindi siamo in una situazione in cui il thread A ha allocato p ma non lo ha ancora inizializzato, viene deschedulato e passa in esecuzione il thread B. Il thread B stavolta scopre che p è diverso da NULL perché A lo ha già allocato, disgraziatamente questo test p == NULL è fatto fuori dalla Lock, quindi può essere svolto dal thread B anche se il thread A ancora detiene la Lock. Il thread B scopre che p è diverso da NULL, esce dal ramo, va ad utilizzare la variabile p, però la variabile p è allocata ma non inizializzata, quindi presumibilmente trarrà delle conclusioni sbagliate, non riuscirà a completare il lavoro correttamente. Dove sta il problema in questo codice? Il problema sta nel fatto che non possiamo testare il valore di p fuori dalla Lock. La nostra struttura dati condivisa è p, qualsiasi accesso a p, incluso quel test p

== NULL deve essere fatto dopo aver acquisito la Lock, non può essere fatto prima. Concettualmente è semplice però il problema può diventare subdolo in un codice molto articolato se non prestiamo bene attenzione. Vediamo un caso più interessante, immaginiamo di avere due thread che cooperano scambiandosi informazioni su un buffer. (la slide che segue non è aggiornata ed ha piccolissime modifiche non significative)

Lock example: Bounded Buffer

```
tryget() {
  item = NULL;
  lock.acquire();
  if (nelem>0) {
      item = buf[front];
      front = (front++)%size;
      nelem --;
  }
  lock.release();
  return item;
}
```

```
tryput(item) {
  lock.acquire();
  if (nelem < size) {
      buf[last] = item;
      last = (last ++)%size;
      nelem ++;
  }
  lock.release();
}
```

Initially: nelem = front = last = 0; lock = FREE;
size is buffer capacity

Abbiamo due thread che cooperano tramite un buffer condiviso che si chiama buf, un thread può fare la get, quindi può leggere da questo buffer, un altro thread invece può fare la put, quindi depositare in questo buffer. Come potrebbe funzionare il codice? Il thread che preleva, inizialmente imposta l'elemento item da prelevare a NULL, dopodiché deve accedere al buffer, quindi acquisisce la Lock, dopodiché verifica se ci sono elementi nel buffer, quindi se nelem > 0, allora preleva un elemento item dal buffer, aggiorna la struttura dati del buffer per rimuovere l'elemento, quindi decrementa anche nelem per indicare che c'è un elemento in meno nel buffer, fatto questo rilascia il Lock e restituisce l'oggetto prelevato. Viceversa chi fa la put, quindi chi deposita, passa in ingresso alla put l'elemento da inserire, quindi acquisisce la mutua esclusione, se il buffer non è pieno, il buffer ha una capacità limitata, quindi se il buffer è già pieno non possiamo scrivere, rischiamo di sovrascrivere un elemento esistente e questo non va bene. Quindi se il buffer non è pieno allora possiamo depositare nella posizione last l'elemento, aggiorniamo il buffer, incrementiamo il numero di elementi ed infine rilasciamo il Lock. In questo modo, usando il Lock, garantiamo che, se viene fatto un accesso alla struttura dati, questo accesso la lascia consistente, quindi non è possibile che un thread legga un dato che non c'è, non è possibile che l'operazione di lettura/scrittura in un buffer avvengano concorrentemente per cui si leggano un valore scorretto o che si lasci il buffer in uno stato inconsistente come abbiamo visto nella lezione scorsa nel caso dello stack. Che problema c'è con questo codice? Il problema non è ad esempio il return item fuori dal Lock, perché è corretto, item non è una variabile condivisa, item è una variabile locale alla get quindi in questo momento appartiene soltanto al thread che sta eseguendo la get, non è una variabile che può essere letta e modificata da chi fa la put, son due variabili che certamente hanno lo stesso nome, ma il loro scope è locale alla funzione, la vera variabile condivisa è tutto il buffer con le variabili di riferimento, quindi buf, front, size, numeroelementi e last, è questa la struttura condivisa, item è un valore locale. Se una variabile è di proprietà di un singolo thread, come nel caso di item, quel thread lo può prendere e modificare a piacimento senza il bisogno di farci sopra nessuna Lock, come del resto abbiamo sempre fatto quando scrivevamo codice in c, questo non è un problema. Il problema si pone soltanto per le variabili che sono condivise, non per quelle private. Quindi item qui è una variabile privata di questo particolare thread che in

questo momento sta eseguendo la get, quindi quel thread la può utilizzare liberamente fuori da qualsiasi Lock. Tornando al resto del codice. Immaginiamo di essere un programmatore che utilizza questa libreria, quindi noi ci affidiamo a questa libreria per far comunicare i nostri thread scambiando informazioni tramite questo buffer. Che cosa succede se vogliamo fare una get e il buffer è vuoto? La get restituirebbe NULL.

Il problema è che se il buffer è vuoto, non mi deve restituire NULL, potrebbe anche farlo, ma in tal caso costringerebbe il processore a fare del lavoro inutile, rallentando gli altri thread, non è efficiente. Se non c'è niente nel buffer è più comodo che la get aspetti. Esattamente come nel caso della Lock, il comportamento che noi ci aspettiamo dalla Lock è questo: se la Lock è occupata, non vado avanti, piuttosto aspetto, aspetto fintanto che non si libera. Vorrei avere la stessa proprietà dalla get. Però questa cosa non la posso fare se l'unico meccanismo che ho a disposizione sono le Lock. Le Lock mi garantiscono la mutua esclusione, ma non mi permettono di sincronizzare due thread per far fare prima l'uno una cosa e poi all'altro, non hanno questa capacità. Le Lock mi servono per dire: se io sto facendo una cosa, tu questa cosa non la fare ed aspetti che io abbia terminato, ma non permettono di dire a due thread "questo lo faccio io e poi dopo quando ho finito questo lo fai tu". Tutto ciò per dire che non ci basta solo la Lock per gestire correttamente l'interazione tra thread concorrenti, ci serve un qualcosa che ci possa far dire "se non c'è nessun elemento, aspetta che arrivi un elemento, quando l'elemento arriva allora estrailo". Stessa cosa per la put, se il buffer è pieno aspetta fintanto che non si svuota almeno un elemento e soltanto a quel punto dammi la possibilità di inserire. Ci serve un altro meccanismo, non può essere la Lock, quest'altro meccanismo esiste e si chiamano variabili di condizione. Queste variabili di condizione permettono ad un thread di aspettare, hanno diversi vincoli nel modo nel quale devono essere utilizzate perché sono hanno un ruolo ben preciso. Il primo vincolo è che le variabili di condizione possono essere utilizzate soltanto all'interno di un blocco lock_acquire / lock_release, non possono essere usate liberamente, non possiamo sincronizzarci al di fuori di una sezione critica. Quindi prima acquisiamo una Lock, dopodiché all'interno di questa lock possiamo utilizzare le variabili di condizione per sospenderci o per riattivarci e poi facciamo la lock_release. Le operazioni che possiamo fare sulle variabili di condizione sono due: una operazione di Wait e una operazione di Signal. La Wait sospende, la Signal riattiva. Si chiamano variabili di condizione ma in realtà non c'è nessuna condizione sull'attesa, quando facciamo una Wait su una variabile di condizione il thread si sospende incondizionatamente. Quindi fare la Wait significa sospendere il thread. La Wait fa due cose, in un colpo solo, in modo atomico, rilascia il Lock e sospende il thread. Quindi noi acquisiamo la Lock, ci rendiamo conto che il buffer è vuoto e quindi non possiamo prelevare, facciamo la Wait e la Wait sospende il thread e contemporaneamente rilascia il Lock. La Signal serve per svegliare un thread che attende, che ha fatto precedentemente una Wait, la Broadcast serve per svegliare tutti i thread che hanno fatto la Wait (se ce ne sono). A questo punto vediamo l'esempio di cooperazione fra thread usando un buffer condiviso con una sincronizzazione sul buffer pieno o vuoto. Questo problema è un problema classico, si chiama Problema del Produttore Consumatore e ha questa definizione: abbiamo uno o più produttori che depositano messaggi in un buffer condiviso e abbiamo uno o più consumatori che prelevano messaggi dal buffer condiviso. Il vincolo che dobbiamo rispettare è che ogni messaggio depositato può essere letto da un consumatore una e una sola volta. Quindi lo stesso messaggio non può esser letto due volte, non può neanche capitare che un messaggio non venga mai letto e vengano letti gli altri. Ogni messaggio depositato deve essere letto e una volta letto deve sparire dal buffer, non può essere riletto da un altro consumatore. Questo vincolo è importante perché altrimenti ricadiamo in un altro caso o in altri casi che sono nati che però prendono un'altra formulazione e un altro tipo di soluzione, poi vedremo tra gli esempi oltre al Produttore Consumatore anche il caso dei Vettori Scrittori che invece hanno vincoli differenti riguardo ai messaggi. Questo è il modo col quale funziona il Produttore Consumatore:

Example: Bounded Buffer

```
get() {
  lock.acquire();
  while (nelem == 0)
        empty.wait(lock);
  item = buf[front];
  front= (front++) % size;
  nelem – –;
  full.signal(lock);
  lock.release();
  return item;
}
```

```
put(item) {
  lock.acquire();
  while (nelem == size)
        full.wait(lock);
  buf[last] = item;
  last = (last ++) % size;
  nelem ++;
  empty.signal(lock);
  lock.release();
}
```

Initially: nelem = front = last = 0; size is buffer capacity
empty/full are condition variables

Nuovamente abbiamo una funzione di put e una funzione di get, la get server per prelevare messaggi dal buffer, la put serve per depositarli. Il buffer è il solito, buf, con le sue variabili d'appoggio front, size, numero_elementi e last.

Quindi l'insieme di queste variabili definisce lo stato del buffer. Abbiamo anche in questo caso una Lock per acquisire la mutua esclusione nell'accesso al buffer ed in più abbiamo due variabili di condizione che sono empty e full. La variabile di condizione empty viene utilizzata per la sospensione dei thread nel caso in cui il buffer sia vuoto. La variabile di condizione full viene utilizzata per sospendere i thread nel caso in cui il buffer sia pieno e quindi non si possa depositare. Com'è fatta questa soluzione: quando facciamo la get per prima cosa di acquisisce la mutua esclusione, dopodiché si controlla se il buffer è vuoto. Quindi fintanto che il numero di elementi è zero, quindi se il buffer è vuoto, allora si fa una Wait sulla variabile di condizione Empty e si passa come parametro a questa Wait il Lock sul quale abbiamo acquisito la mutua esclusione. Quindi che cosa succede se troviamo il buffer vuoto? Se il buffer è vuoto la Wait rilascia il Lock, quindi fa la release del Lock, e contemporaneamente blocca/sospende il thread che ha fatto la Wait. Questo thread che ha fatto la Wait potrà essere riattivato soltanto quando qualcuno farà la Signal. Quindi questa Signal sulla variabile di condizione empty può essere fatta soltanto dopo che si deposita qualcosa. Tizio chiede "perché mettiamo il While? Non potremmo mettere un If?" Concettualmente basta un If, in realtà come vedremo più avanti, ci sono due problemi con l'If. Il primo problema è che potremmo avere più thread che si sospendono, più thread potrebbero essere riattivati e non detto che tutti quanti possano poi riutilizzare. Può passare del tempo dal momento in cui un thread fa la Signal al momento in cui un thread viene effettivamente riattivato e messo in esecuzione quindi nel frattempo lo stato del buffer può essere ancora cambiato e poi c'è un altro motivo, che possono arrivare delle riattivazioni spurie, questo è un problema di sistema/implementazione, in alcuni sistemi è possibile che arrivino delle riattivazioni dei thread spurie non generate dalle Signal ma per altre cause. Quindi ci sono tanti buoni motivi per ritestare nuovamente la condizione, però di questi motivi ne riparleremo più avanti. È meglio mettere il while fin da subito. Il thread che fa la Wait su empty viene sospeso, resta in attesa, fintanto che qualcuno non deposita un messaggio. Quindi quando qualcuno deposita un messaggio, questo thread verrà riattivato, scoprirà che ci sono degli elementi, quindi uscirà dal while, a questo punto potrà andare a prelevare. Quando fa il prelievo è sicuro che c'è un elemento, altrimenti non sarebbe uscito dal while. Quando il thread viene riattivato, riaquisisce la Lock, quindi quando facciamo la Wait, rilasciamo la Lock, ci sospendiamo, alla riattivazione riacquisiamo la Lock e continuiamo. Siccome abbiamo riacquisito il Lock, se ci sono elementi a questo punto siamo sicuri che ci sono e resteranno fintanto che non rilasciamo la Lock, quindi certamente almeno quell'elemento lo andiamo a leggere. Letto questo elemento, tralasciando un attimo la full.signal, possiamo rilasciare il Lock e restituire l'oggetto. Quindi abbiamo implementato quello che volevamo, se non c'è niente aspettiamo, non

appena c'è qualcosa oppure se c'è qualcosa, preleviamo e restituiamo, quindi chi utilizza la get non si deve preoccupare del fatto che restituiamo un valore vuoto NULL, terminata la get certamente ci sarà un valore restituito. Per contro vediamo la put, perfettamente simmetrica, ma al contrario. La put deve acquisire la mutua esclusione, verificare che il buffer non sia pieno, quindi verificare che il numero di elementi non sia uguale a size, se per caso il buffer è pieno facciamo una Wait con parametro Lock sulla variabile di condizione full, quindi se il buffer è pieno il thread rilascia il Lock, si sospende, verrà riattivato quando qualcuno preleverà dal buffer, quindi si sarà liberata una posizione. Quindi quando il thread viene riattivato, riacquisisce il Lock, si assicura nuovamente nel while che effettivamente che una posizione libera ci sia e a questo punto una posizione libera certamente c'è per cui può andare a depositare senza problemi, senza andare a sovrascrivere niente. Quando ha terminato di scrivere il suo elemento, fa una empty.Signal, questo è il meccanismo di sveglia che ci serve perché se per caso un thread si è bloccato sulla empty.Wait perché non c'erano elementi da leggere, quando arriva un processo Produttore che deposita, stavolta c'è almeno un elemento quindi fa la empty.Signal che ha l'effetto di riattivare il nostro thread sospeso su quella Wait e quindi mi permette di prelevare.

Se questo thread lo faccio eseguire da n Produttori e da n Consumatori, può darsi che più thread Produttori siano arrivati e abbiano trovato il buffer vuoto e quindi ci siano un certo numero di thread Consumatori tutti bloccati su questa Wait. Quando poi arriva un produttore, deposita il messaggio e fa la empty.Signal, quale di questi thread viene riattivato? Sappiamo che ne viene riattivato certamente uno solo perché la Signal riattiva un solo thread tra quelli in attesa, in realtà chi viene riattivato dipende dall'implementazione, però normalmente vengono riattivati in ordine FIFO, quindi il primo arrivato è il primo riattivato, è un aspetto legato all'implementazione e non c'è motivo di implementarlo in maniera differente. D'altra parte è vero che in realtà le attivazioni possono arrivare spurie per cui qualche thread potrebbe essere riattivato prima "scorrettamente", cioè potrebbe passare avanti senza rispettare l'ordine, quindi non abbiamo la certezza matematica che i thread poi verranno serviti in ordine FIFO.

Come vediamo il Produttore deve fare la Signal su empty per svegliare i Consumatori che si sono addormentati ma alla stessa maniera i Consumatori devono fare la Signal su full per svegliare gli eventuali Produttori che si sono addormentati perché, se i Produttori sono sospesi sulla Wait vuol dire che il buffer era pieno, quando arriva un Consumatore che preleva, lì darà spazio dal buffer e quindi per ogni elemento prelevato bisogna svegliare un produttore. È importante che quando un thread viene riattivato riacquisisca la Lock e questo è garantito, è nella semantica del meccanismo, delle variabili di condizione. Supponiamo di avere un Consumatore, questo Consumatore acquisisce il Lock, scopre che il buffer è vuoto, fa la Wait, per fare la Wait rilascia il Lock e si sospende. E questo è il thread A. Arriva il thread B, il thread B deposita un messaggio, quindi stavolta c'è un messaggio nel buffer, fa la Signal su empty, la Signal riattiva questo thread, il thread A, quindi il thread A riattivato deve fare di nuovo la lock_acquire () sostanzialmente, la fa implicitamente all'interno della Wait. Siccome però in questo momento la Lock ce l'ha il thread B che non l'ha ancora lasciata, il thread A si blocca di nuovo sulla Lock fintanto che il thread A non la libera. È una Lock a tutti gli effetti, quindi non può mai capitare che per effetto della riattivazione, quando va a riacquisire il Lock mi trova due thread che hanno due Lock contemporaneamente, questo non esiste proprio. C'è un altro aspetto riguardante l'ordine di attivazione: supponiamo di avere due thread Consumatori A e B e un thread produttore C. Allora immaginiamo che inizialmente il buffer sia vuoto, arriva il thread A, fa la lock_acquire (), scopre che il buffer è vuoto, fa la empty.Wait su Lock, quindi rilascia il Lock e A si mette in attesa. Dopodiché arriva il Produttore C, fa la Lock (è libera perché è stata rilasciata da A), trova che nel buffer c'è spazio quindi deposita, a questo punto fa la Signal su Lock e va a riattivare A. Che cosa succede quando ha riattivato A? A è stato riattivato quindi è stato messo nella lista pronti, non è affatto detto che A venga subito messo in esecuzione dal sistema operativo quindi A è pronto, quando verrà messo in esecuzione ripartirà dall'interno della Wait e riacquisirà il Lock, però in questo istante abbiamo C che è in esecuzione ed ha appena terminato la Signal. Ora può succedere di tutto, può capitare che C rilasci il Lock, venga deschedulato, venga messo in esecuzione B, non A, B acquisisce la mutua esclusione, scopre che il

buffer non è vuoto perché c'è il valore appena scritto da C, quindi B va a prelevare il dato dal buffer, fa la lock_release (), a questo punto B viene deschedulato, passa in esecuzione A, A riparte dalla Wait, fa la lock_acquire () e però non c'è più un elemento nel buffer perché nel frattempo B le è passato davanti. Pertanto è costretto a ricontrollare il numero di elementi, scopre che il buffer è vuoto e quindi si risospende. Anche se la riattivazione con la Signal avviene in ordine FIFO, non abbiamo nessuna garanzia che il servizio, l'utilizzo del buffer, avvenga in modo FIFO. Abbiamo guardato questo codice Produttore Consumatore facendo dei casi d'uso, quindi ipotizzando diverse situazioni e cercando di convincerci che in tutte queste situazioni questo codice si comporta correttamente. In effetti di questo codice, come in generale diverse soluzioni standard di problemi classici, esistono delle dimostrazioni di correttezza e in particolare anche per il Produttore Consumatore. Queste dimostrazioni di correttezza si basano sull'analisi di precondizioni/postcondizioni. Per esempio se noi possiamo verificare che alla Lock e alla release, lo stato del buffer sia sempre consistente, questo è un elemento che ci permette poi di dimostrarne la correttezza. È importante capire che se utilizziamo degli schemi di soluzione standard, generalmente ci semplifichiamo la vita e ci garantiamo di evitare tanti problemi. Il Produttore Consumatore è un esempio importante, esiste ovunque: se facciamo un webserver, i webserver sono fatti da un thread che riceve le richieste e le smista a dei thread serventi. Il passaggio delle informazioni delle richieste dal thread che smista ai thread serventi è un problema del Produttore Consumatore. Il problema della comunicazione tra processi utilizzando i meccanismi dei Socket/delle Pipe è un problema del Produttore Consumatore. La gestione dei messaggi che arrivano dalla rete e lo smistamento verso i processi di nuovo è un problema del Produttore Consumatore. È un problema estremamente diffuso, quindi è fondamentale capire questo schema. Lo schema è semplice: si parte da una lock_acquire (), si finisce con una lock_release(), le Wait e le Signal sono incrociate, se da un lato si fa la Wait, dall'altra parte si fa la Signal e viceversa. Riassumendo sulle variabili di condizione, quando si fanno le operazioni di Wait, Signal, Broadcast, bisogna sempre essere all'interno di un Lock perché le variabili di condizione sono degli strumenti per sincronizzarsi su strutture condivise, non su altro, quindi questo è un vincolo importante. Il secondo aspetto importante è che le variabili di condizione non hanno memoria. Che cosa succede qualcuno fa una Signal su una variabile e poi successivamente qualcuno dopo fa una Wait? La Signal si perde, la Signal fatta prima della Wait non lascia traccia, non modifica lo stato, per cui se qualcuno fa la Wait dopo, si sospende. Il senso della Signal è: se c'è qualcuno in attesa lo risveglio, altrimenti non faccio nulla. Quindi non ha stato. Se faccio la Signal quando nessuno aspetta non c'è nessuna operazione, se invece viene fatta la Wait prima della Signal, quando si fa la Signal il thread in attesa viene riattivato. Altra cosa importante da tenere a mente è che la Wait in modo atomico rilascia il Lock e sospende il thread. Perché deve essere fatto in modo atomico? Che cosa succede se prima rilascio il Lock e poi sospendo oppure se prima sospendo e poi rilascio il Lock? Ipotizziamo che la Wait prima sospenda e poi rilasci il Lock. Ci sarebbe un deadlock. Se facciamo la Wait, ci sospendiamo, a quel punto non possiamo più rilasciare il Lock, quindi restiamo sospesi continuando a detenere il Lock, nessun altro può acquisire il Lock quindi nessun altro può fare la Signal e nessun altro ci può riattivare, quindi in realtà restiamo bloccati per sempre e tutti gli altri thread che fanno la lock_acquire () restano pure bloccati. Non possiamo quindi sospenderci e poi rilasciare il Lock. Viceversa, supponiamo di rilasciare prima il Lock e poi di sospenderci. Il problema è che rischiamo di perdere la Signal. Torniamo all'esempio precedente:

Example: Bounded Buffer

```
get() {
  lock.acquire();
  while (nelem == 0)
        empty.wait(lock);
  item = buf[front];
  front= (front++) % size;
  nelem – –;
  full.signal(lock);
  lock.release();
  return item;
}
```

```
put(item) {
  lock.acquire();
  while (nelem == size)
        full.wait(lock);
  buf[last] = item;
  last = (last ++) % size;
  nelem ++;
  empty.signal(lock);
  lock.release();
}
```

Initially: nelem = front = last = 0; size is buffer capacity
empty/full are condition variables

Immaginiamo di essere un thread Consumatore, arriviamo e troviamo il buffer vuoto, facciamo la Wait su empty. Quindi la Wait su empty rilascia il Lock , prima però di sospendere il thread, questo thread viene deschedulato quindi ancora non è sospeso, ha soltanto rilasciato il Lock, passa in esecuzione un Produttore, il Produttore deposita, fa la Signal, la Signal non ha effetto perché non c'è nessuno sospeso, dopodiché ritorna in esecuzione il mio Consumatore, il mio Consumatore completa la Wait e si sospende, però a questo punto la Signal è stata persa, quindi resto bloccato lì mentre invece avrei dovuto continuare. Il rischio è che se se separiamo e facciamo prima la lock_release () e poi la sospensione, rischiamo di perdere una Signal, se facciamo prima una sospensione poi una lock_release (), mandiamo il sistema in blocco. La Wait deve essere un'azione atomica. Riassunto: La Lock fa due operazioni: rilascia la mutua esclusione e sospende il thread, poi quando il thread è riattivato, riacquisisce la mutua esclusione. Il rilascio della mutua esclusione e la sospensione del thread devono essere fatte in un'azione atomica, in un'unica azione indivisibile. Supponiamo che non sia così, supponiamo che io prima sospenda il thread e poi rilasci la mutua esclusione, in questo caso il thread sospeso non può rilasciare la mutua esclusione perché è sospeso quindi non eseguirà mai l'istruzione dopo, quindi il Lock resta bloccato, quindi nessun altro thread può acquisire il Lock per fare la Signal, il mio thread resta bloccato per sempre e tutti i thread che fanno la Lock restano pure bloccati loro per sempre. Altro caso: prima faccio la lock_release () e poi faccio la sospensione, può capitare che io faccia la lock_release (), prima di sospendermi passa un altro thread in esecuzione che modifica la struttura dati e fa la Signal, siccome io però non sono ancora sospeso, quella Signal si perde dopodiché io mi sospendo e a quel punto resto bloccato perché poi la Signal che avrebbe dovuto svegliarmi, è sì stata generata, ma è stata persa. Se prima rilascio la mutua esclusione e poi faccio la sospensione, rischio di perdere le Signal. Quindi le due operazioni devono essere un'azione atomica, quindi non devono essere divisibili.

- When a thread is woken up from wait, it may not run immediately
  - Signal/broadcast put thread on ready list
  - When lock is released, anyone might acquire it
- Wait MUST be in a loop

```
while (needToWait())
   condition.Wait(lock);
```

- Simplifies implementation
  - Of condition variables and locks
  - Of code that uses condition variables and locks

Dobbiamo inserire le Wait all'interno del while perché non siamo sicuri di una cosa: quando li avremo attivati non sappiamo se la Signal è avvenuta poco prima o molto prima, può darsi che passi parecchio tempo dalla Signal al momento nel quale il thread riattivato ottiene il processore e nel frattempo la struttura dati può essere cambiata. Quindi siamo obbligati ad utilizzare un loop di questo tipo per le Wait. Ci sono anche altri motivi legati al fatto che potrebbe essere una Broadcast anziché una Signal, quindi potrebbe aver svegliato più thread, oppure problemi legati alla gestione del sistema operativo dei meccanismi di riattivazione che in certi casi possono prevedere dei segnali spuri. Quindi per tutti questi motivi la Wait deve stare in un loop. Questo semplifica anche l'implementazione delle variabili di condizione e dei Lock e in qualche maniera anche del codice. Questa cosa è talmente importante che ad esempio nel knowledge java c'è proprio scritta una nota per questo punto: quando stiamo aspettando su una condizione una riattivazione spuria potrebbe capitare, in generale come concessione alla semantica della piattaforma sottostante quindi del sistema operativo. Dal punto di vista pratico questo ha un piccolo impatto in molte applicazioni in quanto la variabile di condizione è sempre attesa all'interno di un ciclo che testa che la condizione che deve portarla alla riattivazione sia di nuovo valida, quindi il fatto di utilizzare il while risolve anche il problema delle riattivazioni spurie.

Example: Bounded Buffer

```
get() {
  lock.acquire();
  while (nelem == 0)
        empty.wait(lock);
  item = buf[front];
  front= (front++) % size;
  nelem – –;
  full.signal(lock);
  lock.release();
  return item;
}
```

```
put(item) {
  lock.acquire();
  while (nelem == size)
        full.wait(lock);
  buf[last] = item;
  last = (last ++) % size;
  nelem ++;
  empty.signal(lock);
  lock.release();
}
```

Initially: nelem = front = last = 0; size is buffer capacity
empty/full are condition variables

Mettiamo un If al posto del while, il primo problema è che se noi abbiamo lì l'If, può darsi che la piattaforma sottostante, quindi il sistema operativo più il supporto del linguaggio, possa riattivare il thread anche se in realtà non è stata fatta una Signal dal Produttore, quindi potrebbe darsi che il thread venga riattivato da un segnale spurio, non legato a quella Signal. Se ho messo l'If, non ritesto la condizione e quindi vado a prelevare il messaggio che però non c'è perché io non dovevo essere riattivato. Il secondo

problema è che: supponiamo che qualcuno deposita un messaggio e fa la Signal, io vengo messo nella coda Pronti, ma prima di tornare in esecuzione ci passa del tempo. Quindi in questo momento c'ho un messaggio nel buffer ma non lo posso prendere finché non torno in esecuzione. Se nel frattempo passa in esecuzione un altro thread, quell'altro thread può trovare la Lock libera? Si, perché io riacquisirò la Lock appena tornerò in esecuzione, non prima. Quindi quell'altro thread potrebbe acquisire la Lock, prelevare il valore, soffiarmelo letteralmente da sotto il naso, quando io poi torno in esecuzione, siccome ho fatto la if, non testo quella condizione, vado a prelevare, ma prelevo un valore che non c'è. Quindi il while è fondamentale.

- Identify objects or data structures that can be accessed by multiple threads concurrently
  - In Pintos kernel, everything!
- Add locks to object/module
  - Grab lock on start to every method/procedure
  - Release lock on finish
- If need to wait
  - while(needToWait()) condition.Wait(lock);
  - Do not assume when you wake up, signaller just ran
- If do something that might wake someone up
  - Signal or Broadcast
- Always leave shared state variables in a consistent state
  - When lock is released, or when waiting

In generale com'è che si procede per utilizzare questi meccanismi: prima di tutto individuiamo nel nostro codice gli oggetti condivisi, dopodiché individuiamo le sezioni critiche, quindi tutte le procedure, le funzioni che manipolano questi dati condivisi. Queste procedure, queste funzioni, questi spezzoni di codice devono sempre iniziare con un lock_acquire () e con lock_release (). Se per caso c'è un'operazione che non può essere fatta, quando siamo in sezione critica, quindi quando deteniamo il Lock, perché questa operazione ha bisogno di una sincronizzazione con un altro thread, allora dobbiamo fare una Wait e questa Wait deve stare sempre all'interno di un ciclo. Quando facciamo la Wait, teniamo presente che gli stiamo permettendo a qualcun altro di utilizzare la struttura condivisa, quindi se abbiamo fatto delle modifiche dobbiamo lasciare prima della Wait, la struttura in uno stato consistente altrimenti chi arriva dopo di noi la trova inconsistente. Se facciamo qualcosa che può risvegliare un altro thread dobbiamo fare la Signal. Soprattutto dobbiamo ricordarci che quando rilasciamo il Lock o quando aspettiamo, in entrambi i casi, la struttura va lasciata in ordine consistente, non lasciamo che un altro thread debba aggiustare la struttura per conto nostro perché diventa davvero complicato da gestire.

- Mesa (in textbook, Hansen)
  - Signal puts waiter on ready list
  - Signaller keeps lock and processor
- Hoare
  - Signal gives processor and lock to waiter
  - When waiter finishes, processor/lock given back to signaller
  - Nested signals possible!

A questo punto c'è un aspetto che riguarda l'implementazione della Signal, un aspetto legato alla semantica della Signal, perché ci sono due modi di implementarla: un modo è la semantica Mesa che funziona in questa maniera: quando faccio una Signal e riattivo un altro thread, quel thread viene messo in coda Pronti, quindi viene riattivato, ma io continuo a tenere il Lock. Quando poi cederò il Lock, quando quel thread verrà riattivato, potrà acquisire il Lock e a questo punto svolgere il suo lavoro. Quindi il thread che fa la Signal continua a mantenere il Lock e va avanti. L'altra possibilità invece è la semantica di Hoare che lascia

in eredità il Lock al thread riattivato. Quindi io faccio la Signal, un thread viene riattivato, questo thread prende in eredità il Lock da me e quindi lui può lavorare. Quando lui ha terminato farà la lock_release (), ritornerà prima o poi il controllo a me che potrò continuare il mio codice, la mia sezione critica, e poi farla io la lock_release () a mia volta. Quindi in questo caso diventa più complesso da gestire perché si possono annidare i segnali. Dal nostro punto di vista cosa cambia? Per molte soluzioni non cambia niente, tutte le volte che facciamo una Signal e subito dopo una lock_release (), che sia semantica Mesa o che sia semantica Hoare non cambia niente. Se invece il nostro codice è più articolato allora avere un'implementazione con semantica Mesa o semantica Hoare potrebbe cambiare la questione. In particolare cosa cambia?

```
get() {                          put(item) {
 lock.acquire();                  lock.acquire();
 if (front == last)               if ((last – front) == size)
    empty.wait(lock);              full.wait(lock);
 item = buf[front % size];        buf[last % size] = item;
 front++;                         last++;
 full.signal(lock);               empty.signal(lock);
 lock.release();                 // CAREFUL: someone else ran
 return item;                     lock.release();
}                                }
```

Initially: front = last = 0; size is buffer capacity
empty/full are condition variables

Prendiamo l'esempio del buffer condiviso, del Produttore Consumatore: se la Signal, rispetta una semantica Hoare, succede che tra questa Signal e questa release, qualche altro thread potrebbe eseguire del codice in sezione critica, quindi se tra una Signal e una release continuo a manipolare la struttura dati, devo stare attento perché dopo aver fatto la Signal la struttura dati potrebbe essere stata modificata da qualcun altro e poi il controllo tornerà a me e io farò delle altre modifiche, quindi bisogna che io stia attento perché le mie modifiche vengono sì fatte in mutua esclusione, ma il problema è che questa prima parte e questa seconda parte non sono più atomiche, in mezzo è stata svolta dell'altra attività sulla struttura. E la stessa cosa può capitare qui se la semantica è Hoare, tra questa Signal e questa release. Nel nostro caso, per il Produttore Consumatore, nel nostro codice tra la signal e la release non c'è niente, quindi non è un problema, sia Mesa che Hoare vanno benissimo, il problema si pone se tra la Signal e la release continuiamo a svolgere dell'altro lavoro sulla struttura dati. Con semantica Mesa non c'è problema perché possiamo assumere che la struttura non sia stata modificata da nessun altro, quindi che abbia lo stesso stato lasciato prima della Signal. Se invece abbiamo la semantica Hoare può darsi che lo stato della struttura sia stato modificato dopo la Signal.

- Create a condition variable for every waiter
- Queue condition variables (in FIFO order)
- Signal picks the front of the queue to wake up
- CAREFUL if spurious wakeups!

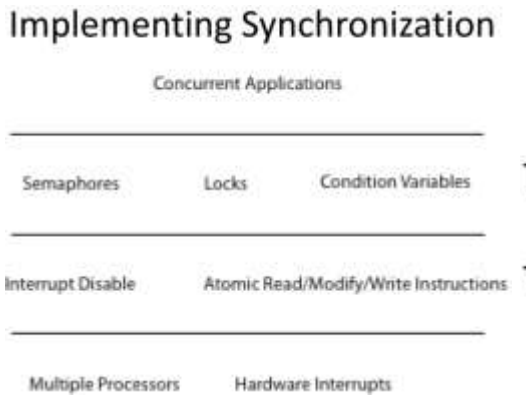
- Easily extends to case where queue is LIFO, priority, priority donation, …
  - With Hoare semantics, not as easy

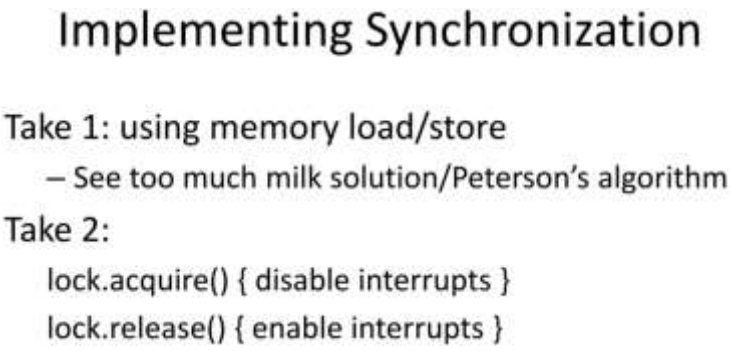
Abbiam visto che il meccanismo di Signal, anche se implementato in ordine FIFO, non garantisce in realtà che i thread vengano riattivati in modo FIFO quindi può darsi che abbiamo messo su un meccanismo, ipotizzando che i thread vengano riattivati in ordine FIFO per la gestione di una certa struttura, ma in realtà

per effetto dello scheduler del sistema operativo, per effetto di tutta un'altra serie di questioni, in realtà i thread vengono riordinati secondo un ordine che non ci aspettiamo. In generale se usiamo la Wait e la Signal non possiamo aspettarci nessun ordine particolare. Però ci sono delle applicazioni nelle quali invece è importante controllare l'ordine di riattivazione. Se vogliamo riattivare in ordine FIFO dobbiamo garantirlo, non possiamo farlo usando soltanto una Signal. È possibile farlo ma dobbiamo lavorare un po' sul codice. Quindi possiamo imporre l'ordine FIFO esplicitamente nel codice. Questa cosa viene particolarmente facile nel caso in cui la semantica sia MESA, nel caso in cui la semantica sia Hoare è appena più complicato, comunque noi lo vediamo per la semantica MESA e vediamo direttamente come funziona la soluzione.

```
get() {                          item = buf[front % size]
 lock.acquire();                 front++;
 if (front == last) {            if (!nextPut.empty())
    self = new Condition;           nextPut.first()->signal(lock);
    nextGet.Append(self);        lock.release();
    while (front == last)        return item;
      self.wait(lock);           }
    nextGet.Remove(self);
    delete self;
 }
Initially: front = last = 0; size is buffer capacity
nextGet, nextPut are queues of Condition Variables
```

Vediamo la get, la put è equivalente. L'idea di questo codice è: quando faccio la get e mi rendo conto che il buffer è vuoto, mi sospendo, ma non mi sospendo sull'unica variabile di condizione insieme a tutti gli altri perché se mi sospendessi in un'unica variabile di condizione insieme a tutti gli altri, quando viene fatta la Signal, devo assumere che ne venga pescato uno a caso. Quindi il trucco è quello di utilizzare una variabile di condizione per ogni thread che si va a sospendere. Quindi se li devo sospendere cosa faccio: creo una nuova variabile di condizione che è mia, privata, la appendo a una coda di variabili di condizione, alla coda che contiene le variabili di condizione di tutti i thread sospesi, dopodiché faccio la Wait sulla mia variabile di condizione. Se chi fa la Signal, fa la Signal sulla mia variabile di condizione, vengo riattivato soltanto io, non un thread a caso. Quindi se chi fa la Signal si preoccupa di estrarre i thread dalla coda nextGet secondo un certo ordine che può essere FIFO/LIFO/a priorità/etc, la riattivazione dei thread avverrà secondo l'ordine stabilito (FIFO/LIFO/etc). Una volta che son stato riattivato, a quel punto rimuovo la mia variabile di condizione dalla lista dei thread in attesa dove la dealloco e a questo punto posso andare a prelevare. Quando ho finito di prelevare, ho liberato potenzialmente una posizione dal buffer, quindi dovrei fare una Signal, ma nuovamente, i thread che stanno facendo la Put, se sono sospesi, si sono sospesi con uno schema simile, quindi ogni thread che ha fatto la Put si è sospeso, ha creato una sua variabile di condizione e si è sospeso sulla propria. Quindi non vado a fare una semplice Signal ma in realtà estraggo una variabile di condizione dalla lista dei thread sospesi sulla Put secondo un certo criterio, per esempio FIFO quindi estraggo il primo, e faccio la Signal. Questo ha effetto di svegliare solo quel thread, non gli altri. E poi a quel punto faccio la lock_release () e via discorrendo. Quindi posso forzare un ordine di riattivazione ricorrendo a delle variabili di condizione private, ogni thread ha la sua variabile di condizione, quando i thread si sospendono depositano le variabili di condizione in una struttura che ha un suo ordine, una sua logica e di conseguenza quando vado a fare la Signal estraggo una variabile di condizione da questa struttura secondo l'ordine stabilito e faccio la Signal soltanto di conseguenza per quel thread su quella variabile di condizione che ho estratto, questo mi garantisce che i thread vengano serviti secondo un ordine predefinito, in questo caso FIFO. Detto questo, vediamo questi meccanismi come sono relizzati.





Abbiamo visto come si usano, qual è la loro semantica quindi cosa ci aspettiamo da loro, ora dobbiamo vedere come si realizzano materialmente, quindi siamo in questa situazione: al di sotto abbiamo l'hardware che ci offre meccanismi di interruzione ed eventualmente anche capacità di calcolo parallelo col multiprocessore (di questo ne dovremo tener conto), al di sopra ci sono le applicazioni concorrenti quindi per esempio il nostro Produttore Consumatore, altre applicazioni concorrenti, in mezzo ci sono i meccanismi realizzati dall'hardware (che sono la disabilitazione delle interruzioni oppure istruzioni speciali per gestire queste situazioni e meccanismi realizzati dal sistema operativo). A noi interessa vedere quali sono i meccanismi realizzati dal sistema operativo che fanno leva sui meccanismi di base offerti dall'hardware. Quindi materialmente come possiamo implementare i meccanismi di Lock: una possibilità è usare algoritmi solitamente software simili a quelli che abbiam visto per l'esempio del latte, questi hanno tutte le problematiche che abbiamo discusso in precedenza, non garantiscono dalle ottimizzazioni del compilatore, dall'ordinamento delle istruzioni da parte del processore, sono molto macchinose/complicate, difficili da gestire su codice complesso e con tanti thread e sono pure inefficienti perché prevedono attesa attiva, quindi queste son da scartare. Un'altra possibilità che ho è quella invece di implementare la Lock disabilitando le interruzioni, quindi se faccio la Lock disabilito le interruzioni, se faccio la release abilito le interruzioni. Funziona? Mi garantisce realmente? Se funziona, ci sono controindicazioni? Se non funziona perché non va bene? Se implemento la lock_acquire () con disabled interrupts e la lock_release () con disabled interrupts, questo vuol dire che se torno al codice della get le interruzioni sono disabilitate alla lock_acquire (), vengono riabilitate alla lock_release (), durante tutta l'esecuzione di questo codice sono disabilitate (se implemento la lock_acquire () e la lock_release () disabilitando le interruzioni). Viene chiesto dal collega se ci sono dei problemi nel caso in cui si utilizzino delle Lock annidate. In realtà no perché la disabilitazione delle interruzioni non prevede annidamento, se le disabilita nessun altro può entrare. Usare questo meccanismo disabilitando le interruzioni vuol dire che disabilitiamo tutte le interruzioni, quindi se faccio la lock_acquire () disabilito le interruzioni, lo scheduler è bloccato, non può passare in esecuzione un altro thread anche per fare altre cose perché le interruzioni sono disabilitate. Un altro collega chiede: ma se utilizzo la disabilitazione delle interruzioni, quindi ho fatto una lock_acquire (), ho disabilitato le interruzioni, poi faccio una printf che prevede una chiamata di sistema, questo lo posso fare? No, non lo può fare, perché se fa una chiamata di sistema con le interruzioni disabilitate questa viene ignorata. Questa è una buona controindicazione però è anche vero potrei avere tante situazioni dove non devo fare chiamate di sistema in una sezione critica. Un altro collega chiede: se disabilitiamo le interruzioni, non entriamo in modalità supervisore? In realtà non è proprio così. Il punto è questo: la disabilitazione delle interruzioni non è permessa in modo utente, è permessa soltanto in modo supervisore quindi in teoria dovrei avere una chiamata di sistema per fare la Lock che fa la chiamata di sistema, disabilita le interruzioni e deve trovare il verso di lasciarle disabilitate anche dopo, però in realtà tutto questo crea un grosso problema: se fossi un programmatore e avessi a disposizione nel mio codice utente una Lock fatta così, potrei chiamare una lock_acquire () come prima riga del main, potrei chiamare una lock_release () come ultima riga del main e il mio codice va come una scheggia perché nel frattempo nessun altro viene messo in esecuzione. Questi meccanismi di lock_acquire () e lock_release () di variabili di condizione non servono

soltanto al nucleo per gestire le sue attività concorrenti ma servono anche tantissimo ai programmatori che possono scrivere dei programmi che definiscono più thread. Quindi i thread che scriveremo lavoreranno in stato utente e condivideranno memoria e avranno bisogno di questi meccanismi di lock_acquire () e lock_release () e di variabili di condizione per sincronizzarsi. Quindi in realtà ci servono dei meccanismi di lock_acquire () e lock_release () che posso offrire ai programmatori in stato utente, non soltanto dei meccanismi che posso usare nel nucleo. Tutto quello di cui abbiamo parlato ora non è un problema del sistema operativo e se non metteremo mai le mani nel sistema operativo non saranno mai affari nostri. Tornando alla disabilitazione delle interruzioni, funziona, questa garantisce la mutua esclusione però a una condizione: che siamo su un UniProcessor. Se siamo su una macchina parallela posso disabilitare le interruzioni su un processore ma non sugli altri. La disabilitazione delle interruzioni in realtà è una cosa che funziona su il singolo processore. Quindi su MultiProcessor (multi-core) non potremo avere delle Lock fatte cose perché non funzionerebbero. Il secondo problema è che non posso offrire il meccanismo di disabilitazione delle interruzioni agli utenti, ai programmatori, perché può essere utilizzato in maniera impropria, non posso offrirlo neanche sottoforma di chiamata di sistema perché quando la chiamata di sistema finisce comunque le interruzioni le devo riabilitare, non posso permettere all'utente di eseguire codice in stato utente a interruzioni disabilitate, questo non si può proprio ammettere. Quindi non va bene offerto ai programmatori in stato utente. Viceversa può essere usato dal nucleo, in effetti il nucleo questo meccanismo lo usa abbondantemente e abbiamo visto degli esempi: quando ci sono da gestire le interruzioni il processore disabilita le interruzioni per gestire l'interruzione. Questo è un meccanismo di mutua esclusione, è la Lock che usata a bassissimo livello per garantire il salvataggio dello stato del processore. E quando riabilitiamo le interruzioni facciamo la lock_release (). Questo meccanismo è valido, è utilizzatissimo però soltanto nel nucleo e per un periodo di tempo molto breve. Usato dagli utenti non è ammissibile sia per questioni di protezione, sia perché non funziona con i multiprocessori. Quindi ci serve un'implementazione della Lock. Come la facciamo?

Lock Implementation, Uniprocessor

```
LockAcquire(){
  disableInterrupts ();
  if(value == BUSY){
    waiting.add(current TCB);
    suspend();
  } else {
    value = BUSY;
  }
  enableInterrupts ();
}
```

```
LockRelease() {
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
    readyList.Append(thread);
  } else {
    value = FREE;
  }
  enableInterrupts ();
}
```

La Lock non la implemento in stato utente, non è una funzione di libreria, è una funzione implementata sotto forma di chiamata di sistema dal nucleo perché nell'eseguire la Lock devo fare alcune operazioni molto delicate. Come funziona la lock_acquire ()? Prima di tutto disabilito le interruzioni per assicurarmi che nessun altro thread concorrentemente possa eseguire questo codice. Dopodiché testo il valore associato al Lock. È evidente che il Lock è associato a uno stato, per forza deve essere associato a uno stato perché il Lock o è libero o è occupato, quindi in realtà il Lock è una variabile del nucleo binaria, può essere libera o occupata. Se il valore del Lock è occupato il thread non può andare avanti, lo devo sospendere, deve bloccarsi, ma per sospendere il thread devo mettere il suo descrittore in una coda di thread in attesa quindi aggiungo il descrittore del thread corrente alla lista dei thread in attesa su una certa specifica coda dei thread in attesa e poi completo la sospensione, quindi metto quel thread in stato sospeso, chiamo lo

scheduler perché metta in esecuzione un altro thread, riabilito le interruzioni, etc. Se il thread ha trovato la Lock occupata si sospende in questo punto e quando il thread verrà riattivato ripartirà da questo punto, quindi completerà l'If, uscirà dall'If, non eseguirà il ramo else, riabiliterà le interruzioni e restituirà il controllo al thread in stato utente, quindi sostanzialmente qui farà una iRet. Quindi il thread che si sospende resta bloccato nella Lock, quando viene riattivato esce dalla Lock e continua a fare l'istruzione successiva alla Lock. Se invece il valore della Lock è libero allora lo metto ad occupato e faccio una iRet (riabilito le interruzioni e vado avanti). Come funziona la release? Devo disabilitare le interruzioni perché ho bisogno di una mutua esclusione nel codice della lock_release (), è una sezione critica, allora la lock_release () deve riattivare l'eventuale thread sospeso sulla Lock se c'è, se non c'è nessun thread la lock_release () mette il valore della Lock a libero. Quindi se non c'è nessun thread in attesa, metto il valore a libero ed esco dalla Lock, se invece c'è un thread in attesa lo devo riattivare ed esco dalla release. Come faccio a riattivare un thread? Accedo alla coda dei thread in attesa sulla Lock, estraggo un descrittore (quindi scelgo un thread), metto il descrittore di questo thread nella coda Pronti, ovviamente manterrò il thread come pronto anziché come in attesa, farò tutte le modifiche necessarie al descrittore di questo thread per metterlo in coda pronti e a questo punto posso uscire dalla lock_release (). C'è da notare una cosa importante di tutto questo meccanismo: se faccio la lock_acquire () e trovo il valore della Lock ad occupato, non tocco il valore della Lock, lo lascio così com'è e quando vengo riattivato esco, quindi assumo, quando vengo riattivato nella lock_acquire () che il valore sia occupato ancora. Infatti quando vado nella lock_release () e riattivo un thread, quindi lo estraggo dalla coda dei thread in attesa e lo riattivo, non metto il valore del Lock a libero ma lo lascio occupato. Questo ha questo effetto: il thread che viene riattivato (la sezione critica) eredita la mutua esclusione da chi sta facendo la release, quindi il thread che viene riattivato sarà l'unico che potrà entrare realmente in sezione critica. Questo è diverso da quello che invece abbiamo visto un istante prima con la Signal, però la lock_release () non è una signal, sono due meccanismi diversi. Perché devo disabilitare le interruzioni all'inizio della lock_acquire () e all'inizio della lock_release ()? Questa lock_ acquire(), come pure la lock_release(), è una sezione critica perché utilizza strutture dati condivise fra tutti i thread che vogliono fare la acquire o la release. Quali sono queste strutture dati? Il valore e le code. Il valore della Lock e la coda di attesa, infatti il valore è testato sia da chi fa la lock_acquire (), sia da chi fa la lock_release () e stesso vale per la coda. Quindi il codice della lock_acquire () e della lock_release () sono sezioni critiche. Siccome sono sezioni critiche, devono essere eseguite in mutua esclusione. Però è il cane che si morde la coda. Avrei bisogno di iniziare il codice della Lock con una lock_acquire (), ma non posso una lock_acquire () altrimenti avrei una chiamata ricorsiva senza una condizione di uscita, non si può proprio fare. Quindi mi serve un modo alternativo di rendere questa Lock una sezione critica, il modo alternativo che allo stato attuale conosco è questo: disabilitare le interruzioni. Ma questa è un'implicazione molto forte, siccome le interruzioni le posso disabilitare nel nucleo, la lock_acquire () e la lock_release () non possono altro che essere funzioni del nucleo, queste due implementazioni della lock_acquire () e della lock_release (). Quindi la lock_acquire () e la lock_release () devono essere due chiamate di sistema. È un problema che siano chiamate di sistema? Non è un problema, anzi è molto comodo che siano chiamate di sistema. Se guardiamo che cosa fa la lock_acquire (), va a manipolare un descrittore di thread e lo va a mettere in una coda di attesa. La lock_release () va a prelevare un descrittore di thread e lo va a mettere nella coda Pronti, nella lista dei thread pronti per l'esecuzione. Il descrittore di thread, la coda dei thread pronti, la coda dei thread di attesa, sono strutture dati del nucleo se i thread sono implementati al livello del nucleo che è il caso più interessante. Quindi c'è un ottimo motivo per cui la lock_acquire () e la lock_release () devono essere chiamate di sistema, perché manipolano strutture dati del nucleo. Quindi non è affatto un problema disabilitare le interruzioni.
Quest'implementazione funziona nel caso del multiprocessore? Se sono su un multiprocessore, la disabilitazione delle interruzioni disabilita le interruzioni su un processore, quindi il thread che è in esecuzione su un processore non è interrompibile su quel processore ma altri thread in esecuzione su altri processori possono eseguire la Lock contemporaneamente. Questo è un problema perché la Lock è una sezione critica e se è una sezione critica deve essere in mutua esclusione per tutti, non soltanto per

qualcuno. Quindi questa Lock andava bene quindici anni fa, oggi non va più bene. Questa è un'implicazione molto forte perché in realtà con quello che sappiamo ora, non abbiamo proprio modo di fare una Lock che vada bene su multiprocessore, il meccanismo della disabilitazione delle interruzioni non basta, la Lock che abbiamo realizzato non va bene perché ha bisogno lei di essere protetta, non può proteggere sé stessa, non possiamo pretendere che protegga gli altri. In realtà l'aiuto di qualche altro elemento, di qualche altro supporto da parte dell'hardware, non ce la possiamo cavare, ci serve un altro meccanismo.

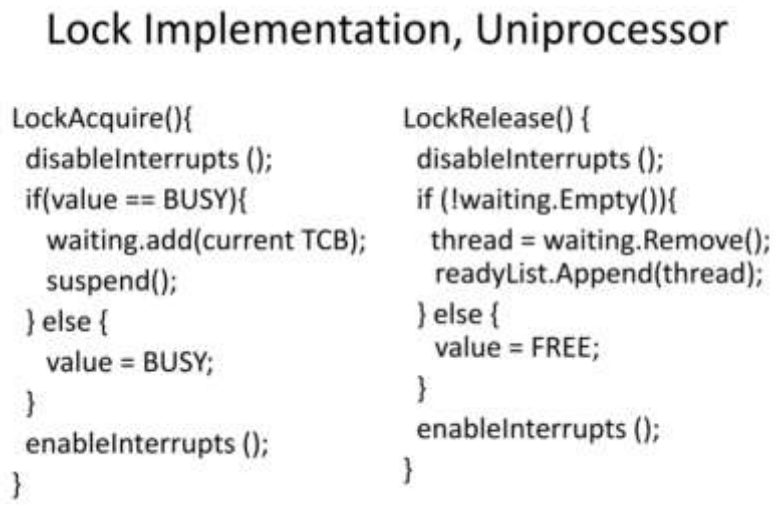


Figura 1

Avevamo visto, la lock la lezione scorsa, nella versione Uniprocessor, vi ricordate? Funzionava in questa maniera: abbiamo due meccanismi che sono di LockAcquire e LockRelease, i due meccanismi prevedono un'attesa passiva, quindi sospendono i thread che non può ottenere la Lock e poi lo riattivano successivamente quando il lock si libera. Per poter effettuare la sospensione è necessario, quindi, prendere il descrittore del thread e metterlo nella coda Pronti, di conseguenza la LockAcquire e la LockRelease devono operare all'interno del nucleo, e per questo motivo sono delle chiamate di sistema.
L'altra cosa importante è che sono delle sezioni critiche a loro volta, il codice, sul quale la LockAcquire e la LockRelease agiscono, in realtà è un codice condiviso che agisce su strutture condivise, in particolare in questo caso, la struttura condivisa è il valore della Lock, la variabile "value" e la coda di sospensione sulla quale vengono inseriti i descrittori dei thread in attesa. Per questo motivo LockAcquire e LockRelease sono sezioni critiche e devono essere a loro volta essere eseguite in mutua esclusione, non si può fare di nuovo mutua esclusione usando la Lock perché sarebbe una cosa ricorsiva. Ci serve un altro meccanismo per garantire la mutua esclusione.
In questo caso, il meccanismo è la disabilitazione delle interruzioni; disabilitando le interruzioni agiamo sul processore, gli impediamo di riconoscere le interruzioni in particolare quella del timer e quindi la riattivazione dello scheduler. In questo modo il thread che sta eseguendo questo spezzone di codice resta in esecuzione per tutto il tempo, fintantoché non riabilita le interruzioni, e chiaramente questa cosa va bene, soltanto perché è un meccanismo del nucleo (la lock è una chiamata di sistema e quindi le interruzioni si possono disabilitare) e poi perché la LockAcquire e la LockRelease sono sezioni critiche piuttosto brevi, per cui se si disabilitano le interruzioni, questo avviene per poco tempo e non causa problemi alla gestione delle interruzioni in generale.
D'altra parte, il fatto che il meccanismo di mutua esclusione in queste due procedure sia affidato alla disabilitazione delle interruzioni ne impedisce di fatto l'uso nel caso del Multiprocessor. Se siamo su un'architettura multiprocessore, è possibile che due thread siano eseguiti contemporaneamente su due processori differenti ed entrambi eseguano la LockAcquire; questo perché ognuno disabilita le interruzioni sul proprio processore, ma non ovviamente sugli altri. Se vogliamo trovare un meccanismo per il multiprocessor, dobbiamo inventarci qualcos'altro. La LockAcquire ovviamente è un meccanismo a livello più alto, ma lei stessa ha bisogno di essere protetta per la mutua esclusione, quindi non può farlo da sola, il meccanismo di più basso livello che è la disabilitazione delle interruzioni funziona soltanto per l'Uniprocessor. Ci manca un nuovo elemento, una qualche cosa che risolva il problema. Allora la soluzione, di nuovo, viene dall'hardware, quindi non è una cosa che si può risolvere a livello del sistema operativo; o

meglio si potrebbe, in linea di principio, risolvere usando uno degli algoritmi simili a quello del Troppo Latte che abbiamo visto un paio di lezioni fa, però ci porteremmo a presso tutti quanti gli svantaggi del caso. Serve invece un meccanismo che sia utilizzabile, più semplice concettualmente da utilizzare.

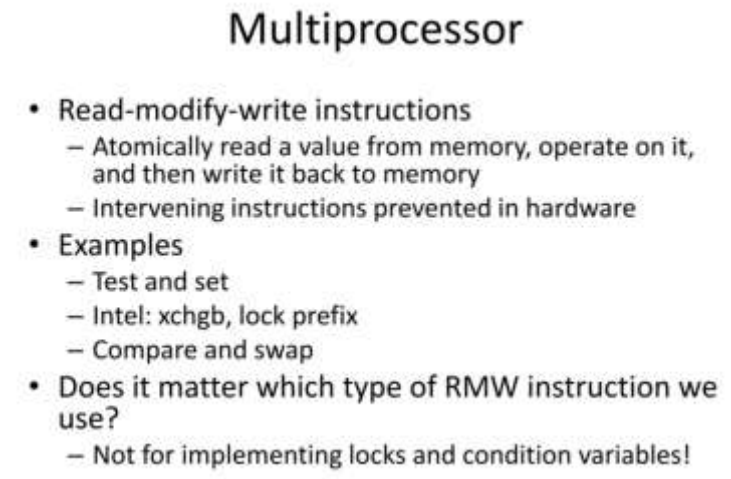


Figura 2

Ci viene in aiuto, di nuovo, l'hardware in termini di istruzioni che prendono forme tipo
READ-MODIFY-WRITE, quindi di istruzioni che sono in grado in un'unica azione atomica (un'unica istruzione del linguaggio macchina) di leggere, modificare e scrivere il contenuto di una cella di memoria. Queste istruzioni sono garantite essere azioni atomiche dall'hardware, in particolare, sul singolo processore sono certamente non interrompibili, perché il processore esegue un'istruzione alla volta, d'altra parte è possibile che un'istruzione di questo tipo sia eseguita contemporaneamente su due processori differenti (questo non lo possiamo impedire), d'altra parte l'accesso in memoria alla cella sulla quali queste istruzioni agiscono, deve passare attraverso un unico BUS ed un'unica Memoria. Quindi in realtà, l'atomicità di questo tipo di istruzioni è garantita dall'hardware attivando dei blocchi sul BUS; quando un processore inizia ad eseguire questa istruzione blocca il BUS e lo riserva per sé, e il BUS resta bloccato fintantoché l'operazione non è conclusa. Ora di esempi di queste istruzioni c'è ne sono tanti, prendono nomi differenti e possono cambiare leggermente la loro definizione d'altra parte non è molto importante né il nome che prendono né, in realtà la specifica semantica che danno alle loro azioni perché tutte quante possono poi essere utilizzate per implementare dei meccanismi di mutua esclusione per i multiprocessori. Noi vedremo come esempio TEST&SET (sulle piattaforme Intel prende altri nomi). Tutte concettualmente fanno la stessa cosa; leggono una cella di memoria, la modificano, e la sovrascrivono.

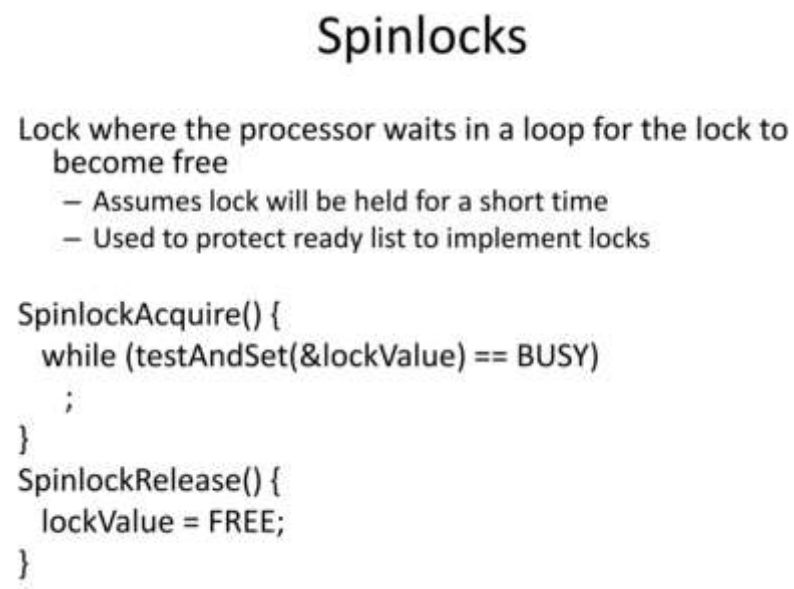


Figura 3

Se abbiamo questo meccanismo, possiamo realizzare via software degli oggetti che si chiamano Spinlock che permettono di acquisire la mutua esclusione nel multiprocessore. Un esempio di Spinlock la trovate qui (<-immagine).
Allora, come funziona la Spinlock? Dunque, la TestAndSet prende come parametro un indirizzo di memoria, una cella, una variabile se volete ma concretamente è un'istruzione Assembler che prende un indirizzo di memoria e fa questo; legge il contenuto della cella di memoria (&lockValue), scrive sulla cella di memoria il valore "occupato" (BUSY), e restituisce il valore letto

precedentemente. Quindi, se voi fate TestAndSet sulla cella e la cella contiene il valore "libero" (FREE), alla fine della TestAndSet chi l'ha chiamata riceve come risultato il valore "libero" ma nella cella lascia scritto il valore "occupato". A questo punto quello che dobbiamo fare è: fintanto che la TestAndSet ci dà risultato "occupato" vuol dire che non possiamo procedere perché, qualcun altro ha acquisito la lock e quindi restiamo impegnati all'interno di questo ciclo. Se invece il valore della $lockValue era "libero", TestAndSet lo mette ad "occupato" in maniera tale che nessun altro dopo di noi possa passare, però il while termina perché viene restituito "libero", quindi fallisce la condizione e a questo punto si può entrare in sezione critica.
Funziona sul multiprocessore, ve lo ripeto, perché anche se, su due processori contemporaneamente due thread tentano di eseguire la TestAndSet, soltanto uno la esegue per primo (non possono eseguirla contemporaneamente per via di un bocco che l'hardware fa sul BUS), di conseguenza trova il valore della cella "libero", la setta ad "occupato" così che il secondo cha arriva la trovi occupata e resti impegnato all'interno di un ciclo.
Ora, se noi abbiamo acquisito lo Spinlock per liberarlo, quando abbiamo terminato la sezione critica, dobbiamo soltanto mettere il valore della lock a "libero". Automaticamente questo libera gli eventuali thread impegnati in un ciclo (il ciclo della Spinlock), ne libera uno (il primo che riesce ad eseguire la TestAndSet sulla lockValue).
Ci sono diverse considerazioni da fare su questa Spinlock; la cosa più importante è che questo Spinlock, è un meccanismo di Attesa Attiva, per cui il thread che non può ottenere la mutua esclusione non si sospende, continua a mantenere impegnato il processore ed esegue un ciclo. Questo è un tipo di comportamento non è auspicabile, è il genere di comportamento che volevamo evitare definendo le LockAcquire e le LockRelease. D'altra parte questo è il prezzo che dobbiamo pagare se vogliamo avere un meccanismo che sia a metà tra l'hardware e il software, che sia abbastanza primitivo da poterci permette di fare mutua esclusione sul codice del LockAcquire.
Allora, a che condizioni questo tipo di meccanismo è accettabile? Certamente va bene se le sezioni critiche sono molto brevi, perché un ipotetico thread acquisisce la mutua esclusione con la Spinlock in tempo molto breve (sperabilmente all'interno del suo quanto di tempo) e quindi può cedere il lock, in questo modo il problema dell'attesa attiva non si pone. Seconda considerazione; è comunque una situazione da usare nel caso del multiprocessore, ora se siamo su un multiprocessore e un thread sta attendendo su un processore, è perché molto probabilmente, un altro processore con un altro thread ha acquisito la sezione critica e sta andando avanti. Quindi di nuovo, se la sezione critica è breve sull'altro processore ci sarà sì, attesa attiva, ma sarà molto breve. A questo aggiungete il fatto che, se le sezioni critiche sono brevi, la probabilità di dover attendere è molto bassa, perché la probabilità che voi troviate un thread già in sezione critica quando voi fate la lock è bassa, perché il thread finisce in tempo. A queste condizioni la Spinlock è accettabile.
Questa Spinlock, secondo voi, si può usare o no a livello utente?
Uno dice NO, perché sennò potremmo acquisire il processore a discapito degli altri thread. L'altro dice Sì perché non tocchiamo le code Pronti, Attesa ecc.
Allora, a che ci serve lo stato supervisore? Ci serve per proteggere il sistema operativo e per proteggere i processi e lo utilizziamo in questa maniera: impediamo a chi non è in stato supervisore di eseguire istruzioni privilegiate e di accedere all'infuori del suo spazio di memoria.
Il punto è: la TestAndSet è un'istruzione privilegiata? Il secondo punto è: la variabile che stiamo usando nella Spinlock è nel nucleo o è in spazio utente?
Immaginate di usarlo in spazio utente; che cosa comporta? Lei ha detto prima "permetterebbe ad un thread l'utilizzo del processore in esclusiva", ma è davvero così? Chi esegue la TestAndSet o la SpinLock.Acquire acquisisce il processore? No, non acquisisce il processore, in realtà se un thread ha il processore, può eseguire la Spinlock e con la Spinlock acquisisce il diritto ad eseguire la sezione critica, che è quello che vogliamo. Quando un thread esegue la Spinlock ed acquisisce la sezione critica, il processore può continuare a ricevere interruzioni e quindi lo scheduler può essere messo in esecuzione; non stiamo

dando ad un thread l'utilizzo del processore, gli stiamo semplicemente permettendo quello che gli permettevamo prima, ovvero, di andare avanti finché non finisce il suo quanto di tempo. Ora, di nuovo, immaginate di usarlo in stato utente. Se devo usarlo in stato utente, questa variabile $lockvalue sulla quale faccio la Spinlock, non può essere una variabile del nucleo; ma chi mi vieta di utilizzare una variabile mia, una variabile che si trova nel mio spazio di memoria, per fare la Spinlock? Se la utilizzo come meccanismo a stato utente, questa lockvalue è una variabile che sta in memoria del processo e che quindi è condivisa con tutti thread di quel processo. Questa lockvalue non è una variabile che i thread usano in maniera libera o sconsiderata, ma è una variabile tramite la quale loro riescono a cooperare. I thread di questo processo non hanno interesse ad usarla male; sarebbe come dire che voi scrivete un programma e poi però per dispetto a voi stessi, ci mettete ogni tanto delle istruzioni sbagliate; dividete a metà le variabili, così, per spregio. Non ha senso.
Quindi, siccome stiamo parlando di entità che cooperano ad ambiente globale e le entità che cooperano ad ambiente globale in stato utente non possono che essere i thread dello stesso processo (processi differenti non condividono memoria, almeno in teoria). Quindi stiamo parlando di thread dello stesso processo e quindi, probabilmente, sviluppati dallo stesso sviluppatore o dallo stesso gruppo di programmatori. Non hanno nessun interesse ad usare male quella variabile.
In realtà, a livello utente non c'è nessun vincolo dal punto di vista della variabile lockvalue (o una qualsiasi cella di memoria). TestAndSet è un'istruzione che legge e scrive una cella di memoria, allo stesso livello di una STORE o di una LOAD e quindi non è un'istruzione privilegiata. Riassumendo, le Spinlock sono uno strumento che potete usare in stato utente.
Ora, può essere usato in stato utente, ma può anche certamente essere usato in stato supervisore. Quindi lo utilizziamo per modificare il codice della LockAcquire e della LockRelease, per avere una versione che funziona anche nel caso del multiprocessore.

Lock Implementation, Multiprocessor

```
LockAcquire(){
  spinLock.Acquire();
  disableInterrupts ();
  if(value == BUSY){
    waiting.add(current TCB);
    suspend();
  } else {
    value = BUSY;
  }
  enableInterrupts ();
  spinLock.Release();
}
```

```
LockRelease() {
  spinLock.Acquire();
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
    readyList.Append(thread);
  } else {
    value = FREE;
  }
  enableInterrupts ();
  spinLock.Release();
}
```

Figura 4

Come funziona la soluzione? Qui in realtà, la disableInterrupts e la Spinlock possono essere invertite, ma non è importante. Siccome la LockAcquire è una sezione critica, noi iniziamo con una Spinlock, acquisiamo la sezione critica, disabilitiamo le interruzioni (in questo testo le interruzioni sono disabilitate perché poi si va ad agire su alcune questioni delicate), dopo che ho acquisito la Spinlock e son sicuro comunque che questo codice verrà eseguito in mutua esclusione, quindi nessun'altro thread potrà eseguire una spinLock.Acquire o una spinLock.Release
contemporaneamente o concorrentemente, a questo punto, il codice resta quello di prima, quindi, vado a testare il valore della lock, se il valore è occupato devo sospendere i thread, quindi metto in coda di attesa il TCB del thread corrente e poi completo la sospensione (devo impostare all'interno del TCB che questo thread è in attesa, devo eventualmente completare il salvataggio dei registri di questo thread all'interno del TCB) , una volta che ho completato questa operazione posso riattivare le interruzioni, rilasciare lo Spinlock e, a questo punto, invoco lo scheduler che sceglierà il prossimo thread da mandare in esecuzione. Quindi la disabilitazione delle interruzioni non mi serve tanto per garantire la mutua esclusione sul valore "value"

della lock, quanto perché in caso di sospensione, devo passare una fase nella quale devo completare la sospensione e questa è una cosa che è bene fare con le interruzioni disabilitate.
Vi faccio notare che: se trovo il valore occupato e non acquisisco la sezione critica, di conseguenza mi sospendo, quando verrò riattivato uscirò dalla LockAcquire e non metterò il valore a BUSY, lo lascio così com'è. Il valore del lock a BUSY lo metto soltanto se lo trovo libero. Quindi se trovo il valore della lock a "libero", vado nel ramo else, lo metto ad "occupato" e a questo punto posso uscire dalla lock; ho acquisito il lock, ho acquisito la sezione critica, posso abilitare le interruzioni e liberare lo Spinlock. Il fatto che io non metta il valore di BUSY della variabile associata alla lock in questo punto, vuol dire che quando qualcun altro mi sveglierà (quindi chi fa la LockRelease), non dovrà mettere il valore a "libero" ma lo dovrà lasciare occupato per me. Devo passare il testimone ed è quello che fa la release, di nuovo, acquisisce la Spinlock, disabilita le interruzioni, se c'è nessuno in attesa sulla coda, quindi se la coda di attesa non è vuota, allora estrae un thread dalla coda , mette il suo descrittore nella coda pronti (dovrà anche marcarlo come pronto) e a questo punto non c'è altro da fare perché il thread che si era sospeso, prima o poi verrà messo in esecuzione (ora è nella coda pronti), quando verrà messo in esecuzione continuerà da questo punto, uscirà dalla lock, ed entrerà in sezione critica e il valore della lock sarà "occupato" (come è giusto che sia) . Se invece nella coda di attesa non c'è nessuno, è vuota, allora l'unica cosa è fare la LockRelease per mettere il valore a "libero", infine riabilita le interruzioni e rilascia lo Spinlock.

Domanda: In questa implementazione, la value che andiamo a cercare è diverso da quello delle Spinlock (lockvalue)?
Ottima domanda. Sì, ogni meccanismo di lock deve avere la propria variabile, quindi, la lock che stiamo seguendo qui (LockAcquire) avrà un suo stato rappresentato dalla variabile "value", in più la spinLock.Acquire e la spinLock.Release che sono un altro meccanismo, faranno leva su un altro valore. Guardate di lock all'interno del sistema e anche all'interno del vostro codice, ce ne saranno tantissime; ogni lock proteggerà una sezione critica, una struttura dati. Ognuna di queste lock, che si riferiscono a strutture differenti, a sezioni critiche di classi differenti, avrà il proprio stato e avrà la propria variabile.

Domanda: Questa lock nella sua interezza è un meccanismo del livello nucleo? Perché agisce sui TCB, sulle code di attesa.
Allora, in realtà, come stanno realmente le cose. Questa qui sì, è un meccanismo di livello nucleo, lo stesso meccanismo lo potete però realizzare a livello utente eliminando alcune cose, per esempio la disabilitazione delle interruzioni, se dovete gestire i thread a livello utente i TCB sono descrittori che stanno nelle strutture a livello utente, quindi, le potete manipolare in quel livello. Però se avete thread a livello utente è più facile fare le lock perché potete agire molto più tranquillamente anche sullo scheduler. In generale questo meccanismo così come lo vedete qui ora è utilizzato ed ha un senso per il nucleo. C'è da chiarirsi, questo meccanismo lo possono usare i thread del nucleo per coordinarsi tra loro, oppure lo possono usare anche i thread dei processi utente se sono implementati a livello del nucleo (è il caso più tipico); i thread che voi userete a laboratorio saranno thread che voi definite a livello utente ma che sono implementati nel nucleo. Quei thread useranno meccanismi di questo tipo che saranno meccanismi del nucleo (Queste lock devono invocare delle chiamate di sistema).
I thread a livello utente hanno anche un'altra possibilità; possono usare semplicemente le Spinlock ***(fig. 3)***.

In effetti, esiste una versione di Spinlock che ammortizza il costo dell'attesa attiva mettendo qua dentro delle Ilde, per cui il thread che si trova a ciclare, cede il processore quindi rilascia il suo quanto di tempo e poi quando lo riacquisisce testa di nuovo la condizione; in questo modo è sempre attesa attiva però costa molto meno perché si fanno meno cicli inutili.
Quindi i thread utente possono usare questo meccanismo, senza invocare chiamate di sistema, oppure possono usare l'altro meccanismo invocandole. Se utilizzano questo ***(fig.4)*** i thread dei processi utente hanno dei meccanismi di mutua esclusione con attesa attiva, altrimenti possono usare questi meccanismi

invocando chiamate di sistema che costano di più (perché l'invocazione costa di più), però prevedono attesa passiva. Per le sezioni critiche lunghe conviene usare questi meccanismi, invece per quelle brevi potete utilizzare le Spinlock. Se io utilizzo questo meccanismo che non prevede attesa attiva, ho un nuovo problema perché questo meccanismo è un'altra sezione critica. Quindi, io devo fare: codice della LockAcquire per decidere se posso entrare o meno nella sezione critica dove manipolo la mia struttura dati, e poi alla fine, codice della LockRelease per cedere questo diritto ad usare la struttura dati. Questo codice molto breve della LockAcquire ***(fig.4)*** è una sezione critica, e va a sua volta protetto con un meccanismo di mutua esclusione, d'altra parte è una sezione critica molto breve. Come la proteggo? Non ho modo di proteggerla con un'altra LockAcquire perché sennò aggiungerei un'altra sezione critica, e poi un'altra ricorsivamente. Non va bene. La LockAcquire non può proteggere sé stessa. Non mi conviene disabilitare le interruzioni perché non funziona nel caso del multiprocessore, quindi mi serve un altro meccanismo, ovvero le Spinlock. Con le Spinlock che succede? Io faccio prima una spinLock.Acquire, piccolissima, per avere il diritto di fare la LockAcquire; se riesco a fare la LockAcquire ottengo il diritto di entrare on sezione critica, chiudo la LockAcquire, chiudo la Spinlock ed entro nella sezione critica molto lunga. Chi arriva nel frattempo che cosa fa? Se io sono il thread A che sta eseguendo la mia sezione critica, quando scade il quanto di tempo può arrivare il thread B che vuole eseguire la lock. Il thread B che cosa farà? Farà la spinLock.Acquire, acquisirà il diritto di far la lock, scoprirà che la lock è già occupata da me, si sospenderà, e da questo momento in poi il thread B, non interviene più, non viene più schedulato, quindi non mi fa più perdere tempo capito? Fintantoché la sezione critica non è completata, viene eseguito soltanto A e gli altri thread che hanno qualcosa da fare, ma non B che sarebbe un inutile perdita di tempo. Quando A ha terminato la sua sezione critica lunga, di nuovo, deve fare la release del lock, quindi acquisisce il diritto di fare la release del lock invocando la spinLock.Acquire, libera il thread B, cede la spinLock.Acquire e chiude la lock che completa la release, da questo momento in poi A, per questa sezione critica grande non ha più niente da dire, ritorna in esecuzione B che viene riattivato, completa la spinLock.Acquire (esce dalla spinLock.Acquire), ha acquisito il diritto di fare la sezione critica lunga, entra in sezione critica e nuovamente può andare avanti a velocità piena senza che il processore perda tempo a schedulare thread inutili.

Domanda: Quindi se ho capito bene, è la variabile della lock che resta a BUSY e quindi la Spinlock può essere acquisita anche da un altro thread.
Sì, è esattamente così. Quando sto eseguendo la sezione critica lunga, il valore della Spinlock diventa "libero" mentre invece il valore della lock è "occupato". Ma questo è logico che sia così, perché la Spinlock è un meccanismo che non protegge la sezione critica grande, protegge la prima parte del codice, ma quando quella parte ha terminato il valore della Spinlock ritorna "libero".
Abbiamo visto, sostanzialmente, due meccanismi che sono il meccanismo della Spinlock e il meccanismo delle Lock, la Spinlock prevede attesa attiva ed è però molto veloce che può essere implementato in stato utente e che se lo Spinlock è libero, in realtà, significa eseguire un paio di istruzioni in linguaggio macchina (una cosa molto veloce), d'altra parte c'è la lock che invece prevede attesa passiva, però comporta l'invocazione di una chiamata di sistema e una quantità di operazioni ben più significative (più invocazioni dello scheduler).

Siccome abbiamo due meccanismi, uno veloce e uno più lento, l'osservazione è che, non abbiamo bisogno di eseguire l'intera lock, se vediamo che è tutto quanto libero allora impostiamo la Spinlock e acquisiamo il diritto ad entrare in sezione critica. Se invece vediamo che è occupato allora, si fa la lock e andiamo a sospenderci, questi sono dei piccoli accorgimenti che vengono utilizzati in Linux per abbattere il costo di queste operazioni.

Figura 5

L'ultimo meccanismo che vi volevo far vedere, è il meccanismo dei semafori:

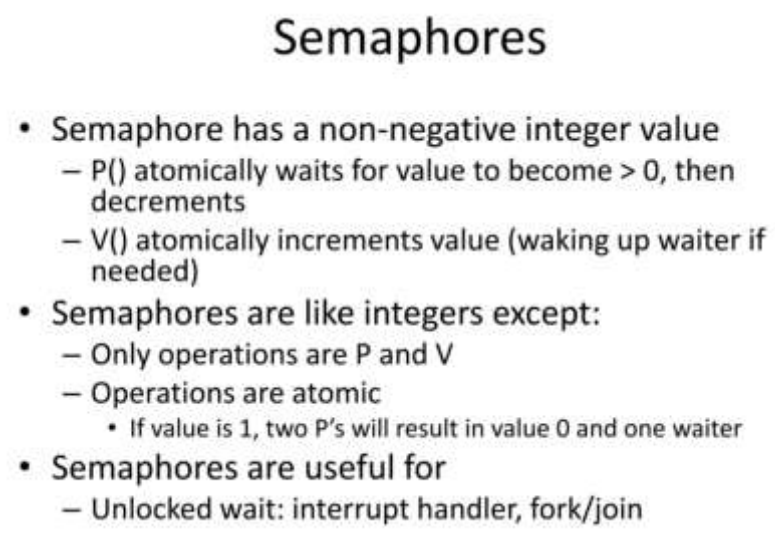


Figura 6

I semafori, storicamente sono nati prima e quando Dijkstra ha proposto la soluzione al problema della mutua esclusione, e poi la criticata, nel criticarla ha proposto proprio il meccanismo dei semafori come possibile soluzione. I semafori per anni sono stati lo strumento principale per risolvere il problema della mutua esclusione all'interno dei sistemi operativi, poi più recentemente, sono stati introdotti i meccanismi più semplici per il programmatore da usare, che sono i meccanismi delle "variabili di condizione". Però i semafori continuano ad esistere perlomeno in alcune fasi della sincronizzazione. Abbiamo visto che le variabili di condizione permettono di sincronizzare due thread (il caso del produttore – consumatore), all'interno di una sezione critica per permettere ad un thread di sospendersi quando non poteva permettersi di fare una certa operazione, e poi per poter essere riattivato quando quell'operazione poteva essere fatta.
Il meccanismo delle "variabili di condizione", per come è fatto, deve stare all'interno di una sezione critica, però se noi vigliamo sincronizzare dei thread al di fuori di una sezione critica, con le variabili di condizione non si può fare. Nei sistemi operativi a volte c'è questa esigenza, soprattutto per sincronizzare thread esterni (tipo i dispositivi), per cui in questi casi si utilizzano i semafori.
Che cosa sono i semafori? I semafori sono delle strutture dati, quindi associano un valore e una coda; una variabile intera positiva e una coda per sospendere i thread e due operazioni atomiche che si chiamano P e

V. Perché si chiamino P e V, sinceramente, non lo so. P è l'equivalente, alla lontana, della sospensione nella LockAcquire e la V sarebbe l'equivalente della release.
P che cosa fa? Testa il valore del semaforo, se il semaforo è maggiore di 0 lo decrementa e non blocca, se il valore del semaforo è uguale a 0, blocca invece il thread che sta eseguendo la P.
La V invece, se ci sono dei thread sospesi in coda del semaforo ne riattiva uno, altrimenti incrementa il valore del semaforo. Potete pensarlo come lo stesso meccanismo delle lock dove il valore è un valore positivo, non è più binario, ma può assumere valori diversi maggiori di 0. Allora, le uniche operazioni permesse sui semafori sono P e V e sono entrambe operazioni atomiche, quindi se il valore del semaforo è 1, il primo thread che esegue la P lo mette a 0 e passa oltre, il secondo thread si sospende. Quindi con valori binari del semaforo, di fatto, si implementa un sistema di mutua esclusione. Se il valore del semaforo è maggiore di 1 per esempio è 3, i primi 3 processi che fanno la P superano la P e possono andare oltre, il quarto processo che esegue la P si sospende.

P&V Implementation, Multiprocessor

```
P(sem){
  spinLock.Acquire();
  disableInterrupts ();
  if (sem.value == 0){
    waiting.add(current TCB);
    suspend(&spinLock); *
} else {
     sem.value --;
     spinLock.Release();
     enableInterrupts ();
}
* Also enables interrupts
```

```
V(sem) {
  spinLock.Acquire();
  disableInterrupts ();
  if (!waiting.Empty()){
    thread = waiting.Remove();
    readyList.Append(thread);
  } else {
    sem.value ++;
  }
  spinLock.Release();
  enableInterrupts ();
}
```

Semaphore Bounded Buffer

```
get() {
  empty.P();
  mutex.P();
  item = buf[front]
  front= (front+1) % size;
  mutex.V();
  full.V();
  return item;
}
```

```
put(item) {
  full.P();
  mutex.P();
  buf[last] = item;
  last = (last +1) % size;
  mutex.V();
  empty.V();
}
```

Initially: front = last = 0; size is buffer capacity
empty/full are semaphores (initialized to 0 and size)
Mutex is a semaphore initialized to 1

Figura 7

Figura 8

Allora l'implementazione delle P e delle V è praticamente identica a quelle delle LockAcquire e LockRelease; l'unica differenza è che il valore del semaforo non è binario, quindi non c'è BUSY o FREE, ma è un valore intero positivo. Se il valore è 0 c'è la sospensione, se il valore è maggiore di 0 c'è il decremento. La V se c'è

qualcuno in coda, riattiva un thread, se non c'è nessuno in coda incrementa. È esattamente la stessa, al netto del valore della variabile.

Vi faccio vedere ***(fig.8)*** come possono essere usati i semafori per risolvere il problema del produttore – consumatore.
Allora, per questo problema uso 3 semafori: un semaforo *mutex* per la mutua esclusione, un semaforo *empty* per sincronizzare produttore e consumatore, in particolare, per sospendere il consumatore se il buffer è vuoto, e un semaforo *full* per sospendere i produttori nel caso in cui il buffer sia pieno. Come devono essere inizializzati? Il semaforo di mutua esclusione deve essere inizializzato con valore 1, perché soltanto un thread può entrare in sezione critica e depositare o prelevare, il valore del semaforo *empty* deve inizializzato a 0, perché inizialmente non ci sono elementi nel buffer da prelevare, il semaforo *full* deve essere inizializzato alla dimensione del buffer perché inizialmente tutte le celle del buffer sono vuote. Ora in questo caso, la dimensione del buffer è data dalla variabile size, quindi *full* prende il valore di size. Se ci sono 10 celle libere nel buffer, il valore di *full* è uguale a 10. Più in generale, il valore del semaforo *full* è pari al numero di celle libere. Il valore del semaforo *empty* è pari al numero di celle occupate nel buffer. Il valore del semaforo *mutex* è 1 se non c'è nessun produttore o consumatore attivo, altrimenti è 0.
Come funziona la get? ***(fig.8)*** Il thread che vuole prelevare deve verificare se ci sono elementi disponibili nel buffer; se lui arriva, inizialmente il buffer è vuoto, quindi lui dovrà sospendersi. Come avviene il meccanismo? Fa una P su *empty*, il valore *empty* è inizializzato a 0 perché non ci sono elementi nel buffer, quindi il thread facendo una P su *empty* scopre che *empty* è 0, e si sospende; se arriva un altro thread consumatore e fa un'altra get, nuovamente farà la P su *empty* e si sospenderà. Quindi fintanto che arrivano consumatori, trovano il valore del semaforo a 0, si sospendono tutti sulla coda associata al semaforo *empty*.
Supponiamo adesso che arrivi un produttore, il produttore come prima cosa fa una P sul semaforo *full*, *full* è inizializzato a size, perché ci sono size celle all'interno del buffer; facendo una P su *full*, *full* è maggiore di 0, decrementa *full,* e continua ad eseguire il codice, quindi andrà a depositare. Avendo decrementato il valore del semaforo *full,* sta in qualche maniera dicendo che, se prima c'erano un certo numero di celle libere adesso ce ne una in meno, perché sta andando a depositare. Prenota una cella libera. Questa prima barriera della *full* di P, può essere superata da tanti thread quante sono le celle libere. Se ci sono 10 celle libere, i primi 10 thread che arrivano fanno la P di *full*, la superano e decrementano *full*; quando è passato il decimo *full* è diventato 0. L'undicesimo thread produttore si sospenderà. Quindi vedete che il semaforo *full* e il semaforo *empty* stanno agendo, non da meccanismi di mutua esclusione, ma da meccanismi di sincronizzazione. Fatta la P su *full,* il produttore va a depositare (lasciamo stare la mutex P e la mutex V), quindi il produttore che ha superato la P su *full,* decrementa *full,* deposita un elemento nel buffer; a questo punto però, avendo depositato un elemento, c'è sì, una cella libera in meno, ma c'è anche una cella occupata in più. Quindi se c'è per caso qualche thread consumatore, che si è sospeso perché tutte le celle erano vuote nel buffer, questo thread deve essere riattivato, a differenza di quanto avviene con le variabili di condizione dove, verificare le condizioni della riattivazione per invocare le signal, in questo caso, nel caso dei semafori, andate a segnalare sul semaforo *empty*.
La V sul semaforo *empty* che cosa fa? Verifica che non ci siano più thread sospesi sul semaforo, se ci sono li riattiva (ne riattiva uno solo). Quindi, noi avevamo un numero di thread consumatori bloccati sulla *empty* di P, il primo produttore ha depositato un elemento, fa la V su *empty* , che scopre che c'è almeno un thread bloccato, lo riattiva , non incrementa *empty,* non ne ha bisogno perché questo thread andrà immediatamente a consumare il dato depositato, il fatto di non incrementare *empty* impedisce ad altri thread che dovessero arrivare ad eseguire la P, di acquisire il diritto di andare a leggere quel dato, soltanto il thread riattivato avrà il diritto di andarlo a leggere.

Supponiamo che invece, arrivi un certo numero di thread produttori. Se il buffer ha dimensione 10, noi possiamo permettere a 10 thread di depositare un messaggio, ma quando il decimo ha depositato il buffer

è pieno e l'undicesimo si deve bloccare. Che cosa avviene in questo caso? Avviene che il valore di *full* era inizializzato a 10, i primi 10 thread hanno superato la P su *full* e l'hanno decrementato, ognuno di loro l'ha decrementato di 1. Quindi, quando è arrivato il decimo thread, ha trovato *full* che aveva valore 1, l'h decrementato portandolo a 0, e ha depositato. Ognuno di questi 10 thread ha fatto la sua V, quindi ci sono state 10 V su *empty* che hanno avuto l'effetto di incrementare il valore del suo semaforo fino a farlo arrivare a 10 (nell'ipotesi che nessun consumatore sia arrivato). Se arriva l'undicesimo produttore, non potrebbe depositare perché il buffer è già pieno, e se ne rende conto perché facendo la P su *full* lo trova a 0 e si sospende. Se a questo punto arriva un consumatore, fa la P su *empty*, trova che *empty* è 10, decrementa *empty,* prenota un elemento da prelevare, lo estrae e fa la V su *full*, questa V ha l'effetto di attivare l'undicesimo produttore, che a questo punto può depositare il suo messaggio nella cella che si è appena liberata.
Il meccanismo di sincronizzazione avviene implicitamente all'interno del semaforo, a differenza delle variabili di condizione che prevede una wait sospensiva a prescindere, e una signal che riattiva se c'è qualcuno sospeso, i semafori hanno uno stato, e questo stato viene usato per decidere se sospendere oppure no.

Domanda: Qual è il guadagno rispetto a prima?
In realtà questi meccanismi sono equivalenti e si possono implementare a vicenda; le lock acquire e release usando i semafori e viceversa.

Storicamente sono nati prima i semafori, e quindi si usavano quelli, in realtà i linguaggi di programmazione moderni non danno ai programmatori i semafori come strumento, ma danno le variabili di condizione, perché i semafori come vi dicevo prima sono associati ad uno stato che è intero e quindi il programmatore deve, in qualche maniera, mappare questo intero nei confronti dell'utilizzo della struttura dati che lui vuole utilizzare, quindi le sue condizioni di utilizzo della struttura dati che può scrivere liberamente con le variabili di condizione nel while, stavolta invece devono essere mappati in una variabile intera di un semaforo. Concettualmente questo è scomodo, è molto più scomodo, e credo sia per questo motivo, principalmente, che ai programmatori ad alto livello, attualmente, vengono come offerti come meccanismi le variabili di condizione, se voi utilizzate i pthread, se voi andate in Java, se voi usate altri linguaggi, il meccanismo di sincronizzazione sono le variabili di condizione.
In pratica, il semaforo è come se includesse il while che incapsula la wait in un unico meccanismo, però quel while che incapsula la wait, ha una condizione che è scritta da voi secondo una logica che riflette il vostro programma, quindi voi lo scrivete sulla base di quello che è logico fare. Nel caso dei semafori, invece, l'utilizzo delle strutture condivise lo dovete gestire tutto in termini della variabile intera.
Una volta che il thread ha acquisito il diritto a depositare, perché ha superato la P di *full*, deve andare materialmente a depositare, ma il deposito stavolta lo deve fare in mutua esclusione perché deve andare a scrivere sul buffer condiviso. Quindi, per fare la mutua esclusione utilizza di nuovo un semaforo, però in modo mutua esclusione; lui fa la mutex.P e la mutex.V per assicurarsi la mutua esclusione su questa sezione critica. Notate che il semaforo di mutua esclusione è usato in questo modo: si fa la P sul semaforo quando si inizia la sezione critica e si fa la V quando si termina, tutti i thread seguono lo stesso schema. Quando invece usate i semafori come sincronizzazione, il valore del semaforo è inizializzato ad un valore che dipende dal problema, da cosa state sincronizzando, da quante risorse ci sono da sincronizzare, e le P e le V avvengono in forma incrociata, chi deve potenzialmente aspettare fa la P, chi deve riattivare fa la V; quindi, il produttore fa la P su *full*, il consumatore che deve riattivare i produttori dà la V su *full.* È incrociato. Viceversa, il consumatore fa la P su *empty,* perché deve aspettare che ci siano elementi, il produttore fa la V su *empty* per segnalare che questi elementi sono arrivati. L'altra differenza sostanziale è che i semafori prevedono stato, per cui, se i produttori inizialmente fanno le V, queste V modificano lo stato del semaforo e poi quando arrivano i consumatori trovano lo stato modificato e superano la P e vanno a consumare. Le variabili di condizione invece non hanno stato, se fate la wait vi sospendete; se fate la signal e c'è qualcuno

sospeso lo risvegliate, altrimenti, se non c'è nessuno in attesa, la signal si perderà. Con i semafori invece, se fate la V e non c'è nessuno sospeso, incrementate lo stato, quindi è preventiva per riattivare già qualcuno dopo.

Ora, in realtà, come dicevo al vostro collega, questi meccanismi sono equivalenti e infatti si possono implementare le P e le V in funzione delle wait e delle signal e vice versa, generalmente avviene il contrario, cioè, si implementano le wait e le signal in funzione delle P e delle V, perché usano dei meccanismi più primitivi (le P e le V le trovate nel nucleo del sistema operativo).
Per farvi vedere che sono equivalenti, vi faccio vedere come si possono trasformare le une in termini delle altre; quindi potrei pormi il problema di, implementare la wait usando la P e di implementare la V usando la signal. La prima cosa che potrei fare è questa:

Implementing Condition Variables
using Semaphores (Take 1)

```
wait(lock) {
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  sem.V();
}
```

Figura 9

Ricordatevi cosa fa una wait su una variabile di condizione; libera il lock, sospende, e poi alla riattivazione acquisisce il lock. Questo è la wait. La signal invece se c'è qualcuno sospeso, lo attiva altrimenti va via.
Quindi potrei implementare la wait facendo così: ***(fig.9)*** LockRelease, sem.P, LockAcquire. E la signal diventerebbe ***(fig.9)*** sem.V.
Questa implementazione non va bene, è molto semplice ma non va bene, perché per quello che vi dicevo prima sulla semantica delle variabili di condizione; se io faccio una signal e non c'è nessuno sospeso, la signal non ha effetto, e se poi dopo di me, qualcuno fa la wait si sospende. Quindi eseguire prima una signal sulla variabile di condizione e poi la wait sospende il thread. Qui invece, se io faccio prima la signal, faccio una V sul semaforo, siccome non c'è nessuno sospeso incremento il valore del semaforo, se poi dopo qualcun altro fa la wait, chi la fa rilascia il lock di mutua esclusione, fa la P di sem, trova che il valore del semaforo è 1, perché è stato incrementato, quindi non si sospende, riacquisisce la mutua esclusione e va

avanti. La wait non è più sospensiva, implementandola in questa maniera vuol dire che, se io faccio prima una signal e una wait dopo, questa non mi blocca.

Questa implementazione non rispetta la semantica. Io la V la dovrei fare solo se c'è qualcuno sospeso in coda.
Dovrei fare qualcosa di questo tipo:

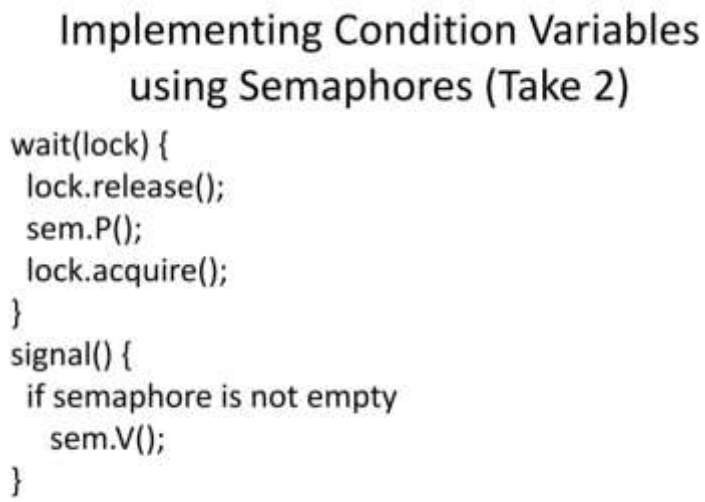


Figura 10

Nella wait, di nuovo, rilascio la mutua esclusione (LockRelease), sem.P per bloccarmi, e poi quando sono riattivato LockAcquire. Invece quando voglio andare a riattivare, a fare la signal, se il semaforo non è vuoto allora faccio la V su sem. In questa maniera evito il fatto che, fare una signal prima, e una wait dopo mi renda la wait non bloccante.

Purtroppo questa soluzione, non funziona neanche, perché ha due problemi: nella wait, il rilascio della mutua esclusione e la sospensione, sono un'istruzione atomica, devono avvenire insieme; qui invece, le ho separate, faccio prima la release e poi la sospensione. In particolare può succedere che arrivi un thread che deve fare la wait, rilascia il lock, viene deschedulato prima di far la sem.P, passa in esecuzione l'altro thread che invece era nella sezione critica, decide di uscirne (fa una signal), la signal scopre che il semaforo è vuoto perché nessuno ancora ha fatto la P, quindi non fa nessuna V e va oltre, a questo punto viene rimesso in esecuzione il thread precedente che fa la sem.P e resta bloccato perché ha perso la signal. L'altro problema è che i semafori sono strutture primitive, vi ho detto prima che il semaforo associa ad una struttura dati due operazioni (P e V), che sono le uniche ammesse ad operare su questa struttura per usarla correttamente. Ma qui, io sto andando a guardare il valore del semaforo con un'operazione diversa da P e da V. Questo non è permesso.

Come si fa ad implementare la wait e la signal in funzione di solo P e V? Si può fare ma bisogna complicare leggermente la soluzione.

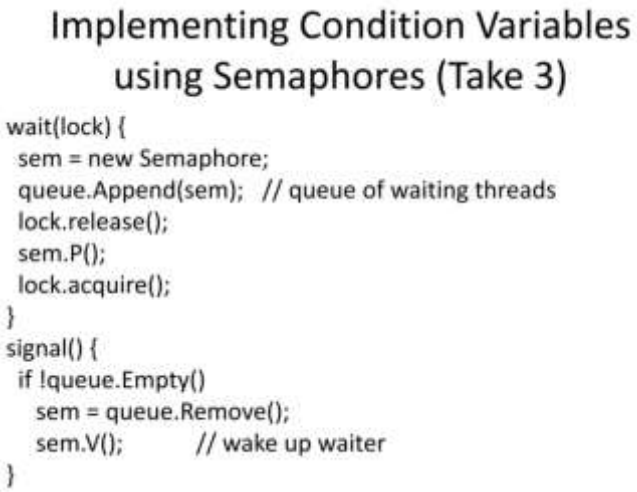


Figura 11

Il problema grosso è che dobbiamo evitare di andare a testare, la coda del semaforo e quindi dobbiamo avere un'altra coda, dobbiamo gestire esplicitamente un'altra coda per andarla a testare. Il secondo punto è che, la sospensione e il rilascio della lock, devono essere fatte atomicamente. Siccome chi fa la wait è già in mutua esclusione, perché quando la eseguite prima avete fatto una LockAcquire (la wait va sempre utilizzata all'interno della LockAcquire e LockRelease), quindi per prima cosa dovete sospendervi e dopo la sospensione potete fare la LockRelease; il problema è che, se vi sospendete non potete fare la lock release, allora che fate? Esattamente come il giochino, che abbiam visto la lezione scorsa, della riattivazione FIFO con le variabili di condizione; create una struttura parallela (coda) sulla quale mettete i thread sospesi, accodate il vostro descrittore su questa coda, che nel caso precedente era una coda di variabili di condizione private, in questo caso è una coda di semafori privati.
Quindi creiamo una coda di semafori privati, creiamo un nuovo semaforo, appendiamo questo nuovo semaforo privato sulla coda, a questo punto, formalmente siamo già sospesi perché ci siamo inscritti "all'albo" dei sospesi (siamo nella coda dei thread sospesi). Concretamente non siamo ancora sospesi perché non abbuiamo rilasciato il processore, ma di fatto lo siamo.
Ci mettiamo nella coda dei thread sospesi, rilasciamo la mutua esclusione e a questo punto, facendo la sem.P, completiamo sul nostro semaforo privato la sospensione.
Chi fa la signal può andare a testare lo stato della coda, se non è vuota allora estrae un elemento e a questo punto fa sem.V sul semaforo privato associato a quel thread che si è sospeso (quindi riattiviamo solo quel thread). Siccome questa sem.V modifica lo stato, che avvenga prima questa sem oppure la sem.P non è più rilevante.

// Una cosa che dovete sempre tenere a mente quando usate la mutua esclusione. Se io ho una struttura dati, su quella struttura dati devo individuare tutte le sezioni critiche che agiscono su di essa, potrebbero essere più di una. Allora, alla struttura dati associo un lock specifico per quella struttura, e tutte le sezioni critiche devono fare una lock acquire su quella variabile di lock. Questo vuol dire che strutture differenti, devono avere la loro variabile di lock. Se io ho 100 strutture dati condivise, allora avrò 100 variabili di lock. //

Sistemi operativi 23/03/17

Vediamo l'ultimo argomento che riguarda ciò che succede quando dobbiamo andare a sincronizzare più thread nel momento nel quale cooperano utilizzando più oggetti condivisi. Quello che abbiamo visto fin ora è ciò che succede quando noi abbiamo un'unica struttura dati (ad esempio nel produttore-consumatore c'è un unico buffer) e più thread contemporaneamente utilizzano questa struttura e abbiamo visto che serve un meccanismo di mutua esclusione e un meccanismo di sincronizzazione, abbiamo visto diversi tipo e abbiamo visto come si gestiscono. Vediamo ora cosa succede nel momento nel quale abbiamo più oggetti differenti (quindi più strutture dati) ognuno dei quali ha le proprie esigenze di sincronizzazione quindi ci troviamo in una situazione nella quale ad ogni struttura dati dobbiamo associare un proprio lock e delle proprie variabili di condizione per gestire correttamente l'interazione su quella struttura però contemporaneamente ne abbiamo diverse. Vediamo quali sono i problemi che nascono quando dobbiamo sincronizzare più risorse, vediamo la definizione di deadlock, vediamo quali sono le condizioni che possono portare al deadlock e che cosa si può fare.

La vera questione che ci stiamo ponendo è se questi meccanismi di sincronizzazione che abbiamo visto (meccanismo di lock, meccanismo dei semafori, meccanismo delle variabili di condizione) sono modulari cioè scalano dal punto di vista della semplicità di utilizzo delle problematiche che possono nascere. Soprattutto un aspetto chiave, che vediamo la lezione di oggi, è lo stallo o deadlock che è una situazione patologica che si presenta in queste circostanze. Quindi sebbene voi utilizzate correttamente gli strumenti che abbiamo visto per sincronizzare ogni singolo oggetto (usa sempre la struttura consistente, utilizza i lock e le variabili di condizione quando utilizza l'oggetto, acquisisce il lock all'inizio della sezione critica, rilascia il lock alla fine della sezione critica, mantieni il lock fintanto che sei all'interno della sezione critica e se per caso devi aspettare fai la wait nel loop rilasciando il lock e soprattutto non utilizzare meccanismi di attesa attiva che possono creare dei difetti di performance) quello che può capitare è di avere comunque problemi legati allo stallo. Vediamo cos'è il deadlock (o stallo). Ci sono un bel po di definizioni da dare in particolar modo ci riferiremo alle risorse come quegli oggetti, strutture passive che devono essere utilizzate dai thread per poter completare il loro lavoro. Quindi cos'è una risorsa? Potrebbe essere una risorsa hardware (tastiera, schermo, finestra schermo), potrebbe essere una risorsa software (struttura dati), può essere spazio disco, può essere disponibilità di processore (quindi il processore stesso). Queste risorse possono essere di due tipi: prerilasciabili e non prerilasciabili. Cosa vuol dire? In generale una risorsa nasce non prerilasciabile; se io penso ad una risorsa hardware come potrebbe essere la tastiera o come può essere il mouse, lo assegno e l'utente ha disponibilità di quella risorsa e non glielo posso togliere così liberamente senza avvisarlo. Lo stesso processore in linea di principio non può essere prerilasciabile. In realtà molte risorse possono essere virtualizzate e rese prerilasciabili (abbiamo visto che questo succede per il processore). Creando i meccanismi di commutazione di contesto, io vorrò creare dei processori virtuali e a quel punto assegno i processi risorse virtuali (non risorse fisiche) in questo modo la risorsa fisica, che di per se non sarebbe prerilasciabile, trasformata in risorsa virtuale diventa prerilasciabile.

Il problema è che questa virtualizzazione non può essere fatta per tutte le risorse. Immaginate di avere una stampante se io inizio a stampare un documento e forzo un prerilascio della stampante, vuol dire che riassegno la risorsa ad un altro thread quindi quell'altro thread inizierà a stampare. Il risultato sarà che la stampa sarà un mix di stampa di un thread e stampa di un altro thread. In generale è molto scomodo se non rinutilizzabile per l'utente. Quindi ci sono alcune risorse che non posso virtualizzare facilmente o che non posso virtualizzare affatto, ci sono altre risorse che posso virtualizzare. Se posso virtualizzarle riesco a renderle prerilasciabili quindi posso riassegnarle in qualche maniera. Se invece non riesco a riassegnarle restano non prelisciabili. Un'altra definizione che ci interessa è quella di attesa indefinita che è una situazione nella quale, i thread aspettano un tempo indefinito che qualche cosa avvenga. Abbiamo visto che i thread occasionalmente devono aspettare. Quando usiamo un lock per acquisire una sezione critica, se per caso il

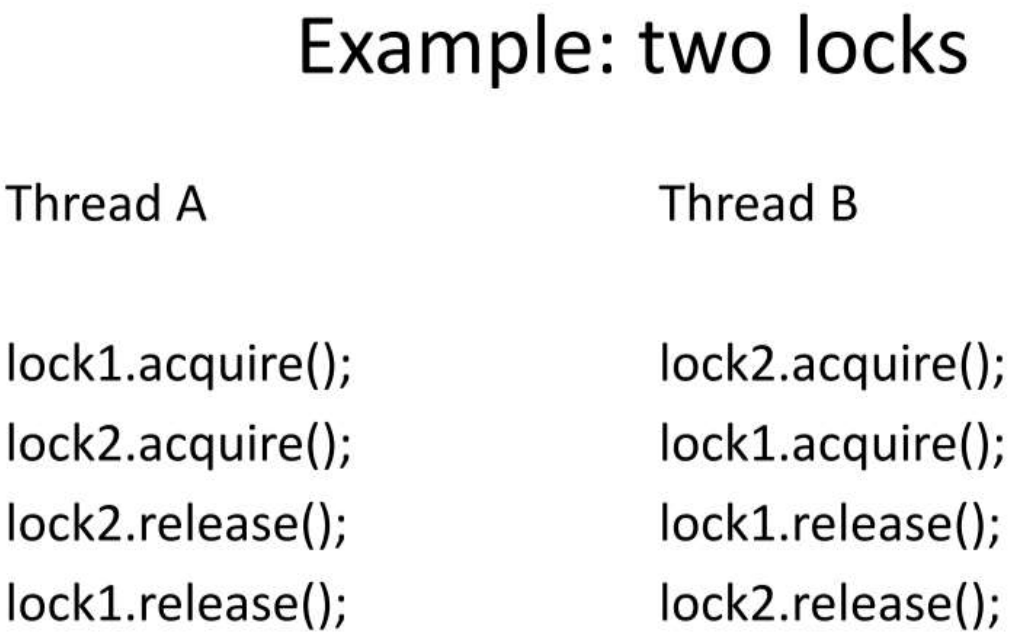
diritto di acquisizione della sezione critica è già stato assegnato a qualcun altro (quindi qualcun altro ha già fatto il lock) il thread che fa la lock_acquire deve aspettare. Abbiamo visto un altro caso nel produttore-consumatore, se anche il thread ha avuto il diritto di utilizzare la sezione critica, nel momento nel quale vuole consumare un dato dal buffer ma non c'è nessun dato nel buffer deve aspettare. Nel caso del produttore-consumatore, nella gestione della mutua esclusione, questa attesa non è indefinita infatti è definita perché uno aspetta fintanto che la condizione di blocco non svanisce e quindi il thread viene liberato. Che significa concretamente? Significa che se un thread sta aspettando la mutua esclusione, il thread che ha ottenuto la mutua esclusione esce dalla sezione critica e fa la lock_release, questo automaticamente libera il thread che stava aspettando. Non sappiamo quanto impiegherà in thread in sezione critica a completare il suo lavoro dentro la sezione critica ma non appena terminerà, sbloccherà e gli altri potranno accedere. Come pure il thread che sta aspettando un dato nel buffer nel produttore-consumatore, non può sapere quando arriverà il produttore a depositare però quando il produttore depositerà si sbloccherà anche il consumatore. Questa situazioni non sono situazioni di attesa indefinita. Una situazione di attesa indefinita invece è quella nella quale un thread resta bloccato ma non c'è più certezza che questo possa mai essere sbloccato quindi il suo tempo di attesa non è più limitato superiormente, non è più dipendete soltanto dal comportamento dei thread che lo potrebbero sbloccare ma ci sono delle altre problematiche che lo stanno portando ad aspettare un tempo indefinito. Infine l'ultima definizione importante è quella di deadlock che è una situazione di attesa circolare nell'utilizzo delle risorse. Cosa vuol dire attesa circolare? Vuol dire che un certo numero di thread compete nell'utilizzo di risorse condivise. Queste risorse condivise non possono essere prerilasciabili perché altrimenti ogni volta che un thread le richiede potrebbero essere tolte da un thread e riassegnate ad un altro quindi devono essere risorse non prerilasciabili. Se un thread trova una risorsa occupata deve andare in stato di attesa. Può capitare se non siamo attenti nel coordinamento di queste azioni di utilizzo delle risorse che in realtà tutti i thread entrano in stato di attesa delle risorse che sono occupati da loro stessi. Per cui i thread entrano in uno stato di attesa circolare. Ogni thread per andare avanti ha bisogno di una risorsa che è già stata impegnata in precedenza da un altro thread e quell'altro thread a sua volta è in attesa di un'altra risorsa che a sua volta è impegnata in uno degli altri thread (che potrei essere io) all'interno di questo (?) di thread. Se tutti i thread sono bloccati in attesa di risorse che solo loro possono sbloccare, abbiamo un'attesa circolare perché l'unica cosa che li possa sbloccare è un'azione che loro possono compiere: il rilascio di una risorsa, ma nessuno di loro rilascerà risorse perché sono tutti bloccati in attesa.

Domanda collega: Quando io entro in attesa non rilascio la risorsa prima di entrarci? Non è detto, non sempre. Se io inizio una stampa e quindi acquisisco la risorsa stampante per poter stampare, mentre sto facendo la stampa ho bisogno di andare a leggere un'ulteriore informazione da dover stampare e me ne rendo conto per qualche motivo solo lì, farò un tentativo di acquisire quell'altra informazione, quella risorsa, quella struttura. Nel far questo se io mi devo bloccare, non posso rilasciare la stampante perché altrimenti la mia stampa dopo che fine fa?

Ritornando alla situazione di prima: il deadlock è una situazione di stallo nella quale tutti i thread attendono che si liberi una risorsa che però loro stessi hanno impegnato in una qualche maniera. La situazione di deadlock in questo caso chiaramente implica un'attesa indefinita perché in realtà questi thread non verranno riattivati da nessuno, resteranno bloccati in eterno. Il deadlock implica attesa indefinita ma non è vero il contrario. Potrei avere una situazione di attesa indefinita senza avere deadlock. Un esempio di attesa indefinita: immaginate di voler fare un concorso pubblico in una qualche repubblica delle banane e di non essere incorsati, questa è una situazione di attesa indefinita perché tutti quanti quelli che avranno l'incorso ci passeranno avanti mentre noi resteremo in attesa in eterno per il posto di lavoro. Tecnicamente non siamo in deadlock, siamo in attesa indefinita.

Facciamo delle ipotesi: assumiamo di avere risorse riutilizzabili (quindi risorse che possono essere richieste, ottenute, utilizzate e dopo che sono state chieste e utilizzate possono essere rilasciate e riutilizzate da qualcun altro.) Se stiamo parlando di risorse non riutilizzabili l'utilizzo delle stesse le distrugge ovviamente

non ci può essere deadlock perché nessuno le può riutilizzare dopo di noi. (13.35) Quindi il problema della sincronizzazione multi oggetto delle problematiche che scaturiscono, nasce nel caso di risorse riutilizzabili. Normalmente noi pensiamo di avere risorse riutilizzabili un esempio è una risorsa hardware: io utilizzo una stampante, stampo, la rilascio e qualcuno la riutilizzerà dopo di me. Un esempio di risorsa riutilizzabile è una mutua esclusione quindi il diritto di eseguire una sezione critica: io acquisisco il diritto di usare una sezione critica per la mutua esclusione per un certo oggetto, quando ho terminato rilascio questo diritto e qualcun altro può acquisire lo stesso diritto dopo di me. Il diritto di eseguire una sezione critica su un oggetto in mutua esclusione è una risorsa ed è riutilizzabile. Un altro aspetto importante è il modo con il quale avviene la richiesta delle risorse. Bisogna che il modello con il quale avviene la richiesta delle risorse segua la sequenza con la quale siamo abituati. Io richiedo di acquisire una risorsa utilizzando un meccanismo di quelli visti la lezione scorsa (ad esempio una lock_acquire), suppongo che questa richiesta sia bloccante per cui se richiedo la risorsa e non la ottengo allora mi blocco in attesa che quella risorsa si liberi, quindi richiedo la risorsa e se non è disponibile mi blocco in stato di attesa mentre se è disponibile la acquisisco, la utilizzo e quando ho terminato di utilizzarla la rilascio. Bisogna che il modo con il quale viene svolto l'utilizzo di risorse segua questo flusso e questo è il modo normale con il quale operiamo nei sistemi operativi quello che normalmente viene offerto. Quando dobbiamo fare una qualsiasi operazione la facciamo invocando le chiamate di sistema, le chiamate di sistema acquisiscono per noi quella risorsa, se la risorsa è libera attuano quell'operazione, in seguito la rilasciano e ci restituiscono il controllo. Se la risorsa è occupata sospendono il nostro thread. Questo modello di funzionamento, che è quello che rende semplice l'uso delle risorse, in realtà è uno degli elementi chiave per produrre potenziali situazioni di deadlock perché ha in mezzo un meccanismo di attesa, causato dall'attesa bloccante, che è quello poi alla fine è la radice del deadlock. Un esempio di sistema che non ha questo tipo di problematica: se io quando ho bisogno di una risorsa la richiedo ma se la risorsa non è disponibile e mi viene detto che la risorsa non è disponibile ed io posso fare altro ovviamente in queste condizioni non ci sarà mai una deadlock perché non andrò mai in stato di attesa. È anche vero che in questo tipi di sistemi, l'utilizzo delle risorse è estremamente scomodo e un thread che non ottiene una risorsa quando la vuole può pure fare altro ma se non ha niente altro da fare che fa? Quindi questo meccanismo di gestione delle risorse vale solo per sistemi speciali con caratteristiche abbastanza diverse alle quali siamo abituati. Nei sistemi generali quelli con cui normalmente abbiamo a che fare questo è il meccanismo normale di accesso ed uso delle risorse e quindi con questo ci dobbiamo confrontare. Vediamo cosa può succedere: immagiamo di avere due thread che devono acquisire due risorse e le acquisiscono (per



esempio sono due sezioni critiche) tramite due lock. Una lock_acquire sulla variabile lock1 e una lock_acquire sulla variabile lock2. Quindi il threadA prima acquisisce la sezione critica 1 (quindi fa la lock1.acquire) poi fa la lock2.acquire quindi acquisisce il diritto a fare la sezione critica di tipo 2, quando poi qua in mezzo avrà la disponibilità di entrambe le mutue esclusioni, farà ciò che deve fare e poi le rilascia. Un altro thread totalmente indipendente realizzato da un altro programmatore potrebbe aver bisogno degli stessi diritti sulle mutue esclusioni ma con un ordine differente quindi vorrà prima acquisire la lock sulla risorsa 2 (quindi il diritto ad eseguire la sezione critica n°2) poi acquisisce il diritto ad eseguire la sezione critica 1 e poi correttamente li rilascia.

Se io li prendo singolarmente il thread A e il thread B fanno tutto correttamente. Se io analizzo singolarmente il thread A quel thread A è corretto e un compilatore o chiunque lo analizzi non avrà niente da ridire. Stesso discorso vale per il codice del thread B. Il problema è che quando questi due codici li mettiamo assieme e li facciamo funzionare contemporaneamente cosa succede?

Risposta collega: vanno il deadlock perché quando abbiamo bisogno della lock1 ce l'ha l'altro thread e viceversa.

Esattamente. Se il thread A acquisisce la lock1 e viene deschedulato a quel punto può passare in esecuzione il thread B acquisisce la lock2 a questo punto fa la lock_acquire e va in stato di attesa perché il lock1 è già stato impegnato dal lock del thread A. Quindi il thread B va in stato di attesa, cede il processore, prima o poi ritorna in esecuzione il thread A, il thread A che aveva acquisito già la lock1 fa la lock_acquire per acquisire il lock2 ma lock2 è già stato acquisito dal thread B, quindi anche il thread A si blocca. A questo punto sia il thread A che il thread B sono sospesi, chi può riattivare A o B? A può essere riattivato solo da B perché B ha acquisito la lock2 e solo B può rilasciare la lock2 mentre B può essere riattivato solo da A perché A ha acquisito la lock1 e solo A può fare la release di lock1. Abbiamo una situazione di diritto incrociati per cui questi due thread sono sospesi su due risorse (lock1 e lock2) che sono detenuti da loro ed ognuno per andare avanti ha bisogno di un'azione dell'altro. Questa è una situazione di attesa circolare.

Ci sono da fare due considerazioni:

1. Da dove nasce questo problema?
2. Considerazione pratica

Partiamo dalla prima: perché si sta verificando questo problema? Risposta collega: le due lock servono assieme. Risposta prof: in realtà non è detto che servono assieme perché può darsi che il thread A abbia bisogno di fare qualcosa all'inizio sulla risorsa associata alla lock1 poi acquisisce anche la lock2 e fa una cosa che riguarda sia la risorsa 1 che la risorsa 2 mentre il thread B ha magari il bisogno di lavorare al contrario: ha bisogno di lavorare prima sulla risorsa 2 e poi lavora contemporaneamente sulla risorsa 1 e 2. Quindi non è detto che servono contemporaneamente, può darsi ma non è necessario.

Il problema nasce da un'altra questione ovvero che sono annidate ovvero che abbiamo eseguito queste due lock nell'ordine opposto ed è proprio questo che causa lo stallo. Se noi forzassimo lock1 e lock2 ad essere acquisite sempre con lo stesso ordine, lo stallo non ci sarebbe mai. Questo cosa vuol dire? Vuol dire che se noi acquisiamo le risorse in maniera ordinata, eliminiamo il problema dello stallo alla radice ma questo per il thread B può essere innaturale perché B è costretto ad acquisire prima la risorsa 1 anche se non gli serve ma d'altra parte se questa piccola forzatura sul comportamento di B permette di evitare lo stallo ben venga. C'è una questione invece di ordine pratico (considerazione 2): qual è la probabilità che uno stallo avvenga con il codice strutturato in questa maniera? (figura su) Immaginate di aver scritto un codice molto lungo e complesso dove ad un certo punto per qualche motivo non ce ne siamo accorti e abbiamo invertito l'ordine dei lock di due thread (e può capitare se il codice è molto complesso) a questo punto debugghiamo il codice provandolo e quindi mandandolo in esecuzione. La disgrazia è questa: se un codice del genere lo mandiamo in esecuzione, può darsi ed è molto probabile in realtà, che non succeda niente o meglio succede esattamente

quello che ci aspettiamo che succeda e quindi che non ci sia nessuno stallo e vada tutto bene. Questo avviene perché lo stallo si verifica in situazioni davvero sfortunate, bisogna che un thread A venga interrotto dopo aver lock1 e prima di aver acquisito lock2 e magari queste due acquisizioni sono molto ravvicinate e la probabilità che il quanto di tempo scada proprio in quel punto è abbastanza bassa. Non solo, se anche lui fosse interrotto in quella fase, bisogna che passi in esecuzione il thread B che, per qualche coincidenza strana, proprio in quell'istante ha voluto acquisire la lock2. Il risultato è che questo codice mandato in esecuzione 1000 volte funziona sempre bene per far vedere che un codice va in deadlock quello che si fa (almeno il professore faceva così tempo addietro) è mettere delle sleep in certi punti del codice per ritardare il codice e forzare l'intervento dello scheduler (al fine del calcolo è inutile) e mettendo le sleep nei punti giusti del codice quello che succedeva era che effettivamente il codice andava il deadlock proprio per forzare questo tipo di situazioni.

Il vero problema è che un debug convenzionale con questo tipo di situazioni non funziona, o meglio se siamo molto fortunati il sistema va in stallo, non capiamo perché, è molto difficile riproporre lo stallo mandandolo in esecuzione però per lo meno ci si è accesa una lampadina che dobbiamo andare a cercare una causa di stallo. Il problema è che non è facilmente riproducibile perché dipende dalla velocità dell'avanzamento dei thread, può capitare di tutto, quindi bisogna avere occhio ed analizzare il codice con molta attenzione per capire perché. Il più delle volte invece non va in stallo nei test, lo consegniamo al nostro committente che lo installa sullo space shuttle ed esplode. Siamo comunque fortunati perché sappiamo che il problema nasce nel momento nel quale i lock vengono utilizzati in maniera invertita per cui per lo meno se stiamo molto attenti e andiamo a cercare nel codice tutti i lock invertiti abbiamo eliminato il problema. Che mi dite di questo?

Two locks and a condition variable

Thread A	Thread B
lock1.acquire();	lock1.acquire();
…	…
lock2.acquire();	lock2.acquire();
while (need to wait)	….
condition.wait(lock2);	condition.signal(lock2);
lock2.release();	lock2.release();
…	…
lock1.release();	lock1.release();

Risposta collega: questo causa anche attesa perché se per esempio il thread A va in attesa perché non può eseguire le sue operazioni a prescindere non ci sarà il thread B a soddisfarlo.

Prima di analizzare il problema notiamo una cosa: in questo esempio sia lock1 che lock2 sono sempre eseguiti in ordine quindi non è un problema di ordine di lock1 e lock2 qui, li sto eseguendo il ordine, qui il problema

di prima non si presenta. Il problema come dice giustamente il vostro collega è questo: immaginate di essere il thread A, il thread A acquisisce la lock1 dopo acquisisce lock2, lo trova libero e va avanti dopo di che si rende conto che nell'utilizzo della risorsa due deve aspettare e quindi lui correttamente fa quello che ci è stato insegnato quindi all'interno di un ciclo fintanto che ha bisogno di attendere fa una wait su lock2. Il problema è che lui si sospende all'interno della lock rilasciando correttamente lock2 ma non rilascia lock1. Quindi se lui per caso si sospende qua dentro tutti gli altri eventuali thread si trovano a non poter accedere a lock1 e quindi un eventuale thread che potrebbe potenzialmente riattivarlo, sbloccando questa condizione (quindi un eventuale thread che potrebbe fare la signal per sbloccare questa condizione), in realtà non può entrare a fare questo sblocco perché viene bloccato su lock1. Lo schema intorno a lock2 è quello che abbiamo utilizzato fino ad ora, non è niente di diverso, acquisiamo lock2 facciamo qualche cosa, ci accertiamo di avere il lock, se non l'abbiamo facciamo la wait all'interno del loop, se veniamo riattivati riacquisiamo lock2 e poi facciamo quello che dobbiamo fare e poi rilasciamo lock2. Qui stiamo astraendo, stiamo ignorando tutte le altre azioni che non hanno a che fare con la risorsa perché qui non ci interessa. Qui ci interessa vedere come stiamo usando il meccanismo. Il thread B qui farà un lavoro differente, potrebbe essere un consumatore o potrebbe produrre un dato che porta a fare una signal e il problema è che questa signal non la può fare perché viene bloccato su lock1. Il problema qui non è che il thread B detiene una lock su qualche cosa, il thread B in realtà non detiene una risorsa, lui è l'unico che può generare una risorsa in questo caso che può portare allo sblocco del thread A mentre invece il thread A è quello che detiene la risorsa in stato di attesa. Quindi il thread A detiene la risorsa 1 in stato di attesa, il thread B è quello che può generare la risorsa che libera la wait del thread A ma si blocca in attesa di lock1. Quindi il problema in realtà è molto più subdolo, non è soltanto un problema di ordine della mutua esclusione è anche un problema di combinazione nell'acquisizione delle risorse che non sono necessariamente sempre associate ad una lock (in questo caso il thread B genera una risorsa che ancora non lo è). È sempre un problema di annidamento in realtà come vedremo però non è un annidamento esplicito sulle lock o sul meccanismo esplicito che abbiamo visto. In questo caso l'annidamento è tra un qualche cosa che il thread B può fare perché concettualmente è lui che fa una certa azione, è lui che fa la signal che sarà legata alla produzione di una certa risorsa ma che non si esplicita nell'acquisizione di un lock a prescindere. Quando il thread B si blocca, si blocca ma non detiene nessuna lock quando si blocca.

Collega: Io avrei dovuto rilasciare lock 1 prima di acquisire lock 2 e poi riacquisirlo sotto?

Prof: Avrebbe dovuto avere un meccanismo di wait che permette rilasciare due lock (lock1 e lock2) che non c'è però perché se ci fosse pure un meccanismo che rilascia lock1 e lock2 vi faccio lo stesso esempio con 3 lock.

Collega: No, io dicevo di rilasciare lock1 prima di acquisire lock2

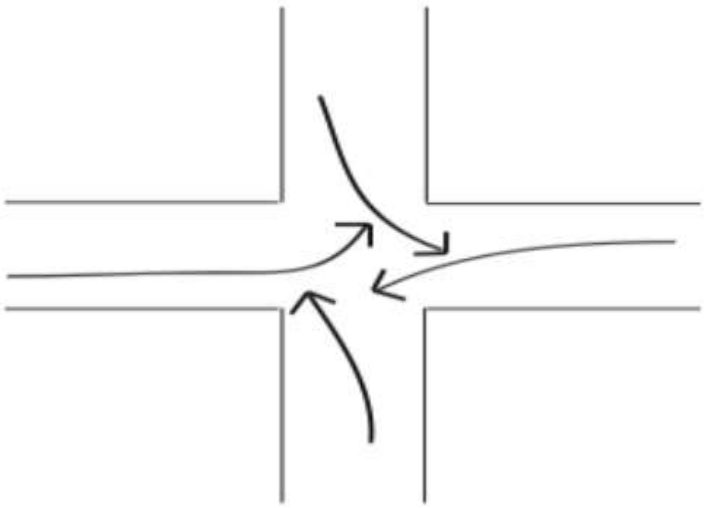
Prof: Eh si, però a quel punto bisogna vedere il codice… Non è detto che il codice sia strutturato per poterlo fare. Se il thread A acquisisce lock1 e lavoro sulla risorsa 1 e però la lascia inconsistente perché deve ancora finire il lavoro, deve prendere un'altra risorsa e poi dopo deve completare il lavoro sulla risorsa uno, non può rilasciare lock1 qui perché per come è strutturato rischia di lasciare la risorsa inconsistente. Questo in realtà è scorretto perché il programmatore ha lavorato male però è molto difficile da correggere perché è un problema proprio concettuale di organizzazione del codice e siccome è un problema concettuale di organizzazione è un problema nasce da lontano e non è sempre facile vederlo e non è sempre facile correggerlo. Se il programma è stato scritto in questa maniera, andare a correggere questa struttura significa andare a riscrivere un bel pezzo di thread A e di thread B. Purtroppo non è soltanto questo il problema.

