UNIVERSITÀ DI PISA

Appunti del corso di Architetture degli Elaboratori e Sistemi Operativi

A.A. 2022/2023

Queste note riguardano il corso di Architettura degli elaboratori e sistemi operativi dell'anno accademico 2022/2023, non c'è distinzione tra il corso A e il corso B in quanto gli argomenti trattati sono gli stessi. I docenti che hanno tenuto il corso sono il Professor Marco Danelutto e Massimo Torquati, in particolare la parte di Architetture degli elaboratori è stata scritta tenendo conto delle lezioni del Professor Danelutto mentre la parte riguardante i Sistemi Operativi si basa sulle lezioni del Professor Massimo Torquati. Questi sono appunti che sono stati rimessi in bella e riletti accuratamente da più studenti, ciò non toglie che possano contenere imperfezioni, inesattezze o errori che potrebbero fuorviare totalmente il lettore — inutile ripetere che queste note sono state scritte per uso personale da parte dello studente.

Ogni commento, correzione o segnalazione di errori è benvenuto.

(Telegram: @Samuele136)

In quanto autore ci tengo a precisare che queste note NON si intendono sostituive del corso e che per avere una comprensione adeguata degli argomenti la frequenza alle lezioni è fondamentale.

Detto questo spero che possano essere utili e che possano agevolare la preparazione dell'esame.

Buono studio,

Samuele Punzo

Il materiale di testo di riferimento per lo studio di questo corso sono i due libri:

- David Money Harris, Sarah L. Harris Sistemi digitali e architettura dei calcolatori Progettare con tecnologia ARM, Zanichelli 2017
- Thomas Anderson, Michael Dahlin, Operating systems Principles & Practice, 2a edizione, Recursive book, 2014
- Slides fornite dai docenti

Da questi due libri e dalle slides sono prese anche delle immagini e delle parti di testo usate per la creazione di questi appunti.

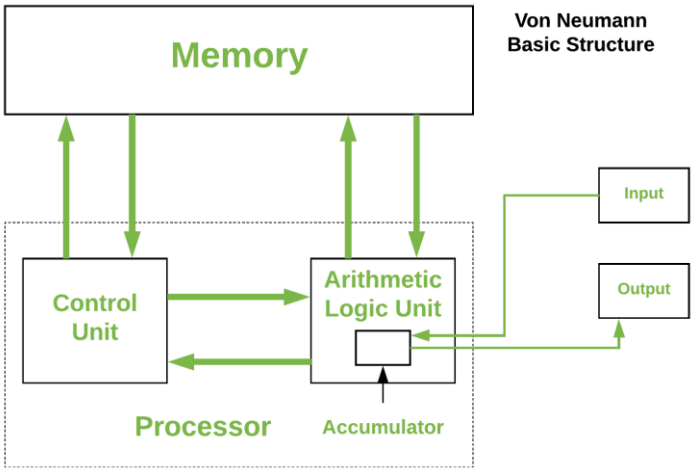
Indice

Architetture degli Elaboratori

Introduzione

Il processore è l'entità che permette di fare i calcoli, abbiamo una componente hardware, la CPU, gestita da una controparte software il sistema operativo.

Attualmente tutti i processori lavorano con il modello Von Neumann, questo modello è composto da 2 parti principali:



- Memoria
- Processore

La maggior parte del lavoro del processore è la gestione del traffico da e verso la memoria.

All'interno del processore come vedremo ci saranno ben altre unità quali la mmu, i registri di stato, il pc ecct…

In particolare il pc (program counter) è un registro molto importante in quanto tiene traccia dell'istruzione corrente.

Von Neumann Bottelneck:

È un rallentamento del traffico a causa dell'eccessivo numero di informazioni.

Ciclo del processore

Il ciclo del processore è il ciclo fetch execute, possiamo immaginarcelo come un while(true) {} al cui interno vengono eseguite delle particolari operazioni.

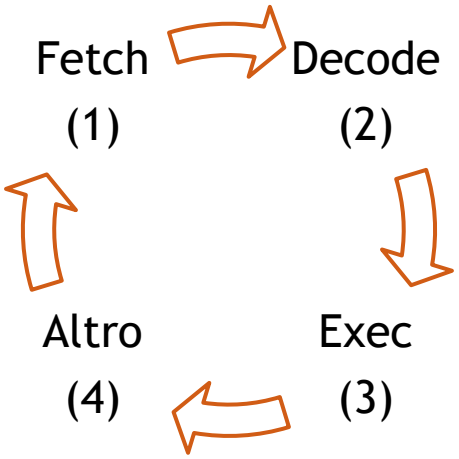
Queste operazioni sono:

1) Fetch: Si prelevano le istruzioni in memoria all'indirizzo indicato dal program counter.

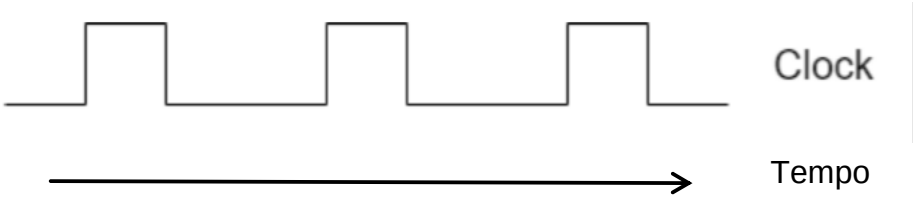
2) Decode: Decodifichiamo le istruzioni

3) Exec: Eseguiamo le istruzioni

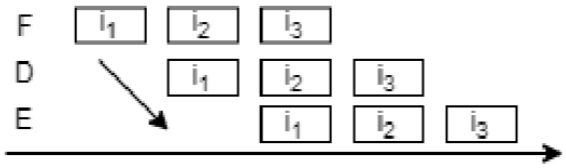
4) Altro: Aggiorniamo il program counter, ci potrebbero anche essere delle eventuali interruzioni (eventi asincroni).



1 clock del processore equivale ad un ciclo del while, ovvero ad un ciclo dello schema qua sopra.



Per ottimizzare i tempi nascono i processori modello pipeline che funzionano come una catena di montaggio (figura sotto).



Legge di Moore

Moore affermò che con l'avanzare del tempo e delle tecnologie saremmo arrivati a raddoppiare le prestazioni dei processori ogni anno e mezzo.

Questo dagli anni 2000 si è rivelato non essere vero, infatti diminuendo le dimensioni dei chip e di conseguenza velocizzando i cicli di clock aumentiamo una resistenza che sviluppa calore.

È stato dimostrato che prendendo un chip di una certa superficie il rapporto superficie / potenza di calcolo è diverso da quello superficie / calore.

Per ovviare al problema è stato creato il multi-core, ovvero l'accoppiamento parallelo di più processori, avendo un area più grande il calore generato è più gestibile.

È da osservare però che con le nuove generazioni ci sta che il numero di core aumenti ma la potenza del singolo core diminuisca rispetto a processori precedenti.

Ma allora come mai ci interessa aumentare il numero di core?

Ci interessa aumentare il numero di core perché in questo modo possiamo spezzettare i nostri programmi ed eseguire ciascun pezzo su un core diverso, questo porta comunque ad una diminuzione della velocità di esecuzione.

Acceleratori

Dobbiamo distinguere 3 tipi di unità di calcolo:

1) Cpu: Central Process Unit
2) Gpu: Graphic Process Unit
3) Fpga: Field Programming Gate Array

La GPU permette di accelerare la creazione di immagini in un frame buffer, destinato all'output su un dispositivo di visualizzazione, ha una sua memoria dedicata in cui gli scambi di lettura e scrittura sono molto veloci.

La FPGA è un dispositivo programmabile, formato da un circuito integrato le cui funzionalità logiche di elaborazione sono appositamente programmabili.

Circuiti elettronici

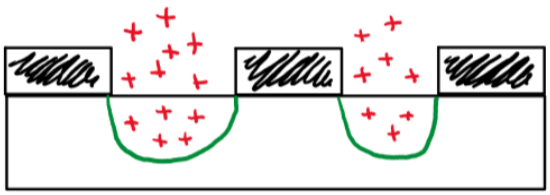
I computer classici rappresentano l'informazione come variabili a valori discreti, in particolare 0 e 1. Ciononostante, le variabili sono rappresentate da quantità fisiche continue come ad esempio la tensione elettrica di un filo.

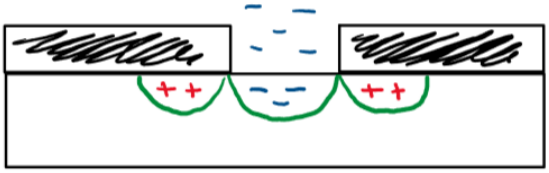
Per rappresentare questa informazione vengono utilizzati dei circuiti elettrici, questi circuiti vengono creati grazie a dei drogaggi su delle piastre di silicio chiamate wafer.

I wafer di silicio sono molto utilizzati grazie alle loro proprietà elettroniche, sono infatti dei semiconduttori.

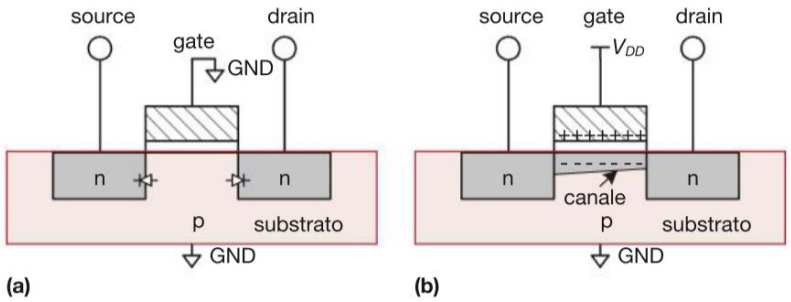
Drogaggio

Il drogaggio è l'aggiunta al semiconduttore puro di piccole percentuali di atomi non facenti parte del semiconduttore stesso allo scopo di modificare le proprietà elettroniche del materiale.

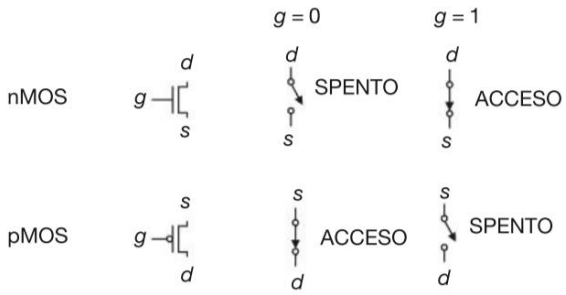




Transistor



Un transistor è composto da un materiale semiconduttore al quale sono applicati 3 terminali che lo collegano al circuito esterno,
il funzionamento è basato sulla giunzione p-n.



Grazie ai transistor possiamo calcolare forme elementari dell'algebra booleana

Ci sono due tipi di mosfet (transistor) gli nMos e i pMos a seconda del tipo di cariche contenute nelle regioni drogate.

Un mosfet si comporta come un interruttore, nel quale la tensione al gate (pin centrale) crea un campo elettrico che apre o chiude il collegamento tra source e drain.

Inizialmente i diodi[1] tra source e drain e il substrato sono polarizzati inversamente, dunque non c'è un percorso che permetta alla corrente di fluire.

Quando viene applicata una tensione dello stesso tipo del substrato, si crea un campo elettrico che attrae le cariche dello stesso tipo della tensione verso la piastra superiore, e le cariche di carica opposta verso quella inferiore.

Se la tensione è sufficiente viene attirata così tanta carica "opposta" verso la piastra inferiore, che il tipo della regione si inverte, creando una regione di inversione chiamata canale (figura (b) sopra).

Adesso il transistore ha un percorso elettrico tutto dello stesso tipo, partendo da source passando per la regione di inversione fino ad arrivare a drain, cosicché gli elettroni siano liberi di scorrere da source a drain (generando corrente) mentre il transistore è acceso.

Giunzione p-n

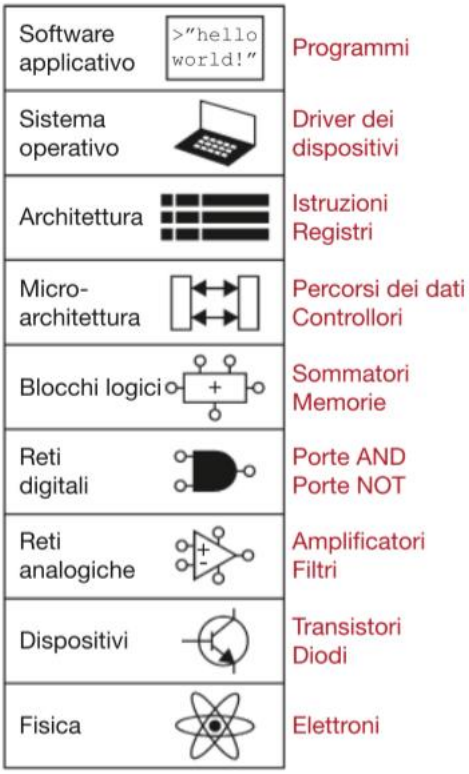
La giunzione p-n è l'interfaccia che separa le parti di un semiconduttore sottoposte a drogaggio differente, è composta da due zone, una con eccesso di lacune (strato p) e una con eccedenza di elettroni (strato n).

La regione di confine è detta zona di carica spaziale, qua i portatori di carica diffondono nel semiconduttore adiacente lasciando non compensati gli atomi ionizzati dei droganti i quali a loro volta genereranno una corrente di trascinamento che si oppone a quella di diffusione.

[1] Un diodo è una componente elettronica la cui funzione è quella di permettere il flusso di corrente elettrica.

Architettura del Compilatore

L'architettura del compilatore segue un modello gerarchico, tutto quello che ho a disposizione nel livello *i-esimo* è messo a disposizione gerarchicamente dal livello *i-1.*



Il So mette a disposizione una serie di programmi messi a diposizione a loro volta dal livello sottostante.

Questo modello segue vari principi:

1) Gerarchia: Implica il dividere il sistema in moduli, e dividere anche questi in sottomoduli finchè i pezzi che li compongono non siano facili da comprendere.

2) Modularità: Dobbiamo cercare di creare moduli intercambiabili.

3) Regolarità: Favorisce l'ottenimento di componenti efficienti con poche risorse di progettazione.

Numerazione logica binaria

Alfabeto binario: $\{0, 1\}$ $\{F, V\}$ $\{False, True\}$

Un bit è un valore nell'insieme {0, 1}

Dati n bit le combinazioni possibili sono 2^n.

$$\underset{bit_0}{2} * \ldots * \underset{bit_1}{2} * \ldots \ldots \ldots \ldots * \underset{bit_{n-1}}{2} * \ldots * \underset{bit_n}{2}$$

Se devo rappresentare #n valori necessito $\log_2 \#n$ bit.[2]

Potenze di 2 notevoli

$2^{10} = 1024 = 1K$

$2^{20} \cong 1'000'000 = 1M$

$2^{30} \cong 1'000'000'000 = 1G$

Esempio:

$8900 = 8K + resto$

$8K = 8 * K = 2^3 * 2^{10} = 2^{13}$

Da cui 8900 ~ 2^{14} quindi lo rappresenteremo su 14 bit.

[2] Arrotondiamo sempre all'intero superiore

Rappresentazioni Numeriche

Rappresentazione Binaria

Sia la base 2 che la base 10 sono sistemi posizionali, cioè il valore dipende dalla posizione delle cifre.

Base 10: $b = 10 \;\; n = 211$

$211 = 200 + 10 + 1$

$211 = 2 * b^2 + 1 * b^1 + 1 * b^0$

Base 2: $b = 2 \;\; n = 101$

$101 = 1 * b^2 + 0 * b^1 + 1 * b^0$

$101 = 4 + 0 + 1$

8 bit prendono il nome di byte, 4 bit prendono il nome di nibble.

Su 8 bit, ovvero un byte posso rappresentare numeri interi positivi nell'intervallo $[0, 255]$.

Trasformazione Decimale → Binario

Abbiamo due metodi:

1) Dato n, sottraggo la prima potenza di 2 che più gli si avvicina e metto ad 1 il bit di indice corrispondente all'esponente della potenza. Continuo fintantoché n non è 0.

   Esempio: $n = 50$

   $$\begin{array}{ll} 50 - 32 = 18 & b_5 = 1 \\ 18 - 16 = 2 & b_4 = 1 \\ 2 - 2 = 0 & b_1 = 1 \end{array}$$

   Il numero 50 su 8 bit sarà rappresentato come 00110010_2.

   Attenzione: l'ultimo bit, il più a destra ha indice b_0 (che corrisponde anche all'esponente della potenza di 2 per cui lo moltiplichiamo).

2) Dato un n, dividiamo per due, se il divisore è pari il bit corrispondente alla *i-esima* divisione sarà 0, altrimenti 1.
   (Praticamente consideriamo se la divisione ha resto, se lo ha: 1, altrimenti 0).

   Esempio: $n = 50$

   $$\begin{array}{ll} {}^{50}/_{2} = 25 & b_0 = 0 \\ {}^{25}/_{2} = 12 & b_1 = 1 \\ {}^{12}/_{2} = 6 & b_2 = 0 \\ {}^{6}/_{2} = 3 & b_3 = 0 \\ {}^{3}/_{2} = 1 & b_4 = 1 \\ {}^{1}/_{2} = 0 & b_5 = 1 \end{array}$$

   Su 8 bit il numero è 00110010_2

   Notiamo che dobbiamo leggere le cifre del numero dal basso verso l'alto ed eventualmente aggiungere degli zeri a sinistra.

Somma tra bit

La somma nel sistema binario è implementata secondo le seguenti regole:

1) $1 + 0 = 1\ con\ resto = 0 \quad e \quad 0 + 1 = 1\ con\ resto = 0$
2) $0 + 0 = 0\ con\ resto = 0$
3) $1 + 1 = 0\ con\ resto = 1$

Moltiplicazione tra bit

È implementata secondo le normali regole di moltiplicazione

1) $0 * 0 = 0$
2) $1 * 0 = 0 \quad e \quad 0 * 1 = 0$
3) $1 * 1 = 1$

Numeri Relativi (positivi o negativi)

Per la rappresentazione di numeri con segno in binario si hanno rappresentazioni diverse:

1) Modulo e segno:
   In questa rappresentazione si utilizza 1 bit aggiuntivo, il più significativo ci indica il segno (0 positivo e 1 negativo).
   Questo crea però dei problemi in particolare non abbiamo più un'unica rappresentazione dello 0.

   Esempio:

$$\begin{aligned} +15 &= 01111 \\ +0 &= 00000 \\ -0 &= 10000 \\ -15 &= 11111 \end{aligned}$$

2) Complemento a 2:
   Se il numero è positivo usiamo la rappresentazione normale, se il numero è negativo invece:
   - Trasformiamo il valore assoluto del numero in binario
   - Facciamo il complemento (invertiamo gli zeri e gli uni)
   - Sommiamo il numero 1_{10} → 0001_2

Vantaggi della notazione in complemento a due:

1. I numeri negativi hanno il bit più significativo sempre ad 1
2. Esiste un unico 0_{10} → 0000_2
3. Somma e sottrazione si fanno nello stesso modo sia per numeri positivi che negativi
4. Per convertire un numero da complemento a due in decimale facciamo nuovamente il complemento a due e poi usiamo il solito metodo.

Gli svantaggi sono che possiamo rappresentare (in valore assoluto) solo la metà dei valori che potrei rappresentare non usando il bit del segno.

Sistema	Range
Senza segno	$[0, 2^N - 1]$
Modulo e segno	$[-2^{N-1} + 1, 2^{N-1} - 1]$
Complemento a due	$[-2^{N-1}, 2^{N-1} - 1]$

Sottrazione tra bit

La sottrazione tra due numeri binari n e m viene vista come la somma tra n ed m negato.

$$n - m = n + (-m)$$

In particolare la negazione del numero viene fatta con il complemento a 2.

Esempio: $5 - 2$

$0101_2 \; - 0010_2$ Il secondo valore va trasformato in complemento a 2. Ottengo: 1110_2

L'operazione diventa quindi:

$$\begin{array}{r} 0101 + \\ 1110 = \\ \hline 0011 \end{array}$$

Questa somma genera un resto di 1 che però possiamo ignorare in quanto il segno non è cambiato.

Esempio problematico

Vogliamo fare la somma della rappresentazione binaria di 5 e 4 in complemento a 2.

$5 + 4$

$$\begin{array}{r} 0101 + \\ 0100 = \\ \hline 1001 \end{array}$$

Sembra che venga fuori un numero negativo, abbiamo infatti un trabocco sul bit del segno, siamo in una situazione di *overflow.*

Come abbiamo visto in complemento a 2 possiamo rappresentare solo i numeri che vanno da: $-2^{n-1} \;\; a \;\; 2^{n-1} - 1$, il -1 è per la rappresentazione dello 0.

Nel corso utilizzeremo processori (e componenti) con una parola da 32 bit, utilizzeremo la rappresentazione binaria per gli interi positivi e il complemento a 2 per i numeri relativi.

Le istruzioni saranno rappresentate come offset.

Rappresentazione Ottale

Esiste una via di mezzo tra la rappresentazione binaria e quella esadecimale, quella ottale, in questa rappresentazione prendiamo gruppi di 3 bit e li rappresentiamo con un simbolo diverso.

Alfabeto $\{0 \, , \ldots \, , 7\}$ dove vengono "tradotti" gruppi da 3 bit.

Rappresentazione Esadecimale

A volte a causa del lungo numero di bit necessari per rappresentare l'informazione, la rappresentazione binaria viene sostituita con il sistema esadecimale $\{0, \ldots, 9, A, \ldots \, , F\}$

Posso prendere gruppi da 4 bit (nibble) e rappresentarli con una singola cifra esadecimale.

$$1011 \mid 1110 \mid 0000 \mid 0101$$

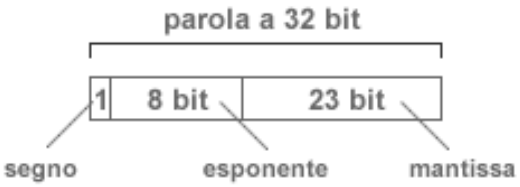
A E 0 9

Rappresentazione dei numeri in virgola mobile

Per rappresentare i floating point number, utilizziamo lo standard IEEE.

Un numero in virgola mobile è rappresentato da un campo esponente e da un campo mantissa.

$1.23\ E^2$ Mantissa = 1.23



Codice ASCII

I numeri compresi nell'intervallo [–128, 127] possono essere memorizzati in un byte invece di occupare un'intera parola di sistema.[3] Dal momento che i caratteri necessari per scrivere in lingua inglese sono molto meno di 256, spesso vengono codificati in un solo byte.

#	Carat-tere	#	Carat-tere	#	Carat-tere	#	Carat-tere	#	Carat-tere	#	Carat-tere
20	spazio	30	0	40	@	50	P	60	`	70	p
21	!	31	1	41	A	51	Q	61	a	71	q
22	"	32	2	42	B	52	R	62	b	72	r
23	#	33	3	43	C	53	S	63	c	73	s
24	$	34	4	44	D	54	T	64	T	74	t
25	%	35	5	45	E	55	U	65	e	75	u
26	&	36	6	46	F	56	V	66	f	76	v
27	'	37	7	47	G	57	W	67	g	77	w
28	(	38	8	48	H	58	X	68	h	78	x
29	)	39	9	49	I	59	Y	69	i	79	y
2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
2F	/	3F	?	4F	O	5F	_	6F	o		

[3] Nel nostro caso la parola di sistema è da 32 bit

Logica Booleana

È composta da 3 operazioni: and, or, not definite sull'alfabeto $\{0, 1\}$.

$A, B, C \in \{0, 1\}$

Congiunzione: $A \wedge B = 1 \Leftrightarrow A = 1 \; \boldsymbol{e} \; B = 1$

Disgiunzione: $A \vee B = 1 \Leftrightarrow A = 1 \; \boldsymbol{o} \; B = 1$

Negazione: $Not\ A = 1 \Leftrightarrow A = 0 \qquad Not\ A = 0 \Leftrightarrow A = 1$

Def a parole di una funzione che fa il confronto tra due bit:
Se sono uguali allora 1, se sono diversi 0.

Tabella di verità

1 colonna per ogni ingresso, tante righe quante solo le combinazioni di ingresso.
1 sola colonna in uscita.

A	B	Z
0	0	1
0	1	0
1	0	0
1	1	1

$$z = (\neg A \wedge \neg B) \vee (A \wedge B)$$

Le espressioni logiche del genere possono essere riscritte in forma canonica come somma di prodotti.

Nella somma di prodotti abbiamo la seguente notazione:

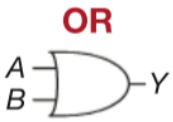
$\wedge = \cdot \quad \neg = \bar{} \quad \vee = +$ con z che diventa quindi $z = \bar{A} \cdot \bar{B} + A \cdot B$

Nelle tabelle di verità abbiamo k variabili, che generano 2^k righe.
Come vedremo con l'algebra booleana potremo ottimizzare la funzione normalizzandola.

Rappresentazione mediante porte logiche

$Y = AB$

A	B	Y
0	0	0
0	1	0
1	0	0
1	1	1



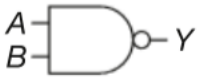
$Y = A + B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	1

$Y = \bar{A}$

A	Y
0	1
1	0

NAND



$Y = \overline{AB}$

A	B	Y
0	0	1
0	1	1
1	0	1
1	1	0

NOR

$Y = \overline{A + B}$

A	B	Y
0	0	1
0	1	0
1	0	0
1	1	0

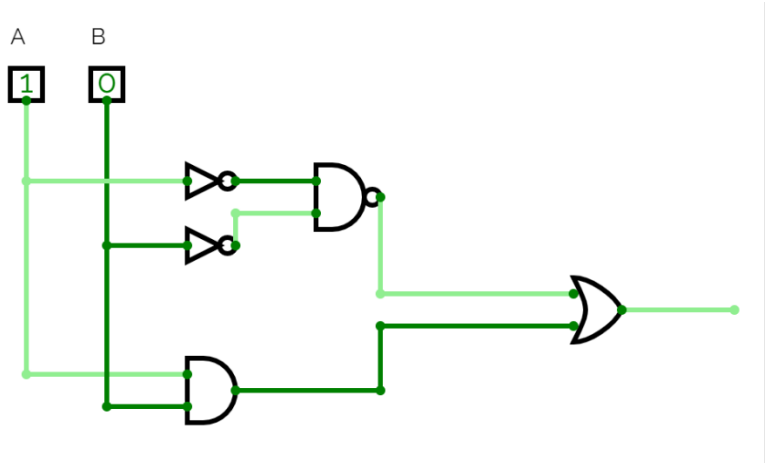
XOR

$Y = A \oplus B$

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Circuiti Logici

Esempio di circuito logico che calcola la funzione: $z = (\bar{A}\,\bar{B}) + (AB)$



Scriviamo ora una tabella di verità rappresentante una funzione logica che restituisce 1 se la maggioranza dei bit in ingresso sono settati ad 1 e 0 altrimenti.

A	B	C	D	E	z
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	0
0	0	1	1	0	0
0	0	1	1	1	1
…	…	…	…	…	…

A	B	C	D	E	z
0	0	0	-	-	0
-	-	0	0	0	0
-	0	0	0	-	0
…	…	…	…	…	…

Le tabelle di verità si possono semplificare con i don't care, i don't care sono rappresentati come ' **–** ' e ci indicano una cella di cui non ci interessa il valore.

Con i don't care non ci importa se quella cella è 0 o 1 siamo infatti in grado di definire già il risultato della riga della tabella, possono essere applicati anche sulla colonna di uscita.

In questo caso specifico per la soluzione della nostra funzione sarebbe meglio il ragionamento inverso, cerchiamo infatti gli 1.

A	B	C	D	E	z
1	1	1	-	-	1
-	1	1	1	-	1
-	1	-	1	1	1
-	-	1	1	1	1
1	-	-	1	1	1
1	-	1	1	-	1
1	-	1	-	1	1
1	1	-	1	-	1
1	1	-	-	1	1
…	…	…	…	…	…

Le tabelle originate con i don't care sono molto più piccole.

Possiamo rappresentare la funzione logica come:

$$z = ABC + BCD + BDE + CDE + \cdots$$

Avremmo potuto anche prendere i termini che generano " 0 " e fare la somma di prodotti per poi invertirla.

Priorità

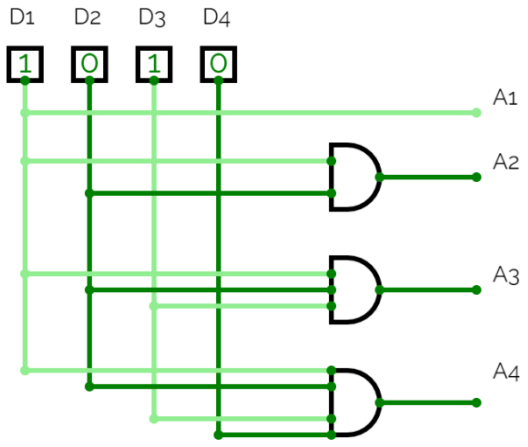
Assumiamo di avere dispositivi diversi, vorremmo poter assegnare delle priorità tramite un arbitro.

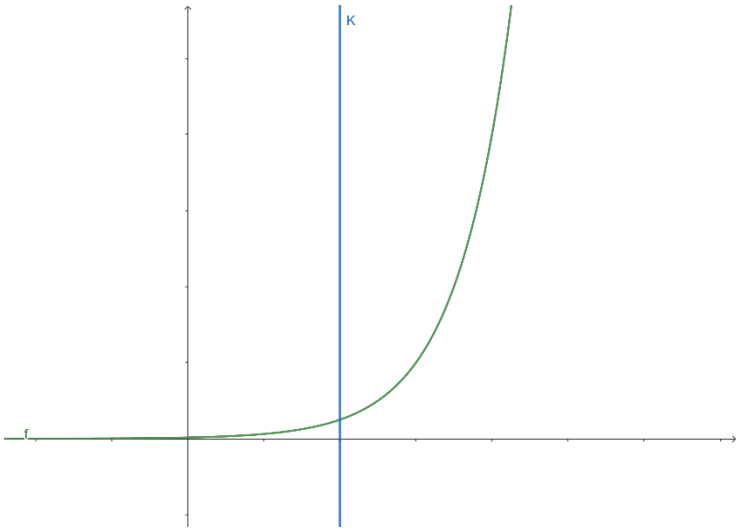
Immaginiamo una funzione logica con annessa tabella di verità che dà priorità agli *i* più piccoli.

D_1	D_2	D_3	D_4	A_1	A_2	A_3	A_4
0	0	0	0	0	0	0	0
1	-	-	-	1	0	0	0
0	1	-	-	0	1	0	0
0	0	1	-	0	0	1	0
0	0	0	1	0	0	0	1

Ogni colonna A_i è una somma di prodotti (e di conseguenza un circuito) è come se fossero 4 tabelle di verità (e quindi 4 circuiti).

$$A_1 = D_1 \;\; A_2 = \overline{D_1} D_2 \;\; A_3 = \overline{D_1 D_2}\, D_3 \;\; A_4 = \overline{D_1 D_2 D_3} D_4$$

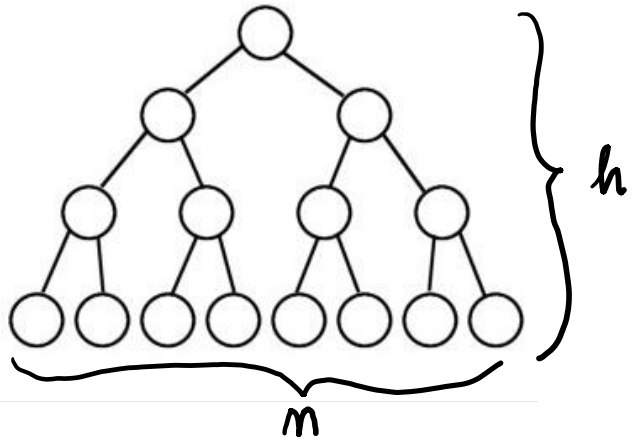




C'è un problema nella costruzione di questi circuiti, all'aumentare del numero di ingressi di una porta logica aumenta il delay.

Nella figura l'asse delle Y rappresenta il delay, l'asse delle x il numero di ingressi logici.

I nostri dispositivi logici sono utilizzabili fino ad un massimo di k ingressi, nel resto del corso considereremo k = 8, quando avremo necessità di utilizzare più ingressi li splitteremo in più porte creando un albero.

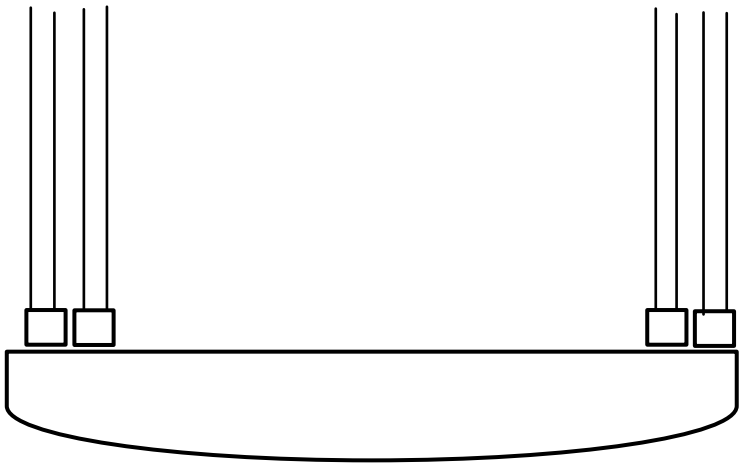


h è il numero di livelli, e n che è il numero di foglie è il numero di ingressi di cui necessitiamo. Se utilizziamo dispositivi logici con porte binarie (ovvero con 2 ingressi) il numero di livelli necessari è dato da: $h = \log_2 n$

Se le porte dovessero avere z ingressi $h = \log_z n = \frac{\log_2 n}{\log_2 z}$

Esempio:

Vogliamo confrontare 2 numeri da 32 bit



Questa in figura è la porta logica di una tabella di verità a 64 ingressi, 2^{64} combinazioni (righe).

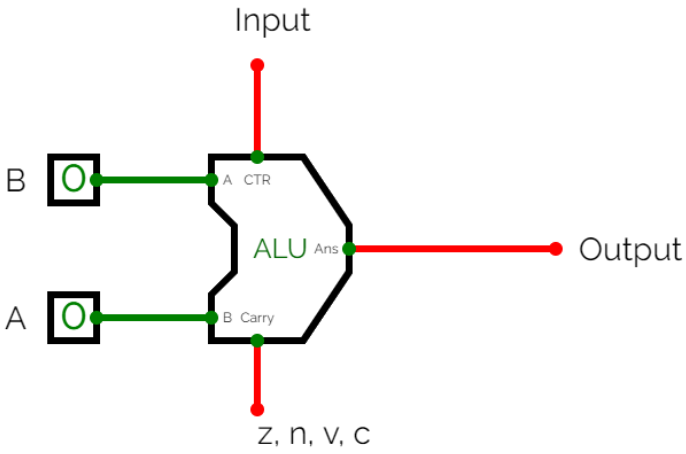
L'ultima è la 32-esima porta, per vedere se sono tutti uguali serve un and con 32 ingressi i quali sono il risultato di un confrontatore con 2 ingressi.

Anziché avere una sola porta con 8 ingressi applichiamo il principio precedente e usiamo 4 porte da 8 ingressi.

Abbiamo 2 livelli logici, ogni livello costa 1 Δt, abbiamo quindi un costo totale di 2 Δt.

Per confrontare due numeri ci sono però metodi più efficienti.

Uno di questi è quello di realizzare questo confrontatore con una alu.



z e n sono dei bit del riporto che mi dicono se il risultato è rispettivamente zero, o negativo.

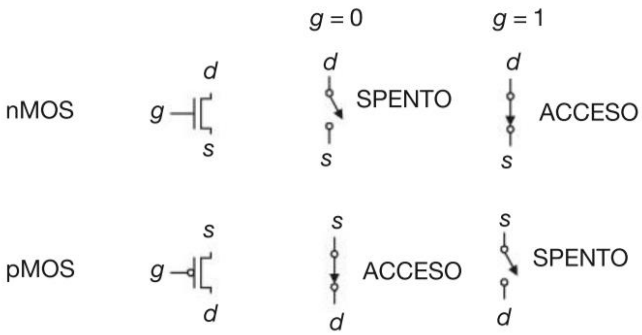
Abbiamo poi v che sta per overflow e c che è il bit del carry.

(Cliccando qui è possibile vedere un approfondimento, non visto a lezione, su come le alu sono implementate)

Reti logiche combinatorie

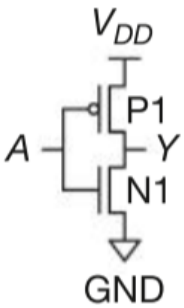
La componente base è il transistor, all'aumentare di questi aumenta l'area ed il costo, aumenta anche il costo di alimentazione.

Transistor schematizzato

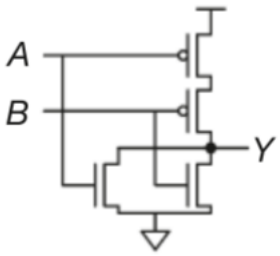


Rappresentazione di porte logiche come transistor

La porta not come circuito è rappresentata nel modo seguente:



La porta nor invece nel modo seguente:



Nel not se aumentiamo la tensione a +1v, apro il circuito mando uno 0 ed esce 1, se invece metto entrambi a 0 metto un 1 ed esce 0.

Algebra Booleana

Assiomi e teoremi, vediamo come usare gli assiomi per ottimizzare i calcoli.

Not: $\bar{0} = 1$ $\bar{1} = 0$

And: $0 \cdot 0 = 0$ $1 \cdot 1 = 1$ $1 \cdot 0 = 0$ $0 \cdot 1 = 0$

Commutatività

Or: $0 + 0 = 0$ $1 + 1 = 1$ $1 + 0 = 1$ $0 + 1 = 1$

Teoremi

Usiamo A,B, C come variabili logiche $\{0, 1\}$

Identità:	$A \cdot 1 = A$	$A + 0 = A$
Elemento nullo:	$A \cdot 0 = 0$	$A + 1 = 1$
Idempotenza:	$A \cdot A = A$	$A + A = A$
Involuzione:	$\bar{\bar{A}} = A$	
Complementi:	$A \cdot \bar{A} = 0$	$A + \bar{A} = 1$
Commutatività:	$A \cdot B = B \cdot A$	$A + B = B + A$
Associatività:	$(A \cdot B) \cdot C = A \cdot (B \cdot C)$	$(A + B) + C = A + (B + C)$
Distributività:	$A \cdot (B + C) = (A \cdot B) + (A \cdot C)$	
De Morgan:	$\overline{A \cdot B} = \bar{A} + \bar{B}$	

Tabella di verità di De Morgan

A	B	$A \cdot B$	$\overline{(A \cdot B)}$	$\bar{A}$	$\bar{B}$	$\bar{A} + \bar{B}$
0	0	0	1	1	1	1
0	1	0	1	1	0	1
1	0	0	1	0	1	1
1	1	1	0	0	0	0

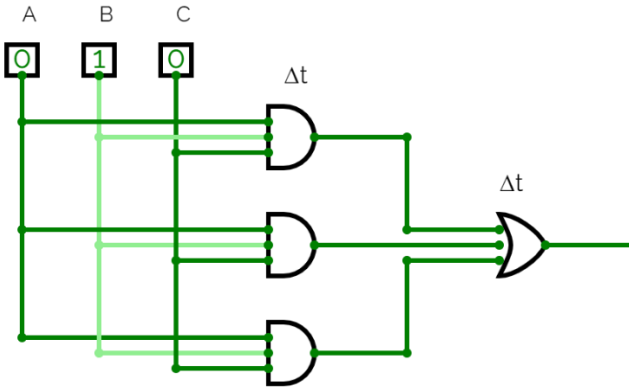
Con l'algebra booleana possiamo semplificare le funzioni portandole in forma normale, in questo modo possiamo diminuire il numero di porte, aumentando quindi la velocità, diminuendo il design time dei circuiti e il costo in termini di silicio e di consumi.

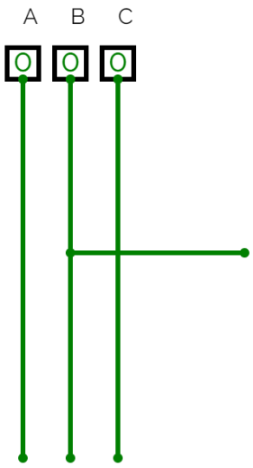
Semplificazione di espressioni

1) $abc + ab\bar{c} + \bar{a}b$
   $ab(c + \bar{c}) + \bar{a}b$
   $ab + \bar{a}b$
   $b(a + \bar{a})$
   b

Il ritardo della funzione non ottimizzata sarebbe stato di $2\Delta t$, il ritardo dell'equazione semplificata con le regole algebriche è 0.

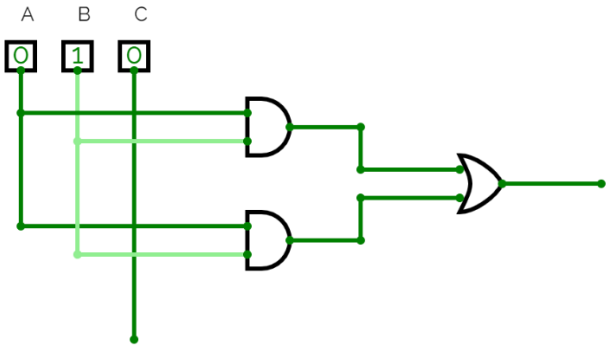
Nella figura vediamo l'immagine relativa al circuito originario a sinistra e quella relativa al circuito ottenuto con la formula semplificata a destra.





A	B	C	ABC	$AB\bar{C}$	$\bar{A}B$	$+$	B
0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0
0	1	0	0	0	1	1	1
0	1	1	0	0	1	1	1
1	0	0	0	0	0	0	0
1	0	1	0	0	0	0	0
1	1	0	0	1	0	1	1
1	1	1	1	0	0	1	1

2) $abc + a\bar{b}\bar{c} + ab\bar{c}$
   $abc + a\bar{b}\bar{c} + ab\bar{c} + ab\bar{c}$
   $ab(c + \bar{c}) + a\bar{c}(\bar{b} + b)$
   $ab + a\bar{c}$
   $a(b + \bar{c})$

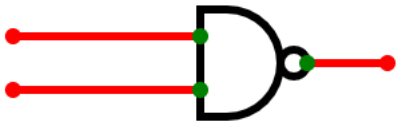


3) $a\bar{b}\bar{c} + ab\bar{c} + \bar{a}bc + abc$
$a\bar{c}(\bar{b} + b) + bc(\bar{a} + a)$
$a\bar{c} + bc$

Combinazione di porte:

A partire da un solo tipo di componente (nand) possiamo creare tutte le altre.

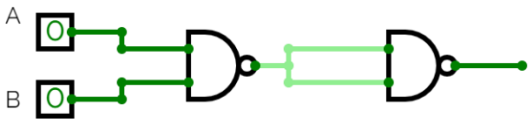
Si perde in termini di porte e di tempo ma utilizziamo solamente un componente.



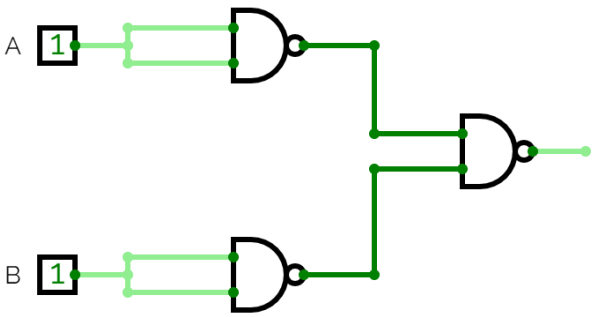
A	*B*	*z*
0	0	1
0	1	1
1	0	1
1	1	0

Porta not:

Porta And:



Porta or:



Ci sono vari modi per minimizzare ed esiste un formalismo per la minimizzazione del numero di porte.

Mappe di karnaugh

Le mappe di karnaugh sono un'alternativa alle tabelle di verità, per un numero basso di ingressi sono buone, si utilizzano per semplificare la funzione originale, raggruppiamo gli "1" in gruppi di potenze di 2

Esempio:

Immaginiamo di avere 2 variabili e la seguente tabella di verità:

A	B	z
0	0	1
0	1	1
1	0	1
1	1	0

AB\CD	0	1
0	1	1
1	1	0

Con 4 variabili avremmo:

AB\CD	00	01	11	10
00	1	1	1	1
01		1	1	
11	1	1	1	1
10	1	1	1	1

Somma

Vediamo la tabella di verità di una funzione che fa la somma bit a bit di 2 numeri x e y rappresentati con 2 bit, la funzione genera il risultato z e il bit di riporto c.

x_1	x_0	y_1	y_0	z_1	z_0	c
0	0	0	0	0	0	0
		0	1	0	1	0
		1	1	1	1	0
0	1	1	0	1	0	0
		0	0	0	1	0
		0	1	1	0	0
		1	1	0	0	1
1	1	1	0	1	1	0
		0	0	1	0	0
		0	1	1	1	0
		1	1	0	1	1
1	0	1	0	0	0	1
		0	0	1	1	0
		0	1	0	0	1
		1	1	1	0	1
		1	0	0	1	1

Le mappe di karnaugh della funzione sono le seguenti:

Z_1:

$x_1x_0\backslash y_1y_0$	00	01	11	10
00	0	0	1	1
01	0	1	0	1
11	1	0	1	0
10	1	1	0	0

$$z_1 = y_1\overline{x_1}\,\overline{x_0} + \overline{x_1}x_0\overline{y_1}y_0 + x_1x_0y_1y_0 + y_1\overline{y_0x_1} + \overline{y_1y_0}x_1 + x_1\overline{x_0y_1}$$

Z_0:

$x_1x_0\backslash y_1y_0$	00	01	11	10
00	0	1	1	0
01	1	0	0	1
11	1	0	0	1
10	0	1	1	0

$$z_0 = y_0\overline{x_0} + \overline{y_0}x_0$$

C:

$x_1x_0\backslash y_1y_0$	00	01	11	10
00	0	0	1	0
01	0	0	0	0
11	0	0	1	1
10	0	1	1	1

$$c = y_1y_0\overline{x_1x_0} + x_1\overline{x_0y_1}y_0 + y_1x_1$$

Algoritmo per costruire le mappe di Karnaugh

Data la tabella di verità per costruire la mappa:

1) Si scelgono due gruppi di letterali (variabili di ingresso). Nelle righe e nelle colonne indichiamo le combinazioni di valori (0, 1) che i letterali possono assumere.
2) A questo punto si individuano le celle della mappa a cui corrispondono le clausole dell'espressione, ovvero si mette un "1" alle celle i cui indici restituiscono 1 nella tabella di verità
3) Si raggruppano le celle confinanti in gruppi di potenze di 2: 1, 2, 4, 8….
   I gruppi possono anche sovrapporsi parzialmente.
4) Si trasformano i gruppi in congiunzioni letterali, prendendo i letterali che non cambiano valore, e se uno di questi ha valore "0" lo nego.
5) Sommare le congiunzioni letterali così da ottenere l'espressione booleana in forma ridotta della funzione.

Nota: è possibile scambiare l'ordine delle colonne per ottenere un migliore raggruppamento di "1".

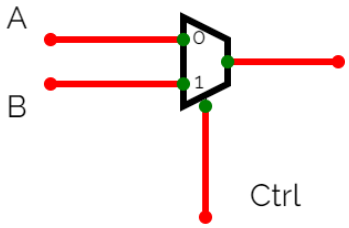
Componenti combinatorie

Multiplexer

Il multiplexer è una componente combinatoria basica che sceglie tra 2 ingressi A e B, la scelta viene fatta grazie ad un ingresso di controllo, se il bit di ctrl è 0 sceglie A, se è 1 sceglie B.

Un multiplexer genera un ritardo di $2\Delta t$.

In realtà i multiplexer possono avere 2^k ingressi, necessitiamo quindi di k bit di controllo.

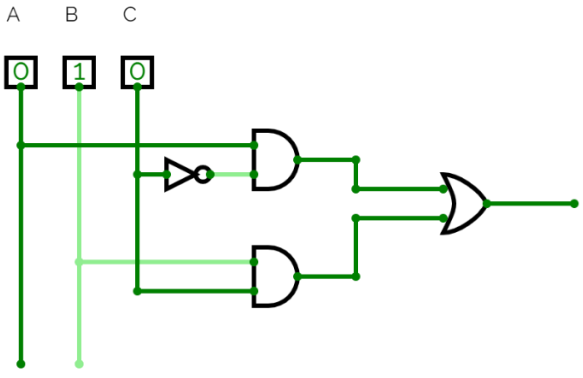


C\AB	00	01	11	10
0			1	1
1		1	1	

$z = a\bar{c} + b$

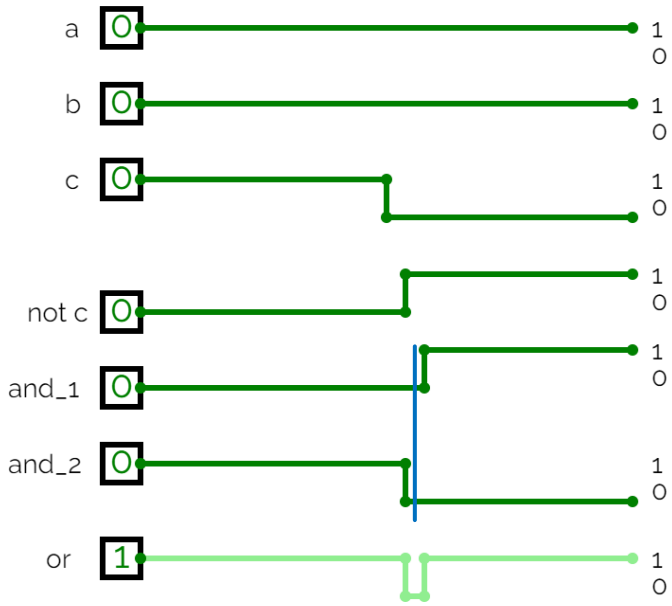
c	a	b	z
0	1	-	1
1	-	1	1

Circuito logico:



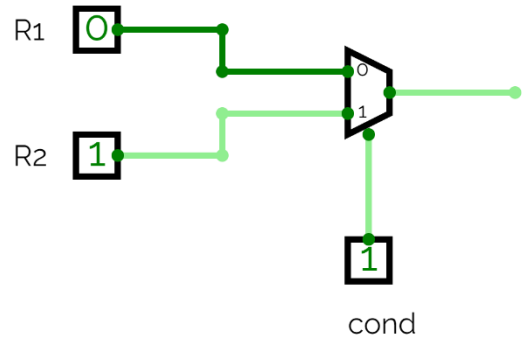
Il multiplexer ha un ritardo di $2\Delta t$

Cosa succede se passiamo da abc = 111 verso abc = 110?



Partiamo che sono tutti a 1, dopo un Δt c scende a 0, passando tra due configurazioni (linea blu) l'uscita cambia quando in realtà non dovrebbe.

Non sempre vogliamo scegliere tra due segnali, potremmo voler scegliere tra due risultati.



È possibile anche avere più ingressi, ad esempio 4 naturali in ingresso e 2 bit di controllo.

In questo caso nella tabella di verità avremmo 6 ingressi.

Questo è possibile implementarlo in 2 modi diversi, o utilizzando 3 multiplexer e generando $4\Delta t$ oppure sfruttando 2 livelli di or + 1 livello di and che genererebbero $3\Delta t$

Esempio

Simuliamo il calcolo su 4 bit della computazione "bit più frequente"

a	b	c	d	m	p
0	0	0	0	0	0
		0	1	0	0
		1	1	-	1
		1	0	0	0
0	1	0	0	0	0
		0	1	-	1
		1	1	1	0
		1	0	-	1
1	1	0	0	-	1
		0	1	1	0
		1	1	1	0

		1	0	1	0
1	0	0	0	0	0
		0	1	-	1
		1	1	1	0
		1	0	-	1

m indica la maggioranza, p la parità

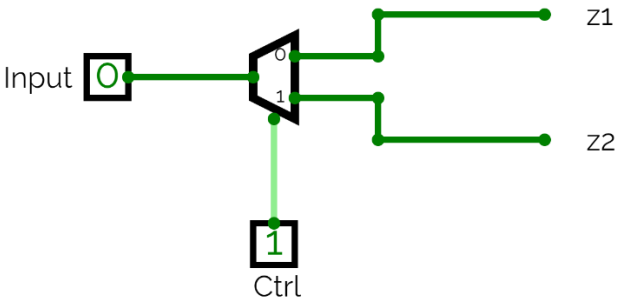
La mappa di karnaugh che andiamo a creare avrà i don't care, siccome i "-" non sono specificati posso assumere che siano 1 o 0 a mia convenienza.

AB\CD	00	01	11	10
00	0	0	-	0
01	0	-	1	-
11	-	1	1	1
10	0	-	1	-

$$m = bd + ac$$

Demultiplexer

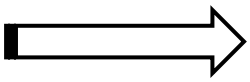
Un demultiplexer manda l'unico ingresso su una delle 2^k uscite, necessiterà al caso generale di k bit di controllo.



$$if(ctrl == 0)\ then\ z_1 = a;\ z_2 = 0;$$
$$else\ z_1 = 0; z_2 = a;$$

La tabella di verità del demultiplexer ha un "1" per colonna nelle colonne di uscita.

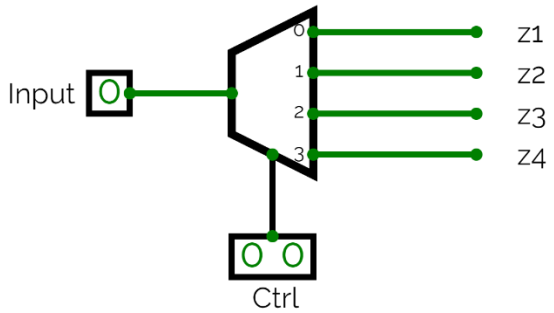
ctrl	a	Z_1	Z_2
0	0	0	0
	1	1	0
1	0	0	0
	1	0	1



ctrl	A	Z_1	Z_2
0	1	1	0
1	1	0	1

$$z_1 = \overline{ctrl} \cdot a \quad z_2 = ctrl \cdot a$$

Se avessimo avuto 4 uscite:



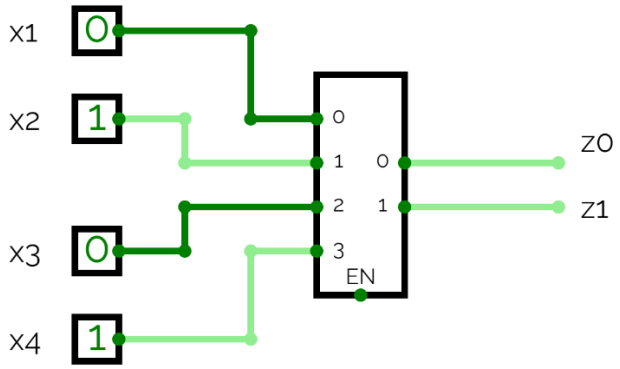
Come si vede dalla figura all'aumentare del numero di uscite aumenta anche la banda (ovvero il numero) di bit di controllo, in particolare la relazione è che per un demultiplexer di 2^k uscite servono k bit.

Per meno di 256 uscite il tempo di un demultiplexer è $1\Delta t$.

Se volessi un ingresso e k bit di controllo dovrei dire che ho 2^k uscite tutte da un bit.

Codificatore

Il codificatore ha *i* ingressi e N uscite con *i* <= 2^N e attiva una delle sue uscite a seconda della combinazione di valori in ingresso, qualcosa ci garantisce che in uno di questi k ingressi ne esiste uno e solo uno che vale 1.



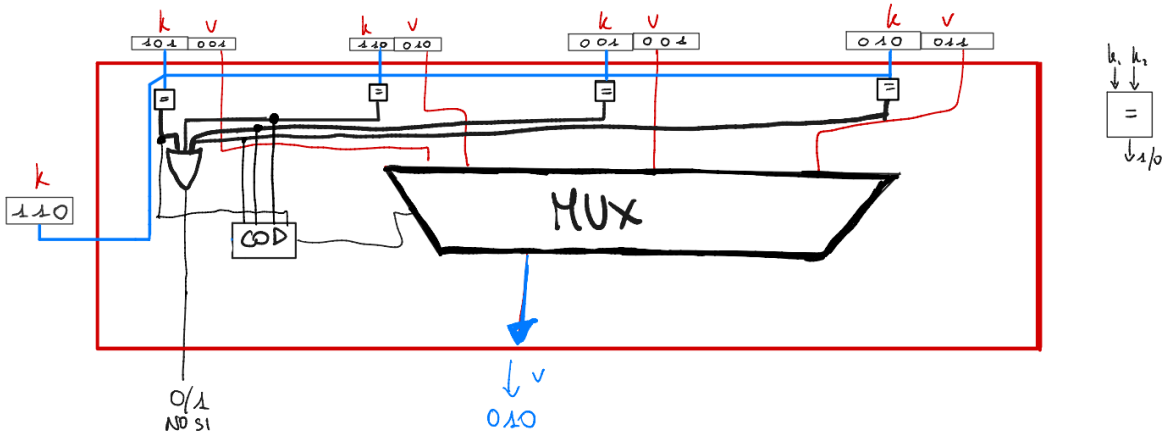
X_1	X_2	X_3	X_4	Z_1	Z_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

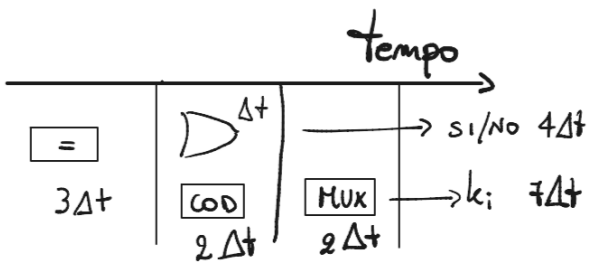
$$z_1 = \overline{x_1}x_2\overline{x_3}x_4 + x_1\overline{x_2}\,\overline{x_3}\,\overline{x_4} \qquad z_2 = \overline{x_1}\,\overline{x_2}\,x_3\,\overline{x_4} + x_1\overline{x_2}\,\overline{x_3}\,\overline{x_4}$$

Nella tabella di verità abbiamo solo k righe e non 2^k in quanto consideriamo solo le configurazioni con un unico bit a 1.

Hash Map

Immaginiamo di avre 4 coppie chiave valore, in cui ogni campo ha 3 bit. Vogliamo sapere se esiste una determinata chiave e se c'è il valore corrispondente. Se almeno un bit dato dall'output dei confrontatori è settato ad 1 allora ho una corrispondenza per la chiave cercata.





Sommatore/confrontatore

Come funziona il confrontatore?

Come verifico se due numeri sono uguali?

Serve un sommatore, infatti per farlo mi basta sommare il primo numero al complemento a 2 del secondo e se il risultato è 0 sono uguali.

Il confronto tra due bit si implementa con uno xor negato.

Se i bit da confrontare sono k, utilizzo k xor negati uniti da un and che deve restituire 1.

Verilog

Per uno studio completo e approfondito al seguente link sono disponibili le dispense del Professore Marco Danelutto:

Verilog è un linguaggio che ci permette di simulare i circuiti

HDL: Hardware description language

RTL: Register transfer language

Il linguaggio Verilog, come tutti gli altri linguaggi "RTL" (Register Transfer Languages), è un linguaggio per la descrizione dell'hardware.
In quanto tale permette di definire componenti che calcolano funzioni, con o senza stato, che possono successivamente essere utilizzati come moduli di altri componenti. I linguaggi RTL possono essere utilizzati per la simulazione o per la sintesi di circuiti.
Nel primo caso, il programma che descrive un certo circuito (componente) viene utilizzato per simularne il comportamento e per controllare dunque che calcoli ciò per cui era stato progettato.

Nel secondo caso, il programma viene utilizzato per generare le specifiche da utilizzare per realizzare un circuito che implementi fisicamente quello descritto dal programma.

Le specifiche possono consistere nello schema di realizzazione di un integrato VLSI o nel file di configurazione di una FPGA.

Questi tipi di linguaggi si trovano a bassissimo livello.

In Verilog si possono utilizzare diversi tipi di dati: costanti (literal), wire, registri, vettori e interi (variabili generiche).

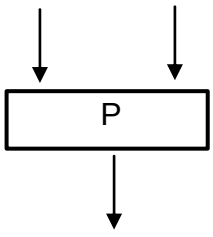
Le operazioni che si possono fare operano su bit, gruppi di bit ecct…

Vediamo come rappresentare gruppi di bit:

[0 : 3] y accediamo agli elementi con y[0], …., y[3]

[3 : 0] y accediamo agli elementi nell'ordine inverso y[3], …, y[0]

Possiamo eseguire dei moduli che eseguono determinate azioni:



Possiamo scrivere dei programmi di test che ci permettono di testare i moduli con valori significativi.