Bidirectional Bounded Buffer

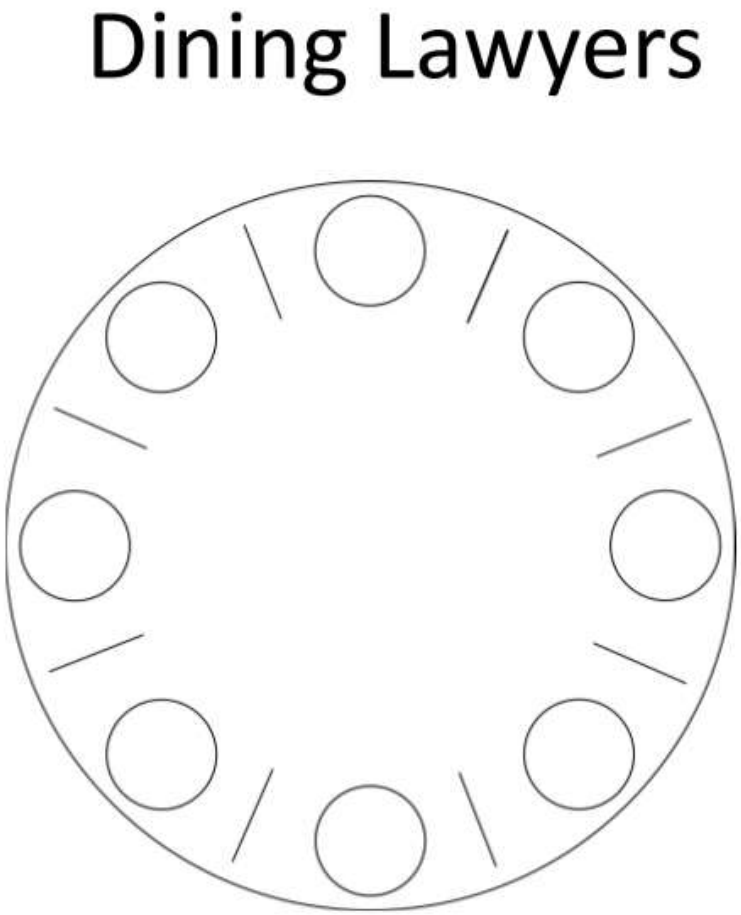
Thread A	Thread B
buffer1.put(data);	buffer2.put(data);
buffer1.put(data);	buffer2.put(data);
buffer2.get();	Buffer1.get();
buffer2.get();	Buffer1.get();

Immaginate di avere due thread che comunicano usando due buffer. Quindi io uso un buffer per spedire dati al thread B e il thread B usa un buffer per spedire i dati a me che sono il thread A. Se questi buffer che utilizziamo sono ad una posizione, se io vado a fare due put, mi blocco fintanto che il thread B non ha fatto la get() ma se anche il thread B fa due put() siamo in una situazione nella quale i due buffer sono pieni, io non posso fare la seconda put() e quindi mi blocco, il thread B non può fare la seconda put() e si blocca e nuovamente siamo in una situazione di stallo e in questa situazione di stallo non c'è una lock da individuare, non c'è un ordine delle lock, non c'è un uso scorretto della wait all'interno della lock. È che sto usando scorrettamente i meccanismi corretti di utilizzo del produttore-consumatore. Situazioni del genere nelle reti si verificano quotidianamente e nelle reti vengono risolte buttando via i pacchetti. Anche nel caso fossimo in un incrocio stradale e tutti e 4 impiegano l'incrocio contemporaneamente con una fila indietro possiamo arrivare facilmente ad una situazione di deadlock.

Yet another Example



Ci sono delle situazioni davvero intricate che sono state modellate con il problema dei filosofi a cena:



Immaginate di avere un certo numero di filosofi che si ritrovano a cena (sono filosofi cinesi quindi mangiano con le bacchette) e per poter mangiare ognuno deve usare due bacchette. Disgraziatamente chi ha apparecchiato la tavola non lo sapeva ed ha messo solo una bacchetta tra un piatto ed un altro per cui ogni filosofo per poter mangiare ha bisogno di acquisire entrambe le bacchette. Il problema qual è? Intanto che fanno i filosofi? Pensano, quando hanno finito di pensare, mangiano però per mangiare hanno bisogno di acquisire le bacchette, se riescono ad ottenere sia le bacchette di destra e di sinistra mangiano altrimenti aspettano che le bacchette si liberino e quando poi le bacchette si liberano iniziano a mangiare. Questo esempio perché è interessante? Se facciamo in passo indietro

Example: two locks

Thread A	Thread B
lock1.acquire();	lock2.acquire();
lock2.acquire();	lock1.acquire();
lock2.release();	lock1.release();
lock1.release();	lock2.release();

Il problema qui stava nell'ordine e come diceva un vostro collega prima si crea un meccanismo annidato e questo concetto di annidamento è quello che abbiamo visto in tutti gli esempi. È bastato dare un ordine per risolvere il problema ma in fin dei conti, anche in questo caso c'è un problema di ordinamento solo in una forma differente e anche in questo caso c'è un problema di annidamento come c'era anche prima. Prendiamo i filosofi a cena, in questo problema l'annidamento non è evitabile. Se io devo prendere le bacchette cosa faccio? Devo prendere la prima bacchetta e poi vado a prendere anche la seconda ma se il mio vicino di sinistra ha già preso la bacchetta di destra e a sua volta sta prendendo la seconda e che a sua volta è stata già presa dal suo vicino, ci troviamo in una situazione di attesa circolare. Per cui ogni filosofo ha preso la sua bacchetta di destra e sono tutti in attesa che si liberi la bacchetta di sinistra. Questo è lo stallo del filosofo e il problema è che non c'è un ordinamento naturale perché la tavola è rotonda quindi non c'è una cosa da fare prima e una cosa da fare dopo, non c'è un filosofo che ha un qualche diritto di poter accedere prima o dopo, sono tutti uguali. È la circolarità delle loro relazioni che è insita nel problema stesso. Questo non vuol dire che non possiamo trovare una soluzione al problema dei filosofi, semplicemente è solo difficile stabilire un ordine che ci aiuta a risolvere il problema.

Quali sono a questo punto le condizioni che ci possono portare ad uno stallo? Sono 4 che sono:

1) Accesso limitato alle risorse e mutua esclusione. Quindi le risorse devono essere utilizzate in mutua esclusione e devono essere illimitate. Se io dispongo di un numero illimitato di risorse non avrò stallo. Se io ho un numero infinito di stampanti posso far stampare tutti i thread che voglio, non ci sarà competizione per quella risorsa e quindi non andranno in stallo. Ovviamente le risorse devono essere usate in mutua esclusione perché se una stessa risorsa può essere utilizzata contemporaneamente tra due o più thread, non ci può essere stallo perché la faccio usare contemporaneamente da tutti i thread e non c'è competizione tra l'assegnamento e quindi non li devo far attendere. Quindi affinché ci sia uno stallo c'è bisogno che le risorse siano limitate e utilizzabili in mutua esclusione.
2) L'altro aspetto è che non siano prerilasciabili perché se le risorse invece sono virtualizzate (e quindi sono prerilasciabili) non ci può essere stallo. Se un altro me la chiede io la riassegno, gliela faccio usare e poi la restituisco al thread precedente come se non fosse successo niente, lui può andare avanti. Quindi sulle risorse prerilasciabili lo stallo non è possibile, non ci sarà stallo sull'utilizzo del processore (mai). Non tutte le risorse sono non prerilasciabili. Una sezione critica non è prerilasciabile. Se io ho acquisito il lock, la mutua esclusione, per eseguire una sezione critica è proprio perché non voglio che sia prerilasciabile. Perché se forzo un prerilascio posso lasciare la risorsa in uno stato inconsistente e poi succede un pasticcio con gli altri thread. Quindi alcune risorse sono intrinsecamente non prerilasciabili.
3) Ci devono essere più richieste indipendenti con un meccanismo di richiesta, acquisizione, rilascio che spesso viene detto "wait while holding". Che vuol dire che se io chiedo una risorsa e questa è occupata mi sospendo però nel sospendermi mantengo occupate tutte le risorse che ho acquisito precedentemente. Se io le rilasciassi tutte non avrei lo stallo. Questa è una cosa normale: se io faccio una malloc e non c'è in questo momento memoria per fare la malloc, non è che cedo tutte le risorse compreso il descrittore del processo. Tengo tutte le risorse che ho già acquisito, aspetto che la memoria si liberi e poi la malloc si completa e vado avanti. Rilasciare tutte le risorse che abbiamo acquisito quando ci sospendiamo è una cosa molto impraticabile. È possibile in casi estremi ma non normalmente.

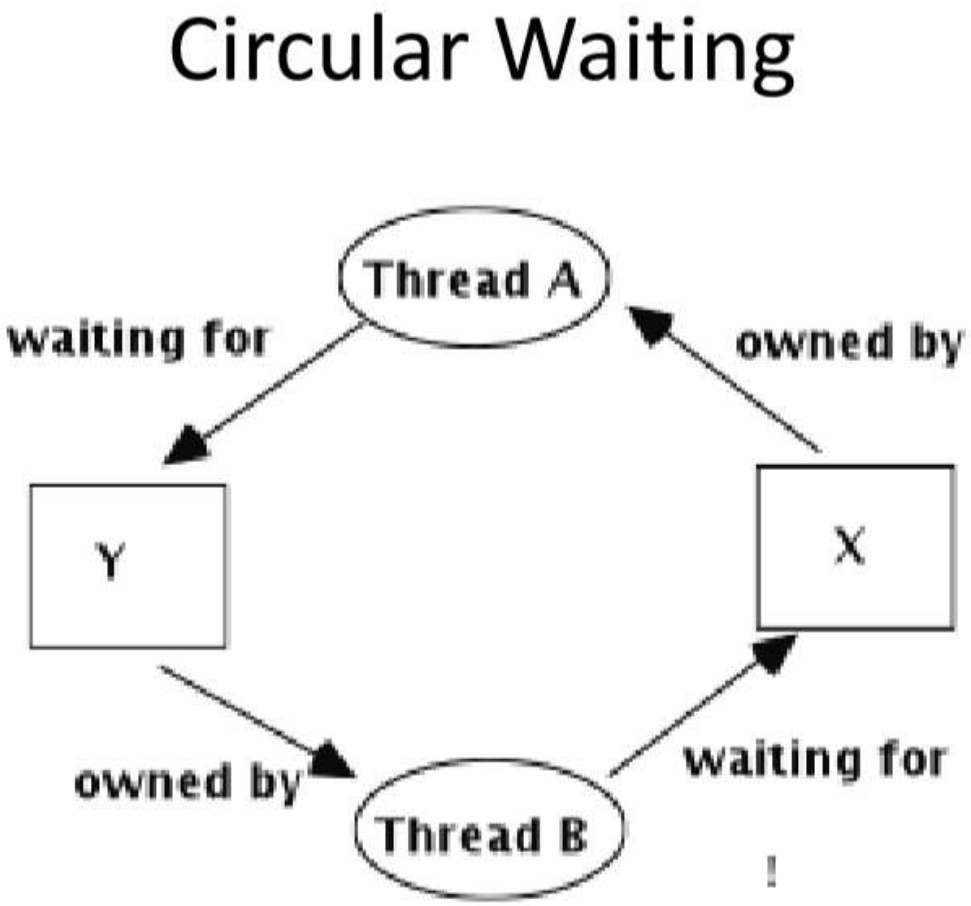
Tutte queste sono condizioni statiche che dipendono da diversi fattori. Questa condizione dipende dal modo con il quale il sistema operativo offre le risorse ai processi. Quindi una scelta precisa di semantica dell'interfaccia che ha il sistema operativo nei confronti dei thread e riflette quella che è la semantica usuale: "richiedo, se è occupato aspetto altrimenti ottengo". È una cosa difficilmente evitabile, è la normalità. Che ci sia prerilascio o non prerilascio dipende dalla natura della risorsa, alcune risorse le posso rendere prerilasciabili come ho fatto il processore, altre risorse sono intrinsecamente non

prerilasciabili quindi è una loro proprietà. La stampante è intrinsecamente non prerilasciabile, la sezione critica è intrinsecamente non prerilasciabile. Il fatto che le risorse vadano usate in mutua esclusione oppure no è una proprietà intrinseca della risorsa, se la risorsa può essere usata contemporaneamente da due thread, è una proprietà che ha lei, non ci posso far niente, non posso agire cambiandone la natura. Tutte queste sono in qualche modo condizioni statiche che dipendono da scelte precise del sistema operativo o della natura delle risorse con il quale abbiamo a che fare. Ma tutte queste condizioni da sole non sono sufficienti perché ci serve un'ulteriore condizione per arrivare allo stallo:

4) Che si sviluppi, nell'esecuzione di più thread concorrenti e nell'accesso di queste risorse, una attesa circolare. Questa condizione di attesa circolare non è nella natura delle risorse, non è nella natura delle scelte procedurali del sistema operativo, è una condizione dipendente dal tempo, può verificarsi oppure no. È completamente fuori controllo ed è difficilmente prevedibile. Un milione di volte non si verifica la milionesima+1 volta si verifica la situazione di attesa circolare.

Tutte queste quattro condizioni sono fondamentali se solo riuscissi a rimuoverne una sola di queste, avrei risolto il problema dello stallo. Non posso cambiare la natura delle risorse quindi sulle prime due è molto difficile agire, se non impossibile. Cambiare questo modello di utilizzo delle risorse è certamente possibile ma costringerebbe poi i programmatori a fare i salti mortali nel loro codice quindi sarebbe molto scomodo usare un metodo del genere e sull'attesa circolare è un altro elemento sul quale è difficile agire però qualcosa in più magari si può dire.

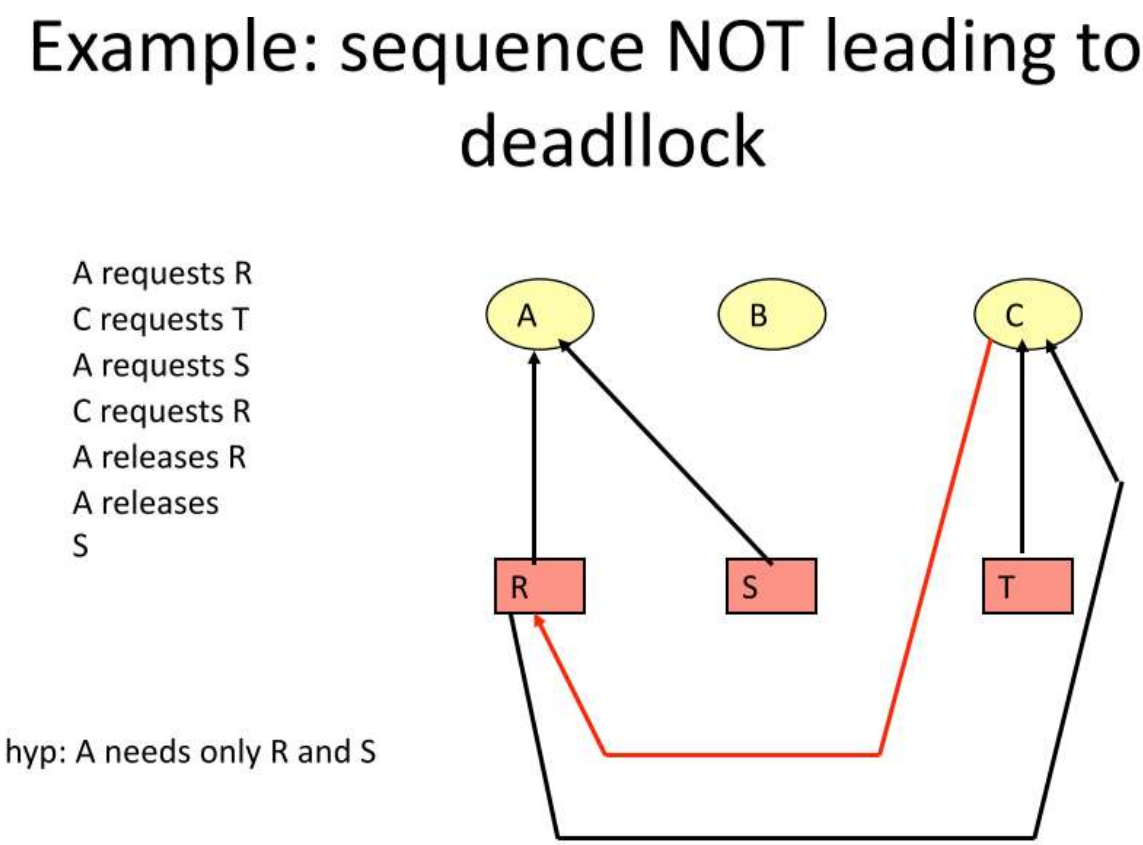
Se voglio affrontare il problema devo prima partire da una rappresentazione, devo poter modellare il sistema per capire cosa sta succedendo. Il modello più semplice si chiama grafo di Holt



Che rappresenta le risorse come quadrati e i thread come cerchi (o ellissi). Per cui se io rappresento lo stato del sistema con questo modello, riesco a rappresentare una situazione di attesa circolare.

PAUSA

Torniamo alla modellazione dello stallo: se noi rappresentiamo i thread con delle circonferenze e le risorse con dei quadrati, l'attesa circolare a questo punto diventa semplice da individuare perché non è altro che un ciclo all'interno di un grafo. In questo grafo rappresento con delle frecce, una freccia orientata da una risorsa verso un thread il fatto che la risorsa è stata acquisita da un certo thread e una freccia orientata da un thread verso una risorsa rappresenta il fatto che il thread è in attesa di acquisire quella risorsa (quindi B ha ottenuto Y e ha richiesto X, quindi è in attesa su X. A ha ottenuto X ed è in attesa di Y) quindi a questo punto la situazione di attesa circolare può essere individuare, agevolmente, andando a cercare un ciclo all'interno di questo grafo. Tutti i thread che sono intrappolati in un ciclo in questo grafo, sono thread in attesa circolare. Immaginiamo di avere tre thread

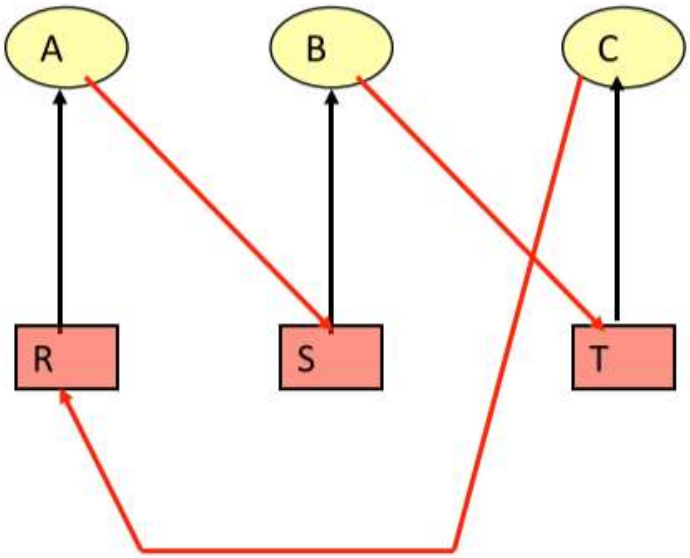


Che devono acquisire le risorse R, S e T. Supponiamo che per qualche motivo A abbia bisogno solo di R e di S, a questo punto A può arrivare per primo e richiedere R, poi può arrivare per primo C e richiede T, poi continua A e richiede S, a quel punto C richiede R e si blocca (quindi in questo momento c'è un blocco ma non ancora uno stallo perché non c'è nessun ciclo e non è detto neanche che si sviluppi) successivamente A potrebbe rilasciare R che a questo punto viene acquisita da C, a questo punto A rilascia S e poi si va avanti e non si è verificata nessuna situazione di stallo però potrebbe capitare una situazione differente.

Example: sequence leading to a deadllock

A requests R
B requests S
C requests T
A requests S
B requests T
C requests R

Deadlock!



Quindi A richiede R, B richiede S, C richiede T, siccome sono tutte e tre libere tutti i thread prendono le risorse che hanno chiesto, a questo punto A richiede S e si blocca, B richiede T e si blocca, C richiede R e si blocca. A questo punto abbiamo una situazione di attesa circolare e quindi uno stallo. Come possiamo risolvere il problema? Il problema è spinosissimo e non è per niente semplice da risolvere. Abbiamo capito che il problema dello stallo nasce da un utilizzo delle risorse singolarmente corretto ma nel complesso potrebbe dare problemi perché non sono gestite correttamente. Questo problema di stallo ci può essere sia in thread che operano con le loro strutture dati quindi noi scriviamo un processo, scriviamo un certo numero di thread e questi thread vanno in stallo tra loro utilizzando le loro strutture. Oppure potremmo avere il problema dello stallo nel sistema operativo: il sistema operativo offre risorse ai thread, le risorse sono limitate, i thread singolarmente richiedono queste risorse correttamente come previsto dal sistema però a causa del fatto che le risorse sono limitate, non prerilasciabili etc… i thread sopra il sistema operativo potrebbero andare in stallo. Quindi il problema si presenta sia a livello di sistema, sia a livello di singolo processo.

A livello di processo come si può intervenire? Certamente il programmatore deve essere ben conscio di quello che fa, adottare delle pratiche di programmazione, analizzare bene il codice e cercare di cavarsela così, sperando di uscire indenne dallo stallo. A livello di sistema operativo non posso affidarmi al programmatore perché il sistema operativo è scritto bene, il problema è quando vado a mettere in esecuzione una serie di thread, questi thread vanno a chiedere risorse in una qualche maniera e vanno in stallo loro. Quindi non dipende dal programmatore del sistema operativo ma dipende dai programmatori delle varie applicazioni che vanno in esecuzione e come li acchiappo quello? Quindi non c'è modo. Servirebbe un meccanismo automatico che mi permetterebbe di risolvere il problema dello stallo alla radice. Avere un supporto automatico che è in grado di risolvere il problema o evitando che i thread vadano in stallo oppure se ci vanno cerco di risolverlo. È di questo che parliamo ora: cercare di capire se esistono dei supporti automatici che può usare il sistema operativo o può usare il supporto a

tempo di esecuzione di un linguaggio per aiutare i programmatori, nel secondo caso, aiutare il sistema operativo automaticamente, nel primo caso, ad evitare o risolvere eventuali problemi legati allo stallo. Ci sono diversi approcci che possono essere catalogati in 3 classi:

1) "Detect and Fix": non mi preoccupo, lascio che i thread vadano avanti, in fin dei conti lo stallo è poco probabile quindi se per caso il loro comportamento può portare a stallo, nella maggior parte dei casi non si verifica mi limito soltanto ad osservare lo stato del sistema e se per caso mi rendo conto che alcuni thread sono andati in stallo allora aggiusto cercando di risolvere a posteriori.
2) Prevenzione statica (static prevention): non adotto nessuna soluzione a tempo di esecuzione come faccio nel "detect and fix" ma agisco su una delle 4 condizioni necessarie per lo stallo (anche più di una nel caso), cerco di rimuoverle in modo tale che lo stallo non si verifiche per una proprietà strutturale del mio sistema. Ad esempio potrei rendere tutte le risorse prerilasciabili (lo posso fare per alcune risorse ma non posso farlo per tutte).
3) Prevenzione dinamica (dynamic prevention): è il più aggressivo di tutti ed utilizza l'algoritmo noto come "algoritmo del banchiere" (banker's algorithm) che agisce in maniera più invasiva. L'algoritmo fa da tramite su tutte le richieste di risorse come se ogni volta che invocassimo una chiamata di sistema che comporta l'uso di una o più risorse, questa richiesta in realtà viene vagliata dall'algoritmo del banchiere e soltanto se questa richiesta ai fini dello stallo è sicura, allora l'algoritmo del banchiere dà l'ok e noi possiamo procedere ad acquisire le risorse ed eseguire la chiamata di sistema. Altrimenti l'algoritmo del banchiere dice no, quindi il thread viene bloccato in attesa che le condizioni di acquisizione di quelle risorse siano tali che non ci sia più stallo.
4) Ci sarebbe un quarto metodo che non è citato nel libro ed è "l'algoritmo dello struzzo" consiste nel mettere la testa sotto la sabbia (fondamentalmente si fa finta che il problema non esista). È l'approccio più utilizzato (su Android, Windows, IOS è largamente utilizzato). È una scelta molto pragmatica e dettata da alcuni fattori: il primo fattore è che i sistemi che noi utilizziamo attualmente hanno una quantità di risorse sproporzionata rispetto all'effettiva necessità che siamo molto vicini ad una disponibilità pressoché infinita di risorse. Lo stallo di conseguenza diventa poco probabile per cui non ha neanche senso porsi il problema. Inoltre la maggior parte dei sistemi che abbiamo a che fare non sono sistemi che operano su dati critici. Se va in stallo il mio computer mentre sto facendo una presentazione basta ammazzare qualche processo, faccio ripartire e dopo qualche minuto è tutto quanto apposto. Quindi non è una situazione critica. La maggior parte di questi sistemi, non operano in situazioni critiche, hanno una sovrabbondanza di risorse, il risultato è che lo stallo è molto poco probabile e che se anche si verifica il danno è talmente basso per cui preoccuparsi sarebbe un inutile dispendio di energie per cui non ci preoccupa. Questo non vuol dire che possiamo fregarci dello stallo sempre e comunque, è che chiaramente le misure, se hanno un costo elevato, in certi casi non vale la pena operare. Ci sono situazioni invece dove il sistema è critico, lavora su dati critici ed è a rischio la vita delle persone o dei soldi, e in questi casi vale la pena farci carico dei costi di risoluzione dello stallo. Per esempio nel caso della mia banca se sapessi che non ci sono dei meccanismi per gestire lo stallo sarei molto preoccupato. Vediamo come funziona la prima soluzione:

Solution #1: Detect and Fix

- Algorithm
  - Scan wait for graph
  - Detect cycles
  - Fix cycles
- How?
  - Remove one thread, reassign its resources
    - Requires exception handling code to be very robust
  - Roll back actions of one thread
    - Databases: all actions are provisional until committed

si - 23 Marzo 2017

L'algoritmo è abbastanza semplice: il mantengo il grafo di Holt. Via via che i thread fanno una richiesta di risorsa, che venga attribuita o che non venga attribuita, ogni richiesta comporta l'aggiornamento del grafo. Per poter applicare questo sistema c'è bisogno che io nel sistema operativo mantenga la struttura dati che rappresenta lo stato di tutti i thread, lo stato di tutte le risorse con le loro relazioni d'uso. Ogni volta che c'ho una chiamata di sistema e questa chiamata di sistema comporta l'allocazione di una risorsa, io aggiorno il grafo. Dopo di che in maniera indipendente da questo (con un meccanismo che potrebbe lavorare in backgroud in un thread separato) potrei avere un thread separato di sistema che analizza questo grafo periodicamente con una certa cadenza e se per caso trova un ciclo segnala il problema al resto del sistema e si cerca una soluzione per i soli thread convolti in questo ciclo. Il rilevamento non è necessario che avvenga in tempo reale, in fine dei conti se un gruppo di thread va in stallo, questi thread sono sospesi perché aspettano qualcosa. La situazione diventa patologica se io non li faccio uscire da questa situazione di stallo, ma se io me ne accorgo dopo 10 secondi, sono stati sospesi solo per 10 secondi e non è un problema reale. Posso anche rilevare lo stallo con un certo ritardo, può non essere drammatico. Il problema non è tanto mantenere aggiornato questo grafo e rilevare lo stallo perché questo in fin dei conti è una cosa che sappiamo fare, si tratta di mantenere una struttura dati a grafo e un algoritmo che visita il grafo e fa una ricerca dei cicli che algoritmicamente parlando è una banalità. I problemi sono di altra natura: il primo problema è che le risorse all'interno del sistema sono davvero uno sproposito perché non sono solo le risorse hardware, qualsiasi variabile all'interno del sistema potrebbe avere i suoi lock, ci possono essere i buffer per comunicare con la rete, insomma qualsiasi struttura all'interno del sistema potenzialmente è una variabile su cui ci può essere stallo. Per cui con la quantità di risorse enorme bisogna avere una catalogazione ben precisa quando si fa una chiamata di sistema per vedere quali sono tutte le risorse che sono effettivamente toccate da quella chiamata di sistema. È un lavoro molto noioso, molto articolato, fattibile ma molto pesante e questo grafo può essere molto grande. Però fin qui siamo nel mondo del dominabile.

Immaginiamo a questo punto di aver rilevato uno stallo, risolviamo dunque il problema ma cosa vuol dire materialmente risolvere uno stallo? Cosa vuol dire fare un fixing? Le soluzioni non sono esaltanti: questi thread se sono in stallo sono in stallo perché stanno aspettando una risorsa, hanno già compiuto del lavoro precedente e devono ancora completare il loro lavoro. Ora potrei decidere di obbligare un thread a rilasciare una risorsa ma se io lo obbligo a rilasciare una risorsa per uscire dallo stallo, in realtà significa che molto probabilmente questa risorsa verrà rilasciata in uno stato inconsistente e quindi è possibile che il risultato dell'elaborazione di uno o più thread non abbia più senso, ha senso farli andare avanti? (domanda retorica).

In maniera più pragmatica potrei uccidere un thread e questo l'abbiamo fatto anche noi quando magari ci trovavamo in un loop infinito e in futuro lo faremo anche perché sono andati in stallo. Una soluzione potrebbe essere appunto questa, tra tanti thread che sono in stallo ne uccido uno, in questo modo libero risorse e gli altri thread possono andare avanti. Sperabilmente la terminazione di questo thread fa uscire dallo stallo tutti gli altri, quindi ne sacrifico uno per salvarne 100. C'è bisogno che il sistema sia davvero molto robusto per poterlo fare perché terminare un thread significa che devo recuperare tutte le risorse che lui aveva acquisito e le devo recuperare in uno stato consistente. Ovviamente il proprietario di quel thread potrebbe non essere molto felice. Quindi in base a che criterio lo scelgo il thread da rimuovere? Oltretutto è possibile anche che il thread che rimuovo non sia quello giusto, potrebbe essere che dopo ne debba rimuovere anche altri. Una tecnica che evita questa soluzione un pò "grossolana" è la tecnica del "roll back". Questa tecnica consiste sostanzialmente in questo: abbiamo una serie di thread, di questi thread periodicamente ne salviamo lo stato per intero e li lasciamo andare avanti. Se per caso questi thread raggiungono uno stallo, allora a quel punto li portiamo ad un punto di ripristino precedente e li facciamo ripartire. Se io riporto indietro dei thread che sono andati in stallo ad un punto precedente, loro è come se tornassero indietro nel tempo e ripartono da quel punto. Ritorneranno in stallo? Non lo possiamo sapere, però siccome lo stallo è poco probabile e per altri motivi che abbiamo discusso anche prima, è poco probabile che io ricada in uno stallo e se per caso mi ritrovo di nuovo in uno stallo riporto di nuovo tutto indietro e lo faccio ripartire. Questa tecnica del roll-back non è una cosa leggera perché significa salvare lo stato di tutti i thread di tutto il sistema ed anche lo stato

del sistema, periodicamente. Riassumendo: la soppressione dei thread è molto grossolano ma ha il vantaggio di essere molto semplice. Posso assegnare le risorse dei thread che ho terminato ad altri thread però devo avere un criterio per selezionare i thread che vado ad uccidere. Ci possono essere vari criteri, qui si adopera quello del minimo danno cioè bisogna trovare il thread la cui soppressione fa il minor danno possibile. Spesso non è facile capirlo e comunque sia questa tecnica non è sempre adottabile. In certe situazioni ogni thread è troppo prezioso o troppo critico per poter semplicemente sopprimerlo. Se mi si verifica una deadlock in una transazione bancaria non posso semplicemente uccidere un thread perché in quel caso il cliente perde tutti i soldi. Spesso lo si utilizza però non è un sistema generalizzabile e va bene per sistemi che sono poco critici.

La roll-back è estremamente costosa, anche questa si utilizza in determinate situazioni e si utilizza in situazioni davvero delicate (nelle transazioni bancarie si utilizza). I sistemi operativi il roll-back lo utilizzano per alcune operazioni molto delicate non tanto per lo stallo ma per evitare eventuali guasti o terminazioni improvvise. Viene utilizzato frequentemente nelle tecniche di logghing delle operazioni di transizioni fatte su file system. I file system moderni (NTFS ma anche quelli di Linux) quando vengono fatte operazioni sui file system in realtà ne tengono traccia, fanno dei log, si segnano le operazioni in corso perché se per caso interviene un crash o un danno, non vogliono che il file system risulti inconsistente, in questo caso sono in grado di ritornare indietro. In contesti molto specifichi quindi viene utilizzato il roll-back non però per il problema dello stallo ma per problemi legati alla consistenza delle strutture critiche del sistema. In generale il roll-back non è una soluzione semplice e nella maggior parte dei casi viene abbandonata. Il roll-back ha un altro problema grosso che lo rende molto complicato perché in realtà c'è il rischio di dover fare un roll-back ricorsivo tornando indietro più e più volte. Immaginate di avere due thread che comunicano mandandosi dei messaggi. Supponiamo che il thread A mandi dei messaggi al thread B, il thread A impiega del tempo per inviare il messaggio al thread B quindi è ancora in volo il messaggio, a questo punto salvo lo stato del thread A e quello del thread B e li mando avanti. Se si verifica uno stallo torno indietro e li faccio ripartire in realtà parto da uno stato in cui il thread A ha mandato il messaggio, il messaggio in realtà si è perso perché era stato mandato prima e il thread B lo ha rimandato in uno stato in cui il messaggio non l'ha ancora ricevuto. Il thread A quindi sa di averlo spedito, il thread B non l'ha ricevuto, quindi li sto ripristinando in uno stato che in realtà è inconsistente con quello che dovrebbe essere perché mi manca il messaggio in volo che non ho potuto salvare nel salvataggio dello stato. Più in generale quando ci sono sistemi distribuiti con comunicazioni in corso con messaggi sulla rete dove i messaggi viaggiano realmente sulla rete (un messaggio sulla rete quando è sul cavo è sul cavo e non lo posso salvare) un roll-back in queste condizioni può costringermi a fare un roll-back ricorsivo ritornando indietro fino a quando non mi rendo conto che sono riuscito a trovare uno stato in cui c'è consistenza anche rispetto ai messaggi in volo. Il problema può diventare estremamente complicato da gestire. Cosa c'è da dire per quanto riguarda la prevenzione del deadlock

Solution #2: Deadlock Prevention

Eliminate one of the four conditions for deadlock

- Lock ordering
  - Always acquire locks in the same order
  - Example: move file from one directory to another
  - Widely used in OS kernels
- Design system to release resources and retry if need to wait
  - No "wait while holding"
  - Example: telephone circuit setup
- Infinite resources?
  - Ex: UNIX reserves a process for the sysadmin to run "kill"
- Acquire all needed resources in advance

La prevenzione statica comporta la rimozione di una delle 4 ipotesi che rendono lo stallo possibile quindi agisce sulla condizione di mutua esclusione, oppure sulla condizione di non prerilasciabilità, oppure sulla condizione di "wait while holding", oppure sulla condizione di attesa circolare. Posso agire solo su una di queste perché se avendo tutte le risorse per esempio utilizzabili senza mutua esclusione, il problema è risolto per tutti non devo fare altro. In problema è che su alcune risorse non posso proprio operare. Quindi se su una risorsa non c'è proprio modo di eliminare la mutua esclusione, non c'è modo punto. In alcuni casi qualcosa si può fare: l'attesa circolare, come abbiamo visto quando si è formato il grafo, dipende dal fatto che l'acquisizione delle risorse avviene secondo un ordine non ben precisato. È un problema di ordine di acquisizione delle risorse. Non è un problema di come noi scriviamo le lock nel codice perché le risorse, come abbiamo visto negli altri esempi, possono essere molto variegate, non necessariamente sono tutte associate ad un lock. Il vero problema è: io individuo le risorse, stabilisco un ordine con il quale queste risorse devono essere acquisite, se tutti i thread acquisiscono le risorse nell'ordine previsto, a questo punto non ho più il problema dello stallo. Se noi prendiamo l'esempio dello stallo che abbiamo visto nell'ora precedente e lo sviluppiamo ponendo il vincolo che tutti i thread devono acquisire prima una risorsa, poi un'altra e poi un'altra e sono tutti obbligati a rispettare questo ordine, ci renderemo conto che lo stallo non si verifica mai. Chi dovrebbe imporre questo ordine? Lo deve imporre il sistema operativo. Il sistema operativo deve stabilire un ordine di priorità per tutte le risorse del sistema ed assicurarsi che se un thread ha acquisito una risorsa 'i', non vada mai ad acquisire la risorsa i-1. Se voglio acquisire la risorsa 100 e poi la risorsa 200, prima acquisisco la risorsa 100 e poi dopo la risorsa 200 non dovrà mai avvenire il contrario. Se il sistema operativo si rende conto che un thread sta facendo l'operazione inversa lo blocca subito. Questo nel sistema operativo per quanto possibile è fatto, non è sempre possibile stabilire un ordine su tutte le risorse. Il sistema operativo lo fa per le sue risorse, non possiamo forzare facilmente le cose per i thread che stanno sopra. Oltretutto questo ordinamento delle risorse sembra gratis ma non lo è. Immaginate che io debba acquisire una stampante ed un file, se io devo prima acquisire la stampante perché devo inizializzare una stampa, poi devo acquisire il file perché devo usare file e stampante contemporaneamente, se il sistema mi dice "no devi fare

al contrario, prima acquisisci il file e poi acquisisci la stampante" io lo faccio pure, ma io sono costretto ad acquisire prima il file quando non mi serve, tenere vincolato il file fino a quando non ho terminato di eseguire il primo lavoro con la stampante e di fatto in questo modo sto limitando altri thread che avrebbero potuto nel frattempo lavorare con altri file. Imporre un ordine significa forzare dei thread ad acquisire delle risorse in anticipo rispetto a quando effettivamente gli serve e questo significa che posso limitare altri thread. Questo ha l'effetto di limitare il grado di parallelismo del sistema ed ha un effetto indiretto sulle prestazioni. Ci saranno più thread che andranno in attesa anche se in realtà non avrebbero strettamente necessità. In molti casi, dov'è possibile, è usato ma in altri casi non è possibile usarlo.

Immaginiamo invece di rilasciare la condizione "wait while holding", supponiamo che il sistema ci dica "ok se tu vuoi acquisire una risorsa me lo chiedi con una chiamata di sistema va bene ma se per caso quella risorsa è occupata tu certamente ti devi sospendere ma devi rilasciare tutto quello che avevi acquisito prima". È come dire che se quella mattina devo prendere la macchina ma quella mattina la macchina l'ha già presa la mia moglie io devo rilasciare tutto: casa, conto in banca, scarpe, vestiti. Esiste un mondo nel quale tutto questo meccanismo si utilizza e si utilizzava nei vecchi metodi telefonici. Come funzionava? In quei sistemi telefonici la comunicazione avveniva a commutazione di circuito, questo vuol dire che erano presenti nelle centraline telefoniche una serie di switch fisici per cui si creava un collegamento fisico tra un telefono ed un altro. Avere un sistema del genere voleva dire che un determinato switch era impiegato per il nostro collegamento e non poteva essere utilizzato da altri. Quando volevamo fare una telefonata cosa succedeva? Che la telefonata partiva dal nostro telefono, iniziava a vincolare alcuni switch per farci arrivare progressivamente da centrale a centrale fino ad arrivare al punto di destinazione e questi ovviamente venivano acquisiti con una certa latenza. Che succede se in questa sequenza di acquisizioni trovo la linea occupata? Io ho già acquisito gli switch che stanno nel mezzo, che faccio li tengo bloccati? Ma se io li tengo bloccati qualcun altro che sta facendo una telefonata li potrebbe trovare bloccati e quindi si blocca a sua volta, si crea un effetto a catena per cui si blocca l'intero sistema. Se faccio una telefonata e trovo occupato in realtà tutti gli switch che io ho acquisito per fare la mia telefonata vengono rilasciati e se io voglio chiamare devo riacquisirli tutti in un momento successivo. Questo meccanismo quindi è stato utilizzato in certi casi e potrebbe anche essere usato un domani certamente ha i suoi svantaggi.

Se un thread che vuole fare una certa operazione e che gli servono tutta una serie di risorse, le potrebbe acquisire in anticipo, quindi è come se dicessimo "quando creo il thread, dichiaro subito di quanta memoria avrà bisogno, di quante risorse hardware avrà bisogno, di quante risorse software avrà bisogno, quante e quali sezioni critiche dovrà utilizzare" le fornisco subito tutte al momento della creazione, quindi quel thread non si dovrà mai sospendere, ha già tutto quello che gli serve e andrà avanti fino alla fine. Questo però nuovamente è molto scomodo per il programmatore perché ci obbliga a sapere in anticipo cosa verrà utilizzato da quel thread.

Risorse infinite: questo in realtà è plausibile e in molti casi lo facciamo. Se noi pensiamo all'esempio del processore, il processore è una risorsa hardware fisica, unica che viene richiesta da più thread. Il problema dell'utilizzo del processore è stato risolto virtualizzandolo quindi da un processore fisico ne abbiamo creati tanti virtuali, mettendo in piedi un meccanismo di time sharing e di commutazione di contesto, per cui quella risorsa l'abbiamo virtualizzata e di fatto abbiamo eliminato il problema dello stallo sull'acquisizione del processore. Risorse infinite in certi casi si può fare, non viene fatto solo per il processore ma viene fatto anche per altre risorse come per esempio le stampanti. La stampante fisica è una però nel sistema operativo viene generata quella che si chiama "spool di stampa". Quando un thread vuole stampare non gli viene data la stampante fisica, gli viene data una stampante virtuale che non è altro che un file sul quale il thread va a stampare. In questo modo i thread non acquisiscono la stampante fisica ma acquisiscono una stampante virtuale e ne posso creare quante me ne pare. Per cui la stampante viene virtualizzata e su questa non ho più problemi di stallo. Non posso però fare lo spool di qualsiasi risorsa, non posso fare lo spool di una sezione critica. Il caso della stampante è tra l'altro emblematico perché io virtualizzo la stampante creando tante

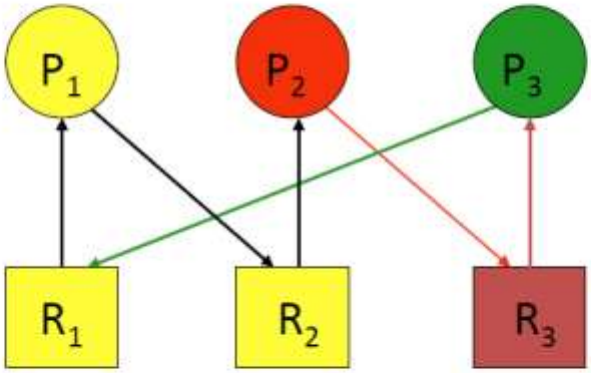
stampanti virtuali tramite lo spool. Quello che sto facendo è spostare il problema, lo stallo non si verifica più sulla stampante fisica ma si può verificare sullo spool perché di fatto cosa sto dicendo? I thread possono stampare su file ma se lo spazio fisico per lo spool si va ad esaurire, potenzialmente potrei avere uno stallo sullo spool. La cosa è abbastanza improbabile perché il disco è enorme, ha una capienza elevata, normalmente lavoro con dischi la cui percentuale di spazio libero è abbondante per cui di fatto il problema lo risolvo "affogandolo" in un mare di risorse. Il problema della mutua esclusione delle risorse infinite in alcuni casi può essere affrontato creando dei meccanismi di virtualizzazione (lo spool è uno di questi) ma non è utilizzabile sempre. Nella prevenzione statica quindi posso fare tante cose, non sempre, ma comunque non ho una risposta definitiva al problema. Alcuni problemi li posso risolvere, altri non li posso risolvere e restano aperti. Certamente adoperando questi metodi posso abbassare la probabilità di eventuali stalli.

Se io impongo un ordinamento nell'acquisizione delle risorse, se prendiamo l'esempio precedente in cui le risorse devono essere acquisite in modo ordinato, succede che

Solution #2: Deadlock Prevention

Lock ordering

- Requests to resources must be ordered



R3: Resource with maximum index

P_3 violates the constraint of sorted requests

P_2 causes circular waiting

P1 acquisisce R1, P1 acquisisce R2, P1 può richiedere R2 perché R2 viene dopo R1 nell'ordine di acquisizione, P3 ha richiesto R3 lo può fare se non ha proprio bisogno di R1 e R2 ma bisogna tener presente che avendo richiesto R3, P3 non potrà andare a richiedere nessuna precedente. Potrà richiedere risorse di indice maggiore. Nel momento nel quale si va a verificare lo stallo (non si è ancora verificato lo stallo perché P2 non ha ancora fatto richiesta per R3) la richiesta che è critica per lo stallo è quella nella quale P3 va a richiedere una risorsa di indice minore dopo che ne ha chiesta una di indice maggiore. In questo punto, che ancora non c'è lo stallo, avviene l'operazione che ci può portare ad una situazione di stallo ed è questa che deve essere proibita e la possiamo proibire se imponiamo l'ordimento nell'acquisizione delle risorse perché questa verrebbe impedita. Lo stallo materialmente si verifica qui (linea rossa che collega P2 a R3) ma P2 può richiedere R3 dopo aver chiesto R2 perché P2 sta rispettando l'ordine. P1 e P2 hanno agito correttamente,

P2 è quello che agendo correttamente scatena lo stallo ma la vera causa è la richiesta illecita di P3 che ha violato l'ordinamento.

Riassumendo abbiamo diverse condizioni: se vogliamo agire sulla condizione legata alle risorse infinite e alla muta esclusione possiamo adottare metodi di spooting e in generale di virtualizzazione e per certe risorse lo possiamo fare. Se vogliamo agire sulle tecniche di "wait while holding" possiamo richiedere tutte le risorse in anticipo, questo è molto scomodo però in certi casi lo possiamo fare, in casi un po' limitati. Se voglio agire sulla condizione di attesa circolare devo imporre un ordine nell'acquisizione di risorse.

FINE

Sistemi operativi 27/03/17

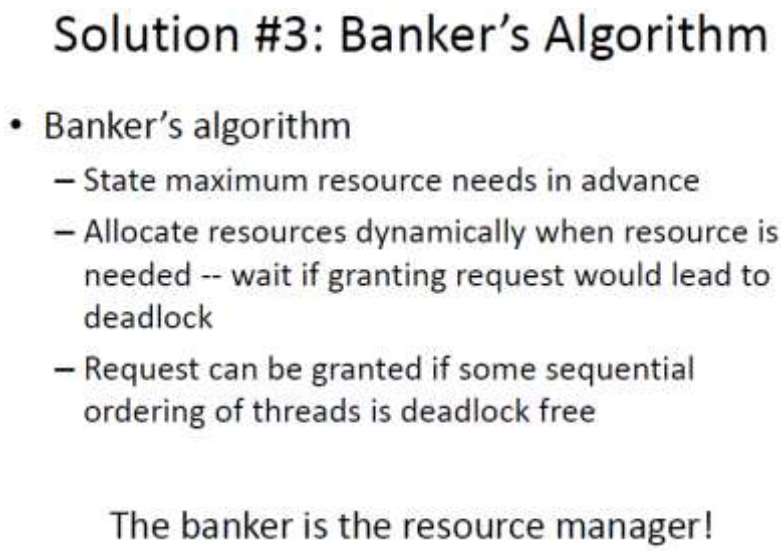
Avevamo iniziato a vedere il problema dello stallo, avevamo visto le tecniche di prevenzione statica che consistevano nel prevenire lo stallo, impedendo che una delle 4 condizioni necessarie per lo stallo si potesse verificare. Questi meccanismi di protezione statica, abbiamo visto, che in alcuni casi si possono utilizzare (e vengono poi effettivamente utilizzati), però non danno una garanzia totale, nel senso che: aggiungendo in maniera drastica certamente sì, potrebbero darla, ma ci sono dei casi in cui non è consigliabile intervenire. Per cui comunque restano delle risorse che sono scoperte, sulle quali non possiamo o non vogliamo applicare la prevenzione statica. Quindi resta l'ultima alternativa: è quella della prevenzione dinamica. Prima di introdurre la prevenzione dinamica c'è un aspetto da aggiungere: tutto quello che abbiam visto fino ad ora, soprattutto per quanto riguarda la rilevazione, presupponeva che nel sistema fossero presenti un certo numero di risorse, ma sostanzialmente con molteplicità 1. In realtà la cosa non è sempre così, alcune risorse possono essere presenti con una molteplicità maggiore di 1. Questo vuol dire che di un certo tipo di risorsa possiamo averne diverse istanze; per cui in realtà la competizione è un po' più articolata. Quindi in un contesto di questo tipo, dove di una risorsa sono presenti più istanze non possiamo più usare il grafo di Hault per rappresentare lo stato di allocazione delle risorse, ma ci serve una rappresentazione leggermente diversa, che è quella che utilizziamo adesso per vedere la prevenzione dinamica.

Multiplicity of resources

- Resources classified according to their type
  - Multiplicity M:
    - M is the number of resources of a given type
  - Availability D:
    - Number of resources of a given kind that are currently available
- Method of request:
  1. Single resource
  2. Multiple resources
     - A process requests k resources of a given kind
     - If k ≤ D then all k resources are assigned
     - If k > D then no resource is assigned (and the process waits)

Quindi in generale noi possiamo classificare le risorse secondo il loro tipo e per tipo di risorsa definiamo molteplicità e disponibilità. Immaginatevi in un sistema di avere tre stampanti, se qualcuno vuol stampare, per lui in effetti può essere indifferente una stampante o l'altra. Quindi il sistema almeno in linea di principio può assegnargli uno a qualsiasi stampante. Quindi il tipo di risorsa è stampante e la molteplicità in questo caso è 3. Ora, se abbiamo tre stampanti nel sistema può darsi che una sia stata già allocata a qualche processo e due siano libere. Quindi ci servono a tenere molteplicità e disponibilità. La disponibilità ci dice quante risorse di quel tipo in questo momento sono effettivamente libere, disponibili. Se non vi piace l'esempio delle stampanti pensate alla memoria: noi abbiamo una certa quantità di memoria RAM, per assegnare questa memoria RAM ai processi, dal loro punto di vista è indifferente che gli venga assegnato un blocco di RAM o un altro blocco di RAM, tutta la memoria è uguale. Quindi la RAM è una risorsa che è presente con molteplicità molto maggiore di 1 e a seconda di come è gestita può essere allocata in blocchi o a byte, questo è indifferente. Quindi anche per la memoria è importante sapere, oltre alla molteplicità, sapere anche quanta è effettivamente allocata e quanto è libera e quindi, anche in questo caso, conoscerne la disponibilità. Un'altra cosa da chiarire: il metodo di allocazione delle risorse. Fino ad ora abbiamo supposto, per semplicità, che le risorse venissero richieste in un' unica istanza e che venissero assegnate in un'unica istanza. In realtà è anche possibile far richiesta un'unica richiesta che chiede più risorse e quindi di conseguenza fare l'assegnamento su più risorse. Di nuovo: pensate alla memoria. Se viene fatta una Fork io devo allocare lo spazio per un nuovo processo, questo processo ha bisogno di spazio

in memoria principale. La richiesta di allocazione conseguente alla Fork è una richiesta di allocazione di risorse multiple, perchè andrò ad allocare un Mega di RAM, due Mega un Giga o quello che serve. Allora in questo caso come funziona normalmente il meccanismo: noi facciamo la richiesta di k risorse di un dato tipo, se tutte quelle k risorse sono disponibili, allora le assegno tutte e k in un' unica azione. Altrimenti se la richiesta è superiore alla disponibilità questa richiesta non è, diciamo, esaudita in questa fase e il processo che ha fatto la richiesta si sospende in attesa che questa appunto possa essere soddisfatta. Una nota: in questa parte del corso, quindi relativamente alle risorse, parlo di processi e non di thread, perchè in effetti i thread non dispongono di risorse proprie, ma dispongono delle risorse che sono state allocate al processo del quale loro fanno parte. Quindi parlare in questo caso di processi è più generale, perchè i processi sono le scatole che contengono le risorse. Quindi, tornando al caso delle risorse multiple io posso richiedere un Mega di memoria libera (principale): se nel sistema è presente più di un Mega allora tutto questo MB di memoria è assegnato al processo, altrimenti il processo resta in attesa e quando viene liberato almeno quello spazio di memoria il processo viene riattivato e gli viene dato l'uso, gli viene assegnato. Allora detto questo vediamo, a questo punto la soluzione dell'Algoritmo del Banchiere (AB) per la prevenzione dinamica. Allora, intanto ci sono alcune premesse da fare.

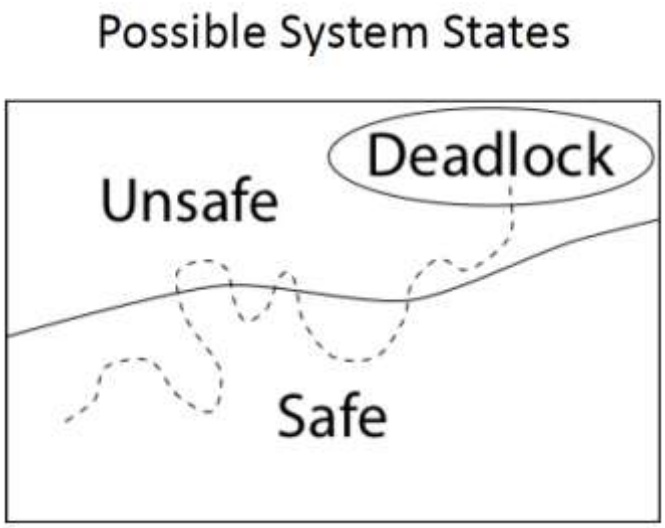


La prima premessa è che utilizzare una soluzione di questo tipo impone ai processi di dichiarare in anticipo le loro necessità. E' un po' come quando fate una fork e la fork deve allocare diverse cose: dovrà allocare il Descrittore di Processo, dovrà allocare memoria, dovrà andare ad acquisire l'uso di certi file per poter prendere le informazioni che (???) i file (???) la fork. Ma insomma dovrà allocare tutta una serie di risorse. Ecco nel momento nel quale viene fatta questa operazione bisogna dichiarare in anticipo quali sono le risorse delle quali questo processo avrà bisogno, tutte quante. Ora, come si fa a sapere quale è il numero massimo di risorse delle quali un processo ha bisogno e di saperlo in anticipo. Il sistema non lo può calcolare da solo. Anche avendo a disposizione il codice che verrà eseguito successivamente con la Exec, non è impossibile staticamente riuscire a sapere quante risorse utilizzerà quel processo. Quindi questa è una cosa, in realtà, che deve essere dichiarato dal programmatore. Quando il programmatore scrive il codice, lo scrive pensando a certe funzionalità, di conseguenza stima le risorse che gli servono e queste vanno dichiarate, presumibilmente con le direttive del compilatore. Ora, vi faccio notare che una cosa del genere non l'avete mai fatta e mai la farete, ne discutiamo dopo. Quindi, in qualche maniera, questa informazione non è facile da ottenere ed obbliga il programmatore a fare uno sforzo aggiuntivo e dare questa informazione. Quindi quando il processo viene messo in esecuzione questa informazione è disponibile e viene utilizzata dall'Algoritmo del Banchiere. Allora l'idea dell'algoritmo è questa: si allocano le risorse dinamicamente, quindi su richiesta, quindi non c'è una allocazione preventiva di tutte le risorse necessarie, ma le risorse vengono allocate quando vengono richieste. Però stavolta, con un po' di accortezza, quindi se ci rendiamo conto che ci sono delle richieste che potrebbero portarci ad una situazione pericolosa dal punto di vista dello stallo (non necessariamente in stallo), ma anche in una

situazione pericolosa ai fini dello stallo (dopo vedremo cosa significa), allora questa richiesta non viene esaudita e il processo che ha fatto questa richiesta resta sospeso, fintanto che non si verificano le condizioni necessarie affinchè questa richiesta sia sicura ai fini dello stallo e quindi possa essere (???). Quindi tutto il problema si riduce a questo, se un processo mi richiede l'allocazione di una risorsa, come faccio a capire se quella richiesta di allocazione è sicura o insicura ai fini dello stallo? Badate bene, qui non sto andando a verificare se quella richiesta mi porta in stallo, che è un'informazione differente. Voglio sapere se quella richiesta la posso esaudire e dal punto di vista dello stallo sono tranquillo. Ora, in effetti questo algoritmo di chiama Algoritmo del Banchiere, perchè questa cosa i banchieri la fanno da quando esistono, quindi è un algoritmo che in realtà si ispira ad un altro mondo. (10.39)
Qual è in realtà in problema del banchiere nel contesto della banca? Il banchiere ha una certa quantità di risorse (nel suo caso sono soldi). E ci sono persone che vengono a chiedere dei prestiti per finanziare le loro imprese. Questi prestiti poi verranno restituiti quando le imprese avranno successo. Però può capitare che una impresa faccia richiesta di una certa quantità di soldi: ci sono certi aspetti che portano alla necessità di avere ulteriori investimenti per evitare che tutta l'impresa fallisca. Quindi in questo caso il banchiere cosa deve fare: lui sa che se non fa il secondo prestito quella impresa fallisce e quindi non gli restituirà neanche i soldi del primo prestito, quindi lui deve valutare se ha senso continuare a finanziare nella speranza di avere poi indietro tutto quanto, ovviamente nel suo caso con gli interessi. Quindi il banchiere si deve porre questo problema, anche perchè lui deve fare i conti sulla disponibilità dei soldi che effettivamente ha. Quindi lui deve pensare, va bene, "*Tu mi stai chiedendo, in questo momento, degli altri soldi per rifinanziare la tua impresa, perchè altrimenti rischieresti di fallire. Ma potrebbe venire qualcun altro subito dopo di te che mi chiede degli altri soldi per finanziare o per rifinanziare la sua impresa. E se io presto soldi a tutti quanti, senza stare attento a non fallire io, appunto, rischio di fallire*"; e questo è il suo Deadlock. Quindi la situazione è molto simile ed è un po' quella che abbiamo in questo caso. Il sistema operativo presta le risorse ai vari processi, i processi continuano a chiedere delle ulteriori risorse per poter andare avanti e poter completare il loro lavoro, con la promessa che se il sistema presta loro tutte le risorse che loro chiedono, loro termineranno e potranno restituire il tutto. Ovviamente nel sistema operativo non ci sono interessi, non facciamo pagare interessi ai processi. Però possono restituire il tutto. Quindi il sistema operativo deve prestare le risorse assicurandosi di poter portare tutti a terminazione, in maniera tale da riavere indietro le risorse. Il rischio che corre è che se non sta attento, appunto, manda il sistema in Deadlock. Quindi come il banchiere gestisce le sue risorse finanziarie, l'AB nel sistema operativo deve gestire le risorse, prestandole con accuratezza. Ora, c'è una piccola differenza tra il banchiere che lavora in banca e il banchiere nel nostro sistema.
Perchè se il banchiere nel nostro sistema non può, in questo momento, prestare alcune risorse ad un processo perchè rischia, appunto, di andare in una situazione pericolosa per lo stallo, quel processo resta bloccato in attesa fintanto che non ci sono le condizioni per attuare il prestito. Nella realtà, se la banca non presta i soldi, quell'impresa fallisce, questo è un altro discorso. Nel nostro caso i processi non falliscono se non viene esaudita la loro richiesta, ma restano in attesa che questa possa essere esaudita. Allora, ai fini di questo gestore delle risorse, quindi, gli stati che ci interessano sono tre:

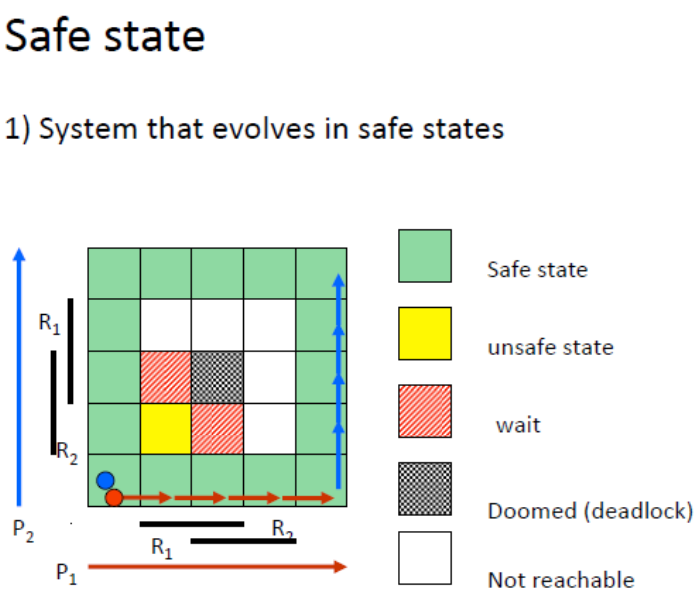
- lo stato di Deadlock,
- uno stato Insicuro
- e lo stato Sicuro.

Ora, intuitivamente che differenza c'è.



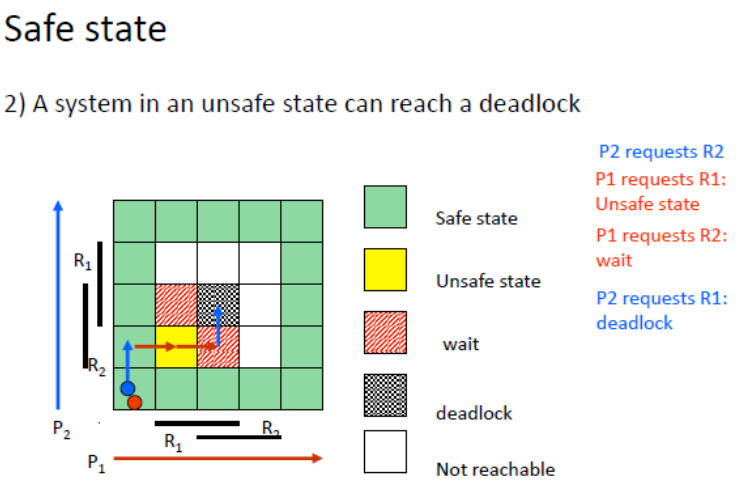
Vi faccio notare che i processi nella loro globalità definiscono lo stato Sicuro, Insicuro, Deadlock; quindi questo non è lo stato di un singolo processo, ma rappresenta lo stato dell'intero sistema. Questo stato oscilla, quindi, in realtà, SAFE, non è, se volete, uno stato singolo, ma è un insieme di stati che hanno la proprietà di essere sicuri e lo stato del sistema si muove, percorre una traiettoria tra questi sottostati sicuri. Ora se noi non usiamo l'algoritmo del banchiere e quindi non facciamo nessun controllo sul modo col quale le risorse vengono allocate, il percorso, la traiettoria che lo stato del sistema può seguire può oscillare tra stato Sicuro, Insicuro e poi eventualmente può anche arrivare al Deadlock. Allora, la cosa importante da tenere a mente è che finanto che il sistema è nello stato Sicuro ed Insicuro, ancora non è Deadlock, perchè i processi non sono ancora sospesi in attesa circolare. Nello stato Insicuro i processi continuano a svolgere del lavoro, possono richiedere l'allocazione di ulteriori risorse o possono deallocare delle risorse. Quindi ai fini dello stallo, lo stato Insicuro non è stallo, i processi sono ancora attivi. La differenza quale è? La differenza è che se io permetto al sistema di andare in uno stato Insicuro, può darsi, che a quel punto, una serie di richieste particolarmente sfortunata, da parte dei processi, mi conduca allo stallo e io non posso più farci niente a quel punto. Perchè ho già impegnato e distribuito troppo le mie risorse e non sono più in grado di far fronte a certe sequenze di richieste che porterebbero automaticamente il sistema in stallo. Invece se mantengo il sistema in uno stato sicuro, in qualche modo io mi sto tenendo una riserva, idealmente, per cui sono in grado di far terminare tutti i processi senza portarli in uno stato insicuro e quindi senza farli arrivare allo stallo. Lo stato di stallo invece è quello che conosciamo, uno stato di attesa dove i processi non si risvegliano più. Definiamo lo stato Sicuro: è Sicuro se, per ogni possibile futura sequenza di richieste, l'AB può individuare quali richieste di queste vanno soddisfatte e quali no, ai fini di far terminare tutti i processi con successo, ovviamente. Quindi lo stato è Sicuro se, per ogni possibile sequenza di richiesta di allocazione di risorse futura, l'AB è in grado di soddisfarne alcune prima, alcune dopo secondo un proprio criterio, in maniera tale da portare tutti i processi a terminazione. Questo vuol dire che, e questo è molto importante, in alcune circostanze, può darsi, che qualche processo faccia richiesta di una risorsa, quella risorsa è disponibile, ma l'AB non gliela segna subito e lo tiene sospeso, perchè assegnarla potrebbe condurre il sistema in uno stato Insicuro e quindi è un potenziale stallo. Quindi vuol dire che, in pratica, stiamo introducendo delle limitazioni ai processi non esplicite perchè dipende dall'evoluzione dinamica del sistema; la differenza è tutta lì. Se non avessi l'AB quando c'è una risorsa libera e qualcuno me la chiede io gliela do sempre. Con l'AB in qualche caso, ma davvero in qualche caso, potrei non assegnare una risorsa libera a qualcuno che me la chiede, perchè quella risorsa mi potrebbe servire per assicurarmi che qualche altro processo possa terminare e se io la do rischio di non fare terminare né lui né quell'altro che me la sta chiedendo ora. Lo stato invece è Insicuro se, a partire da questo stato, esistono delle sequenze di richiesta che portano il sistema ad uno stallo. E se queste sequenze si presentano, a quel punto, l'AB non può più farci niente. Infine lo stato è Doomed, se tutte le possibili computazioni portano allo stallo. Questo stato di Doomed, Dannato, non è mostrato nello schema precedente, ma ve lo potete immaginare come una palla che è inclusa nello stato Insicuro e che racchiude lo stallo. Il punto quale è?

Dallo stato Insicuro si può ritornare nello stato Sicuro se siamo fortunati. Se per caso qualche processo rilascia anticipatamente alcune risorse, ovviamente non possiamo sapere che faranno i processi in futuro, noi ci aspettiamo che richiedano altre risorse, ma magari le cose cambiano, qualche processo rilascia in anticipo delle risorse, perchè a quel punto ha cambiato idea, non gli servono più e questo ci può portare in uno stato Sicuro. Invece lo stato è Dannato, Doomed appunto, se invece a quel punto non siamo ancora in stallo, però qualsiasi cosa succeda, qualsiasi richiesta di allocazione delle risorse, certamente ci portano allo stallo. Vediamo di farvelo capire con questo schema:
immaginiamo di avere due processo P1 e P2. Allora, i due processi si comportano in questa maniera: P1 fa una certa computazione che mostro con una certa freccia che si sviluppa da sinistra a destra, quindi



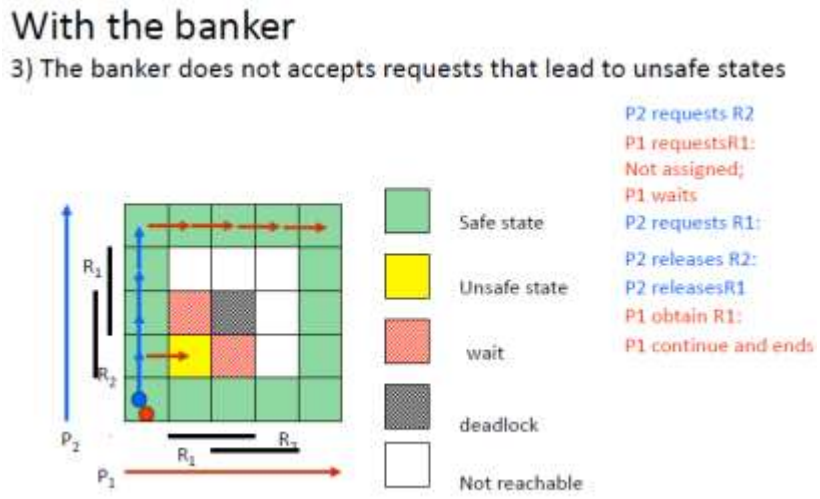
orizzontalmente; nel corso della sua esistenza fa questo: inizialmente fa una sua elaborazione per i fatti propri, poi acquisisce la risorsa R1, poi acquisisce la risorsa R2, poi rilascia la risorsa R1 e detiene solo R2. Infine rilascia anche R2 e poi va a terminazione. Abbiamo un altro processo P2 che invece rappresento con una evoluzione temporale con una freccia verso l'alto, quindi verticale, che invece ha un comportamento opposto: fa qualche cosa per i fatti suoi, poi acquisisce R2, poi acquisisce R1, poi rilascia R2, poi rilascia R1 e poi va a terminazione. Ho chiaramente preso, per farvi questo esempio, un caso di due processi che hanno il codice scritto in maniera tale da poter portare, in qualche caso lo stallo: se il codice fosse stato scritto in maniera tale da non portare mai allo stallo, questo esempio non avrebbe avuto senso. Tenete presente che qui non stiamo usando tecniche di prevenzione statica, quindi permettiamo ai processi di essere scritti come pare a loro. Il problema ce lo stiamo ponendo dentro l'AB quindi questo comportamento dei processi lo possiamo permettere, perchè è il banchiere che deve gestirlo, non è la prevenzione statica. Ora, che cosa può succedere quindi? I processi, nella loro evoluzione possono attraversare diversi stati; per esempio, in questo stato, P2 sta svolgendo del compito ma non ha ancora acquisito nessuna risorsa. Mentre invece P1 ha acquisito R1. Se noi siamo in questo stato, P1 ha già utilizzato R1 e R2 e le ha liberate, quindi non ha niente e sta andando a terminazione, mentre invece P2 tiene impegnate sia R1 che R2. Allora, entrambi i processi partono dallo stato iniziale, quindi, se volete, da quest'angolo iniziano a muoversi disegnando delle traiettorie che attraversano questi riquadri. Ora, non tutti i riquadri sono uguali: per esempio, se noi prendiamo questa situazione, in questo riquadro abbiamo che P1 ha acquisito R1, P2 ha acquisito R2 e da quello che sappiamo questo ancora non è stallo, perchè P1 e P2 non sono in attesa, però questo è certamente uno stato Insicuro, perchè se P1 va avanti a fare quello che ha previsto di fare, quindi di chiedere R2, P2 va avanti a fare quello che ha previsto di fare, quindi richiedere R1, a quel punto andranno prima in questo stato oppure prima in quest'altro a seconda di chi fa prima cosa, quindi uno dei due inizia ad aspettare, poi l'altro porterà il sistema in stallo, che è appunto questo stato. Questi stati bianchi sono non raggiungibili, perchè chiaramente, il tempo avanza sempre o verso destra o verso l'alto, quindi una volta raggiunto questo stato, su questi qui non ci posso più andare.

Da qui non posso salire verso l'alto perchè questo stato P1 detiene R2, quindi se P2 va a richiedere R2 viene bloccato, qui c'è una attesa perchè R2 è impegnato. Qui, di fatto, tutta questa zona bianca non è raggiungibile. Gli stati sicuri ai fini dello stallo sono gli stati verdi, gli unici stati nei quali devo mantenere il sistema se applico l'AB. Vediamo che cosa può succedere. Questo è il primo caso nel quale il sistema evolve in uno stato sicuro, quindi evolve prima P1, arriva quasi a terminazione o a terminazione e poi evolve P2.



Oppure potrei avere un sistema che va in uno stallo, transitato per lo stato insicuro: quindi potrei avere che P2 richiede R2, poi P1 richiede R1, se non esercito alcun controllo e assegno R1 a P1 a questo punto vado nello stato Insicuro. Per esempio, poi, P1 richiede R2 e va in attesa P1, a quel punto P2 richiede R1 e va in attesa anche lui e finiscono entrambi in stallo. Il momento nel quale deve agire il sistema di allocazione delle risorse per prevenire lo stallo in maniera dinamica, con l'AB è in questo punto. Quindi questa assegnazione non deve essere permessa. Vi faccio notare, lo dicevo prima ma ve lo ribadisco, se io quando P1 mi chiede R1 in questo punto non gliel'assegno, sto facendo una cosa diversa da quello che facevo prima perchè in questo istante R1 è libera, quindi in teoria glielo potrei assegnare, ma non glielo assegno, quindi impedisco questo assegnamento e metto P1 in attesa, perchè assegnare R1 a P1 in questo caso comporta la transizione da stato sicuro a insicuro e quindi mi può portare ad uno stallo. Non è detto che lo stallo avvenga, non è detto, perchè questo esempio ovviamente è troppo semplice per poterlo valutare, però in realtà i processi potrebbero prendere delle decisioni nell'utilizzo delle risorse che possono cambiare sulla base dell'input, sulla base di tanti fattori. Quindi ad un certo punto P1 potrebbe cambiare idea e non utilizzare R2: quindi in questo caso uscirei da uno stato insicuro e ritornerei in uno stato sicuro. Il problema è che in questa fase io non posso fare previsioni su quello che farà P1. Quindi se P1, quando sono qui, e mi richiede R1 io so poi che avrà bisogno di R2 e so che se lui manterrà fede alle sue promesse a quello che ha

dichiarato di voler usare ed effettivamente vi chiederà tutte le risorse che ha dichiarato allora io rischio di



andare in stallo. Quindi, quando avviene questa richiesta, P1 viene sospeso, R1 non viene assegnato e il sistema continua ad avanzare, a questo punto, nell'altro modo possibile, quindi avanza P2 e quando P2, a questo punto, libera R1, soltanto a questo punto, mi rendo conto che assegnare R1 a P1 non comporta la transizione nello stato insicuro, ma comporta una transizione sicura, quindi a questo punto attivo, risveglio P1, gli assegno R1 e P1 può andare avanti. Quindi applicare l'Algoritmo del Banchiere comporta, in realtà, una penalizzazione piccola di prestazioni, perchè, in certi casi, devo sospendere i processi anticipatamente. Quindi l'Algoritmo del Banchiere come funziona? Alloca le risorse soltanto se lo stato che risulta dall'allocazione delle risorse è uno stato sicuro. Lavorando in questo modo, l'AB permette di poter portare avanti dei thread anche se le loro risorse necessarie complessive sono superiori alle risorse effettivamente disponibili. Quindi può darsi che io faccia terminare felicemente 5 processi ognuno dei quali ha bisogno di un Giga di memoria, pur avendo 3 Giga a disposizione. Lo posso fare perchè è vero che la loro necessità complessiva è superiore alla mia effettiva disponibilità, ma l'AB è in grado di distribuire la loro esigenza effettiva nel tempo in maniera tale che non superi mai, in un dato istante, l'effettiva disponibilità. Quindi, sostanzialmente, un thread, un processo può andare avanti se il numero delle risorse libere resta comunque >= del massimo numero di risorse che mi servono per poter far terminare i processi, i thread. Detto questo, come è organizzata la cosa?

Banker's algorithm for resources of the same type

- First each process declares the number of resources it needs
- At a request of process P the banker checks that the resource assignment keeps a safe state. To this purpose:
  - Considers the state S reached if request is granted
  - For each process computes the residual requirement R (the number of resources it still needs)
  - Sorts the processes for an increasing value of R
  - For each process Q not already marked:
    - Check if the availability of resources is bigger than the needs
    - If this is the case mark the process as terminated, frees its resources and considers the next one
  - If at the end of the loop all processes are marked the state is safe
- If the state S is safe and the request can be granted
- Otherwise the process P waits until there are enough resources to let it proceed

Prima di tutto ogni processo deve dichiarare le risorse delle quali avrà bisogno durante la sua elaborazione (tutte le risorse ai fini dello stallo). Ogni volta che il processo fa richiesta di risorse, in realtà, questa richiesta di risorse è una richiesta che fa al SO tramite chiamata di sistema. Quindi quando la chiamata di sistema si rende conto che per la sua esecuzione servono un certo numero di risorse le richiede al gestore delle risorse, che implementa l'AB. E l'AB, in realtà, fa una simulazione: lui dice: "*Benissimo, questo è lo stato iniziale: ora tu mi richiedi queste risorse, se io ti do queste risorse vado in un nuovo stato S. Vediamo se questo stato è sicuro"*. Quindi l'AB appartiene da questo stato ottenuto assegnando le risorse richieste, va verificare se questo è sicuro. E per far questo applica un procedimento iterativo: quindi va a vedere se con le risorse che lui ha disponibili attualmente, a partire da questo stato, riesce a trovare una sequenza di richieste di assegnazione che permette a tutti i processi di terminare. Se la trova almeno una, allora lo stato è sicuro e quindi quelle risorse richieste possono essere assegnate, altrimenti il processo viene messo in stato di attesa. Vi dico, prima di vedere l'algoritmo vero e proprio, che l'AB fa una simulazione, non attua nulla di per sè. Non fa terminare nessun processo. Lui ha ricevuto una richiesta, fa finta di assegnarla, vede lo stato risultante e da quello stato simula una possibile evoluzione del sistema per verificare che tutti i processi in quella evoluzione possano terminare. Il gioco è che lui riesce a trovare almeno una sequenza di richiesta di assegnazione (???) che porta alla terminazione, allora lo stato è sicuro. Non ha bisogno di garantirlo per tutte le sequenze, basta che ne trovi almeno una; tanto alla prossima richiesta di allocazione lui di nuovo verrà eseguito. Quindi è la continua esecuzione ad ogni richiesta di allocazione dell'AB che garantisce che il sistema resti sempre sicuro.
Per eseguire l'AB ci servono un po' di strutture dati, in particolare, ci serve sapere

Banker's algorithm for multiple resources of multiple types

For each resource R_k: D: availability vector
(D_k number of available units of resource R_k)

For each process P_j : Aj: assignment vector; Ej: vector of residual requirements;
$E_j \leq D$ if $E_{jk} \leq D_k$ for each k

```
Initially each process Pj is not marked
while (∃ non marked processes) {
   if (∃ Pj that satisfies Ej≤D) {
    mark Pj;
    Dj=Dj+ Aj ;
   } else ends while, the state is not safe;
}
success: the initial state is safe
```

quale è il vettore delle risorse disponibili (chiamo D). All'interno di questo vettore D, per ogni risorsa Rk mantengo il numero di unità disponibili di quella risorsa, quindi Dk mi dice quante unità disponibili ci sono della risorsa Rk. Poi, per ogni processo, Pj, devo mantenere il vettore Aj delle risorse che gli ho assegnato: se ho 10 risorse il vettore Dk ha 10 elementi, ogni elemento mi dice la disponibilità per quello specifico tipo di risorsa e il vettore Aj del processo Pj, nuovamente, ha 10 elementi. Aj mi dice quante risorse di tipo j sono state già allocate al processo Pj. Per il processo Pj ho anche il vettore dell'esigenza, che mi dice quante risorse di un certo tipo il processo Pj ha necessità. Quindi, uso questa notazione, e dico che Ej <=D se ogni elemento di Ej < del corrispondente elemento del vettore D. Quindi tutti gli elementi di Ej devono essere < dei rispettivi elementi del vettore D. Vediamo questo esempio.

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (1.1)

Un sistema con processi P1, P2, P3, P4 e risorse dei tipi R1, R2, R3, R4, rispettivamente di molteplicità [4, 5, 5, 5], i processi dichiarano inizialmente le seguenti esigenze:

ESIGENZA INIZIALE				
	R1	R2	R3	R4
P1	2	3	1	1
P2	2	1	1	2
P3	0	1	0	2
P4	0	2	5	2

Stato (sicuro) raggiunto dal sistema al tempo *t*:

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	1	1	0

	R1	R2	R3	R4
P1	0	2	0	0
P2	0	1	0	0
P3	0	0	0	2
P4	0	0	3	0

ESIGENZA RESIDUA

Io ho una matrice che mi rappresenta le esigenze iniziali. Per ogni processo P, in questo caso ho 4 processi, ho un vettore E che mi dice l'esigenza iniziale di quel processo per ogni possibile tipo di risorsa. Quindi ho risorse di tipo R1 e a P4 non servono. A P4 servono 2 risorse di tipo R2, 5 risorse di tipo R3 e 2 risorse di tipo R4. Questo è la matrice A delle assegnazioni attuali: la matrice A è formata da tante righe, una per ogni processo, per esempio la riga A1 mi dice l'assegnazione attuale del processo P1, quindi P1 attualmente ha acquisito 2 risorse di tipo 1, 1 risorsa di 2, 1 risorsa di tipo 3, una risorsa di tipo 4. Questo è il vettore delle risorse esistenti, quindi ho 4 risorse di tipo 1 nel sistema, 5 di tipo 2, 5 di tipo 3, 5 di tipo 4, ma soprattutto la cosa importante è questa, la disponibilità attuale. Attualmente ho 0 risorse libere di tipo 1, quindi le 4 risorse di tipo 1 le ho tutte assegnate. Ho una sola risorsa di tipo 2, quindi io devo avere 4 risorse di tipo 2 già assegnate, ed infatti ne ho assegnate 4. Ho una risorsa di tipo 3 libera e 0 risorse di tipo 4 assegnate. Di conseguenza l'esigenza residua dei thread dei processi è quella mostrata in questa tabella. Banalmente sono tutte somma e sottrazione, quindi per esempio prendiamo la riga 1 dell'esigenza residua: questo è ciò che serve a P1 per terminare. Come lo calcolo? L'esigenza iniziale di P1 era 2, 3, 1, 1. Gli ho già assegnato il vettore 2 1 1 1 , che sono risorse che gli ho già assegnato, quindi la differenza tra l'esigenza iniziale e l'assegnazione attuale mi restituisce 0,2,0,0, che è ciò di cui P1, nel peggiore dei casi avrà bisogno per poter terminare. Queste sono le strutture dati che devo avere con l'AB. Ovviamente questo esempio in scala molto ridotta, in un SO con centinaia, migliaia, decine di migliaia di risorse di tipo differente e con molteplicità molto arbitraria, queste tabelle prendono tutt'altra dimensione. Come funziona a questo punto

l'algoritmo? Questo algoritmo fa una simulazione. A partire da uno stato di assegnazione, quindi a partire da uno stato simile a questo, stabilisce se questo stato è sicuro oppure no. Quindi prima di eseguire questo algoritmo vero e proprio io ho avuto una richiesta, ho fatto finta di assegnare la risorsa richiesta, ho prodotto uno stato (che è rappresentato da tutte queste matrici e vettori che uso nell'algoritmo) e a questo punto a partire da questo stato, faccio una simulazione per vedere se sono in grado di far terminare nel caso peggiore tutti i processi. Inizialmente, utilizzo una marca binaria per sapere se i processi in questa simulazione li posso far terminare oppure no. Quindi inizialmente, non lo so e allora nessun processo è marcato; via via che mi rendo conto che ho risorse sufficienti per far terminare un processo, quello lo marco. Ovviamente non vuol dire che lo faccio terminare, però idealmente è come se dico: "*Questo processo può terminare e quindi terminerà*". A questo punto, inizialmente nessun processo è marcato: entro in questo ciclo nel quale valuto: se ci sono processi che non sono ancora stati marcati, quindi in questa simulazione ancora non hanno terminato, sostanzialmente, allora verifico se tra tutti i processi non marcati ce n'è uno la cui esigenza attuale, residua è <= della disponibilità. Cosa vuol dire che la sua esigenza

Banker's algorithm for multiple resources of multiple types

For each resource R_k: D: availability vector
(D_k number of available units of resource R_k)

For each process P_j : Aj: assignment vector; Ej: vector of residual requirements;
$E_j \leq D$ if $E_{jk} \leq D_k$ for each k

```
Initially each process Pj is not marked
while (∃ non marked processes) {
   if (∃ Pj that satisfies Ej≤D) {
    mark Pj;
    Dj=Dj+ Aj ;
   } else ends while, the state is not safe;
}
success: the initial state is safe
```

residua è <= della sua disponibilità? Vuol dire che attualmente le mie risorse libere sono sufficienti per farlo terminare, quindi se lui mi fa delle ulteriori richieste che sono quelle rappresentate all'interno di questo vettore, io ho disponibilità sufficiente per poterle soddisfare, quindi gli posso assegnare tutto quello di cui lui ha bisogno, da qui fino alla sua fine. Siccome ho disponibilità sufficienti per poterlo far terminare, cosa faccio? Lo marco. Perchè potrei decidere, se mi trovo messo alle strette, di usar tutte le mie disponibilità per far terminare prima lui, per permettergli di terminare, cosicchè quando lui sarà terminato mi restituirà tutto quello che lui ha assegnato attualmente. Quindi io posso farlo terminare, nella mia simulazione, marcarlo e liberare le sua risorse, quindi di assegnarle al vettore della disponibilità attuale. Ho fatto sostanzialmente questo calcolo: ho detto "*Benissimo, se sono messo male, sono a rischio che posso fare?*" Posso assegnare Ej al processo Pj, quindi faccio D-Dj. Lui ha questo punto ha tutte le risorse, non mi chiederà più niente, prima o poi terminerà perchè ha tutte le risorse che gli servono, quando sarà terminato mi restituirà questa Ej e mi restituirà anche quelle che aveva già prima, le Aj. Questo significa che il mio vettore di disponibilità diventerà D + Aj. Sarebbe T + Aj + Ej - Ej . Quindi se lui termina mi restituisce quello che aveva già prima. Ora però il fatto di aver restituito queste risorse che lui deteneva mi va ad incrementare il vettore di disponibilità, quindi è possibile che avendo aumentato la mia disponibilità io abbia risorse sufficienti per far terminare qualcun altro e quindi completo il loop e vado a guardare se ci

sono altri processi non marcati. E' possibile che degli altri processi non marcati, di nuovo, adesso possa soddisfarne qualcun altro.

Ora continuo così finchè non succedono due cose: o li marco tutti (se li ho marcati tutti vuol dire che a quel punto, sarei idealmente in grado di farli terminare tutti, anche se con una sequenza di esecuzione sequenziale però sarei in grado di farli terminar tutti nel caso peggiore. Quindi se li ho marcati tutti, esco da questo While e a questo punto ho successo: lo stato iniziale è sicuro. L'altra possibilità è invece che ad un certo punto resto con un gruppo di processi che restano non marcati perchè per nessuno di loro vale questa condizione. Quindi per tutti i processi che mi restano vale che la loro esigenza è maggiore della disponibilità. E allora, in questo caso, se questi processi chiedessero tutte le risorse che hanno dichiarato nel loro vettore delle esigenze, io potrei trovarmi in uno stallo. Quindi in questo caso, se restano processi non marcati che non riesco a marcare, allora lo stato non è sicuro. Una nota: questo AB io lo posso usare in due modalità: o in modo preventivo, appunto per impedire le assegnazioni che mi portano in uno stato insicuro, oppure lo posso usare come meccanismo di Detect&Fix. Per cui io potrei usare l'AB per andare a vedere lo stato dei processi con risorse multiple e capire se sono in stallo. Se lo uso in questa seconda modalità, tutti i processi non marcati alla fine dell'algoritmo sono in stallo, in realtà, o sono Doomed, a

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (1.1)

Un sistema con processi P1, P2, P3, P4 e risorse dei tipi R1, R2, R3, R4, rispettivamente di molteplicità [4, 5, 5, 5], i processi dichiarano inizialmente le seguenti esigenze:

ESIGENZA INIZIALE				
	R1	R2	R3	R4
P1	2	3	1	1
P2	2	1	1	2
P3	0	1	0	2
P4	0	2	5	2

Stato (sicuro) raggiunto dal sistema al tempo *t:*

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

	R1	R2	R3	R4
P1	0	2	0	0
P2	0	1	0	0
P3	0	0	0	2
P4	0	0	3	0

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	1	1	0

seconda dei casi. Vediamo a questo punto l'esempio. Immaginiamo di partire da questo stato.
Se io ho utilizzato l'AB correttamente (e certamente se lo utilizzo posso usarlo correttamente).

Verifichiamo che quello stato effettivamente sia sicuro.

VERIFICA: LO STATO RAGGIUNTO AL TEMPO t E' SICURO (Ver 1)

Stato raggiunto dal sistema al tempo *t:*

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	0	2	0	0
P2	0	1	0	0
P3	0	0	0	2
P4	0	0	3	0

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	1	1	0

Verifica dello stato sicuro:

1) l'esigenza di P2 può essere soddisfatta e P2 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	2	1	1
P2	-	-	-	-
P3	0	1	0	0
P4	0	2	2	2

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	0	2	0	0
P2	-	-	-	-
P3	0	0	0	2
P4	0	0	3	0

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
2	1	2	2

Quindi applichiamo l'AB per verificare che, come vi dicevo, è sicuro. Cosa devo fare? Inizialmente tutti e quattro i processi sono non marcati e inizio il ciclo: devo cercare se esiste un processo non marcato per cui la sua esigenza attuale è <= della disponibilità. Ne trovate almeno uno? P1 no, la sua esigenza attuale non è minore della sua disponibilità. Perchè avrebbe bisogno di 2 risorse di tipo R2 e non ce le ho. Neanche P3 e P4, P3 avrebbe bisogno di 2 risorse di tipo 4 e ne ho 0; P4 addirittura ha bisogno di 3 risorse di tipo 3 e ne ho soltanto 1. Quindi P1, P3, P4 non soddisfano questa condizione. Invece P2, per terminare, ha bisogno solo di una risorsa di tipo R2, e quindi io posso far terminare soltanto P2 in questa situazione, in questo stato. Allora che faccio? Simulo la terminazione di P2. Lo marco e di conseguenza tutte le sue risorse allocate diventano magicamente libere. Quindi il vettore di disponibilità diventa improvvisamente pari a 2 1 2 2. Ho sommato alla disponibilità attuale l'assegnazione attuale di P2, quindi questa diventa la mia disponibilità attuale.

VERIFICA: LO STATO RAGGIUNTO AL TEMPO t E' SICURO (Ver 2)

Verifica dello stato sicuro:
1) l'esigenza di P2 può essere soddisfatta e P2 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	2	1	1
P2	-	-	-	-
P3	0	1	0	0
P4	0	2	2	2

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	0	2	0	0
P2	-	-	-	-
P3	0	0	0	2
P4	0	0	3	0

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
2	1	2	2

Verifica dello stato sicuro:
1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
2) l'esigenza di P3 può essere soddisfatta e P3 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	-	-	-	-
P3	-	-	-	-
P4	0	2	2	2

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	0	2	0	0
P2	-	-	-	-
P3	-	-	-	-
P4	0	0	3	0

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
2	2	2	2

A questo punto, (figura) siccome ne ho marcato 1, continuo il ciclo e mi chiedo: tra tutti questi processi ancora non marcati ne esiste uno per cui la sua esigenza residua è <= della disponibilità attuale? La risposta è: P1 certamente no, perchè vorrebbe 2 di tipo 2; P3 sì, perchè ha bisogno soltanto di 2 risorse di tipo 4; mentre invece P4 no, perchè ha bisogno di 2 risorse di tipo 3 e non ce le ho. Quindi P3 posso farlo terminare. Se per caso P1 e P4 mi chiedono qualcosa li metto in attesa; se P3 mi chiede qualcosa gli posso assegnare tutto quello di cui ha bisogno, questa è l'idea. Siccome gli posso assegnare tutto, lui potrà terminare: lo posso marcare come terminato e il vettore di disponibilità attuale diventa 2 2 2 2, perchè vado a sommare l'assegnazione attuale di P3.

VERIFICA: LO STATO RAGGIUNTO AL TEMPO t E' SICURO (Ver 3)

Verifica dello stato sicuro:

1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
2) l'esigenza di P3 può essere soddisfatta e P3 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	-	-	-	-
P3	-	-	-	-
P4	0	2	2	2

	R1	R2	R3	R4
P1	0	2	0	0
P2	-	-	-	-
P3	-	-	-	-
P4	0	0	3	0

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
2	2	2	2

Verifica dello stato sicuro:

1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
2) l'esigenza di P3 può essere soddisfatta e P3 può terminare
3) l'esigenza di P1 può essere soddisfatta e P1 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	0	2	2	2

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	0	0	3	0

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
4	3	3	3

A questo punto con questa disponibilità posso far terminare P1? Sì, ma non P4. Se P4 mi dovesse chiedere delle risorse lo bloccherò. Mentre invece a P1 posso assegnar tutto, siccome gli posso assegnar tutto posso lo posso far terminare in questa simulazione. La disponibilità, a questo punto, diventa 4 3 3 3 e con questa posso far terminare anche P4.

VERIFICA: LO STATO RAGGIUNTO AL TEMPO t E' SICURO (Ver 4)

Verifica dello stato sicuro:

1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
2) l'esigenza di P3 può essere soddisfatta e P3 può terminare
3) l'esigenza di P1 può essere soddisfatta e P1 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	0	2	2	2

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	0	0	3	0

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
4	3	3	3

Verifica dello stato sicuro:

1) l'esigenza di P2 può essere soddisfatta e P2 può terminare
2) l'esigenza di P3 può essere soddisfatta e P3 può terminare
3) l'esigenza di P1 può essere soddisfatta e P1 può terminare
4) l'esigenza di P4 può essere soddisfatta e P4 può terminare

STATO SICURO !

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
4	5	5	5

Terminato e marcato anche P4, ho marcato tutti i processi e ho finito. Lo stato è sicuro.

A questo punto tutto ciò che ho fatto in questa simulazione lo annullo, ritorno al punto di partenza e i processi continuano da dove li avevo lasciati. Ora immaginiamo di avere quest'altra situazione: partiamo nuovamente da uno stato sicuro.

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (1.2)

Stato (sicuro) raggiunto dal sistema al tempo *t*:

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	0	2	0	0
P2	0	1	0	0
P3	0	0	0	2
P4	0	0	3	0

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	1	1	0

Il processo P1 richiede una risorsa di tipo R2: stato raggiunto dopo l'ipotetica assegnazione:

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	2	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	0	1	0	0
P2	0	1	0	0
P3	0	0	0	2
P4	0	0	3	0

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	0	1	0

Supponiamo che in questo stato il processo P1 richieda una risorsa di tipo 2. Quindi cosa avviene nella realtà? P1 invoca una Chiamata di Sistema, che nella sua esecuzione si rende conto di aver bisogno di una risorsa di tipo 2, invoca il gestore delle risorse del SO, che a sua volta esegue l'AB. Quindi simula l'assegnazione di questa risorsa di tipo 2 a P1 e verifica se lo stato risultante è sicuro oppure no. Se noi simuliamo l'assegnazione di questa risorsa cosa succede? Nella assegnazione attuale di P1, la sua assegnazione diventa 2 2 1 1, perchè gli ho assegnato una risorsa di tipo 2 e di conseguenza la sua esigenza residua diventa 0 1 0 0, perché gli ho assegnato una risorsa di tipo 2. Era 0 2 0 0, quindi diventa 0 1 0 0. Alla stessa maniera la disponibilità attuale diventa 0 0 1 0. La domanda è: questo stato risultante dopo l'assegnazione di quella risorsa a P1 è sicuro? Andiamo a vedere.

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (1.3)

Il processo P1 richiede una risorsa di tipo R2: stato raggiunto dopo l'ipotetica assegnazione:

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	2	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	0	1	0	0
P2	0	1	0	0
P3	0	0	0	2
P4	0	0	3	0

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	0	1	0

Verifica dello stato sicuro:
1) l'esigenza di P1 non può essere soddisfatta
2) l'esigenza di P2 non può essere soddisfatta
3) l'esigenza di P3 non può essere soddisfatta
4) l'esigenza di P4 non può essere soddisfatta

LO STATO NON E' SICURO

Devo confrontare l'esigenza residua di tutti i processi con la disponibilità. Inizialmente tutti i processi sono non marcati. Esiste un processo per cui l'esigenza residua è <= della disponibilità? Per P1 no, gli manca una di tipo 2 e ne ho 0. Per P2 no, anche lui ha bisogno di 1 di tipo 2 e ne ho 0. P3 no, perchè ha bisogno di 2 di tipo 4 e ne ho 0. E P4 neanche perchè ho bisogno di 3 di tipo 3 e ne ho soltanto 1 di tipo R3. Non posso marcare nessun processo: quindi termino dal ciclo e lo stato non è sicuro. Che faccio? Quando P1 mi chiede la risorsa di tipo R2 lo metto in stato di attesa. Quando qualche processo libererà delle risorse andrò ad eseguire l'AB per verificare se questa richiesta la posso soddisfare. Vediamo un altro esempio:

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.1)

Un sistema con processi P1, P2, P3, P4 e risorse dei tipi R1, R2, R3, R4, rispettivamente di molteplicità [4, 5, 5, 5], i processi dichiarano inizialmente le seguenti esigenze:

ESIGENZA INIZIALE				
	R1	R2	R3	R4
P1	2	3	1	1
P2	2	0	1	3
P3	0	1	0	2
P4	0	2	3	2

Stato raggiunto dal sistema al tempo *t:*

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	0	2	0	0
P2	0	0	0	1
P3	0	0	0	2
P4	0	0	1	0

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	1	1	0

supponiamo di avere questo stato (differente) con quella esigenza residua, disponibilità e assegnazione attuale. A partire da questo stato supponiamo che P1 mi chieda una risorsa di tipo 2. Simuliamo.

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.2)

Stato raggiunto dal sistema al tempo *t:*

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	1	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	0	2	0	0
P2	0	0	0	1
P3	0	0	0	2
P4	0	0	1	0

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	1	1	0

Al tempo *t* processo P1 richiede una risorsa di tipo R2: stato raggiunto dopo l'ipotetica assegnazione:

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	2	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

ESIGENZA RESIDUA

	R1	R2	R3	R4
P1	0	1	0	0
P2	0	0	0	1
P3	0	0	0	2
P4	0	0	1	0

DISPONIBILITA' ATTUALE

R1	R2	R3	R4
0	0	1	0

Assegniamo nella simulazione la risorsa di tipo R2 a P1, modifichiamo di conseguenza la sua assegnazione attuale, la sua esigenza residua e la disponibilità attuale. A questo punto verifichiamo se questo stato è sicuro.

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.3)

Al tempo *t* processo P1 richiede una risorsa di tipo R2: stato raggiunto dopo l'ipotetica assegnazione:

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	2	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	0	2	2	2

	R1	R2	R3	R4
P1	0	1	0	0
P2	0	0	0	1
P3	0	0	0	2
P4	0	0	1	0

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	0	1	0

Verifica dello stato sicuro:
1) l'esigenza di P4 può essere soddisfatta e P4 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	2	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	-	-	-	-

	R1	R2	R3	R4
P1	0	1	0	0
P2	0	0	0	1
P3	0	0	0	2
P4	-	-	-	-

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	2	3	2

Devo confrontare la disponibilità attuale con l'esigenza residua di ogni processo. Nel caso di P1 l'esigenza residua non è <, perchè ha bisogno di una risorsa di tipo R2 e non ce l'ho. Neanche per P2 e P3. Per P4 invece vale, perchè la sua esigenza residua è <= della disponibilità attuale, quindi quello che posso fare è: simulare l'assegnazione di una risorsa di tipo 3 a P4 (quello di cui P4 ha bisogno per terminare); quando P4 terminerà rilascerà tutta la sua assegnazione attuale, per cui il vettore disponibilità diventerà 0 2 3 2 (libero 0 2 2 2 risorse che si vanno a sommare a quelle che avevo disponibili attualmente) e ovviamente la sua esigenza residua diventa niente perchè lo posso far terminare e una volta terminato non avrò esigenza residua.
A partire da questo stato c'è qualche altro processo che può terminare?

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.4)

Verifica dello stato sicuro:
1) l'esigenza di P4 può essere soddisfatta e P4 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	2	2	1	1
P2	2	0	1	2
P3	0	1	0	0
P4	-	-	-	-

	R1	R2	R3	R4
P1	0	1	0	0
P2	0	0	0	1
P3	0	0	0	2
P4	-	-	-	-

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
0	2	3	2

Verifica dello stato sicuro:
1) l'esigenza di P4 può essere soddisfatta e P4 può terminare
2) l'esigenza di P1 può essere soddisfatta e P1 può terminare

MOLTEPLICITA' →

R1	R2	R3	R4
4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	2	0	1	2
P3	0	1	0	0
P4	-	-	-	-

	R1	R2	R3	R4
P1	-	-	-	-
P2	0	0	0	1
P3	0	0	0	2
P4	-	-	-	-

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
2	4	4	3

Di nuovo, confronto la disponibilità attuale con l'esigenza residua e qui addirittura potrei farli terminare già tutti quanti: quindi ne prendo uno, l'esigenza di P1 può essere soddisfatta e quindi P1 può terminare e come P1 termina libera le risorse e la disponibilità attuale diventa >.

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.5)

Verifica dello stato sicuro:

1) l'esigenza di P4 può essere soddisfatta e P4 può terminare
2) l'esigenza di P1 può essere soddisfatta e P1 può terminare

MOLTEPLICITA' →

	R1	R2	R3	R4
	4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	2	0	1	2
P3	0	1	0	0
P4	-	-	-	-

	R1	R2	R3	R4
P1	-	-	-	-
P2	0	0	0	1
P3	0	0	0	2
P4	-	-	-	-

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
2	4	4	3

Verifica dello stato sicuro:

1) l'esigenza di P4 può essere soddisfatta e P4 può terminare
2) l'esigenza di P1 può essere soddisfatta e P1 può terminare
3) l'esigenza di P2 può essere soddisfatta e P2 può terminare

MOLTEPLICITA' →

	R1	R2	R3	R4
	4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	0	1	0	0
P4	-	-	-	-

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	0	0	0	2
P4	-	-	-	-

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
4	4	5	5

A questo punto a partire da questo stato posso far terminare anche qui P2. E quindi la diponibilità diventa 4 5 5 5 e con questa posso far terminare anche P3. Ho marcato tutti e 4 i processi, lo stato è sicuro.

ALGORITMO DEL BANCHIERE CON RISORSE MULTIPLE DI PIU' TIPI (2.6)

Verifica dello stato sicuro:

1) l'esigenza di P4 può essere soddisfatta e P4 può terminare
2) l'esigenza di P1 può essere soddisfatta e P1 può terminare
3) l'esigenza di P2 può essere soddisfatta e P2 può terminare

MOLTEPLICITA' →

	R1	R2	R3	R4
	4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	0	1	0	0
P4	-	-	-	-

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	0	0	0	2
P4	-	-	-	-

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
4	4	5	5

Verifica dello stato sicuro:

1) l'esigenza di P4 può essere soddisfatta e P4 può terminare
2) l'esigenza di P1 può essere soddisfatta e P1 può terminare
3) l'esigenza di P2 può essere soddisfatta e P2 può terminare
4) l'esigenza di P3 può essere soddisfatta e P3 può terminare

STATO SICURO !

MOLTEPLICITA' →

	R1	R2	R3	R4
	4	5	5	5

ASSEGNAZIONE ATTUALE →

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

	R1	R2	R3	R4
P1	-	-	-	-
P2	-	-	-	-
P3	-	-	-	-
P4	-	-	-	-

ESIGENZA RESIDUA

DISPONIBILTA' ATTUALE

R1	R2	R3	R4
4	5	5	5

Ora vedremo un altro problema noto di programmazione concorrente: il problema dei Lettori/Scrittori. Tale problema è parente del Produttore consumatore, ma un po' più complicato.

Il problema

Una struttura dati è condivisa da un insieme di thread, che appartengono a una delle due seguenti categorie:

- *Thread lettori*: accedono alla struttura dati esclusivamente per leggere, senza modificare i dati;
- *Thread scrittori*: accedono alla struttura dati senza restrizioni, con la possibilità di modificare i dati.

Per evitare interferenze, sono imposti seguenti vincoli:

- i thread scrittori accedono alla struttura dati in mutua esclusione, rispetto agli altri scrittori e ai lettori;
- i thread lettori accedono alla struttura dati in mutua esclusione rispetto agli scrittori, ma senza vincolo di mutua esclusione rispetto agli altri lettori (in altre parole, più lettori possono utilizzare la struttura dati concorrentemente).

L'idea è: noi abbiamo un certo numero di thread che cooperano tra di loro tramite una struttura dati. Alcuni thread la scrivono e altri la leggono. Qual è la differenza rispetto al Produttore/Consumatore? La differenza è che i lettori leggono ciò che c'è scritto ed eventualmente possono leggere anche più volte le stesse informazioni se nessuno ha aggiornato la tabella. Gli scrittori aggiornano la tabella in concorrenza tra loro e con i lettori. Immaginatevi, come esempio, il gioco in borsa. Gli indici di borsa sono sostanzialmente delle tabelle con le quotazioni delle azioni. Quello è un classico esempio di lettore/scrittore perchè le persone di tutto il mondo osservano quelle tabelle per decidere come investire, quindi leggono. Tutti leggono le stesse informazioni, quindi non ho il problema che ogni informazione deve essere letta una e una sola volta. Tutti quelli collegati in quel momento leggono esattamente le stesse informazioni, quello che vogliono leggere leggono. Eventualmente più di una volta. Dall'altra parte ci sono invece degli scrittori, quindi l'agenzia che gestisce la borsa che periodicamente con una certa cadenza aggiorna gli indici e quindi aggiorna queste tabelle. Può capitare che se sto dormendo e hanno aggiornato gli indici, non li ho visti e mi sono perso un po' di informazioni: ma poco male perché quello che conta è l'ultima informazione visualizzata, non quella vecchia. Quindi i lettori possono perdere alcune informazioni (alcune informazioni possono essere scritte e lette da nessuno) oppure altre informazioni possono essere scritte e lette da tanti. Vi ricordo che nel Produttore/Consumatore, ogni informazione deve essere letta una ed una sola volta: una volta letta viene estratta dal buffer. Nel caso del Lettore/scrittore non si estrae niente, si legge e basta quello che c'è. In questo contesto, per evitare interferenze, i lettori possono leggere tranquillamente in parallelo perchè non modificano la struttura dati, quindi tra di loro non c'è interferenza (due lettori possono leggere senza avere mutua esclusione tra loro). Per quanto riguarda invece gli scrittori invece è diverso: gli scrittori devono essere in mutua esclusione tra loro e con i lettori (mentre uno scrittore sta scrivendo non posso avere un lettore che legge). Quindi o c'è attivo uno scrittore oppure sono attivi uno o più lettori, oppure nessuno. Non posso avere 2 scrittori attivi contemporaneamente e non posso avere uno scrittore e un lettore. Questa è la definizione del problema.

Come funziona l'accesso alla risorsa?

Il problema

Il problema viene risolto con una politica che consente l'accesso alternativamente a uno scrittore e a un insieme di lettori, ed evita l'attesa indefinita per gli scrittori. Precisamente, per le richieste valgono le seguenti clausole:

1. Quando la struttura dati non è utilizzata da nessun processo:
   - accede il primo processo (lettore o scrittore) che ne fa richiesta;
2. Quando la struttura dati è utilizzata da uno scrittore, oppure da uno o più lettori:
   - se un lettore richiede l'accesso e la struttura dati è utilizzata da uno scrittore, il lettore richiedente viene sospeso;
   - se un lettore richiede l'accesso e la struttura dati è utilizzata da uno o più lettori, il lettore ottiene l'accesso *a condizione che non vi siano scrittori in attesa*;
   - se un lettore richiede l'accesso, la struttura dati è utilizzata da uno o più lettori e vi è almeno uno scrittore in attesa di accedere, il lettore richiedente viene sospeso *(questa clausola evita l'attesa indefinita per gli scrittori)*;
   - se uno scrittore richiede l'accesso e la struttura dati è utilizzata da uno scrittore oppure da uno o più lettori, lo scrittore si sospende.

Sia che si tratti di un lettore che di uno scrittore, il thread prima verifica se può accedere in lettura o scrittura (e le due cose sono differenti). A quel punto accede alla struttura e poi segna il fatto che sta uscendo, che la sta rilasciando. Se la struttura dati non è usata da nessun processo il primo processo, lettore o scrittore, che ne fa richiesta entra. Se uno scrittore entra nell'utilizzo di una risorsa, e quindi la risorsa viene bloccata per tutti gli altri, se entra un lettore viene bloccata per gli scrittori, ma viene lasciata per trattare (???) i lettori. Se la struttura dati è utilizzata da uno scrittore oppure da uno o più lettori che succede? Se un lettore chiede l'accesso e c'è già uno scrittore dentro il lettore viene sospeso. Se un lettore chiede l'accesso e la struttura è usata da uno o più lettori, il lettore ottiene l'accesso, però per evitare l'attesa indefinita degli scrittori si utilizza questa ulteriore condizione; che non ci siano scrittori in attesa. Qual è il problema? Se arriva un lettore, poi ne arriva un altro, lo scrittore che arriva si può sospendere, resta in attesa, però ora se arrivano altri lettori e a questi lettori noi continuiamo a permettere di entrare, via via che escono dei lettori, ne entrano altri e lo scrittore rischia di restare in attesa indefinita. Quindi in caso ci sia uno scrittore in attesa gli viene data la priorità. E' una condizione che serve per evitare l'attesa indefinita, ma non altera la correttezza da tutti gli altri punti di vista della soluzione.

Se uno scrittore richiede l'accesso alla struttura dati usata da uno scrittore oppure da uno o più lettori lo scrittore ovviamente si sospende.

Il problema

Per i rilasci valgono le seguenti clausole:

3. quando uno scrittore rilascia la struttura dati:
   - se esistono lettori in attesa, tutti questi lettori sono riattivati e ottengono l'accesso;
   - se non esistono lettori in attesa ma esiste almeno uno scrittore in attesa, il primo di questi scrittori ottiene l'accesso;
   - se non esistono lettori o scrittori in attesa, si torna al caso 1).
4. quando un lettore rilascia la struttura dati:
   - se altri lettori stanno ancora utilizzando la struttura dati, continua l'accesso in lettura da parte di questi thread;
   - se non vi sono altri lettori che utilizzano la struttura dati ed esiste almeno uno scrittore in attesa, il primo di questi scrittori ottiene l'accesso;
   - altrimenti si torna al caso 1).

Per quanto riguarda invece il rilascio se è uno scrittore che rilascia una struttura dati allora prima si riattivano i lettori: se non ci sono lettori in attesa allora si guarda se ci sono scrittori in attesa (e ne viene riattivato 1). Se non ci sono neanche scrittori in attesa, la struttura viene lasciata libera. Quando invece un lettore rilascia una struttura dati, se ci sono altri lettori che agiscono sulla struttura allora la struttura resta in uso a quei lettori. Se invece è l'ultimo lettore, quindi quello che sta uscendo ed è l'ultimo (l'ultimo chiude la porta): se ci sono scrittori in attesa ne riattiva uno altrimenti la struttura viene lasciata libera. Alcuni suggerimenti per scrivere la soluzione.

Il problema

Per la soluzione del problema, si utilizzano i seguenti dati condivisi da tutti i thread:

- activeReaders : intero non negativo; valore iniziale 0
- waitingReaders : intero non negativo; valore iniziale 0
- activeWriters : intero non negativo; valore iniziale 0
- waitingWriters : intero non negativo; valore iniziale 0

e le seguenti variabili:

- Lock mutex: per la mutua esclusione;
- Cond readGo: variabile di condizione utilizzata per la sospensione dei lettori;
- Cond writeGo : variabile di condizione utilizzata per la sospensione degli scrittori;

Utilizziamo alcune variabili: non sono variabili di condizione, ma sono variabili che rappresentano lo stato del nostro sistema, che sono condivise e che voi dovrete utilizzare per aiutarvi a scrivere la soluzione di questo problema. Queste variabili sono variabili che memorizzano lo stato degli accessi, per cui abbiamo ActiveReaders, che ci dice quanti sono i lettori attivi nella struttura, inizialmente 0, WaitingReaders, che ci dice quanti lettori sono in attesa di poter leggere (inizialmente 0), ActiveWriters, che ci dice quanti scrittori

sono attivi in scrittura (inizialmente è 0 e assume al massimo 1), ActiveWriters, che ci dice quanti scrittori sono in attesa di poter scrivere (inizialmente 0). Poi usiamo queste variabili: Lock Mutex, che è una variabile di mutua esclusione, e poi le condizioni ReadGo e WrtieGo, che sono due variabili di condizione sulle quali si sospendo i lettori e gli scrittori, rispettivamente.

Lettore

```
...
While (true) {
   RWLock::startRead();
   <accede in lettura alla struttura condivisa>
   RWLock::doneRead();
   <usa i dati letti>
}
...
```

La procedura che segue il lettore e lo scrittore segue questo schema: il lettore normalmente ha un ciclo, esattamente come avviene per gli indici di borsa, uno fa un ciclo, osserva gli indici (prende l'informazione), decide l'investimento, quale investimento e di nuovo va a leggere gli indici per passare al prossimo. Quindi lo scrittore è in un ciclo infinito dove, ad un certo punto, inizia la lettura, quindi esegue la funzione startRead() che stabilisce se può acquisire il diritto a leggere la struttura dati; dopodichè accede in lettura alla struttura. Badate bene che questa parte qui è sezione critica rispetto agli scrittori, ma non è sezione critica rispetto agli altri lettori. Quando ha terminato di leggere esegue una funzione doneRead() che termina la lettura e poi fa quello che deve fare. Viceversa, lo scrittore avrà una funzione startWrite() con la quale inizia la scrittura, poi accede alla struttura in scrittura (questa è una sezione critica con tutti, sia con i lettori che con gli scrittori) e poi fa doneWrite(). Una soluzione banale, che però non è realmente performante è quella di risolvere il problema dei lettori/scrittori con un unico mutex, per cui o c'è un lettore o uno scrittore attivo, però questo tipo di di soluzione (lockAcquire al posto della startRead e lockRelease al posto della doneRead) garantisce sì, la correttezza nell'uso della struttura, ma non è performante perchè impedisce ai lettori di leggere contemporaneamente, cosa che invece voglio permettere, perchè i lettori, in realtà, non vanno in mutua esclusione tra loro. Quindi la soluzione banale di usare un unico lock è corretta però non è accettabile come soluzioni del problema perchè non garantisce le prestazioni massime.

Scrittore

```
...
While (true) {
   <prepara dati da scrivere>
   RWLock::startWrite();
   <accede in scrittura alla struttura condivisa>
   RWLock::doneWrite();
}
...
```

Scrivere startRead, doneRead, startWrite, doneWrite.

Lettore – startRead(), doneRead()

```
void RWLock::startRead() {
    .....
    waitingReaders++;
    while (activeWriters > 0 ||
           waitingWriters > 0) {
        .....
    }
    waitingReaders --;
    activeReaders++;
    .....
}
```

```
void RWLock::doneRead()
{
    .....
    activeReaders--;
    if (activeReaders ==0 &&
    waitingWriters > 0) {

    .....
    }
    .....
}
```

Scrittore – startWrite(), doneWrite()

```
void RWLock::startWrite() {
    .....
    waitingWriters++;
    while (activeWriters > 0 ||
           activeReaders > 0) {
        .....
    }
    waitingWriters --;
    activeWriters++;
    .....
}
```

```
void RWLock::doneWrite() {
    .....
    activeWriters--;
    if (waitingWriters > 0) {
        .....
    }
    else
    readGo.Broadcast(&mutex);
    .....
}
```

Partiamo dalla startRead.

Prima osservazione: la startRead è quella che il lettore esegue se può leggere. Esso deve verificare che non ci siano scrittori già attivi o che non ci siano scrittori in coda, quindi deve andare ad analizzare il contenuto delle variabili ActiveWriters o WaitingWriters. D'altra parte le variabili WaitingWriters o ActiveWriters, da chi verranno modificate? Verranno modificate dagli scrittori. Quindi AW, WW sono variabili condivise da lettori/scrittori. I lettori le leggono per capire se possono poi entrare in lettura e gli scrittori le scrivono per segnare il loro stato. Quindi, essendo condivise, queste variabili vanno protette. Questo pezzo di codice è una sezione critica e quindi lo dobbiamo proteggere con delle lock su Mutex; non soltanto questo test, ma anche la modifica di WR e di AR, perché se date un'occhiata al codice dello scrittore, il codice dello scrittore andrà a leggere AR, andrà a modificare WR e AW. Quindi anche queste operazioni vanno protette all'interno della mutua esclusione. È tutta questa che è una sezione critica. Come si fa a renderla in mutua esclusione? Questa è mutua esclusione tra lettore e lettore, sia tra lettore e scrittore, è mutua esclusione per tutti. Badate bene che questo è un prologo, qui ancora non stiamo andando a leggere la struttura dati, qui stiamo

decidendo se possiamo entrare la struttura dati. Quindi l'accesso alla struttura dati e la sezione critica che materialmente esegue l'accesso sono due cose differenti.
Noi abbiamo bisogno di una sezione critica per proteggere la funzione che stabilisce chi può entrare o chi no nella struttura dati. Qui che cosa dobbiamo mettere? Dobbiamo mettere in alto un Acquire() e in basso una Release(). L'unica variabile che vi ho suggerito nel testo di utilizzare per la mutua esclusione è la Lock Mutex. Lo stesso lo devo fare quando vado a completare la lettura perchè qui vado a modificare AR, vado a leggere AR, vado a leggere WW, quindi anche qui dovrò fare una mutex.Acquire e release. Stesso discorso per quanto riguarda lo scrittore. Mutex.Acquire all'inizio, mutex.Release alla fine. Vi ricordo che questa Mutex non è una Mutex che regola l'accesso alla struttura: l'accesso alla struttura è regolato da tutto questo prologo e l'uscita dalla struttura è regolato da tutto questo epilogo. Quindi l'accesso alla struttura non è sotto la protezione della Mutex, ma è sotto la protezione di un epilogo e di un prologo scritto correttamente, sia per i lettori che per gli scrittori. Un suggerimento: se capitano esercizi di questo genere la soluzione di questi schemi è molto standard, ma è volutamente molto standard, perchè nel 99% dei casi, quando uscirete da qui, dovrete applicare questo schema di soluzione. Quindi anche se uno non sa né leggere né scrivere, è buona norma iniziare una sezione critica con una lockAcquire e terminare con una lockRelease. Una volta che abbiamo acquisito la mutua esclusione per decidere se possiamo leggere, andiamo a vedere delle condizioni per la lettura. Che cosa fa questo codice in realtà? Mette subito WR++ perchè nel caso in cui il lettore si debba sospendere segna il fatto che si è sospeso in lettura, tanto poi alla fine di questo prologo se poi non mi sono sospeso, decrementerò subito WR, queste operazioni si annulleranno a vicenda all'interno del prologo. A questo punto verifico se devo aspettare oppure no: le condizioni ve le ho date prima nel testo. Se ci sono scrittori in attesa oppure se ci sono scrittori in coda, allora aspetto, altrimenti entro in lettura. Per entrare in lettura, quindi, mi cancello dai lettori in attesa e mi aggiungo ai lettori attivi. L'unica cosa che devo fare in realtà qui, è soltanto mettermi in attesa. Come faccio a mettermi in attesa all'interno di questo While? Dovrò fare una readGo.Wait. Ricordate però come si utilizzano le Wait. La Wait prende come parametro una variabile di mutua esclusione, nel nostro caso la Mutex (readGo.wait(&Mutex)). Il codice della readDone cosa deve fare? Se io sono un lettore e ho smesso di leggere dovrò riattivare qualcuno. Come faccio a riattivare e chi devo riattivare? In effetti io avrò due tipologie di thread in attesa: avrò dei thread lettori in attesa oppure avrò i thread scrittori in attesa. Se io sono un lettore e sto terminando di utilizzare la struttura dati, chi dovrò riattivare? Se voi andate ad analizzare tutti i possibili casi, scoprite che non ci sono tante alternative. Basta elencarle tutte.
Caso 1 è che ci sia almeno uno scrittore in attesa e almeno un lettore in attesa: in questo caso devo svegliare lo scrittore, quindi dovrò fare la signal sulla variabile di condizione degli scrittori. Vediamo gli altri casi. Potrei avere almeno uno scrittore in attesa e nessun lettore in attesa, in questo caso che devo fare? Una Signal sugli scrittori. Prendiamo l'altro caso. Non ho nessuno scrittore in attesa e ho almeno un lettore in attesa, cosa devo fare? Non è possibile questo caso. Perchè se non c'è nessuno scrittore in attesa, i lettori non si sospendono. Io sono un lettore dentro la struttura, quindi non può esserci uno scrittore: se arriva un altro lettore e non può trovare nessun scrittore attivo perchè c'è già un lettore dentro; se non trova nessuno scrittore in attesa il lettore va avanti. Quindi non esiste il caso in cui ci siano 0 scrittori in attesa e almeno 1 lettore in attesa. Quindi questo caso non mi interessa. Se non c'è nessuno scritture in attesa e nessun lettore in attesa non devo fare niente. Quindi in realtà quando completo la Read l'unico caso che mi interessa è questo. Se sono l'ultimo lettore e ci sono scrittori in attesa faccio la Signal su WriteGo. La vostra collega dice "*Gli scrittori però così non risvegliano in ordine in cui sono arrivati, in realtà se metto la Signal risveglio uno scrittore"*. Nel 99.9% dei casi l'implementazione della Signal riattiva il primo scrittore. Quindi in realtà, quasi certamente sarà un ordine FIFO; però se voi volete obbligare l'ordine FIFO potete combinare questa soluzione con lo schema che vi avevo dato tempo fa per realizzare un ordine FIFO nella riattivazione dei thread, quindi le cose si possono combinare fra loro. Per cui la soluzione del lettore è questa.

Lettore – startRead()

```
void RWLock::startRead()
{
    mutex.Acquire();
    waitingReaders++;
    while (activeWriters > 0 || waitingWriters > 0) {
        readGo.Wait(&mutex);
    }
    waitingReaders --;
    activeReaders++;
    mutex.Release();
}
```

Lettore – doneRead()

```
void RWLock::doneRead()
{
    mutex.Acquire();
    activeReaders--;
    if (activeReaders ==0 && waitingWriters > 0) {
        writeGo.Signal(&mutex);
    }
    mutex.Release();
}
```

Notate che il lettore che ha terminato la lettura si decrementa, segna che lui non è più attivo (quindi decrementa AR) dopodiché questo test lo può fare correttamente, che non ci siano lettori attivi e che ci sia almeno uno scrittore. Se ci sono lettori attivi è scorretto svegliare lo scrittore, perchè lui non può fare nulla comunque.

Scrittore – startWrite()

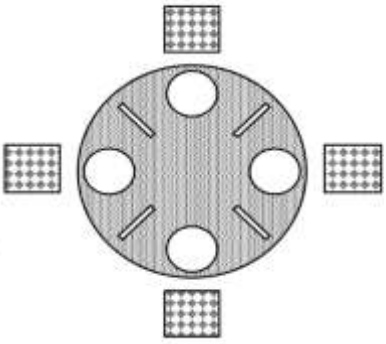
```
void RWLock::startWrite()
{
    mutex.Acquire();
    waitingWriters++;
    while (activeWriters > 0 || activeReaders > 0) {
        writeGo.Wait(&mutex);
    }
    waitingWriters --;
    activeWriters++;
    mutex.Release();
}
```

Scrittore – doneWrite()

```
void RWLock::doneWrite()
{
    mutex.Acquire();
    activeWriters--;
    assert(activeWriters == 0);
    if (waitingWriters > 0) {
        writeGo.Signal(&mutex);
    }
    else readGo.Broadcast(&mutex);
    mutex.Release();
}
```

Per quanto riguarda lo scrittore in realtà il codice è molto simile, con l'unica differenza che varia la condizione per sospendersi oppure per riattivare. Quindi lo scrittore, quand'è che si sospende? Si deve sospendere se ci sono scrittori attivi oppure se ci sono lettori attivi. In questo caso si sospende in attesa, altrimenti si segna come scrittore attivo ed entra in scrittura. Vi faccio notare che per l'effetto di questo prologo gli scrittori sono in mutua esclusione fra loro, non posso avere due scrittori attivi contemporaneamente, perchè questa condizione blocca immediatamente il secondo che tenta di entrare. Non ho bisogno della mutua esclusione qui esplicita per proteggere la scrittura, perchè la proprietà del prologo mi garantisce che lo scrittore è sempre 1. Quando lo scrittore termina di scrivere, non sono più uno scrittore attivo: se ci sono scrittori in coda ne sveglio uno, altrimenti se ci sono lettori in coda sveglio tutti i lettori. In realtà siccome la Signal e la Broadcast, se non c'è nessuno in coda non hanno effetto mi sono risparmiato un if. Però a voi l'ho risparmiato anche per non allungare troppo la soluzione di questo testo. Se voi dovete scrivere del codice un po' più efficiente, vi conviene scrivere un if perché vi risparmiate una chiamata di sistema per fare una Signal (in questo caso è una broadcast). Diciamo che non è scorretto se non la metto lì.

N filosofi si ritrovano a cena in un ristorante cinese, occupando un tavolo circolare, apparecchiato con un piatto per ogni filosofo e un bastoncino interposto fra ogni coppia di piatti adiacenti.



Per mangiare, ogni filosofo deve avere a disposizione i due bastoncini che si trovano rispettivamente alla sua sinistra e alla sua destra, entrando così in competizione con i due filosofi che siedono ai suoi lati. Il filosofo che non riesca ad ottenere ambedue i bastoncini (perché almeno uno è in possesso di un suo vicino) si sospende in attesa di ottenerli.

Abbiamo il problema dei filosofi, quindi n filosofi che si trovano ad un tavolo circolare, con un piatto a testa e una bacchetta a testa e per poter mangiare hanno bisogno di acquisire entrambe le bacchette, questo crea una situazione di conflitto all'accesso alle risorse perché non ci sono bacchette sufficienti per far mangiare tutti contemporaneamente, in particolare se un filosofo sta mangiando, questi due che gli stanno affianco dovranno attendere perché non hanno la disponibilità della bacchetta, una alla sua sinistra, l'altra alla sua destra. Il protocollo che seguono i filosofi è questo: fanno un ciclo infinito, quando ad un certo punto decidono di mangiare, eseguono una sequenza di azioni per acquisire le bacchette, a quel punto possono mangiare, quando hanno terminato devono rilasciare e questo è lo schema. Quando poi hanno rilasciato le bacchette, pensano, fanno altro, e poi ad un certo punto ritornano in cima, gli tornerà fame e torneranno ad acquisire e cercheranno nuovamente di mangiare.