Uno degli usi che facciamo di verilog è quello di simulazione, non è però il solo, infatti una volta che abbiamo la descrizione dei moduli possiamo avviare un processo di sintesi che dal codice verilog descrive un circuito, testiamo chip senza produrli.

FPGA

Le fpga sono un insieme di celle di memoria coordinate da un programma, le colonne dell'fpga sono in parte ram in parte dsp, le colonne dsp calcolano somme e prodotti in virgola mobile.

Questi dispositivi permettono di realizzare funzioni logiche anche molto complesse e sono caratterizzati da un'elevata scalabilità.

Una fpga può essere calcolata tramite verilog/reti combinatorie, oppure tramite un programma fatto da moduli.

Sintassi ed esempi

Moduli

I moduli in Verilog rappresentano l'astrazione di una componente. Un modulo può rappresentare una funzione (rete logica) o una funzione con stato (rete sequenziale). I moduli possono essere ricorsivamente definiti come composizione di altri moduli.

Ci sono due modi per dichiarare i moduli in verilog, entrambi seguiti da end *tipo_modulo*

I tipi di moduli che possiamo dichiarare sono:

- Primitive: In cui mettiamo a disposizione una tabella di verità
- Module: Che ci permette di scrivere del codice

Ogni modulo primitive calcola un solo bit di uscita, se ci dovessero essere più bit di uscita servirebbero tanti moduli quanti sono i bit.

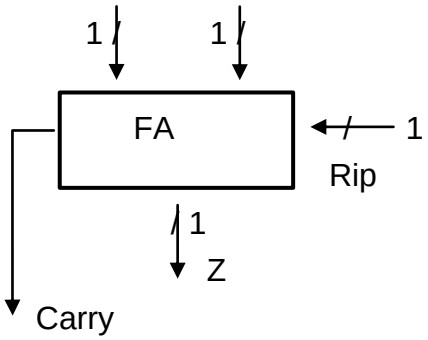
Per i moduli ti tipo module invece è possibile usare in uscita dei registri o dei wire di dimensione variabile.

I moduli prevedono:

- Un nome
- Una lista di parametri formali
- Un corpo, che definisce come i parametri di uscita vengono calcolati a partire dai parametri in ingresso.

Es: primitive A(parametri); oppure module B(parametri); seguiti poi rispettivamente da end primitive e end module.

Esempio full Adder da 1 bit



x	y	rip	carry
0	0	0	0
0	1	1	1

x	y	rip	Z
0	0	0	0
0	1	1	0

Per fare una tabella di verità di verità in verilog la racchiudiamo tra table e end table.
Descriviamo per ogni riga: *input* : *output*

Esempio:

table
0 0 : 0 ;
0 1 : 1 ;
1 0 : 1 ;
1 1 : 0 ;
endtable

Ricordiamo che se ci sono n uscite servono n tabelle di verità.

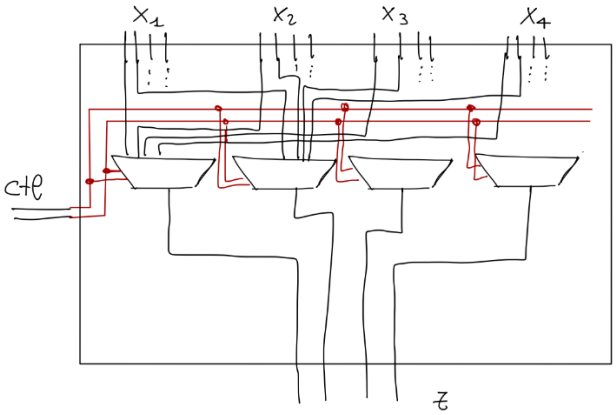
Al posto dei don't care "-" mettiamo i "?"

Di per se queste due tabelle di verità non fanno un full adder, devono infatti essere collegate, per collegarle si necessità di utilizzarle in un module, dentro possiamo usare le primitive, usiamo i wire per collegare i module.

Multiplexer
Se avessi avuto un modulo mux1 avrei potuto scrivere il codice seguente:

```
module mux4(output [3:0]z, input [3:0]x1, input [3:0]x2, input [3:0]x3, input [3:0]x4, input [1:0]ctrl);
    mux1 M4(z[3], x1[3], x2[3], x3[3], x4[3], ctrl);
    mux1 M3(z[2], x1[2], x2[2], x3[2], x4[2], ctrl);
    mux1 M2(z[1], x1[1], x2[1], x3[1], x4[1], ctrl);
    mux1 M1(z[0], x1[0], x2[0], x3[0], x4[0], ctrl);
endmodule
```

La logica è rappresentata dal seguente schema:



Test

Come possiamo istanziare i moduli e testarli?

Creiamo dei “test bench”, possiamo creare dei moduli appositi come ad esempio:

module testfa(); senza parametri, si comporta come un main.

All’interno di un modulo di test:

- Dichiariamo tante variabili di tipo reg (registri) quanti sono gli ingressi, specificando le dimensioni e i nomi.
  - Esempio: nome[] = 1 bit (controllare che sia vero)
- Dichiariamo tanti wire (cavi) quante sono le variabili di output
- Creiamo un’istanza del modulo dandogli un nome.
  - Esempio: fa nomemodulo(variabili)
- Nel modulo di test dopo la dichiarazione di queste variabili scriviamo un corpo racchiuso tra *initial* e *begin* in cui assegniamo dei valori ai registri, possiamo inoltre far variare il valore delle variabili ogni tot unità di tempo attraverso il comando #t *var = newValue* con t che rappresenta il tempo che deve trascorrere prima di ri-assegnare la variabile.

```
module testFA();
    reg inx, iny, inr;
    wire z, c;

    FA test(c, z, inx, iny, inr);

    initial begin
        inx=0; iny=1; inr=0;
        #1 inx=1;
    end

endmodule
```

Dichiariamo tante variabili (registri) quanti sono gli ingressi, dichiariamo poi tanti wire quanti sono gli output, creiamo un’istanza del modulo e in seguito inizializziamo le variabili.

Direttive

I comandi che possiamo utilizzare per la simulazione sono tutte direttive che iniziano col segno dollaro $. Le direttive permettono di salvare la traccia di esecuzione di una simulazione, di terminare la simulazione stessa o di stampare il valore delle variabili utilizzate sul terminale.

Fra i comandi che possiamo utilizzare per controllare la simulazione, citiamo:

```
initial begin

    $dumpfile("nomefile");   //crea file per supporto visuale con gtkwave
    $dumpvars;      //ci mostra le variabili che cambiano
    $time;      //restituisce il valore del tempo corrente
    $display(formato, lista variabili);
    /*mostra il contenuto delle variabili nella lista, secondo il formato
(opzionale).
```

```
    La stringa di formato (simile a quella della printf del C) utilizza %d, %b,
%t e %h
    per visualizzare valori in decimale, binario, di tempo e esadecimale,
rispettivamente*/
    $monitor(formato,lista variabili);
    //funziona come la display, ma esegue la stampa ogni volta che le variabili
cambiano valore
    $finish;        //a fine test prima di end

end
```

Compilazione ed esecuzione:

Per compilare si scrive il seguente comando sulla shell [4]:

iverilog nomefile.v test.v

Per eseguire lanciamo il seguente comando:

./a.out

È possibile vedere visualmente come si comporta il programma con il comando:

gtkwave file.vcd

Esempio: implementazione full Adder

Calcoliamo la somma tra due bit con un eventuale riporto iniziale:

La somma di due bit e un bit di riporto fa 0 se tutti i bit sono 0 o se solo 2 dei tre bit sono 1, fa 1 se esattamente uno dei tre ingressi è 1 oppure se lo sono tutti e tre.

Quindi il modulo può essere scritto come segue:

```
primitive fa_somma ( output s , input r , input x1 , input x2 );

 table
 0 0 0 : 0 ;
 0 0 1 : 1 ;
 0 1 0 : 1 ;
 0 1 1 : 0 ;
 1 0 0 : 1 ;
 1 0 1 : 0 ;
 1 1 0 : 0 ;
 1 1 1 : 1 ;
 endtable

endprimitive
```

Possiamo osservare che il riporto è 1 quando:

- il riporto iniziale è 0 e entrambi gli ingressi sono 1
- il riporto iniziale è 1 e almeno uno degli ingressi è 1.

[4] Assicurarsi di avere iverilog impostato come variabile d'ambiente per poterlo richiamare da shell.

```
primitive fa_riporto ( output s , input r , input x1 , input x2 );

 table
 0 0 0 : 0 ;
 0 0 1 : 0 ;
 0 1 0 : 0 ;
 0 1 1 : 1 ;
 1 0 0 : 0 ;
 1 0 1 : 1 ;
 1 1 0 : 1 ;
 1 1 1 : 1 ;
 endtable

endprimitive
```

Per definire il nostro full adder utilizziamo come componenti i due moduli primitive fa_somma e fa_riporto.

Entrambi i moduli prendono in ingresso gli stessi parametri del modulo che calcola la somma, ma uno calcola il bit di risultato e l'altro calcola il bit di riporto.

```
module fulladder ( output riporto , output risultato ,
 input riportoiniziale , input x1 , input x2 );

 fa_riporto m1 ( riporto , riportoiniziale , x1 , x2 );
 fa_somma m2 ( risultato , riportoiniziale , x1 , x2 ) ;

endmodule
```

Generate

Un caso particolare di comando che si può utilizzare per la definizione di un modulo è il blocco generate. Un blocco generate può essere utilizzato per generare un certo numero di istanze di componenti.

L'esempio che segue fa vedere come possiamo utilizzare il generate per istanziare una serie di multiplexer da due ingressi di un singolo bit per realizzare un multiplexer da due ingressi da N bit ciascuno.

```
module commutatore_nbit_generative (z ,x ,y , alpha );
 parameter N = 32;

 output [N-1:0] z;
 input [N-1:0] x;
 input [N-1:0] y;
 input alpha ;

 genvar i;

 generate
 for (i=0; i<N; i=i+1)
      begin
            mux2 t(z[i],x[i],y[i], alpha );
      end
 endgenerate
endmodule
```

Assign

In verilog anziché utilizzare le tabelle di verità per descrivere le nostre funzioni potremmo usare la somma di prodotti, questo viene fatto con il comando assign, ogni volta che una delle variabili a destra dell'uguale cambierà, z verrà aggiornata. (si chiama assegnamento continuo).

assign

z = formula;

Possiamo riscrivere il full adder nel modo seguente:

```
module fa(output c, output z, input x, input y, input r);
   assign { c, z } = x + y + r;
endmodule
```

Behavioural

Possiamo utilizzare statement del linguaggio per descrivere il comportamento di una cosa

```
initial begin

      //comandi

end
```

```
always @(x, y, z) begin
    //comandi
//Vengono eseguiti ogni volta che una variabile tra le parentesi cambia, se l'evento viene omesso, viene eseguito in continuazioine
end
```

```
always begin

end
```

Registri

I registri sono variabili che possono mantenere un valore

Reg x,a;

Reg [7:0] B;

Tutto quello a sx di "=" deve essere un registro.

I numeri si possono rappresentare specificando la base ed il numero di cifre, utilizzando la notazione <n>'<b>xxxx dove <n> in decimale rappresenta il numero di bit, <b> è un singolo carattere che rappresenta la base (d per decimale, b per binario, o per ottale e h per esadecimale).

Costrutti:

```
   if(B==255) A=1;
   else A=0;
```

```
  for(   ;   ; i=+/- );
```

```
case ( espressione )
valore1 : begin ... end
valore2 : begin ... end
 ...
default : begin ... end
endcase
```

Dentro ad un begin le istruzioni vengono eseguite in modo sequenziale,

```
begin

1    A=1;
2    B=0;

end
```

Prima viene eseguita l'istruzione 1 e poi la 2

A volte però vorremmo poter fare una cosa diversa:

```
begin

    A<=1;
    B<=0;

end
```

In questo caso l'assegnamento avviene in contemporanea

Codificatore 4x2

Indica la posizione dell'unico bit a 1.

a	b	c	d	z_1	z_0
0	0	0	1	0	0
0	0	1	0	0	1
0	1	0	0	1	0
1	0	0	0	1	1

```
module primitve z1(output z, input
a,b,c,d);
    table
        0000 : 0
        0001 : 0
          .
          .
          .
        0100 : 1
        1000 : 1
    endtable
endmodule
```

Avremmo potuto scrivere

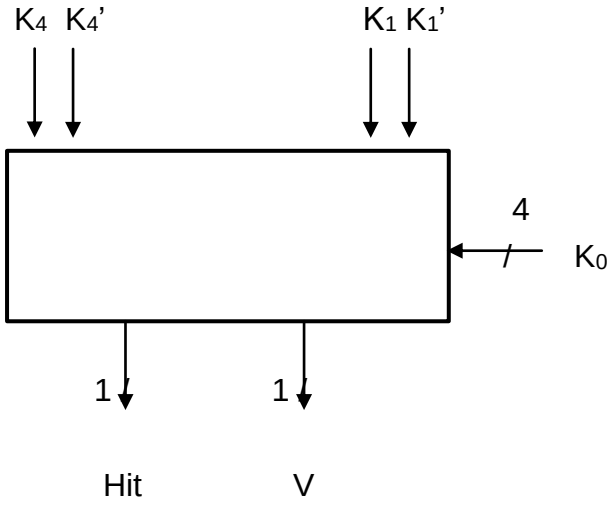
```
module z1(output z, input a,b,c,d);
    assign
        z= ~a and b and ~c and ~d || a
and ~b and ~c and ~d;
endmodule
```

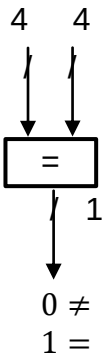
Avremmo potuto scriverlo anche come:

```
module z1(output reg[1:0] z, input
a,b,c,d)
    always @(a, b, c, d)
    begin
        case({a, b, c, d})
            1000:z=11
            0100:z=10
            0010:z=01
            default:z=00
        endcase
    end
endmodule
```

Le graffe servono a considerare i 4 bit uniti.

Confrontatore a 4 bit





Confrontatore di chiavi

Come si scrive in verilog?

```
module comp4(output z, input[3:0]x, input[3:0]y);
    assign
        z=~(x[3]^y[3])&&~(x[2]^y[2])&&~(x[1]^y[1])&&~(x[0]^y[0]);
endmodule
```

In questo codice usiamo:

- l'operatore XOR " ^ "
- l'operatore AND " && "
- l'operatore NOT " ~"

Alcune varianti potrebbero essere:

1)

```
module
    reg ris;
    ris = 1;
        for()
            ris=ris&&(~(x[i]^y[i]))
endmodule
```

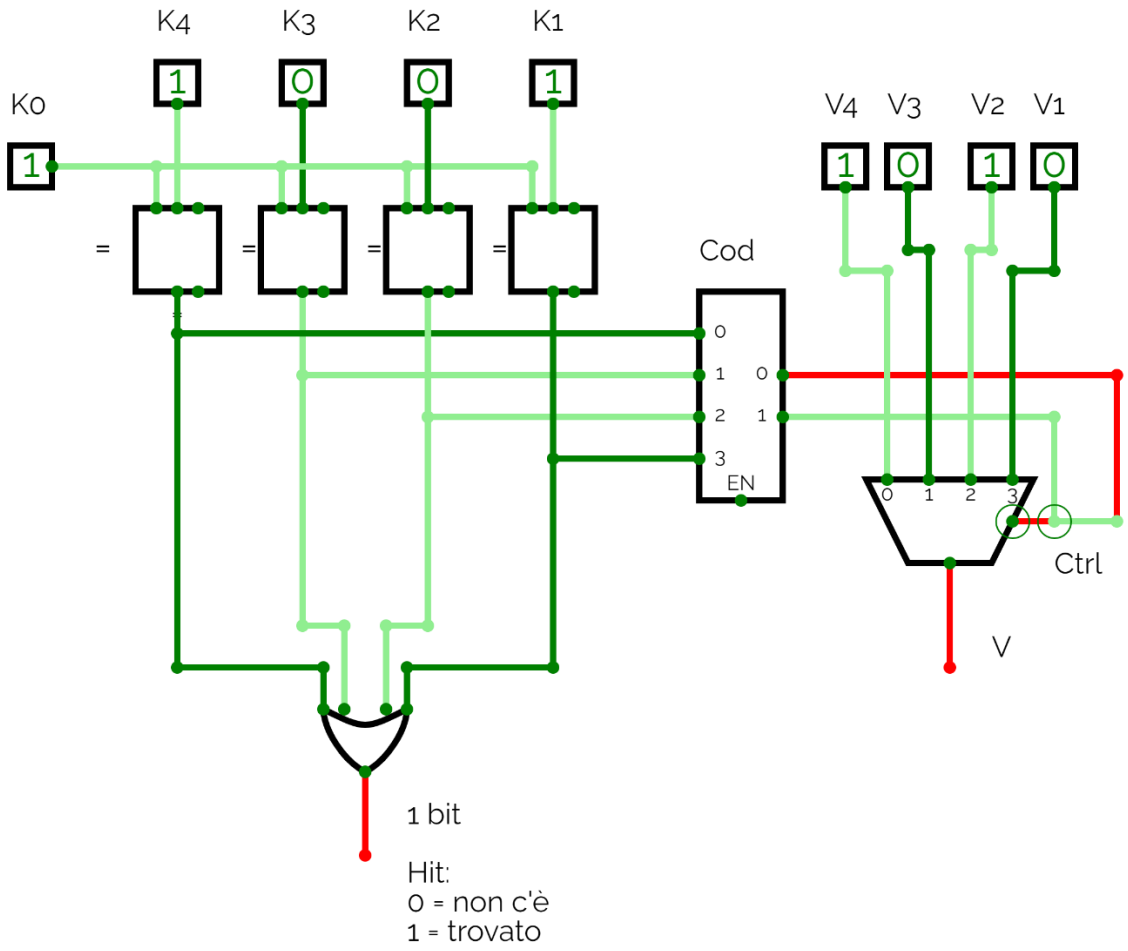
2)

```
module comp4(output z, input[3:0]x, input[3:0]y);
    reg ris;
    initial begin
        always@(x, y)
            begin
                integer i;
                for(i=0; i <4; i=i+1)
                    ris=ris&&(~(x[i]^y[i]));
            end
    end
endmodule
```

3)

```
module comp4(output z, input[3:0]x, input[3:0]y);
    assign
        z=(x==y ? 1'b1 : 1'b0);
endmodule
```

Circuito logico



Memorie

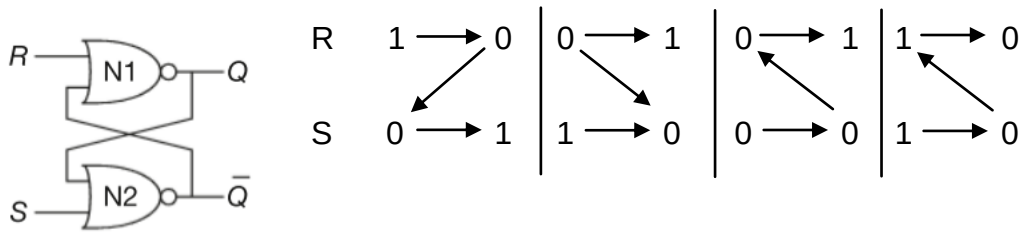
Dobbiamo essere in grado di memorizzare degli stati e gestire delle memorie.

Latch SR

Il latch SR rappresenta una delle reti sequenziali più semplici ed è composto da due porte NOR collegate a croce, il latch ha due ingressi, S e R, e due uscite, Q e $\bar{Q}$.

Il suo stato può essere controllato mediante gli ingressi, S e R, che attivano (o “settano”, set) e disattivano (o “resettano”, reset) l’uscita Q.

Set e reset permettono di mantenere 1 bit di informazione con 2 segnali, set permette di metterlo a “ 1 “, reset a “ 0 “, se entrambi vanno a 0 si ha il bit memorizzato precedentemente.



Caso	S	R	Q	$\bar{Q}$
IV	0	0	Q_{prec}	$\bar{Q}_{prec}$
I	0	1	0	1
II	1	0	1	0
III	1	1	0	0

Quando entrambi i bit vanno a 0 si genera la configurazione precedente.

Vediamo in dettaglio le quattro possibili combinazioni di R e S:

• Caso I: R = 1, S = 0:

N1 vede almeno un ingresso VERO, R, quindi produce un'uscita FALSA su Q.

N2 vede sia Q sia S come FALSI, quindi produce un'uscita VERA su $\bar{Q}$.

• Caso II: R = 0, S = 1

N1 riceve come ingressi 0 e $\bar{Q}$. Dal momento che il valore di $\bar{Q}$ a questo punto è ancora sconosciuto, non è possibile determinare che valore assume Q.

N2 riceve almeno un ingresso VERO, S, quindi produce un'uscita FALSA su $\bar{Q}$.

A questo punto è possibile tornare a N1 e, sapendo che entrambi gli ingressi sono FALSI, calcolare il valore dell'uscita Q, che è VERO.

• Caso III: R = 1, S = 1

N1 e N2 vedono entrambi almeno un ingresso VERO (R oppure S), quindi entrambi producono un'uscita FALSA. Di conseguenza, sia Q sia $\bar{Q}$ sono FALSE.

• Caso IV: R = 0, S = 0

N1 riceve gli ingressi 0 e $\bar{Q}$. Dal momento che il valore di $\bar{Q}$ è sconosciuto, non è possibile determinare il valore dell'uscita.

N2 riceve a sua volta gli ingressi 0 e Q, ma visto che anche il valore di Q è sconosciuto, anche in questo caso non è possibile determinare il valore dell'uscita.

Il problema è lo stesso dei negatori collegati a croce.
Si sa che il valore di Q deve essere 0 oppure 1, quindi è possibile risolvere il problema indagando le conseguenze delle due possibilità.

• Caso IV(a): Q = 0

Dal momento che S e Q sono entrambe FALSE, N2 produce un'uscita VERA su $\bar{Q}$. A questo punto N1 riceve almeno un ingresso VERO, $\bar{Q}$, quindi la sua uscita Q è FALSA, come era stato presupposto.

• Caso IV(b): Q = 1

Se Q è VERA, N2 produce un'uscita FALSA su $\bar{Q}$.

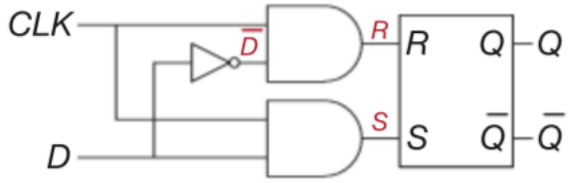
Ora N1 riceve due ingressi FALSI, R e $\bar{Q}$, quindi la sua uscita, Q, è VERA, come da presupposto.

Dobbiamo però evitare di perdere l'informazione se sia R che S sono 1.

Per risolvere questo problema introduciamo quindi il D-latch.

D-Latch

Questo latch ha due ingressi: un ingresso dati, D, che controlla il prossimo stato, e un ingresso clock, CLK, che controlla invece il momento del cambio di stato.



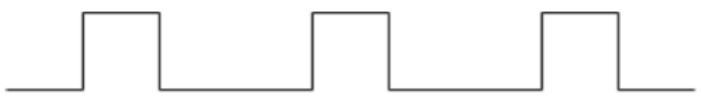
CLK	D	$\bar{D}$	S	R	Q	$\bar{Q}$
0	X	$\bar{X}$	0	0	Q_{prec}	$\bar{Q}_{prec}$
1	0	1	0	1	0	1
1	1	0	1	0	1	0

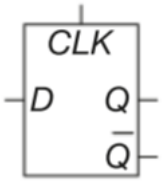
Quando CLK = 0, sia R sia S sono FALSI, indipendentemente dal valore assunto da D. Se invece CLK = 1, allora una porta AND produce un valore VERO e l'altra un valore FALSO, a seconda del valore di D.

Dati i valori di S e di R, è possibile determinare Q e $\bar{Q}$. Si osservi che, quando CLK = 0, Q ricorda il suo valore precedente, Q_{prec}, mentre quando CLK = 1, Q = D.

Il latch D è in grado di evitare il caso anomalo in cui gli ingressi S e R vengano attivati simultaneamente, questo è realizzato grazie al clock, se è basso non succede niente, ma se è altro grazie al not non potò avere due ingressi che sono " 1 ".

Quando CLK = 1, il latch è detto trasparente. I dati scorrono da D verso Q, come se il latch fosse un buffer. Quando invece CLK = 0, il latch è opaco: viene bloccato il passaggio dei dati verso Q, che mantiene il valore precedente.





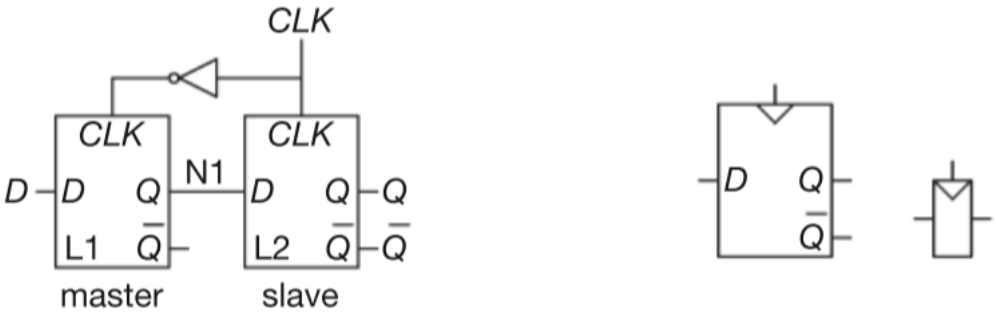
D = 1 S = 1 D = 0 R = 1

Clock = 1 R = 1 Clock = 0 Clock = 1 S = 0

Il problema di questa componente è che cambia continuamente il suo stato quando il clk è ad 1, noi vorremmo una componente che cambia solo in determinati momenti.

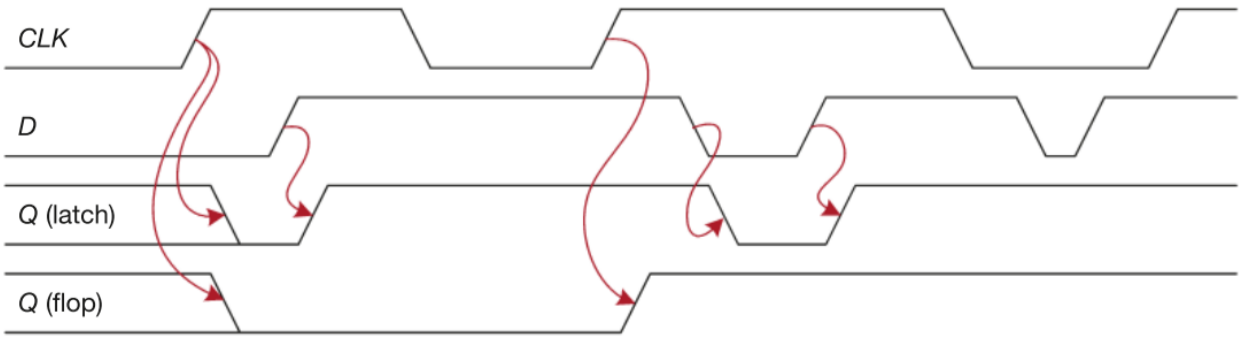
D flip-flop

Un flip-flop D può essere costruito a partire da due latch D in cascata controllati da due segnali di clock complementari. Il primo latch, L1, viene detto master, mentre il secondo latch, L2, viene detto slave. Quando CLK = 0, il latch master è trasparente, mentre il latch slave è opaco. Di conseguenza, qualsiasi valore di D viene portato a N1. Quando invece CLK = 1, il latch master diventa opaco e quello slave trasparente. In questo caso, il valore di N1 viene trasmesso a Q, ma N1 resta isolato da D. Quindi, qualunque sia il valore di D subito prima del fronte di salita (passaggio da 0 a 1) del clock, questo è il valore che viene trasferito a Q al momento di tale fronte. In tutti gli altri casi, Q mantiene il suo valore precedente, dal momento che c'è sempre un latch opaco che blocca il passaggio di dati tra D e Q.



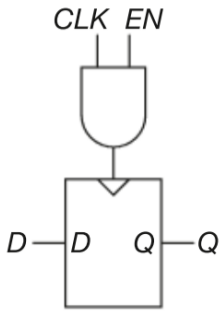
Essendo segnali elettrici dobbiamo considerare il tempo di propagazione, il passaggio da 0 ad 1 non è immediato.

In questa immagine mostriamo i segnali letti da un D Latch e da un D Flip-Flop a confronto, come possiamo vedere il D-Latch varia al variare di D (con un certo ritardo) quando il clk è ad 1, mentre il D Flip-Flop memorizza il segnale che avevamo durante il fronte di salita del clock e lo mantiene.



Abilitazione

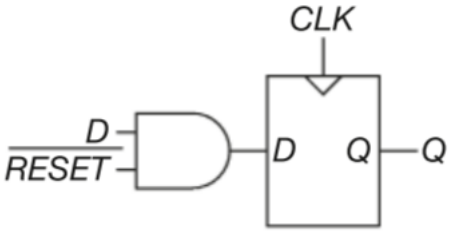
Possiamo modificare il flip-flop precedentemente mostrato aggiungendo un segnale che ci permette di abilitare o meno la scrittura.



Questo segnale viene implementato aggiungendo un altro ingresso, chiamato EN o ENABLE, per determinare se memorizzare o no il dato sul fronte del clock. Quando EN è VERO, il flip-flop con abilitazione reagisce come un normale flip-flop D; quando invece EN è FALSO, il flip-flop con abilitazione ignora il clock e mantiene il proprio stato.

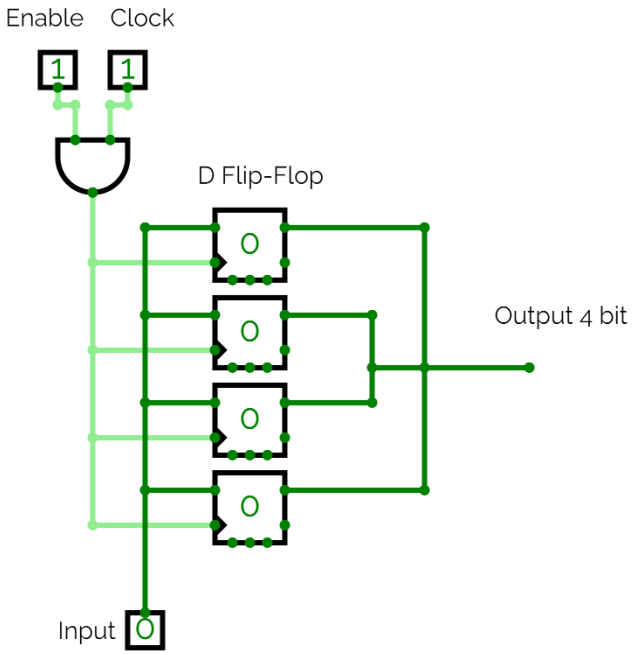
Reset

Un flip-flop resettabile aggiunge un altro ingresso, chiamato RESET. Quando l'ingresso RESET è FALSO, il flip-flop resettabile si comporta come un normale flip-flop D. Quando invece RESET è VERO, il flip-flop resettabile ignora D e, appunto, resetta l'uscita a 0. Questa tipologia di flip-flop è utile nel caso in cui si desideri forzare uno stato noto (cioè 0) in tutti i flip-flop della rete quando viene accesa.



Questi flip-flop possono essere resettabili in modo sincrono o asincrono. I flip-flop resettabili in modo sincrono si resettano solo al fronte di salita di CLK, i flip-flop resettabili in modo asincrono si resettano nel momento in cui RESET diventa VERO, indipendentemente dal valore assunto da CLK.

Registri

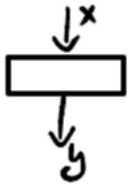


Se vogliamo implementare un sistema a N bit servono N D Flip-Flop che condividono un ingresso CLK comune, in modo che tutti i bit vengano aggiornati allo stesso tempo.

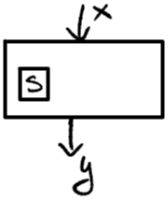
Reti logiche

Ci sono due tipi di reti logiche:

- Reti logiche combinatorie (funzionali)



- Reti logiche sequenziali (funzioni con stato)



In generale, le reti sequenziali includono tutte le reti che non sono combinatorie, cioè quelle la cui uscita non può essere determinata guardando semplicemente i valori presenti in quel momento agli ingressi.

Registri

In un registro il passaggio dell'informazione da 0 ad 1 o da 1 a 0 non è immediato, questo è dovuto a diversi fattori: Il primo è quello dei tempi di propagazione del segnale che non sono istantanei, come neanche il clock lo è, infatti non va alto o basso immediatamente, un altro fattore è il tempo necessario ad eseguire i calcoli logici, come abbiamo visto infatti ogni porta logica induce un ritardo nella rete combinatoria.

Il tempo che passa dall'inizio del fronte di salita (o di discesa) all'effettiva stabilizzazione del segnale (calcolo dell'uscita) viene chiamato tempo di propagazione.

Il tempo di contaminazione, invece, è il tempo necessario affinché un segnale in ingresso ad una porta logica si rifletta sulle altre porte logiche della rete, contaminandone il segnale.

Il segnale di ingresso deve essere stabile prima che il clock vada alto, se questo non succede possono verificarsi degli errori (i registri potrebbero leggere bit invertiti) o entrare in uno stato detto metastabile che sull'uscita non ha né 0 né 1.

Bisogna scrivere nei registri prima che il clock risalga.

Il ciclo di clock scandirà le nostre operazioni.

Circuito Sequenziale

Una rete sequenziale ha una serie finita di stati discreti $\{S_0, S_1, \ldots, S_{k-1}\}$. Una rete sequenziale sincrona ha un ingresso di clock i cui fronti di salita indicano una sequenza di istanti di tempo nei quali hanno luogo le transizioni di stato. Vengono spesso utilizzati i termini stato presente e stato prossimo per distinguere lo stato in cui il sistema si trova al momento attuale dallo stato in cui si porterà al prossimo fronte di clock.

Le regole di composizione delle reti sequenziali sincrone stabiliscono che una rete è una rete sequenziale sincrona se è formata da elementi circuitali interconnessi in modo tale che:

- ogni elemento della rete è o un registro o una rete combinatoria;
- deve essere presente necessariamente almeno un registro;
- tutti i registri ricevono lo stesso segnale di clock;
- ogni percorso ciclico contiene almeno un registro.

Esistono anche reti sequenziali asincrone, che talvolta sono necessarie, per esempio quando la comunicazione avviene tra due sistemi con segnali di clock differenti o quando si ricevono gli ingressi in momenti arbitrari.

Macchine a stati finiti o Automi

Le reti sequenziali sincrone possono essere rappresentate come macchina a stati finiti (FSM o Automi).

Un automa è composto da due blocchi di logica combinatoria, la logica di stato prossimo e la logica d'uscita, e da un registro che immagazzina lo stato.
A ogni fronte di salita del clock, la FSM avanza allo stato prossimo, definito in base agli ingressi e allo stato presente.

Esistono due classi generali di macchine a stati finiti, ognuna caratterizzata dalla propria specifica funzionale: nelle macchine alla Moore, le uscite dipendono esclusivamente dallo stato presente della macchina; nelle macchine alla Mealy, invece, le uscite dipendono sia dallo stato presente della macchina sia dagli ingressi attuali.

$$Mealy \begin{cases} z = F_z(S, x) \\ S' = F_s(S, x) \end{cases} \qquad Moore \begin{cases} z = F_z(S) \\ S' = F_s(S, x) \end{cases}$$

L'automa di Mealy è più rapido in quanto risponde in base allo stato e all'input, per dare la risposta in quello di Moore serve uno stato in più (questo non è sempre vero).

In generale, le macchine di tipo Moore richiedono più stati rispetto alle macchine di tipo Mealy solo quando è necessario che lo stato corrente produca un'uscita corretta per tutti gli input possibili. Al contrario, nelle macchine di tipo Mealy, è possibile utilizzare lo stato corrente e l'input corrente per produrre l'uscita corretta in meno stati.

Esempio

Vediamo l'esempio di un automa che legge le sequenze *aba* su un alfabeto $\{a, b, c\}$

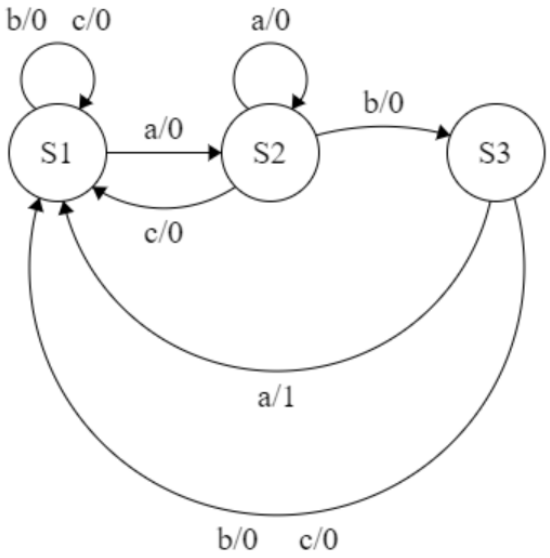
Input

⟶ *abcabbaba*

Output

_____|‾‾‾ L'uscita appena riconosciuta la stringa passa da 0 ad 1

Mealy

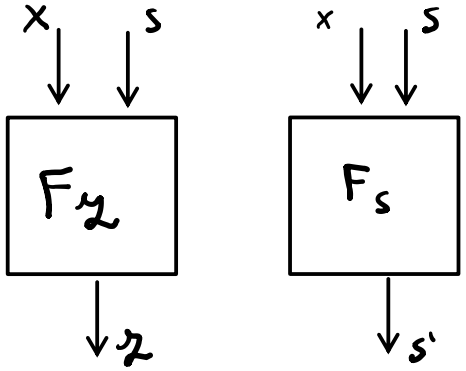


	a	b	c	a	b	b	a	b	a
S_1	S_2	S_3	S_1	S_2	S_3	S_1	S_2	S_3	S_1
0	0	0	0	0	0	0	0	0	1

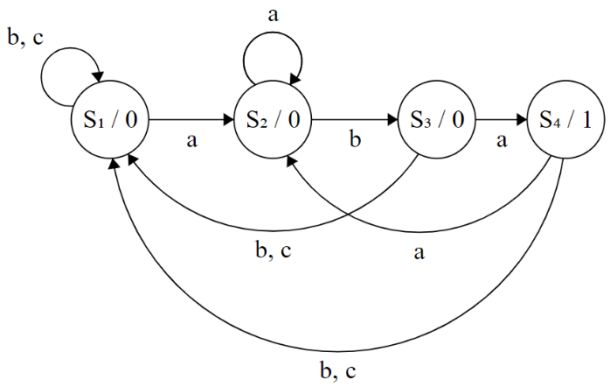
In mealy posso usare una rete combinatoria che prende l'ingresso e lo stato e mi restituisce le uscite.

Per lo stato dovremmo usare un registro da 2 bit.

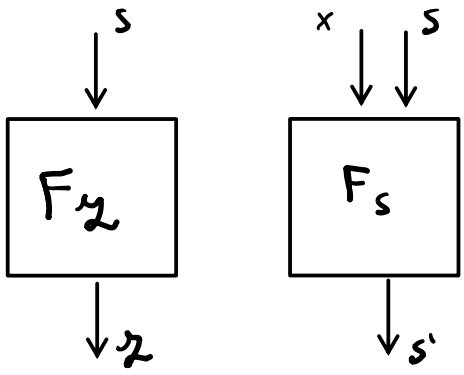
Sono due blocchi di logica combinatoria perché sono funzioni, dobbiamo ricordarci lo stato quindi si usa un registro da 2 bit, abbiamo infatti 3 stati possibili, quindi $\log_2 3 \sim 2$.



Moore

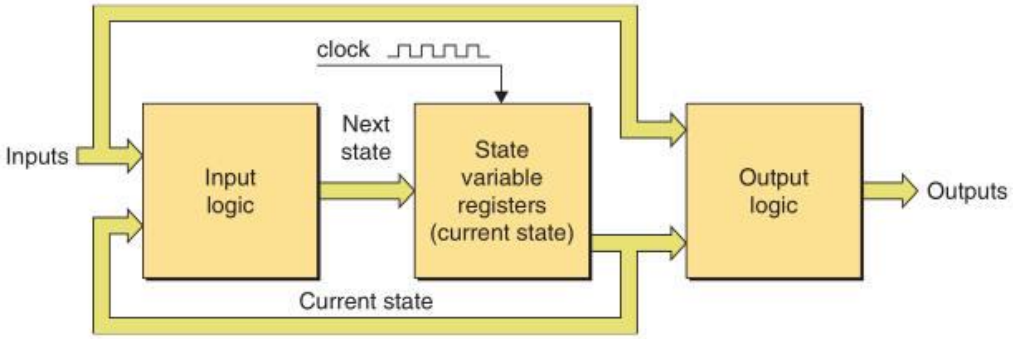


Nell'automa di Moore la funzione delle uscite ha solo lo stato corrente



Una volta che io ho “chiuso” il mio circuito in una scatola gli stati non mi interessano, il segnale di clock scandirà le operazioni.

Implementazione rete di mealy



b_1	b_0	
S_1 = 0	0	
S_2 = 0	1	
S_3 = 1	0	

x_1	x_0	
a = 0	0	
b = 0	1	
c = 1	0	

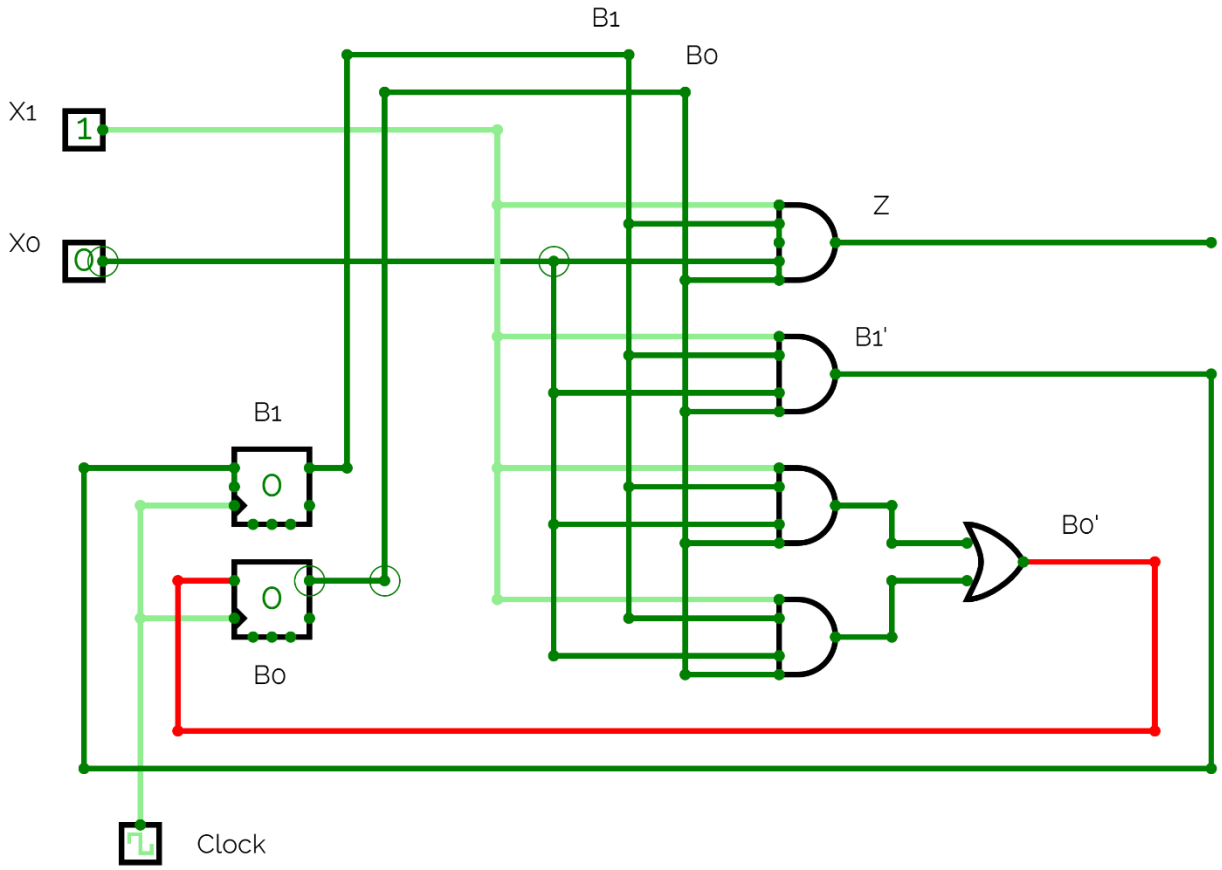
FS

b_1	b_0	x_1	x_0	b_1’	b_0’
0	0	0	0	0	1
		0	1	0	0
		1	0	0	0
0	1	0	0	0	1
		0	1	1	0
		1	0	0	0
1	0	0	0	0	0
		0	1	0	0
		1	0	0	0

$$b_1' = \overline{x_1} x_0 \overline{b_1} b_0$$
$$b_0' = \overline{x_1 x_0} \overline{b_1}$$
$$z = \overline{x_1 x_0} b_1 \overline{b_0}$$

FZ

b_1	b_0	x_1	x_0	z
1	0	0	0	1

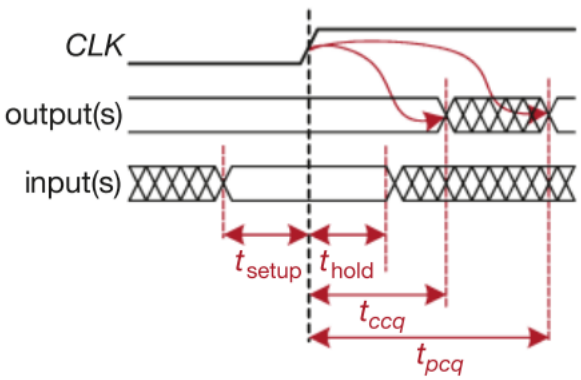


Ritardo reti sequenziali

Cosa accade se D sta cambiando nello stesso momento in cui il clock passa da 0 a 1?

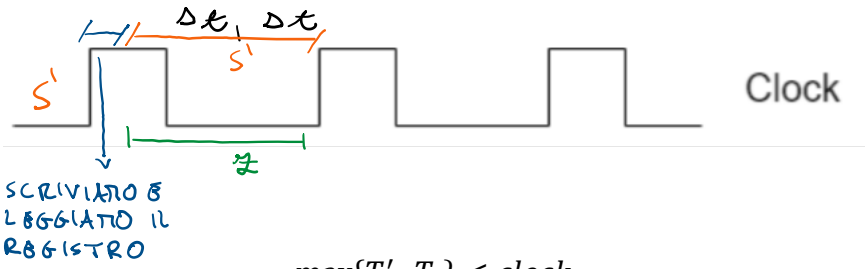
Un elemento sequenziale ha un tempo di apertura intorno al fronte del clock durante il quale l'ingresso deve essere stabile affinché il flip-flop produca un'uscita ben definita. Il tempo di apertura di un elemento sequenziale viene definito da un tempo di setup e da un tempo di hold, rispettivamente prima e dopo il fronte del clock.

Il periodo del clock deve essere abbastanza lungo da permettere a tutti i segnali di stabilizzarsi, il che costituisce un limite per la velocità del sistema. Nei sistemi reali, il clock solitamente non raggiunge tutti i flip-flop allo stesso tempo e questa differenza di tempo, detta sfasamento del clock, aumenta ulteriormente il periodo di clock necessario.



Quando il clock presenta il fronte di salita, l'uscita o le uscite iniziano a cambiare dopo il ritardo di contaminazione da clock a Q (chiamato t_{ccq}) e devono stabilizzarsi sul valore definitivo entro il ritardo di propagazione da clock a Q (chiamato t_{pcq}) Questi ritardi rappresentano rispettivamente il ritardo più rapido e il più lento di attraversamento della rete. Perché la rete interpreti in maniera corretta l'ingresso o gli ingressi, questi devono essersi stabilizzati almeno entro il tempo di setup (o tempo di attivazione) tsetup prima del fronte di salita del clock e devono rimanere stabili per la

durata almeno del tempo di hold (tempo di mantenimento) thold dopo il fronte di salita del clock. La somma del tempo di setup e del tempo di hold è detta tempo di apertura perché rappresenta il tempo totale durante il quale l'ingresso deve rimanere stabile.



$$\max\{T_s' , T_z\} < clock$$

Se s' fosse 2 livelli di porte ⇒ $2\Delta t$, dunque un ciclo di clock dovrebbe durare almeno $2\Delta t$ + il tempo di scrittura e lettura del registro, se il clock fosse minore:

Quando calcolo l'uscita?

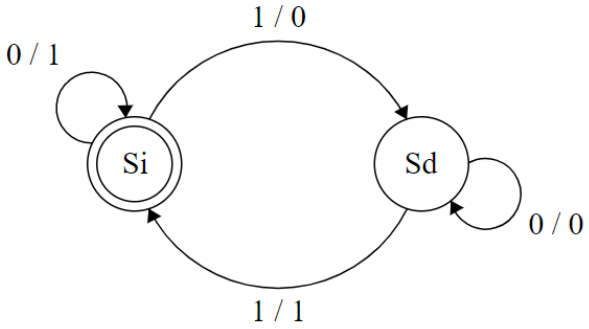
L'uscita deve essere calcolata prima

Rete che calcola la parità di una sequenza di bit

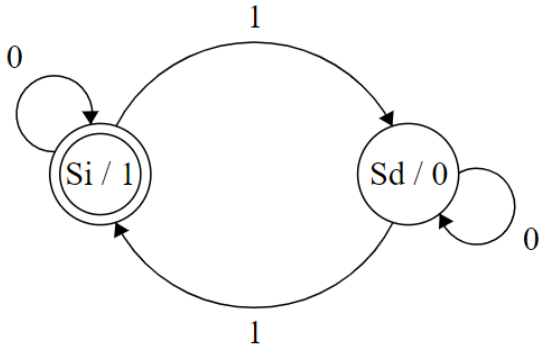
Input: 0 1 1 0 1 0 1 1 1 1 0 1 0 0 0

Z: 1 0 1 1 0 0 1 0 1 0 0 1 1 1 1

Mealy



Moore



L'ordine schematico dei passaggi da effettuare è il seguente:

1) Calcolare il numero di bit del registro di stato
2) Calcolare la tabella di verità di F_s' e F_z
3) Ottimizzazioni
4) Calcolare il ritardo delle reti logiche combinatorie
5) Definire la durata minima del ciclo di clock

Nell'esempio precedente abbiamo:

1) 1 bit

2) F_s'

S	x	S'
0	0	0
0	1	1
1	0	1
1	1	0

Fz

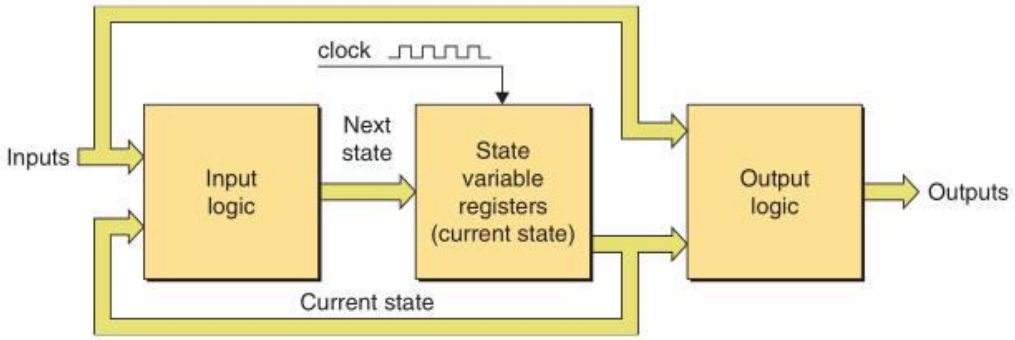
S	X	z
0	0	1
0	1	0
1	0	0
1	1	1

3) $S' = \bar{S}x + S\bar{x} \qquad z = Sx + \bar{S}\bar{x}$
4) $2\Delta t$
5) $r \geq \max\{2\Delta t, 2\Delta t\} + tempo\ lettura\ e\ scrittura\ del\ registro$

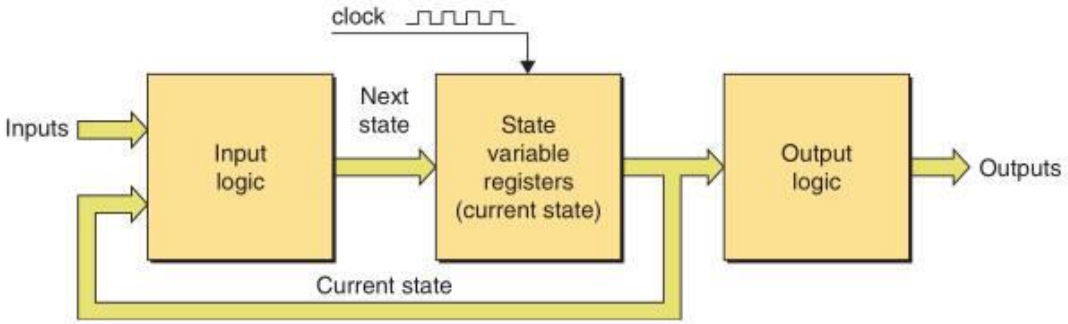
Implementare reti sequenziali in Verilog

Innanzitutto partiamo col descrivere l'automa, creiamo il circuito corrispondente, prendiamo in input gli ingressi, lo stato interno precedente e generiamo in un registro il successivo che andrà in un'altra rete combinatoria z che ci permetterà di calcolare l'uscita, se la rete è di mealy il la funzione delle uscite prende anche l'input come ingresso.

Rete di mealy

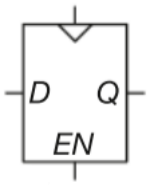


Rete di moore



In verilog le variabili si possono dichiarare come reg *nome.*

Reg sta per registro:



Il registro in verilog sarà un module con vari parametri, in output n bit e in input un segnale di enable, un input di clock e n bit.

Esempio di un registro in verilog

Attenzione questo codice ha un solo scopo illustrativo e non è stato testato.

```
module reg(output [n-1:0]out, input enable, input clock, input [n-1:0]in);
    parameter n = 8;
    reg[n-1:0]R;
    initial
        begin
            R = 0;
        end
    always @(posedge clock)
        begin
            if(enable == 1) R=in;
        end
    assign
        out = R;
endmodule
```

In tutto il modulo posso usare n come se fosse 8, possiamo usarla quindi sia per l'ingresso che per l'uscita.

Se dovessimo aver bisogno di un registro da 1 bit anziché n, il parametro nel module potremmo cambiarlo in fase di istanziamento del modulo.

Quando assegno R = 0, " 0 " viene inteso come decimale, dunque viene rappresentato come 0 indipendentemente dal numero di bit.

Durante la fase di salita del clock, quando l'input è stabile scriviamo il registro.

Il blocco Always mi permette di scrivere il registro quando il clock inizia a salire (posedge).

Per considerare anche il segnale di enable necessitiamo di un if.

Esempio in verilog del riconoscitore di stringhe

Vediamo l'implementazione in verilog dell'automa di mealy scritto in modo strutturale dell'esercizio sul risconoscere la sequenze aba in una data stringa di input.

```
// codifica stati e ingressi
// s1 00 s2 01 s3 10
// a  00 b  01 c  10

module stato(output [1:0] ns, input [1:0] x, input [1:0] s);
   assign
     ns[1] = ~x[1] &&  x[0] && ~s[1] &&  s[0];
   assign
     ns[0] = ~x[1] && ~x[0] && ~s[1];
endmodule // stato

module uscita(output z, input [1:0] x, input [1:0] s);
   assign
     z = ~x[1] && ~x[0] && s[1] && ~s[0];
endmodule // uscita

module registro(output [1:0]z, input  [1:0]inval, input clock);
   reg [1:0] stato;
   initial
     begin
  stato <= 2'b00;
     end
   always @(posedge clock)
     begin
        stato <= inval;
     end
   assign
     z = stato;
endmodule // registro

module fsm(output z, input [1:0] x, input clock);
   wire [1:0] rin;
   wire [1:0] rout;

   registro regst(rout, rin, clock);
   stato    fs   (rin,  x,   rout);
   uscita   zeta (z,    x,   rout);
endmodule // fsm
```

Il modulo stato modella la rete che calcola il nuovo stato.

Il modulo uscita calcola con i parametri l'uscita del modulo.

Il cavo rout si sdoppia va in stato e uscita.

Rappresentazione automa a stati finiti

Abbiamo due modi per rappresentare in verilog un automa a stati finiti:

Modello strutturale:

Si programmano come reti formate da 3 componenti:

Le due reti combinatore per il calcolo delle uscite e del prossimo stato interno e il componente registro di stato.

Modello Behavioural

Si programmano usando componenti quali: assegnamenti, if-then-else, blocchi always, generate o assign. Questo metodo è più simile alla programmazione classica che siamo abituati a conoscere.

Cosa serve per una MFS behavioural?

Servono 3 cose:

Un registro dello stato, un always e un assign

MFS strutturale del riconoscitore di stringhe

Uso un modulo stato dove ci sono il bit1 e il bit0 del nuovo stato.

Uso un modulo uscita che calcola z a partire dagli input dello stato.

Usiamo un registro che quando il clock sale legge l'input.

Nel modulo fsm si dichiara un registro, un modulo stato e un'uscita.

Se fosse una macchina di moore avremmo due moduli fsm e fsm1 entrambi scritti in modo bheavioural.

Dichiariamo stato s e nuovo stato ns, inizializzandoli a 0, cambieranno quando il clock andrà alto, se siamo nello stato 0 e vediamo una "a" andiamo nello stato 1, in tutti gli altri casi rimaniamo in 0.

Tramite dei case abbiamo tradotto le transizioni.

L'assegnamento è: se alla fine lo stato corrente è 1 allora do 1, 0 altrimenti.

L'assegnamento su una rete di mealy e più complicato in quanto deve controllare stato e ingresso, dunque serve una congiunzione di condizione su stato e ingressi.

Riconoscitore di stringhe Moore Strutturale

```
module fsm1(output z, input [1:0] x, input clock);
   reg [1:0] s;
   reg [1:0] ns;

   initial
     begin
    s <= 0;
    ns <= 0;
     end
```

```
   always @(posedge clock)
     s <= ns;

   always @(*)
     begin
    case (s)
      2'b00:
        begin
           case(x)
         2'b00: ns <=2'b01;
         default: ns<= 2'b00;
           endcase // case (x)
        end
      2'b01:
        begin
           case (x)
         2'b00: ns <= 2'b01;
         2'b01: ns <= 2'b10;
         default: ns <= 2'b00;
           endcase // case (x)
        end
      2'b10:
        begin
           case (x)
         2'b00: ns<=2'b11;
         default: ns<=2'b00;
           endcase // case x
        end
      default: ns<= 2'b00;
    endcase // case (s)
     end // always @ (*)

   assign
     z = (s == 2'b11 ? 1'b1 : 1'b0);

endmodule // fsm1

module fsm(output z, input [1:0] x, input clock);
   reg [1:0] rin;

   always @(posedge clock)
     rin = x;

   fsm1 afsm(z,rin,clock);

endmodule // fsm
```

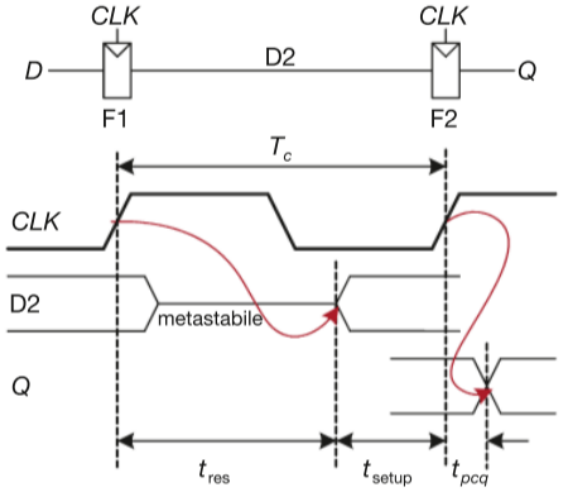
Sincronizzatori

Per garantire livelli logici corretti tutti gli ingressi asincroni dovrebbero essere fatti passare attraverso sincronizzatori.

Il sincronizzatore è un dispositivo che riceve un ingresso asincrono D e un clock, restituendo un valore.

Se D è stabile durante il tempo di apertura, Q assume lo stesso valore di D. Se invece D cambia durante il tempo di apertura, Q può assumere un valore ALTO o BASSO, ma non può assumere un valore metastabile.