Diamo un'occhiata a questa soluzione:

Soluzione 1

Filosofo i

```
...
while (true) {
        // il filosofo di indice i decide di mangiare: protocollo per mangiare //
    lockBastoncino[i].Acquire();
        //il filosofo i si sospende se non può ottenere il bastoncino situato alla sua sinistra //
     lockBastoncino[(i+ 1) mod N].Acquire();
        // il filosofo i si sospende se non può ottenere il bastoncino situato alla sua destra //

    < il filosofo di indice i mangia >

        //il filosofo di indice i  ha finito di mangiare: protocollo per pensare //
    lockBastoncino[i].Release();
    lockBastoncino[(i+ 1) mod N].Release();
        // il filosofo rilascia i due bastoncini situati alla sua sinistra e alla sua destra //
}
```

Questa soluzione utilizza un array di variabili di mutua esclusione, quindi un array di Lock che si chiama lockBastoncino e ogni variabile di Lock è associata ad un bastoncino. Questa soluzione è SBAGLIATA, non va certamente in Deadlock, il Deadlock si può presentare in un caso molto sfortunato, quando il primo filosofo acquisisce il primo bastoncino (quello che gli sta alla destra), quindi fa la Lock_acquire() su questo bastoncino, viene deschedulato, passa in esecuzione quest altro filosofo, che fa la Lock_acquire() su questo

bastoncino, viene deschedulato, passa in esecuzione quell'altro filosofo, che fa la Lock_acquire() su quest'altro bastoncino (di destra),viene deschedulato, passa in esecuzione questo filosofo che normalmente fa la Lock_acquire() sul suo bastoncino di destra e da questo momento in poi i thread ancora non sono in stallo, nel momento nel quale il prossimo thread farà l'acquire del suo bastoncino di sinistra, si sospenderà. Via via si sospenderanno tutti gli altri, quando tutti gli altri thread avranno richiesto l'acquisizione, avranno fatto la Lock_acquire() per il loro bastoncino di sinistra, saranno tutti sospesi e il sistema sarà in stallo. Quindi abbiamo una situazione in cui possiamo vedere lo stato di stallo, lo stato di Doomed, quindi uno stato nel quale ancora i thread non sono in stallo ma ci andranno certamente e questo è lo stato dove ogni thread ha acquisito il suo bastoncino di destra ma non ha ancora acquisito quello di sinistra, quindi questo è lo stato dannato, andremo certamente in stallo. "Riuscite a vedere quali sono gli stati sicuri e insicuri?" Quand'è che si transita da uno stato sicuro ad uno stato insicuro? La domanda è lecita, perché inizialmente lo stato è sicuro, nessun filosofo ha acquisito il lock. Ed è chiaro che se io permetto di acquisire entrambi i bastoncini li può mangiare, poi smette di mangiare, poi andranno avanto gli altri secondo un certo ordine, quindi lo stato iniziale è certamente sicuro. Quindi evidentemente se possiamo andare in stallo c'è un istante nel quale passiamo da uno stato sicuro ad uno insicuro. **Qual è questo istante?** Non stiamo parlando di filosofi che stanno applicando l'algoritmo del banchiere, perché altrimenti non andrebbe in stallo. Quindi non stiam parlando di filosofi che controllano le conseguenze delle loro azioni, stiamo parlando dei filosofi che eseguono esattamente questo codice, secco e duro. Quindi abbiamo visto che da questo codice si può arrivare ad uno stallo, siccome inizialmente lo stato è sicuro e però possiamo arrivare a uno stallo, evidentemente in quella sequenza di azioni che ci porta allo stallo, c'è un punto dal quale passiamo da uno stato sicuro ad uno insicuro e vi sto chiedendo qual è questo istante. **Soluzione**: quando il terzo ha preso la bacchetta di destra e viene deschedulato, abbiamo tre bacchette impegnate ma nessuno dei quattro filosofi è ancora sospeso, quindi quando descheduliamo il terzo non è mica detto che viene messo per forza in esecuzione il quarto e se viene messo in esecuzione il quarto, magari lui vuole ancora pensare, quindi pensa e non piglia nessuna bacchetta. Questo ancora non è uno stato insicuro perché può darsi che venga messo in esecuzione a questo punto nuovamente il terzo, ovvero viene messo in esecuzione il primo, il primo si sospende perché trova la bacchetta di sinistra occupata, il secondo si sospende perché trova la bacchetta occupata, viene rimesso di nuovo in esecuzione il terzo che trova la bacchetta di sinistra libera, il terzo mangia e a questo punto finiranno tutti felicemente. C'è un altro errore importante: la transizione da stato sicuro a stato insicuro non può mai avvenire quando un thread viene deschedulato, la transizione da stato sicuro a insicuro avviene sempre quando qualcuno prende qualcosa, quando c'è un'acquisizione di risorse perché quello è l'elemento delicato ai fini dello stallo. Posso ancora applicare l'algoritmo del banchiere, anche se vengo deschedulato in mezzo all'acquisizione di una risorsa e di riferire ad altri di acquisire quelle risorse che manderebbero tutti in stallo quindi quando io poi torno in esecuzione posso completare a fare ciò che dovevo fare, quindi la transizione non è mai in circostanze diverse dall'acquisizione di una risorsa. Però se questa soluzione non funziona, "siete in grado voi di proporre una soluzione alternativa che funzioni? Vi do un po' di suggerimenti: dovete trovare una rappresentazione conveniente per la struttura dati che rappresenta lo stato di allocazione delle bacchette e dovete eseguire una funzione di acquisizione delle bacchette che permetta ai thread di acquisirle entrambe contemporaneamente e poi di rilasciarle entrambe contemporaneamente perché la vera chiave di questo problema è il fatto che questa soluzione, le bacchette non vengono acquisite in un'azione atomica ma vengono acquisite separatamente, quindi il nostro problema è rendere l'acquisizione atomica, per rendere l'acquisizione atomica avete già tutti gli strumenti, avete le Lock e le variabili di condizione". Ogni volta che si propone una soluzione l'attegiamento normale che uno ha è quello di convincersi che la soluzione è corretta, è invece vi invito a fare esattamente il contrario, cercando di dimostrare che la soluzione è sbagliata, perché se non ci riuscire almeno quella soluzione ha passato un test di robustezza. Quindi chiedo: "Nell'ipotesi in cui uso una spinlock_acquire() all'inizio e una spinlock_release() alla fine, questa soluzione ci convince? Riusciamo a mettere in difficoltà, riusciamo a produrre uno stallo? È un *errore* ed è importante capire dove stia il problema". Immaginiamo questo: arriva

un primo thread, quindi un primo filosofo, il primo filosofo acquisisce la Spinlock_acquire() e a questo punto siccome è il primo che arriva, è chiaro che i bastoncini saranno liberi e lui potrà acquisire, uscire e mangiare. Quindi il primo filosofo certamente non avrà mai problemi. Se dovete cercare un problema, lo dovete cercare negli altri filosofi. Se il primo filosofo riesce ad acquisire due bastoncini, a fare la release e ad andare avanti, cosa può succede al secondo filosofo che arriva ad eseguire la Spinlock? Trova una spinlock libera ma il bastoncino occupato, quindi il primo filosofo acquisisce i bastoncini, trova la spinlock libera e acquisice il bastoncino di sinistra e di destra, libera la spinlock e inizia a mangiare, arriva quest altro filosofo, acquisisce la spinlock, cerca di acquisire il bastoncino di destra ma si sospende perché il bastoncino di destra è occupato. Ora succede che tiene impegnata la spinlock. Questo vuol dire che tutti gli altri thread che cercano di accedere alla struttura dati si bloccano perché non possono entrare, ogni volta che vado ad eseguire la spinlock per acquisire o liberare i bastoncini si trovano bloccati.

In generale quando dobbiamo affrontare un problema del genere, prima di tutto è importante trovare una struttura dati che rappresenti lo stato delle risorse che ci aiuti. Se ragioniamo bene sulla struttura dati, troviamo una struttura che rappresenta bene lo stato delle risorse, questo problema non diventa piu' artigianato (soluzioni trovate, specifiche per il problema, ma poco universali), ma diventa un'industria, cioè c'è una soluzione chiave, netta e semplice, da adottare sempre, uno schema di soluzione pressocchè universale che va solo adattato alle nostre esigenze. Quindi il primo grosso problema è trovare una struttura dati che rappresenti lo stato, in realtà possiamo rappresentare lo stato di questi filosofi e delle bacchette in questa maniera: possiamo rappresentare lo stato delle bacchette come libere o occupate, o possiamo rappresentare lo stato dei filosofi come "mangiano, pensano o stanno aspettando le bacchette". Nell'esempio precedente, quello che non funzionava, come stavo rappresentando lo stato delle bacchette? Stavo usando lo stato delle variabili di lock che implicito non lo posso osservare nel mio codice. Quindi in realtà in questa soluzione il problema è che lo stato del sistema è si rappresentato, ma non è manipolabile da me che devo gestire l'acquisizione/il rilascio delle bacchette, è tutto quanto implicito nello stato delle variabili di lock che sono gestite però dal nucleo del sistema operativo, quindi non posso in anticipo andare a vedere se una bacchetta è libera o è occupata in questo modo, perché dovrei andare a vedere lo stato di una variabile di lock. Primo suggerimento: lo stato lo rappresentate con una struttura dati che voi potete gestire e controllare, questo significa consumare piu' memoria perché dovete allocare una struttura dati, in questo caso è gestita implicitamente, ma non è un problema, è molto piu' importante scrivere la soluzione corretta che allocare una manciata di bytes. Quindi troviamo una rappresentazione adeguata, quella che vi propongo è una rappresentazione di questo tipo:

Il problema

In questa soluzione i filosofi condividono il vettore *stato[i]* (i= 0 .. N-1), dove lo stato del filosofo di indice i può assumere i valori *"ha fame", mangia", "pensa"*, con il seguente significato:

- *stato "ha fame" == > richiede i due bastoncini*
- *stato "mangia" == > possiede i due bastoncini*
- *transizione "mangia" → "pensa" == > rilascia i due bastoncini*

Si utilizzano inoltre le seguenti variabili:

- Lock mutex: per la mutua esclusione sul vettore *stato;*
- Cond attesaFilosofo[N]: vettore di variabili di condizione utilizzate per la sospensione dei filosofi. La variable attesaFilosofo[i] è usata per l'attesa del filosofo di indice i.

Un vettore che rappresenta lo stato dei filosofi, per ogni filosofo lo stato assume i valori "mangia", "pensa", "sta aspettando le bacchette". Siccome rappresento la struttura dati con questo stato, con questo array,

per decidere se posso mangiare o meno, devo andare ad analizzare questo array, a leggerne il contenuto, capire se posso mangiare, e se posso mangiare modificarlo di conseguenza. Lo stesso lo devono fare tutti i filosofi, quindi questa struttura dati che rappresenta lo stato complessivo del sistema, è una struttura dati condivisa, di conseguenza non può essere utilizzata liberamente ma dev'essere utilizzata in mutua esclusione. Quindi mi serve una Lock di mutex per poter avere il diritto di andare a leggere o scrivere quella struttura dati. C'è un altro problema: se per caso io filosofo scopro, leggendo la struttura dati, che non posso mangiare, cosa mi deve succedere? Dovrò sospendermi in attesa. Il meccanismo che mi sospende in attesa di una condizione, son le variabili di condizione, quindi mi servono anche delle variabili di condizione. Nel mio caso decido di utilizzare un vettore di n variabili di condizione, una per ogni filosofo, in maniera tale che posso fare le sveglie personalizzate, ma la stessa soluzione la posso fare anche con una sola variabile di condizione. Adesso che abbiamo stabilito la struttura dati, possiamo scrivere il codice per acquisire il diritto di mangiare e per rilasciare le bacchette, quindi per acquisire le bacchette e per rilasciare le bacchette.

Soluzione – filosofo i

Protocollo per mangiare

```
...
while (true) {
        // il filosofo di indice i decide di mangiare: protocollo per mangiare //
    mutex.Acquire();
    stato[i]= HaFame;
    while  (stato[(i- 1) mod N] == mangia) || (stato[(i+ 1) mod N] == mangia )   {
                attesaFilosofo[i].Wait(&mutex);
    }
    stato[i]= mangia;      //ha ottenuto ambedue i bastoncini //
    mutex.Release();

    < il filosofo di indice i mangia >
        // il filosofo di indice i  ha finito di mangiare: protocollo per pensare //
```

Quindi il filosofo dovrà pensare, poi esegue un protocollo per acquisire le bacchette, poi mangia e poi esegue un protocollo per rilasciare le bacchette. Questi due protocolli per acquisire le bacchette e per rilasciare le bacchette devono iniziare e finire con una mutex_acquire e una mutex_release, questo è meccanico perché quest'analisi del vettore di stato è una sezione critica e quindi deve essere protetta, ho dichiarato la variabile di mutex apposta per questo. Questa variabile di mutex deve essere la stessa per tutti (quindi NON bisogna utilizzarne una per ogni filosofo), quindi non posso avere una mutex per ogni filosofo perché tutti i filosofi stanno manipolando la stessa struttura dati e quindi tutti quanti si devono scontrare contro questa barriera e deve essere la stessa per tutti. Una volta che ho acquisito la mutua esclusione, metto lo stato in "ha fame" perché il filosofo era in stato "pensa", adesso vuole acquisire le bacchette, non le ha ancora acquisite perché sta verificando se le acquisirà o meno, allora preventivamente mette il suo stato a "ha fame", perché vuol dire che se resta a "ha fame" sarà in attesa di acquisire le bacchette appunto. Quand è che questo filosofo deve bloccarsi in attesa? Deve bloccarsi in attesa se le bacchette a destra e a sinistra sono occupate, vuol dire che il filosofo di sinistra e quello di destra stanno mangiando, quindi se il filosofo ddi sinistra e quello di destra stanno mangiando allora ci sospendiamo su una variabile di condizione, il filosofo si sospende sulla sua variabile di condizione, ma come vi dicevo si può fare anche utilizzando una variabile di condizionamento. Se invece il vicino di sinistra e di destra non stanno mangiando, allora il filosofo può prendere le bacchette e mangiare. Quindi il suo stato diventa "mangia" e

questo significa che ha acquisito le due bacchette e a questo punto rilascia la mutua esclusione e può mangiare. Quando ha finito di mangiare deve rilasciare le bacchette.

Soluzione – filosofo i

Protocollo per pensare

```
mutex.Acquire();
stato[i]=pensa;
if (stato[(i - 1) mod N]== HaFame) && (stato[(i - 2) mod N]<> mangia) {
        // riattiva il filosofo (i-1) mod N se può ottenere entrambi i bastoncini //
    stato[(i - 1) mod N] = mangia;
     attesaFilosofo[(i- 1) mod N].Signal(&mutex);
}
if (stato[(i + 1) mod N]== HaFame) && (stato[(i + 2) mod N]<> mangia) {
        // riattiva il filosofo (i+1) mod N se può ottenere entrambi i bastoncini //
    stato[(i + 1) mod N]= mangia;
     attesaFilosofo[(i + 1) mod N].Signal(&mutex);
    }
mutex.Release();
}
```

Per rilasciare le bacchette, nuovamente, deve modificare lo stato della struttura dati, quindi dovrà acquisire la mutua esclusione, modificare la struttura dati specificando che adesso lui ha smesso di mangiare, sta pensando, quindi stato[i] = “pensa”, dopodiché deve riattivare gli altri filosofi, potrebbe fare una Broadcast banalmente, oppure fa una Signal puntuale sul filosofo da riattivare, per esempio: se il mio vicino di sinistra ha fame e certamente se ha fame stava aspettando sicuramente me perché la bacchetta la tenevo impegnata io, uno delle due bacchette gliela tenevo impegnata io, quindi se il mio vicino di sinistra ha fame e quello che gli sta alla sinistra non sta mangiando, vuol dire che tutte e due le bacchette per lui son libere e allora lo posso riattivare con una Signal. Stessa cosa per il vicino di destra: se il mio vicino di destra ha fame e il vicino che gli sta a destra non sta mangiando, allora lo posso riattivare, rilascio la mutua esclusione e ho finito. Alcune considerazioni tra questa soluzione e le soluzioni che hanno proposto i colleghi: per quanto riguarda la soluzione che previene lo stallo ordinando le risorse, quindi che prevede che l’ultimo filosofo abbia un codice differente. È corretta, si può utilizzare, non presenta particolari svantaggi ai fini della prestazione, però NON è uno schema di soluzione generale che posso adottare in tanti contesti derivati in qualche maniera dai filosofi o mescolanze dei filosofi con altri problemi, è difficile, perché adottare quella soluzione significa ogni volta reinventare la specifica per quel problema. C’è un problema ulteriore: in questa soluzione tutti i filosofi fanno la stessa cosa tranne uno che fa una cosa differente, questo vuol dire che quando voi la andate a scrivere, dovete scrivere del codice concorrente, dovete scrivere n thread che fanno tutti la stessa operazione e un thread che fa un’operazione differente. L’ordinamento va bene ed è corretto però costringe il programmatore che deve sviluppare del codice concorrente ad avere un thread che si comporta diversamente dagli altri e quindi a scrivere un codice differente per un thread. Che succede poi se il numero dei thread cambia o un thread viene ucciso, c’è una gestione dinamica dei thread, ci sono tutta una serie di complicazioni dietro, che in questo caso non abbiamo perché il comportamento di tutti i thread è perfettamente *simmetrico*. La soluzione con gli spinlock va anche bene ma presenta problemi di prestazioni. Immaginiamo che il primo filosofo abbia acquisito i bastoncini e stia mangiando, arriva il secondo thread, quello che gli sta a destra, fa la spinlock_acquire(), trova un bastoncino impegnato e si sospende su questo bastoncino mantenendo bloccata la spinlock. Abbiamo vista prima che ai fini dello

stallo questo non è un problema, perché quando il thread che sta mangiando avrà smesso di mangiare, lascerà il bastoncino, il nostro thread verrà riattivato, acquisirà tutti e due i bastoncini e andrà a mangiare pure lui. Il problema è un altro, che è un problema di prestazioni, non di Deadlock. Immaginiamo che questo filosofo non stia mangiando, mentre mangia viene deschedulato, passa in esecuzione questo filosofo, acquisisce la spinlock e si sospende per cercare di acquisire questo bastoncino. C'è qualche altro filosofo che può mangiare ora? No. Ci son filosofi che in teoria potrebbero mangiare? Si, tanti. Il problema è che quello schema di soluzione non glielo permette perché fa una sospensione all'interno della mutua esclusione senza rilasciare la mutua esclusione. Se ci fate caso, è proprio per questo che sono state inventate le variabili di condizione. Quindi qui è scorretto fare quest'operazione, qui avrei dovuto dire "mi sospendo se il bastoncino è occupato, ma rilascio la spinlock". Il problema è se usate la spinlock non avete un meccanismo di variabili di condizione associato per gestirla, perché non c'è, dovete usare una lock_acquire() regolare, spinlock o lock non fa alcuna differenza, e qui dovete usare la Wait e quindi dovete guardare lo stato dei bastoncini. Altri errori tipici che qualcuno fa, quando facciamo la acquire abbiamo un protocollo per acquisire il bastoncino, spesso uno pensa: "va bene, ma prima di impelagarmi in questa questione di capire se devo fare la lock, se devo sospendermi, etc, vado a testare lo stato della variabile per capire se mi devo sospendere o meno in anticipo, quindi lo faccio fuori da lock, capita spesso di vedere IF(possoMangiare) then mangia() else mutex lock". Quindi capita spesso di vedere dei test o degli accessi alle strutture condivise che rappresentano lo stato fuori dal Lock e questo è sbagliatissimo, capita in altre circostanze di vedere delle soluzioni dove si cerca di guardare se ci sono filosofi in coda sulla Wait e guardarlo però fuori dalla mutex_acquire(), anche questo è sbagliato, non potete vedere se ci son filosofi sospesi sulla Wait perché la Wait è una struttura del nucleo, non è visibile, non la potete esplorare. Tutto ciò che riguarda le variabili di condizione e i Lock sono informazioni private, interne, non possono essere guardate al di fuori delle operazioni previste per quelle strutture. Perché questa soluzione permette a piu' filosofi di mangiare contemporaneamente? Non è una proprietà di stallo per cui devo analizzare tutti i casi possibili, basta far vedere che in alcuni casi effettivamente questa proprietà vale. Supponiamo che arrivi il primo filosofo, il primo filosofo trova tutto libero, va in stato di "mangia", quindi blocca il bastoncino di sinistra e di destra, rilascia la mutua esclusione ed entra a mangiare. Quindi il primo filosofo acquisisce la mutua esclusione, acquisisce il diritto di mangiare, rilascia la mutua esclusione e mangia con la mutua esclusione *libera.* Arriva il secondo thread, è il mio vicino di sinistra. Il mio vicino di sinistra acquisisce la mutua esclusione, scopre che non può mangiare, si sospende, ma rilascia la mutua esclusione, quindi non impedisce a qualsiasi altro filosofo di acquisire la mutua esclusione per vedere se può mangiare. Se arriva un filosofo che non ha niente a che vedere con me, che non è né alla mia sinistra, né alla mia destra, questo filosofo può acquisire la mutua esclusione perché mentre io mangio non è bloccato, può verificare che i suoi bastoncini sono liberi, li può acquisire, ed entra a mangiare, quindi in questo momento stiamo mangiando sia io che lui. L'unico momento nel quale le cose non si possono fare contemporaneamente è quando acquisiamo i bastoncini, o li acquisisco io o li acquisisce lui, ma non possiamo acquisire contemporaneamente i bastoncini, questo è l'unico vincolo che poniamo. E per lo stesso motivo poniamo il vincolo che non si possono acquisire e rilasciare contemporaneamente i bastoncini, se uno acquisce, nessuno può rilasciare, se uno sta rilasciando, nessuno può acquisire, però questo è un vincolo necessario per la mutua esclusione, perché per rilasciare e per acquisire dobbiamo agire su una struttura dati condivisa, questo è il minimo che dobbiamo pagare, meno di questo esistono delle soluzioni ma non le vedremo in questo corso e sono molto specialistiche e limitate ad alcuni campi.

Ora si passa a vedere un esercizio sulle Upcall (su richiesta di un tizio).

ESERCIZIO - Upcall

Si consideri un processore che dispone dei registri speciali PC (program counter) e PS (program status), dello stack pointer SP e dei registri generali R1 e R2. In stato utente, ogni processo dispone di uno stack ad uso generale e di uno stack riservato per gestire le upcall chiamato *Signal Stack* (per semplicità assumiamo che ogni processo abbia un unico

thread).

Per notificare una upcall ad un processo, il nucleo salva il program counter, lo stack pointer del processo e tutti i registri generali nel *Signal Stack*, quindi modifica lo stack pointer per puntare al Signal Stack, e il program counter per puntare alla procedura di gestione della upcall. La upcall si conclude ripristinando i registri generali dal Signal Stack, e quindi con l'istruzione RETU che ripristina PC e SP sempre dal Signal Stack.

Al tempo t, il nucleo riceve la richiesta di notifica di una upcall al processo P che corrisponde alla funzione di gestione che inizia all'indirizzo 3100 (nello spazio di memoria del processo). In questo istante di tempo, il signal stack punta alla locazione 2A19, il processo è in stato di pronto e il contenuto dei suoi registri speciali e generali è conservato nel suo descrittore come mostrato in tabella.

DESCRITTORE DI P	
Stato	Pronto
PC	5F80
PS	16F2
SP	FA00
R1	56A9
R2	52BE

Mostrare:

a) Il contenuto dei descrittori, dei registri generali e speciali, dello stack del nucleo e lo stato del processore durante la fase di estrazione della prima istruzione della funzione di gestione della upcall;
b) Il contenuto dei descrittori, dei registri generali e speciali, dello stack del nucleo e lo stato del processore durante la fase di estrazione dell'istruzione RETU con la quale termina la upcall;
c) Il contenuto dei descrittori, dei registri generali e speciali, dello stack del nucleo e lo stato del processore durante la fase di estrazione dell'istruzione eseguita subito dopo la RETU.

Consideriamo un processore che c'ha i registri PC, PS, SP, due registri generali. Ogni processo dispone di uno stack per le operazioni normali e poi invece uno stack per gestire i segnali (Signal Stack), in questo esercizio per semplicità supponiamo che ogni processo abbia un singolo thread. Come funziona la notifica delle upcall a un processo: il nucleo salva il Program counter, lo Stack pointer del processo e i registri generali nello Signal Stack. Perché deve fare questa operazione? L'upcall viene inviata dal nucleo del sistema operativo al processo, quando il processo è in uno stato di pronto/attesa, non quando è in esecuzione, perché se è in esecuzione allora il nucleo non è in esecuzione e quindi non può inviargli materialmente l'upcall. Il nucleo invia l'upcall al processo, come conseguenza dell'upcall il processo esegue una funzione di handler, ma al termine di questa funzione di handler il processo deve tornare ad eseguire l'istruzione normale, quella che stava eseguendo prima quando è stato interrotto. Quindi la gestione della terminazione dell'upcall non può essere gestita con l'assistenza del sistema operativo, non c'è un iRet qui, bisogna che sia gestita con il supporto del linguaggio, quindi bisogna che il termine dell'handler dell'upcall sembri con un ritorno da procedura/funzione. Il ritorno da funzione opera andando a prendere i registri PC, PS dalla cima dello stack, per questo motivo andiamo a mettere queste informazioni alla cima dello stack. Fatto questo il nucleo deve preparare tutto quanto per restituire il controllo al processo, quindi metterlo in esecuzione, però deve essere messo in esecuzione a partire dall'handler e utilizzando lo Signal Stack, non utilizzando lo stack del processo, assumo che il termine dell'handler, il supporto a tempo di esecuzione del linguaggio, abbia messo tutto quello che serve per uscire dall'handler correttamente, prendere le informazioni dallo stack e ritornare all'esecuzione del programma. L'upcall si conclude ripristinando i registri generali dello Signal stack quindi con l'istruzione RETU. A questo punto al tempo t, il nucleo riceve la richiesta di segnalazione di una upcall verso il nostro processo e questa richiesta di invio dell'upcall corrisponde a mettere in esecuzione l'handler che si trova alla locazione 3100, ovviamente nello spazio di memoria del processo, l'handler è una funzione utente. Quando arriva questa notifica di upcall, lo Signal stack punta alla locazione 2A19 (questo ci serve per configurare correttamente l'upcall), il processo è in stato di pronto e questo è il suo descrittore con i registri generali. Dobbiamo mostrare il contenuto dei

descrittori, dei registri generali, etc, durante la fase di estrazione della prima istruzione della funzione di gestione dell'upcall, quindi siamo nel momento nel quale il nucleo ha già fatto l'iRet, quindi siamo già in stato utente e il processore si sta apprestando ad eseguire la prima istruzione dell'handler. Poi dobbiamo mostrare il contenuto del processore, lo stato del processore del sistema, nel momento nel quale stiamo chiudendo l'ultima istruzione dell'handler, stiamo facendo questa RETU per chiudere l'handler, e poi il contenuto dei descrittori, dei registri, etc, subito dopo aver fatto questa RETU, quindi essere ripartiti ad eseguire l'istruzione del processo che ci si sarebbe aspettati di eseguire fin da subito, quindi il processo nel punto in cui era stato interrotto quando è stato passato in stato di pronto. Quello che dovete fare è riempire queste tabelle. Questo esercizio, essendo tale, semplifica tutta una serie di questioni per proporre qualcosa di simile da fare, normalmente queste upcall vengono gestite appoggiandosi molto al modo nel quale il linguaggio gestisce la terminazione delle funzioni e il ritorno delle funzioni. Facciamo tutta una serie di ipotesi semplificative, una delle quali è la RETU che in realtà non esiste ma ai fini dell'esercizio si, un'altra è il fatto che comunque mettiamo subito in esecuzione i thread quando arriva la notifica di upcall. Che cosa succede quindi quando mandiamo la upcall e mettiamo in esecuzione quel processo: intanto il descrittore cambia perché dobbiamo mettere in stato esecuzione, il contenuto del descrittore in realtà non deve cambiare perché non è modificato il suo stato, il processo si ferma lì. Quello che sta succedendo in realtà è che di questo processo stiamo mettendo in esecuzione una sua funzione, un suo handler, in maniera asincrona rispetto alla sua esecuzione, quindi questo processo era in attesa di eseguire l'istruzione all'indirizzo 5F80, però in realtà lo mandiamo ad eseguire l'istruzione all'indirizzo 3100 che è l'istruzione dell'handler, quindi questo valore di PC, PS, SP, R1, R2, in realtà devono essere utilizzati quando termineremo di eseguire la funzione dell'handler (3100). Per mettere in esecuzione l'handler dobbiamo quindi caricare PC (indirizzo iniziale dell'handler), nella PSW dobbiamo prendere la PSW del processo, perché tra l'altro codifica il fatto che siamo in stato utente a interruzioni abilitate e via discorrendo. Lo stato del processo di conseguenza è stato utente e nei registri SP, R1, R2 mettiamo: in SP, visto che stiamo eseguendo l'handler e ci dobbiamo predisporre per la corretta terminazione dell'handler che dovrà ritornare ad eseguire il codice alla locazione 5F80, allora dovremo mettere un po' di informazioni in cima allo stack per la gestione dei segnali e di conseguenza lo Stack Pointer deve puntare alla cima dello stack dei segnali, invece i valori di R1, R2 non sono significativi a differenza degli altri casi nei quali non ci sono le upcall, perché non stiamo mettendo in esecuzione il processo nel punto in cui lui aveva interrotto l'esecuzione, ma stiamo eseguendo una upcall da zero, quindi stiamo eseguendo un handler da zero, i valori dei registri generali per questo handler non hanno nessun significato, comunque l'handler li ripulirebbe e inizierebbe la sua esecuzione pulita. Siccome quando terminerà l'handler però dobbiamo ripristinare lo stato che si trova nel descrittore di P, lo stato del descrittore lo andiamo a mettere nello stack dei signali e in particolare ci mettiamo PC, SP, PSW, R1, R2, a questo punto passa in esecuzione l'handler, quando l'handler si appresta alla terminazione, alla terminazione dell'handler dobbiamo riprendere l'esecuzione del processo esattamente dove era stata interrotta l'istruzione 5F80 con i registri R1, R2, per cui le ultime istruzioni dell'handler, le ha scritte il linguaggio e prevedono il ripristino di R1, R2 dalla cima dello stack per cui la cima dello stack restano soltanto PC, SP che devono essere ripristinati assieme nell'ultima istruzione, quella che chiude l'handler (RETU). A questo punto il PC conterrà l'indirizzo finale dell'handler (non sappiamo quale). Fatto questo eseguiamo la RETU, la RETU prende come scritto nel testo dell'esercizio, i valori di PC e SP dalla cima dello stack e li copia in PC e SP del processore e in questo modo abbiamo ripristinato correttamente l'esecuzione del processo a partire dal punto in cui era stato interrotto.

Soluzione

a) Contenuto dei descrittori, dei registri generali e speciali, dello stack del nucleo e lo stato del processore durante la fase di estrazione della prima istruzione della funzione di gestione della upcall:

DESCRITTORE DI P	
Stato	Esecuzione

SIGNAL STACK DI P	
2A19	5F80

REGISTRI SP, R1, R2	
SP	2A15

PC	5F80		2A18	FA00		R1	??
PS	16F2		2A17	56A9		R2	??
SP	FA00		2A16	52BE			
R1	56A9		2A15				
R2	52BE		2A14				

PROCESSORE: Registri speciali e stato							
PC	3100		PS	16F2		Stato	Utente

b) Contenuto dei descrittori, dei registri generali e speciali, dello stack del nucleo e lo stato del processore durante la fase di estrazione dell'istruzione RETU con la quale termina la upcall:

DESCRITTORE DI P			SIGNAL STACK DI P			REGISTRI SP, R1, R2	
Stato	Esecuzione		2A19	5F80		SP	2A17
PC	5F80		2A18	FA00		R1	56A9
PS	16F2		2A17			R2	52BE
SP	FA00		2A16				
R1	56A9		2A15				
R2	52BE		2A14				

PROCESSORE: Registri speciali e stato							
PC	3100+??		PS	16F2		stato	Utente

c) Contenuto dei descrittori, dei registri generali e speciali, dello stack del nucleo e lo stato del processore durante la fase di estrazione dell'istruzione eseguita subito dopo la RETU.

DESCRITTORE DI P			SIGNAL STACK DI P			REGISTRI SP, R1, R2	
Stato	Esecuzione		2A19			SP	FA00
PC	5F80		2A18			R1	56A9
PS	16F2		2A17			R2	52BE
SP	FA00		2A16				
R1	56A9		2A15				
R2	52BE		2A14				

PROCESSORE: Registri speciali e stato							
PC	5F80		PS	16F2		stato	Utente

Sistemi operativi 20/04/17

Oggi parliamo dello scheduling. Lo scheduling sappiamo che è quella attività che il sistema operativo deve svolgere per allocare il processore. Nel momento nel quale, per un motivo o per l'altro il processore si libera il sistema operativo esegue un algoritmo, un programma, un proprio codice che quello dello scheduler per decidere, tra tutti quanti i thread presenti nel sistema, quale è il prossimo al quale assegnare il processore, quindi il prossimo che andrà in esecuzione. Questo algoritmo ovviamente è soggetto ad una serie di vincoli e di aspetti legati alle prestazioni; c'è da vedere, per capire come avviene lo scheduling, un po' di definizioni, legate a quelli che sono gli aspetti prestazionali, soprattutto dello scheduling, che riguardano il tempo di risposta, il (???), il (???), e via discorrendo.

Main Points

- Scheduling policy: what to do next, when there are multiple threads ready to run
  - Or multiple packets to send, or web requests to serve, or ...
- Definitions
  - response time, throughput, predictability
- Uniprocessor policies
  - FIFO, round robin, optimal
  - multilevel feedback as approximation of optimal
- Multiprocessor policies
  - Affinity scheduling, gang scheduling
- Queueing theory
  - Can you predict a system's response time?

Quello che vediamo sono soltanto tecniche di scheduling per il caso dell'uniprocessor, non vediamo il caso del multiprocessor. In effetti l'ultimo algoritmo di scheduling che vediamo che è lo scheduling a code multiple è anche adottato nei sistemi multiprocessore. Dal punto di vista dello scheduling ci sono un po' di cose con le quali lo scheduler ha a che fare: i task, i job. Che cosa sono?

Spesso in passato ho utilizzato il termine "task" o "job" in maniera più libera, in questo caso parlando dello scheduling, mi riferisco ai task job relativamente a dei compiti specifici che un qualche thread deve svolgere in risposta a una precisa richiesta dell'utente. Quindi, per esempio, quando l'utente inserisce qualcosa in un form e preme invio questo scatena un'azione da parte di un thread che probabilmente vorrà anche fare altro, però nell'immediato dovrà produrre una risposta a quello che l'utente sta chiedendo. Oppure quando con il mouse cliccate sul pulsante per avviare una certa procedura, quella procedura è quella che qualche thread dovrà eseguire e con quelle identifichiamo il task. Chiaramente quindi il task è un sottoinsieme, è una certa azione svolta da un thread, eventualmente anche da più thread in cooperazione che va a produrre un risultato. Quello che succede è che quando l'utente vi chiede l'esecuzione di un task specifico al sistema si aspetta che questo task venga eseguito nel modo più rapido possibile, si aspetta di avere una risposta più rapidamente possibile. Non solo: generalmente l'utente si aspetta anche di avere una certa proporzionalità in questa risposta. Qualsiasi utente, lavorando col computer, sa che ci sono certe operazioni che comportano un carico di lavoro maggiore (e quindi la risposta arriverà dopo) e ci sono invece delle operazioni che richiedono una risposta immediata. Se voi iniziate a scuotere il mouse voi vi aspettate che la freccina segua in maniera molto precisa e accurata i movimenti che voi fate

con la mano. Qualsiasi ritardo in questi movimenti viene percepito dall'utente come uno scatto, come una carenza nelle prestazioni de sistema. D'altra parte se l'utente dà un comando premendo un pulsante che è associato ad un'operazione che lui sa essere lunga e complessa e che richiede del tempo in questo caso l'utente è disposto ad aspettare del tempo. Quindi bisogna che la risposta che il sistema dà alle richieste dell'utente, quindi all'esecuzione dei task, avvenga in tempi compatibili con quelle che sono le esigenze dell'utente e con tempi proporzionali rispetto alla complessità dei task. Dei task hanno in effetti una latenza, un tempo di risposta, il tempo che passa dal momento nel quale l'utente richiede l'esecuzione di quel task all'istante nel quale quel task è completato e l'utente ha la sua risposta.

Definitions

- **Task/Job**
  - User request: e.g., mouse click, web request, shell command, ...
- **Latency/response time**
  - How long does a task take to complete?
- **Throughput**
  - How many tasks can be done per unit of time?
- **Overhead**
  - How much extra work is done by the scheduler?
- **Fairness**
  - How equal is the performance received by different users?
- **Predictability**
  - How consistent is the performance over time?

Questo è ciò che riguarda la percezione dell'utente. Dall'altra parte, dal punto di vista del sistema, il sistema può avere più utenti contemporaneamente attivi che possono fare diverse richieste o lo stesso utente contemporaneamente può fare più richieste al sistema, quindi può richiedere l'attivazione di più task. Quindi dal punto di vista del sistema è anche importante la capacità di poter portare a compimento il numero maggiore di task per unità di tempo: questo è il throughput. E' la quantità di task che riusciamo a completare per una certa unità di tempo. Il throughput era estremamente importante nei sistemi batch. È ancora importante nei sistemi attuali, nel momento nel quale il sistema deve rispondere a più stimoli, a più richieste contemporaneamente. Se voi pensate ad un web server il suo throughput è un parametro molto importante, perché misura la quantità di richieste di pagine che il web server riesce a far uscire per unità di tempo (e quindi presumibilmente la quantità di utenti che riesce a soddisfare per unità di tempo). Per fare ciò che deve fare lo scheduler, ovviamente, deve fare dei calcoli, delle elaborazioni, gestire delle strutture dati: questo comporta un overhead. Quindi ai fini dello scheduling è importante l'overhead, perché questo è ciò che noi vogliamo minimizzare. Idealmente noi vogliamo avere uno scheduling che ha 0 overhead: questo non è ovviamente possibile perché lo scheduler deve comunque poter eseguire per fare delle scelte. Tenete presente che l'overhead non è soltanto il tempo di esecuzione dell'algoritmo di scheduling, ma riguarda anche effetti indiretti dello scheduling. Mi spiego meglio: quando lo scheduler decide di interrompere un thread in esecuzione e ne mette un altro in esecuzione, il passaggio da un thread all'altro comporta una commutazione di contesto che non è azione svolta dallo scheduler, ma è l'effetto indiretto di una scelta fatta dallo scheduler. Questo comporta un overhead dell'algoritmo di scheduling. Quindi quanto più frequentemente l'algoritmo di scheduling comporta la commutazione di contesto, tanto maggiore sarà il suo overhead. E l'overhead non finisce lì perchè la commutazione di contesto sappiamo anche che ha associato un overhead, quindi una perdita di tempo, dovuta al fatto che le cache del

sistema del processore non sono aggiornate. Il sistema di gestione della memoria, come vedremo più avanti nel resto del corso, non ha immediatamente a disposizione tutte le informazioni aggiornate e quindi questo causa ulteriori rallentamenti. Quindi quando si valuta uno scheduler dobbiamo valutare sì il suo tempo sulle prestazioni di thread, ma valutiamo anche il suo overhead perché questo ha un effetto sulle prestazioni complessive del sistema.

Altri aspetti sono l'aspetto di equità e di predicibilità. Equità perchè vorremmo che lo scheduler mantenga, offra a tutti i thread lo stesso tipo di servizio, le stesse possibilità. Ovviamente non è desiderabile uno scheduler che sistematicamente penalizza senza alcun motivo alcuni thread a favore di altri. Apparentemente può sembrare scontato e ovvio definire uno scheduler che garantisce proprietà di equità, ma in realtà, come vedremo, non è così semplice perchè intervengono diversi aspetti, diverse proprietà dei thread che poi comportano delle problematiche che devono essere gestite dalle scheduler (di questo ne parleremo tra un pochino). Predicibilità è quello che vi dicevo prima: vorremmo che lo scheduler non alteri le prestazioni dei thread, per cui un thread che impiega un certo tempo ad eseguire un certo task vorremmo che impieghi sempre lo stesso tempo. Non vogliamo che un thread in certi casi impieghi molto meno tempo e in certi altri casi sia rallentato a dismisura. Poi ci sono un altro po' di definizioni che ci servono.

More Definitions

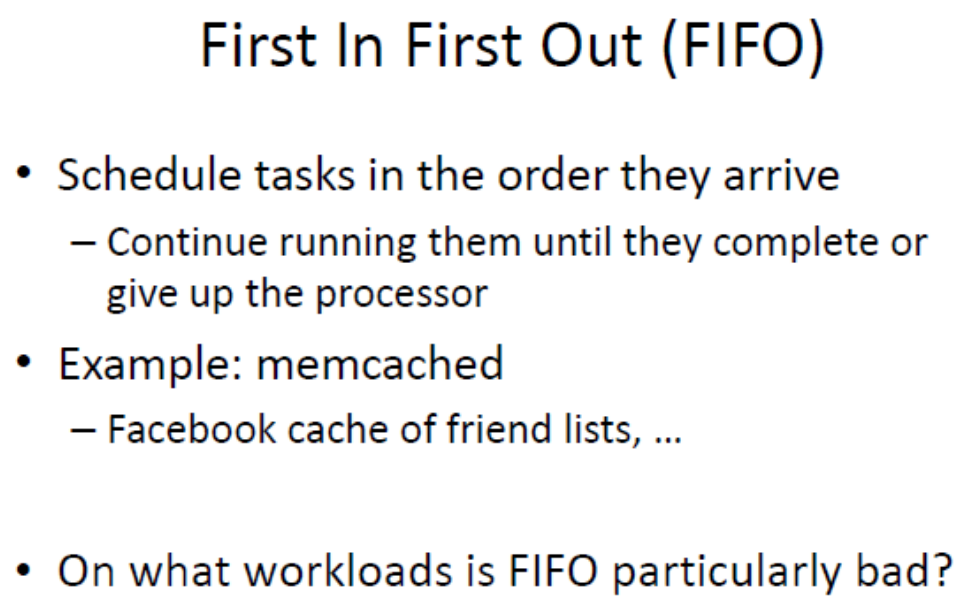
- **Workload**
  - Set of tasks for system to perform
- **Preemptive scheduler**
  - If we can take resources away from a running task
- **Work-conserving**
  - Resource is used whenever there is a task to run
  - For non-preemptive schedulers, work-conserving is not always better
- **Scheduling algorithm**
  - takes a workload as input
  - decides which tasks to do first
  - Performance metric (throughput, latency) as output
  - Only preemptive, work-conserving schedulers to be considered

Workload, o insieme dei task, che sono presenti nel sistema e che ai quali lo scheduler deve dare una risposta, quindi deve in qualche maniera gestire e assegnare il processore per poterli portare a compimento. Poi lo scheduler può essere con o senza prerilascio, questa è una proprietà dello scheduler e riguarda il fatto che lo scheduler ha la possibilità di forzare il rilascio anticipato del processore da parte di un thread. Tutti i sistemi moderni utilizzano lo scheduling con prerilascio, quindi ci sembrano un dato scontato,

però non è sempre stato così. In certi casi tutt'ora non è così. Il fatto di avere uno scheduling con prerilascio significa che lo scheduler ha la facoltà, quando lo ritiene opportuno di revocare l'assegnazione del processore ad un thread e di riassegnarlo ad un altro thread. Per poter realizzare uno scheduling con prerilascio serve ovviamente avere un supporto da parte del processore, da parte del SO per realizzare la commutazione di contesto e questo è tutto quello che abbiamo visto nella parte iniziale del corso. Poi riguardo allo scheduler desideriamo che sia word conserving o detto in termini molto semplici: non vogliamo che lo scheduler lasci dei tempi morti sul processore, ma vogliamo invece che lo scheduler massimizzi l'utilizzo del

processore. Anche questo può essere un fatto scontato, può sembrare ovvio. Se il processore è libero sembra ovvio poterlo assegnare ad un thread per poterlo sfruttare immediatamente. Chiaramente il fatto di avere uno scheduler work-conserving, generalmente, va a massimizzare le prestazioni del sistema, perchè tende ad utilizzare il più possibile il processore e quindi a portare avanti più task. Ci possono essere dei casi molto particolari nei quali lo scheduler non è work-conserving: in certi casi potrebbe non essere conveniente per lo scheduler riassegnare il processore quando è in idle, ma gli può convenire tenerlo in idle per riassegnarlo più rapidamente ad un thread che si deve riattivare, magari a breve termine. D'altra parte questo tipo di scheduling sono scheduling a utilizzo speciale (i quali in realtà non ci interessano in questo corso, per cui non ne parleremo). In questo corso parleremo di scheduling prevalentemente con prerilascio, farò qualche cenno a scheduling senza prerilascio e soprattutto work-conserving. A questo punto l'algoritmo di scheduling è un algoritmo che prende come input un workload, quindi una serie di task da eseguire (questi task in effetti sarà la lista dei thread pronti). L'algoritmo decide quale è il task da mandare in esecuzione per primo, decide quanto tempo, eventualmente, dovrà stare in esecuzione se con prerilascio sulla base di metriche prestazionali, per esempio throughput o latenza. Questo è quello che vedremo in queste due ore.

Quando si parla di scelte tra task (quindi thread) da mandare in esecuzione, il primo algoritmo ovvio che vi viene in mente è l'algoritmo FIFO. Questo algoritmo viene in mente, ovviamente,



perché è uno di quelli più usati anche nella vita comune: se andate al supermercato o alle poste l'ordine naturale è quello FIFO. Non è usato soltanto in quei contesti: per esempio un web server se ha delle richieste in ingresso è naturale che utilizzi l'ordine FIFO per evaderle. E quindi per associare un thread che dia una risposta (ci sono anche esempi su Facebook citati ecc.). La questione riguardante il FIFO è: questo algoritmo di scheduling per i nostri scopi, va bene oppure no? Tenendo conto di tutti i paramentri prestazionali dei quali vi ho dato le definizioni prima, il punto è: l'algoritmo FIFO è davvero ragionevole, è davvero valido in tutte le circostanze oppure ci sono delle circostanze nelle quali l'algoritmo FIFO non è desiderabile? Sicuramente l'algoritmo FIFO ha dei pregi: è "fair", è equo. Garantisce lo stesso tipo di trattamento: un thread sa che quando arriva ci sono i thread che sono arrivati prima di lui che dovranno essere eseguiti e poi quando arriverà il suo turno lui passerà in esecuzione. Quindi dà a tutti lo stesso tipo di trattamento e non produce situazioni di attesa indefinita. Un thread sa che dovrà aspettare ad andare in esecuzione un tempo proporzionale ai thread che sono nella coda pronti prima di lui. Quindi sicuramente questo tipo di scheduling ha una sua

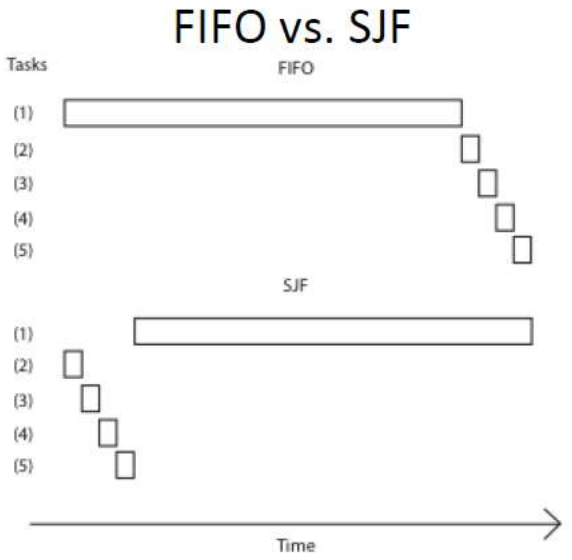
motivazione: è utilizzato non a caso perchè almeno ha queste due proprietà molto importanti e molto utili. Riuscite ad individuare dei casi nei quali l'algoritmo di scheduling FIFO non è adeguato? O potrebbe aver dei limiti? (*Silenzio di tomba*). A me è capitato un caso la settimana scorsa di scheduling FIFO inadatto. (*Il Chessa strozza*). È più facile di quanto pensate.
<Uno studente risponde>
"Ci sono Job più pesanti o altri più leggeri? Magari ci sono dei lavori che richiedono più tempo di altri ad essere eseguiti che però sono eseguiti prima: potremmo invece eseguire prima quelli più veloci."
...
Ma ad esempio quando scegliete degli esami da dare per primi voi state facendo scheduling. Voi sapete che ad ogni esame è associata una quantità di lavoro che dovete dedicare e sapete che la strategia vincente è fare gli esami più pesanti con i compitini, poi all'appello successivo avete poco tempo per studiare e vi limitate ad un esame più piccolo o complementare. In questo caso non state utilizzando uno scheduling FIFO. Quanti di voi hanno seguito nello scheduling degli esami esattamente la lista presentata nelle pagine del Dipartimento? Nessuno. Quindi nessuno di voi ha fatto scheduling FIFO. Quello che dice il vostro collega è giustissimo. La settimana scorsa mia moglie mi ha mandato di corsa al supermercato a prendere un pacco di uova. Sono corso al supermercato, ho preso le uova e sono andato alla cassa: mi sono ritrovato in una fila con delle persone con carrelli carichi di roba. Il tizio davanti, molto gentile, mi ha detto "Eh, si vede che sua moglie l'ha mandata a prendere le uova. Passi avanti." Questo è un esempio calzante con ciò che diceva il vostro collega: eseguire prima alcuni task brevi, in alcune circostanze è un vantaggio perché ovviamente il tempo che impiegate ad eseguire i task nella loro interezza è sempre lo stesso. Che io abbia pagato le uova prima o dopo della persona con il carrello più pesante non cambia il tempo totale di lavoro della cassiera, però ha migliorato le cose perché il tempo di risposta complessivo medio si è abbassato notevolmente. In particolare: il mio tempo di completamento è diventato piccolissimo; il tempo di completamento della persona che mi ha fatto passare avanti è aumentato, ma di una frazione per lui impercettibile, davvero minima, perché lui avrebbe comunque dovuto aspettare un tempo consistente. Quindi, in generale, non è una cattiva idea quando avete un carico molto disomogeneo mettere in esecuzione dei task più brevi e dopo fare i task più lunghi. Questo probabilmente lo fate anche quando dovete studiare o comunque nella gestione del vostro tempo. A me capita spesso: se so che devo fare diverse cose è più semplice per me concettualmente concentrarmi prima sui task brevi che posso liberare rapidamente, sgombrare la mente e poi dedicarmi ai task più lunghi e più complessi. Quindi ci sono delle situazioni oggettive nelle quali il FIFO non è particolarmente adeguato, indicato. Per questo motivo ci sono degli algoritmi di scheduling diversi dal FIFO. Proprio sulla base di quello che diceva il vostro collega, è più comodo eseguire i task brevi prima, perché in questo modo riduco il tempo di completamento medio. Proprio sulla base di questa osservazione è nato l'algoritmo Shortest Job First (SJF).

Shortest Job First (SJF)

- Always do the task that has the shortest remaining amount of work to do
  - Often called Shortest Remaining Time First (SRTF)

- Suppose we have five tasks arrive one right after each other, but the first one is much longer than the others
  - Which completes first in FIFO? Next?
  - Which completes first in SJF? Next?

Questo algoritmo, in realtà, è stato tradotto in algoritmo di scheduling già agli albori dei sistemi operativi, nel momento nel quale i sistemi erano, sostanzialmente, sistemi Batch dei quali bisognava portare i Job all'inizio della giornata (vi ho già descritti questi sistemi all'inizio del corso). Chiaramente i sistemisti che facevano a mano, in realtà, lo scheduling si sono resi conto subito che era molto più conveniente mettere in esecuzione prima i Job brevi rispetto ai Job lunghi. Loro lo potevano fare perchè, per questioni di Debug, i Job che venivano portati al centro di calcolo erano associati a delle informazioni che il programmatore dava: il programmatore portava il Job e diceva: "Questo Job impiega 2 ore per essere eseguito.. quest'altro dovrà impiegare 10 minuti.. ". Quindi le informazioni sul tempo di completamento, in questi Job, è nota a priori. Di conseguenza era abbastanza comodo poter fare uno scheduling di tipo SJF, mandando in esecuzione di Job brevi prima. In questo modo cosa succede quello che si è detto prima: il tempo di completamento medio dei lavori, dei Job si riduce. Il tempo odi completamento di un Job, di un Task è il tempo che passa dall'istante nel quale l'utente lo offre al sistema all'istante nel quale l'utente prende il risultato. Quindi in questo tempo di completamento non incide soltanto il tempo effettivo di esecuzione, ma incide anche il tempo di attesa, che è quel Task/Job/Thread che deve passare prima di essere messo in esecuzione.



Se confrontiamo il FIFO con SJF succedono cose di questo genere: se ci si presenta un carico di lavoro disomogeneo, con dei Task molto lunghi e dei Task brevi e disgraziatamente il Task lungo capita all'inizio, prima di poter completare i Task brevi, devo aspettare il completamento del Task lungo. Quindi il tempo medio di completamento è più o meno la metà di questo intervallo. Se invece manda in esecuzione prima i task brevi e poi il task lungo, il mio tempo di completamento diventa pari a quasi un quinto del tempo totale. È evidente che con l'SJF miglioro le cose. Il punto è: posso dimostrare che l'SJF è ottimo dal punto di vista del tempo di completamento? Esiste un algoritmo migliore che può ridurre il tempo di completamento oppure l'SJF è ottimo sotto questo punto di vista? L'SJF è ottimo. Perché? Suggerimento: quali sono i casi nel quale l'SJF è ottimo? Risposta: quando tutti i Task hanno esattamente lo stesso tempo di esecuzione. In questo caso, il FIFO diventa equivalente all'SJF, banalmente. Domanda: in quale caso il FIFO è pessimo? Risposta: quando l'ordinamento è strettamente decrescente, quindi quando il primo task è il più lungo e poi via via l'ultimo task è il più breve. E l'SJF? Esso ottimizza il tempo di completamento medio, quindi il Turnaround, ma ha altri problemi? Sì, esso è esposto al problema dell'attesa indefinita. Bisogna stare un po' attenti riguardo l'SJF, dipende in quali condizioni noi pensiamo di applicarlo. Se voi applicate l'SJF in condizioni statiche, per cui avete l'insieme di Task da eseguire (questo insieme è statico) e lo organizzate, lo ordinate secondo l'algoritmo SJF e lo mandate in esecuzione ovviamente in questo caso non si ha Starvation. Il problema è il caso nel quale, invece, l'SJF viene usato in modo dinamico: i Task possono arrivare a tempo di esecuzione in maniera dinamica, si aggiungono alla lista dei task da mandare in esecuzione e in questo caso l'SJF interviene per continuare a riordinare l'elenco dei Task da mandare in esecuzione. Quindi in questo caso, come dice il vostro collega, se continuano ad arrivare Task brevi, questi continuano a passare avanti, i Task più lunghi rimangono in coda e i questo caso abbiamo una situazione di attesa indefinita dei Task lunghi, perchè non sappiamo se e quando questi verranno messi in esecuzione, non possiamo fare più previsioni. L'SJF, quando è nato, è nato per il contesto statico; successivamente è stato utilizzato anche in modo dinamico, in questo caso prendeva il nome di Shortest Remaining Time First (che sostanzialmente è un SJF dinamico). Tornando all'SJF, questa è la dimostrazione che minimizza il tempo del Turnaround. (Immagine mancante).

Vediamo come si calcola. Supponiamo di avere 4 job: A, B, C, D, i quali hanno tempo di esecuzione appunto a, b, c, d. Poi supponiamo che la sequenza di scheduling sia A, B, C, D. Il tempo di Turnaround del Job "A", è "a": esso viene messo in esecuzione all'istante 0 e dopo un tempo pari ad A ha terminato. Il Job "B", invece, deve aspettare prima il completamento di A e poi l'esecuzione di B, quindi il suo completamento è "a+b". Il Job C dovrà aspettare, per completare, un tempo pari ad "a+b+c" e il Job D dovrà aspettare un tempo pari ad "a+b+c+d". Questa è l'ipotesi nella quale tutti i Job arrivano all'inizio, contemporaneamente, vengono schedulati e poi ogni Job aspetterà il suo turno; quindi il suo tempo di completamento sarà dato dal tempo di attesa più il suo tempo di esecuzione. Se noi sommiamo tutto questo il tempo di completamento medio è dato dalla somma di tutto questo diviso 4, quindi sarà 4a+3b+2c+d diviso 4. Questa quantità quando è minimizzata? Essa è banalmente minimizzata quando il parametro che ha coefficiente massimo è il più piccolo. Quindi quando si ha a < b < c <d.
Questo è esattamente il caso dello scheduling SJF.
Abbiamo quindi visto che il problema principale dell'SJF è la starvation, se è utilizzato in un contesto dinamico. Esso ha però anche altre questioni , altre problematiche accessorie, in particolare: la variazione nel tempo di esecuzione è molto aleatoria, dipende molto da quello che stanno facendo gli altri thread. Se mi trovo davanti una serie di Job più piccoli il mio tempo di completamento si allunga; se invece non ho Job più piccoli davanti il mio tempo di completamento si accorcia e questo comporta un'alta variabilità nei tempi di completamento, una scarsa predicibilità.

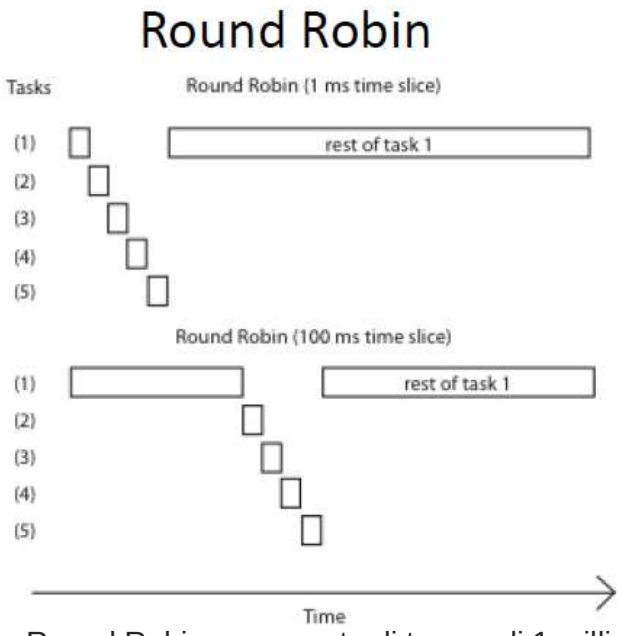
In effetti l'SJF è stato in voga per un po' di anni, ma è stato surclassato soprattutto sulla spinta di una serie di evoluzioni che sono capitate nel mondo dell'informatica dei sistemi operativi degli anni '60 grosso modo. E' capitato che hanno iniziato a diffondersi i video terminali. Quindi gli utenti, a questo punto, avevano la possibilità di interagire direttamente con i loro programmi. Nel mondo precedente, senza video terminali, in realtà l'interazione con il calcolatore non c'era. Il programmatore scriveva il suo programma, predisponeva, affiancati al programma, una serie di dati sui quali il programma doveva andare in esecuzione. Tutto questo veniva raccolto in un insieme di schede perforate, lo portava al centro di calcolo e l'unica interazione era alla fine, quando il programma era completato l'utente riceveva la risposta. In questo contesto un SJF aveva assolutamente senso. Con l'introduzione dei video terminali la situazione cambia radicalmente: intanto i Task arrivano dinamicamente, col fatto che l'utente interagisce con il video terminale in qualsiasi istante può richiedere l'esecuzione di un Task. Quindi ci troviamo in una condizione nella quale
l'SJF deve essere applicato in maniera dinamica e non statica, come prima. In secondo luogo, il programma non parte necessariamente avendo i dati a disposizione a priori, ma potrebbe richiederli all'utente, potrebbe interagire con l'utente per chiedere degli ulteriori dati o per sapere come procedere che direzione prendere (quello che normalmente facciamo quando oggi interagiamo con un'interfaccia grafica). Questo fa sì che si aggiunge un'ulteriore variabile all'algoritmo di scheduling (l'utente), la quale è veramente una variabile impazzita. Il problema è che a queste condizioni l'obiettivo principale dello scheduling non è più quello di massimizzare le prestazioni del sistema, ma diventa quello di massimizzare la soddisfazione dell'utente, il che vuol dire che se l'utente mi dà un comando o se l'utente preme un tasto sulla tastiera devo cercare di eseguire, di dare un feedback, una risposta in tempi più rapidi possibili e comunque compatibili con la complessità dell'operazione che l'utente mi sta chiedendo. Quindi, messo tutto quanto questo sulla brace, è venuto fuori che l'SJF non era chiaramente più utilizzabile come algoritmo di scheduling e si è dovuto pensare a qualche cosa di nuovo. Questo qualche cosa di nuovo in realtà l'abbiamo già visto quando vi citavo i sistemi Time Sharing ed è l'algoritmo di Round Robin.

Round Robin

- Each task gets resource for a fixed period of time (time quantum)
  - If task doesn't complete, it goes back in line
- Need to pick a time quantum
  - What if time quantum is too long?
    - Infinite?
  - What if time quantum is too short?
    - One instruction?

Nell'algoritmo del Round Robin si lavora in questa maniera: ad ogni thread/task associo un quanto di tempo e lascio in esecuzione quel task (gli assegno il processore) soltanto per quel quanto di tempo. Se alla fine del quanto di tempo quel task non ha terminato poco male: lo metto in fondo alla coda dei thread pronti, assegno il processore ad un altro task per un altro quanto di tempo e vado avanti così.
In questa maniera ho la garanzia di ridistribuire le capacità del processore in maniera equa tra tutti quanti i thread, perchè ogni thread prende un quanto di tempo, sta in esecuzione quel quanto, dopodichè aspetta in coda un tempo proporzionale alla quantità dei thread che sono

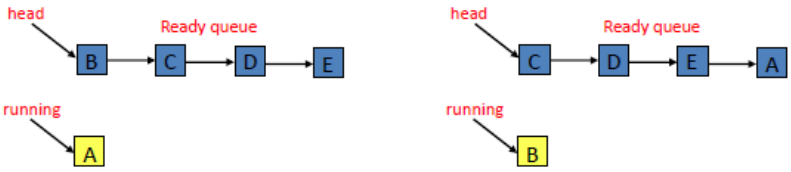
nella coda pronti, poi ritorna in esecuzione e va avanti. In questo modo è come se ogni thread prendesse una frazione del tempo del processore e quindi evolve, va avanti con una velocità che è una frazione del tempo della velocità di avanzamento del processore, dell'hardware. Questa frazione dipende dalla quantità di thread che sono presenti nel sistema: maggiore è il numero di thread nel sistema, più piccolo sarà la frazione di tempo che il mio thread prenderà per l'esecuzione. Questo è ragionevole: maggiore è il carico e più il sistema rallenta. Per utilizzare questo tipo di algoritmo di scheduling ho bisogno, ovviamente, di un po' di supporto dall'hardware, e questo è già stato introdotto negli anni '60 sotto forma di Timer abbinato al meccanismo di interruzione che permette di fare questo scheduling. A questo punto l'unico parametro dell'algoritmo di scheduling è il quanto di tempo: una volta che io adotto l'algoritmo Round Robin, l'unico parametro sul quale posso agire per modificarne le prestazioni, modificare il comportamento dello scheduling è la dimensione del quanto di tempo. Il punto è: quanto deve essere grande il quanto di tempo? Cosa succede se è troppo grande o troppo piccolo? <Una studentessa prova a rispondere> La vostra collega dice: "Se lo faccio troppo piccolo mi trovo a pagare un costo di commutazione di contesto che avviene alla fine di ogni quanto di tempo, quindi aumento l'overhead; questo mi fa capire che il quanto di tempo non può diventare troppo piccolo perchè altrimenti l'overhead rischia di diventare preponderante: perdo più tempo a fare commutazione di contesto che a far avanzare i thread. D'altra parte se faccio il quanto di tempo troppo grande ritardo il tempo che impiega un thread a tornare in esecuzione una volta che è finito in coda pronti. Quindi quando il mio thread va in esecuzione sono contento perchè un bel quanto di tempo lo posso far avanzare parecchio, però quando finisce il suo quanto di tempo, prima di tornare in esecuzione ci vorrà molto più tempo. Questo può essere un problema? Sì, perché se ho dei thread che devono interagire con l'utente tramite il video terminale degli anni '60, avere un quanto di tempo molto lungo significa che il thread che sta interagendo con quell'utente ha delle fasi in cui è particolarmente responsivo, e quindi può interagire rapidamente con l'utente, e poi ci sono delle fasi in cui l'utente preme un tasto e prima che abbia una risposta passa parecchio tempo. Quindi devo trovare un equilibrio tra quelli che sono i tempi di risposta e l'overhead, è un tradeoff che devo risolvere. Banalmente, se il quanto di tempo diventa infinito che succede al Round Robin? Succede che ricadiamo nel caso FIFO, con tutte le problematiche del FIFO. Vediamo che succede col Round Robin.



Supponiamo di avere un Round Robin con quanto di tempo di 1 millisecondo e siamo in una situazione in cui abbiamo 4 Task che hanno tempo di completamento di 1 ms e un Task che ha un tempo di 200 ms. Quello che succede è che il primo Task usa 1 quanto di tempo, poi passano in esecuzione gli altri Task, finita questa prima mandata ritorna in esecuzione il primo

Task, tutti gli altri sono completati e da questo momento in poi siccome non ci sono altri Task resta in esecuzione permanentemente fino al completamento del Task 1. Se invece utilizziamo un quanto di tempo di 100 ms, mettiamo in esecuzione il Task 1 per 100 ms, dopodichè i Task 2, 3, 4 possono passare in esecuzione subito dopo. Loro, tra l'altro, hanno un tempo di esecuzione molto breve (1 ms). Per cui non hanno bisogno di un quanto di tempo così lungo: terminano subito, si alternano rapidamente e poi torna in esecuzione il Task 1. Questi sono due esempi di comportamento del Round Robin considerando lo stesso Workload, quindi lo stesso set di thread. Vediamo quali sono gli elementi del Round Robin.

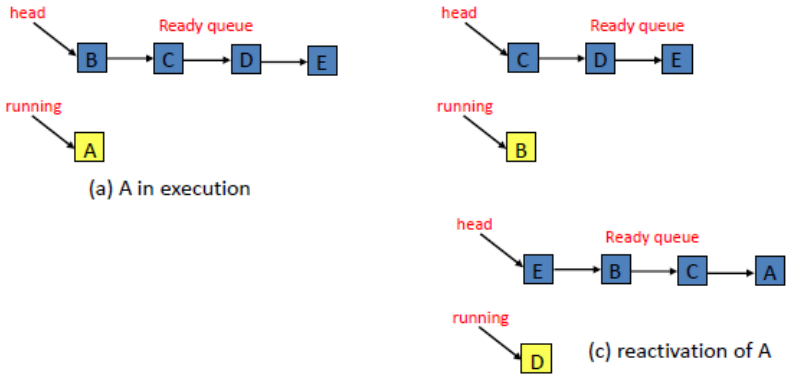
Round Robin



- Proportionality: turnaround (time spent in the system) proportional to task length
- Response time upperbounded by the number of processes
  - # ready processes * time share

Abbiamo una coda di thread in attesa, essa ha una cima, in questo caso è B e abbiamo un thread in esecuzione che è il thread A. Quando A completa il suo quanto di tempo viene messo in fondo alla coda e passa in esecuzione il thread B. In questo caso abbiamo un po' di proprietà interessanti perché il sistema diventa proporzionale: il tempo che un thread passa in attesa di tornare in esecuzione nella coda è proporzionale alla quantità di thread in coda (n°thread in coda moltiplicato il quanto di tempo). E' misurabile del proporzionale a carico del sistema. Il tempo di risposta è limitato dal numero di thread dei processi, perché se per caso, quando A è in fondo alla coda il thread A sta interagendo con l'utente (l'utente preme un tasto sulla tastiera), quel tasto verrà recepito dal thread A quando A tornerà in esecuzione. Per tornare in esecuzione ci vorrà un tempo che è proporzionale al numero di thread in coda. Quindi abbiamo un limite superiore al tempo di risposta che un thread può dare all'utente. Qui ci sono due casi del Round Robin (figura a pagina successiva): in questo caso A è in esecuzione e questa è la coda pronti. Se A va in stato di attesa, per esempio perché esegue una P() su un semaforo con valore 0 o esegue una wait() su una variabile di condizione, A, a questo punto, non compare più né nella coda pronti, né in esecuzione (andrà in una coda di attesa da qualche parte); la coda pronti a questo punto contiene solo C, D, E e in esecuzione ci va B (il primo della coda). Se invece quando poi A viene riattivato, quando qualcuno fa una V() sul semaforo, viene messo in fondo alla coda pronti. Alla riattivazione con il Round Robin il thread va sempre in fondo alla coda pronti, non c'è nessuna forma di priorità per i thread che vengono riattivati.

Round Robin



La fine del quanto di tempo è segnalata dalle interruzioni del timer: quando scatta il timer scatta l'interruzione, ritorna in esecuzione lo scheduler che attua la commutazione di contesto e prende il primo thread dalla coda. Quando lo scheduler fa questo e rimette un thread in esecuzione riattiva anche il timer per il prossimo quanto di tempo. Però non è tutto qui. Se per caso il thread si sospende o termina durante il suo quanto di tempo ovviamente lui avrebbe ancora del tempo da passare in esecuzione, ma questo tempo non lo può sfruttare perché si sospende: allora anche in questo caso passa in esecuzione lo scheduler che sceglie il prossimo thread da mandare in esecuzione. Quindi non ci sono tempi morti. Se un thread non

Round Robin

- End of time share signaled by the timer
  - Timer interrupt causes the activation of the scheduler
  - The scheduler restarts the timer
- Scheduler takes over also in case of suspension of the running process
  - Reassigns the CPU and restarts timer
- In current systems time share around 20-120 msec

usa tutto il suo quanto di tempo tanto meglio, il quanto di tempo del prossimo thread inizia prima (però dura sempre la stessa quantità). Nei sistemi attuali il quanto di tempo varia tra i 20 e i 120 ms, dipende un po' da quello che vogliamo ottenere. Tipicamente nei sistemi più interattivi, quindi se prendete il vostro sistema Android o se prendete un Laptop, Windows un computer con iOS, tipicamente il quanto di tempo è piccolo per privilegiare l'intera attività. Sui server, che invece devono svolgere spesso i compiti più in background, quindi di meno interesse in interattività, il quanto di tempo tende ad essere più grande. Ci sono server da cui vi potete aspettare quanti da 100-120 ms. Domanda: se supponiamo di non avere costi di overhead, quindi legati alla commutazione di contesto, il Round Robin è sempre meglio del FIFO? Bisogna pensare a cosa state misurando, quale è la vostra misura di prestazione? Pensate a cosa può essere rilevante misurare, quindi in base a cosa state misurando (cioè in base all'indice prestazionale) pensate se è meglio il Round Robin o il FIFO. <Uno studente>

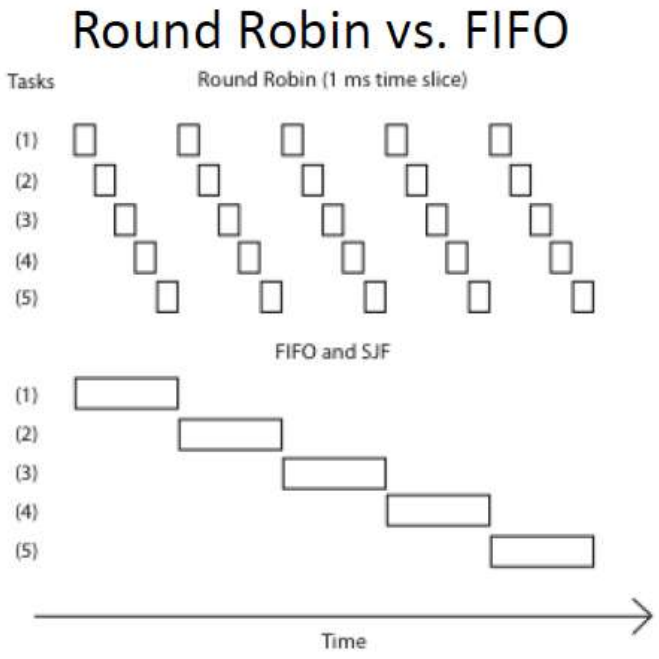
Se i Job sono più brevi rispetto al quanto di tempo assegnato il tempo rimanente è perso? <Chessa> No, non è perso, riparto subito con un altro Job. <Altre risposte random in sottofondo> La vostra collega vi ha dato un indizio: si sta concentrando sul tempo di completamento. Quindi possiamo valutare il tempo di completamento medio di una serie di thread. Prendete un Workload, quindi definite quanti sono i thread e quanto è il loro tempo di esecuzione e provate a schedularli col FIFO e con il Round Robin. Cosa è meglio?

Pausa

Round Robin vs. FIFO

- Assuming zero-cost time slice, is Round Robin always better than FIFO?

Stavamo cercando un caso in cui il FIFO può funzionare meglio del Round Robin: questo è il caso.



Abbiamo 5 Task che hanno tutti esattamente lo stesso tempo di esecuzione. In questo caso particolare il FIFO si comporta come l'SJF e ha le stesse prestazioni riguardo ai tempi di completamento. Vi ricordo che il tempo di completamento è la media di questi tempi qui. Quello che succede nel caso del Round Robin, invece, è che siccome tutti i Task hanno esattamente lo stesso tempo di esecuzione vengono messi in un ciclo di esecuzione dove vanno avanti per un quanto di tempo ognuno e però in questa maniera nessun Task finisce prima, ma finiscono tutti quanti più o meno nello stesso istante di tempo, con una breve differenza l'uno dall'altro. (???) il tempo di completamento medio è la media dei tempi di completamento di questi task. Questo ci insegna che quando ci sono tanti parametri da ottimizzare e tanti aspetti da considerare, non è più detto che sia possibile trovare un ottimo assoluto. Ci potrebbero essere dei casi nei quali un algoritmo funziona bene e invece altri casi nei quali lo stesso funziona peggio. La scelta deve riguardare diversi aspetti, non necessariamente un unico parametro prestazionale, ma un insieme di essi. La scelta dipende molto da quale è quello da noi ritenuto più importante. Quindi sebbene in alcuni casi il Round

Robin possa comportarsi peggio del FIFO, alla fine della storia gli algoritmi più utilizzati sono tutti quanti basati sul Round Robin (sono in effetti una sua evoluzione). Vuol dire che questo caso particolare non è ritenuto né un caso rappresentativo né in realtà quello sul quale si gioca la scelta. A vantaggio del Round Robin c'è in maniera molto più pressante e forte il fatto che è più adatto per i sistemi interattivi, perché garantisce un avanzamento uniforme di tutti quanti i thread e in questo modo non penalizza nessun processo interattivo. Quindi un'interazione si sviluppa per tutti gli quanti gli utenti in maniera equa, da questo punto di vista. C'è anche un'altra questione: il Round Robin è sempre equo? Questa è una domanda subdola.

Round Robin vs. Fairness

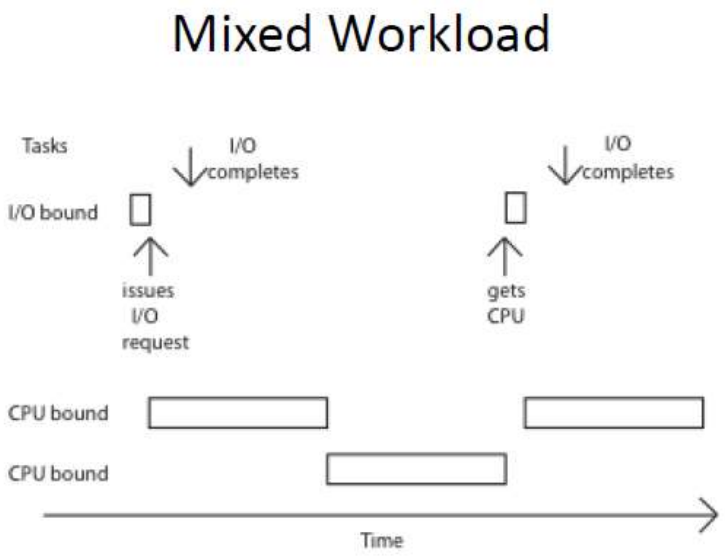
- Is Round Robin always fair?

<Uno studente> Dipende se abbiamo delle priorità dei thread. <Chessa> Non ci sono priorità. Il Round Robin, per definizione, non ha priorità. Tutti i thread sono trattati equamente dando ad ognuno lo stesso quanto di tempo. <Raffaele risponde> Magari può succedere che l'algoritmo del Round Robin non vada a prendere sempre allo stesso modo Task in attesa? Nel senso che non è pari nel riprendere i Task che sono in attesa. <Chessa> Ma in Task in attesa vanno sempre in una coda FIFO, quindi si estrae sempre dalla testa e si inserisce sempre in coda. Quindi quando un thread entra nella coda dei thread pronti, esso sa già quando uscirà, cioè quando saranno messi in esecuzione tutti i thread che gli stanno davanti alla coda. Chi gli sta dietro non gli può passare avanti, proprio per definizione di Round Robin. <Raf che spara altre fesserie> Ma se un task finisce prima, l'altro task avrà più tempo per essere eseguito. Se un task finisce prima.. <Chessa> Ma se un task ha finito vuol dire che ha finito. Chiariamo un attimo. Cosa succede quando un thread termina la sua esecuzione prima del quanto di tempo? Lui termina, libera il processore, si mette in esecuzione lo scheduler, prende il primo thread nella coda pronti e lo mette in esecuzione per un quanto di tempo. Quindi il quanto di tempo è sempre lo stesso, non è che si allarga per andare ad utilizzare il quanto di tempo lasciato libero da un altro thread. Vedo che un altro vostro collega sostiene invece che il Round Robin sia equo. Vi dicevo che la domanda è subdola perché il Round Robin fa tutto per essere equo: mette i thread nella coda che è gestita rigorosamente FIFO, il quanto di tempo è uguale per tutti, non c'è nessuna priorità. Il problema sta come al solito in ciò che viene dato in pasto al Round Robin. Andiamo a vedere i processi che possiamo avere nel sistema. Li possiamo classificare in due tipi (questa classificazione, tra l'altro, non ha un confine netto, è molto sfumata): possiamo individuare i processi di tipo CPU Bound e i processi di tipo I/O Bound.
Una parentesi: sto utilizzando il termine processo/thread/task in una maniera totalmente indistinguibile, ma non è molto importante in realtà. Quello che ci interessa è l'unità di schedulazione, che sia il processo, che sia il task ecc. è equivalente. Allora, potrei avere i processi che sono CPU Bound, ciò vuol dire che la loro velocità di esecuzione è limitata dalla velocità di esecuzione del processore. Sono processi che svolgono la maggior parte del loro

Mixed Workload

- Two kind of processes:
  - CPU-bound – long CPU bursts with sparse I/O
  - I/O bound – short CPU bursts with frequent I/O

compito sul processore. Sono, generalmente, processi che fanno molti calcoli. Un processo che fa moltiplicazione di matrici, che trova il determinante di una matrice è un processo CPU Bound. Viceversa ci possono essere processo I/O Bound, la cui velocità di esecuzione è limitata dalla velocità di input/output. Supponete di avere un processo che fa la copia di un file: per fare la copia di un file deve leggere un file da disco e poi scriverlo. Non fa calcoli, sul processore fa davvero poco. Quello che fa sono soltanto comandi al disco di lettura e di scrittura. La sua velocità di esecuzione è limitata dalla massima velocità di esecuzione dei comandi di lettura e scrittura su disco. È importante vedere queste classi di processi perché se noi li mettiamo in uno scheduling Round Robin possono succedere cose di questo genere.



Supponiamo di avere 3 processi, 2 sono CPU Bound, 1 è I/O Bound e supponiamo che il quanto di tempo sia lungo così. Supponiamo anche che per primo passa in esecuzione il processo I/O Bound: succede che esso esegue poche istruzioni sul processore e dà il comando per l'esecuzione di un'operazione di input/output. Fatto questo va in sospensione, non termina, ma si sospende: va in stato di attesa. A questo punto lo scheduler prende il primo processo in coda, che è CPU Bound e lo mette in esecuzione. In questo momento abbiamo due processi in parallelo in esecuzione: abbiamo un processo nel processore e un altro sul dispositivo. Successivamente il dispositivo completa l'operazione di I/O, manda l'interruzione al SO che riattiva il primo processo. La riattivazione del primo processo sospeso nel Round Robin significa che quel processo viene messo in fondo alla coda pronti: quindi viene messo in coda pronti dietro questo processo.
A questo punto questo processo va avanti nell'esecuzione e termina il quanto di tempo, interviene poi lo scheduler che prende il primo processo della coda pronti, che è questo qui. Resta in esecuzione tutto il quanto di tempo; quando finisce il quanto di tempo interviene lo scheduler che prende il primo della coda pronti: esso è il processo I/O Bound che viene messo in esecuzione. Questo processo esegue poche istruzioni, dà nuovamente un comando al dispositivo, si sospende e quando verrà riattivato ritornerà in coda pronti. Se facciamo due calcoli vediamo che, in realtà, questo processo I/O Bound per tornare in esecuzione sta aspettando 2 quanti di tempo. I processi CPU Bound invece aspettano poco più di un quanto di tempo per tornare in esecuzione. Quello che sta succedendo è che in realtà i processi CPU Bound si stanno avvantaggiando, stanno sfruttando le risorse (il processore) molto più del processo I/O Bound. e quest'ultimo di conseguenza viene rallentato. Se voi fate questo gioco, aumentando il numero di processi CPU Bound, vi renderete conto che la penalizzazione del processo I/O Bound aumenta: i processi I/O Bound ritardano quasi nulla i processi CPU Bound. Mentre invece i processi CPU Bound ritardano molto il ritorno in esecuzione dei processi I/O Bound. Quindi da questo si capisce che in realtà lo scheduling Round Robin non è fair se abbiamo un carico misto di I/O Bound e CPU Bound. Il problema è, a questo punto, come facciamo a gestire questi carichi misti: tali carichi non sono cose strane, la maggior

parte dei sistemi hanno a che fare con un carico di questo genere. I processi I/O Bound sono processi che devono sì gestire tanto input/output, un processo che va a stampare, un processo che gestisce trasferimenti di informazioni da e per il disco, ma sono anche processi interattivi; un processo che aspetta che l'utente prema i tasti della tastiera o che muova il mouse è un processo I/O Bound. È speciale perché è interattivo, però rimane sempre un I/O Bound. I processi CPU Bound invece sono quelli che fanno calcoli. D'altra parte i processi possono cambiare il loro comportamento nel tempo: se voi siete appassionati di elaborazione video e avete un'applicazione per fare manipolazione/editing di video, il processo che voi state utilizzando di che tipo è? Finché state facendo editing, decidete dove tagliare, copiare, incollare, cucire ecc, è un processo pesantemente interattivo. Quando però poi date il comando di fare la codifica in un certo formato improvvisamente diventa un processo CPU Bound; quando poi ha terminato di fare la codifica ridiventa nuovamente un processo interattivo. Il comportamento di un processo non è prevedibile, può cambiare nel tempo e normalmente i sistemi hanno a che fare con processi con questa doppia identità. Trovare un algoritmo di scheduling che sia equo in queste condizioni è importante.
L'approccio è quello di Max/Min Fairness.

Max-Min Fairness

- How do we balance a mixture of repeating tasks:
  - Some I/O bound, need only a little CPU
  - Some compute bound, can use as much CPU as they are assigned
- One approach: maximize the minimum allocation given to a task
  - Schedule the smallest task first, then split the remaining time using max-min

L'idea è: cercare, prima di tutto, di isolare i processi CPU Bound e schedulare loro e poi il tempo rimanente ridividerlo tra i processi CPU Bound. Quindi abbiamo 2 problemi: il primo problema è quello di individuare le caratteristiche del processo, se appartiene ad una classe o all'altra; il secondo problema è quello di avere una politica di scheduling che dia la priorità ai processi I/O Bound.

Multi-level Feedback Queue (MFQ)

- Goals:
  - Responsiveness
  - Low overhead
  - Starvation freedom
  - Some tasks are high/low priority
  - Fairness (among equal priority tasks)
- Not perfect at any of them!
  - Used in Linux (and probably Windows, MacOS)

Per risolvere il secondo problema si utilizzano algoritmi di tipo Multi-Level Feedback Queue, che sono algoritmi utilizzati da Windows (anche Linux li utilizza in qualche forma). L'obiettivo è migliorare la responsività (il tempo di risposta, il tempo che impiega un thread a ritornare in esecuzione quando è uscito in coda pronti), tenere basso l'overhead, evitare la starvation e gestire la priorità alta o bassa per venire in contro alla questione dello scheduling dei thread CPU Bound, I/O Bound. La cosa curiosa è che con tutti questi obiettivi non c'è un algoritmo che può ottimizzarli tutti, non esiste proprio. Questo algoritmo a code multiple trova un compromesso su tutti questi obiettivi: esso non fa particolarmente male, non fa danno, però in realtà non è ottimo su alcuno di essi. Gli algoritmi che ottimizzano uno di questi aspetti generalmente sono disastrosi sotto gli altri punti di vista.

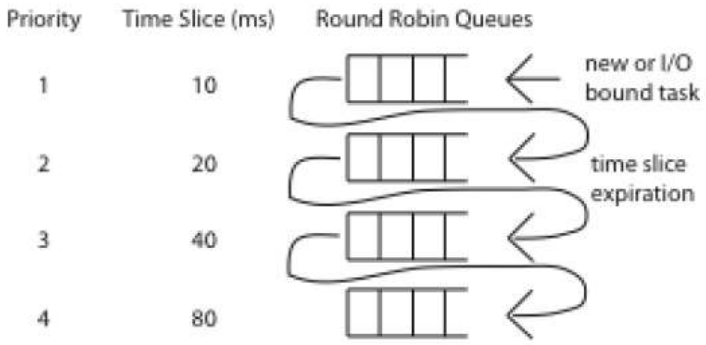
MFQ

- Set of Round Robin queues
  - Each queue has a separate priority
- High priority queues have short time slices
  - Low priority queues have long time slices
- Scheduler picks first thread in highest priority queue
  - Round robin in each queue
- Tasks start in highest priority queue
  - If time slice expires, task drops one level

Utilizziamo una serie di code Round Robin: non una sola coda pronti, ma tante code pronti ognuna gestita con politica Round Robin. Ogni coda è associata ad un livello di priorità. Quindi si inizia a schedulare i thread che stanno nella coda di priorità più elevata e soltanto quando questa coda si svuota si vanno a schedulare i thread delle code di priorità più bassa. L'algoritmo non è finito qui, perché il comportamento dei thread cambia nel tempo. Quindi bisogna dei meccanismi che prevedano il passaggio dei thread da una coda all'altra. 19.00 Quando il thread viene creato inizialmente sta, per esempio nella coda di priorità maggiore. In questa coda di priorità maggiore i quanti di tempo sono più piccoli perché essi sono più adatti per gestire i thread I/O Bound; idealmente dovrebbero essere thread I/O Bound quelli che stanno nelle code di priorità maggiore. Se un thread inizia ad utilizzare tutto il suo quanto di tempo vuol dire che probabilmente non è un thread I/O Bound, ma è CPU Bound con qualche gradazione di CPU Bound. Per cui se un thread usa tutto il suo quanto di tempo viene scalato nella coda inferiore, dove il quanto di tempo è un pochettino più grande; se quando viene schedulato in questa coda usa tutto il suo quanto di tempo vuol dire che è ancora più CPU Bound di com'è e quindi lo facciamo scalare nuovamente nelle code inferiori (se non lo utilizza tutto scalerà ai livelli superiori invece).

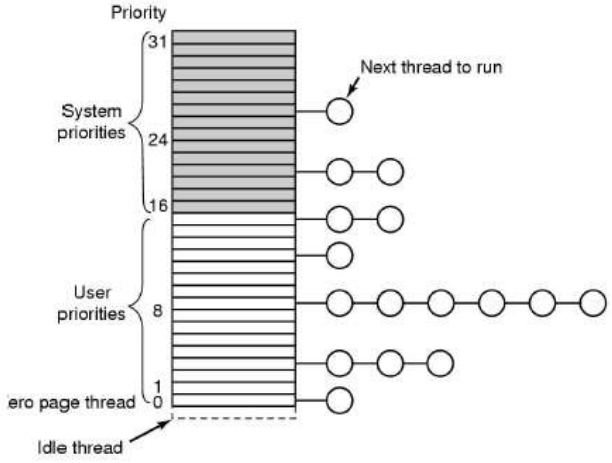
Abbiamo con questo esempio (figura a pagina successiva) 4 code, gestite con politica Round Robin, la coda di priorità maggiore ha quanti di tempo più piccoli in questo esempio di 10 ms, la coda di priorità più bassa ha quanti di tempo maggiore, in questo esempio 80 ms, che sono più adatti a gestire thread CPU Bound. Se un thread non utilizza tutto il suo quanto di tempo viene portato nella coda di livello inferiore. Quali sono le questioni con questo tipo di algoritmo? Certamente la starvation. L'avevamo risolta e adesso ritorna, l'attesa indefinita. Se al sistema continuano ad arrivare thread I/O Bound è evidente che i thread CPU Bound rischiano di andare in attesa indefinita. Perché comunque con questo algoritmo noi

MFQ



scheduliamo sempre e soltanto i thread delle code a priorità maggiore della coda non vuota a priorità maggiore. Se continuano ad arrivare thread I/O Bound o se ce ne sono molti rischiamo di non mettere mai in esecuzioni i thread CPU Bound. E poi l'altro aspetto è che ci servono delle politiche per aumentare la priorità: abbiamo visto qual è il meccanismo per abbassare la priorità; ci serve un meccanismo per alzarla, per riconoscere i thread che da CPU Bound sono ridiventati I/O Bound e devono risalire. Vi faccio direttamente l'esempio di Windows che è un esempio particolarmente pulito di algoritmo a code multiple.

Example: scheduling in Windows



Questo esempio non è nel libro però lo potete trovare nel Tanenbaum o nel Silbershatz o in un qualunque testo di sistemi operativi. Questo algoritmo utilizza 32 livelli di priorità, dei quali, però, la priorità a livello 0 è riservata ad alcuni thread particolari (ci interessa poco). Le priorità più alte, da 16 a 31, sono riservate ai thread che stanno lavorando in modo supervisore, o ai thread di sistema; quindi sono thread che generalmente stanno svolgendo dei servizi per conto di thread utente. I thread utente lavorano con priorità che va da 1 a 15. Ognuno di questi livelli di priorità è associato ad una coda Round Robin, esattamente come il Multi-LevelFeedback Queue e l'algoritmo fa questo: individua la coda di priorità maggiore non vuota e schedula con scheduling Round Robin i thread in quella coda, finché ce ne sono; se per caso un thread di priorità maggiore viene messo in una coda di priorità maggiore allora lo scheduler fa un prerilascio e passa ad eseguire i thread Round Robin sempre nella coda a priorità maggiore. Immaginiamo questa situazione: sta schedulando un thread che ha livello di priorità 26, viene riattivato un thread a priorità 31 immediatamente il thread a priorità 26 viene deschedulato (anche se non ha finito il suo quanto di tempo viene interrotto), si fa un prerilascio a favore di un thread a priorità più alta e viene schedulato quello.

Già qui notiamo un potenziale problema di starvation tra i thread che hanno priorità elevate (i thread di sistema) e i thread utente:

MFQ

- Issues with MFQ:
  - Starvation:
    - If all the upper queues are always full of I/O-bound processes
  - A process may change its "habits": from CPU-bound to I/O-bound and vice-versa
- Need for policies to raise up the priority of:
  - I/O bound processes
  - CPU-bound processes that are starving
- MFQ usually combined with dynamic priorities

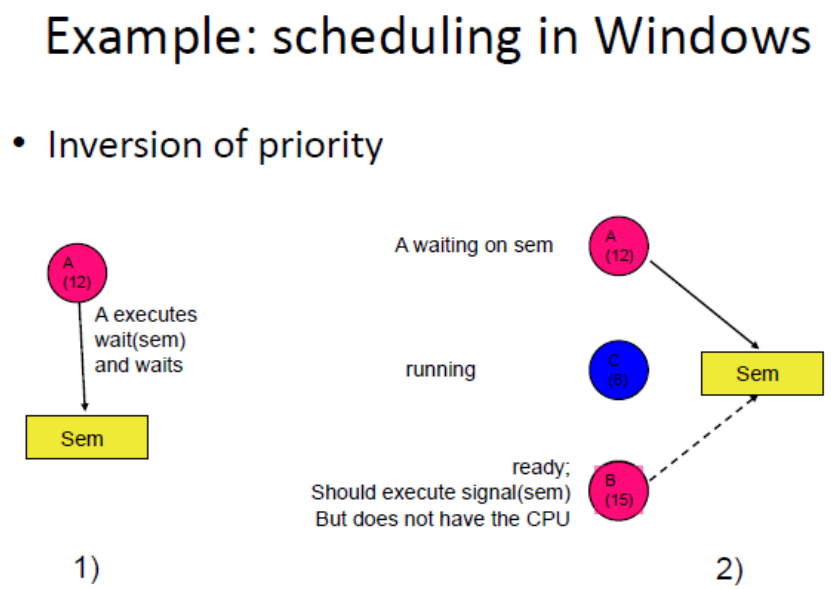
in realtà questo problema non si pone perché i thread di sistema non svolgono attività così per caso, ma vengono messe in esecuzione per svolgere attività di servizio richieste dai thread utente. Per cui se voi non mettete in esecuzione i thread utente, da un certo punto in poi non ci sarà neanche più bisogno di mettere in esecuzione thread di livello di sistema perché non ci saranno servizi da svolgere, nessuno li ha richiesti. Qui non c'è realmente starvation tra questi thread a causa di questi altri. Ecco come funziona il meccanismo:

Example: scheduling in Windows

- A new thread starts with priority 8
- Priority raised up if :
  - thread reactivated after I/O operation (disk : +1, Serial line: +6, Keyboard: +8, Audio card: +8, ...)
  - thread reactivated after waiting on a mutex/semaphore (+1 if in background, +2 if in foreground)
  - Thread didn't run for a given amount of time (priority goes to 15 for two time shares)
- Priority lowered if thread uses all time share (-1)
- When a window goes in foreground the time share of its threads is enlarged

quando un thread viene creato viene messo nella coda di priorità 8. Quando ad un certo punto tutte le code superiori sono vuote, cosa tutt'altro che improbabile poiché succede abbastanza frequentemente, si inizia lo scheduling Round Robin dei thread a priorità 8. A questo punto ci sono dei meccanismi che permettono di aumentare o ridurre le priorità dei thread. In particolare la priorità di un thread può essere aumentata in alcuni casi: se un thread viene riattivato in seguito ad una sospensione e durante questa sospensione stava aspettando informazioni da alcuni tipi di dispositivi allora in funzione del tipo di coda di attesa, in funzione del dispositivo con il quale stava colloquiando la sua priorità viene aumentata di più o di meno. Se viene riattivato dopo un'operazione su disco viene incrementata la sua priorità di 1. Se stava aspettando sulla linea seriale viene incrementata di 6, sulla tastiera viene incrementata di 8, sulla scheda audio viene incrementata di 8 e via discorrendo. Perché viene data la priorità ai thread riattivati sulla scheda audio rispetto ai thread riattivati in seguito ad un'attesa di un'operazione su disco? Risposta: per l'interazione con l'utente. La scelta di Windows,

essendo un sistema fortemente orientato a utenti desktop o laptop, privilegia fortemente i thread interattivi. C'è da dire di più in realtà: un thread che sta utilizzando una scheda audio è molto facilmente soggetto a vincoli di real time, non forti ma deboli vincoli di real time, perché se lo rallentate chiaramente la musica suona male, non vogliamo che vada a scatti. Per cui c'è anche una necessità: deve mantenere un flusso costante di controllo del dispositivo, lo deve nutrire in modo costante per suonare la musica in maniera uniforme e per questo motivo ha senso aumentarne maggiormente la priorità. Viceversa le operazioni su disco sono indici di un thread I/O Bound ma non di un thread interattivo. Il thread che sta usando il disco è probabilmente I/O Bound, ma non è interattivo perché il disco non è uno strumento per interagire con l'utente; quindi il livello di incremento di priorità è inferiore. Ci sono degli altri casi. Se un thread è stato riattivato in seguito ad un'attesa su un semaforo, su una mutex, su una variabile di condizione allora il suo incremento è di 1 se è in background oppure di 2 se è in foreground. In background vuol dire se la finestra alla quale lui è associato è in background. In foreground se la sua finestra, quindi se l'attenzione dell'utente è rivolta all'interazione verso questo thread e quindi in questo caso il suo incremento è maggiore. Anche questa differenza è appunto fatta per privilegiare i thread interattivi che stanno interagendo con l'utente in questo momento rispetto ai thread meno interattivi o comunque pur interattivi ma che in questo momento non interagiscono con l'utente. C'è un caso particolare però: (28.47) per evitare problemi di attesa indefinita se un thread non è stato in esecuzione per un tempo molto lungo, prefissato allora la sua priorità viene portata al valore massimo per i thread utente (che è 15) e questo intervento viene garantito per 2 quanti di tempo. La logica è: negli approcci Multi-Level Feedback Queue abbiamo il problema dell'attesa indefinita nel caso in cui ci siano tanti processi I/O Bound, il problema è che i processi CPU Bound si trovano ad andare in attesa indefinita in questa situazione. Quindi se un thread è a rischio di attesa indefinita perché non è stato portato in esecuzione per un periodo molto lungo, allora gli si dà priorità massima tra quelle utente; in questo modo si garantisce il fatto che lui tornerà in esecuzione in un tempo ragionevole e lo si manda in esecuzione per 2 quanti di tempo. In questo modo lui, seppur lentamente, comunque va avanti, evolve. Questo meccanismo è fatto per evitare l'attesa indefinita, ma in realtà viene fatto anche per controbilanciare un fenomeno negativo che è noto come il fenomeno di inversione di priorità.



Comunque tutti questi criteri vengono utilizzati per alzare la priorità dei thread, ma d'altra parte la priorità deve essere anche abbassata. Vediamo come si gestisce l'abbassamento di priorità, perché se non l'abbassiamo ad un certo punto tutti salgono e poi ricadiamo di nuovo nel caso del Round Robin semplice. Se un thread usa tutto il suo quanto di tempo la sua priorità viene decrementata di 1, esattamente come avveniva nello schema precedente. Non solo: questi incrementi di priorità non vengono dati per sempre ma valgono soltanto per 1-2 quanti di

tempo. Quindi se un thread viene riattivato, ha ottenuto un incremento di priorità, il quale vale per 2 quanti di tempo e poi ritorna al suo livello di priorità precedente. Infine, i thread che avevamo in foreground hanno un quanto di tempo più ampio; questo per privilegiare i thread che stanno gestendo finestre che stanno interagendo con l'utente. Per quanto riguarda il fenomeno di inversione di priorità abbiamo questa situazione. 31.46
Immaginate di avere un thread A con priorità 12 che esegue una P() su un certo semaforo e quindi si blocca in attesa (potrebbe essere anche una wait() su una variabile di condizione). Supponiamo ora che ci sia un altro thread di priorità 8 che a questo punto è quello che va in esecuzione. Il problema è: A ha una priorità maggiore di C, d'altra parte non può andare in esecuzione perché sta aspettando su un semaforo; quindi in questo momento, in realtà, la possibilità di A di andare avanti dipende da thread che farà la V() sul semaforo e in qualche maniera è come se A stesse acquisendo la priorità del thread che farà V(). Quindi se in questa situazione (???) può fare la V() e il thread B che ha priorità 4 quello che succede è che B la V() non la può fare, perché non può passare in esecuzione a causa del thread C, il quale ha priorità 8 ed è quello che viene messo in esecuzione. Ora immaginatevi non un unico thread C, ma magari una serie di thread C di priorità intermedia quello che succede è che A si trova bloccato a cause di thread di priorità più bassa di lui perché il thread che lo può sbloccare ha una priorità troppo bassa per passare in esecuzione. Questo appunto è noto come fenomeno di inversione di priorità perché è come se la priorità di A si fosse invertita, anziché essere massima fosse diventata minima. Questo è quello che vogliamo evitare, ma può essere evitato utilizzando la regola che abbiamo visto prima. Quando un thread non è stato in esecuzione per un certo periodo tempo sulla base di un parametro prefissato gli viene data priorità 15 e quindi quello che succede è che ad un certo punto a B verrà data priorità 15, tornerà in esecuzione, potrà fare la V() sul semaforo, sbloccherà A e quindi in questo modo può riacquisire la sua reale priorità. Riassumiamo tutto quello che ci siamo detti oggi.

Uniprocessor Summary

- FIFO is simple and minimizes overhead.
- If tasks are variable in size, then FIFO can have very poor average response time.
- If tasks are equal in size, FIFO is optimal in terms of average response time.
- Considering only the processor, SJF is optimal in terms of average response time.
- SJF is pessimal in terms of variance in response time.

Gli algoritmi di scheduling ne esistono diversi: abbiamo visto il FIFO, l'SJF, il Round Robin e l'algoritmo a code multiple. Il vantaggio del FIFO è che è semplice e ha un overhead molto basso, però se abbiamo thread che hanno tempo di completamento variabile il FIFO non è ottimo e presenta dei limiti. Se invece i thread hanno tempo di completamento identico allora il FIFO è ottimo nel tempo di risposta medio. Se conserviamo l'utilizzo del processore, soprattutto in contesti statici e non dinamici, l'SJF è ottimo in termini di tempo di risposta media e in questo senso è da preferire al FIFO. Esso ha garanzie di ottimizzare il tempo di risposta medio, mentre il FIFO non ce l'ha. D'altra parte l'SJF presenta li problema dell'attesa indefinita e ha un'elevata varianza in termini di tempi di risposta medi, può variare parecchio. Il FIFO e l'SJF sono prevalentemente limitati ai sistemi Batch. Se si esce dai sistemi Batch e si va ai sistemi interattivi si deve cambiare algoritmo di scheduling e in queste circostanze ha senso considerare il Round Robin.

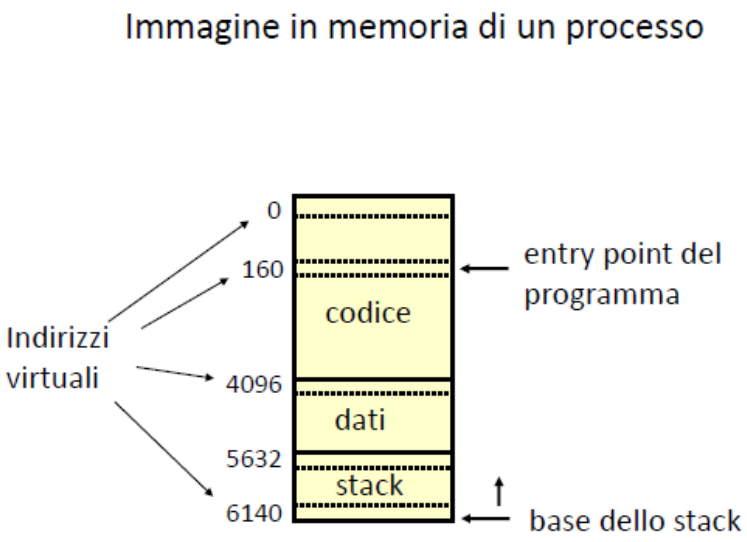
Uniprocessor Summary

- If tasks are variable in size, Round Robin approximates SJF.
- If tasks are equal in size, Round Robin will have very poor average response time.
- Tasks that intermix processor and I/O benefit from SJF and can do poorly under Round Robin.
- Max-min fairness can improve response time for I/O-bound tasks.
- Round Robin and Max-min fairness both avoid starvation.
- By manipulating the assignment of tasks to priority queues, an MFQ scheduler can achieve a balance between responsiveness, low overhead, and fairness.

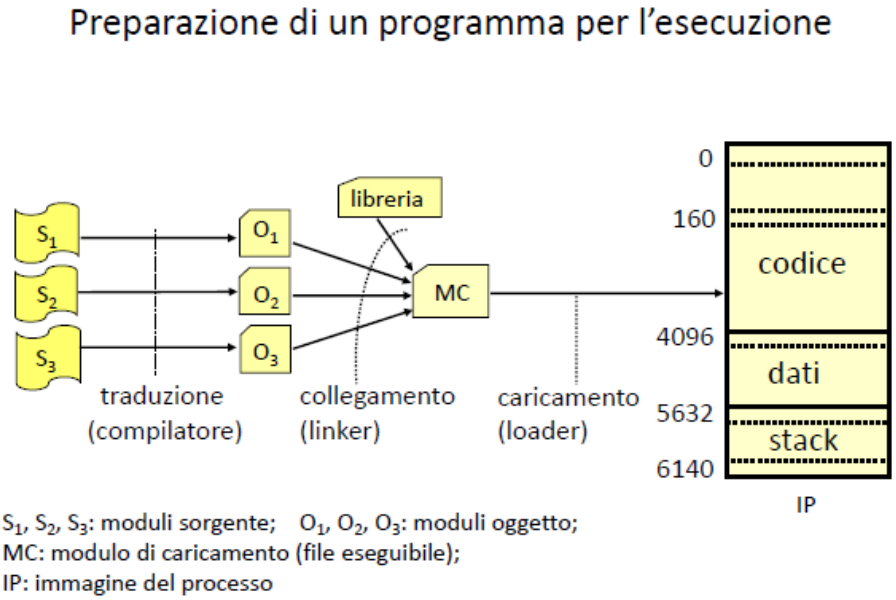
Quello che succede col Round Robin è che se i task hanno dimensione molto variabile, differente tende ad approssimare l'SJF se i task invece hanno tutti la stessa dimensione il Round Robin si comporta male in termini del tempo medio di completamento. In realtà si può pensare ad una soluzione intermedia che cerca di sfruttare l'idea dell'SJF di dare la priorità ai processi più brevi con l'idea del Round Robin di cercare di garantire una qualche forma di equità: sulla base di questa idea vengono fuori algoritmi a code multiple, dove fondamentalmente si utilizza un Round Robin con priorità. Gli algoritmi attuali dei SO, quelli più diffusi, sono varianti di algoritmi a code multiple. Con questo terminiamo con l'argomento dello scheduling.

Sistemi operativi 27/04/17

Oggi parliamo della traduzione degli indirizzi. Poi vedremo aspetti legati alla gestione della memoria virtuale, della sezione caching e infine poi l'ultimo argomento sarà quello legato ai dispositivi e al (???). Quello che vedremo in queste due lezioni saranno: concetti relativi alla traduzione degli indirizzi, quindi come facciamo a tradurre un indirizzo virtuale in uno fisico e per questo vedremo una serie di tecniche per la traduzione di indirizzi a partire da quelle iniziali, registro base-limite, segmentazione, paginazione e traduzione degli indirizzi multilivello. E poi la prossima lezione vedremo come la traduzione degli indirizzi può essere resa efficiente.
Partiamo da qui. Questo grafico vi fa vedere, tipicamente, qual è la struttura di un programma

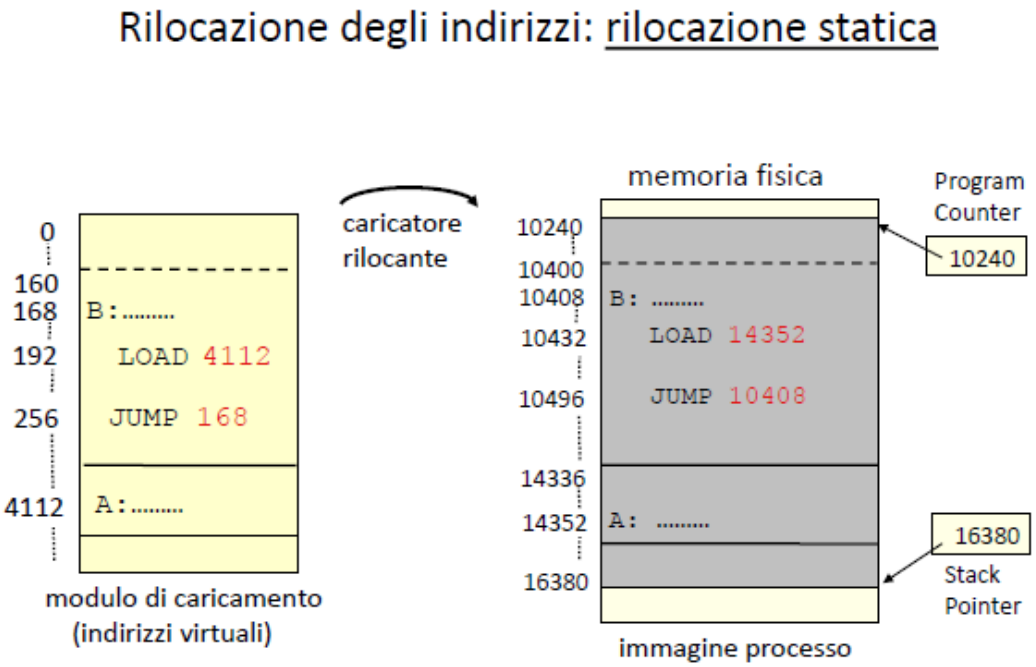


presente in memoria. Quindi quando voi attivate un processo a quel processo viene assegnata della memoria e quella memoria deve trovare spazio: il codice, i dati statici, lo heap e lo stack. Per esempio potrebbero essere disposti in questa maniera: abbiamo un codice degli indirizzi bassi, il main comincia ad un certo indirizzo, poi i dati che iniziano subito dopo il codice; subito dopo i dati preallocati (quelli statici) ci sono i dati che vengono allocati dinamicamente (lo heap); poi nella parte alta c'è lo stack che tende a crescere verso il basso e quindi va a scontrarsi prima o poi con lo heap che cresce nella direzione opposta. Quando il programma viene scritto, però, succede questo: voi scrivete il programma per esempio in C sotto forma di una serie di file indipendenti; avete un primo processo di traduzione, la compilazione che



produce i file oggetto. Ora, vediamo che cosa avviene materialmente al vostro codice durante questa traduzione. Il codice viene tradotto in linguaggio macchina, e ciò va bene. Il problema è che quando fate la traduzione in linguaggio macchina le istruzioni in linguaggio macchina non fanno riferimento a variabili, ma fanno riferimento a indirizzi e gli indirizzi fanno riferimento alla memoria. Quando però siete nella fase di compilazione, la memoria ancora non esiste, non c'è. Voi state compilando il codice in astratto, ma non avete ancora memoria nella quale il codice verrà effettivamente caricato. Per cui se volete saltare una certa istruzione dovreste utilizzare l'istruzione in codice Assembler "Jump" associata ad un certo indirizzo, ma non sapendo dove quel programma verrà allocato in memoria non sapete ancora qual è l'indirizzo che potrebbe associare. Come pure: se volete andare a leggere una variabile sapete che in linguaggio macchina non ci sono le variabili, ci sono le celle di memoria. Per cui se voi dovete andare a leggere il contenuto della variabile A voi stabilite, nel compilatore, che la variabile A sarà allocata all'indirizzo di memoria 10000 e quindi andate a fare "Load 10000, R1", caricare il contenuto dell'allocazione 10000 nel registro R1. Di nuovo il problema è che in questa fase di compilazione l'indirizzo della variabile A non è noto, perchè, di nuovo, non avete ancora la memoria sulla quale collocare il vostro programma. Quindi in questa fase di compilazione, in realtà, tutti i riferimenti alla memoria restano sospesi sotto forma di simboli, di legami simbolici. Successivamente, oltretutto avete un altro problema: compilando separatamente i singoli moduli, i singoli file, se ci sono delle invocazioni reciproche, per esempio se da questo file, da questo pezzo di codice va ad invocarti una funzione definita in S2 o in S1, compilandoli separatamente ancora non sapete qual è l'indirizzo nella quale quella funzione verrà caricata e quindi non potete completare questa compilazione. Ci sono tutta una serie di cose che nel codice oggetto restano sospese, quello che fate successivamente è fare i link, nei link unite tutti i codici oggetto e le librerie. Quando fate la fase di link voi prendete i vari moduli oggetto compilati che hanno tutti i riferimenti esterni e i riferimenti alla memoria in formato simbolico, non esplicitati, li riunite tutti assieme in un unico file eseguibile. In tale eseguibile voi decidete quali sono le allocazioni relative delle varie funzioni, per cui voi potete sapere: la certa funzione F definita nel modulo S1 si troverà in una certa posizione e di conseguenza potete risolvere i legami simbolici. Per fare tutto questo il linker fa varie cose: ipotizza di avere uno spazio di memoria che parte dall'indirizzo 0 e arriva fino ad un certo indirizzo massimo e all'interno di questo spazio di memoria va a ipotizzare di caricare, di collocare i vari componenti, i vari pezzi del codice. In questa maniera può tradurre i riferimenti del codice, può trasformare tutto quello che era un riferimento implicito simbolico in un indirizzo, per cui se dovrà saltare ad una certa istruzione, ad una certa funzione avrà un'istruzione Assembler di Jump e ad essa può associare l'indirizzo al quale deve saltare facendo riferimento ad uno spazio di indirizzamento che non è quello reale, ma è uno spazio di indirizzamento ipotetico che parte da 0 fino ad un certo indirizzo massimo. Fatto questo il linker produce il codice eseguibile che contiene dentro si sè tutto il codice, i dati statici e poi un po' di informazioni accessorie ed il codice è stato scritto utilizzando indirizzi che fanno riferimento a quello spazio di memoria. Siccome tutto il codice C, scritto da chiunque viene sempre compilato e linkato alla stessa maniera, se due di voi, sullo stesso computer, compilano e linkano il proprio programma, per entrambi questi programmi verrà prodotto un codice eseguibile che per entrambi farà riferimento ad uno spazio di indirizzamento che parte da 0 fino ad un certo indirizzo massimo. Ovviamente questi due programmi possono essere caricati ed eseguiti contemporaneamente, questo sì, ma non possono essere eseguiti tutti e due a partire dall'indirizzo 0, perchè c'è uno o c'è l'altro. Quindi, più in generale, lo spazio di indirizzamento utilizzato per le fasi di compilazione e di link non sarà mai quello reale, utilizzato poi nella vera e propria esecuzione. Una volta prodotto il codice eseguibile quello che resta da fare è mandare in esecuzione il codice eseguibile: quindi attivare un processo, caricare il programma

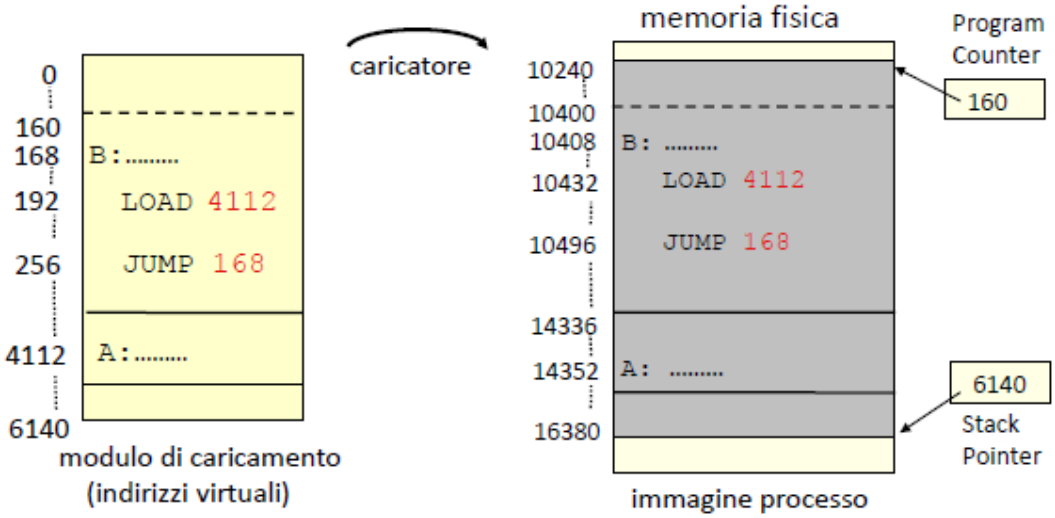
nel modulo eseguibile in memoria principale. Questa operazione viene fatta dal Loader, dal Caricatore.



In questo lucido vediamo la situazione che c'era negli anni '50, primi anni '60. Voi avete un modulo di caricamento, quindi un file eseguibile che è stato compilato facendo riferimento ad indirizzi che partono da 0 fino ad un certo indirizzo massimo, per cui tutti i riferimenti alla memoria presenti nel codice sono stati risolti dal Linker sotto queste ipotesi: quando il codice, l'istruzione all'indirizzo 256 deve fare un salto alla istruzione di etichetta B all'etichetta B avrà sostituito il valore 168 che corrisponde all'indirizzo di memoria dell'istruzione alla quale bisogna saltare. Quando il codice deve andare a caricare nella variabile A che è stata messa nell'indirizzo 4112, anzichè fare Load A verrà sostituito dal Linker il valore, il Load 4112. ipotizzando che A si trovi a questo indirizzo e che l'istruzione alla quale dobbiamo saltare sia all'indirizzo 168. Questa è la situazione del codice nel modulo eseguibile. A questo punto però quando lo mandiamo in esecuzione creiamo un processo, il SO trova uno spazio memoria abbastanza grande per contenere quel processo e lo mette in esecuzione. Questo spazio di memoria può trovarsi ovunque nella memoria fisica. Per esempio, il SO può trovare uno spazio di memoria fisica che parte dall'indirizzo 10240, che è sufficientemente grande per contenere tutto il programma più lo heap, lo stack per i dati dinamici. Se noi andiamo a prendere questo codice e lo carichiamo pari pari qua dentro, senza cambiarlo noi ci troveremmo una istruzione Load che prima era all'indirizzo 192, verrà caricata all'indirizzo 10432 (in questo esempio) e però l'indirizzo di salto è 4112. Quindi se noi prendiamo il codice così com'è pari pari e lo copiamo qui avremo un Load 4112 e qui avremo un Jump 168. Quando mandiamo in esecuzione questo codice non funziona nulla, perchè anzichè ad andare a caricare la variabile A, andremo a caricare qualcos'altro, chissà cosa c'è nell'indirizzo 4112. Quando andiamo a fare un salto, anzichè saltare correttamente all'indirizzo 10408 andiamo a saltare all'indirizzo 168 e chissà cosa c'è lì. Quindi il codice non può essere preso e copiato pari pari in memoria, ma dobbiamo, nella fase di caricamento, prendere tutti gli indirizzi riferiti dal codice e li dobbiamo tradurre, questo lo faceva il caricatore rilocante in quel periodo. Per cui il caricatore rilocante andava a leggere il codice, trovava tutte le istruzioni Afssembler, in linguaggio macchina, che corrispondevano, che contenevano riferimento alla memoria e sostituiva a quel riferimento alla memoria il riferimento sommato all'indirizzo iniziale a partire da quale il programma viene caricato. Per cui il caricatore rilocante anzichè caricare Load 4112 carica Load 14352. In questo modo il codice è messo in memoria principale con tutti gli indirizzi corretti e ora lo possiamo mandare in esecuzione e funziona correttamente. Se questa

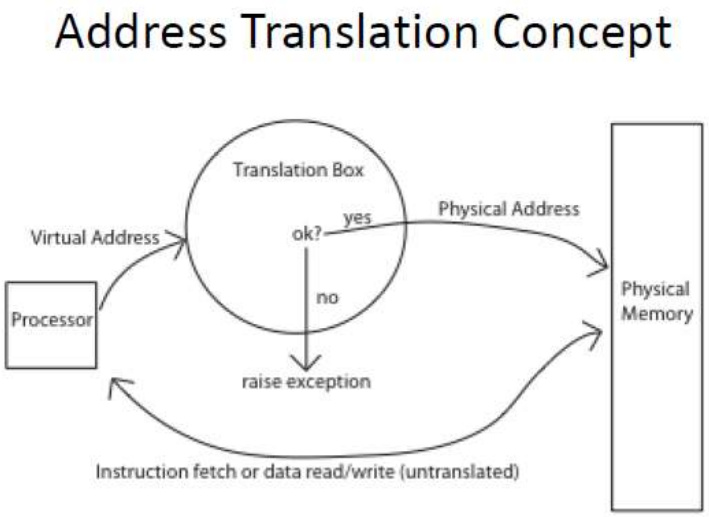
è una tecnica che si usava negli anni '50, primi anni '60 e poi è stata abbandonata, ci sono tanti buoni motivi per questo. Ci sono due motivi che sono molto importanti: il primo è legato alla protezione. Una volta che io metto in esecuzione un codice di questo genere, se il programmatore è stato abbastanza abile, ma non ce ne vuole uno estremamente abile, da modificare il codice a tempo di esecuzione per sostituire agli indirizzi predisposti al caricatore rilocante degli altri indirizzi, questo programma potrebbe andare ad accedere liberamente alla memoria (andare a danneggiare il SO, andare a prendere informazioni da altri programmi in esecuzione o ad alterarli o a danneggiarli).
Questo è certamente un grosso problema, col caricatore rilocante in questo modello non possiamo fare protezione, ce lo possiamo scordare. Il secondo problema è che (ora è difficile ma vi sembrerà più chiaro dopo) andando a caricare tanti processi in memoria, poi magari qualcuno ogni tanto termina, qualcuno riparte e via discorrendo, in realtà la memoria diventa un po' una groviera, con tante zone vuote di memoria libera e tante di memoria piena. Può capitare che ogni tanto il SO debba rimettere un po' in ordine tutto questo per ricompattare la memoria: questo meccanismo si può fare però se io devo prendere questo processo e spostarlo da un'altra parte in memoria devo andare a rimodificare nuovamente tutti gli indirizzi e ciò è un lavoro molto pesante. Questo meccanismo, questo sistema è stato abbandonato a favore di un altro approccio, che è quello con la rilocazione dinamica.

Rilocazione degli indirizzi: rilocazione dinamica



Mentre con il caricatore rilocante abbiamo una rilocazione statica perchè tutti gli indirizzi del modulo di esecuzione, del modulo di caricamento vengono trasformati staticamente in indirizzi fisici a tempo di caricamento (e una volta modificati non vengono più toccati) con la rilocazione dinamica invece tutto questo lo dimentichiamo, prendiamo il codice dal modulo dal codice di caricamento così com'è e lo carichiamo in memoria principale. Però questo ovviamente il problema che avevamo prima: questi indirizzi non hanno più senso in questo spazio di memoria. Il senso glielo dà il fatto che stavolta abbiamo a tempo di esecuzione la rilocazione. Quindi quando il programma viene messo in esecuzione, ogniqualvolta il programma, il processo esegue un'istruzione che contiene un riferimento alla memoria, questo riferimento alla memoria viene ripreso e tradotto. Quindi per ogni singola istruzione eseguita dal processore dobbiamo avere una verifica che non ci siano indirizzi di memoria e se sono presenti indirizzi di memoria allora in quel momento li traduciamo. Quindi, in questo modello il processore lavora con gli indirizzi virtuali, non con gli indirizzi fisici: ogni indirizzo virtuale che esce dal processore deve essere tradotto a tempo di esecuzione. Vediamo cosa succede in questo caso. Quando facciamo partire questo processo carichiamo nel program counter

l'indirizzo iniziale del programma, che è 160. Il processore che va a caricare dalla memoria il contenuto della locazione 160, aspettandosi di trovare un'istruzione da eseguire. Questo indirizzo 160 è un indirizzo virtuale che non può andare alla memoria così com'è, viene preso, tradotto, trasformato in 10400. A questo punto questo diventa un indirizzo fisico, va alla memoria principale ed essa restituisce il valore corrispondente e il processore lo esegue. Quindi quando abbiamo la rilocazione dinamica il processore lavora con indirizzi virtuali, quando generano un indirizzo questo viene tradotto. Chi è che fa questa traduzione? Ci serve un oggetto, una scatola di traduzione che fa da tramite tra processore e memoria fisica.



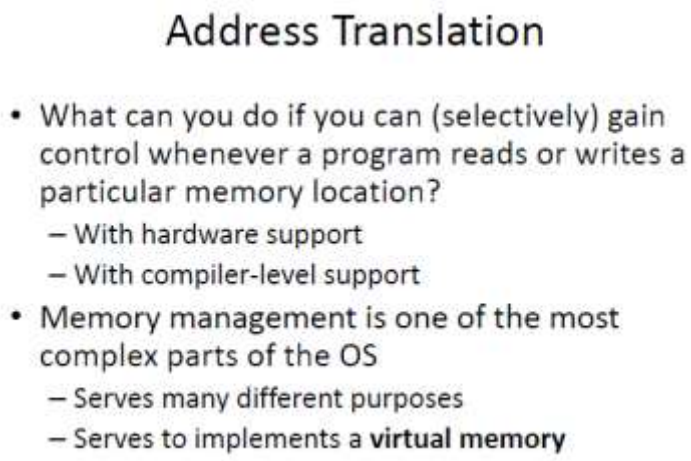
Quando il processore genera un indirizzo virtuale questo passa da questo componente, che poi lo conoscete già da Architetture (l'MMU), che fa la traduzione. Nel fare questa traduzione da indirizzo virtuale a fisico abbiamo un ulteriore elemento: possiamo fare il controllo di protezione. Possiamo verificare che l'indirizzo virtuale generato dal processore sia compreso nell'insieme degli indirizzi ammissibili, quindi sia compreso tra 0 e 6140 (che è l'indirizzo massimo possibile per quel processo). Se questo è vero allora possiamo tradurre l'indirizzo, andare in memoria fisica e prendere il dato. Oppure scrivere il dato su (???) scrittura. Se invece l'indirizzo virtuale non rientra nei limiti, l'MMU solleva un'eccezione che va sulla linea delle eccezioni verso il processore e a questo punto si scatena il meccanismo delle interruzioni, entra in gioco il SO che usa il processo di aver violato la protezione e lo termina con un Segmentation Fault. Ci serve un meccanismo di traduzione degli indirizzi dinamico.

Address Translation Goals

- Memory protection
- Memory sharing
- Flexible memory placement
- Sparse addresses
- Runtime lookup efficiency
- Compact translation tables
- Portability

Questo sistema di traduzione degli indirizzi dinamico a tempo di esecuzione ha un po' di obiettivi, il primo è garantire la protezione della memoria. Poi ci sono tutta una serie di sotto-

obiettivi: permetterci di condividere la memoria in certi casi; abbiamo visto che in certi casi la condivisione della memoria è utile, riprenderemo poi l'esempio della Fork per questo motivo. Poi dovrebbe permettermi di gestire la memoria in maniera un po' flessibile: tenete presente che quando un processo deve essere eseguito devo trovare memoria fisica libera sufficiente per poterlo eseguire. Questo può non essere un compito così banale, a seconda di come funziona il sistema di traduzione degli indirizzi, questo compito può essere più semplice o più complicato. Poi deve gestire indirizzi sparsi, deve essere efficiente nel tradurre gli indirizzi a tempo reale, non deve occupare troppa memoria, deve essere portabile. Portabile vuol dire che se io ho scritto un SO che si basa su un certo concetto di traduzione degli indirizzi voglio che questo SO con tale concetto possa essere portato su una architettura diversa senza troppi problemi.



D'altra parte questo meccanismo di traduzione degli indirizzi è evidente che interviene nell'esecuzione di ogni singola istruzione. Questo ci dà un potere enorme: immaginatevi tutto quello che si può fare se voi poteste acquisire il controllo ogniqualvolta il processore esegue un'istruzione. Del processo che è in esecuzione in questo momento voi potete sapere vita morte e miracoli. C'è una quantità di cose, di possibilità che si aprono nel momento nel quale utilizzate un sistema di traduzione degli indirizzi dinamico che vi permette di acquisire il controllo per ogni singolo indirizzo generato. Questa cosa può essere fatta interamente con il supporto hardware oppure può essere fatta con l'aiuto del compilatore: in certi linguaggi viene fatta, addirittura, con un supporto molto pesante del compilatore. Pensate ai linguaggi di tipo Java che funzionano su una macchina virtuale o C# e via discorrendo. Siccome la possibilità di avere un controllo così stretto su ciò che state eseguendo apre una montagna di possibilità e molte di queste, se non tutte, vengono sfruttate dai SO, non è un caso che alla fine il sistema di gestione della memoria diventi l'elemento più complesso del sistema operativo e anche quello più critico ai fini delle prestazioni. Il motivo è che serve a tanti diversi scopi, critico per le prestazioni, soprattutto serve per implementare una memoria virtuale. Nel momento nel quale avete un sistema di traduzione degli indirizzi di questo tipo la memoria diventa interamente virtualizzata, quindi i vostri processi possono eseguire ipotizzando o avendo a disposizione memorie che hanno caratteristiche completamente diverse da quelle fisiche effettivamente disponibile. Esempi di quello che potete fare con un controllo così fino sulla generazione degli indirizzi da parte del processore, quindi sull'esecuzione di processi.

Address Translation Uses

- Process isolation
  - Keep a process from touching anyone else's memory, or the kernel's
- Efficient interprocess communication
  - Shared regions of memory between processes
- Shared code segments
  - E.g., common libraries used by many different programs
- Program initialization
  - Start running a program before it is entirely in memory
- Dynamic memory allocation
  - Allocate and initialize stack/heap pages on demand

Potete prima di tutto isolare i processi, quindi impedire ad un processo di uscire dal proprio spazio di indirizzamento (realizzate materialmente la protezione). Lo stesso meccanismo, però, lo potete utilizzare per effettuare comunicazione tra processi efficienti, perchè potete prendere pezzi della memoria di un processo e spostarli direttamente, con una singola operazione, all'interno dello spazio di memoria di un altro processo. Quindi non avete il problema di fare delle Send che comportano la copia di informazioni, ma potete materialmente manipolare lo spazio allocato ai processi in maniera tale da spostare qualcosa da un processo ad un altro, o meglio attribuire una zona di memoria ad un altro processo. In questo modo, implicitamente, potete realizzare dei meccanismi di comunicazione molto efficienti. Potete condividere pezzi di codice, per esempio librerie; potete sfruttarla per l'inizializzazione dei programmi, per esempio potete mettere un programma in esecuzione, quindi attivare un processo, far partire un processo anche se la sua memoria non è stata completamente inizializzata. Potete mettere in esecuzione un processo anche se non avete caricato in memoria neanche una riga di codice di quel processo. Il processo avrà un PC, il processore andrà ad eseguire la prima istruzione puntata dal PC: siccome quell'istruzione genererà un indirizzo che ufficialmente appartiene al processo, ma che ancora non è stato inizializzato o addirittura neanche è stato allocato, il sistema di traduzione degli indirizzi se ne rende conto ed attiva una procedura che a quel punto inizia il caricamento selettivo dei dati del processo. Potete fare allocazione dinamica della memoria, quindi far crescere la dimensione allocata al processo, lo spazio allocato al processo. Se voi scrivete un codice C con una procedura ricorsiva questa procedura ricorsiva può far crescere a dismisura lo stack, il problema è che magari questo lo fate in 3 righe di codice. Come si fa realmente ad allocare uno stack per un programma del genere? È molto difficile, perchè non sapete poi materialmente quanto cresceva lo stack. Avendo meccanismi di traduzione dinamica degli indirizzi potete far crescere lo stack o lo heap a dismisura per tener conto delle effettive esigenze del processo che nascono durante la sua esecuzione, non prima.

Ma non è finita qui: potete gestire meglio la cache associata al processo, quindi sia la cache di

Address Translation (more)

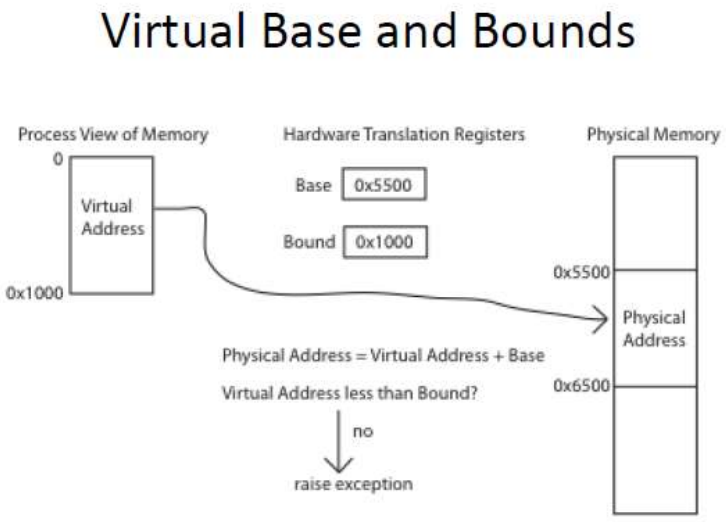
- Cache management
  - Page coloring
- Program debugging
  - Data breakpoints when address is accessed
- Zero-copy I/O
  - Directly from I/O device into/out of user memory
- Memory mapped files
  - Access file data using load/store instructions
- Demand-paged virtual memory
  - Illusion of near-infinite memory, backed by disk or memory on other machines

processore che le cache di vario livello. Probabilmente le avete utilizzate per fare il debugging ai corsi di laboratorio, per mettere dei breakpoint in certi punti del codice, potete utilizzarlo per fare la Zero-Copy I/O, quindi trasferire informazioni dai dispositivi ai processi senza dover copiare troppe volte le informazioni, con un unico traferimento; per gestire (???) memoria, per gestire caricamento di pagine a domanda sulla memoria virtuale (molte di queste cose le vedremo poi via via che andiamo avanti), per gestire il checkpointing dei processi, quindi

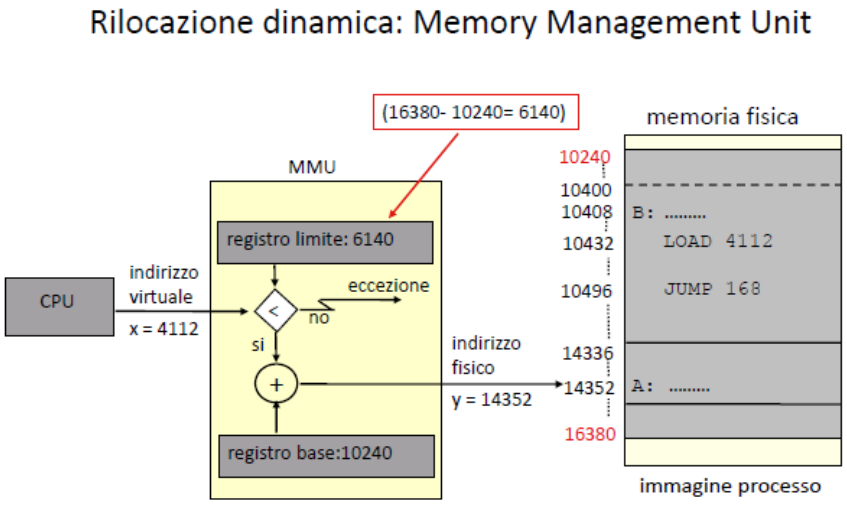
Address Translation (even more)

- Checkpointing/restart
  - Transparently save a copy of a process, without stopping the program while the save happens
- Persistent data structures
  - Implement data structures that can survive system reboots
- Process migration
  - Transparently move processes between machines
- Information flow control
  - Track what data is being shared externally
- Distributed shared memory
  - Illusion of memory that is shared between machines

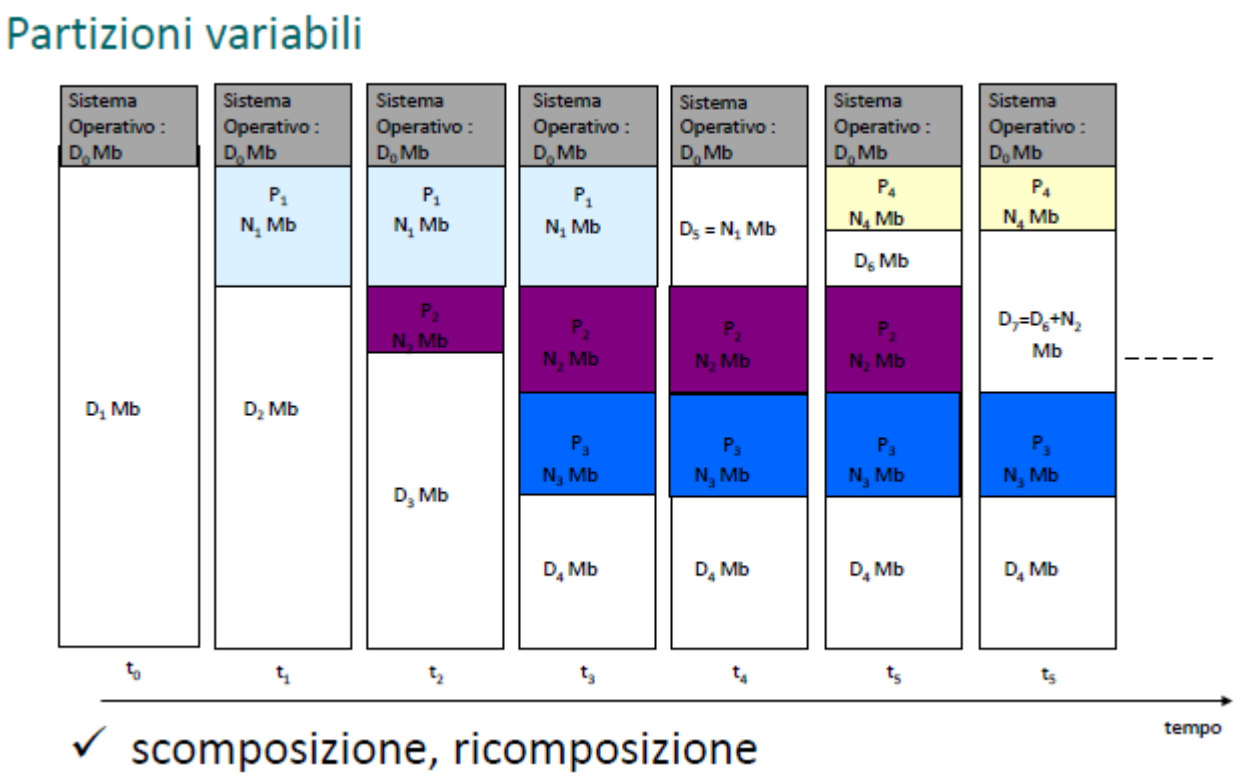
interromperli e salvare lo stato; per creare strutture dati persistenti, per migrare i processi tra macchine differenti, per controllare il flusso controllo, per realizzare dei sistemi che gestiscono memoria distribuita su più macchine. Quindi vi rendete conto che avere un meccanismo di traduzione dinamica degli indirizzi permette di fare una tale quantità di cose che viene caricato di una tale quantità di responsabilità che non a caso diventa un sistema realmente complesso ed importante. Detto questo vediamo quali sono questi sistemi di traduzione degli indirizzi.
Il primo sistema che vi faccio vedere in realtà ve l'ho già raccontato, ve l'ho già anticipato nella lezione introduttiva ed è quello basato sulla coppia di registri base-limite (è detto anche Base & Bound).



L'idea è: il processo vede la memoria come un'area di memoria contigua che parte da 0 fino ad un certo indirizzo massimo. Ovviamente questo non è la sua memoria fisica, ma è la sua memoria virtuale, perchè la sua memoria fisica in realtà se ne andrà da qualche altra parte, partirà da un indirizzo (in questo caso 5500) e arriverà all'indirizzo 6500, quindi sarà diversa la sua memoria virtuale. Associato al processo abbiamo 2 registri che sono il registro base ed il registro limite. Quest'ultimo specifica quanto è la dimensione dello spazio virtuale assegnato al processo e di conseguenza quanta è la dimensione dello spazio fisico assegnato al processo. Il registro base, invece, ci dice a partire da quale indirizzo lo spazio di memoria virtuale del processo è stato caricato. Quindi in questo esempio l'indirizzo base è 5500 che è l'indirizzo a partire dal quale il processo è effettivamente caricato in memoria fisica. Occupa 1000 byte a partire dall'indirizzo 5500 fino al 6500 (l'ultimo estremo escluso). Questi 2 registri dove stanno materialmente? Essi devono essere conservati nei descrittori dei processi. Ogni processo avrà la sua copia di registri. Quando il processo, però, viene messo in esecuzione durante la commutazione di contesto i valori base-limite di quel processo vengono presi dal descrittore e vengono copiati in 2 registri del processore o se volete dell'MMU, per cui base-limite in realtà a tempo di esecuzione stanno nell'MMU; quando il processore che lavora con indirizzi virtuali genera un indirizzo virtuale che fa riferimento a questo spazio passa attraverso l'MMU e viene prodotto l'indirizzo fisico, dato da indirizzo virtuale generato dal processore più la base. Questa è la traduzione. Contestualmente viene anche il controllo di protezione. Si verifica che l'indirizzo virtuale sia minore del limite: se è così tutto bene, l'indirizzo viene inviato alla memoria, se invece è maggiore viene sollevata un'eccezione. Qui abbiamo un esempio.



La Cpu genera l'indirizzo virtuale 4112, viene confrontato col registro limite, se va tutto bene 4112 viene sommato al registro base, questo fa uscire dal sottosistema processore l'indirizzo fisico che (???). Questo modello di traduzione degli indirizzi ha dato luogo alle tecniche di gestione della memoria che spesso sono note anche come partizioni variabili. Che cosa comporta in effetti l'utilizzo dei registri base-limite? Comporta che lo spazio di memoria fisica allocata ad un processo debba sempre essere una porzione di memoria fisica contigua: parte da un certo indirizzo fisico ed arriva ad un altro indirizzo fisico e tutti gli indirizzi intermedi sono tutti allocati al processo. Vediamo cosa succede quando avete questo modello.



Immaginiamo un sistema che ha una certa memoria principale in cui una parte è riservata al SO e il resto di 1MB sono riservati ai processi in stato utente. Quando viene messo in esecuzione il processo P1 sappiamo che P1 ha bisogno di un certo numero di MB, questi vengono allocati in uno spazio libero. Quindi la memoria libera del sistema viene divisa in 2 parti, in 2 frammenti: un frammento libero, di dimensione pari a D2 e un frammento occupato, assegnato a P1 di dimensione N1. Se successivamente viene creato il processo P2 di dimensione N2 questo troverà spazio in un'altra zona di memoria (ovviamente non sovrapposta a quella di P1) per cui ridurremmo ulteriormente una memoria libera e una parte va al (???); e così via viene allocato P3. Ora supponiamo in questa situazione che ad un certo punto P1 termini: quando esso termina lo spazio di memoria fisica allocato a P1 si libera, diventa parte della memoria libera. Il problema è che però stavolta abbiamo una situazione in cui le zone di memoria libere sono 2, non è più una. Una porzione di dimensione pari a N1 (con (???) a P1) è una dimensione pari a D4 (???) rimasto libero prima. Se successivamente viene eseguito un altro processo P4 di dimensione N4 il SO deve cercare una porzione di memoria fisica, contigua, libera di dimensione almeno N4. In questo caso scopre che la zona D5 è più grande del necessario, quindi prende una porzione di questa, pari a N4 alloca P4 e il residuo lo lascia libero. Quindi abbiamo una porzione di memoria libera pari a D6 e una porzione di memoria libera pari a D4. Se poi P2 termina libera il suo spazio di memoria e allora in questo caso gli spazi di memoria libera si riuniscono e si crea un nuovo spazio di memoria libero un po' più grande. Si chiama "partizioni variabili" perchè la memoria viene partizionata in un numero di partizioni pari al numero di processi in esecuzione o attivi nel

sistema, in esecuzione (???). Dopo per ogni partizione può avere una dimensione variabile, non è fissa, dipende dalla quantità di memoria richiesta da quel particolare processo. In questo contesto ci serve avere un sistema per allocare la memoria ai processi. Quando un processo parte il caricatore va a leggere il file eseguibile, scopre quanta memoria serve al processo e di conseguenza informa il resto del SO. Quindi il SO ha bisogno di un algoritmo per trovare memoria libera sufficiente. Ci sono un po' di algoritmi, ma non è che siano particolarmente interessanti.

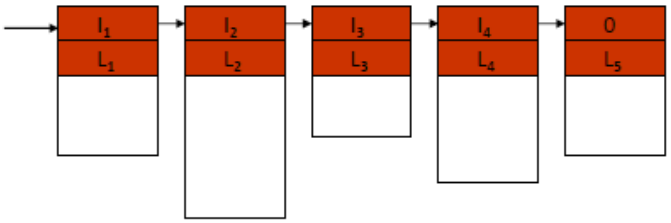
Criteri di allocazione delle partizioni libere

- First-fit

  fra tutte le partizioni libere di dimensioni sufficienti a soddisfare la richiesta viene scelta quella di indirizzo minimo.

- Best-fit

  fra tutte le partizioni libere di dimensioni sufficienti a soddisfare la richiesta viene scelta quella di lunghezza minima



Lista di partizioni libere

Si chiamano First Fit o Best Fit per esempio. L'idea del First Fit è questa: se io devo cercare in questo esempio uno spazio di memoria di dimensione pari a X MByte. Se X è maggiore di D6 scorro tutti i frammenti di memoria libera, appena trovo una zona di memoria libera più grande di X li alloco. Invece il Best Fit va a cercare tra tutte le zone di memoria libera quella più piccola di dimensione sufficiente per accogliere quel processo. In passato sono stati fatti tanti studi per vedere quale era meglio o peggio, ma in realtà non ce n'è uno migliore o peggiore, dipende molto dai casi. In ogni caso quello che succede con questa tecnica delle partizioni variabili è il fenomeno della frammentazione.

Partizioni e frammentazione

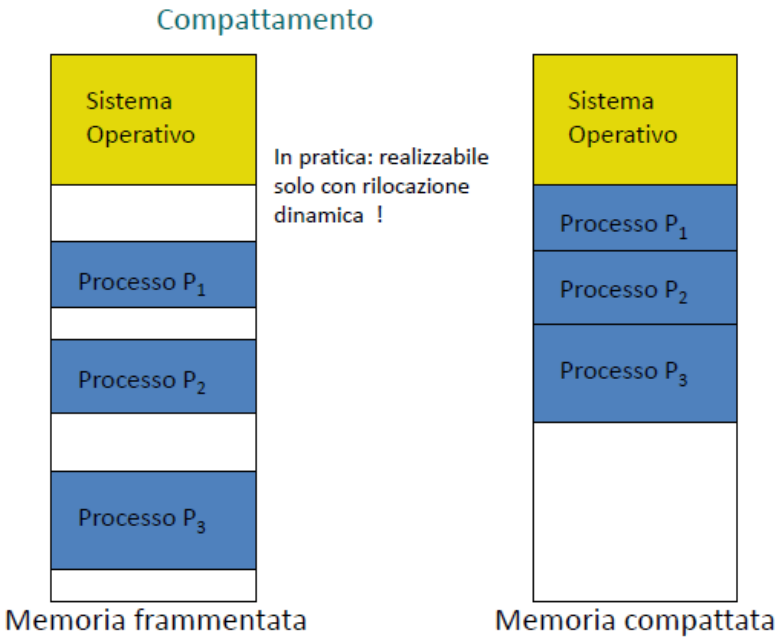
- Frammentazione interna

  frazione della memoria non utilizzata nella tecnica delle partizioni fisse: corrisponde alla differenza tra la somma delle dimensioni delle partizioni allocate e la somma delle dimensioni delle partizioni richieste.

- Frammentazione esterna

  frazione della memoria non utilizzata nella tecnica delle partizioni variabili: corrisponde alla somma delle dimensioni delle partizioni libere quando ciascuna di esse non è da sola sufficiente a soddisfare una richiesta di memoria.

Abbiamo già visto prima cosa significa, ma lo vediamo qui. Per effetto di una serie di allocazioni e disallocazioni dei processi quello che succede è che la memoria libera che inizialmente era tutta contigua dopo un po' di tempo diventa frammentata. Quindi ci troviamo ad avere tante zone di memoria libera separate da loro, quindi che non possono essere congiunte. Questo fenomeno della frammentazione che è detto frammentazione esterna può diventare un fenomeno patologico. Immaginate di avere tanti frammenti piccoli di memoria fra i vari processi: complessivamente potrei anche avere tanta memoria libera, il problema è che sono tanti frammenti non contigui. Quindi se al SO viene chiesto di caricare un processo grande, tale processo potrebbe anche starci se tutti questi frammenti fossero uniti, ma siccome sono separati questo processo non può essere caricato e quindi dobbiamo aspettare a mandarlo in esecuzione. Quindi il problema della frammentazione esterna si presenta in queste situazioni: quando la memoria libera viene frammentata troppo e quindi diventa inutilizzabile per eseguire i processi. Magari col tempo lo ridiventa se viene di nuovo ricompattata, però rallenta i processi. In effetti ci sono due tipi di frammentazione: esterna (che riguarda gli spazi di memoria libera esterni ai processi); altre tecniche di allocazione della memoria invece hanno il problema della frammentazione interna, per cui voi potreste allocare al processo più spazio di quello che gli serve, per vari motivi, e chiaramente lo spazio che quel processo ha allocato ma non usa è memoria sprecata. Per ovviare al problema della frammentazione esterna con la tecnica delle partizioni multiple si fa il compattamento.



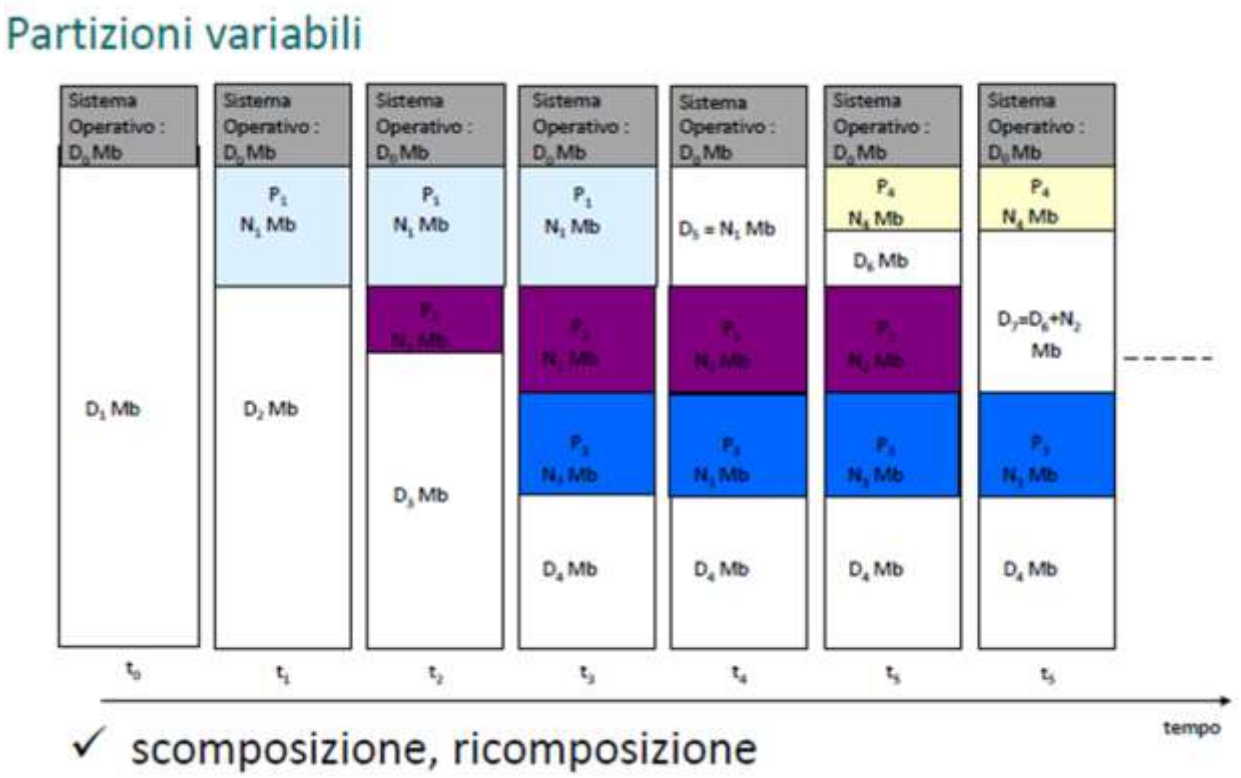
Quello che fate voi quando dovete mettere in ordine una libreria: mancano un po' di libri nel mezzo mettete una mano all'estremo, li schiacciate tutti e fate il compattamento. Il problema è che però nel SO non basta semplicemente con una mano spingere tutti i processi verso l'alto, vanno proprio ricopiati, byte per byte. Quindi questa operazione di compattamento se in un sistema che usa le partizioni variabili la frammentazione esterna diventa patologica e problematica, questa operazione di compattamento può essere fatta però è molto costosa. È bene non abusarne.

Virtual Base and Bounds

- Pros?
  - Simple
  - Fast (2 registers, adder, comparator)
  - Can relocate in physical memory without changing process
- Cons?
  - Can't keep program from accidentally overwriting its own code
  - Can't share code/data with other processes
  - Can't grow stack/heap as needed

Con la tecnica dei registri base-limite abbiamo certamente un po' di vantaggi: quello più grosso è che è un sistema estremamente semplice, perchè abbiamo, in fin dei conti, da aggiungere soltanto 2 registri al processore e 2 campi all'interno del descrittore del processo. Di conseguenza è anche molto veloce, poichè quello che dobbiamo fare è soltanto una somma, un confronto, che sono operazioni che si fanno molto velocemente, per di più in aritmetica (???). <Il Chessa scivola e bestemmia internamente> Ci permette di caricare i processi in memoria fisica senza dover andare a modificare il codice, quindi a differenza della rilocazione statica dobbiamo soltanto ricopiare il processo ma non abbiamo bisogno di rimodificare tutti gli indirizzi. Problemi o limiti di questo sistema (39.55) Questo sistema è fatto per proteggere i processi nella loro interezza però, per esempio, non impedisce ad un processo di modificare il proprio codice. Ora uno potrebbe anche fregarsene oggettivamente, non è che ci interessi tanto che un processo modifichi il proprio codice. Se modifica il proprio codice nella peggiore delle ipotesi danneggia sé stesso. È anche vero, però, che mandare in esecuzione un programma che accidentalmente ha modificato il proprio codice è un'inutile perdita di tempo, che va a detrimento delle prestazioni del sistema e va a rallentare gli altri processi. In fin dei conti un processo che ha modificato il proprio codice non produrrà niente di buono. Voi avete mai scritto programmi che modificano il proprio codice? Sì e no. No, non ne potete aver scritti perchè se avete scritto programmi Unix, esso protegge il codice difende il Base & Bound, quindi non lo potete fare. Sì, se non ci fosse stato quel meccanismo di protezione di Unix, sono pronto a mettere la mano sul fuoco, che tutti quanti, se avete scritto il codice avete scritto il codice che era a rischio di modificare sé stesso. Tutte le volte che avete utilizzato i puntatori male che siete andati a scrivere in una zona di memoria arbitraria, se per caso tale zona coincideva con una porzione di codice, in assenza di meccanismi di protezioni voi sareste andati a modificare il vostro codice. Quindi se voi prendete il vostro codice bacate e lo eseguite sul sistema che fa Base & Bound, modificate il vostro codice, il vostro programma continua a funzionare, potreste non ricevere dal SO il maledettissimo Segmentation Fault, ma in compenso tirate fuori risultati impossibili da interpretare. In realtà avere un SO che impedisca ai programmi di modificare il proprio codice è di aiuto perchè

permette di rilevare subito i problemi, interrompere subito un programma e iniziate prima il debug. L'altro problema, l'altro limite è che non si può condividere dati con altri processi: siccome se voi volete condividere un dato e questo dato si trova a metà dello spazio di memoria allocato ad un processo, come fate a condividerlo? Tenete presente che lo spazio allocato ai processi deve essere contiguo col Base & Bound. Quindi potete anche sovrapporre 2 spazi di memoria di 2 processi, ma dovete essere molto molto fortunati perchè nello spazio che andate a sovrapporre ci siano dei dati che volete davvero condividere. La realtà è che non si può condividere memoria. L'altro problema è che quando fate partire un processo dovete stabilire a priori quanto quel processo dovrà utilizzare di memoria, perchè voi dovete inizializzare i registri base-limite gli allocate quella memoria, da lì in poi parte la memoria libera ed essa la potete allocare ad un altro processo. Guardiamo questo esempio.



P2 è stato messo in esecuzione, qui c'è la memoria libera ma non gli serviva in quel momento. Dopodiché per i suoi motivi si è allargato, è diventato così grosso finchè non è stato allocato P3. In questa situazione se P2 ha bisogno di allocare altra memoria perchè deve far crescere lo stack o lo heap non posso farlo, perchè non posso dargli altra memoria contigua che si va a sovrapporre a P3; l'unica soluzione è prendere P2 e metterlo da un'altra parte, ma è una soluzione molto costosa. Domanda. Quando c'è una commutazione di contesto, in un sistema di questo genere cosa dovete salvare nel descrittore? <Uno studente> "Il Base & Bound?".

Virtual Base and Bounds

- Pros?
  - Simple
  - Fast (2 registers, adder, comparator)
  - Can relocate in physical memory without changing process
- Cons?
  - Can't keep program from accidentally overwriting its own code
  - Can't share code/data with other processes
  - Can't grow stack/heap as needed

<Chessa> Sì dovete salvare anche quelli. Il Base and Bound sono associati al processo, quindi quando caricate un nuovo processo, oltre a quello che abbiamo già visto nelle lezioni scorse, andate anche a prendere i valori registro base-limite e li caricate nei registri del processore. Se poi durante l'esecuzione quel processo ha allocato altra memoria o se Base è stato spostato, Base & Bound possono essere modificati e quindi dovete risalvarli. La cosa curiosa di questo sistema è che quando è stato proposto, credo agli inizi degli anni '60, in realtà richiedeva alcune modifiche nell'hardware, perchè prima che esistessero, ovviamente i processori non avevano i registri base-limite al loro interno, quindi non si poteva realizzare. Per poterlo realizzare bisognava modificare il processore, bisognava andare da chi produceva processori e dirgli: "Guarda, il processore lo fai diversamente." Questo tipo di modifica, all'epoca, era ritenuta troppo costosa (quindi c'è chi si è opposto): si riteneva che tutto il lavoro per modificare il processore e aggiungere 2 registri fosse un costo eccessivo rispetto ai vantaggi che poteva portare. La storia invece è andata nella direzione opposta: non soltanto non era un costo eccessivo, ma addirittura ora è diventato ridicolo e con i processori attuali il problema del costo per questi sistemi di gestione della memoria proprio non si pone, perchè il costo va a pesare su altri aspetti relativi alla prestazioni, alla proteione, alla sicurezza di sistema..C'è un forte vantaggio, una forte spinta per rendere più complessi e più efficaci questi

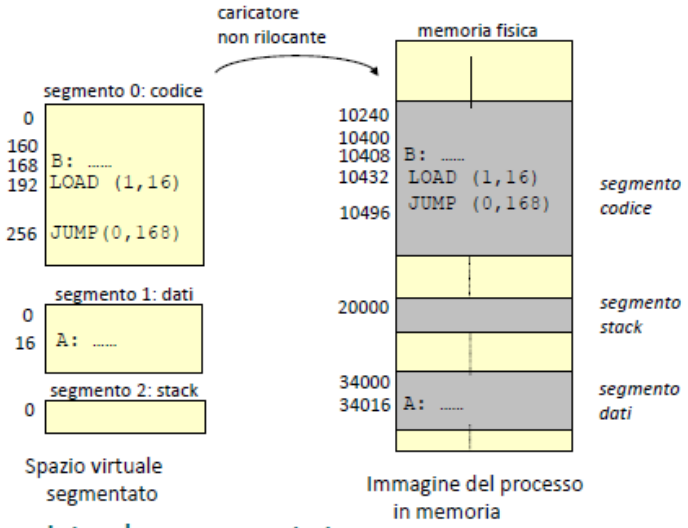
meccanismi. Per diversi motivi, quindi, il (???) Base & Bound è stato poi abbandonato a favore di altre tecniche: la prima di queste è la tecnica della segmentazione.

Segmentation

- Segment is a contiguous region of memory
  - Virtual or (for now) physical memory
- Each process has a segment table (in hardware)
  - Entry in table = segment
- Segment can be located anywhere in physical memory
  - Start
  - Length
  - Access permission
- Processes can share segments
  - Same start, length, same/different access permissions

Intanto definiamo cos'è un segmento. Un segmento è una zona contigua di memoria, quindi esattamente come capitava prima con le partizioni variabili, un segmento è una partizione di memoria fisica contigua. Lo spazio di memoria di un processo è dato da un insieme di segmenti: non è più se volete come nel caso precedente, lo spazio di memoria al processo era un unico segmento. Ora, con la segmentazione, lo spazio di memoria di un processo è dato dall'unione di un certo numero di segmenti. Questi segmenti non hanno l'obbligo di stare uniti l'uno all'altro, di essere contigui tra loro, possono essere sparpagliati e di conseguenza è necessario, da parte del gestore della memoria, sapere per ogni processo quali e dove si trovano i segmenti allocati ad ogni processo. Per questo motivo ogni processo dispone di una tabella, detta tabella dei segmenti, che contiene per quel processo l'indirizzo iniziale, la lunghezza di ogni segmento allocato. La descrizione del singolo segmento equivale ai registri base-limite, ogni segmento ha associato informazioni sul suo indirizzo iniziale e lunghezza. Oltre a questo ogni segmento ha anche dei permessi specifici, quindi in questo modo posso marcare alcuni segmenti come sola lettura o altri come a sola esecuzione e via discorrendo. Questo mi dà la possibilità, con la segmentazione, di stabilire, di proteggere il codice, di impedire al programmatore di andarlo a modificare, per esempio. Siccome le informazioni stavolta sono distribuite sui vari segmenti il singolo segmento lo posso condividere: questo mi permette di avere condivisione di memoria tra processi differenti in maniera agevole. Vediamo come funziona.

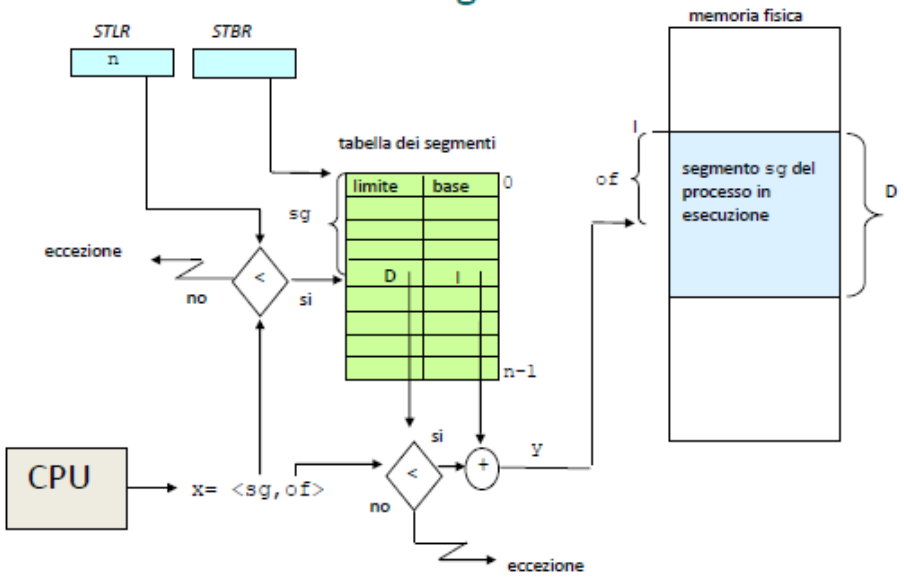
Segmentazione



Spazio virtuale segmentato

Questo è uno spazio di memoria segmentato. In questo esempio lo spazio di memoria del processo è lo stesso dell'esempio iniziale e diviso in 3 segmenti: un segmento contiene il codice, un segmento contiene i dati e lo heap e l'ultimo contiene lo stack. Questi 3 segmenti hanno il loro indirizzo iniziale, che è sempre 0, il loro indirizzo massimo nello spazio virtuale, dopodichè sono allocati in memoria fisica in punti arbitrari, dove capita. Per cui in questo caso il segmento dati è allocato qua, lo stack qui e il codice lì. Nuovamente il caricatore non è rilocante, per cui quando si carica il processo per metterlo in esecuzione si allocano i segmenti, si copia il codice eseguibile all'interno del/dei segmento/i codice e a quel punto si manda in esecuzione il processo. La traduzione degli indirizzi avviene con questo meccanismo.

Segmentazione: traduzione degli indirizzi

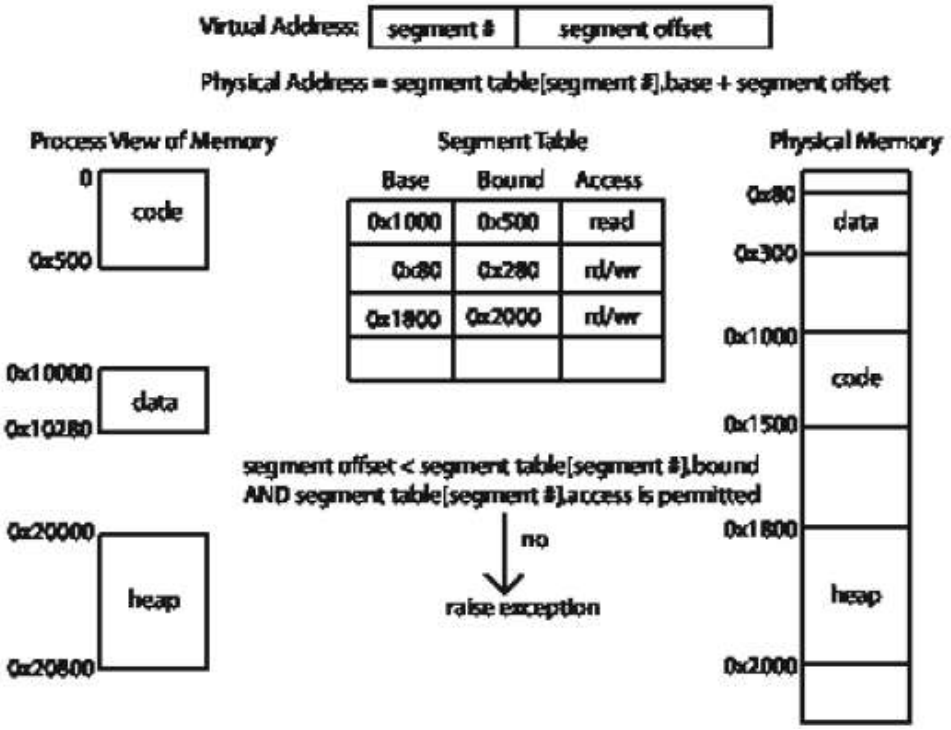


==> Tabella dei segmenti residente in memoria

- Per evitare l'accesso alla tabella dei segmenti ad ogni riferimento alla memoria: cache della MMU
  --> accesso associativo; algoritmo di sostituzione

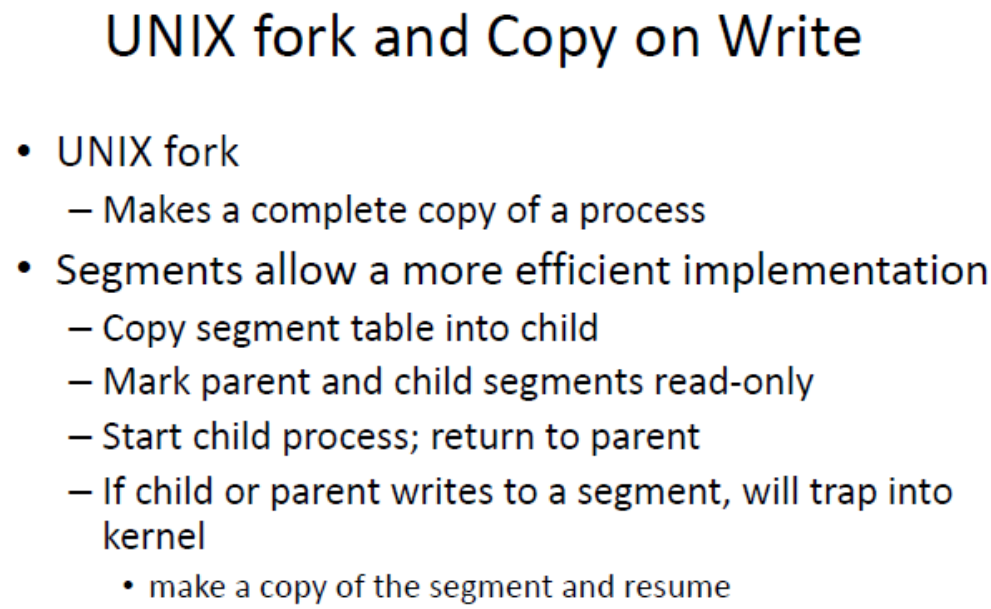
Intanto noi dobbiamo sapere dove sta la tabella dei segmenti e quanto è lunga: questa informazione generalmente è contenuta nel descrittore del processo. (i registri azzurri). Il processore genererà un indirizzo (è sempre una stringa di bit di dimensione fissa), però viene interpretato in realtà come una coppia di valori Indice di segmento e Offset. L'Indice di

segmento viene estratto dall'indirizzo e si confronta con il contenuto del registro limite: se il numero del segmento eccede il numero di segmenti allocati al processo allora c'è un'eccezione. Il processo sta cercando di accedere ad un segmento che non ha allocato. Se invece l'Indice di segmento rientra nel limite allora va bene e può essere utilizzato per indicizzare la tabella dei segmenti. Quindi tramite l'indice del segmento noi andiamo nella tabella dei segmenti, estraiamo una riga e in questa riga troviamo la coppia di valori base-limite per quello specifico segmento. A questo punto si fa un ulteriore controllo di protezione che è l'Offset, cioè che la posizione del byte che vogliamo leggere all'interno di quel segmento sia minore del limite del segmento. Se è sì tutto bene, altrimenti violazione di protezione. Se l'Offset è minore del limite produciamo un indirizzo fisico andando a sommare alla base l'Offset. Questo ci dà un indirizzo di memoria fisica. <Uno studente> "Il limite della tabella è il numero di segmenti che ho". <Chessa> Esattamente. Non possiamo confrontare il numero crudo perchè la tabella chiaramente ha una sua dimensione, occupazione. Notate il ruolo dell'Offset rispetto all'indice del segmento: l'Offset ci dice per quel segmento a quale Byte voglio andare ad accedere all'interno del segmento ipotizzando che il segmento parta all'indirizzo 0. Quindi alla posizione relativa del dato che voglio accedere di quel segmento e questo deve essere sommato alla base.



In realtà la tabella dei segmenti è un po' più ricca di tutto questo perchè contiene il campo di accesso, con i diritti di protezione, quindi per ogni riga della tabella dei segmenti, oltre a base-limite, abbiamo anche i diritti di accesso. Questo vuol dire che una volta che abbiamo estratto la riga dalla tabella dei segmenti non controlliamo soltanto che l'Offset sia minore del limite, ma controlliamo anche che l'operazione sia permessa. Ora, il fatto di avere un meccanismo di traduzione degli indirizzi a segmentazione permette la condivisione delle informazioni tra processi: in teoria questo non dovrebbe essere possibile perchè i processi possono accedere al loro spazio di memoria chiuso, non possono andare in altri spazi, però se lo comunicano appropriatamente al SO, quest'ultimo può permettere a 2 o più processi di utilizzare lo stesso segmento. Perchè vogliamo la condivisione dei segmenti in questa maniera? Per esempio nel codice potrei fare riferimento ad una libreria di sistema: tale libreria, presumibilmente, sarà usata su tanti processi, quindi allocare la libreria su un segmento separato è vantaggioso perchè la carico una sola volta in memoria e tutti i processi che la utilizzano condividono lo stesso segmento, per esempio. Oppure potrei utilizzare il segmento per memorizzare una struttura dati, un segmento per una struttura dati che voglio far condividere tra due processi e

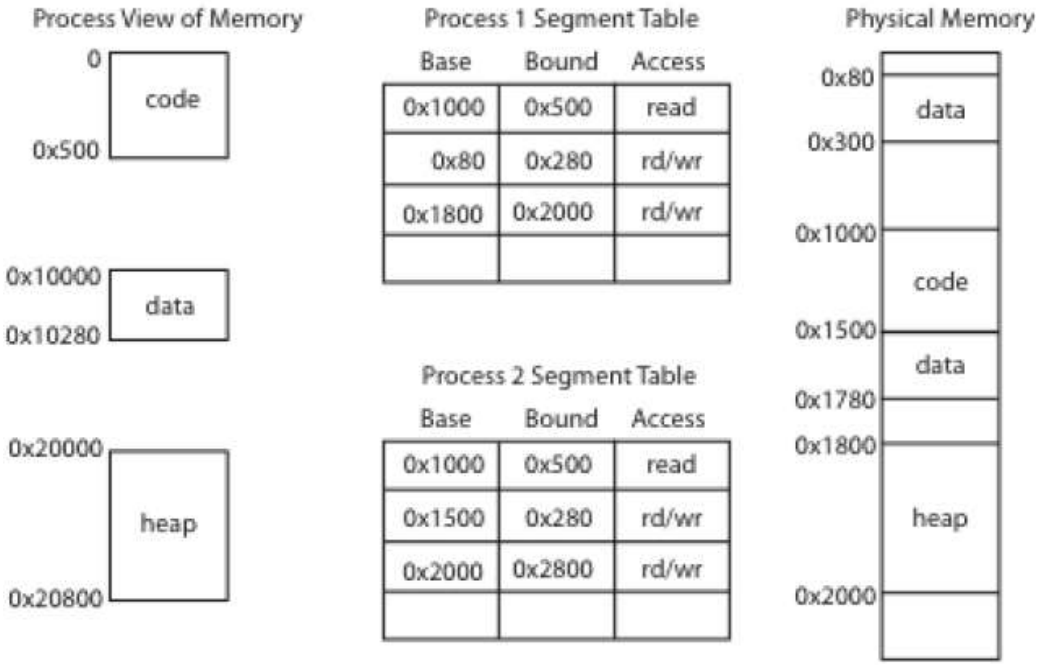
in questo modo tali processi possono comunicare molto facilmente e velocemente usando una memoria condivisa (ovviamente con tutti i meccanismi che abbiamo discusso nella prima parte del corso). Uno dei meccanismi che si avvantaggia della segmentazione è il meccanismo di Unix delle Fork. Un processo invoca la Fork e viene creato, generato un processo figlio identico al padre: per generare tale figlio, il SO deve duplicare tutte le strutture del nucleo associate al padre (il descrittore di processo e tutte le tabelle usate dal figlio), ma non solo; deve duplicare anche la memoria. Quindi prende lo spazio di memoria associato al padre e ne fa una copia affinchè il figlio lo possa utilizzare, con codice, dati e stack. Tutto questo è un enorme spreco di tempo, perchè in effetti, prima che il figlio inizi a lavorare, questa memoria è esattamente la stessa e tra l'altro il figlio potrebbe anche non modificarne nè codice nè dati nè stack. Potrei fare una Exec e poi dopo devo di nuovo ricambiare tutto quanto. Se invece il figlio non fa una Exec comunque il codice resta uguale, non cambia sicuramente. Per questo motivo con la segmentazione in effetti si può ottimizzare pesantemente la Fork in Unix utilizzando la tecnica della Copy on Write.



Diciamo che a Copy on Write è un'implementazione più efficiente della Fork. Quando si fa la Fork si vanno prima di tutto a duplicare le strutture dati, quindi si duplica il descrittore del processo, si alloca una tabella dei segmenti per il figlio e tale tabella la si inizializza esattamente identica a quella del padre. Ciò vuol dire che quando il figlio andrà in esecuzione utilizzerà la tabella dei segmenti identica a quella del padre per riferire la sua memoria e di conseguenza andrà ad accedere alla stessa memoria del padre. In questa maniera non abbiamo bisogno di fare la copia della memoria perchè il figlio potrà tranquillamente andare ad accedere alla stessa memoria del padre, che è tutta condivisa. Fintanto che il figlio fa accessi in memoria in sola lettura nessun problema. Il problema è se il figlio e il padre iniziano a fare accessi in scrittura a questo punto potrebbero interferire l'uno con l'altro, questo bisogna evitarlo. <Uno studente> Come si fa ad indicare quali processi possono accedere ad un segmento condiviso? <Il Chessa perde il filo> Le risponderò dopo. Quindi io ho duplicato la tabella dei segmenti del padre nel figlio, il figlio accede alla stessa memoria del padre: se tutti e due accedono in lettura nessun problema, se però il figlio o il padre iniziano a scrivere sono guai perchè potrebbero inteferire l'uno con l'altro, questo lo devo impedire. Quando faccio la Fork marco tutti i segmenti utilizzabili in sola lettura, per cui non ho bisogno di copiare memoria: il figlio e il padre possono andare avanti se devono leggere la memoria, ma se qualcuno dei due la scrive causa un'eccezione. Stavolta però l'eccezione, quando viene intercettata da SO non produce un Segmentation Fault, perchè il SO si è tenuto a mente il fatto che un processo ha eseguito e l'altro ha, diciamo, subito la Fork. Per cui il sistema operativo sa che in questo caso la violazione di protezione è in realtà un effetto della Copy on

Write, che ha disabilitato i diritti di scrittura. Quindi soltanto in questa circostanza il SO prende il segmento e lo duplica. Fatto questo ripristina i diritti originari di entrambi i segmenti e da questo punto il processo padre e il processo figlio possono continuare ad andare avanti, modificando stavolta la copia della loro memoria, non l'originale. <Uno studente> (???) viene (???) il riferimento all'interno della tabella dei segmenti e (???) Supponiamo che il processo figlio faccia una scrittura: la scrittura dà eccezione, il SO interviene e si rende conto che quello è un processo figlio che stava cercando di accedere a della memoria, ma quella memoria era stata messa in sola lettura per effetto della Copy on Write. Quindi prende quel segmento, lo copia, lo duplica, nella tabella dei segmenti del figlio ci scrive il riferimento a questo nuovo segmento duplicato con il diritto di accesso corretto in scrittura e il figlio può ripartire. Da questo momento in poi il padre e il figlio sono separati: il padre ha il suo segmento che a questo punto può andare a scrivere perchè il diritto di scrittura gli è stato ripristinato e il figlio ha il suo segmento separato nel quale può scrivere o leggere. Il vantaggio è che tutti i segmenti di codice non li devo duplicare perchè quelli non si potrenno comunque scrivere, quindi quelli non dovranno essere copiati. Nel caso peggiore dovrò copiare soltanto i segmenti dati e i segmenti per lo stack: nel caso peggiore però. Perchè se i processi padre e figlio non vanno a modificare i segmenti dati quelli non li devo duplicare. Quindi complessivamente ho risparmiato un sacco di tempo. Provo a riformulare la domanda di prima del vostro compagno. Il vostro collega mi chiedeva "Supponiamo di utilizzare la segmentazione per condividere un segmento di una libreria, come fa il SO a saperlo e a configurare tutto questo?" Intanto dobbiamo partire dal link. Il codice che utilizza una libreria avrà un riferimento (una #include) per includere un file di libreria. Quindi il Linker lo sa che è stata utilizzata una libreria per quel codice. Le informazioni sulla libreria utilizzata vengono scritte nel modulo di caricamento, nel file eseguibile. Quando quel programma viene messo in esecuzione, diventa un processo, durante il caricamento il SO sa, si rende conto che quel processo utilizza una libreria e può anche sapere dove è stata allocata nello spazio virtuale di quel processo. Se si utilizza la segmentazione sa che è in un segmento separato. Il Linker stesso quando ha creato i collegamenti ha allocato quella libreria in un segmento separato. Quindi il SO questa informazione ce l'ha. A questo punto, al SO serve soltanto sapere se quella libreria è già presente in memoria per un altro processo, ma siccome tutti i processi sono stati caricati in memoria da SO, quando il SO va a caricare un processo che utilizza una libreria se lo ricorda, se lo segna. Per cui se dopo c'è da caricare un altro processo che usa la stessa libreria scorre la lista delle librerie caricate, vede in quali segmenti di memoria fisica sono caricate, se la trova inizializza coerentemente la tabella dei segmenti del processo che sta allocando adesso. <Uno studente> Quindi quando includiamo una libreria standard (IO, ecc) sono tutte già presenti. <Chessa> Chiaramente il SO non è obbligato a farlo e se ne può fregare, può caricare interamente e tutte le librerie sono duplicate per tutti i processi che le utilizzano, però vi rendete conto che è uno spreco. Per cui se un SO cerca di ottimizzare tutte le risorse (i sistemi attuali lo fanno) trae vantaggio da tale possibilità. I sistemi moderni in realtà non utilizzano segmentazione, utilizzano altre tecniche di gestione della memoria. <Uno studente> Quindi la copia viene effettuata prima che venga fatta la scrittura, la copia del segmento? <Chessa> Eh sì. Lui cerca di fare la scrittura, solleva l'eccezione, interviene il SO, capisce che è stata fatta la Copy on Write, ovviamente anche lui si sarà segnato quello che aveva da segnarsi. Quindi individua il padre e il figlio, scopre qual è il segmento sul quale il figlio voleva fare la scrittura, prende quel segmento e lo duplica, modifica la tabella dei segmenti del figlio in maniera tale da puntare al duplicato, setta il diritto di scrittura, va sulla tabella dei segmenti del padre e per quel segmento setta il diritto di lettura e fa ripartire il figlio. Ovviamente deve sapere che su quel segmento il diritto originale si poteva scrivere: se era un segmento codice

non si poteva scrivere neanche prima e neanche dopo. Quando si fa la Fork con la Copy on Write ci sono tante informazioni da tenere a mente. Niente di complicato.



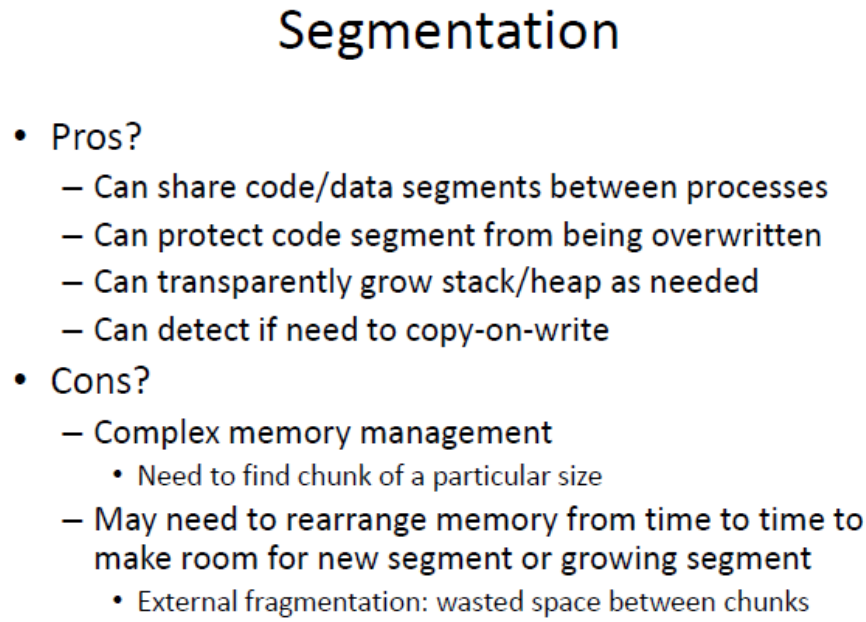
Questo è un esempio di 2 processi che condividono un segmento, in particolare il processo 1 nella tabella dei segmenti ha il primo segmento 0 che parte dall'indirizzo 1000 ed è lungo 500 ed è usabile in lettura e questo è effettivamente la stessa informazione presente nella tabella dei segmenti del processo 2, per cui questo è un segmento condiviso. Mentre invece gli altri segmenti non sono condivisi e tutti questi segmenti (che in totale sono 5) sono allocati in punti arbitrari della memoria, ovviamente in modo disgiunto. Domanda. Quando caricate un processo per l'esecuzione, quanta memoria dovete allocare per lo stack e quanta per lo heap? Perchè per il codice è facile. Il codice è stato compilato e sapete esattamente quante sono le istruzioni, quanto occupano e via discorrendo. Quindi il segmento codice, o i segmenti codice se ce n'è più di uno sappiamo esattamente quanto sono lunghi e come devono essere allocati. Se sono presente i dati statici lo sappiamo pure. È stato dichiarato dal codice, il compilatore e il Linker lo sanno e si può predisporre un segmento per i dati statici. Però i dati dinamici, quelli allocati con la malloc, non sappiamo quanti saranno, non sappiamo quanto dovrà crescere lo stack (?). Quindi quando andiamo a inizializzare un processo e dobbiamo allocare il segmento per lo heap e quello per lo stack, come lo allochiamo, quanto grande lo dobbiamo allocare?

Zero-on-Reference

- How much physical memory do we need to allocate for the stack or heap?
  - Zero bytes!
- When program touches the heap
  - Segmentation fault into OS kernel
  - Kernel allocates some memory
    - How much?
  - Zeros the memory
    - avoid accidentally leaking information!
  - Restart process

Zero. Possiamo non allocarlo. Non gli allochiamo un bel niente, perchè non appena il processo va a toccare il segmento stack, poi il segmento heap capiamo che quel segmento va allargato. Il processo ha diritto ad allargare il suo segmento heap/stack. Esse sono strutture che si devono poter allargare dinamicamente. Quindi andare a riferire il segmento heap/stack non è un problema se non è stato allocato.
Noi inizializziamo un stack/heap di dimensione 0, proibiamo l'accesso (escludiamo qualsiasi operazione), nel momento nel quale il processo fa riferimento a quel segmento automaticamente, soltanto in quel momento lo allochiamo. Mi correggo: non è proprio una mossa furba allocarlo di dimensione 0, perchè nel segmento stack c'è almeno un record di attivazione, che è quello del main, però quanto esso sia grande lo sappiamo, basta vedere la dichiarazione del main per saperlo. Lo stack può quindi essere allocato di dimensione pari al main. Lo heap può essere allocato di dimensione pari a 0. I SO offrono delle chiamate di sistema per allargare i segmenti. In Unix ha un nome impronunciabile, mi sembra si chiami SDRK. E' una chiamata di sistema che serve ad allargare la dimensione di un segmento. Per cui se c'è bisogno questi segmenti si possono allargare. <Uno studente> Ma il segmento dello stack o dello heap è un segmento solo? <Chessa> Il segmento dello stack è opportuno che sia un segmento solo perchè sarebbe molto scomodo gestire uno stack sparpagliato su più segmenti. Il vostro collega ha chiesto due cose: ma il segmento dello stack è uno solo o più di uno? Ma il segmento dello heap è uno solo o più di uno? Vi dico che in realtà la risposta non la so, però posso fare un ragionamento. Se io ho un processo Unix con un solo thread, avere uno stack diviso su più segmenti è molto scomodo, perchè lo stack si utilizza con delle Push e delle Pop che vanno a incrementare e decrementare automaticamente lo stack pointer. Se lo sparpaglio su più segmenti c'è da spararsi. D'altra parte ha senso, assolutamente, allocare un segmento per ogni stack di ogni thread. Quindi se ho un processo multithread, ogni thread potrebbe avere il proprio stack allocato su segmenti differenti. <Uno studente> Se finisco il segmento? <Chessa> Ma il segmento non finisce, perchè dovrebbe finire? <Studente> Se faccio troppe Push <Chessa> Sì, ma quante ne deve fare per finire il segmento? Molte. A tutto c'è un limite, però il segmento può crescere molto. Provo a darvi dei numeri se riuscite a tenerli a mente, sennò ve li scrivo alla lavagna. Prendiamo un sistema che ha indirizzi a 32 bit, quindi dal processore il campo indirizzi esce a 32 bit, nudo e crudo. Nei sistemi a segmentazione quello che si dice è: una parte di questi bit sono l'Indice del segmento, una parte è l'Offset. Con la segmentazione, in generale, il numero dei segmenti è limitato, quindi voi riservate un numero limitato di bit per il segmento. Supponiamo di imporre un tetto massimo del numero di segmenti che il processo può avere pari a 16. Questo vuol dire che per rappresentare l'Indice del segmento mi bastano 4 bit. Questo vuol dire che mi restano 28 bit per l'Offset. Con 28 bit posso avere un segmento grande 256 MByte. 2^20 è un Mega, 2^8 è 256. Se voglio avere più segmenti potrei variare questi numeri qui, ma di poco. Quindi generalmente con la segmentazione il numero di segmenti può essere un po' alto, ma non enorme. Per cui il singolo segmento può essere anche piuttosto grande, per riempire 256 MByte sullo stack ci vuole un certo impegno (è comunque possibile). D'altra parte, però, tenete presente che qualsiasi sistema di gestione della memoria che avete, se voi esagerate voi riempite la memoria... <Uno studente> Per lo heap? <Chessa> Per lo heap lo stesso ragionamento. C'è anche un'altra questione. Il numero di segmenti che utilizzo per allocare gli stack, lo heap, ecc, non dipende tanto dal SO, ma dipende molto dal linguaggio. Perchè è il linguaggio in fase di compilazione e linking che decide quanti sono i segmenti. Ed è il supporto a tempo di esecuzione che riceve la malloc e che poi le implementa con le chiamate di sistema sottostanti che decide se deve allocare un nuovo segmento oppure no. Il SO non ha la malloc come chiamata di sistema, la malloc viene eseguita da una funzione di libreria, dal supporto a tempo di esecuzione. Il SO offre delle chiamate di sistema per allargare i segmenti, ma sono elementari, come la SDRK che allarga o restringe un segmento. Per cui, in realtà, è il

supporto a tempo di esecuzione che decide se deve allocare un nuovo segmento oppure no: ecco perchè non so la risposta in realtà, ma ci posso arrivare a logica, perchè io insegno SOL e non insegno Linguaggi, sono tenuto a non saperlo. Possiamo allocare 0 memoria per un segmento e lasciare che questo cresca dinamicamente. Tra l'altro così facendo possiamo anche riazzerare la memoria dinamicamente, in maniera tale da dare ad un processo memoria azzerata in modo tale che non possa acquisire informazioni di rimando dalla memoria deallocata agli altri processi. Punti di forza e di debolezza della segmentazione. Possiamo



condividere segmenti e la condivisione non soltanto è facile, ma segue anche una logica: abbiamo visto almeno 2 modi per condividere librerie o segmenti con la tecnica Copy on Write usata con la Fork, che è facile anche dare informazioni al SO sui segmenti da condividere, si fa molto bene con la segmentazione. L'altro vantaggio è che possiamo avere protezione specifica per il singolo segmento: tale segmento lo possiamo proteggere in scrittura e in lettura. Poi abbiamo dei diritti specifici per i segmenti, quindi possiamo proteggere il codice, differenziarlo dai dati, dallo heap e via discorrendo. Possiamo far crescere i segmenti di heap e di stack dinamicamente, quindi non abbiamo bisogno di sapere a priori quanto spazio allocare. Problemi. Certamente è più complesso rispetto al Base & Bound: se prima avevamo 2 registri stavolta dobbiamo gestire una tabella dei segmenti, con una traduzione che diventa un pochettino più articolata, perchè dobbiamo estrarre dalla tabella dei segmenti e poi dopo abbiamo almeno controlli di protezione (se non 3): il primo sull'Indice del segmento, il secondo sull'Offset, il terzo sul diritto di accesso lettura-scrittura. E poi abbiamo il Lookup della tabella. L'altro problema della segmentazione è che in realtà il problema della frammentazione esterna, che si osservava con la tecnica delle partizioni variabili, in realtà è sempre presente. Con le partizioni variabili, il problema della frammentazione esterna nasceva dalla allocazione di processi interi, stavolta nasce dall'allocazione di segmenti: siccome i segmenti devono essere contigui per effetto delle successive de/allocazione dei segmenti, potrei trovarmi ad avere la memoria libera, divisa in tanti pezzettini, frammenti troppo piccoli per poter essere utilizzati per allocare nuovi segmenti. Il problema con la segmentazione è minore rispetto al problema che avevo con la tecnica delle partizioni variabili, perchè con la segmentazione, presumibilmente, i segmenti sono più piccoli e quindi sono più facilmente collocabili, però è un problema sempre presente. <Uno studente> Se io faccio i segmenti della stessa dimensione <Chessa> A frammentazione interna <Studente> Se io ho uno spazio libero di tutto un segmento o è libero tutto o no. <Chessa> Il vostro collega dice: "Io potrei fare i segmenti tutti

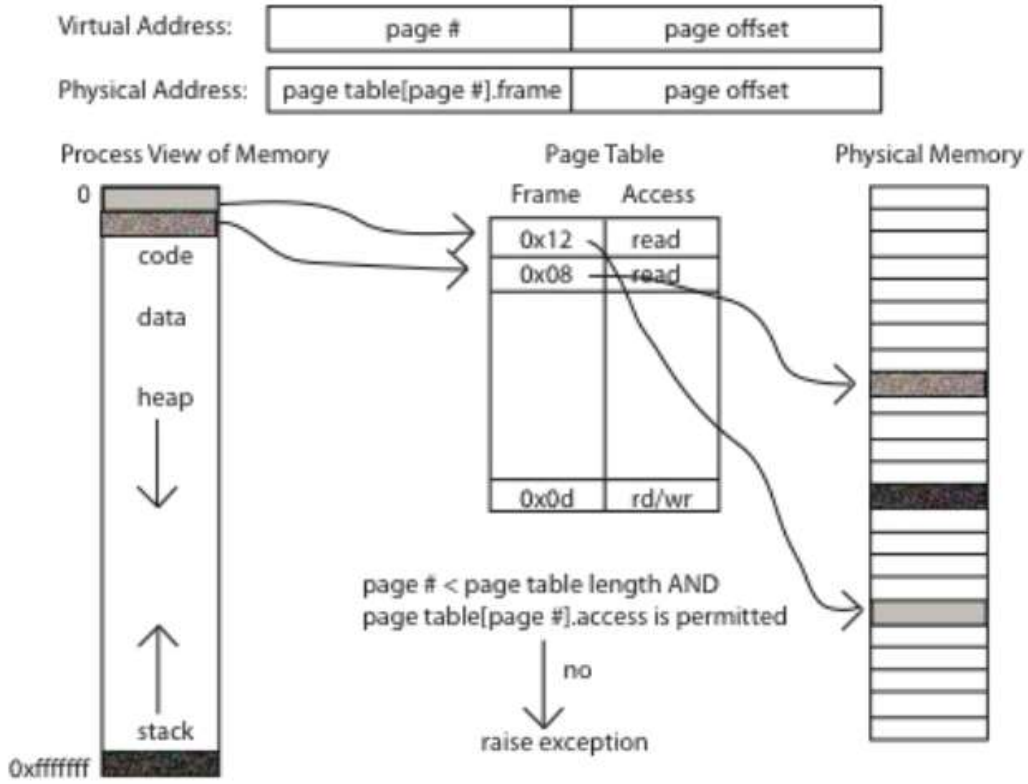
della stessa dimensione, così li metto bene". Come la collezione di Topolino di mio figlio, sono tutti quanti della stessa dimensione, quindi ne togli uno e ne metti un altro. Però il problema è grosso, perchè se io impongo che tutti i segmenti debbano avere la stessa dimensione posso scrivere con pochissima fatica codice che occupa 100 Byte e che utilizza 20 Mega di dati. Se impongo che tutti i segmenti abbiano la stessa dimensione faccio i segmenti di 20 Mega, quindi spreco quasi 20 Mega nel segmento codice. Sarebbe una buona idea, ma va declinata diversamente. E' il prossimo argomento. Comunque sia la segmentazione presenta il problema della frammentazione, quindi dobbiamo cambiare approccio. Ci sono sistemi che sono nati e vissuti con la segmentazione: le stesse piattaforme Intel, quelle che sono all'origine degli attuali i3, i5, i7, i vecchissimi Intel (???) ecc. sono nati con la segmentazione. Il commento del vostro compagno viene proprio perfetto. Per evitare la frammentazione esterna, come ha acutamente osservato il vostro collega, avete tutto più facile se tutti quanti i segmenti avessero la stessa dimensione. Però, come vi dicevo prima, con i segmenti tutti della stessa dimensione ho un problema, perchè li devo fare troppo grande per poter contenere le informazioni e rischio di sprecare tanta memoria quando li alloco. Quindi la vera chiave che ci permette di fare un ulteriore passo è di slegare l'idea di un segmento associata ad una qualche semantica: in questo momento con la segmentazione voi avete l'idea che il segmento sia il segmento codice, il segmento dati, segmento heap, segmento stack, segmento libreria. In qualche modo i segmenti hanno un loro significato e questo significato ne determina anche la dimensione e questo ci impedisce di avere una allocazione efficiente perchè ogni volta devo trovare lo spazio per allocare i segmenti e produco frammentazione esterna. Se io invece divido la memoria in cubettini tutti della stessa dimensione (questo lo ha reinventato l'Ikea la stessa cosa) li posso certamente allocare più agevolmente, però questo fa a pugni con l'idea che ogni blocchetto deve avere una sua semantica, in qualche maniera. Come si risolve questa cosa? E' stata risolta con la paginazione.

Paged Translation

- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length

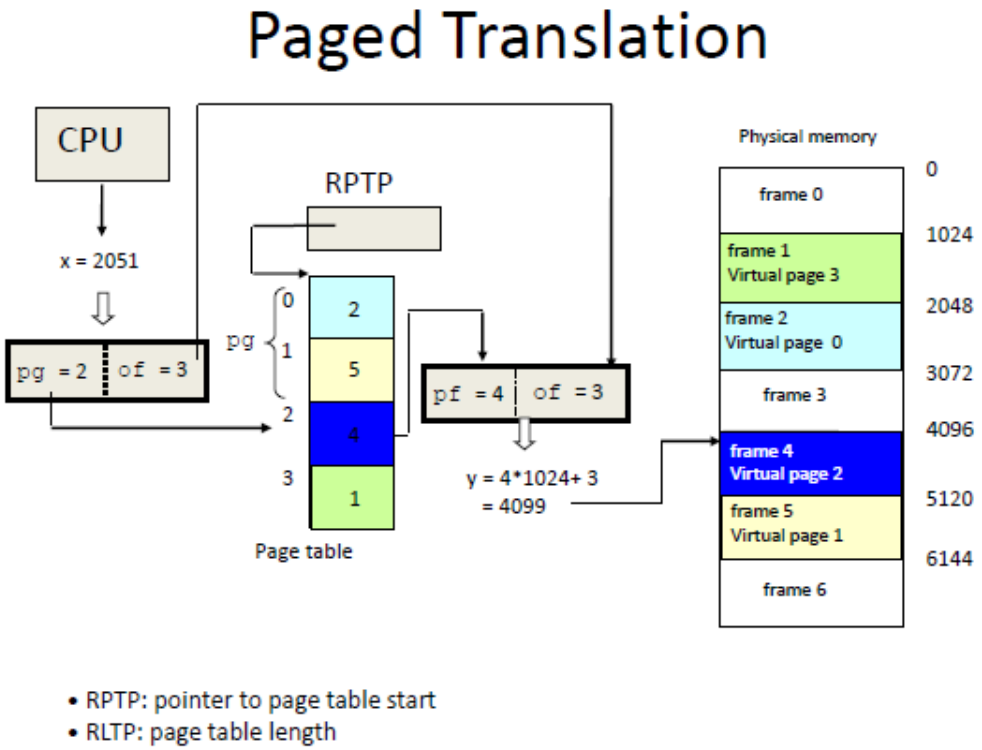
Con la paginazione l'idea è questa: divido la memoria in blocchi di dimensione fissa, tutti uguali e tutti molto piccoli; ogni blocco tipicamente ha una dimensione di pochi K, potrebbe essere 1K, 2K, 4K, le dimensioni tipiche sono queste. Quando mi si chiede di allocare memoria alloco un blocco intero oppure non lo alloco. La memoria fisica è divisa in questa maniera. Per sapere se un blocco è libero o occupato non ho più bisogno come nella segmentazione di mantenere delle strutture dati che mi dicono quali sono i frammenti liberi,

perchè posso usare delle strutture dati molto efficienti: delle Bitmap (mappe di bit). Ogni bit è associato ad un blocco: se il bit è 1 il blocco è occupato, se il bit è 0 il blocco è libero. Quando qualcuno mi chiede di allocare 10 Mega e so che i blocchi hanno tutti dimensione 1K, a me basta trovare 10000 blocchi liberi, dove stanno stanno. Il processo viene allocato in un certo numero di blocchi di memoria fisica che sono sparpagliati ovunque in memoria principale: devo sapere dove sono e in che ordine sono, perchè se non so queste cose non posso ricostruire la struttura della memoria del processo. Questa informazione la metto in una struttura dati che si chiama tabella delle pagine. Vediamola dal punto di vista del processo.



Con la paginazione la memoria virtuale del processo è uno spazio di memoria contiguo che parte dall'indirizzo 0 e arriva fino ad un certo indirizzo massimo (che è molto grande nel caso della paginazione), pari al massimo indirizzo rappresentabile. Quindi se ho un processore a 32 bit con tali bit posso indicizzare 4 GByte, lo spazio virtuale assegnato ad un processo è 4 Gbyte, tutta la memoria è indirizzata. Il processo non lo sa ma la sua memoria in realtà è divisa in pagine che sono blocchettini di dimensione fissa nell'ordine di 1, 2, o 4 K. Diciamo 1K. Queste pagine che nello spazio virtuale sono contigue, ognuna di loro in realtà è allocata in un blocco fisico della stessa dimensione che può essere in un punto arbitrario. Questo vuol dire che se noi prendiamo questo processo che sta utilizzando 3 pagine, da 0 a 1 è sempre ffff e tutto il resto della sua memoria virtuale è libera, non l'ha ancora utilizzato, queste 3 pagine possono essere allocate in blocchi fisici arbitrari. Tramite la tabella delle pagine è possibile ricostruire l'associazione tra pagine virtuali e blocchi fisici che le contengono. Per cui se il processo genera un indirizzo che fa riferimento a una cella di memoria che sta nello stack, quindi genererà l'indirizzo ffffff2, il meccanismo di traduzione degli indirizzi tramite la tabella delle pagine scopre che questa pagina è allocata in questo blocco fisico e quindi è in grado di andare a prendere la cella corrispondente a quell'indirizzo. In questo modo ho eliminato completamente il problema della frammentazione, perchè appunto tutte le pagine sono allocate in blocchi di dimensione uguale: se il processo deve far crescere lo stack (in questo caso ha bisogno di un ulteriore pagina) dovrò allocare un'ulteriore pagina fisica in memoria principale, però questa può essere una qualsiasi. <Uno studente> Praticamente sarebbe lo

stesso meccanismo del primo ma ho diminuito la dimensione. (???) segmenti (???) <Chessa> Apparentemente sì, poi concretamente cambia tutto. Perchè quello che succede è che (???) semantica, in qualche maniera. E questo ha forti implicazioni. Il meccanismo di traduzione degli indirizzi non è molto differente, è molto simile in realtà. Come funziona la traduzione degli indirizzi con la paginazione?



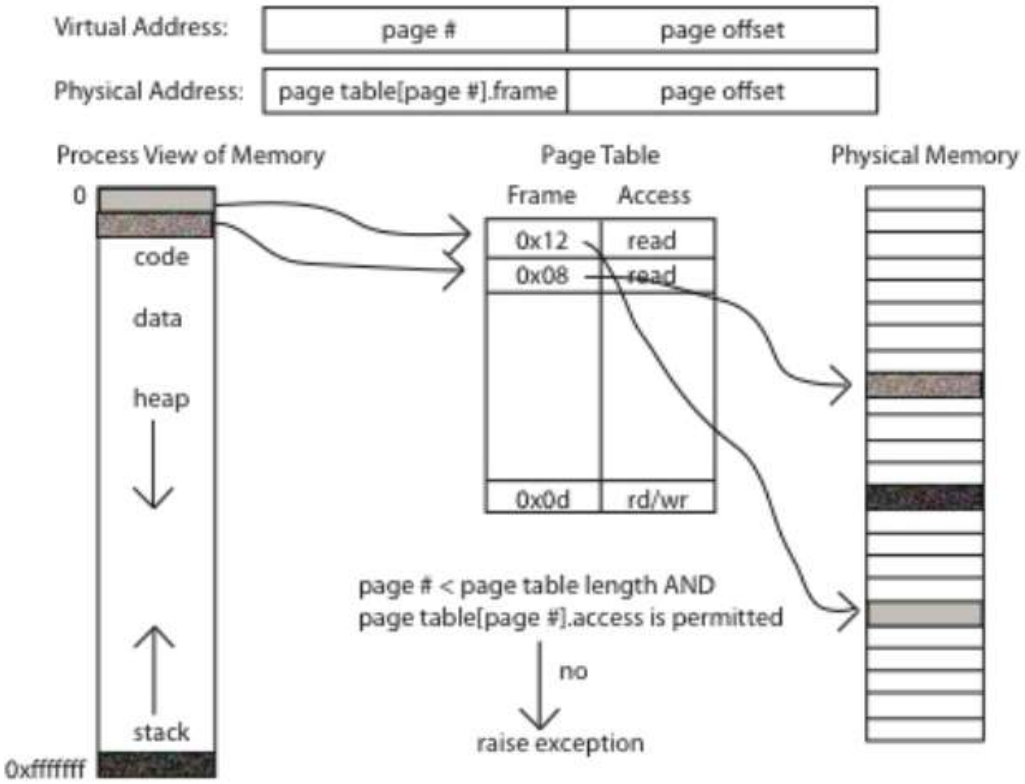
- RPTP: pointer to page table start
- RLTP: page table length

Il processore genera un indirizzo virtuale, tale indirizzo è una stringa di bit che viene diviso in due campi, Indice di pagina e Offset. Stavolta il primo è abbastaza grosso perchè di pagine ne ho tante, in generale. L'Offset, tipicamente, potrebbe essere una decina di Bit, quindi se avete indirizzi a 32 bit l'Indice di pagina può essere formato da 22 bit. Un indice di pagina di 22 bit vuol dire che avere a disposizione 4 milioni di pagine circa. Il processore genera un indirizzo virtuale, viene diviso in due campi Indice di pagina e Offset. Il primo viene utilizzato per puntare alla tabella delle pagine: qui non ho bisogno di fare un controllo di protezione perchè il processo ha la possibilità di indicizzare una qualsiasi pagina nel suo indirizzo virtuale. Quindi qualsiasi indice di pagina va bene. L'Indice di pagina punta ad una qualsiasi pagina, non c'è una verifica di protezione. Dalla tabella delle pagine, in corrispondenza della riga corrispondente all'Indice di pagina si estrae l'Indice della pagina fisica, (in questo caso la pagina 2 è caricata nella pagina fisica 4) , si prende questo indice di pagina fisica, lo si combina con l'Offset, stavolta è una semplice giustapposizione (?), vengono uniti l'uno con l'altro e questo produce un indirizzo fisico che va a puntare in memoria principale. L'unica cosa che ci serve è l'indirizzo della tabella delle pagine per sapere dove si trova.

Paged Translation

- Manage memory in fixed size units, or pages
- Finding a free page is easy
  - Bitmap allocation: 0011111100000001100
  - Each bit represents one physical page frame
- Each process has its own page table
  - Stored in physical memory
  - Hardware registers
    - pointer to page table start
    - page table length

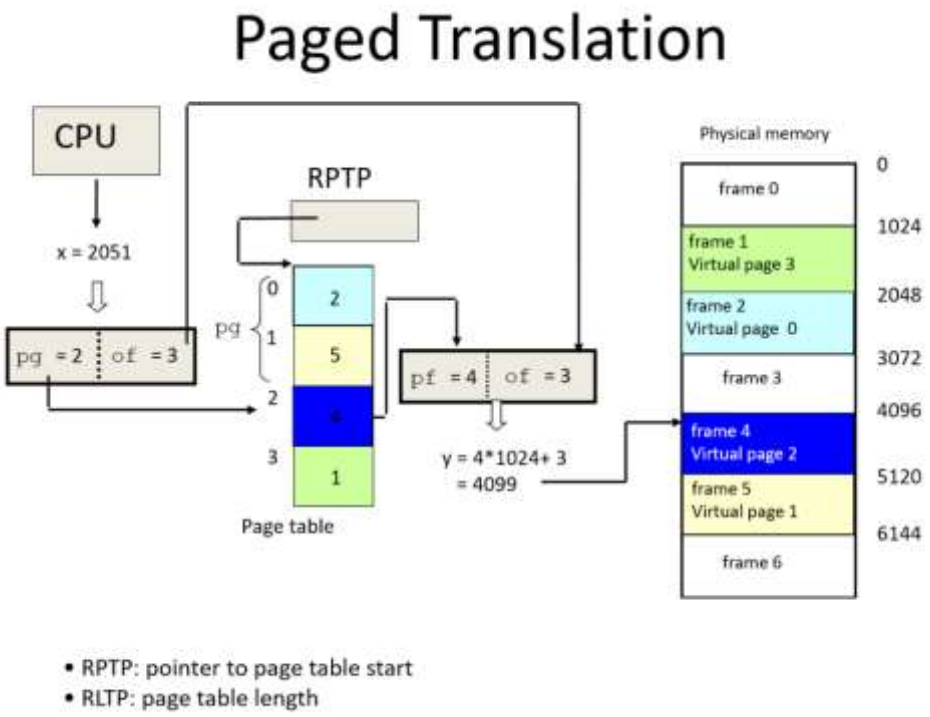
Ripartiamo dalla Paginazione, se vi ricordate prima della Paginazione abbiamo visto altre tecniche della gestione della memoria che riguardano: le partizioni variabili oppure la Segmentazione, ed entrambi questi sistemi avevano il limite di dover allocare la memoria a blocchi contigui, di dimensioni variabili, per cui creavano delle difficoltà di allocazione al sistema operativo, soprattutto dovute al fenomeno della frammentazione esterna.



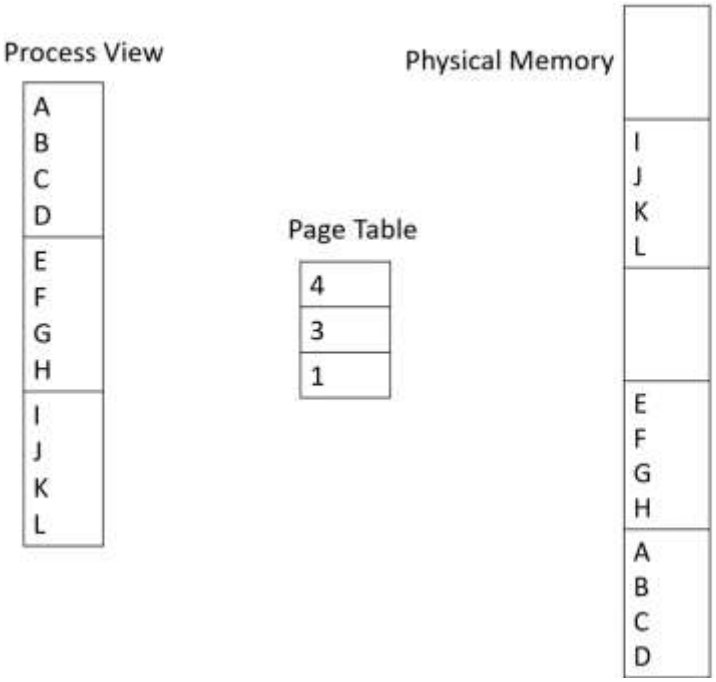
Questo problema viene risolto con la tecnica della Paginazione. Nella Paginazione, lo spazio di memoria visto dal processo non è più segmentato, ma è un unico spazio di memoria contiguo che va dall'indirizzo 0

fino all'indirizzo massimo rappresentabile, quindi con 32 bit -> 4GB di memoria virtuale interamente a disposizione del processo, 64 bit -> Milioni/Miliardi di GB. Il processo vede questo spazio virtuale che in qualche modo gestiste. Lo spazio virtuale, in qualche modo non visibile al processo è diviso in pagine che hanno dimensione fissa (tipicamente 4KB), a sua volta la memoria fisica è divisa in blocchi della stessa dimensione delle pagine, ed il risultato è che ogni pagina di ogni processo può essere allocato in un qualsiasi blocco della memoria. La relazione tra la pagina del processo nello spazio virtuale e il blocco in memoria fisica che la contiene, viene mantenuta nella tabella delle pagine.

La tabella delle pagine mantiene questa associazione, in questo modo: contiene una riga per ogni pagina del processo, quindi viene indicizzata con l'indice della pagina virtuale e all'interno della tabella delle pagine troviamo, per ogni riga, un descrittore di pagina che contiene l'indice di pagina fisica (l'indice del blocco fisico su cui la pagina è allocata, per esempio la pagina virtuale 0 è allocata sul blocco fisico 12) e i diritti di accesso. In questo modo abbiamo una granularità molto fine nel definire i diritti di accesso allo spazio di memoria perché possiamo isolare pagina per pagina.



Il meccanismo complessivo di traduzione degli indirizzi avviene secondo questo schema; il processore genera indirizzi virtuali (32-64 bit), l'indirizzo virtuale che fa riferimento ad un unico spazio virtuale contiguo viene visto dall'MMU come composto da due campi: indice di pagina e un Offset. L'indice di pagina viene usato per indicizzare la tabella delle pagine, dalla quale si estrae il descrittore di pagina, dal descrittore si prende l'indice del blocco fisico, e a questo punto, si combina l'indice del blocco fisico con l'offset per produrre l'indirizzo fisico che poi viene inviato alla memoria.



Quindi, dal punto di vista del processo (ora, questo è un esempio molto semplificato, è un processo che ha un spazio virtuale fatto di 3 pagine formate da 4 byte) lo spazio virtuale ha questa struttura, (se andiamo a riempire i vari byte con questi caratteri vanno in questo ordine). D'altra parte, ogni pagina è allocata in un blocco fisico da qualche parte in memoria principale tramite il meccanismo dato dalla tabella di traduzione, per cui in memoria fisica lo spazio virtuale è distribuito in una maniera arbitraria.

Per esempio, qual è l'indirizzo fisico corrispondente all'indirizzo virtuale 6?

Se prendete l'indirizzo virtuale 6, questo corrisponde alla stringa binaria 000110, ora, siccome le pagine sono formate da 4 byte, l'offset è formato da 2 bit (con 2 bit posso indicizzare 2^2=4 posizioni), quindi i primi 2 bit generati dal processore sono l'offset in questo esempio (Chessa legge da dx verso sx per qualche motivo), i primi 2 bit sono 10 quindi tradotto in decimale 2. Invece l'indice di pagina è 0001, in decimale resta 1. Quando vado nella tabella delle pagine nella posizione 1 (l'indice di pagina è 1) estraggo il valore "3", questo vuol dire che mi rimanda al blocco fisico 3 con Offset 2. Quindi 3 in binario è 0011 e 2 è 10 se ricompongo tutto questo ottengo la stringa 001110 che corrisponde a 14.

INDIRIZZO VIRTUALE = 6

INDIRIZZO FISICO = 14

Paging Questions

- What must be saved/restored on a process context switch?
  - Pointer to page table/size of page table
  - Page table itself is in main memory
- What if page size is very small?
- What if page size is very large?
  - Internal fragmentation: if we don't need all of the space inside a fixed size chunk

Quando c'è una commutazione di contesto, ammettendo di usare la paginazione, che cosa devo salvare? Come al solito devo pensare a quello che deve stare nel processore per gestire la paginazione, che cosa ci sta? L'MMU per tradurre l'indirizzo ha bisogno di sapere dove sta la tabella delle pagine, quindi in linea di principio, il puntatore è alla tabella delle pagine. Tenete presente che in realtà la traduzione delle pagine, l'MMU non la può fare realmente andando a leggere la tabella delle pagine. La tabella delle pagine è grande e sta in memoria principale, l'MMU utilizzerà delle CACHE per questa tabella delle pagine, ne parleremo dopo. È chiaro che per fare questa traduzione comunque l'MMU deve un accesso alla tabella, se non altro per scaricarsi la sua CACHE interna, i dettagli li vediamo dopo però, serve comunque conoscere l'indirizzo della tabella delle pagine, proprio per permettere questo scambio di informazioni. Ci serve quindi di salvare il puntatore della tabella delle pagine ed eventualmente anche la dimensione della tabella.

Quindi, rispetto alla commutazione di contesto che abbiamo già visto, ci sono queste informazioni in più che vanno salavate e che vanno caricate per il nuovo processo.

Ora, qual è la dimensione delle pagine? Io vi ho detto che nella paginazione, una pagina può essere nell'ordine di 1Kb – 4Kb. In Windows la dimensione delle pagine è di 4KB, tanto per darvi un'idea. Ma da dove viene fuori questo numero e cosa succede se si prendono dei valori più piccoli? Oppure cosa succede se si prendono dei valori molto grandi? Vi faccio notare che se prendete delle pagine molto grandi, la memoria viene allocata in pagine intere, perché una pagina viene allocata in memoria fisica, non posso allocare mezza pagina. Ora, come vengono gestite queste pagine? Immaginate di dover allocare lo spazio per il codice, difficilmente che il codice abbia dimensione tale da pareggiare la dimensione di un'intera pagina, d'altra parte voi dovete sempre allocare un numero intero di pagine. Questo vuol dire che, quando andate ad allocare il codice o una struttura dati o uno stack, c'è sempre una porzione dell'ultima pagina che non è mai utilizzata interamente in media, mezza pagina è inutilizzata. Se il vostro codice sono 3KB e le pagine sono di 4KB, questo vuol dire che 1Kb resta inutilizzato. Se il codice sono 6KB dovete allocare 2 pagine e sprecare 2KB.

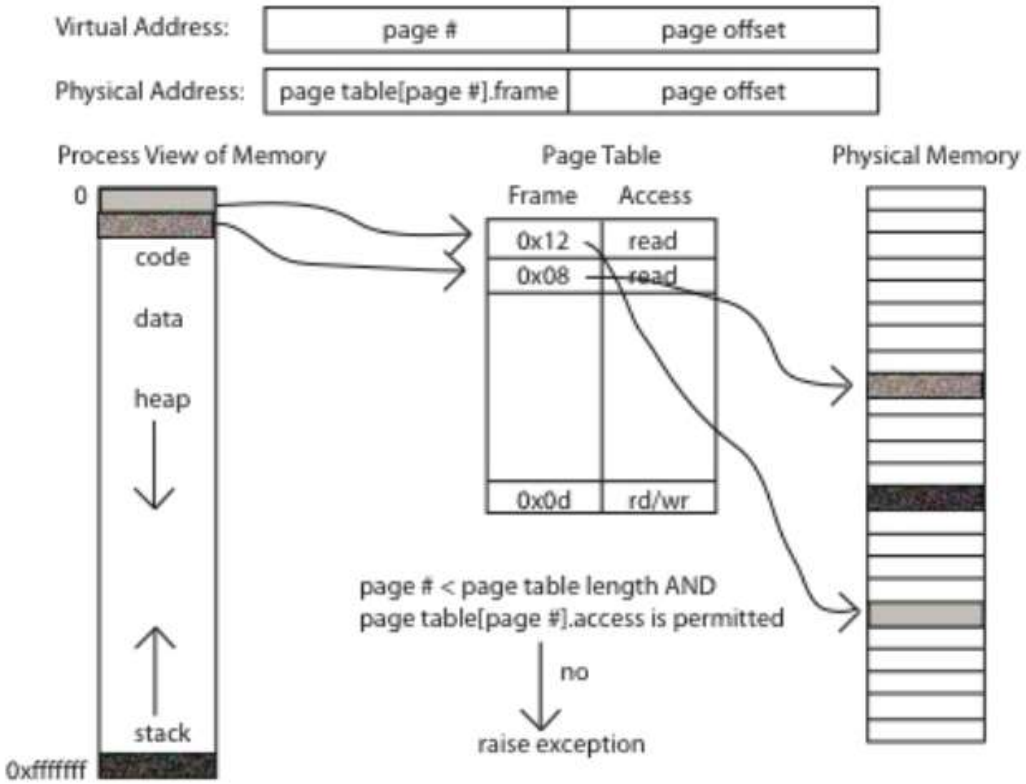
Se le pagine sono piccole questo spreco, in realtà è trascurabile; con pagine di 4KB se ogni tanto andate a sprecare qualche KB non è un problema. Se però la dimensione delle pagine aumenta, questo spreco di memoria può non essere più trascurabile. Questo spreco di memoria è detto FRAMMENTAZIONE INTERNA

per distinguerlo da quella Esterna, se vi ricordate la Frammentazione Esterna è quella che abbiamo con la Segmentazione o con le Partizioni virtuali quando ci ritroviamo ad avere degli spazi virtuali liberi di memoria tra due aree occupate ma questi spazi di memoria sono piccoli e non possono essere usati singolarmente, quindi contribuiscono ad uno spreco. Frammentazione Esterna perché si tratta di memoria allocata all'esterno della memoria dei processi allocati. Invece nel caso della Paginazione, se usate pagine molto grandi aumentate il fenomeno della frammentazione interna (spazio che è allocato ai processi ma che essi non usano).

Quindi aumentare la dimensione della pagina aumenta la frammentazione interna, questo significa che è conveniente avere pagine piccole, ma che succede se le pagine sono davvero molto piccole? Potrei avere pagine di 1 byte?

COLLEGA: No, perché dovrei fare tanti accessi in memoria.

CHESSA: Questa cosa qui non ha significato nella paginazione. Guardiamo questo processo:



Questo processo ha uno spazio virtuale che è tutto mappato in memoria principale ad un qualche indirizzo, in realtà, a me non interessa che un certo byte da caricare stia in una pagina piuttosto che in un'altra, se devo andare a leggere un byte qui o un byte lì non fa nessuna differenza. L'MMU prende l'indirizzo virtuale, lo traduce con la tabella delle pagine e va in memoria principale, quindi che io stia andando a leggere informazioni che stanno nella stessa pagina o in due pagine differenti (allo stato attuale) non cambia niente.

Il problema sta nella dimensione della tabella delle pagine, se io faccio pagine di 1Byte dovrei avere una tabella che ha un descrittore di pagina per ogni pagina e quindi per ogni Byte. Ora, quanto è grande ogni riga di questa tabella? Beh, ogni riga deve contenere: l'indice di un blocco fisico (quasi un indirizzo fisico, circa 4 Byte) e poi un certo di indicatori relativi ai diritti di accesso e alla protezione. Quindi una riga di ogni pagina può tranquillamente occupare 4 Byte o 8Byte (a seconda dell'architettura). Se io facessi pagine di 1 Byte la tabella delle pagine occuperebbe 8 volte la memoria che sto allocando al processo. Non ha molto

senso. La tabella dei processi deve servire per facilitare i processi, ma se va ad occupare più spazio di quanto ne occupano i processi per fare il loro lavoro, allora non ci siamo.

Quindi se faccio pagine troppo piccole, il problema è che occupo troppo spazio. Utilizzando come fa Windows pagine di 4Kb: Una pagina è di 4KB, il suo descrittore potrebbe essere di 4 Byte, questo vuol dire che la tabella delle pagine occupa idealmente l'1/1000 dello spazio occupato dal processo. È più ragionevole.

Restano tutta una serie di altre questioni aperte con la Paginazione; va bene, diciamo di aver risolto tutti i problemi della paginazione, di farla funzionare, di aver creato la tabella delle pagine, di poter fare la traduzione degli indirizzi in maniera efficiente, di essere soddisfatti con l'overhead. Se vi ricordate con la Segmentazione avevamo alcuni vantaggi: potevamo fare la Copy on Write nel caso della Fork, potevamo condividere segmenti. Questa cosa la possiamo fare ancora, se utilizziamo la Paginazione? Beh, la risposta è sì, la possiamo fare con un po' di accortezze.

Paging and Copy on Write

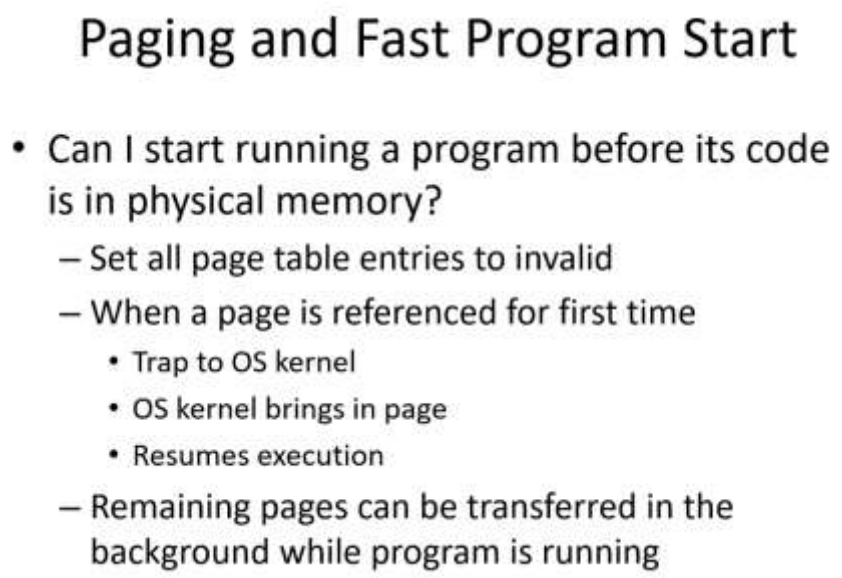
- Can we share memory between processes?
  - Set entries in both page tables to point to same page frames
  - Need core map of page frames to track which processes are pointing to which page frames
- UNIX fork with copy on write at page granularity
  - Copy page table entries to new process
  - Mark all pages as read-only
  - Trap into kernel on write (in child or parent)
  - Copy page and resume execution

Se io voglio far condividere una pagina tra due processi, basta che nella tabella delle pagine di quei due processi faccia puntare allo stesso blocco fisico, quindi in questo modo la pagina resta condivisa, per cui posso implementare con questo accorgimento la Copy on Write della fork di Unix, quando creo il processo figlio, duplico la tabella delle pagine (padre e figlio hanno memorie virtuali mappate sulla stessa memoria fisica), dopodiché marco tutte le pagine in sola lettura, per cui non appena il padre o il miglio cercano di scrivere, si solleva un eccezione e a questo punto il sistema operativo deve ricordarsi che non è una violazione di protezione, ma una eccezione dovuta alla copy on write e quindi soltanto a questo punto, il sistema operativo duplica la pagina. Abbiamo in effetti un vantaggio rispetto alla segmentazione, facendo la copy on write, perché nel caso della segmentazione se il figlio vuole modificare un segmento, devo duplicare l'intero segmento, se il figlio vuole modificare 1Byte devo duplicare soltanto la pagina che contiene quel byte. Quindi in questo modo la Copy o write, viene a costare molto meno, perché duplico molto meno.

Per implementare correttamente la copy o write, immaginate quello che succede quando dopo aver applicato questa tecnica, il processo padre per esempio, va a modificare un byte di una certa pagina, quella pagina è protetta in sola lettura quindi si solleva un'eccezione, a questo punto il sistema dice "benissimo devo fare la copy on write, quindi devo prendere la pagina il cui accesso ha causato l'eccezione e copiarla,

quindi devo differenziare questo descrittore di pagina dalla tabella delle pagine del figlio", è tutto fatto? Fino ad un certo punto, perché devo sapere quella pagina che il processo padre ha cercato di modificare a quale tabella delle pagine di quale figlio fa riferimento, ed io l'informazione che ho è: l'indirizzo virtuale generato dal padre e l'indirizzo fisico del blocco fisico associato alla memoria principale. Per risalire alla tabella delle pagine del figlio mi servirebbe l'informazione inversa, dall'indirizzo fisico sapere quali sono i processi che hanno una pagina su quell'indirizzo fisico, quindi avere un'informazione inversa. Questa informazione inversa spesso nei sistemi operativi viene memorizzata (per tanti motivi) in strutture dati che si chiamano Core map. Quindi la Core Map mi dice per ogni pagina fisica quale pagina virtuale di quale processo c'è allocata sopra, se è condivisa di quali processi. Avendo questa informazione posso implementare correttamente la copy on write.

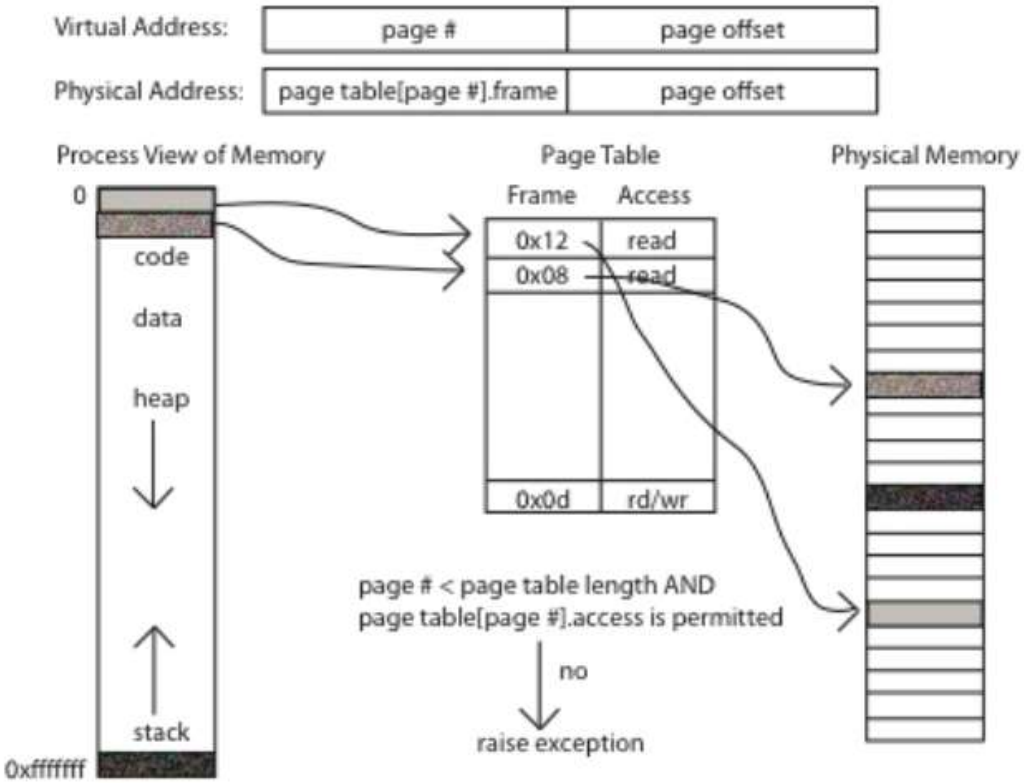
Ci sono altre questioni, con la Segmentazione se vi ricordate posso mettere in esecuzione un processo senza aver caricato tutti i suoi segmenti, nella tabella dei segmenti marco tutti i segmenti come invalidi, metto in esecuzione un processo, non appena questo genera un indirizzo l'MMU va a tradurre questo indirizzo, ma nella traduzione dell'indirizzo si rende conto che non può tradurlo perché il descrittore del segmento riferito non è valido, questo provoca un'eccezione e il sistema operativo in risposta va a caricare il segmento dal disco tipicamente. Quindi questo permette di mettere in esecuzione rapidamente i processi. La stessa cosa e anche meglio, la posso fare con la Paginazione.



Quando devo mandare in esecuzione un processo non ho bisogno di caricare tutte le pagine del processo in memoria principale. Inizializzo tutti i descrittori nella tabella delle pagine come invalidi, per cui non appena il processo genera un indirizzo, la MMU non può tradurlo perché il descrittore associato a quell'indirizzo è settato come invalido, questo comporta l'intervento del sistema operativo che va a caricare la pagina richiesta. Di tutto questo, in particolare di questo aspetto, come si implementa la paginazione a domanda avremo modo di parlarne.

Una volta che abbiamo caricato la pagina del processo di cui abbiamo bisogno, mentre il processo continua la sua esecuzione usando i dati in questa pagina, nel frattempo possiamo caricare le altre pagine che gli stanno intorno.

Ok, tutto bene? NO. C'è un piccolo problema, e questo piccolo problema, in effetti, è stato anticipato qui:



Guardate questo processo, ha uno spazio virtuale molto grande, giustamente pari allo spazio indirizzabile, d'altra parte è stata utilizzata una porzione molto piccola, lui sta utilizzando:

- Pagina 0 -> Codice
- Pagina 1 -> Dati
- Pagina ffffffff -> Stack

Tutte le altre pagine in questo momento sono inutilizzate.

Che cosa succede in realtà quando i processi lavorano? I processi devono allocare una parte di codice nota a priori, in questa parte di codice probabilmente dovranno caricare anche delle librerie ma per permettere la condivisione delle librerie con altri processi non è bene che queste librerie stiano attaccate al codice, è bene che stiano un po' separate; quindi quando il compilatore va a stabilire come deve essere allocata la memoria per quel processo, in effetti già al momento della compilazione va a distribuire nella memoria virtuale: codice, librerie e altri pezzi di codice in una maniera un po' sparpagliata per poterlo utilizzare meglio, non solo immaginate che questo processo sia pure multithread, se è multithread avrà bisogno di un certo numero di stack (1 per ogni thread). Dove li alloco gli stack? Non li posso allocare tutti quanti a partire dall'indirizzo ffffffff (si possono sovrapporre), non posso nemmeno metterli uno attaccato all'altro perché se lo stack di un thread deve crescere mi fa comodo avere lo spazio per farlo crescere. Quindi un altro accorgimento che adottano i compilatori è quello di allocare gli stack un po' distanziati all'interno dello spazio virtuale. Quando poi fate una malloc per allocare una grossa struttura dati, di nuovo, dovete prendere dello spazio dallo heap e questo finirà in un'altra regione dello spazio virtuale, separata dalla altre.

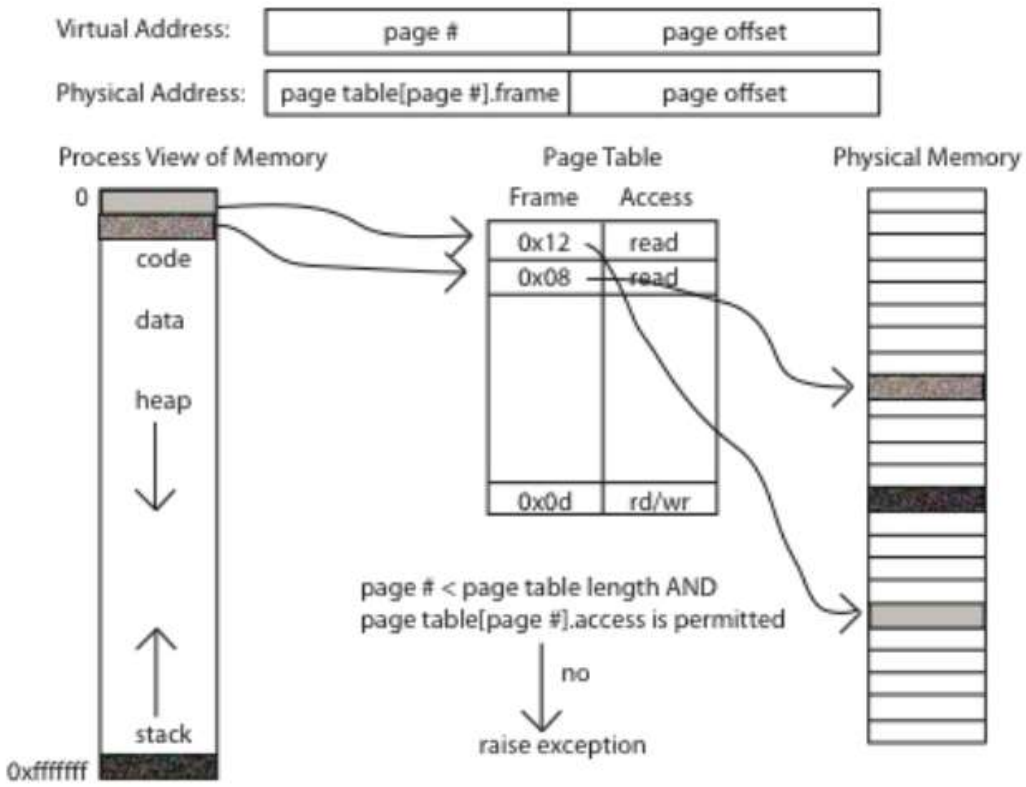
Il risultato è che, in realtà, lo spazio di memoria virtuale dei processi è una gran groviera, ha una serie di buchi e una serie di parti allocate.

Sparse Address Spaces

- Might want many separate segments
  - Per-processor heaps
  - Per-thread stacks
  - Memory-mapped files
  - Dynamically linked libraries
- What if virtual address space is sparse?
  - On 32-bit UNIX, code starts at 0
  - Stack starts at 2^31
  - 4KB pages => 500K page table entries
  - 64-bits => 4 quadrillion page table entries

Quindi in realtà, sebbene non stiamo utilizzando la tecnica della segmentazione ma anche semplicemente con la Paginazione pura, di fatto lo spazio virtual del processore è naturalmente segmentato. Questa questione qui ci pone, in qualche maniera un problema ma è anche un'opportunità. Perché lo spazio virtuale abbiam che è naturalmente segmentato, perché il compilatore distribuisce informazioni segmentate in questo spazio virtuale, di conseguenza il compilatore dovrà tener traccia dello spazio di indirizzamento occupato, il sistema operativo invece si troverà ad avere una tabella delle pagine che, o è in larga misura inutilizzata oppure presenta dei grandi buchi.

Nel caso di questo processo:



La tabella delle pagine è una tabella molto grande, che contiene una riga per ogni pagina ma soltanto alcune di esse sono utilizzate, in questo caso l'ultima e le prime 2.

COLLEGA: Professore quann'è che n'a tabella delle pagine viene mappata, cioè quann'è che una pagina virtuale viene mappata in una pagina fisica, solo quando l'alloco?

(il mitico Stefano): Sì, esattamente. Quando io vado a caricare il programma, genero il processo vado a caricare il programma, il programma lo prendo da un file eseguibile, e nel file eseguibile c'è scritto che occupa un certo numero di byte, il sistema operativo si fa due conti e dice "va bene, per allocare questo programma mi servono 10 pagine. Quindi alloca 10 pagine dello spazio virtuale, che vuol dire che prende 10 descrittori di pagina presumibilmente contigui nello spazio virtuale, li riserva per allocare codice e va in memoria principale a cercare 1 blocchi fisici per poter allocare queste 10 pagine virtuali. Dopo di che dice "Benissimo, c'è da caricare anche questa libreria, che occupa 2 pagine", trova altre 2 pagine virtuali contigue nello spazio virtuale, alloca i 2 descrittori e cerca 2 pagine fisiche per poterle contenere e ci mette la libreria. Come risultato questo programma semplice che ha codice, una libreria, dei dati statici e uno stack si ritrova ad avere: un certo numero di pagine (una 20ina diciamo) allocate nello spazio virtuale in questo modo:

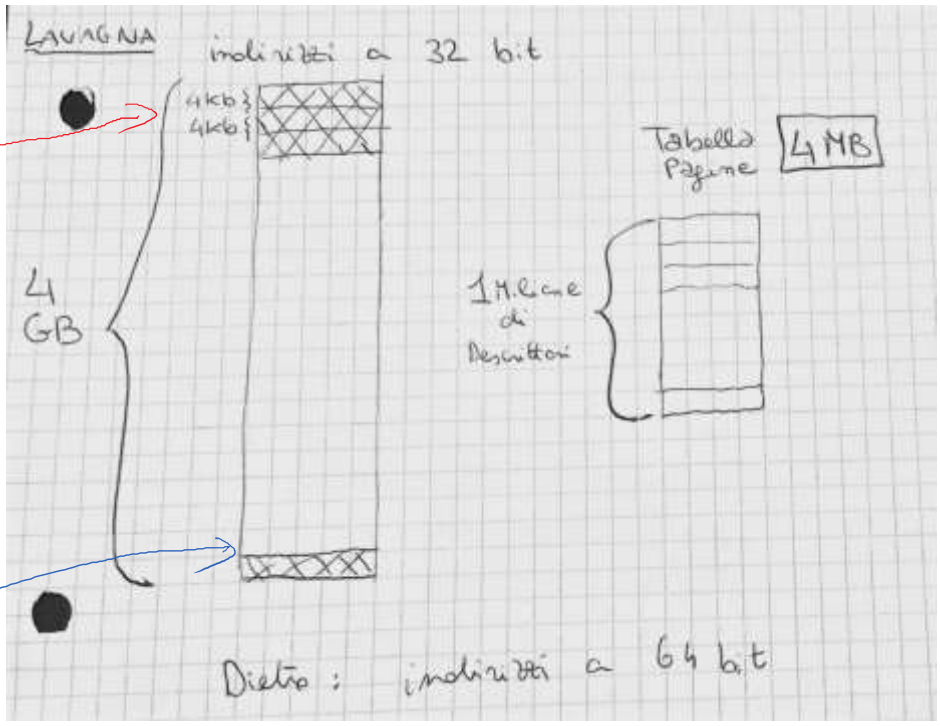
- 10 pagine riservate al codice (magari le prime 10)
- 2 pagine riservate per la libreria (da 11 in giù)
- 1 pagina occupata per i dati statici
- Altre pagine allocate per lo stack

Il risultato è che la tabella delle pagine, non è utilizzata interamente, però la devo allocare per intero, perché il metodo di accesso della tabella delle pagine è quello degli array, prendo l'indice di pagina virtuale e tramite questo indicizzo la tabella, è un array. Quindi tutta la tabella delle pagine deve essere allocata, ma di questa tabella delle pagine, soltanto 20 vettori sono in uso e i restanti 3 milioni e fischia non sono utilizzati.

In memoria principale sono allocate soltanto le pagine effettivamente allocate nello spazio virtuale, questa pagina logica che non è utilizzata nello spazio di memoria fisica non è presente non è mappata, se ad un certo punto a tempo di esecuzione il codice fa una Malloc e il supporto a tempo di esecuzione decide che vuole utilizzare questa pagina virtuale, in realtà, il supporto ad esecuzione può decidere quello che gli pare, ma se poi il sistema operativo non lo fa, son dolori. Quindi, il supporto a tempo di esecuzione farà la malloc allocando una pagina virtuale e di conseguenza il sistema operativo, andrà a cercare una pagina fisica che può contenere quella pagina virtuale e andrà ad inizializzare il descrittore di quella pagina virtuale in maniera tale da legarla alla pagina fisica che ha appena allocato. È evidente che in tutto questo io devo sapere quali sono le pagine nello spazio virtuale allocate e quali no, perché altrimenti rischierei di perdere memoria, e questo è anche uno dei motivi per cui è buona pratica fare la free dopo che avete fatto la malloc. Nel caso di Java si preoccupa il Garbage collector.

Il risultato è che lo spazio virtuale è utilizzato in modo molto sparso. Perché questo è un problema e perché diventa anche un'opportunità?

Facciamo due conti, ma li prenderemo in maniera più precisa. Supponiamo di prendere un sistema a 32 bit, dove lo spazio virtuale è di 4GB, se noi utilizziamo pagine di 4KB vuol dire che lo spazio virtuale si compone di 1GB pagine. Ognuna di queste pagine, per ogni processo (ogni processo ha 1 Milione di pagine di spazio virtuale), ha una tabella delle pagine che contiene 1Milione di righe; ogni riga potrebbero essere 4Byte. Quindi, ogni tabella delle pagine occupa 4MB, il risultato è che se io devo allocare un processo che utilizza memoria per 10 KB, comunque sia in memoria principale devo allocare 4MB, per poter contenere la sua tabella delle pagine. Tenendo presente che la maggior parte di queste righe sono inutilizzate, perché quel processo utilizza soltanto 10KB.



Qual è in realtà il problema, per il quale incide il fatto che lo spazio sia segmentato? La questione è questa: se il compilatore, giustamente, va ad allocare memoria in punti arbitrari e lo stack me lo mette più distante possibile dal codice perché in questo modo lo stack può crescere molto, è chiaro che in questo modo il mio spazio virtuale è impegnato sia questi indirizzi sia su questi altri, con un grosso vuoto nel mezzo; la tabella delle pagine non può rappresentare comodamente questa struttura, perché è una tabella rigida.

Siccome lo spazio di memoria virtuale viene, giustamente, gestito in modo frammentato per tutta una serie di motivi, dal compilatore, dal supporto a tempo di esecuzione del linguaggio, il risultato è che la tabella delle pagine io non posso compattarla. Se io sapessi che la memoria è tutta allocata all'inizio potrei dire "benissimo, la tabella delle pagine la alloco soltanto fino ad un certo punto", ma il problema è che siccome lo spazio virtuale è fatto in questa maniera, devo allocare tutto. Ho un certo spreco, questo spreco si moltiplica per il numero di processi (più sono piccoli i processi maggiore è lo spreco).

Con indirizzi a 64Bit lo spazio di indirizzamento diventa (momento di panico per Chessa),4 milioni di miliardi di byte (4'000'000'000'000'000), la dimensione della singola pagina resta sempre quella, perché anche se gli indirizzi sono a 64bit non mi cambia nulla, una dimensione della pagina più grande aumenta la frammentazione interna. Il risultato è che se io vado a dividere 4 milioni di miliardi per 4K, vengono 4000 T descrittori di 4Byte quindi ho circa 16000 TB di tabella delle pagine per il singolo processo. Non ce lo possiamo permettere.

Si utilizzano le tabelle delle pagine multilivello.

Multi-level Translation

- Tree of translation tables
  - Paged segmentation
  - Multi-level page tables
  - Multi-level paged segmentation
- All 3: Fixed size page as lowest level unit
  - Efficient memory allocation
  - Efficient disk transfers
  - Easier to build translation lookaside buffers
  - Efficient reverse lookup (from physical -> virtual)
  - Page granularity for protection/sharing

Con le tabelle di pagina multilivello, in realtà, la tabella delle pagine diventa una struttura più complessa, una struttura ad albero. In effetti con la traduzione delle pagine multilivello, ci sono diverse tecniche, si può usare la paginazione segmentata, e questa è la tecnica utilizzata nelle architetture x86. Si possono usare delle tabelle delle pagine multilivello oppure si possono utilizzare tabelle delle pagine combinate alla segmentazione. In ogni caso, tutte queste tecniche come meccanismo di base hanno la Paginazione.

Quindi alla base di tutto c'è la paginazione, la singola unità di allocazione in memoria principale è la pagina, lo spazio virtuale è sempre paginato. La dimensione delle pagine resta la stessa, d'altra parte utilizzando queste strutture più complesse, garantisco una gestione della memoria più efficiente (si evita per esempio lo spreco dalla porzione della tabella delle pagine non utilizzata). Soprattutto si mantiene il livello di granularità delle pagine per quanto riguarda la protezione e la condivisione. In pratica si combinano due mondi, i vantaggi della paginazione con i vantaggi della segmentazione.

Paged Segmentation

- Process memory is segmented
- Segment table entry:
  - Pointer to page table
  - Page table length (# of pages in segment)
  - Access permissions
- Page table entry:
  - Page frame
  - Access permissions
- Share/protection at either page or segment-level

Vediamo prima di tutto la segmentazione paginata. Nella segmentazione paginata, lo spazio del processo è segmentato, quindi il processo vede i segmenti; quando voi allocate o il compilatore quando compila, in realtà dispone il codice, lo stack, i dati in segmenti separati. Quindi, abbiamo una tabella dei segmenti che serve per fare la traduzione degli indirizzi, stavolta però il singolo descrittore del segmento non contiene più BASE/LIMITE ma contiene il puntatore alla tabella delle pagine per quel segmento. Immaginatevi cosa succede:

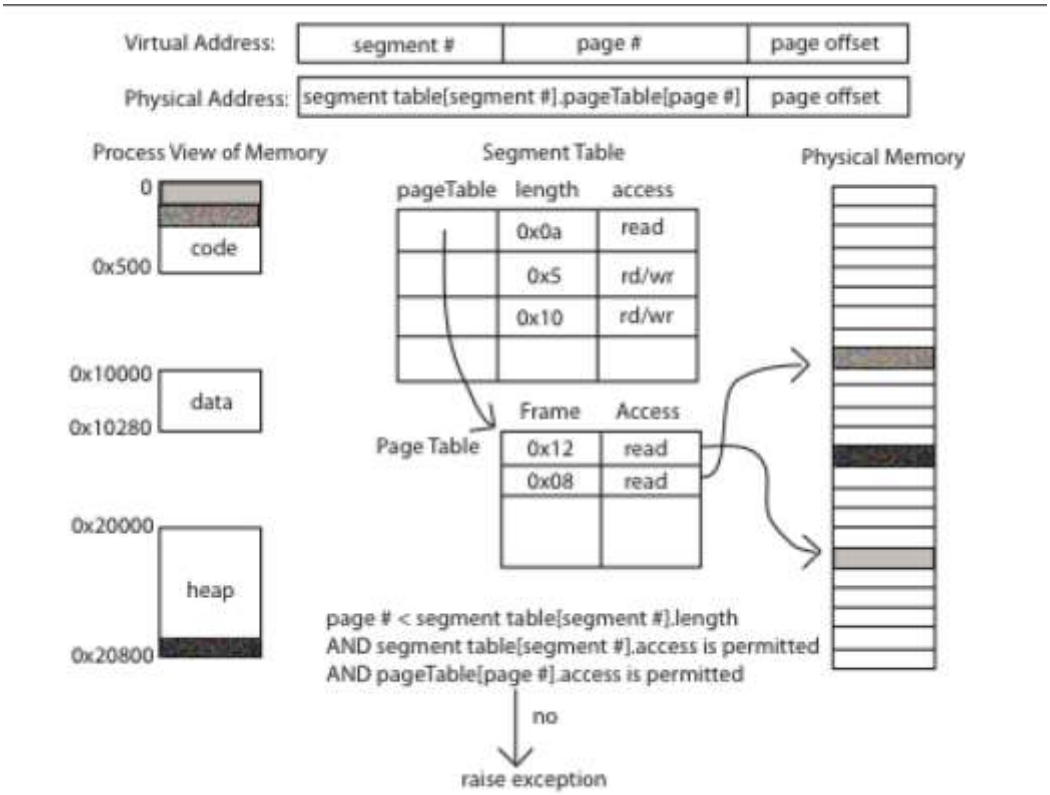
- Genero un indirizzo
- L'indirizzo nella prima parte specifica un indice di segmento
- Da questo indice di segmento, dalla tabella dei segmenti estraggo il descrittore del segmento e quel descrittore del segmento contiene la tabella delle pagine associata al segmento.

// non si capisce una mazza ma ho scritto quello che ha detto. (min. 55:19)

Quindi il resto dell'indirizzo me lo posso elaborare sulla base della tabella delle pagine (per quello specifico segmento). Quindi io ho una tabella delle pagine per ogni segmento di ogni processo, quindi, aumento a dismisura il numero delle tabelle delle pagine ma, il trucco qual è? Il trucco è che i segmenti più piccoli dello spazio virtuale della paginazione, quindi le tabelle delle pagine sono più piccole; non solo, ma i segmenti sono allocati da un indirizzo 0 fino ad un certo indirizzo massimo e sono utilizzati in toto perché i segmenti vengono allocati sulla base di una qualche semantica; allocate il segmento per il codice e sapete quanto è grande, allocate una struttura dati e sapete quanto è grande, allocate lo stack e non lo sapete ma ne allocate una certa dimensione e poi lo fate crescere.

Il risultato è che la tabella delle pagine di questi segmenti, stavolta, non deve gestire uno spazio virtuale sparso, ma deve gestire uno spazio virtuale compatto e quindi alloco la tabella delle pagine per quello che effettivamente mi serve per rappresentare quel segmento. In questo modo posso avere sia i permessi di accesso per il singolo segmento e poi permessi d'accesso anche per la singola pagina, e quindi posso condividere o gestire la protezione sia con la granularità della pagina, sia con la granularità del segmento.

Che cosa succede?



Questo è lo spazio di memoria di un processo segmentato, in questo esempio ci sono 3 segmenti, di conseguenza abbiamo 3 descrittori di segmento che contengono:

- La lunghezza del segmento
- I diritti di accesso, di protezione
- Tutti gli "accessori" per far funzionare la segmentazione
- Il puntatore alla tabella delle pagine

Quindi, in questo caso, il primo segmento che va dall'indirizzo 0 all'indirizzo 500 che in questo momento è stato allocato in più pagine, ma ha allocato materialmente soltanto le prime due pagine, corrisponde una tabella delle pagine che descrive le pagine allocate all'interno di quel segmento che sono la 0 e la 1, con diritti che possono essere differenti. In linea di principio come si fa la traduzione dell'indirizzo? Il processore genera un indirizzo, che vedete scritto lassù in alto. L'indirizzo considerato è un indirizzo virtuale che viene, suddiviso logicamente in 3 campi: indice di segmento, indice di pagina, Offset. Dall'indice di segmento si va a puntare alla tabella dei segmenti, si estrae l'indice della tabella delle pagine, a questo punto si va nella tabella delle pagine la si indicizza con l'indice di pagina, si estrae l'indice del blocco fisico, lo si giustappone all'offset e si ottiene l'indirizzo fisico.

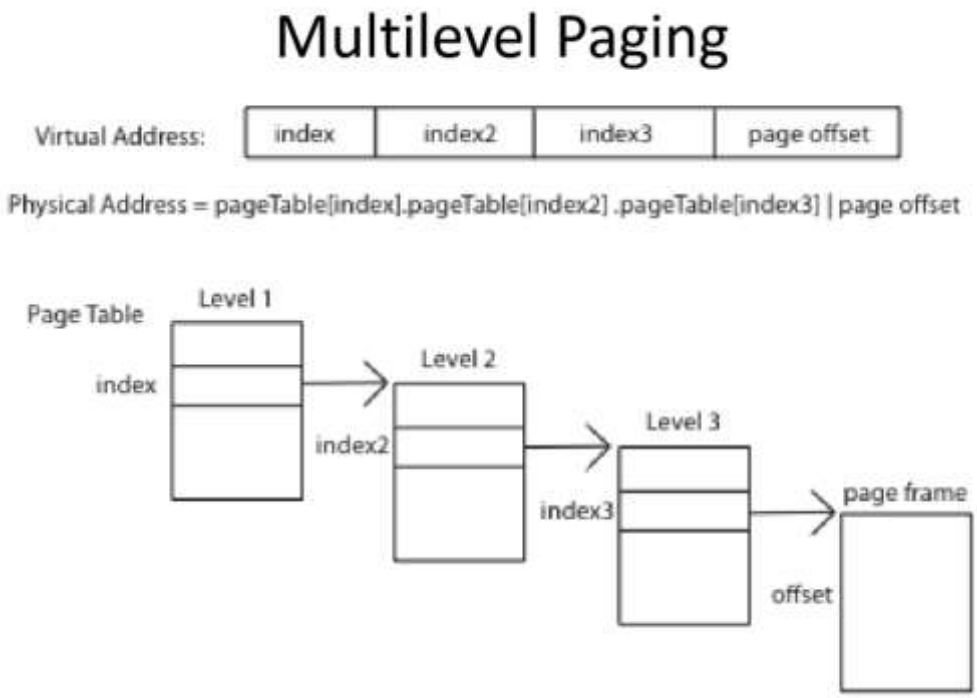
In teoria per questo servono due accessi in memoria:

- Primo accesso: tabella dei segmenti
- Secondo accesso: tabella delle pagine

Quindi, per ogni Byte che vogliamo leggere dalla memoria principale, dobbiamo leggerne 3, ovviamente questo non è la realtà, perché sebbene la tabella dei segmenti e la tabella delle pagine stiano in memoria principale, la traduzione avviene fatta facendo delle CACHE di queste informazioni.

Questa era la tecnica della segmentazione paginata, che unisce i vantaggi di segmentazione e paginazione; ogni segmento ha una sua semantica in realtà, imposta dal compilatore o dal supporto a tempo di esecuzione quando si va ad allocare i segmenti.