È possibile costruire in modo semplice un sincronizzatore a partire da due flip flop



F1 campiona D al fronte di salita di CLK: se D cambia in quel momento, l'uscita D2 potrebbe momentaneamente assumere un valore metastabile. Se il periodo del clock è abbastanza lungo, D2 con elevata probabilità si stabilizza su un livello logico valido prima della fine del periodo. Successivamente F2 campiona D2, che a questo punto è stabile, producendo un'uscita accettabile Q. Si dice che un sincronizzatore fallisce se Q, l'uscita del sincronizzatore,diventa metastabile.

Questo può succedere se D2 non si stabilizza su un livello valido prima di essere campionato da F2 (e cioè se $t_{res} > t_c \ t_{setup}$).

Nelle reti sequenziali usiamo come sincronizzatore un registro che abbia come ingresso lo stesso clock del registro di stato e l'input. Questo registro speciale farà in modo che se anche l'input cambia durante il ciclo di clock la sua uscita rimarrà stabile dunque avremo un input stabile nella rete.

Quando useremo memoria e processore vi sarà un'interfaccia che regola entrambi, sia memoria che processore sono infatti macchine a stati finiti, per farli lavorare bene dobbiamo usare o lo stesso segnale di clock e far sì che le cose accadano nel momento giusto oppure usare un sincronizzatore.

Memorie

Un componente memoria dal punto di vista ideale si ricorda dei valori, ci possiamo immaginare una memoria come un vettore di parole (da x bit).

Abbiamo due operazioni, scrivi e leggi:

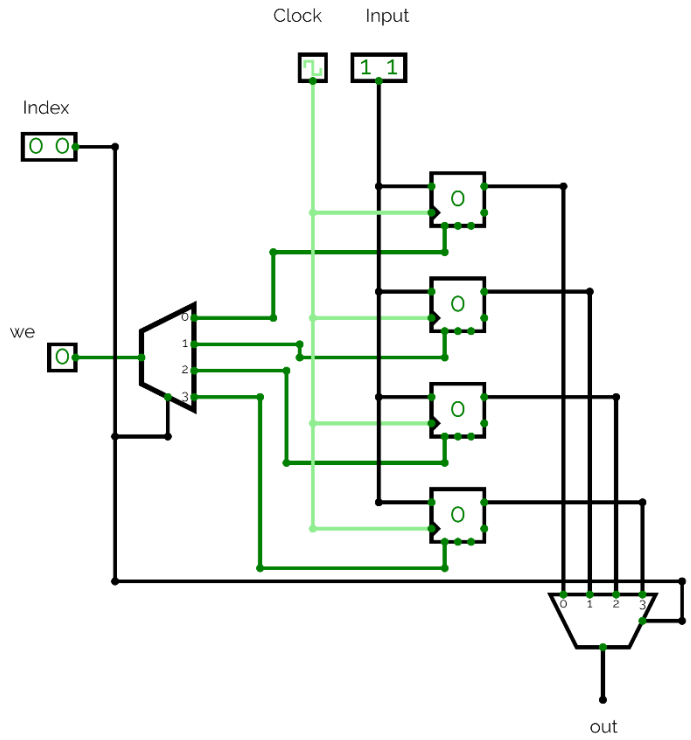
Write(l, x) Read(l) dove per l intendiamo la locazione e con x il dato da scrivere.

Es: Write(3, 123) Write(4, 456) read(4) → 456

La memoria conterrà i programmi e i dati su cui vogliamo lavorare.

Come si cotruisce una memoria?

Es: memoria con 4 posizioni da 4 bit

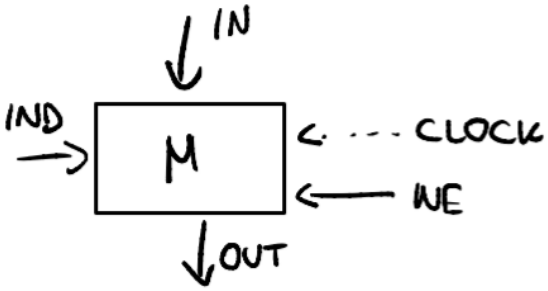


- Prendiamo 4 registri ognuno da 4 bit
- Dobbiamo poter avere un modo per leggere uno di questi registri, ogni registro ha un'uscita, usiamo quindi un multiplexer per sceglere l'uscita del registro
- Per scrivere un registro usiamo un segnale write-enable, ve ne sarà 1 per ogni registro, collegati ad un demultiplexer.
- Quello che voglio scrivere lo posso mandare a tutti tanto solo quello che ha il we = 1 scriverà.

Es: write(2, 12)

Mandiamo sull'ingresso in "12" (1100_2) questo va a finire su tutti gli ingressi dei registri, sull'entrata indirizzo mandiamo "2" (0010_2) che fungerà da we per il registro 2.

Abbiamo poi un out che vale sempre M[ind] l'indirizzo di scrittura funge da controllo per il multiplexer che è come se mi leggesse l'indirizzo appena scritto.



Siccome il costo del multiplexer è maggiore di quello del demultiplexer secondo questa implementazione leggere sarebbe più oneroso che scrivere, questa implementazione descritta è un modello teorico e nella pratica si usa poco.

La memoria in realtà la rappresentiamo come:

- Un ingresso che è un indirizzo
- Un demultiplexer
- Una griglia (wordLine e bitLine)

La memoria è organizzata come una matrice bidimensionale di celle di memoria.
A ogni accesso la memoria può leggere o scrivere il contenuto di una riga della matrice. Questa riga viene specificata da un indirizzo (address). Il valore letto o scritto nella memoria viene chiamato dato (data). Un componente con un numero N di bit di indirizzo e un numero M di bit di dato possiede 2^N righe e M colonne. Ogni riga di dati viene chiamata parola. Quindi tale componente contiene 2^N parole da M bit.

I componenti di memoria vengono realizzati come matrici di celle di bit, ognuna delle quali può contenere un bit di dato. Ogni cella di dato è connessa a una linea di parola e a una linea di bit. Per ogni configurazione dei bit di indirizzo, la memoria attiva una sola linea di parola che, a sua volta, attiva le celle di bit presenti nella riga corrispondente.
Quando la linea di parola è ALTA, il bit memorizzato viene inviato alla linea di bit o prelevato dalla stessa. Altrimenti, la linea di bit è disconnessa dalla cella di bit.
La circuiteria per l'immagazzinamento del bit varia a seconda del tipo di memoria.

Per leggere una cella di bit, la linea di bit viene inizialmente lasciata elettricamente fluttuante (Z). Viene quindi attivata la linea di parola, che permette al valore immagazzinato di forzare la linea di bit a 0 o a 1.

Per scrivere la cella di bit, invece, la linea di bit viene forzata in modo deciso al valore desiderato. Viene poi attivata la linea di parola, che collega quindi la linea di bit alla cella di bit in cui memorizzare il dato. La linea di bit forzata in modo deciso sovrasta il contenuto della cella di bit, scrivendo il valore voluto nel bit in questione.

Durante la lettura da memoria, la linea di parola è attiva e la riga corrispondente di celle di bit porta le linee di bit a un valore ALTO o BASSO. Durante la scrittura in memoria, invece, per prima cosa vengono portate le linee di bit a un valore ALTO o BASSO, e solo successivamente viene attivata la linea di parola, permettendo così ai valori delle linee di bit di essere immagazzinati in quella riga di celle di bit.

Le RAM (random access memory) sono fatte così, si chiamano random perché il costo di un lettura di ogni riga è lo stesso.

Esistono due tipi di ram:

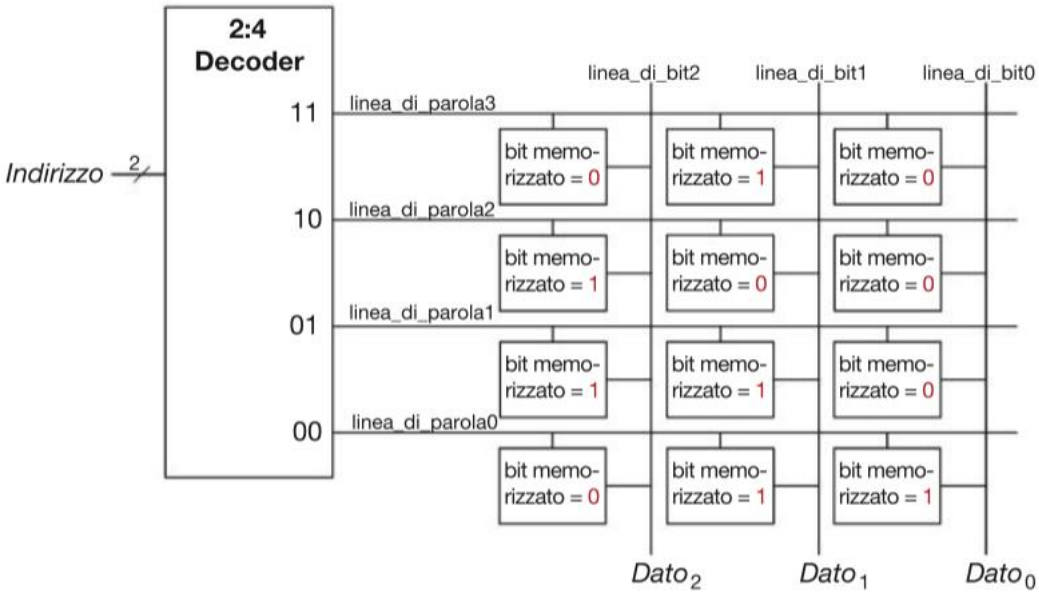
- Dinamiche: Per mantenere il bit nella griglia va "refreshato" sia periodicamente che dopo averlo letto, sennò viene perso, i dati vengono memorizzati come una carica su un condensatore.

- Statiche: I bit che andiamo a mettere nella griglia nelle ram statiche vengono mantenuti finché alimentati, i dati vengono memorizzati utilizzando una coppia di negatori collegati a croce.

La classificazione più generale distingue le memorie ad accesso casuale (RAM) dalle memorie a sola lettura (ROM). La RAM è una memoria volatile, cioè una memoria che perde traccia dei suoi dati una volta spenta. Invece, la ROM è una memoria non volatile, cioè una memoria che trattiene i suoi dati per un tempo indefinito, anche in assenza di alimentazione.

La RAM è chiamata memoria ad accesso casuale perché si può accedere a qualsiasi parola con lo stesso ritardo di qualsiasi altra. Al contrario, una memoria ad accesso sequenziale, come per esempio un registratore a nastro, accede ai dati più vicini più velocemente rispetto ai dati più lontani (ovvero ai dati che si trovano all'estremo opposto del nastro). La ROM viene invece chiamata memoria a sola lettura perché, storicamente, poteva essere esclusivamente letta, e non scritta. Questi nomi possono risultare ambigui

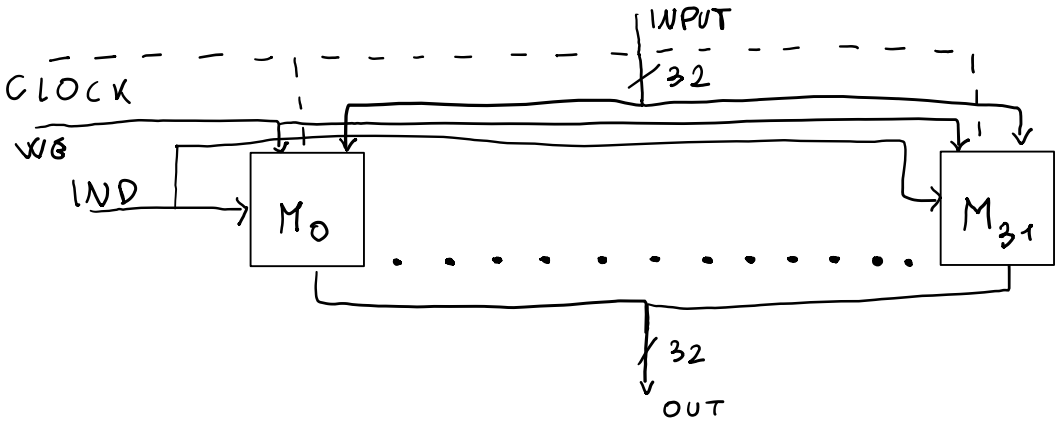
dal momento che anche le memorie ROM sono ad accesso casuale. Ancora peggio, la maggior parte delle ROM moderne possono essere sia lette sia scritte. Quindi, la distinzione importante da tenere a mente è che le memorie RAM sono volatili mentre le ROM non lo sono.



Nella ram dinamica mettiamo un transistor comandato dalla wordline e dalla gridline.

La RAM dinamica (DRAM, Dynamic RAM) memorizza un bit come presenza o assenza di carica in un condensatore. Il valore del bit viene memorizzato in un condensatore. Il transistore nMOS si comporta come un interruttore che connette o disconnette il condensatore dalla linea di bit. Quando la linea di parola è attiva, il transistore nMOS si accende e il valore del bit immagazzinato viene trasferito alla o dalla linea di bit. Quando il condensatore viene caricato, il bit immagazzinato è 1; quando invece viene scaricato, il bit immagazzinato è 0. Il terminale del condensatore è dinamico perché non viene portato a un valore ALTO o BASSO da un transistore tenuto a ALTO o BASSO. In caso di lettura, il valore del dato viene trasferito dal condensatore alla linea di bit.
Al contrario, in caso di scrittura, il valore del dato viene trasferito dalla linea di bit al condensatore. La lettura distrugge il valore del bit immagazzinato nel condensatore, quindi la parola di dato deve essere rinfrescata (riscritta) dopo ogni lettura. Anche quando la DRAM non viene letta è necessario ricaricarne i contenuti (cioè leggerli e riscriverli) ogni pochi millisecondi poiché la carica sul condensatore si perde gradualmente.

Nella ram statica è come se ci fosse un latch fatto da 6 transistor in grado di aprire o chiudere un interruttore messo tra la wordline e la bitline a seconda che ci sia un 1 o uno 0, non vi è più bisogno della carica se scrivo 1 l’interruttore si chiude.



Prendiamo 32 moduli dando a tutti lo stesso indirizzo, lo stesso we, lo stesso clock, e nel mio IND che è fatto da 32 bit do il primo bit al primo modulo, il secondo bit al secondo fino al 31-esimo che andrà all'ultimo modulo, i bit di uscita saranno singoli e dovrò farne un fascio da 32.

Sia una ram da 1G x 8 bit come facciamo a fare 2G da 8 bit?

Utilizziamo due moduli, il primo giga andrà nel primo modulo ed il secondo giga nel secondo, facciamo comandare poi un mux dal bit più significativo.

Usando ad esempio delle memorie da 8 posizioni per 1 bit ci posso implementare una funzione da 8 bit in 1 che ad ogni indirizzo scrive 0 o 1, una volta che la memoria mi ha salvato i risultati della T.V, quando avrò un indirizzo che corrisponde ad una riga della tabella di verità la memoria mi restituirà il risultato calcolato da quella funzione.

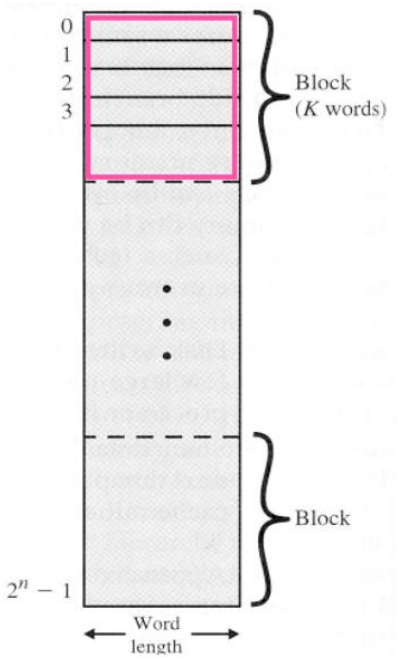
Memorie modulari

Supponendo che un modulo abbia una certa capacità, i bit dell'indirizzo saranno $\log_2 capacità + 1$, il bit aggiuntivo posso prenderlo come il più significativo o il meno e ci serve per scegliere il modulo da leggere grazie ad un multiplexer.

Ci sono due modi per raggruppare moduli di memoria:

- Modo sequenziale: (figua a dx)
- Modo interallacciato:

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17



Gli indirizzi dei moduli sequenziali sono "verticali" e vanno da 0 a n-1, n a 2n-1, 2n a 3n-1.

Vengono indirizzati come $id = {}^{ind}/_{C}$ dove id è l'indice del modulo, indirizzo è l'indirizzo e C la capacità del modulo, all'interno del modulo indirizziamo id nel modo seguente $ind_{modulo} = id \,\%\, C$

Gli indirizzi dei moduli interallacciati sono in fila l'un l'altro 0, 1, 2.

Vengono indirizzati come $id = ind \,\%\, m$ dove m è il numero di moduli, all'interno del modulo id salviamo ind all'indirizzo $ind_{modulo} = {ind}/{m}$.

La differenza in questi due utilizzi risiede nel come leggiamo parole consecutive.

Se sullo schema sequenziale voglio accedere a 2 parole consecutive, mi servono 2 tempi di memoria (ovvero due cicli di clock).

Sullo schema interallacciato invece se voglio accedere a 2 parole successive sono sempre una in un modulo e l'altra nel modulo successivo, dunque i due accessi possono essere fatti contemporaneamente nello stesso ciclo di clock.

Memoria Associativa

Finora abbiamo parlato di una memoria in cui vi è un indice e in quell'indice vi è una parola, in realtà molto spesso non vogliamo leggere un intero registro di indirizzi, bensì coppie di chiavi valore data la chiave.

Si usano due moduli di memoria, uno con le chiavi ed uno con i valori, nella posizione ind ho una chiave k_i e un valore v_i dal punto di vista logico vogliamo che passando una chiave come parametro questa memoria ci deve restituire il valore v con indirizzo ind t.c k_i = chiave (se esiste).

Se si vuole scrivere dobbiamo verificare che esista k_i = chiave e settare il valore di v_i = v.

Dal punto di vista logico sono realizzate usando tanti comparatori, a questi comparatori va la chiave che intendo cercare, i comparatori sono legati alle celle del modulo di memoria k, i comparatori restituiscono valori 0 o 1, tutti messi in or, che mi confermano l'esistenza della chiave cercata.

Prima dell'or c'è un codificatore legato al modulo dei valori che mi restituisce il valore cercato. Non posso però effettuare tutti i confronti contemporaneamente dunque il modulo k non è una memoria.

Leggi(key) → V con indice i t.c K_i = key

Scrivi(key, V) → sse $\exists\ k_i = key \Rightarrow v_i = v$

Forme di parallelismo

Anche per questa parte è consigliata la lettura del materiale didattico fornito dal professor Marco Danelutto disponibile al seguente link.

Molto spesso è necessario dover fare più cose contemporaneamente, questo ci permette di ridurre un po' il tempo rispetto ad eseguire i compiti in modo sequenziale.

Le istruzioni per essere eseguite parallelamente devono avere però delle determinate caratteristiche.

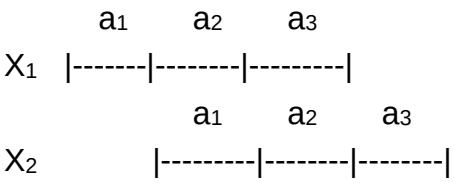
Le forme di parallelismo si dividono in:

- Temporale
- Spaziale

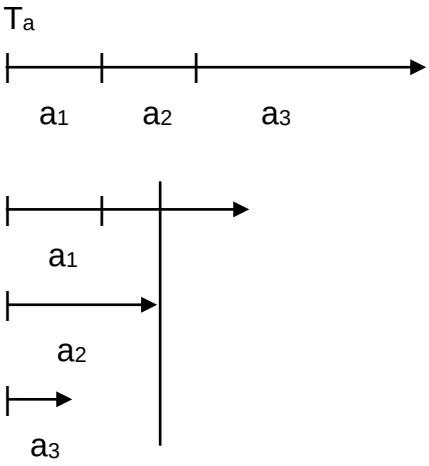
Per parallelismo *temporale* prenderemo in considerazione uno stream di dati (stream/flusso) e avremo l'idea che ad un tempo t_1 avremo un dato, ad un tempo t_2 un altro dato ecct...

Per il parallelismo *spaziale* anziché avere stream sui dati si parla di collezione sui dati che esistono in un momento unico.

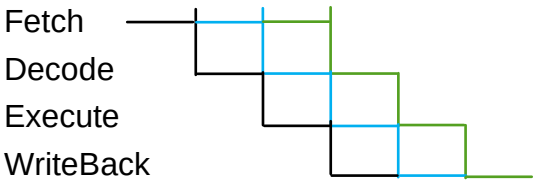
Parallelismo Temporale



Parallelismo Spaziale



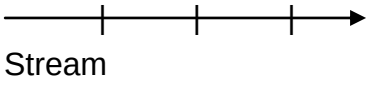
Parallelismo Pipeline per il ciclo di clock



Misure

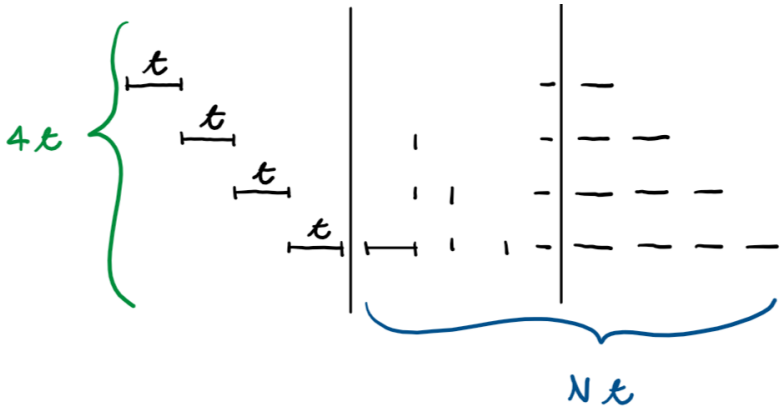
Aumentando il numero di elementi che lavorano dovrei ridurre il tempo di lavoro.

Se ho una catena di montaggio e secondo il modello stream monto delle cose al più posso ottenere un miglioramento di tempo pari a quante posizioni di lavoro ho.



Stream

$N \cdot 4t \longrightarrow 3t + Nt$ otteniamo un miglioramento pari a $^{N \cdot t}/_{N \cdot w}$



Misure primitive

Consideriamo tre misure:

La latenza: è per un singolo pezzo di lavoro il tempo che ci impiego (Δt)

Tempo di servizio: è ogni quanto riesco a finire un lavoro

La banda: è 1 / tempo di servizio

Misure derivate

Definiamo adesso due misure, lo speedup e la scalabilità.

Lo SpeedUp è una funzione del grado di parallelismo che usiamo ed è data dal tempo del miglior algoritmo sequenziale / tempo dell'algoritmo parallelo (n*w)

La scalabilità è il tempo che prendo con l'algoritmo parallelo / tempo parallelo con parallelismo nw

Latenza: $L = \Delta t \qquad \Delta t = t_1 - t_0$

Tempo di servizio: *variabile*

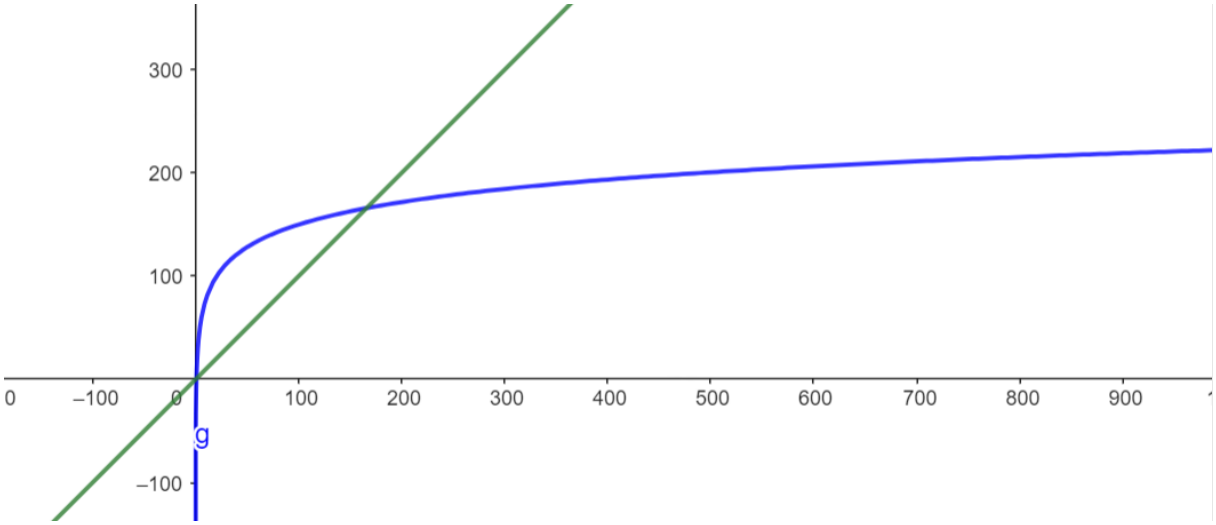
Banda: $^{1}/_{t_{serv}}$

SpeedUp: $Sp(nw) = {}^{T_{miglior\ seq}}/_{T_{par\ (nw)}}$

Scalabilità: $Sc(nw) = {}^{T_{par(1)}}/_{T_{par(nw)}}$

Efficienza $\varepsilon(nw) = {}^{Sp(nw)}/_{n}$

Vediamo adesso come si comporta lo SpeedUp all'aumentare del numero di worker.



La funzione g rappresenta lo speedUp, sull'asse delle y abbiamo i valori dello speedUp, sull'asse delle x il numero di worker.

Efficienza

L'efficienza è il rapporto tra il tempo di ideale n (c.a T_{seq} / T_{par} (nw)) e il tempo parallelo con nw gradi di parallelismo.

L'efficienza ci dice quanto riusciamo a sfruttare le cose che abbiamo, un'efficienza buona vuol dire che sfrutto le risorse che ho ottimamente, quando ho una risorsa la sfrutto al meglio.

$$\varepsilon(nw) = {T_{id}(nw)}/{T_{par}(nw)} \qquad T_{id}(nw) = {T_{seq}}/{nw}$$

$$\varepsilon(nw) = \frac{\frac{T_{seq}}{nw}}{T_{par}(nw)} = \frac{T_{seq}}{T_{par}(nw)} \cdot \frac{1}{nw} \qquad sp(nw) = \frac{T_{seq}}{T_{par}(nw)}$$

$$\varepsilon = \frac{Sp(nw)}{nw}$$

Come si comporta il pipeline?

Ogni processo ha un tempo t_i

La latenza è il tempo che impiego da quando inizio a processare a quando finisce, corrisponde alla sommatoria dei t_i.

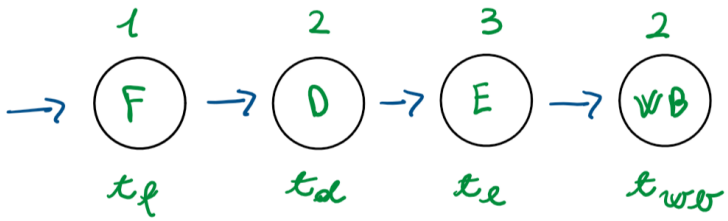
Il tempo di servizio dipende dalle unità di tempo dei t_i e corrisponde al $\max\{t_f, t_d, t_e, t_w\}$

Alla fine si allinea su quella che è l'unità di tempo più lunga (ovvero sul processo più lento)

Lo SpeedUp è T_S / T_{Par}

Diminuendo il numero di worker aumenta l'efficienza.

Lo SpeedUp diminuisce se la Pipeline ha stati sbilanciati (ovvero tempi per processo diversi).



$$L = t_f + t_d + t_e + t_{we} \qquad T_s = \max\{t_f, t_d, t_e, t_{wb}\}$$

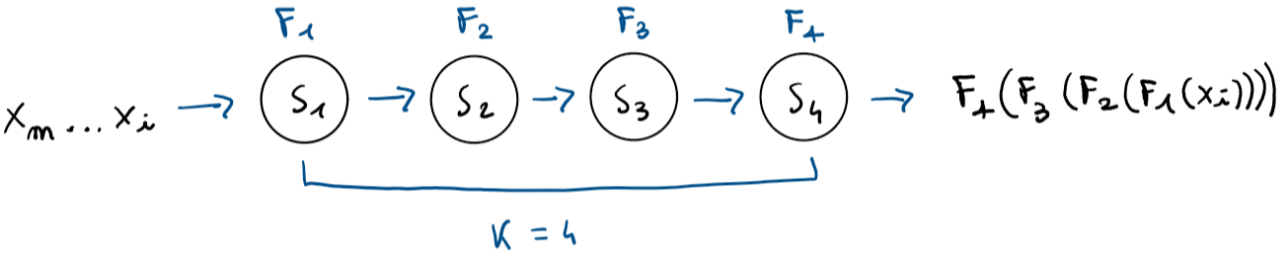
$$Sp(nw) = \frac{T_{seq}}{T_{par}(nw)} \qquad Sp(4) = \frac{8}{3}$$

Forme di parallelismo

Si possono utilizzare forme di parallelismo combinate.

Ogni stato calcola una funzione, ogni funzione ha una latenza t_1, t_2, t_3, t_4.

La latenza è il tempo che intercorre tra quando arriva un dato e quando lo rispediamo fuori (dopo averci applicato tutte le funzioni) la latenza è dunque la somma dei t_i degli stadi.



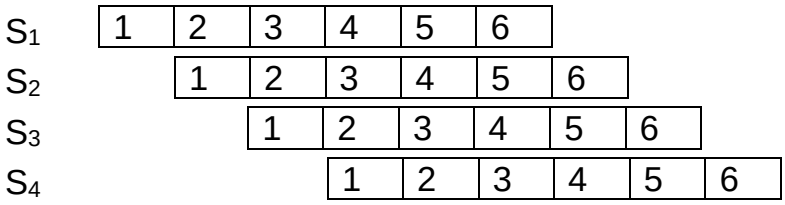
Le funzioni F_1, F_2, F_3, F_4 hanno rispettivamente tempi t_1, t_2, t_3, t_4.

$$L = \sum_{i=1}^{4} t_i$$

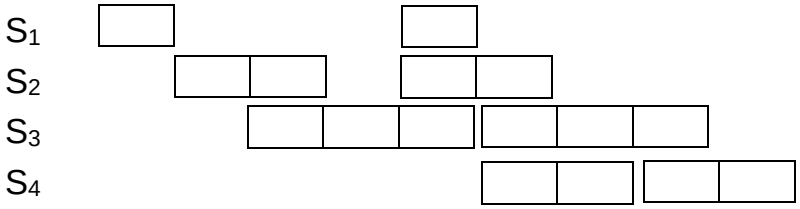
Un pipeline è fatto per lavorare su una sequenza di oggetti la cui elaborazione è totalmente indipendente gli uni dagli altri.

Cerchiamo di calcolare il tempo di completamento.

Possiamo osservare che spendiamo 1 tempo iniziale per riempire il pipeline (potrebbe essere una latenza) da lì in poi dobbiamo attendere che l'ultimo stato completi tutti i processi, in realtà dobbiamo solo attendere che lo stato più lento li completi tutti.



$$T_c = \sum t_i + (m-1)t_4$$



$$T_c = \sum t_i + (m-1)\max\{t_i\}$$

In entrambi i casi di solito m è molto più grande di k, quindi ad un certo punto alcune parti di questo conto diventano irrilevanti (come ad esempio L che dipende da m).

Se invece del tempo di completamento considerassimo quello di servizio, ovvero il tempo necessario per mandare in uscita il buffer attuale, o per accettare in ingresso il prossimo potremmo anch'esso approssimarlo con il tempo massimo dei vari stadi.

Il tempo di completamento lo potremmo approssimare come il numero di task moltiplicato per il tempo di servizio, questa è una misura tipica per le computazioni che operano su uno stream di dati.

Una delle cose che possiamo pensare come regola di refactoring è quella di raggruppare gli stadi quando il raggruppamento porti ad una riduzione del numero di entità necessarie al calcolo, senza portare ad un aumento del tempo di servizio.

Per calcolare lo speedUp dovremmo considerare che se cambiamo il numero di esecutori dovremmo dire che se il numero di esecutori è maggiore o uguale al numero degli stadi allora i valori sono quelli visti prima, lo speedIUp e l'efficienza sono i valori che deriviamo.

Se il numero di esecutori dovessere essere minore di k, non possiamo più derivare queste misure.

Lo speedUp per un certo grado di parallelismo lo abbiamo definito come il miglior tempo sequenziale fratto il tempo di parallelismo con nw gradi.

Se consideriamo il tempo di completamento potremmo andare a prendere $m*\max\{t_i\}$, dunque otteniamo una formula con sopra una sommatoria e sotto il max di quei valori.

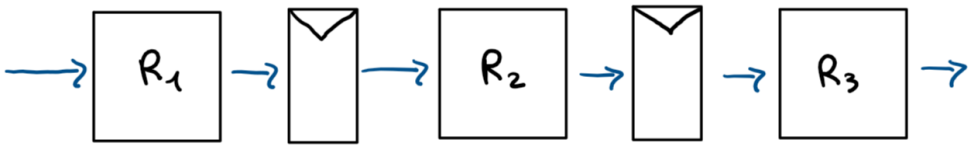
$$sp \begin{cases} n° \; esecutori \; \geq k \\ n° \; esecutori \; \leq k \end{cases} \qquad Sp(nw) = \frac{T_{seq}}{T_{par}(nw)} = \frac{m(\sum t_i)}{m(\max\{t_i\})} \qquad Sp(3) = \frac{3(\sum t_i)}{3(\max\{t_i\})}$$

$$2\bar{t} \quad 2\bar{t} \quad | \quad 3\bar{t} \quad 1\bar{t} \qquad nw = 2$$

$$2 \quad | \quad 2 \quad | \quad 4 \qquad nw = 3$$

$$4 \quad | \quad 3 \quad | \quad 1 \qquad nw = 3$$

Quello che dobbiamo fare per permettere l'esecuzione pipeline è spezzare l'esecuzione delle reti combinatorie inserendo dei registri tra l'una e l'altra.



Farm

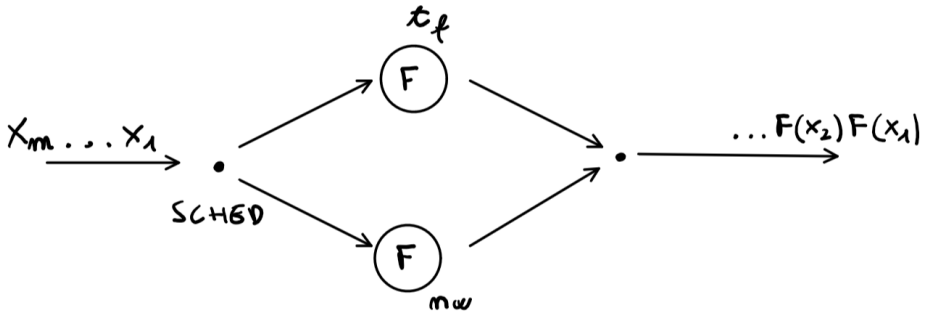
È la seconda forma di parallelismo, sia uno stream di dati $x_m \ldots x_1$, posso utilizzare uno schedulatore e una serie di oggetti che calcolano F e a round robin posso passare x_i all'i-esimo esecutore, x_{m+1} di nuovo al primo ecct…

Così che questi buttino fuori il risultato dell'esecuzione sul valore di input.

Fintanto che lo schedulatore riceve dati e riesce a buttarli fuori ad un worker nuovo, ogni t_f/t_c torno al primo worker.

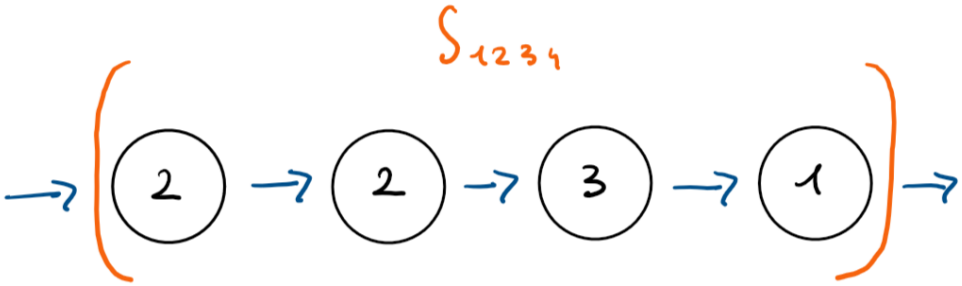
Possiamo vederlo come un pipeline di 3 stadi con rispettive latenze.

Lo SpeedUp ideale che viene fuori dipende dal numero di worker che ho messo, tutte le volte che abbiamo molte cose da calcolare possiamo applicare un farm e sperare di avere un tempo di speedUp ideale.



$$T_\varphi = \max\left\{T_{sched}, \; \frac{t_f}{nw}, \; t_{coll}, \; t_a\right\}$$

$$T_c = nwT_e + \frac{m}{nw} + t \approx \frac{m}{nw} t_f = m\left(\frac{t_f}{nw}\right) \qquad Sp = \frac{m\, t_f}{m\left(\frac{t_f}{nw}\right)} = nw$$



$$T_S = 1\bar{t} \quad Farm(S_{1234}\,,\ 8)\ T_c = m\bar{t}$$

$$T_s = \max\{2,2,3,1\}\ Pipe(S_1,\ S_2,\ S_3,\ S_4) \Longrightarrow Pipe(S_1,\ S_2,\ \ Farm(S_3,2),\ S_4)$$

$$T_s = \max\left\{2,\ \ 2,\ \ \frac{3}{2}\,,\ \ 1\right\}\ T_c = 2m\bar{t}$$

$$Pipe(Farm(S_1,2), Farm(s_2,2), Farm(S_3,3), S_4)$$

$$T_s = 1\bar{t} \quad T_c = m\bar{t}$$

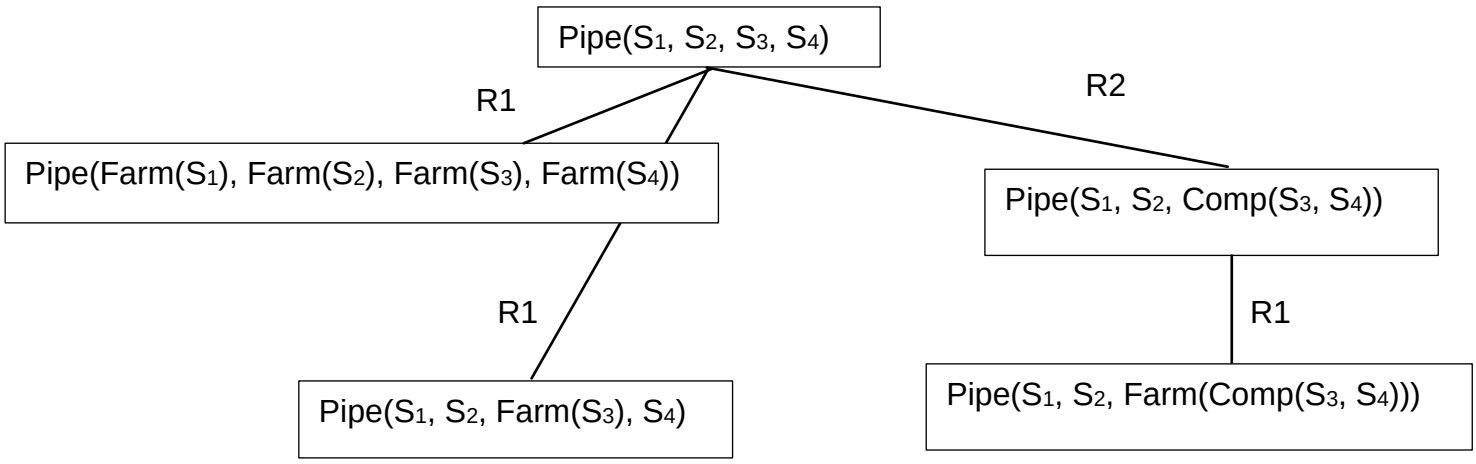
Se avessimo una grammatica che mi dice che un programma o è un sequenziale o è una farm o è un pipeline oppure una composizione sequenziale, potrei dire che data un'espressione posso usare delle regole di riscrittura che mi dicono che qualsiasi programma lo posso far diventare una farm di programmi, e che un pipeline di due programmi lo posso convertire in una composizione sequenziale.

$$Prog = Seq \mid Pipe(prog, prog) \mid Farm(prog) \mid Comp(prog, prog)$$

$Regole\ di\ riscrittura$:

$R1 : prog \ \leftrightarrow \ Farm(prog)$

$R2 : Pipe(prog, prog) \ \leftrightarrow Comp(Prog, prog)$

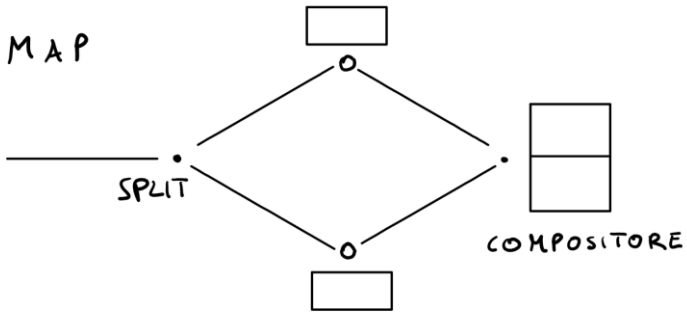


Map

La realizziamo con uno splitter ed un compositore.

Il tempo di servizio nella map non ha senso, consideriamo solo le latenze, queste sono date dal tempo per dividere la nostra struttura dati più il tempo che ho assegnato per lavorare sulla struttura.

N sarà il numero di elementi della collezione e t_f il tempo per calcolare un elemento.



$$m = Grandezza\ della\ cella$$
$$T_f = tempo\ per\ calcolare\ un\ elemento$$
$$L = t_f + T_{comp} + \left\lceil \frac{m}{nw} \right\rceil t_f$$

Reduce

È l'operazione che si usa per "sommare" una certa serie di numeri, si può infatti eseguire una somma ad albero, il costo (in numero di somme) per sommare n bit diventa $\log_2 n$.

Assembler

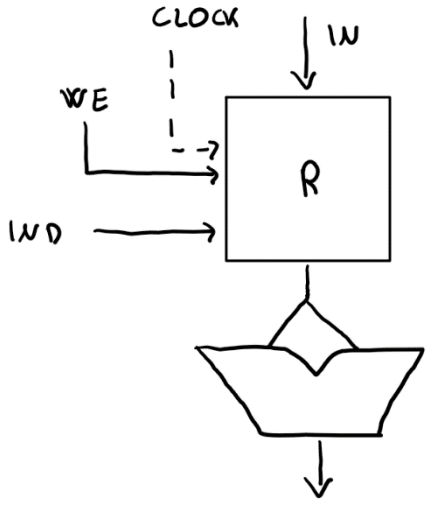
Assembler è un linguaggio di programmazione.

Il livello del linguaggio assembler mette a disposizione una serie di istruzioni che possono essere eseguite dal sistema operativo, le istruzioni andranno ad agire direttamente sulle componenti hardware.

Le istruzioni verrano convertite in binario, ovvero il linguaggio macchina, ciascuna istruzione può essere rappresentata da una parola in 32 bit.

Microarchitettura: è quella che ci fa vedere i componenti della macchina che ci permettono di eseguire le istruzioni.

Es: ADD R0, R1, R2



Architettura arm è un acronimo di acorn risc machine.

Arm era un'azienda che faceva calcolatori, il nome poi diventa advanced risc machine.

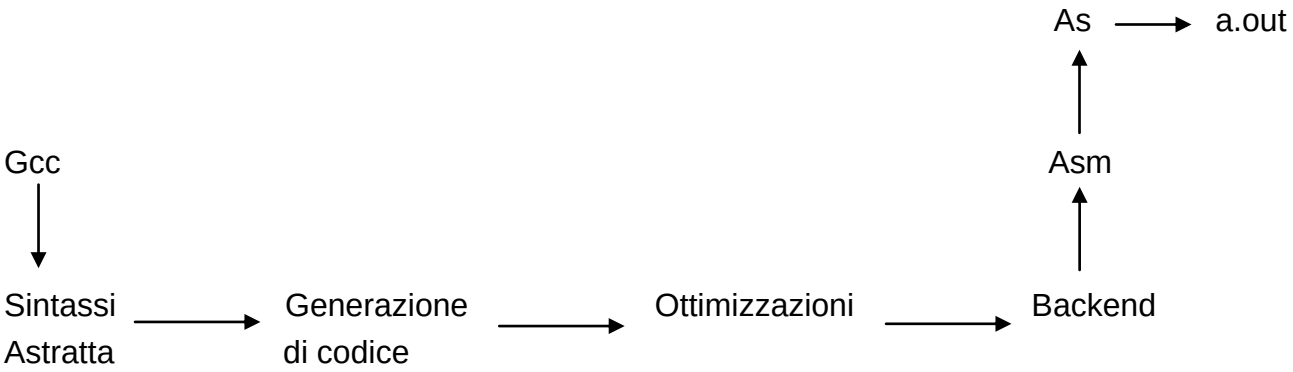
Tutto il linguaggio assembly si basa su istruzioni che eseguono operazioni e che accedono alla memoria.

Principi:

- Regolarità supporta semplicità, ovvero per tutte le istruzioni cerchiamo di dare la stessa struttura.
- Il caso comune deve essere veloce, ovvero le istruzioni più frequenti devono essere implementate in modo da essere eseguite il più velocemente possibile.
- Piccolo è bello, ovvero se posso dimensionare le risorse in modi diversi, ridimensionandole in piccolo andrò più veloce.
- Buon progetto se e solo se si hanno buoni componenti.

Assembly ha una sua sintassi e una sua semantica, normalmente compila in una cosa che il livello firmware (componenti) riescono a capire, ovvero in linguaggio macchina.

Armv7 prevede parole da 32 bit, in ogni parola ci sono dei pezzi che mi identificano l'istruzione.



L'assembler mi permette di eseguire le istruzioni del ciclo fetch-execute.

Il program counter restituisce l'indirizzo di memoria delle istruzioni da eseguire.

Abbiamo registri da 32 bit, in totale sono 16, gli ultimi 3 registri possono essere usati con degli alias:

- R13 Stack Pointer
- R14 Link Register
- R15 Program Counter

Tipi di istruzioni:

- Operative:

  Sono quelle che calcolano e che lavorano sempre sui registri.

  Add, Sub, And, Or, Cmp, Lsl (logical shift left), tutte queste istruzioni prendono due registri e scrivono il risultato in un nuovo registro.

- Memoria:

Sono quelle che operano sulla memoria:

Ldr (Load register) Prende due registri e un offset, a partire da un registro e dall'offset calcola un indirizzo di memoria e inserisce il valore della locazione ottenuta nell'altro registro.

Str (Store register) Prende due registri e un offset, a partire da un registro e dall'offset calcola un indirizzo di memoria e inserisce il valore che ho nella locazione di memoria trovata.

Ci sono due varianti che sono la Ldrb e la Strb con il significato che caricheranno non l'intera parola ma solo un byte all'indirizzo puntato.

- Salto:

Sono quelle che ci permettono di spostarci in punti diversi del programma:

B (branch) *etichetta*, se sto eseguendo un codice con tot istruzioni posso aggiungere un'etichetta seguita da altre istruzioni, la prima istruzione con branch che trovo salta all'etichetta selezionata. Il salto può essere in avanti o indietro.

Per fare questo salto assegniamo al program counter un offset che rappresenta la distanza in bit dall'etichetta.

Bl (branch and link) *etichetta*, è come la branch ma il pc+1 va a finire nel registro LR in modo che la procedura, per ritornare legge l'indirizzo in LR e lo sposta nel pc, in modo di ritornare al vecchio pc+1.

- Condizioni:

Le istruzioni condizionali ci permettono di modificare il flusso di esecuzione, possono essere effettuate sulle branch ma anche sulle operative, per verificare la condizione che testano utilizzano un insieme di bit flag.

Eq (==), Ne (!=), Lt (<), Gt (>)

Architettura arm

Abbiamo 16 registri da 32 bit ciascuno, noi li vedremo come un vettore di posizioni che vanno da R0 a R15, questi registri sono tutti uguali ma alcuni hanno un significato diverso.

I registri da R0 a R3 sono i registri temporanei, vengono utilizzati dal programma ma anche per le chiamate delle funzioni, questi vengono infatti utilizzati per i parametri della funzione.

Se i parametri dovessero essere più di 3 allora occorrerebbe usare lo stack.

In R0 verrà salvato il risultato dell'esecuzione della funzione.

I registri da R4 a R11 sono le saved variables e servono per le variabili permanenti del nostro codice.

R12 è un altro registro temporaneo.

R13 è lo stack pointer (SP)

R14 è il link register e conterrà l'indirizzo di ritorno della procedura (LR)

R15 mantiene l'indirizzo dell'istruzione (PC)

Quando facciamo le operazioni i numeri su cui eseguiamo le operazioni devono essere contenuti nei registri.

La memoria dal punto di vista di assembler è anch'essa un vettore di celle, che va da ind_0 a ind_{max}, con max = $2^{32} - 1$ (4 G).

Diversamente da altri calcolatori l'indirizzo che andiamo a riferire è al byte, cioè alla cella 0 non ci sono 32 bit ma ce ne sono 8, altri 8 all'indirizzo 1, altri 8 all'indirizzo 2 e così via, per cui se vogliamo leggere davvero un registro, dobbiamo prendere l'indirizzo ind, ind+1, ind+2, ind+3, prendere questi 4 byte e metterli nel registro, le operazioni che andranno a leggere o a memorizzare dati in memoria rispettano le convenzioni dell'arm.

La memoria gestita da arm è little endian ovvero il bit meno significativo si trova nella cella di memoria di indirizzo minore, questo significa che se prendo un registro che ha 4 byte non vengono messi in memoria il primo in posizione ind, il secondo ind+1 ma vengono invertiti.

I valori costanti, cosi dettì immediati, saranno semplicementi dei numeri preceduti da #.

Isa (Istruction Set Architecture)

Abbiamo già classificato le istruzioni in:

- Operative
- Di memoria
- Di salto

Vediamole adesso nel dettaglio.

Istruzioni operative (o Aritmetico logiche)

I tipi di operazioni che possiamo fare o sono aritmetiche o sono logiche.

Solitamente queste operazioni hanno 3 operandi:

L'operando di destinazione e due operandi sorgenti.

La destinazione diventerà il risultato dell'operazione tra source1 source2.

Le due sorgenti solitamente sono vincolate, la prima deve essere un registro, mentre la seconda può anche essere una costante.

Es: ADD R1, R2, R3 o ADD R1, R2, #1

Nome	Descrizione	Operazione
AND Rd, Rn, Src2	AND bit a bit	Rd ← Rn & Src2
EOR Rd, Rn, Src2	XOR bit a bit	Rd ← Rn ^ Src2
SUB Rd, Rn, Src2	Sottrazione	Rd ← Rn - Src2
RSB Rd, Rn, Src2	Sottrazione rovesciata	Rd ← Src2 - Rn
ADD Rd, Rn, Src2	Somma	Rd ← Rn+Src2
ADC Rd, Rn, Src2	Somma con riporto	Rd ← Rn+Src2+C
SBC Rd, Rn, Src2	Sottrazione con riporto	Rd ← Rn - Src2 - $\overline{C}$
RSC Rd, Rn, Src2	Somma con riporto rovesciata	Rd ← Src2 - Rn - $\overline{C}$
TST Rn, Src2	Controllo	Set flags based on Rn & Src2
TEQ Rn, Src2	Controllo di equivalenza	Set flags based on Rn ^ Src2
CMP Rn, Src2	Confronto	Set flags based on Rn - Src2
CMN Rn, Src2	Confronto con negativo	Set flags based on Rn+Src2
ORR Rd, Rn, Src2	OR bit a bit	Rd ← Rn | Src2

MUL Rd, Rn, Rm	Moltiplicazione	Rd ← Rn × Rm (low 32 bits)
MLA Rd, Rn, Rm, Ra	Moltiplicazione con accumulo	Rd ← (Rn × Rm)+Ra (low 32 bits)
UMULL Rd, Rn, Rm, Ra	Moltiplicazione di long senza segno	{Rd, Ra} ← Rn × Rm (all 64 bits, Rm/Rn unsigned)
UMLAL Rd, Rn, Rm, Ra	Moltiplicazione di long senza segno con accumulo	{Rd, Ra} ← (Rn × Rm)+{Rd, Ra} (all 64 bits, Rm/Rn unsigned)
SMULL Rd, Rn, Rm, Ra	Moltiplicazione di long con segno	{Rd, Ra} ← Rn × Rm (all 64 bits, Rm/Rn signed)
SMLAL Rd, Rn, Rm, Ra	Moltiplicazione di long con segno con accumulo	{Rd, Ra} ← (Rn × Rm)+{Rd, Ra} (all 64 bits, Rm/Rn signed)

Traslazioni:

MOV Rd, Src2	Copia	Rd ← Src2
LSL Rd, Rm, Rs/shamt5	Traslazione logica a sinistra	Rd ← Rm << Src2
LSR Rd, Rm, Rs/shamt5	Traslazione logica a destra	Rd ← Rm >> Src2
ASR Rd, Rm, Rs/shamt5	Traslazione aritmetica a destra	Rd ← Rm>>>Src2
RRX Rd, Rm, Rs/shamt5	Rotazione a destra con estensione	{Rd, C} ← {C, Rd}
ROR Rd, Rm, Rs/shamt5	Rotazione a destra	Rd ← Rn ror Src2
BIC Rd, Rn, Src2	Cancellazione bit a bit	Rd ← Rn & ~Src2
MVN Rd, Rn, Src2	NOT bit a bit	Rd ← ~Rn

Vediamo alcuni esempi di comportamento di queste istruzioni:

LSL R1, R2, R3 Prende il contenuto di R2 lo sposta a sinistra di tante posizioni quanto vale R3 e lo mette in R1.

Spostare a sinistra un valore significa moltiplicare il valore per (2^numero di spostamenti).

Spostare a destra un valore significa dividere il valore per (2^numero di spostamenti).

L'arithmetic shift right preserva il segno mettendo un 1 a sinistra ogni volta che shifto a destra.

ROR R1, R2, R3 Un ror prendere 3 registri e tutti i bit scorrono di R3 posizioni, quelli che escono da destra entrano a sinistra.

BIC R1, R2, R3 La bic cancella in R2 i che stanno ad 1 in R3 e mette il risultato in R1.

MOV R1, R2 Sposta il valore di R2 (che può anche essere un immediato) in R1.

Si ha un problema con la moltiplicazione, raddoppiamo infatti il numero di cifre necessarie per la rappresentazione, da 32 a 64.

MUL R1, R2, R3 Prende due numeri da 32 e genera un risultato da 32, in R3 NON può esserci una costante.

Per ovviare al problema della dimensione quello che fa la mul è di prendere i bit meno significativi e metterli in R1.

C'è un'altra versione MULL (dove la seconda l sta per long) dove davanti ha una S o una U che significano rispettivamente Signed e Unsigned.

MULL R1, R2, R3, R4 Prende 4 registri, e quello che fà è andare a mettere nella concatenazione di R1 e R2 i 64 bit della moltiplicazione tra R3 e R4.

I 32 bit meno significativi saranno memorizzati in R1, gli altri in R2.

MLA R1, R2, R3, R4 Moltiplica e somma i valori in un registro che contiene il risultato parziale inizializzato a 0. Anche questa operazione ha la versione Signed oppure Unsigned.

In questa versione di arm (armV7) non esiste la divisione, in altre versioni sì.

Flag

Arm prevede solo 4 tipi di flag rappresentati da bit (ovvero variabili booleane che rappresentano le proprietà del risultato di un'operazione).

Z	zero	vero *sse* ris == 0
N	Negative	vero *sse* ris < 0
C	Carry	Settato ad 1 se ho avuto un riporto
V	Overflow	

Abbiamo codici mnemonici per testare le condizioni sui flag.

Ad ognuno dei codici corrisponde un'espressione dell'algebra booleana sui flag appena definiti.

Eq (==), Ne (!=), Lt (<), Gt (>), Le(<=), Ge(>=)

Come vengono settati i flag?

Per settare i flag utilizziamo un'operazione che è la compare.

CMP R1, R2 è come se fosse una sub di cui buttiamo via il risultato, R1 – R2 → Flag Uguali sse il risultato è 0.

A tutte le istruzioni operative possiamo aggiungere una s, questo ci indica che vogliamo settare i flag.

Es: ADDS R1, R2, R3 equivale sempre a fare R1 = R2 + R3 ma setta i flag.

Possiamo utilizzare il flag insieme ad alcune istruzioni, modificandole nel seguente modo:

BEQ, BNE, BLT ecct…

ADDEQ (somma il contenuto se e solo se i flag rappresentano la condizione eq)

Esempio: Vediamo come scrive un if in assembler

Assumiamo x in R0 e y in R1

If(x == 0)	CMP R0, #0
y++	ADDEQ R1, R1, #1

Istruzioni di salto

Abbiamo la b (branch) e la bl (branch and link).

Le istruzioni in assembler sono una per riga e vengono eseguite sequenzialmente.

Le istruzioni in arm sono codificate con un formato da 4 byte (32 bit).

3 istruzioni sono 12 byte che dovrò sommare al program counter per dirgli dove andare a prendere la prossima istruzione.

Le istruzioni di salto a differenza di quelle operative e di memoria modificano il PC (program counter) sommandogli l'offset dell'istruzione puntata.

Esempio: Primo loop

```
loop:   mov r1, #1
        mov r2, #2
        add r3, r1, r2
        b loop
```

Quando arriviamo ad eseguire l'istruzione b loop, il programma farà la seguente cosa:

pc + offset → pc con offset che, dovendo tornare indietro di 3 istruzioni sarà -12 byte.

La bl mette l'indirizzo dell'istruzione successiva puntato dal pc nel link register (dunque pc+4) e poi salta all'istruzione puntata.

Bl loop pc + 16 → pc = pc + loop

(pc + 16 perchè assumiamo che pc punti alla prima istruzione ovvero alla prima mov, dunque sommiamo 4 per ogni istruzione successiva).

Se avessi una funzione che prende x e y come argomenti e li somma, i parametri vengono passati in R0 e R1, e il risultato dovrà essere in R0.

Spostiamo il link register nel program counter per far ritornare effettivamente la funzione, nel link register abbiamo l'indirizzo di ritorno che si trova nello stack di attivazione.

Spostandolo nel program counter ci sposteremo effettivamente a quell'istruzione.

```
add r2, r0, r1
mov r0, r2
mov pc, lr
```

Se ho z = fun 3, 2 per compilare questa chiamata dovrei compilare mettendo 3 e 2 nei posti dove si aspetta di trovare i parametri attuali, a quel punto dovrei chiamare la funzione e mettere il risultato nel registro che voglio.

```
mov r0, #3
mov r1, #2
bl fun
mov r4,r0
```

Istruzioni di memoria

Abbiamo visto che esistono due operazioni per operare su locazioni di memoria, la ldr e la str, che servono rispettivamente per andare a leggere e a scrivere in una locazione di memoria.

Se ho un vettore v[i] e voglio l'indirizzo successivo farò ind + qualcosa, dunque serve sapere dov'è l'inizio (i) più l'offset che sarebbe la dimensione.

Esempio:

Se volessi fare un'operazione del genere in assembler si tradurrebbe in:

M[R[b] + R[i]] → R[dest] Ldr R_{dest} , [R_b , R_i]

(dobbiamo moltiplicare l'offset per 4, in quanto la memoria è indirizzata al byte)

In questo esempio R_b sta per base e R_i sta per indice.

A questo esempio ci possono essere delle varianti, ad esempio al posto del registro R_i ci possono essere degli immediati.

Dobbiamo stare attenti a cosa stiamo indicizzando, infatti se questo metodo di indicizzazione va bene per struct o stringhe, con i vettori di interi dobbiamo prestare attenzione, dobbiamo sempre considerare infatti il numero di byte con cui il dato viene memorizzato in memoria, nel caso di vettori di interi infatti il primo elemento si troverà in posizione 0, ma il secondo elemento non si troverà in posizione 1, bensì in posizione 1*4, in quanto 4 byte sono la dimensione allocata per la rappresentazione degli interi, sapendo questo è facile capire che l'elemento i-esimo di un vettore, a partire dalla base potrà essere recuperato con offset i*4.

Per fare questo nella nostra istruzione anziché scrivere R_i possiamo fare R_i , lsl #2 .

Lo shift logico a sinistra infatti moltiplica il contenuto in R_i per 4, la nostra istruzione diventa quindi: Ldr R_{dest} , [R_b , R_i , lsl #2]

Ci sono diverse combinazioni possibili per accedere alla memoria, ne vediamo 3:

Ldr R_{dest} , [R] Ovvero in questo caso accediamo direttamente con base *dest* e offset R.