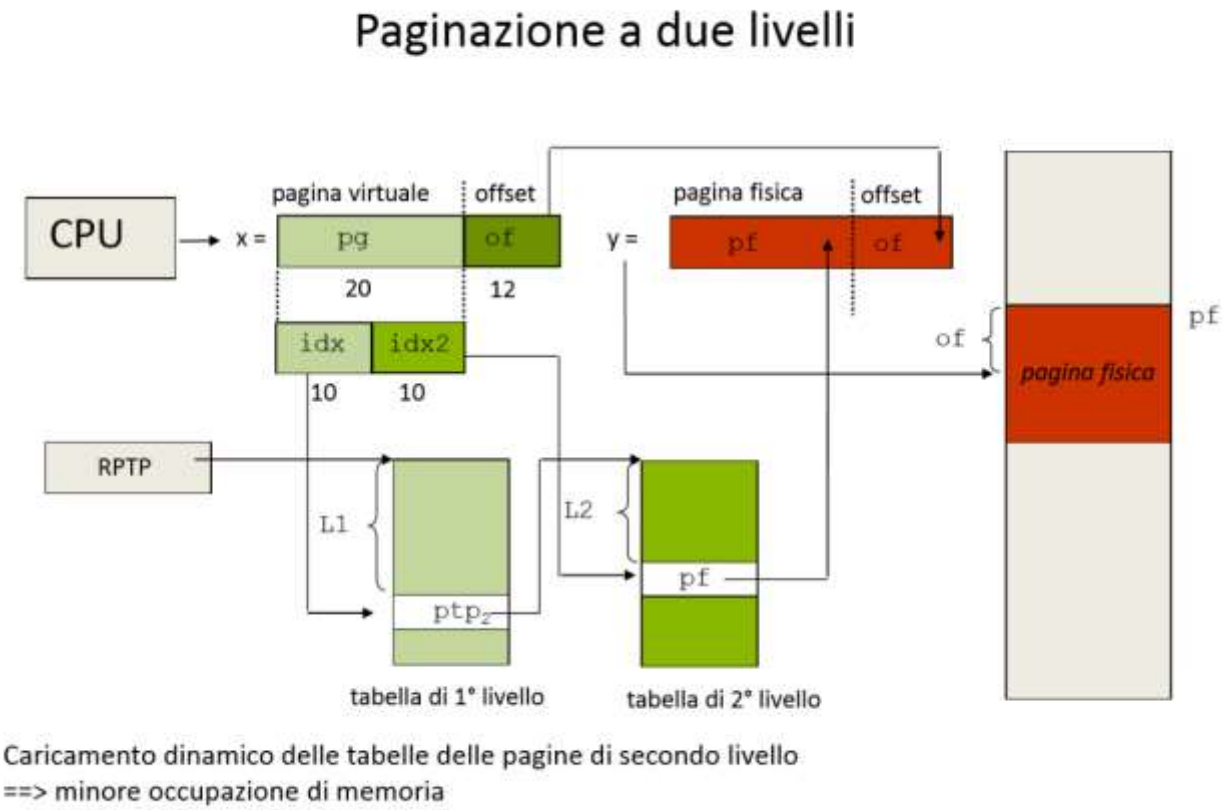
L'altra tecnica è la paginazione multilivello:



Per certi versi può sembrare simile alla segmentazione, però c'è una differenza fondamentale: non c'è più semantica associata ai singoli segmenti. Nella paginazione multilivello lo spazio virtuale è sempre un unico grande spazio continuo, da 0 fino all'indirizzo massimo, questo spazio virtuale viene gestisto dal supporto a tempo di esecuzione del linguaggio come pare a lui, quindi allocando informazioni dove gli pare, e quindi sarà naturalmente segmentato ma non perché il sistema fa segmentazione, ma perché è nella logica di funzionamento del linguaggio. Indipendentemente da questo, il sistema operativo adotta la tecnica delle pagine multilivello, quindi che cosa fa? Quando il processore genera un indirizzo virtuale, questo indirizzo virtuale viene visto come diviso in più campi (per esempio in questo caso viene diviso in 4 campi), i primi 3 campi sono gli indici di pagina (indice di primo livello, di secondo livello ecc…) e l'ultimo è l'offset.

Come funziona la traduzione? Dal primo campo INDEX, indicizzo la tabella delle pagine di primo livello, all'interno dalla quale estraggo un puntatore alla tabella delle pagine di secondo livello che indicizzo tramite il secondo campo dell'indirizzo virtuale, in questo modo dalla tabella delle pagine di secondo livello estraggo un puntatore alla tabella delle pagine di terzo livello che indicizzo con il terzo campo dell'indirizzo virtuale, e da questa estraggo il descrittore di pagina che contiene l'indice del blocco fisico. A questo punto lo compogno con l'offset e ottengo l'indirizzo fisico.

Vediamo un esempio un po' più concreto: La paginazione a due livelli.



Caricamento dinamico delle tabelle delle pagine di secondo livello
==> minore occupazione di memoria

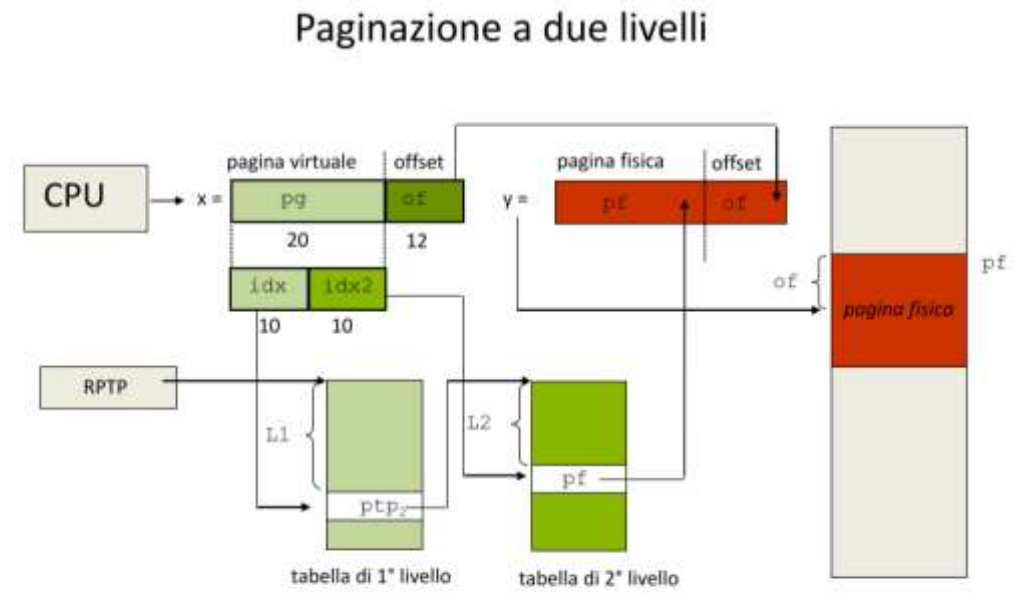
Allora, il processore genera l'indirizzo virtuale, l'indirizzo virtuale con la paginazione a due livelli è diviso in prima battuta in 2 campi (indice di pagina virtuale e offset), a sua volta l'indice di pagina virtuale è diviso in 2 campi perché dobbiamo indicizzare due tabelle. Quindi se io ho un indirizzo a 32 bit, i primi 20 sono l'indice di pagina, i 12 sono l'offset, dopo di che dei 20 dell'indice di pagina, 10 sono l'indice di pagina di primo livello e i secondi 10 sono l'indice di pagina di secondo livello. Fatto questo, ho il puntatore alla tabella delle pagine di primo livello, la indicizzo con il primo campo ed estraggo il puntatore alla tabella

delle pagine di secondo livello che indicizzo con il secondo campo, estraggo il descrittore di pagina fisica e compongo con l'offset e ottengo l'indirizzo fisico.

Il vantaggio del multilivello è che se ho un indirizzo invalido in una tabella di primo livello non ha senso che io allochi una tabella di secondo livello riferita a quell'indirizzo. Quindi risparmio memoria se il processo è piccolo, cosa che non accedeva nel caso della paginazione/ segmentazione classica. Nel caso peggiore(quando il processo occupa tutto il suo spazio virtuale) però questo sistema occuperà più memoria perché deve allocare tutto lo spazio del processo, sommato alle varie tabelle di ogni livello.

Sistemi operativi 08/05/17

Stavamo vedendo la paginazione multithread in particolare abbiamo visto un meccanismo di traduzione delle pagine nel caso di paginazione a due livelli.



Caricamento dinamico delle tabelle delle pagine di secondo livello
==> minore occupazione di memoria

Quindi vi ricordo come funziona la cosa: anziché utilizzare l'indice di pagina virtuale come unico indice per indicizzare un'unica grande tabella, questa tabella in realtà viene strutturata in un albero con una radice e un solo livello sotto la radice per cui l'indice di pagina virtuale si compone di due campi: il primo campo serve per indicizzare la radice (quindi la tabella di primo livello), dalla tabella di primo livello sappiamo qual è la tabella delle pagine di secondo livello che dobbiamo utilizzare perché ne estraiamo l'indirizzo e a questo punto la tabella delle pagine di secondo livello è indicizzata con la seconda parte dell'indirizzo. Da lì e dalla tabella di pagine di secondo livello estraiamo l'indirizzo fisico che componiamo con l'offset e otteniamo l'indirizzo fisico per indirizzare la memoria.

Confronto tra tabelle delle pagine a 1 o 2 livelli

IPOTESI:

- Indirizzi logici di 32 bit; pagine logiche e fisiche di 4 kByte.
  ==> lunghezza del campo offset : 12 bit; indice di pagina codificato con 20 bit
- Descrittori di pagina (elementi della tabella delle pagine) codificati con 4 byte, di cui:
  - 3 byte (24 bit) per la codifica dell'indice di blocco;
  - 1 byte riservato agli indicatori

TABELLA DELLE PAGINE A 1 LIVELLO:

- Numero di elementi della tabella delle pagine: 2^{20}
- Spazio occupato dalla tabella delle pagine: $2^{20} * 4 = 2^{22}$ byte= 4 Mbyte
- Massima dimensione della memoria fisica: 2^{24} blocchi
  ==> $2^{24} * 2^{12} = 2^{36}$ byte = 64 Gbyte

Avevamo fatto a mano alla lavagna un paio di conti per vedere qual è il vantaggio che si ha quando si utilizza la paginazione a multilivello. In particolare un confronto tra le tabelle della pagine a singolo livello e a doppio livello ed avevamo preso questo esempio: avevamo considerato un sistema con indirizzi a 32 bit e pagine logiche e quindi di conseguenza le pagine fisiche di 4k, il fatto che le pagine siano di 4k significa che l'offset deve essere di 16 bit perché 4kbyte per indicizzare correttamente (quindi 12 bit di indirizzo) questo significa che l'indice di pagina sono i restanti 20 bit dell'indirizzo logico.

Supponiamo anche che i descrittori di pagina siano a 4byte di cui 3byte codificano l'indice di blocco (il prof ci fa notare che se utilizziamo 3 byte per codificare l'indice di blocco, questo significa che l'indice di blocco è a 24 bit. Se noi combiniamo questi 24 bit con 12 di offset, possiamo generare indirizzi a 36 bit, quindi possiamo indirizzare una memoria di 2^36 byte che è molto più grande di quella indirizzabile direttamente dal processore virtualmente.) e 1 byte per codificare gli indicatori. Gli indicatori sono dei bit di protezione. Se uso le tabelle delle pagine ad un solo livello quello che succede è che la tabella delle pagine deve avere un descrittore per ogni pagina quindi una riga per ogni pagina. Abbiamo 2^20 pagine (perché abbiamo 20 bit di indice di pagina) quindi vuol dire che la tabella ha 2^20 elementi quindi quasi 4 milioni di descrittori. Siccome ogni descrittore occupa 4byte lo spazio occupato dalla tabella delle pagine è 2^20 * 4 = 4Mbyte. La massima dimensione della memoria fisica indirizzabile da cosa è data? È data dall'indirizzo fisico che si ottiene componendo l'indice di blocco estratto dalla tabella delle pagine con l'offset. Quindi in totale sono 24 + 12 bit quindi sono 36 bit quindi la memoria la memoria fisica è indirizzabile su 2^36 byte quindi 64 giga. Questo a fronte di un indirizzo virtuale uscito dal processore che è di 32 bit quindi con una memoria virtuale di 4 gb. Questa è la massima che potrei gestire con questo tipo di sistema, con questa configurazione. Se nelle stesse ipotesi utilizzo una tabella delle pagine a due livelli cosa cambia?

Confronto tra tabelle delle pagine a 1 o 2 livelli

IPOTESI (come nel caso precedente)

- Indirizzi logici di 32 bit; pagine logiche e fisiche di 4 kByte.
  ==> lunghezza del campo offset : 12 bit; indice di pagina codificato con 20 bit
- Elementi di ogni tabella delle pagine (di primo o secondo livello) codificati con 4 byte, di cui:
  - 3 byte (24 bit) per individuare un indice di blocco;
  - 1 byte riservato agli indicatori

TABELLA DELLE PAGINE A 2 LIVELLI:

Ipotesi: 2^{10} tabelle delle pagine di secondo livello;

==> La tabella di primo livello ha 2^{10} elementi

==> ripartizione dell'indirizzo logico:

- 12 bit per offset;
- 10 bit per indirizzare la tabella di primo livello
- 10 bit per indirizzare la tabella di secondo livello selezionata;

Bhe devo avere ulteriori informazioni. Devo sapere l'indice di pagina come viene ripartito sui due livelli quindi devo sapere come è strutturato l'albero che mappa la tabella delle pagine. Nell'esempio che abbiamo fatto, abbiamo ipotizzato che la tabella delle pagine di secondo livello avesse 2^10 elementi. Che vuol dire? Vuol dire che il secondo indice quello delle tabelle delle pagine di secondo livello è formato da 10 bit dell'indirizzo quindi l'indice di pagina che è di 20 bit si decompone in due parti: un indice di primo livello di 10 bit e un indice di secondo livello di 10 bit. Questo fa si che ogni tabella delle pagine di primo e di secondo livello abbia 2^10 elementi. Quindi qui vediamo come è ripartito l'indirizzo logico.

Confronto tra tabelle delle pagine a 1 o 2 livelli

TABELLA DELLE PAGINE A 2 LIVELLI:

==> ogni elemento di tabella di primo livello corrisponde a una tabella di secondo livello
- 3 byte: indice di blocco nel quale risiede la tabella di secondo livello (se presente)
- 1 byte: indicatori (tra cui indicatore di presenza).

==> ogni elemento di tabella di secondo livello corrisponde a una pagina
- 3 byte: indice di blocco nel quale risiede la pagina (se presente)
- 1 byte: indicatori (tra cui indicatore di presenza).

- **lunghezza di ogni tabella di primo o secondo livello: 2^{10} elementi ==> 2^{10}*4 = 4 Kbyte**
- **massima dimensione della memoria fisica : 2^{24} blocchi ==> 2^{24} * 2^{12} = 2^{36} byte = 64 Gbyte.**

Ogni tabella di primo livello corrisponde a una tabella di secondo livello. La tabella di secondo livello è quella che contiene il descrittore della pagina quindi che deve contenere l'indice del blocco fisico sul quale la pagina virtuale è allocata. Quindi questo elemento deve avere lo stesso contenuto del descrittore di pagina nel caso di tabelle a singolo livello. Mettiamoci nelle stesse condizioni quindi supponiamo che ogni tabella di secondo livello contenga 3 byte per indicizzare il blocco fisico nel quale risiede la pagina virtuale e un byte di indicatore. Quindi la tabella delle pagine di secondo livello contiene 2^10 elementi * 4 byte ognuno e supponiamo che questo valga anche per la tabella di primo livello. La tabella di primo livello cosa deve contenere? Deve contenere un puntatore in memoria a una tabella di secondo livello quindi deve contenere un indirizzo. Ora in questo caso ogni tabella di secondo livello sta esattamente in una pagina quindi a noi basta memorizzare in una tabella di primo livello l'indice di blocco fisico nel quale la tabella di secondo livello corrispondente è stata memorizzata quindi ci bastano sempre 3 byte sostanzialmente. Un byte di indicatori per sapere se quella tabella delle pagine di secondo livello è allocata o meno ed eventuali altre informazioni.

Quindi il risultato è che una tabella di primo o di secondo livello contiene sempre 2^10 elementi ognuno di 4byte e quindi occupa 4Kbyte. La massima dimensione della memoria fisica resta uguale perché l'indirizzo fisico si compone sempre (passa l'aereo) di 2^24 * 2^12 dove 12 rappresenta l'offset e quindi sono sempre 64 giga. Ai fini della memoria fisica, della memoria virtuale del processo non cambia niente che si utilizzino tabelle a livello singolo o tabelle a multilivello, per il programmatore non cambia niente vede la paginazione sempre esattamente nella stessa maniera, ciò che cambia è all'interno del sistema operativo che riesce a gestire meglio la sua memoria quindi deve riservare meno spazio per allocare le tabelle delle pagine o meglio deve riservare uno spazio di memoria proporzionale alla dimensione del processo per allocare le tabelle delle pagine.

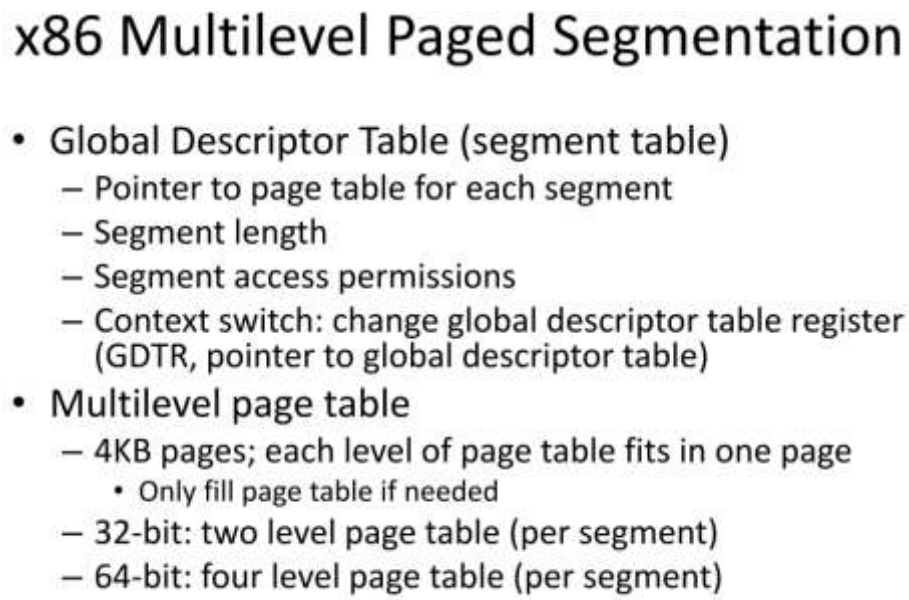
Domanda: potrebbe rispiegare del perché 2^36?

Il discorso è questo: se io vi chiedo quanto è la dimensione massima della memoria fisica, la memoria fisica è limitata dalla dimensione dell'indirizzo con il quale la indicizzate. Se avete 10 bit di indirizzo fisico non potete avere una memoria più grande di 2^10 byte. Quindi per sapere qual è la massima dimensione della memoria fisica, dovete sapere quanto è grande l'indirizzo fisico che spedite alla memoria. Quanto è grande questo indirizzo fisico? Questo indirizzo fisico è ottenuto componendo l'offset con l'indice di pagina fisica che è contenuto all'interno della tabella delle pagine di secondo livello. Quindi per sapere quanto è l'indirizzo fisico dovete sapere quanto è grande l'offset e quanto è grande l'indice di pagina fisica contenuto

qui dentro. Nell'esempio visto l'offset e di 12 bit e poi vi ho detto che la tabella delle pagine di secondo livello riserva 3 byte per codificare l'indice di pagina fisica quindi questo campo è formato da 24 bit quindi l'indirizzo fisico è dato dalla composizione di questi 24 bit con i 12 di offset in totale 36 bit che formano l'indirizzo fisico.

Domanda: ma non avevamo detto che i nostri indirizzi erano a 32 bit?

Virtuali. Quelli generati dal processore, infatti il processore genera indirizzi virtuali a 32 bit però poi alla memoria fisica dovete spedire un indirizzo della memoria fisica. Il processore indirizza la memoria virtuale del processo, questa memoria è virtuale quindi non esiste in realtà, viene mappata nella memoria fisica in punti arbitrari e per fare questo mappaggio l'indirizzo virtuale viene trasformato in un indirizzo fisico per la memoria fisica in modo tale che quell'indirizzo fisico generato contiene quel byte della memoria virtuale di quel processo che ci interessa. Quindi il processore genera indirizzi virtuali di 32 bit che fanno riferimento ad uno spazio di 4 giga. Questi indirizzi virtuali (in questo esempio) vengono tradotti in indirizzi di 36 bit questo vuol dire che la memoria fisica al massimo può avere 2^36 byte quindi 64 giga. Se io avessi detto che invece che riservare 3 byte per l'indice di pagina fisica ne avessi riservati soltanto due, avrei avuto un indirizzo fisico a 38 bit e quindi una memoria fisica di 256 mega. Quindi dipende dall'architettura e dal particolare esempio.



Alcune architetture per esempio l'architettura dell'x86 e tutta la famiglia dei processori Intel utilizza, anche per motivi storici, la segmentazione paginata con tabelle delle pagine multilivello. Utilizza la segmentazione paginata per motivi storici perché già nelle prime versioni questi processori adottavano i metodi della segmentazione. Di conseguenza in questi sistemi dovete gestire: la tabella dei segmenti tramite questo global descriptor table che è appunto la tabella dei segmenti e all'interno del processore c'è il puntatore alla tabella dei segmenti di ciascun processo. La tabella dei segmenti contiene un po di informazioni in particolare il puntatore alla tabella delle pagine per ogni segmento. Questa tabella delle pagine non è una tabella delle pagine a livello singolo ma è una tabella delle pagine multilivello. In particolare nelle architetture a 32 bit di questa famiglia di questi processori, la tabella delle pagine ha due livelli per segmento oppure nelle architetture a 64 bit abbiamo una tabella delle pagine a 4 livelli. Il meccanismo di traduzione degli indirizzi è lo stesso portato anche su più livelli quindi dalla tabella delle pagine di primo livello si trova una tabella delle pagine di secondo livello che viene indicizzata verso una tabella delle pagine

di terzo livello e da quella si va a quella al quarto livello e infine si trova l'indirizzo fisico.

Multilevel Translation

- Pros:
  - Allocate/fill only as many page tables as used
  - Simple memory allocation
  - Share at segment or page level
- Cons:
  - Two or more lookups per memory reference

Quali i sono i vantaggi della allocazione multilivello rispetto quella a singolo livello? Il vantaggio più grosso è che possiamo allocare le tabelle delle pagine soltanto per la parte che effettivamente ci interessa allocare quindi risparmiamo memoria e tempo per pagine che non ci servono. Utilizzando il multilivello con segmentazione più paginazione abbiamo la possibilità di andare a condividere interi segmenti o singole pagine a seconda dei casi. Tutto questo agevola alcune funzioni del sistema operativo per esempio il meccanismo della fork con la copy on write è molto agevolato nell'avere la segmentazione con le pagine multilivello perché se ad un certo punto un processo figlio rispetto al padre deve andare a differenziare andando a scrivere la sua memoria andrà a ricopiare soltanto le pagine di dati che il processo va a sovrascrivere/modificare. In questo modo la fork diventa molto più efficiente. Qual è invece lo svantaggio di usare la traduzione multilivello? Per fare una traduzione da indirizzo virtuale a fisico devo avere tanti accessi alla tabella delle pagine tanti quanto sono i livelli. Mi si complica quindi questo meccanismo di traduzione degli indirizzi. Tenete presente che il meccanismo di traduzione degli indirizzi non può assolutamente essere fatto ad hardware andando a caricare la tabella delle pagine nella sua interezza perché semplicemente non ci sta nell'MMU (dispositivo che traduce gli indirizzi). In effetti la traduzione degli indirizzi viene fatta tramite una cache che sta dentro l'MMU quindi non carichiamo tutta la tabella delle pagine ma carichiamo soltanto la cache. Soltanto nel caso in cui l'MMU non sia in grado di tradurre l'indirizzo perché non trova nella cache l'indirizzo virtuale da poter tradurre, allora in questo caso bisogna andare in memoria principale ad esplorare la tabella delle pagine per poter fare la traduzione. Ed è questo il caso in cui si paga di più però se la cache è gestita correttamente questa operazione di traduzione fatta in memoria principale la si fa raramente quindi non incide molto sui costi.

Portability

- Many operating systems keep their own memory translation data structures
  - List of memory objects (segments)
  - Virtual -> physical
  - Physical -> virtual
  - Simplifies porting from x86 to ARM, 32 bit to 64 bit
- Inverted page table
  - Hash from virtual page -> physical page
  - Space proportional to # of physical pages

Con questi meccanismi di traduzione degli indirizzi siamo molto legati all'hardware, al processore. In effetti la scelta del meccanismo di traduzione degli indirizzi e quindi della rappresentazione dello spazio virtuale del processo e della gestione dello spazio fisico è dettata dall'hardware, dal processore. Questo vuol dire che se scrivete un sistema operativo per un certo processore poi avete grosse difficoltà a portare quel sistema operativo su un altro processore. Se prendete Windows, se prendete Linux, se prendete iOS questi sistemi operativi sono portabile su una grande classe di processori, ve li potete trovare su processori tipici da smartphone, veli potete trovare su processori desktop etc… Ognuno di questi processori ha la propria architettura, fa le proprie scelte per rappresentare la memoria virtuale e questo vuol dire che ogni volta che vogliamo portare il nostro sistema operativo su un'altra piattaforma, su un altro processore, dobbiamo cambiare il sistema operativo, dobbiamo cambiarne il funzionamento interno. Questo non è chiaramente accettabile, l'investimento per sviluppare un sistema operativo è enorme e non vogliamo ributtare tutto all'aria e rifare tutto da capo ogni qual volta dobbiamo adattarci ad un nuovo processore.

Per questo motivo i sistemi operativi sono realizzati un po a buccia di cipolla, a strati e in particolare per non dipendere dal meccanismo di gestione della memoria imposto dal processore e quindi per non dipendere dalla tabella delle pagine scelta dal processore con quel particolare formato, il sistema operativo utilizza una propria tabella delle pagine in qualche modo universale. Quindi il sistema operativo rappresenta lo spazio virtuale dei processi utilizzando una propria rappresentazione interna per la tabella delle pagine, una propria rappresentazione interna per l'allocazione della memoria fisica e quello che si fa quando si porta questo sistema operativo da un'architettura ad un'altra è soltanto modificare i meccanismi di passaggio, di traduzione dalla tabella del sistema operativo alla tabella imposta dal processore.

Perché i sistemi operativi fanno questo? Certamente lo fanno per portabilità e poi lo fanno perché i sistemi operativi devono svolgere tutta una serie di compiti di gestione che vanno al di là della semplice traduzione degli indirizzi. Dal punto di vista del processore è importante tradurre gli indirizzi ma il sistema operativo deve invece caricare i processi in memoria, allocare memoria fisica, stabilire dove allocare un processo, agevolare la condivisione di segmenti, porzione di codice, porzione di dati. Quindi il sistema operativo deve mantenere molte più informazioni di quelle che sono strettamente necessarie al processore per fare la traduzione degli indirizzi. Quindi in ogni caso al sistema operativo servono strutture dati aggiuntive.

Che strutture dati utilizzano i sistemi operativi? Ne utilizzano diverse, in particolare utilizzano spesso e volentieri tabelle delle pagine inverse. Perché tabelle delle pagine inverse? Perché spesso si basano su associazioni inverse tra blocco fisico e pagina virtuale. La tabella delle pagine, così com'è usata dal processore, serve per tradurre un indice di pagina virtuale in un indice di pagina fisica quindi in qualche modo il suo utilizzo è unidirezionale e in effetti la tabella delle pagine a livello singolo oppure la tabelle delle pagine multilivello, si utilizzano molto bene se voi avete come chiave d'accesso l'indirizzo virtuale. Però questo fa si che la loro memorizzazione sia un po dispendiosa (si sprechi un po di memoria), dovete avere una tabella delle pagine per ogni processo quindi spesso ridondanti e poi al sistema operativo serve spesso anche l'informazione inversa: dato un blocco fisico vorrei sapere a quale processo quel blocco è allocato perché devo fare delle altre operazioni sui blocchi fisici, devo trovare i blocchi fisici liberi, se un processo termina devo liberarli quindi devo sapere di quel processo cosa è allocato e via discorrendo. Per questo motivo si usano spesso tabelle delle pagine inverse e ci sono diverse implementazioni una è per esempio sotto forma di tabella hash per cui si usa un hash da una pagina virtuale verso una pagina fisica e in questo modo posso avere un'unica tabella hash per tutti i processi del sistema che ha una dimensione proporzionale alla memoria fisica non più alla memoria virtuale.

Strutture alternative sono le core map che sono delle tabelle che hanno un descrittore per ogni blocco fisico e quel descrittore mi dice qual è la pagina virtuale allocata su quel blocco fisico e a quale processo/processi appartiene. Tenendo presente che il sistema operativo deve usare queste strutture dati aggiuntive per rappresentare lo stato di allocazione della memoria fisica etc. c'è da porsi questa domanda: c'è davvero bisogno delle tabelle delle pagine multilivello?

Do we need multi-level page tables?

- Use inverted page table in hardware instead of multilevel tree
  - IBM PowerPC
  - Hash virtual page # to inverted page table bucket
  - Location in IPT => physical page frame
- Pros/cons?

Cosa vuol dire questo? Il processore impone di avere delle pagine fatte in una certa maniera. Queste tabelle delle pagine non le può usare direttamente per tradurre gli indirizzi perché nella MMU non ci stanno quindi utilizza in realtà una cache per tradurre gli indirizzi. Di fatto queste tabelle delle pagine devono stare in memoria principale, devono essere mantenute dal sistema operativo per permettere al processore di completare la traduzione nel caso in cui con la cache interna il processore non possa tradurre l'indirizzo virtuale in indirizzo fisico. Anziché utilizzare queste tabelle non potrei utilizzare delle tabelle inverse che sono più comode per il sistema operativo? In effetti alcuni processori, su alcune architetture è stata tentata questa strada quindi c'è ad esempio il PowerPC dell'IBM, un processore dell'inizio anni 2000, che adottava questa strategia. Quindi la tabella delle pagine era in realtà una tabella hash per cui il processore manteneva nell'MMU una cache della tabella delle pagine per tradurre velocemente gli indirizzi ma nel caso in cui non fosse in grado, tramite questa cache, di tradurre l'indirizzo andava ad accedere a questa tabella hash per andare a caricare il descrittore di pagina dentro l'MMU.

Questo tipo di approccio non ha avuto poi molto successo, le architetture attuali non lo utilizzano e il motivo fondamentalmente è che gestire una tabella hash ad hardware è abbastanza complesso. Quindi se vogliamo che sia l'MMU ad andare a scandire la tabella delle pagine nella memoria principale utilizzando l'hashing questo complica parecchio il progetto. Si sono preferite altre strade a favore delle tabelle delle pagine multilivello nel processore.

Efficient Address Translation

- Translation lookaside buffer (TLB)
  - Cache of recent virtual page -> physical page translations
  - If cache hit, use translation
  - If cache miss, walk multi-level page table
- Cost of translation =

  Cost of TLB lookup +

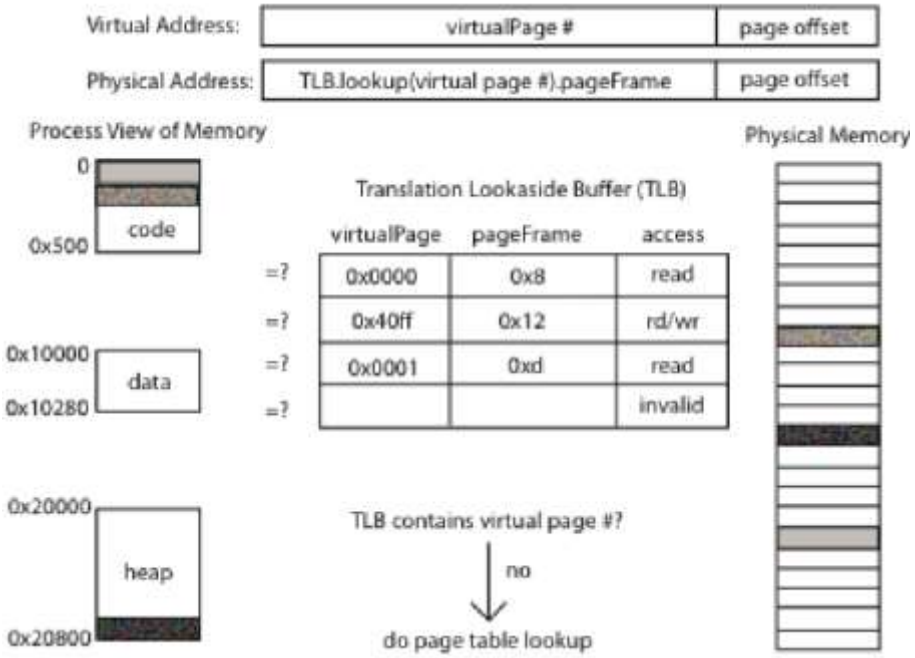
  Prob(TLB miss) * cost of page table lookup

Come si fa in realtà la traduzione degli indirizzi ad hardware? Non si utilizza la tabella delle pagine memorizzata nell'MMU ma si utilizza una cache. Questa cache si chiama TLB (translation lookaside buffer), è una memoria abbastanza piccola che mantiene l'associazione tra alcune pagine virtuali e i blocchi fisici sui quali queste pagine virtuali sono allocate. Nella TLB non dobbiamo conservare quindi tutta la tabella delle

pagine del processo in esecuzione ma conserviamo soltanto i descrittori delle pagine che quel processo sta utilizzando in un certo periodo di tempo, in una certa fase della sua esecuzione.

L'idea è che se il processore genera un indirizzo virtuale e questo indirizzo virtuale ha una corrispondenza all'interno di questa cache (TLB) vuol dire che esaminando TLB trovo l'indirizzo di pagina fisica e posso fare immediatamente la traduzione con il meccanismo che abbiamo visto nella lezione precedente. Questo funziona sia che la tabella delle pagine sia a livello singolo sia che utilizzi una tabella delle pagine multilivello. In effetti questa cache è indipendente dal modello di rappresentazione della tabella delle pagine perché in questa cache sto caricando solo il descrittore di pagina quindi l'associazione tra pagina virtuale e pagina fisica. Che succede se all'interno della TLB non trovo il descrittore della pagina cercata? A questo punto devo andare a cercare il descrittore in memoria principale esaminando la tabella delle pagine. Se la tabella delle pagine è a singolo livello lo faccio in un solo accesso, se la tabella delle pagine è a multilivello lo faccio in più accessi alla memoria e usando esattamente il meccanismo di traduzione visto nelle lezioni scorse.

Quindi è il processore, nel particolare l'MMU, che deve avere la capacità di andare ad esaminare la tabella delle pagine. Qual è il costo della traduzione? Il costo della traduzione è dato dal costo di andare a cercare all'interno della TLB (costo piccolo) sommato al costo di andare ad esaminare la tabella delle pagine nel caso in cui ci sia una TLB miss (cioè nel caso in cui non sia riuscito a fare la traduzione).



Quindi la TLB è strutturata in questa maniera: è una tabella che contiene per ogni riga: un indice di pagina virtuale, un indice corrispondente alla pagina fisica e ovviamente i diritti di protezione. In questo caso abbiamo una segmentazione paginata quindi la memoria virtuale è segmentata e i segmenti a loro volta sono paginati. Nella TLB non ci interessa di come si ha il modello di memoria virtuale del processo, quindi non ci interessa che ci sia paginazione, segmentazione, uno o più livelli. È assolutamente irrilevante perché siccome l'ultimo livello è quello dato dalla paginazione, noi per tradurre l'indirizzo dobbiamo soltanto sapere a un indirizzo virtuale quale pagina fisica corrisponde.

Quindi indipendentemente da quale è il livello di memoria virtuale offerto ai processi la TLB resta soltanto l'associazione tra indice di pagina virtuale e indice di pagina fisica con i rispettivi diritti di accesso. Che cosa succede? Generato un indirizzo virtuale dal processore si estrae l'indice di pagina (che è un indice di pagina virtuale) e ad hardware lo si confronta in maniera parallela con tutti questi campi. Quindi la TLB dispone di un numero di comparatori pari al numero di righe della TLB e organizzata in forma associativa, quindi voi

date una chiave in ingresso e in uscita ottenete il valore associato a quella chiave (questo lo si può fare in maniera molto efficiente). Se questo confronto da esito positivo vuol dire che la TLB contiene una riga associata a quella pagina virtuale e per quella pagina virtuale estrae un indirizzo di pagina fisica quindi (???) e posso controllare anche i diritti di accesso. Se invece questo confronto da esito negativo vuol dire che la pagina virtuale cercata potrebbe essere presente in memoria principale ma non ho possibilità di tradurla attualmente con le informazioni che sono presenti nella TLB. Quindi in questo caso ho una cache miss e per procedere con la traduzione devo prima caricare nella TLB il descrittore della pagina. In questo caso se la TLB non contiene una riga per la pagina virtuale, la MMU va in memoria principale, accede alla tabella delle pagine, va a cercare il descrittore associato a quella pagina virtuale, lo carica all'interno della TLB (in uno spazio vuoto oppure rimuove una riga a seconda dei casi) e a questo punto può fare la traduzione.

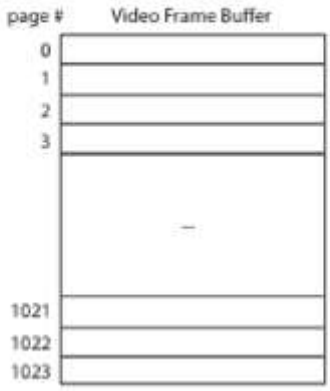
In questo caso ci possiamo porre un'altra domanda: abbiamo bisogno di avere un'MMU così complessa che va ad accedere direttamente alla memoria principale a scandire la tabella delle pagine per caricare la TLB? Potrei fare una cosa differente. Se nella TLB non c'è l'associazione tra pagina virtuale e pagina fisica, potrei sollevare un'eccezione. Metto in esecuzione il sistema operativo che a questo punto capisce qual è la pagina virtuale che ha causato l'eccezione, va lui a scandire la tabella delle pagine e va a caricare il descrittore delle pagine nella TLB. Questa soluzione è stata adottata in alcune architetture per esempio

Software Loaded TLB

- Do we need a page table at all?
  - MIPS processor architecture
  - If translation is in TLB, ok
  - If translation is not in TLB, trap to kernel
  - Kernel computes translation and loads TLB
  - Kernel can use whatever data structures it wants
- Pros/cons?

Nei MIPS con l'obiettivo di avere un processore più semplice di questi meccanismi. Quindi se dalla TLB possiamo tradurre allora tutto bene, se non possiamo tradurre interruzione al kernel, il kernel estrae il descrittore delle pagine dalla tabella delle pagine, carica nella TLB e fa ripartire il processo. Questo tipo di approccio è stato utilizzato ma non è risultato particolarmente efficiente per il problema della trap al kernel, cioè generare l'interruzione verso il kernel e far partire il sistema operativo. Questa operazione porta via centinaia di operazioni che devono essere eseguite dal processore, porta via tempi dell'ordine di centinaia di millisecondi mentre se invece lo faccio fare direttamente al processore (all'MMU) questo lavoro di aggiornamento della TLB i tempi sono estremamente più rapidi. Per questo motivo questo genere di soluzione è stata abbandonata e nelle architetture moderne l'analisi della tabelle delle pagine viene fatta direttamente dall'MMU.

When Do TLBs Work/Not Work?



Ovviamente la TLB è una cache e come tutte le cache in certi casi funziona bene in certi casi funziona male. Funziona bene se gli indirizzi che vengono riferiti hanno una qualche proprietà di località (una qualche logica) di questo genere. Se voi immaginate di avere una memoria video, dove ogni riga del video è codificata all'interno di una pagina, quindi la pagina 0 codifica tutta la prima riga, la pagina 1 codifica tutta la seconda riga… se io vado a leggere questa schermata per righe, la traduzione degli indirizzi è efficiente perché prima genererò sempre indirizzi riferiti alla pagina 0 per scandirla tutta e quindi il primo potrà causare un TLB miss ma poi tutti gli altri riferimenti saranno sempre sulla stessa pagina virtuale e quindi andrà tutto bene, la traduzione verrà fatta rapidamente. Però se io invece vado ad eccedere verticalmente, quindi vado ad accedere ad un bit per pagina, vuol dire che ad ogni istruzione potenzialmente potrei avere dei TLB miss quindi è un uso molto meno efficiente della TLB (questo rappresenta la normalità dell'utilizzo delle cache).

Superpages

- TLB entry can be
  - A page
  - A superpage: a set of contiguous pages
  - x86: superpage is set of pages in one page table
  - x86 TLB entries
    - 4KB
    - 2MB
    - 1GB

Un'altra caratteristica che si usa spesso nei sistemi è quella di utilizzare riferimenti a "super pagine". Cosa vuol dire? All'interno della TLB io dovrei conservare un sistema per la traduzione di ogni singolo indirizzo di pagina virtuale però se il sistema operativo è stato accorto e ha caricato una serie di pagine virtuali consecutive in un insieme di blocchi fisici consecutivi (il sistema operativo questa cosa la può fare tranquillamente se ha la possibilità di farlo), per tradurre gli indirizzi di tutte queste pagine non ho bisogno di avere una riga nella TLB per ogni pagina ma basta avere una riga collettiva per tutto questo insieme di pagine. In questo modo risparmio spazio nella TLB e sono più efficiente. Per esempio in questo modo utilizzando super pagine se io ho l'accortezza di allocare tutto questo buffer (in riferimento al video frame

buffer nella slide precedente) in una serie di blocchi fisici consecutivi, in realtà la traduzione di ognuna di queste pagine può avvenire con una sola riga della TLB. Quindi per questo motivo le architetture in accordo con i sistemi operativi, che gestiscono questa cosa, possono rappresentare delle super pagine che non sono altro che gruppi di pagine allocate in modo consecutivo su blocchi fisici per cui possono essere descritte da un super descrittore.

Questa caratteristica è usata ad esempio per architetture x86 che permettono di avere la TLB entry di 4k (quindi una riga può essere una singola pagina) oppure può essere un blocco di memoria di 2MB (512 pagine) oppure può essere un blocco di 1Gb.

Cosa succede con le TLB quando c'è una commutazione di contesto?

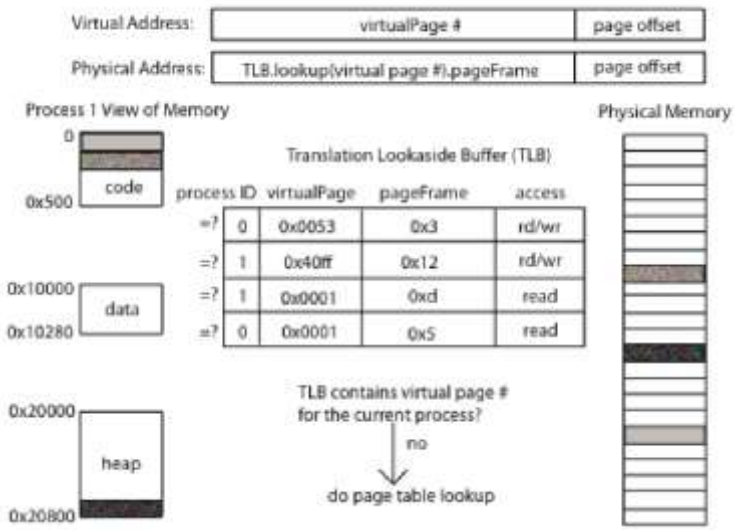
When Do TLBs Work/Not Work, part 2

- What happens on a context switch?
  - Reuse TLB?
  - Discard TLB?

- Motivates hardware tagged TLB
  - Each TLB entry has process ID
  - TLB hit only if process ID matches current process

Potrei riutilizzare il contenuto della TLB. Quando c'è una commutazione di contesto sappiamo che tolgo un thread dall'esecuzione e ne metto un altro. Se l'altro thread appartiene ad un altro processo, siccome gli spazi di indirizzamento sono disgiunti, il contenuto della TLB andrà completamente invalidato perché il nuovo thread si riferirà ad indirizzi che sono completamente differenti. Se però il thread fa parte dello stesso processo, quindi la commutazione di contesto avviene tra due thread dello stesso processo, allora il secondo thread potrebbe utilizzare indirizzi già traducibili con contenuti della TLB attuale. Quindi invalidare la TLB, in questo caso, non è un vantaggio ma mi conviene tenerla perché il nuovo thread messo in esecuzione riferisce in realtà allo stesso spazio di memoria, fa riferimento alla stessa tabella delle pagine.

Il problema però è che se voglio conservare il contenuto della TLB oppure se lo devo scartare perché ho messo in esecuzione un thread di un altro processo, questo nell'MMU lo devo sapere quindi non mi basta avere le informazioni che ho attualmente della TLB ma le devo arricchire con informazioni aggiuntive. Per questo motivo nella TLB aggiungo un altro indicatore che è l'indice del processo attualmente in esecuzione.



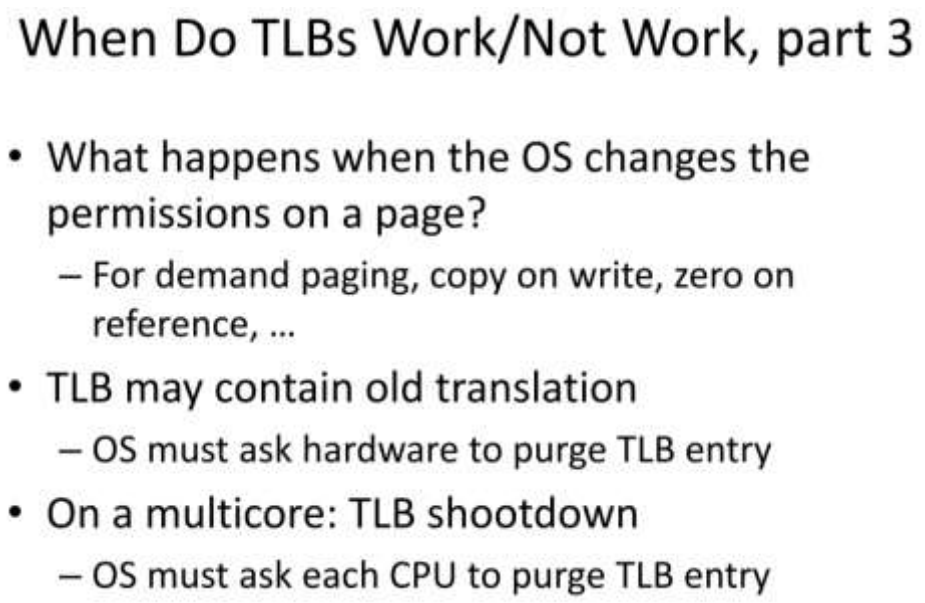
Non è un indice del thread, è un indice del processo perché sebbene nel sistema operativo si mettono in esecuzione i thread e non i processi, tutti i thread dello stesso processo condividono la stessa memoria e quindi la stessa tabella delle pagine. Quindi non mi interessa sapere a quale thread fa riferimento quella riga della TLB ma mi interessa sapere il processo di riferimento. Quindi in questo modo io ho un indicatore che è in realtà è un campo e contiene l'identificatore del processo e questo per ogni riga della TLB. Quando c'è una commutazione di contesto non mi devo preoccupare di aggiornare la TLB, metto in esecuzione il nuovo thread e lascio che lui generi i suoi indirizzi. Ora cosa succede? Il thread genera un indirizzo, l'indice di pagina virtuale viene cercato all'interno della TLB, supponete che il thread attualmente in esecuzione faccia capo al processo di identificatore 1. È chiaro che le uniche righe che devo confrontare con l'indirizzo virtuale con le uniche due righe che hanno come process ID: 1. Le altre due righe fanno riferimento ad un altro processo e non posso utilizzarle per la traduzione dell'indirizzo attuale.

Domanda: Ma quanti bit in più sono? (si riferisce al process ID)

Tanti quanti per rappresentare un identificatore di processo (capitan ovvio). Quindi se un identificatore di processo è a 16 bit dovrò mettere 2 byte.

Domanda: quando cambio tra un thread ed un altro e cambio processo la tabella non è che la elimino? Cioè lo faccio passo passo?

No non la elimino. Posso fare questo perché? Supponiamo di mettere in queste situazioni un thread corrispondente ad un process ID pari a 4, quel thread andrà in esecuzione, genererà i suoi indirizzi virtuali, (chiaramente dovrò aver caricato in un registro del processore l'identificatore del processo altrimenti non funziona, lo caricherò nell'MMU) nessuno dei valori contenuti all'interno della TLB può corrispondere, perché non corrisponde l'identificatore del processo, questo vuol dire che ci sarà subito una TLB miss. L'MMU andrà nella tabella del processo 4, prenderà il descrittore di pagina cercato e lo caricherà in una di queste righe secondo una politica di sostituzione quindi da questo momento in poi quel thread andrà a riempire la TLB con i dati che lo riguardano. Se però dopo che ho lasciato in esecuzione questo processo ho riempito la TLB di righe corrispondenti al processo di indice 4, c'è una commutazione di contesto e passa in esecuzione un altro thread sempre del processo 4 ovviamente sono avvantaggiato.

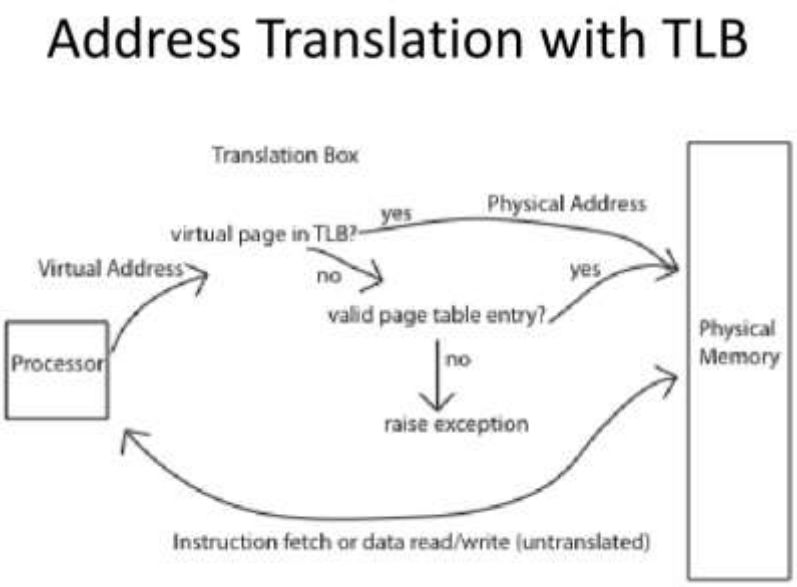


Cosa succede quando il sistema operativo deve cambiare i diritti relativi ad una pagina? Per esempio il sistema operativo può stabilire che una pagina non ha più diritto di scrittura ma soltanto in lettura. O potrà fare delle operazioni (che vedremo dopo) relative alla gestione delle pagine. Quindi cambia il contenuto della tabella delle pagine. In questi casi, può darsi che quel gestore delle pagine che il sistema operativo ha

modificato nella tabella delle pagine, abbia una sua copia in una qualche TLB su qualche processore. E questo è un problema perché se quella copia viene utilizzata, non è aggiornata. Immaginate una situazione nella quale il sistema operativo modifica i diritti di accesso di una pagina, scrive che non c'è più per quella pagina il diritto di scrittura ma solo di lettura. Il sistema operativo questa modifica la fa nella tabella delle pagine in memoria principale. Nell'MMU potrebbe essere rimasta una riga associata proprio a quel descrittore e all'interno della TLB resta segnato il fatto che quella pagina è accessibile sia in lettura che in scrittura quindi con ancora i diritti di scrittura. Se passa in esecuzione un thread che genera gli indirizzi relativi a quella pagina e cerca di scrivere lui potrà scrivere nonostante il sistema operativo l'abbia espressamente proibito nella tabella delle pagine.

Questo può avvenire anche quando c'è la copy on write ad esempio. Quando ci sono queste situazioni in realtà non si può soltanto modificare la tabella delle pagine ma bisogna avvisare l'MMU che va invalidata TLB (nel caso più brutale) se abbiamo un meccanismo più "fine" bisogna specificare che certe linee della TLB vanno invalidate.

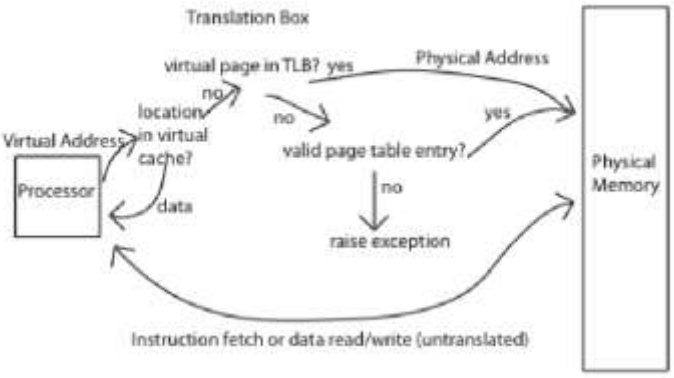
Se siamo su un sistema multicore, per cui ogni core ha la propria TLB, dobbiamo andare ad invalidarle tutte quante, dobbiamo andare a togliere da tutte le TLB potenzialmente quell'informazione non più valida. Come è fatto a questo punto il metodo di traduzione degli indirizzi usando la TLB.



L'indirizzo virtuale va dentro l'MMU, viene confrontato con il contenuto della TLB e si vede se è presente oppure no. Se quell'indirizzo virtuale è presente nella TLB possiamo, con le informazioni contenute nella TLB, fare la traduzione e inviamo questo indirizzo in memoria fisica. Altrimenti se non presente in TLB, l'MMU va ad analizzare la tabella delle pagine, se nella tabella delle pagine trova il descrittore valido per quella pagina, lo carica nella TLB, fa la traduzione e manda l'indirizzo fisico in memoria principale. Se invece nella tabella delle pagine, in memoria principale, il descrittore di pagina associato a quella pagina virtuale non è valido perché per esempio quella pagina virtuale non è allocata, allora il processore (l'MMU) non può farci nulla, non può risolvere questa situazione, manda un'eccezione al sistema operativo e poi il sistema operativo la gestisce (segmentation fault oppure quello che gli pare e piace).

Questa è la situazione semplice perché il processore non ha soltanto la TLB ma utilizza anche delle cache per memorizzare i dati, non soltanto per gli indirizzi. Quindi la situazione diventa questa

Virtually Addressed Caches



Il processore genera un indirizzo virtuale, questo indirizzo virtuale viene prima controllato con la cache dati di primissimo livello e si verifica se a fronte di questo indirizzo virtuale c'è già il dato che può essere prelevato dalla cache. Se si bene, se no l'indirizzo virtuale va alla TLB, si cerca di fare la traduzione, se si può tradurre bene, se non si può tradurre si guarda la tabella delle pagine, se dalla tabella delle pagine non c'è un descrittore valido, di nuovo eccezione.

Questo meccanismo in realtà è replicato più e più volte a vari livelli, non lo si fa quindi solo a livello del processore ma lo si fa anche a livello del sistema operativo per creare una gerarchia di memoria. In effetti nei sistemi attuali abbiamo una gerarchia di memoria molto profonda.

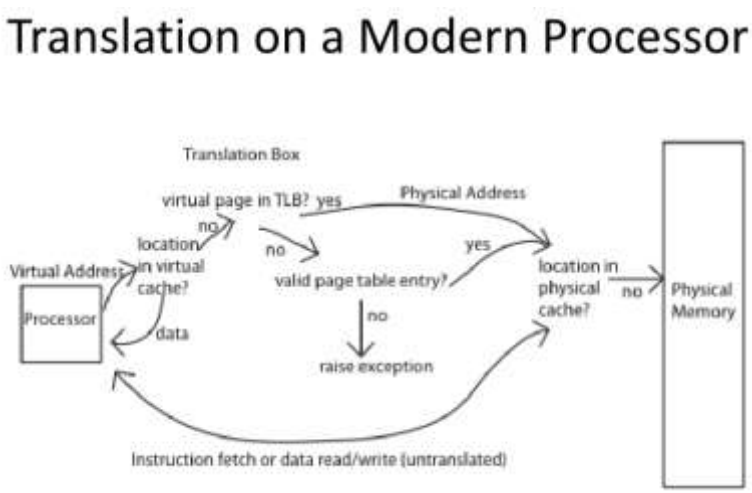
Memory Hierarchy

Cache	Hit Cost	Size
1st level cache/first level TLB	1 ns	64 KB
2nd level cache/second level TLB	4 ns	256 KB
3rd level cache	12 ns	2 MB
Memory (DRAM)	100 ns	10 GB
Data center memory (DRAM)	100 μs	100 TB
Local non-volatile memory	100 μs	100 GB
Local disk	10 ms	1 TB
Data center disk	10 ms	100 PB
Remote data center disk	200 ms	1 XB

i7 has 8MB as shared 3rd level cache; 2nd level cache is per-core

Partiamo dalla cache di primo livello che lavora su indirizzi virtuali quindi cache dati e primo livello della TLB. I vantaggi di questa cache è che ha l'accesso dell'ordine del nanosecondo però è anche molto piccola. C'è una cache di secondo livello sia per i dati sia per la TLB di nuovo con tempo di accesso molto ridotti e con una dimensione maggiore e poi si va alla cache di terzo livello che è spesso una cache della memoria fisica con tempi di accessi un po più alti e dimensione maggiore e soltanto come quarta battuta si ricorre alla memoria principale, la RAM che ha tempi di accesso molto più lenti rispetto alla cache di primo livello però ha una capienza dell'ordine di decine di gigabyte quindi decisamente superiore. Non ci si ferma qui perché a livello di sistema operativo si gestisce il caching per realizzare una gerarchia sui livelli superiori. Quindi la memoria viene vista come una cache di altri supporti di memoria o remoti o locali. Quindi via via si va ad incrementare la dimensione del supporto di memorizzazione e aumentare anche i tempi di accesso.

Quindi quando guardiamo un sistema operativo dobbiamo tenere a mente tutte queste gerarchie. In realtà il gioco è sempre lo stesso ovvero realizzare una cache per i vari livelli. (noi vedremo solo il rapporto tra memoria e disco). Il meccanismo è sempre quello della cache quindi la memoria principale viene vista come una cache di un qualcosa memorizzata nel disco, al livello di gestione non è più una cache gestita come una cache gestita a primo livello gestita dall'hardware. A livello di sistema operativo abbiamo una diversa flessibilità nell'utilizzo delle strutture dati, nella traduzione degli indirizzi… quindi anche se il concetto è quello di una cache i meccanismi sono differenti.



Quindi la traduzione nei processori moderni avviene in modo molto più articolata, il processore genera un indirizzo virtuale, se il dato è nella cache virtuale di primo o di secondo livello viene restituito altrimenti si manda l'indirizzo alla TLB, se si può tradurre l'indirizzo della TLB benissimo si traduce in indirizzo fisico e a questo punto si va alla cache fisica se nella cache fisica si trova il dato si manda al processore altrimenti si accede alla memoria principale.

Seconda parte:

Vediamo come viene gestita realmente la paginazione dal punto di vista del sistema operativo. Quello che abbiamo visto finora della paginazione è principalmente il meccanismo imposto dall'hardware, dal processore però questo non basta. Il processore per poter tradurre gli indirizzi si aspetta che qualcuno per lui abbia caricato la tabella delle pagine e gli metta a disposizione tutte le informazioni che gli servono. In realtà queste devono essere messe a disposizione dal sistema operativo. In tutto questo il sistema operativo ha due approcci: un approccio "teorico" che sarebbe: quando viene creato un processo alloca spazio in memoria fisica, allora la tabella delle pagine, inizializza la tabella delle pagine per allocare tutto lo spazio virtuale del processo e a questo punto mette in esecuzione il processo. In realtà questo non viene mai fatto perché è in larga misura inutile. Il processo non è detto che utilizzi tutte le pagine del suo spazio virtuale a maggior ragione con processi interattivi in cui le scelte fatte dal processo dipendono dalle scelte fatte dall'utente, andare a caricare in anticipo tutto lo spazio virtuale può rilevarsi un inutile spreco di tempo. Può darsi che alcune istruzioni non vengano utilizzate, alcuni dati non vengano letti, alcuni codici non vengono mai eseguiti. Per questo motivo alcuni sistemi operativi moderni adottano sistemi di paginazione on-demand ovvero quando il processo viene generato e viene messo in escuzione non vengono in realtà caricate le pagine del suo spazio virtuale quindi la tabella delle pagine del processo appena generato viene inizializzata interamente come invalida (certamente la tabella delle pagine deve essere allocata per lo meno la tabella delle pagine di primo livello) e si lascia che il processo passi in esecuzione, generi degli indirizzi e a seguito delle eccezioni che vengono fuori perché questi indirizzi corrispondono a pagine non caricate, il sistema operativo procede poco alla volta a caricare le pagine che servono per quel processo. Fa quindi una paginazione su richiesta. Questo permette di mettere in esecuzione molto rapidamente il processo (perché non abbiamo bisogno di aspettare che sia caricato il suo

spazio virtuale) e abbatte i costi di caricamento nel ciclo di vita completo del processo. Se si utilizza la tecnica di on-demand paging

dobbiamo cambiare un po la struttura del descrittore di pagina, in particolare il descrittore di pagina è un record che contiene un indice di pagina fisica e poi una serie di indicatori; in particolare alcuni indicatori relativi alla protezione (R-W) a questi dobbiamo aggiungere una serie di indicatori in particolare M-U che sono gli indicatori di pagina modifica o meno e di pagina in uso o meno e soprattutto il bit di presenza che è quello fondamentale per implementare la paginazione on-demand.

Ci sono diversi modi di implementarla ma supponiamo di avere un'implementazione di questo tipo se la pagina non è presente in memoria caricata (non è stata proprio caricata) il bit di presenza assume il valore 0, se la pagina è invece presente (è stata caricata in memoria principale) allora il suo bit di presenza è a 1. Nel caso in cui il processo generi un indirizzo di pagina virtuale corrispondente ad una pagina che è caricata (quindi il bit di presenza è 1) è tutto esattamente come prima questo descrittore può essere caricato nella TLB, la TLB può utilizzare l'indice di pagina fisica e può tradurre l'indirizzo. Se invece la pagina non è caricata in memoria principale e il bit di presenza è pari a 0 allora vuol dire che se il processore genera questo indirizzo virtuale, l'MMU usando la TLB non può tradurre l'indirizzo perché non sarà presente il descrittore nella TLB quindi l'MMU andrà a cercare il descrittore di pagina all'interno della tabella delle pagine, scoprirà che l'indicatore è pari a 0 e quindi non potrà procedere a caricare questo descrittore all'interno della TLB ma dovrà avvisare il sistema operativo, genererà un'eccezione. Il sistema operativo cosa fa a questo punto quando passa in esecuzione? Scopre che è stata riferita a questa pagina, scopre che non è presente, capisce se è una pagina che non è presente ma dovrebbe essere presente e si può caricare oppure è una pagina che non è presente perché non è stata mai allocata e quindi è un errore e si comporta di conseguenza. Se è un errore del processo da un segmentation fault, abortisce il processo altrimenti se la pagina non è caricata ma può essere presa dal disco allora procede al caricamento ma deve sapere dove si trova.

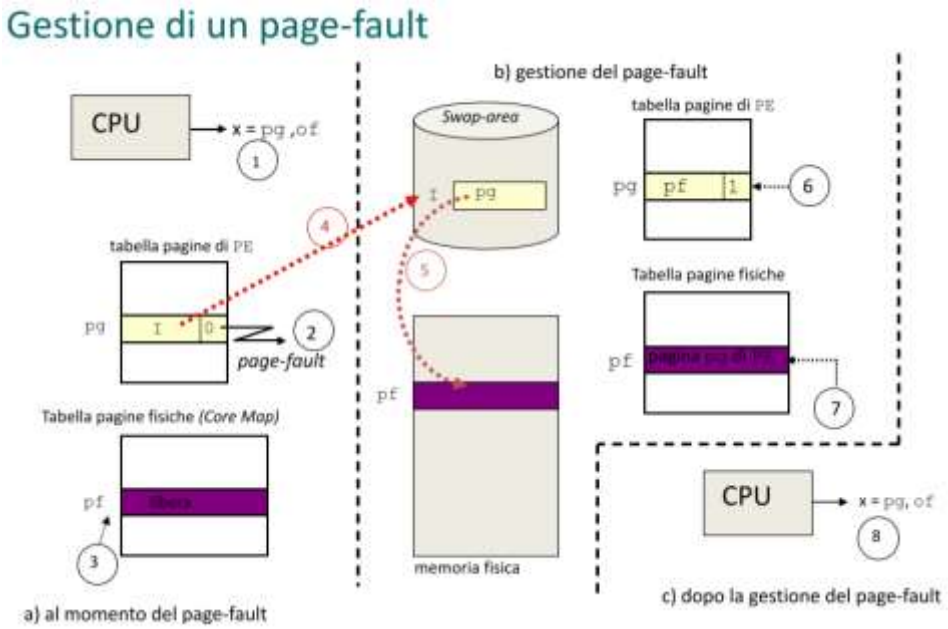
Per sapere dove si trova può usare il campo dell'indice di pagina fisica che in questo caso codifica uno spazio su disco dove andare a riferire la pagina.

Domanda: come è possibile che ho il mio indirizzo virtuale che è mappato in un indirizzo fisico che non c'è?

Perché quando il processo viene generato, non alloco memoria fisica per quel processo, non carico il codice dal suo file eseguibile, non faccio niente. Prendo la tabella delle pagine e la inizializzo scrivendo che tutti i descrittori sono non presenti. Nel descrittore di questo processo carico l'indirizzo della prima istruzione da eseguire e scrivo che quel processo è pronto, può essere messo in esecuzione. Ad un certo punto quel

processo passerà in esecuzione, il sistema operativo prende il suo descrittore lo copia nei registri del processore, fa l'iRet e a questo punto si carica nel processore l'indirizzo della prima istruzione da eseguire. Il processore genera questo indirizzo virtuale, questo indirizzo virtuale va alla TLB, non si può tradurre perché la TLB non ha mai visto questo thread quindi va a guardare la tabelle delle pagine (va in memoria principale ad esaminare il contenuto della tabella delle pagine), trova il descrittore di questa pagina virtuale, scopre però che non è stato caricato in memoria principale (nessuno l'ha fatto) e il bit di presenza è uguale a 0.

A questo punto il processore non può più fare niente, la MMU non può fare nulla, si arrende, genera un'eccezione (quindi solleva un'eccezione al processore) che mette in esecuzione il sistema operativo (quindi l'handler del sistema operativo abilitato a gestire questa eccezione), il sistema operativo capisce che: c'è stata un'eccezione di indirizzo generata dall'MMU, generata da un thread appartenente a un certo processo, quindi capisce qual è l'indirizzo che ha causato questa eccezione, va nella tabella delle pagine, capisce dov'è memorizzato in memoria secondaria lo carica etc…



Il processore genera un indirizzo virtuale formato da pagine ed offset (1). Non si può tradurre l'indirizzo perché non è presente questo descrittore di pagina virtuale. L'MMU a questo punto va nella tabella delle pagine del processo corrispondente e trova il descrittore della pagina ma con codice di presenza pari a 0. Genera un'eccezione verso il processore, questa eccezione si chiama eccezione di page-fault (manca la pagina). Questa eccezione causa l'esecuzione di un handler del sistema operativo (l'handler predisposto a gestire i page-fault). Questo handler capisce che era in esecuzione il processo PE e il fault è stato generato quando è stato generato un indirizzo della pagina pg, guarda la tabella delle pagine e si rende conto che questa pagina pg non è caricata in memoria principale ma è caricata in memoria secondaria presente sul disco ad un certo indirizzo sul disco. Cosa fa? Deve andare a caricarla ma per caricarla deve trovare uno spazio in memoria fisica dove poterla mettere, deve cercare un blocco fisico libero. Cosa fa a questo punto il sistema operativo? Va ad analizzare la tabella delle pagine inverse (ad esempio la core-map) che mi dice per ogni blocco fisico se è libero o se è occupato e se è occupato mi dice quale processo è stato allocato e quale pagina virtuale contiene. Andando quindi ad analizzare la core-map, il sistema operativo trova un blocco fisico libero per esempio il blocco fisico pf. A questo punto che fa? Prende il valore che era memorizzato nel descrittore della pagina (pg) questo valore a cose normali doveva essere l'indice di blocco fisico, siccome la pagina non è caricata in memoria principale questo valore è un valore che mi codifica una posizione di questa pagina nel disco da qualche parte. Il sistema operativo da comando al disco di andare a

leggere dall'indirizzo 'I' il contenuto della pagina pg e di andare a caricarla in memoria principale nel blocco fisico pf.

Fintanto che questa operazione di copia non si è completata, il sistema operativo mette in stato di blocco il thread che ha causato l'eccezione (resta in stato di wait). Manderà in esecuzione degli altri thread, faranno quello che dovranno fare, quando ad un certo punto questa copia dal disco alla memoria fisica si è completata, il disco lancia un'interruzione al sistema operativo, il sistema operativo torna in esecuzione, capisce che si è completata la copia della pagina richiesta dal processo PE all'interno della pagina fisica pf, a questo punto il sistema operativo aggiorna le strutture dati e in particolare cosa fa? Nella core-map va a scrivere che la pagina pf è allocata al processo PE e contiene la pagina logica pg del processo PE. Nella tabella delle pagine del processo PE va a scrivere che la pagina pg è caricata (quindi mette il bit di presenza pari ad 1) e nel campo di traduzione ci scrive l'indice della pagina fisica PE. Fatto questo il thread che ha causato l'eccezione viene di nuovo messo nello stato di pronto, viene riattivato, quando quel thread tornerà in esecuzione, genererà lo stesso indirizzo (< pg , of >) stavolta però l'indirizzo potrà di nuovo causare una TLB-miss perché non è presente nella TLB, la MMU andrà nella tabella delle pagine e stavolta associato a pg troverà il valore di pf, scoprirà che quella pagina è caricata nel blocco fisico pf quindi prende l'indice pf, lo combina con l'offset e traduce l'indirizzo.

(SPOILER)

Utilizzato questo meccanismo fino in fondo cosa succede in realtà? Succede che io non ho bisogno di tenere in memoria tutto l'intero processo per poterlo mandare in esecuzione, potrei eseguire un processo usando soltanto una pagina fisica. Questo vuol dire che in memoria principale io di processi ne posso mettere di più. Se io ho 4 giga di memoria fisica e i processi hanno una memoria virtuale di 4 giga, basta un processo che vuole utilizzare tutti e 4 i giga e o ci sto io o ci stanno gli altri, potrei avere un solo processo in memoria principale. In realtà utilizzando questo meccanismo io in memoria principale di processi ne potrei avere centinaia, migliaia caricati in memoria principale perché per ogni processo mi basta avere soltanto le pagine che effettivamente gli servono. Questo permette in realtà di non caricare per intero un processo quando lo mando in esecuzione una prima volta ma mi permette anche di non caricare dati o codice che quel processo durante la sua esecuzione non userà mai (quindi risparmio sicuro), mi permette anche di mantenere in memoria principale un numero molto più alto di processi. Tenere un numero molto più alto di processi vuol dire che posso soddisfare i bisogni di più utenti quindi ho più processi interativi in funzione, un sistema che globalmente ha prestazioni migliori. Aumento il grado di multiprogrammazione del sistema. I vantaggi sono innumerevoli ed infatti non è un caso che tutti i sistemi operativi moderni usano questo meccanismo (si chiama paginazione a richiesta).

(END SPOILER)

Quindi completato il meccanismo il thread tornerà in esecuzione genererà quell'indirizzo e questo punto lo posso tradurre. Riassumendo: qui c'è tutta la lista di eventi che avvengono nel caso in cui si utilizzi la paginazione on-demand e si riferisca ad una pagina che non è caricata in memoria principale.

Demand Paging

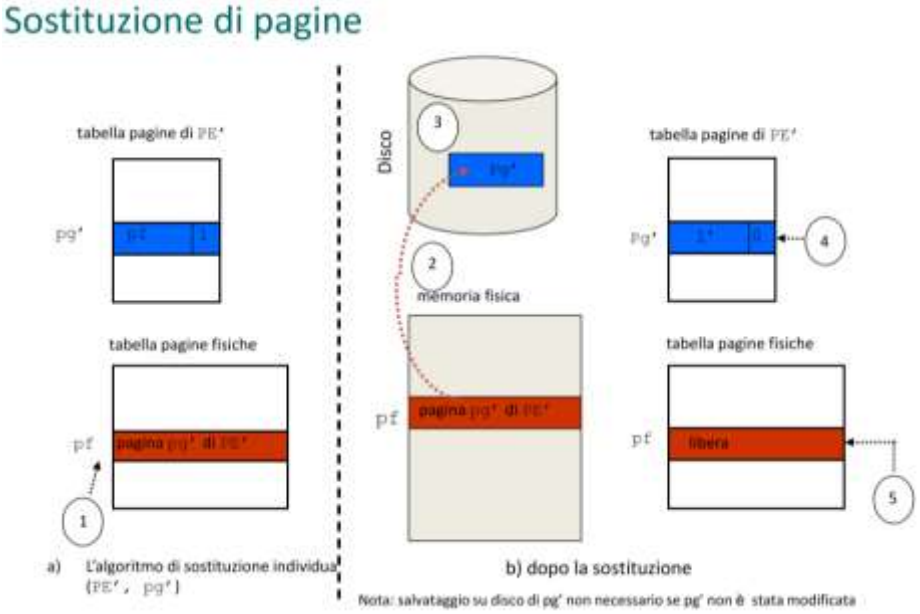
1. TLB miss
2. Page table walk
3. Page fault (page invalid in page table)
4. Trap to kernel
5. Convert address to file + offset
6. Allocate page frame
   - Evict page if needed
7. Initiate disk block read into page frame
8. Disk interrupt when DMA complete
9. Mark page as valid
10. Resume process at faulting instruction
11. TLB miss
12. Page table walk to fetch translation
13. Execute instruction

La prima cosa (1) che succede è la TLB miss quindi il thread in esecuzione genera un indirizzo, questo indirizzo non si può tradurre nella TLB. Può darsi che sia perché il descrittore di quella pagina non è caricato nel TLB quindi l'MMU va a scorrere la tabella delle pagine per cercare quel descrittore (2). Lo trova ma scopre che non è valido, che quella pagina non è caricata quindi l'MMU genera un page-fault (3) e fin qui siamo nell'hardware. Generato il page-fault, che è un'interruzione sostanzialmente, si passa al nucleo del sistema operativo, in particolare nell'handler del page-fault (4). L'handler del page-fault traduce l'indirizzo virtuale di pagina generato da quel thread in un indirizzo ad un file (5). Quindi quando il processo è stato generato le sue pagine non sono state caricate in memoria principale però il sistema operativo si è segnato dove trovarle.

(esempio concreto) Quando metto in esecuzione il processo lo invoco specificando il nome del file eseguibile, questo vuol dire che il suo codice è già scritto nel file eseguibile. Fino ad ora prima di usare la paginazione on-demand io dovevo allocare delle pagine fisiche in memoria e copiare il file eseguibile all'interno di queste pagine fisiche. Ora questa operazione non la sto facendo, questo vuol dire che queste pagine virtuali (che sono ipoteticamente presenti ma non sono ancora allocate fisicamente) sono si presenti ma sul disco, sul file eseguibile. Quando quindi viene generato un indirizzo di codice, questo indirizzo di codice corrisponde ad una pagina non caricata in memoria principale ma che può essere trovata andando a prendere dal file eseguibile ed è da li che viene presa (questo per quanto riguarda il codice). Quindi codice, dati statici e libreria li trovo nei file eseguibili o nei file di librerie. Per quanto riguarda i dati invece (quelli che il processo ha generato di suo) se non fossero presenti vanno memorizzati da un'altra parte (lo vediamo dopo).

Per ora l'handler del sistema operativo, l'handler del page-fault traduce l'indice di pagina virtuale in un indirizzo ad un file (5). Trova una pagina fisica libera eventualmente se tutta la pagina fisica è occupata rimuove una pagina allocata in precedenza per avere una pagina fisica dove mettere la richiesta (anche questo lo vediamo dopo) (6). Trova una pagina fisica libera, da comando al disco per leggere la pagina virtuale dal disco verso la pagina fisica allocata (7). Quando il disco ha completato la copia genera un'interruzione, torna in esecuzione il sistema operativo che capisce che si è completata l'operazione di caricamento di questa pagina (8). Nella tabella delle pagine marca la pagina come pagina valida(9). Nella core-map segna che la pagina fisica è stata allocata a quel processo. A questo punto mette in esecuzione il thread che riparte da dove è stata generata l'eccezione (quindi dall'indirizzo che aveva causato l'eccezione)(10). Questo indirizzo nuovamente genera una TLB miss (11) perché non può essere già stato caricato nella TLB ovviamente. L'MMU va nella tabella delle pagine per caricare il descrittore della pagina virtuale, stavolta lo trova ed è valido, lo carica nella TLB e a questo punto può fare la traduzione e si può andare avanti.

Il meccanismo è unico, non ci sono tante alternative, è una sequenza obbligata usando la paginazione on-demand. Cosa facciamo se la memoria fisica è tutta occupata? Dobbiamo andare a cercare una pagina fisica da liberare. Ovviamente non possiamo liberarla così allegramente perché quella pagina fisica era stata allocata da qualcun altro e di conseguenza bisogna stare attenti a lasciare tutte le pagine in uno stato consistente. Vediamo il meccanismo di sostituzione delle pagine



Supponiamo che in questa situazione (al punto 3 della slide precedente) non ci sia una pagina libera in memoria principale quindi la core-map segna tutte le pagine come occupate. Utilizziamo un algoritmo di sostituzione (lo vedremo più avanti) che tra tutte le pagine fisiche occupate ne sceglie una da liberare. Quindi supponiamo tramite un algoritmo di aver individuato una pagina fisica pf che attualmente è utilizzata dal processo PE' per conservare la pagina virtuale pg'. Vogliamo liberare questa pagina fisica (per qualche motivo abbiamo scelto questa). Questa pagina virtuale pg' in realtà è rappresentata in due punti: la sua locazione è rappresentata in core-map ma è anche rappresentata nella tabella delle pagine PE'. Nella tabella delle pagine del processo PE' in corrispondenza della pagina pg' abbiamo segnato che questa pagina virtuale è caricata, è presente in memoria nella pagina fisica pf. Rimuoverla vuol dire andare ad aggiornare entrambe le tabelle: sia la tabella delle pagine che la core-map. C'è una complicazione: può darsi che la pagina pg' sia stata modificata dal processo PE' quindi se noi semplicemente la cancelliamo dalla memoria principale, PE' perde delle informazioni che lui ha scritto su quella pagina. Non possiamo quindi semplicemente andare a prendere questa pagina e svuotarla e caricarci qua sopra la pagina richiesta dal processo PE. Dobbiamo prima di tutto salvarla quindi cosa si fa? Il sistema operativo se la pagina pg' di PE' è stata modificata (se e solo se è stata modificata perché se non è stata modificata è già presente nel disco aggiornata e quindi non ho bisogno di fare questo) per prima cosa si dà il comando al disco di andare a scrivere, da qualche parte, il contenuto della pagina pf.

Quindi prima si salva la pagina pg' di PE' da qualche parte in qualche indirizzo nel disco. Fatto questo la pagina fisica può essere marcata come libera e nella tabella delle pagine del processo PE' segniamo che la pagina pg' non è più presente e mettiamo il bit di presenza pari a 0 e nel campo "indirizzo" andiamo a scrivere l'indirizzo " I' " dove in memoria secondaria è presente la pagina pg'. Supponiamo che questa pagina pg' contenga un pezzo dello stack di PE', lo stack contiene dei dati che sono stati modificati a tempo di esecuzione e non avevano una zona nel disco per essere memorizzati. Noi sappiamo che il codice lo possiamo andare a prendere dal file eseguibile ma il contenuto dello stack non può stare nel file eseguibile. Lo stack è stato modificato ed aggiornato durante la sua esecuzione dal processo PE' con dei dati che sono stati scritti ex-novo. Per questo motivo i dati devo essere messi nella memoria secondaria nel disco in una qualche parte fatta apposta per poterli contenere. Questa parte può essere o una partizione del disco separata (come si fa spesso in linux): si utilizza un'area del disco separata dove vengono memorizzate tutte le informazioni scaricate dalla memoria principale oppure (come nel caso di Windows) si utilizza un file speciale che contiene tutte queste informazioni. Quale sia il meccanismo non fa differenza, c'è un'area di swap nel disco dove queste informazioni vengono depositate e questi indirizzi fanno riferimento a quell'area.

Riassumendo: per il codice possiamo sempre fare riferimento al file eseguibile utilizzato per caricare il processo, per i dati facciamo riferimento ad un'area speciale del disco (che può essere o un file speciale o una partizione) dove le informazioni scaricate alla memoria principale vengono conservate. Ogni processo in ogni instante di tempo avrà alcune pagine virtuali caricate in memoria principale e tutte le altre pagine virtuali caricate nel disco in punti differenti con il codice memorizzato nel file eseguibile e i dati memorizzati nell'area di swap. Il sistema operativo per ogni pagina deve sapere dove sta, se è allocata in memoria principale, se è allocata nel disco e dove.

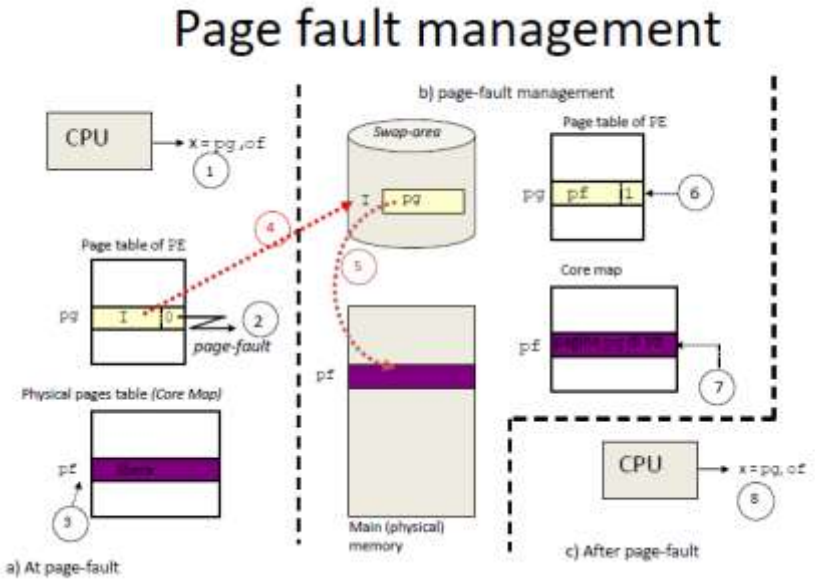
Se pg' era presente nella TLB, andando a fare questa operazione bisogna che il sistema operativo avvisi la MMU di andare a cancellare dalla TLB l'eventuale riga che descrive pg', questo va fatto assolutamente ed è il caso in cui bisogna andare ad annullare il contenuto della TLB (questo riguarda l'hardware c'è bisogno che il sistema operativo abbia modo di passare questa informazione alla TLB). Nel punto (1) una pagina, chiamiamola pg, non può essere presente nella TLB perché pg fa riferimento ad una pagina che non è mai stata caricata in memoria principale quindi nella TLB non ci può proprio essere. Per caricarla nella TLB bisogna che ci sia stata una TLB miss in precedenza riferendo la stessa pagina e l'MMU avrebbe provocato il caricamento in memoria principale.

Domanda: può rispiegare la core-map?

È un concetto molto semplice: noi abbiamo una memoria fisica che è divisa in pagine fisiche, la core-map è una struttura dati (un array) che contiene un descrittore per ogni pagina fisica (quindi 100 pagine fisiche-> 100 elementi dell'array). Ogni riga della core-map corrisponde ad una pagina fisica e contiene informazioni relative alla locazione di questa pagina fisica. In particolare se la pagina fisica è libera c'è scritto che è libera, se la pagina fisica invece è occupata vuol dire che è stata allocata ad un processo e di quel processo mappa una pagina virtuale. Nel descrittore di questa pagina fisica conserviamo queste due informazioni: è allocata al processo PE e contiene l'indice di pagina virtuale pg. In realtà nella core-map (sento puzza di spoiler) ci sono altre informazioni in più (eccolo, spoiler) che servono per gestire i meccanismi di sostituzione delle pagine. Se la memoria principale è tutta occupata uso un algoritmo che si chiama algoritmo di sostituzione per andare a cercare una pagina fisica che posso liberare ma questi algoritmi di sostituzione per trovare una buona pagina fisica da liberare hanno bisogno di informazioni aggiuntive, queste informazioni aggiuntive non le abbiamo ancora viste ma le vediamo dopo e devono essere memorizzate nella core-map. La core-map ha una dimensione proporzionale alla memoria fisica quindi facciamo un'ipotesi: supponiamo di avere memoria fisica di 4 gb con indirizzi a 32 bit. Supponiamo inoltre che le pagine fisiche siano di 4k quindi abbiamo 1 milione di pagine e la core-map ha 1 milione di righe. In questo caso la core-map occupa 8mbyte a fronte di una memoria fisica di 4 gb. Però la core-map è unica per tutto il sistema, non ho una core-map per ogni processo viceversa le tabelle delle pagine ne devo avere una per ogni processo (100 processi -> 100 tabelle delle pagine). Quindi non è un overhead enorme. (il Chessa, non soddisfatto, spara una h8frase prima di terminare) Tutte le tabelle a loro volta potrebbero essere sul disco e paginate a loro volta.

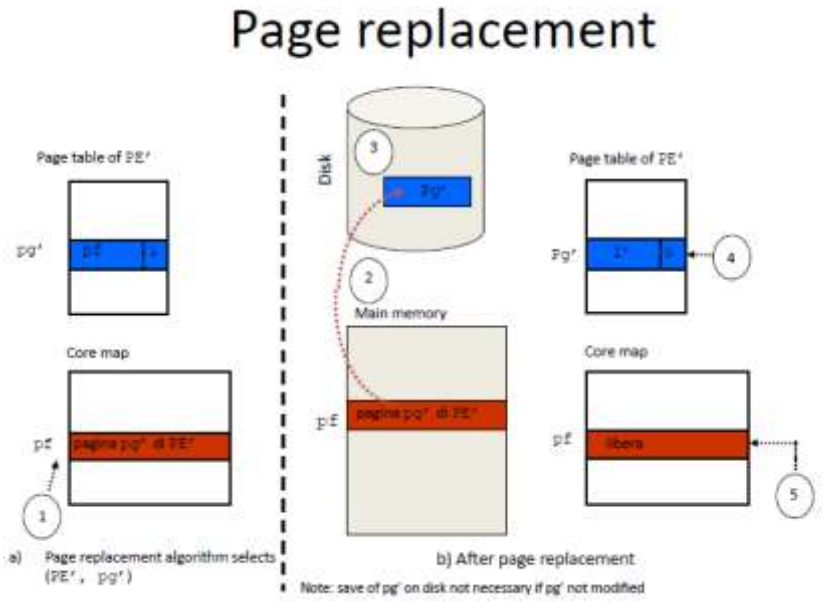
Sistemi operativi 11/05/17

Riprendiamo il discorso sulla paginazione On Demand. Se vi ricordate, il meccanismo era questo: quando viene generato un indirizzo virtuale se l'MMU non riesce a tradurre l'indirizzo, perchè non lo trova nella TLB, esso va a guardare la tabella delle pagine: se in tale tabella quella pagina è marcata come non valida vuol dire che non è caricata in memoria principale. La MMU, a questo punto, non può fare niente, genera un Fault di Pagina, interviene il SO che va a caricare la pagina sul disco e per far questo va ad individuare una pagina fisica libera sulla quale allocare la pagina richiesta.



Fatto questo periodicamente, a questo punto, ripristina le strutture dati, inizializza la tabella delle pagine del processo in maniera tale da creare l'associazione tra pagina virtuale e pagina fisica che la contiene; aggiorna la Core Map e a questo punto ripristina il processo.

Ora, per poter far questo bisogna avere spazio libero in memoria principale per poter andare a caricare la pagina richiesta dal processo. Non è detto che ci sia sempre questo spazio libero: per questo motivo, se la memoria è tutta occupata bisogna decidere quale pagina rimuovere ed è per questo motivo che esistono algoritmi di sostituzione delle pagine.



Per cui si va a cercare (analizzando la core map) una pagina che si può rimuovere. Eventualmente questa pagina è stata modificata rispetto all'ultimo utilizzo, va salvata sul disco da qualche parte in memoria secondaria; una volta che questo salvataggio è stato completato, a quel punto, la pagina fisica è libera e può essere utilizzata per completare l'algoritmo di sostituzione. Quindi, con questo algoritmo, abbiamo bisogno di poter fare una scelta: individuare tra tutte le pagine presenti in memoria principale, quale può essere rimossa.

Allocating a Page Frame

- Select old page to evict
- Find all page table entries that refer to old page
  - If page frame is shared
- Set each page table entry to invalid
- Remove any TLB entries
  - Copies of now invalid page table entry
- Write changes to page to disk, if necessary

Tale scelta non è una scelta banale, perchè se rimuoviamo una pagina che poi verrà riferita nuovamente entro breve tempo, saremo costretti a ricaricarla. Questa è esattamente la stessa problematica che si incontra quando si realizzano le cache. Bisognerebbe cercare di rimuovere le pagine dalle cache soltanto quando non verranno usate per un certo tempo, possibilmente il più lungo possibile. Ora, per poter eseguire questo tipo di algoritmo è necessario avere un po' di informazioni relative all'uso delle pagine. In particolare dobbiamo sapere due cose:

- se la pagina è stata usata, magari ci farebbe anche piacere sapere quando è stata usata (dopo vedremo i dettagli su questo);
- l'altra informazione che ci serve sapere per ogni pagina è se è stata modificata, perchè se così fosse andrebbe salvata su disco prima di rimuoverla.

Come facciamo a ottenere queste informazioni?

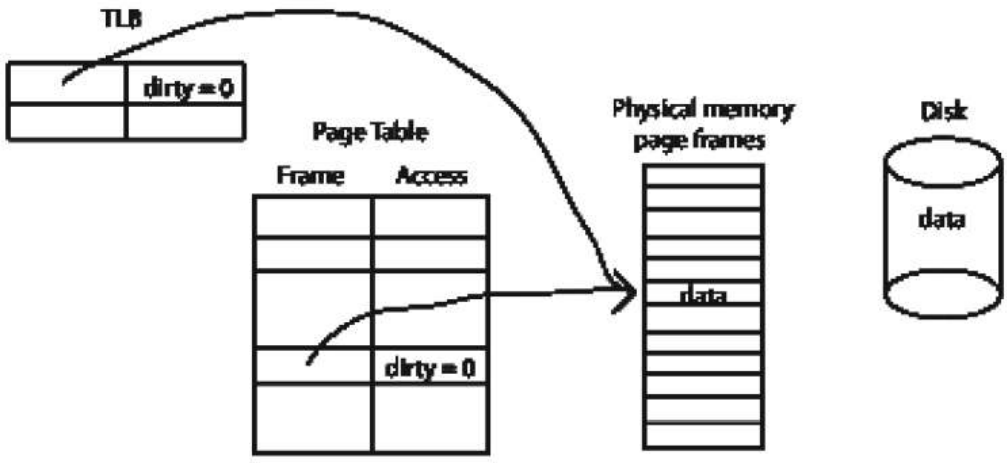
How do we know if page has been modified?

- Every page table entry has some bookkeeping
  - Has page been modified?
    - Set by hardware on store instruction to page
    - In both TLB and page table entry
  - Has page been used?
    - Set by hardware on load or store instruction to page
    - In page table entry on a TLB miss
- Can be reset by the OS kernel
  - When changes to page are flushed to disk
  - To track whether page is recently used

Badate bene che sapere se la pagina è stata modificata, questo vuol dire che è un'informazione che va creata quando la pagina viene materialmente modificata e per sapere se la pagina è stata usata è un'informazione che va creata quando la pagina viene effettivamente usata. Chiusa la pagina chi la modifica è il processo quando è in esecuzione e materialmente chi fa l'operazione di accesso o di modifica è il processore. Quindi dovremmo intervenire quando il processore fa l'accesso in memoria, capire qual è la memoria riferita e di conseguenza mappare, segnalare nella pagina corrispondente che è stata riferita oppure modificata. Per esempio, per quanto riguarda il fatto di sapere se la pagina è stata modificata, bisogna avere un aiuto dall'hardware, la cosa si fa in questa maniera: bisogna aggiungere un ulteriore bit tra gli indicatori nel descrittore di pagina da aggiungere a quelli usuali.

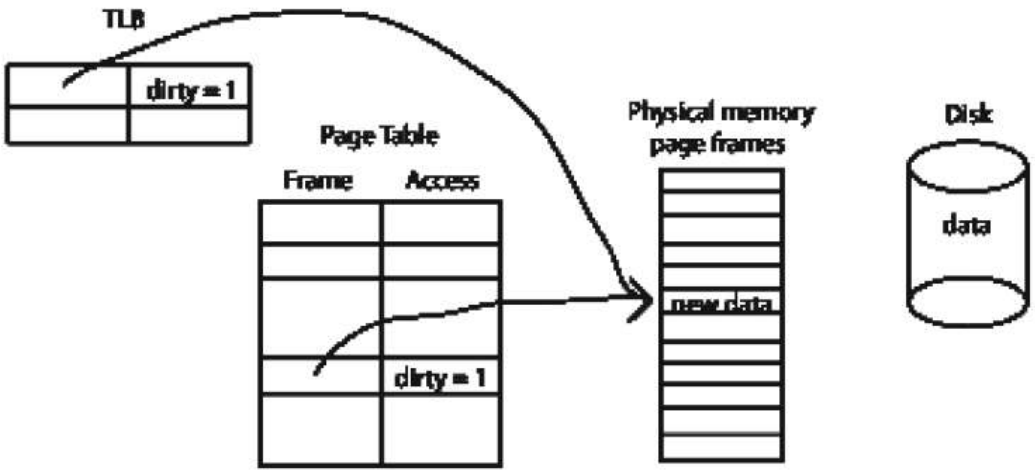
Oltre agli indicatori di protezione e di presenza ci serve un bit per rappresentare il fatto che la pagina è stata modificata o meno. Questo bit spesso si chiama Dirty Bit. Questo bit viene aggiunto a tutti i descrittori di pagina, quindi deve stare nella tabella delle pagine. Quando il processore genera un indirizzo e la MMU cerca di tradurla andando ad esaminare la TLB, l'MMU va, in pratica, ad estrarre dalla TLB un descrittore di pagina. Quindi quando fa questa operazione può andare a settare il bit di "modificata" nel descrittore della pagina contenuto all'interno della TLB. Quando poi questa pagina, questo descrittore dovrà uscire dalla TLB per un motivo o per l'altro, dovrà essere aggiornato anche il descrittore della pagina nella tabella delle pagine in memoria principale. E' l'hardware che setta che la pagina è stata modificata e nella stessa maniera, quando c'è un'operazione di scrittura su una certa pagina, e nella stessa maniera l'hardware fa la stessa cosa quando c'è un'operazione di accesso generico a pagina, che sia di lettura o di scrittura. Quindi il bit di uso è l'altro bit che utilizziamo all'interno del descrittore delle pagine e anche questo viene settato dall'hardware (dalla MMU) ogniqualvolta intercetta un indirizzo virtuale.

L'hardware, in questa maniera, può andare a settare questi bit, però è evidente che poi quando la pagina viene scaricata o quando la pagina non serve più, questi bit vanno rimessi a zero. Questa operazione di rimetterli a zero non la fa l'hardware, ma viene fatta dal sistema operativo, nell'ambito dei suoi algoritmi di sostituzione, quindi quando è opportuno farlo ci pensa il sistema operativo a resettarli. In pratica i bit di uso e di modifica di ogni singola pagina vengono settati dalla MMU ogniqualvolta c'è un riferimento e ho una modifica ad una pagina e questo bit viene settato sia nella TLB che nella tabella delle pagine. Questo bit viene analizzato dall'algoritmo di sostituzione per poter fare i suoi ragionamenti sulla sostituzione delle pagine presenti in memoria e nel far questo è il sistema operativo che li resetta questi bit, li riporta a zero.



Se noi abbiamo questa situazione dove nella TLB e nella tabella delle pagine abbiamo un riferimento ad una certa pagina caricata in una pagina fisica in memoria, tale pagina avrà anche un Back in Store (o Backing Store, non si capisce) sul disco dove viene conservata. Vedete che il Dirty Bit (quindi i bit di modifica) è

tenuto a zero. Se per caso questa pagina viene modificata con dei nuovi dati, la pagina viene modificata in memoria principale, nel disco continuiamo a conservare una copia che non è aggiornata, perchè non faccio ogni volta che si modifica una pagina, quello sarebbe molto costoso. Però ci segniamo sia nella TLB che nel descrittore della pagina, nella tabella delle pagine, il fatto che questa pagina è stata modificata.



Per cui se ad un certo punto dovessimo rimuoverla dalla memoria principale, analizzando questo bit sapremmo che lo dovremo salvare. Alcune architetture, in realtà, non hanno un supporto per gestire i bit di modifica e di uso.

Emulating a Modified Bit

- Some processor architectures do not keep a modified bit in the page table entry
  - Extra bookkeeping and complexity
- OS can emulate a modified bit:
  - Set all clean pages as read-only
  - On first write, take page fault to kernel
  - Kernel sets modified bit, marks page as read-write

O meglio diciamo tutte le architetture più recenti hanno acquistato funzionalità o qualcosa di equivalente, però non è sempre stato così e potrebbe non essere così, magari in un'architettura più a basso costo o meno performanti. D'altra parte questi bit possono essere emulati dal SO. Come si può fare ad emularli? E' un trucco abbastanza semplice: la pagina la si marca come non valida, se per caso c'è un tentativo di accesso con la pagina la MMU non riesce a tradurre l'indirizzo perchè la pagina è stata marcata come valida, genera un Fault di Pagina. Il SO a questo punto si rende conto che quel fault di pagina non è un vero fault di pagina perchè la pagina in realtà è caricata, ma deve semplicemente settare lui i bit di uso o di modifica a seconda dell'operazione che è stata richiesta. Il SO attua questa modifica e poi ripristina il thread in esecuzione. Chiaramente emulare i bit di uso e di modifica senza supporto dell'hardware è molto

costoso, perchè ogni volta che una pagina viene modificata bisogna gestire un'interruzione, la prima volta che una pagina viene utilizzata nuovamente bisogna gestire un'interruzione. Dopo che la pagina è stata modificata o utilizzata il bit viene settato, la pagina viene marcata come valida, poi successivamente non c'è bisogno di generare nuove interruzioni per quella pagina. Quindi la cosa è possibile però ovviamente non è molto comodo. Questo può esser fatto sia per il bit modificata sia per il bit di uso.

Come è ormai ampiamente chiaro, questo meccanismo di paginazione On Demand ricalca le stesse problematiche della gestione delle cache. Ora riprenderei un parallelo brevissimo con le cache per farvi capire dove sta il punto di incontro.

Definitions

- Cache
  - Copy of data that is faster to access than the original
  - Hit: if cache has copy
  - Miss: if cache does not have copy
- Cache block
  - Unit of cache storage (multiple memory locations)
- Temporal locality
  - Programs tend to reference the same memory locations multiple times in a given period of time
  - Example: instructions in a loop
- Spatial locality
  - Programs tend to reference nearby locations
  - Example: data in a loop

La cache una struttura che contiene dei dati e che serve a garantire un accesso più rapido a questi dati. I dati normalmente hanno un posto dove vengono conservati, da qualche parte.. (OK) La cache è una memoria spesso più piccola, più rapida, che permette un accesso più veloce. Quindi il gioco è che se troviamo il dato nella cache possiamo, diciamo, fornirlo più rapidamente al processore e quindi il processore va avanti più speditamente, senza doverlo andare a reperire. La cache l'avete vista ad Architetture nella gestione della memoria principale, quindi si frappone tra la memoria principale e il processore.

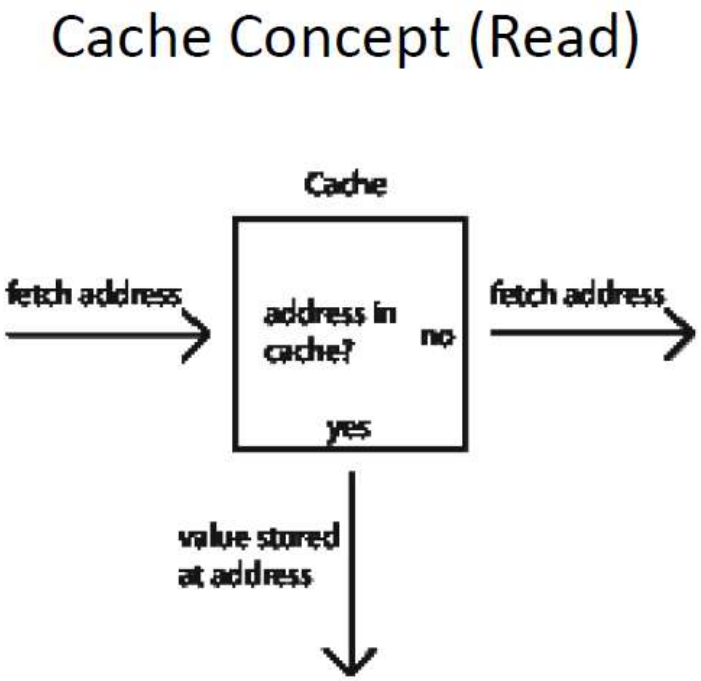
Quello che succede con la paginazione on demand è che in realtà la memoria principale, stavolta, diventa una cache di una memoria più grande, più lenta, più capiente che non è altro che il disco. In realtà quello che stiamo dicendo è che la memoria vera, virtuale del processo sta sul disco in memoria secondaria da qualche parte e la memoria principale non è altro che la sua cache. Quindi le pagine stanno su disco, vengono portate in memoria principale a richiesta e nella gestione di tutto questo ogni tanto devono ritornare indietro se dobbiamo liberare spazio nella cache.

Questo meccanismo, come nel caso delle cache del processore, funziona se l'accesso alla memoria virtuale avviene con proprietà di località temporale e di località spaziale. Questo perchè una volta che abbiamo caricato un dato nella cache, che sia cache del processore o cache della memoria rispetto alla memoria secondaria, quel dato resta facilmente accessibile, quindi se viene riferito nuovamente in futuro, l'accesso diventa più rapido. Il problema è che noi abbiamo pagato di più per gestire la cache, quindi per andarlo a caricare. Una volta caricato guadagniamo perchè l'accesso diventa molto più rapido. Questo guadagno è tanto maggiore quanto più questo valore, caricato in cache, viene riutilizzato nel tempo. Questo avviene se

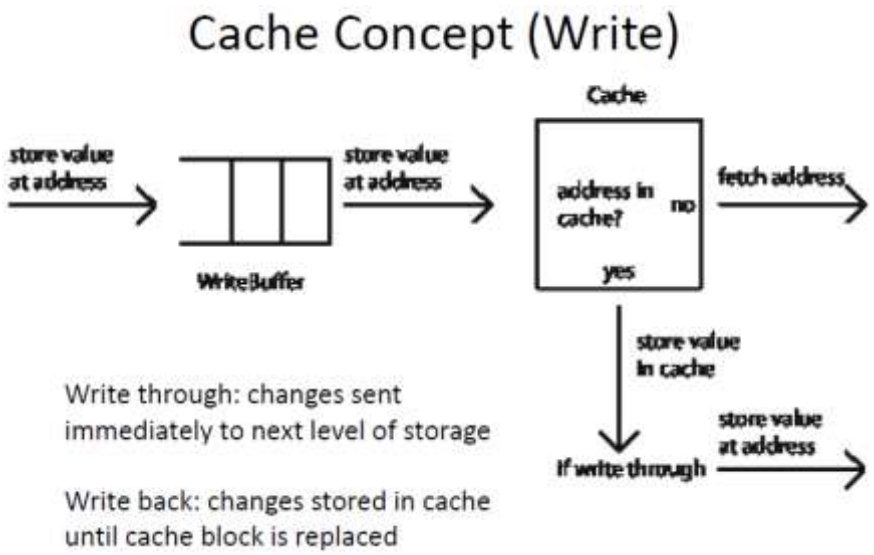
l'utilizzo dei dati in memoria appunto ha proprietà di località temporale: quindi i dati riferiti vengono con alta probabilità, riferiti nuovamente anche in tempi successivi.

Stesso discorso per la località spaziale: anch'essa è importante perchè noi in realtà in memoria principale andiamo a caricare una pagina intera.

Quindi andiamo a caricare la memoria a blocchi e una pagina rappresenta una sequenza contigua di byte utilizzati da quel thread. Quindi se quel thread ha proprietà di località spaziale nel riferimento della memoria, una volta caricata la pagina molto probabilmente i dati contenuti in quella pagina verranno riferiti più e più volte e quindi avremo un guadagno di prestazioni, perchè risparmieremmo tempo. La cache normalmente come funziona?



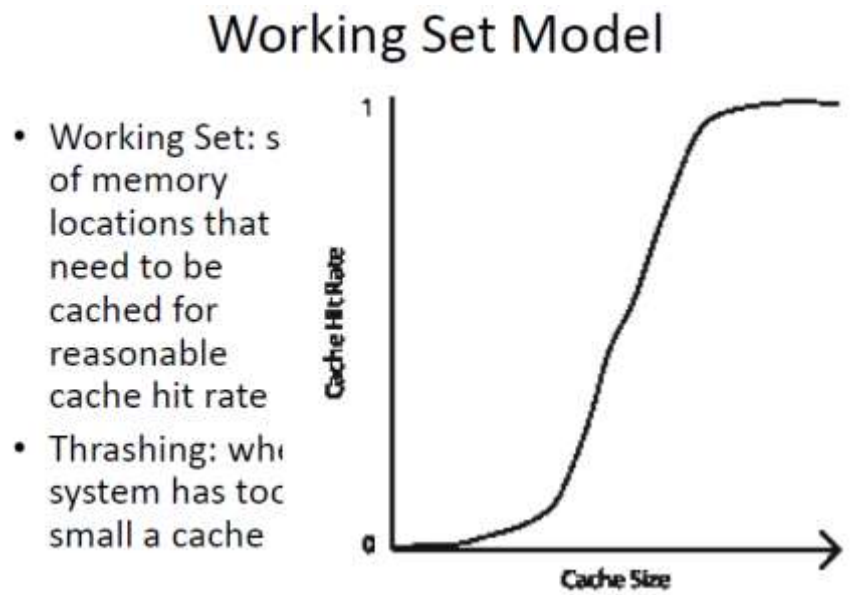
È una black box, voi mandate un indirizzo di un dato da caricare se state facendo un'operazione di lettura; se quell'indirizzo è caricato in cache allora dalla cache possiamo astrarre il valore allocato in quella allocazione riferita e possiamo restituirlo al processore. Altrimenti dobbiamo andare a caricare il dato dalla memoria principale, lo montiamo nella cache e a quel punto possiamo rispondere. Quindi abbiamo un overhead nel caso in cui il dato non sia in cache, abbiamo un guadagno molto forte nel caso in cui il dato sia in cache. Nel caso in cui, invece, la cache venga utilizzata in scrittura, quindi c'è un riferimento a una locazione, ad un indirizzo virtuale, con la richiesta di andare a scrivere un dato, anche in questo caso la cache aiuta.



Ciò perchè se un indirizzo riferito sul quale vogliamo scrivere in cache benissimo, la scrittura la possiamo fare direttamente in cache, senza perdere tempo ad andare in memoria principale. Se invece il dato non è presente in cache allora siamo costretti a fare la scrittura in memoria principale.

Una cosa: quando la scrittura viene fatta direttamente nella cache, l'azione ovviamente è rapida e quindi si può subito dire che ho fatto l'operazione di scrittura. Però in realtà, per ridurre possibili inconsistenze nei dati soprattutto nel caso in cui abbiamo un processore, se la cache funziona in write through viene fatta la scrittura della cache, ma subito dopo (in maniera asincrona rispetto all'attività del processore della cache) viene anche memorizzato il dato nella memoria principale. In questo modo risparmiamo il tempo di accesso: in questo caso perchè restituiamo subito il processore, però ci troviamo il dato aggiornato nella cache sia in memoria principale (questo con la write through).

Gli stessi meccanismi li troviamo nella paginazione. Quindi la paginazione opera facendo esattamente queste stesse operazioni, in collaborazione con la MMU. Se c'è un riferimento ad un indirizzo virtuale di una pagina caricata in memoria principale benissimo, ce l'abbiamo in memoria principale, altrimenti andiamo a fare il caricamento. Ovviamente nel caso della paginazione queste operazioni di caricamento dell'indirizzo, di salvataggio del valore di quell'indirizzo, nel caso in cui l'accesso sia in lettura, tali operazioni sono molto più costose perchè in questo caso non stiamo andando in memoria, ma stiamo andando su disco. Quando si parla di cache bisogna tener presente qual è il modello di riferimento della memoria utilizzato dai processi e si osserva che spesso i processi, i thread hanno comportamenti molto simili dal punto di vista della gestione della memoria. In particolare quello che ci interessa tenere a mente è il concetto di working set.



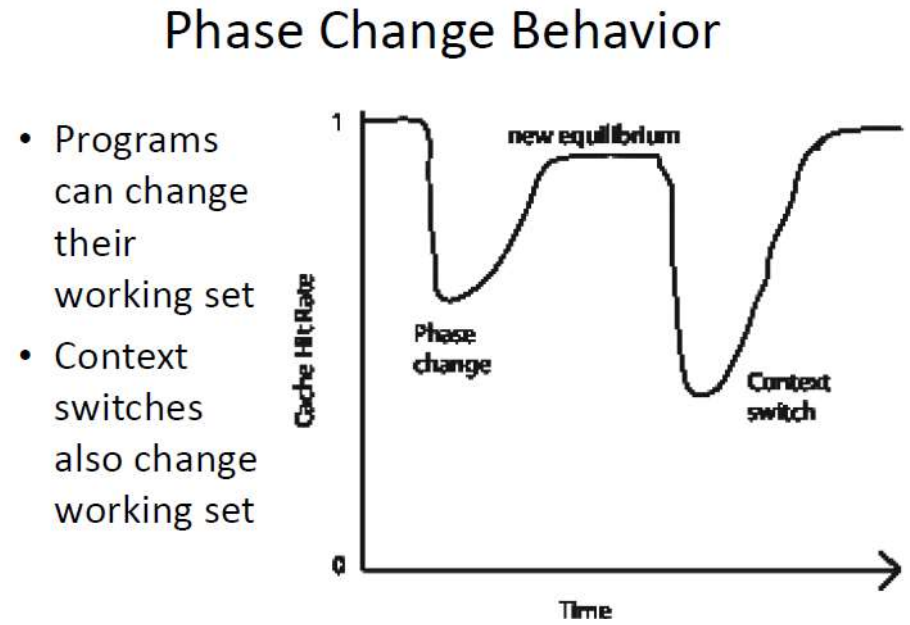
Informalmente il Working set di un processo (e da qui in poi parlo di processi perchè tanto la memoria è allocata ai processi e non ai thread, poiché i thread riferiscono la memoria, ma sono i processi che materialmente lavorano) non è altro che l'insieme delle pagine di memoria virtuale che quel processo sta utilizzando in una certa fase della sua esistenza. Siccome quel processo le sta utilizzando, per migliorare le prestazioni di quel processo è importante che quelle pagine siano caricate in memoria principale. Quindi il nostro obiettivo è quello di individuare qual è il working set di ogni processo presente nel sistema e di allocare ad ogni processo una quantità di pagine fisiche sufficienti per poter allocare tutti i working set di tutti i processi attivi.

Se andiamo a vedere in funzione della dimensione della cache qual è la probabilità di trovare la pagina in memoria principale, in funzione della cache size, ovviamente più grande è la memoria fisica e più è probabile che le informazioni siano reperibili direttamente in memoria principale, perchè una volta caricate restano lì, abbiamo spazio per tenerle. Quindi se il processo le riutilizza anche dopo tanto tempo è più facile ritrovarle. Quando invece la dimensione della cache (nel nostro caso della memoria principale) si riduce, la probabilità di trovare informazioni di pagine richieste caricate in memoria si abbassano.
La cosa interessante (che probabilmente vi è stata detta ad Architetture) è che il passaggio tra un hitrate (una probabilità di trovare i dati) elevata (prossima a 1) oppure un hitrate pari a 0 (la probabilità di non trovare i dati) è molto repentino. Quindi c'è una dimensione della memoria per cui se superiamo quella dimensione riusciamo a raggiungere un hitrate pari a 1 e quindi elevate prestazioni; se andiamo al di sotto, con alta probabilità, ci ritroviamo in un sistema in cui l'hitrate è praticamente nullo (quindi la probabilità di trovare dei dati in memoria principale) e quindi il sistema va in thrashing.

La dimensione della memoria fisica diventa importante più per questo motivo, perchè se è sufficiente riusciamo a tenere in memoria fisica il working set dei processi e di conseguenza andiamo ad operare in questa zona con un hitrate molto elevato e quindi con poca necessità di andare a caricare nuove pagine dal disco. Se invece una memoria fisica non è sufficiente, è troppo piccola ci troviamo facilmente in questa situazione per cui il sistema va in thrashing.

La memoria fisica ovviamente non è un parametro sul quale possiamo agire, il computer si compra con una certa memoria fisica e quella resta, magari la potete espandere però una volta espanso quella memoria è quella che è. In realtà il SO non può agire sulla dimensione della memoria fisica, per lui quello è un parametro (quindi non lo può modificare), ma il SO può agire invece su altri elementi, per esempio sulle dimensioni globali dei working set dei processi presenti. Il SO non può imporre ad un processo di avere un working set, se il processo vuole usare certe pagine e quelle fanno parte del suo working set il SO non ci può fare niente. Però il SO può decidere se caricare o meno un processo oppure se sospendere o meno un processo e quindi a passare i suoi requisiti. Tenete presente che questa problematica di thrashing è un fenomeno che si verifica quando globalmente tutti i working set dei processi sono troppo grandi per poter star nella memoria fisica. Quindi quello che può fare il SO è ridurre il numero di processi, praticamente.

Se la vediamo dal punto di vista del singolo processo il working set non è un elemento fisso, costante che non cambia nel tempo, ma in realtà cambia nel tempo. Può capitare che il processo entri in un loop, all'interno di questo ciclo va a utilizzare certe strutture dati, utilizza sempre la stessa porzione di codice e di conseguenza finchè resta intrappolato in questo ciclo continua ad utilizzare sempre le stesse pagine. Quando poi esce dal ciclo e magari inizia ad eseguire un 'altra funzione da qualche altra parte inizierà ad utilizzare un'altra porzione di codice, magari altre strutture dati. Il risultato è che l'insieme delle pagine che fanno parte di un working set di un processo non è una cosa statica, ma cambia nel tempo, in funzione di quello che il processo fa. Vediamo cosa succede a livello di sistema.

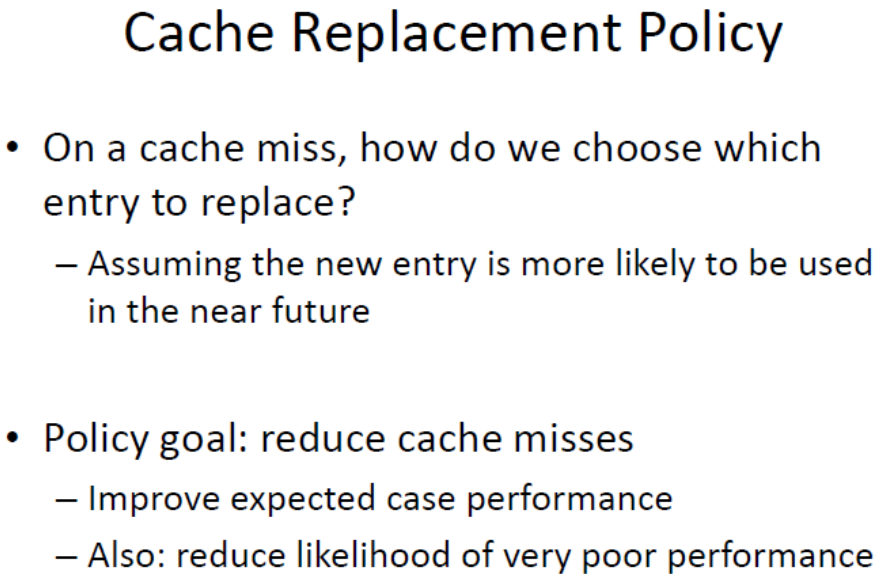


Supponete, ad un certo istante di tempo di aver scoperto qual è il working set del processo, sapete quali sono le pagine che sto usando e gliele caricate tutte in memoria principale: fintanto che il working set resta quello tutte le pagine del working set restano caricate in memoria principale e il processo va avanti spedito perchè non genera fault di pagina. Quando il processo cambia comportamento entra in un'altra zona di codice: il suo comportamento può cambiare (può cambiare il suo working set) e a questo punto mi trovo in una situazione nella quale le pagine che avevo caricato precedentemente non mi servono più, ma me ne servono delle altre. Quindi se andate a vedere dal punto di vista del pthread (nella nostra terminologia questo è il numero di fault di pagina) quello che succede è che ad un certo punto, per esempio in una fase iniziale abbiamo raggiunto un equilibrio abbiamo caricato il working set, quindi l'hitrate è 1 o prossimo a 1. Abbiamo caricato tutte le pagine, quindi non ci sono page fault. Il processo cambia comportamento, inizia molto repentinamente ad utilizzare le nuove pagine e a non usare più le vecchie e questo comporta un elevato numero di page fault, un missrate più alto (più basso hitrate) sulla memoria, perchè non ci sono le nuove pagine che il processo deve riferire. Però progressivamente tutti questi page fault causano un caricamento delle pagine richieste e quindi progressivamente la situazione tende di nuovo a ristabilizzarsi, riportando un punto di equilibrio, nel quale il nuovo working set è stato caricato in memoria principale e nuovamente abbiamo pochissimi page fault.

Dal punto di vista del processo la cosa evolve in questa maniera: quindi si passa da zone in cui l'hitrate è 1 (quindi non ci sono quasi page fault) a zone in cui invece ce ne sono molti, poi ci si riprende e si continua in questa maniera. Che cosa succede invece quando c'è una commutazione di contesto? Il SO decide di sostituire il processo in esecuzione e di mettercene un altro. In realtà sappiamo che il SO schedula i thread e non i processi, però dal punto di vista della gestione della memoria è la stessa cosa: se mette in esecuzione un thread dello stesso processo dal punto di vista della memoria è sempre lo stesso processo in esecuzione. Se mette in esecuzione un thread di un altro processo, dal punto di vista della gestione della memoria la cosa cambia.

Quindi questa commutazione di contesto ha un effetto sulla gestione della memoria: questo perchè mettere in esecuzione un thread di un altro processo comporta che quest'altro processo riferirà la sua

memoria, userà il suo working set. Il punto è: il working set del nuovo processo che abbiamo messo in esecuzione è caricato in memoria o no? Può darsi che durante il periodo nel quale questo processo non era in esecuzione (così come i suoi thread) la memoria sia stata liberata per gli altri thread che erano in esecuzione, per gli altri processi. Per cui il working set del processo che adesso andiamo a mettere in esecuzione (abbiamo un contest switch) non trova subito il suo working set disponibile, per cui la commutazione di contesto è un punto delicato. In questo momento potrei avere un alto missrate (quindi un basso hitrate) con elevato numero di fault di pagina in seguito proprio a commutazione di contesto per andare a ricaricare le pagine del working set del processo che adesso viene messo in esecuzione. In tutto questo gioca un ruolo chiave il modo con il quale noi scegliamo le pagine da rimuovere dalla memoria principale.



È evidente che prima o poi delle pagine della memoria principale le devo rimuovere perchè i processi sono tanti, hanno spazi virtuali che possono essere anche molto ingombranti e molto probabilmente tutti quanti non ci staranno in memoria principale. Se io guardo il mio sistema (mi sembra che abbia 4 giga di ram, ma la quantità di processi attualmente in funzione sono uno sproposito, ci saranno almeno una ventina di processi), globalmente questi processi occupano molta più memoria fisica. Quindi se io non rimuovessi pagine dalla memoria principale mi troverei in condizioni di non dover più lavorare: le pagine vanno certamente rimosse.

L'algoritmo con il quale si rimuovono le pagine dalla memoria principale è estremamente importante, perchè se rimuove delle pagine che poi saranno riutilizzate a breve comporterà un ulteriore page fault, un ulteriore overhead per il caricamento della nuova pagina. Vorremmo fare in modo di rimuovere delle pagine quando siamo obbligati, ma far sì che le pagine che vengono rimosse dalla memoria principali siano pagine che non ci servono per un periodo di tempo abbastanza lungo. L'obiettivo degli algoritmi di sostituzione è quello di ridurre il numero di cache miss, quindi il numero di page fault. Non è quindi un caso il fatto che di algoritmi di sostituzione sembra ne siano stati proposti tantissimi per migliorare tutta una serie di aspetti a seconda dei requisiti o anche dell'evoluzione tecnologica dei sistemi.

A Simple Policy

- Random?
  - Replace a random page
- FIFO?
  - Replace the page that has been in the cache (main memory) the longest time
  - What could go wrong?

Due possibili politiche molto semplici sono: sostituzione totalmente casuale, nella quale se siamo fortunati va bene, se siamo sfortunati va male, però è difficilmente controllabile questo tipo di meccanismo, quindi ovviamente non viene utilizzato. Un'idea potrebbe utilizzare una regola di tipo FIFO: andiamo a rimuovere le pagine che sono in memoria principale da più tempo. L'idea potrebbe essere che se una pagina è in memoria principale da tanto tempo può darsi che nel frattempo il processo che ce l'aveva nel working set nel frattempo abbia cambiato working set e quindi quella pagina non ci sia più. In realtà non è un caso che il FIFO non sia più utilizzato perchè ci sono dei problemi: in particolare,è facile trovare il caso peggiore del FIFO, il caso in cui il FIFO funziona davvero male.

FIFO in Action

Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			

Worst case for FIFO is if program strides through memory that is larger than the main memory

Immaginiamo di avere 4 blocchi fisici in memoria principale e un processo che riferisce 5 pagine dello spazio virtuale, ABCDE. Se utilizziamo la FIFO quello che succede è che il processo va a caricare progressivamente queste 5 pagine del working set carica prima A, poi B, poi C poi D. Quando gli viene richiesto di caricare E, perchè il processo riferisce alla pagina E, il sistema va a rimuovere la pagina che è in memoria da più tempo. Ma così facendo mette E al posto di A: così facendo il prossimo riferimento, che sarà ad A, causerà di nuovo un altro page fault. Quindi di fatto tutti questi riferimenti fatti in sequenza vanno a causare un page fault, perchè di volta in volta col FIFO andiamo rimuovere la pagina peggiore, proprio quella che verrà utilizzata subito dopo.

In effetti il fatto che una pagina sia caricata da molto tempo non vuol dire che sia fuori dal working set. Può darsi che quel codice, per come è strutturato, continui a riutilizzare quella pagina perchè magari lì c'è una struttura dati che è essenziale per tutta quanta la sua durata di esecuzione. Quindi sarà stata caricata all'inizio dei tempi ma deve restare sempre caricata perchè resta parte del working set.

L'algoritmo ideale dovrebbe cercare di individuare la pagina che non verrà utilizzata in futuro o che se in futuro verrà utilizzata ciò avverrà dopo il tempo più lungo, perchè in questo modo il tempo del page fault lo sposto più avanti possibile.

MIN, LRU, LFU

- MIN (ideal, optimal)
  - Replace the page that will not be used for the longest time into the future
  - Optimality proof based on exchange: if evict a page used sooner, that will trigger an earlier page fault (=cache miss)
- Least Recently Used (LRU)
  - Replace the page that has not been used for the longest time in the past
  - Approximation of MIN
- Least Frequently Used (LFU)
  - for Zipf distributions
  - Replace the page used the least often (in the recent past)
  - Suitable for web proxies, not meaningful for page replacement algorithms

Questo tipo di algoritmo che è in realtà un algoritmo ideale, non è reale, perchè il SO non ha una sfera di cristallo dover può guardare, leggere il futuro e capire quale sarà la pagina che non verrà riferita più recentemente nel futuro. D'altra parte ha senso pensare a quale è la sequenza ideale, ottima perchè questo è un buon banco di prova: se io devo testare il funzionamento di un algoritmo di sostituzione lo posso confrontare con l'algoritmo ideale. Non potrà fare meglio, ma se ci si avvicina vuol dire che è un buon algoritmo di sostituzione. In realtà è possibile, dato un processo, sapere quali sono le pagine ideali da rimuovere: lo faccio eseguire una volta, mi segno quali sono i riferimenti che fa; se lo faccio eseguire una seconda volta li riferirà una memoria nella stessa maniera e poi a quel punto posso prevedere quali sono i riferimenti futuri e potrei implementare l'algoritmo ideale. Ovviamente questo va bene in una simulazione, nell'uso reale questo non lo si può fare. L'algoritmo ideale non è utilizzabile nella realtà, ma è utilizzabile in una simulazione.

Se prendo un programma posso sapere quale potrebbe essere idealmente la sequenza migliore per avere meno fault di pagina possibili. Concretamente visto che questo non si può utilizzare si utilizzano degli algoritmi che sono delle sue approssimazioni. Una approssimazione, che funziona piuttosto bene, è il Least Recent Used.

MIN, LRU, LFU

- MIN (ideal, optimal)
  - Replace the page that will not be used for the longest time into the future
  - Optimality proof based on exchange: if evict a page used sooner, that will trigger an earlier page fault (=cache miss)
- Least Recently Used (LRU)
  - Replace the page that has not been used for the longest time in the past
  - Approximation of MIN
- Least Frequently Used (LFU)
  - for Zipf distributions
  - Replace the page used the least often (in the recent past)
  - Suitable for web proxies, not meaningful for page replacement algorithms

Esso si basa pesantemente sul principio di località. L'idea dell'LRU è questa: se io non posso prevedere il futuro guardo al passato, immaginando che il passato tenda a riprodursi nel futuro. Se il processo ha caratteristiche di località evidentemente continuerà a comportarsi come si è comportato nell'ultimo periodo. Quindi guardare all'ultimo periodo è utile per capire come si comporterà dopo. L'idea dell'LRU è quella di, se necessario, andare a sostituire la pagina che non è stata usata per il periodo di tempo più lungo nel passato. Quindi, anzichè come nel caso del FIFO andare a rimuovere la pagina caricata da più tempo, vado a rimuovere la pagina che non è stata usata da più tempo. Se la pagina non è stata usata da tanto tempo è facile che sia fuori da working set del processo.

Ovviamente potrei sbagliarmi con LRU, questo è possibile. Ed è possibile costruire dei casi ad hoc nei quali l'LRU funziona male. D'altra parte nel comportamento medio dei processi l'LRU funziona piuttosto bene ed è forse la migliore approssimazione che si conosca dell'algoritmo ideale. In realtà per poter implementare l'LRU abbiamo bisogno per ogni pagina di ogni processo di sapere qual è l'istante di ultimo riferimento, sulla base di un tempo assoluto (oppure si può fare sulla base di un tempo relativo). Comunque c'è bisogno di sapere il tempo di ultimo riferimento: questa informazione chi la crea? Chi la mantiene e dove viene conservata? Ogniqualvolta che il processore riferisce una pagina, idealmente, dovrei andare a scrivere nel descrittore della pagina l'istante di riferimento. Tale istante, tra l'altro, deve essere un tempo con una risoluzione buona, con un'informazione non piccola da conservare.

In realtà l'LRU non viene utilizzato dai sistemi attuali, non so neanche se in passato è stato usato in sistemi significativi perlomeno, perchè è sempre stato ritenuto molto complesso da realizzare (anche con i sistemi attuali, poi è anche uno spreco), certamente è molto complicato pretendere dall'MMU di andare a conservare l'istante di ultimo riferimento di una pagina. E nella tabella delle pagine va a pesare normalmente questo dato perchè come minimo sono parecchi byte che bisogna aggiungere ad ogni descrittore. Quindi è oggettivamente molto costoso. Per questo motivi gli algoritmi più utilizzati in realtà si basano su approssimazioni dell'LRU. Quindi l'LRU è un'approssimazione dell'algoritmo ideale.

Gli algoritmi che si utilizzano concretamente sono algoritmi approssimati rispetto all'LRU.
APERTA PARENTESI: Troverete una parte nei lucidi che in realtà non ho intenzione più di fare perchè mi sembra abbastanza inutile: tra le quali rientrano la descrizione della Zipf, la descrizione del Least Frequently Used (questo non è un algoritmo che in realtà ha molto senso se applicato al caso della paginazione). Questo tipo di algoritmi che si basano sulla frequenza dell'utilizzo vanno bene per situazione di popolarità, quindi per la caching delle pagine web per esempio. Però non sono certamente adatti nel SO per gestire la paginazione. Quindi questa parte la potete eliminare, come anche tutta la parte che riguarda l'organizzazione delle cache e la distribuzione zipf (guardatevela da soli, non ho intenzione di spiegarvela). CHIUSA PARENTESI;

Ritorniamo all'LRU. Come vi dicevo l'LRU è un'approssimazione dell'algoritmo ideale. Se noi lo confrontassimo con l'algoritmo ideale cosa succede?

LRU/MIN for Sequential Scan

							LRU								
Reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A				E				D				C		
2		B				A				E				D	
3			C				B				A				E
4				D				C				B			
							MIN								
1	A					+					+			+	
2		B					+					+	C		
3			C					+	D					+	
4				D	E					+					+

Per esempio stesso scan sequenziale visto prima per il FIFO, quello era un caso particolarmente svantaggioso anche per l'LRU. Avendo 4 pagine e riferendo le pagine virtuali ABCDE in sequenza l'LRU in questo caso particolare si comporta esattamente come il FIFO, non ha alternative. Quando si arriva ad aver caricato D abbiamo saturato la memoria principale, andiamo a riferire E e la pagina non utilizzata più a lungo è la pagina A. Quindi in questo caso particolare l'LRU si comporta esattamente come il FIFO e ogni riferimento ad ogni pagina nuova va a causare un page fault. L'algoritmo ideale cosa avrebbe fatto?

Guardando il futuro e sapendo che subito dopo E viene riferita A, poi B, poi C poi D, l'algoritmo ideale saprebbe che non conviene caricare E al posto di A perchè in realtà il riferimento futuro sarà sulla pagina D, non su A. Quindi lascia A in memoria e va a sostituire D con E. In questa maniera i riferimenti successivi ad A, B e C non causano page fault perchè AB e C sono ancora in memoria. Dopodichè abbiamo il riferimento a D che non è più caricata perchè è stata rimossa. Quindi questo obbligatoriamente causa un page fault e la pagina D viene stavolta caricata al posto di C perchè quando siamo in questo punto la pagina che verrà riferita nel tempo più remoto nel futuro è proprio C. Quindi qui si sostituisce C con D; a questo punto il riferimento a A, a E e B non causa page fault; poi abbiamo un altro page fault causato da C che viene caricato al posto di B e via dicendo.

Quindi vedete che in realtà un algoritmo ideale migliora parecchio le prestazioni, perchè andiamo a risparmiare in questo esempio 9 page fault (che sono davvero tanti). L'LRU non è il MIN, non è un algoritmo ideale, anche l'LRU ha dei casi svantaggiosi, sfavorevoli. Poter avere informazioni, quindi cercare di ottimizzare l'algoritmo di sostituzione produce enormi vantaggi, perchè permette di risparmiare davvero tanti fault di pagina. D'altra parte se l'LRU non è ideale come l'algoritmo MIN funziona, diciamo, mediamente bene. Quindi se anzichè prendere il caso peggiore prendessimo il caso medio l'LRU funzionerebbe decentemente. In questo caso c'è un esempio di come funzionano l'LRU, il FIFO in un altro esempio.

LRU															
Reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+				+			+	
2		B			+								+		
3				C					E			+			
4						D		+		+					C
FIFO															
1	A		+				+		E						
2		B			+						A			+	
3				C								+	B		
4						D		+		+					C
MIN															
1	A		+				+				+			+	
2		B			+								+		C
3				C					E			+			
4						D		+		+					

La sequenza di riferimento delle pagine è quella che trovate scritta su in alto. Con l'LRU vengono caricate prima A, poi B, poi viene riferita nuovamente A (questo non è un page fault perchè è già stata caricata). Viene riferita C, quindi non è presente, deve essere caricata questo page fault. Poi B è già presente in memoria, va tutto bene, viene riferita D, questo è un page fault perchè non è presente. Da questo punto in poi si riferisce A e siamo esattamente a metà. Fino a questo punto sono tutti riferimenti che sono stati fatti con una memoria che non era ancora satura, per cui sono page fault causati dal fatto che stiamo andando a caricare il working set del processo in memoria principale. Di conseguenza LRU, FIFO e MIN fino a questo punto si comportano esattamente nella stessa maniera; sono i page fault strettamente necessari per caricare il working set del processo in memoria principale. Da questo momento in poi però quando abbiamo saturato la memoria principale, ogni ulteriore page fault dovrà causare una sostituzione e quindi la differenza tra LRU, MIN e FIFO la vedete da questo punto in poi.

Il primo riferimento ad A è caricato in memoria e quindi non è un problema. Il secondo riferimento è a D è caricato in memoria e quindi nuovamente non c'è page fault. Il problema nasce quando si riferisce E. E non è caricato in memoria, fa parte del working set, va caricata e per permettere il caricamento di E bisogna prima rimuovere una pagina. E qui si vede la differenza: il MIN va a sostituire C con E, l'LRU va a sostituire C con E (in questo caso, perchè C è stato quello usato meno tempo), il FIFO invece va a sostituire A con E, perchè A è quella che è stata caricata da più tempo. Qui c'è la prima differenza. Il risultato è che, da questo punto in poi in realtà, il MIN e l'LRU si comportano alla stessa maniera e fanno pochissimi page fault fino al riferimento di C, mentre invece per effetto di questa scelta del FIFO abbiamo 3 page fault per andare a ricaricare A B e C.

(Pausa)

Come vi dicevo l'algoritmo LRU è un algoritmo che è in realtà un po' complesso da implementare, più che altro perchè richiede effettivamente la memorizzazione del tempo associato ad ogni pagina e l'aggiornamento di questa informazione temporale è una cosa abbastanza (starnuto). Quindi per questo motivo si preferiscono utilizzare degli algoritmi che offrono delle approssimazioni dell'LRU. Uno di questi, tra l'altro è stato anche ampiamente utilizzato, è l'algoritmo dell'orologio (si chiama così per il modo con il quale è implementato, ma il suo vero nome sarebbe second chance).

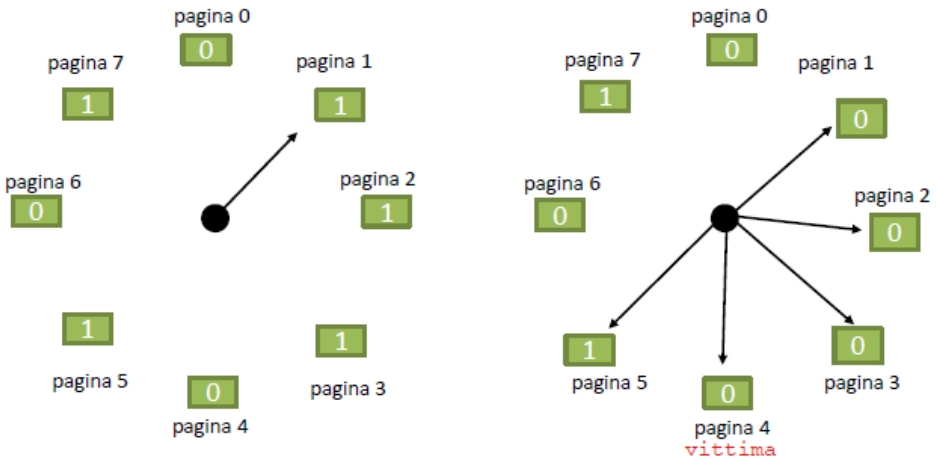
Clock Algorithm: Estimating LRU

- Periodically, sweep through all pages
- If page is unused, reclaim
- If page is used, mark as unused

A differenza dell'LRU questo algoritmo per funzionare ha bisogno soltanto di 2 bit nei descrittori delle pagine, quelli che abbiamo visto prima, il bit di uso e il bit di modifica. In particolare quello critico è il bit di uso, che abbiamo visto si può implementare abbastanza facilmente con l'aiuto della MMU, dell'hardware e poi viene gestito dall'algoritmo Second Chance.

L'idea è quella di andare periodicamente a scandire tutte le pagine: è un algoritmo che normalmente viene implementato in un Demone, quindi in un processo che sta in background e periodicamente viene messo in esecuzione. Questo processo analizza le tabelle delle pagine e per ogni singolo processo scandisce tutte le pagine: se ho una pagina che non è stata utilizzata questo vuol dire che la pagina non è stata utilizzata dall'ultimo istante di tempo in cui questo algoritmo è stato eseguito, quindi tra un'esecuzione e l'altra di questo algoritmo la pagina non è stata riferita. Quindi non è stata utilizzata da un po' di tempo e quindi si può rimuovere. Altrimenti se la pagina è stata utilizzata in questo intervallo di tempo viene marcata come utilizzata e il bit di uso della tabella delle pagine viene messo a 0. Si chiama algoritmo Clock per il modo con il quale spesso viene rappresentato.

Page replacement "*second-chance*" (*clock algorithm*)



a) all'inizio dell'algoritmo

b) Come procede l'algoritmo

Immaginate di avere tutte le pagine messe in una sorta di lista circolare e di questa lista circolare tenete il puntatore, che sarebbe la lancetta dell'orologio. Supponiamo che la lancetta dell'orologio, ad un certo punto, dopo l'ultima esecuzione dell'algoritmo di sostituzione punti per qualche motivo alla pagina 1. In questo istante (nel quale l'algoritmo parte in esecuzione abbiamo che le pagine del processo avranno il loro bit di uso) alcuni saranno ad 1, altri saranno a 0.

L'algoritmo procede così: scandisce tutte le pagine a partire da quella puntata attualmente; ogniqualvolta trova una pagina che ha il bit a 1, lo mette a 0 e manda avanti la lancetta e prosegue in questa maniera, andando ad azzerare tutti i bit di riferimento, di uso delle pagine, fintanto che non trova una pagina in cui il bit di uso è 0. In questo caso, siccome questa pagina non è stata utilizzata almeno dall'ultimo intervento di

questo algoritmo (ricordatevi che questo algoritmo viene eseguito periodicamente con un periodo fissato) allora riteniamo che essa sia fuori da working set, poiché non è stata usata da un certo tempo (tuttavia non siamo sicuri che questa sia quella che non è stata usata da più tempo, quindi non è detto che questo sia lo stesso calcolo fatto dalla MMU). In ogni caso questa la riteniamo buona per essere sostituita quindi è quella che individuiamo come vittima.

Fatto questo la lancetta viene spostata alla pagina successiva e di questa pagina vittima viene dato il comando di scaricamento e su questa pagina fisica (pagina 4), sulla quale questa pagina è caricata, potrà essere caricata la pagina richiesta. Se c'è un page fault questa sarà stata liberata e quindi poi farà (???). Se la pagina era modificata andava salvata su disco, se la pagina non era modificata può essere marcata (???). (Chiaro no?)

Questo algoritmo di sostituzione, come vedete, è molto semplice, niente di complicato. Intanto perchè si chiama second chance? In questo caso queste pagine avevano il bit di uso ad 1, quindi non vengono rimosse (come avrebbe fatto il FIFO), ma viene azzerato il bit di uso, ma rimangono lì presenti. Per cui se al prossimo ciclo non verranno riferite a quel punto potranno essere rimosse e da qui appunto la seconda possibilità. Normalmente una pagina viene rimossa dopo due giri di orologio. Il primo giro, resetta l'indicatore e nel secondo giro la si rimuove.

Quest'idea del second chance può essere utilizzata in diverse varianti, per esempio con la tecnica dell'ennesima chance, quindi invece di usare indicatori binari usate indicatori interi e contate il numero di giri che la lancetta ha fatto su quella pagina.

Nth Chance: Not Recently Used

- Periodically, sweep through all page frames
- If page hasn't been used in any of the past N sweeps, reclaim
- If page is used, mark as unused and set as active in current sweep

Questo tipo di algoritmi vengono detti anche Not Recently Used perchè il loro obiettivo non è di rimuovere la pagina che è stata usata meno di recente, ma di rimuovere una tra le pagine che sono state usate meno di recente. Quindi non è l'LRU, ma è una sua buona approssimazione. Questi algoritmi possono essere utilizzati in 2 modi: come algoritmi globali o come algoritmi locali.

Local and global page replacement

- Global algorithms:
  - The page selected for removal is selected among all pages in main memory
  - Irrespective of the owner
  - "past distance" of a pages defined based on a global time (absolute clock)
  - May result in trashing of slow processes
- Local algorithms
  - The page selected for removal belongs to the process that caused the page fault
  - Fair with "slow" processes about trashing
  - Past distance of a page based on relative time
    - The time a process has spent in running state

Il punto è che quando vado a fare quel lavoro di analisi delle pagine quali pagine effettivamente sto guardando? Detto in altri termini, quando eseguo l'algoritmo su queste pagine, queste pagine quali sono e a chi appartengono?
Se queste sono tutte le pagine di tutti i processi caricati in memoria principale, questo è la forma globale dell'algoritmo. Quindi l'algoritmo applicato ad una sua versione globale, nel caso di fault di pagina può andare a rimuovere una pagina di un qualsiasi processo, anche di un processo che non ha causato il page fault. Se invece utilizzano la sua versione locale allora queste pagine sono tutte pagine di uno stesso processo. Quindi questo algoritmo dovrà essere applicato per tutti i processi separatamente.

Uno Studente: Però quanno un processo entra in esecuzione dopo un cambio di contesto, le pagine sue non ce l'ha angora, devo portarcele dentro ...

Dopo un cambio di contesto no. Il vostro collega dice, quando c'è un cambio di contesto se il processo viene messo in esecuzione dopo il cambio di contesto le sue pagine non sono in memoria principale

Lo Studente: Se io lo applico solo alle pagine mie, non ce le ho angora perchè sono appena entrato in … devono fare angora un fault.

No, non è così. Quando il processo viene caricato, viene messo in esecuzione la prima volta dovrà caricare le sue pagine. Quando poi viene deschedulato, c'è una commutazione di contesto, le sue pagine restano in memoria principale, non vengono tolte. Può capitare che alcune pagine gli vengano tolte se si utilizza l'algoritmo di sostituzione forma globale. Per cui per effetto delle richieste dei fault di pagina degli altri processi, può capitare che le pagine del nostro processo vengano rimosse, ma questo non è detto. Quindi se l'algoritmo è in forma globale, quando poi questo processo ritorna in esecuzione probabilmente molte pagine nel suo working set sono ancora in memoria principale ed è possibile che qualche pagina sia stata scaricata. Non possiamo sapere con certezza. Se è stato fatto non è stato fatto con intenzione di rimuoverla. Se invece l'algoritmo di sostituzione è locale, allora quando il processo viene rimesso in esecuzione tutte le sue pagine sono certamente in memoria principale, perchè l'algoritmo di sostituzione locale per rimuovere va a rimuovere le pagine dello stesso processo che ha causato il page fault.

Lo Studente: Però dovrei avere abbastanza memoria per tutte le pagine dei processi.

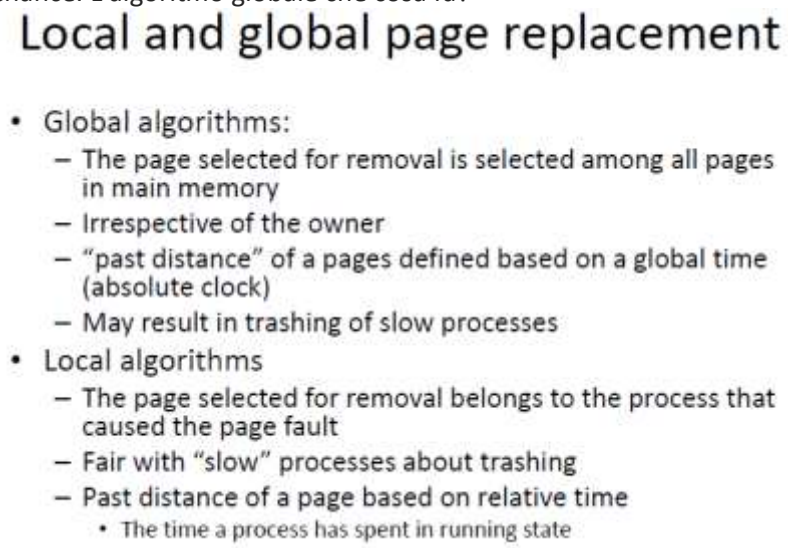
Quello sempre e comunque. Deve sempre avere memoria fisica per contenere il working set di tutti i processi presenti nel sistema: se non ne ha abbastanza di memoria fisica funziona comunque ma va in thrashing.

Lo Studente: Perché uso l'algoritmo globale?

Non è un problema di algoritmo di sostituzione: è semplicemente il problema che nella memoria principale non ci stanno i working set. Voi il thrashing lo potete sperimentare allegramente (sul vostro computer). Quando ero studente per un esame dovevamo fare un progetto. Questo progetto si svolgeva su un computer unico utilizzato su tanti terminali. Al tempo i terminali a disposizione erano 12. Erano gli anni ‘90 quindi vi potete immaginare i computer che roba erano. Questi terminali erano terminali grafici ed erano già tra i primi terminali grafici: immaginatevi la grafica era abbastanza pesante da gestire all'epoca. dovevamo fare un progetto, dovevamo fare una certa cosa per questo esame e ovviamente gli studenti lavoravano tutti contemporaneamente su tutti i terminali. Il sistema era regolarmente in thrashing. Quindi si arrivava a muovere il mouse o premere qualcosa per tracciare, spostare una finestra e il tracciamento della finestra lo potevate seguire in tempo reale, durava quanto una partita di calcio. Voi avevate le righe che venivano tracciate una ad una, con molta flemma da parte del terminale.

Questa cosa la potete tranquillamente riprodurre. La colpa non è dell'algoritmo di sostituzione, è che c'è poca memoria principale. E se vi volete divertire a fare thrashing dovete scrivere un programma che alloca tanta memoria, un bell'array bello grande e utilizza questo array utilizzando locazioni di memoria casuali. Quindi andate a scrivere o a leggere in locazioni casuali. Ovviamente succede che in queste condizioni il vostro programma non ha località, quindi la cache del processore inizia a funzionare male; la cache data dal gestore della memoria inizia a funzionare male perchè sparate un po' dappertutto, caricate una cinquantina di questi processi (dipende da quanto è capiente la vostra memoria) e poi vedere che succede. Fatelo per esercizio per la prossima volta (è istruttivo).

Torniamo al second chance. L'algoritmo globale che cosa fa?



Se il mio processo ha fatto un fault di pagina non c'è spazio in memoria principale, vado a rimuovere dalla memoria principale una pagina scelta tra tutti i processi indifferentemente. Quindi, per esempio, il mio processo ha causato un page fault e vado a rimuovere una pagina del suo processo. Facendo in questo modo però devo stare attento perchè i processi hanno un loro tempo di avanzamento, hanno un loro clock interno proprio. Non ce l'hanno materialmente, però concretamente il tempo del processo avanza quando lui è in esecuzione. L'algoritmo di sostituzione in forma globale deve analizzare invece le pagine sulla base di un tempo globale, perchè il tempo deve essere comparabile per tutti i processi. Anche se usiamo il second chance, per cui il tempo materialmente non c'è, di fatto è come se ci fosse, perchè il passare del tempo è segnato dall'esecuzione dell'algoritmo second chance che va a scandire tutte le pagine e va a vedere se dall'ultima esecuzione sono state azzerati o meno i bit di uso. Di fatto siamo in condizioni di tempo globale e potremmo andare a rimuovere una pagina di un qualsiasi processo.

Con l'algoritmo di sostituzione globale abbiamo dei vantaggi, perchè non mi devo preoccupare di allocare pagine fisiche ad un processo, sarà lui a richiederle; quando mi chiede di caricare una pagina virtuale io la carico, senza starmi a preoccupare tanto. Il problema, però, è che se ci sono dei processi più lenti, per esempio quelli che hanno priorità più bassa (CPU Bound) che vengono eseguiti meno frequentemente, questi processi rischiano di vedere le loro pagine sostituite a vantaggio dei processi invece che sono più attivi, che hanno priorità maggiore. Quindi con gli algoritmi globali la gestione è più semplice, però i processi che hanno priorità inferiore rischiano maggiormente di andare in thrashing.

Viceversa, con gli algoritmi locali: se il mio processo causa un page fault benissimo, vado a rimuovere una pagina tra quelle del mio processo. In questo modo il page fault fatto da un processo non va ad incidere sugli altri processi, però questo causa un problema, perchè per il mio processo devo decidere quante pagine fisiche allocargli. D'altra parte l'algoritmo di sostituzione applicato in forma locale è più equo nei confronti dei processi lenti. Nel caso degli algoritmi locali, tra l'altro, questo non è nè un vantaggio nè uno svantaggio, è semplicemente una caratteristica intrinseca, il tempo di riferimento delle pagine non è più quello globale, non è più basato su una sorta di clock globale (implicito o esplicito), ma è basato sulla velocità di avanzamento interno del processo, quindi su un tempo di riferimento locale, che ha valore per il singolo processo, non ha valore per tutti.

Uno studente: C'è un motivo?

Viene naturalmente così, non è che ci sia un motivo. Se quel processo è stato messo in esecuzione ora per un secondo, poi sta sospeso per un'ora e poi viene messo in esecuzione per un altro secondo, la pagina che ha riferito ora non è riferita da un'ora e passa, non è stata riferita, in realtà, da due secondi, perchè quel processo è stato in esecuzione 2 secondi, non un'ora. Quindi naturalmente deve essere così, devo fare riferimento ad un tempo..

Lo Studente: Io pensavo nell'ottica di fare il confronto tra una pagina un'altra, per vedere quale togliere.

Proprio nell'ottica di fare il confronto tra i tempi di riferimento tra una pagina e l'altra, devo tenere presente che questo tempo di riferimento è un tempo di riferimento relativo al processo, non è un tempo di riferimento assoluto. Se fosse assoluto potrei incorrere in problemi di valutazione. E' facile fare un esempio, non mi costa niente, è che poi potrebbe confondere le idee a qualcun'altro.

Local vs global page replacement

a)

	T
A0	10
A1	7
A2	5
B0	9
B1	6
C0	12
C1	4
C2	3

b)

	T
A0	10
A1	7
A2	5
B0	9
B1	6
C0	12
C1	4
C2	3

c)

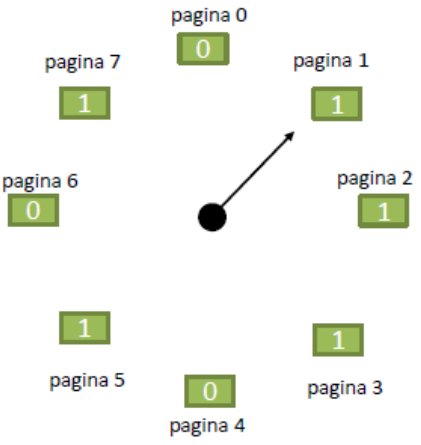
	T
A0	10
A1	7
A2	5
B0	9
B1	6
C0	12
C1	4
C2	3

T: time of last reference

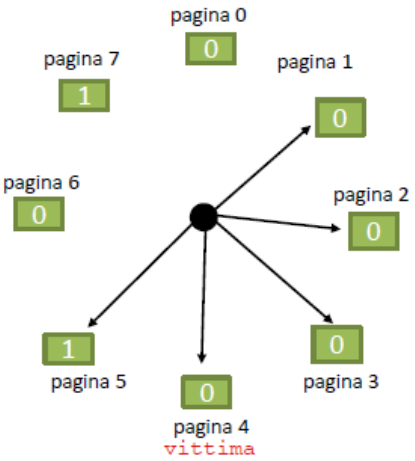
a) Initial configuration
b) Page replacement with a local polity (WS, LRU, sec. chance)
c) Page replacement with a global policy (LRU, sec. Chance)

Se noi abbiamo questo sistema che ha caricato in memoria principale queste pagine, le pagine 0-1-2 del processo A, 0-1 del processo B, 0, 1, 2, del processo C e quelli nella colonna di destra sono gli istanti di ultimo riferimento. Se io utilizzo una politica locale, quando il processo A causa un page fault, vado a rimuovere la pagina utilizzata meno recentemente tra quelle del processo A e quindi vado a rimuovere la pagina 2, che è stata riferita all'istante 5 (quello meno recente). Sto ipotizzando una LRU, però, diciamo, il second chance avrebbe un comportamento che approssima questo. Se invece utilizzo un algoritmo nella forma globale, a quel punto il page fault causato da A va a produrre la rimozione della pagina 2 di C, perchè quella è riferita meno di recente tra tutte quante. Riprendiamo un attimo questo esempio.

Page replacement *"second-chance" (clock algorithm)*



a) all'inizio dell'algoritmo



b) Come procede l'algoritmo

C'è un'altra questione su dove effettivamente agisce il working set: abbiamo detto, se è globale agisce su tutte le pagine di tutti i processi, se è locale agisce sulle pagine del singolo processo. Concretamente significa che un processo ha le sue pagine descritte nella tabella delle pagine: quindi per eseguire l'algoritmo working set dovrei andare a scorrere tutta la tabella delle pagine. D'altra parte la tabella delle

pagine, per molti processi, in larga misura è inutilizzata. Molti descrittori non hanno senso, non corrispondono a pagine caricate. Quindi andare ad eseguire l'algoritmo di questo tipo sulla tabella delle pagine non è conveniente, si rischia di andare a perdere tempo e andare ad analizzare pagine che non sono mai state allocate. Quindi in linea di principio si può far nella tabella delle pagine, concretamente no: lo si fa soltanto sulle pagine allocate.

Come si gestisce questa cosa? Le pagine allocate possono essere collegate fra loro in una lista circolare: quindi senza modificare i descrittori di pagina, lasciandoli dove sono, i descrittori di pagina possono avere un ulteriore campo che implementa una lista circolare di pagine. In alcuni sistemi lo si osserva. Oppure, in alternativa, questo algoritmo lo eseguo sulla core map. Se io utilizzo l'algoritmo in forma globale tutte le pagine caricate le trovo nella core map e quindi lo posso eseguire direttamente sulla core map, che è più comodo. (La prossima lezione faremo un esercizio). La core map è una tabella delle pagine inversa, che associa ad ogni blocco fisico la pagina virtuale che c'è caricata, con il processo al quale appartiene. Quindi se io voglio vedere le pagine caricate di tutti i processi le trovo nella core map, non ho bisogno di andare a guardare la tabella delle pagine. Questo algoritmo LRU o Second Chance era utilizzato un po' di tempo fa nei sistemi operativi di tipo Unix o anche altri sistemi. I sistemi più recenti utilizzano tecniche un po' più sofisticate, in particolare la tecnica del working set.

Working set algorithm

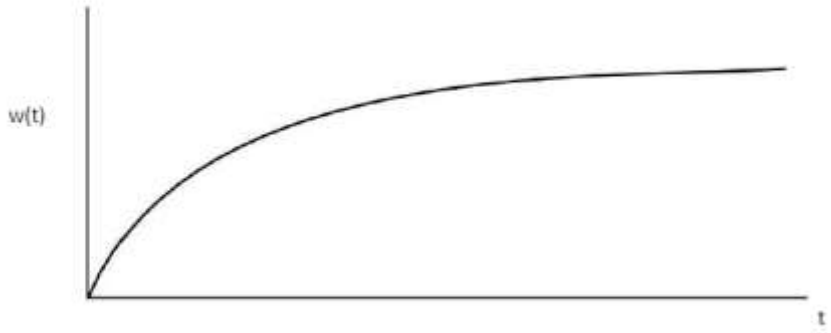
- Keep in memory the *working set* of a process:
  - the pages that the process is currently using
- Working set defined as:
  - The set of pages referred in the last k memory accesses
    - difficult to implement
  - The set of pages referred in the last period T
    - Usually implemented in this way, using the «use» bit

Il working set a differenza del second chance, che non si preoccupa in realtà di definire un working set, ma va a rimuovere una pagina a patto che sia riferita a un tempo sufficientemente remoto. L'algoritmo working set, invece, cerca di individuare con maggiore precisione il working set dei processi e va a rimuovere le pagine che stanno fuori dal working set. (Frame mancante) …son sempre gli stessi. E' sempre il bit di uso, in più c'è un indicatore che viene gestito dal working set e che è un tempo di ultimo riferimento nel blocco. Per poter definire meglio il working set intanto il concetto di working set stesso va definito.

Io prima vi ho detto "sono le pagine che un processo sta utilizzando in una certa fase della sua esecuzione", ma questa definizione non è utilizzabile completamente. Se noi vogliamo creare un algoritmo di working set che cerca di approssimare con una certa precisione il working set dei processi dobbiamo dare una definizione rigorosa, precisa di quello che è il working set. Di definizione di working set ce ne sono un po': una è quella di definirla sulla base degli accessi in memoria.

Quindi, per esempio, potrebbe essere definito come l'insieme di pagine che sono state riferite negli ultimi k accessi in memoria, dove k è un parametro. Il problema è che preso in questa forma la definizione del working set non è molto comoda, perchè dovrei contare gli accessi in memoria e questo è scomodo. In realtà si preferisce una definizione di questo tipo: "l'insieme delle pagine riferite nell'ultimo periodo t", dove t stavolta è un periodo fissato e un parametro di sistema sul quale si può giostrare per modificare le prestazioni, per fare un tuning delle prestazioni del sistema. Normalmente la definizione adottata di algoritmi working set è questa. Per implementare questo concetto utilizzano il bit di uso. Domanda: come varia il working set nel tempo? Qui vi mostro in funzione del tempo la dimensione del working set.

Working set



- working set: The set of pages referred in the last k memory accesses
- w(t) is the size of the working set as function of time

Questa è la dimensione del working set, non è l'insieme del working set. Questo grafico vi dice che, a parte una fase di inizializzazione nella parte iniziale dei processi, poi il working set tende a stabilizzarsi, non come insieme, ma come dimensione. Quindi normalmente i processi, una volta che arrivano a regime, tendono ad avere un working set di dimensioni stabili. Due processi differenti avranno working set di dimensioni differenti, però lo stesso processo dato a regime tenderà a mantenere un working set di dimensione stabile e per questo motivo il grafico della dimensione del working set rispetto al tempo tende a crescere secondo questa legge. Ovviamente poi nel tempo, se il working set cambia può darsi che oscilli, che raggiunga diversi stati di equilibrio, però questi restano stabili.

Working set algorithm

- Each process has a number of physical pages reserved to upload its working set
  - WS replacement policy is inherently local
- Resident set:
  - is the actual set of virtual pages in main memory
  - some of them may be out of the working set
  - $\neq$ working set

Gli algoritmi che implementano il concetto di working set sono algoritmi prettamente locali, perchè vanno ad individuare l'insieme delle pagine del working set del singolo processo e di conseguenza operano andando a rimuovere per quel processo le pagine che stanno fuori dal working set, quindi l'algoritmo working set è intrinsecamente locale. Siccome è intrinsecamente locale bisogna che io ad ogni processo assegni un certo numero di pagine fisiche per poter contenere il suo working set. Negli algoritmi locali, come vi dicevo prima, se un processo fa page fault io rimuovo una pagina di quel processo per dargliene un'altra. Questo vuol dire che il numero di pagine di quel processo resta costante nel tempo e questo vuol dire che io gli devo aver allocato ad un certo punto quelle pagine.

Quindi ho il problema con gli algoritmi locali di stabilire quante sono le pagine fisiche che assegno a quel processo. Questo problema esiste anche nel working set: ad ogni processo assegno un certo numero di pagine fisiche (poi vedremo dopo come e quante). Su queste pagine fisiche l'algoritmo cerca di mantenere caricate le pagine del working set. Le pagine del working set sono pagine virtuali. In un dato istante di tempo quel processo avrà un insieme di pagine residenti in memoria principale, ma non è detto che queste pagine siano tutte del working set: alcune pagine potrebbero essere fuori da working set e quindi andranno

rimosse e altre pagine saranno dentro il working set. Quindi in buona sostanza, io al processo do un certo numero di pagine fisiche: per quanto possibile su queste pagine fisiche cerco di tenere pagine virtuali nel working set, inevitabilmente alcune pagine staranno fuori. Collettivamente tutte queste pagine formano un insieme residente, l'insieme delle pagine caricate in memoria principale per quel processo.

Working set algorithm

- WS defined as the set of pages referred in the last period P
  - P is a parameter of the algorithm
- For each page:
  - R bit (called “referred” or “use” bit) indicates whether the page had been referred in the last time tick
  - Keep an approximation of the time of last reference to the page
    - At the end of each time tick resets bit R for each page and updates the approximation of time of last reference
- At page fault:
  - For each page checks bit R and time of last reference
    - If R=1: set last reference time to current time and resets R
  - The pages referred in the last period P are in the working set and (if possible) are not removed

Il working set è definito come l'insieme delle pagine che sono state riferite nell'ultimo periodo di riferimento P, dove P è questo parametro. Cosa dobbiamo fare per ottenere questa informazione? Mi servono un po' di informazioni da conservare nelle tabelle delle pagine e in particolare utilizzo 2 informazioni: una è il bit di uso o bit di riferimento, prende nomi diversi a seconda di delle architetture (è esattamente il bit di uso che abbiamo visto prima) e poi un'altra informazione che riguarda il tempo di ultimo riferimento della pagina.

Fate attenzione: questo tempo di ultimo riferimento non è lo stesso tempo di ultimo riferimento usato nella LRU. Nell'LRU, il tempo di ultimo riferimento doveva essere calcolato e scritto dall'MMU apposta nella tabella delle pagine. Nel caso del working set in realtà mi basta un'approssimazione del tempo di ultimo riferimento. Quindi questo tempo in realtà è un'approssimazione che conservata nella tabella delle pagine è gestita dal SO interamente a software.

In particolare l'algoritmo working set viene eseguito con periodo fisso, a cicli. Ad ogni ciclo va a resettare i bit di riferimento e se trova il bit di riferito a 1 va ad aggiornare l'approssimazione del tempo di ultimo riferimento. Quando c'è un page fault che cosa fa? Per ogni pagina testa se il bit R è uguale a 1, setta il tempo di ultimo riferimento al tempo attuale e resetta il bit R. Di conseguenza tutte le pagine che sono state analizzate hanno un tempo di ultimo riferimento che dipende dalla frequenza di esecuzione dell'algoritmo. Questi tempi vengono confrontati con il periodo di riferimento P. Tutte le pagine che hanno un'anzianità maggiore di P, quindi che sono state riferite prima del tempo P vengono dichiarate fuori dal working set e possono essere rimosse. Tutte le pagine che invece sono state riferite dopo l'ultimo periodo P si assume che siano dentro il working set e quindi non vengono rimosse. Vediamo un esempio di come viene eseguito.

Working set algorithm

Current virtual time: 2204

Page table

Time of last reference	Bit R	...
2084	1	
2003	0	
1980	1	
1213	0	
2014	1	
2020	1	
1604	0	

For each page:

if(R==1)
time of last reference =
current virtual time; R=0

if (R==0 and age>P)
removes the page

---

if (age<=P for each page)
removes the page with smaller
time of last reference

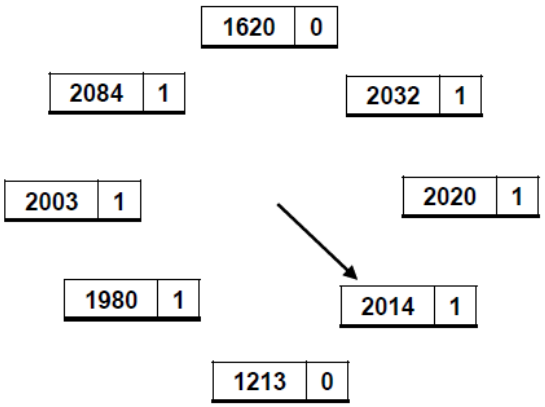
Age: current virtual time – time of last reference

Prendiamo un processo che ha queste pagine e di queste pagine ad un certo punto il SO ha inizializzato il tempo di ultimo riferimento e l'hardware ha settato i bit di uso. In questo istante viene messo in esecuzione l'algoritmo del working set: questo è l'istante 2204, che è il tempo virtuale per questo processo.
Tutti questi tempi di riferimento possono anche essere maggiori di 2204, perchè questo è l'istante attuale del processo che stiamo considerando. L'algoritmo va ad analizzare ogni pagina, ogni descrittore: se trova il bit R pari 0, l'istante di ultimo riferimento viene lasciato così. Se trova il bit di riferimento pari ad 1, l'istante di ultimo riferimento viene aggiornato al tempo attuale (2204) e il bit di riferimento viene resettato.

Fatta questa scansione, tutte le pagine riferite dall'ultima esecuzione erano tempo di ultimo riferimento pari al tempo attuale. Tutte le altre pagine hanno un tempo precedente. A questo punto, dato P, vado a calcolare il tempo attuale di P. Supponiamo che P sia 100, vado a calcolare 2204 - 100: trovo 2104. Quello che dico è che tutte le pagine che hanno un tempo di ultimo riferimento maggiore di 2104 sono nel working set; tutte le pagine che hanno un tempo di ultimo riferimento minore di 2104 sono fuori dal working set.

WSClock (working set + clock)

Current virtual time : 2204



- Considers only the pages in main memory
  - More efficient than scanning the page table
- Pages in a circular list
- At page fault looks for a page out of the WS
  - Better if not "dirty"
  - If selects a dirty page, the page is saved before its actual removal

Questo algoritmo può essere eseguito anche lui nella forma ad orologio: se lo vediamo in tale forma (questi che vediamo sono esattamente gli stessi numeri di prima) lo eseguiamo al tempo attuale 2204. L'orologio ad un certo punto punta a questa pagina ed essa è riferita, quindi questo bit va a 0 e qui ci va messo il tempo attuale, si sposta in avanti la lancetta; questa pagina non è riferita, quindi il suo tempo resta quello che è, si sposta in avanti la lancetta e si va a questa. Quello che succede dopo la prima esecuzione è che

avrà individuato questa pagina come candidata vittima per la rimozione e invece questa pagina avrà resettato il bit R e avrà aggiornato il tempo di ultimo riferimento. Ecco che succede se andiamo avanti.

Questa pagina è portuale(?), il bit riferita è a 0; essa, che era candidata come vittima, è stata rimossa. Al suo posto è stata caricata una nuova pagina: siccome è stata caricata ora il suo tempo di ultimo riferimento è il tempo attuale e il bit di riferimento è a 1 perchè è stata appena riferita.

Uno Studente: C'è una possibilità che siano tutte riferite?

Sì. Se sono riferite tutte quante abbiamo un caso interessante. In questo caso cosa succede?

Uno studente: Succede che il processo è a pieno regime? Nel senso che il suo working set è dato dalla dimensione massima e quindi sta usando tutte le pagine insieme?

Esattamente. In questa ipotesi abbiamo che tutte le pagine attualmente caricate in memoria per quel processo sono tutte nel working set. Il processo ha causato un page fault, quindi deve caricare un'ulteriore pagina nel working set: siamo in una situazione nella quale il working set è più grande dell'insieme residente. Se non facciamo nulla e ci limitiamo a scaricare una pagina per caricare quella appena riferita rischiamo di portare questo processo in thrashing. Questo perchè siccome tutte le pagine sono nel working set probabilmente riferirà la pagina appena scaricata nel futuro e quindi causerà un altro page fault, che causerà lo scaricamento di un'altra pagina del working set. Quindi se si trova in questa situazione la risposta da dare non è "scarico una pagina a caso", ma dovrebbe essere "allargo l'insieme residente".

Uno Studente: La dimensione dell'insieme residente la scelgo prima?

Ve lo racconto adesso. Utilizzo un algoritmo di sostituzione locale, per esempio il working set: questa è una scelta del SO uguale per tutti i processi. Viene creato un processo, in qualche maniera stabilisco che quel processo ha bisogno di dieci pagine fisiche e gliele assegno. Queste dieci pagine fisiche ce le ha a disposizione, però inizialmente non ha nessuna pagina virtuale caricata.

Quel processo passa in esecuzione ha iniziato a usare i page fault. I primi dieci non danno problemi, i primi dieci page fault causano il caricamento delle prime dieci pagine. A questo punto il processo va avanti e causa un altro page fault (prima o poi); questo page fault causa l'esecuzione dell'algoritmo working set, che applica l'algoritmo di sostituzione, quindi inizia a sostituire pagine che stanno fuori dal working set e si va avanti così. Il processo evolve, lavora con le pagine che stanno nel working set, alcune entrano altre ne escono. Questo processo continua a lavorare con 10 pagine fisiche, il sistema continua a tenere il suo working set all'interno di queste pagine.

Ora, in realtà, i sistemi non funzionano in questa maniera per due motivi: il primo è ovvio, è molto difficile sapere che quel processo ha bisogno di 10 pagine anzichè di 20; il secondo problema è che quel processo, durante la sua esistenza può cambiare dimensione del working set. Guardate che la cosa non è banale. Io potrei prendere un processo che occupa pochi K di codice e questo processo potrebbe avere un working set molto grande, perchè magari va ad allocare dinamicamente memoria, la utilizza in una maniera per cui tiene impegnate tutte le pagine e questo significa che un processo anche piccolo, un codice eseguibile piccolo può avere un working set molto grande. Quindi non posso guardare la dimensione del codice per sapere quanto è grande il processo. Viceversa un processo, un codice molto grande potrebbe sempre lavorare con poche pagine e quindi potrebbe sempre avere un working set molto piccolo. E' molto difficile guardando il codice capire quante pagine devo allocare, ci sono problemi anche di decidibilità, non posso avere un algoritmo che mi decide lui di quanta memoria ha bisogno quel processo, non è decidibile.

Secondo problema. Il processo può cambiare comportamento: lui può partire che in un dato istante di tempo gli servono 10 pagine fisiche e poi, dopo un po', gliene servono 3 e va avanti così per chissà quanto tempo, poi gliene servono 50 e via dicendo, dipende da come evolve. Quindi la dimensione dell'insieme residente deve essere dinamica nel tempo: devo poter avere un sistema che mi permetta di far crescere o restringere la dimensione dell'insieme residente. Questa è una questione. Un'altra questione è che quando si verifica il fault di pagina se non ci sono pagine libere fisiche devo eseguire l'algoritmo di sostituzione, eventualmente andare a salvare su disco la pagina che voglio rimuovere, andare a caricare la pagina

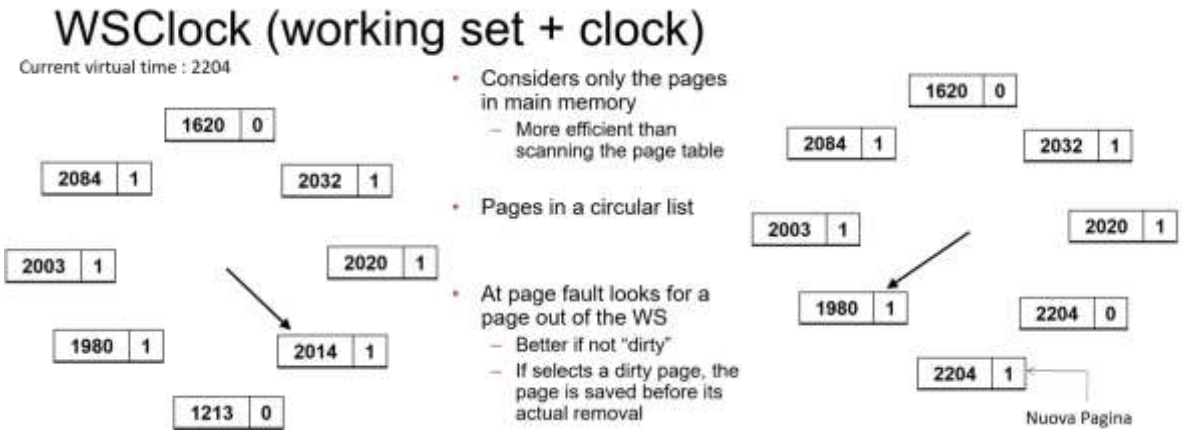
richiesta e poi, a quel punto, posso ripristinare il thread che ha causato il page fault. Questa cosa fa perdere troppo tempo.

Di nuovo: i sistemi non lavorano in questa maniera. Cercano di, invece, avere sempre un pool di pagine fisiche libere per soddisfare le richieste. Quindi in realtà quello che vedremo alla prossima lezione è questo, sono questi due aspetti: uno, come si fa ad avere un insieme residente che varia dinamicamente e secondo, l'algoritmo di sostituzione non viene eseguito su richiesta quando c'è il page fault, ma viene eseguito preventivamente per mantenere un numero di pagine fisiche libere in maniera tale da poter gestire rapidamente i page fault. Tutto questo non cambia l'algoritmo di sostituzione, io posso avere il working set o posso avere l'LRU o posso avere il second chance. Quello che cambia è il modo con il quale lo utilizzo nel sistema.

Working set algorithm

- WS defined as the set of pages referred in the last period P
  - P is a parameter of the algorithm
- For each page:
  - R bit (called "referred" or "use" bit) indicates whether the page had been referred in the last time tick
  - Keep an approximation of the time of last reference to the page
    - At the end of each time tick resets bit R for each page and updates the approximation of time of last reference
- At page fault:
  - For each page checks bit R and time of last reference
    - If R=1: set last reference time to current time and resets R
  - The pages referred in the last period P are in the working set and (if possible) are not removed

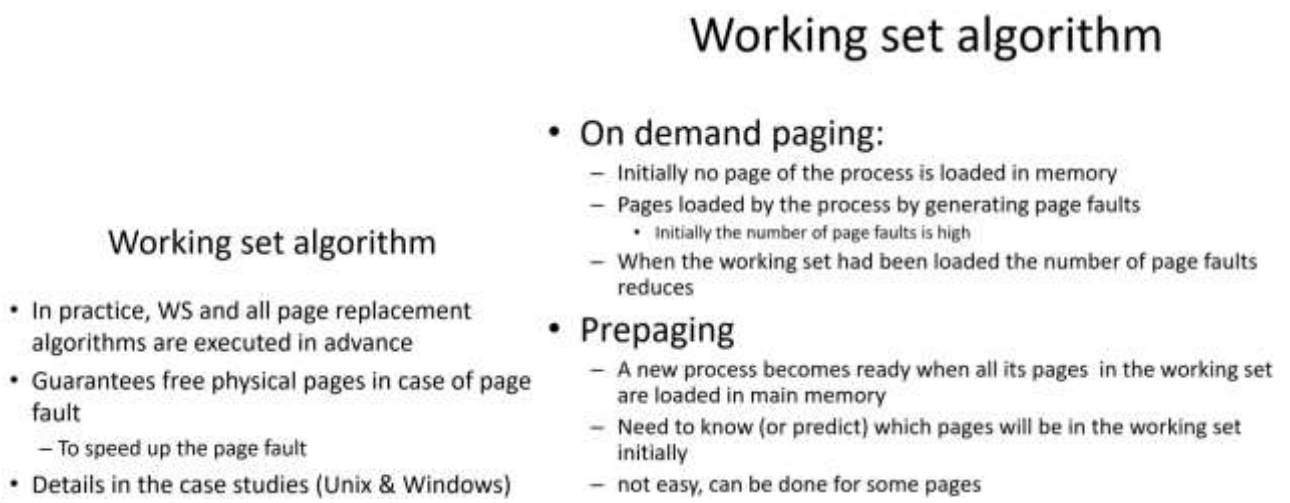
Ripartiamo dall'algoritmo Working Set che stavamo vedendo nella lezione scorsa, ve lo riprendo brevemente: il Working Set utilizza due informazioni: la prima informazione è un bit nella tabella delle pagine(il bit di uso) che viene settato dalla MMU e resettato periodicamente dall'algoritmo Working Set; il bit di uso viene settato dall'hardware ogni qualvolta il processore riferisce una certa pagina, che sia in lettura o in scrittura; quando viene messo in esecuzione l'algoritmo working set.



Questo scorre tutte le pagine caricate in memoria principale per un certo processo, nella figura ve lo faccio vedere nella sua versione clock, verifica se il bit riferito è a 1 oppure a 0, se il bit è a 1, azzera il bit e aggiorna l'istante di ultimo riferimento al tempo attuale, se il bit è a 0, allora controlla il tempo di ultimo riferimento memorizzato nel descrittore della pagina, se questo ultimo tempo di riferimento della pagina precede il tempo attuale meno un valore di riferimento, quindi la pagina è stata riferita oltre un certo tempo rispetto al tempo attuale, allora può essere rimossa.

Quindi in questo caso, quando analizza la prima pagina, questo è il tempo di riferimento che è stato impostato l'ultima volta che l'algoritmo del Working Set è stato eseguito, trova il bit riferito ad 1, quindi azzera e aggiorna il tempo di ultimo riferimento a questa pagina al tempo corrente. Questa pagina che invece non era riferita, era un tempo di riferimento abbastanza vecchio quindi che supera un certo limite, viene rimossa, al suo posto viene caricata la nuova pagina che ha tempo di riferimento fissato al tempo

attuale e viene marcata come riferita. Quindi le informazioni son due: un bit di riferimento e un tempo (una marca temporale, che in realtà non è una marca temporale molto accurata, è approssimata, è gestita dall'algoritmo Working Set e viene aggiornata ogni qualvolta l'algoritmo Working Set esamina una pagina). In realtà questi algoritmi non vengono eseguiti reattivamente quando si verifica un fault di pagina, ma vengono eseguiti preventivamente per far sì che in memoria principale ci siano sempre un po' di blocchi fisici liberi e questo permette di gestire più rapidamente i fault di pagina. In altri termini, quando si verifica un fault di pagina, il processo che lo causa viene penalizzato perché dobbiamo sospenderlo fintantoché la pagina che lui ha riferito non verrà caricata. Quindi per evitare un ulteriore rallentamento dei processi che causano fault di pagina, è bene avere pagine fisiche libere in maniera tale da eseguire il più rapidamente possibile la gestione del fault di pagina. Questo significa che, per far questo, l'algoritmo Working Set o qualsiasi altro algoritmo di sostituzione delle pagine deve essere eseguito in anticipo su base periodica, quindi non su base reattiva.



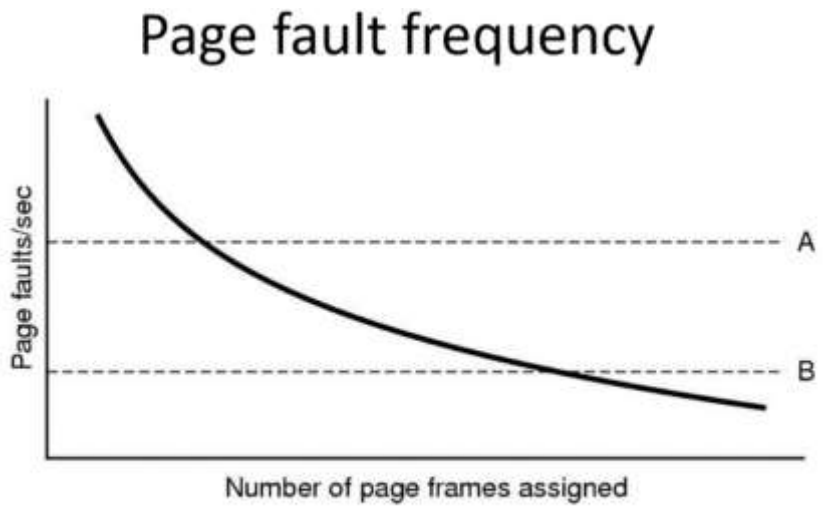
Vediamo alcuni esempi di come viene fatto nei casi di studio di UNIX e di Windows che adottano questa strategia. Un altro aspetto della paginazione dinamica riguarda il modo col quale il processo viene messo in esecuzione: un modo è di metterlo in esecuzione prima ancora di aver caricato qualsiasi pagina del suo Working Set, noi in effetti possiamo mettere un processo in esecuzione anche se la sua memoria non è stata affatto allocata, non c'è ancora nessuna pagina caricata per quel processo, questo vuol dire che non appena quel processo passerà in esecuzione, già alla prima istruzione genererà un fault di pagina e a furia di generare fault di pagina, questo processo andrà a caricare naturalmente il suo Working Set, dopo una fase transitoria entrerà a regime e andrà alla massima velocità. Questa tecnica è la tecnica dell'On Demand paging, e ha senso perché è molto difficile, quando viene messo un processo in esecuzione, sapere qual è il suo Working Set. L'alternativa invece è quella di adottare un modello con Prepaging: nel modello con Prepaging invece cerchiamo di prevedere qual è il Working Set del processo e a questo punto carichiamo le pagine del suo Working Set e lo mandiamo in esecuzione in maniera tale che già dall'inizio alla sua esecuzione non causi troppi fault di pagina. Ora in realtà dove sta la verità? La verità sta un po' nel mezzo tra queste due tecniche, è molto difficile, quando un processo viene messo in esecuzione, sapere quali sono le pagine del suo Working Set, questo resta molto difficile, però alcune pagine sappiamo che vanno caricate, per esempio la pagina che contiene la parte iniziale del Main, del codice main, o la pagina che contiene il record di attivazione sullo stack legato al main e magari con un po' di sforzo il compilatore può individuare altre cose che possono essere caricate subito quando il processo viene messo in esecuzione. D'altra parte è difficile sapere qual è tutto il suo Working Set, una volta che viene messo in esecuzione può prendere un'esecuzione difficilmente prevedibile e chissà di quali pagine avrà bisogno. Quindi questo vi dice che non dobbiamo preoccuparci troppo, in realtà. Quindi possiamo caricare, del processo, le pagine che sappiamo che gli son necessarie all'inizio e poi lasciamo che la natura faccia il suo corso e che lui con i fault di pagina provochi il caricamento delle pagine che gli servono. Se il processo però a un certo punto

della sua esistenza viene congelato e magari viene spostato in memoria secondaria per liberare spazio in memoria, quando viene poi ricaricato in memoria, ripartirà da dove era stato congelato il suo stato e a quel punto però il suo Working Set lo conosciamo bene e quindi possiamo caricarlo tutto quanto, in quel caso.

Working set algorithm

- What should be the number of physical pages allocated to a process?
- Not easy to say in advance
- Page allocation algorithms
  - Static algorithms do not work
  - Dynamic algorithms: Page Fault Frequency (PFF)

Infine resta aperta un'altra questione, anche questa importante, con la paginazione On Demand noi siamo in grado di scoprire nel tempo quali sono le pagine utilizzate dal processo in un dato periodo di tempo, quindi qual è il suo Working Set, ma per scoprire questo ci vuole del tempo. D'altra parte noi non abbiamo solo il problema di caricare le pagine virtuale del processo ma abbiamo bisogno anche di allocargli delle pagine fisiche. Il punto è che l'algoritmo Working Set è un algoritmo locale per cui, l'idea è questa: noi assegniamo a un processo un numero di pagine fisiche, su queste pagine fisiche il processo andrà a caricare le sue pagine virtuali che fanno parte del suo Working Set. Il problema è: quante pagine fisiche gli dobbiamo allocare, perché se gliene allochiamo troppo poche il Working Set potrebbe non starci, se gliene allochiamo troppe abbiamo uno spreco di memoria, abbiamo delle pagine fisiche allocate al processo che lui non userà perché non riuscirà a riempirle, perché il suo Working set è più piccolo. C'è un problema ulteriore: che il comportamento del processo nel tempo cambia, può darsi che in un dato istante il suo Working Set sia formato di tre pagine e tra dieci minuti sia formato da cento pagine, perché ha cambiato comportamento. Quindi quando dobbiamo andare ad allocare le pagine fisiche, i blocchi fisici a un processo, non abbiamo molte informazioni, non sappiamo qual è il suo Working Set e sappiamo che potrà cambiare in maniera arbitraria nel tempo. Quindi decidere in maniera rigida le pagine fisiche da allocare è la strategia sbagliata: i sistemi moderni soprattutto utilizzano un metodo di allocazione dei blocchi fisici dinamico, quindi osservano il comportamento del processo e in funzione del suo comportamento gli allocano più pagine fisiche oppure gliele tolgono. Per far questo ci serve un ulteriore algoritmo, un algoritmo che capisce quante sono le pagine fisiche necessarie a quel processo, che è ovviamente legato al tutto il resto del meccanismo. Un algoritmo che si utilizza è quello del Page Fault Frequency, che in realtà non è un vero e proprio algoritmo, è un'euristica e funziona in questa maniera:



Number of page faults per unit of time, depending on the number of physical pages assigned to the process

Se noi osserviamo il numero di fault di pagina generati da un processo in funzione delle pagine fisiche allocate, vediamo che il comportamento è di questo tipo: se assegniamo molte pagine fisiche, il numero di fault di pagina generati da quel processo in un dato intervallo di tempo tende a scendere, non raggiungerà mai lo zero presumibilmente ma tenderà a zero (a dire il vero se il numero di pagine fisiche è maggiore di quello che al processo mai servirà durante la esecuzione, poi alla fine diventerà zero). Viceversa se il numero di pagine fisiche è assegnato ed è molto piccolo, il numero di page fault che generiamo per unità di tempo, in questo caso al secondo, tende a crescere esponenzialmente. L'idea è che esiste un range delimitato da due soglie, queste due soglie son due parametri del sistema operativo e tararli è un'arte. Comunque da due soglie, per le quali diciamo che: se il numero di fault di pagina eccede la soglia A, allora presumibilmente con elevata probabilità stiamo operando in questa zona, per cui il numero di pagine fisiche assegnate al processo è troppo piccolo e allora dobbiamo aumentare questo Pool, dobbiamo dargli più pagine fisiche. Viceversa se il numero di fault di pagina scende al di sotto della soglia B, vuol dire che stiamo operando in questa area, ovviamente il processo è felice (😊), perché genera pochi fault di pagina, ma se lo guardate dal punto di vista del sistema, il fatto che il processo generi troppo pochi fault di pagina vuol dire che presumibilmente gli abbiamo allocato più memoria di quella che effettivamente a lui serve e quindi presumibilmente lui sta tenendo in memoria principale anche pagine che non sta più utilizzando, che non sono più del Working Set. Questo perché è un problema? È un problema perché la memoria fisica principale ha una dimensione limitata ed è condivisa fra tutti i processi: se un processo usa più memoria di quanta effettivamente gli serve, implicitamente la sta togliendo agli altri, quindi dal punto di vista del sistema, se il numero di fault di pagina scende sotto B, le pagine fisiche gli vengono troppe. Quindi quello che si cerca di fare è di mantenere i processi in un'area di funzionamento compresa tra A e B nel quale il numero di fault di pagina sono quelli fisiologici. Questo non è un algoritmo, è un'euristica, ciò vuol dire che in realtà questo è un comportamento che più o meno sappiamo andare bene per tutti i processi ma ciò non toglie che si possono scrivere processi che non hanno questo comportamento e questi ovviamente funzionano male, possiamo creare dei processi che, o per la natura del problema che stanno affrontando oppure perché li abbiam fatti apposta in quella maniera, vanno a generare un numero di fault di pagina enorme dovuto al fatto che in realtà noi forziamo artificialmente un Working Set molto grande, per esempio se volete mettere in crisi questi sistemi, allocate strutture dati enormi e riferite a caso vari pezzi di queste strutture dati e questo ovviamente come viene visto dal sistema operativo? Viene visto dal sistema operativo come un processo che ha bisogno di un Working Set molto grande, in realtà glielo state forzando voi questo comportamento e questo ovviamente va a causare dei problemi. Ci possono essere delle applicazioni che proprio naturalmente hanno bisogno di un Working Set molto grande e che magari hanno

bisogno di lavorare con le strutture dati in maniera molto sparpagliata, però in casi particolari. Come viene utilizzato l'algoritmo di Page fault frequency?

Page fault frequency

- Dynamically determines the number of physical pages assigned to a process
  - Guarantee that resident set ≥ working set
- When frequency of page faults >> «natural frequency»
  - Increase the size of the resident set
- When frequency of page faults << «natural frequency»
  - Reduces the size of the resident set

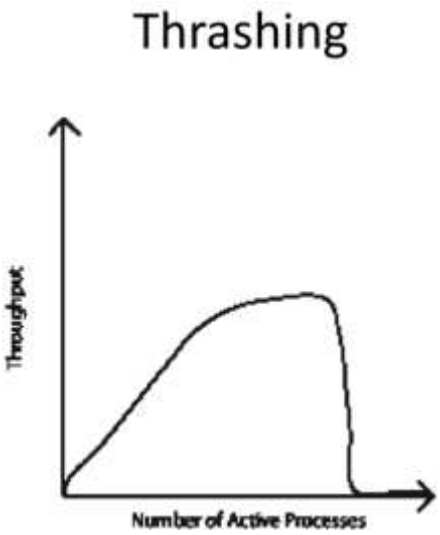
Determina il numero di numero di pagine fisiche da assegnare a un processo facendo in modo che l'insieme residente sia sempre maggiore del Working Set, quindi l'insieme di pagine fisiche allocate a un processo sia sempre maggiore o uguale del Working Set e questo proprio perché se il processo va a lavorare in questa zona, il numero di pagine fisiche viene incrementato. Si cerca di tenere quindi il numero di fault di pagina compreso tra la frequenza naturale (all'interno di un'area di frequenza naturale). Questa area di page fault naturale non esiste, è una scelta arbitraria che fa parte del tuning del sistema operativo.

Question

- What happens to system performance as we increase the number of processes?
  - If the sum of the working sets > physical memory?

A dispetto dell'impegno che mette il sistema operativo per calcolare il numero di pagine fisiche da assegnare a un processo e mantenere il Working Set di ogni processo in memoria, può capitare che a un certo punto l'insieme dei Working Set di tutti i processi ecceda la memoria fisica.

Se siamo in questa situazione evidentemente non possiamo, non ce la facciamo proprio, a mantenere tutti i Working Set di tutti i processi in memoria principale. Il risultato di questo è che i processi entrano selvaggiamente in competizione fra di loro e quello che si produce è il thrashing.



Che cos'è il thrashing? Se andiamo a mappare il numero di processi attivi sulla X e mappiamo il Throughput, quindi seguendo la velocità di avanzamento dei processi sulle Y, vediamo che in una fase iniziale, finché le risorse sono sovrabbondanti il throughput grossomodo cresce linearmente con il numero dei processi finché non si arriva alla saturazione, qui il sistema, il processore ha raggiunto la saturazione quindi non può andare materialmente più veloce di così e quindi il sistema raggiunge questa zona di saturazione lineare, che è ancora una zona buona per lavorare, vuol dire che le richieste fatte al sistema sono leggermente superiori alla sua capacità però lavorando in quella zona riusciamo a sfruttare tutte le risorse del sistema, quindi il sistema è sfruttato al massimo. Il singolo processo probabilmente sarà un po' rallentato ma il sistema è sfruttato al massimo delle sue possibilità, questo se utilizziamo un server va bene. Se però eccediamo e aumentiamo ancora il numero di processi allora arriviamo alla fase in cui l'insieme del Working Set globalmente è maggiore della memoria fisica e improvvisamente le prestazioni collassano, non c'è una performance degradation, non calano gradualmente le prestazioni, ma letteralmente collassano. Cosa deve fare il sistema in questa circostanza? Intanto il sistema ha il primo problema di proteggere sé stesso, quando le prestazioni collassano in questa maniera, anche il sistema ha difficoltà ad essere eseguito, lo stesso sistema operativo non riesce ad essere eseguito, perché sta succedendo che non appena qualcosa torna in esecuzione, si genera di nuovo un fault di pagina che provoca il blocco del processo, impegna il sistema operativo in un'operazione di I/O e a questo punto la coda di Input/Output sul disco è lunghissima, le stesse operazioni del sistema operativo vengono rallentate. I sistemi normalmente cercano di proteggersi un minimo nei confronti di questa situazione di thrashing, riservando un po' di risorse, riservando un po' di memoria fisica per i meccanismi che devono intervenire a salvare la situazione nel caso del thrashing, però ovviamente il problema è grosso, non è solamente un problema di memoria o di processore, è anche un problema di Input/Output sul quale è molto più difficile agire. Quindi in queste situazioni anche il sistema operativo soffre terribilmente (☹). Che cosa si può fare comunque?

Thrashing

- When, from the PFF algorithm, comes out:
  - Some processes require more memory
  - No process requires less memory
- The number of page faults raises up
  - The system almost halts…
- Solution: reduce the number of processes in main memory
  - Reduces competition for memory (reduces the degree of multiprogramming)
  - Swaps out some process on disk

Questa situazione in realtà è irrilevabile perché se l'algoritmo di Page fault frequency ci dice che tutti i processi hanno bisogno di più memoria, quindi tutti i processi stanno generando un numero di fault di pagina che eccede il limite, nessun processo sta generando fault di pagina al di sotto del limite, quindi nessun processo può liberare memoria, questo indice è chiaro che il sistema sta andando in thrashing e l'intervallo temporale nel quale il sistema va in thrashing è brevissimo perché basta davvero poco, poi il collasso è immediato. Se ci rendiamo conto di questa situazione, che cosa possiamo fare? L'unica soluzione possibile è quella di ridurre il numero di processi in memoria principale, il problema è che ci son troppi processi, le risorse non sono sufficienti, quindi dobbiamo ridurre il peso, il carico del sistema, ridurre il carico vuol dire prendere alcuni processi e toglierli dalla memoria principale. In linea di principio toglierli vorrebbe dire ucciderli, terminarli (☹), però in realtà non è questo quello che si fa, c'è un'altra possibilità: li si può congelare, li si può mettere in stato di attesa, e spostare interamente sul disco dove abbiamo memoria a sufficienza. Si fa la cosiddetta operazione di Swap Out, quindi alcuni processi, scelti in base a una qualche politica, vengono bloccati, copiati interamente in memoria e tenuti lì. In questo modo il numero di processi attivi si abbassa, si libera memoria fisica, il sistema si può riprendere. Quando poi il sistema ritorna a regime, magari qualche processo termina o riduce le sue esigenze di memoria, a quel punto possiamo fare lo Swap In, quindi andare a riconsiderare i processi che sono congelati sul disco, in memoria secondaria, li si può riconsiderare e li si può riportare in memoria principale, quindi li si può di nuovo rendere attivi. La scelta dei processi da swappare o da ricaricare viene fatta sulla base di euristiche, son delle cose molto arbitrarie.

Where pages are stored

- Every process segment backed by a file on disk
  - Code segment -> code portion of executable
  - Data, heap, stack segments -> temp files
  - Shared libraries -> code file and temp data file
  - Memory-mapped files -> memory-mapped files
  - When process ends, delete temp files
- Provides the illusion of an infinite amount of memory to programs

Infine, ultima questione, quando vi si dice di portare una pagina sul disco oppure di portare l'intero processo su disco, materialmente che cosa vuol dire? Vuol dire che deve essere presente su disco un'area di scambio che contiene gli spazi virtuali dei processi, quindi in realtà lo spazio virtuale dei processi realmente si trova sul disco e soltanto una parte di questo sta in memoria principale e appunto è la memoria principale che opera da cache nei confronti del disco che tiene lo spazio virtuale dei processi. Però materialmente questa area di swap sul disco in realtà ha diverse forme e diverse configurazioni perché si possono fare diverse ottimizzazioni, per esempio: se consideriamo un processo segmentato o comunque del quale possiamo individuare le pagine di codice per ognuno, le pagine di codice hanno una loro collocazione naturale nel disco che è all'interno dei file eseguibili, quindi se noi dobbiamo scaricare una pagina di codice, non la dobbiamo salvare perché il codice è protetto e non può essere modificato, possiamo riutilizzare la pagina fisica che conservava una pagina virtuale di codice, se poi la dobbiamo ricaricare, la andiamo a prendere dal file eseguibile. I dati, lo heap e i segmenti di stack, questi vengono messi in file temporanei dove questi file temporanei in realtà che cosa sono? Nel caso di UNIX o almeno nel caso di alcune versioni di Linux, questi file temporanei in realtà stanno in una partizione del disco separata oppure potrebbero essere dei file che stanno nella stessa partizione con il sistema e con i dati ma che sono vincolati ad essere utilizzati per mappare gli spazi virtuali dei processi e questa è per esempio la scelta fatta da Windows. Se voi potete osservare una partizione Windows da vicino, scoprirete che c'è un file che non può essere spostato, che è vincolato a stare in una porzione ben precisa del disco e quella è l'area di swap, lì risiedono gli spazi virtuali dei processi. Le librerie condivise, anche queste possono stare invece nei file eseguibili delle librerie, son proprio file eseguibili, sono nel codice compilato. Quando il processo termina, dobbiamo ricordarci che non dobbiamo soltanto liberare la memoria principale, ma dobbiamo anche svuotare le aree di swap che gli son state assegnate. Per i file eseguibili non c'è problema, ma per le zone dell'area di swap destinate ad accogliere dati per stack, queste vanno liberate. Tutto questo serve per offrire l'illusione di una quantità di memoria infinita data ai programmi, in realtà non è proprio infinita la quantità di memoria perché comunque i programmi hanno una quantità di memoria limitata dal massimo indirizzo rappresentabile, quindi posso usare una memoria che è al massimo pari alla quantità di memoria indirizzabile dal processore, che comunque può essere di diversi ordini di grandezza maggiore rispetto alla memoria fisica effettivamente presente.

Gestione della memoria in UNIX

- Da BSD v.3 in poi:
  - segmentazione paginata
  - **memoria virtuale** con paginazione a domanda

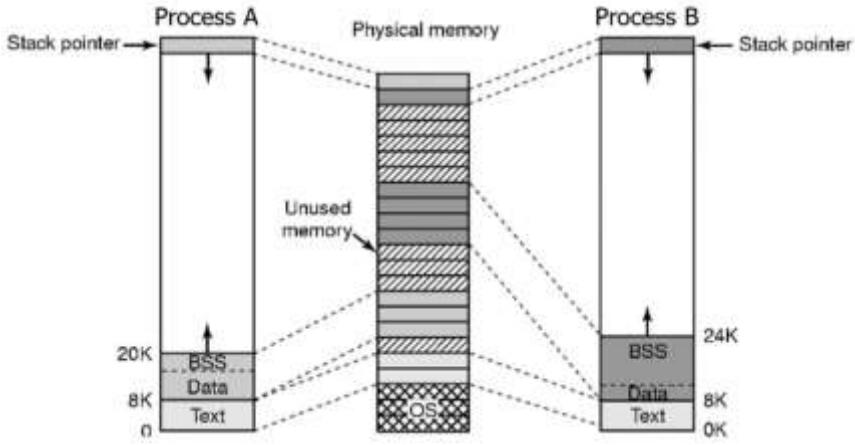
- Paginazione a domanda:
  - ***Core map***: struttura dati interna al kernel che descrive lo stato di allocazione dei blocchi e che viene consultata in caso di page fault.
  - **Sostituzione delle pagine:** algoritmo Second Chance

Alcuni cenni legati a UNIX e a Windows, questi lucidi che vi mostro in questa fase, sono informazioni che in questa forma così estesa le trovate nel Tanenbaum o nel Silberschatz o anche nel "Lancillotto Boari".

UNIX è una costellazione di sistemi che hanno soluzioni interne spesso un po' differenti fra loro e che in comune hanno l'interfaccia e quindi il livello delle chiamate di sistema che rispettano lo standard POSIX.

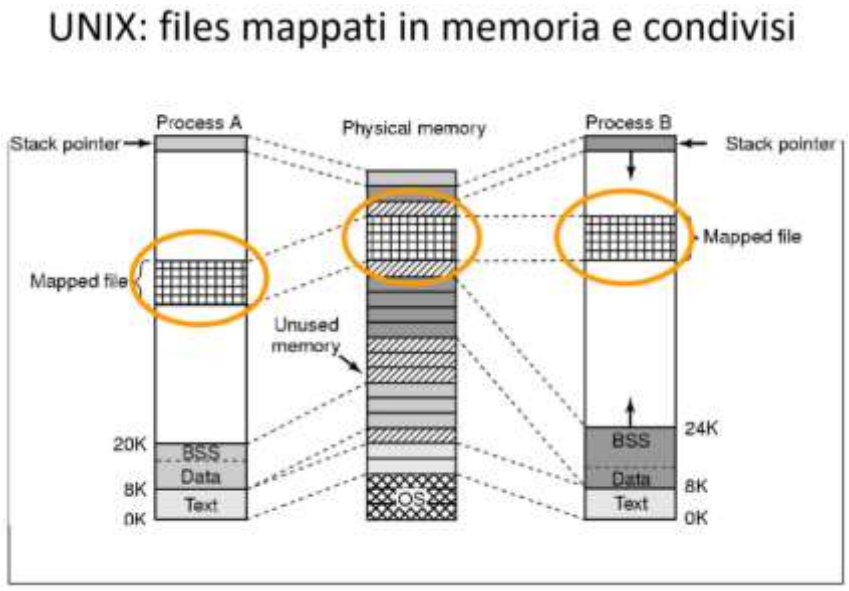
Alcuni sistemi UNIX, in particolare BSD, adottano il metodo della segmentazione paginata con paginazione a domanda, quindi ad alto livello i processi sono segmentati, i segmenti sono paginati, e il meccanismo dinamico è dato dalla paginazione a domanda. Il sistema oltre a memorizzare obbligatoriamente una tabella delle pagine specifica per quel particolare processore sul quale sarà messo in funzione, per scopi suoi interni, rappresenta gli spazi virtuali e lo spazio fisico tramite una struttura che si chiama Core Map, è una struttura dati interna al nucleo che descrive la locazione dei blocchi, quindi la Core Map è una tabella delle pagine inversa, per ogni blocco fisico ho un descrittore, questo descrittore, tra le altre informazioni che contiene, contiene l'indice della pagina virtuale e l'indice del processo che in questo momento sta utilizzando questo blocco fisico. L'algoritmo di sostituzione era l'algoritmo Second Chance, ora nelle versioni moderne non so più quale sia.

UNIX: organizzazione della Memoria



Dal punto di vista dei processi, la memoria è organizzata in questa maniera: lo spazio virtuale alloca codice, dati e heap nella parte bassa, lo stack nella parte alta. Nel caso in cui ci siano più thread all'interno del processo, avremmo più stack caricati in punti intermedi dello spazio virtuale per evitare che gli stack si sovrappongano fra loro. Dal punto di vista della memoria fisica però che cosa succede: se prendiamo diversi processi, per esempio questi due, questi processi hanno i loro segmenti, questi segmenti sono mappati in un certo numero di pagine in memoria principale, alcune pagine mancano al sistema operativo e

chiaramente alcune pagine di memoria restano lì. Per comodità tutte le pagine di un segmento ve le ho messe contigue, però non è necessario, ovviamente la paginazione dinamica permette che le pagine di un segmento siano sparpagliate, altrimenti questo grafico diventa estremamente confuso. Esiste una possibilità in UNIX che è quella di mappare in memoria dei file, e se due processi mappano in memoria lo stesso file, in realtà si ritrovano a condividere memoria, per cui abbiamo anche questa possibilità:



Due processi A e B potrebbero aver mappato in memoria principale lo stesso file, quindi dal punto di vista del processo A il file è mappato in una zona di memoria virtuale, lo stesso file dal punto di vista del processo B è mappato in un'altra zona di memoria virtuale, però concretamente si trova nella stessa zona di memoria fisica perché è condiviso. Non entrerò nel dettaglio, comunque esistono delle chiamate di sistema apposta per mappare file in memoria e questo viene fatto perché in questo modo potete molto agevolmente manipolare un file come se fosse un array, quindi le operazioni sul file sono molto efficienti e quando avete terminato di lavorarci viene salvato sul disco, in questo modo l'accesso al file è più rapido rispetto alle chiamate di sistema Read e Write, ogni accesso risparmiate sicuramente una chiamata di sistema. Oltretutto se mappate lo stesso file, permettete a due processi di condividere memoria dati per cui anche UNIX, nonostante il modello a processi isolati (quindi modello ad ambiente locale), in realtà questa è una deroga che permette un modello ad ambiente globale.

Ora, come viene rappresentata la memoria?

1) Core Map + Tabelle delle pagine

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
SO	SO	SO	SO	SO	SO		A,1 2	B,0 10	C,1 3		B,6 5	C,7 8		C,3 6	A,5 9	C,5 7	B,2 1	A,7 4

Processo, pagina
Tempo ultimo riferimento

Blocco

Pagina	Blocco
0	-
1	7
2	-
3	-
4	-
5	15
6	-
7	18

Processo A

Pagina	Blocco
0	8
1	-
2	17
3	-
4	-
5	-
6	11
7	-

Processo B

Pagina	Blocco
0	-
1	9
2	-
3	14
4	-
5	16
6	-
7	12

Processo C

Tabelle delle pagine indicizzate da indice di pagina

2) Core Map = Tabella delle pagine inversa

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
SO	SO	SO	SO	SO	SO		A,1 2	B,0 10	C,1 3		B,6 5	C,7 8		C,3 6	A,5 9	C,5 7	B,2 1	A,7 4

Processo, pagina
Tempo ultimo riferimento

Blocco

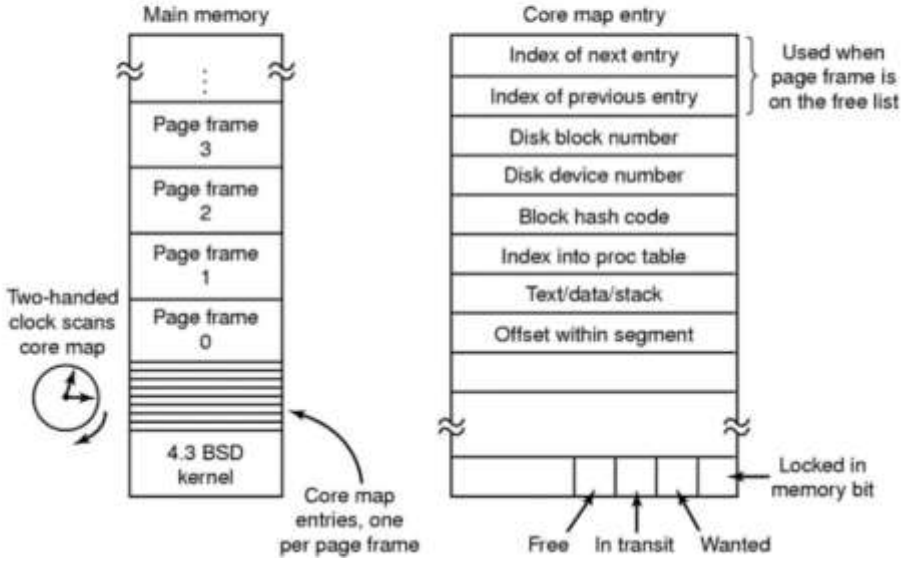
Core Map indicizzate da indice di blocco
--> accesso con funzione hash

In entrambi i casi: vettore circolare dell'algoritmo *Second Chance* realizzato su *Core Map*, con i soli descrittori di blocchi assegnati ai processi

Questa è un'astrazione molto semplificata che spesso utilizzo negli esercizi per proporvi degli scenari e dei casi da risolvere, però possiamo avere in memoria contemporaneamente la tabella delle pagine e allora ogni processo ha la sua tabella delle pagine (vi ricordo che la tabella delle pagine conserva l'associazione <pagina virtuale, blocco fisico>) ed oltre a questo il sistema mantiene la Core Map che è la tabella su in alto che non è altro che un array che per ogni pagina fisica ci dice il nome del processo e la pagina virtuale alla quale è allocata, quindi per esempio il blocco fisico 18 contiene la pagina virtuale 7 del processo A e poi a questa sono associate informazioni legate al tempo di ultimo riferimento per esempio (per l'algoritmo Working Set), quindi per esempio della pagina 7 del processo A, che è l'ultima a destra, sappiamo che l'algoritmo Working Set gli ha impostato come tempo di ultimo riferimento 4.

Paginazione in UNIX BSD

- Usa una tabella delle pagine inversa o *core map*



Nel caso di UNIX BSD, in memoria fisica abbiamo le pagine fisiche che contengono le pagine virtuali dei processi, e da qualche parte in memoria principale abbiamo la Core Map sulla quale si utilizza l'algoritmo tipo Second Chance, in realtà si usa una variante dell'algoritmo Second Change (con due lancette), quindi sarebbe una sorta di Third Chance, e ogni riga della Core Map contiene le informazioni legate alla pagina fisica, quindi due puntatori per legare le pagine fra di loro, da qualche parte il puntatore alla tabella delle pagine del processo, ci sono informazioni nel caso in cui quella pagina qual è la zona in back in store(?), quindi nel disco sul quale quella pagina può essere scaricata, ci serve in qualche parte l'indice del processo, l'offset all'interno del segmento dice qual è la pagina virtuale caricata e via discorrendo, quindi nel descrittore di pagina, nel caso di BSD, ci sono tante informazioni.

Sostituzione delle pagine in UNIX (BSD) - I

Per selezionare pagine da scaricare:
- algoritmo *SecondChance* (globale)
- o la sua variante *Two-handed Clock Algorithm*

Eseguito da *PageDaemon*
PageDaemon:
- utilizza parametri *lotsfree*, *desfree*, *minfree con:*

  lotsfree > *desfree* > *minfree*

- Interviene periodicamente

Come funziona l'algoritmo di sostituzione? In questo caso l'algoritmo di sostituzione è una variante del Second Chance, ma non ci interessa tanto l'algoritmo di sostituzione in sé ora, ci interessa il modo col quale questo algoritmo è utilizzato. Nel caso di UNIX BSD, l'algoritmo di sostituzione è eseguito periodicamente, non reattivamente quando avviene un fault di pagina, quindi l'obiettivo è quello di mantenere un pool di pagine fisiche libere in memoria per gestire rapidamente i fault di pagina, questo vuol dire che l'algoritmo di sostituzione deve essere eseguito da un demone in via preventiva. Un demone non è altro che un processo che sta in background che periodicamente viene messo in esecuzione e va ad eseguire esattamente l'algoritmo di sostituzione, in particolare il PageDaemon. Questo PageDaemon utilizza tre parametri che sono: lotsfree, desfree e minfree. Con questa relazione, lotsfree > desfree > minfree.

Sostituzione delle pagine in UNIX (BSD) - II

Algoritmo del *PageDaemon* (esistono diverse varianti nelle diverse versioni di UNIX)

- **IF** *NumeroBlocchiLiberi* ≥ *lotsfree* ritorna senza scaricare pagine
- **IF** *minfree*≤*NumeroBlocchiLiberi*< *lotsfree* **OR**
  (*NumeroBlocchiLiberi*< *minfree* **AND**
  media(NumeroBlocchiLiberi, ΔT)≥ *desfree*)
  scarica pagine fino ad ottenere
  NumeroBlocchiLiberi= lotsfree+ k, con k≥0
- **IF** *NumeroBlocchiLiberi*< *minfree* **AND**
  media(NumeroBlocchiLiberi, ΔT) < *desfree*
  esegue **swapout** di uno o più processi

Come funziona il PageDaemon? Quando viene messo in esecuzione, lui vede se il numero di blocchi liberi nel sistema è maggiore o uguale a lotsfree. Se questo è vero, vuol dire che ci sono blocchi liberi in memoria fisica sufficienti per andare avanti per un po' e quindi non deve eseguire l'algoritmo di sostituzione, può lasciare il sistema così com'è e si interrompe. Se invece il numero di blocchi liberi in memoria principale è minore di lotsfree, allora vuol dire che il numero di blocchi fisici inizia a essere poco, potrebbe scarseggiare a breve, e bisogna liberarne un po', quindi cerca di evitare problemi, se lascia che il numero di blocchi fisici si decrementi troppo, rischia di andare in una situazione nella quale il numero di fault di pagina, ad un certo punto, è troppo alto e non lo può più gestire velocemente. Se il numero di blocchi liberi è minore di lotsfree ed è però maggior o uguale a minfree, quindi è compreso tra queste due soglie, oppure se il numero di blocchi liberi è di minfree (minore della soglia minima), però nell'ultimo periodo/intervallo tra due esecuzioni successive dell'algoritmo di sostituzione, è successo che il numero medio di blocchi liberi era maggior o uguale di desfree. Se il numero di blocchi liberi è minore di lotsfree ed è maggiore o uguale di minfree, allora esegue l'algoritmo di sostituzione fintantoché il numero di blocchi liberi non diventa uguale a lotsfree + k, dove k è un certo parametro. Quindi libera un certo numero di pagine fisiche in maniera tale che quelle liberate siano maggiori di lotsfree e in questo modo per un po' di tempo non dovrà rieseguire l'algoritmo di sostituzione. Questa sostituzione la fa in due casi: o il numero di pagine fisiche è compreso tra le due soglie, oppure se è minore della soglia minfree, che è una soglia da allarme rosso, però nell'ultimo periodo il numero medio di pagine fisiche libere è stato maggiore di desfree, che è un parametro intermedio, vuol dire che probabilmente siamo in una situazione un po' Borderline, ma ancora non estremamente critica, perché comunque mediamente il numero di pagine fisiche libere era ancora ragionevole, quindi in questo caso si limita semplicemente ad applicare l'algoritmo di sostituzione. Se però invece il numero di blocchi liberi è minore di minfree e nell'ultimo intervallo di tempo il numero medio di blocchi liberi è stato minore di desfree, allora questa è davvero una situazione molto critica ed in questo caso procede a fare lo Swap Out di uno o più processi. Quindi questa situazione è quella nella quale presume che i Working Set dei processi siano maggiori della memoria fisica e allora abbassa il grado di multiprogrammazione del sistema, prende qualche processo e interamente lo congela e lo sposta in memoria secondaria. Perché fa questo nel caso in cui il numero medio di blocchi liberi sia minore di desfree e non lo fa anche semplicemente se è minore di minfree, cioè perché considera anche questa condizione invece che considerare solo questa? Il problema è che lo Swap Out è un'operazione molto costosa che incide molto pesantemente sull'avanzamento di processi, quelli che vengono swappati restano sul disco per un bel po' di tempo, prima di tornare in esecuzione passerà parecchio tempo, quindi prima di prendere una decisione così grave (questa è l'arma finale), quindi prima di usare quest'arma finale cerca di dare possibilità al sistema utilizzando l'algoritmo di sostituzione per quanto possibile, questa è l'idea.

Sostituzione delle pagine in UNIX (BSD): relazione con la teoria del Working Set

- Se *NumeroBlocchiLiberi< minfree*
  elevata frequenza di errori di pagina nell'ultimo periodo di attivazione di *Page Daemon*
  ==> esistono processi con *#RS < #WS:* provocano thrashing

- Se *media(NumeroBlocchiLiberi, ΔT)< desfree*
  il fenomeno persiste da un certo periodo
  - Eseguendo *swapout* si liberano risorse di memoria
  - i processi con *#RS < #WS*, per effetto dei propri errori di pagina, possono incrementare *#RS*

Detto in altri termini, se il numero di blocchi liberi è minore di minfree vuol dire che la frequenza di fault di pagina è elevata e ci stiamo avvicinando a una situazione nella quale i Working Set dei processi non riescono più a stare in memoria principale. Se però anche la media dei blocchi liberi è minore di desfree, allora questo è un fenomeno che persiste a un certo periodo, quindi vuol dire che il sistema si trova spesso a lavorare un numero di blocchi liberi sotto minfree e allora in questo caso bisogna liberare le risorse.

Swapout e Swapin dei processi in UNIX (BSD):

Swapout

Se *NumeroBlocchiLiberi< minfree* e *NumeroMedioBlocchiLiberi< desfree* (media calcolata in un opportuno intervallo di tempo),
PageDaemon:

- Seleziona processi candidati allo scaricamento con criterio basato su:
  - tempo trascorso senza andare in esecuzione
  - quantità di memoria necessaria
- Esegue Swapout di uno o più processi fino a ottenere *NumeroBlocchiLiberi≥ lotsfree*

Swapin

Se *NumeroBlocchiLiberi* sufficientemente grande, *PageDaemon:*

- Seleziona uno o più processi in stato *swapped*, con criteri basati su:
  - tempo trascorso in stato *swapped*
  - quantità di memoria necessaria
- Esegue swapin di uno o più processi, rispettando la condizione
  NumeroBlocchiLiberi≥ lotsfree

Per quanto riguarda lo Swap Out, quindi lo spostamento in memoria secondaria di un processo, su che base viene scelto? Questa è davvero un'euristica, c'è poco da dire, processi differenti possono fare scelte completamente differenti e soprattutto è difficile trovare un ottimo, si cerca di utilizzare dei criteri ragionevoli cercando di individuare i processi il cui spostamento fa meno danno complessivamente al sistema. Tenete presente che i sistemi portati in memoria secondaria, non è che terminano, non è che vengono abortiti/buttati via, ma resteranno congelati, quindi senza aver la possibilità di essere eseguiti per un tempo indefinito, fintantoché il sistema non si libera, non si scarica, quindi quei processi subiscono un rallentamento molto pesante. Su che criterio li possiamo scegliere? Se scegliamo processi piccoli liberiamo poca memoria e potrebbe non essere sufficiente, quindi un criterio potrebbe essere scegliere processi che

occupano molta memoria, però magari processi che occupano molta memoria potrebbero essere processi che invece stanno interagendo molto con l'utente per cui chi ne paga le spese maggiormente è l'utente, quindi bisogna trovare un po' un equilibrio tra scegliere processi che occupano molta memoria, processi che presumibilmente sono CPU bound, non sono interattivi per cui l'utente risente meno di questo Swap Out e via discorrendo. Normalmente si fa lo Swap Out di uno o più processi finché il numero di blocchi liberi non supera lotsfree mai di una certa quantità, appunto per evitare di ritrovarci poi subito dopo ad avere problemi. Quand'è che si fa poi lo Swap In, quand'è che si riporta invece un processo che è in memoria secondaria in memoria principale? Lo si fa quando il numero di pagine fisiche libere diventa significativamente maggiore di lotsfree, per cui a questo punto portare il processo in memoria non è più un problema. E su che base si scelgono i processi da riportare? Potremmo averne più di uno sul disco. Qui potremmo adottare una politica FIFO, d'altra parte chi è da più tempo lì congelato ha sofferto più degli altri e quindi potrebbe essere riportato prima oppure potrebbero esserci dei criteri legati alla quantità di memoria necessaria. Quando si fa lo Swap In si può riportare in memoria principale uno o più processi, però bisogna sempre fare in modo che il numero di blocchi liberi resti maggiore di lotsfree, altrimenti rischiamo di ripiombare nuovamente nella situazione precedente. PAUSA.

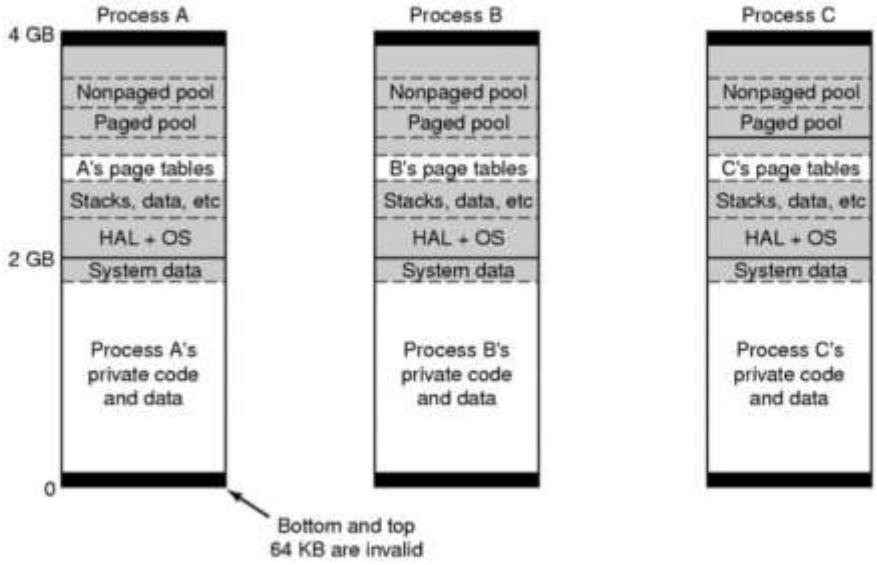
Gestione della memoria in Windows (32 bit)

- Dimensione della memoria virtuale: 4 Gbyte (indirizzo virtuale di 32 bit).
- Memoria virtuale paginata (paginazione a domanda) con pagine di dimensioni fisse (le dimensioni della pagina dipendono dalla particolare macchina fisica).
- *Spazio virtuale* suddiviso in due sottospazi di 2 Gbyte ciascuno
  - il *sottospazio virtuale* inferiore è privato di ogni processo
  - il *sottospazio virtuale* superiore è condiviso tra tutti i processi e mappa il sistema operativo.

Come ultimo caso di studio della memoria vediamo il caso di Windows, la versione a 32 bit, mi rendo conto che ormai tutti i sistemi stanno passando a 64 bit, però queste informazioni spesso vengono rese note con un po' di ritardo quindi non ho ancora trovato dei testi che vi spieghino com'è fatta la gestione della memoria in Windows a 64 bit, quindi vi dovete accontentare.

Con 32 bit di indirizzamento abbiamo a disposizione 4Gb di spazio virtuale, lo spazio virtuale è paginato con paginazione a domanda quindi a livello di sistema operativo non abbiamo la segmentazione, sappiamo che però poi se Windows è posizionato sopra alcune architetture che hanno segmentazione ovviamente il livello di adattamento all'hardware al processore gestirà anche segmentazione. Nel caso di Windows lo spazio virtuale è suddiviso in sottospazi di 2Gb ciascuno e questo viene fatto perché i 2Gb alti mappano il sistema operativo e i 2Gb bassi invece mappano lo spazio virtuale del processo. In buona sostanza che cosa succede in Windows? Succede che ogni processo ha il suo spazio virtuale e tutto il sistema operativo mappato all'interno della sua memoria virtuale. Siccome anche la memoria del sistema operativo è paginata, in questo modo nella tabella delle pagine del processo compare anche il mapping delle pagine del sistema operativo e questo permette al sistema di gestire più rapidamente le chiamate di sistema, quindi quando il processo vuole passare al nucleo, in realtà, invoca una chiamata di sistema ma continua la sua elaborazione in stato supervisore all'interno del nucleo, quindi eseguendo procedure del nucleo, continuando a riferire il suo spazio virtuale, quindi senza che questo causi scompensi nella TLB sostanzialmente, quindi il modello è questo:

Struttura della memoria virtuale in Windows



Le aree bianche sono private; le aree scure sono condivise

La parte bassa quindi è riservata al processo (circa 2Gb in realtà), poco meno di 2Gb la parte riservata alta riservata al sistema operativo. La parte riservata al sistema operativo è condivisa fra tutti i processi perché il sistema operativo è lo stesso, quindi la parte alta di tutti i processi in realtà mappa le stesse pagine con un'unica eccezione: il sistema operativo voi sapete che deve contenere la tabella delle pagine. La tabella delle pagine del processo A non è condivisa con la tabella delle pagine del processo B, C, etc, quindi ogni processo nella sua tabella delle pagine, mappa sé stesso, il sistema operativo e anche la sua stessa tabella delle pagine. Sembra una cosa autoreferenziale ma in realtà non lo è: la tabella delle pagine deve stare in memoria principale, deve poter essere utilizzata e quindi di conseguenza occuperà alcuni blocchi fisici e starà in alcune pagine virtuali del processo. Dobbiamo fare in modo che le pagine della tabella delle pagine del processo siano sempre raggiungibili, quindi alcune pagine del processo sono vincolate a stare in memoria principale, non possono essere sostituite, altrimenti rischieremo di andare in confusione anche nell'accesso alla tabella delle pagine, il processo è un po' più rigido.

L'altra cosa curiosa è che esistono due banchi di 64Kb in basso ed in alto con indirizzi bassi e indirizzi alti, che sono invalidi, non possono essere usati per nessuno scopo. Il motivo è che quando un codice fa confusione con gli indirizzi (il famoso errore di Segmentation Fault che spesso e volentieri avete beccato) era dovuto al fatto che avete utilizzato come puntatore un'area di memoria che non conteneva un puntatore. Molto spesso quando si fanno errori del genere, la cosa più probabile è che l'area di memoria usata erroneamente come puntatore fosse un'area vuota, non inizializzata, per cui in realtà molti dei Segmentation Fault sono causati da puntatori nulli (o quasi nulli). Rendendo invalida questa parte è un meccanismo per permettere al sistema di rendersi conto in anticipo del fatto che il programma causerà un Segmentation Fault, se qua dentro ci fosse stato del codice del processo A e voi avete utilizzato erroneamente quel puntatore, il sistema non si rende conto di un errore perché questa zona la potete utilizzare, la utilizza, ma a quel punto il processo probabilmente otterrà un risultato completamente scassato, quindi andrà avanti per un po' ma dopo un po' dovrà abortire. Per evitare di farlo andare avanti una volta che tanto ormai è già scassato, utilizzano questo trucco, lo si fa fallire in anticipo. Il punto è che nella maggior parte dei casi i puntatori prodotti per errore, utilizzati per errore, contengono con elevata probabilità il valore 0. Se voi nel vostro codice utilizzate per errore dei puntatori nulli, andate a riferire questa zona di memoria, questi qui sono gli indirizzi 000..0, 000..0, 00000..00, 0000..01, 000….02, 0000..03, …, comunque sono gli indirizzi per i quali in larga misura il valore dell'indirizzo è nullo. Rendendoli invalidi, non appena il processore utilizza uno di questi indirizzi, causa subito Segmentation Fault. Viceversa, se non

li rendete invalidi, può darsi che il sistema li abbia allocati al processo, per esempio il processo potrebbe averci messo del codice, e allora cosa succede in questo caso? Il processo ha preso un puntatore sbagliato, riferisce la memoria in maniera sbagliata e quindi prima o poi causerà un errore per dati inconsistenti. Però siccome l'indirizzo 0, che è quello che ha preso sbagliato, è allocato, il sistema non dà un Segmentation Fault perché sta riferendo a una zona di memoria allocata, sbagliata ma allocata. Questo vuol dire che il processo, riferendo queste zone di memoria, se non è bloccato, continuerà ad andare avanti, ma il suo andare avanti è una perdita di tempo perché comunque non potrà produrre risultati sensati. Per evitare questa perdita di tempo, se mappiamo quella zona di memoria come invalida, con probabilità maggiore il processo che deve andare in Segmentation Fault ci andrà prima. Stesso discorso per la zona di memoria alta, per quel blocco nero vicino ai 4Gb. Quegli indirizzi lì sono tutti indirizzi che sono fffff..ff, quindi sono tutti 1, con una probabilità leggermente inferiore, molti indirizzi che generano Segmentation Fault vanno a cadere in quella zona lì.

Memoria Virtuale in Windows

Spazio virtuale unico, suddiviso in *regioni*

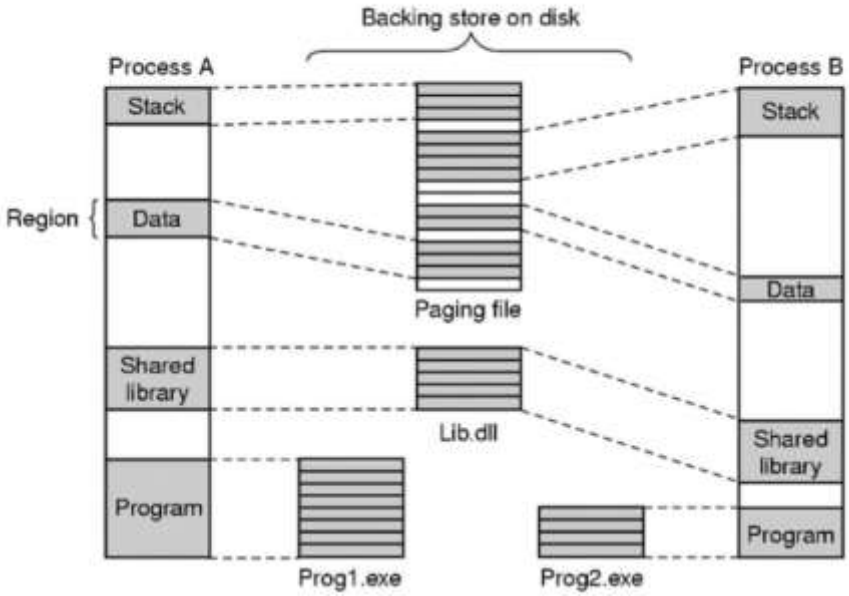
Ogni pagina logica può essere :

- *free* : se non è assegnata a nessuna regione
  - un accesso a una pagina free determina page fault non gestibile
- *reserved* : è una pagina non ancora in uso ma che è stata riservata per espandere una regione
  ==> non mappata nella tabella delle pagine
  - esempio: riservata per l'espansione dello stack
  - non utilizzabile per mappare nuove regioni
  - un accesso a una pagina reserved determina page fault gestibile
- *committed* : se appartiene a una regione già mappata nella tabella delle pagine
  - un accesso a una pagina *committed* non presente in memoria risulta in un page fault, *che determina il caricamento della pagina solo se questa non si trova in una lista di pagine eliminata dal working set*

Com'è organizzato lo spazio virtuale dei processi? È diviso in regioni, queste regioni grossomodo sono i segmenti, non sono propriamente segmenti (come abbiamo visto con la segmentazione), ma giocano un po' il ruolo dei segmenti. Chiaramente lo spazio virtuale, anche se di 2Gb è molto grande, per molti processi è enorme, in questo spazio virtuale i processi allocano i loro thread, allocano i codici, allocano le librerie, allocano lo heap (nello heap magari diverse strutture dati a mandate successive) e tutto questo va ad occupare delle zone dello spazio virtuale un po' a macchie di leopardo per cui il sistema operativo deve sapere quali zone dello spazio virtuale sono libere e quali sono occupate. Le zone occupate si chiamano regioni. Di conseguenza ogni pagina logica dello spazio virtuale può essere: o libera (se non è allocata nessuna regione), oppure 'committed' se appartiene a una regione, oppure potrebbe essere riservata. Che vuol dire riservata? Supponiamo che io debba allocare memoria per uno stack, ora lo so già che lo stack probabilmente crescerà e in una certa direzione però in questo momento lo stack è ancora piccolo, non ho bisogno di allocare le pagine per far crescere lo stack quindi materialmente quelle pagine per far crescere lo stack non le alloco, alloco solo quelle che servono. Però ho un problema: se successivamente qualcuno mi chiede di allocare memoria, io inavvertitamente la alloco troppo vicino allo stack, impedirò poi a quello stack di crescere. Allora la soluzione è quella di avere le pagine riservate, quindi quando alloco lo stack che faccio: alcune pagine le riservo per lo stack per contenere le informazioni che mi servono ora, le pagine che vengono dopo, che mi serviranno per far crescere lo stack, le marco come riservate, non sono concretamente allocate, non ancora, ma non possono essere usate se non per far crescere lo stack, quindi se c'è una richiesta di allocazione di memoria, quelle non le posso utilizzare. Quando poi lo stack crescerà,

riempirà le pagine committed e continuerà a crescere, andrà a riferire le pagine riservate a quel punto faccio una vera e propria allocazione.

Windows: copia del programma residente su disco



Quindi lo spazio virtuale del processo è gestito in questa maniera, con una serie di aree di regioni che svolgono il ruolo di segmenti (appunto, non si chiamano segmenti) e che in memoria secondaria possono essere memorizzati in punti differenti, in particolare il codice come vi dicevo prima sta nei file eseguibili, le librerie stanno nei file .dll che sono file che contengono codice di libreria, i dati e lo stack stanno nel file di paging che nel caso di Windows è un file speciale vincolato a stare in un punto ben preciso del disco.

Windows: gestione degli errori di pagina (1)

Algoritmo di sostituzione *Working Set*
==> è un algoritmo (fondamentalmente) locale
==> *working set* inteso come *insieme residente del processo (RS)*
==> dimensione ammissibile compresa tra valori *min* e *max*

- *min* e *max* intesi come massima e minima stima di # WS
- valori iniziali uguali per tutti i processi, ma modificabili dal *memory manager*

Se *x* è l'ampiezza corrente del *working set* (intesa come *x*= #*RS*) e si verifica un page fault, il *memory manager*:

- carica la pagina riferita in un blocco libero;
- se *x*<*max* assegna *x*= *x* +*1;*
- se *x*=*max*
  ==> #*RS* uguale a *max*
  - *x* non viene incrementato; le pagine in eccesso saranno scaricate dal *working set manager*

La disponibilità di un numero sufficiente di blocchi liberi è assicurata dai demoni *balance set manager* e *working set manager*.

Ora, come fa Windows a gestire i Fault di pagina, adotta un algoritmo di tipo Working Set, prevalentemente locale anche se in realtà è un po' un ibrido, c'è un elemento di globalità nell'algoritmo di Working Set di Windows. In questi algoritmi si fa sempre distinzione tra Working Set che sono le pagine virtuali in uso e insieme residente che son le pagine effettivamente allocate, quindi blocchi fisici a disposizione del processo sui quali sono stati allocati delle pagine. Questi due insiemi son disgiunti, quindi in Windows quest'algoritmo di sostituzione li tiene disgiunti. Ciò vuol dire che in un dato istante di tempo io potrei aver caricato in memoria molto più del Working Set, ma non è un problema perché tanto poi periodicamente interviene il Working Set Manager e fa pulizia. L'obiettivo dell'algoritmo di Working Set è di mantenere la memoria fisica, quindi il numero di pagine fisiche allocate ad un processo, comprese tra due soglie: minimo e massimo. Questi valori di minimo e massimo sono inizialmente uguali per tutti, però poi via via che il processo va avanti, riferisce alle sue pagine e il sistema operativo osserva il suo comportamento, questi valori di minimo e massimo vengono aggiustati in maniera tale da contenere meglio il processo. Supponiamo che x sia l'ampiezza iniziale del Working Set, quindi x è inizialmente anche il valore dell'insieme residente, supponiamo di essere in uno stato in cui il Working Set è l'insieme residente ed è caricato in memoria, come caso iniziale/ideale. Se si verifica un Fault di pagina, senza preoccuparsi di nulla, il Sistema Operativo prende la pagina richiesta e la carica, questa non comporta la rimozione di una pagina dal Working Set, la pagina richiesta viene presa e caricata e di conseguenza x viene incrementato, fintantoché non si arriva a max, al valore massimo. Se a questo punto il processo genera un altro Fault di pagina, il valore di x non viene più incrementato, però la pagina richiesta viene caricata, quindi può darsi che il processo in un dato istante abbia un numero di pagine caricate in memoria maggiore del massimo e questo di per sé non è un problema: finché ci sono pagine libere e il processo causa Fault di pagina, per risolvere rapidamente il suo problema di riferimento della pagina, la pagina viene caricata. Però poi, quando dopo, periodicamente viene messo in esecuzione il Working Set Manager, a quel punto il Working set Manager analizza questo processo, scopre che il numero di pagine caricate eccede il massimo e va a rimuovere tutte le pagine che stanno fuori dal Working Set per riportarlo nei ranghi. In questo modo il sistema può gestire rapidamente i fault di pagina di un processo perché comunque non cerca di scaricare niente quando ci sono fault di pagina e non cerca di scaricare niente se quel processo sta occupando troppa memoria, però poi dopo periodicamente interviene il Working Set Manager, rimuove le pagine in eccesso e fa in modo che il sistema abbia sempre un pool di pagine fisiche libere a disposizione. Questo meccanismo del Working Set Manager è gestito da un demone che si chiama Balance Manager.

Windows: gestione degli errori di pagina (2)

Balance Set Manager:

- Attivato periodicamente
- Se il numero di pagine libere è inferiore a una certa soglia, attiva *working set manager*

Questo Balance Manager viene attivato periodicamente, verifica se il numero di pagine libere è minore di una certa soglia e questa è un'analisi globale, il numero di pagine libere dell'intero sistema, allora va tutto bene anche se c'è qualche processo che sta occupando più pagine di quello che è il suo Working Set non è un problema: fintantoché ci sono pagine libere sufficienti si può lasciare tutto così. Se invece il numero di pagine fisiche libere scende sotto una certa soglia, allora si attiva il Working Set Manager e il Working Set Manager utilizza l'algoritmo del Working Set con una strategia inizialmente locale, però in una seconda fase adotta un meccanismo di tipo globale. Quindi che fa il Working Set Manager?

Windows: gestione degli errori di pagina (3)

Working Set Manager

- considera i processi con *x>min e*, per ogni processo, considera per ogni pagina *pag* caricata in memoria il bit di uso *R*
  - se *R=1*, azzera *R* e il contatore *cont(pag)*
  - se *R=0* incrementa *cont(pag)*

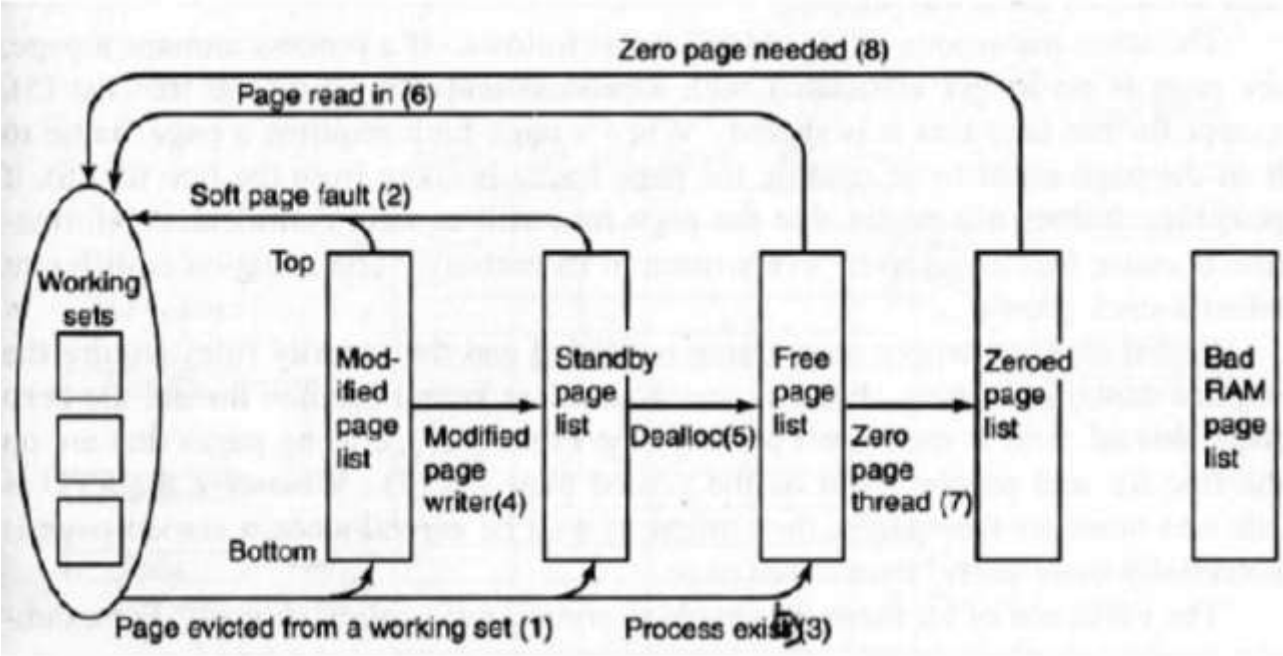
  ==> *cont(pag)* è un'approssimazione della distanza passata *globale*
- finchè esistono processi che hanno nel working set un numero di pagine maggiore di *max* *(#RS> max)*, marca per la rimozione pagine scelte nei *working set* *(= RS)* in ordine decrescente di *cont(pag)*, fino a raggiungere il numero desiderato di blocchi disponibili.
- Se è necessario per raggiungere il numero desiderato di blocchi disponibili, il procedimento viene applicato anche per gli altri processi con *x > min*.

È esattamente l'algoritmo che abbiamo visto prima: scorre per ogni processo la sua tabella delle pagine, va a aggiornare il tempo di ultimo riferimento di ogni pagina andando a guardare il bit di riferita. Dopodiché, e fin qui è l'algoritmo di Working Set. C'è una piccola differenza però: anziché utilizzare l'istante di ultimo riferimento, utilizza un contatore, quindi ogni volta che viene eseguito l'algoritmo Working Set, il contatore della pagina viene incrementato di 1. Quindi, se la pagina è stata riferita il contatore viene azzerato, se la pagina non è stata riferita il contatore viene incrementato, quindi più grande è il valore di questo contatore, da più tempo la pagina non è stata riferita. Le pagine che sono fuori dal Working Set vanno cercate tra quelle che hanno i valori più alti di questo contatore, quindi questo contatore in realtà non misura il tempo relativo al processo, ma è un'indicazione globale del tempo perché quest'algoritmo va ad incrementare sempre nello stesso periodo per tutti i processi, quindi di conseguenza questo ha l'effetto di un contatore di un tempo globale. Fatto questo, va ad analizzare il mio processo e vede: se c'è un processo che ha caricato in memoria un numero di pagine maggiore del massimo, quindi i processi per cui il numero di pagine caricate in memoria è maggiore del massimo, allora di questi bisogna andare a rimuovere pagine dal loro Working Set fino a riportarle al di sotto del massimo, però le pagine non vengono rimosse subito,

vengono messe in una lista di pagine da rimuovere, quindi quando voi analizzate ogni singolo processo, se un processo è minore del massimo non lo toccate, se un processo è maggiore del massimo individuate un certo numero di pagine da rimuovere per poterlo riportare all'interno del massimo numero di pagine fisiche allocate. Fate questo per ogni processo e alla fine vi trovate un insieme di pagine potenzialmente da rimuovere. Di tutto questo insieme di pagine da rimuovere, sono pagine che appartengono a tanti processi, non ad uno specifico processo, ma son pagine di un po' di tutti i processi. Queste pagine andate a rimuoverle a partire dalla più anziana, quindi da quella che ha il valore di contatore maggiore, quindi la rimozione in realtà avviene su base globale. Quindi in pratica state garantendo ad ogni processo di mantenere in memoria il suo Working Set, gli state permettendo di crescere, le pagine in più vengono rimosse su base globale, son scelte su base locale e vengono rimosse su base globale. In questo modo un processo più veloce che riferisce più spesso le sue pagine non potrà erodere il Working Set di un processo più lento, perché al processo più lento, comunque il suo Working Set viene garantito, gli viene rimosso soltanto quello che eventualmente è in eccesso. Quindi si vanno a rimuovere le pagine da questa lista globale fintantoché non abbiamo raggiunto il numero desiderato di blocchi disponibili. Se però questo non dovesse essere sufficiente, Windows non adotta la tecnica dello Swapping, ma va rimuovere blocchi anche dai processi che hanno x maggiore del minimo, quindi riconsidera di nuovo i processi e va a ridurre il loro Working Set sostanzialmente. Questa situazione potenzialmente potrebbe portare il sistema in thrashing, c'è da dire che Windows è un sistema pensato per macchine che hanno una sovrabbondanza di risorse quindi in realtà il thrashing non viene tanto risolto meccanismi di sistema ma viene risolto semplicemente assumendo che la memoria sia più che sufficiente. Diversi anni fa avevano intervistato Bill Gates e l'intervistatore gli contestava il fatto che programmi applicativi tipo word (word processor di Windows) fossero programmi molto più pesanti del necessario, che occupassero troppa memoria, insomma che andassero ingegnerizzati, sostanzialmente diceva "potreste investire un po' di briciole di quello che avete per ingegnerizzare Word e farlo diventare molto più efficiente". La sua risposta è stata "si è vero potremmo farlo, però in realtà se aspettiamo un anno o poco più, le risorse dei sistemi saranno talmente aumentate per cui in realtà questa necessità di reingegnerizzare Word scomparirà naturalmente e il tempo che dobbiamo aspettare probabilmente è paragonabile al tempo che ci porta via a reingegnerizzarlo e di farlo più efficiente". Quindi in certi casi i problemi si risolvono semplicemente mettendosi lì ad aspettare. Non vi invito a far questo <risata di sottofondo>.

Windows: gestione degli errori di pagina (4)

Liste delle pagine e transizioni tra le liste



Quest'ultimo diagramma vi fa vedere tutto il meccanismo di gestione dei rifiuti di Windows: la gestione dei rifiuti riguarda le pagine che a un certo punto l'algoritmo di sostituzione ha deciso di rimuovere. In realtà come viene anche nella vita quotidiana, i rifiuti passano attraverso un processo molto lungo, per esempio: in ufficio per prima cosa se ci sono dei fogli da buttare via, li butto nel mio cestino, dopodiché una volta al giorno/alla settimana passa l'agenzia delle pulizie, prende i cestini e li svuota e li mette in un cestino più grande da dove poi vengono presi, raccolti periodicamente dalla raccolta dell'immondizia comunale, che li porta nel centro riciclo e lì vengono riutilizzati definitivamente. Che cosa comporta in realtà tutto questo meccanismo? Comporta che se io prendo un foglio e lo accartoccio e lo butto nel cestino, se dopo 10 minuti mi viene in mente che ho fatto un errore e che quel foglio mi serviva, posso andare a cercarlo nel mio cestino e lo posso recuperare. Se però me lo ricordo il giorno dopo è nel frattempo l'agenzia di pulizie è già passata e l'ha raccolto, ho ancora una speranza: potrei cercare il cassonetto dove tutti questi fogli vengono buttati, mi ci butto là dentro e lo posso andare a cercare, eventualmente lo recupero, se però quel cassonetto è già stato preso dall'agenzia di pulizie comunali a quel punto non c'è più speranza, probabilmente è irrecuperabile. Qui alle pagine scartate selezionate dai processi per la rimozione avviene esattamente la stessa cosa. Questi sono i Working Set dei processi, da questi Working Set dei processi per effetto dell'esecuzione periodica del Working Set Manager, ogni tanto escono alcune pagine. Queste pagine che escono in realtà sono dei blocchi fisici che hanno caricato dentro ancora la pagina virtuale di quel processo e che quella pagina è in realtà ancora utilizzabile da quel processo. Quindi prima di buttarli via definitivamente, vengono messi in due liste: se son stati modificati vengono messi nella lista delle pagine modificate, se non son stati modificati vengono messe nella lista delle pagine in standby. Che caratteristiche hanno queste due liste? Queste due liste contengono pagine fisiche che sono già uscite dal Working Set dei processi per le quali però io ancora mi ricordo a quale processo appartengono e quale pagina virtuale contengono. Per cui se per caso un processo causa un page Fault e mi rendo che quella pagina che lui ha riferito sta qua dentro, anziché andare a prenderla dal disco e fare un macello, la riprendo direttamente da qui, quindi in questo caso il page fault costa molto poco, è quello che viene detto Soft-page-Fault. Perché la pagina è ancora riconoscibile, sebbene eliminata dal Working Set del processo, dalla tabella delle pagine, ancora sta in memoria e ancora so a chi appartiene e che cosa contiene, quindi è

immediatamente riutilizzabile senza andare a caricarla dal disco, questo Soft-Page-Fault comporta soltanto il riaggancio della pagina alla tabella delle pagine del processo che ha causato il Fault di pagina. Questo meccanismo mi permette di essere un po' più grossolano nella scelta delle pagine da rimuovere perché se per caso rimuovo una pagina che invece non dovevo rimuovere perché stava ancora nel Working Set, poco male perché il page Fault si gestisce rapidamente. Le mette in due liste separate perché le pagine che son state modificate devono essere salvate, via via che vengono salvate da un demone che opera in background, vengono spostate nella lista delle pagine in standby. È chiaro che queste pagine ad un certo punto dovranno davvero essere liberate, le posso tenere lì in memoria principale, però se poi dopo la memoria libera mi si riduce troppo, poi queste pagine le pagine le devo riutilizzare per forza di cose. Quindi c'è un demone che opera periodicamente e che procede alla de-allocazione delle pagine, di quelle che sono in standby, via via che le dealloca, quindi cancella l'informazione legata all'indice del processo, all'indice di pagina, queste pagine diventano effettivamente libere. Però badate bene, son libere ma non sono ancora azzerate, contengono ancora il contenuto originale, sebbene però io non sappia più a chi appartengono, questo è il mio foglietto accartocciato che ormai è finito nel camion dei rifiuti, chissà a chi appartiene, le informazioni ce le ha ancora scritte, però andare a scoprire che apparteneva a me a quel punto diventa impossibile. Se per caso c'è una richiesta di una pagina da parte del processo (nel caso di un page Fault ha bisogno di una pagina da allocare), questa pagina può essere allocata al processo, non è stata azzerata, ma tanto dovrà essere sovrascritta da qualcosa caricato dal disco, quindi io la posso allocare, do il comando di caricamento dal disco, viene sovrascritta, e a quel punto entra nella disponibilità del processo che l'ha richiesta. Oltre a questo c'è un altro demone che prende le pagine libere e le azzera, le pagine azzerate a cosa servono? Servono a quando faccio una malloc, quando voglio allocare memoria e voglio avere memoria libera e azzerata, vado a prendere pagine di quella free(?). Oltre a tutto questo può darsi che alcune pagine siano danneggiate/non siano più utilizzabili, queste il sistema operativo se le segna perché non le può utilizzare e le mette nella lista delle pagine cattive (☹).

Domanda da uno studente: In una macchina virtuale, se per esempio avessi Linux installato su una macchina virtuale, come è gestita la memoria dal sistema operativo?

Risposta: è un bel divertimento, perché in realtà si sovrappongono due sistemi di gestione della memoria: siccome sotto c'è Windows, dal punto di vista di Linux la macchina virtuale Linux (sarebbe anche vero il contrario ovviamente) è un processo. Per cui la gestione della memoria fisica la fa Windows, quel processo che sta eseguendo la macchina virtuale Linux, in realtà lui vede la sua memoria virtuale ma la interpreta come la sua memoria fisica. Dal suo punto di vista è irrilevante perché lui gestirà i suoi processi che son tutti processi virtuali, andandoli a mappare nella sua memoria virtuale, quindi lui gestirà: memoria virtuale dei suoi processi, dove ogni suo processo avrà uno spazio virtuale di 4Gb diciamo, che sarà mappato in una memoria fisica di 1Gb. A questo punto il sistema Linux gestirà i suoi Fault di pagina andando a caricare le pagine richieste dai processi nel suo spazio virtuale, per lui è una memoria fisica ma in realtà lo sta facendo nel suo spazio virtuale. Al livello sottostante Windows si renderà conto che il processo macchina virtuale sta facendo riferimento a delle pagine del suo spazio virtuale che sono nel disco e gliele andrà a caricare lui. Quindi in realtà il caricamento fisico dal disco lo gestirà sempre il sistema sottostante con il suo meccanismo di paginazione, questo meccanismo verrà forzato implicitamente dal fatto che al livello di Linux sulla macchina virtuale, vengono generati dei fault di pagina virtuali e questo sistema andrà a fare dei caricamenti nella sua macchina virtuale.

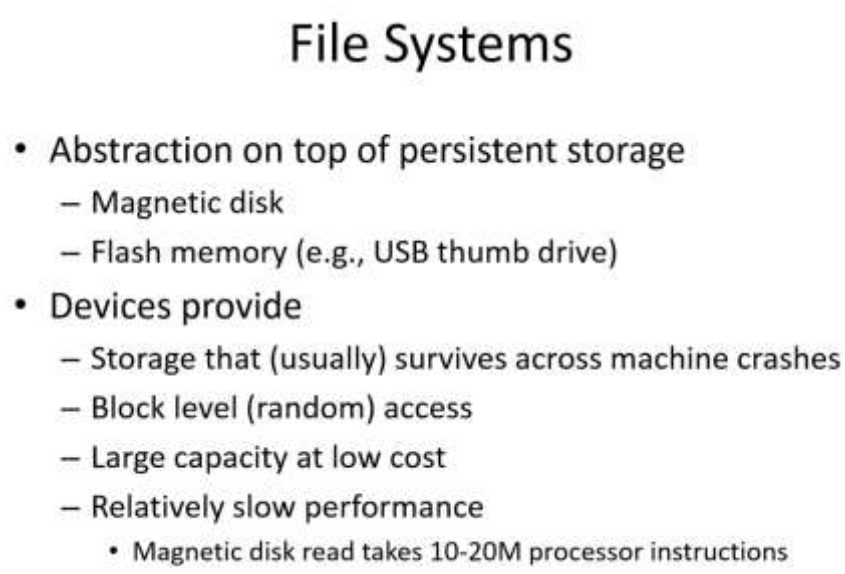
Supponiamo che un processo Linux causi un Fault di pagina, ovviamente quella pagina non è presente nello spazio virtuale e il sistema sottostante non può sapere che quella lì è una pagina da caricare nello spazio virtuale, quindi il sistema Linux darà ordine al disco di andare a caricare quella pagina in quella pagina virtuale, questo comando lo dà al sistema Windows sottostante. D'altra parte lo spazio virtuale di Linux, dal punto di vista di Linux è memoria fisica, quindi è interamente caricato, dal punto di vista di Windows no, quindi se il processo Linux riferisce una pagina del suo spazio virtuale e risulta caricata in memoria

principale, per Linux questo non causa un Fault di pagina, ma questo invece poteva essere un Fault di pagina di Windows perché per qualche motivo aveva deciso per quella pagina di sostituirla. Questi meccanismi funziona bene se c'è davvero tanta memoria, ci deve essere abbastanza memoria fisica perché la macchina virtuale di Linux in realtà, alternando processi differenti, si trova ad essere un processo che ha un Working Set enorme. Il vero Working Set della macchina virtuale è: le pagine utilizzate dalla macchina virtuale per poter lavorare lei, più l'unione di tutti i Working Set di tutti i processi Linux, quindi in realtà il Working Set dell'intera macchina virtuale è molto molto grande. Per funzionar bene bisogna che tutto il Working Set di tutta questa baracca sia caricato nella memoria fisica di Windows, ma per far questo bisogna che la memoria fisica di Windows sia molto grande, perché oltre ad avere il Working Set di tutto il sistema Linux, deve avere poi il Working Set di tutti i processi Windows quindi ci vuole tanta memoria, oppure fate funzionare soltanto la macchina virtuale e bloccate gli altri processi.

Sistemi operativi 18/05/17

Per quanto riguarda i sistemi di memorizzazione, vediamo due cose oggi:

- Informazioni generali sul File System
- Caratteristiche di alcuni dispositivi di memorizzazione



Intanto, cos'è il file system?

Il file system è un'astrazione che è messa sopra i dispositivi di memorizzazione. Un dispositivo di memorizzazione, tipo un disco, ha un'interfaccia molto primitiva, permette di andare a leggere un blocco sulla base di un indice o addirittura di una tripla che codifica la posizione di quel blocco all'interno del disco fisico.