Ldr R_{dest} , [R_i , R_j] ! equivale a $R_{dest} = M[R_i + R_j]$; $R_i = R_i + R_j$

Questa operazione vuol dire che come offset uso la somma di questi due registri e la somma la calcolo e la mantengo in R_i . (pre incremento)

Ldr R_{dest} , [R_i] , R_j equivale a $R_{dest} = M[R_i]$; $R_i = R_i + R_j$

(post incremento)

Per quanto riguarda le store invece:

Str R_1, [R_2, R_3] equivale a M[R[2] + R[3]] = R[1]

Compilazione comandi C in Assembler

If-then

In C:

```
if(cond){
    then
}
```

In Assembler:

```
        //condizione
        bne cont
        //ramo then
cont:   //altre istr
```

If-then-else

In C:

```
if (cond) { Is_then }
else { Is_else }
```

In Assembler:

```
        //condizione
        bne else
        //ramo Is_then
else:   //ramo Is_else
```

For loop

Il for può essere che non si esegua mai quindi la condizione deve essere verificata prima. Assumiamo che *i* sia contenuto in R0 e *n* in R1

In C:

```
for (i = 0; i < n; i++){
    //istr
}
```

In Assembler:

```
        mov r0, #0
loop:   cmp r0, r1
        bge fine
        //istr
        add r0, r0, #1
        b loop
fine:   //finefor
```

While loop

In C:

```
while(cond){
    //istruzioni
}
```

In Assembler:

```
while:  //condizione
        bne cont
        //corpo del while
        b while
cont:   //altre istr
```

Do-while loop

In C:

```
do{
    //istruzioni
}while(cond);
```

In Assembler:

```
while:  //corpo del while
        //condizioni
        beq while
```

Chiamata di funzione

In C:

```
f(arg1, arg2);
```

In Assembler:

```
        mov r0, arg1
        mov r1, arg2
        push{//eventuali altri arg}
        bl f
//in R0 troviamo il valore restituito
```

Dobbiamo ricordarci che possiamo fare la mov di massimo 4 argomenti in quanto i registri temporanei sono da R0-R3, se necessitiamo di ulteriori argomenti dovremo pusharli sullo stack.

All'inizio del codice di ogni funzione dobbiamo salvare sullo stack i registri non temporanei (>= 4) che andremo ad utilizzare, questo perché sono solo i registri temporanei R0-R3 che si resettano per ogni record di attivazione, gli altri andranno salvati con un push e poi ripristinati con una pop prima di ritornare.

Esempio: *Come si verifica se un numero è pari?*

Basta verificare l'ultimo bit, se è a 0 sono tutte somme di potenze di 2 dunque numeri pari.

Come facciamo a controllare l'ultimo bit?

Per farlo si usa uno shift a sinistra così che l'ultimo bit (quello a destra) arrivi in prima posizione (a sinistra) e in tutte le altre posizioni abbiamo 0.

Possiamo poi fare una compare con #0, se il bit era 0 infatti tutti i bit saranno a 0.

Vediamo come usarlo in un if per incrementare un contatore che conta se il numero è pari:

In questo caso in R0 abbiamo il numero che vogliamo controllare e in R1 il contatore.

```
        lsl r0, r0, #31
        cmp r0, #0
        bne cont
        add R1, R1, #1
cont:   //altre istr
```

Come possiamo prendere esattamente un bit nel registro?

Possiamo usare una bitmask.

Se vogliamo isolare un bit all'interno del registro potremmo fare degli and con costanti che ci restituiscono 1 se i due valori sono uguali, 0 altrimenti.

Facendo un and con una bitmask che ha un " 1 " in una qualche posizione, ad es: i.
Tutti i bit del valore che stiamo confrontando con la bitmask che non si trovano in posizione i vengono cancellati, ovvero messi a 0. Nel risultato il valore del bit in posizione i, dipenderà dal valore in posizione i del numero iniziale, se questo era un 1 infatti ci sarà un 1, se era 0 ci sarà 0.

Questo è possibile in quanto l'operatore and esegue un and bit a bit.

Il codice per la parità precedente può essere quindi riscritto come:

```
        and r0, r0, #1
        cmp r0, #0
        bne cont
        add r1, r1, #1
cont:   //istruzioni
```

Potrei togliere il compare utilizzando l'istruzione operativa che setta i flag.

```
        ands r0, r0, #1
        bne cont
        add r1, r1, #1
cont:   //altre istr
```

Un'altra alternativa potrebbe essere quella di usare operatori condizionali:

```
ands r0, r0, #1
addeq r1, r1, #1
```

Esempio

Vogliamo contare le occorrenze di una x all'interno di un vettore.

Assumiamo di avere: x = R1, n = R2, count = R3, i = R4, V[] = R5

In C:

```
int count = 0;

for(int i = 0; i < n; i++){
    if(v[i] == x) count++;
}
```

In Assembler:

```
        mov r3, #0      //Inizializzo count
        mov r4, #0      //Inizializzo i
for:    cmp r4, r2      //Verifico la condizione del for
        beq endfor
        ldr r5, [r0, r4, lsl #2]   //leggo v[i]
        cmp r5, r1      //controllo che v[i] == x
        addeq r3, #1    //Se ho superato il controllo incremento il contatore
        add r4, r4, #1  //incremento la i
        b for           //salto all'inizio per verificare nuovamente la condizione
endfor: //fine
```

Cosa succederebbe se usassimo la ldr come: ldr r5, [r0], #4 ?

Il registro 5 va a prendere il valore in memoria indirizzato dal registro 0, e poi incrementa il registro 0 di 4.

Stringhe

In assembler possiamo manipolare anche stringhe di caratteri.

Possiamo memorizzarle nei registri R0 → str1 = “abcd” R1 → str2 = “qztf”

Come possiamo accedere ai singoli caratteri?

Utilizziamo una versione della load che è la load register byte.

Esempio: Scriviamo un programma che confronta due stringhe

Immaginiamo di avere la prima stringa in r0 e la seconda in r1.

```
loop:   ldrb r4, [r0]
        ldrb r5, [r1]
        cmp r4, r5
        blt //etichetta1        //str1 < str2
        beq loop
        b //etichetta2          //str1 > str2
```

In r4 (ugualmente in r5) vanno a finire 3 nibble e 1 gruppo da 8 bit (1 byte) che è il codice del carattere ASCII della prima stringa.

printf

Come possiamo fare per far scrivere qualcosa ai nostri programmi?

Vorremo fare una cosa tipo printf(“y vale %d\n”, y).

Per chiamare funzioni dobbiamo usare la branch and link, e i parametri dobbiamo metterli nei registri temporanei R0, R1, R2.

In questo caso la printf prende due parametri, la stringa di formattazione e il valore della variabile da stampare.

Si aspetta quindi il primo parametro in R0 e il secondo parametro in R1, la printf in C restituisce il numero di caratteri stampati.

Se volessi effetivamente chiamare una printf devo preparare la chiamata, è possibile farlo nel modo seguente:

Prima di finire il codice vogliamo fare la printf, per farlo dobbiamo prepararci i parametri, nel parametro R0 dobbiamo mettere l'indirizzo della stringa, che però dobbiamo definire da qualche parte, utilizziamo quindi una direttiva utilizzando la .data dove scriviamo il nosto dato che è la stringa, in questo caso la etichettiamo come:

str: .string "y vale %d\n"

Questa è una pseudo istruzione e ci permette di riservare da qualche parte un'area di memoria che sarà lunga 12 byte e che potremo recuperare con l'etichetta str, per metterlo in un registro possiamo fare una mov: mov R0, =str

Mettiamo poi in R1 l'altro parametro e facciamo la bl alla printf.

Possiamo chiamare la printf perché facendo gcc abbiamo a disposizione una parte della standard library del C.

Esempio di chiamata della printf:

```
        .text
        .global main

        .data
        str: .string "y vale %d\n"

main:   mov r0, =str
        mov r1, #7
        bl printf
        pop {lr}
        mov pc, lr
```

Compilazione

Per compilare un programma assembler abbiamo bisogno di un toolchange:
di compilatore, esecutore, debugger, object dump ecct…

Noi usiamo i tool della gnu, che sono disponibili sulla macchina virtuale @laboratorio2.di.unipi.it

Noi dobbiamo creare il nostro file assembler che solitamente ha suffisso .s, il file lo compiliamo con il comando gcc, in realtà il comando giusto da usare sarebbe as che sta per assembler, è il compilatore assembler che leggere il testo ASCII e produce un file binario che conterrà i dati per la microarchitettura. La maggior parte delle macchine non sono arm, usiamo quindi un toolchain cross-compiler, usiamo un programma scritto in x86 che compila per arm.

Per fare questo dobbiamo dire che sono istruzioni assembler, esistono delle pseudoistruzioni che iniziano per . e che vanno messe precedute da un tab, queste pseudoistruzioni si chiamano anche direttive.

Compilando il file.s otteniamo un a.out che potrà essere eseguito, per eseguirlo siccome siamo su una macchina x86 dovremmo usare un altro strumento che è il qemu, ovvero un emulatore che ci permetterà di eseguire l'a.out.

Per il debugging possiamo usare gdb, per usarlo dobbiamo fare dei passi particolari in quanto anch'esso è scritto per x86, quando compiliamo con gcc abbiamo bisogno di flag particolari, -ggdb3 e -static.

Per far partire il debugger mentre in C facciamo gdb a.out con eventuali parametri, qua serviranno due passi, dobbiamo far partire il sorgente con qemu e dei flag particolari e poi il debugger multiarch, questi si parleranno tramite un socket e ci permetteranno di fare debugging.

Utilizziamo una toolchain che si chiama:

arm-linux-gnueabihf-gcc file.s -static

Gnueabihf sta per gnu, abi è armv7, hf è hard floating point.

Questo comando ci genera un file a.out, che però non possiamo eseguire semplicemente con uno ./a.out, per poterlo eseguire dobbiamo utilizzare il comando:

qemu-arm a.out

Possiamo creare uno script bash per permettere una più facile compilazione/esecuzione.

Script.sh (NON TESTATO)

```
#
echo "Compiling $*"
arm-linux-gnueabihf-gcc -static -ggdb3 $*
qemu-arm -g 34788 a.out &
gdb-multiarch -q --nh -ex 'set architecture arm' -ex 'file a.out' -ex 'target
remote localhost:34788'
```

Questo script prima scrive che si sta compilando l'argomento che gli abbiamo passato, poi attiva il compilatore statico con il flag di debugger di quello che gli abbiamo passato da linea di comando e procede a creare il file a.out.

Qemu-arm esegue l'eseguibile con i flag che servono per fare il debugging.

Notare che per non avere conflitti (sulla macchina virtuale) conviene cambiare la porta.

Lanciamo poi il debugger multiarch con un po' di flag, il primo flag ci indica che lo stiamo lanciando per arg, il secondo gli diamo il nome del file, e alla fine c'è la porta a cui connettersi.

Lanciando questo script ci parte il debugger, possiamo chiedere di farci vedere le informazioni significative scrivendo:

tui reg gen

In questo modo possiamo avere una panoramica su tutti i registri utilizzati per poter controllare il loro cambiamento.

Usiamo poi gli stessi comandi del gdb, anziché dare run diamo cont.

Possiamo settare breakpoint con break, e andare avanti con next o step.

Direttive

Queste direttive non sono funzioni assembler, sono direttive al compilatore.

La text non ha parametri e serve ad indicare dove sono le istruzioni, si mette text e sotto le istruzioni scritte in ASCII, indica che quello che segue è il testo di un programma.

Se pensiamo alla memoria come vettore di celle, la direttiva text indica la porzioni di celle che contengono le istruzioni binarie ottenute dalla compilazione del programma.

La seconda che usiamo è .global *etichetta* per dire che quell'etichetta deve essere riconosciuta anche fuori, potremmo linkare quell'etichetta con altro codice, ci servirà per quando scriveremo funzioni in assembler che verranno utilizzate da programmi C.

La direttiva function serve per dichiarare il tipo di una certa etichetta in modo che possa essere chiamata da fuori.

L'area data corrisponderà ad un'altra area di memoria nella quale potremmo prendere gli indirizzi con delle etichette.

La direttiva .data ha altre direttive proprie:

La .word, .string, .fill

Queste sub-direttive ci permettono di riservare porzioni di memoria e riempirle con dei valori.

Ad esempio, se dopo la word mettiamo 4 numeri quell'oggeto definisce 4 celle di memoria con i rispettivi 4 valori.

Esempio:

.data

etcihetta: .word 1, 2, 3, 4

Nella .string ci mettiamo una stringa racchiusa tra le " ", qesta direttiva riserva un area di memoria grande tanti byte quanti sono i caratteri con uno \0 in fondo.

Nella .fill invece ci metto un numero e quella riserva un area di memoria che non dovrebbe essere inizializzata ma che normalmente il compilatore inizializza con degli zeri lunga quanto il valore che passo alla fill.

Stack

Vediamo adesso le operazioni push e pop e come operano sullo stack.

Push e pop sono delle pseudo istruzioni compilate con ldrm e strm dove m sta per mul.

L'operazione push permette di prendere un registro o una serie di registri che denotano indrizzi di memoria e salvarli sullo stack.

L'operazione pop esegue esattamente l'operazione contraria ovvero sposta dallo stack in un registro o in un insieme di registri degli indirizzi di memoria che erano stati precedentemente pushati.

L'ordine in cui i registri vengono caricati e scaricati è LIFO.

Dal punto di vista di come lo organizzo in memoria lo stack può crescere verso l'alto o verso il basso a seconda di dove è posizionato, si parla di stack ascending o stack descending.

Per lavorare sullo stack serve uno stack pointer, questo può puntare o alla prima posizione vuota o alla prima posizione piena.

Abbiamo 4 possibili combinazioni.

Nella ldrm e strm potremmo usare ulteriori due lettere che sono "fa" e "fd" che ci indicano quale tipo di stack vogliamo utilizzare.

Noi però non ce ne preoccuperemo in quanto useremo solo push e pop, l'unica cosa a cui bisogna prestare attenzione è la seguente:

Posso scrivere push{r1, r2, r4-r8, lr}

Possiamo mettere un numero qualunque di registri, indicandoli:

Per nome, per range, o per nome simbolico.

Il problema è come riusciamo a codificare un'istruzione del genere usando solo 32 bit?

Dobbiamo usare un po' di bit all'inizio per dire che questa istruzione è una push, e i rimanenti per rappresentare un numero arbitrario di registri, ciascuno dei quali dovrebbe, a regola, richiedere 4 bit.

Questo ovviamente non è possibile, l'implementazione è infatti diversa, usiamo una bitmap, ovvero 16 bit di cui il bit i-esimo se è 0 vuol dire che non prendiamo il registro i-esimo, se invece è 1 lo prendiamo.

Questo ha un'implicazione ovvero potrei fare una push{R1, R2} e una pop{R2, R1}

Questo non si può fare in quanto R1 e R2 o R2 e R1 sono la stessa bitmap, non viene tenuto conto dell'ordine in cui li scriviamo.

Possiamo però fare:

push{lr}

pop{pc}

In questo caso stiamo spostando il contenuto del link register sullo stack e poi lo stiamo togliendo per inserirlo nel program counter.

Esempio: Fattoriale

Fact(1) →1

Fact(n) →n * Fact(n – 1)

Quando chiamiamo la funzione n è l'unico parametro, ma in r0 ci serve anche n-1, e il risultato di Fact(n-1), per ricordarmi il valore precedente posso salvarlo sullo stack. Posso quindi fare la push{R0} prima della chiamata ricorsiva, poi una volta fatta facciamo una pop in {R1} e il calcolo MUL R1, R0 .

In realtà dobbiamo passare anche il link register in quanto stiamo facendo una chiamata ricorsiva per salvarci l'indirizzo di ritorno.

Dunque:

push{R0, lr}

pop{R1, lr}

mul R1, R0

Strutture Dati

Dobbiamo decide come memorizzare la struttura dati.

Ad esempio, se ho una matrice A[N][N] se dovessi prendere l'elemento i, j dovrei fare i*N+j per trovare l'indice della cella di memoria con il valore voluto.

Se avessi delle struct con dei campi potrei utilizzare come offset la sizeof del campo sommata alla base.

Questo immaginando che le strutture abbiano una base in memoria e i campi memorizzati sequenzialmente in modo contiguo.

Come si realizzano queste strutture?

Possiamo chiedere l'allocazione di memoria con la direttiva .data, oppure possiamo fare una chiamata di funzione a malloc.

Esercizio: Fattoriale ricorsivo

```
fact:   cmp r0, #1      //caso base
        moveq pc, lr    //se siamo nel caso base ritorniamo
        push{r0, lr}    //altrimenti salvo n e l'indirizzo di ritorno
        sub r0, r0, #1  //calcolo n-1
        bl fact         //faccio la chiamata ricorsiva
        pop{r1, lr}     //stiamo srotolando lo stack dei record di attivazione
        mul r0, r0, r1  //calcoliamo man a mano i risultati
        mov pc, lr      //ritorniamo
```

Parametri da linea di comando

In C abbiamo due parametri nel main, argc e argv, questi occupano rispettivamente R0 e R1, in R0 è contenuto il numero di elementi (n) e in argv è contenuto l'indirizzo ad un'area di memoria dove sono contenuti questi elementi.

Il primo parametro passato si troverà in argv[1] e sarà un puntatore ad un'area di memoria che conterrà i caratteri ASCII dell'argomento.

Esempio: Fibonacci

Scriviamo la funzione di fibonacci che accetta valori da riga di comando.

Carichiamo il puntatore al primo parametro in un registro con: ldr R0, [R1, #4]

Su R0 chiamiamo poi la atoi così da trasformare il numero da stringa a intero: bl atoi

Il codice è il seguente:

```
        .text
        .global main
        .data
        stringa: .string "Fibo(%d) = %d\n"

main:   push{lr}                //salvo l'indirizzo di ritorno
        ldr r0, [r1, #4]        //carico l'argomento
```

```
        bl atoi                 //casto l'argomento a intero
        push{r0}                //salvo il valore iniziale
        bl Fibo                 //faccio la chiamata alla funzione
        mov r2, r0              //sposto il valore calcolato in r2
        pop{r1}                 //scarico il valore iniziale in r1
        ldr r0, =stringa        //in r0 carico la stringa formattata
        bl printf               //chiamo la printf
        pop{pc}                 //ritorno

Fibo:   cmp r0, #1              //se siamo al caso base
        movle pc, lr            //restituisco direttamente r0
        push{lr}                //salvo l'indirizzo di ritorno
        sub r0, r0, #1          //calcolo n-1
        push{r0}                //salvo n-1
        bl Fibo                 //calcolo fib(n-1)
        pop{r1}                 //recupero l'ultimo n-1
        push{r0}                //salvo fib(n-1)
        sub r0, r1, #1          //calcolo n-2
        bl Fibo                 //calcolo fib(n-2)
        pop{r1}                 //recupero  fib(n-1)
        add r0, r0, r1          //calcolo fib(n-2)+fib(n-1)
        pop{pc}                 //ritorno al chiamante
```

Esempio: Liste

Usiamo un programma C che definisce la struct lista, vogliamo scrivere una funzione crea che a partire da un vettore crea una lista. Il main.c crea un vettore di oggetti e ce lo stampa facendoci vedere gli elementi uno dietro l'altro.

Utilizziamo una direttiva .type che serve per la toolchain cross compiler di arm.

Dobbiamo specificare che è una funzione con: *.type nomefun, %function*

```
    .text
    .global crea
    .type crea, %function

        @ r0 = vettore @ r1 = n  (lunghezza del vettore)
crea:   push {r4-r8,lr}
        mov r4, r0      @ &v[0]
        mov r5, r1      @ n
        mov r0, #8      @ sizeof int + sizeof void *
        bl malloc
        mov r6, r0      @ puntatore alla lista
        ldr r0, [r4]    @ v[0]
        str r0, [r6]    @ lista->valore
        mov r0, #0       @ assegniamo il valore NULL
        str r0, [r6, #4]    @ lista->next
        mov r7, #1      @ contatore del for
        mov r8, r6      @ prev
```

```
loop:   cmp r7, r5       @ se siamo arrivati ad n finiamo
        beq endfor       @ altrimenti facciamo gli stessi passi
        mov r0, #8
        bl malloc        @ r0 = nuovo
        ldr r1, [r4, r7,lsl #2] @ v[i]
        str r1, [r0]
        mov r2, #0
        str r2, [r0, #4]
        str r0, [r8,#4]     @ prev->next = nuovo
        mov r8, r0          @ prev = nuovo
        add r7, r7, #1      @ incrementiamo il contatore
        b loop

endfor: mov r0, r6
        pop {r4-r8,pc}
```

A questi due link sono disponibili altri due esempi di funzioni che implementano rispettivamente un albero e una hash map.

Funzione che visita un albero binario: File_tree

Funzione che calcola un hash function: File_Hash

Chiamate di sistema

Esistono due tipi di funzioni, le funzioni di libreria e le funzioni di sistema, fare una scanf o una getc sono chiamate di libreria, se noi facciamo una read(fd, buffe, n_byte) questa è una chiamata di sistema, cioè fa cambiare lo stato del processo da una modalità utente alla modalità kernel, in cui il processore agisce con poteri da superutente.

Per fare un SO il processore deve avere almeno due modi operativi, un modo user e un modo superutente, il passaggio tra questi due modi avviene tramite un'interruzione che però non è un interruzione generata dai dispositivi di I/O ma è un'interruzione software ovvero un'istruzione assembler che genera un'interruzione, il trattamento di questa interruzione si traduce in un'azione del so.

Questo passaggio avviene con un'istruzione particolare che nel nostro assembler si chiama svc e ha un parametro che è il tipo di interruzione che può generare, che per le chiamate di sistema è sempre 0.

Svc 0 vuol dire che stiamo invocando il sistema operativo, il sistema operativo come conseguenza dell'invocazione da parte della svc 0 fa una syscall il cui numero è nel registro R7, questa syscall ha dei parametri che sono r0 r1… (usa il solito schema di passaggio di parametri) e restituisce un valore in R0.

Ad esempio:

La read restituisce come valore il numero di byte letti, e il numero di byte letti dal descrittore fd sono trasferiti nel buffer.

Per i descrittori, lo 0 è lo stdin, l'1 è lo stdout, il 2 è lo stderr.

Potremmo fare ad esempio la funzione fibonacci in cui il numero anziché passarlo come argomento lo leggiamo da tastiera.

Esempio: Fibonacci con syscall

```
        .text
        .global main
        .data
        buff:   .fill 8192
        stringa: .string "Fibo(%d) = %d\n"
      strstr: .string "Il nummero immesso vale %d, in hex %x (era la stringa \"%s\")\n"

main:   push{lr}                //salviamo l'indirizzo di ritorno
        mov r0, #0              //selezioniamo il primo descrittore =stdin
        ldr r1, =buff           //salviamo l'indirizzo del buffer
        mov r2, #8192           //lunghezza del buffer
        mov r7, #3              //numero della syscall read è 3
        svc 0                   //faccio l'interruzione ed eseguo la syscall

        sub r0, r0, #1          //tolgo lo \n alla fine
        mov r1, #0              //e sostituire con \0
        ldr r2, =buff           //nel buffer appena letto
        str r1, [r2, r0]

        mov r0, r2
        bl atoi
        mov r1, r0
        mov r2, r0
        ldr r3, =buff
        ldr r0, =strstr
        push{r1}
        bl printf

        pop{r0}
        bl Fibo                 //faccio la chiamata alla funzione
        mov r1, r0              //sposto il valore calcolato in r2
        ldr r0, =stringa        //in r0 carico la stringa formattata
        bl printf               //chiamo la printf
        pop{pc}                 //ritorno

Fibo:   cmp r0, #1              //se siamo al caso base
        movle pc, lr            //restituisco direttamente r0
        push{lr}                //salvo l'indirizzo di ritorno
        sub r0, r0, #1          //calcolo n-1
        push{r0}                //salvo n-1
        bl Fibo                 //calcolo fib(n-1)
        pop{r1}                 //recupero l'ultimo n-1
        push{r0}                //salvo fib(n-1)
        sub r0, r1, #1          //calcolo n-2
        bl Fibo                 //calcolo fib(n-2)
        pop{r1}                 //recupero  fib(n-1)
        add r0, r0, r1          //calcolo fib(n-2)+fib(n-1)
        pop{pc}                 //ritorno al chiamante
```

Linguaggio Macchina

La macchina comprende istruzioni che hanno un formato su 32 bit.

Ci sono 3 classi di formato, un formato per le operative, uno per load e store e uno per le operazioni di salto.

Sono 3 formati diversi per come sono composte le operazioni.

Es: le operative hanno 1 o 2 sorgenti e una destinazione.

Le load e le store hanno un registro che viene letto o scritto e un indirizzo in cui si va a scrivere o leggere.

I 3 formati hanno una prima parte *identica* formata da due campi, un campo da 4 bit e un campo da 2 bit, il campo da due bit è **op**.

Op vale 00 se sono operative, 01 se sono ldr o str, e vale 10 se sono operazioni di salto, l'op 11 è riservata ad usi futuri.

Il campo op ci servirà per decidere cosa fare, per distinguere i 3 tipi di operazioni e usare circuiti diversi.

Se sono operative dobbiamo usare una ALU, se sono ldr/str ci servirà una memoria, se sono b/bl dovremo calcolare quale istruzione prendere anziché pc+4.

Gli altri 4 bit sono i bit **cond**, che vengono rappresentati come una lettera esadecimale, cond sarà una qualche configurazione di quei 4 bit che ci dice se quell'istruzione è un' istruzione condizionale e come dobbiamo valutare quella condizione, questo cond è la codifica di condizioni come LE, GE, EQ ecc…

Ci sarà una configurazione speciale *none* che corrisponde al nibble e_{16} ovvero 1110 e questo significa che l'istruzione non è un'istruzione condizionale.

Nelle operative ci sono altri campi in particolare c'è un campo da 6 bit che il **funct** che specifica ulteriormente quale tra le istruzioni operative dobbiamo fare, es ror, shift ecc…

Siccome le add possono essere addeq o adds, allora ci dovrà essere un qualcosa che mi dice se è condizionale e se i flag dovranno essere settati, dopo servono i registri sorgente, destinazione.

Il resto è il source 2 sono 12 bit, ovvero bit in più rispetto a quelli che servono per denotare un registro, si usano infatti per specificare il 3° registro oppure per specificare una costante o una sorgente shiftata.

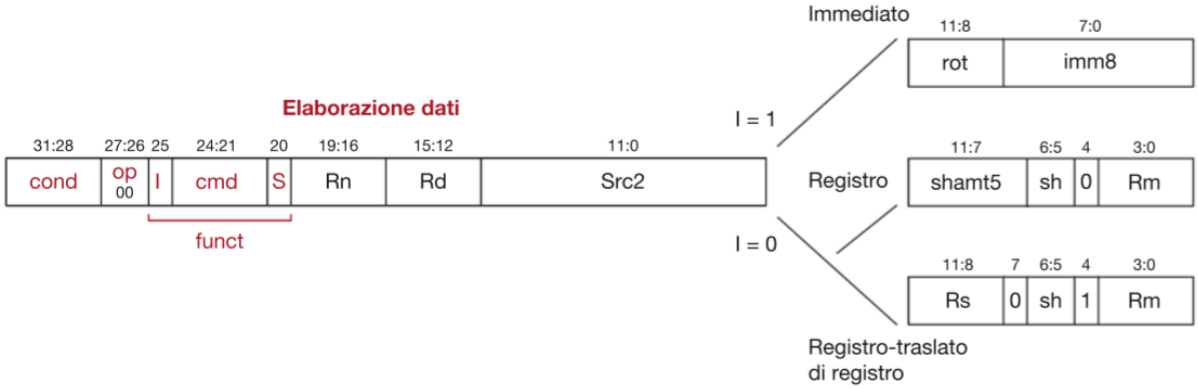
Le tre varianti di Src2 consentono di avere come secondo operando:

1) un immediato;

2) un registro (Rm) eventualmente traslato di una costante (shamt5, da shift amount su 5 bit)

3) un registro (Rm) traslato del contenuto di un altro registro (Rs). Per le ultime due possibilità, il sottocampo sh codifica il tipo di traslazione

Istruzioni operative nel dettaglio:



Il campo **cmd** è da 4 bit, è come una ALU che dati due ingressi calcola un'uscita tra tutte le possibili operazioni immaginabili, il campo cmd dice quale uscita prendere, questo campo ci dice dunque il tipo di operazione.

Gli altri due bit **I** e **S** sono due bit che ci dicono rispettivamente se il terzo operando è un immediato oppure no (I), e S che ci dice se è una opS oppure no, S = 1 ci dice di settare i flag, l'ALU come tutte le alu ha un'uscita per settare i flag.

Esiste un registro speciale detto **cpsr** (*current program status register*) che ha 4 bit per memorizzare i flag.

Questo viene chiamato anche registro di stato e conterrà i valori calcolati per i flag.

I = 1 significa che c'è un immediato, il valore numerico però utilizza al più 8 bit ciò significa che posso scrivere un numero tra 0 e 255, gli altri, bit 11 10 9 8 sono di quanto devo ruotare quel numero per avere effettivamente la costante che mi serve.

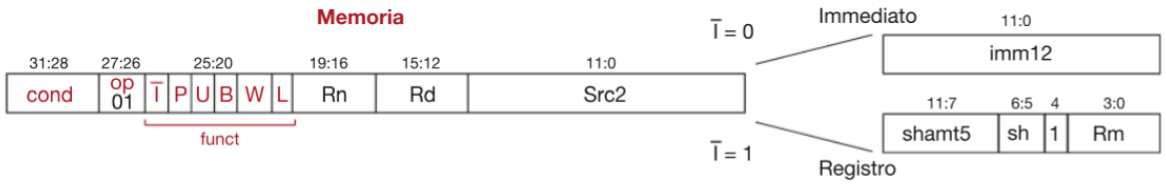
Se I = 0 bisogna leggere il valore da un registro, questo registro può essere letto in due modi diversi, a seconda di quanto vale il bit tra **sh** e **Rm** (vedere figura).

Gli ultimi 4 bit sono il registro sorgente, questo registro può essere sottoposto ad uno shift o ad una rotazione di un valore che può essere una costante o un registro.

Il bit sh può essere 4 cose, lsl, lrl, asl, rtr.

Questi sono codificati rispettivamente come 00 01 10 11.

Istruzioni di memoria nel dettaglio:



Queste sono le istruzioni di memoria, i primi due campi sono uguali alle operative, **funct** è sempre 6 bit, dunque il formato è praticamente identico, solo che il funct ha struttura diversa, P, U, B, W, L sono dei flag speciali, ad esempio B dice se vogliamo fare un byte o una word. Ognuno di questi flag forma uno dei parametri che ci dice come è fatta un'istruzione di accesso in memoria.

a memoria.

Bit	Significato	
	I	U
0	Spiazzamento immediato in Src2	Sottrae lo spiazzamento dalla base
1	Spiazzamento a registro in Src2	Somma lo spiazzamento alla base

P	W	Modo di gestione indice
0	0	Post-indice
0	1	Non supportato
1	0	Spiazzamento
1	1	Pre-indice

L	B	Istruzione
0	0	STR
0	1	STRB
1	0	LDR
1	1	LDRB

Il source2 ha gli stessi bit che avevamo nelle operative ma sono usati in maniera un po' più semplice, perché se dice che è un immediato tutti e 12 i bit sono un valore numerico (si ha il complemento a 2).

Istruzioni di salto nel dettaglio:

Abbiamo sempre il campo cond da 4 bit, l'op da 2 bit, il funct da soli 2 bit, e i 24 bit che rimangono, sono un immediato, in particolare uno dei bit del funct è sempre uno, mentre l'altro, variabile, distingue tra B e BL.

ESEMPI

ADD R0, R1, R2

e0810002 cond => e op => 08 rs => 1 rd => 0 src2 => 002

MICROARCHITETTURA
PROCESSORE SINGLE CYCLE

La microarchitettura è un insieme di componenti hardware che implementano un interprete armv7.

Implementano le istruzioni del linguaggio macchina ed eseguono quello che è codificato in quelle istruzioni.

Quella di cui stiamo discutendo è una parte che ha componenti hardware (memorie, alu, ecc…).

Il tutto è organizzato come una rete sequenziale, quello che questo livello mette a disposizione del livello superiore è la capacità di interpretare istruzioni assembler.

Questo hardware implementa il ciclo fetch-execute:

1. ***fetch***
2. ***decode***
3. ***execute***
4. ***wb***
5. **altre op**

Cos'è lo stato architetturale?

Lo stato architetturale sono le cose che definiscono lo stato di un programma assembler.

In questo stato includiamo un po' di cose, prima tra tutte un program counter che ci dirà quale sarà la prossima istruzione da eseguire.

Avremo dei registri, r0-r3, r4-r11, r12, r13, r14 e r15.

Si parla di stato architetturale perché se prendo un'architettura diversa avrò un numero diverso di registri con nomi e funzioni diverse.

Avremo poi una memoria dati, dove ci metto spazio per contenere valori.

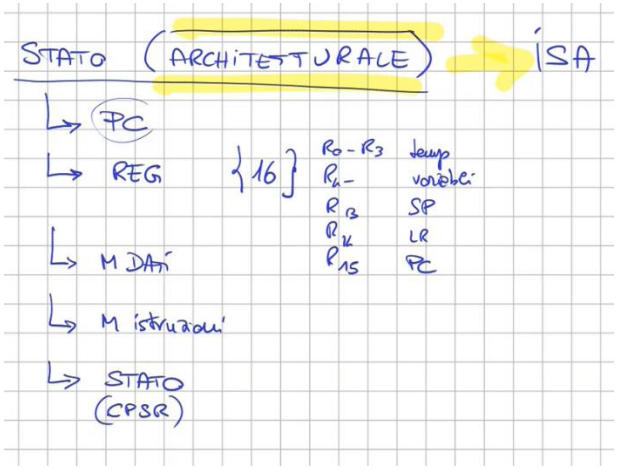
Avremo anche una memoria istruzioni dove memorizzo le istruzioni da eseguire, questa potrebbe essere implementata come quasi una rom, mentre la memoria dati deve poter essere letta e scritta, perché i dati devono poter essere modificati durante l'esecuzione del programma.

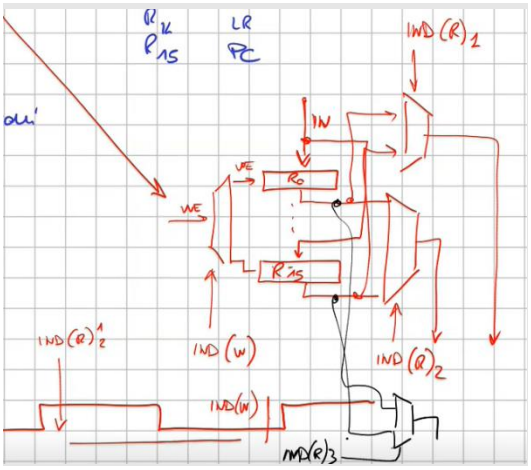
Lo stato si definisce architetturale perché fa riferimento all'instruction set.

Nello stato architetturale manca ancora qualcosa, il cpsr, che ci permette di verificare lo stato dei flag settati dal programma.

In altre versioni ci potrebbero essere dei registri non architetturali, dove si possono scrivere dei risultati parziali ottenuti da alcune istruzioni.

Lo stato non architetturale non dipende dall'implementazione (es: le varie versioni di arm).





Questi componenti nello stato architetturale sono tutti realizzati con le componenti viste sino ad ora, il pc è un registro (da 32 bit) e lo stato pure anche se basterebbero 4 bit per le informazioni che codifica (i flag).

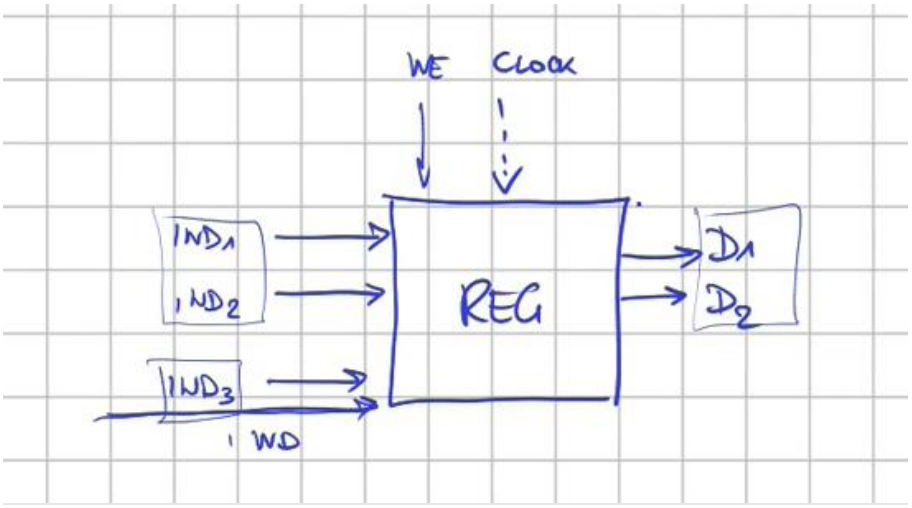
I registri fanno parte di una memoria, questa memoria deve essere realizzata con la stessa tecnologia dei registri, perché la lettura e la scrittura (di registri diversi) a volte è necessaria nello stesso ciclo di clock.

La memoria è dunque un insieme di registri comandati da un demultiplexer, con un ingresso duplicato su tutti i registri che setta il we, e uscite collegate ad un multiplexer che ne manda in out solo una, sia il multiplexer che il demultiplexer sono comandati da indirizzi.

Abbiamo due multiplexer che leggono, in modo che con 2 indirizzi diversi riusciamo nello stesso ciclo di clock a leggere due valori.

Non sono 16 registri messi a caso, bensì 16 registri organizzati in una memoria multi porta che ha 3 ingressi uno per la lettura e uno per la scrittura, per la store serve però un terzo multiplexer che prende gli stessi segnali degli altri, ma ha un indirizzo di lettura (ind_3) che serve per tirare fuori il terzo dato.

Il register file lo vedremo come una cosa che ha due indirizzi per la lettura uno per la scrittura, un dato per la scrittura, un enable per dire se deve scrivere o no, e due uscite che corrispondono ai due indirizzi degli ingressi corrispondenti, ci sarà anche un ingresso di tipo clock.



Le memorie che vedremo saranno due, una data memory e una instruction memory, queste verranno rappresentate come memorie normali, nel seguito del corso le vedremo rappresentate come cache.

Entrambe avranno un indirizzo che dice dove operare, la DM avrà un data in e un data out, ovvero un ingresso che ci dice dove leggere, e un'uscita che ci dice cosa è stato letto, ci sarà anche un we e prenderà un clock per funzionare.

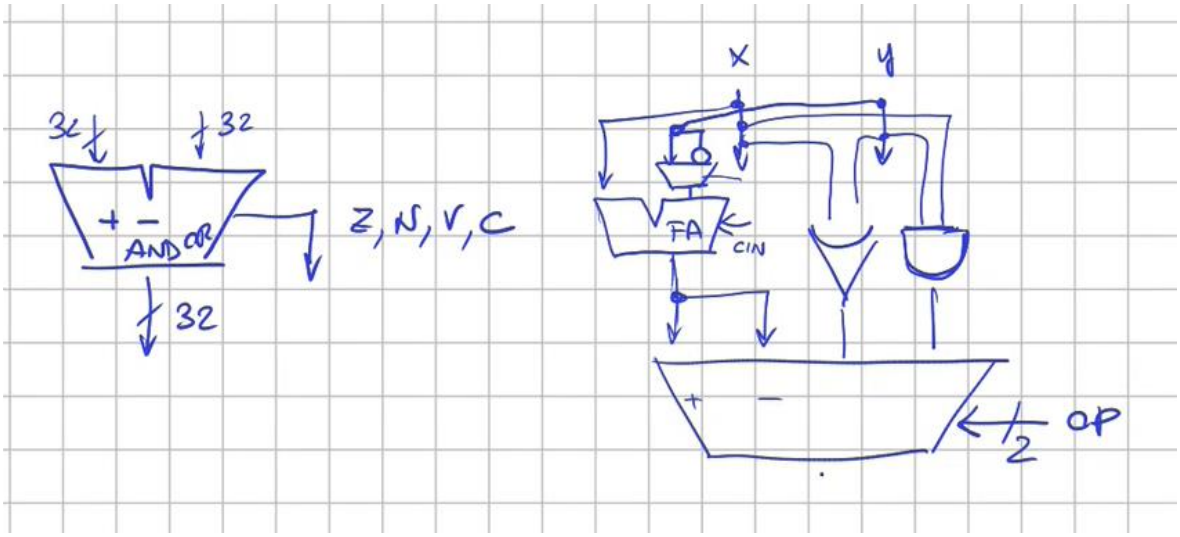
Nella IM abbiamo un'uscita che sarà un'istruzione, avremo il clock, ma non avremo né il data in, né il we.

Adesso abbiamo bisogno di una alu, immaginiamo che possa fare solo 4 operazioni tra due numeri da 32 bit, più, meno, and e or.

Generando i segnali Z, N, V, C e un risultato a 32 bit.

Questa ALU è composta da 3 data path, in cui se devo fare l'or o l'and tra due ingressi X e Y, li prendo e metto i due segnali che fanno l'AND o l'OR bit a bit, l'uscita di questi due circuiti va in un multiplexer che sceglie il risultato che mi serve.

L'altro ingresso è quello che uso per la somma o per la sottrazione, ci metto un full adder collegato a due ingressi, il secondo ingresso lo faccio andare normale o negato, questo è deciso dal bit $cntrl_{in}$ che mi dirà se devo fare una somma (=0) o una sottrazione (=1)



Adesso mettiamo questi componenti in fila e vediamo come possiamo eseguire delle istruzioni scritte in linguaggio macchina.

Il nostro ciclo inizia con la fetch, infatti per prima cosa dobbiamo recuperare l'istruzione dalla instruction memory, l'indirizzo che dovremo leggere nella IM è dato dal program counter.

Avremo quindi PC seguito da Instruction memory, questa istruzione presa sarà da 32 bit, suddivisi come abbiamo visto.

Supponendo di avere una add mi serve un componente memoria per recuperare il contenuto delle due sorgenti, e una alu per eseguire il calcolo. L'uscita della alu deve essere collegata a sua volta alla memoria dati per poter scrivere il risultato nel registro.

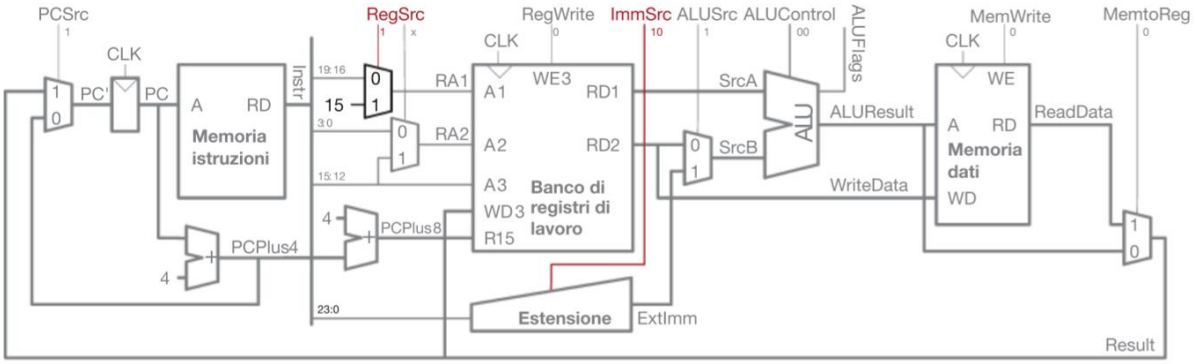
Uno dei due registri può essere un immediato, dobbiamo aggiungere quindi prima della ALU un multiplexer che sceglie l'ingresso tra registro e immediato, quest'ultimo però (l'immediato) ha meno bit rispetto al valore nel registro (8 vs 32). Quello che facciamo dunque è aggiungere un estensore che prende gli 8 bit e li fa diventare 32 preservando il segno e tutte le proprietà.

Se voglio fare l'estensione da 4 a 8 bit e il numero è -2, prendo due, lo nego, gli sommo 1 e questo è -2, per farlo diventare a 8 bit, copio i 4 bit del numero, nei 4 bit meno significativi e poi copio il bit più significativo nei 4 bit restanti (vedere immagine con l'esempio).

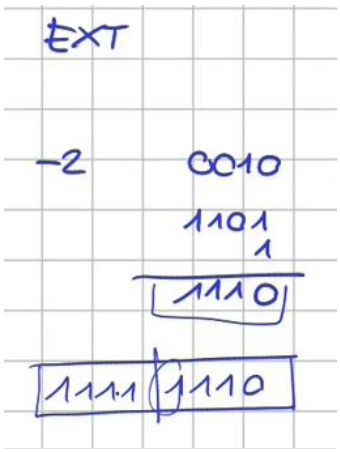
(Per fare l'esempio con 32 bit, se il numero è negativo dobbiamo fare il complemento-2 del numero, prendere gli n bit e copiarli negli n bit meno significativi, poi si prende il bit più significativo del numero complementato e si copia nei 32 - n bit restanti)

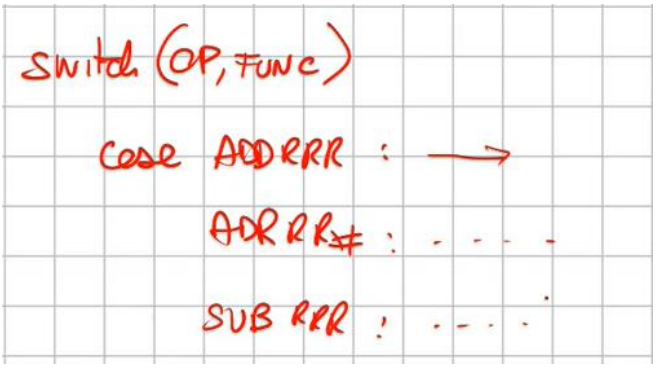
Se invece di essere una add fosse stata una sub, avremmo dovuto dare un segnale diverso alla alu.

Questo raffigurato in figura si chiama dataPath, questo percorso ha tutto un insieme di segnali di controllo, we del register file, op della alu, i vari controlli dei multiplexer. Sono i segnali che permettono di calcolare esattamente l'istruzione di cui abbiamo fatto il fetch anziché un'altra.



ES estensione di -2





Ci serve però un'entità esterna che decida quali sono i segnali di controllo.

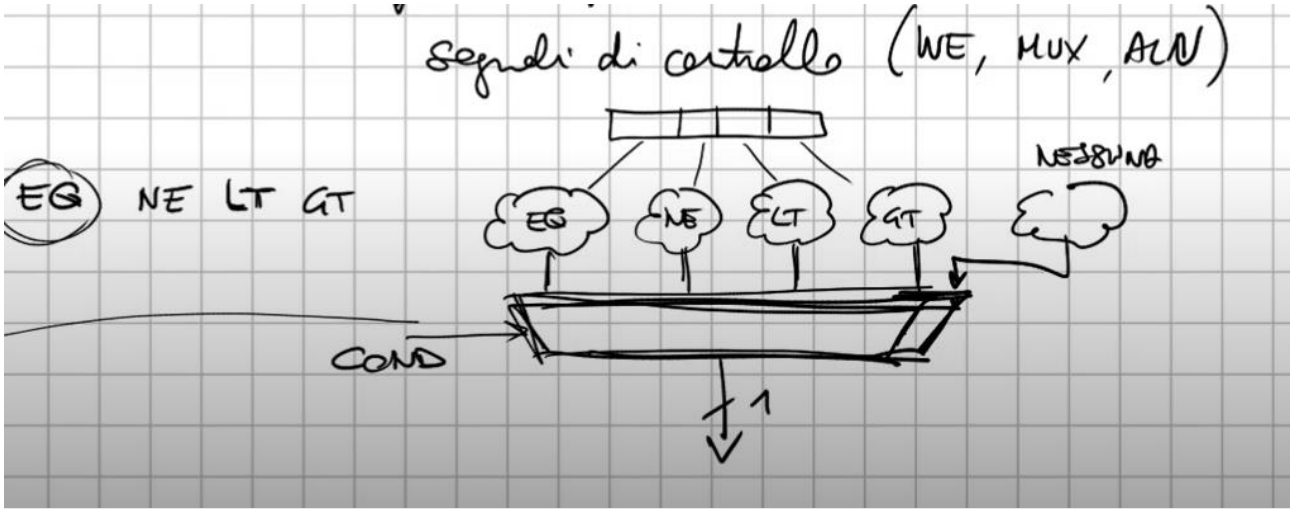
Questa unità è la parte controllo, che non fa altro che generare tutti i segnali per configurare il data path in modo da eseguire l'operazione richiesta.

La parte controllo riceve parte dell'istruzione ed è in grado tramite uno switch, di selezionare i controlli giusti.

Nel caso di una addeq devo decidere se la eseguo oppure no, se la eseguo il we dei registri deve essere 1 se non la eseguo deve essere 0, per sapere se devo eseguirla devo controllare i flag. Questo complica le cose perché nella parte controllo io ho tutta una serie di controlli che dai bit dell'istruzione mandano i segnali di controllo, ma finora immaginavamo di avere solo cose combinatorie, adesso invece ci serve immaginare 4 registri, z, n, c e v che rendono dunque la rete sequenziale.

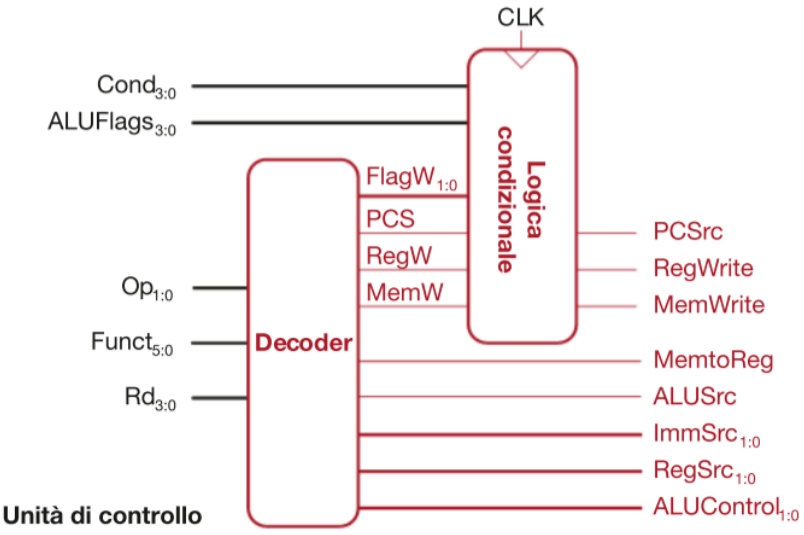
Uso delle reti logiche per calcolare GT, EQ, LT ecc..

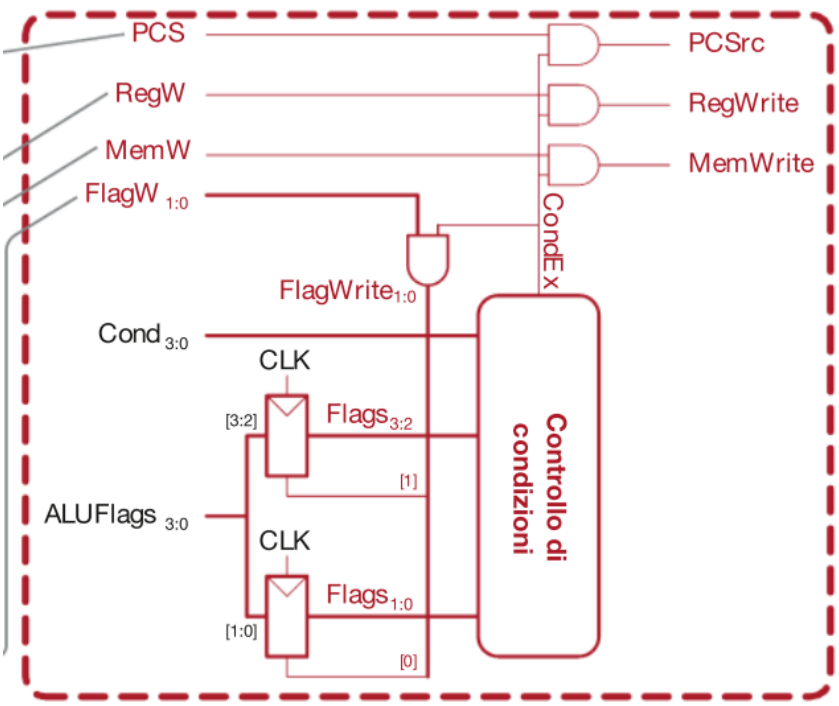
Hanno in ingresso quei 4 bit e producono 1 se la condizione è vera 0 se è falsa, i bit di cond vengono usati come bit di controllo di un multiplexer.



Il libro usa un decoder e un cond logic, nel blocco cond logic vanno i bit del campo cond dell'istruzione e i flag, nel blocco decoder vanno invece i bit op, funct e le altre cose che ci servono per capire cosa fa l'istruzione.

Entrambi questi oggetti generano dei segnali ma quelli di controllo li genera tutti il decoder. Siccome abbiamo la logica delle condizioni i segnali che determinano il cambiamento dello stato architetturale passano nel decoder e in caso vengono modificati.





(c) Logica condizionale

Ogni volta che faccio un ciclo del clock, la decode la fa la parte controllo, e l'esecuzione la fa il datapath, con i segnali dati dalla parte controllo aggiornati.

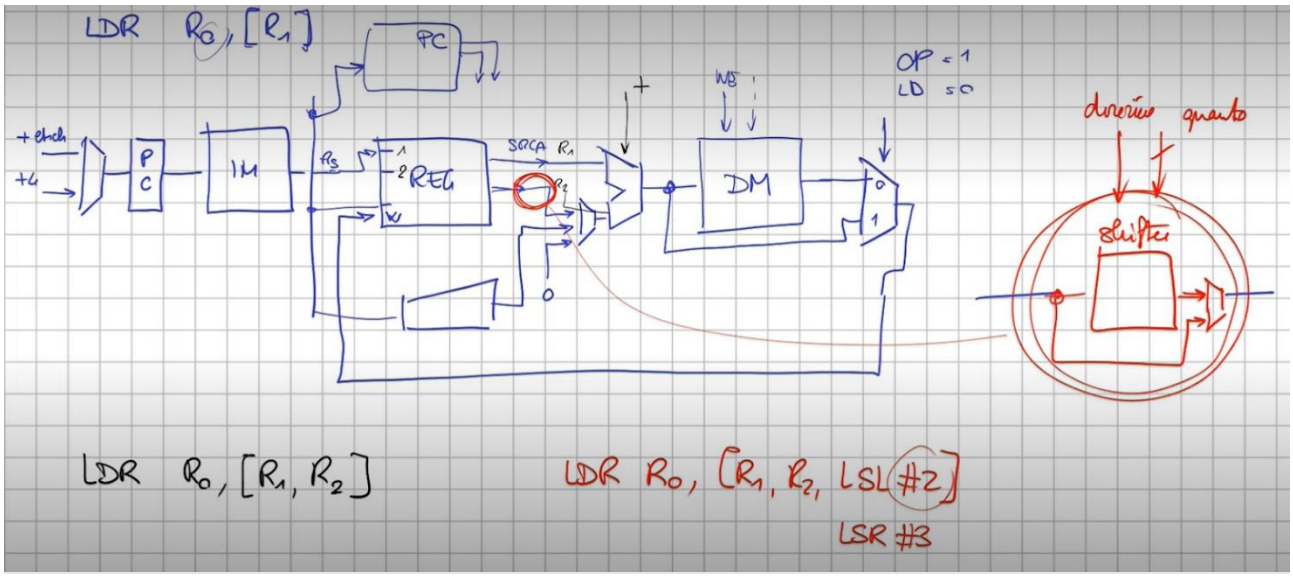
Manca la memoria dati.

Vediamo come fare la ldr r0 [r1]

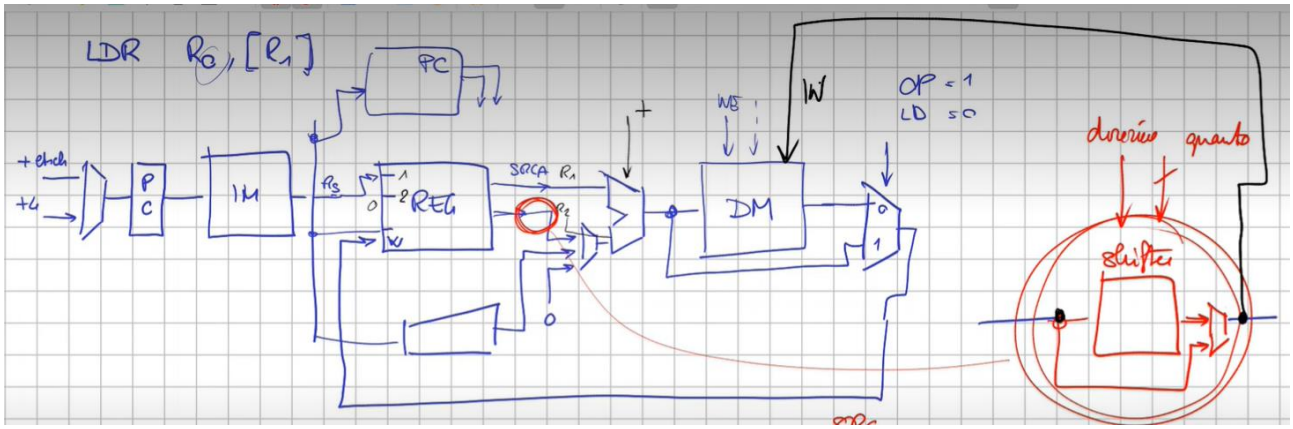
Anche qua abbiamo un program counter con davanti un multiplexer che fa scegliere etichetta o +4, abbiamo la instruction memory e sopra la parte controllo, dopo c'è il register file, un multiplexer, la alu e poi la memoria dati. A quest'ultima do un indirizzo per leggere qualcosa e mi darà il risultato che devo andare a scrivere nel registro, la logica è la solita di una add.

Quando faccio un'operativa la data memory non serve, fuori ci sarà quindi un ulteriore multiplexer che sceglie se rimandare indietro (per salvarlo nel registro) quello che abbiamo letto in memoria oppure quello che abbiamo calcolato con la alu, se è operativa sceglie 1 se e una load sceglierà 0.

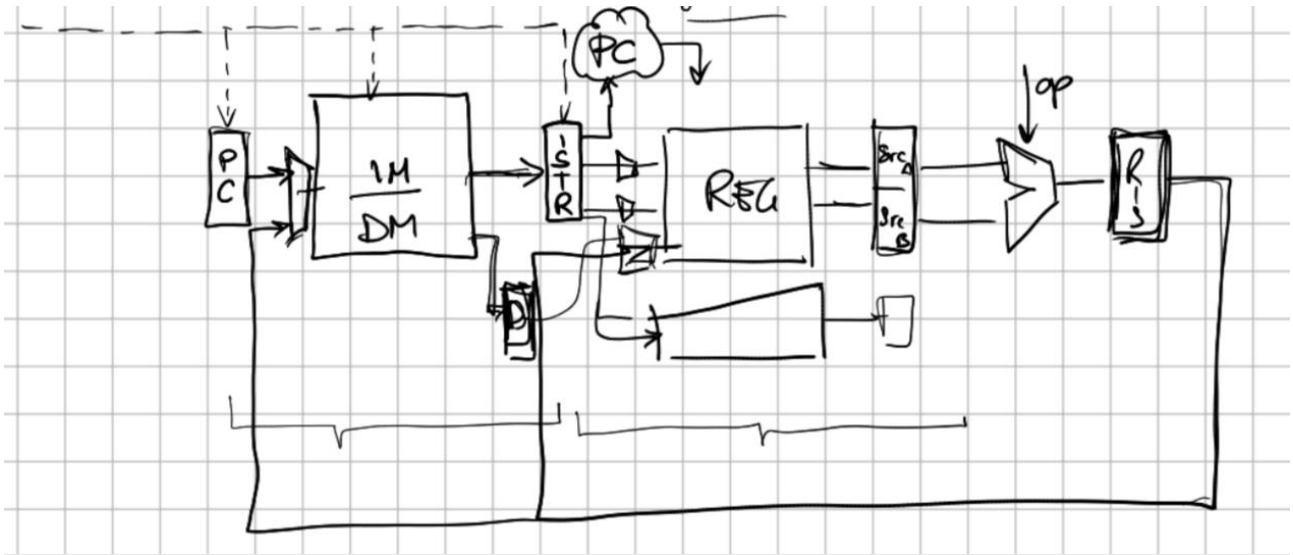
Prendendo di esempio l'istruzione LDR R0, [R1, R2, LSL#2] ci serve uno shifter tra l'uscita di reg e il multiplexer.



Per eseguire una store mandiamo il risultato dello shifter dentro la data memory come ingresso di input.



PROCESSORE MULTI CYCLE



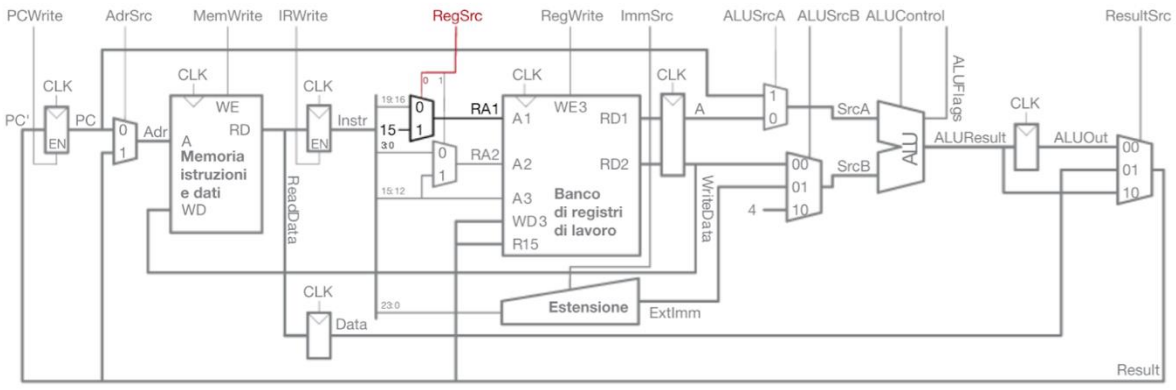
L’idea è quella di usare i cicli di clock in maniera diversa, ad esempio se voglio fare un’istruzione operativa, uso un primo ciclo di clock per fare quello che si faceva nella fetch, un secondo per fare quello che si faceva nella decode, un altro per l’esecuzione e un altro per il wb.

Nel fetch prima avevamo un pc che indirizzava una instruction memory la quale tirava fuori un’istruzione in cui andavamo a prendere i bit corrispondenti alla codifica dell’istruzione, noi al posto della decodifica dell’istruzione usiamo un registro istruzione, questo mi permette di dire che in un ciclo di clock si fa solo quella cosa, nel ciclo di clock successivo si farà la decodifica, ovvero si prendono i bit dell’istruzione e ad esempio si mandano alla parte controllo, mentre altri vanno nell’unità registri, dove ci sono dei multiplexer per selezionare i bit, e altre cose andranno a finire come ingresso dell’unità di estensione degli immediati.

Una volta recuperati i dati prima finivamo alla alu, prima della alu, qua scriviamo ulteriori 2 registri, che sono il srcA e il srcB che saranno gli operandi della alu insieme ai segnali di controllo, lo stesso potrebbe valere per quello che tiriamo fuori dall’estensore.

Complessivamente spendiamo 3 tau (cicli di clock) che però sono molto più piccoli.

Una volta calcolato il risultato, metto un altro registro che memorizza il risultato della alu, e questo registro o lo mando alla memoria dati, oppure lo uso per mandare indietro quello che devo scrivere, questo registro, o va a finire come ingresso all’unità registri, o va a finire come ingresso a una cosa che questa volta può essere sia memoria dati che memoria istruzioni, usando un multiplexer posso indirizzare queste due memorie.



Subito dopo la memoria istruzioni e dati ci sono due registri uno per le istruzioni e uno per i dati.

Dopo la memoria dei registri c’è un registro per memorizzare le due sorgenti lette, e dopo la alu un registro per salvare il risultato.

Se dobbiamo fare un salto, in un primo ciclo vado a leggere la branch e la scrivo nel registro istruzione, una parte va all’unità di controllo che manderà i segnali opportuni, a questo punto viene calcolato l’operando immediato. In questo caso vengono spesi 3 tau perché il risultato non viene riscritto nel registro della alu.

Qual è il problema di questa architettura?

Abbiamo aggiunto 4 registri non architetturali che servono a spezzare la nostra esecuzione, e presumibilmente abbiamo scorciato molto il ciclo di clock, adesso in generale sono tutti di tipo multiplexer+memoria o multiplexer+alu, il problema è però che alcune cose che accadono dentro è perché abbiamo già eseguito dei cicli di clock, i segnali che la parte controllo manda dipendono dalla fase che stiamo eseguendo.

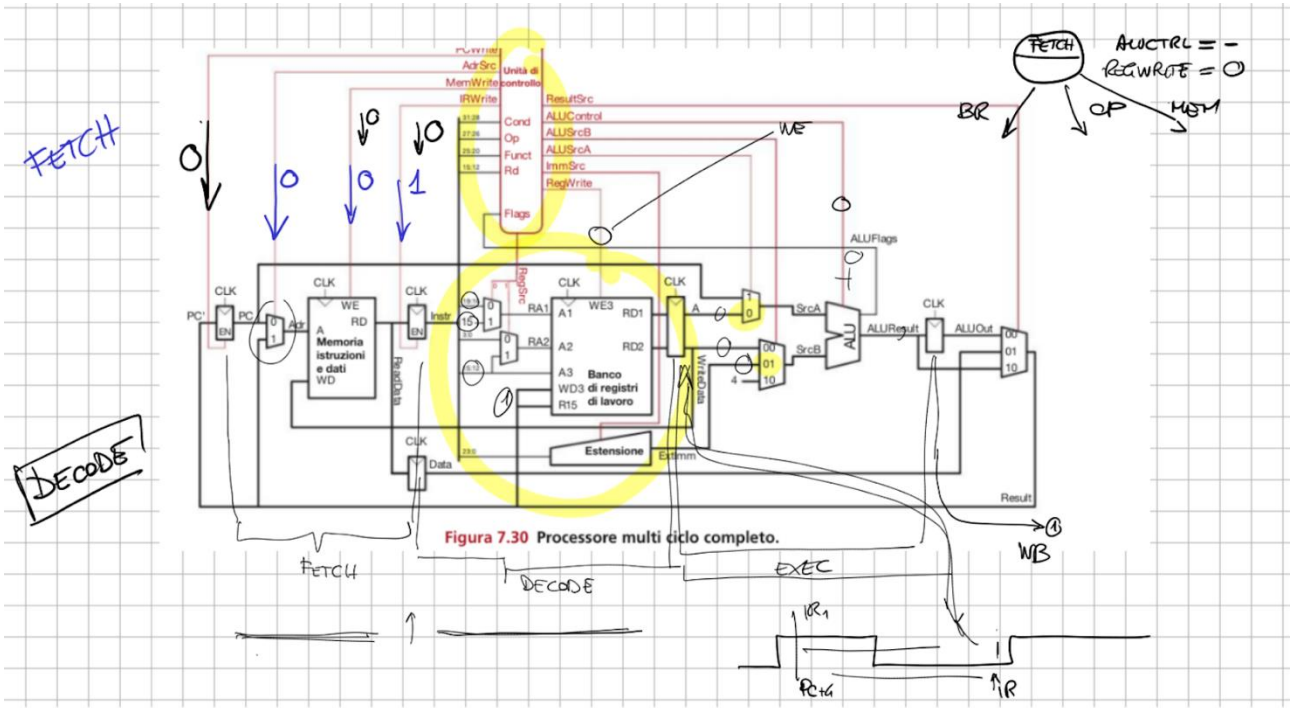
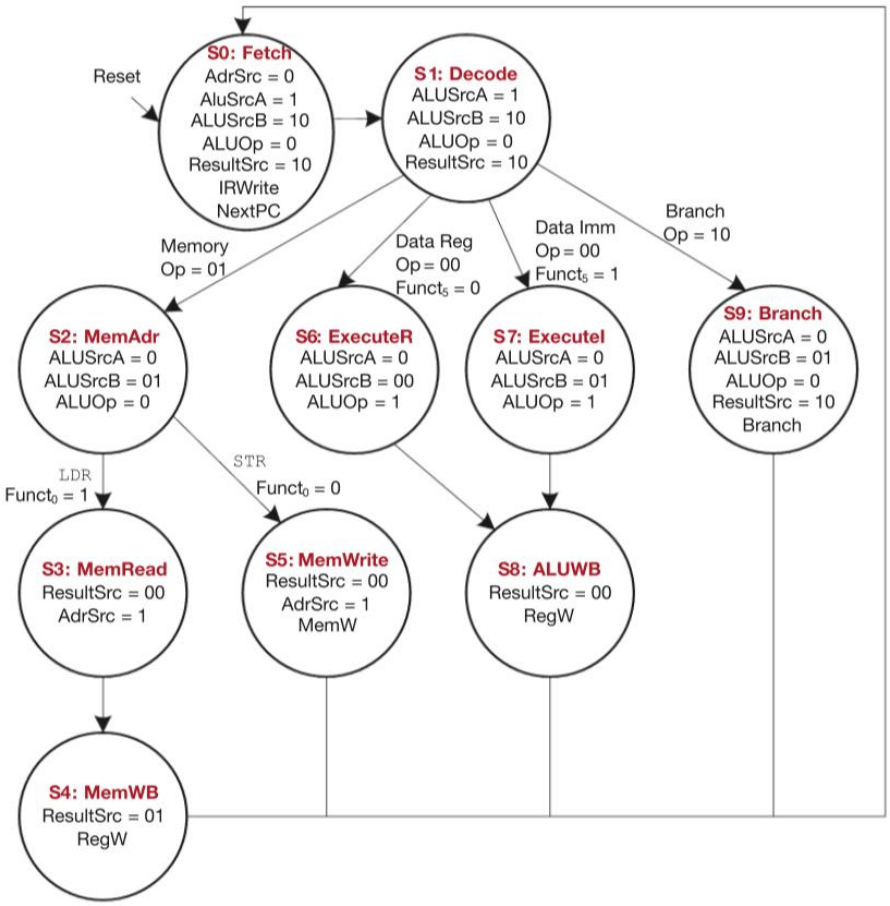


Figura 7.30 Processore multi ciclo completo.

Possiamo vedere i segnali di controllo diversi nelle fasi fetch e decode (blu e nero). La parte controllo questa volta è un automa ovvero una rete sequenziale, è un automa che dice, se sono nella fase di fetch, alcuni segnali possono essere non specificati (es alucontrol).

Quando finisco la fase di fetch dipende dall'istruzione che ho ottenuto.

La parte controllo del multi cycle è ancora abbastanza semplice ma deve avere una struttura articolata come un automa.



State	Datapath µOp
Fetch	Instr ← Mem[PC]; PC ← PC+4
Decode	ALUOut ← PC +4
MemAdr	ALUOut ← Rn + Imm
MemRead	Data ← Mem[ALUOut]
MemWB	Rd ← Data
MemWrite	Mem[ALUOut] ← Rd
ExecuteR	ALUOut ← Rn op Rm
ExecuteI	ALUOut ← Rn op Imm
ALUWB	Rd ← ALUOut
Branch	PC ← R15 + offset

Questo oggetto significa che se io devo fare un'operazione a seconda delle 3 classi che ho, ho percorsi diversi che mi concludono l'automa.

Es:

Operativa, faccio fetch, decode, poi o s6 o s7, e poi ho un wb, a questo punto sono in fondo e posso tornare in cima.

L'operativa sono 4 stati attraversati che corrispondono a 4 cicli di clock (4 tau).

Se faccio un'operazione di salto, faccio una fetch, una decode, una branch e poi torno su, un salto sono 3 tau ovvero 3 cicli di clock.

Se faccio un'operazione di memoria bisogna distinguere tra le load e le store, se ho una load faccio una fetch, una decode, memAdr, load, wb, 5 tau.

Se devo fare una store, faccio una fetch, una decode, una memAdr e un write, 4 tau.

Mentre con il single cycle se avevo 10 istruzioni spendevo 10 cicli di clock, con il multi cycle devo andare a vedere come sono fatte le istruzioni e poi posso dire il costo, il singolo ciclo di clock però dura molto meno in quanto devo fare molte meno cose, (vedere nell'immagine sopra quali pezzi usano le varie fasi).

Esempio fare le seguenti istruzioni

LDRB R0, [R2], #1

CMP R0, #0

BEQ fine

Nel single cycle essendo 3 istruzioni costano 3 tau (CPI (cycles per instruction) = 3).

Nel multi cycle abbiamo 5 tau' + 4 tau' + 3 tau' = 12 tau'.

Abbiamo un guadagno (nell'usare un multi cycle rispetto ad un single cycle) solo se 12 tau' sono meno di 3 tau ovvero se $tau' < {}^{tau}/_{4}$.

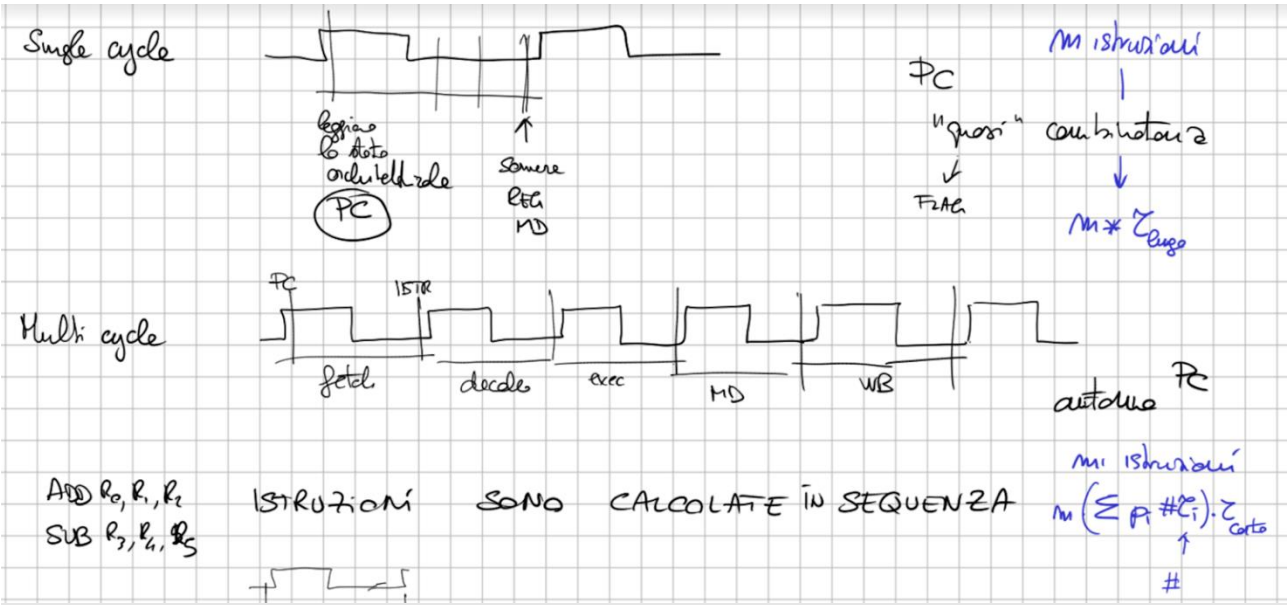
Quindi a parità di numero, impiego meno tempo a fare 1 tau' che 1 tau.

Nel processore single cycle quello che succede è che in un ciclo di clock, leggiamo lo stato architetturale e prima che il ciclo di clock torni alto abbiamo computato un'istruzione assembler.

Nel processore multi cycle quello che succede è che grazie a dei registri non architetturali riusciamo a spezzare il ciclo di clock in tante parti in modo che in ogni ciclo di clock venga elaborata solo una parte dell'istruzione.

Nel single cycle m istruzioni richiedono m*tau_Lungo cicli di clock per completarsi.

Nel multi cycle m istruzioni richiedono una media pesata delle istruzioni per il numero dei cicli di clock dell'istruzione corrispondente moltiplicata per il numero di istruzioni tutto moltiplicato per tau_corto.



MICROARCHITETTURA PIPELINE

Il processore pipeline è quello che usa il parallelismo temporale, cioè quello a catena di montaggio, useremo più o meno le fasi che avevamo nel single cycle, ma invece di farle in fila, e quindi di concludere l'esecuzione di un'istruzione alla volta, useremo dei registri non architetturali fra queste varie fasi come se stessimo spezzando per il multi cycle ma poi ciascuno di questi pezzi lo utilizziamo per fare un pezzetto di un'istruzione diversa.

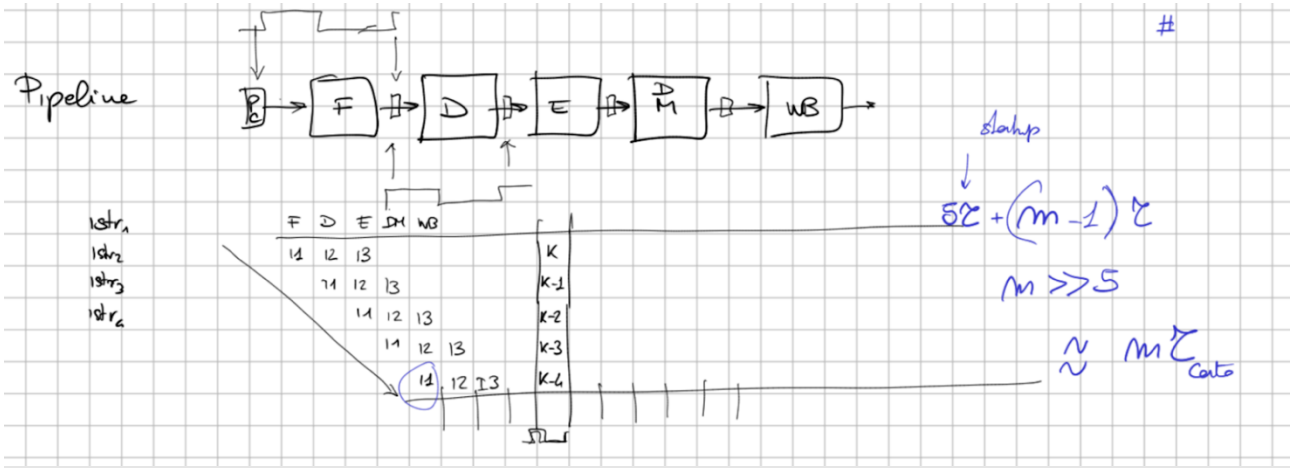
Il program counter dà un indirizzo, il fetch lo va a prendere nella memoria istruzioni e passa quello che ha letto alla decode; quindi, fondamentalmente avremo tanti registri che servono a passare quello che abbiamo fatto allo stadio successivo, quello che accade tra la lettura di un registro non architetturale e la scrittura del prossimo è un ciclo di clock.

Ad esempio mentre sto facendo il fetch di una istruzione posso fare la decode di quella di cui ho fatto il fetch prima.

Se devo eseguire 4 istruzioni posso immaginare di avere le fasi di: fetch, decode, execute, data memory e write back, e di far passare le istruzioni nel tempo durante le varie fasi.

Da un certo punto in poi avremo che in un ciclo di clock riusciremo a fare parte dell'istruzione k, k-1, k-2, k-3, k-4. Servono 5 cicli di clock per far iniziare a lavorare tutti.

Per il pipeline se non ci fossero problemi avremmo 5 tau_{corto} (tempo di startup) + (m-1) istruzioni ognuna delle quali richiede un tau, siccome m normalmente è molto più grande di 5 è possibile approssimare come m*tau_{corto}.

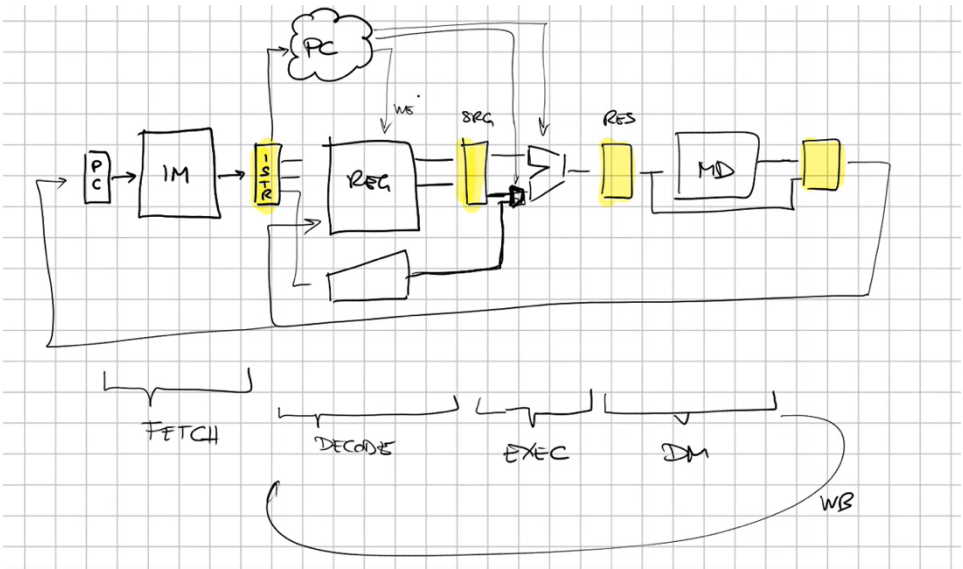


Questo è il principio generale, il problema è che non ci chiediamo quale stato sia più lento o più veloce, il fatto è che tutti gli stadi devono funzionare con lo stesso ciclo di clock, quindi esso deve essere lungo abbastanza ovvero almeno pari allo stato più lento.

Il concetto è che tutto quello che succede in una fase venga scritto in dei registri, in modo che tra due registri non architetturali si possano fare delle operazioni relative a istruzioni diverse, è interessante capire come si può fare.

Avremo come sempre il program counter la cui uscita andrà in una instruction memory, questa volta siccome la instruction memory e la data memory le usiamo in fasi diverse dovremmo riconsiderarle separate, la data memory sarà in fondo prima del write back in modo da poter fare le load e le store una volta che sappiamo qual è l'indirizzo a cui operare nella data memory.

Un primo passo che sarà quello di fetch legge una cosa che è il registro istruzioni, (che non contiene altro che il codice in linguaggio macchina della mia istruzione) l'indirizzo in parte sarà utilizzato per creare i segnali di controllo, e verrà usato dal banco dei registri per produrre i due operandi, abbiamo poi due registri non architetturali per salvare i sorgenti, avremo una alu, dove arriveranno le uscite dei due registri e l'uscita dell'estensore, la alu scriverà su una cosa che è un registro non architetturale per salvare il risultato del calcolo appena eseguito, questo risultato potrà servire per indirizzare la memoria dati nel caso in cui sia una load o una store, o eventualmente per passare allo stadio successivo, anche qui avremo un registro che mi separa la fase di accesso alla memoria dati. Infine avremo un po' di traffico che dall'ultimo registro va nel banco dei registri o nel program counter.



Quando la parte controllo capisce che deve mandare i segnali per una certa istruzione?

Lo capisce quando riesce a leggere dalla IM il codice binario dell'istruzione.

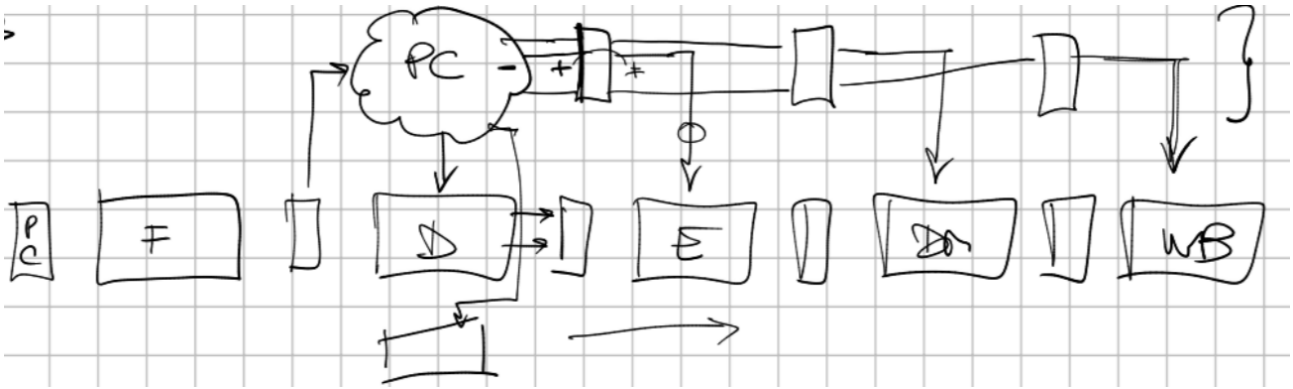
Una volta che so il codice in linguaggio macchina dell'istruzione so in tutto e per tutto cosa quell'istruzione fa e cosa gli devo mettere a disposizione.

Il problema è che in un certo istante nelle varie fasi abbiamo istruzioni diverse, con segnali di controllo dunque, diversi.

Questa cosa non funziona, nella fase di decode capisco cosa fare ma devo poi applicare i segnali giusti nelle fasi successive.

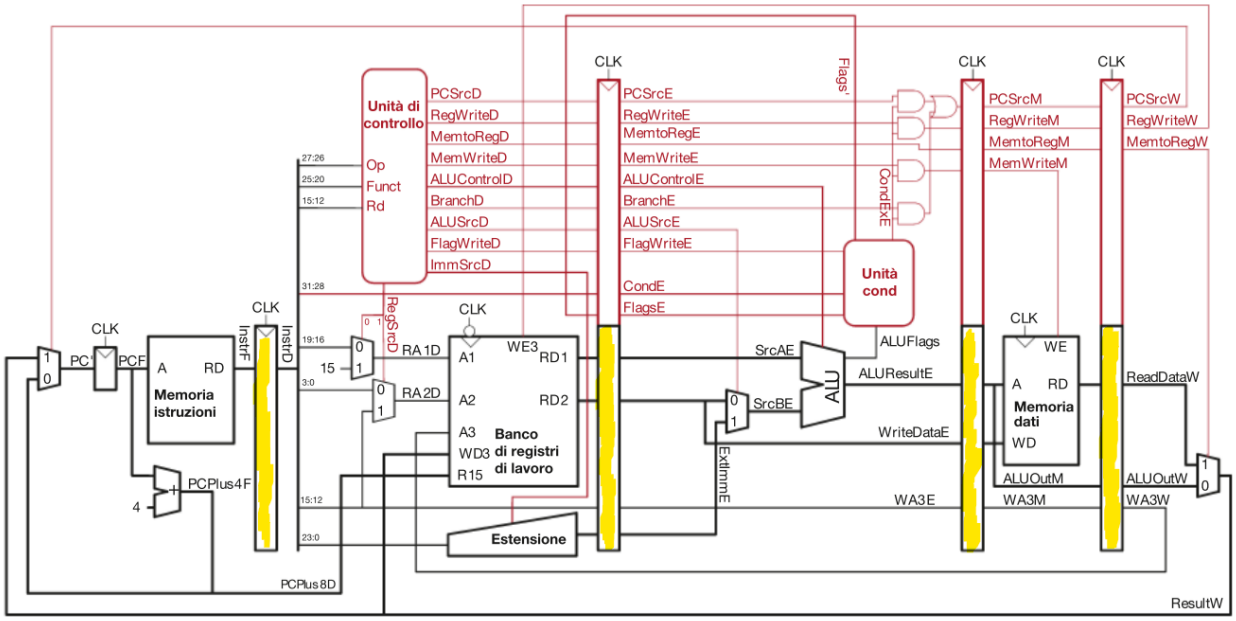
Come si ritarda di un ciclo di clock l'esistenza di un segnale? Si usano dei registri.

Eventuali segnali che vengono generati per la decode vengono inviati immediatamente, gli altri segnali vengono ritardati mettendo dei registri in corrispondenza dell'inizio delle varie fasi.



Nel processore pipeline non ho bisogno di un automa, ma di una rete di controllo che è quasi una rete combinatoria (non lo è per colpa dei flag che devono essere memorizzati in dei registri che possono essere visti come dei veri e propri stati) ma quello che produce la rete combinatoria viene consumato nei cicli di clock successivi perché questi registri dei segnali di controllo mi permettono di schedulare i comandi nel tempo nella stessa maniera in cui i dati nel tempo passano da uno stadio all'altro.

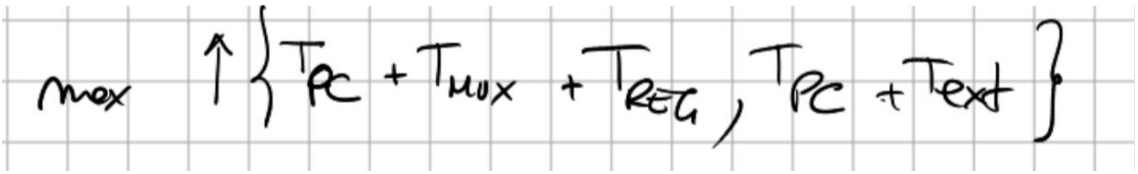
È come se avessimo un *pipeline di controllo* e un *pipeline di datapath* dove in quest'ultimo succedono le cose che vengono comandate dalla parte controllo.



Il data path è identico a quello del single cycle tranne per l'aggiunta di alcuni registri (evidenziati), il primo per contenere le istruzioni, il secondo per contenere gli operandi della alu, il terzo per contenere i risultati della ALU, il quarto per contenere il risultato di quello che facciamo in memoria, o con questo bypass (aluresult) che arriva fino al pc.

In aggiunta a questo abbiamo altri 3 registri nella parte controllo, uno che ritarda la fase di esecuzione, uno che ritarda i comandi che servono per la memoria, e un altro che ritarda i comandi che servono per dire come fare il write back.

Sebbene i multiplexer siano sotto la parte controllo, dal momento in cui leggo il registro istruzione pago un t_{pc} per stabilizzare le uscite della parte controllo, più un tempo del multiplexer, poi pago il tempo di un'unità registri e solo a quel punto posso scrivere qualcosa nel registro finale. In realtà devo tener conto che questi tempi dovrebbero essere il massimo tra questo e l'altro giro, che è quello di far lavorare la parte controllo e generare qualcosa che va all'estensore senza passare dalla register file.



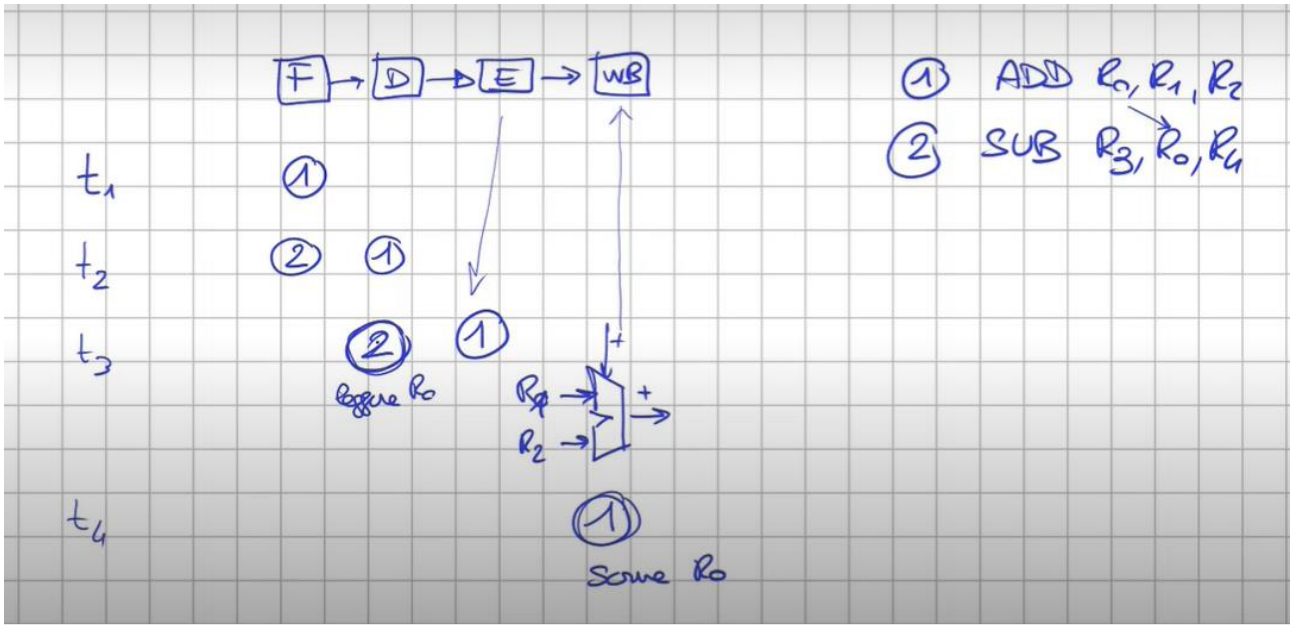
Dipendenze logiche e di controllo

Consideriamo adesso 4 stati, fetch, decode, execute, e write back.

E due operazioni ADD R0, R1, R2 e SUB R3, R0, R4

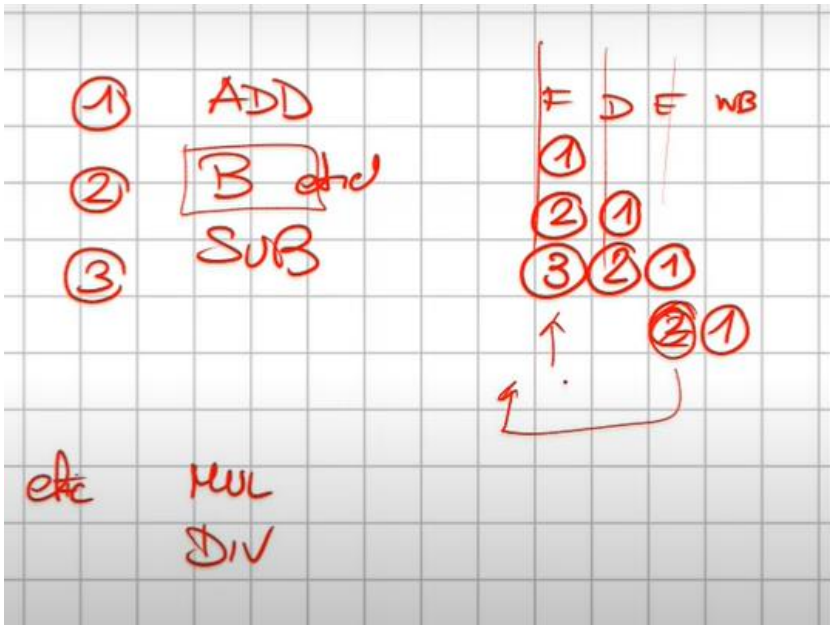
Se io faccio queste operazioni nel processore pipeline, al primo ciclo di clock faccio la fetch della add al secondo la add viene decodificata e io faccio la fetch della sub, al terzo ciclo di clock la add è in esecuzione e la sub in decodifica, e qua sorge un problema perché se la add è in esecuzione vuol dire che ci troviamo nella situazione in cui R1 e R2 sono davanti ad una ALU che produrrà un risultato che è una somma, ma il risultato finirà dentro il registro R0 solo durante la fase di write back ovvero devo fare un ulteriore passo in tempo t4 in cui l'istruzione add scrive in R0 il contenuto giusto, ma in realtà nella decodifica della sub io sto

già leggendo il valore del registro R0 e quello che trovo non è quindi l'R0 giusto perché questo verrà calcolato tra 2 cicli di clock.



C'è anche un altro caso, se le istruzioni fossero state una ADD, B, SUB

Faccio il fetch della Add, quando questa va nella decode faccio il fetch della B, quando vado a finire questo io sto facendo il fetch della Sub, il branch però è incondizionato, se l'etichetta fosse sotto la sub, nel momento in cui faccio il fetch della sub questa operazione non mi serve a nulla. Io saprò infatti cosa mi serve veramente solo quando avrò eseguito l'istruzione di salto.



Questo non accade solo con i salti ma anche con le istruzioni condizionali.

Questi sono i problemi delle **dipendenze logiche e dipendenze di controllo**.

Cosa possiamo fare per evitare che la necessità di avere un risultato da un'istruzione che ancora deve essere eseguita blocchi tutto?

Per fare questo abbiamo 2 opzioni, una è quella di cercare di procurarci in maniera anticipata i risultati, e un'altra è quella di fermare la catena di montaggio. Se ho bisogno di sapere come nel caso di prima che cosa produce la add per fare la sub, posso bloccare la parte di catena che non è quella della add, mando avanti la add, quando la add ha riscritto (wb) faccio ripartire la catena di montaggio.

1° tecnica:

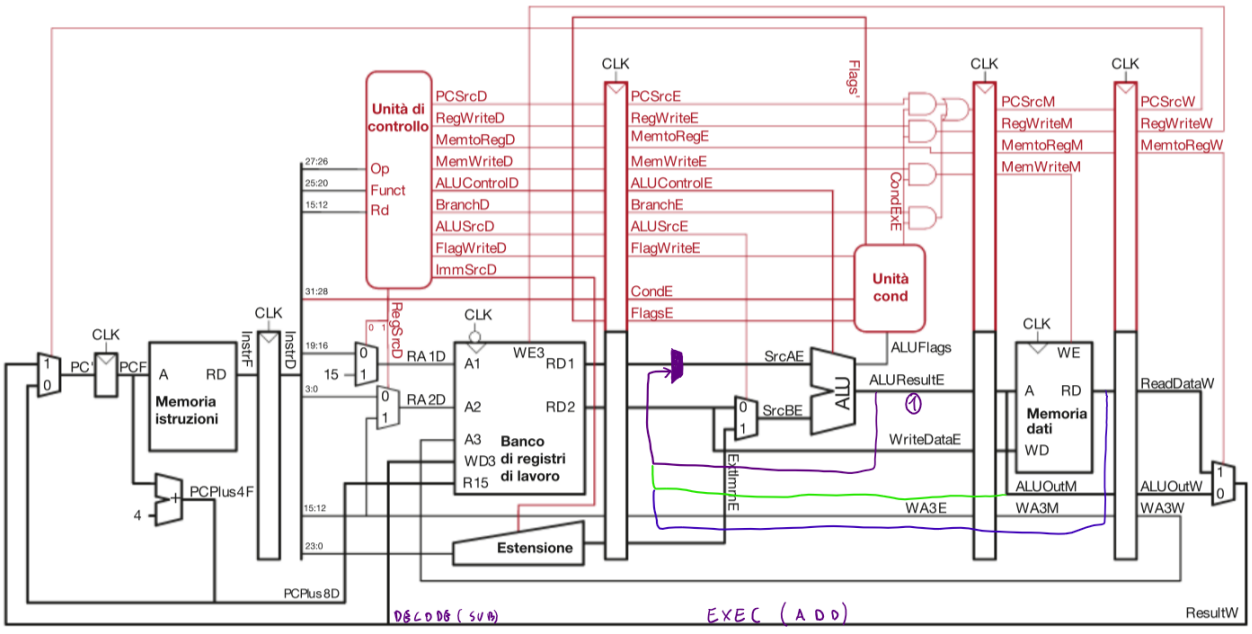
Dobbiamo cercare di capire meglio dove vengono prodotti i dati e dove vengono consumati.

Riprendiamo l'esempio ADD R0, R1, R2 e una SUB -, R0, - (non ci interessa cosa usa, basta solo che come sorgente ci sia il registro interessato ovvero R0).

Il primo momento in cui abbiamo il valore di r0 è nel tratto alu result ([1] nella figura sotto).

La alu calcola la somma tra R1 e R2, mentre sto facendo questo sono nello stato execute, ciclo di clock i-esimo, in quello stesso ciclo di clock, nello stato precedente il banco dei registri sta cercando di leggere i registri necessari alla sub, tra cui r0. Però il contenuto di R0 non è quello buono, è sempre quello di prima, non è ancora stato riscritto. Quello che possiamo fare è aggiungere un multiplexer tra banco dei registri e ALU (mult viola in figura), a cui facciamo prendere qualcosa che viene dal banco dei registri o dalla alu result (percorso viola).

Il problema è che questo è un ciclo, non abbiamo registri intermedi, e dunque abbiamo problemi di stabilizzazione, per risolvere però anziché prenderlo subito dopo la ALU potremmo prenderlo dopo il suo registro (percorso verde).



Questa cosa si chiama **tecnica del forwarding** e risolve tutti i problemi in cui l'istruzione *i-1* usa una cosa prodotta dall'istruzione *i*, tranne in un caso, la *load*, se avessi una

LOAD R0, R1, #4,

SUB R4, R0, R5,

In questo caso anche se lo facessi tornare indietro e aggiungessi un collegamento dalla data memory al multiplexer (percorso blu) dovrei comunque aspettare il prossimo ciclo di clock (questo perché la load termina dopo la fase di DM per cui non posso fornire il dato alla sub che lo necessita nella fase di execute), dunque non è più sufficiente solo il forwarding ma ho bisogno di fare uno **stallo**, una bolla del pipeline, ovvero faccio passare la load, la sub siccome ha dipendenze la fermo, ovvero smetto di scrivere nel 1 e 2 registro non-architetturale.

Il primo pezzo della pipeline lo freezo, quando al ciclo dopo la memoria dati mi manderà in uscita il risultato corretto allora sblocco la sub ed eseguo l'istruzione.

Uno stallo dal punto di vista del programma assembler, è come fare in assembler:

LDR R0, [R1, #4]

NOP

SUB R4, R0, R5

Nop è l'istruzione che non fa nulla ma fa passare un ciclo di clock; dunque, il tempo per eseguire queste due istruzioni anziché 2 tau sono 3 tau.

Come si implementa la nop?

Si mandano a 0 tutti i segnali di controllo in modo che tutto quello che c'è prima dell'esecuzione della load rimanga com'è senza essere modificato.

Anche i salti sono un problema, immaginando una sequenza di istruzioni 1, 2, 3, …

	t1	t2	t3	t4	t5	t6	t7	t8	t9	t10	t11	t12	
f	1	2	3	4	5	6	7	25	26				
d		1	2	3	4	5	6	7	25	26			
e			1	2	3	4	5	6	7	25	26		
dm				1	2	3	4	5	6	7	25	26	
wb					1	2	3	4	5	6	7	25	26

Supponiamo che la 5 sia un'istruzione di branch, la decodifico, e la devo eseguire ovvero devo fare pc+offset in realtà quando finisco di eseguirla ho che all'uscita della alu ho il pc' che è il valore che devo sostituire al pc iniziale, quindi anche qui posso aggiungere dei collegamenti così da fare tornare il valore fino al program counter (percorso blu immagine sotto) dove ci sarà un multiplexer che dato un valore dirà o prendi il +4 (per passare all'istruzione successiva) o prendi il target del salto (pc').

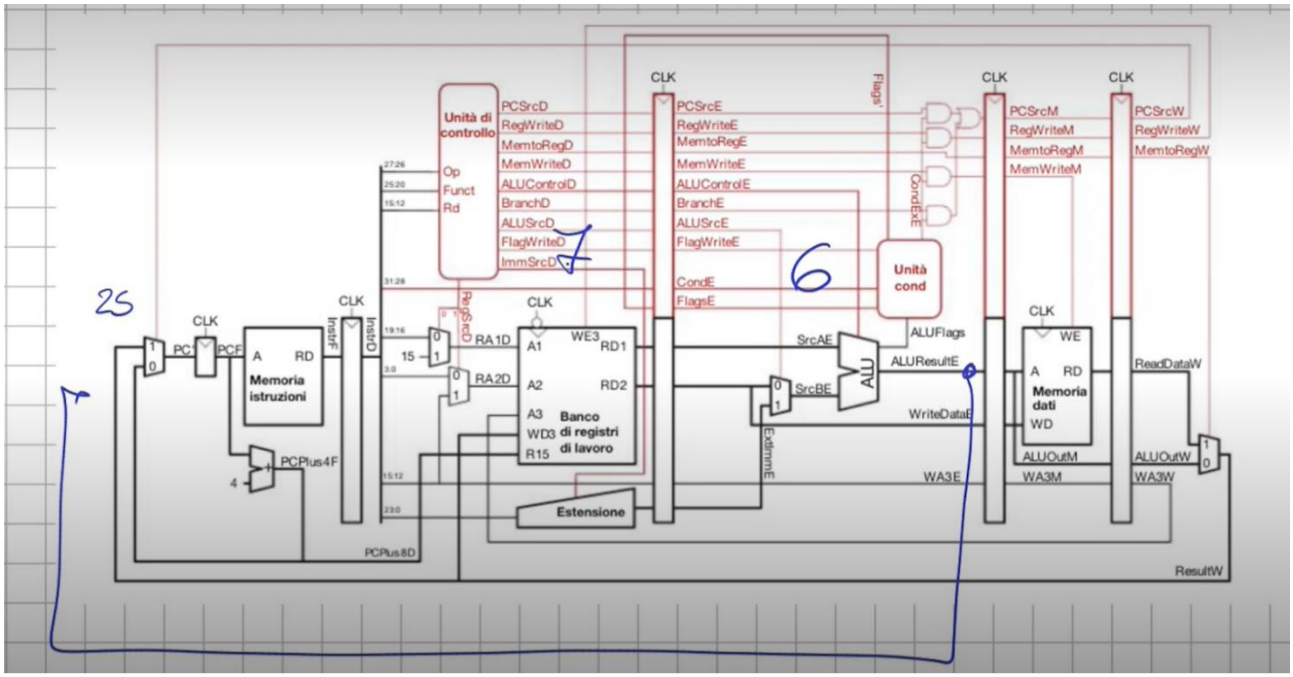
Questa cosa qua accade al tempo t7, e questo ha un effetto su quello che vado a calcolare in t8, infatti se il target del salto è l'istruzione 25 in t8 devo iniziare il fetch di 25.

Il problema è che da quando capisco che devo fare un salto a quando inizio a farlo c'è una bolla da due slot (evidenziata in rosso), in cui probabilmente io sto facendo le istruzioni 6 e 7 che però non dovrei fare, il problema è che sono già nella catena di montaggio, l'unico modo per non farle dunque è stallare il pipeline.

È simile al buttare via i risultati però non è uno stallo vero e proprio in quanto non è fermo, bensì quello che sta succedendo nell'ultimo pezzo lo butto via.

Quando ho capito che il risultato della alu torna indietro, all'inizio ho il fetch dell'istruzione 25, ho la 7 in decode e la 6 in exec.

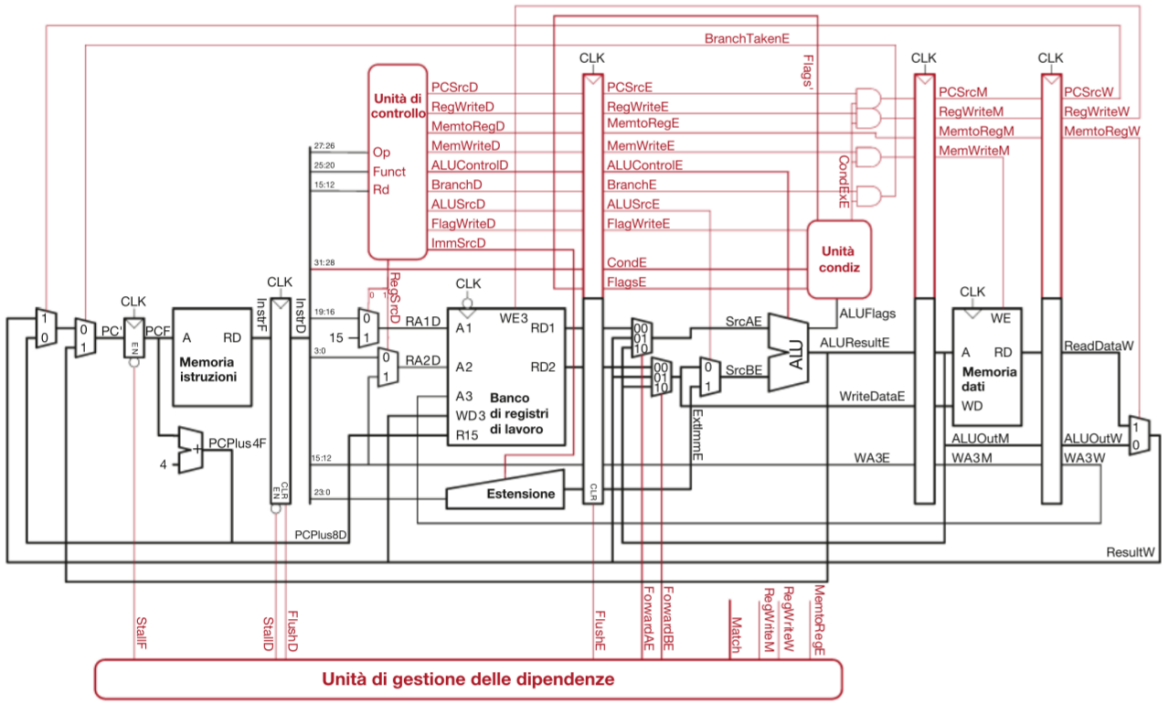
Per stallare la 6 e la 7, devo evitare che vengano scritti i risultati della 6, ovvero devo evitare che vengano scritti gli ALUflag dentro l'unità condizionale, e devo evitare che venga scritto qualcosa nel 3° registro non architetturale, inoltre se la 6 voglio fermarla tutti i segnali sui registri non architetturali 3 e 4 che avrebbero comandato l'esecuzione al ciclo successivo devo eliminarli (azzerarli).



Ci servono ancora due pezzi oltre a tutti i collegamenti extra per fare il forwarding.

Chi deve decidere se fare lo stallo o un forwarding non vede un'istruzione sola ma vede almeno le due istruzioni che hanno causato la dipendenza, ci si deve ricordare un minimo di cronologia delle istruzioni, si aggiunge quindi la **hazard unit**, manda i segnali di controllo che non ci sarebbero se non ci fossero le dipendenze, ma che mi servono per fare o il forwarding o lo stallo, ad esempio i we dove prima non c'erano o i bit di controllo dei multiplexer che non sono comandati dall'unità di controllo (quelli che usiamo ad esempio per scegliere se prendiamo i dati dal percorso normale o dal forwarding).

Lo schema diventa:



L'unità di gestione delle dipendenze se per esempio manda un we al program counter sul percorso StallF, significa che è il bit di controllo per stallare la fase di fetch. ForwardAE e ForwardBE sono dei bit di controllo che dicono se fare il forward del primo o del secondo source.

Se bisogna fare un loop da i = 0; i < n; i++ in arm diventa:

Cmp //condizione

Beq fine

//corpo del for (C)

ADD //incremento il contatore (i)

B loop

Immaginiamo che C (corpo del for) sia uguale ad A[i] = B[i] con A e B aree di memoria.

Per farlo useremo una ldr seguita da una str

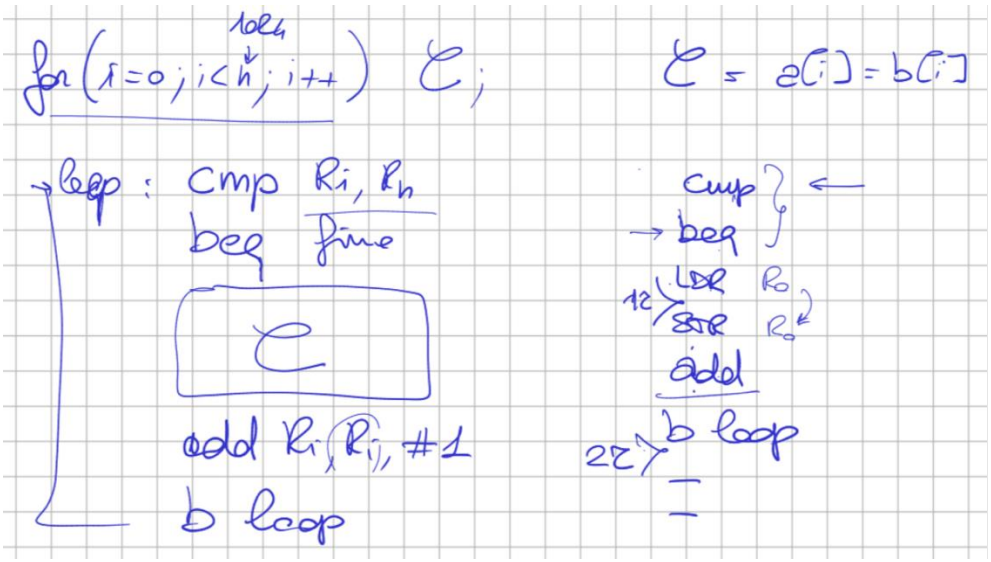
Quante *dipendenze* ci sono in questo ciclo?

La beq è un salto non preso

La ldr carica r0 che viene letto dalla store, questo è il caso in cui si ha un tau in più per avere lo stallo che serve a rendere il risultato della load subito visibile alla store.

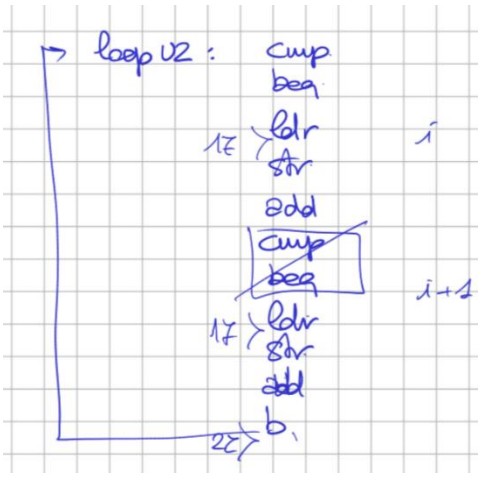
Forse abbiamo un problema anche nella cmp, infatti la cmp è implementata come una sub tra Ri e Rn, buttando via il risultato, questo significa che la fase execute della compare produce qualcosa che deve leggere la beq. I flag non sono un problema unicamente perché l'unità condizionale insieme all'esecuzione legge il contenuto dei flag e quelli che sono i bit delle condizioni, viene letto e processato mentre lo produco, poi genera un segnale che va in and con i segnali che passano dopo, es i we della memoria ecc…

Per eseguire un'iterazione di questo ciclo mi ci vogliono 6 cicli di clock, ma in realtà impiego 6 tau + 1 tau della dipendenza da load + 2 tau della dipendenza sul controllo che fanno 9 tau che è ½ in più rispetto alle istruzioni originarie.



$$6\tau + 1\tau + 2\tau = 9\tau$$

Per risolvere una cosa di questo genere ad hardware possiamo fare ben poco, quello che possiamo fare è un **unrolling del loop**.



Se so che devo fare un for da i a n, posso scrivere questo loop in maniera diversa posso dire:

Ovvero posso eseguire due volte il corpo del for in un unico ciclo, in modo da risparmiare i tau del salto incondizionato alla fine.

Qua abbiamo 2 tau in più, su 11 istruzioni abbiamo 11 tau + 2 tau delle load + 2 tau del salto.

Se so che il numero è pari posso buttare via una cmp e una beq, quindi utilizzo 9+2+2 ovvero 13 tau per fare 2 iterazioni, quando prima ne utilizzavo 18, questa cosa la fa il compilatore.

Questa è un'ottimizzazione del numero di cicli in più che facciamo per via degli stalli e delle dipendenze logiche.

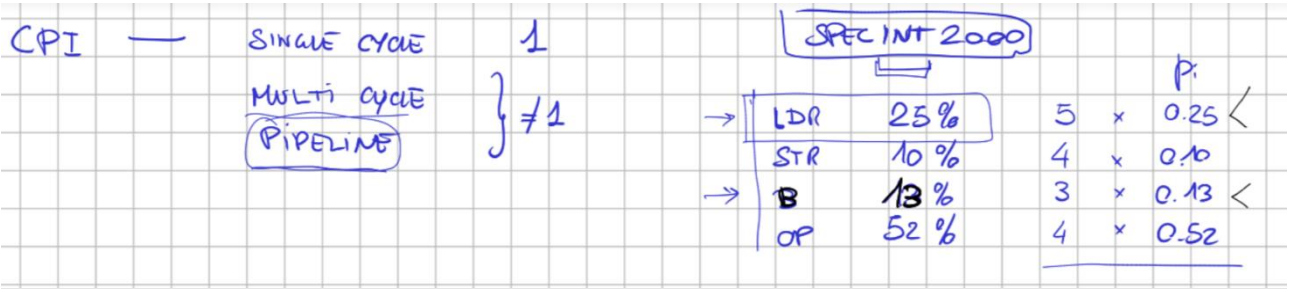
Quando consideriamo il cpi (cycles per instruction) se andiamo a parlare di single cycle, per definizione il cpi è uno, nel multi cycle e nel pipeline questo è un valore diverso da 1, nel multicycle per costruzione, in quanto usiamo pezzi diversi del processore per pezzi diversi del ciclo fetch-execute, qua (nel multi cycle) il cpi dipende esclusivamente da quali istruzioni sono presenti nel nostro codice.

Quando consideriamo il cpi del pipeline, questo dipende sia da quali istruzioni abbiamo sia da come sono messe, in quanto dipende fondamentalmente dal fatto se queste istruzioni inducono o no delle dipendenze logiche e se i salti che abbiamo sono presi oppure no.

Se consideriamo un processore pipeline dobbiamo tener conto di questi due fattori.

Spec int 2000

Il libro riporta un benchmark, spec int 2000



Spec int è un tipo particolare di benchmark che non considera le istruzioni a virgola mobile.

Questo benchmark da un'idea di quante istruzioni vengano utilizzate, lo spec int distingue tra operazioni di load, store, op di salto, e operative, in questa classe di programmi ci sono delle percentuali abbastanza tipiche che dicono che ci sono un 25 % di load, un 10% di store, un 13% di salti e il resto (52%) sono operative.

Se avessi queste percentuali e volessi dire qual è il cpi del multicycle, do per ogni istruzione il numero di tau necessari (ovvero il numero di cicli necessari) e lo moltiplico per la percentuale.

Quando andiamo a considerare un processore pipeline dobbiamo considerare qualcosa di diverso, in particolare non possiamo andare a prendere questi numeri in quanto, di per sé sapere che ci sono un quarto di istruzioni che sono di load non ci dice niente.

Senza nessun tipo di dipendenza né sul controllo né sui dati ognuna di queste istruzioni verrebbe completata in un ciclo di clock perché vanno tutte a catena di montaggio e alla fine di ogni passo della mia catena di montaggio esce il risultato dell'istruzione, dunque dovrei dire che il cpi è 1.

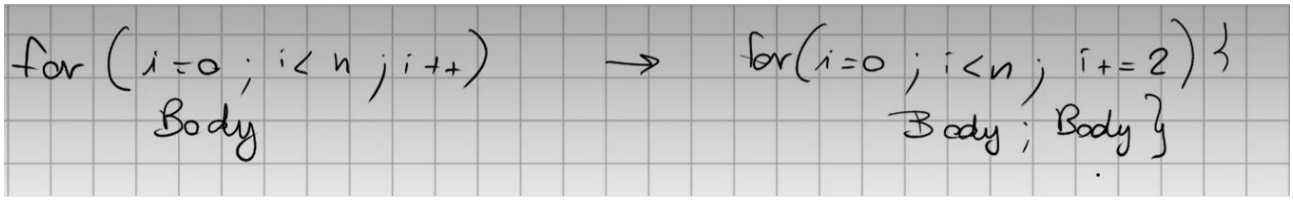
Il problema è se le load sono seguite o no da un'istruzione che utilizza il risultato della load, in quel caso devo aggiungere l'utilizzo di una bolla, lo stesso devo fare per i salti presi, perché ogni volta che ho un salto preso, sono costretto anche utilizzando forwarding nel program counter a perdere 2 cicli.

Il libro ci dice che assumendo che un 40% delle loadr siano seguite da istruzioni che leggono ad esempio il registro caricato, allora in questo caso vuol dire che questo 25% delle istruzioni che dovrebbe avere un cpi di 1, un pochino meno della metà di queste di suo queste istruzioni richiedono 2 cicli di clock, dovrei quindi prendere il 40% del 25% e farlo costare 2 cicli di clock mentre il 60% di quel 25% mi costa 1 ciclo di clock in quanto non è seguito immediatamente da un'istruzione che utilizza il risultato della loadr.

Anche per i salti si ha un discorso analogo, avendo il 50% dei salti presi, questi sono o branch, o branch and link, oppure branch condizionati, si ha che il 50% del 13% pesa 3 cicli di clock, mentre l'altra metà di quel 13% pesa un ciclo di clock.

LOOP E UNROLLING

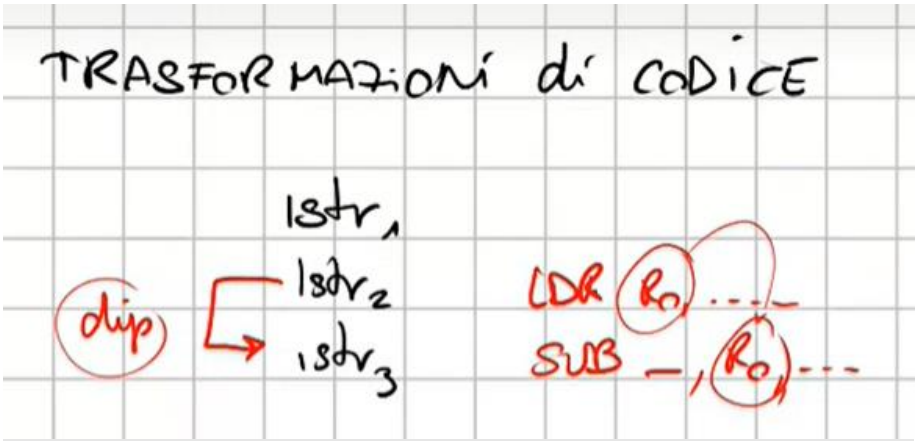
Srotolare un loop è la trasformazione di un loop in questo modo



n deve essere pari, altrimenti devo copiare anche i controlli, dunque il numero di istruzioni raddoppiano.