Ovviamente nessuno utilizza i dischi in questa maniera, o meglio, gli utenti normalmente non usano i dischi in questa maniera e neanche i programmatori, e neanche grande parte del sistema operativo ma si basa su delle astrazioni che sono fornite dal file system, volendo, tramite queste astrazioni il file system permette di rappresentare supporti di memorizzazione molto differenti tra di loro quali possono essere i dischi rigidi classici oppure i dischi SSD però per entrambi offre la stessa visione quindi lo stesso metodo di accesso. D'altra parte il dispositivo offre: supporto alla memorizzazione persistente che quindi può sopravvivere anche a crash della macchina, accesso a blocchi all'informazione, grande capacità di memorizzazione a basso costo e prestazioni relativamente basse. Tenete presente che un accesso ad un disco magnetico porta via 10 20 milioni di (??? min: 2.46) del processore, quindi enormemente più lento.

File System as Illusionist: Hide Limitations of Physical Storage

- Persistence of data stored in file system:
  - Even if crash happens during an update
  - Even if disk block becomes corrupted
  - Even if flash memory wears out
- Naming:
  - Named data instead of disk block numbers
  - Directories instead of flat storage
  - Byte addressable data even though devices are block-oriented
- Performance:
  - Cached data
  - Data placement and data structure organization
- Controlled access to shared data

Ora su questo dispositivo per poter offrire la stessa astrazione, il file system agisce da illusionista quindi:
- Garantisce che le informazioni possano sopravvivere anche ad un crash mantenendo all'interno del disco informazioni ridondanti e utilizzando dei codici correttori nel caso i cui ci siano degli errori o che ci siano o dei difetti sul dispositivo fisico.

- Offre l'utilizzo dei nomi, quindi ci dà l'impressione, l'illusione di avere a disposizione informazioni associate a dei nomi, creando delle strutture all'interno del disco per raggruppare informazioni sulla base dei desideri dell'utente e appunto, utilizzando l'astrazione dei nomi per poter garantire l'accesso

- Agisce da illusionista anche dal punto delle prestazioni. Utilizzando un caching piuttosto aggressivo aumenta dal punto di vista dell'utente le prestazioni del disco avvicinandole a quelle della memoria.

File System Abstraction

- File system
  - Persistent, named data
  - Hierarchical organization (directories, subdirectories)
  - Access control on data
- File: named collection of data
  - Linear sequence of bytes (or a set of sequences)
  - Read/write or memory mapped
- Crash and storage error tolerance
  - Operating system crashes (and disk errors) leave file system in a valid state
- Performance
  - Achieve close to the hardware limit* in the average case

In tutto questo crea delle astrazioni; in particolare

- Crea l'astrazione di file system che è una struttura gerarchica organizzata ad informazioni (rappresentate da nomi).
- Crea l'astrazione del file che è una collezione di dati associata ad un nome, generalmente una sequenza lineare di byte.
- Fornisce caching e storage per la (??? guasti?: min 4.37) e migliora le prestazioni.
- Le altre astrazioni che crea sono quelle di direc("ma questo suono ? è terapia musicale?")

(Ho capito, stavo cercando un video sulla Agricoltura di precisione per una presentazione che devo fare e avevo aperto una decina di pagine sotto)

File System Abstraction

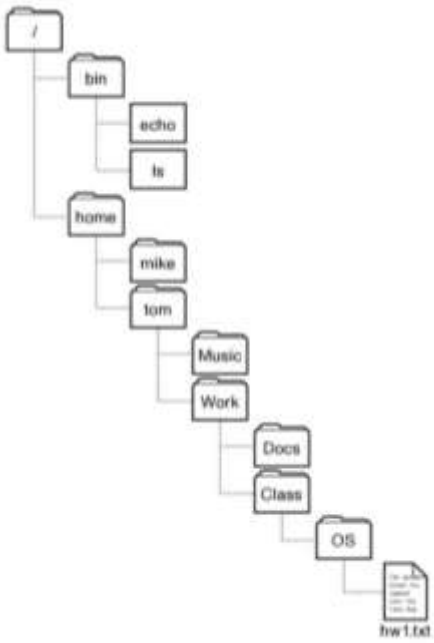
- Directory
  - Group of named files or subdirectories
  - Mapping from file name to file metadata location
- Path
  - String that uniquely identifies file or directory
  - Ex: /cse/www/education/courses/cse451/12au
- Links
  - Hard link: link from name to metadata location
  - Soft link: link from name to alternate name
- Mount
  - Mapping from name in one file system to root of another

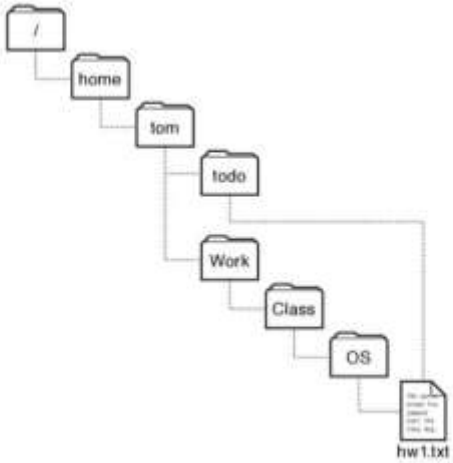
- Allora, crea una serie di astrazioni tra cui la Directory che è un contenitore per file, per gestire le informazioni in maniera più strutturata, più organizzata.

- L'astrazione di PATH che è una stringa che identifica univocamente all'interno del file system un file o una directory

- I link che sono dei collegamenti a file, possono essere HARD o SOFT. I Soft link sono quelli usati ampiamente su Windows (i collegamenti), per cui viene creato un file che contiene il Path verso un altro file. Le Hard link invece sono dei veri e propri collegamenti a file da directory differenti che avvengono inserendo proprio un entry all'interno della directory equivalente.

- L'ultima astrazione è quella del montaggio di diversi supporti di memorizzazione che permette quindi di vedere all'interno di un unico file system omogeneo diversi dischi, diverse memorie flash e via discorrendo.





(slide cambiata. min: 6.42)

Questa è una struttura ad albero di directory di un sistema unix, come vedete, ci sono le directory che son dei contenitori per altre sottodirectory oppure per file, in questo caso in questa directory contiene due sotto directory e un file, e la struttura si riproduce in questa maniera ad albero.

(figura a dx) In questo caso avete, un file "hw1.txt" che appare in due directory distinte, questo badate non è un collegamento, perché il collegamento in realtà è un file differente che contente un'informazione differente dal file originario, contiene il path verso il file originario, per cui, voi aprite il collegamento e il sistema operativo sa che deve aprire quel file, leggere il path, seguire il path e andare al file originario. In questo caso invece, è esattamente lo stesso file e queste due sono esattamente la stessa entry di directory che punta allo stesso file (HARD LINK).

Chiaramente, in due regioni differenti il file può avere nomi differenti. In Unix si fa uso di Hard link anche nel sistema operativo.

(slide mancante: min 08.20)

Allora, che cosa sono le directory? Una Directory è un file che contiene una tabella che crea un'associazione tra nomi di file e il file vero e proprio inteso come area del disco che contiene i dati associati al file.

Ci sono diversi modi di organizzare le directory, un modo che è quello utilizzato nel file system di tipo FAT (quello che usate generalmente per le chiavette USB); nelle directory di quel file system trovate una tabella di questo genere, per ogni riga avete il nome del file(la stringa che rappresenta il nome del file) associato ad una serie di attributi, che possono essere : tipo di file, lunghezza del file, data di creazione, data di ultima modifica e poi anche, posizione all'interno del disco nel quel il file si trova.

Quindi la directory non è altro che una sequenza di record tutti con la stessa struttura. Questo nel caso delle directory del file system FAT, se voi invece andate nelle directory del file system Unix all'interno

delle cartelle trovate una tabella che contiene un'associazione per nome di file e il puntatore ad una struttura dati ulteriore che si trova nel disco.

(slide mancante: min 09.55)

In effetti nel file system unix c'è una struttura che si chiama I-List che contiene un descrittore di per ogni file, questo descrittore contiene tutti gli attributi del file e la directory contiene il puntatore al descrittore. Il descrittore nel caso di unix si chiama I-Node.

(slide mancante: min 10:24)

Per quanto riguarda gli accessi ai file, chiaramente creare un'astrazione significa definire delle strutture dati delle funzioni per poterle utilizzare. Quindi, nel caso dei file la struttura dati è il descrittore del file, quindi l'i-node oppure le informazioni contenute nella directory associate al file, e poi una zona del disco dove quel file effettivamente risiede, ma questo non basta per creare un'astrazione; bisogna anche fornire ai programmatori delle funzioni per poterle utilizzare.

Queste funzioni sono quelle che avete visto a laboratorio per l'accesso dei file, quindi sono funzioni che permettono di accedere ai file in lettura e scrittura, di aprirli di manipolarli di copiarli e via discorrendo. Allora astraendo un po' rispetto a quello che è avete viso nello specifico nel caso di unix, le operazioni sono in termini di lettura o scrittura di record logico. Che cos'è un record logico? Nel caso di Unix è il singolo byte, quindi voi potete fare letture o scritture di un singolo byte all'interno di un file, ci sono certi file system dove il record logico è una struttura più complessa.

Ora per poter eseguire queste operazioni di lettura o scrittura di record logici nel file è necessario che il sistema operativo sappia quale file ci stiamo riferendo, abbia caricato in memoria principale, i riferimenti al file, tutti i suoi attributi e sappia dove questo file è fisicamente collocato nel disco. Per evitare di fare queste operazioni di raccolta informazioni di file tutte le volte che vogliamo fare una lettura/scrittura, i sistemi operativi prevedono un'operazione preliminare che è la open: una operazione con la quale il programmatore dice al sistema operativo "Guarda, voglio utilizzare questo file", in risposta a questa richiesta, il sistema operativo carica tutte le strutture dati necessarie per lavorare su quel file e a questo punto le operazioni successive di lettura/scrittura sono molto rapide, efficaci, perché già in memoria principale c'è tutto quello che serve per poterla eseguire. E siccome c'è questa operazione di apertura, necessariamente serve anche una operazione di chiusura che serve a deallocare tutto quello che riguarda il file che è stato messo in memoria quando abbiamo smesso di usarlo.

L'accesso al file può essere sequenziale o diretto.

(slide mancante. Min 13:09)

Sequenziale se andiamo a leggere ogni singolo record logico uno dopo l'altro; invece l'accesso è diretto se possiamo leggere i record logici spostandoci in punti arbitrari del file. Normalmente in Unix e in Windows, l'accesso di base dei file è Sequenziale voi aprite un file, sapete che la prima read leggerà il primo byte, la seconda read leggerà il secondo byte e via discorrendo. Potete anche fare un read a blocchi e in quel caso leggerete un certo numero di byte sequenziali. Però vi viene anche fornito un metodo di accesso diretto con la seek, voi tramite la seek potete spostarvi in un'altra parte del file senza aver letto tutto quello che viene prima. Ora badate bene che questo tipo di acceso che voi fate in realtà è indipendente dalla natura del file e del supporto fisico. D'altra parte, il supporto fisico al di sotto potrebbe essere intrinsecamente sequenziale, (i primi file system erano posizionati su nastri).

Negli anni '60/'70 le informazioni venivano messe su grossi nastri, e su questi nastri l'accesso era intrinsecamente sequenziale non potevate accedere fisicamente ad un punto arbitrario del nastro, prima lo dovevate scorrere. Nel caso dell'accesso diretto è completamente diverso, e quello lo potete fare su un disco; voi dite "voglio andare a questa posizione" e il disco rapidamente va in quella posizione senza obbligarvi a leggere tutto quello che viene prima. La cosa notevole è che indipendentemente dal fatto che a livello fisico l'accesso sia sequenziale o possa essere diretto, il file system vi offre spesso entrambi i metodi di accesso e poi si fa lui carico di assorbire le differenze. Questo vuol dire che se avete un file system messo su un nastro, potete comunque avere operazioni di accesso diretto perché il file system si preoccupa di implementare quell'accesso diretto in termini di accesso sequenziale.

(slide mancante min 16:02)

L'accesso sequenziale, appunto quello che prevede l'accesso ai record logici uno dietro l'altro (l'operazione è tipo ReadNext) e ogni volta che fate una lettura viene spostato il puntatore del record logico in modo che possa puntare al record logico successivo.

(slide mancante min 16:35)

Nell'accesso diretto invece voi potete specificare la posizione, e quindi il tipo di operazione che fate: andate a leggere da un file a partire dalla posizione i e poi voglio depositare il risultato nel buffer. In Unix questa posizione i non la specificate perché è implicita. Per fare l'accesso diretto voi fate un'operazione differente con la seek spostate il record logico e poi con la read fate un'operazione di lettura sequenziale a partire da quel record logico.

UNIX File System API

- create, link, unlink, createdir, rmdir
  - Create file, link to file, remove link
  - Create directory, remove directory
- open, close, read, write, seek
  - Open/close a file for reading/writing
  - Seek resets current position
- fsync
  - File modifications can be cached
  - fsync forces modifications to disk (like a memory barrier)

In un file system di tipo Unix, le operazioni che vi vengono date a disposizione sono operazioni per creare un file (voi avrete visto la Open), link e unlink che servono per creare Hard link ai file, per creare o rimuovere directory, per aprire file e poi anche operazioni di tipo differente tipo da fsync che serve a memorizzare i contenuti dei buffer in memoria con ciò che sta nel disco.

Perché la fsync è utile? Spesso il sistema operativo per accelerare le operazioni, quando c'è un'operazione di scrittura, non la implementa subito, ma adotta una politica di tipo writeback (prima o

poi scrivo), questo vuol dire che alcune informazioni non vengono scritte nel disco ma sono in realtà conservate in una cache in memoria; dal punto di vista dell'utente è come se fossero scritte nel disco perché in realtà qualsiasi accesso al disco passa prima alla cache e quindi l'utente accede alle informazioni in cache. Il problema è che però se manca la corrente o se comunque il sistema collassa il disco rischia di non essere più consistente, allora per questo motivo potete forzare la scrittura di tutte le informazioni che sono ancora in cache nel disco in maniera tale da avere le operazioni al sicuro. La fseek è una di quelle cose che si fanno per esempio quando si chiude una sessione, quindi quando di spegne il computer.

File System Interface

- UNIX file open is a Swiss Army knife:
  - Open the file, return file descriptor
  - Options:
    - if file doesn't exist, return an error
    - If file doesn't exist, create file and open it
    - If file does exist, return an error
    - If file does exist, open file
    - If file exists but isn't empty, nix it then open
    - If file exists but isn't empty, return an error
    - …

Vabbè, la open la conoscete, e qui dice che è un coltellino svizzero perché in effetti con la open, con un'unica chiamata potete fare tantissime cose. Quindi se il file non esiste allora dà un errore altrimenti se non esiste allora crealo e aprilo, oppure se il file esiste fai un errore ecc... Quindi la open vi permette di, diciamo, semplificare l'apertura del file andando già ad isolare tutta una serie di cause che possono dare problemi o degli errori.

Interface Design Question

- Why not separate syscalls for open/create/exists?
  - Would be more modular!

```
if (!exists(name))
    create(name);   // can create fail?
fd = open(name);   // does the file exist?
```

Se voi non aveste una open fatta in questa maniera a la Unix, voi dovreste scrivere del codice abbastanza noioso andando a scrivere cose come: se il file non esiste allora crealo, dopo che l'hai creato allora eseguilo ecc... Con la open vi viene restituito un codice di errore e la risolvete prima.

Device Access

- A software stack that provides ways to access a wide range of I/O devices
- A common interface
  - In posix equivalent to file access
  - In terms of open/close, read/write system calls
- Device drivers for each specific device
  - Upper part HW-**independent**
  - Lower part HW-**dependent**

Per quanto riguarda invece l'accesso ai dispositivi; l'accesso ai dispositivi è offerto da un software che sta nel sistema operativo e che in realtà, è strutturato a due livelli:

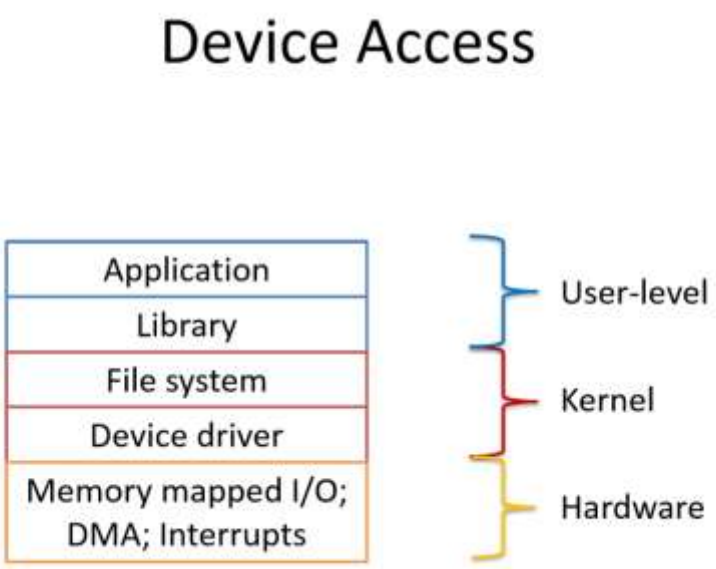
- da un lato offre un'interfaccia, per esempio l'interfaccia POSIX in termini di open, close, read, write.

- dall'altra parte interagisce a livello fisico con il dispositivo usando le interruzioni

Per far queste cose, in realtà, il sistema di accesso ai dispositivi è organizzato in due parti:

- una parte a basso livello che è dipendente dall'hardware

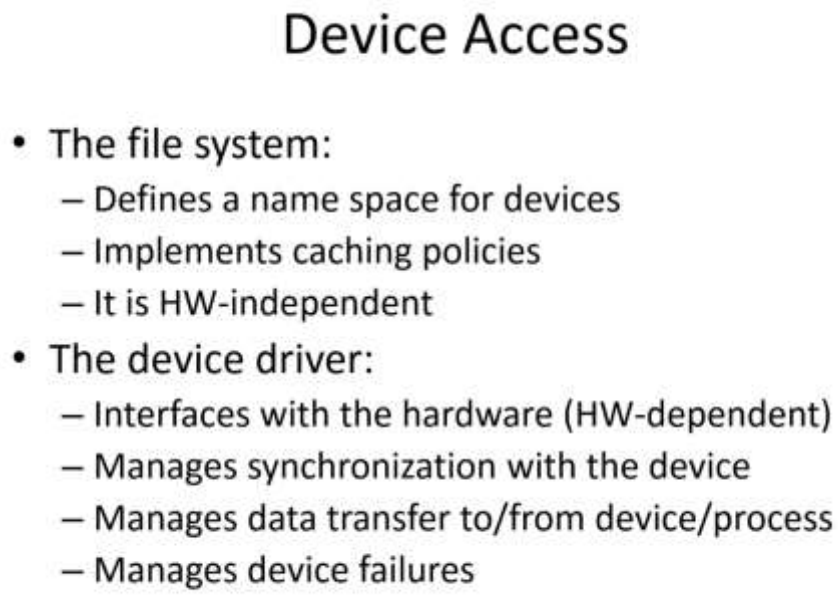
- una parte indipendente dall'hardware



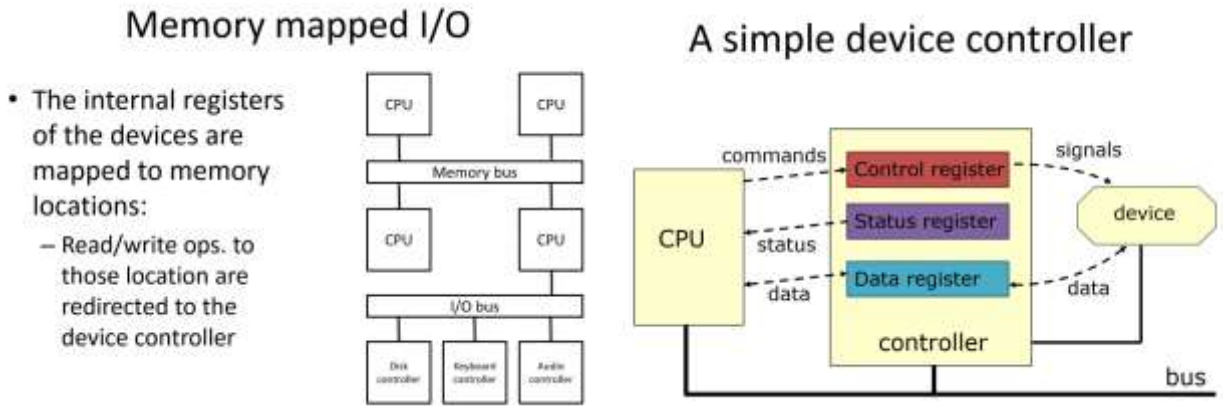
Quindi, all'interno del sistema voi trovate, ci sono le vostre applicazioni le vostre librerie qui siete al livello utente e questo è il livello, diciamo questa è l'interfaccia data dal sistema operativo, l'interfaccia POSIX.

Sotto questa interfaccia POSIX trovate il file system che definisce le politiche di alto livello ed implementa le astrazioni (file, directory, e via discorrendo) e il file system si posa sul device driver che poi è quello che materialmente va a comunicare con i dispositivi fisici.

Il device driver a sua volta ha due parti: una parte dipendente dal dispositivo e una indipendente. La parte dipendente dal dispositivo è quella che effettivamente comunica con il dispositivo che gestisce le interruzioni, gestisce il trasferimento di dati. la parte indipendente dal dispositivo è quella che gestisce la politica di accesso al dispositivo, quindi controlla i diritti di accesso al dispositivo o gestisce il trasferimento di informazioni a più alto livello verso processi e via discorrendo.

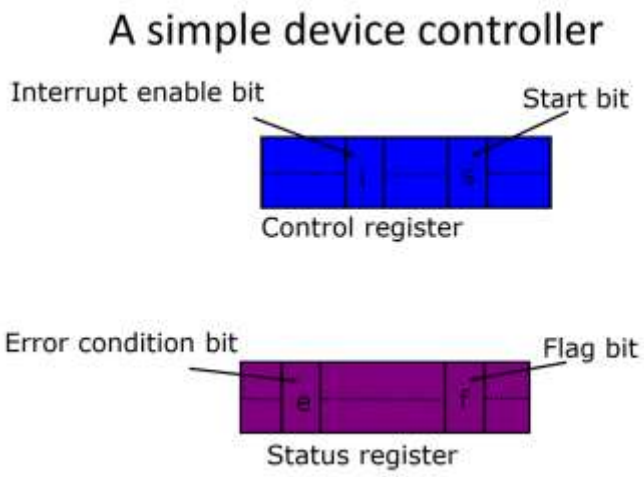


Ok, c'è scritto qui. Il file system definisce uno spazio di nomi, implementa le politiche di caching, è indipendente dall'hardware. Il device driver ha l'interfaccia con l'hardware che è HW- dependent quindi dipende dall'hardware, si sincronizza con il dispositivo, gestisce il trasferimento tra processi e device. Una parte del device driver è indipendente dal dispositivo e una parte è dipendente. Una parte per esempio della gestione dei guasti può essere generica e può valere per dispositivi differenti, la parte invece specifica di interfaccia che dà i comandi, ovviamente se si danno comandi ad un SSD o si danno comandi ad un disco, saranno differenti. E se il disco è più grande o più piccolo di nuovo ci saranno delle differenze.



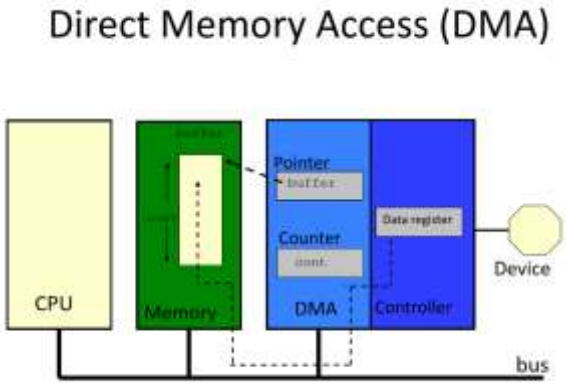
I dispositivi in realtà sono gestiti da un controller, questo controller ha dei registri e voi programmate il dispositivo andando a scrivere o leggere su questi registri, per cui dovete avere un metodo dato dal processore per accedere ai registri più interni, e la cosa viene realizzata a livello hardware mappandoli su certe porzioni di memoria. Per cui il processore scrive su certe locazioni di memoria e automaticamente questo comporta una scrittura sui registri del controllore o una lettura.

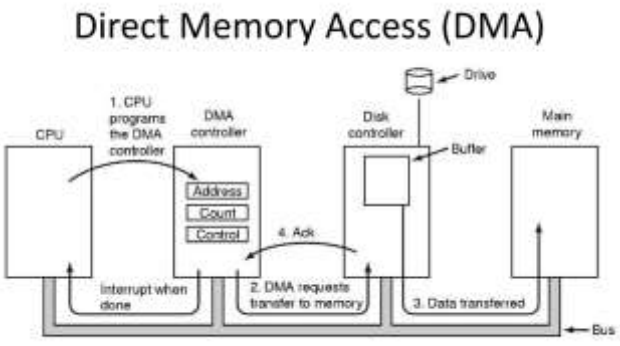
Per cui un controllore molto semplice (figura a dx) può essere gestito in questa maniera, potrebbe avere un registro di controllo, un registro di stato e un registro dei dati. Il registro di controllo in cui il processore scrive i comandi, quindi il processore scrive una certa stringa di bit e questo significa che richiede una certa operazione a quel dispositivo. Il controller a questo punto legge questo registro di controllo, implementa l'operazione sul dispositivo che normalmente si può tradurre con un trasferimento di dati, dal dispositivo verso il controller o dal controller al dispositivo a seconda che sia un'operazione di lettura o di scrittura, e poi successivamente il processore può andare a leggere questi dati sempre inviando degli altri comandi. Inoltre il processore può leggere lo stato del dispositivo accendendo ad un altro registro di stato, questo registro di stato conterrà informazioni riguardo ad eventuali guasti, mal funzionamenti oppure fasi dell'esecuzione di alcuni comandi oppure dà informazioni sul fatto che il dispositivo è ancora in fase di inizializzazione e che quindi non può essere usato e via discorrendo.



Qui ci sono esempi di che cosa può contenere un registro di controllo, tipicamente è organizzato come maschera di bit, qualche bit rappresenta il segnale di inizio dell'operazione e qualche altro bit invece specifica altre informazioni o cose che il controller può fare e il registro di stato contiene bit di condizione e i bit di flag. I bit di condizione rappresentano lo stato del dispositivo fisico e del controller, i bit di flag sono dei parametri.

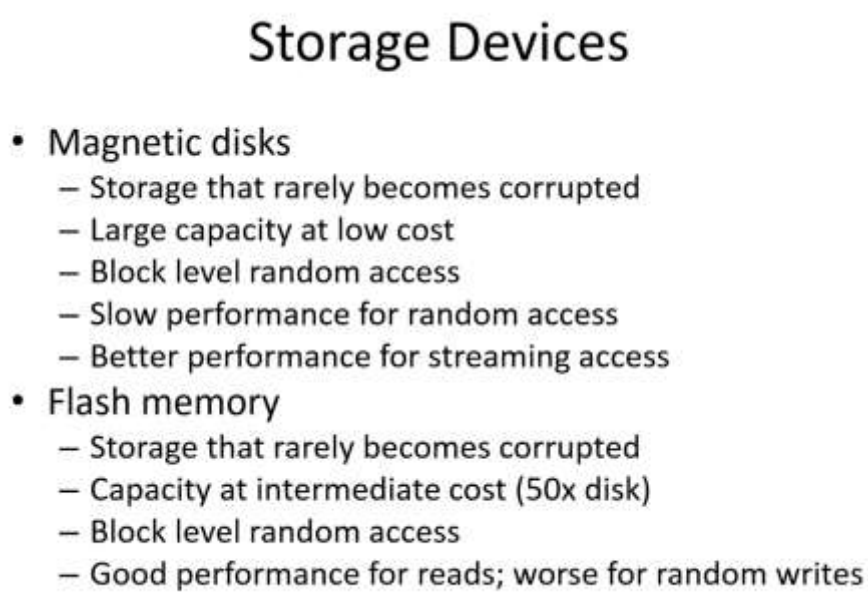
Questo per un dispositivo molto elementare, i dispositivi più complessi vengono gestiti col DMA. Il DMA permette un trasferimento automatico dal dispositivo alla memoria principale senza coinvolgere il processore per questo è utilizzato coi dispositivi più potenti perché permette di liberare il processore del carico di dover trasferire i dati lui uno ad uno verso la memoria principale, quindi il processore programma il DMA dicendo "fai questa operazione, per esempio una lettura da un certo dispositivo e le informazioni che hai letto le copi direttamente in memoria in questa zona, la zona di memoria dove avviene la copia tipicamente è una zona di memoria assegnata al processo, quindi quando voi fate una read dal disco e passate come input un buffer, questo buffer sta nella memoria virtuale del processo che però è mappata su pagine fisiche, per cui il sistema operativo può dare la richiesta al DMA di trasferire direttamente nelle pagine fisiche e il DMA a seconda di quanto è sofisticato può usare indirizzi fisici o logici, per questo motivo nel descrittore di pagina, le pagine che sono impegnate nella operazione di DMA vengono bloccate, c'è un bit particolare nel descrittore di pagina che è il bit di pinning che serve per dire che quella pagina non può essere rimossa finché non è stata completata l'operazione di I/O.





Quindi il DMA contiene il puntatore di buffer verso la memoria, ha un contatore che serve per gestire la copia e lui in realtà va a programmare il controller, per cui il controller legge i dati dal dispositivo li mette in un buffer interno e poi fa la copia direttamente in memoria tramite il DMA.

Il flusso delle operazioni è questo: la CPU programma il controller del DMA, il controller del DMA a sua volta richiede al controller del disco una certa operazione (per esempio dei dati alla memoria) Quindi il trasferimento di dati alla memoria avviene dal disco verso la memoria sotto il controllo del controller del disco, quando questa operazione è completata e il controller del disco (???????) DMA che a questo punto genera un'interruzione verso il processore e gli comunica che l'operazione è completata.



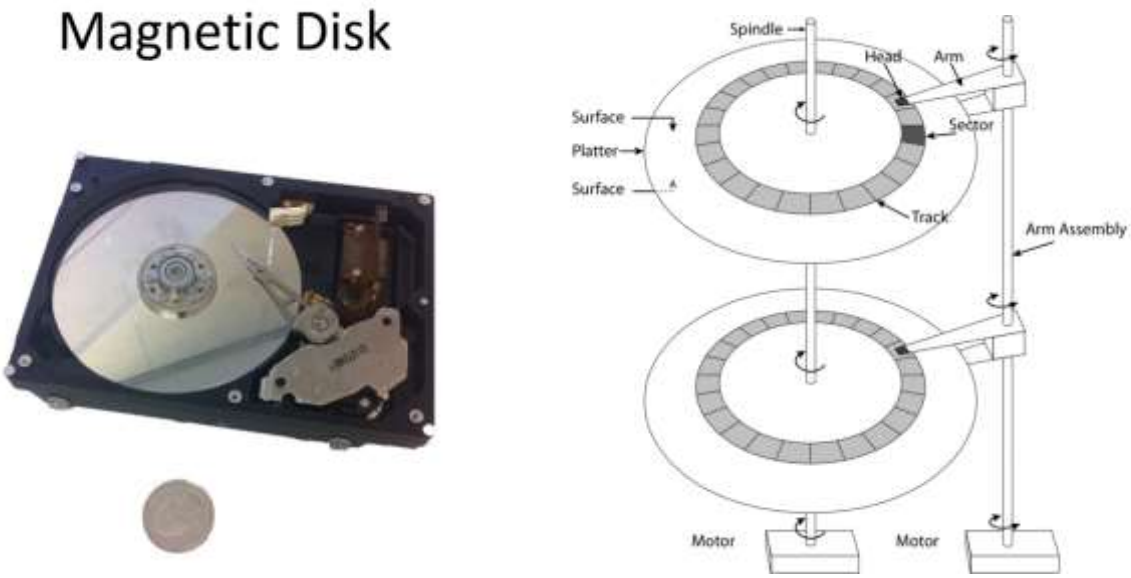
A questo punto, vediamo invece i dispositivi di memorizzazione. Allora, i primi che vediamo sono i dischi magnetici, in realtà i dischi magnetici stanno scomparendo da un utilizzo consumer quindi i vostri portatili hanno ancora qualche disco magnetico ma molti di questa generazione e tutti quelli della prossima non li avranno più. Nei cellulari, negli smartphone ormai si lavora soltanto con dischi di tipo SSD. Restano ancora invece nei server, soprattutto in quelli di fascia alta che in realtà possono usare diverse gerarchie di disco perché possono avere uno strato di dischi SSD e poi successivamente un altro stato di dischi magnetici, questo perché i dischi magnetici hanno ancora un costo molto basso e una capienza molto più alta, per cui se dovrete gestire un server che deve avere una capacità di memorizzazione enorme può essere conveniente usare i dischi magnetici come ultima risorsa e pooi in mezzo mettere un livello di dischi SSD che quindi in qualche maniera rendono tutto il sistema più veloce e i dischi SSD faranno da Cache per i dischi magnetici e poi la memoria e via discorrendo.

Quindi insomma, ha ancora senso parlare di dischi magnetici, tra l'altro i dischi magnetici erano dati per morti già nel '99 e come vedete ce li siamo ritrovati tra i piedi per un ventennio e ancora ce li abbiamo tra i piedi, quindi mai darli per spacciati.

Viceversa, le memorie flash sono anche queste memorie che non si corrompono facilmente, riescono a mantenere informazioni persistenti nel lungo tempo, hanno capacità a costi intermedi, ormai le memorie SDD avranno un costo doppio o triplo rispetto ai dischi rigidi, però il loro costo si sta abbastando notevolmente a causa della loro diffusione.

Comunque, che differenza c'è tra le memorie flash e i dischi magnetici? Beh, entrambi permettono accesso diretto però mentre nelle memorie flash l'accesso è rapidissimo, è come andare in memoria principale, nel caso dei dischi magnetici le prestazioni per l'accesso diretto sono un po' più lente, ma comunque sempre accettabili rispetto ad altri sistemi che sono intrinsecamente sequenziali come i cd o furono i nastri, quindi intendiamoci, lento o veloce in questo caso è sempre relativo, dipende con che cosa ci si compara.

D'altra parte le prestazioni dei dischi sono buone, se si fa un accesso in STREAM (in sequenza). Viceversa le memorie flash hanno una caratteristica un po' strana, hanno delle ottime prestazioni in lettura ma hanno delle prestazioni decisamente peggiori in scrittura, e questo è proprio un problema legato alla specifica tecnologia usata per questo tipo di memoria flash, in futuro questo potrebbe cambiare con nuove generazioni di memorie flash, allo stato attuale, questo è la situazione.



Perché è importante sapere come è fatto il disco? È importante perché altrimenti non si capisce perché poi viene gestito in un certo modo. Quindi mi rendo conto che una cosa che riguarda più i meccanici che gli informatici ma facciamo questo sforzo.

Come funziona? Sono una serie di piatti, poggiati tutti sullo stesso asse e ruotano tutti in sincrono, dopo di che hanno una testina in lettura/scrittura che si posiziona su ogni piatto

quindi in realtà su ogni disco abbiamo due testine, anche loro sincronizzate, e le testine si muovono lungo un raggio. Quindi per effetto della rotazione quando la testina è posizionata in un certo punto andrà naturalmente a leggere un settore circolare. Questo settore in realtà ha un nome, si chiama TRACCIA, per cui quando noi spostiamo la testina più avanti o più indietro in realtà ci posizioniamo su una Traccia differente.

Le informazioni nella traccia non sono scritte un Byte dietro l'altro perché sarebbe molto difficile da leggere ma sono organizzati a blocchetti che sono chiamati SETTORI.

Perché esistono i settori nel disco? Fate questo esperimento quando andate in autostrada, vi mettete un paraocchi da cavallo guardate lateralmente mentre state viaggiando velocemente e vedrete che davanti vi scorrono gli alberi dell'autostrada. È molto difficile riuscire a distinguere i dettagli di ciò che guardate in questa maniera, perché vi passano velocissimi sotto il naso. Per poter fare una lettura accurata, intanto la testina di lettura/scrittura si trova esattamente nella stessa situazione, lei è ferma e sotto gli stanno scorrendo delle informazioni e lei riesce a leggere un bit alla volta che le passa sotto il naso, il problema è che per poter fare la lettura accurata, la testina deve capire quando comincia e quando finisce ogni bit, e badate bene neanche voi ci riuscite quando vi passano gli alberi voi vedete una strisciata non vedete le singole foglioline, giusto? A meno che non siate più bravi di me.

Per cui, come fa la testina a distinguere i singoli bit se voi non ci riuscite?

Ovviamente la testina ha una frequenza molto elevata però deve sapere quando comincia e quando finisce ogni bit, perché se inizia la lettura sfasata non riuscirà a capire se ha letto un bit o se ha letto due bit, immaginate che ci sia una sequenza di 5 bit a 1, quando finisce di leggere dice "erano 5 o erano 4, erano 6?" è difficile. Per questo motivo ogni tanto deve essere risincronizzata quindi, che cosa succede? Ognuno di questi settori è fatto in questa maniera:

- Ha una prima fase di sincronizzazione, quindi non ci sono bit in questa fase ma ci sono delle sequenze di magnetizzazione della superfice tali per cui la testina riesce a capire con che velocità arriveranno poi dopo i bit, e riesce a sincronizzarsi con l'inizio e con la fine di ogni bit. Questa sincronizzazione non dura in eterno perché basta un minimo errore che alla lunga si desincronizzano, per questo all'inizio di ogni settore è necessario risincronizzarsi.

- Finita questa fase di sincronizzazione ci deve essere: il numero del settore che sta leggendo, il numero della traccia (la testina deve sapere se si trova su una traccia o su un’altra), quindi informazioni relative a ciò che si sta leggendo, almeno il numero di settore e il numero della traccia.

- Poi arrivano i veri e propri byte e poi alla fine di questi byte, siccome la lettura può dare diversi errori c'è un codice correttore d'errore.

Ora, se voi guardate quelle testine è come se voi prendeste una cosa grande come un elefante per leggere una fila di formiche; quella testina è molto più larga della traccia per cui quello che può succedere è che la testina posizionata in un certo punto vada a leggere un po' di più di quello che effettivamente serve, però siccome la traccia che gli interessa è quella al centro è quella che ha una forza maggiore, e quindi quella che alla fine viene letta o scritta. D'altra parte in questa operazione ci sono dei disturbi dati dalle tracce adiacenti per cui in fase di lettura si possono fare degli errori ed è per questo motivo che è bene avere dei codici correttori di errore per poter evitare di dover rileggere lo stesso settore più e più volte.

Quindi noi abbiamo che il disco è composto da diversi piatti, ogni piatto in realtà ha due facce, quindi in ultima analisi l disco è composto da un certo numero di facce in questo caso ne ha 4; su ogni faccia abbiamo le tracce e ogni faccia è divisa in settori. In realtà, spesso si usa anche quest'altra notazione perché le tracce che sono tutte omologhe (che si trovano alla stessa posizione nelle varie facce) se le prendete tutte assieme formano un cilindro, quindi l'insieme delle tracce che hanno tutte lo stesso indice nelle varie facce sono anche dette "cilindro". Quindi, se volete trovare un’informazione in questo disco dovete specificare su che faccia è, su quale cilindro si trova (equivalente a dire su quale

traccia si trova) e poi qual è il numero di settore. Ogni informazione è codificata quindi da una tripla: CILINDRO, FACCIA e SETTORE.

Un po' di parametri legati alle tracce, una traccia ha dimensione nell'ordine di 1micron (larghezza), tenete presente che la lunghezza d'onda della luca in media è di 0.5 micron. La risoluzione dell'occhio umano è di circa 50 micron, quindi vi faccio vedere che voi vedete la punta della testina ma non riuscite ad apprezzare poi la reale dimensione della testina che è molto più piccola ed è sotto, e la dimensione della traccia era (2008) di circa 100KB. Ora per evitare troppi problemi, le tracce sono separate tra loro da delle zone non utilizzate quindi non sono appiccicate l'una a l'altra ma sono distanziate proprio per ridurre la quantità di errori che fate quando andate a leggere, e la lunghezza della traccia varia lungo il disco perché le tracce più esterne sono ovviamente più lunghe e le tracce più interne sono ovviamente più corte, d'altra parte la velocità di rotazione è fissa questo vuol dire che la testina impiega lo stesso tempo a leggere una traccia più interna o una più esterna e questo vuol dire che sulla traccia più esterna, in realtà, la sua velocità rispetto alla superfice è maggiore (la sua velocità angolare è la stessa). D'altra parte siccome lo spazio che abbiamo nella traccia esterna è maggiore, possiamo far stare più settori nelle tracce esterne, e questo normalmente viene fatto nei dischi rigidi, perché nei dischi rigidi in realtà, le tracce non sono tutte omogenee le tracce più interne hanno un numero minore di settori e le tracce più esterne hanno un numero maggiore di settori.

Per evitare una situazione davvero ingestibile, ci sono 3 aree del disco, tutte le tracce dell'area interna hanno lo stesso numero di settori, le tracce dell'area media hanno un altro numero di settori e le tracce nell'area esterna ne hanno un numero ancora maggiore. Inoltre, la parte più interna del disco non viene utilizzata perché troppo piccola.

Il disco si chiama disco magnetico perché la scrittura dei bit viene fatta magnetizzando una parte della superfice, quindi la testina induce un campo magnetico che va a magnetizzare una parte della superfice oppure va a leggere il campo magnetico presente sul disco.

Sectors

Sectors contain sophisticated error correcting codes
  - Disk head magnet has a field wider than track
  - Hide corruptions due to neighboring track writes
- Sector sparing
  - Remap bad sectors transparently to spare sectors on the same surface
- Slip sparing
  - Remap all sectors (when there is a bad sector) to preserve sequential behavior
- Track skewing
  - Sector numbers offset from one track to the next, to allow for disk head movement for sequential ops

I settori contengono dei codici di correzione perché la testina magnetica ha una larghezza maggiore della traccia e questo vuol dire che ci possono essere degli errori quando si va a leggere. Ora in realtà questi dischi possono avere dei problemi nel senso che alcune zone del disco durante il processo produttivo in fabbrica possano non essere magnetizzate in maniera perfetta, quindi può capitare che poi durante l'uso o già al primo uso alcune zone del disco risultino inutilizzabili. Questo è possibile e in effetti spesso capita, ma in realtà non è un problema perché il controller a basso livello via via che si rende conto che ci sono settori non utilizzabili, li scarta e può anche procedere ad una rinumerazione dei settori in maniera tale da mantenere la proprietà di adiacenza al fine di migliorare le prestazioni.

Tenete presente che il disco è molto quando voi accedete alle informazioni sequenzialmente perché in realtà l'unico movimento che dovete fare è aspettare la rotazione del disco, quindi le informazioni sulla stessa traccia vengono lette molto velocemente. Il disco diventa lento quando vi dovete spostare da una traccia all'altra; sapendo questo se ci sono dei settori rotti che potrebbero danneggiare la sequenzialità, il controller del disco può anche operare uno spostamento dell'informazione in maniera tale da non alterare troppo le prestazioni.

Sectors & blocks

- At low level the controller accesses individual sectors
  - Typical sector size is 256/512 bytes
  - Identified by a triple: <#cylinder, #face, #sector>
- The disk driver group a set of contiguous sectors in a block
  - Typical block size is 2/4/8 Kbytes
  - identified by a pointer on a contiguous address space (from 0 to max-blocks)

Quindi, per motivi ovvi, visto che è necessario sincronizzarsi con la velocità dei bit, usare dei codici correttori, le informazioni sul disco possono essere un settore alla volta, non potete scrivere 1 Byte nel disco, potete scrivere un settore per intero come pure potete leggere un settore intero. La dimensione del settore tipicamente è 256/512 Byte e come vi dicevo prima è definita e indirizzata dalla tripla (# faccia, # cilindro, # settore). Questo modo però di accedere alle informazioni del disco perché per tanti motivi il sistema operativo vorrebbe gestire le informazioni con strutture che per lui sono più adeguate. Quindi, a basso livello la lettura o scrittura avviene sul singolo settore, per sua comodità il file system, raggruppa il settore in blocchi, per cui a livello di file system, la lettura è sul singolo blocco.

Tipicamente un blocco può essere di 2/4/8 KB e la dimensione del blocco dipende dalle necessità del sistema operativo, slegata invece dalle proprietà fisiche del supporto che prevedono invece dimensione dei settori differenti. I blocchi tra l'altro non sono indicizzati con una tripla (faccia,clindro,settore) ma sono indicizzati con un numero di sequenza, quindi dal punto di vista del file system i blocchi sono un array numerati da 0 fino ad un numero massimo; concretamente ogni blocco di mappa su un gruppo di settori caratterizzati da indirizzi in forma cilindro, faccia, settore.

Sectors & blocks

Given a block number b, and a triple $< t, f, s >$:

$$b = t(\#faces \cdot \#sectors) + f(\#sectors) + s$$

- $\#faces$ is the number of faces in the disk
- $\#sectors$ is the number of sectors per trace

Consequently:

$$t = b \; div \; (\#faces \cdot \#sectors)$$

$$f = \big(b \; mod \; (\#faces \cdot \#sectors)\big) div \; \#sectors$$

$$s = \big(b \; mod \; (\#faces \cdot \#sectors)\big) \; mod \; \#sectors$$

Se ipotizziamo che il blocco abbia la stessa dimensione del settore per fare comodamente questo calcolo, altrimenti basta usare un piccolo fattore di scala. Cmq se ipotiziamo che il blocco corrisponda al settore, dato un numero di settore "b" e la tripla (cilindro, faccia e settore) il numero del blocco è dato da

cilindro(#facce*#settori)+faccia(#settori)+settore

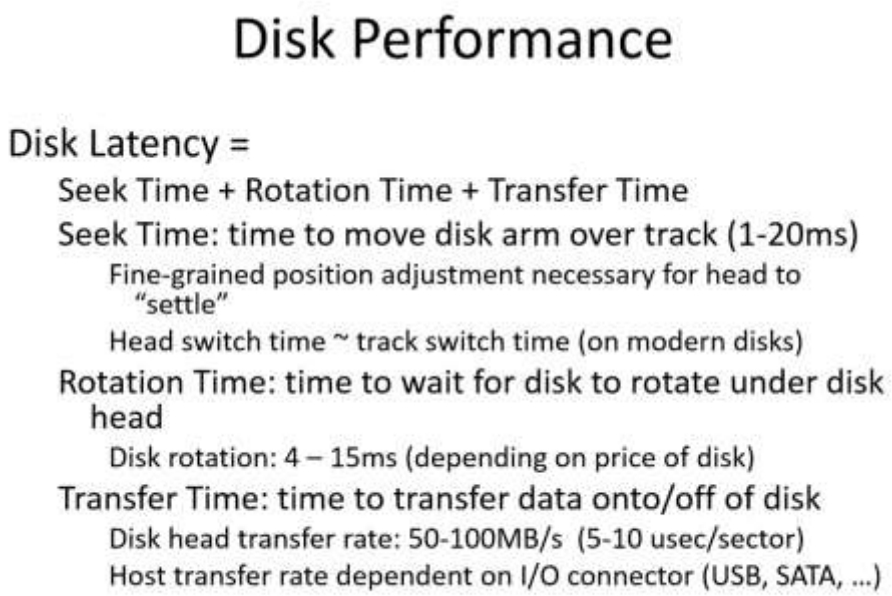
e di conseguenza se io so il numero del blocco ottengo:

cilindro= blocco/(#facce*#settori)

faccia=(blocco mod(#facce*#settori))/ #settori

settore=(blocco mod(#facce*#settori)) mod #settori

Questo tipo di associazione ha un vantaggio perché blocchi vicini si ritrovano in settori adiacenti e questo è un grosso vantaggio, perché quando vado a memorizzare le informazioni dei file spesso le metto in blocchi sequenziali e di conseguenza siccome blocchi che sono sequenziali sono anche molto probabilmente su settori sequenziali l'accesso sequenziale al file diventa efficiente e per certi aspetti anche l'accesso diretto.



Come variano a questo punto le prestazioni del disco? In base a che cosa si possono stimare?

Noi sappiamo che per raggiungere un'informazione dobbiamo portare la testina sulla traccia giusta, poi dobbiamo aspettare che la rotazione del disco ci porti il settore che vogliamo leggere sotto la testina e poi una volta che il settore arriva sotto la testina allora dobbiamo effettivamente leggere, prendere quei byte e trasferirli verso la memoria. Quindi se vogliamo sapere quanto tempo impiega l'operazione dobbiamo sapere questi 3 tempi che sono:

- Tempo di seek, il tempo che serve per spostare la testina sulla traccia

- Tempo di rotazione, che serve per aspettare il settore che passi sotto la testina

- Tempo lettura settore e trasferimento

Il tempo di seek, ovviamente varia a seconda delle tecnologie, del disco ecc... Può variare dall'1 ai 20 millisecondi, dipende oltretutto anche da quanto lontano stiamo andando, se la testina è attualmente nella traccia 0 e vogliamo andare nell'ultima traccia, dobbiamo attraversare tutto il disco, quindi il tempo di seek sarà ovviamente maggiore rispetto al caso in cui dobbiamo raggiungere un settore vicino, quindi questo tempo di seek varia e dipende dalla tecnologia.

Possiamo andare nell'ordine del 1/ 20 millisecondi a seconda dei casi.

Il tempo di rotazione invece, dipende dal disco; ci sono dischi che girano a 5600 giri al minuto, dischi che girano a 7200 giri al minuto e difficilmente si va a velocità di rotazioni molto più alte.

Il tempo di rotazione può essere tra i 4 e i 15 millisecondi, dipende da quanto è veloce la rotazione. Ovviamente però, questo è il tempo per fare un giro intero ma quando noi capitiamo su una traccia magari siamo fortunati e becchiamo subito il settore oppure dobbiamo aspettare una parte della rotazione, non necessariamente tutta. Quindi spesso quello che viene dato è il tempo **medio** per raggiungere il settore che è circa la metà del tempo di rotazione totale. Nella metà dei casi impiegate

poche rotazioni nell'altra metà dei casi siete sfortunati e impiegati tante rotazioni, in media aspettate mezza rotazione.

Infine il tempo di trasferimento, è davvero molto piccolo perché una volta che siete posizionati sul settore l'attraversamento del settore è molto rapido, è molto più piccolo del tempo di rotazione ovviamente. Il tempo di trasferimento è equivalente ad una frazione intera del tempo di rotazione e può essere dell'ordine dei 5/10 microsecondi a settore. Questo poi ovviamente dipende anche dalla velocità del connettore, se siete su dischi SERIAL ATA o USB ecc.

Vi porto questo esempio del 2008, ma non che sia particolarmente rilevante perché se prendete un disco più moderno potete prendere questi numeri e scalarli in qualche fattore, tanto il concetto non cambia.

Toshiba Disk (2008)

Size	
Platters/Heads	2/4
Capacity	320 GB
Performance	
Spindle speed	7200 RPM
Average seek time read/write	10.5 ms/ 12.0 ms
Maximum seek time	19 ms
Track-to-track seek time	1 ms
Transfer rate (surface to buffer)	54–128 MB/s
Transfer rate (buffer to host)	375 MB/s
Buffer memory	16 MB
Power	
Typical	16.35 W
Idle	11.68 W

Allora questo qui è disco che ha 2 piatti quindi 4 facce, una capacità di 320 GB, una velocità di rotazione di 7200 giri per minuto. Il tempo di seek medio per un'operazione di lettura/scrittura è nell'ordine di 10,5 millisecondi, qui dice che varia a 12 millisecondi, presumo che varia a seconda del modo col quale questo tempo di seek medio viene calcolato, queste sono le specifiche del costruttore, quindi magari hanno i numeri per calcolarlo. Il tempo massimo di seek è 19 millisecondi e il tempo minimo di seek per passare da una traccia a quella adiacente è di 1 millisecondo. Tenete presente che la testina per spostarsi da una traccia ad un'altra deve accelerare, raggiungere una velocità di crociera, poi decelerare e fermarsi sula traccia, per cui non potete prendere il tempo di seek e dividerlo a metà, perché in ogni caso dovete considerare il ritardo dovuto all'accelerazione e alla decelerazione. Per cui il tempo medio è un po' più alto della metà cruda del tempo massimo. Il rate di trasferimento dalla superfice del disco al buffer varia da 54-128 MB/s e il tempo di trasferimento dal buffer alla memoria del PC è di 375 MB/s, quindi è evidente che il collo di bottiglia non è questo. Il suo buffer in memoria è di 16 MB. Come alimentazione questo consumicchiava , l'assorbimento è di 16,35W, quando è in iddle è di 11.68W. Tenete presente che non so come si questo disco, ma ci sta che in iddle mantenga una

velocità di rotazione per evitare di dover riaccelerare da 0. Se fermate il disco lo dovete riportare a velocità di regime prima di poterlo utilizzare.

Domanda: Quanto ci si impiega a fare 500 letture a caso in ordine FIFO?

Siccome sono posizioni scelte a caso e sono eseguite nello stesso ordine in cui sono state generate, non c'è nessuna relazione tra una richiesta e quella dopo. Quindi per ogni richiesta voi che dovete fare? Dovete fare:

- Seek per posizionare la testina

- Aspettare metà rotazione in media

- Trasferimento

Quindi questo qui sono 500*(tempo medio di seek + tempo rotazione medio + trasferimento) se facciamo due conti:

Seek medio: 10,5 millisecondi

Tempo rotazione medio: 4,5 millisecondi

Tempo di trasferimento: 5/10 microsecondi

Quello che succede è che voi li sommate tutti, li dividete per 1000 perché sono millisecondi e li moltiplicate per 500, e quindi per fare 500 letture sparpagliate nel disco impiegate 7,3 secondi.

500*(10,5+4,5+0,01) /1000= 7,3 secondi

Vi sembra tanto o poco? È un'enormità.

Che cosa succede se io invece faccio la lettura di 500 settori sequenziali?

Se sono sequenziali, noi in realtà dobbiamo fare soltanto un seek e poi a quel punto tutti i settori sono uno dietro l'altro, quindi aspettato il mezzo tempo di rotazione da quel momento in poi troviamo tutti settori uno di fila all'altro, se abbiamo la fortuna che questi settori siano tutti sulla stessa traccia abbiam finito qui. Se i settori sono sparpagliati su più tracce (saranno tracce adiacenti) e dovremmo aggiungere un tempo di seek e di rotazione per ogni passaggio di traccia. Il tempo di trasferimento completo è 500*512byte / 128 MB/s = 2ms.

500*512 ci dice quanti byte dobbiamo leggere in totale e siccome 128 MB/s è il throughput , allora bisogna dividere, per ottenere il tempo totale.

Quindi: se noi prendiamo gli stessi valori di prima, otteniamo:

10,5+4,15+2 =16,7ms

Quasi un fattore di 500 a 1 rispetto alla lettura in ordine casuale. Se poi per caso tutti questi settori non sono sulla stessa traccia, ma sono su due tracce adiacenti dobbiamo aggiungere un ulteriore 1ms per raggiungere la traccia adiacente.

Quindi insomma il tempo in ogni caso è di 16,7ms o poco più. Che cosa ci dice tutto questo? Ci dice che dobbiamo stare attenti a dove mettiamo le informazioni, perché a seconda di come il file system mette le informazioni nel disco potete ottenere delle prestazioni pessime o potete ottenere delle prestazioni favolose.

Forse voi non ve ne rendete più conto, perché tutti i file system di tutti i sistemi operativi più moderni utilizzano tutta una serie di politiche e di stratagemmi per fare in modo che tutte le informazioni siano adiacenti, però quando i vecchi sistemi operativi utilizzavano il vecchio file system FAT, in quei file system che erano molto elementari, non c'erano criteri sensati per distribuire le informazione del disco e questo voleva dire che dopo un di tempo che ci lavoravate soprattutto se il disco era piuttosto pieno le informazioni si trovavano tutte sparpagliate in diversi punti del disco, un singolo file poteva occupare settori in posizioni molto distanti nel disco e le prestazioni degradavano paurosamente.

Quindi ha senso per il file system porsi come utilizzare le informazioni inteso come distribuire i dati, i file, all'interno del disco in maniera tale da garantire queste prestazioni e non quelle altre.

Question

- How large a transfer is needed to achieve 80% of the max disk transfer rate?

  Assume y rotations are needed, then solve for y:

  0.8 (10.5 ms + (1ms + 8.4ms) y) = 8.4ms y

  80% of time required to read y tracks with overhead =
  = time to read y tracks without overhead

Total: y = 9.1 rotations, 9.8MB

Un’altra domanda: Quanto deve essere grande un trasferimento dati dal disco alla memoria per poter garantire l'80% della prestazione massima del disco?

Supponiamo di dover fare Y rotazioni, perché potrei dover leggere un’intera tracciai poi un’altra traccia, potrei dover leggere Y tracce per raggiungere l'80% delle prestazioni, potrebbe non bastarvi un'unica traccia.

Se leggo Y tracce, in tempo di accesso per leggere interamente Y tracce è: 1 seek iniziale per raggiungere la prima traccia e poi per ogni traccia che vado a leggere dovrò pagare un seek breve (1ms) + tutto il tempo di rotazione della traccia per leggerla, non ho bisogno di aspettare il tempo di rotazione per andare al primo settore perché se tanto leggo tutta la traccia, posso leggerla da qualsiasi posizione poi, una volta che ce l'ho in memoria la riordino. Quindi questo è il tempo reale che impiego a leggere Y tracce, la massima prestazione invece è data 8,4*Y. Se io leggo Y tracce senza pagare nessun overhead il tempo di lettura di queste Y tracce è dato dal tempo di rotazione che è 8,4ms moltiplicato per Y. Io voglio raggiungere l'80% di questo, quindi quello che succede se io risolvo per Y questa equazione è che viene fuori Y= 9.1 rotazioni quindi questo equivale a leggere 9.8 MB/s. Quindi se io riesco a fare un'operazione di circa 10 MB supero l'80% delle prestazioni nel disco. Per quanto possibile devo leggere le informazioni a blocchi non a singoli settori, perché sono troppo penalizzato.

Quindi sono avvantaggiato se le informazioni sono vicine e se le leggo a blocchi più grandi possibile; in queste condizioni, il disco ha delle buone prestazioni, possiamo dire anche ottime, altrimenti le prestazioni degradano.

Quindi è importante, far due cose:

- organizzare le informazioni su disco in maniera efficiente

- trovare degli algoritmi di scheduling che se ne caso noi ricevemmo più richieste contemporaneamente, riordinino queste richieste in maniera tale da eseguirle nel minor tempo possibile.

Disk Scheduling

- FIFO
  - Schedule disk operations in order they arrive
  - Downsides?

Quindi ritorniamo agli algoritmi di scheduling stavolta però applicati al disco e nuovamente ci si presentano i soliti casi, per esempio il FIFO che lo è lo scheduling base, però sappiamo già che il FIFO non è un'idea particolarmente brillante. Tenete presente che il sistema operativo riceve richieste di lettura di blocchi dal disco da parte di più thread o di più processi. Ogni processo andrà a chiedere le informazioni che vuole e non ci sarà coerenza tra queste; quindi le richieste che provengono da processi differenti presumibilmente saranno come se scelte a caso nel disco, quindi senza un criterio e questo sappiamo che con lo scheduling FIFO porta a cattive prestazioni.

Disk Scheduling

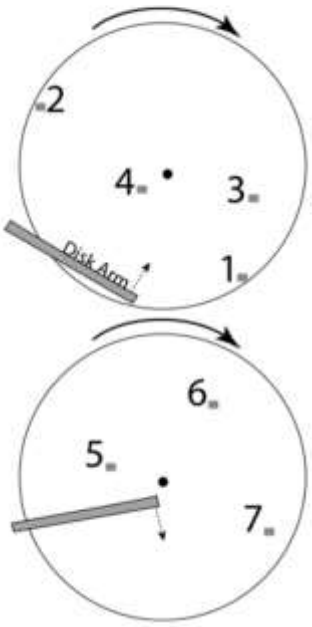
- Shortest seek time first
  - Not optimal in response time
    - … suppose two clusters of requests at far end of disk
  - Downsides?

Un esempio di tecnica che può aver senso è la SHORTEST SEEK TIME FIRST che ricorda un po' la SHORTEST JOB FIRST dei processori. La SSTF che cosa dice? Siccome quello che pesa molto è il tempo di seek (tempo di rotazione non lo posso accelerare e quello di trasferimento è trascurabile) vado a servire prima le richieste che sono su tracce più vicine; quindi riordino le richieste pendenti sulla base del tempo di seek rispetto alla posizione attuale. Il problema però è che con questo approccio ho molta disparità nel servire le richieste, e poi oltretutto rischio che le richieste che si riferiscono ai lati estremi del disco vadano in STARVATION perché se continuano ad arrivare richieste per tracce intermedie, continuerò a servire quelle e non servirò mai le altre.

Quindi in realtà si utilizzano altre tecniche quali la SCAN che è anche nota come algoritmo dell'ascensore.

Disk Scheduling

- SCAN: move disk arm in one direction, until all requests satisfied, then reverse direction

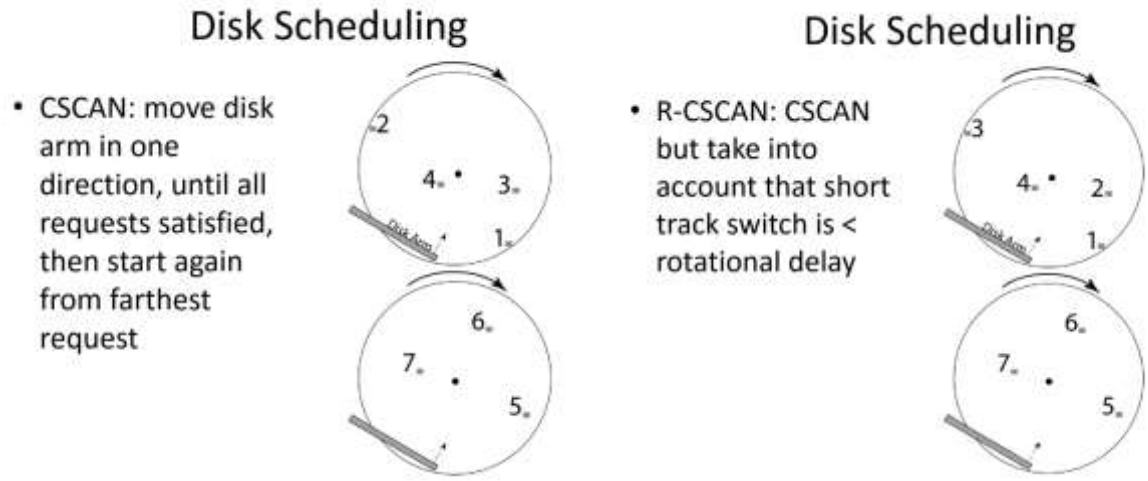


Quindi in realtà si utilizzano altre tecniche quali la SCAN che è anche nota come algoritmo dell'ascensore.

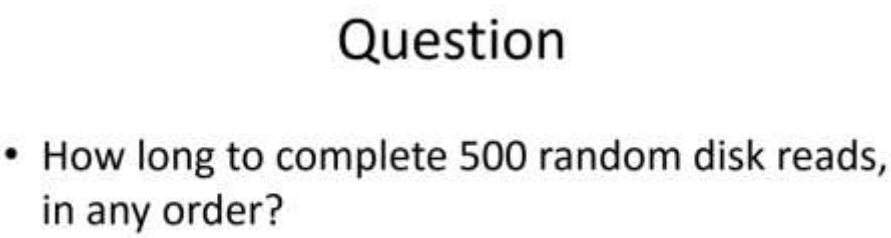
Lavora in questa maniera: la testina scandisce le tracce (per esempio dall'indice più basso all'indice più alto) e serve tutte le richieste pendenti via via che le incontra. Quando ha completato questa operazione inverte la direzione e serve le richieste che incontra nell'ordine inverso.

Per esempio supponendo di avere queste richieste pendenti 1 2 3 4, la testina parte dal lato estremo, quindi serve prima 1 e 2 poi continuando a scorrere serve 3 poi continuando a scendere va a servire. Quando è arrivato a 4 inverte la direzione e a questo punto se ci sono di nuovo delle altre richieste pendenti che sono arrivate proprio in questo punto, le serve in ordine inverso a partire da 5 poi 6 e poi 7. Dopo che ha servito 1 e 2, e si è spostata verso 3 magari a quel punto è arrivata una nuova richiesta nella zona di 1 (nelle tracce basse) ma qui la testina non torna indietro per servirle ma continua nella sua direzione.

Anche con la SCAN ci potrebbe essere attesa indefinita, perché se si trova sulla traccia 3 e continuano ad arrivare richieste sulla stessa traccia, la testina rischia di restare lì per sempre, in attesa indefinita. Quindi in realtà con la SCAN, se sto servendo la traccia 3 ed arrivano nuove richieste per la traccia 3 le metto nel buffer e le servirò nella fase di ritorno.



Ci sono delle varianti della SCAN tipo la CSCAN che serve le richieste sempre nella fase di salita, poi quando ha finito la fase di salita ritorna alla traccia 0 e riparte di nuovo con la fase di salita. E poi c'è la R-CSCAN che è come la CSCAN però con un piccolo trucchetto; supponete che io debba ricevere richieste 1 3 2, ora mentre il giro sta ruotando io troverò prima 1 e poi dovrò servire 3 a quel punto dovrei fare una seek per andare a 2. Siccome però la traccia 2 è abbastanza vicina, dopo aver servito 1 potrei fare una piccola seek per andare a 2 e poi faccio un'altra seek per andare a leggere 3, siccome il tempo di rotazione è più lungo del tempo di seek dovrei riuscire (se riesco a fare questo piccolo doppio salto) prima di arrivare a 3, a soddisfare 2 e così risparmio tempo.



Supponiamo di riprendere la richiesta di prima: *Quanto tempo impiego a soddisfare 500 richieste scelte in ordine casuale?*

Prendiamo per esempio il CSCAN, (ora questo conto è fatto davvero in maniera molto grossolana), se sono 500 richieste molte di queste saranno vicine come tempo di seek, perché se posso andare a servire richieste vicine il tempo di seek sarà più probabilmente 1ms piuttosto che altro, per ogni richiesta dovrò comunque pagare mezzo tempo di rotazione, questo vuol dire che il tempo che impiego sarà, molto probabilmente, per ogni richiesta un tempo di rotazione+1ms di seek, ipotizzando di metterli in ordine in base alla distanza dalla testina e questa cosa qui mi produce un tempo atteso di 2,2 secondi.

Question

- How long to complete 500 random disk reads, in any order?
  - Disk seek: 1ms (most will be short)
  - Rotation: 4.15ms
  - Transfer: 5-10usec
- Total: 500 * (1 + 4.15 + 0.01) = 2.2 seconds
  - Would be a bit shorter with R-CSCAN
  - vs. 7.3 seconds if FIFO order

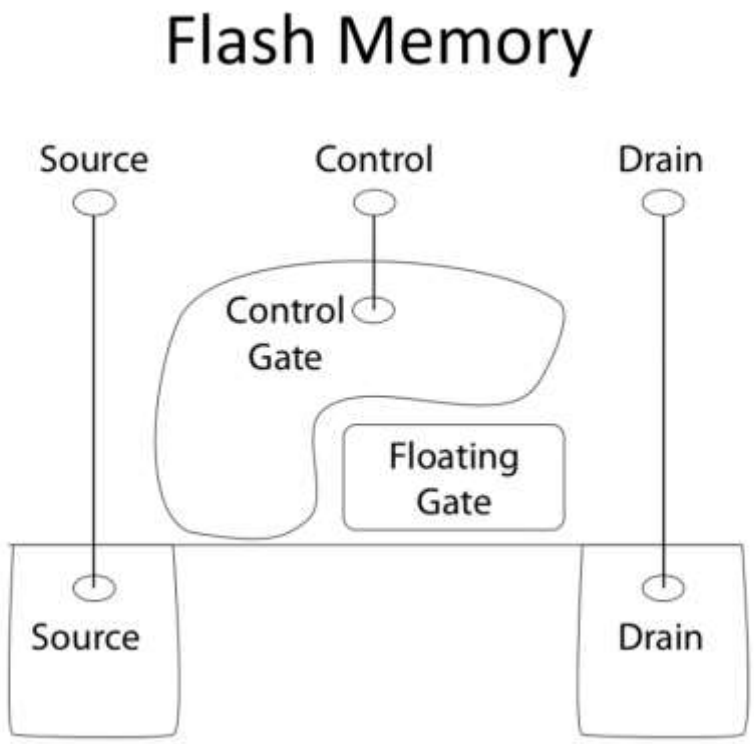
Quindi soltanto cambiando l'algoritmo di scheduling ho più che triplicato le prestazioni del disco.

Un'altra domanda giusto per curiosità: *Quanto ci si mette a leggere tutto il disco?*

Se il disco ha la capacità di 320 GB e la banda di accesso al disco è di 128 MB/s il tempo di trasferimento è dato dalla capacità/ banda media.

320GB/128MB = 1 ora

Con questo chiudo il disco.



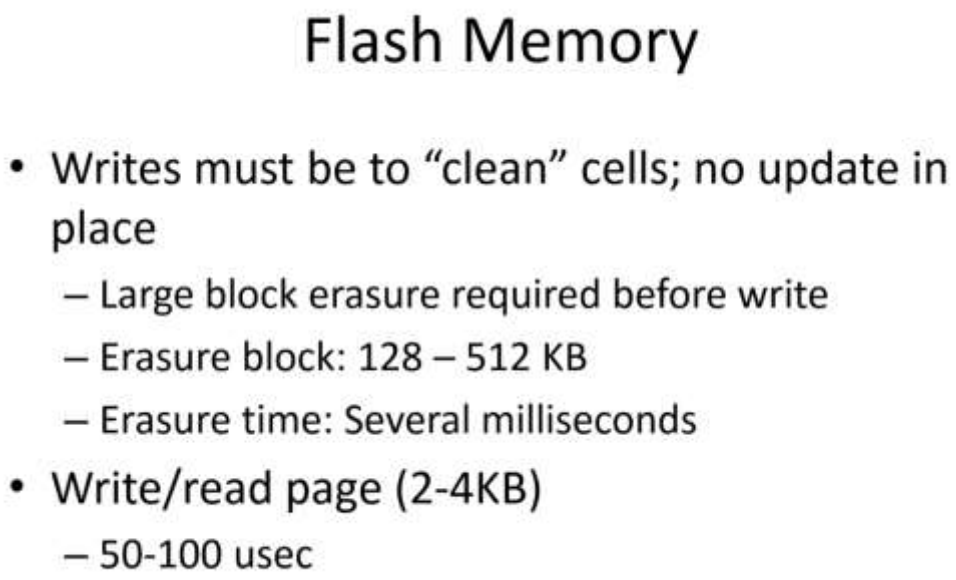
Passiamo adesso alle memorie flash. Intanto come sono fatte le memorie flash? Di memorie flash ce ne sono di diverso tipo; questo schema vi mostra come funziona la memorizzazione di un bit in una memoria flash con le tecnologie attuali. Tenete presente che c'è un'evoluzione fortissima, già ora esistono delle tecnologie differenti da questa e si va avanti così, nel giro di poco tempo avremo una nuova generazione di memorie flash.

Questo qui sarebbe lo schema di un transistor che funziona in questa maniera: un transistor ha 3 piedini (sorgente, controllo e drain). Voi potete regolare il flusso di corrente dalla sorgente al drain dando corrente sul piedino di controllo, praticamente se questo è alimentato potete interrompere il passaggio oppure potete permetterlo.

Nel caso delle memorie flash l'idea che sta alla base è che al control è attaccata questa **floating gate** che è isolata elettricamente dal controllo. Vi faccio notare che comunque il controllo è isolato elettricamente dalla sorgente e dal drain, semplicemente la presenza di tensione sul controllo inibisce il passaggio di corrente tra questi due poli. La floating gate è isolata elettricamente però, per un principio fisico degli elettroni possono essere intrappolati qua dentro, per cui se non abbiamo intrappolato qua dentro degli elettroni, è come se avessimo memorizzato un bit qua dentro che può essere 1 o 0, di conseguenza se qui non ci sono intrappolati elettroni il passaggio di corrente quando viene alimentato dalla sorgente al drain avverrà, se invece qui sono intrappolati elettroni il passaggio di corrente, non avverrà. E questo ci permette di leggere l'informazione sul fatto che il passaggio avviene o non avviene, e quindi leggere 1 o 0.

Per poter scrivere dovrei intrappolare degli elettroni qua dentro, per poter leggere invece non devo fare niente, mi limito soltanto ad alimentare la sorgente e il drain per vedere se passa corrente. Il risultato di tutto questo, per quanto strano possa sembrare è che le letture sono agevoli, facili ma la

scrittura che è complicatissima (perché per poter scrivere un bit prima devo azzerare la floating gate. L'operazione di azzeramento della memoria flash è costosa e viene fatta a blocchi) è lenta.



Le scritture devono essere su celle pulite, non posso aggiornare e la cancellazione (la pulizia delle celle) avviene a blocchi (128-512 KB) e il tempo di cancellazione è nell'ordine di diversi ms. Quindi, quando vado a scrivere su una cella dovrei, in teoria, prima cancellare tutto il blocco grande e poi riscriverlo; non è pensabile. Per questo motivo quello che succede in queste memorie è che se voglio scrivere una cella in realtà smetto di utilizzarla prendo una cella già cancellata e scrivo in quella cella. Per questo motivo le informazioni in scrittura si spostano in continuazione. Quindi la cancellazione è nell'ordine di diversi ms, mentre invece se faccio una lettura, di nuovo, non posso lavorare sul singolo byte ma devo usare i blocchi.

La lettura porta un tempo nell'ordine dei 5-10 microsecondi, la scrittura fatta su una pagina CANCELLATA impiega lo stesso tempo.

Vediamo alcuni parametri di un flash drive:

Flash Drive (2011)

Size	
Capacity	300 GB
Page Size	4KB
Performance	
Bandwidth (Sequential Reads)	270 MB/s
Bandwidth (Sequential Writes)	210 MB/s
Read/Write Latency	75 μs
Random Reads Per Second	38,500
Random Writes Per Second	2,000 (2,400 with 20% space reserve)
Interface	SATA 3 Gb/s
Endurance	
Endurance	1.1 PB (1.5 PB with 20% space reserve)
Power	
Power Consumption Active/Idle	3.7 W / 0.7 W

Flash Memory

- Writes must be to "clean" cells; no update in place
  - Large block erasure required before write
  - Erasure block: 128 – 512 KB
  - Erasure time: Several milliseconds
- Write/read page (2-4KB)
  - 50-100 usec

Le memorie flash sono delle tecnologie che vanno a sostituire progressivamente i dischi rigidi, quelli che sono basati sul piatto magnetico sul quale vengono memorizzate le informazioni per magnetizzazione della superficie. Nelle memorie flash invece le informazioni vengono memorizzate andando a intrappolare degli elettroni all'interno della Floating Gate. Il risultato di questa tecnologia è che per poter leggere non c'è nessun problema, la lettura del singolo bit avviene agevolmente, per quanto riguarda invece la scrittura, questa può avvenire soltanto dopo che la porta è stata cancellata, quindi quando si opera in scrittura bisogna prima cancellare. Generalmente le memorie flash sono organizzate in questa maniera: le cancellazioni sono molto costose e si fanno operando su banchi di memoria un po' grossi, nell'ordine dei 128-512 KB e portano via un tempo piuttosto lungo, nell'ordine dei diversi millisecondi. Mentre invece le letture e le scritture avvengono a blocchi (vengono dette pagine a questo livello), sarebbero l'equivalente dei settori nel caso delle memorie flash che hanno la dimensione di 2-4KB, in questi casi il tempo necessario per l'operazione è nell'ordine dei 50-100 usec (microsecondi). Anche le scritture impiegano più o meno lo stesso tempo, a patto che la pagina sulla quale si scrive sia stata già cancellata a priori. Se non è stata cancellata bisogna prima cancellarla, il problema è che siccome si cancella a blocchi, non cancello soltanto la pagina sulla quale voglio scrivere, ma devo cancellare tutte le pagine che stanno nello stesso blocco, per cui questo porta dei problemi di gestione del disco.

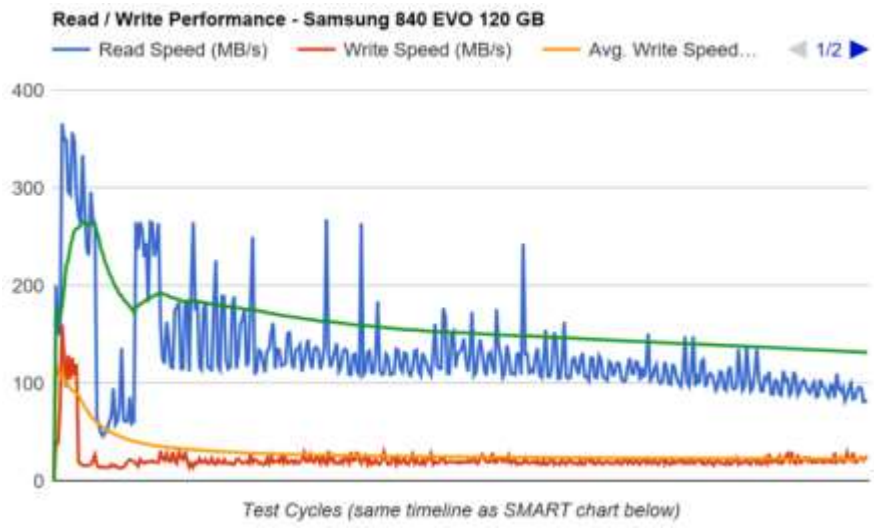
Flash Drive (2011)

Size	
Capacity	300 GB
Page Size	4KB
Performance	
Bandwidth (Sequential Reads)	270 MB/s
Bandwidth (Sequential Writes)	210 MB/s
Read/Write Latency	75 μs
Random Reads Per Second	38,500
Random Writes Per Second	2,000 (2,400 with 20% space reserve)
Interface	SATA 3 Gb/s
Endurance	
Endurance	1.1 PB (1.5 PB with 20% space reserve)
Power	
Power Consumption Active/Idle	3.7 W / 0.7 W

Qui ci sono dei parametri di un disco flash del 2011, un disco da 300GB che ha una banda in lettura o scrittura piuttosto elevata (270/210 MB/s), una latenza di lettura/scrittura su pagina già cancellata di 75 usec (microsecondi), se noi su questo disco andiamo a fare letture o scritture in ordine casuale, viene fuori la differenza: in particolare il numero di letture che possiamo fare ad accesso casuale per secondo sono nell'ordine di 38.500, il numero di scritture ad accesso casuale che facciamo per secondo sono nell'ordine

di 2.000, da dove viene fuori questa differenza? Viene fuori dal fatto che questo numero nasconde la necessità di fare le erasure di tanto in tanto, quindi le scritture fatte ad accesso casuale (non è un pattern molto ricorrente nei sistemi reali, normalmente le cose non funzionano così), però se noi testiamo il disco andando a fare delle scritture in ordine casuale, finché andiamo a scrivere su una pagina cancellata, bene, quando ci capita di dover scrivere una pagina che non è cancellata, dobbiamo prima procedere con la erasure e poi dopo la possiamo scrivere e da qui viene fuori questo rate di scrittura per secondo molto diverso. Un altro parametro interessante per questi dischi è la Endurance (la durata), può essere misurata in tante maniere, il fatto è che queste memorie flash non possono essere sovrascritte in eterno, dopo un certo numero di scritture la singola pagina perde le sue proprietà e non è più utilizzabile. Il numero di scritture che si possono fare sul singolo bit sono piuttosto basse (nell'ordine del migliaio), per questo motivo in realtà il controller del disco SSD opera cercando di ripartire le scritture in maniera omogenea su tutte le pagine in maniera tale da allungare il più possibile la durata. Vengono fatti dei test di durata di questi dischi per vedere quanto effettivamente possono essere utilizzati, in questo caso la durata è misurata in numero di byte che possono essere scritti, il disco è di 300GB, ma voi potete scrivere/sovrascrivere idealmente all'infinito, in realtà dopo aver scritto in totale 1,1 Petabyte, o in condizioni migliori, 1,5 Petabyte se abbiamo l'accortezza di riservare il 20% di spazio libero), dopo questo numero di scritture le prestazioni del disco degradano a tal punto che non è più utilizzabile. Questo problema di degrado non ce lo abbiamo soltanto per i dischi SSD, ma ce lo abbiamo anche per i dischi magnetici, proprio due giorni fa ho incontrato un collega al CNR che appeso al muro ha un disco magnetico: il singolo piatto è di vetro in quel disco e sopra c'è lo strato magnetico, lo strato magnetico su tutta una fascia esterna è stato completamente piallato ed è rimasto soltanto un vetro, effettivamente è trasparente, comunque quel disco ancora funzionava, ma lo spazio di memorizzazione era ridottissimo perché erano rimaste soltanto le tracce più in basso, tutta la fascia di tracce esterne era completamente inesistente. Quindi anche sui dischi magnetici abbiamo un problema di durata, la differenza è che nei dischi magnetici possiamo scrivere molto di più, quindi durano molto di più e normalmente non ci si pone il problema, nel caso invece di dischi SSD il degrado avviene prima e di conseguenza si fanno questi test. Su internet trovate alcuni test di durata di alcuni dischi SSD, ne ho preso uno a caso da questo sito:

http://ssdendurance.com/ssd-endurance-test-report/Samsung-840-EVO-120



Questi sono test fatti apposta per mettere molto il disco sotto stress, quindi il tipo di utilizzo fatto da questi test non combacia con l'utilizzo usuale, in un utilizzo usuale si vanno a scrivere 20GB al giorno (più o meno, comunque non è così drammatico), in ogni caso vedete da questo test che in seguito a tutta una serie di letture/scritture fatte in continuazione, le prestazioni potrebbero degradare sia in lettura/scrittura. C'è questo degrado perché alla lunga il sistema si trova a dover gestire settori danneggiati che deve escludere, via via che scrivete si trova a gestire la cancellazione di blocchi, le informazioni potrebbero essere sparpagliate in varie maniere all'interno del disco e chiaramente se deve fare una cancellazione, deve assicurarsi che tutte le pagine che stanno in un blocco di cancellazione siano libere altrimenti non può procedere con la cancellazione, per questo motivo le prestazioni col tempo decadono.

Flash Translation Layer

- To avoid the cost of an erase for each write, pages are erased in advance
  - Clean pages always available for a new write
- This means that a write cannot be directed to an arbitrary page in the memory, but only to one previously erased page
- What happens if you rewrite a block of a file?
  - The page storing the block cannot be rewritten immediately…
  - It need to be erased first but with surrounding pages!
  - A clean page is used to apply the write, but this page is somewhere in the disk…
    - The old page goes to garbage for recycling
- How to know where the pages of my file are?

In particolare riguardo alla cancellazione abbiamo un problema piuttosto grosso: posso cancellare in blocchi di 128-512KB, ma le pagine sono di 2-4KB. Immaginiamo di avere allocato un file contiguo nel disco e su questo file a un certo punto volete cambiare un byte, per cambiare un byte dovete sovrascrivere una pagina, il problema è che tutte le pagine adiacenti alla pagina che vogliamo sovrascrivere sono occupate dal file, quindi non possiamo sovrascriverle, dovremmo cancellare 512KB del file e poi riscriverlo nuovamente. Questo in realtà non viene fatto così ma si utilizza nei controller di questi dischi uno strato intermedio di traduzione che è in grado di rimappare i blocchi fisici del disco in blocchi virtuali, quindi spostare i blocchi da una parte all'altra a seconda di dove è conveniente farlo, quindi quando avete una scrittura in questa situazione, in mezzo a un file, la cosa più semplice da fare è operare la scrittura su un altro blocco e segnarsi il fatto che quel blocco di quel file è stato spostato da un'altra parte nel disco, poi successivamente quella pagina che prima apparteneva al file, ora non gli appartiene più, viene marcata come riutilizzabile e quando poi sarà possibile verrà cancellata e riutilizzata. In qualche maniera il controller del disco deve mantenere un processo di Garbage Collection per cui le pagine sovrascritte in realtà vengono spostate su altre pagine, sparpagliate, e il Flash Translation Layer deve ricordarsi quali sono le pagine che son state liberate, quando un gruppo di pagine liberate diventa abbastanza grande da coprire tutto un blocco di erasure, a questo punto quelle pagine possono essere cancellate e da quel momento in poi possono essere riutilizzate per le scritture.

Flash Translation Layer

- Flash device firmware maps logical page # to a physical location
  - Move live pages as needed for erasure
    - Garbage collect empty erasure block by copying live pages to new location
  - Wear-levelling
    - Can only write each physical page a limited number of times
  - Avoid pages that no longer work
- Transparent to the device user

Per fare quest'operazione, quindi per sapere dove stanno fisicamente le pagine il Flash Translation Layer contiene una mappa che mappa il numero di pagina su una locazione fisica. Questo Flash Translation Layer cerca anche di equilibrare le scritture per cui se dobbiamo fare una scrittura su una pagina, cerca una pagina che è stata scritta meno volte, in maniera tale da bilanciare il numero di scritture in maniera equa su tutte le pagine, per evitare che alcune pagine collassino troppo rapidamente, se alcune pagine collassano rapidamente poi si produce un effetto a catena sulle altre per cui le prestazioni del disco si riducono anzitempo, mentre invece se noi bilanciamo le scritture su tutte le pagine, ci sarà un momento in cui il disco improvvisamente collasserà, però questo istante sarà spostato un po' più avanti nel tempo. Questo livello oltretutto gestisce le pagine che non funzionano più, quindi quelle le mette in una BadList e non le utilizza. Tutto questo avviene in maniera trasparente. Trasparente vuol dire che è trasparente anche allo stesso sistema operativo. Il sistema operativo è guidato dal controller, il controller implementa il Flash Translation Layer per realizzare tutte queste funzionalità e dare visione al sistema operativo di un disco come una sequenza di pagine utilizzabili. Su questo array di pagine il sistema operativo, il file system nello specifico, va a memorizzare i suoi file, le sue strutture dati, ecc, ma non si cura direttamente di questo problema, questo è risolto interamente ad hardware.

File System – Flash

- File systems on magnetic disks do not need to tell the disk what blocks are in use:
  - When a block is no longer used it is marked free in the bitmap
  - The file system reuse it whenever it wants
- When these FS were used first on flash drives the performances decayed over time

Quindi il fatto di mantenere un'indipendenza tra controller e file system, ovviamente semplifica di molto le cose ai progettisti dei sistemi operativi, separa i problemi, quindi semplifica la complessità di gestione dell'intero sistema. Quest'approccio ha creato un problema grosso perché quando i dischi flash son stati introdotti, il primo impulso è stato quello di utilizzarli con i file system esistenti all'epoca, quindi con NTFS, FAT, FFS e via discorrendo. Questi file system in realtà non si curavano, erano stati progettati per dischi rigidi, non si curavano dei problemi interni dei dischi flash, in particolare ignoravano completamente la gestione che il Flash Translation Layer doveva fare delle pagine. Cosa succedeva in realtà? Quando

cancellate un file, il sistema operativo in una sua struttura dati segna che i blocchi associati a quel file sono disponibili/liberi, ma quest'informazione la sa il sistema operativo, il file system, non viene comunicata al controller, d'altra parte un disco rigido convenzionale non sa che farsene di quest'informazione, per lui è irrilevante. Se invece anziché avere un disco rigido convenzionale, sotto avete un disco flash, qual è il problema? Il problema è che il file che avete cancellato, dal punto di vista del controller nell'SSD non è stato cancellato, sono ancora blocchi utilizzati perché son scritti, nessuno gli ha detto che quei blocchi non li usa più nessuno. Quindi dal punto di vista del controller dell'SSD, quei blocchi sono occupati e questo vuol dire che quando deve organizzarsi per liberare spazio per fare una erasure, ci sta che ogni tanto si metta a spostare dei blocchi scritti da qualche altra parte nel disco per appunto poter fare la erasure. Questo lavoro coinvolge anche blocchi/pagine che in realtà non sono più in uso ma lo sa soltanto il file system, non lo sa il controller. Il risultato è che, applicati i file system convenzionali su questi dischi, le prestazioni collassavano in maniera inspiegabile molto rapidamente, dal punto di vista del controller era come operare con un disco quasi completamente pieno, quindi riuscire a spostare le pagine per avere spazio sufficiente per fare una erasure diventava un problema.

File System – Flash

- The Flash Translation Layer got busy on garbage collection:
  - Live blocks must be remapped to a new location …
  - … to compact the free pages in order to proceed with block erasure
- This even with large amount of free space
  - For example, if the FS move a large file from a range of blocks to another one …
  - … the storage does not know that the old blocks can go to garbage …
  - … unless the FS tells it!

Lo storage, quindi il controller non sa quali sono i blocchi vecchi che possono essere recuperati (andare al Garbage Collection) a meno che il file system non glielo dica. Nel 2011 proprio per questo motivo è stato introdotto un comando aggiuntivo, una funzione aggiuntiva che è la TRIM:

File System – Flash

- TRIM command
  - File system tells device when pages are no longer in use
  - Helps the File Translation Layer to optimize garbage collection
  - Introduced in between 2009/2011 in most OSs

Tramite la TRIM il file system può comunicare al controller il fatto che certe pagine non sono più in uso, quindi quando cancellate un file oltre a segnarlo nelle sue strutture dati, il file system va anche a segnalarlo al controller e in questa maniera il controller mantiene una sua tabella Bitmap di pagine libere e può eseguire in maniera più efficiente le operazioni di cancellazione e la Garbage Collection.

Dischi RAID

Redundant Array of Independent Disks

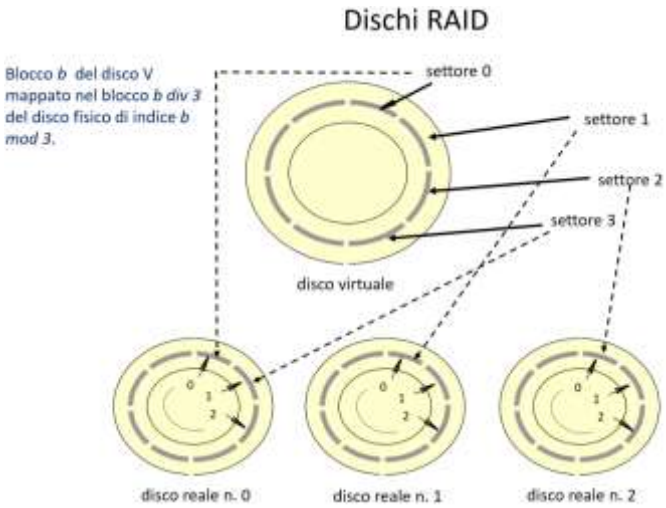
Architettura RAID:

- Realizza un *disco virtuale* di capacità superiore a quella dei singoli dischi
  l'interfaccia è quella di un unico disco
- sfrutta il parallelismo per ottenere un accesso più veloce
  i blocchi consecutivi di uno stesso file sono distribuiti sui dischi dell'array in modo da permettere operazioni contemporanee
- sfrutta la ridondanza per accrescere l'affidabilità
  la ridondanza permette di correggere gli errori di certe classi

Diversi livelli di architettura RAID, con diversi livelli di ridondanza

L'ultimo argomento sui dispositivi sono i RAID, i RAID sono nati per rendere più efficienti, per virtualizzare i dischi rigidi però nulla toglie le stesse tecnologie/soluzioni dei RAID possano essere adottate anche per gestire i dischi flash. In ogni caso lo scopo primario quando son nati era proprio quello di virtualizzare i dischi fisici, quindi renderli più capienti e più veloci. Come spesso succede questo tipo di problematica viene risolta aumentando il grado di parallelismo. Se volete far diventare un computer più veloce gli potete aggiungere una CPU, esattamente la stessa cosa viene fatta in dischi RAID: per rendere dei dischi fisici più veloci, li si parallelizza, si utilizzano più dischi contemporaneamente. In effetti RAID è l'acronimo di Redundant Array of Independent Disks, appunto vuol dire un array, una serie di dischi che operano in maniera indipendente l'uno dall'altro ma coordinata a un livello superiore. Un disco RAID è organizzato con un controller che ripartisce le operazioni di lettura e scrittura sui vari dischi fisici in funzione di un proprio algoritmo. In questa maniera il controller realizza un disco virtuale, ciò significa che il controller offre al file system la visione di un unico disco visto come un array di blocchi, concretamente però questi blocchi non sono allocati tutti sullo stesso disco fisico ma sono allocati su dischi fisici differenti e l'allocazione dei blocchi del disco virtuale sui dischi fisici è nota al controller. Questa allocazione nel disco virtuale sui blocchi fisici dei dischi viene fatta in maniera tale da ripartire il carico su tutti i dischi, in questa maniera le prestazioni complessive del disco aumentano. Idealmente se avete 10 dischi le prestazioni dovrebbero diventare 10 volte tanto, in certi casi lo diventano pure. L'altro elemento è quello di sfruttare la ridondanza per far crescere l'affidabilità. Storicamente i dischi RAID sono stati sempre legati a un problema di affidabilità e di ridondanza. Il problema è che se avete un disco solo, normalmente questo disco ha un tempo medio di fallimento, un tempo che impiega prima di guastarsi e questo tempo generalmente è abbastanza lungo. Questo tempo che viene detto MinTimeToFailure (tempo medio al primo fallimento) normalmente viene misurato da chi produce l'hardware ed è nell'ordine degli anni per un disco normale. Se prendete tanti dischi e li utilizzate contemporaneamente, la probabilità che almeno uno di questi si rompa in un tempo più breve aumenta, perché il tempo di fallimento è un tempo di fallimento medio, ci sarà qualche disco che durerà di più e qualche disco che durerà di meno, se voi prendete più dischi è facile beccarne uno che si rompe prima, almeno uno. Basta che se ne rompa almeno uno di disco per crearvi un problema, perché le informazioni le distribuite equamente su tutti quanti i dischi, quindi se uno solo si rompe siete spacciati. Per convincervi di questo potete fare due cose: o vi mettete a studiare la probabilità di fallimento oppure seguite questo esempio piuttosto macabro: pensate di giocare alla roulette russa, la roulette russa si fa con un revolver a 6 colpi, uno carica un solo colpo alla pistola, fa girare la rotella e poi si spara. La probabilità che uno si suicidi è 1/6. Immaginate che anziché utilizzare un solo revolver a 6 colpi, di

usare 10 revolver a 6 colpi in parallelo. Quindi prendete tutti questi 10 revolver a 6 colpi, girate la rotella su tutti quanti e poi contemporaneamente vi sparate con tutti questi 10. Qual è la probabilità che vi suicidiate? Io non ci giocherei, spero neanche voi (e invece lo spera). La stessa cosa avviene con il RAID, più ne avete e più è facile che uno si rompa prima. Per questo motivo noi possiamo aumentare il numero di dischi per aumentare le prestazioni ma il problema è che il disco virtuale ha una durata inferiore ed è per questo motivo che dobbiamo aumentare la ridondanza, perché aumentando la ridondanza possiamo aumentare l'affidabilità per controbilanciare questo fenomeno. In realtà di RAID ci son diversi livelli (diversi livelli di architettura), ognuno con un diverso livello di ridondanza da tarare in funzione delle specifiche e delle necessità del problema.



Questo vi fa vedere come è organizzata la cosa, quindi abbiamo un certo numero di dischi fisici, ogni disco fisico ha i suoi settori, le sue tracce e abbiamo un disco virtuale. Questo disco virtuale in realtà non ha settori e tracce fisiche, ha settori e tracce virtuali che si mappano sui settori e sulle tracce dei dischi fisici, quindi per esempio il settore 0 della traccia 0 si trova nel settore 0 della traccia 0 del disco 0. Il settore 1 della traccia 0 del disco virtuale si trova nel settore 0 del disco 1 della traccia 0 del disco 1, il settore 2 della traccia 0 del disco virtuale si trova nel settore 0 della traccia 0 del disco 2 e via discorrendo. Come avviene materialmente il mapping è banale ma comunque neanche particolarmente interessante, questo lo fa l'hardware direttamente, lo fa direttamente il controller, al livello superiore il controller presenta il disco virtuale. Concretamente le prestazioni di questo disco virtuale dipendono dalle prestazioni di questi 3 dischi fisici, essendo 3 possono fare operazioni in parallelo e quindi possono rispondere a delle richieste fatte sul disco virtuale in tempi brevi. Le architetture RAID sono organizzate a livelli, in particolare:

Dischi RAID

Livello 0: Dischi asincroni, nessuna ridondanza

- Si possono effettuare contemporaneamente operazioni indipendenti
- Anche detto JBOD (just a bunch of disks)

Livello 1: Dischi asincroni, disco con copie ridondanti *(mirror)*

- Si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Livello 2: Dischi sincroni, i dischi ridondanti contengono codici per la correzione degli errori

- si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Livello 3: Dischi sincroni, un solo disco ridondante

- contiene la parità del contenuto degli altri dischi
- si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Utilizzatissimo è il livello 0, nessuna ridondanza, se comprate un raid per casa è facile che questo raid sia configurato a livello 0, questo è ragionevole se il RAID è formato da pochi dischi, il RAID 0 tipicamente va bene se avete uno, due, tre dischi, non molti di più. I tre dischi sono indipendenti, quindi posso operare letture e scritture indipendentemente su settori e tracce differenti, ognuno ha le proprie testine, sono coordinati a livello più alto dal controller, questo tipo di livello viene detto anche JBOD (just a bunch of disks), vuol dire semplicemente un insieme di dischi, quindi il controller qui davvero fa poco lavoro. Non c'è nessuna ridondanza, quindi se un disco si guasta perdete l'informazione di quello però gli altri continuano a funzionare. Sebbene siano in RAID, i dischi comunque continuano ad avere dei meccanismi ridondanti interni, ogni settore continua ad avere il suo codice correttore d'errore e il controller mappa i settori danneggiati del singolo disco in maniera tale da non utilizzarli, per cui in realtà il danneggiamento che può capitare (normale è il settore che si guasta), in realtà i settori nei dischi non si guastano istantaneamente, ma si guastano progressivamente, per cui il controller si può rendere conto che un settore si sta danneggiando, a quel punto lo ricopio da un'altra parte, lo esclude e il disco continua a funzionare. Quello che ci preoccupa quando parliamo di RAID, non sono tanto questo tipo di guasti dovuti all'usura dei settori, quanto ai guasti che portano il disco a collassare istantaneamente in maniera brutale, per esempio si rompe il controller, oppure la testina, il controller per qualche motivo perde il controllo della testina che danneggia completamente la superficie oppure problemi meccanici per cui il disco si grippa e il disco non gira più, cose di questo genere. Il livello 1 invece prevede sempre dischi asincroni, quindi i dischi sono indipendenti, ma ogni disco ha una copia ridondante. Ora se andate a vedere, su internet, i dischi di livello 1 esistenti, c'è un po' di ambiguità su che cosa sia effettivamente il livello 1. Su alcuni testi viene scritto una cosa, su altri ne viene scritta un'altra, su internet altro ancora, c'è ambiguità sul livello 1. In alcuni casi il livello 1 s'intende un solo disco principale con la sua copia, in alcuni casi invece il livello 1 viene inteso come un gruppo di dischi con la loro copia. Stabiliamo da questo momento in poi che il livello 1 lo intendiamo come un disco principale con una sua copia, fine. Qual è il vantaggio del fatto di avere un disco livello 1? Il vantaggio è che un disco collassa, si rompe, posso continuare a utilizzare l'altro, per cui le informazioni sono al sicuro. Però quando un disco si rompe, non posso andare avanti così in eterno perché prima o poi si romperà anche l'altro e perderò tutte le informazioni. Il RAID si usa in questa maniera: partite con tutti e due i dischi pienamente efficienti, li utilizzate duplicando tutte le informazioni, non appena un disco si rompe il controller se ne accorge, lo segnala al gestore del sistema che deve correre a comprare un altro disco, sostituire il disco rotto e a questo punto far ripartire il RAID. Siccome però quest'operazione di manutenzione porta via del tempo e magari invece questo disco RAID è importante che continui a

funzionare perché sta svolgendo un servizio, in realtà il servizio continua a svolgerlo tramite il disco ridondante, potrebbe rompersi il disco ridondante e restare vivo il Master, è equivalente. Per cui in realtà il disco RAID può andare avanti anche in caso di guasto a prestazioni degradate. A prestazioni degradate vuol dire che sarà un po' più lento e non avrà a questo punto più il livello di sicurezza/affidabilità. Nel momento nel quale il nuovo disco viene inserito al posto di quello rotto, si può ripristinare una funzionalità normale, per ripristinarla bisogna ricopiare tutto il disco integro nel nuovo disco inserito, fatta questa operazione di copia il sistema può ripartire. Il livello 1 di nuovo è un tipo di livello che potete configurare anche nei RAID di fascia bassissima, quelli che potete trovare al MediaWorld, in qualsiasi negozio elettronico. I RAID al livello 2 e 3 per quanto ne so non sono più utilizzati, in passato sono stati usati in alcuni sistemi, in particolare il livello 2 era utilizzato nella Connection Machine che era una vecchissima macchina parallela degli anni '80. Nel 2 e 3 i dischi sono sincroni quindi in realtà son tanti dischi nei quali le testine si muovono esattamente in sincrono, tutte le testine vanno esattamente a leggere lo stesso settore sulla stessa traccia anche se i dischi son separati, quindi è come avere un controller unico per tutti quanti questi dischi. Nel livello 2 abbiamo tanti dischi ridondanti, possono essere una decina e di questi, 3-4 possono essere interamente ridondanti, ridondanti però utilizzando dei codici, non utilizzando la copia. Nel livello 3 invece i dischi son sincroni, sono tanti ma c'è un solo disco ridondante. Il disco ridondante è quello che contiene informazioni aggiuntive che vi servono per recuperare da eventuali guasti di un disco, quindi se un disco si guasta, tramite il disco ridondante potete ricostruire l'informazione del disco rotto. Questa ridondanza che si utilizza nel livello 2-3 è molto diversa dalla ridondanza che si usa nel livello 1, nel livello 1 la ridondanza è la copia, ogni informazione è duplicata. Nel livello 2-3 invece si utilizza un codice, un codice correttore d'errore che è più efficiente rispetto alla duplicazione. Vi faccio vedere un codice, quello usato nel livello 3-4-5 che è un codice basato su parità. Il codice utilizzato per i RAID di livello 2 è un codice più complesso ma essendo livelli che non si utilizzano non mi sembra il caso di insistere.

Dischi RAID

Livello 4: Dischi asincroni; un disco ridondante

- contiene la parità del contenuto degli altri dischi
- si possono effettuare contemporaneamente operazioni indipendenti e correggere errori
- Il disco ridondante è sovraccarico nei piccoli aggiornamenti

Livello 5: Come livello precedente, ma parità distribuita tra tutti i dischi

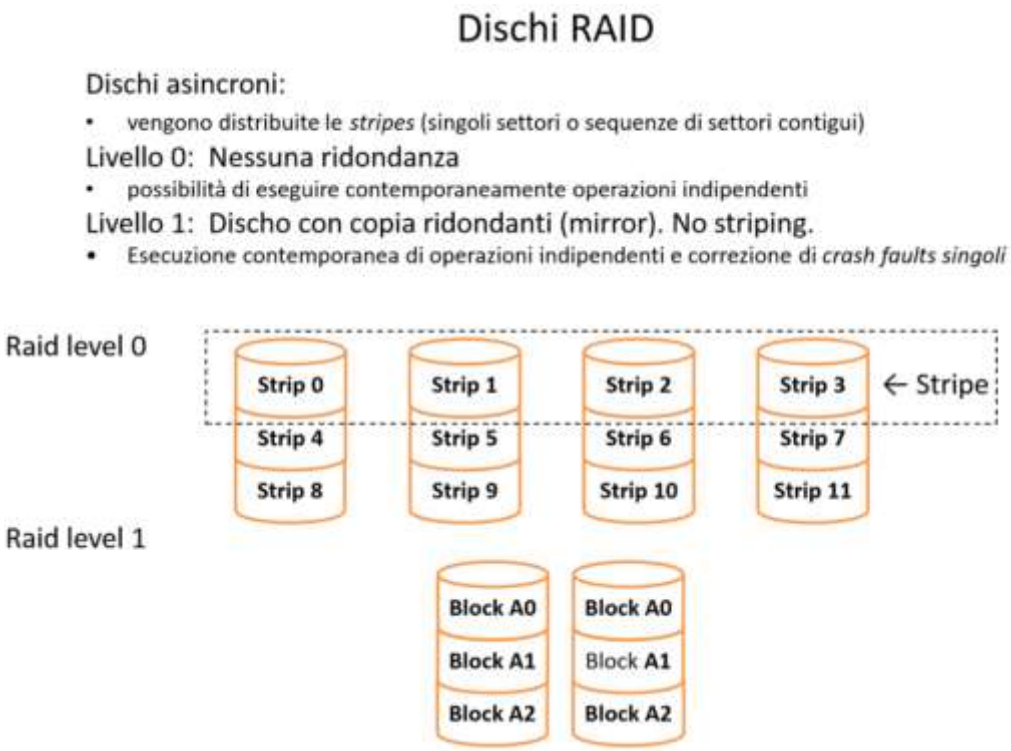
- permette un miglior bilanciamento del carico tra i dischi

Livello 1+0: Dischi asincroni, Mirror di stripes
Livello 0+1: Dischi asincroni, Stripe di mirror

Per la correzione di errori: ipotesi di *errori singoli* e *crash faults*

I livelli più interessanti sono 4, 5, 1+0, 0+1, in realtà il 4 non è utilizzato però è utile per capire poi come funziona il 5, quelli normalmente più usati dalle aziende che hanno informazioni più delicate e sulle quali vogliamo avere più garanzie, normalmente utilizzano il 5 o 1+0 o 0+1, molto spesso si usano 1+0 e 0+1 o addirittura si possono utilizzare combinazioni di questi livelli. In effetti 1+0 e 0+1 sono già una combinazione di livelli 0 e 1. Nel livello 4 abbiamo dischi indipendenti, un disco contiene codici ridondanti, in particolare contiene dati di parità per cui col livello 4 succede che, se abbiamo 5 dischi, 4 dischi contengono le informazioni vere e proprie, un disco, l'ultimo, non contiene informazioni, non potete

andare a leggere e scrivere dati sull'ultimo disco, ma andate a scrivere/leggere soltanto dei codici che permettono di recuperare il guasto di uno degli altri 4 dischi. Qual è l'utilizzo normale? Quando andate a fare operazioni di lettura, l'operazione di lettura la dovete mandare sul disco nel quale ci sono i dati da leggere, nei primi 4 dischi non ci sono informazioni ridondanti quindi un dato è stato scritto in uno di questi 4 dischi, quando ho fatto un'operazione di lettura, quest'operazione deve andare in uno di questi 4 dischi necessariamente però gli altri 3 dischi restano liberi, per cui voi potete effettuare fino a 4 operazioni di lettura contemporaneamente. Quando invece effettuate un'operazione di scrittura, l'operazione di scrittura sarà diretta a uno specifico settore di uno di questi 4 dischi ma non è sufficiente perché se scrivete e basta poi non avete affidabilità, dovete anche aggiornare il codice correttore quindi l'operazione di scrittura coinvolge sempre un disco e il disco ridondante. Se uno di questi 4 dischi si rompe, sfruttando il contenuto degli altri 3 dischi sani e il disco ridondante, è possibile avviare una procedura di ripristino che vi recupera interamente il contenuto del disco rotto. Il RAID di livello 5 è come il livello 4, l'unica differenza è il modo col quale questa informazione di parità è memorizzata. Nel disco 4 sta tutto in un unico disco, nel disco 5 è ripartita in maniera differente. 1+0 o 0+1 sono due RAID in cascata: 1+0 sono più RAID di livello 1 organizzati in un RAID di livello 0, invece 0+1 sono più RAID di livello 0 organizzati in un RAID di livello 1.



Nel Raid di livello 0, immaginatevi questa situazione: ognuno di questi è un disco, questi sono i blocchi del disco (blocco 0, blocco 1, blocco 2, ecc), nella notazione a RAID si chiamano strip, se prendiamo tutti i blocchi, tutte le strip che occupano la stessa posizione in tutti i dischi, questa forma una stripe e quindi le strip all'interno del RAID vengono numerate trasversalmente, seguendo l'ordine delle stripe. Se volete andare a scrivere informazioni su un file nei blocchi 0, 1, 2, 3, in realtà materialmente non li scrivete tutti sullo stesso disco ma le ripartite in tutti e 4 i dischi. Quando andate a fare una lettura di un file, potete leggere più blocchi contemporaneamente dai vari dischi, questo dipende poi anche dal file system e di come gestisce l'informazione. Nel Raid di livello 1 invece, avete un disco che ha i suoi blocchi e questo è il disco principale, ogni scrittura su disco principale viene duplicata nel disco ridondante, per cui se volete leggere il blocco 0 e il blocco 2, in realtà è indifferente da quale disco lo leggete per cui potete operare le due letture contemporaneamente su due dischi, quindi nel RAID di livello 1 le prestazioni in lettura raddoppiano, le prestazioni in scrittura sono equivalenti a quelle del disco singolo.

Dischi RAID

Dischi sincroni: vengono distribuiti i bit

Livello 2: Ridondanza con codice correttore di errori, stripe bit-level

Esempio bit 0, 1, 2, 3 e 4 di informazione, bit 5, 6 e 7 ridondanti

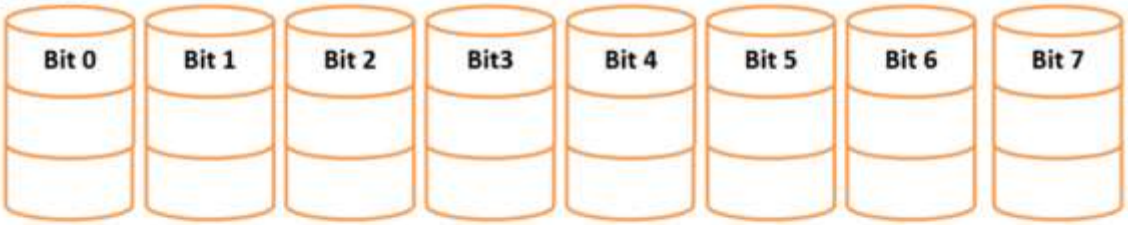
- Lettura o scrittura contemporanea di tutti i bit della strip; *rilevazione errori doppi e correzione di errori singoli (Hamming ECC Code)*

Livello 3: Ridondanza con parità, stripe bit-level (può anche essere byte-level)

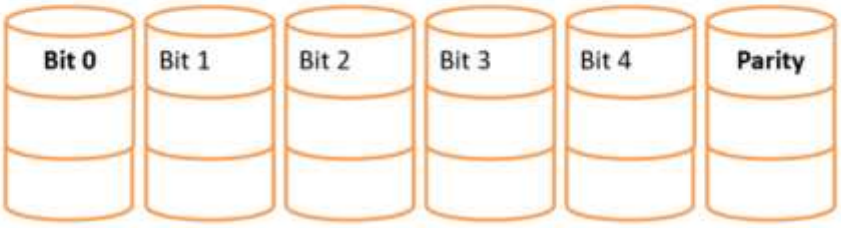
Esempio: bit 5 parità dei bit 0, 1, 2, 3, 4

- Lettura o scrittura contemporanea di tutti i byte della strip e *correzione di crash faults singoli*

Raid level 2



Raid level 3



RAID di livello 2 e 3 in realtà sono dischi singoli, per cui le testine sui vari dischi son sempre posizionate tutte nella stessa posizione, non sono indipendenti i dischi, non potete fare letture in posizioni arbitrarie e in questo caso i dati vengono ripartiti a bit, come se tutti questi dischi fossero un'unità per cui, ora in questo caso sono 8 dischi, se il primo byte fosse sparpagliato sul primo bit/il primo blocco di tutti quanti i dischi. La stessa cosa nel RAID di livello 3, che cosa cambia nel RAID di livello 3 da quello di livello 2? Che nel 2, un certo numero di dischi, in questo caso potrebbero essere 3 o 4, dipende dal codice, non conservano dati ma conservano informazioni ridondanti per cui i dati vengono ripartiti tra i bit dei dischi non ridondanti. Nel caso del RAID di livello 3 invece, c'è un unico disco ridondante che è l'ultimo e che contiene un'informazione di parità. C'è la ridondanza nel RAID, concentriamoci sulla singola striscia di bit, dove i bit da 0 a 4 hanno le informazioni che devono avere, queste le avrà scritte l'utente, il controller va a scrivere nell'ultimo disco, nella posizione omologa, un valore di parità, la parità di questi bit, quindi se il numero di bit a 1, tra 0 e 4, è dispari, il valore di parità è 1, altrimenti il valore di parità è 0, in maniera tale che il numero di 1 in questa striscia sia sempre pari. Supponiamo a questo punto che questo disco si rompa. Il controller si rende conto che questo disco è rotto perché non riesce più a leggere nulla da questo disco, come fa a ricostruire l'informazione? Siccome sa che il numero di bit a 1 è pari, lui legge tutti i bit da tutta la striscia, ovviamente tranne quello rotto, se legge un numero pari di 1, vuol dire che qui c'era scritto 0, se legge un numero dispari di 1, vuol dire che qui ci doveva esser scritto 1. In questa maniera ricostruisce il contenuto del bit. Funziona come codice correttore perché sappiamo dov'è l'errore, soltanto sapendo qual è il bit danneggiato o non accessibile, soltanto in queste condizioni, la parità funziona come codice correttore. In altre situazioni invece, per esempio nelle reti, nelle comunicazioni, quando noi non sappiamo qual è il bit danneggiato, la parità non funziona come codice correttore, al massimo se siamo fortunati ci dice che c'è un errore ma non sappiamo dove sta quindi non possiamo correggere. Nel caso dei RAID invece, sapendo qual è il bit danneggiato, con un solo bit di parità possiamo correggerlo. Lo stesso codice a parità è utilizzato nei RAID di livello 4 e nei RAID di livello 5:

Dischi RAID

Dischi asincroni, vengono distribuite le *stripe*

Livello 4: Ridondanza con *strip* di parità

Esempio: disco 4 contiene le strip di parità delle strips omologhe dei dischi 0, 1, 2, 3

- esecuzione contemporanea di operazioni indipendenti e *correzione di crash faults singoli*

Livello 5: Come livello 4, ma strip di parità distribuite nei vari dischi

- migliore bilanciamento del carico tra i dischi

Raid level 4

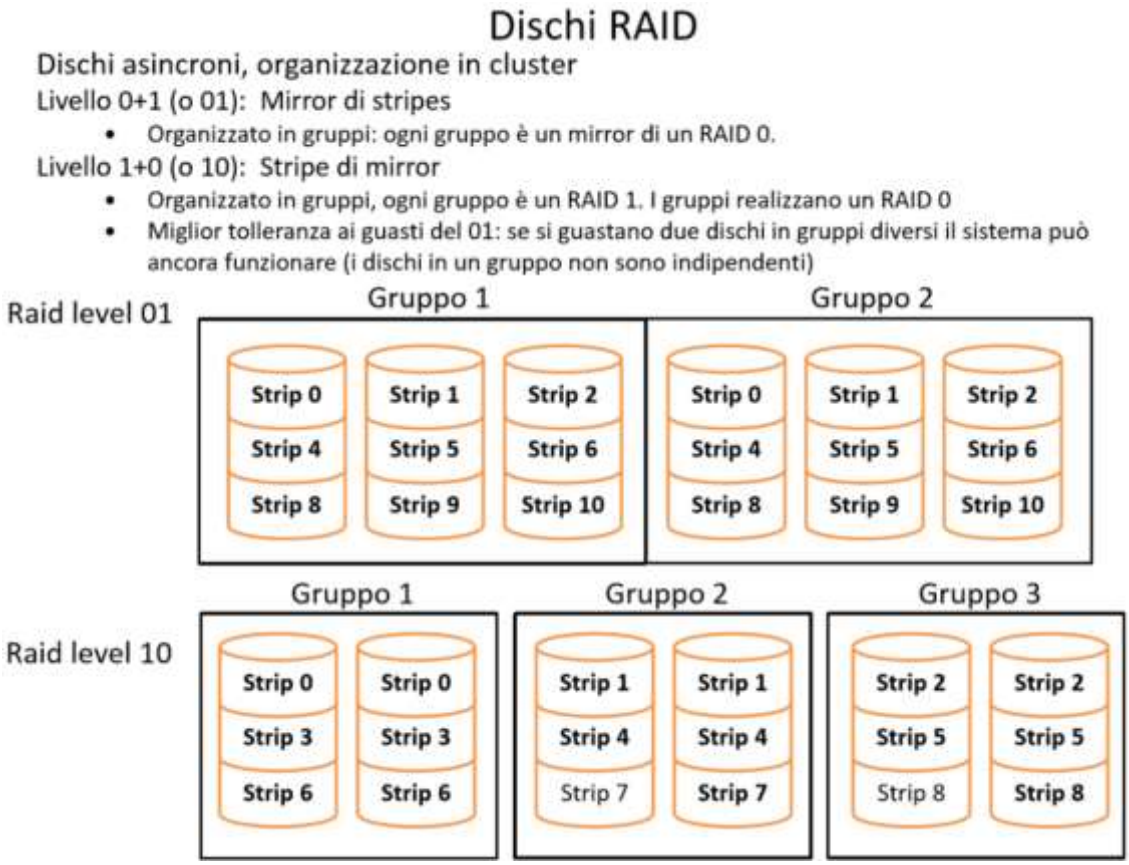
Strip 0	Strip 1	Strip 2	Strip 3	P 0-3
Strip 4	Strip 5	Strip 6	Strip 7	P 4-7
Strip 8	Strip 9	Strip 10	Strip 11	P 8-11

Raid level 5

Strip 0	Strip 1	Strip 2	Strip 3	P 0-3
Strip 4	Strip 5	Strip 6	P 4-7	Strip 7
Strip 8	Strip 9	P 8-11	Strip 10	Strip 11
Strip 12	P 12-15	Strip 13	Strip 14	Strip 5
P 16-19	Strip 16	Strip 17	Strip 18	Strip 19

I RAID di livello 4 e 5 sono organizzati con dischi indipendenti, quindi nel RAID di livello 4 in questo caso viene 5 dischi, 4 dischi sono indipendenti e su questi mettiamo le informazioni andando a scriverle direttamente nelle strip, immaginate la strip come un settore/blocco, ogni stripe del disco ridonante ha il suo codice di parità. Di nuovo, questo codice ridondante son di nuovo delle parità, sono i tipi di parità di tutta la stripe, quindi ogni bit di ognuna di queste 4 strip ha il suo bit di parità in questa strip. La correzione dell'errore funziona esattamente come nel caso del RAID di livello 3, se c'è un disco danneggiato possiamo ricostruire interamente la strip 1 sfruttando le informazioni della strip 0, 2, 3 e della parità, esattamente come avveniva prima, facendolo bit per bit su tutta la strip. Qual è il vantaggio del RAID al livello 4? Sul RAID al livello 4 posso fare fino a 4 letture contemporaneamente, quando faccio una scrittura, la scrittura deve essere fatta sul disco sul quale scrivo e sul disco di parità contemporaneamente e quando faccio questa scrittura però non posso modificare le altre strip, per esempio se vado a modificare strip 1, devo andare a scrivere su strip 1 e sulla parità da 0 a 3, ma in questa fase non posso andare a modificare strip 2, strip 3, strip 0, perché altrimenti avremmo problemi di concorrenza, potrei alterare il dato di parità e lasciare il disco in stato inconsistente. Se vado a scrivere strip 1, posso andare a scrivere strip 6 contemporaneamente ma non posso andare contemporaneamente a scrivere strip 2, per farlo devo in realtà coordinare le due scritture e andare a calcolare la parità in maniera tale da gestire entrambe le scritture (se sono in grado di farlo), altrimenti devo farle separatamente. Sebbene siano 5 dischi, col RAID 4 al massimo posso quadruplicare le prestazioni, non di più, questo perché l'ultimo disco è usato soltanto per la parità, quindi non lo posso usare in lettura. Nel RAID di livello 5 fanno una cosa semplice: riprendono il disco di parità ma il suo contenuto lo sparpagliano in questo modo sulle diagonali tra tutti i dischi in maniera uniforme. In questa maniera tutti i dischi sono utilizzabili per un'operazione di lettura. Nel RAID 5, posso parallelizzare un'operazione di lettura fino a 5 volte in questo caso, ho 5 dischi. Con un piccolo accorgimento ho aumentato le prestazioni rispetto al RAID 4 ma non ho aggiunto posti o complicazioni particolari, ho soltanto gestito diversamente la parità. Quindi il RAID 4 non viene utilizzato, ciò che viene utilizzato è il RAID 5, il RAID 5 non è il plus-ultra della sicurezza dal punto di vista dei livelli RAID classici, successivamente è stato introdotto il RAID 6 che in realtà utilizza un codice correttore un po' più complesso e può correggere errori doppi. Idealmente il RAID 5 dovrebbe essere il massimo, sia in termini di affidabilità, sia in termini di prestazioni, perché non penalizza le prestazioni, permette di ottenere tutti i

vantaggi dati dalla parallelizzazione dei dischi sfruttandoli tutti in maniera uniforme e riesce a corregge il danneggiamento del disco singolo. Però chi ci lavora sa che i RAID non si rompono soltanto perché si rompe un disco, il problema è che i RAID hanno anche un controllo che è un Single Point Of Failure, un componente elettronico, più affidabile del singolo disco che però anche lui si può danneggiare. Se si danneggia il controller tutto il disco risulta inaccessibile. Se volete avere delle cose più affidabili, dovete duplicare tutto l'hardware compreso il controller, questa cosa la potete fare agevolmente strutturando i RAID su più livelli e questa è l'idea del 0+1 o 1+0.



Nel livello 01 l'idea è quella di dividere i dischi in gruppi, ogni gruppo è un RAID di livello 0, quindi il gruppo 1 è un RAID di livello 0, il gruppo 2 è un RAID di livello 0, quindi hanno il loro controller che opera a livello 0 (due controller), però sopra c'è un altro controller che organizza questi due gruppi come se fossero un RAID di livello 1, quindi in realtà tutte le informazioni memorizzate qui sono duplicate qui, se si rompe il controller del gruppo 1, il sistema può ancora funzionare con il solo gruppo 2, se si rompe un disco del gruppo 1, il sistema continua a funzionare perché nel gruppo 2 ho il disco ridondante. In realtà è semplicissimo, niente di complicato. Invece il livello 10 funziona al contrario: dividiamo i nostri dischi in gruppi, ogni gruppo è un RAID di livello 1, quindi il gruppo 1 è come se fosse un unico disco ma in realtà è un disco ridondante, stesso per il gruppo 2, stesso per il gruppo 3. Tutti quanti vengono messi in un RAID 0. In questo caso se si rompe il controller del gruppo 1 queste informazioni non sono più accessibili perché il gruppo 1, gruppo 2, gruppo 3 insieme formano un RAID 0 quindi non c'è ridondanza tra 1 e 2, l'unica ridondanza è quella interna. Vi faccio vedere un esempio di RAID di livello 4.

Dischi RAID di livello 4: esempio (1)

Un disco RAID di livello 4 è composto da 5 dischi fisici, numerati da 0 a 4. I blocchi del disco virtuale V sono mappati nei dischi 0, 1, 2, 3: precisamente il blocco *b* del disco V è mappato nel blocco *b div 4* del disco fisico di indice *b mod 4*. Il disco 4 è ridondante e il suo blocco di indice *i* contiene la parità dei blocchi di indice *i* dei dischi 0, 1, 2, 3.

Il gestore del disco virtuale accetta comandi (di lettura o scrittura) che interessano più blocchi consecutivi : ad esempio *read(buffer, PrimoBlocco, NumeroBlocchi)* legge un numero di blocchi pari a *NumeroBlocchi* a partire da quello di indice logico *PrimoBlocco* e li scrive nel buffer di indirizzo iniziale *buffer.*

Ad esempio, l'operazione *read(buffer 12, 3)* legge i blocchi *12, 13, 14* del disco virtuale, mappati nel blocco *3* dei dischi fisici *0, 1, 2*

✓ trattandosi di un'operazione che interessa dischi fisici indipendenti, può essere eseguita in un solo tempo di accesso.

- SALTO DELLA PAUSA –

Per quanto riguarda il RAID di livello 4 vediamo un esempio, negli esercizi trovate molti esercizi sul RAID di livello 4, non perché sia importante ma perché scrivere un esercizio per il RAID di livello 4 è molto più semplice, per me che lo scrivo e per voi che lo dovete leggere in 5 minuti, capirlo e risolverlo.

Negli esercizi spesso trovate questo, ma è rappresentativa di tutta quanta la categoria dei RAID. Dal punto di vista lavorativo, se avete mai a che fare con i RAID, vi interesseranno probabilmente quelli 5, quelli 01 o quelli 10.

Prendiamo questo disco RAID 4 composto da 5 dischi che sono mappati secondo un regola scritta qui: il blocco b del disco virtuale è mappato sul blocco b diviso 4 del disco fisico di indice b modulo 4. Quindi per esempio il blocco 0 del disco virtuale mappato sul blocco 0 del disco 0, il blocco 5 del disco virtuale è mappato sul blocco 1 del disco 0, il blocco 6 è mappato sul blocco 1 del disco 1 e via discorrendo. Il gestore del disco virtuale accetta comandi (lettura o scrittura) usuali, simili a quelli che abbiamo visto, per cui si può leggere andando a depositare informazioni sul buffer, a partire da un certo blocco.

Per esempio l'operazione read(buffer 12, 3) legge i blocchi 12, 13, 14 che sono mappati nel blocco 3 dei dischi fisici 0, 1, 2. Basta che fate le operazioni di cui sopra e ottenete queste informazioni.

Non che nei dischi RAID si utilizzino istruzioni di questo genere, i dischi RAID li utilizzate esattamente come utilizzate i dischi normali, con le read di UNIX che avete già visto, questa read è inventata ad Hoc per fare l'esercizio ma non è poi molto diversa. Dato che i dischi RAID nel livello 4 sono indipendenti, quest'operazione può essere anche eseguita in un solo tempo di accesso perché può essere ripartita su più dischi.

Dischi RAID di livello 4: esempio (2)

Supponiamo che i blocchi di indice 3 dei dischi fisici *0, 1, 2* e *3* abbiano i contenuti mostrati in tabella: di conseguenza il blocco di indice 3 del disco fisico 4 contiene la parità del contenuto dei blocchi omologhi dei dischi fisici *0, 1, 2* e *3*.

Disco 0	0	1	0	0	1	1	0	1
Disco 1	1	0	1	1	0	0	0	1
Disco 2	0	1	1	0	1	0	0	1
Disco 3	0	1	1	1	1	0	0	1
Disco 4	1	1	1	0	1	1	0	0

Se la lettura dal disco fisico *1* fallisce a causa di un *crash fault*, l'evento viene rilevato dal controllore, che restituisce un blocco vuoto. Il contenuto del blocco 3 del disco fisico 1 può essere ricostruito come parità dei contenuti dei blocchi omologhi dei dischi *0, 2, 3* e *4*.

CONTENUTO RESTITUITO

Disco 1	-	-	-	-	-	-	-	-

CONTENUTO RICOSTRUITO

Disco 1	1	0	1	1	0	0	0	1

Se si esegue l'operazione *write(buffer 13, 1)*, che scrive il contenuto del *buffer* nel blocco di indice *3* del disco fisico 1 e il buffer contiene *1 1 0 1 0 1 1 1*, è necessario modificare come mostrato in tabella anche la parità contenuta nel blocco omologo del disco Fisico 4. La parità può essere ricalcolata in base alla differenza tra il vecchio e il nuovo contenuto del blocco 3 del disco *1*, senza la necessità di leggere i blocchi omologhi dei dischi *0, 2* e *3*.

PRIMA DELL'OPERAZIONE

Disco 0	0	1	0	0	1	1	0	1
Disco 1	1	0	1	1	0	0	0	1
Disco 2	0	1	1	0	1	0	0	1
Disco 3	0	1	1	1	1	0	0	1
Disco 4	1	1	1	0	1	1	0	0

DOPO L'OPERAZIONE

Disco 0	0	1	0	0	1	1	0	1
Disco 1	1	1	0	1	0	1	1	1
Disco 2	0	1	1	0	1	0	0	1
Disco 3	0	1	1	1	1	0	0	1
Disco 4	1	0	0	0	1	0	1	0

Supponiamo che 5 dischi nel blocco di indice 3 contengano questa informazione che è rappresentata lì. Ovviamente il blocco è molto più grande, qualche KB, però per fare l'esercizio ve l'ho ridotto a pochi bit, concettualmente non cambia molto. Se dovete fare un'operazione di lettura andate a leggere il contenuto di quei 3 blocchi in nero dai dischi fisici 0, 1, 2, 3, questo non è un problema. Supponiamo però che la lettura del disco fisico 1 fallisca perché il disco fisico 1 collassa, subisce un crash fault, un guasto non recuperabile, non è più utilizzabile, non ci si riesce a leggere. Questo tipo di guasto viene rilevato dal controller che non riesce a dialogare col disco fisico o che non riesce a tirare fuori dei dati per i quali il codice correttore gli dica che l'informazione è corretta. Quindi il controller sa benissimo che ciò che ha letto dal disco fisico 1, non è utilizzabile, è danneggiato, ammesso che sia riuscito a leggere qualcosa. A questo punto allora, per completare quest'operazione, il controller può andare a leggere il contenuto del blocco 3 del disco 4 che è quello ridondante, siccome ha letto il contenuto dello stesso blocco dei dischi 0, 2, 3, sfruttando l'informazione ridondante può ricostruire. Come ricostruisce? Prendete per esempio il bit in verticale, prendete ad esempio il contenuto di questa colonna, da questa colonna il controller legge 0 dal disco 0, niente dal disco 1, 0 dal disco 2, 0 dal disco 3 e legge 1 dal disco 4, quindi ha letto 0, nulla, 0, 0, 1, siccome sa che il numero di bit a 1 deve essere pari perché quello è un dato di parità, il contenuto del disco 1 deve necessariamente essere 1 e quindi in questo modo ricostruisce il primo bit del disco 1. Fa lo stesso lavoro per tutti gli altri bit letti in questa mandata e in questo modo ricostruisce interamente il disco 1. In questo modo il disco può andare avanti, il disco virtuale può andare avanti operando fintantoché poi il sistemista, arriva un disco nuovo, lo sostituisce, fa partire l'operazione di recovery e quindi il disco nuovo viene ripristinato con tutte le informazioni corrette e a quel punto il RAID riparte a prestazioni normali. Supponiamo invece di fare una scrittura, andiamo a scrivere un certo valore, per esempio 11010111 nel blocco 13 del disco virtuale. Cosa succede, il blocco 13 del disco virtuale è mappato nel blocco 3 del disco fisico 1, quindi supponiamo che il disco prima dell'operazione contenga quest'informazione, noi dobbiamo andare a sovrascrivere questa stringa di bit con quest'altra, se noi però facciamo la scrittura soltanto nel disco 1, la parità non torna più. Per esempio nella seconda colonna dobbiamo scrivere un 1, se vi scriviamo un 1 la parità non torna più, ho 5 bit a 1, è sbagliato. Quindi scrivere soltanto nel disco 1 non basta, devo anche scrivere nel disco 4 il codice di parità, il problema è che il codice di parità come lo calcolo? Devo mettere la parità nuova. La parità nuova è in funzione di tutto il contenuto di tutti e 4 i dischi, quindi per

fare la scrittura in teoria dovrei leggere questo blocco di tutti dischi, calcolare la nuova parità e poi andare a scrivere disco 1 e disco 4, quindi un'operazione di scrittura mi comporterebbe 3 letture e 2 scritture, non ho bisogno di leggere il contenuto di disco 1 perché tanto il contenuto cambia, la parità la faccio direttamente sul valore nuovo, quindi in teoria dovrei fare 3 letture e 2 scritture, in realtà posso essere più efficiente perché per calcolare la nuova parità la posso fare a partire dalla vecchia, quindi nella realtà che faccio? Leggo il vecchio valore del disco 1, leggo il vecchio valore del disco 4, a questo punto prendo il nuovo valore che voglio scrivere, guardo che il primo bit è uguale al primo bit, quindi la parità qui non cambia, il secondo bit è differente, quindi qui la parità la devo cambiare, anziché 1 ci devo scrivere 0. Il terzo bit è differente, è diverso, quindi anche qui devo cambiare la parità, quindi in realtà per aggiornare la parità a noi basta leggere il vecchio contenuto del disco 1, il vecchio contenuto del disco 4, ricalcolo la nuova parità e a questo punto scrivo il contenuto del disco 1 e del disco 4, anziché 3 letture e 2 scritture, posso cavarmela con 2 letture e 2 scritture. Una scrittura mi comporta in realtà 2 letture e 2 scritture. La cosa non è drammatica perché in realtà sto quadruplicando le prestazioni in lettura e sto dividendo per 4 le prestazioni in scrittura, è un bel guadagno perché la maggior parte delle operazioni sono in lettura e poi le scritture non le devo eseguire subito, le scritture spesso vengono fatte in cache e poi dopo quando il disco è scarico vengono effettuate, quindi in realtà le operazioni in scrittura, molto spesso, non le percepisco perché vengono fatte nei tempi morti. A questo punto abbiamo visto i dispositivi, vediamo dettagli sul File System.

FILE SYSTEM

File System Workload

- File sizes
  - Are most files small or large?
  - Which accounts for more total storage: small or large files?

Quando si progetta un file system, bisogna pensare, se vogliamo ottimizzare le prestazioni, non basta pensare a quali sono le caratteristiche del disco che stiamo utilizzando, ma bisogna pensare anche a come verrà utilizzato, per esempio: se voi prendete il file system del vostro computer, la maggior parte dei file sono piccoli o grandi? Nella maggior parte dei casi la maggior parte dei file memorizzati in un file system sono piccoli, i file grandi in realtà sono pochi, d'altra parte i file grandi occupano più spazio, quindi se voi andate a vedere nel disco quant'è lo spazio occupato da file grandi e quant'è lo spazio occupato da file piccoli, scoprite che la maggior parte dello spazio è occupato da file grandi ma la stragrande maggioranza dei file sono molto piccoli. C'è molta disparità nei file che si memorizzano, un file che memorizza un video può tranquillamente occupare qualche gigabyte. Per occupare un gigabyte con file piccoli bisogna davvero averne una quantità spropositata. Questo ci dice che dobbiamo stare attenti perché a seconda di come organizziamo le informazioni, l'accesso ai file piccoli può essere privilegiato oppure può essere più efficiente l'accesso ai file grandi, il problema è che abbiamo a che fare con entrambi questi oggetti, i file piccoli sono presenti in quantità maggiore, i file grandi sono presenti in quantità minore ma occupano più spazio. C'è un altro problema: se guardo il numero degli accessi, utilizzo più spesso i file piccoli o utilizzo più spesso i file grandi?

File System Workload

- File access
  - Are most accesses to small or large files?
  - Which accounts for more total I/O bytes: small or large files?

E se vado a vedere la quantità di byte che vengono spostati sul bus dal disco alla memoria, sposto più byte per spostare file piccoli oppure file grandi? La maggior parte degli accessi sono per i file piccoli, perché? Soprattutto con i sistemi moderni che hanno delle interfacce grafiche, ogni qualvolta voi muovete il mouse e fate un click su una cartella, in realtà aprite una nuova cartella e lì ci sono decine e decine di nuove icone fa visualizzare, ogni icona è un file piccolo ma in realtà non è che andate soltanto a leggere le icone, magari per fare quella operazione, concettualmente semplice, in realtà avete letto una valanga di informazioni per sapere come l'utente vuole che quelle informazioni vengano visualizzate, disposte e via discorrendo. Tutte queste sono informazioni piccole, di natura differente, ma memorizzate in file differenti e tutti piccoli. Quindi in realtà operazioni anche semplici nascondono tantissime letture di file piccoli, però se guardate la quantità di byte complessivi trasferiti, nuovamente i file grandi pesano di più, perché su 10.000 letture basta leggere una volta un file grande, ma quel file grande m'ha spostato qualche gigabyte, mentre invece le altre 9999 letture di file piccoli, complessivamente hanno spostate poche decine di KB. Ho due problemi nel file system: come organizzare le informazioni in maniera tale che siano efficienti dal punto di vista dello spazio e come organizzarle in maniera tale che siano efficienti dal punto di vista dell'accesso, dell'utilizzo. Per ottimizzare lo spazio devo pensare ai file grandi, per ottimizzare l'accesso/la velocità devo privilegiare i file piccoli.

File System Workload

- How are files used?
  - Most files are read/written sequentially
  - Some files are read/written randomly
    - Ex: database files, swap files
  - Some files have a pre-defined size at creation
  - Some files start small and grow over time
    - Ex: program stdout, system logs

Come si utilizzano i file? Anche qui ci sono delle differenze, per esempio la maggior parte dei file sono letti e scritti sequenzialmente, è vero che noi abbiamo come metodo d'accesso quello diretto, ma in realtà nella maggior parte dei casi i file si leggono sequenzialmente. Ogni qualvolta mandate in esecuzione un file eseguibile, quel file eseguibile viene letto sequenzialmente; quando dovete visualizzare un'immagine, quell'immagine la leggete sequenzialmente, quando riproducete un video, quel video viene letto sequenzialmente, quando caricate il file system, i file che contengono il nucleo vengono letti sequenzialmente. Gli accessi diretti ai file in realtà sono sporadici e avvengono per alcuni file molto particolari, per esempio i database, un database è facile che debba accedere in ordine casuale. Anche i file di swap, quelli dove vado a mappare lo spazio virtuale dei processi, anche quelli sono utilizzati ad accesso diretto, però nella maggior parte dei casi l'accesso è sequenziale. Quindi se devo ottimizzare, devo partire ottimizzando l'accesso sequenziale più che quello diretto, perché è quello sequenziale che mi pesa di più sulle prestazioni. Quando devo stabilire come devo memorizzare un file, ovviamente potrei dire che se il file è grande allora utilizzo una certa strategia di memorizzazione, se il file è piccolo ne utilizzo un'altra, il problema è che spesso dei file non so che dimensione avranno quando vengono creati. Il file viene creato però poi dopo può crescere, può allungarsi, quindi è difficile sapere all'atto della creazione sapere qual è la strategia migliore per gestire quel file se non so quando sarà grande e soprattutto se non so se può

crescere, perché potrebbe essere piccolo inizialmente e poi diventare grande dopo, per esempio i log di sistema, quelli che vanno a memorizzare tutte le operazioni che son state svolte per questioni di diagnostica o di affidabilità o anche di sicurezza, sono file che tendono a crescere nel tempo.

File System Design

- For small files:
  - Small blocks for storage efficiency
  - Concurrent ops more efficient than sequential
  - Files used together should be stored together
- For large files:
  - Storage efficient (large blocks)
  - Contiguous allocation for sequential access
  - Efficient lookup for random access
- May not know at file creation
  - Whether file will become small or large
  - Whether file is persistent or temporary
  - Whether file will be used sequentially or randomly

Quindi nella progettazione del file system devo tener conto di tutto questo, ma non è facile come vi rendete conto, in particolare per file piccoli sarebbe meglio avere blocchi di allocazione piccoli, perché in questo modo la memorizzazione delle informazioni è più efficiente, per i file piccoli sarebbe opportuno avere blocchi di 1KB per esempio. Per i file grandi invece i blocchi di 1KB sono troppo piccoli perché è una memorizzazione inefficiente, mi servirebbero blocchi più grandi, d'altra parte la dimensione del blocco è un parametro fisso del file system. Nei file piccoli, essendo tipicamente un file piccolo memorizzato in un singolo blocco, le operazioni concorrenti sono efficienti, mentre invece nei file grandi l'accesso efficiente è sequenziale, abbiam visto che gli accessi sequenziali su disco rigido son molto più veloci e questi hanno senso, li possiamo fare se il file è grande, se il file è piccolo "non posso fare" l'accesso sequenziale, tanto letto un blocco o letto un file, è finito lì. Come si risolvono tutti questi problemi? Bisogna trovare un punto di equilibrio che non penalizzi le prestazioni per nessuno di questi aspetti, né per i file piccoli né per i file grandi e che mi permetta una memorizzazione efficiente sia dei file piccoli che dei file grandi, il risultato è che i file system hanno avuto una evoluzione fortissima negli anni e tutt'ora in effetti sul file system c'è ancora un notevole sviluppo, forse uno degli argomenti sui quali nei sistemi operativi si è investito di più ultimamente.

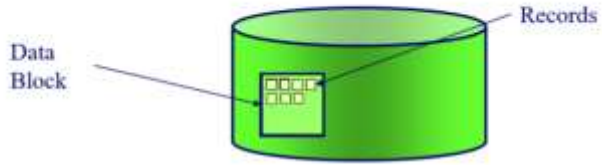
File System Design

- Data structures
  - Directories: file name -> file metadata
    - Store directories as files
  - File metadata: how to find file data blocks
  - Free map: list of free disk blocks
- How do we organize these data structures?
  - Device has non-uniform performance

Come si fa a memorizzare i dati sui dischi, intanto per poterle memorizzare bisogna avere delle strutture dati che ci permettano di capire dove le informazioni materialmente sono, noi sappiamo che un file è una sequenza di byte, per lo meno in UNIX è così, questi byte sono memorizzati in blocchi quindi un file è una sequenza logica di blocchi, insomma un certo numero di blocchi che devo leggere secondo un certo ordine per ricostruire le informazioni del file, il problema è che questi blocchi potrebbero trovarsi ovunque nel disco, devo sapere dove stanno, per avere quest'informazione devo mantenere delle strutture dati che sono mantenute dal file system e mi permettono di sapere, a partire dal nome del file, dove quel file fisicamente è collocato nel disco. Una parte di queste strutture dati sono le directory che contengono l'associazione tra nome del file e metadati. I metadati sono informazioni sul singolo file (data di creazione, data di modifica, utente, proprietario, ecc) unite anche a informazioni legate alla posizione che le informazioni di quel file occupano nel disco, quindi quali blocchi sono allocati a quel file. E poi da qualche parte ci dev'essere anche una Free map, una lista dei blocchi liberi, perché quando vado ad allocare un file dobbiamo individuare dei blocchi liberi sui quali metterlo e devo sapere quali blocchi nel disco sono liberi o occupati, questo normalmente viene gestito da una ulteriore struttura dati che è una tabella. Come si organizzano tutte queste strutture dati?

Data blocks and records

- Data in files are accessible in records
  - E.g. in Unix a record is a single byte
- Data are physically stored (and accessed) in blocks
  - In windows blocks are called clusters
- Usually block size >> record size
  - A block is often a few sectors



Partiamo dal basso, noi sappiamo quindi che il file è una sequenza di record logici, nel caso di UNIX i record logici sono byte. A livello più basso un gruppo di byte viene messo all'interno di un blocco e questo blocco è un'unità indicizzabile nel disco, tipicamente un blocco può avere 2-4 KB di dimensione, quindi un blocco è molto maggiore di un record. I blocchi in realtà sono memorizzati in settori contigui, in realtà a questo livello non ci interessa, la cosa più importante è che i blocchi nel disco sono numerati sequenzialmente, da 0 a un indice di blocco massimo, quindi se vogliamo andare a reperire il blocco che contiene questi record specifici del nostro file, ci basta sapere qual è il suo numero, qual è la sua posizione nella lista dei blocchi, quindi ci serve sapere il suo indirizzo nel disco.

Design Challenges

- Index structure
  - How do we locate the blocks of a file?
- Index granularity
  - What block size do we use?
- Free space
  - How do we find unused blocks on disk?
- Locality
  - How do we preserve spatial locality?
- Reliability
  - What if machine crashes in middle of a file system op?

Dal punto di vista della progettazione del file system dobbiamo stabilire quali sono i puntatori, quindi qual è il dominio dei puntatori, qual è l'insieme dei puntatori utilizzabili per indicizzare il disco, qual è la dimensione del blocco, qual è lo spazio libero.

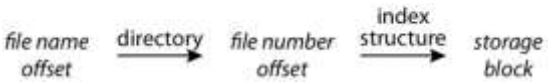
Tra l'altro sui file grossi dobbiamo privilegiare l'accesso sequenziale perché è quello che viene fatto più spesso ed è quello che maggiormente aumenta le prestazioni, dobbiamo cercare di allocare blocchi dello stesso file quanto più possibile vicini gli uni agli altri, perché in questo modo, riportati sul disco, si trovano i settori vicini e in questo modo l'accesso è molto più rapido. Non dobbiamo soltanto stabilire la dimensione dei blocchi ma quando andiamo ad allocare i blocchi per i file, dobbiamo cercare di allocarli in maniera tale da preservare proprietà di località spaziale, quindi blocchi dello stesso file devono stare più vicini possibile l'uno dagli altri, questo è vero se utilizzo dischi rigidi, se utilizzo dischi SSD questo problema non esiste più perché l'accesso ai dischi SSD è indipendente dalla località spaziale, però nei dischi SSD ho altri problemi legati alla gestione degli erasure e alle scritture.

File System Design Options

	FAT	FFS	NTFS
Index structure	Linked list	Tree (fixed, asym.)	Tree (dynamic)
granularity	block	block	extent
free space allocation	FAT array	Bitmap (fixed location)	Bitmap (file)
Locality	defragmentation	Block groups + reserve space	Extents Best fit defrag

Tutte queste problematiche hanno soluzioni differenti nei vari file system, in particolare vi presento 3 file system, sono scelti sulla base del loro utilizzo, il file system FAT è il vecchio file system dei primi personal computer, che ancora sopravvive perché di fatto è un file system talmente semplice che può essere integrato anche in dispositivi estremamente elementari ed è utilizzato ampiamente per le chiavette USB, gli altri due file system sono FFS che è il file system di UNIX, tenete presente che i file system di UNIX o di LINUX negli ultimi anni hanno avuto un'evoluzione fortissimi, e poi NTFS che invece è file system di Windows. A tutte quelle domande che vi ho posto prima, questi file system propongono approcci/soluzioni differenti. Per esempio la struttura dati che serve per tener traccia dei blocchi allocati al file, nel caso del FAT è una lista linkata, nel caso dell'FFS è un albero asimmetrico (una struttura ad albero asimmetrica e fissa), nel caso invece dell'NTFS è una struttura ad albero dinamica. La granularità delle informazioni è a livello di blocco per FAT e FFS ed è a livello di extent (che sarebbe un blocco dinamico) nel caso NTFS. Per quanto riguarda l'allocazione dello spazio libero, nel caso della FAT i blocchi liberi sono rappresentati da un array che si trova in FAT, nel caso invece dell'FFS è una Bitmap (questa è in una posizione ben precisa della partizione), nel caso dell'NTFS è una Bitmap che viene conservata in un file. Per quanto riguarda la proprietà di località delle informazioni, nel caso della FAT è lasciata al caso fino al punto in cui non se ne può più, si fa la deframmentazione che ricompatta e ripristina la proprietà di località spaziale dell'informazione. Nel caso dell'FFS invece si raggruppano i blocchi a gruppi e si riserva dello spazio per permettere ad un file di crescere in maniera localizzata. Nel caso invece di NTFS di utilizza l'extent, quindi si cerca di tenere gli extent compatti e uniti in una serie di blocchi adiacenti e in casi eccezionali si può anche qui ricorrere alla deframmentazione per ridurli. La storia della deframmentazione in NTFS è interessante perché vi fa capire come spesso ci sono delle dinamiche che si sviluppano tra progettisti e utenti che portano dei risultati davvero imprevedibili. Alle versioni originarie dell'NTFS (fine anni '90) non c'era nessun tool di deframmentazione perché i progettisti di NTFS avevano previsto tutta una serie di politiche di allocazione per cui avevano stabilito che la deframmentazione non doveva essere necessaria. È successo che gli utenti di Windows erano abituati alla deframmentazione, erano un po' come il loro piccolo cucciolotto che ogni tanto accarezzavano, quando hanno iniziato ad utilizzare NTFS nei loro sistemi e hanno visto che non c'era la deframmentazione, sono andati nel panico perché dovevano fare la deframmentazione almeno una volta al mese nella vita. Il risultato è stato che per effetto di questa necessità, più psicologica che reale, sono nate delle compagnie che hanno iniziato a produrre deframmentatori per Windows, ovviamente per Windows questo era un vero schiaffo morale, per cui a un certo punto Windows ha deciso di aggiungere il deframmentatore mettendolo a disposizione agli utenti di Windows sebbene non fosse strettamente necessario. Dopo tanti anni la deframmentazione è un po' passata di moda.

Named Data in a File System

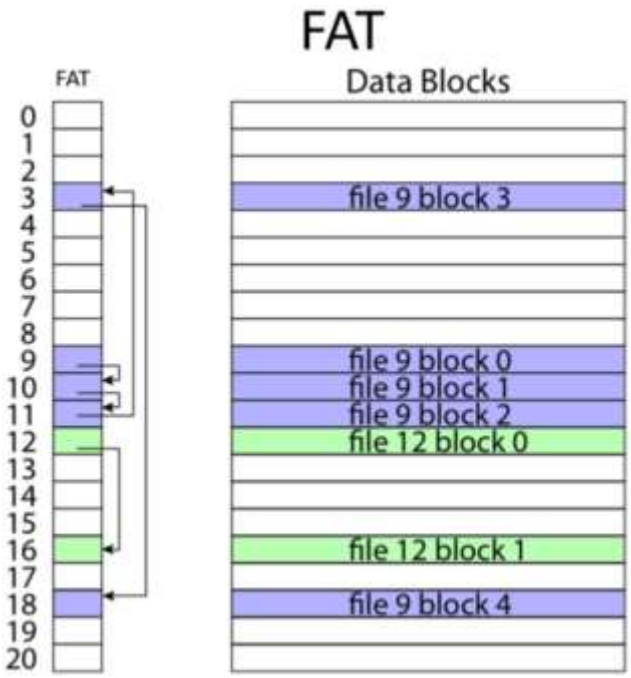


Le strutture dati, un po' in tutti i file system, sono organizzate in questi due livelli: la prima struttura dati è la directory, tramite la directory manteniamo l'associazione tra nome di file e un qualche puntatore/elemento di legame verso un'altra struttura dati che contiene i metadati e tra questi metadati in particolare, l'informazione essenziale è i blocchi allocati al file, quindi svolgendo tutta questa catena, da un nome del file possiamo sapere quali sono i blocchi fisici nel disco che lo memorizzano e in quale ordine sono messi. Quindi il passaggio è doppio: dal nome del file si accede alla directory, si tira fuori in qualche maniera un puntatore ad una struttura dati che contiene i blocchi del file. Vediamo come questo è realizzato nel file system FAT, iniziamo da lì perché è la cosa più semplice.

Microsoft File Allocation Table (FAT)

- Linked list index structure
  - Simple, easy to implement
  - Still widely used (e.g., thumb drives)
- File table:
  - Linear map of all blocks on disk
  - Each file a linked list of blocks

Il file system FAT adotta come tecnica di allocazione dei file la lista linkata. Perché la lista linkata? Perché è semplice, facile da implementare, ancora ampiamente utilizzata; questa lista linkata, però, a differenza di alcuni file system ancora più primitivi, è memorizzata all'interno di una tabella che si chiama FAT, per cui in realtà FAT è l'acronimo, File Allocation Table è il nome di questa tabella, ma siccome questa tabella è così importante in questo file system, anche l'intero file system si chiama FAT. FAT è sia il nome dell'intero file system, sia il nome della struttura dati che contiene le liste linkate dei blocchi di tutti i file presenti nel disco. Perché vi faccio questa distinzione? Perché quando si parla di file system non basta parlare soltanto della tabella che gestisce i blocchi allocati ai file, bisogna anche parlare di come sono organizzate le directory, di come è organizzata la lista dei blocchi liberi, di come viene fatta l'allocazione e via discorrendo.



La FAT è un array di puntatori, quindi tanti puntatori quanti sono i blocchi del disco, se il disco è formato da 1000 blocchi la FAT contiene 1000 elementi, ogni elemento è un puntatore. In questo esempio abbiamo due file memorizzati nel disco, il file 9 e il file 12; il file 12 è memorizzato nel blocco 12 e nel blocco 16, esattamente in questo ordine, prima il blocco 12 e poi il blocco 16, il file 9 invece è memorizzato invece a partire dal blocco 9, poi occupa il blocco 10, il blocco 11, il blocco 3, il blocco 18 e poi termina. Come vedete il file 9 ha tre blocchi adiacenti, quindi quando andiamo a fare un accesso sequenziale per questi tre blocchi l'accesso sarà veloce, dobbiamo pagare il tempo di seek soltanto all'inizio e poi dopo non lo paghiamo praticamente per i restanti tre blocchi, poi però letto il blocco 2 dobbiamo fare un seek per andare nella traccia dove c'è il blocco 3 e letto il blocco 3 dobbiamo fare un altro seek per andare a leggere il blocco 4. Da un punto di vista dell'efficienza dell'accesso sarebbe stato più utile avere tutti i blocchi compatti, però questo non era possibile farlo perché quando è stato allocato il file 9, presumibilmente i blocchi adiacenti a questi tre erano occupati e quindi il file system è andato a cercare delle altre posizioni. In realtà il file system FAT non ha fatto neanche questo, perché quando gli dite di allocare un file, lui va a scorrere la lista dei blocchi liberi, il primo blocco libero che trova lo utilizza senza preoccuparsi di nessun aspetto di località ed è questo il motivo per il quale i file nel file system FAT possono essere distribuiti in maniera così arbitraria ed è per questo che le prestazioni possono degradare selvaggiamente ed è per questo che serve la deframmentazione , per ricompattare i blocchi dei file. La locazione di questi due file è memorizzata all'interno di questa struttura dati della FAT in questo modo: sappiamo che il primo blocco del file 9 è il blocco fisico numero 9, l'associazione tra file 9 e blocco numero 9 è mantenuta all'interno della directory che contiene il file, quindi quando voi date il nome del file, dalla directory estraete il primo blocco nel quale questo file è allocato e dal valore 9 voi potete leggere il primo blocco del file e potete anche leggere l'elemento numero 9 della FAT. Dell'elemento numero 9 è l'inizio della lista linkata, quindi contiene il puntatore al blocco successivo, quindi all'interno dell'elemento 9 della FAT c'è scritto 10, all'interno dell'elemento 10 della FAT c'è il puntatore al blocco successivo che è 11, all'interno dell'elemento 11 della FAT c'è il puntatore al blocco successivo che è 3, all'interno dell'elemento 3 della FAT c'è il puntatore al blocco successivo che è 18 e all'interno dell'elemento 18 della FAT c'è scritto che questo è l'ultimo, quindi qui ci sarà scritto -1 (oppure 0, insomma un valore non utilizzabile) in maniera tale che chi scorre la lista linkata sappia che quello è l'ultimo blocco del file. Per il file 12, invece, nella directory c'è scritto che il file 12 comincia nel blocco 12, quindi letta la directory sappiamo che inizia il blocco 12, possiamo leggere il primo blocco del file 12 e però possiamo anche leggere l'elemento 12 della FAT nel quale troviamo il puntatore 16 al blocco successivo e nell'elemento 16 della FAT troviamo il valore che ci indica terminazione

di file. Spesso negli esercizi questo è indicato con -1, con 0, che sono valori non ammissibili per il puntatore. Come vedete la struttura dati per allocare file è semplicissima, i vantaggi sono che è facile trovare un blocco libero; come è memorizzata la lista dei blocchi liberi? In realtà i blocchi liberi sono memorizzati in questa tabella stessa marcati come liberi, quindi il valore contenuto per i blocchi liberi sarà un valore specifico che indica che quel blocco è libero. Se devo allocare un nuovo file e mi servono 4 blocchi liberi, scandisco la FAT dall'inizio, trovo che i primi 4 posti liberi sono utilizzabili per quel file e quelli lì li alloco al file e di conseguenza li leggo in una lista linkata. In questo modo è facile trovare un blocco libero, è facile aggiungere informazioni a un file, se io voglio far crescere un file, trovo degli altri blocchi liberi, li collego alla lista linkata e in questo modo allungo il file liberamente ed è anche facile cancellare un file, per cancellare un file devo marcare come liberi tutti quanti i blocchi in una lista linkata. Quali sono invece i punti di debolezza della FAT? La FAT dove si trova, dove deve essere conservata? La tabella FAT deve stare obbligatoriamente sul disco, sta nel disco nei primi blocchi del disco: se aprite un disco e una sequenza di blocchi, a parte il blocco 0 e il blocco 1 che sono riservati per scopi particolari, i blocchi successivi contengono la FAT e devono stare lì perché le informazioni fatte nella FAT sono critiche: se perdete la FAT poi come fate a sapere dove sta allocato un file? Perdete il contenuto del disco, le informazioni ci sono ancora ma non avete più un ordine, una coerenza. Se la FAT è memorizzata nel disco, se voi sapete che il file inizia dal blocco 9, potete intanto leggere il blocco 9 (un primo accesso al disco), poi però dovete leggere il blocco del disco che contiene questo pezzo della FAT, quindi un secondo accesso al disco, da qui avete il puntatore al blocco successivo, magari stanno nello stesso blocco che avete letto sennò dovete leggere un altro blocco della FAT per sapere qual è la posizione del blocco successivo, quindi se siete molto sfortunati, in realtà, per leggere 5 blocchi di un file rischiate di dover fare 10 letture dal file, 5 per leggere i blocchi e 5 per leggere i blocchi della FAT nei quali ci sono i puntatori. Prendiamo il file 12, supponiamo che debba leggere il file 12, il file 12 inizia al blocco 12, quindi leggo il blocco 12 e ottengo il primo blocco del file, poi devo leggere il blocco che contiene il puntatore 12 per sapere qual è l'elemento successivo e ho fatto la seconda lettura, qui scopro che il blocco successivo è 16 quindi posso leggere il blocco 16 e ho letto il secondo blocco del file (terza lettura), a questo punto devo leggere il blocco della FAT che contiene l'elemento 16 (quarta lettura), fatta la quarta lettura scopro che il file è terminato. Quindi in realtà per leggere due blocchi ne ho letti 4, non è un modo efficiente di usare il disco, per questo motivo in realtà la FAT, quando voi inserite la pendrive usb, come prima cosa viene caricata in memoria principale, caricando la FAT in memoria principale tutte le operazioni sulla FAT si fanno in memoria quindi son velocissime e a questo punto leggere il file mi comporta soltanto la lettura dei blocchi effettivi del file, niente di più, niente di meno.

FAT

- Pros:
  - Easy to find free block
  - Easy to append to a file
  - Easy to delete a file
- Cons:
  - FAT size
    - Should be uploaded in main memory
    - Limitation to file system size
  - Limited metadata and no protection
  - Fragmentation
    - File blocks for a given file may be scattered
    - Files in the same directory may be scattered
    - Problem becomes worse as disk fills

Ora però abbiamo un problema: se la FAT sta normalmente sul disco, ma quando il computer acceso deve stare in memoria principale, quant'è grande questa FAT? Con le tecnologie attuali fa ridere perché la tecnologia FAT non si è più evoluta, se lo riportiamo nel periodo nel quale la FAT era utilizzata, non fa più tanto ridere. La dimensione della FAT è un problema e la FAT in memoria me la devo tenere tutta, indipendentemente dai file che voglio effettivamente leggere o scrivere, perché non so dove sono collocati a priori, quindi comunque me la dovrò caricare, quindi in memoria dovrà stare per forza. Poi altri problemi piuttosto gravi sono il fatto che nel file system FAT non ho tanti metadati, ho poche informazioni sui file ed in particolare non ho informazione di protezione, questo è un limite molto forte, d'altra parte la FAT era pensata per personal computer ad uso personale, non si pensava che un utente dovesse proteggersi da sé stesso, nei sistemi attuali invece si, gli utenti vanno anche protetti a sé stessi, questo è un limite grosso della FAT e l'altro limite grosso è legato alla frammentazione, non fa niente per cercare di mantenere i file compatti, di tenere i blocchi allocati ai file compatti, e questo vuol dire che allocando a caso i blocchi può peggiorare in maniera drammatica la prestazioni.

Limitazioni del file system FAT

Posto:
- L lunghezza (in bit) degli elementi della FAT
- B la dimensione(in byte) dei blocchi del disco,

Il numero di blocchi indirizzabili è 2^L (capacità del disco, o partizione)

la massima estensione del file system è:
2^L blocchi, ovvero a $B \cdot 2^L$ byte.

se ogni elemento occupa N byte (solitamente L è multipla del byte), la FAT occupa complessivamente: $N \cdot 2^L$ byte

Facciamo due conti riguardo la dimensione della FAT; supponiamo di avere puntatori di L bit, un valore tipico di L è 16 o 32 e supponiamo che il blocco abbia dimensione B. Siccome ho L bit per gli indirizzi, il numero di blocchi che posso indirizzare è 2^L e questo 2^L è la massima capacità del disco, della partizione che posso gestire in termini di blocchi. In termini di byte la dimensione massima di quella partizione è B*2^L, ho 2^L blocchi moltiplicato per la dimensione del blocco. Quant'è grande la FAT? La FAT deve contenere 2^L elementi ognuno dei quali è un puntatore, se L è un multiplo di un byte, per esempio L è 16 bit o 32 bit, la dimensione della FAT è un multiplo di 2^L, potrebbe essere 2*2^L o 4*2^L, N tipicamente è Log2(L) arrotondato all'intero superiore, quindi se ho un puntatore a 32 bit, il puntatore a 32 bit sta in 4Byte e quindi ogni riga della FAT contiene 4Byte, siccome ci sono 2^L elementi, 4*2^L è la dimensione della FAT. Vediamo di fare due conti più precisi;

Limitazioni del file system FAT

Esempio:

con N=2 (→ FAT 16) e B= 2^{10} (blocchi di 1 Kbyte)

La massima estensione del file system è 2^{16} blocchi, ovvero 2^{26} byte (= 64 Mbyte)

la FAT occupa complessivamente 2* 2^{16} byte= 128 Kbyte

- con una memoria paginata e pagine di 1Kbyte, la FAT occupa 128 pagine

dato che gli elementi che descrivono un file possono essere distribuiti su molte pagine diverse, possono verificarsi frequenti errori di pagina quando si percorre un file.

→ Per realizzare file systems più estesi si usano blocchi di dimensioni maggiori.

Supponiamo di avere FAT16, quindi puntatori a 2 Byte, e supponiamo che i blocchi abbiano dimensione di 1Kb, quindi B=2^10, allora la massima estensione del file system è 2^16 blocchi, quindi 2^26 Byte che vuol dire 64 MByte, poco, il secondo computer che ho avuto aveva un disco di 20Mb e aveva FAT16, stiamo parlando dei primi anni '90. In queste condizioni quanto spazio occupa la FAT? La FAT occupa

2^16 righe * 2 Byte perché sono 2^16 blocchi, quindi ogni blocco deve avere una riga della FAT, ognuno contiene un puntatore di 2 Byte, quindi la FAT occupa 128 Kb, pochissimo. Nel computer dei primi anni '90 avevo un disco da 20Mb e una memoria principale di 512Kb, con un disco di 20Mb la FAT era lunga grossomodo 40Kb, questo vuol dire che la mia memoria principale per circa 1/5 era occupata dalla FAT, 1/5 della memoria principale per memorizzare la FAT è un'enormità, adesso è come se vi dicessero che avete una memoria principale di 5Gb ma 1Gb è occupato dalla FAT e poi il resto c'è il S.O. e tutte le altre strutture dati. Quindi è un Overhead molto pesante non trascurabile. Ora fa ridere perché il mondo si è evoluto, siamo andati avanti, le memorie hanno aumentato di capienza, 64Mb sono ridicoli, però sono aumentate anche le dimensioni dei dischi, se dovessi gestire un disco da qualche Terabyte con la FAT, l'occupazione di memoria con la sola FAT non è più tanto trascurabile, inizia a incidere per un 10% della memoria attuale, però sempre troppo oggettivamente. Questi son due calcoli fatti su una FAT16, in realtà di FAT ne esistono almeno tre tipi: FAT12, FAT16, FAT32

Limitazioni dei file systems FAT

Block size	FAT-12	FAT-16	FAT-32
0.5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

Massima dimensione del File System per diverse ampiezze dei blocchi

In questa tabella vedete la dimensione del blocco e di conseguenza la massima dimensione del disco che posso gestire, con FAT16, usando blocchi da 32Kb che è un'enormità, posso avere dischi da 2 Tb, questi dischi da 2Tb sono ottenuti al costo di avere blocchi da 32 Kb, questo vuol dire che se allocate un file di 100 Byte, quel file di 100 Byte è conservato in un blocco di 32 Kb nel quale in larga misura questo blocco è inutilizzato, quindi non è molto consigliabile lavorare in questa fascia, ma neanche in questa, la dimensione ragionevole per i blocchi tipicamente è di 2-4 Kb, questo vuol dire che con FAT16 possiamo gestire al massimo dischi da 256 Mb e con FAT32 possiamo gestire dischi da 1 Tb. Ci sono queste limitazioni, c'è questo tappo sopra al 2, perché in realtà nella dimensione del disco che si può gestire intervengono tanti colli di bottiglia, la dimensione del puntatore non è l'unico collo di bottiglia, ci sono anche altri limiti in termini di strutture dati interne al file system che indicano la massima dimensione dei dischi, in ogni caso, a meno di casi realmente eccezionali non è consigliabile utilizzare la FAT per gestire dischi da 1-2 Tb, è stato fatto in passato però ora non è più consigliabile da fare.

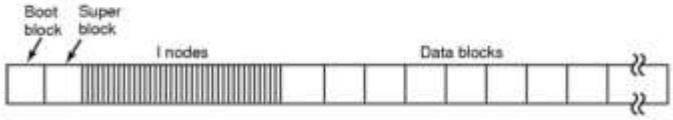
Berkeley UNIX FFS (Fast File System)

- inode table
  - Analogous to FAT table
- inode
  - Metadata
    - File owner, access permissions, access times, …
  - Set of pointers to data blocks

FFS è un file system che è nato, si è sviluppato, all'interno del mondo UNIX parecchi anni fa e di nuovo questo va a suo merito, i file system che utilizziamo attualmente nei sistemi Linux/UNIX sono evoluzioni dell'idea originale, vi presento uno di questi file system intermedi, le ultimissime versioni che trovate su Linux sono ulteriori evoluzioni, hanno diversi meccanismi in più. Nel caso dell'FFS tutto quanto gira intorno alla tabella degli i-Node, la tabella degli i-Node è la struttura dati in FFS che contiene associazione tra file e blocchi che lo allocano nei quali lui è contenuto. L'i-node non contiene soltanto quest'associazione ma contiene anche tanti metadati, tante informazioni associate ai file, inclusi i bit di protezione.

Quindi l'i-Node contiene metadati e contiene puntatori.

Physical disk organization in UNIX

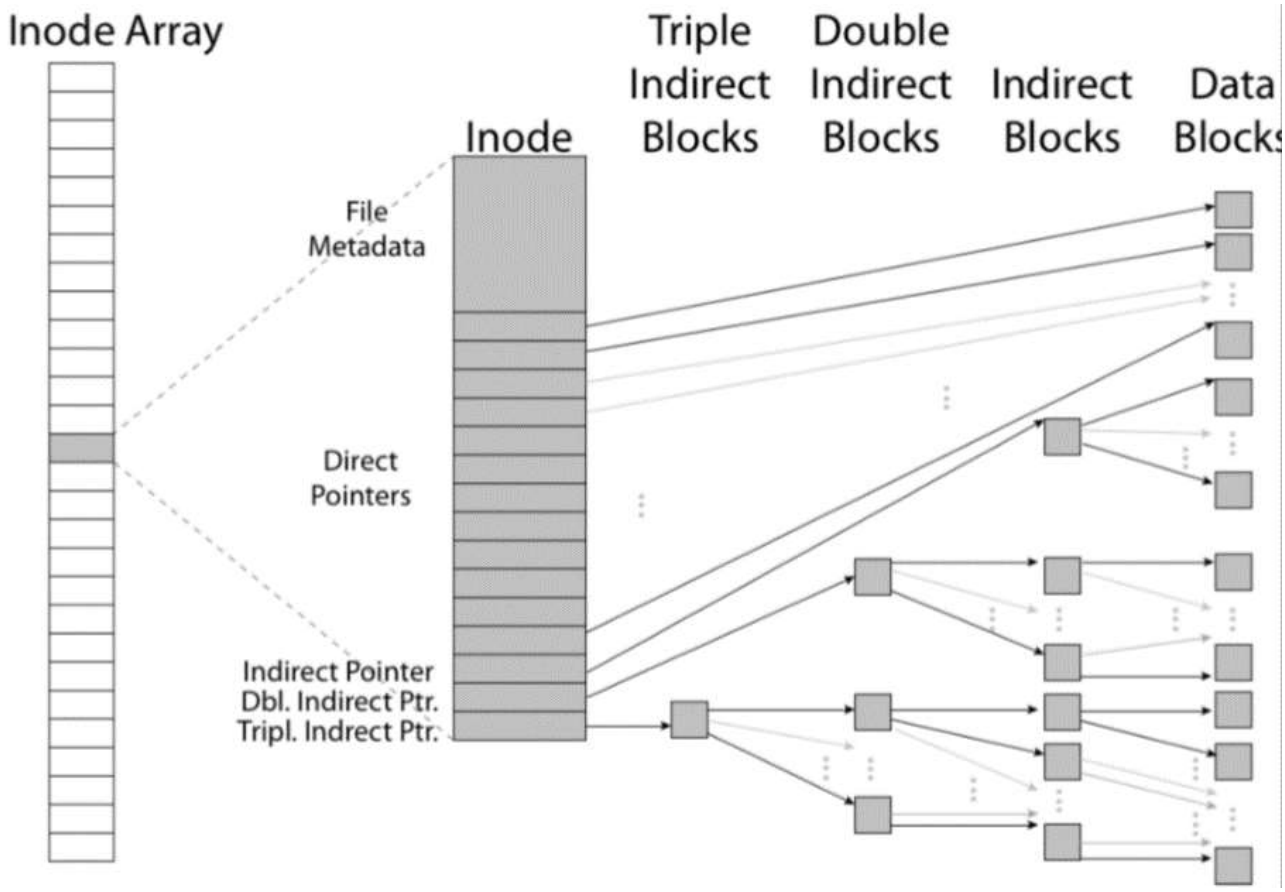


Dal punto di vista fisico, per lo meno in astratto, un disco UNIX prende questa struttura: i primi due blocchi sono riservati, in particolare il blocco 0 è il blocco di Boot, il blocco 1 è il superblocco, il blocco di Boot contiene informazioni che abbiamo visto nella prima lezione per fare il boot del sistema, il superblocco invece contiene informazioni relative a questo file system, quindi ci dice che qua dentro c'è FFS di una certa versione e ci dà i parametri di questo file system. Dopo il superblocco ci sono una serie di blocchi (i blocchi hanno la stessa dimensione) però in realtà ci sono una serie di blocchi che memorizzano la tabella degli i-Node (la i-List) e che quindi sono organizzati in record, terminata la i-List, in realtà ci sarebbe la Bitmap (la mappa dei blocchi liberi/occupati del disco) e poi ci sono questi dati.

FFS inode

- Metadata
  - File owner, access permissions, access times, …
- Set of 12 data pointers
  - With 4KB blocks => max size of 48KB
- Indirect block pointer
  - pointer to disk block of data pointers
  - 4KB block size => 1K pointers to data blocks => 4MB
- Doubly indirect block pointer
  - Doubly indirect block => 1K indirect blocks
  - 4GB (+ 4MB + 48KB)
- Triply indirect block pointer
  - Triply indirect block => 1K doubly indirect blocks
  - 4TB (+ 4GB + 4MB + 48KB)

L'i-Node contiene queste informazioni: informazioni sulle proprietà del file, informazioni su permessi di accesso, istante di ultimo accesso, istante modifica, creazione, numero di hard link, tutta una serie di informazioni e poi in particolare ci sono i puntatori ai blocchi nei quali il file è allocato. Questi puntatori sono organizzati nell'i-Node usando una struttura ad albero fisso e asimmetrico, in particolare nell'i-Node son memorizzati 15 puntatori, di cui 12 puntatori sono puntatori a blocchi diretti, quindi i primi 12 puntatori contenuti nell'i-Node puntano direttamente a dei blocchi dati, chiaramente questo permette di rappresentare dei file piccoli, un file piccolo è interamente rappresentato dal suo i-Node, non ho bisogno di altro, un file piccolo che è contenuto in 12 blocchi. Se però il file è un po' più grande, questi 12 puntatori diretti non mi bastano, in realtà dovrei avere un numero di puntatori che cresce in funzione della dimensione del file, però questo è un problema in UNIX perché gli i-Node sono strutture dati a dimensione fissa, non sono a dimensione variabile, il fatto che gli i-Node abbiano dimensione fissa permette di gestire bene il loro utilizzo all'interno della i-List per cui se il file è più lungo, i puntatori aggiuntivi ai blocchi ulteriori non sono memorizzati nell'i-Node ma sono memorizzati in alcuni blocchi dati che si trovano nel resto del disco e l'i-Node ne contiene il puntatore, in particolare l'i-Node contiene 3 puntatori a blocchi di indirizzi, che sono puntatori diretto singolo doppio e triplo.



Qui vedete l'i-List, questo è un i-Node dell'i-List, quest'i-Node è esploso, quindi dimensione ingrandita, l'i-Node contiene metadati, poi contiene 12 puntatori diretti e tramite questi puntatori posso risalire i primi 12 blocchi del disco; i blocchi del disco li sto rappresentando piccoli, ma ognuno di questi sono 4 Kb, l'i-Node potrebbe essere 128-256 Byte, quindi l'i-Node è molto più piccolo di ognuno di questi blocchi, appunto è ingrandito, gli ultimi 3 puntatori dell'i-Node sono i puntatori indiretti: il primo puntatore indiretto punta a un blocco indice di terzo livello, questo blocco indice è un blocco dati, quindi potrebbe essere grosso 4 Kb o 2 Kb a seconda della dimensione dei blocchi in questo file system e contiene puntatori a blocchi dati del disco. Quindi se i puntatori sono di 4 Byte (a 32 bit), i blocchi sono di dimensione 4 Kb, qua dentro troviamo 1024 puntatori che indicizzano 1024 blocchi dati del file. Se il file non occupa più di

1024 + 12 = 1036 blocchi, quindi grossomodo se non è più grande di 4 Mb o poco più, per rappresentare il file mi basta soltanto questo blocco qui e l'i-Node, quindi gli ultimi due puntatori non mi servono, se il file è più grande invece, posso ricorrere all'indirizzamento indiretto doppio, per cui il puntatore indiretto doppio punta a un blocco indice che contiene puntatori ad altrettanti blocchi indice ognuno dei quali contiene altrettanti puntatori a blocchi dati, quindi il numero di blocchi dati che indicizzo con un indirizzamento indiretto doppio, cresce di un ordine quadrato, e per il triplo è lo stesso gioco fatto tre volte. Il risultato è che la struttura dati che rappresenta questo file è un albero asimmetrico, non è simmetrico infatti, per il terzo puntatore abbiamo tre livelli, per il secondo puntatore due livelli, per il primo puntatore abbiamo un solo livello, quindi asimmetrico e perché è asimmetrico? La struttura è fissa perché deve crescere secondo questa struttura preordinata, non posso cambiarla, perché in UNIX si adotta questo modello con albero asimmetrico fisso? Perché questa struttura è efficiente per rappresentare file di dimensioni differenti, file piccoli son memorizzati con poche informazioni con solo i-Node, file medi mi richiedono un po' più di informazioni, file grandi ne richiedono ancora di più. Quindi in qualche modo la dimensione della struttura che rappresenta un file è legata alla dimensione del file, più è grande il file, più costa la struttura in termini di Overhead di memoria e tempo di accesso. Non abbiamo finito parlando dell'i-Node, parlando del file system FFS, finiamo la prossima lezione.

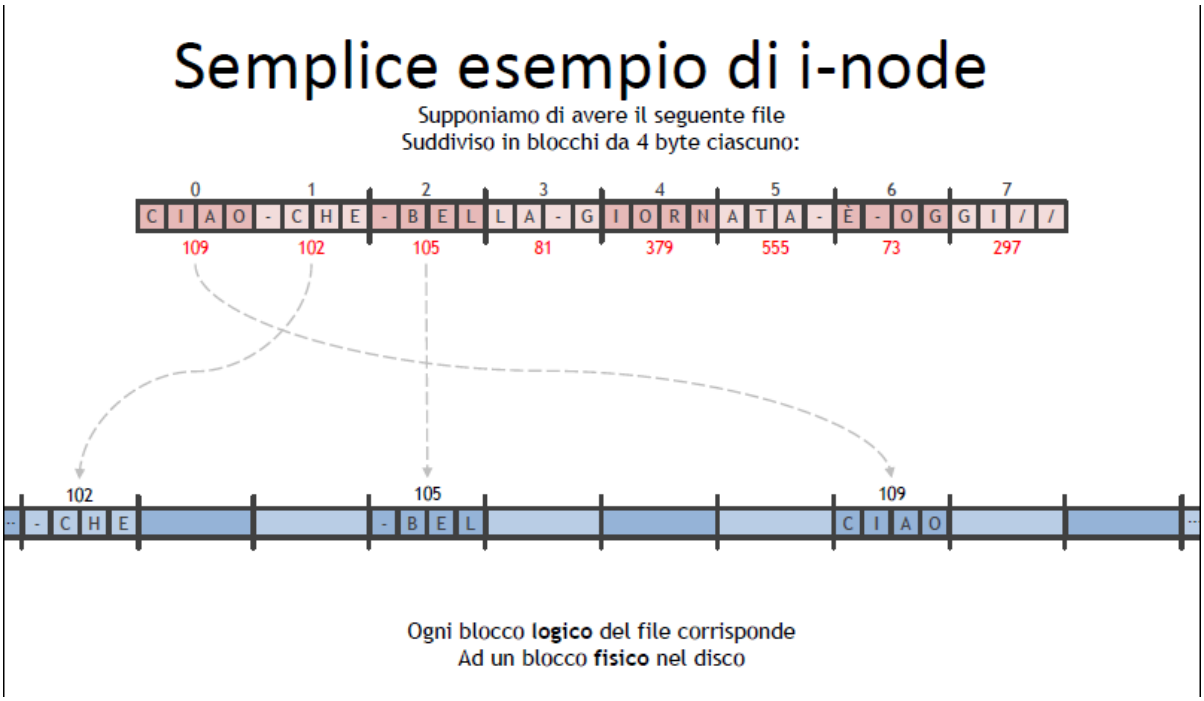
Sistemi operativi 25/05/17

Vediamo un esempio di come funziona la struttura I-Node. (Questi lucidi me li ha fatti un vostro collega l'anno scorso dato che all'esame aveva qualche difficoltà con l'i-node, dopo che gliel'ho rispiegato mi ha prodotto questi lucidi). Facciamo un esempio molto semplice.

Esempio (banale) di i-node

- Blocchi di 4 byte
- Puntatori di 2 byte
- File di 26 byte (quindi 7 blocchi)
- I node con:
  - 2 puntatore diretti
  - 1 puntatore indiretto singolo
  - 1 puntatore indiretto doppio

Ipotizzando di avere blocchi di 4 byte, puntatori fatti di 2 byte e prendiamo un file di 26 byte: tale file vuol dire che è composto da 7 blocchi di cui tutti i blocchi sono pieni e l'ultimo blocco occupa soltanto 2 byte e 2 byte invece sono vuoti. Supponiamo che l'i-node abbia 2 puntatori diretti un puntatore diretto singolo e un puntatore indiretto doppio. Quello che succede è questo:



Supponiamo di avere il file in figura, che contiene una serie di stringhe ("Ciao che bella giornata è oggi"): queste stringhe sono partizionate in blocchi e i blocchi a loro volta sono allocati su blocchi fisici del disco in posizione arbitraria. Per esempio il primo blocco (il blocco 0) del file allocato sul blocco fisico 109, il blocco logico 1 del file allocato sul blocco fisico 102 hanno i contenuti come in figura. Come potete vedere, in

realtà, il file è frammentato e le informazioni sono memorizzate all'interno del disco in maniera disordinata. In realtà logicamente le informazioni sono ordinate perchè il file ha la struttura che vedete in alto, per cui il contenuto dei blocchi deve essere preso e messo esattamente in quell'ordine quando ne facciamo l'accesso. Chi gestisce la corrispondenza tra file e blocchi fisici sul quale il file è mappato sappiamo che è l'i-node. A questo punto scrivete l'i-node di questo file. Lì in rosso vedete che ci sono i puntatori ai vari blocchi che compongono il file.

(Passati due minuti)

Non sapete che pesci prendere? L'i-node è una struttura dati che contiene 2 puntatori diretti, un puntatore diretto singolo e un puntatore indiretto doppio (poi contiene dei metadati che però in questo esercizio qui non ci interessano, quindi non li riempiamo). Per scrivere l'i-node dovete scrivere i valori dei contenuti dei 4 puntatori. Quello che dovete fare è una tabella con 4 elementi e vi dovete scrivere il contenuto di essi (sono 4 puntatori). I primi due sono puntatori diretti, quindi:

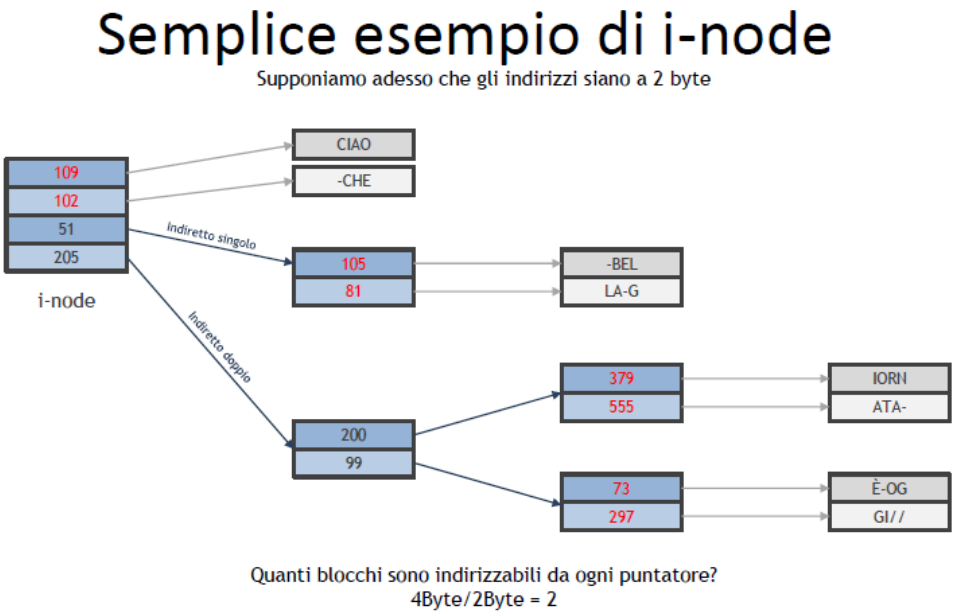
- il primo contiene 109, perchè è diretto al primo blocco del file;
- il secondo è un puntatore diretto al secondo blocco del file e contiene 102;
- il terzo puntatore nell'i-node è un puntatore ad un blocco indice di terzo livello.

Se vogliamo completare la struttura dovremmo decidere dove si trova questo blocco indice di terzo livello e che cosa contiene. Esso si trova in un punto qualsiasi del disco che contenga un blocco libero (potrebbe essere un indirizzo qualsiasi, basta che non sia un indirizzo occupato). Ora vi chiedo: che cosa contiene il blocco indice di terzo livello, quindi associato al puntatore indiretto singolo.

(Passati due minuti)

Faccio 2 passi indietro. Ricordatevi che il file è diviso nei blocchi logici 109, 102, 105, 81, ... Sappiamo questo. L'i-node è fatto in questa maniera: ha un certo numero di puntatori diretti a blocchi del file (il nostro i-node ne ha 2 di puntatori e questi puntano ai primi 2 blocchi del file, quindi contengono 109 e 102). Il blocco dati successivo al 102 è 105 e quello successivo ancora è 81. Consideriamo 105 e 81: in questo blocco indiretto noi manteniamo i puntatori al blocco 105 e all'81. Il contenuto del blocco indiretto di terzo livello non sono altro che i puntatori a questi due blocchi del file 105 e 81. Non ce ne stanno di più perchè i blocchi sono formati da 4 byte, gli indirizzi sono di 2 byte ciascuno, quindi posso tenere 2 indirizzi in un blocco indice. Ora ho finito, nel blocco indice ci sta 105 e 81, devo decidere dove metterlo questo blocco indice.

Dove lo metto? Dove c'è spazio. Potrei metterlo nel blocco 103, come anche nel 107, nel 111... Potete sceglierlo liberamente, basta che sia un blocco libero. Nell'i-node ci sarà il puntatore di questo blocco indice, per esempio il 103, l'i-node conterrà primo puntatore 109, secondo puntatore 102, terzo puntatore 103 (è il puntatore al blocco indice) che al suo interno contiene 105 e 81. Con un blocco indice di terzo livello, quindi con un puntatore indiretto singolo, sono riuscito a rappresentare i primi 4 blocchi del file, ma il file ha altri 4 blocchi. Devo ricorrere al puntatore indiretto doppio. Esso punta ad un blocco che contiene, nel nostro esempio, i puntatori a due blocchi indiretti di terzo livello ognuno dei quali contiene i puntatori a 2 blocchi dati, che sono quelli che vengono dopo, nello stesso ordine in cui li avete visti. Ci devono stare 379, 555, 73 e 297. Questo vuol dire che il puntatore indiretto doppio punta ad un blocco indice di secondo livello che punta a 2 blocchi indice di terzo livello che contengono il primo 379 e 555 e il secondo gli ultimi 2. Quello che devo fare è trovare il posto nel disco dove piazzare questi 3 blocchi, legarli tra loro e inizializzare il puntatore indiretto doppio dell'i-node. Trovo 3 posti liberi nel disco, metto questi 3 blocchi e di conseguenza fisso gli indirizzi. Il puntatore indiretto triplo non mi serve perchè il file è finito lì. Questa è quindi la struttura che vado a creare:

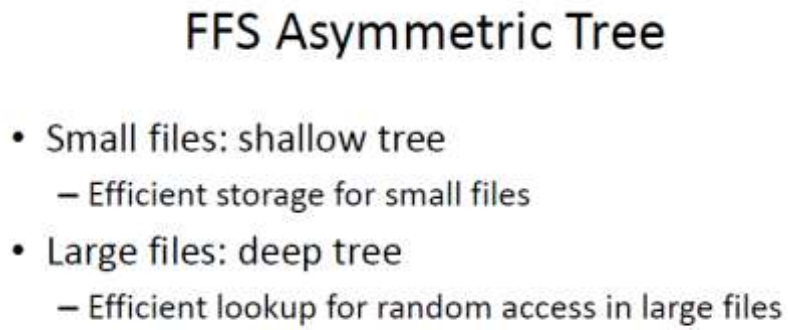


dove alcune scelte sono totalmente arbitrarie, in particolare:

il fatto che qui ci sia 51 e qui 205 è arbitrario, l'ho scelto io trovando un posto libero nel disco

Il fatto che qui ci sia scritto 105 e 81 non è arbitrario, perchè questi sono i blocchi del file, sono obbligato a metterli così.

Anche questi (200, 99) sono arbitrari, li ho scelti in maniera tale da poter allocare dei blocchi indice. La struttura che rappresenta il mio file nel suo complesso è questa: quelli grigi sono i blocchi dati con il loro contenuto che posso scorrere nell'ordine giusto (posso ricostruire l'ordine corretto dei vari record logici del file, anche se fisicamente sono tutti sparpagliati. Questo è il contenuto dell'i-node e questi sono i contenuti dei blocchi indici. Sui blocchi indici che hanno i numeri scritti in rosso non posso farci niente, sono obbligato a metterli così, i blocchi indice che hanno i numeri scritti in nero li ho allocati io dove ho trovato spazio nel disco.

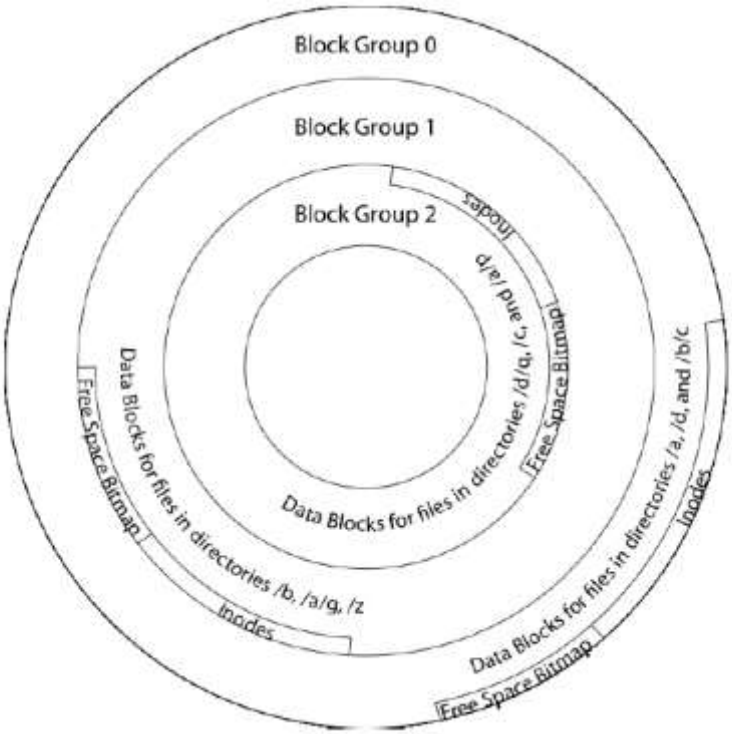


Fatto in questo modo l'FFS utilizza gli alberi asimmetrici: questo permette di essere efficiente per memorizzare file piccoli e d'altra parte non penalizza la memorizzazione dei file grandi. I file grandi hanno un albero più profondo, perchè utilizzano il secondo, il terzo livello, quindi l'accesso sui file grandi è un pochettino più pesante, un po' più lungo, ma d'altra parte questo è accettabile perchè quando si legge un file grande si sa che si dovrà aspettare più tempo. Quindi la penalizzazione maggiore per i file grandi incide poco rispetto al tempo totale di accesso al file. Un'ultima questione sempre legata all'FFS è l'aspetto della località.

FFS Locality

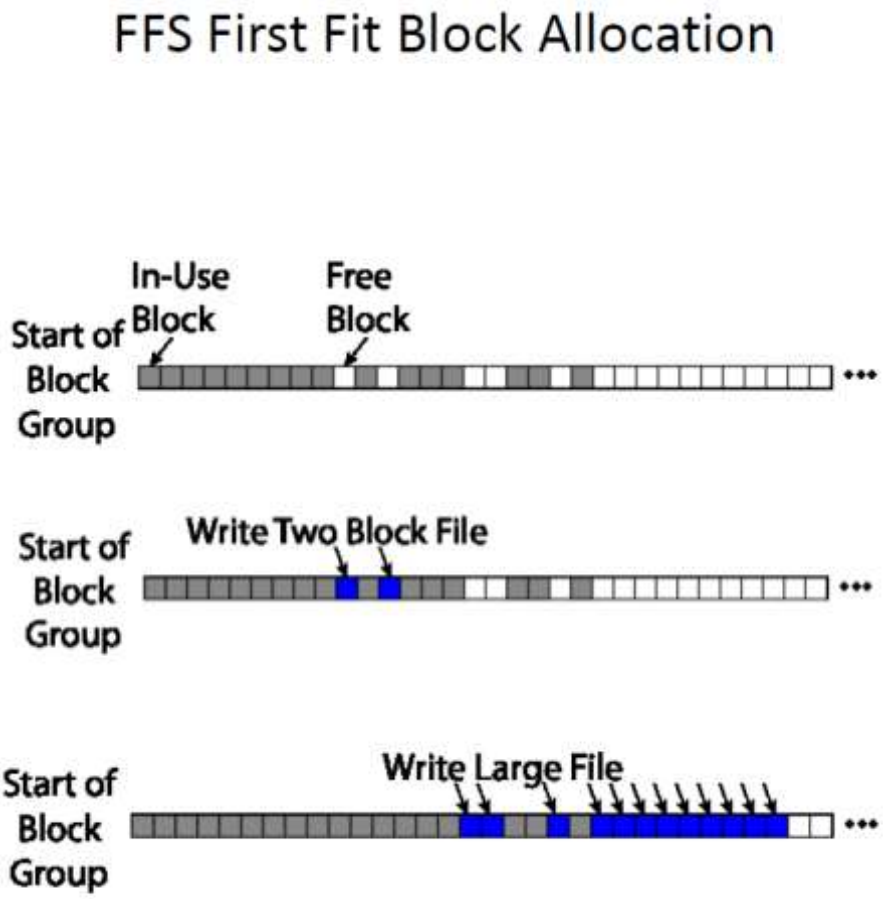
- Block group allocation
  - Block group is a set of nearby cylinders
  - Files in same directory located in same group
  - Subdirectories located in different block groups
- inode table spread throughout disk
  - inodes, bitmap near file blocks
- First fit allocation
  - Small files fragmented, large files contiguous

Sappiamo che per accedere efficacemente ad un file grande è necessario che tutti i suoi blocchi siano ravvicinati l'uno con l'altro, però non solo: per leggere un file grande devo leggere anche i suoi blocchi indice e devo leggere anche il suo i-node. Quindi quando vado a fare l'accesso, in realtà, dovrò contare l'accesso alla directory che contiene l'i-node, nella quale trovo il puntatore al suo i-node, accedo al suo i-node; a questo punto posso leggere un po' di blocchi dati, poi dovrò leggere i blocchi indice, poi dovrò leggere i blocchi dati puntati dai blocchi indice e tutte queste operazioni comportano dei seek nel disco, degli spostamenti della testina. Se voglio essere efficiente bisogna che la directory, l'i-node, i blocchi indice e i blocchi dati del file stiano tutti quanti abbastanza vicini nella stessa zona del disco. Per fare questo unix FFS divide il disco in macro aree e per quanto possibile cerca di allocare un'intera directory con tutti i file in essa contenuti all'interno della stessa area.



Per cui il disco materialmente non ha un'unica zona dove sono presenti tutti gli i-node, un'unica i-list che non ha un'unica vittma ma in realtà ha tante aree e in ogni area trovate l'ilist, la vittima per quest'area e via

discorrendo. Per quanto possibile cerca di mettere una directory file in essa contenuti tutti all'interno della stessa area. Ovviamente nei limiti del possibile. In questo modo un accesso ad un file o a più file della stessa directory comporta dei tempi di seek ridotti perchè ci stiamo muovendo all'interno di una stessa area del disco. Oltre a questo utilizza un metodo di allocazione first fit, per cui ad allocare i blocchi a gruppi in maniera tale da tenerli il più possibile vicini gli uni agli altri.



Questa slide sul first fit mi sono reso conto non è fatta particolarmente bene, quindi la rifarò per il futuro, ma non è argomento nel quale vi propongo esercizio nè al compito nè all'orale. Se volete avere più informazioni riguardo al first fit fate riferimento al libro oppure anche ad altri testi dove è descritto meglio.

FFS

- Pros
  - Efficient storage for both small and large files
  - Locality for both small and large files
  - Locality for metadata and data
- Cons
  - Inefficient for tiny files (a 1 byte file requires both an inode and a data block)
  - Inefficient encoding when file is mostly contiguous on disk (no equivalent to superpages)
  - Need to reserve 10-20% of free space to prevent fragmentation

I punti di forza dell'FFS:

- è efficiente nella memorizzazione, l'overhead per file piccoli è piccolo (per cui utilizzo una struttura dati ridotta);
- garantisco la località per i file piccoli e grandi, per i dati ed i metadati perchè tendo ad allocare all'interno dello stesso gruppo di tracce, file e directory vicini;

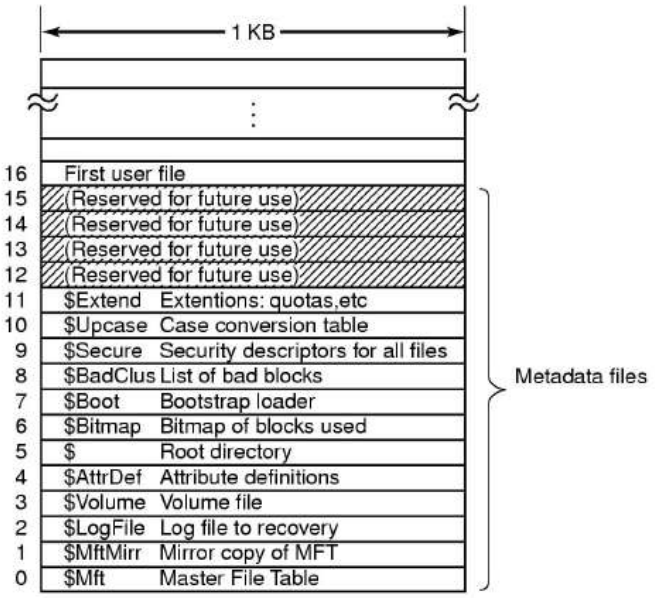
Può però essere inefficiente per file davvero molto piccoli perchè un file di un byte richiede comunque un i-node e un blocco di dati. C'è una piccola inefficienza legata al fatto che se il file nel disco è contiguo, quindi tutti i suoi blocchi sono uno di seguito all'altro io comunque devo memorizzare nella struttura dell'i-node e dei blocchi indice tutti questi puntatori, in effetti se un file fosse contiguo potrei memorizzarlo in maniera molto efficace ricordandomi il blocco iniziale e la sua lunghezza, ma questa rappresentazione in FFS non è utilizzata. Mi servirebbe introdurre in FFS un concetto di superblocco, superpagina, quindi una serie di blocchi tutti contigui. Questo concetto esiste in NTFS (si chiama Extent) e credo che sia stato introdotto anche in FFS nelle ultime versioni. Poi per prevenire la frammentazione bisogna comunque rilavorare con un disco che ha il 10-20% di spazio libero, ma questo in realtà vale per tutti i file system. Se voi su un file system lavorate con il disco pressochè pieno vi sfido a trovare un file system che funzioni bene. Ultima informazione riguardo invece l'NTFS.

NTFS

- Master File Table
  - A table of records (one x file)
  - Flexible 1KB storage for a file metadata and data
  - MFT itself is a file!
- Extents
  - Block pointers cover runs of blocks
  - Linux (ext4) adopts a similar approach
  - File create can provide hint as to size of file
- Journalling for reliability
  - Will not be discussed

L'NTFS è il file system che è nato con le versioni più recenti di Windows nella metà degli anni '90, progettato ex-novo: tenete presente che FFS e la struttura dell'i-node si rifà ad una struttura pensata negli anni '70, quindi c'è un po' differenza. E' stato pensato in maniera piuttosto flessibile, in particolare: nell'NTFS tutto quanto gira intorno ad un'unica tabella (che si chiama Master File Table) e che è la tabella che dice tutto del file system, contiene tutte le informazioni che sono all'interno di questa tabella (vedi figura).

MFT



Gioca un po' il ruolo della FAT oppure il ruolo della i-list, ma in realtà fa molto di più, perchè nella MFT ci stanno molte più informazioni, molti più metadati sull'intero file system. Nell'i-list di Unix ci stanno soltanto i puntatori agli i-node. La cosa notevole è che l'MFT stessa è un file (sembra una fesseria, ma in realtà questa è un'implicazione molto forte). L'MFT è una tabella che descrive per intero tutto il contenuto del file system ed è essa stessa un file. Per cui essa stessa è descritta all'interno della MFT. Per sapere dove sta l'MFT nel disco dovete leggere l'MFT. Detta così sembra strana, ma è realmente così. È come se nel vostro diario segreto aveste scritto dove lo avete nascosto e per sapere dove è nascosto dovete trovare il vostro diario segreto. (La classe guarda male il Chessa) Siccome l'MFT è un file lo potete allocare dove vi pare. Supponete di prendere un disco e di avere in questo disco danneggiato il blocco 3. In questo disco non potete installare FAT perchè nel blocco 2 e nel blocco 3 c'è la FAT e se i blocchi sono danneggiati la FAT è danneggiata e il disco è inutilizzabile. Non potete mettere il file system di tipo FFS come quello visto prima perchè nel blocco 2 e 3 il file system si aspetta di trovare l'i-list e se quei blocchi sono danneggiati l'i-list è danneggiata e non ci potete mettere il file system.

Ci potete mettere invece NTFS perchè la MFT la potete piazzare dove vi pare nel disco perchè essa stessa è un file e quindi può essere allocata dove vi pare. Siccome è allocata dove vi pare dovete però tenere traccia di quali sono i suoi blocchi e questi sono descritti nella MFT. Quindi in realtà per avere un accesso al file system dovete trovare prima l'MFT: ma come la trovate? L'indice del primo blocco dell'MFT è conservato nel Master Boot Record, per cui da lì riuscite a trovare l'MFT e iniziando a leggere il primo blocco di essa potete torvare poi le informazioni per ricostruire il resto della MFT (quindi in realtà avete lasciato un piccolo uncino per andare a riagganciarvi). L'altro elemento che introduce sono gli Extent che sono questi superblocchi e quindi permette di rappresentare blocchi di un file allocati in modo contiguo in modo molto efficiente, andando a memorizzare il primo blocco e la lunghezza della Extent. In questo modo un file di dimensioni enormi potrebbe, se contiguo, essere rappresentato soltanto con due informazioni: blocco iniziale e lunghezza. Questo è il massimo dell'efficienza alla quale possiamo aspirare. Se invece il file è molto frammentato viene ovviamente dovrò memorizzare molte più informazioni per poterlo rappresentare.

Infine utilizza delle tecniche che sono dette journaling, che sono state introdotte anche nelle ultime versioni dei file system Linux per l'affidabilità. Non entrerò nel merito di queste tecniche, vi dico soltanto quale è l'idea: ogniqualvolta si fa una modifica al file system questa modifica viene memorizzata in un file di log (il journal). Viene fatto questo perchè spesso le modifiche del file system da fare sono diverse e tutte assieme rendono il file system coerente. Se cancellate un file dovete andare ad eliminare il suo i-node e marcare i blocchi del file come libere e quindi dovete fare tante modifiche al file system, ogni volta che fate una modifica anche piccola alle directory, ai file e via discorrendo. Se mentre state facendo queste modifiche intervenisse un crash e il sistema collassasse il file system rimarrebbe inconsistente. Quando riavviate le strutture dati non daranno più informazioni coerenti. Questo era il minimo che vi potevate aspettare nei sistemi fat, era un gran macello.

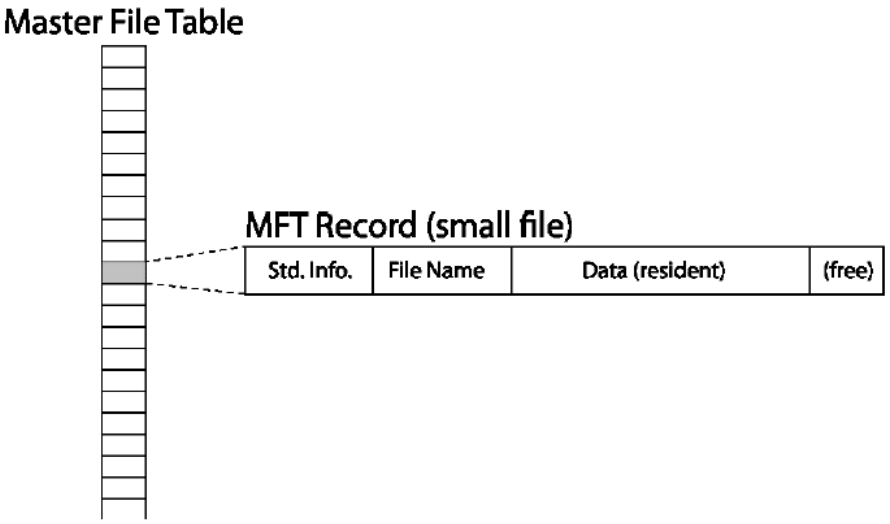
Utilizzando il journaling, invece, voi vi segante tutte le operazioni che avete fatto prima di farle e le annullate dopo che le avete compiute. In questo modo se interviene un crash nel frattempo, andando a vedere il journal potete ricostruire esattamente lo stato dell'ultima operazione e annullarla oppure completarla. Quindi anche in seguito ad un crash potete ripristinare il file system in una maniera coerente e quindi il sistema diventa più robusto. Vi invito da ora in poi a spegnere il vostro computer Windows per verificare che il journal funzioni bene (però sono fatti vostri se lo fate).

L'MFT concretamente è un file, logicamente è una tabella di record: ogni record è lungo 1 kbyte e ogni record descrive un file, quindi ogni record dell'MFT è l'equivalente di un i-node, contiene metadati del file e posizione nel disco. Vi dicevo prima che l'MFT è un file, perchè i primi due record dell'MFT descrivono la MFT stessa e se volete sapere dove essa si trova dovete leggere il blocco 0 della MFT e lì c'è scritto quali sono i blocchi che compongono tutto il resto di essa. Siccome questa informazione è critica, se per caso questo blocco viene danneggiato o viene distrutto per errore (rip file system) questa informazione è duplicata nei primi due record. Vi ho detto che ogni record occupa 1 kbyte, tipicamente la dimensione di un settore è 256, 512 byte e questo vuol dire che se anche un settore venisse danneggiato comunque la copia della MFT resta valida.

Chiaramente se vi rendete conto che un settore del record 0 dell'MFT è danneggiato la prima cosa da fare è allocare un altro blocco iniziale per l'MFT, andarci a ricopiare le informazioni e a questo punto ripristinare tutto quanto. In questo modo è tollerante ai guasti. Dopodichè i primi 16 record nell'MFT sono utilizzati per conservare metadati sul file system, per esempio: noi del file system dobbiamo sapere quali sono i blocchi liberi e quali occupati e questa informazione viene memorizzata come una bitmap, una tabella dove ogni bit mi dice se il blocco omologo è libero o occupato.

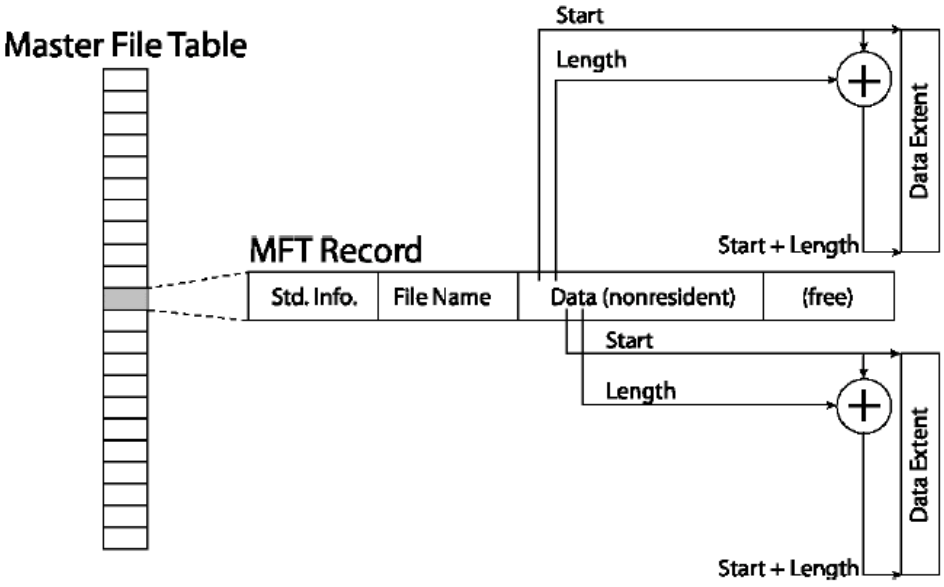
In Unix questa bitmap è allocata staticamente subito dopo l'i-list, in NTFS è conservata in un file. I blocchi nei quali questo file è allocato sono descritti nel record 6. Questo qui è il file di log utilizzato per il journaling, per esempio per ripristinare eventuali crash del sistema; il file numero 8 della MFT è il file che contiene la lista dei blocchi danneggiati del disco e via discorrendo. un certo numero di file 12, 13, 14, 15 sono riservati per utilizzi futuri (questo nel 2010, non so se allo stato attuale sono stati utilizzati oppure sono ancora liberi). I record che descrivono invece i file veri e propri, quelli degli utenti (file e directory) partono dal 16 in poi, presumibilmente il file 5 è la directory radice, quindi la directory principale di tutto il file system, quello dal quale poi parte tutta la struttura ad albero del file system; tutte le altre sottodirectory, file e via discorrendo sono descritti nei record dal 16 in poi.

NTFS Small File

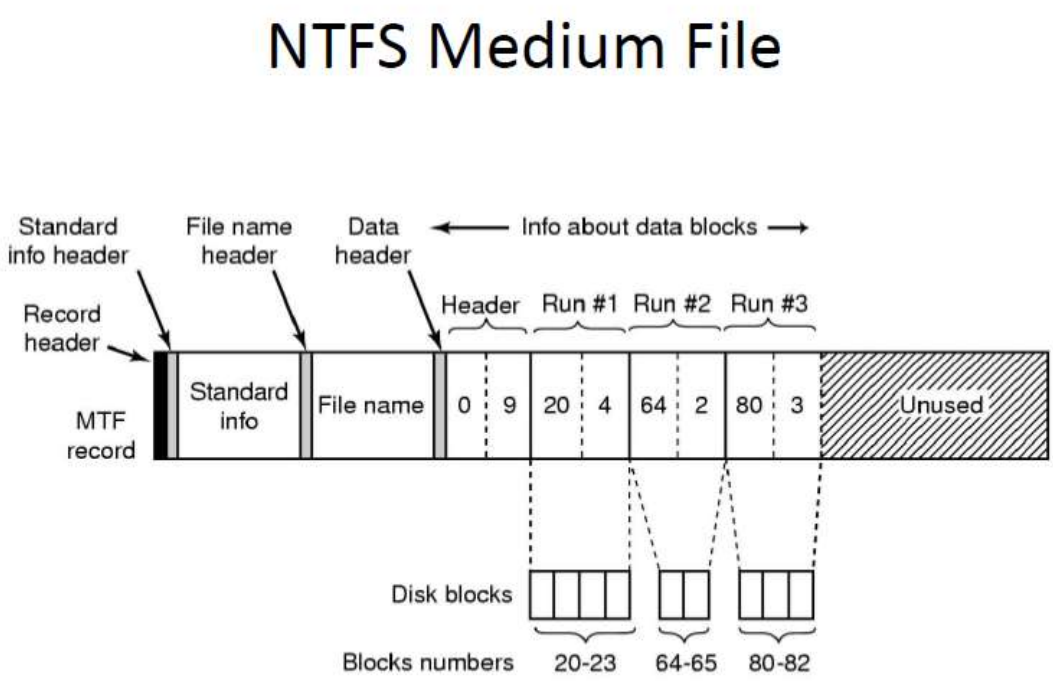


Ogni record contiene metadati e dati: in realtà contiene anche il nome del file all'interno della MFT record. Nel record MFT quindi voi trovate: il nome del file, i suoi attibuti e poi se è molto piccolo il contenuto del file, se è un po' più grande trovate i puntatori ai blocchi che compongono il file. In questo caso il file è piccolo, contiene informazioni di attibuti vari, il nome del file e i dati stessi. Se voi create un file vuoto esso non allocherà nessun blocco nel disco, ma resterà tutto quanto incluso nel suo MFT record. Fintanto che questo file ha dimensioni che stanno nell'MFT record, lui è memorizzato là dentro. Se invece il file, ad un certo punto, inizia ad avere più informazioni di quante possono essere contenute nell'MFT record allora a questo punto almeno il suo contenuto viene estratto dall'MFT record e scritto in un blocco del disco e l'MFT record ne contiene i puntatori.

NTFS Medium File



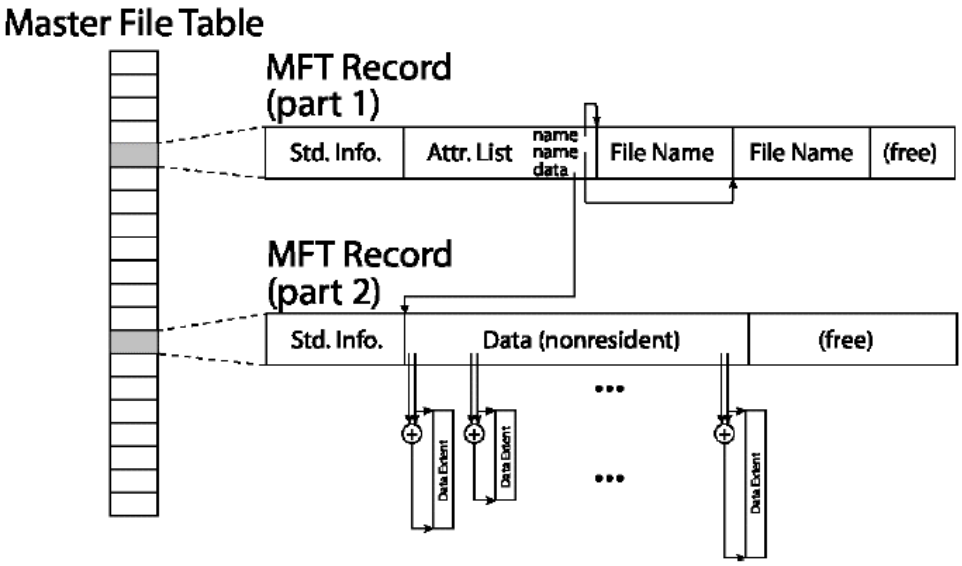
Questo è il caso di un file medio. L'MFT record continua a contenere attributi e nome del file, dopodichè i dati non sono più residenti ma sono memorizzati in sequenze di blocchi contigui, gli Extent, i quali sono descritti all'interno di questo campo, che quindi non contiene più dati ma che contiene i riferimenti ai vari Extent, alle varie sequenze di blocchi. Vi faccio un esempio un po' più concreto:



Immaginiamo un file che ha allocato 9 blocchi memorizzati in 3 Extent: il primo Extent sono i blocchi dal 20 al 23, il secondo dal 64 al 65 e il terzo dall'80 all'82. Nella parte che descrive gli Extent le informazioni sono memorizzate in questo modo: nell'intestazione c'è scritto 0-9, vuol dire che il file inizia dal blocco logico 0 e occupa 9 blocchi logici. Tali blocchi sono memorizzati in questa maniera: i primi 4 blocchi logici (da 0 a 3) sono memorizzati a partire dal blocco fisico 20, poi due blocchi logici successivi (il 4 e il 5) sono memorizzati a partire dal blocco fisico 64 ed infine gli ultimi tre (il 6, 7, 8) sono memorizzati a partire dal blocco fisico 80. Questo è un caso di un file che è diviso in 3 Extent, occupa 9 blocchi logici e occupa tutti i blocchi logici dallo 0 fino all'8.

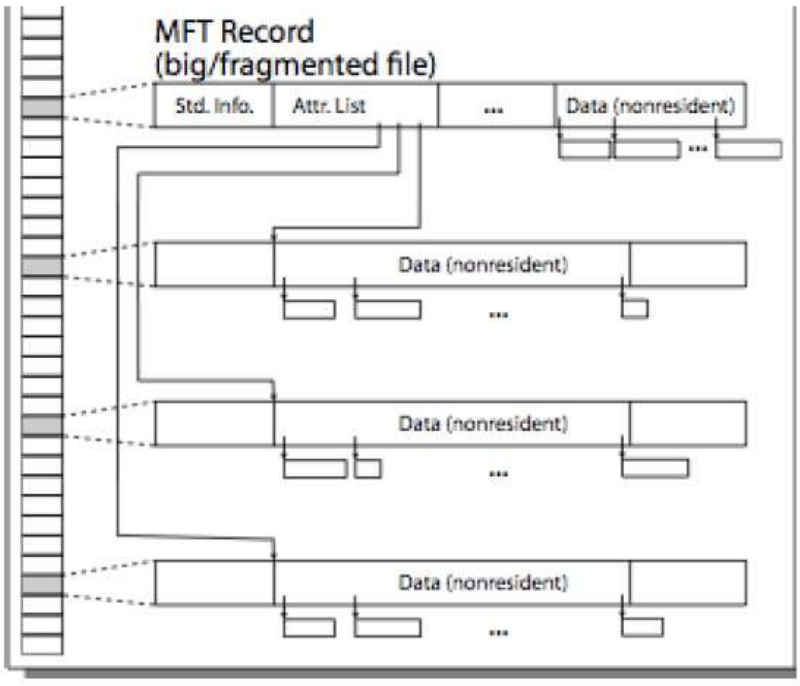
Vi voglio complicare le idee: voi in realtà avete la possibilità di creare dei file con dei buchi, cioè create un file vuoto, fate una seek su 1 milione e andare a scrivere questo file a partire dal byte 1 milione in poi. Vuol dire che tutti i byte precedenti in realtà non esistono e non sono allocati, questo con questa rappresentazione lo potete fare, perchè qui potete dire che il file inizia da un certo blocco logico, non inizia dal blocco logico 0, ma inizia dal blocco logico 80000. Vuol dire che i blocchi logici dallo 0 al 79999 non esistono, non sono mai stati allocati e non contengono niente. Il file può logicamente avere dei buchi e voi questi buchi li rappresentate replicando questa struttura, più e più volte, tutte le volte che serve. Se il file è memorizzato in modo contiguo io di queste informazioni ne devo memorizzare soltanto una: blocco iniziale e lunghezza. Se il file è molto frammentato invece di questi Run ne devo memorizzare parecchi. Che succede se il numero di Run che devo memorizzare supera la capacità dell'MFT record, se sforo un kbyte? tutte le informazioni aggiuntive le devo mettere da qualche altra parte. Per questo motivo in realtà gli MFT record possono estendersi con altri MFT record.

NTFS Indirect Block

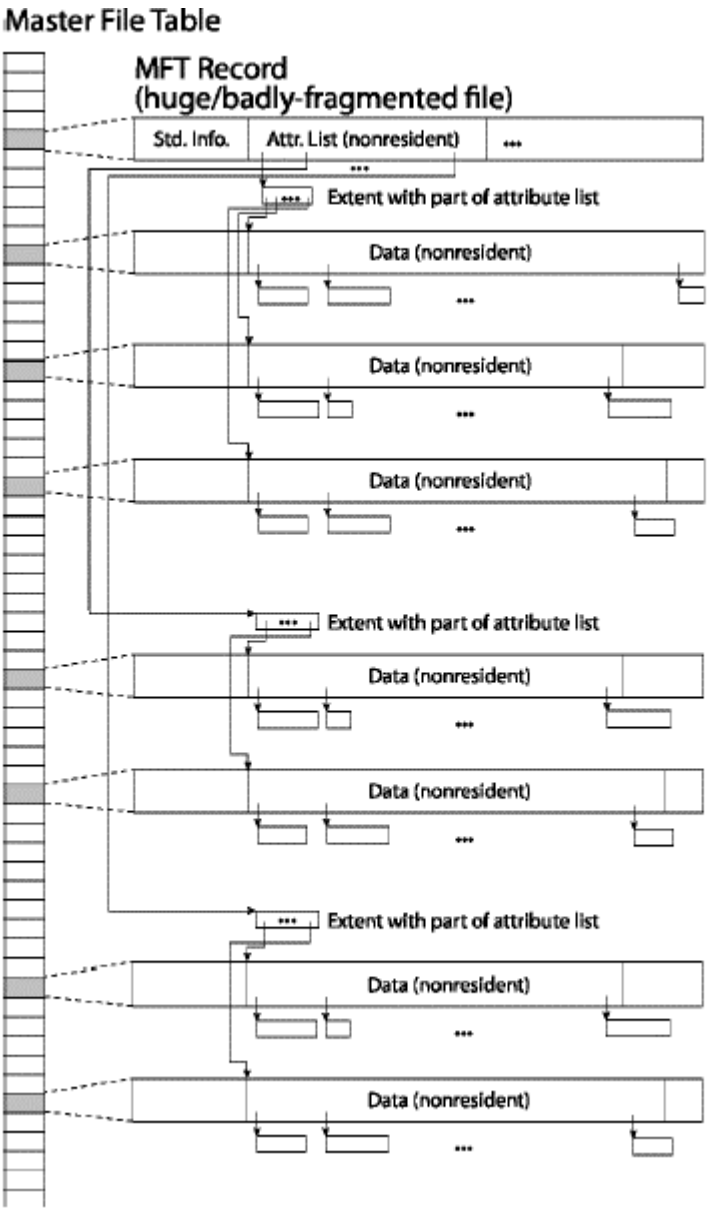


Per cui questo MFT record non è da solo sufficiente per memorizzare tutta una serie di informazioni e allora si appoggia a degli altri MFT record che in qualche maniera li estendono. in questa maniera posso andare a rappresentare file di dimensione e lunghezza arbitraria. Non entro più nel dettaglio su questo argomento perchè poi diventa una struttura davvero molto intricata, però chiaramente è giusto per dirvi che si tiene conto anche di questi casi. Per cui la dimensione del file può crescere arbitrariamente e posso avere situazioni dove in realtà di MFT record aggiuntivi ne ho parecchi.

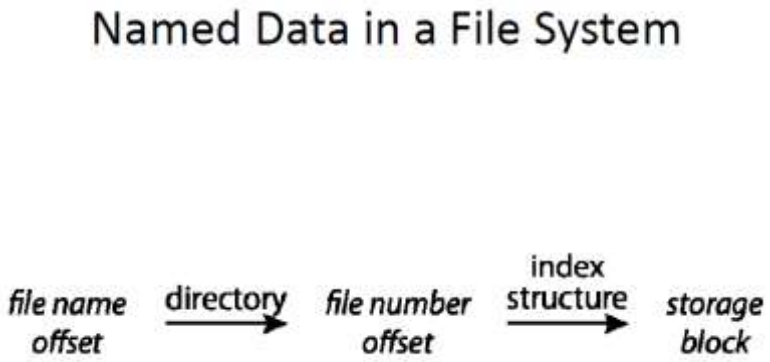
NTFS Multiple Indirect Blocks



Il file fa capo al primo MFT record, se possibile si mette tutto là dentro, se non è possibile questo MFT record contiene puntatori ad altri MFT record nei quali ci sono informazioni aggiuntive (poi la struttura si può complicare ulteriormente a dismisura).

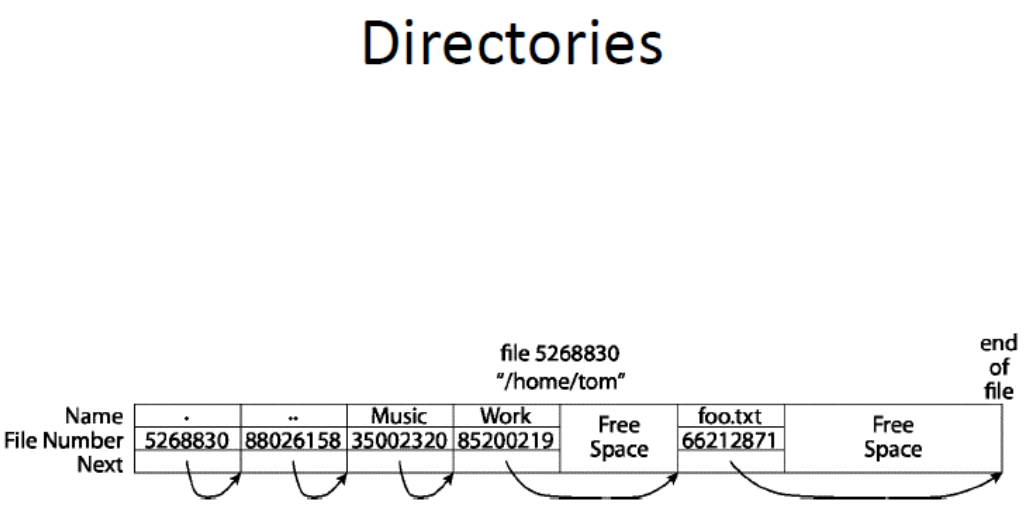


Vi do l'ultimissima informazione e poi chiudo qui.



Riguardo a quando si fa accesso ad un file: l'utente normalmente vi dà il nome di un file e una directory dove questo si trova; quando voi compilate e mandate in esecuzione un programma scrivete ./nomeFile, ma cosa state dicendo alla shell? mette in esecuzione questo file associato a questo nome che si trova nella directory attuale. In alternativa voi potreste scrivere un path intero /home o /Ste/nomeFileEseguibile e in questo modo voi dite: "metti in esecuzione questo file con questo nome che si trova nella directory associata al path". Quindi tutte le volte che il SO materialmente deve andare ad accedere ad un file conosce sempre una directory e il nome del file. Ogni accesso al file, che sia un eseguibile, un file di dati, quello che vi pare, viene fatto sempre a partire dalla directory.

Il file system che cosa fa? legge la directory di riferimento, ne scorre il contenuto, cerca il nome del file, associato al nome del file nella directory trova un puntatore (in Unix è il puntatore all'i-node) e a questo punto sposta l'attenzione del disco sull'i-node, va dal disco a leggere l'i-node, lo estrae, scorre la struttura, trova i puntatori ai blocchi ed è poi in grado, da questi, di leggere il contenuto effettivo del file. Se siamo in Windows, in NTFS non cambia molto: nuovamente specificate il nome di una directory e il nome del file, che cosa fa? Il sistema operativo dal nome della directory risale all'MFT record della directory, legge l'MFT record della directory, trova i vari puntatori agli MFT record dei vari file in essa contenuti, li legge e controlla quali di questi corrisponde al nome del file che sta cercando. Una volta che l'ha trovato legge tutto l'MFT record associato a questo file, trova i puntatori ai blocchi del file, rappresentati sotto forma di Run e accede al file. Quindi è importante sapere come sono strutturate le directory per poter fare tutto questo, perchè se non sappiamo come sono strutturate non possiamo trovare i file.



In realtà le directory sono file, quindi la directory è memorizzata nel file system come un file (come gli altri) quindi per trovare una directory dovete leggere la directory madre o padre. Una volta che avete trovato la directory e ne leggete il contenuto esso è memorizzato sotto forma di (in Unix) Nome associato a puntatore all'inode e poi un puntatore all'elemento successivo e le directory hanno normalmente questa struttura. In Unix avete chiamate di sistema apposta per manipolare le directory, per leggere le directory: in realtà esse sono file, quindi le potreste leggere al byte e per semplicità ci sono delle chiamate di libreria per poter manipolare le directory con questa struttura direttamente, per agevolare il compito. E finiamo qui. Questo è un esercizio facile facile sulla FAT.

Esercizio FileSystem 1

In un disco con blocchi di 1 Kbyte (= 2^{10} byte), è definito un file system FAT.
Gli elementi della FAT sono in corrispondenza biunivoca con i blocchi fisici del disco.
Ogni elemento ha lunghezza di 3 byte e indirizza un blocco del disco.
Ogni file è descritto da una lista concatenata di indirizzi di blocchi, realizzata sulla FAT.
Il primo blocco di ogni file è identificato dalla coppia (*nomefile, indiceblocco)* contenuto nella rispettiva directory.

1. Qual è la massima capacità del disco, espressa in blocchi e in byte?
2. Quanti byte occupa la FAT?
3. Supponendo che il file *pippo* occupi (ordinatamente) i blocchi fisici 15, 30, 16, 64 e 40, quali sono gli elementi della FAT che descrivono il file e quale è il loro contenuto?

Soluzione:

1. Capacità del disco: ..... blocchi ..... byte
2. Lunghezza della FAT: ..... byte
3. Elementi della FAT che indirizzano il file e loro contenuto:

ELEMENTO FAT	CONTENUTO

Abbiamo un disco che ha blocchi di 1 kbyte, abbiamo su questo disco un file system FAT e l'esercizio in realtà vi ricorda come è organizzata. "Gli elementi della FAT sono in corrispondenza biunivoca con i blocchi fisici del disco" (questo non dovrebbe essere necessario ricordarvelo) "Ogni elemento della FAT ha lunghezza 3 byte e indirizza un blocco del disco" Questo vuol dire che siamo con una FAT 24, perchè i puntatori sono 3 byte (3 * 8 bit = 24 bit) "ogni file è descritto da una lista concatenata di indirizzi di blocchi realizzata sulla FAT (anche questo già lo sappiamo) Il primo blocco di un file è identificato dalla coppia (nome file, indice di blocco) contenuto nella sua directory. Vogliamo sapere quant'è la massima capacità del disco espresso in blocchi e in byte, quanti byte occupa la FAT e poi supponendo che il file Pippo occupi i blocchi fisici 15, 30, 16, bla bla bla, quali sono gli elementi della FAT che descivono il file e qual è il loro contenuto.

APERTA PARENTESI: La settimana scorsa è venuta una vostra collega a ricevimento e mi ha fatto alcune domande sulla fat. Il suo approccio per risolvere gli esercizi è questo: ai compiti degli anni passati ci sono tutta questa serie di domande, quindi per questo tipo particolare di domanda sulla FAT esiste questa formula che devo applicare. I suoi appunti erano una lista di formule da applicare su una casistica molto ampia di casi. Vi dico onestamente: l'ho fatto pure io da studente. Vi dico anche però, proprio perchè l'ho fatto so cosa significa, questi esercizi in realtà molto spesso sono concettualmente estremamente semplice e se voi partite dalla comprensione della fat, di che cos'è (questo vale sia per la fat, per l'inode, per l'ntfs) non avete bisogno di ricordare a memoria tutte quelle formule. Non ho niente in contrario al fatto che uno faccia la casistica e si memorizzi le formule, fate come vi pare, però potete essere davvero molto più efficaci se anzichè ricordarvi a memoria tutte quelle formule cercaste di capire quale è la FAT e trovate la formula giusta direttamente "on the fly".
CHIUSA PARENTESI

SOLUZIONE.

Esercizio FileSystem 1

In un disco con blocchi di 1 Kbyte (= 2^{10} byte), è definito un file system FAT.
Gli elementi della FAT sono in corrispondenza biunivoca con i blocchi fisici del disco.
Ogni elemento ha lunghezza di 3 byte e indirizza un blocco del disco.
Ogni file è descritto da una lista concatenata di indirizzi di blocchi, realizzata sulla FAT.
Il primo blocco di ogni file è identificato dalla coppia (*nomefile, indiceblocco)* contenuto nella rispettiva directory.

1. Qual è la massima capacità del disco, espressa in blocchi e in byte?
2. Quanti byte occupa la FAT?
3. Supponendo che il file *pippo* occupi (ordinatamente) i blocchi fisici 15, 30, 16, 64 e 40, quali sono gli elementi della FAT che descrivono il file e quale è il loro contenuto?

Soluzione:

1. Capacità del disco: 2^{24} blocchi → 2^{34} byte
2. Lunghezza della FAT: $2^{24} * 3$ byte → 48 MByte
3. Elementi della FAT che indirizzano il file e loro contenuto:

ELEMENTO FAT	CONTENUTO
15	30
30	16
16	64
64	40
40	Ø

Qual è la massima dimensione del disco in blocchi e in byte. Stiamo parlando di massima capacità teorica: sappiamo dal testo che i puntatori ai blocchi sono fatti da 3 byte, da 24 bit perchè sappiamo che ogni elemento della FAT ha lunghezza 3 byte e indirizza un blocco del disco, quindi i puntatori ai blocchi del disco sono fatti da 3 byte. Quant'è il numero massimo di puntatori che possiamo generare con 3 byte? 2^{24}. Questo vuol dire che il disco non può avere più di 2^{24} blocchi perchè se ce ne fosse uno in più non avrei un indirizzo per rappresentarlo. Quindi gli indirizzi sono fatti da 24 bit, di conseguenza ho 2^{24} indirizzi differenti e quindi il numero massimo di blocchi del disco è 2^{24}. Questa è anche la massima capacità teorica del disco. Quanto è grande un blocco? C'è scritto nel testo, 1 Kbyte. Quindi se nel disco posso avere 2^{24} blocchi di 1 Kbyte ciascuno la sua capienza è 2^{24} per 1 Kbyte.

$2^{10}\ byte \times 2^{24} = 2^{34}\ byte = 16\ Gbyte$ (è la massima capacità di questo disco).

Quanto è lunga la FAT? Questi calcoli sulla FAT non sono altro che l'area del rettangolo, la FAT è una matrice che ha una certa lunghezza (il numero di elementi) e la larghezza (la dimensione del singolo elemento). quindi la dimensione della FAT non è altro che l'area del rettangolo. Il numero di elementi è 2^{24}, perchè ogni elemento della FAT è in corrispondenza biunivoca con un blocco, la FAT deve poter indicizzare tutti i blocchi del disco, la FAT deve avere tanti elementi quanti sono i blocchi del disco. Al massimo sono 2^24, ogni elemento sono 3byte, quindi la FAT occupa fisicamente

$3\ byte \times 2^{24} = 48\ Mbyte$

Per quanto riguarda il contenuto della FAT, a meno che qualcuno non voglia mettersi a scrivere tutti gli elementi 2^24 potete invece rappresentare un estratto della FAT per gli elementi che ci interessano per rappresentare questo file. Questo quindi in realtà non è la FAT, ma un estratto della FAT per rappresentare gli elementi che mappano quel file. Il file occupa i blocchi fisici 15, 30, bla bla bla. Sono nell'ordine giusto, quindi 15 è il primo blocco del file, il blocco logico numero 0 del file. Noi sappiamo che nel file system FAT il blocco logico 0 del file è puntato dalla directory, quindi se noi andiamo nella directory che contiene il file troviamo l'associazione tra nome del file e puntatore al primo blocco, quindi il valore 15 sta nella directory. Sappiamo anche che per trovare il puntatore al secondo blocco del file, quindi il valore 30, lo troviamo nella FAT nella posizione corrispondente al blocco precedente. Detto in altri termini se io vado nella FAT e vado a

leggere l'elemento numero 15 ci trovo il puntatore al blocco successivo, che è il 30. A questo punto posso andare a leggere il blocco logico numero 1 dal file andando a leggere il blocco 30, poi posso andare a leggere l'elemento numero 30 della FAT e lì troverò il puntatore al blocco successivo che è il 16. Quindi in realtà costruire i valori contenuti nella FAT è semplice, basta scalare la lista linkata. In 15 trovo 30, in 30 trovo 16, in 16 trovo 64, in 64 trovo 40 e siccome il file termina col blocco 40 il contenuto della FAT nell'elemento 40 non può rappresentare un puntatore valido perchè non c'è un puntatore ad un blocco successivo, quindi questo c'è un codice che mi rappresenta la fine del file. Può essere -1, 0, qualcosa che faccia capire che il file è finito. Volete vedere qualche altro esercizio in particolare?

Andrea si sfrega le mani:

"*L'esercizio M10 per favore*".

ESERCIZIO M-10 Segmentazione

In un sistema che gestisce la memoria con segmentazione a domanda, l'indirizzo logico è di 28 bit, dei quali i primi 6 bit codificano l'indice di segmento e i restanti 22 bit definiscono l'offset. Ogni elemento della tabella dei segmenti è formato da 7 byte, dei quali 1 byte contiene indicatori utili al gestore della memoria, 3 byte contengono il valore base, e i restanti 3 byte contengono il valore limite.
Si chiede:

1. Il massimo numero di segmenti di un processo;
2. La massima dimensione di un segmento (in byte);
3. La massima dimensione della tabella dei segmenti (in byte);
4. Il massimo valore possibile per l'indirizzo di origine di un segmento;

SOLUZIONE

1. Massimo numero di segmenti di un processo: ..... segmenti
2. Massima dimensione di un segmento (in byte): ..... byte
3. Massima dimensione della tabella dei segmenti (in byte): ..... byte
4. Massimo valore possibile per l'indirizzo di origine di un segmento: .....

Siamo in una situazione dove abbiamo un sistema che gestisce la memoria con segmentazione a domanda. Questo vuol dire che la memoria dei processi è segmentata e i segmenti non stanno tutti necessariamente in memoria, ma li posso caricare quando vengono riferiti. Quindi come si fa con la paginazione a domanda, lo stesso lavoro lo si può fare con la segmentazione a domanda. Gli indirizzi sono fatti da 28 bit, di cui i primi 6 codificano l'indice di segmento e i restanti 22 bit definiscono l'offset. Ogni elemento della tabella dei segmenti è formato da 7 byte dei quali 1 byte contiene gli indicatori utili per il gestore della memoria, quindi sono i flag segmento-riferito, segmento-in-uso, segmento-modificato, tutto quello che serve per gestire il caricamento dinamico dei segmenti. 3 byte contengono il valore di base, il valore base sarebbe l'indirizzo iniziale dell'area di memoria che contiene il segmento e i restanti 3 byte contengono il valore limite, che sarebbe la lunghezza del segmento. Vogliamo sapere il massimo numero di segmenti per ogni processo, la massima dimensione di un segmento in byte, la massima dimensione della tabella dei segmenti in byte e il massimo valore possibile per l'indirizzo di origine di un segmento.

Andrea: "*Domanda: quindi se io sommo base + offset, diciamo che al massimo posso ottenere, ad esempio, più di 2^{22}?*"

Andrea guarda Fabio chiedendo approvazione...

Fabio: "*Ma che stai facendo? Non so cosa stai facendo..*"

Andrea: "*Comunque l'offset ci dice al massimo quante posizioni ho in quel segmento. La seconda domanda ci chiede la massima dimensione di un segmento..*"

Fabio: "*..in byte. Quindi noi abbiamo 2^{22} cosa?*"

Chessa: Sono 22 bit.

Fabio: "*Possiamo avere 2^{22} nel segmento, ma ogni singola cella..*"

Chessa: ma scusi, l'offset a che serve? (Ecco che parte l'orale)

Fabio: "*per scorrere il segmento..*"

Chessa: non mi soddisfa come risposta. A che vi serve l'offset materialmente? Pensate a cosa succede: voi generate un indirizzo e generato quell'indirizzo volete leggere un byte che sta in memoria a quell'indirizzo. Quindi ad ogni indirizzo corrisponde un byte.

Fabio: "*Quindi noi vogliamo leggere un byte.*"

Chessa: ad ogni indirizzo corrisponde un byte, dal punto di vista del processore nel nostro corso. Poi ci sono processori che ad ogni indirizzo associa 4 byte, ma a noi non interessa. Ogni indirizzo vogliamo leggere un byte. Ogni indirizzo è composto da indice di segmento e offset: l'indice di segmento lo sostituisco con l'indirizzo iniziale di segmento. Ci appiccico l'offset e ottengo un indirizzo per la memoria che mi restituisce un byte. Quindi una volta che ho fissato la base con l'offset posso generare tutti gli indirizzi, tutti i byte che compongono il segmento. Quindi il segmento può avere al massimo 2^{22} byte.

Fabio: "*oooh.*"

Andrea: "*Quindi all'offset appiccico l'indice ma non la base? o è la stessa cosa?*"

Chessa: No. Sommate l'offset alla base: ma questo vuol dire che fissata la base (che è fissa) tutti gli indirizzi del segmento sono dati da tutti i possibili valori ammissibile per l'offset, al massimo quanti sono questi indirizzi?

Andrea: "*2^22. Invece nella paginazione è un po' diverso. "*

Chessa: No. Nella paginazione è sempre lo stesso: ogni offset indicizza una pagina e quindi la pagina ha tanti byte quanti li potete rappresentare con l'offset.

Andrea: "*..e anche lì si prende un byte alla volta.*"

Chessa: La nostra ipotesi è: quale che sia il sistema di gestione della memoria, il processore genera un indirizzo e legge o scrive un byte. Per farla semplice: avrei potuto dire un indirizzo ha 4 byte e sarebbe stata la stessa cosa.

Andrea: "*OK.*"

(Cinque minuti censurati perché si sparla del Chessa)

Soluzione.

ESERCIZIO M-10 Segmentazione

In un sistema che gestisce la memoria con segmentazione a domanda, l'indirizzo logico è di 28 bit, dei quali i primi 6 bit codificano l'indice di segmento e i restanti 22 bit definiscono l'offset. Ogni elemento della tabella dei segmenti è formato da 7 byte, dei quali 1 byte contiene indicatori utili al gestore della memoria, 3 byte contengono il valore base, e i restanti 3 byte contengono il valore limite.
Si chiede:

1. Il massimo numero di segmenti di un processo;
2. La massima dimensione di un segmento (in byte);
3. La massima dimensione della tabella dei segmenti (in byte);
4. Il massimo valore possibile per l'indirizzo di origine di un segmento;

SOLUZIONE

1. Massimo numero di segmenti di un processo: 2^6 segmenti = 64 segmenti
2. Massima dimensione di un segmento (in byte): 2^{22} byte = 4 Mbyte
3. Massima dimensione della tabella dei segmenti (in byte): 64*7= 448 byte
4. Massimo valore possibile per l'indirizzo di origine di un segmento: 2^{24}-1

Abbiamo 2^6 segmenti perchè abbiamo 6 bit per codificare l'indice del segmento. Con 6 bit possiamo rappresentare 2^6 differenti configurazioni che vuol dire 2^6 differenti segmenti al massimo. Massima dimensione di un segmento in byte, di questo ne ho parlato con tanti di voi. Mi sono reso conto che c'è una questione, un'ambiguità di fondo con qualcuno di voi nell'interpretazione di queste cose. Quello che noi abbiamo sempre assunto è che il processore genera un indirizzo alla memoria e da questo indirizzo ottiene 1 byte, quindi ad ogni indirizzo corrisponde 1 byte in memoria, potrebbero non essere così, potrebbe essere che ad un indirizzo corrispondono 2, 4 byte, in questo corso facciamo l'ipotesi che ad ogni indirizzo corrisponda 1 byte a meno che non sia stato scritto diversamente da qualche parte.

Guardiamo questo segmento: l'indirizzo che riferisce al segmento si forma sommando l'offset alla base del segmento. La base del segmento è fissa, l'unica cosa che varia è l'offset. Quindi i vari byte di questo segmento li indirizzate facendo variare l'offset, non la base, la base del segmento è quella, è fissa. L'offset del segmento è formato da 22 bit, quindi posso rappresentare al massimo 2^{22} indirizzi; di conseguenza il numero massimo di indirizzi ammissibili per l'offset è 2^{22}, la massima dimensione del segmento è 2^{22} byte.

Badate bene che però c'è un piccolo inghippo: la dimensione del segmento non è limitata soltanto dall'offset, ma anche dal registro limite. Se il registro limite fosse di 10 bit vuol dire che qualsiasi indirizzo che eccede 2^{10} viene segnalato come violazione di protezione. Quindi in realtà la massima dimensione del segmento è limitata dal massimo indirizzo rappresentabile come limite e dal massimo indirizzo rappresentabile come offset, il minimo tra questi due massimi. Siccome il registro limite è formato da 3 byte, quindi 24 bit, quindi al massimo posso avere segmenti di 2^{24} per quanto riguarda il limite.

Per quanto concerne l'offset al massimo posso avere segmenti di 2^{22} byte, il minore tra i due è 22 e quindi la massima dimensione del segmento è 2^{22} byte. La massima dimensione dell'offset è limitata da 2 fattori: il registro limite e il massimo numero di indirizzi che possiamo generare con l'offset. Perchè l'MMU dice questo: se il valore dell'offset è maggiore del registro limite allora eccezione. Quindi il massimo valore dell'offset se supera il valore del registro limite viene troncato e dà errore. Il registro limite è fatto da 3 byte. Quindi al massimo ho 2^{24} byte nel segmento, la dimensione massima. D'altra parte l'offset 22 bit, quindi al massimo il segmento può essere lungo 2^{22} byte. Il minore dei 2 è 2^{22} e quindi il segmento più di 2^{22} non va.

Massima dimensione della tabella dei segmenti in byte. Sappiamo dal punto 1 che la tabella dei segmenti contiene 2^6 elementi, al massimo, ogni elemento occupa 7 byte (perchè lo dice il testo dell'esercizio) quindi la massima dimensione della tabella dei segmenti è 7 byte * 2^6. Riguardo al quarto punto: se il segmento ha

lunghezza M byte la massima base è pari alla massima dimensione della memoria virtuale meno M. Supponiamo di avere una memoria virtuale di 1 kbyte. Supponiamo di avere un segmento che occupa 512 byte, se io dico che la base è 900 dovrei andare a riferire alla memoria oltre il suo limite massimo e questo non sarebbe possibile. Quindi il valore massimo della base per poter allocare quel segmento è dato dalla massima dimensione della memoria virtuale, quindi massimo indirizzo rappresentabile meno lunghezza effettiva del segmento. Siccome la lunghezza minima del segmento ipotizzo che sia 1 (effettivamente potrebbe essere anche 0) questo vuol dire che il massimo valore della base è 2^{24} -1.

ESERCIZIO M-17 Tabella delle pagine

Un sistema gestisce la memoria con paginazione e usa indirizzi logici di 32 bit, blocchi di memoria di 1 kbyte e tabelle delle pagine a due livelli. La tabella delle pagine di 1° livello (o *directory)* e quelle di 2° livello hanno uguale dimensione. Ogni elemento della tabella di primo livello e di quelle di secondo livello contiene l'indice di un blocco di memoria e 2 bit di controllo.
La memoria fisica ha una capacità di 4 Gbyte (= 2^{32} byte)
Domande:

1. Nell'indirizzo logico, quanti bit sono riservati all'offset nella tabella di 1° livello, quanti all'offset nella tabella di 2° livello e quanti all'offset nella pagina logica?
2. Quanti bit sono necessari per codificare un indice di blocco e quanti bit contiene ogni elemento dei tabella di primo o di secondo livello?
3. Quanti byte e quanti blocchi di memoria occupa ogni tabella?

SOLUZIONE

Il numero di blocchi fisici è .....

1. Offset in Tab 1° livello (numero di bit): .....
   Offset in Tab 2° livello (numero di bit): .....
   Offset in pagina logica (numero di bit): .....
   Bit necessari per codificare un indice di blocco: .....
   Lunghezza in bit di ogni elemento di tabella: .....
2. Numero di byte occupati da ogni tabella:
   Numero di blocchi occupati da ogni tabella: .....

Abbiamo gestione della memoria con paginazione. Indirizzi logici a 32 bit, blocchi di memoria di 1 k, di conseguenza la dimensione delle pagine nello spazio virtuale è di 1 k. I blocchi e le pagine hanno tutte la stessa dimensione. Tabelle delle pagine a due livelli. La tabella delle pagine di primo livello che spesso viene detta anche directory e quella di secondo livello hanno la stessa dimensione. Ogni elemento della tabella di primo e di secondo livello contiene le stesse informazioni, in particolare: l'indice di un blocco di memoria e 2 bit di controllo. Poi sappiamo che la memoria fisica ha la capacità di 4 Gbyte (2^{32} byte) vogliamo sapere nell'indirizzo logico quanti bit sono riservati all'offset nella tabella di primo livello e quanti all'offset nella tabella di secondo livello e quanti all'offset nella pagina logica. Poi quanti bit sono necessari per codificare un indice di blocco e quanti bit contiene ogni elemento della tabella di primo e di secondo livello e poi quanti byte e quanti blocchi di memoria occupa ogni tabella. Sono in cascata, se non fate il primo poi non riuscite a fare gli altri. Quindi concentratevi prima di tutto nel capire questa paginazione a due livelli come prevede che sia fatta la divisione dell'indirizzo.

SOLUZIONE.

ESERCIZIO M- 17 Tabella delle pagine

Un sistema gestisce la memoria con paginazione e usa indirizzi logici di 32 bit, blocchi di memoria di 1 kbyte e tabelle delle pagine a due livelli. La tabella delle pagine di 1° livello (o *directory)* e quelle di 2° livello hanno uguale dimensione. Ogni elemento della tabella di primo livello e di quelle di secondo livello contiene l'indice di un blocco di memoria e 2 bit di controllo.
La memoria fisica ha una capacità di 4 Gbyte (= 2^{32} byte)
Domande:

1. Nell'indirizzo logico, quanti bit sono riservati all'offset nella tabella di 1° livello, quanti all'offset nella tabella di 2° livello e quanti all'offset nella pagina logica?
2. Quanti bit sono necessari per codificare un indice di blocco e quanti bit contiene ogni elemento dei tabella di primo o di secondo livello?
3. Quanti byte e quanti blocchi di memoria occupa ogni tabella?

SOLUZIONE

Il numero di blocchi fisici è 2^{22}.

1. Offset in Tab 1° livello (numero di bit): 11 bit

   Offset in Tab 2° livello (numero di bit): 11 bit

   Offset in pagina logica (numero di bit): 10 bit
2. Bit necessari per codificare un indice di blocco: 22 bit

   Lunghezza in bit di ogni elemento di tabella: 24 bit (22 bit per l'indice di blocco + 2 bit di controllo)
3. Numero di byte occupati da ogni tabella:

   Numero di blocchi occupati da ogni tabella: 2^{11} elementi di 3 byte ➔ 6 * 2^{10} byte ➔ 6 blocchi

Vi ricordo che nella paginazione a due livello l'indirizzo è diviso in 3 campi: 1 indicizza la tabella delle pagine di primo livello, il secondo indicizza la tabella delle pagine di secondo livello e il terzo è l'offset. La memoria fisica è di 4 Gbyte (2^{32} byte). Ogni blocco di 10 kbyte quindi il numero di blocchi è

$$\frac{2^{32}\ byte}{2^{10}\ byte} = \text{2^22 } blocchi \text{ (circa 4 milioni).}$$

Poi offset di una tabella di primo, di secondo livello è l'offset della pagina logica. Per trovarli dobbiamo capire come si scompone l'indirizzo virtuale: l'indirizzo virtuale è di 32 bit, dei quali: 10 bit sappiamo che sono l'offset della pagina, mentre sappiamo anche che la tabella di primo livello e la tabella di secondo livello hanno la stessa dimensione hanno le stesse informazioni nei singoli descrittori, quindi per forza di cose devono avere anche lo stesso numero di elementi. Quindi se io da 32 bit tolgo i 10 bit di offset me ne restano 22, la metà sono per la tabella di primo livello e l'altra metà sono per quella di secondo. Il risultato è che sono 11, 11, 10. Poi i bit necessari per codificare un indice di blocco: i blocchi abbiamo visto che sono 2^{22} (l'abbiamo visto anche prima). Quindi per codificare 2^{22} indirizzi differenti mi servono 22 bit.

Lunghezza in bit di ogni elemento di tabella, di primo o di secondo livello. Ogni elemento della tabella di primo o di secondo livello contiene un indice di blocco, quindi 2^{22} bit più 2 bit di controllo, quindi 24. Numero di byte occupati da ogni tabella: ogni tabella contiene 2^{11} elementi perchè l'indice di primo livello è formato da 11 bit, quindi l'offset nella tabella di primo livello è di 11 bit, quindi essa può contenere 2^{11} elementi. Stesso discorso per la tabella di secondo livello. Ogni tabella deve avere 2^{11} elementi ognuno di 3 byte. Questo fa 6 Kbyte, ogni blocco può contenere 1 Kbyte e quindi ho 6 blocchi.

Nel microfono con tono minaccioso: "*A NICCOLO' E' ANDATA MOLTO BENE*.."

L'indirizzo è formato da 32 bit: l'offset è quella parte dell'indirizzo che mi serve per indicizzare i byte della pagina. L'offset significa appunto scostamento, esso mi dice che a partire dall'inizio della pagina quale è il byte che voglio andare a leggere. Quindi il termine offset ha un'accezione più ampia, non è soltanto da

riferire alla posizione all'interno di una pagina. Ogniqualvolta voi avete uno scostamento a partire da una posizione iniziale quello è un offset. Per esempio nella mia bicicletta ho un certo rapporto, se scalo di marcia ho un offset di 1, 2, 3, 4, a seconda della marcia in cui scalo. In questo esercizio il termine offset è usato questa maniera quando è riferito alla tabella di primo e secondo livello. Quando io vi dico che l'offset in tabella di primo livello è formato da 11 bit vi sto dicendo che i primi 11 bit servono per indicizzare la tabella di primo livello che vuol dire che a partire dal primo elemento della tabella di primo livello vado a prendere un elemento scostandomi di una quantità pari al contenuto di quegli 11 bit, quindi quelli sono un offset in realtà. Anche se l'offset lo abbiamo sempre riservato riferendoci alla posizione all'interno di un segmento o di una pagina in realtà parlando in maniera un po' più ampia il termine offset lo posso usare per riferirmi alla posizione all'interno di un array o all'interno di una tabella. Detto questo, 10 bit sono per l'offset per determinare lo scostamento all'interno della pagina, i restanti 22 bit li devo ripartire equamente tra l'indice della tabella di primo livello e quello della tabella di secondo. Questi due indici sono di fatto 2 offset.

Domanda: "*I bit restanti li ripartiamo equamente perchè ce lo dice il testo?*"

Certo, solo perchè lo dice questo testo in questo esercizio. È per questo che vi dicevo che la vostra collega venuta a ricevimento aveva tantissime formule: perchè qualcuno ha preso la briga di codificare tutti gli esercizi e se qui avessi scritto 3/4 avrebbe trovato in quegli appunti la formula da applicare nel caso in cui ci fosse scritto 3/4. Ovviamente questo si può fare, però non è utile in realtà. Come si fa a ripartire se il testo non dice nulla? Questo riguarda il mestiere del professore che deve dare i compiti, perchè io ho questo problema in realtà con voi e in generale. Preparo 6 compiti all'anno: pensare a 6 compiti differenti all'anno è difficilissimo. Un mio collega spagnolo ne fa 1 compito all'anno, quindi per me il tempo scorre 6 volte più veloce, 1 anno per me equivale a 6 anni per lui. Io insegno SOL da 17 anni, 17 * 6 senza contare i compitini, fanno un centinaio di compiti che ho preparato. Fare tutti sti compiti all'anno vuol dire che dopo un po' passa la fantasia e uno deve riutilizzare gli schemi degli anni precedenti, ma io so bene che se utilizzo troppo gli schemi degli esercizi precedenti voi, giustamente, vi addestrate a risolvere gli esercizi e non studiate Sistemi Operativi. È questo il motivo per il quale la vostra collega e alcuni di voi si sono preparati una serie di soluzioni e schemi per esercizi tipici: vi capisco, lo facevo pure io. D'altra parte se il professore non ha fantasia si sfrutta. Per limitare questo fenomeno, che non mi permette di valutarvi in maniera oggettiva, occasionalmente si introducono delle variazioni nei compiti, proprio per vedere che uscendo dagli schemi negli eserciziari ve la cavate spesso. Questo è il motivo per il quale ogni tanto non c'è scritta questa cosa qui, ma ce n'è scritta un'altra.

"*Ma quindi qualcosa ce deve stà scritto?*"

Eh certo, se no in base a cosa lo calcolate, però io ogni volta mi devo inventare qualcosa di diverso in maniera tale che voi non lo possiate mappare su un esercizio già predisposto.