Nel caso generale possiamo utilizzare molte cose per annullare il peso delle dipendenze, tra cui trasformazioni di codice, ovvero immaginiamo di poter scambiare o cambiare l'ordine di esecuzione delle istruzioni preservando la semantica ma diminuendo il peso delle dipendenze.

Supponiamo che date 3 istruzioni, l'istruzione 2 generi una dipendenza sull'istruzione 3.



<u>Posso cercare un riordinamento delle istruzioni in modo che questa dipendenza non abbia effetto.</u>

Ad esempio, se posso mettere l'istruzione 1 tra la due e la 3. senza che essa interferisca sul loro risultato, avrò che la load carica R0 quindi nel momento in cui R0 lo posso usare per la sub, questa richiesta con l'istruzione nel mezzo accade quando la load sta già producendo il risultato; quindi, con il forwarding la sub può usare il valore nuovo senza creare la bolla che avremmo avuto se la load fosse seguita dalla sub.

Questo si può fare solo quando il riordinamento non altera la semantica del programma.

Condizioni di Bernstein

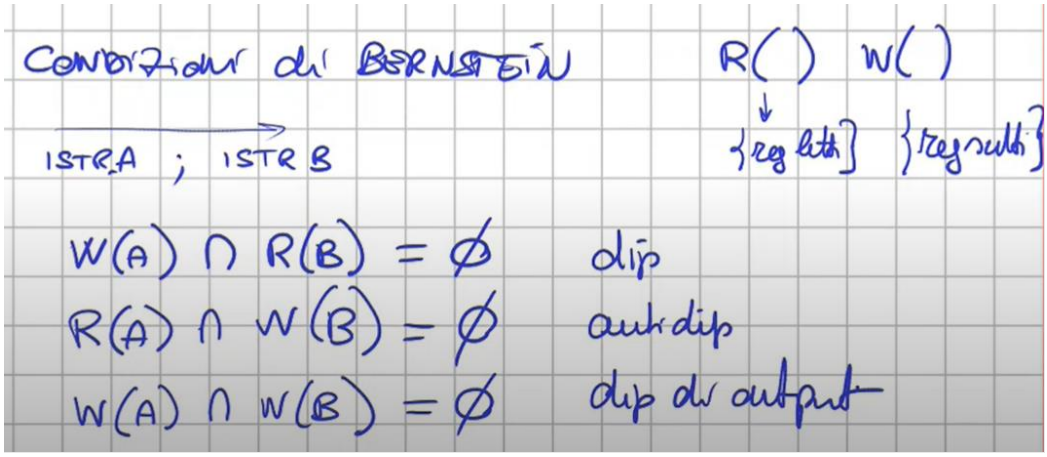
Andiamo a considerare quelle che sono le **condizioni di Bernstein**, ovvero delle condizioni sul dominio e sul codominio di un'istruzione, cioè su quello che un'istruzione legge e su quello che un'istruzione scrive, che dettano le condizioni per cui quelle istruzioni si possono eseguire in un ordine qualunque, in particolare le condizioni di Bernstein dicono questo:

Usiamo R(), e W() per definire l'insieme dei registri letti e l'insieme dei registri scritti, se io ho un'istruzione A e un'istruzione B che in qualche maniera suppongo di eseguire una dopo l'altra, in realtà queste istruzioni posso eseguirle anche nell'ordine inverso se e solo se ciò che scrivo nella A, intersecato ciò che leggo nella B risulta essere l'insieme vuoto.

Devo stare attento anche che quello che legge A intersecato quello che scrive B sia anch'esso vuoto.

Perché invertendole sennò creerei una dipendenza.

Infine, quello che scrive l'istruzione A intersecato quello che scrive l'istruzione B deve essere di nuovo l'insieme vuoto.

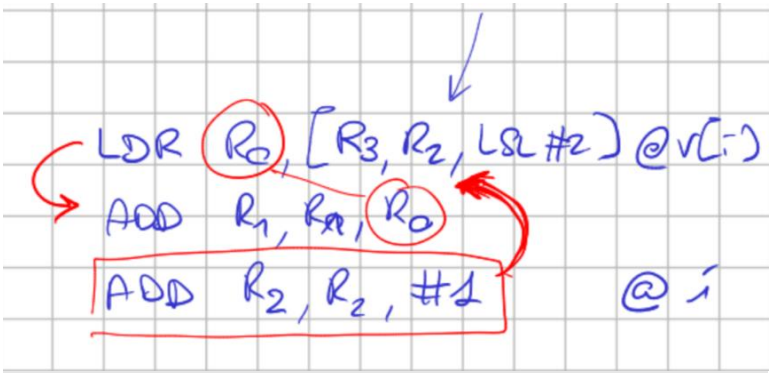


Noi considereremo solo le dipendenze ovvero il primo caso, perché fondamentalmente per le dipendenze di output, se stiamo considerando due cose che scrivano un registro senza che nessuno lo legga nel frattempo, stiamo scrivendo un programma scritto male (sto scrivendo due volte con valore diversi un registro senza che nessuno dopo la prima scrittura lo legga).

L'antidipendenza non la consideriamo perché se anche avessi una cosa che la seconda istruzione scrive e la prima legge se cerco di farle insieme, nello stesso ciclo di clock, non ho problemi, perché quella che va a scrivere in quel ciclo di clock scrive alla fine, mentre quella che va a leggere, legge all'inizio, quindi legge e scrive il valore giusto se eseguite contemporaneamente.

In realtà infatti le istruzioni le consideriamo in parallelo (siamo nel pipeline).

Esempio:



Posso spostare la seconda ADD in quanto il suo readset è R2 e il writeset è R2, il readset di quella sopra è R0, R1 e il writeset è R1 dunque la prima condizione di Bernstein è soddisfatta.

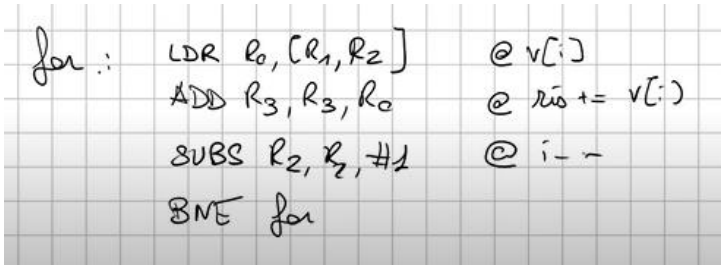
Stiamo facendo un'ottimizzazione del codice assembler che è composta da due passi:

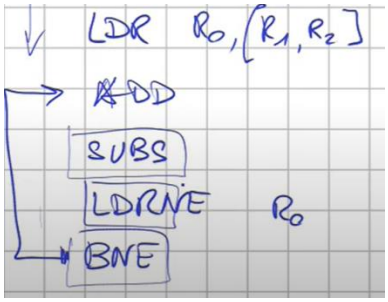
1. Identifico la causa dei ritardi, ovvero vado a vedere dove sono le dipendenze.
2. Trovo uno spostamento di codice che risolva il problema di ritardo

Queste ottimizzazioni non sono garantite ma possiamo provare a farle.

Essendo un riordinamento non modifichiamo il codice come il loop unrolling.

Esempio:





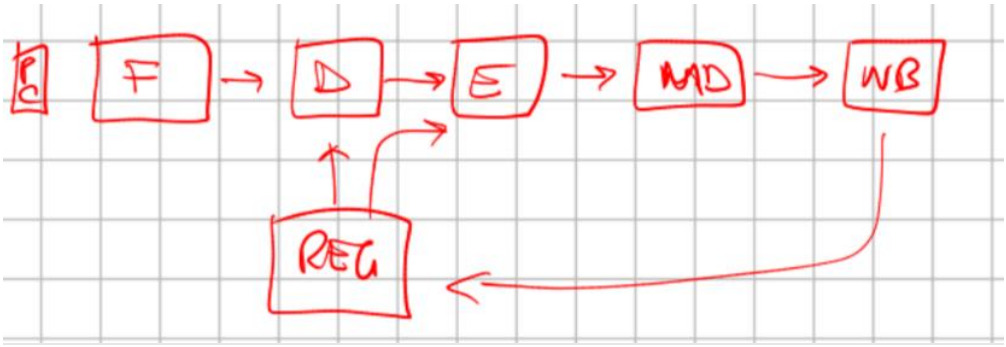
Qua ho sempre il problema della dipendenza che mi crea una bolla, e come prima posso spostare la sub. Sempre a livello delle trasformazioni di codice però posso pensare di fare cose diverse, faccio la ldr fuori dal loop, faccio la add, la sub aggiungo una ldr, e il bne punta alla add.

Quando vado a fare il salto ho comunque una bolla da 2 però mi permette di avere il risultato della loadr che era in R0 direttamente nel registro cercato.

C'è però un problema, in questo caso la loadr va a lavorare su R2, che è lo stesso che uso nella subs, se la subs fosse un'altra operazione avrei un problema di dipendenza.

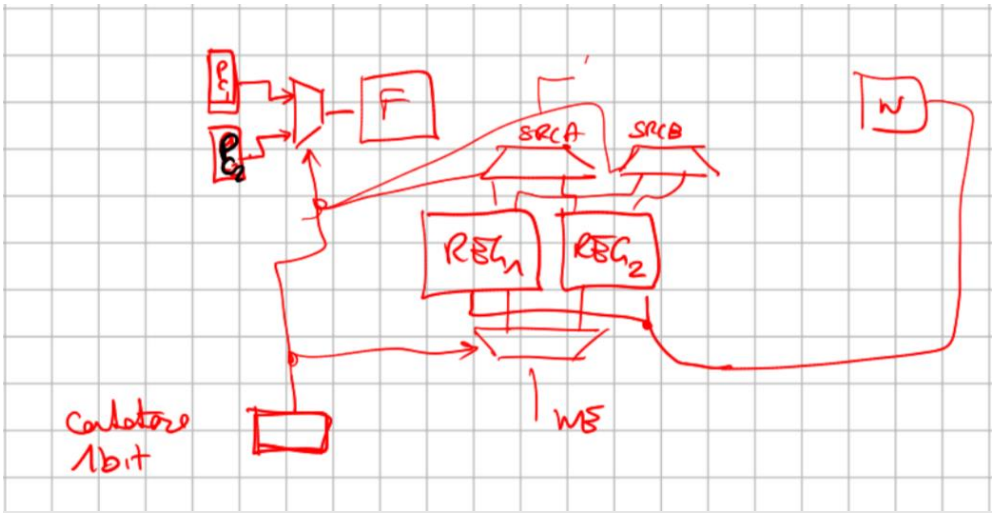
Un modo classico per togliersi il problema delle dipendenze che creano le bolle da 1 è il seguente:

Abbiamo immaginato di avere una pipeline composta dai seguenti stadi,



Questo esegue un flusso di controllo alla volta, supponiamo di avere 2 program counter, e di questi program counter abbiamo un selettore (un multiplexer), che alla fine ha un collegamento con tutti gli stadi (vedere figura sotto) abbiamo poi anche due unità registri, dove sul sourceA e sul sourceB, che sono quelli che vanno a finire nello stadio di esecuzione abbiamo due selettori che scelgono se prendere i valori dal primo o dal secondo insieme di registri, quando andiamo a fare il write back abbiamo un segnale che arriva a tutti e due i registri ma il write enable lo mandiamo con un demultiplexer o all'uno o all'altro.

Supponiamo poi di avere un contatore da 1 bit, ovvero una cosa che ad ogni ciclo di clock fa 0 o 1 e poi di nuovo 0 o 1, questo segnale lo usiamo per comandare tutti i selettori e i write enable che abbiamo aggiunto.



Che cosa fa questa cosa?

Manda due programmi determinati da due program counter diversi, alternando un'istruzione di uno e un'istruzione dell'altro.

Prendendo l'esempio del single cycle abbiamo che al ciclo di clock 1, il contatore vale 0, quindi prendo pc1 il reg1 e faccio le operazioni con quello, secondo ciclo, il contatore vale 1, quindi prendo pc2 (quello sotto in figura, in nero) e reg2, e faccio operazioni con questi.

Quindi vuol dire che io riesco a mandare un'istruzione presa da un programma alternata ad un'istruzione presa da un altro.

Non programmi generici perché la memoria è la stessa per entrambi, non sono processi ma sono thread.

Se in questo caso normalmente nei due programmi avessi delle bolle da 1, queste bolle sarebbero sparite tutte, se dal pc1 tiro fuori una load seguita da una add e dal pc2 una sub e una add, l'ordine di esecuzione delle istruzioni sarebbe: load sub add add

Abbiamo riempito il buco che creava uno stallo.

Questo è quello che fanno alcuni produttori di processori quando fanno *hyperthreading*.

Spostando le istruzioni il problema dei salti rimane, nelle versioni di assembler vecchie del corso c'era un'istruzione chiamata salto ritardato, per risolvere il problema delle istruzioni di salto quello che fanno alcune architetture è di mettere a disposizione dei salti con una semantica particolare, la semantica di questi salti è: fra tot istruzioni vai ad eseguire l'etichetta, c'è di solito qualche flag particolare che si mette sull'istruzione ma immaginiamo di avere un branch con questa semantica e immaginiamo che sia un delayed branch di due posizioni, questo significa che se io scrivo branch a loop seguito da istruzione x e istruzione y, fintanto che calcolo il target del salto eseguo anche x e y, sfrutto i 2 slot di istruzioni che avrei dovuto buttare via perché il salto è preso, ci metto però delle cose che sono comunque logicamente appartenenti a quello che io ho eseguito prima di arrivare al loop, se questo fosse possibile, (e in altri calcolatori lo è) utilizzo quelle due posizioni che sarebbero state una bolla dal punto di vista del pipeline per metterci qualcosa di utile, questo vuol dire che in linea di massima dovrei andare a cercare due istruzioni prima del branch, e spostarle sotto, a patto che non modifichino il pc (che è una delle poche usate dalla branch).

Un’altra cosa che frequentemente si utilizza è la branch prediction, il problema con i salti si ha quando si prendono, i salti incondizionati hanno il problema delle bolle, e i salti condizionati li hanno solo nel caso di condizione presa, la branch prediction cerca di capire da che parte vanno e sostanzialmente si tiene, in qualche maniera, da che parte andavano in modo da avere già il valore dell’etichetta calcolato da qualche parte.

Immaginando di avere del codice for:

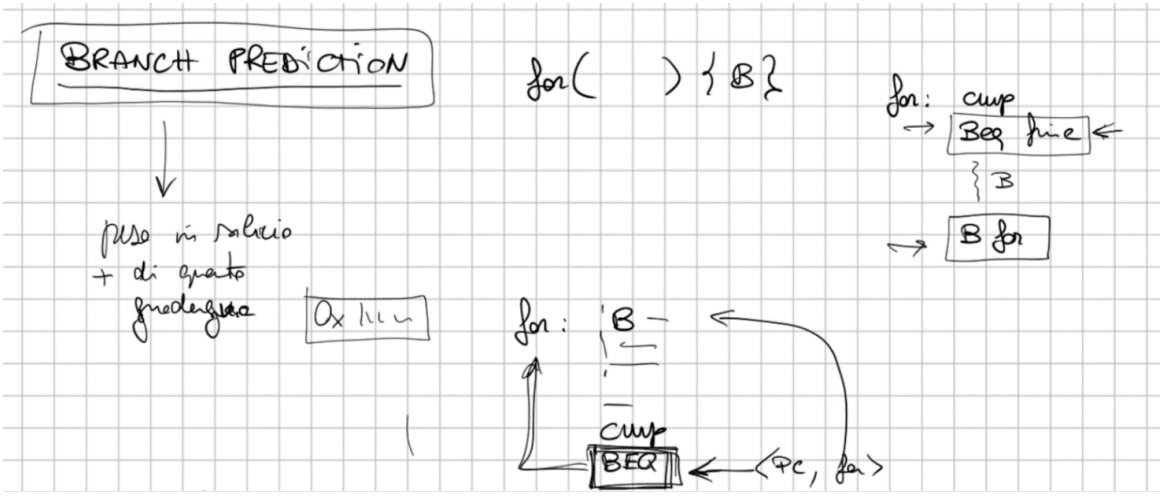
Ho un’etichetta for, faccio una compare per sapere se sono alla fine, un branch if equal o qualche altra condizione, poi ho la compilazione del body del loop e per ultimo aggiorno le variabili di iterazione e faccio un branch a for.

Questa cosa qua ha un salto sempre preso, e un salto condizionale che viene preso un’unica volta quando io esco dal ciclo.

Questo è il caso buono, non serve fare la predizione perché nel 99% il primo è sicuramente un salto non preso (quello di uscita dal ciclo), mentre il secondo è sempre preso in quanto incondizionato.

Posso decidere di fare dei cicli in cui invece dell’etichetta for decido che un for lo faccio perché lo voglio fare almeno una volta.

Per cui un modo che risparmia un po’ di istruzioni è quello di compilare quello che faceva il body del for prima, poi di fare l’aggiornamento della variabile d’iterazione, mettere la compare alla fine e fare un bcond a for, questo in qualche modo risparmia il fatto che a destra (nell’immagine) ho due salti, mentre qua ne ho uno solo, risolvo lo stesso problema, avendo avuto cura di assicurarmi che almeno una volta devo entrare nel ciclo, questa volta ho una beq, ovvero ho un salto condizionale, una volta che dopo il primo ciclo ho capito che devo tornare indietro all’indirizzo di memoria rappresentato dell'etichetta, tutte le altre volte meno una devo ritornarci. Se avessi modo di ricordare l’indirizzo di memoria in cui andare, la prima volta che eseguo la beq me la calcolo, pago i miei due cicli di clock, e salto.



Però una volta calcolata posso utilizzare una cache per ricordarmi l’indirizzo, in realtà quello che mi ricordo è una coppia, <program counter dell’istruzione, valore dell’etichetta>, quando ci ripasso la seconda volta faccio in modo di avere un piccolo adder (?) aggiuntivo in modo che se la coppia è già dentro la cache, in qualche maniera non sto ad eseguire tutta la branch ma mando direttamente quel valore lì e quindi evito di passare (nel ciclo fetch-execute) dalla parte finale di decode, dalla exec e da tutto il write back.

Con il problema che se quella in cui sono è l’ultima iterazione sto facendo una cazzata (eheheh) perché se fosse l'ultima iterazione avrei dovuto saltare la beq, devo avere quindi un meccanismo che in due cicli mi va a considerare la condizione.

Se non è verificata devo disfare quello che ho fatto, perché ho supposto erroneamente che ci fosse da fare un'altra iterazione.

In qualche maniera si ha la necessità di avere una cosa che mi ricorda il target dei salti che ho preso, o comunque le direzioni dei salti, e una cosa che mi permette di disfare quello che sto facendo se la condizione non è più verificata, per disfare il salto devo evitare di scrivere nello stato architetturale, ovvero le scritture del program counter e dei registri devono essere mantenuti in qualche altra struttura fino a che non so se la condizione è verificata o no, e lasciare poi che vengano scritti nei registri.

La branch prediction è una tecnica importante e di fatto ci basta avere 4 stati che riguardano i salti, sono quelli che rappresentano i salti;

sempre presi, quasi sempre presi, quasi mai presi, mai presi

Si può organizzare un automa per cui se sono sempre presi e una volta non lo prendo vado nel quasi sempre preso.

In modo da poterci muovere in questi 4 stati e a seconda dello stato in cui mi trovo, mi comporto in un modo o nell'altro nella microarchitettura, però si ha comunque bisogno di un supporto a livello di componenti, si necessita ad esempio di una cosa che una volta calcolato il target di un salto possa in qualche maniera riutilizzarlo successivamente.

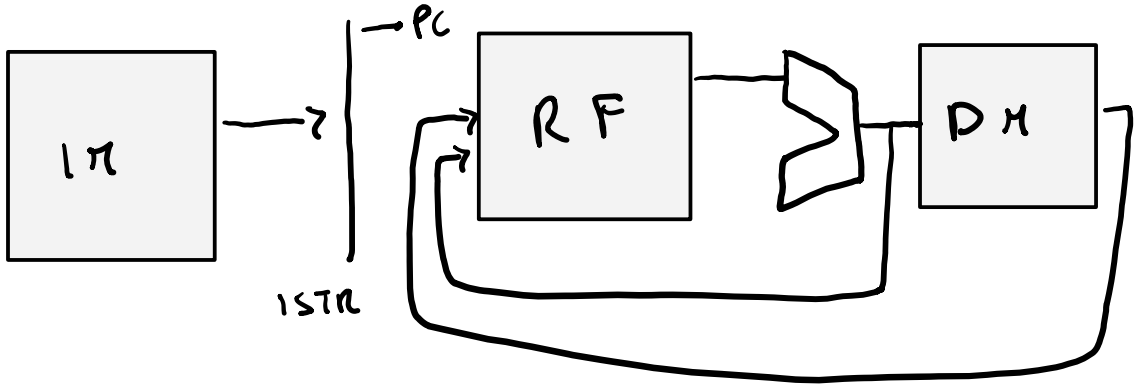
La branch prediction pesa in silicio molto di più di quanto ci permette di guadagnare.

Sul libro c'è un grafico che ci dice che il ciclo di clock non è accorciabile all'infinito, c'è un minimo che quando lo oltrepasso per via dei ritardi non può più essere accorciato.

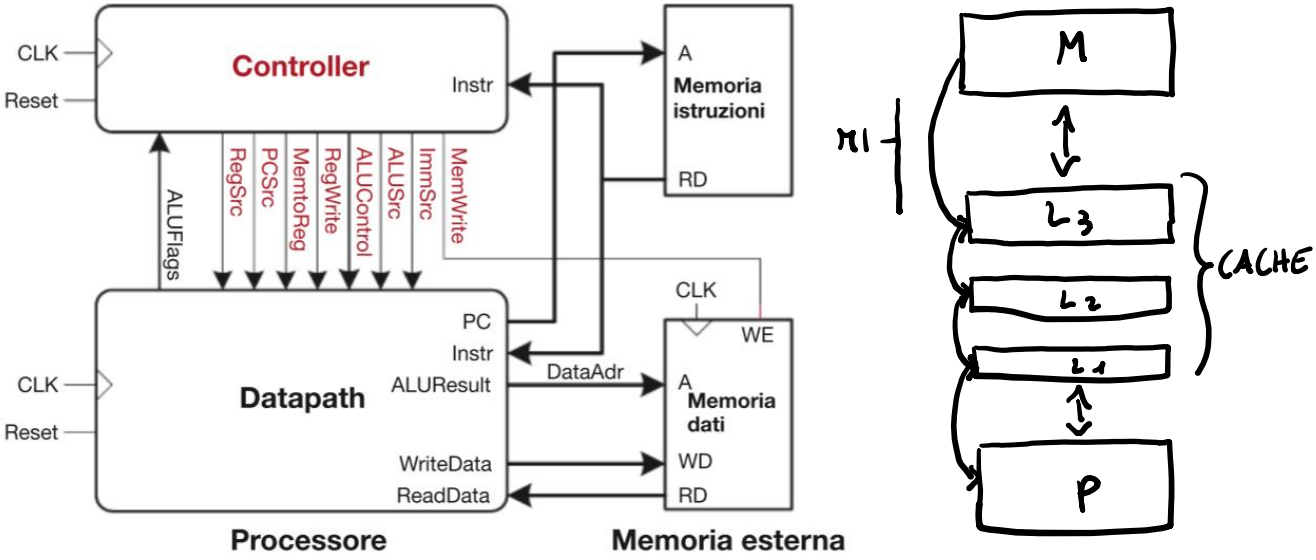
Non si possono fare le cose arbitrariamente, i componenti di calcolo posso anche pensare di farli a pipeline (le alu a virgola mobile lo sono quasi tutte) quelle che hanno stato come il file dei registri e le memorie, si possono fare un po' più veloci riducendole ma alla fine il limite è il tempo che impiego a leggere o scrivere e oltre quello non posso andare.

Memoria (microarchitetture del libro vs. modelli reali)

Le memorie che abbiamo visto erano blocchi dove indirizzo, un dato, o quello che passo in ingresso viene scritto nella cella se il we è 1. Il problema di queste cose è che sono all'interno dell'architettura.



Le memorie (IM e MD) non sono lì, le due memorie sono componenti grandi, vengono infatti usate come componenti esterne, quello che ci mostra il libro è che lo schema del single cycle è il seguente



Il datapath è quello di prima, ci sono le stesse componenti ma non c'è la data memory, quelli che sono i collegamenti con la data memory sono evidenziati in nero, lo stesso vale per la instruction memory.

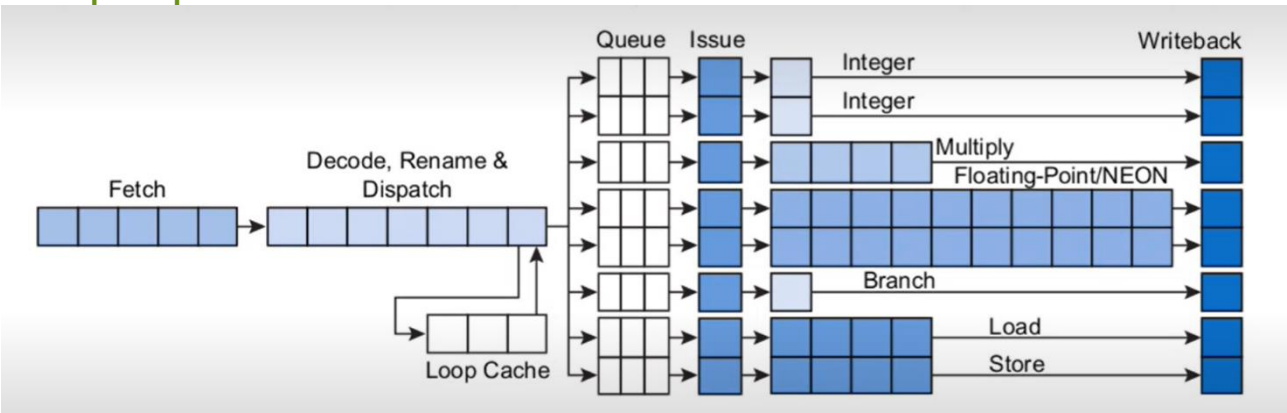
La data memory e la instruction memory in realtà non sono memorie bensì delle gerarchie di memorie, ovvero stack di memorie di cui quella che sta in cima, la più lontana da quello che è il nostro processore è veramente una memoria base. Tra questa memoria e i registri che stanno al livello del processore nel mezzo ci sono livelli di memoria di altro genere che chiamiamo memorie cache. Le cache non sono altro che memorie più piccole e più veloci, che contengono il working set, ovvero i dati e le istruzioni che servono in un certo momento al processore per andare avanti nel programma.

Le cache più basse di livello sono dispositivi che riescono a dirti se stai cercando di leggere o scrivere un dato in 1-2 cicli di clock, la cache di 2 livello ce ne mette 6-7.

Riusciremo a ragionare come se tutto quello che abbiamo si trovasse al livello uno.

Per accedere ai livelli superiori devo attendere un po' quindi ho bisogno di fare qualcosa al livello del processore che implementi i tempi di attesa, questi rallentamenti però rallentano appunto l'esecuzione del data path. Questa situazione non è diversa dalla situazione in cui abbiamo delle dipendenze nel ciclo di pipeline.

Deep Pipeline



Cosa succede realmente nel numero di stadi del pipeline?

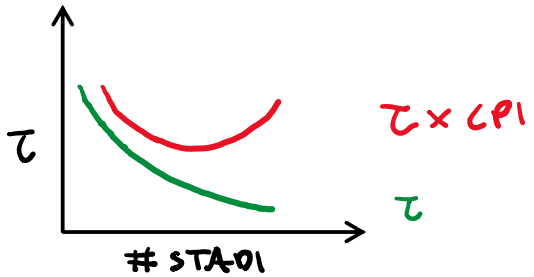
Noi ne immaginavamo 5, questo in figura è un processore arm, dove però ogni quadratino tranne le code, rappresentano le cose che cambiano in un ciclo di clock ovvero uno stadio.

Per fare il fetch di un'istruzione abbiamo 5 stadi, Decode Rename & Dispatch sono degli stadi che capiscono cosa fa quell'istruzione e si preparano per l'esecuzione, una volta capito qual è l'operazione da eseguire, ci sono gli altri stadi che la implementano.

Se facciamo una moltiplicazione ci vogliono un po' più di stadi, ancora di più se abbiamo la virgola mobile, nel caso più lungo abbiamo 24 stadi il che vuol dire 24 cicli di clock per fare un'operazione.

Ogni stato è piccolo ed è stato spezzato perché semplice, tra gli stati ci saranno dei registri architetturali dove prendere e scrivere i dati (come nel pipeline classico).

Se riesco a spezzare l'operazione in tante fasi e riesco a spezzarla in *n* pezzi ci metto 1/*n* del tempo, più li spezzo più riesco ad ottenere un tau molto corto.



Riporto sul grafico la lunghezza del ciclo di clock e il numero di stadi, più stadi ho, più il ciclo di clock scende (verde).

Ci sono però dei problemi come ad esempio il costo di realizzazione del processore, il problema sono che avendo delle dipendenze e avendo aumentato il numero di stadi ho aumentato il numero di processi intermedi che ho effettuato senza doverli fare.

La gestione delle dipendenze si complica molto.

L'andamento tau*cpi (rosso) ad un certo punto scende ma poi aumenta in quanto il cpi aumenta più di quando non diminuisca la lunghezza del ciclo di clock.

Micro Operazioni

LDR R2, [R3], #4

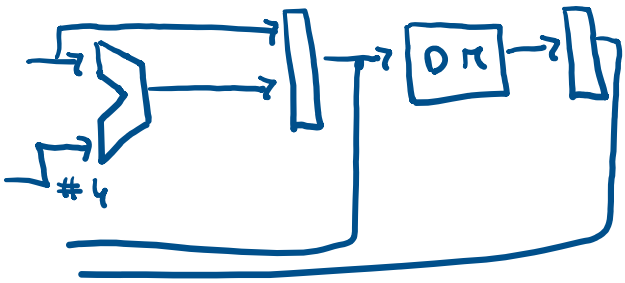
⇩

LDR R2, [R3]

ADD R3, R3, #4

$R_2 \leftarrow MEM[R3]$

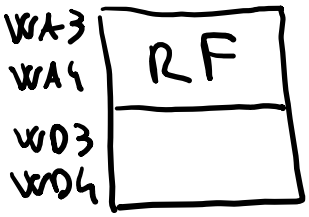
$R_3 \leftarrow R_3 + \#4$



L'esecuzione di questa istruzione ha bisogno di due scritture all'interno del register file, che però noi non possiamo fare.

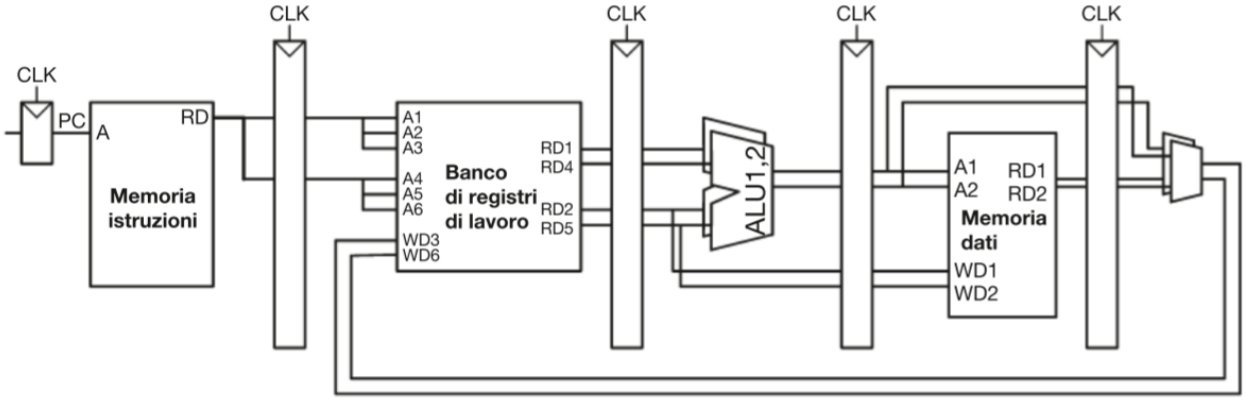
Nella fase di write back dobbiamo poter avere la possibilità di scrivere nel register file sia quello che arriva dalla memoria, sia quello che arriva dalla alu.

Devo avere qualcosa che nel register file abbia non uno ma due indirizzi di scrittura.



Tutti i processori moderni hanno definito delle micro-operazioni, che possono essere utilizzate per comporre cose più complicate, come la branch and link, che deve prendere un target e deve recuperare un *program counter + il valore del target* e contemporaneamente il vecchio *program counter + 1* deve andare a finire nel link register, queste sono due micro operazioni che compongono l'operazione di branch and link, se le facessimo insieme richiederebbero l'uso doppio dell'alu che al momento non abbiamo.

Architettura superscalare



Finora ci siamo mossi nell'idea di avere il parallelismo temporale, questa architettura pretende di usare il parallelismo spaziale, cerchiamo dunque di fare più istruzioni alla volta.

Usiamo una memoria modulare interallacciata, che ci permette di avere due istruzioni alla volta individuate da pc e pc+4, ciascuna di queste due istruzioni ha bisogno di avere degli operandi, dunque ho bisogno di un register file con 6 entrate e 4 uscite (vedere figura), ho poi due alu.

La memoria dei dati se vogliamo poter fare due istruzioni di load e due di store deve anche lei avere più ingressi, qua le cose sono un po' più complicate in quanto non è detto che basti una memoria modulare interallacciata perché se devo andare a scrivere due cose che sono nello stesso modulo, ho bisogno di spendere due cicli di clock. Infine, ho la fase di write back e devo riscrivere due cose all'inizio.

Il problema delle dipendenze persiste, due istruzioni dipendenti possono essere eseguite una insieme all'altra, dobbiamo assicurarci quindi che non ci siano istruzioni dipendenti che vengono eseguite contemporaneamente in quanto qua a differenza del pipeline il forwarding non funziona.

ADD R8, R1, R2
SUB R8, R1, R8
} NOP

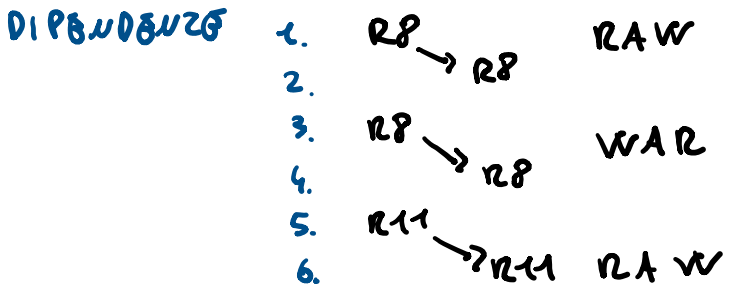
Tutte le dipendenze sono esasperate, quando ho una dipendenza come quella sopra, o metto una nop tra le due istruzioni, oppure usiamo una di quelle tecniche che abbiamo visto per il pipeline, possiamo effettuare secondo le condizioni di Bernstein un riordinamento delle istruzioni in modo che venga eseguita quell'istruzione al posto della sub e fare poi un forwarding.

Anche la hazard unit, diventa più complicata in quanto non deve gestire una sequenza di un'istruzione per ciclo bensì di due istruzioni per ciclo.

Questo ci pone un po' di problematiche e la possibilità di aprire ad altri due tipi di concetto.

Dato uno stream di istruzioni cerco di realizzare un'esecuzione *out of order*.

1. **LDR R8, [R0, #40]**
2. **ADD R9, R8, R1**
3. **SUB R8, R2, R3**
4. **AND R10, R4, R8**
5. **ORR R11, R5, R6**
6. **STR R7, [R11, #80]**



Abbiamo una read after write e una write after read, e poi una nuova read after write. Possiamo però fare degli scambi, se spostiamo la penultima istruzione (5) tra la (1) e la (2), la loadr con la orr le posso mandare insieme, a questo punto mi rimangono la add e la sub che però non posso mettere insieme.

La store una volta che è stato prodotto il risultato della orr può essere mandata insieme alla and.

LDR R8, [R0, #40]
ORR R11, R5, R6
ADD R9, R8, R1
STR R7, [R11, #80]
SUB R8, R2, R3
nop
AND R10, R4, R8
nop

Per fare questo abbiamo bisogno di due cose, la prima è una cosa che mi permette di vedere lo stream delle istruzioni così da avere una visione globale di quello che bisogna fare, e l'altra devo avere un modo per analizzare le istruzioni e calcolare le condizioni di Bernstein per capire se sono interscambiabili o no.

L'unità che fa questa cosa si chiama *scoreboarding* e lavora in parallelo con il datapath. Questa unità ha però un costo.

L'altra tecnica è il *register renaiming*, in alcuni casi succedono delle cose particolari, in questo esempio sia la load che la sub usano lo stesso registro R8, questo da' fastidio perché se voglio fare degli spostamenti di codice quel registro è lo stesso.

Io in questo caso avrei potuto usare un altro registro in quanto vado a sovrascrivere il valore; dunque, avrei potuto scegliere un qualsiasi altro registro.

L'architettura mette a disposizione dei registri temporanei (sono 20 registri da 0 a 18) che non possiamo nominare, e sono usati dall'unità di scoreboarding per impedire ad una cosa che sarebbe una dipendenza di avere effetto dopo il rimescolamento delle istruzioni.

Posso ad esempio rinominare il registro R8 come se fosse il temporaneo T_0, purchè tutte le cose che usano R8 dopo la sub usino T_0.

Istruzioni Thumb

Sono istruzioni codificate su 16 bit anziché su 32, sono state fatte un certo numero di scelte che hanno permesso di trasformare le istruzioni, innanzitutto non possiamo accedere a tutti e 16 i registri contemporaneamente bensì solo ad 8, in modo da avere 3 bit anziché 4 per rappresentare gli operandi, di solito le istruzioni possono far riferimento allo stesso indirizzo sorgente e destinazione, ovvero ADD R0, #4 non thumb sarebbe ADD R0, R0, #4 butto via uno dei registri e risparmio altri 3 bit, altra cosa attuata è quella di usare costanti più corte.

Buttiamo inoltre via le istruzioni condizionali.

La *compare* diventa l'unica istruzione a settare i flag, le uniche istruzioni che rimangono condizionali sono i salti.

15															0	
0	1	0	0	0	0	funct			Rm		Rdn					<funct>S Rdn, Rdn, Rm (elaborazione dati)
0	0	0	ASR		LSR	imm5			Rm		Rd					LSLS/LSRS/ASRS Rd, Rm, #imm5
0	0	0	1	1	1	SUB	imm3		Rm		Rd					ADDS/SUBS Rd, Rm, #imm3
0	0	1	1	SUB		Rdn		imm8								ADDS/SUBS Rdn, Rdn, #imm8
0	1	0	0	0	1	0	0	Rdn [3]	Rm		Rdn[2:0]					ADD Rdn, Rdn, Rm
1	0	1	1	0	0	0	0	SUB	imm7							ADD/SUB SP, SP, #imm7
0	0	1	0	1	Rn			imm8								CMP Rn, #imm8
0	0	1	0	0	Rd			imm8								MOV Rd, #imm8
0	1	0	0	0	1	1	0	Rdn [3]	Rm		Rdn[2:0]					MOV Rdn, Rm
0	1	0	0	0	1	1	1	L	Rm		0	0	0			BX/BLX Rm
1	1	0	1	cond			imm8									B<cond> imm8
1	1	1	0	0	imm8											B imm11
0	1	0	1	L	B	H	Rm		Rn		Rd					STR(B/H)/LDR(B/H) Rd, [Rn, Rm]
0	1	1	0	L	imm5				Rn		Rd					STR/LDR Rd, [Rn, #imm5]
1	0	0	1	L	Rd			imm8								STR/LDR Rd, [SP, #imm8]
0	1	0	0	1	Rd			imm8								LDR Rd, [PC, #imm8]
1	1	1	1	0	imm22[21:11]											1 1 1 1 1 imm22[10:0] BL imm22

Thumb esiste perché nei micro controllori avere le istruzioni da 32 bit anziché da 16 è uno spreco, queste sono utili però anche per implementare i super scalari in quanto possiamo rappresentare due istruzioni con le stesse dimensioni di memoria normali.

Multithreading

I thread sono flussi di controllo ovvero parti di codice che appartengono allo stesso programma e che lavorano sulla stessa memoria in modo indipendente.

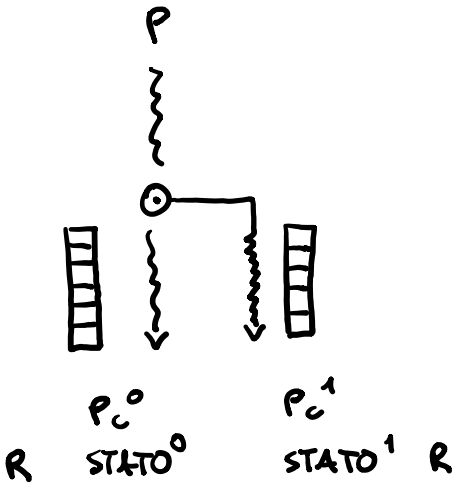
Come possiamo realizzare questo nel processore?

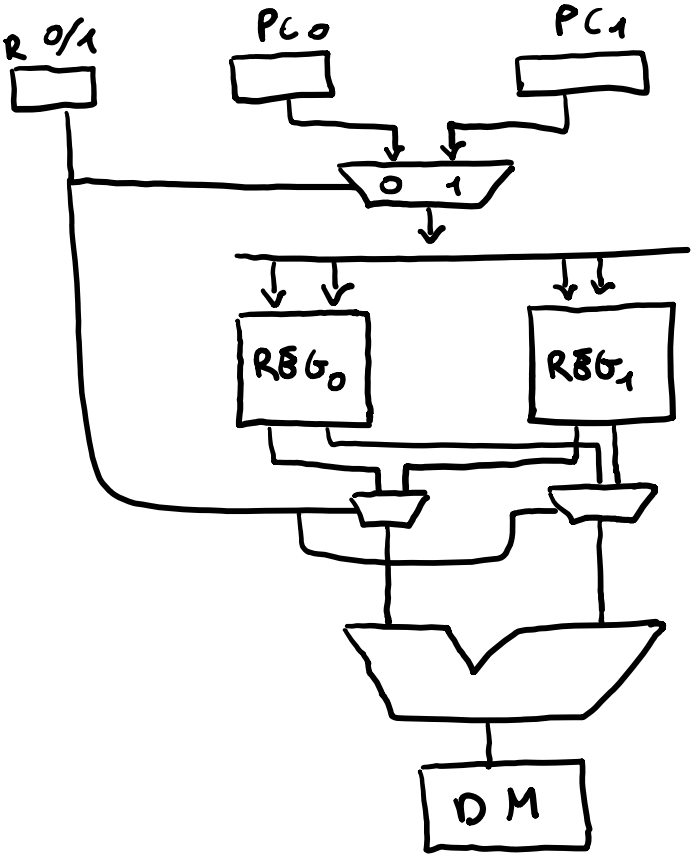
Ognuno avrà un proprio program counter e un proprio stato.

Potrei immaginare un meccanismo che dice che va avanti ad eseguire quel thread fintanto che quel thread non si blocca, quando si blocca va ad eseguirne un altro.

Cosa serve per fare questo?

Un meccanismo per riconoscere se il thread si è bloccato e un meccanismo per passare da un thread all'altro.





Posso immaginare un multiplexer che sceglie o il primo o il secondo PC, c'è poi la parte di decodifica che manda i segnali a tutti e due i register file, le cui uscite vanno in due multiplexer, comandati con la stessa cosa che comandava il multiplexer iniziale, quanto costa passare da un thread all'altro? Praticamente 0, in quanto basta cambiare il bit di controllo del multiplexer che esegue le scelte.

Sistema operativo

Vedremo le gerarchie di memoria, vedremo poi come vengono trattate le unità di I/O, e poi vedremo il concetto di sistema operativo che non è altro che una collezione di software che permettono di presentare al livello superiore un'interfaccia ovvero un'astrazione che permette di avere delle system call che interagiscono con il SO.

Parleremo poi di processi e thread, di protezione, e di interruzioni.

Parleremo di traduzione di indirizzi e memoria virtuale.

Parleremo di concorrenza e sincronizzazione, scheduler, storage e file system

Gerarchia di Memoria

Se non ci fossero le gerarchie di memoria tutto quello che abbiamo non funzionerebbe, tutto quello che adesso viene fatto in un secondo andrebbe moltiplicato per un fattore 100.

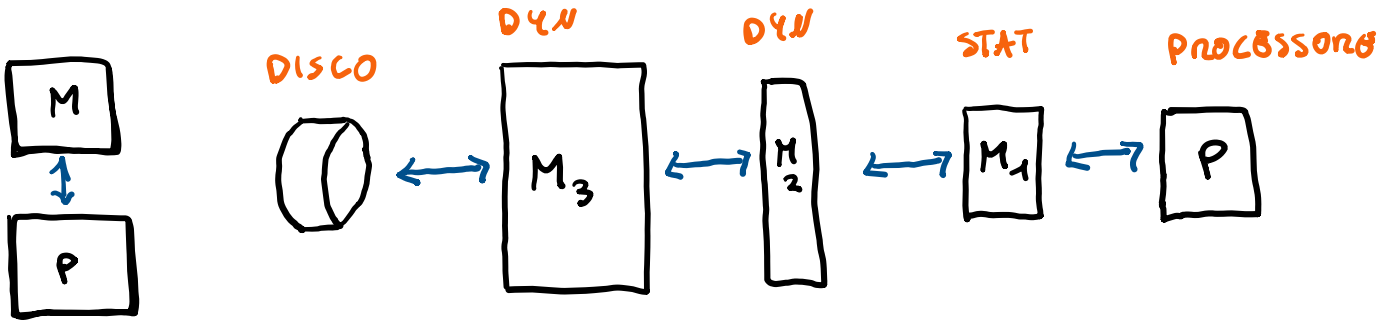
Siamo partiti dai flip flop oggetti che hanno tanti componenti ma che riescono ad essere molto veloci siamo infatti in ordini di tempo del nanosecondo. Con i flip flop abbiamo fatto dei registri i cui tempi sono rimasti più o meno i soliti, sopra a questo abbiamo la ram statica SRam questa è quella in cui il singolo bit di informazione era fatto da un ordine di porte che era sopra la decina e che riescono a garantire anche loro, ordini di nanosecondi, abbiamo poi la DRam ovvero la dynamic ram, queste ultime costano molto poco in termini di componenti anche se a questo punto i tempi di accesso sono dell'ordine di 60/80 nanosecondi.

Quando la memoria non basta più si va sul disco, abbiamo infatti la memoria secondaria dove abbiamo i dischi, che sono di due tipi, hard disk e ssd, hanno tempi di accesso dell'ordine del millisecondo per gli hd, un po' meno per gli ssd. Molto spesso non possiamo mettere quanti hard disk vogliamo, in quanto la mole di dati da memorizzare a volte è troppo grande.

Il modello di macchina che consideriamo è quello di von Neumann, in questo modello abbiamo due componenti M e P dove nel P abbiamo la control unit e il data path, Il processore accede alla memoria tramite un unico canale di comunicazione in cui passano tutti i codici delle istruzioni, da una parte questo canale è critico perché è un po' il collo di bottiglia della situazione, dall'altra parte, posso scegliere di costruire M come voglio.

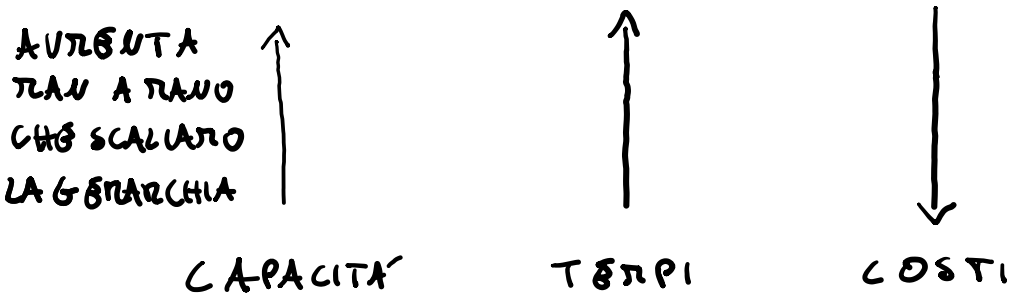
La memoria su disco costa poco, il problema è che se usassi solo questa ram, ogni volta per eseguire un'istruzione pagherei il tempo di attesa che è il più elevato tra le altre.

La ram dinamica è una via di mezzo tra le due (SRam e Disco) , costa un po' di più del disco ma non è così densa come la tecnologia che si usa per la memoria statica, potremmo usarla ma non tutta per fare la memoria.



Ci immaginiamo dunque una gerarchia di memoria fatta in questo modo, abbiamo il processore, una memoria piccola M1 una memoria un po' più grande M2, una ancora più grande M3, e quando non abbiamo più livelli da sfruttare mettiamo il disco.

Questi livelli hanno capacità crescente andando verso l'alto (sinistra), lo stesso per i tempi di accesso, come costi invece abbiamo qualcosa che cresce andando verso il basso (destra)



Per esempio per il livello più vicino al processore potremmo usare ram statica, al secondo e terzo livello potremmo avere ram dinamica e all'ultimo livello un disco.

Per cercare i dati si cerca di livello in livello (salendo) finchè non viene trovato il dato, ogni livello mantiene una copia del dato e poi scendiamo verso il processore.

Applicando i principi di località spaziale e temporale posso sperare che questa istruzione mi serva ancora, così da poterla chiedere subito al primo livello dopo averla memorizzata e pagando dunque meno in termini di tempo.

La località spaziale e temporale ci dice che quando vado ad eseguire un certo codice su un programma, i dati e le istruzioni si comportano in maniera tale da essere utilizzati in locale.

Località spaziale vuol dire che se io sto eseguendo delle istruzioni o sto accedendo a dei dati, nell'immediato futuro molto probabilmente andrò ad accedere ad istruzioni vicine all'istruzione che stavo eseguendo, se sto accedendo ad un dato molto probabilmente accederò ai dati vicini.

La *località temporale* mi dice che se ho utilizzato un certo dato o un'istruzione, questo dato o questa istruzione molto probabilmente saranno riutilizzati.

Normalmente quando vado a chiedere un dato non mi limito a prendere quel dato o quell'istruzione, in realtà prendo un blocco di cose lì attorno.

Posso immaginare che avendo chiesto I1 vicino al processore abbia un blocchetto di istruzioni.

Quali sono i problemi dal punto di vista concettuale?

C'è un problema di capienza, ma soprattutto il problema è che se io ho un'istruzione nell'ultimo livello, le istruzioni sono indicizzate, se le sposto in altri blocchi quei blocchi non sono più ad accesso diretto, come faccio ad indirizzare i nuovi blocchi che sposto?

Ci serve un meccanismo di indirizzamento che permetta di fare un'operazione molto velocemente, ovvero mi serve sapere se il dato che cerco c'è o non c'è.

Tutto questo modo di mantenere la gerarchia di memoria non è a carico del programmatore, bensì è a carico dell'hardware, il primo livello di memoria che è una cache ha un meccanismo per riconoscere se il dato c'è o non c'è e se non lo trova si occupa di andarlo a cercare nel livello superiore, questa cosa da l'illusione di avere tanta memoria quanta ne abbiamo in cima (disco) alla velocità che abbiamo in fondo (cache) e ad un costo che è più simile a quello di sopra che a quello di sotto.

SRAM	DRAM	FLASH	DISK
0,5 - 1 nSec	10 - 50 nsec	20K nsec	5 msec
$/GB 5000	$/GB 7	$/GB 0,4	$/GB 0,05

Amat è la formula del tempo medio di accesso che andiamo a spendere per andare in memoria (*average memory access time*) è quello che spendiamo effettivamente dal processore per andare ad accedere ad una singola locazione di memoria.

H sta per hit rate, ovvero qual è la probabilità che troviamo qualcosa in un livello.

Se becco subito quello che cerco nel primo livello ho semplicemente un hit time pari a 1, se non la trovo là dentro vado a pagare la probabilità di trovarla nel secondo livello e così via.

Questo funziona tanto meglio quanto alto riusciamo a tenere l'hit rate. Solitamente abbiamo valori del 98-99% .

Se consideriamo di avere un hit rate del 90% con un tempo di accesso di 1 ns, e di avere sopra un solo altro livello che contiene il 10% dei casi, il cui accesso però costa 100 ns, abbiamo dunque un tempo di accesso che è circa 11 ns.

$$t_a = h_1 t_1 + (h_2 t_2 + h_3 t_3)$$

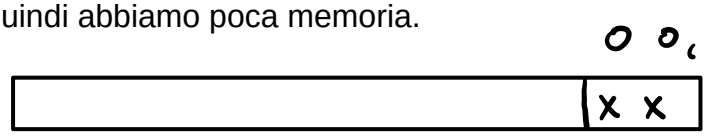
Località temporale: se uso qualcosa devo tenerlo perché ci sta che venga riutilizzato tra poco.

Località spaziale: uso l'indirizzo e anche gli indirizzi che stavano attorno, questa ha anche un'altra conseguenza, abbiamo che l'interfaccia da P e M fa viaggiare una sola parola, tutte le altre interfacce invece sono un certo numero di parole per permettere appunto questo meccanismo.

Assumiamo che un blocco siano 4 parole ovvero 16 byte quando faccio la load l'indirizzo sono 32 bit di una configurazione qualunque, io considero gli ultimi bit, siccome le cache ragionano in termini di blocco dato questo indirizzo lo voglio andare a prendere i blocchi di memoria attorno, che sono 4 parole.

Prendo quindi le 4 parole che hanno gli ultimi 2 bit diversi tra cui quella che avevo passato.

Il livello 1 sono 16/32 kb per i dati e altri 16/32 per le istruzioni, sono due cache separate per istruzioni e dati, quindi abbiamo poca memoria.



Se chiedessi il fetch dell'istruzione 5, 5 finisce con 01, vado a caricare dunque 00, 01, 10, 11

Memorie cache

Immaginiamo n = 4 **For(i = 0; i < n; i++)** **S+=v[i]**

```
            mov R0, #0
loop:       cmp R0, R1
            beq end
            ldr R4, [R3, R0, lsl #2]
            add R2, R2, R4
            add R0, R0, #1
            b loop
end:
```

Immaginiamo di avere sopra la cache una memoria interallacciata che ci permette di scegliere 4 parole alla volta, che però devono essere allineate sui blocchi.

La prima cosa che facciamo è tentare di fare il fetch della mov, cerchiamo quindi nel primo livello di memorie la mov, che però non trovo, chiedo quindi sopra l'indirizzo 0 e mi viene risposto con i primi 4 indirizzi (corrispondenti alle prime 4 istr) che vengono salvati nel livello inferiore.

Assumiamo che il compilatore sia in grado di allineare i vettori, così che comincino sempre allineati.

Assumiamo che la cache non finisca la memoria, per questo ci bastano solo 3 letture della parte di memoria sopra.

Per eseguire questo codice ho fatto tutti gli accessi che dovevo fare, 3 li ho pagati tanto (lettura delle 7 istruzioni e del vettore), tutti gli altri li ho pagati il tempo di accesso alla cache, il costo di accesso in memoria per 4 iterazioni sono 31 accessi di cui (3+1) pagati tanto e 27 accessi in cache.

Qual è la località spaziale? Il vettore e la condizione del for.

Qual è la località temporale? Sulla variabile i sulla variabile s e sul codice.

I trasferimenti di dati sono tra processore e il primo livello di cache, tra livelli di cache inferiori e livelli di cache successivi, poi ad un certo punto abbiamo la memoria ed infine il disco.

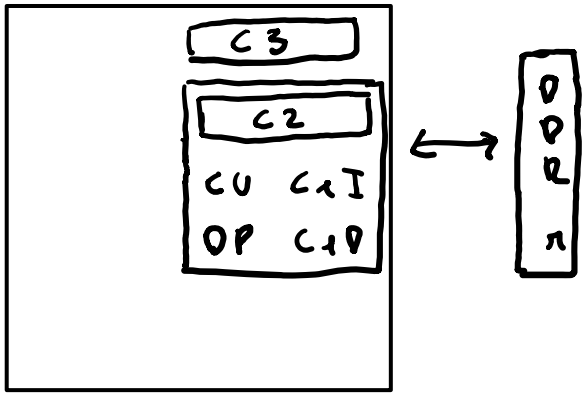
Chi si occupa di questi trasferimenti? I primi trasferimenti sono merito del compilatore, scrivo infatti roba nei registri o nell'instruction register.

I trasferimenti tra i livelli di cache e tra cache e memoria sono hardware, dal punto di vista del processore non succede nulla.

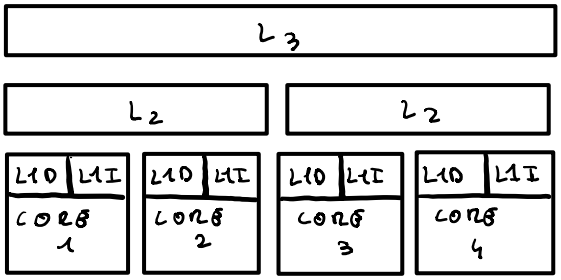
Quando arriviamo alla memoria centrale le interazioni tra quest'ultima e il disco sono gestite dal sistema operativo, se viene richiesto qualcosa che in memoria non c'è viene provocato un errore che scende giù tra i livelli e arriva al processore, quest'ultimo se ne accorge e manda in esecuzione quel pezzo di sistema operativo che cerca nel disco l'informazione, che viene poi mandata al processore ed eseguita.

Possiamo immaginarci di avere processori che abbiano un certo numero di livelli di cache direttamente sul chip dove abbiamo implementato control unit e data path, li mettiamo lì in quanto finchè siamo sullo stesso pezzo di silicio la comunicazione è molto veloce, abbiamo una migliore sincronizzazione e lo stesso ciclo di clock.

Uscite dal chip ci sono le interfacce che collegano il processore alle memorie, nel processore ci sono due cache in un core, con 4 core abbiamo una ripetizione di questo hardware.



Se devo utilizzare architetture multi core c'è una variante dello schema in cui il primo livello è privato mentre gli altri sono condivisi, i core oltre ad avere un data path hanno una l1 dati e una l1 istruzioni private e una l2 condivisa a due a due (può anche non essere così), per finire con una l3 condivisa, l'unico modo per comunicare è utilizzare una memoria condivisa tra tutti, o utilizzare un bus condiviso.



Terminologia:

Ho una *hit* quando vado a cercare qualcosa in una memoria piccola, che è un sottoinsieme della memoria complessiva e lo trovo.

L'*Hit rate* è la percentuale di volte che troviamo quello che stiamo cercando.

L'hit si risolve in un tempo di accesso alla cache, molto spesso questo tempo è detto *hitTime* (Tc1)

Abbiamo un *miss* quando vado a cercare in una cache un qualcosa e quello che cerco non lo ha, il *miss rate* è una percentuale di volte che mi dice quante volte non troviamo quello che stiamo cercando. Se non troviamo quello che stiamo cercando immaginiamo di poter

chiedere la stessa cosa al livello superiore, mentre la richiesta tra processore e cache di primo livello è una richiesta di parola, tra livelli di cache superiore si trasferiscono blocchi di parole. Nel caso di miss il tempo è un hitTime + qualcos'altro.

Dobbiamo distinguere due concetti il *missTime* e la *missPenalty*, il misstime è il tempo che devo spendere per ottenere quell'elemento in caso di miss, la missPenalty è il tempo che ci metto a richiedere il blocco di dati e a salvarlo nella cache.

Il tempo che impiego per andare a fare l'amat, è l'hitTime del livello 1, sommato al missRate del livello 1, moltiplicato per l'hitTime del livello successivo se lo troviamo sommato al missRate moltiplicato per i livelli successivi ecct…

La moltiplicazione dei livelli successivi è una miss Penalty.

Tutto questo funziona perché vale il principio di località spaziale e temporale.

$$AMAT = t_{C_1} + MR_1 \cdot \left(t_{C_2} + MR_2 \cdot (\dots)\right)$$

Le cache sono implementate ad hardware come hash map che funzionano a ciclo di clock, ovvero in un ciclo di clock vogliamo sapere se il dato che cerchiamo c'è o non c'è.

Supponiamo di avere un vettore e una computazione che tocca posizioni consecutive e quando arriva in fondo torna in cima, posso leggere dei valori e incrementare la posizione indicizzata da modulo lunghezza del vettore.

Se so che un certo programma ha un certo numero di istruzioni, ha un cpi e un ciclo di clock sul processore, il cpi è una cosa del processore, e il tau è relativo all'hardware, se devo fare 5mld di istruzioni moltiplico per il tau e per il cpi e ottengo il tempo di completamento.

Tempo di completamento = *#I * Cpi * Tau*

In realtà questa cosa è una situazione ideale non stiamo infatti tenendo conto dei tempi di attesa che abbiamo quando non troviamo i dati nelle cache, questo è il conto che faremmo senza considerare i cache miss, nel caso dei cache miss ho da pagare la miss Penalty.

Potremmo dividere il cpi in un cpi perfect e un cpi stall che rappresenta quello che perdiamo quando stalliamo un processore, ovvero quando aspettiamo che arrivi qualcosa dalla cache. Il processore non sa che sotto c'è la cache, il meccanismo tra cache e processore ha dei sincronizzatori con un'interfaccia che restituisce l'esito dell'operazione che però devo attendere.

Il cpi stall si calcola considerando il numero di accessi effettuati all'interno del programma complessivamente, moltiplicato per la miss rate, e tutto questo verrà moltiplicato per la miss penalty.

$$Cpi_{Stall} = \#accessi * MissRate * MissPenality$$

$$Cpi = Cpi_{Perfect} + Cpi_{Stall}$$

Cache Performance Example

- Assume a miss rate of 2% for the instruction cache and of 4% for the data cache, a miss penalty of 100 cycles for all misses, and a frequency of 36% of loads and stores. If the CPI is 2 without memory stalls (i.e., $CPI_{Perfect}$), determine how much faster the processor runs with a perfect cache that never misses.

- CPI memory stalls for instructions and data accesses:
  $CPI_{Stall-Instr} = 1 * 0.02 * 100 = 2$ cycles
  $CPI_{Stall-Data} = 0.36 * 0.04 * 100 = 1.44$ cycles
  $CPI_{Stall} = 2 + 1.44 = 3.44$
  Therefore, the total CPI including memory stalls is: $CPI = 2 + 3.44 = 5.44$

$$\frac{CPU_{time\ with\ stalls}}{CPU_{time\ perfect}} = \frac{IC * (CPI_{Perfect} + CPI_{Stall}) * ClockCycleTime}{IC * CPI_{Perfect} * ClockCycleTime} = \frac{5.44}{2}$$

- The performance with the perfect cache is better by a factor of 2.72

Fare un sistema con le cache richiede di risolvere tutta una serie di problemi.

Quanti devono essere i blocchi?

Se io richiedo l'indirizzo, questo indirizzo lo considero con degli zeri in fondo, e vado a prendere gli indirizzi con suffisso simile.

Quante parole devo prendere nel blocco?

Non so quante parole effettivamente userò

Come facciamo a sapere se un dato è nella cache?

Serve un modo efficiente per creare la tabella cache

Nel caso di fault come trovo il blocco mancante?

Servono degli algoritmi per reperire l'informazione che non c'è

Cosa faccio se ho conflitti sui blocchi?

Es lo spazio non è sufficiente, o è già occupato

Che politiche uso per il rimpiazzamento?

Se sono arrivato alla fine della mia cache e ho bisogno di altro devo rimpiazzare qualcosa, quale scelgo?

Consistenza?

Ovvero se viene modificato un dato che è in una cache come faccio ad aggiornare lo stesso dato che si trova nelle altre cache nel caso di core multipli?

Snoopy bus

La consistenza delle cache consuma molte risorse, aumenta il tempo di accesso alla cache e serve però solo in alcuni casi.

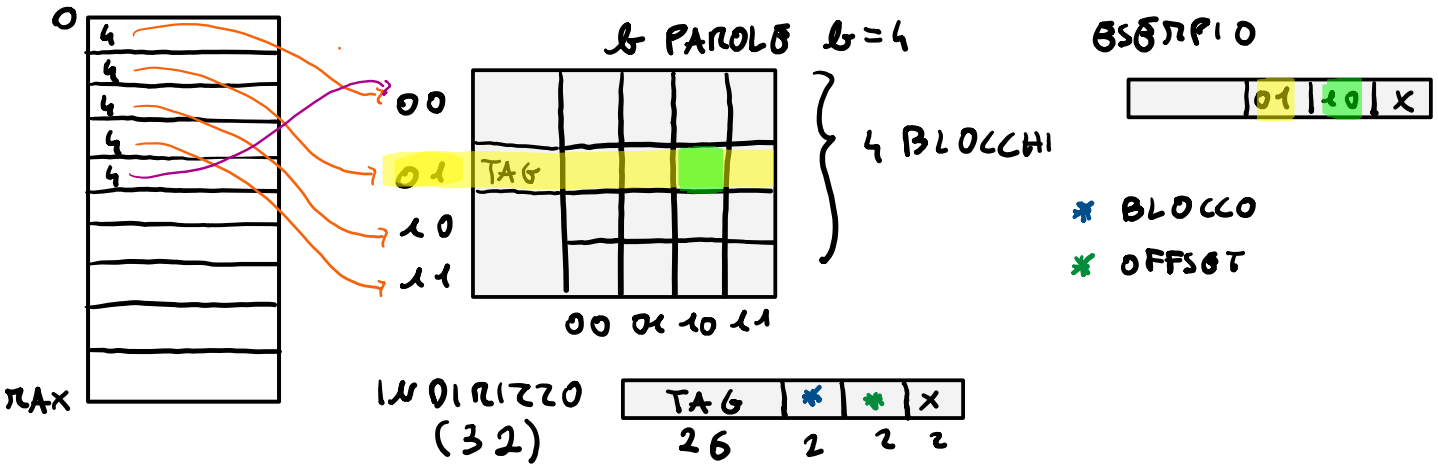
Indirizzamento diretto

Abbiamo una memoria che va da 0 a max e abbiamo un certo numero di posizioni, abbiamo poi una cache, all'inizio abbiamo un po' di informazioni di controllo e poi abbiamo b parole, in quanto lavoriamo su blocchi, supponiamo b = 4 e di avere 4 blocchi nella cache, stiamo parlando di parole, ci viene dato un indirizzo che ci serve per accedere alla cache, il processore vuole accedere alla parola a quell'indirizzo, come facciamo a fare questo con un indirizzamento diretto?

L'indirizzo è da 32 bit, l'ultima parte dell'indirizzo la useremo come offset, ovvero come qualcosa che comandi un multiplexer, che una volta individuato quale tra i gruppi di 4 parole è quello giusto, se c'è, ci individua la parola giusta, alla fine nell'offset abbiamo 2 bit che sono l'offset byte, e altri 2 che sono l'offset che effettivamente utilizzeremo (colonna).

Per sapere quale posizione prendere siccome sono ad accesso diretto utilizziamo ancora altri 2 bit che mi individuano il numero di blocco (riga).

Tutto quello che non viene utilizzato per l'indirizzamento, si chiama tag, questi bit vengono copiati nella colonna information control, e li utilizzeremo comparandoli con quello che troviamo all'indirizzo cercato per verificare se sono la stessa cosa.



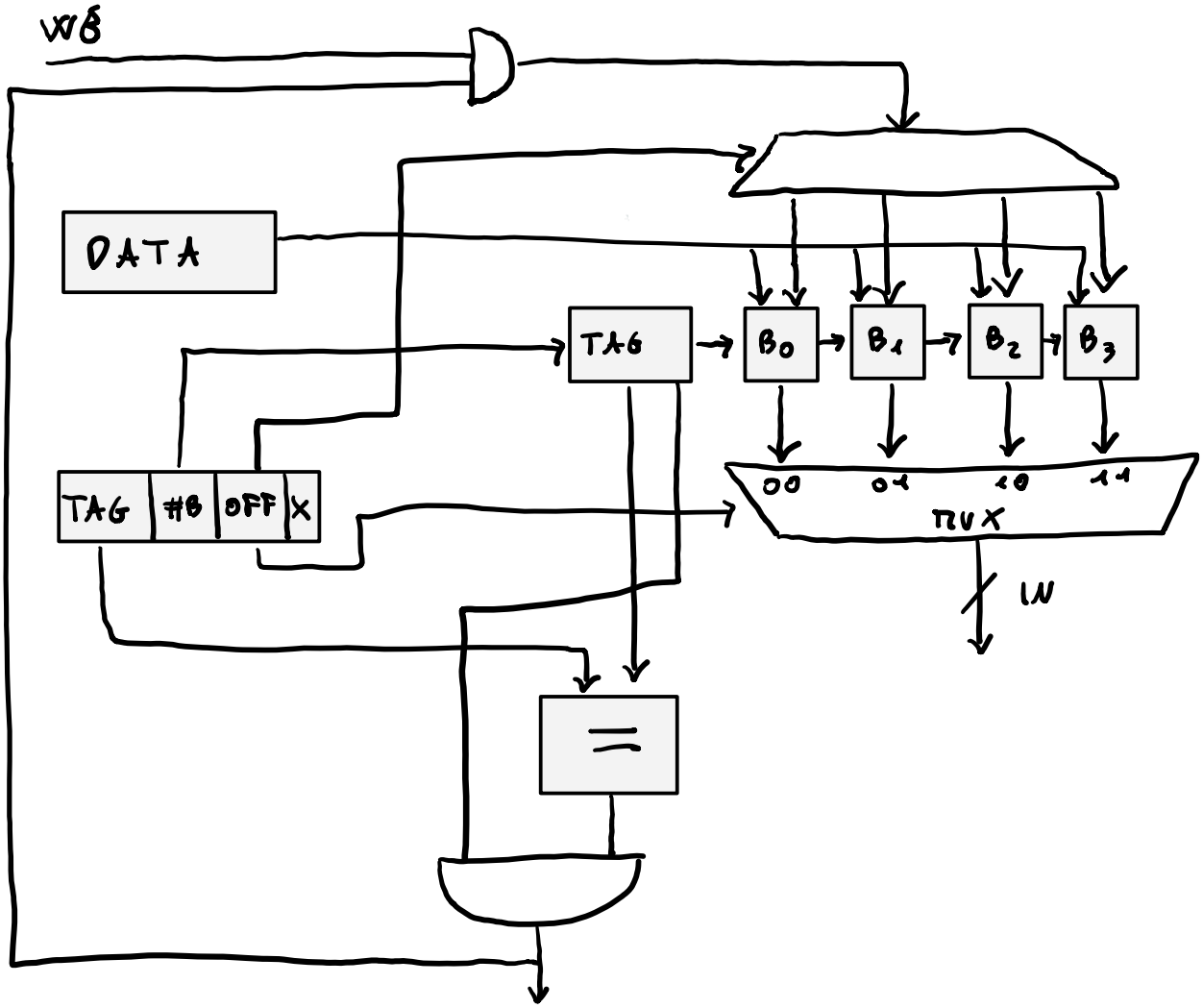
Come realizziamo la memoria?

Sarà una memoria fatta da un certo numero di moduli, mettiamo un modulo dove andiamo a mettere i tag, e 4 moduli dove inseriremo le parole, utilizziamo il numero di blocco dell'indirizzo come indirizzamento per tutti i moduli, troviamo quindi un tag e poi parola 0, 1, 2, 3, dobbiamo tirare fuori la parola giusta e poi verificare che sia quello che stiamo cercando. Per far ciò prendo il tag dell'indirizzo e il tag in uscita dal primo modulo, utilizzo quindi un confrontatore, se restituisce vero va tutto bene, se restituisce falso abbiamo un miss, in aggiunta al tempo di accesso alla memoria ci aggiungiamo il tempo di accesso al confrontatore che però è poco, contemporaneamente siccome è una memoria modulare posso scegliere una delle altre 4 parole usando i bit che prendo dal campo offset dell'indirizzo, utilizzo un multiplexer per recuperare la parola che stavo cercando se il comparatore è vero.

Questo vale sempre tranne quando siamo all'inizio e non abbiamo cose caricate sopra, ovvero quando la cache è ancora vuota.

Nel campo di controllo prima di cominciare a caricare le parole, un po' lo utilizziamo per il tag e un bit lo usiamo come bit di validità che vale 1 se e solo se quella riga contiene qualcosa di valido, 0 altrimenti.

Il bit di validità lo possiamo utilizzare in and con il segnale che ci genera il confrontatore.



Quanto impiega questo sistema per restituire qualcosa o per dirci che c'è un fault?

$$\text{HitTime} = T_a + max\{T = T_{and}\ ,\ T_{mux}\}$$

Qual è il problema di queste cache?

Il fenomeno del trashing, immaginiamo di avere una cache di 32kb organizzate con b = 16, ci sono 1k blocchi (righe) ognuno da 16 parole (colonne)

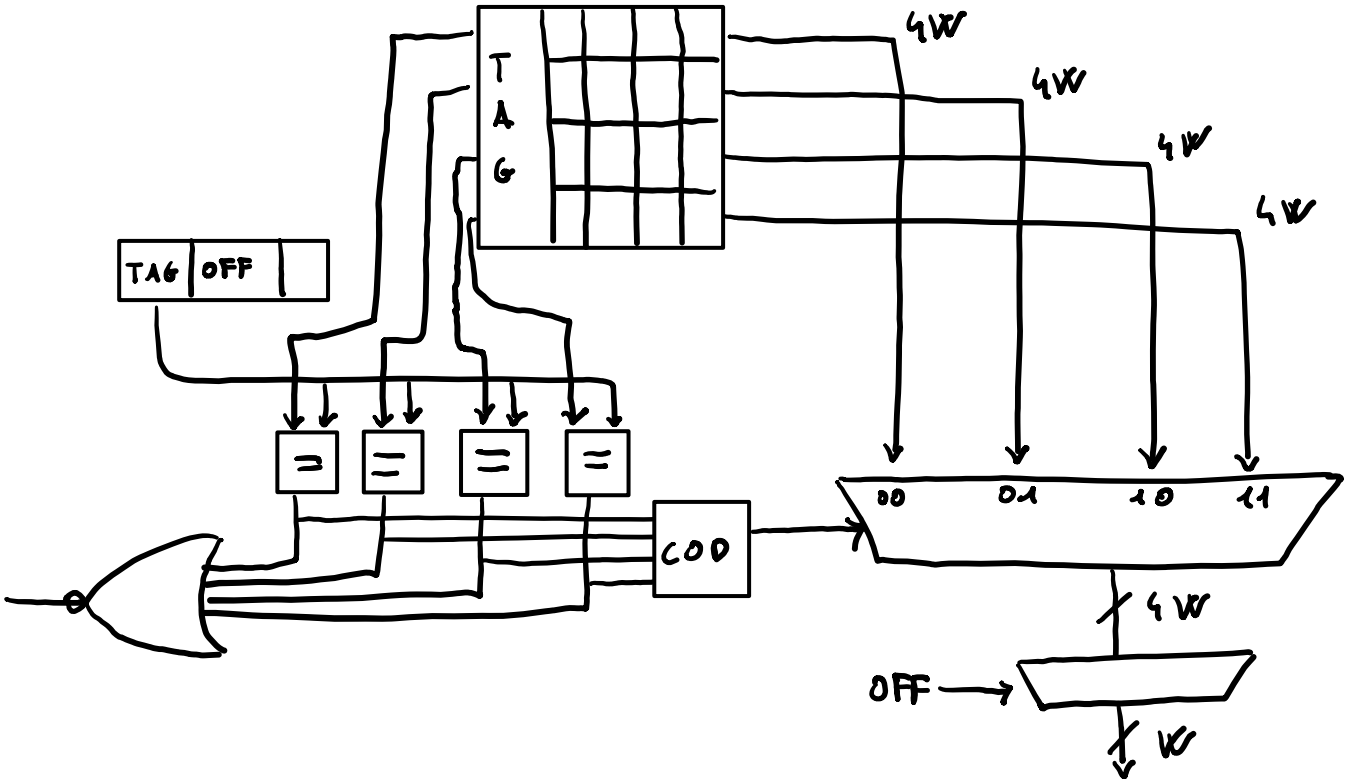
Ad ogni ciclo del for nella cache ho qualcosa che potrei riutilizzare che però non riutilizzo perché ad ogni ciclo la cache deve essere sovrascritta, avendo però tutto il resto vuoto. Questo accade in quanto i dati vengono mappati sempre nei soliti indirizzi.

Esempio:

Immaginando un for da i a n che esegue il seguente calcolo A[i] = B[i] + C[i] ad ogni ciclo dovrò recuperare i valori di A, B e C

Indirizzamento Associativo

I dati possono essere inseriti in una qualunque riga della cache, per cercare un blocco dobbiamo però confrontare contemporaneamente tutte le righe a differenza di come facevamo nell'indirizzamento diretto dove avevamo i due bit che ci indicavano il blocco.



Abbiamo 4 parole, e 4 righe, nella parte di controllo abbiamo un tag e un bit di qualità, data una chiave (il tag) diciamo se l'elemento c'è o non c'è, l'indirizzo che presento è un tag e un offset.

Avendo 4 linee mi servono 4 comparatori che prendono ognuno il tag corrispondente e il tag dell'indirizzo come secondo ingresso, restituendo poi un segnale che mi dice se è uguale o no.

Metto tutti i valori dei confrontatori in or, lo nego e ottengo il fault, tra i 4 gruppi devo scegliere il gruppo di parole che mi interessa e poi su quelle con l'offset scelgo la parola che stavo cercando.

Come comando il primo multiplexer?

Prendiamo il risultato dei confrontatori che vengono mandati ad un codificatore che restituisce il segnale di controllo del primo multiplexer.

I tag non possono essere memorizzati in una memoria normale, perché ho bisogno di leggerli contemporaneamente, devo usare quindi dei registri.

Questa cosa è una cosa molto costosa che però risolve completamente il problema del trashing.

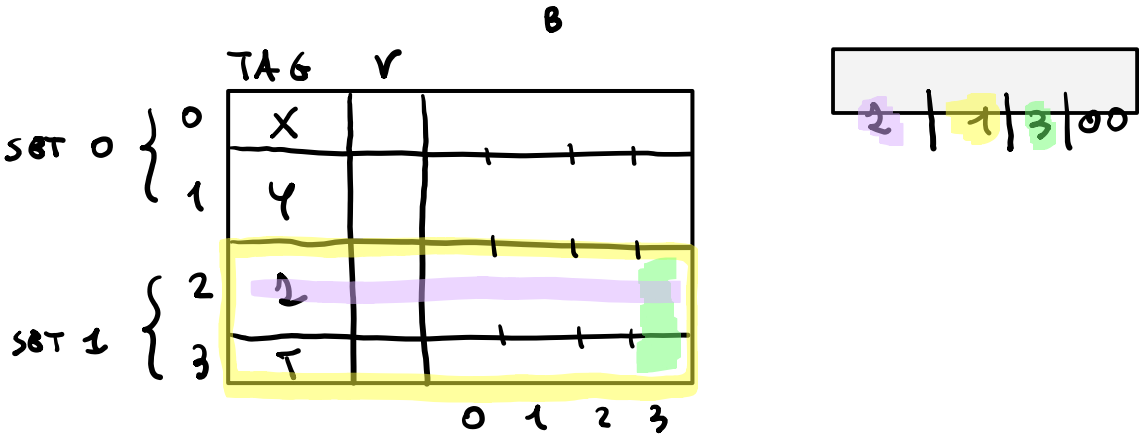
Indirizzamento associativo su insiemi

Usa un indirizzamento diretto per accedere ad un piccolo insieme, e al suo interno si comporta in modo associativo.

Ci sono quindi due fasi:

1) Usiamo un indirizzamento diretto e con questo individuiamo un insieme di linee di cache.

2) Usiamo un indirizzamento completamente associativo e da questo insieme di linee troviamo una linea che è quella dove presumibilmente troviamo quello che stavamo cercando oppure un fault.



Come funziona questo tipo di indirizzamento?

Immaginiamo di avere una cache così fatta:

4 linee di cui ognuna ha una parte tag, un bit di validità e b parole. In realtà le prime due righe le chiameremo il set_0 e le altre 2 il set_1, si parla di indirizzamento associativo a k vie, con k che è il numero di linee per insieme.

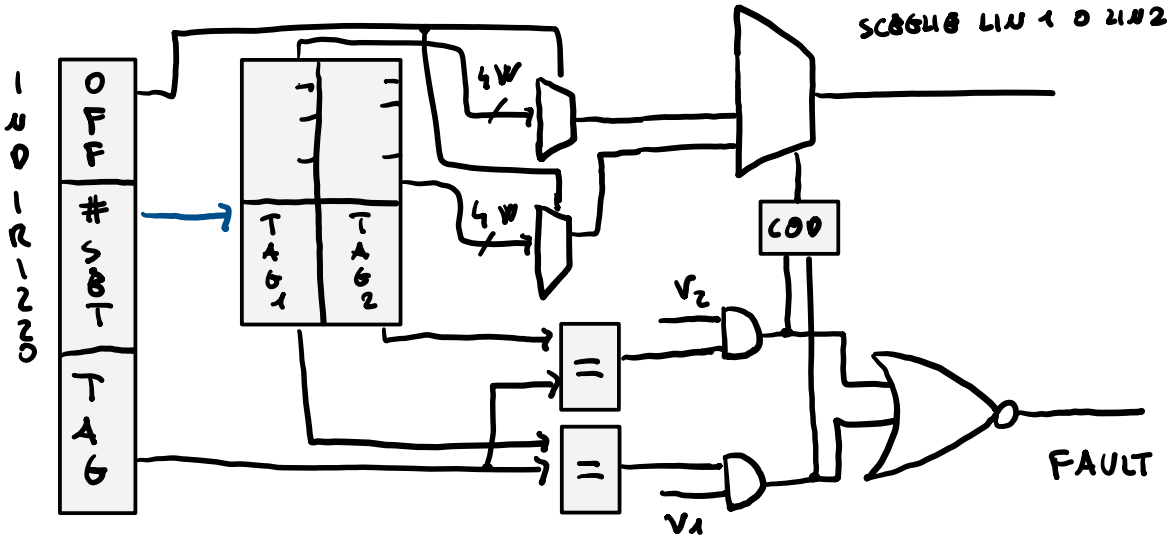
Come funziona l'indirizzamento?

L'indirizzo ha la solita struttura che noi abbiamo descritto come divisa in 3 parti, una parte inizializza l'offset, una il numero di insieme, e poi il tag.

Come si compone questo oggetto?

Prendiamo il numero di insieme e individuiamo la coppia di righe, abbiamo due righe con due tag diversi, a questo punto entriamo nella seconda fase, prendo i due tag e li mando a due comparatori, i quali prendono anche il tag dell'indirizzo e questo oggetto insieme ai bit di validità genera due segnali in and, messi poi in or per dirmi se l'ho trovato, aggiungendo un not dopo l'or ottengo il segnale di fault.

I due segnali prima dell'or possono essere usati in un codificatore che comanda un multiplexer per scegliere tra la linea uno e la linea due, le entrate di questo multiplexer saranno le uscite di altri due multiplexer sopra, che prendono le 4 parole della prima e della seconda riga e con l'offset che prendiamo dall'indirizzo scegliamo o una parola della prima riga o una parola della seconda riga, l'offset sceglie di prendere la parola o dal primo set o dal secondo set.



Se nella fase di indirizzamento diretto avessi il fenomeno del trashing, andrei comunque a cercare in quel set, dove però potrei usare l'altra riga e mantenere quindi le informazioni precedenti, sostituendo solo le informazioni meno recenti, senza eliminare le ultime eseguite.

Esempio: Se z e w fossero due strutture dati, e quindi per il principio di località vengono lette contemporaneamente, sebbene mappino sullo stesso indirizzo, in modo diretto, ci sono due spazi disponibili e quindi possiamo memorizzare sia A che B senza stare tutte le volte a sovrascrivere.

Con questo metodo di indirizzamento si hanno i vantaggi del metodo diretto ovvero semplicità, e del metodo associativo ovvero versatilità.

Con i costi che sono piccoli come nell'indirizzamento diretto perché la parte associativa vale poco, il numero di comparatori varia solo a seconda di quante linee (tag) abbiamo nell'insieme.

Abbiamo un costo che è quello di due comparatori, riesco a fare un accesso associativo dopo aver individuato l'insieme in cui devo andare ad operare.

Adesso si usano cache ad 8-vie ovvero 8 linee per insieme.

Posso scegliere il numero di vie e le parole da associare in un blocco.

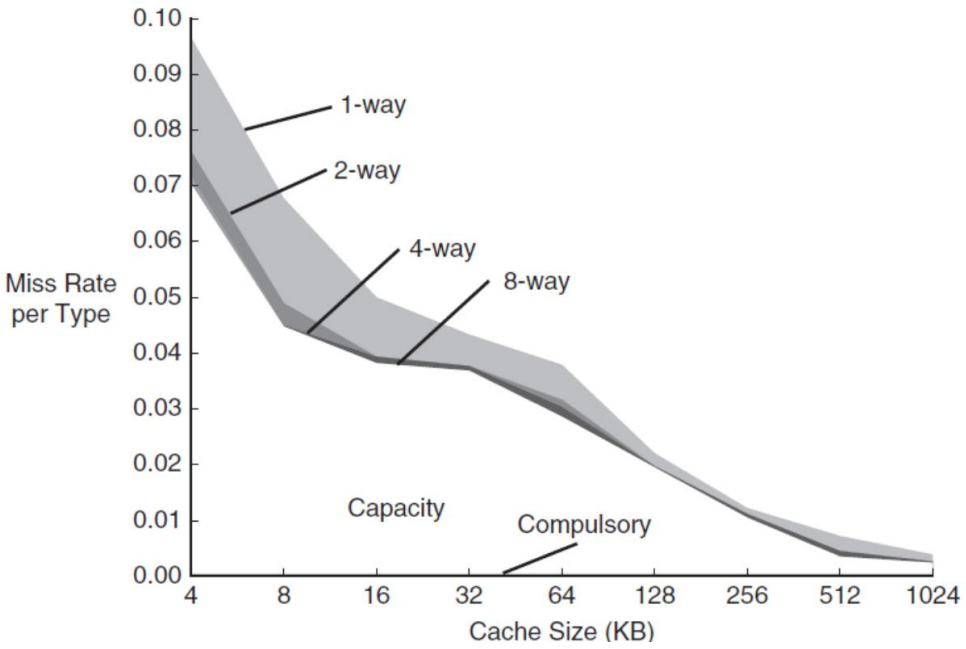
Cache Miss

Tipi di cache miss:

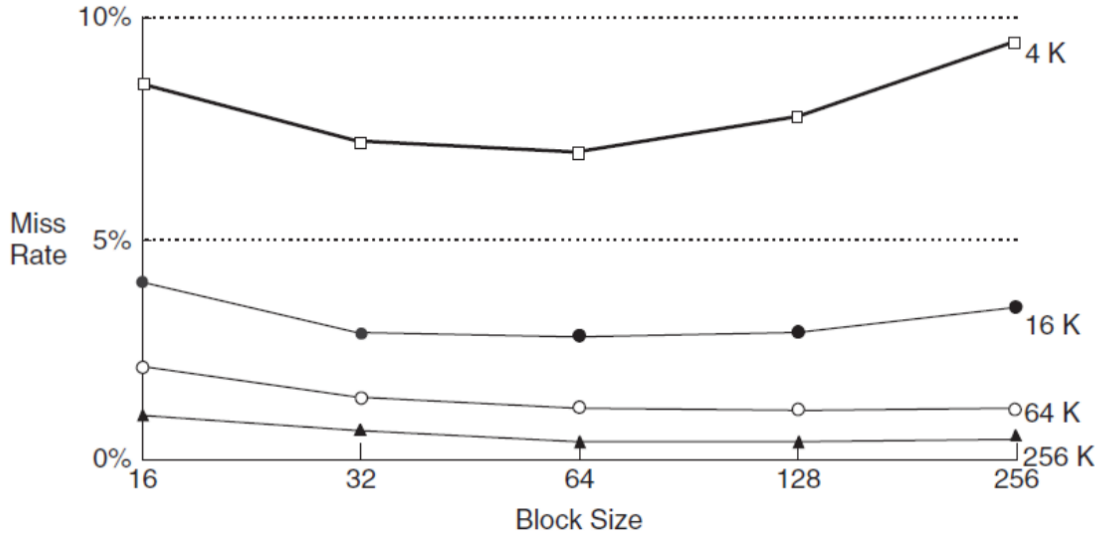
- Miss inevitabili, sono quei miss che abbiamo quando accediamo per la prima volta ad un certo indirizzo, che dunque sarà vuoto.

- Miss legati alla capacità della cache, sono quei miss che non avrei se avessi una cache più larga. Si ha infatti un capacity_miss quando l'insieme su cui lavoro, il working set, non entra nella cache, per cui ho un insieme di indirizzi di dati che riferisco in modo ripetitivo, non avendo abbastanza spazio in cache, temporaneamente dovrò togliere e riaggiungere istruzioni.

- Miss di conflitto, si hanno solo con le cache di tipo diretto e associativo su insiemi. per via delle operazioni di mapping posso avere dei conflitti perché più indirizzi potrebbero convergere sulla stessa linea di cache dello stesso insieme.



All'aumentare del numero di vie (dimensione) non ho un incremento significativo, ovvero un decremento significativo del miss rate. Questo ci dice che difficilmente troveremo cache con 16/32 vie (righe).



Il numero tipico di parole in un blocco sono 8-16 parole, nel grafico possiamo vedere che fissata una certa capacità variando da 16 a 256 il numero di byte possiamo variare il numero di parole che posso memorizzare in una linea, notiamo però come dopo un po' non sia più conveniente aumentarlo, più ne metto e più mi costa caricarne.

Come possiamo diminuire l'amat?

Per farlo avevamo pensato di ridurre il miss rate, un modo è quello di incrementare la dimensione del blocco, ma non più di tanto perché sennò aumenta la miss penalty, potrei incrementare la dimensione della cache, che però ha un costo economico maggiore, potrei aumentare l'associatività che diminuisce i conflitti ma genera un hit time maggiore.

Il miss rate si può minimizzare cercando di scrivere gli algoritmi nel modo migliore possibile, ovvero nel modo che mi permette di sfruttare al meglio la località spaziale e temporale.

Gestire i cache miss

Se ho un cache miss il processore si ferma perché si aspetta che la load venga completata, anche se non ha trovato il dato nella cache.

Esistono organizzazioni dell'architettura che permettono di sovrapporre istruzioni (out of order architecture).

Cosa avviene in caso di fault?

Se ho un fault al livello *i* devo chiedere al livello *i+1*[5] il dato che mi serve, quando arriva faccio ripartire l'operazione e genero un cache hit.

L'architettura (hardware) sposta i dati a blocchi tra i livelli di cache e la ram, e singole parole tra il chip e la cache.

Tra disco e memoria invece il trasferimento è a carico del sistema operativo.

[5] *i+1* è il livello successivo della cache, quello più costoso in termini di accesso.

Gestire le store

Anche nel caso delle store posso avere una hit o un miss, dove scrivo il dato?

Solo nel primo livello o in tutti?

Ovviamente va scritto in tutti, infatti se non prendiamo precauzioni la cache e la memoria sarebbero inconsistenti, ovvero potrei avrei due versioni del dato diverse.

Ci sono due tecniche per il caso di hit, la *write trough* e la *write back*.

Nel caso di miss invece? (Abbiamo un miss quando non c'è spazio disponibile)

Due diverse tecniche *write allocate* e *write non allocate*, nella write allocate il blocco è scritto prima nella cache e poi la memoria viene aggiornata, questa cosa ha più senso quando abbiamo il write back, l'altra politica se ho un write miss semplicemente non carica il blocco in cache ma lo scrive nel livello superiore, e questa tecnica si usa principalmente con il write trough.

Tecniche di gestione dei write hit

La politica *write trough* ogni volta che scriviamo qualcosa in cache non solo la scrive nella cache ma aggiorna anche i livelli superiori.

Questo sistema ha un grosso svantaggio, ogni volta che scrivo un dato in cache siccome lo scrivo anche nei livelli superiori devo aspettare che il caricamento sia completato, la scrittura rischia quindi di costarmi non quanto l'accesso in cache bensì quanto l'accesso alla memoria più lenta.

La tecnica del *write back* quando fa una scrittura aggiorna solo il livello che stiamo scrivendo, i livelli superiori e più lenti verranno aggiornati solo quando rimuoverò il dato dalla cache.

Non ho nessuno degli svantaggi che avevo nella politica write trough, ma ne ho altri, i contro sono che se anche scrivo una sola parola mi devo ricordare che quella linea di cache è sporca, se la devo rimpiazzare devo poi copiare il valore nei livelli superiori, in generale l'implementazione è più complicata e il costo di rimpiazzamento di una linea di cache è più alto.

Suppose a cache has a block size of four words. How many main memory accesses are required by the following code when using each write policy: write-through or write-back?

```
MOV R5, #0
STR R1, [R5]
STR R2, [R5, #12]
STR R3, [R5, #8]
STR R4, [R5, #4]
```

Solution: All four store instructions write to the same cache block. With a write-through cache, each store instruction writes a word to main memory, requiring four main memory writes. A write-back policy requires only one main memory access, when the dirty cache block is evicted.

Per la politica write trough si utilizza un write buffer, ovvero una memoria aggiuntiva, che si trova nel MMU (memory management unit) e permette di ricordarsi di bufferizzare le memorie verso i livelli superiori, anziché aggiornare subito questi livelli, scriviamo il dato nel buffer.

Abbiamo posticipato il tempo di stallo, mediamente il costo di scrittura è quello della cache anche per la politica write trough proprio grazie a questo meccanismo.

Ho dei problemi quando il write buffer è pieno, la dimensione del buffer è infatti limitata.

Il write buffer permette anche di avere ottimizzazioni nella politica write back,

Il costo di un write è ovviamente più alto se si ha un cache miss, dobbiamo infatti prima scrivere il blocco che viene dalla memoria (se il bit è settato a 1, ovvero se la memoria era già occupata). Questo richiede almeno due cicli di clock anche per un write hit, un ciclo per controllare se abbiamo avuto un hit o meno e un altro ciclo per fare effettivamente la write.

Alternativamente possiamo usare un write buffer, che ci permette di mantenere temporaneamente il dato da scrivere mentre vediamo se nel blocco di cache abbiamo una hit. Il processore controlla la cache e mette il dato da scrivere nel write buffer durate il ciclo di accesso alla cache. Assumendo di aver avuto un cache hit, il nuovo dato è scritto dal buffer dentro la cache nel prossimo ciclo di accesso (pipelining of access).

Mentre per la politica write trough il write buffer è obbligatorio, nella politica write back il write buffer è davvero un'ottimizzazione, il write buffer crea però molti problemi a livello software, le architetture sono sempre più spinte, per cercare di minimizzare la miss penalty e rendere gli accessi più veloci.

Rimpiazzamento di cache

Con una cache di una certa di dimensione il working set potrebbe non essere contenuto totalmente nella cache, immaginando un loop con un insieme di istruzioni dati, l'ottimo sarebbe che nella cache entrasse tutto il working set così da avere per un lasso di tempo tutti hit. Questo può non accadere se il working set è più grande della dimensione della cache, per questo capiterà che accederò a dei dati che non sono in cache ma la cache è piena, dovrò quindi rimuovere una linea di cache.

Nelle politiche associative posso scegliere cosa rimpiazzare, in quella totalmente associativa posso infatti scegliere quale linea di cache buttare fuori, e pure in quella associativa su insiemi ho il range di blocchi per un dato insieme e devo capire quale selezionare per rimpiazzarla.

Come vengono fatte queste scelte?

La scelta migliore sarebbe rimpiazzare la linea di cache a cui accederò più lontanamente nel tempo, questo però non sappiamo farlo, dunque rimuoviamo la linea di cache che si è usata meno recentemente nel passato (Last Recently Used). Questa politica gioca sul fatto che se il programma ha località temporale è più probabile che io acceda a linee di cache a cui ho acceduto recentemente.

Per cache totalmente associative useremo una politica random, che ci fornisce circa le stesse performance della LRU.

Consider a small cache with 4 blocks, b=1. Find the number of misses for a **direct-mapped** cache for the following ordered sequence of block addresses 0, 8, 0, 6, 8.

Block address	Cache block
0	(0 modulo 4) = 0
6	(6 modulo 4) = 2
8	(8 modulo 4) = 0

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		0	1	2	3
0	miss	Memory[0]			
8	miss	Memory[8]			
0	miss	Memory[0]			
6	miss	Memory[0]		Memory[6]	
8	miss	Memory[8]		Memory[6]	

Consider a small cache with 4 blocks, b=1. Find the number of misses for a **2-way set-associative** cache for the following ordered sequence of block addresses 0, 8, 0, 6, 8 (**LRU replacement policy**).

Block address	Cache set
0	(0 modulo 2) = 0
6	(6 modulo 2) = 0
8	(8 modulo 2) = 0

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Set 0	Set 0	Set 1	Set 1
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[6]		
8	miss	Memory[8]	Memory[6]		

Consider a small cache with 4 blocks, b=1. Find the number of misses for a **fully associative** cache for the following ordered sequence of block addresses 0, 8, 0, 6, 8.

Address of memory block accessed	Hit or miss	Contents of cache blocks after reference			
		Block 0	Block 1	Block 2	Block 3
0	miss	Memory[0]			
8	miss	Memory[0]	Memory[8]		
0	hit	Memory[0]	Memory[8]		
6	miss	Memory[0]	Memory[8]	Memory[6]	
8	hit	Memory[0]	Memory[8]	Memory[6]	

Riprendiamo adesso l'esempio a pag 46.

Suppose we speed up the clock rate by a factor of 2 (e.g., 2 -> 4 GHz

$CPI_{Stall-Instr} = 1 * 0.02 * 200 = 4$

$CPI_{Stall-Data} = 0.36 * 0.04 * 200 = 2.88$

$CPI_{Stall.} = 4 + 2.88 = 6.88$

$CPI = 2 + 6.88 = 8.88$

The fraction of time spent on memory stalls rises from 63% to 77%

- 3.44/5.44 = 0.63 → 6.88/8.88 = 0.77

The execution time improves of a factor 1.23:

$$\frac{CPU_{time1}}{CPU_{time2}} = \frac{IC * CPI_{Stall1} * ClockCycleTime}{IC * CPI_{Stall2} * \frac{ClockCycleTime}{2}} = \frac{5.44}{\frac{8.88}{2}} = 1.23$$

Ho raddoppiato la frequenza ma ho migliorato solo di un 1.23 su un valore di 2, si ha quindi un'efficienza del 60% è dunque inutile fare il processore con un tau più basso se lasciamo il sistema di memorie invariato, rischiamo di guadagnarci poco.

Come si può migliorare l'efficienza?

Bisogna per forza migliorare la miss penalty, se ho un miss per caricare i dati in cache devo cercare di ridurre il tempo, un modo per farlo è quello di avere più livelli di cache. L'obbiettivo è cercare di diminuire quanto mi costa la miss penalty.

Cache multilivello

Se il livello superiore dopo un miss è gestito da una cache mi può essere utile così da avere un cache hit in pochi cicli di clock, accedere alla memoria esterna costerebbe molto di più, i livelli di cache da poter mettere sono però limitati, infatti dopo un po' il guadagno non riesce a giustificare il costo.

- $t_M = 50ns$ (main memory service time), $t_{L1} = 1ns\ t_{L2} = 6ns\ t_{L3} = 10ns$
- Miss rates: 10%, 1.5% and 0.4% for L1,L2, and L3, respectively

1. No cache: *AMAT*= 50ns
2. Only L1 cache: *AMAT*= 1+0.1*50= 6ns
3. L1 and L2 caches: *AMAT*= 1+0.1*(6+0.015*50)= 1.675ns
4. L1, L2 and L3 caches: *AMAT*= 1+0.1*(6+0.015*(10+0.004*50))= 1.6153ns

In generale se consideriamo il CPI, per calcolare il CPI stall possiamo usare la seguente formula.

$$CPI_{Stall} = Miss\ rate\ memory\ instr. * Miss\ penalty$$

$$CPI_{Stall} = MissRate_{L1} * MissPenalty_{L1} + GlobalMissRate_{L2} * MissPenalty_{L2} + GlobalMissRate_{L3} * MissPenalty_{L3} + \ldots$$

$$GlobalMissRate_{LN} = MissRate_{L1} * MissRate_{L2} * \ldots * MissRate_{LN}$$

La miss penalty del livello 1 può essere considerata come l'access time del livello 2

- Suppose a 2-level cache system with:
  - $MissRate_{L1} = 2\%, \quad MissRate_{L2} = 20\%$
  - L2 cache access time 20 cycles, main memory access time 200 cycles

  compute a) the $GlobalMissRate_{L2}$ and b) the CPI_{Stall}.

a) $\text{GlobalMissRate}_{\text{L2}} = \text{MissRate}_{\text{L1}} * \text{MissRate}_{\text{L2}} = 0.02*0.2=0.04 \ (4\%)$

b) A miss in the L1 cache can be satisfied either by L2 cache or by the main memory. The miss penalty is 20 cycles if we have an hit in L2, and 200 cycles if we have a miss in L2, thus:

$$MissPenalty = 20 + 0.2 * 200 = 60 \ cycles$$

Therefore, $CPI_{Stall} = MissRate_{L1} * MissPenalty = 0.02 * 60 = 1.2 \ cycles$

(in a different way)

b) By considering $MissPenalty_{L1} = 20$ and $MissPenalty_{L2} = 200$ and using the previous formula (previous slide), we have:

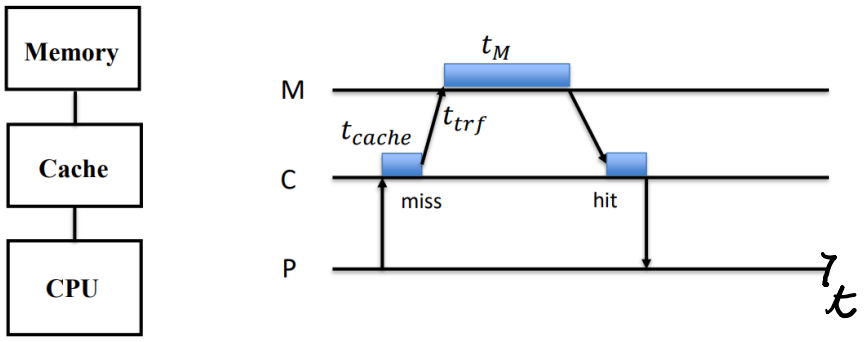
$$CPI_{Stall} = 0.02 * 20 + (0.02 * 0.2) * 200 = 1.2 \ cycles$$

Disegnare i sistemi di memoria

Il miss penalty può essere ridotto incrementando il bus tra dram e cache, ci sono tre possibili organizzazioni:

- L'organizzazione semplice in cui si legge dalla memoria una parola alla volta
- L'organizzazione wide-memory in cui vengono lette dalla memoria N parole
- L'organizzazione interleaved (interallacciata) in cui ci sono K memorie indipendenti che possono gestire varie richieste contemporaneamente

Quando ho un miss devo recuperare la parola che mi serve e chiedere alla memoria di inviarmela.



Spendiamo un tempo per accedere alla cache, un certo tempo per fare il trasferimento della richiesta, un tempo per leggere la parola e un tempo per portarla giù.

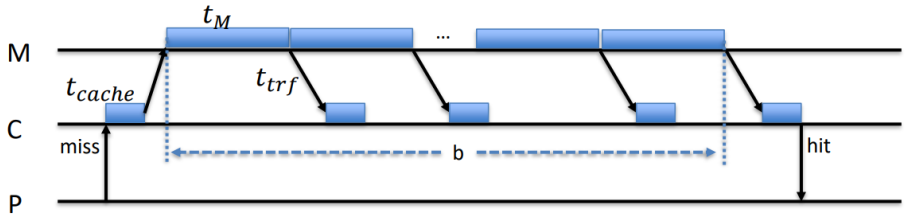
Se uno somma tutti i tempi, il tempo di accesso è circa:

$$t_a = 2\,t_{trf} + 2\,t_{cache} + t_M$$

In presenza di cache, in generale, abbiamo un numero di parole >1 contemporaneamente.

Se seguissimo lo stesso schema, possiamo fare un po' di pipelining tra il tempo di lettura della memoria e quello di trasferimento/cache.

$$t_{miss} = 2t_{trf} + 2t_{cache} + b\,t_M \approx b\,t_M$$



Considering $b = 8$ and $t_M = 80$, t_{cache}= 1 and $t_{trf} = 6$ clock cycles, the cost of the cache miss due to the transferring of all data in the block is very high (>650 clock cycles)

- The CPU stalls for several hundred cycles!

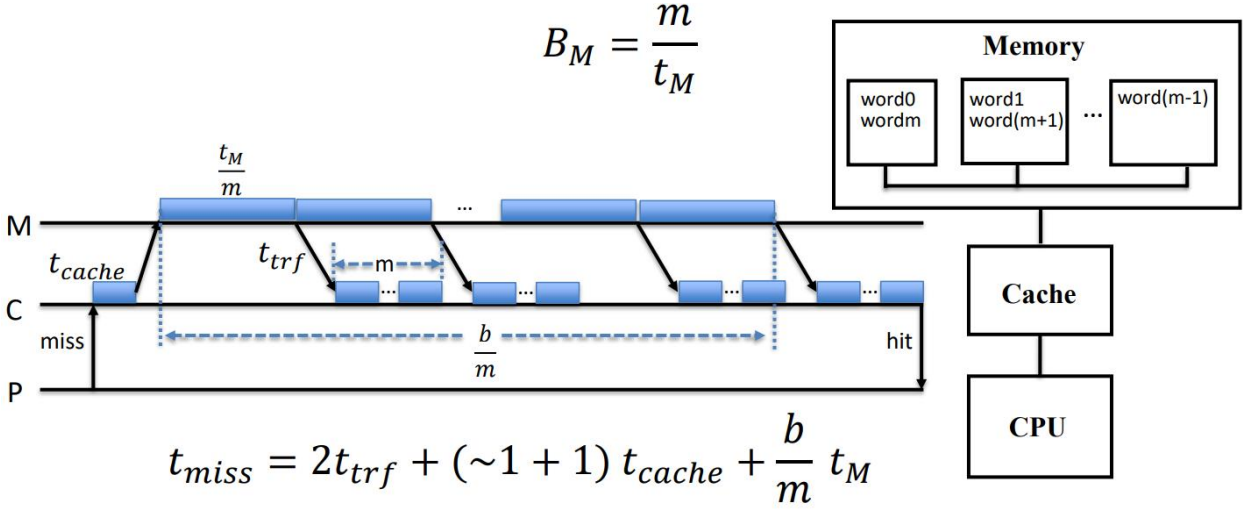
Un'organizzazione semplice non è accettabile, con l'organizzazione interallacciata, in cui si suppone che ci siano blocchi di memoria indipendenti, è come se il sottosistema di memoria avesse aumentato la banda passando da $1/t_m$ a m/t_m con m che è il numero di moduli di memoria con cui l'abbiamo costruita.

In un tempo semplice riusciamo a servire m parole quindi il tempo di servizio diventa m/t_m

Il tempo finale che dovrebbe essere m per t $_{cache}$, in realtà si suppone che siamo in grado di scrivere m parole in un ciclo.

Se mettiamo che b = m il tempo di un miss è circa il tempo di un accesso in memoria.

$$B_M = \frac{m}{t_M}$$



$$t_{miss} = 2t_{trf} + (\sim 1 + 1)\, t_{cache} + \frac{b}{m}\, t_M$$

- If we set $m = b$ we have again $t_{miss} \approx t_M$

Problemi delle cache

Ci sono aspetti negativi legati alla coerenza, nei sistemi multi-core il fatto di avere cache e core, cache condivisa o comunque di avere cache separate per ogni core, dove sui core ci girano programmi che condividono dati, crea dei problemi. C'è infatti il problema che su una cache potrei avere un dato aggiornato e su un'altra cache lo stesso dato non ancora aggiornato.

Con cache coherence si pone il problema di mantenere e tenere le cache sempre aggiornate.

Un altro problema è il false sharing che pero non approfondiamo.

Cache Coherence Problem

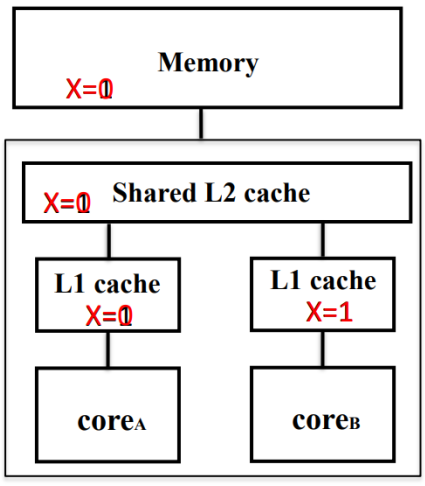
Problema molto complesso, supponiamo di avere un sistema multi core, (SMP symmetric multiprocessor) con due core e due processori che hanno una cache privata L1, una cache condivisa L2 più una memoria, l'avere una cache condivisa dà dei vantaggi.

Il vantaggio di una memoria condivisa è che programmi diversi che sfruttano gli stessi dati possono trovare i dati in un livello condiviso, aumentiamo dunque la possibilità di riuso dei dati (località spaziale e temporale) il problema è: che succede se un programma che gira sul core_A, scrive un dato e questo dato è condiviso dal programma che gira sul core_B?

Ad esempio lui scrive e l'altro lo vuole leggere.

Write-through caches

Time	Event	L1-core1	L1-core2	M
0				X=1
1	A reads X	X=1		X=1
2	B reads X	X=1	X=1	X=1
3	A writes X	X=0	X=1	X=0



Supponiamo di avere una variabile x in memoria principale, una volta che il core A vi accede, questa variabile viene trasferita dalla memoria in L2 e poi in L1, la stessa cosa la fa dopo un po' il coreB, che però non dovrà prenderla dalla memoria perché la troverà già nella L2, a questo punto però il processo sul coreA scrive x, e la mette a 0, supponiamo adesso che tutta la gerarchia sia aggiornata, tranne la parte del core B (L1) che ha x ancora ad 1, in questo caso le due cache si dicono non coerenti, ci serve un protocollo per dire all'altra cache di buttare via quello che ha, oppure di sovrascriverlo.

Due opzioni: la cache L1 in qualche modo quando scrive il suo valore lo trasferisce a tutte le altre, un'altra opzione potrebbe essere invece di dire a tutte le altre cache che quel valore non è più valido.

Questi sono rispettivamente i protocolli di aggiornamento e di invalidazione, noi vedremo quello più semplice, il secondo.

Quando scriviamo un valore viene detto a tutte le cache che quella cella non è più valida, poniamo il bit di validità a 0.

Se ci sono tante invalidazioni si ha un tempo in cui il processore sta fermo, stalla.

Write-back caches

Event	Bus activity	L1-core1	L1-core2	M
				X=1
A reads X	Miss for X	X=1		X=1
B reads X	Miss for X	X=1	X=1	X=1
A writes X	Invalidation for X	X=0	not valid	X=1
B reads X	Miss for X	X=0	X=0	X=0

False sharing

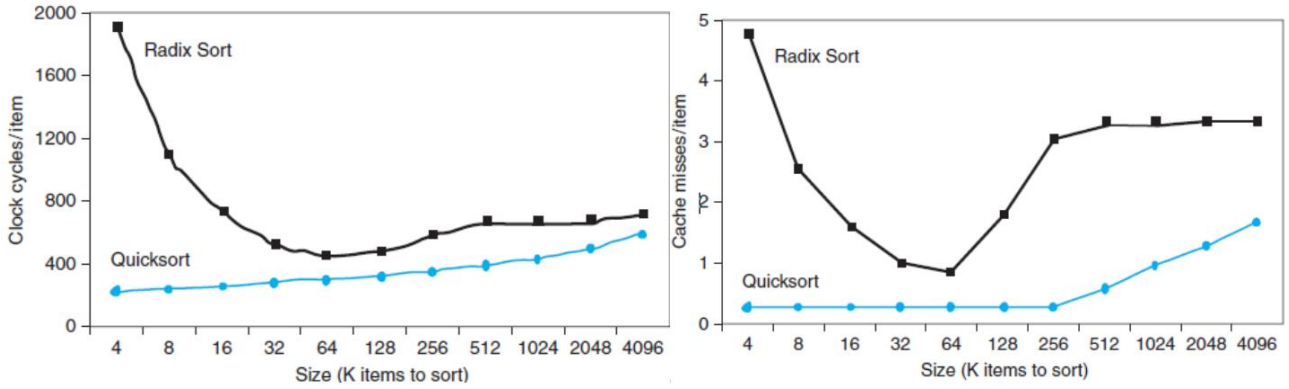
Una linea di cache ha più parole (per sfruttare la località spaziale). Supponiamo di avere due processi di uno stesso programma che però girano su due core diversi, il processo A e il processo B, uno accede alla variabile A e uno accede alla variabile B, non c'è nessuno sharing, però se le due variabili per qualche motivo dovessero capitare sulla stessa linea di cache di fatto il problema del false sharing si verrebbe a creare, non sono condivise però quando il processo A scrive A, la linea di cache viene invalidata per il processoB. Si crea questo meccanismo di mutua invalidazione e quindi il trashing, perché due cose che erano scollegate sono finite sulla stessa linea di cache.

Come si risolve di solito?

Di solito forziamo il fatto che le variabili siano distanziate in memoria di almeno una linea di cache. I compilatori cercano di risolvere automaticamente questo problema ma non sempre ci riescono.

Interazioni cache con il software

Un esempio per provare a capire il problema.



L'algoritmo radix sort così com'era descritto non riusciva a sfruttare bene le cache, ed è stato dovuto riscrivere, le cache se usate male possono innescare il fenomeno del trashing, il costo di gestire i miss non è basso quindi perdiamo molti cicli del processore.

Anche questo ha un impatto serio sui programmi software.

I compilatori devono sapere com'è fatta l'architettura e devono compilare il codice in modo da minimizzare i fault di cache.

Ottimizzazioni Software

Quelo che vogliamo fare è migliorare il codice il più possibile in modo da poter sfruttare al massimo la località spaziale e temporale.

Le due tecniche che vedremo sono il loop interchange e il data blocking.

Loop Interchange

Before

```
for (j=0;j<100;++j)
  for(i=0;i<1000;++i)
     A[i][j] = 2*A[i][j];
```

After

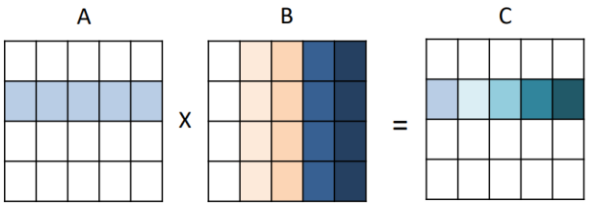
```
for (i=0; i<1000; ++i)
  for(j=0; j<100; ++j)
     A[i][j] = 2*A[i][j];
```

Il loop interchange come possiamo vedere dalla figura ci permette di implementare la località spaziale

Data blocking

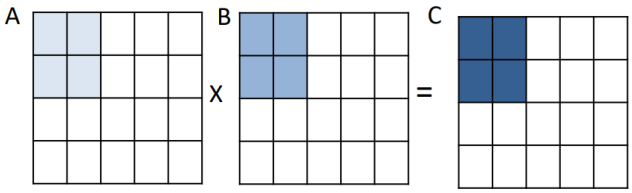
Before (textbook algorithm)

```
for (i=0;j<N;++i)
  for(j=0;j<N;++j) {
     c = 0;
     for(k=0;k<N;++k)
       c+= A[i][k] * B[k][j];
     C[i][j] = c;
  }
```

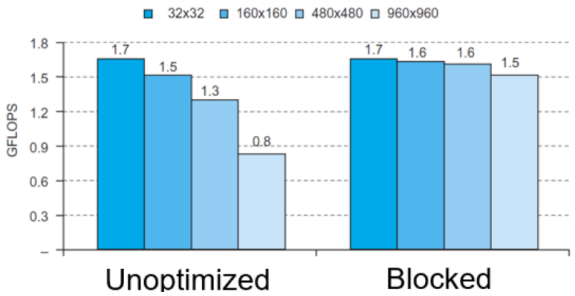


After

```
for (jj=0; jj<N; jj+=B)
 for(kk=0; kk<N; kk+=B)
  for(i=0;i<N;++i)
   for(j=jj; j < min(jj+B,N); ++j) {
    for(c=0,k=kk; k<min(kk+B,N); ++k)
     c+= A[i][k] * B[k][j];
    C[i][j] += c;
   }
```



block size BxB (2x2)



Il Data blocking ci permette di implementare la località temporale.

A volte è possibile riscrivere gli algoritmi a blocchi, queste versioni sono più complicate ma lavorano su porzioni piccole in modo tale che accendendoci più volte riescano a stare in uno dei blocchi di cache.

Se uno vuole ottimizzare veramente dei codici bisogna tenere conto dell'architettura sottostante, i compilatori cercano di darci un livello dove l'architettura vera è astratta, possiamo programmare liberamente senza troppi problemi, a volte però il compilatore non riesce a fare questo per nostro conto.

I BUS DI I/O

I dispositivi di I/O sono tutti i dispositivi esterni al disco, mouse, tastiera, schede di rete ecct

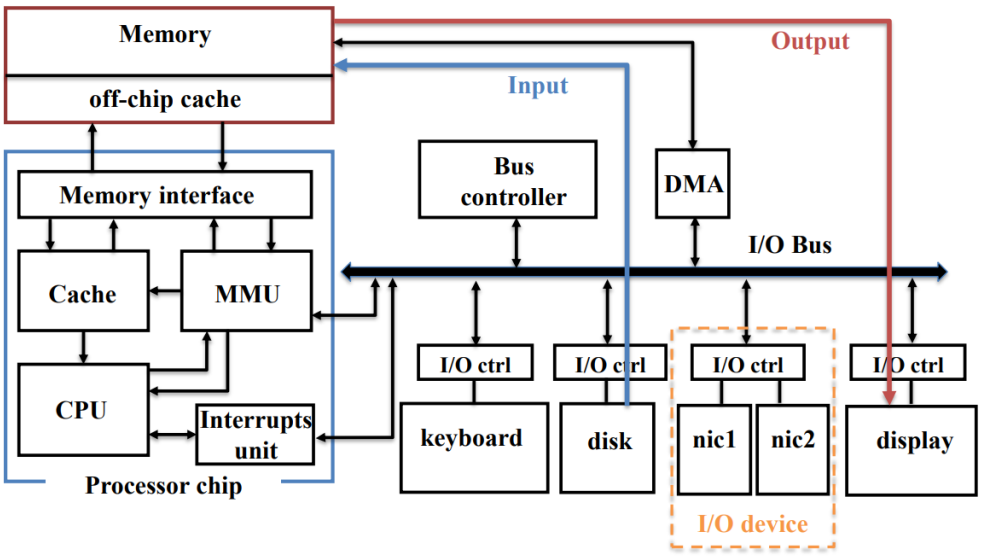
La nostra architettura rimane quella Di Von Neumann solo che fuori ci sarà il bus di I/O.

Alcuni dispositivi sono esclusivamente di input, alcuni esclusivamente di output, alcuni sia di input che di output, altri ancora richiedono un'interfaccia con l'uomo, altri invece interagiscono direttamente con la macchina.

Ci sono molti dispositivi diversi, e la cpu deve interfacciarsi quindi con dispositivi enormemente lenti o molto veloci.

Quando parliamo di dispositivi di I/O parliamo in generale, ovvero senza riferirci a qualche dispositivo in particolare.

Avevamo visto l'MMU (Memory Management Unit) come dispositivo interno al chip per interfacciare il chip con le memorie, adesso si interfaccia anche con il bus, ovvero un canale a cui sono attaccati i dispositivi esterni.



Quello che si fa è trasferire dati che sono sui dispositivi esterni e caricarli in memoria oppure trasferire dati che sono in memoria all'interno del dispositivo.

La cosa interessante è che a questo bus è attaccato il dispositivo interno alla cpu che è l'MMU. Questi dispositivi di I/O li vogliamo trattare come se stessimo indirizzando la memoria, le operazioni che vogliamo andare a fare dal punto di vista della cpu sono essenzialmente load e store, questa MMU capirà se queste operazioni sono riferite alla memoria o ai dispositivi, in modo da smistarle correttamente.

C'è anche una parte dove alcune delle unità si interfacciano, il DMA (Direct Memory Access), che ci permette di fare delle load e delle store nella memoria principale senza utilizzare il processore.

Inoltre abbiamo l'interrupt unit, perché tra i meccanismi per segnalare alla cpu che qualcosa è pronto i dispositivi di I/O hanno le interruzioni e c'è un'unità che ci permette di gestirle.

- The I/O impact on program execution time may be quite high
- Let's suppose $T_{exe} = T_{cpu} + T_{I/O}$ and $T_{I/O} = \frac{1}{10} T_{cpu}$, therefore $T_{exe} = \frac{11}{10} T_{cpu}$
- If we speed up T_{cpu} by 10 times leaving unaltered $T_{I/O}$, we have $T_{cpu}^{enhanced} = \frac{1}{10} T_{cpu}$, thus $T_{exe} = \frac{1}{10} T_{cpu} + \frac{1}{10} T_{cpu} = \frac{1}{5} T_{cpu}$
- Let's consider the $Speedup = \frac{Exec.time\ before\ enhancement}{Exec.time\ after\ enhancement}$
- The speedup obtained is $\frac{11}{2} = 5.5$

An optimization of 10-fold on the T_{cpu} produced only about 5-fold enhancement on T_{exe}! Why?

Legge di Amdahl

Si usa nel calcolo parallelo e ci dice che se abbiamo un algoritmo e una macchina con tantissimi core, se anche riusciamo a dividere la nostra app tra i core, se questa app impiega un'ora e mezza e la parte parallelizzabile dura solo un'ora, la mezz'ora dovremo pagarla sempre.

La legge di Amdhal ci dice che il guadagno è limitato dalla parte che non ottimizziamo.

$$Speedup = \frac{Exec.time\ before\ enhancement}{Exec.time\ after\ enhancement} = \frac{T(1-f)+Tf}{T(1-f)+\frac{Tf}{N}} = \frac{1}{(1-f)+\frac{f}{N}}$$

A volte può non valere la pena fare il miglioramento.

Le prestazioni nei dispositivi di I/O non sono l'aspetto più importante, si guardano anche altri aspetti come l'affidabilità e la disponibilità, un disco può anche essere velocissimo ma se si rompe spesso o non è in grado di isolare automaticamente le parti del disco è inutile.

Lo stesso vale per la memoria, deve essere veloce ma anche affidabile, se ci sono delle zone rovinate il sistema di memoria deve gestirle non facendomele utilizzare.

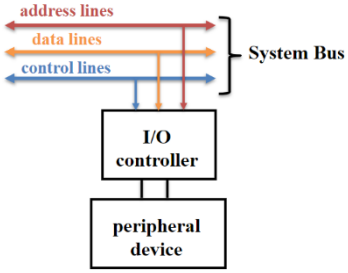
Si parla di mean time to failure MTTF e mean time to repair MTTR.

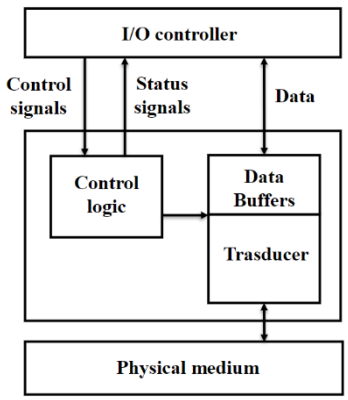
I sistemi raid sono quei sistemi che mettono insieme più dischi contemporaneamente perché magari voglio tollerare che se si rompe un disco il sistema non si ferma, oppure voglio creare un sistema che ha 5 dischi e voglio tollerare almeno la rottura di 2 dischi, esistono dei controller raid che gestiscono i dischi.

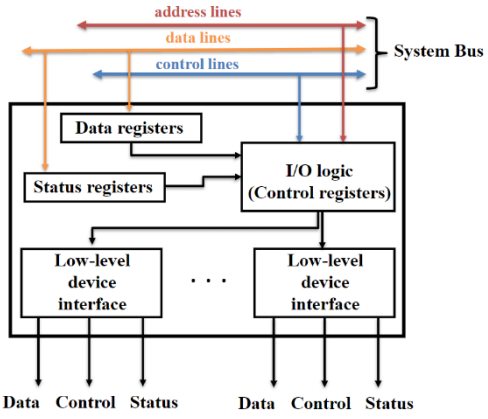
Nel sistema RAID lo spazio di memorizzazione è quasi pari alla somma degli spazi dei dischi.

Struttura dispositivi I/O

In generale un dispositivo di I/O può essere visto come un insieme di porte, ne ha almeno due, una control port e una data port, la porta di controllo è la porta in cui posso andare a scrivere comandi o leggere informazioni di stato di quel dato dispositivo, la data port invece è la porta attraverso cui posso scrivere e leggere dati.







Un dispositivo di I/O ha un controller che è in grado di interfacciarsi con la parte fisica del dispositivo e ad un bus di comunicazione, ovvero ad un insieme di linee che permettono al controller di ricevere comandi e permettono di ricevere e trasmettere da parte del controller dati relativi a quel dispositivo.

Il bus è a sua volta un dispositivo dove però i canali sono classificati, alcuni canali trasportano indirizzi e dati, altri canali che sono collegati alla porta di controllo trasmettono comandi e ci permettono di leggere lo stato.

Il controller è attaccato alla periferica non con un bus ma con dei “fili”, il bus permette di interfacciare tutti o alcuni dei dispositivi verso la memoria o la cpu.

Le funzioni principali del controller di un dispositivo sono la parte di controllo del tempo, ovvero la sincronizzazione rispetto al bus e rispetto alla periferica, ma anche la parte di controllo relativa alla ricezione di comandi e all’invio di messaggistiche di stato, il controller può avere anche funzioni di memoria, può avere al suo interno un buffer, ovvero un po' di memoria per migliorare le prestazioni.

Che cos’è questo bus di comunicazione?

Un bus di comunicazione è un mezzo di trasmissione, in generale condiviso da più dispositivi

Fisicamente è un insieme di linee che permettono la comunicazione, si chiama system bus il bus che connette componenti importanti come il processore la memoria ed alcuni dispositivi eventualmente di I/O.

Il bus collega processore, memoria e MMU ma anche processore, memoria e DMA.

Classi di BUS

Esistono 3 classi di BUS:

Il system bus che deve essere estremamente veloce perché si interfaccia con il processore, tipicamente connette pochi dispositivi e per questo è veloce, viene implementato in modo sincrono ovvero c’è un clock condiviso tra tutti i dispositivi collegati a quel bus.

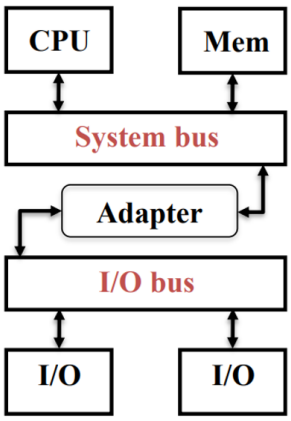
Di solito quando si parla di system bus non si ha uno standard, perché ogni casa produttrice ha dei suoi metodi.

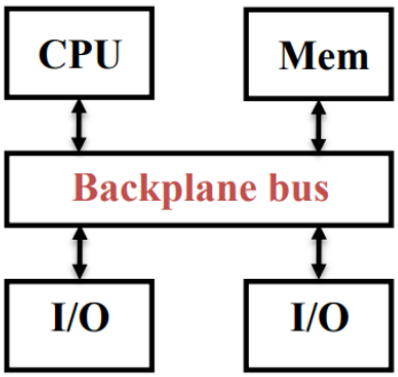
Si parla di standardizzazione quando si parla di I/O bus perché io voglio poter collegare sistemi di I/O diversi con un’unica interfaccia.

L’usb è uno standard di I/O.

Il bus di I/O si differenzia dal system bus per il fatto che posso collegare più dispositivi e posso avere lunghezze fisiche dell’ordine dei metri.

Il system bus e l’I/O bus possono essere connessi tramite un modulo adapter.





Un altro tipo di bus è il backplane bus che può essere utilizzato in alcuni sistemi per collegare vari dispositivi e creare mini-computer.

Si parla anche qui di standardizzazione, tipicamente è più lento del system bus ma ha la caratteristica di poterci attaccare più dispositivi.

Le prestazioni di un bus sono caratterizzate dal numero di dispositivi connessi e dalla lunghezza del bus.

Essendo questo bus un medium condiviso, mi servono delle tecniche di arbitraggio, più dispositivi ci metto e più difficile sarà gestirli.

BUS DESIGN

Vogliamo dispositivi performanti e magari standardizzati, a basso costo.

Ci sono molti problemi di progettazione per raggiungere questi 3 obiettivi.

Un aspetto è: quanto fare ampio il bus?

Un bus largo permette di avere alte bande di comunicazione ma sono più costosi e sono più suscettibili a problemi di sfasamento, ad esempio nei bus sincroni che hanno un clock, più li faccio ampli e più rischio di avere sfasamenti del clock.

Un aspetto più di progettazione è: come acquisisco il permesso di utilizzare il bus?

Idealmente sul bus devo fare 3 operazioni.

Come li realizzo?

Un modo è con operazioni atomiche, ovvero una volta che acquisisco il bus lo utilizzo io (dispositivo) per tutto il tempo finchè non finisco l'operazione.

Questa gestione è semplice ma non permette un ottimo utilizzo del bus.

Un'altra possibilità potrebbero essere le split transactions, ovvero vengono mandate una o più richieste di lettura/scrittura che vengono eseguite alternativamente, si spezzano le operazioni in operazioni più semplici e si alternano (interleaving).

Ci permette di ottimizzare l'utilizzo del bus ma abbiamo anche dei costi di gestione più alti.

Il bus ha un clock o no?

Si possono fare sia bus sincroni che bus asincroni.

Nei bus sincroni si ha un clock che dà il tempo a tutti i dispositivi che si collegano a quel bus.

Nei bus sincroni solitamente abbiamo pochi dispositivi (come nel system bus) quindi non corro il rischio di avere sfasamenti di clock.

Il vantaggio è che questi bus sono molto veloci, in quanto le operazioni vengono gestite tramite i cicli di clock.

I più diffusi sono però quelli asincroni, in cui se anche c'è un clock non è un clock che coordina tutti i dispositivi che si interfacciano con il bus.

C'è un problema sulla gestione del bus, non posso più fare affidamento al clock, (fronte di salita, cicli di clock passati) qua la durata dell'operazione dipende da chi sta usando il bus, si necessita quindi di un protocollo detto handshaking che permette di far accordare i dispositivi sull'utilizzo del bus.

I bus asincroni con protocolli di handshaking sono più lenti.

Chi decide chi usa il bus in caso di richieste molteplici?

Chi utilizza il bus in un certo momento prende il nome di master, gli altri dispositivi sono slave, ci possono però essere più richieste in contemporanea, serve quindi un arbitro, potremmo pensare che questo arbitro sia la CPU, però non è ragionevole che sia lei perché perderemmo della potenza di calcolo per regolare il traffico di altri dispositivi, di solito si elegge tra i dispositivi un arbitro, che è un'unità dedicata a fare solo quello. Garantisce che ci sia un master attivo per il bus in ogni istante e non più di uno, l'aspetto importante è il numero di arbitri da utilizzare, se ne usiamo solo uno si parla di implementazione centralizzata.

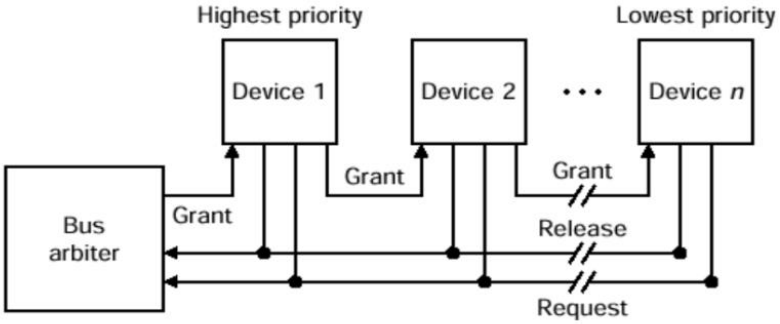
L'arbitro potrebbe essere distribuito tra tutti i dispositivi che si affacciano al bus allora mi serve un ulteriore protocollo di arbitraggio per decidere chi fa l'arbitro in un dato istante.

Arbitri centralizzati e distribuiti sono complessi e tipicamente si implementano mediante sistemi software, a livello hardware spesso si utilizzano arbitri centralizzati perché distribuiti costerebbero troppo.

Ci sono diversi modelli:

- Daisy chain
- Arbitro a richieste indipendenti

Daisy chain



È un modo semplice per implementare l'arbitraggio ma ha molti difetti, esiste un dispositivo che fa da arbitro che dice agli altri a chi tocca, è un particolare schema in cui viene data priorità ai dispositivi più vicini all'arbitro, l'ultimo è quello che ha priorità più bassa. Quando uno o più dispositivi vogliono acquisire la possibilità di usare il bus, scrivono sul filo delle richieste. L'arbitro mette in or tutte le richieste di tutti i dispositivi e cerca se c'è un 1, se c'è vuol dire che almeno uno ha fatto richiesta anche se lui non sa chi.

Quando qualcuno fa richiesta l'arbitro scrive dentro alla linea grant dando possibilità di scrittura al primo dispositivo, se questo non ha fatto richiesta forwarda il segnale al secondo dispositivo e così via finchè non si incontra il dispositivo che voleva utilizzare il bus.

Quando il dispositivo che ha fatto richiesta riceverà il grant interromperà la trasmissione.

Schema con poche porte da realizzare che però non è equo, l'ultimo potrebbe essere ritardato anche per un tempo indefinito nell'acquisizione del bus.

Arbitro a richieste indipendenti

L'arbitro a richieste indipendenti invece è sempre di tipo centralizzato però anziché avere un'unica linea di richieste ed una linea ad anello per dare il controllo del bus, ha tutti fili indipendenti, cioè ha richiesta e risposta indipendenti.

È quello più usato anche se più costoso e più lento.

Gestione I/O

Come possiamo gestire un dispositivo di I/O?

Dobbiamo considerare 3 aspetti:

- come diamo i comandi
- come possiamo sapere della terminazione del comando
- come avviene il trasferimento dei dati

Guardiamo il dispositivo dal punto di vista della cpu.

Comandi

I comandi vengono mandati dal S.O, noi che scriviamo un programma e che scriviamo printf, la printf non va a scrivere direttamente sul dispositivo schermo, ma chiede questo servizio al S.O, perché i dispositivi sono gestiti unicamente da lui, ci potrebbero essere conflitti e dobbiamo gestire lo scheduling delle richieste.

Dobbiamo sempre mediare l'accesso a qualunque dispositivo mediante il S.O che fa tutto per nostro conto. *Come possiamo fare questo?*

Possiamo farlo attraverso le chiamate di sistema, ovvero possiamo far fare al sistema operativo per nostro conto delle operazioni.

Ci sono sostanzialmente due opzioni per inviare i comandi:

La prima, poco usata oggi, è quella di avere istruzioni dedicate per l'I/O.

Oltre alle load, store, jump e compare si usano operazioni apposite per fare la scrittura e la lettura di certi dati da un dispositivo, queste si chiamano I/O instructions.

Oggi, quasi tutti (se non tutti) i sistemi usano il secondo approccio che è il memory mapped I/O, non abbiamo istruzioni dedicate, ma i dispositivi per la cpu sono visti come parti di memoria, in questo modo possiamo usare le load e le store verso indirizzi dedicati.

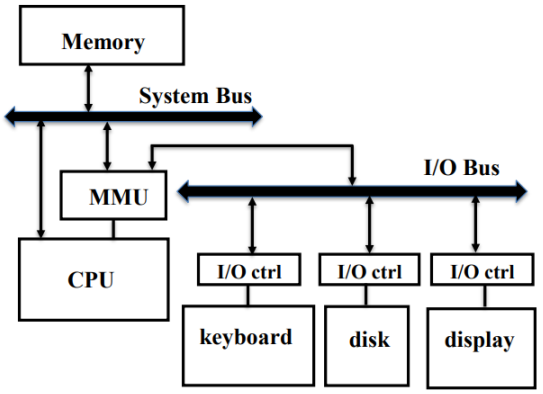
Riservo una parte dello spazio di indirizzamento del sistema (che va da 0 a 2^32-1), in modo che un certo range di indirizzi riferiscano il disco, altri il monitor e così via, quando leggo o scrivo su questi particolari indirizzi è come se stessi facendo delle istruzioni di I/O.

Nell'arm gli indirizzi I/O riferiti sono gli utlimi.

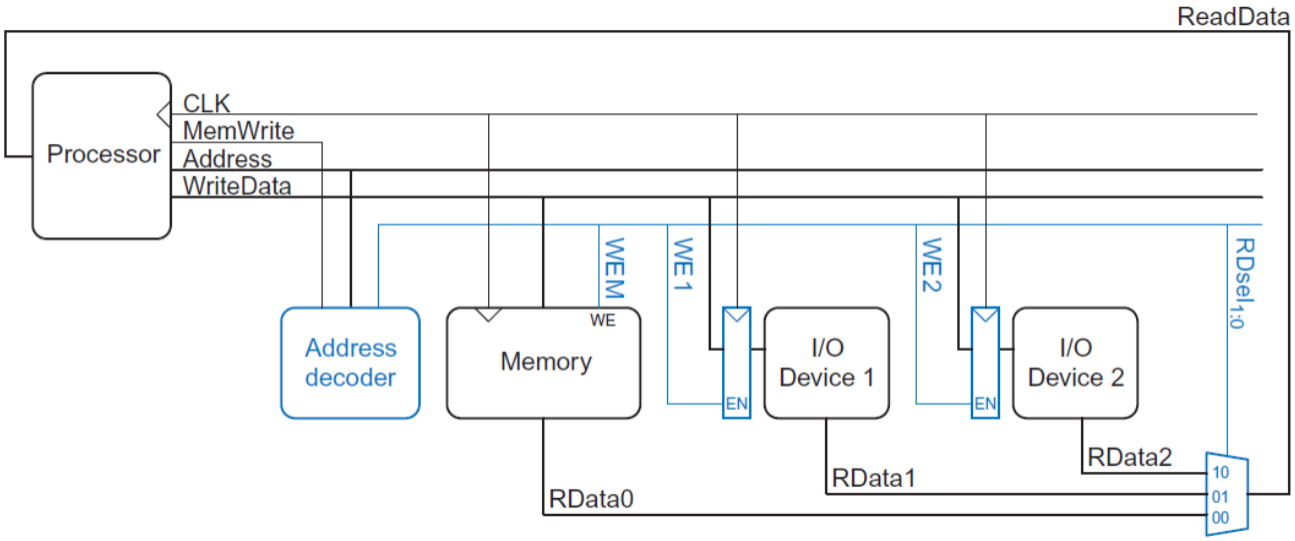
Memory mapped I/O

Il dispositivo è rappresentato come un set di indirizzi.

La CPU invia comandi di load e store all'MMU, quest'ultima capisce se quei dati indirizzi che sta ricevendo sono indirizzi di un particolare dispositivo, per la CPU non cambia nulla, la MMU capisce se ci stiamo riferendo ad una cache o ad un dispositivo.



La MMU prende infatti un indirizzo che sono 32 bit, guarda i 4 bit più significativi, se i primi due bit sono 11 l'operazione viene girata al bus di I/O, se sono qualsiasi altra cosa l'operazione va fatta sulla memoria.



Example e9.1 COMMUNICATING WITH I/O DEVICES

Suppose I/O Device 1 in Figure e9.1 is assigned the memory address 0x20001000. Show the ARM assembly code for writing the value 7 to I/O Device 1 and for reading the output value from I/O Device 1.

Solution: The following assembly code writes the value 7 to I/O Device 1.

```
      MOV R1, #7
      LDR R2, =ioadr
      STR R1, [R2]
ioadr DCD 0x20001000
```

The address decoder asserts *WE1* because the address is 0x20001000 and *MemWrite* is TRUE. The value on the *WriteData* bus, 7, is written into the register connected to the input pins of I/O Device 1.

To read from I/O Device 1, the processor executes the following assembly code.

```
LDR R1, [R2]
```

The address decoder sets $RDsel_{1:0}$ to 01, because it detects the address 0x20001000 and *MemWrite* is FALSE. The output of I/O Device 1 passes through the multiplexer onto the *ReadData* bus and is loaded into R1 in the processor.

Gestione delle operazioni

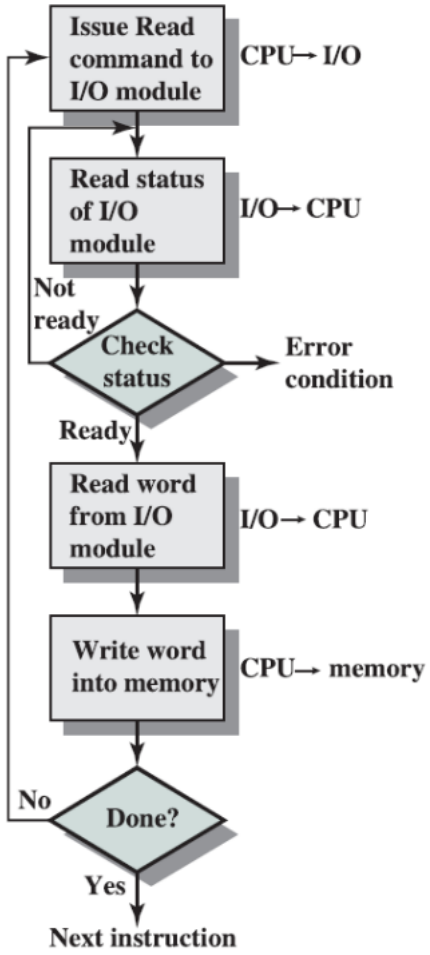
Un'interruzione è un evento asincrono che un dispositivo invia per notificare un evento ad un altro dispositivo, tipicamente alla cpu. Spesso si sente parlare del termine eccezione, sono cose diverse anche se può capitare che venga usato il termine exception per riferire sia l'una che l'altra, un'eccezione è però un evento sincrono che avviene a seguito di un particolare evento deterministico.

La distinzione principale è che le interruzioni sono eventi asincroni quindi non posso prevedere quando arriveranno.

Dei due approcci, programmed I/O e interruzioni partiamo dal primo, vogliamo sapere quando una certa operazione è terminata, con il modello programmed I/O la cpu svolge un ruolo attivo e il dispositivo un ruolo passivo, in particolare la CPU esegue un ciclo che testa continuamente lo stato di quel dispositivo, si chiede ogni volta se ha finito, se non ha finito potrebbe fare qualcos'altro, però poi deve testare nuovamente se quella particolare operazione è terminata, ripete questo controllo fino a che l'operazione non è completata.

Esiste una sequenza di istruzioni che viene fatta per testare continuamente se quella data operazione è terminata, questo ciclo di testing si chiama anche polling.

Polling



Oltre che polling si parla anche di busy waiting, facciamo continuamente un'attesa attiva.

Pro e contro:

Il vantaggio di questo approccio è che è molto semplice da implementare e richiede poche istruzioni, il costo della sua implementazione è relativamente basso.

Ha però un enorme svantaggio, ovvero il fatto che la cpu quando si trova ad eseguire questo piccolo programma per il testing di una condizione sull'I/O non fa altro, perde tutti i cicli di clock per testare una condizione che certamente si verificherà, ma magari tra tantissimi cicli di clock che andranno persi.

Come tecnica è una tecnica utilizzabile ma dato che la cpu è enormemente più veloce di un dispositivo di I/O si usa in pochissimi casi, si usa quindi quasi sempre l'altro modello, ovvero il modello guidato dalle interruzioni.

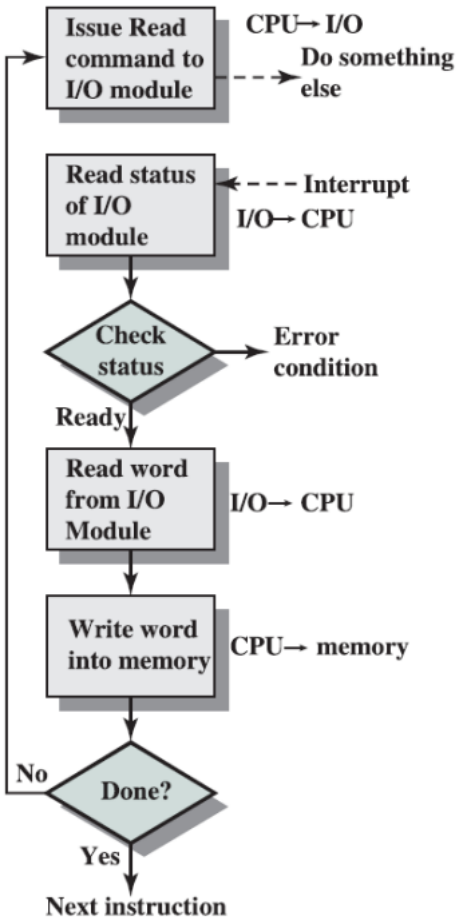
Interruzioni

Questo modello cerca di risolvere il problema principale del polling, non vogliamo stare a testare continuamente la condizione bensì nel frattempo vogliamo fare altro, ci dovrà essere però qualcuno che ci dica che l'operazione è terminata, questo meccanismo è l'interruzione ovvero un evento asincrono inviato dal dispositivo stesso per dire che ha finito.

La cpu sente questa notifica, questa interruzione e sa che una certa operazione che era ancora pendente è stata completata.

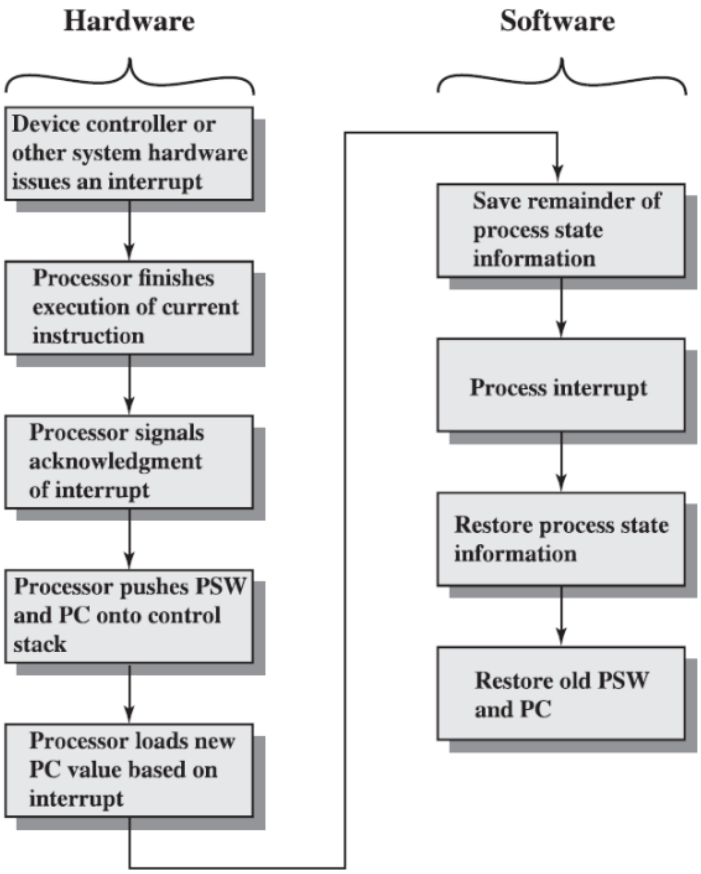
C'è un'unità che ci invia un'interruzione quando ha finito, l'interruzione fa entrare in gioco l'esecuzione di qualcos'altro. Per cambiare dal contesto attuale all'esecuzione di qualcos'altro serve avere dei privilegi particolari che solo il sistema operativo ha, dunque queste interruzioni sono fatte al livello kernel del sistema operativo.

Essendo le interruzioni urgenti da gestire, vedremo che alcune parti della gestione delle interruzioni, alcune sequenze sono fatte in modo atomico, in alcuni casi la gestione di un'interruzione non è interrompibile da un'altra interruzione.



PSW is the Program Status Word register, the equivalent of the **CPSR/APSR** register for ARM processors

```
while(true) {
  try {
    IR = fetch(PC);
    decode(IR);
    execute(IR);
    update(PC);
  } catch (exception e) {
    exception_management();
  }
  if(interrupt) {
    interrupt_management();
  }
}
```



Trasferimento dei dati

Il data transfer con DMA (direct memory access) richiede le interruzioni.

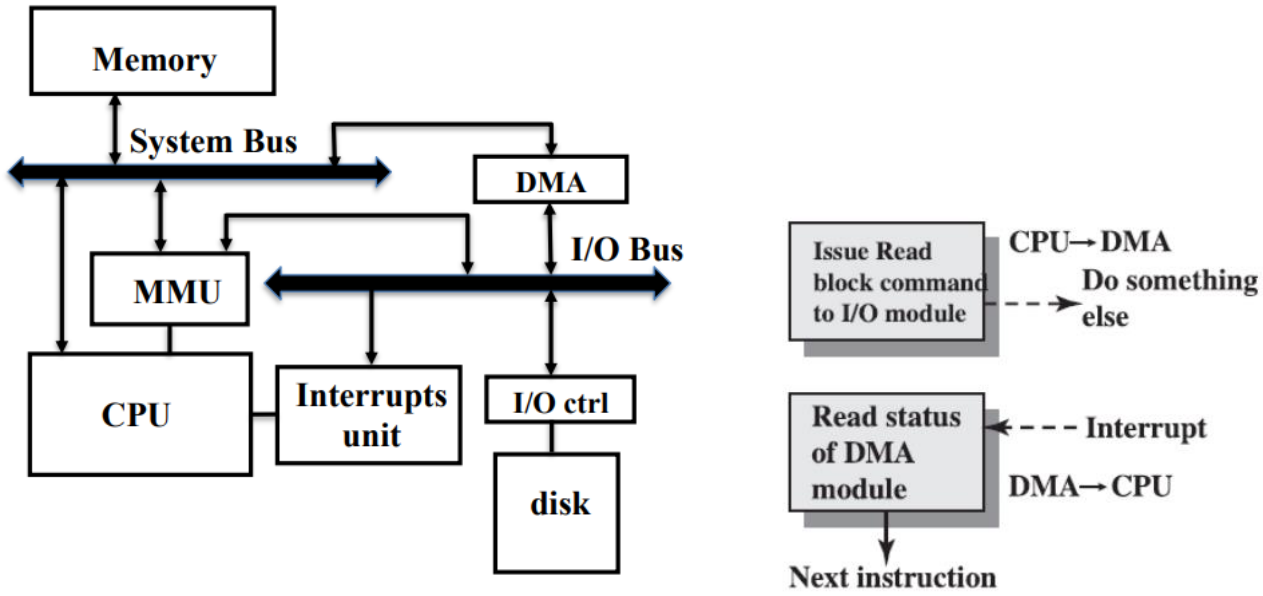
Il DMA si occupa di fare il trasferimento dal buffer dei dispositivi direttamente in memoria, adesso posso sovrapporre non solo la lettura dei dati del dispositivo ma anche il trasferimento in memoria, il controller del dispositivo ha una logica più complessa ed è in grado anche di accedere al bus, fare il protocollo di arbitraggio del bus, chiedere alla memoria di scrivere i dati ed inviare un'interruzione alla cpu tramite le unità di interruzione, ha una logica più complessa rispetto a semplicemente controllare i buffer del dispositivo.

In questo modo risparmiamo cicli di cpu.

È costoso ma ne vale la pena perché alleggerisce il carico della CPU.

Voglio avere qualcuno che mi notifichi quando qualcosa è pronto.

Il concetto del DMA è che è direttamente l'unità di I/O a spostare i dati tramite un controllo specializzato, dunque il vero diagramma è:



La cpu emette il comando di trasferimento, quando gli arriva l'interruzione non va più a testare lo stato del dispositivo interno ma testa lo stato del Dma per verificare se il trasferimento è andato a buon fine, se sì avrò già i dati in memoria dove la cpu ha richiesto di metterli.

Il DMA non solo gestisce le periferiche ma trasferisce anche i dati in memoria.

La cpu dovrà scrivere in certe zone di memoria per mandare un comando di tipo DMA su un certo dispositivo, quindi manderà sicuramente l'operazione che vogliamo sia effettuata, non eseguo direttamente load o store ma un'operazione identificata come trasferimento DMA, chiedendo a lui poi di fare load o store, ovvero di scrivere dal dispositivo alla memoria o viceversa.

Devo dire anche da dove prendere i dati se faccio una store, o dove mettere i dati se faccio una load, la CPU quando fa il setup del DMA-transfer passa tutte queste informazioni tramite il meccanismo del memory mapped, cioè in particolari zone di memoria, il DMA legge queste operazioni e opera in autonomia, quando ha finito manda l'interruzione alla CPU dicendo che ha finito.

A questo punto la CPU sa di poter trovare i dati all'indirizzo specificato.

Abbiamo liberato la CPU dal fare operazioni ripetitive o poco utili come trasferimenti e polling, grazie al meccanismo delle interruzioni e del DMA.

Non abbiamo però risolto tutti i problemi in quanto da un lato abbiamo aumentato il costo, ci costa gestire le interruzioni e avere controller in grado di fare DMA, però abbiamo introdotto anche un altro problema, adesso abbiamo due unità che possono leggere e scrivere dalla memoria, se leggo posso leggere in parallelo, se scrivo chi scrive per prima?

Il bus come abbiamo visto è gestito tramite protocolli, se l'accesso lo vuole avere la cpu o l'MMU, il DMA aspetta e viceversa. Da un lato abbiamo migliorato le prestazioni perché riusciamo a sfruttare meglio tutti i cicli, dall'altro abbiamo aumentato le probabilità di stallare il processore, perché quando il bus è occupato se la cpu vuole fare una store o una load sta ferma e deve aspettare, in questo caso si parla di DMA e interrupt breakpoint.

Nel normale ciclo di fetch-execute, questa cosa vuol dire che quando la CPU vuole fare il fetch della prossima istruzione o del prossimo operando potrebbe in quel momento non riuscire a farlo perché il bus è occupato, questa attività potrebbe essere quindi ritardata per un po'.

La gestione delle interruzioni invece è voluta, in quanto non è stallo del processore bensì si passa a gestire l'interruzione dopo aver fatto altro.

Per quanto sia costosa la gestione dell'interruzione faccio del lavoro utile, mentre se il bus è occupato perché lo sta usando un'altra unità io non posso fare nulla.

- **Assumptions**:
  - 500MHz CPU, disk device with 4MB/s transfer rate, 16B interface, 50% utilization
  - Interrupt handling takes 400 cycles
  - Data transfer takes 100 cycles
  - DMA setup takes 1600 cycles, it transfers a 16KB block at a time
- Processor overhead for **interrupt-driven I/O without DMA**
  - The processor is involved for each data transfer (16 B)
  - 0.5 * (4 MB/s) / (16 B) * [(400+100) cycles / 500M cycles/s] = **12.5 %**
- Processor overhead for **interrupt-driven I/O with DMA transfer**
  - The processor is involved once per block transfer (16 KB)
  - 0.5 * (4M B/s)/ (16KB) * [(1600+ 400) / (500 M cycles/s)] = **0.05 %**

Che succede tra DMA e gerarchie di memoria?

La cpu dice al DMA controller dove leggere o scrivere i dati.

Finora abbiamo sempre pensato di operare con indirizzi fisici, in quanto non ci eravamo posti il problema di avere un indirizzo diverso da quello della locazione vera dove sono memorizzati i dati. In realtà le cose sono più complicate perché per avere flessibilità nell'esecuzione dei programmi la cpu non lavora con gli indirizzi fisici della memoria, ma lavora con indirizzi logici, ovvero un'astrazione, dove non è detto che vi sia una corrispondenza 1:1 tra indirizzo logico ed indirizzo fisico, vedremo più avanti che esisterà una funzione di traduzione tra indirizzo logico ed indirizzo fisico.

Dal punto di vista della memorizzazione non cambia nulla, i dati sono in locazioni fisiche, ma l'istruzione che si trova ad un determinato indirizzo logico non è detto che si trovi a quell'indirizzo anche nella memoria fisica.

Se opero con indirizzi logici il DMA come la CPU deve essere in grado di effettuare la traduzione, se invece operiamo con indirizzi fisici, il DMA opera con indirizzi fisici, la CPU sicuramente opera con indirizzi logici. Dovremo trovare il modo nella fase di setup di fare la traduzione da indirizzo logico a fisico, ci sono due soluzioni:

Una è la virtual DMA e l'altra è la physical DMA; la virtual DMA è tutto quello che abbiamo visto finora a parte il fatto che il DMA è in grado di lavorare con indirizzi logici, quindi deve essere in grado di effettuare la traduzione degli indirizzi.

Se invece si ha un physical DMA, questa vuole un indirizzo di memoria fisico dove andare a leggere e a scrivere. Ovviamente il primo è più complesso e costoso da realizzare ma è molto più flessibile, il secondo è più semplice ma meno flessibile.

Il setup del virtual DMA sicuramente mi costerà di più sia in termini economici che temporali, avrà ad esempio bisogno di una cache interna con tutti i relativi costi di gestione, si ha un trade off tra molti fattori, non esiste una soluzione lineare perfetta.

L'altro aspetto è l'interazione con le cache, immaginiamo una cache di tipo write back, in cui gli aggiornamenti li scriviamo in cache ed evitiamo di aggiornare il livello superiore.

Questo in presenza di DMA rischia di creare qualche problema, avrei dei dati scritti in cache con politica WB e potrei aver ordinato al DMA di scrivere su disco degli indirizzi che sono in cache aggiornati, dovrei quindi invalidare la linea di cache prima di fare il setup del DMA, invalido la linea di cache così da far propagare le scritture, ma in questo modo la cache andrà più lenta perché butto fuori qualcosa che eventualmente avrei ritardato, scrivere qualcosa in memoria significa occupare il bus di sistema ecct…

La cache migliora le prestazioni, però se non c'è località spaziale o temporale le cache ci fanno andare peggio, al solito è un trade off di scelte complesse che dipendono da molti fattori.

Driver

I dispositivi li gestisce il sistema operativo, le applicazioni fanno system call e il S.O gestisce i vari device.

Il S.O gestisce i dispositivi utilizzando i device driver, ovvero un software che gira in kernel space che si interfaccia con i vari dispositivi e ha diritti speciali.

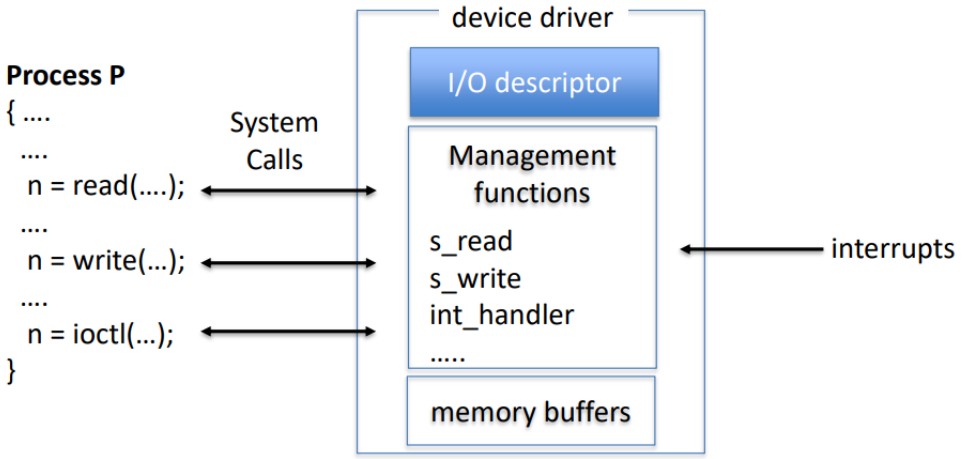
In generale l'avere tanti dispositivi crea anche un problema lato software.

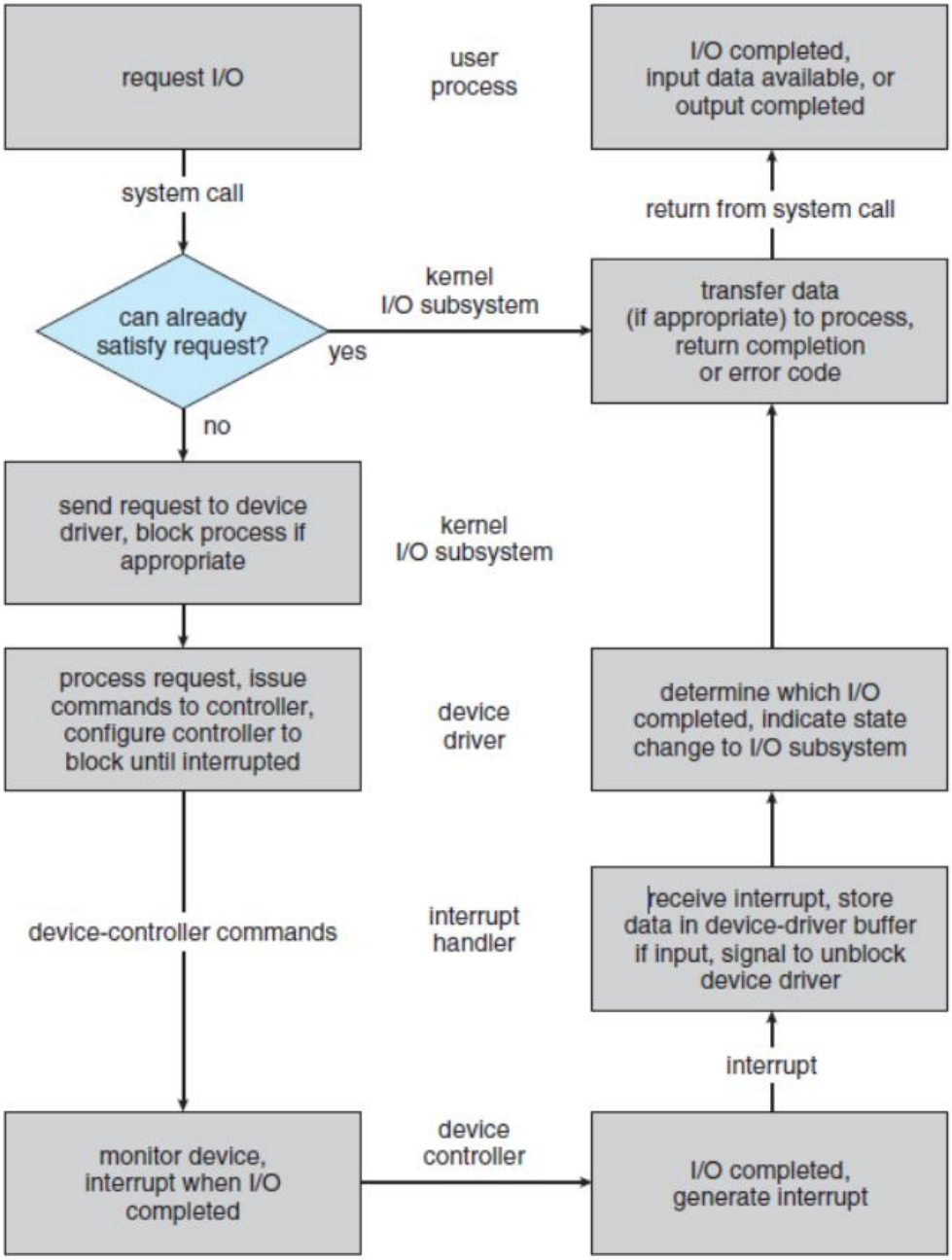
Questi driver sono software che hanno il compito di interfacciarsi con il dispositivo vero e proprio, facendo memory mapped I/O e gestendo le interruzioni, la sincronizzazione e la gestione dei fallimenti.

Un device driver ha una sua parte che chiamiamo I/O descriptor ovvero una sorta di descrittore che ha tutti i metadati che descrivono il dispositivo nello specifico.

In più avrà un insieme di funzioni di handle per le singole chiamate.

Ha anche un buffer per fare bufferizzazione in kernel space.


device driver
Process P
{ ....
....
n = read(....);
....
n = write(...);
....
n = ioctl(...);
}
System Calls
I/O descriptor
Management functions
s_read
s_write
int_handler
.....
memory buffers
interrupts


request I/O
user process
I/O completed, input data available, or output completed
system call
return from system call
can already satisfy request?
yes
kernel I/O subsystem
transfer data (if appropriate) to process, return completion or error code
no
send request to device driver, block process if appropriate
kernel I/O subsystem
process request, issue commands to controller, configure controller to block until interrupted
device driver
determine which I/O completed, indicate state change to I/O subsystem
device-controller commands
interrupt handler
receive interrupt, store data in device-driver buffer if input, signal to unblock device driver
interrupt
monitor device, interrupt when I/O completed
device controller
I/O completed, generate interrupt

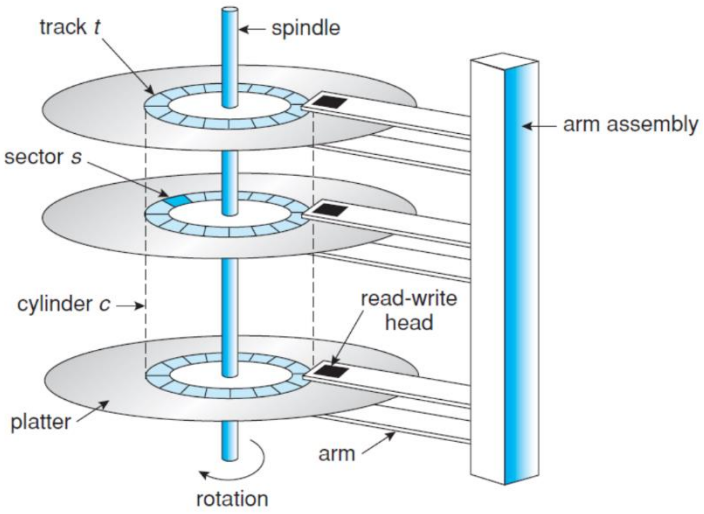
Dischi

I dischi si dividono in dischi magnetici e memorie flash.

I dischi meccanici sono dei dischi impilati che girano ad alta velocità con una testina che permette di leggere i dati, sono dischi molto affidabili e hanno grandissime capacità, l'accesso al disco non viene fatto al byte bensì al blocco, di solito si prendono 256 byte, funzionano molto bene se facciamo scritture e letture sequenziali, con scritture e letture random no.

L'altra tecnologia moderna che abbiamo ovunque sono i dischi basati su tecnologia flash memory che sono gli ssd, non hanno parti meccaniche a differenza degli hard disk, quindi sono più affidabili in quanto ad esempio possono resistere agli urti, in generale hanno una capacità di memorizzazione minore, si accedono anch'essi a livello di blocco e hanno buone prestazioni per qualsiasi tipo di accesso.

Hard Disk



Il disco si divide in un certo numero di piatti che ruotano ad altissima velocità, ogni piatto si può leggere e scrivere su entrambe le facce, le testine scorrono entrambe le facce e sono in grado di leggere e scrivere, la circonferenza di un piatto si chiama traccia, ogni traccia ha un certo numero di settori, l'insieme delle tracce alla stessa distanza dal centro di tutti i piatti si chiama cilindro.

Siamo in grado di leggere e scrivere settori in una traccia.

Le dimensioni sono dell'ordine dei micron, i vari settori delle tracce non sono appiccicati bensì distanziati, visto che il braccio è meccanico e lo spostamento è meccanico serve un certo margine di tolleranza, anche i settori hanno una parte iniziale che permette di fare il setting del braccio, per ogni settore, abbiamo poi una parte di dati magnetizzata e una parte finale che contiene dei codici di errore per controllare se quello che c'è stato scritto è corretto, la dimensione di queste tracce se consideriamo dischi di questa dimensione varia, quelle esterne sono infatti più grandi di quelle interne, la densità più alta si ha al centro dunque non va bene, per questo le tracce usate sono solo quelle esterne.

Cerchiamo di avere una densità di settori per traccia uguale per tutti. La tecnologia di questi dischi è molto sofisticata, ci sono settori e tracce di riserva, se una parte del disco si danneggia riusciamo ad isolare tracce e settori danneggiati per utilizzarne altri, un'altra cosa interessante è che se consideriamo una data traccia il settore logico della traccia più interna è disallineato rispetto al settore della traccia più esterna per permettere di fare lo spostamento.

In generale i dischi magnetici al contrario dei dischi ssd per quanto si possano smagnetizzare hanno una durata maggiore.

Dal punto di vista logico, come si gestiscono?

I valori tipici sono dell'ordine di ½ k o ¼ k o 1 k, ogni settore è identificato da una tripla, il cilindro, la faccia e il settore, il controller del disco usa questi 3 identificatori per individuare il settore che deve essere letto o scritto. Ad alto livello il driver che si interfaccia con il disco lavora a blocchi logici che sono più grandi del settore che deve essere scritto, 2-4-8k ci deve essere una funzione di traduzione tra il blocco logico che il driver chiede al controller e la tripletta che identifica il particolare settore sul disco.

Data una tripla <cilindro, faccia, settore> possiamo recuperare il blocco logico, è come se srotolassimo tutte le facce.

Given a sector number b, and a triple < c, f, s>:

$$b = c(\#faces \cdot \#sectors) + f(\#sectors) + s$$

Se guardiamo l'aspetto delle prestazioni di un disco la sua capacità di fornire informazioni dipende dal tempo di rotazione da quello di trasferimento e dal tempo di movimento della testina sul disco.

Per quanto veloce difficilmente riesco a stare sotto 1ms, la rotazione avviene in ordine (1-20 ms) questa differenza si ha perché dobbiamo considerare il caso peggiore ovvero quando la cella che devo leggere è la precedente di quella appena letta (dovrò fare quindi un giro completo).

Il transfer rate ha ordini di 5-10 us (nano secondi).

How long to complete 500 random disk reads, in FIFO order?

- Seek: average 10.5 msec
- Rotation: average 4.15 msec
- Transfer: 5-10 usec

500 * (10.5 + 4.15 + 0.01)/1000 = 7.3 seconds

Con letture o scritture random il disco è davvero troppo lento 500 letture da 1-2k sono quasi 1 megabyte, davvero poco in troppo tempo.

- How long to complete 500 sequential disk reads?
  - Seek Time: 10.5 ms (to reach first sector)
  - Rotation Time: 4.15 ms (to reach first sector)
  - Transfer Time: (outer track)

    500 sectors * 512 bytes / 128MB/sec = 2ms

Total: 10.5 + 4.15 + 2 = 16.7 ms

Might need an extra head or track switch (+1ms)

Track buffer may allow some sectors to be read off disk out of order (-2ms)

Con letture o scritture sequenziali il tempo diminuisce di molto, ci sono 3 ordini di grandezza di differenza.

Il modo di organizzare i dati sul file system (deciso dal sistema operativo) è fondamentale.

Disk with 100 cylinders, 4 faces, 2000 sectors per track.

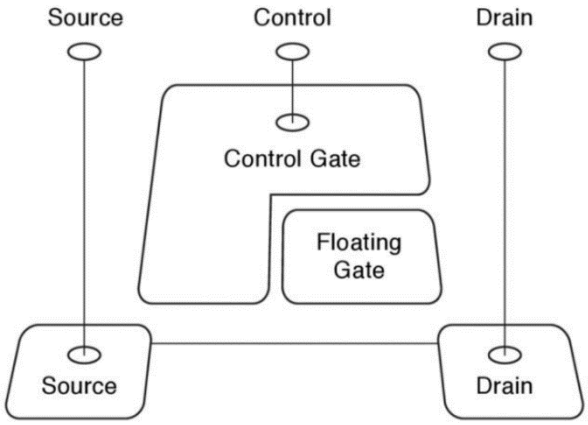
A file uses blocks 94421,94422,94423 (<11,3,421> <11,3,422> <11,3,423>). A sector is read in 0.01ms, the seek time between two consecutive tracks is 0.01ms, and the average time to reach the desired sector is half the rotation time. Considering that the arm is at cylinder 22, and that the DMA controller has enough buffers, compute the time to read the file blocks.

(22-11)*0.01 + 20/2 + 3*0.01 = 10.14ms

Dischi SSD

Tutti i problemi dell'hard disk sui dischi ssd non ci sono perché non hanno nulla di meccanico, ogni accesso costa sempre uguale, c'è però una differenza tra il costo delle letture e quello delle scritture, queste ultime sono molto più lente.

Dal punto di vista tecnologico gli ssd hanno 3 porte, la porta di controllo serve per scrivere, per caricare i dati uso la floating way (floating perché non è attaccata a nulla) una volta che è stata caricata permette di scaricarsi dopo molto tempo, è una specie di transistor evoluto, se sottopongo una corrente abbastanza forte riesco a caricare il floating gate, se questa parte è carica, e faccio passare una corrente da source verso drain, là il floating gate crea un canale tra le due parti grazie al campo elettrico.



È un sistema molto più costoso rispetto ai dischi.

Non importa dove vado a scrivere, è indipendente dalla posizione, il problema non è la lettura che non è particolarmente costosa perché si tratta solo di dare una corrente, sono le scritture invece ad essere molto delicate, non possiamo andare a scrivere in una certa zona se questa non è clean, per pulire una data zona del disco non lo posso fare in modo puntuale, ovvero per le poche celle che voglio usare, ma lo devo fare per celle abbastanza grandi perché le correnti che sto usando sono molto forti.

Se voglio scrivere una pagina di memoria, ovvero un blocco di dati, che sono di 2-4k il disco per motivi tecnologici mi può scrivere solo zone di 128-512k, quindi anche se voglio scriverne solo 4 è come se ne scrivessi 128-512, il controllo del disco deve quindi verificare che queste zone clean ci siano, altrimenti dovrei prima cancellare una zona, ripulirla, e poi fare la scrittura.

La cosa difficile è l'aggiornamento, in quanto non posso aggiornare una singola parte, devo scrivere il nuovo blocco in una parte pulita e poi rimuovere l'altra, per un blocco piccolo stiamo consumando tantissime risorse.

I consumi elettronici sono comunque minori di quelli del disco.

Le varie parti del disco non posso scriverle per quante volte voglio, è come se avessi un numero massimo di scritture che posso fare prima che il disco si rompa. Il controllore deve

tenere conto di quante volte ha scritto una zona rispetto ad un’altra per evitare di avere zone più consumate rispetto ad altre.

Esiste una parte del controller che si chiama flash translation layer che si occupa della gestione delle scritture, c’è una profonda differenza tra le scritture fatte dall’hard disk che sono gestite dal sistema operativo, negli ssd non è il sistema operativo ma il FTL a tenere traccia di tutti i blocchi puliti/scritti.

Nei primi dischi i driver si occupavano di tenere traccia di quali blocchi logici erano occupati oppure liberi.

Quando gli stessi driver sono stati utilizzati per la gestione dei dischi ssd si notò che dopo un po' di utilizzo le prestazioni degli ssd calavano notevolmente perché l’FTL mappava sempre in cerca di nuove aree, ma quando il S.O cancellava non veniva comunicato all’FTL e lui ignorava dunque la nuova capacità disponibile, si è quindi dovuto modificare i driver per permettere questa comunicazione così da far diventare le operazioni uniformi, questa cosa è stata introdotta solo nel 2009/2011 dopo che fossero stati introdotti gli ssd.

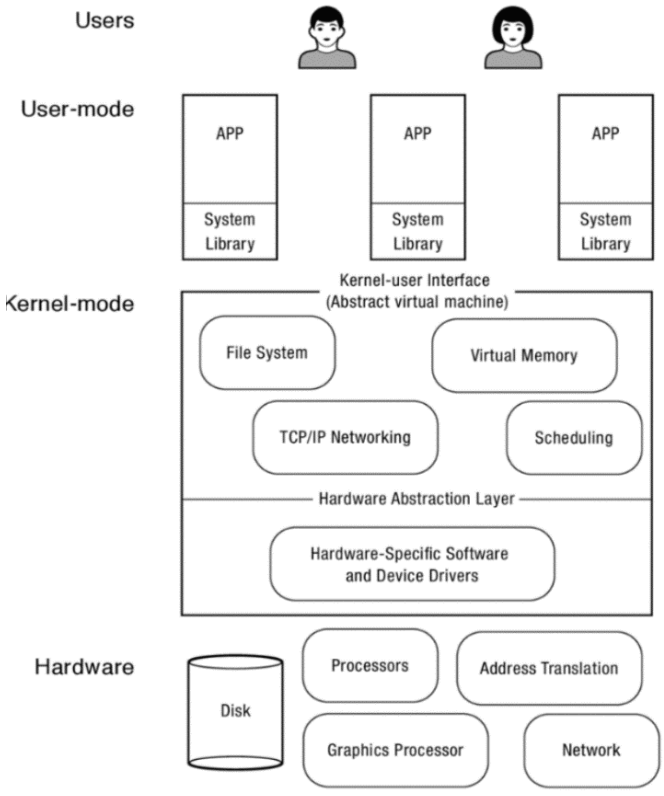
Il sistema operativo

E’ l’insieme delle risorse software che permettono di gestire le risorse del computer, ovvero non solo la cpu ma anche le periferiche e tutto quello che sfrutta il compilatore, ha il compito di mettere a disposizione queste risorse all’utilizzatore, il s.o infatti astrae dalle risorse e permette di utilizzarle in modo più semplice.

Il sistema operativo è un’interfaccia tramite cui le applicazioni riescono a sfruttare le sue risorse, ovvero le chiamate del sistema operativo.

I compiti più importanti del S.O sono:

- La gestione delle richieste e delle risorse
- La gestione della memoria virtuale
- Gestione dei processi e delle interazioni tra questi.



Possiamo distinguere 3 tipi di ruoli del sistema operativo:

Il primo è quello dell’arbitro, il sistema operativo fa da arbitro per le richieste di accesso alle risorse. Ma c’è anche un ruolo di arbitraggio nella gestione della memoria allocata dalle applicazioni. Deve anche risolvere i problemi legati alle comunicazioni tra utenti e applicazioni.

Il secondo è quello dell’illusionista, è il ruolo fondamentale, illudiamo infatti l’utente facendogli credere di avere a disposizione delle risorse che in realtà non ha.

Il terzo è quello di collante, cioè il S.O ci permette di mettere le cose insieme e di farle funzionare, mette a disposizione interfacce tra le applicazioni in modo da farle lavorare in modo uniforme sulle risorse messe da lui a disposizione.

Example: File Systems

• Referee:
  – Prevent users from accessing each other's files without permission
  – Even after a file is deleting and its space re-used

• Illusionist:
  – Files can grow (nearly) arbitrarily large
  – Files persist even when the machine crashes in the middle of a save

• Glue:
  – Named directories, printf, …

Design patterns

Cloud Computing:

L'arbitraggio è la gestione delle risorse tra le varie app.

L'illusione è che le macchine cambiano continuamente come si può isolare un'applicazione dalla sua evoluzione?

Le macchine nel cloud sono disomogenee, ho bisogno dunque di far sì che appaiano omogenee.

Il collante è l'interfaccia che mi permette di accedere al cloud.

Web Services:

L'arbitraggio è garantire un tempo minimo di risposta.

L'illusione è data dal fatto che sono generalmente distribuiti ma dobbiamo farli funzionare come se fossero tutti insieme.

Il collante è come far funzionare script su browser che provengono da SO e HW diversi.

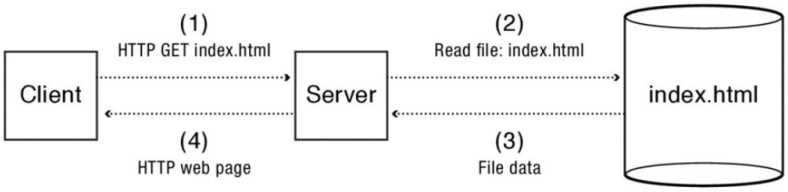
Multi-user database systems
- Referee: how to enforce data access and privacy to different users ?
- Illusionist: how to mask failures so that data remains consistent and available to users?
- Glue: what common services to programs development?

Internet
- Referee: guarantee differentiated services to users and protect against DoS, spam, phishing etc…
- Illusionist: internet appears as a unique, world-wide network but it is not!
- Glue: internet protocols make applications independent of the underlying network architecture

Tutte queste cose servono per mascherare processi che accadono di nascosto.

Ci sono molti processi che fanno affidamento sul fatto che ci sia il sistema operativo a gestire varie cose, un esempio sono le richieste, ma anche i processi ecct…

Esempio: Web Service



• It defines a simple behavior but …
• How does the server manage many simultaneous client requests?
• How do we keep the client safe from spyware embedded in scripts on a web site?
• How do make updates to the web site so that clients always see a consistent view?

However:
• Multiple users issue requests at the same time
  – These must be managed simultaneously
• Many requests involve data and computations
  – Think about search engines, a request may involve deep computations over large clusters of machines
• The server uses caches to speed up
  – Cache is shared among users, need for synchronized access mechanisms
• Servers send to clients scripts for pages customization
  – How does the client can protect itself from the execution of third party code that may embed viruses/spyware?
• Web sites need to be updated
  – How to manage consistency with concurrent read requests?
• Client and server may run at different speeds
  Need for speed decoupling
• Hardware supporting the web site may be updated
  – How to take advantage of this without rewriting the web server code?

Sfide del S.O

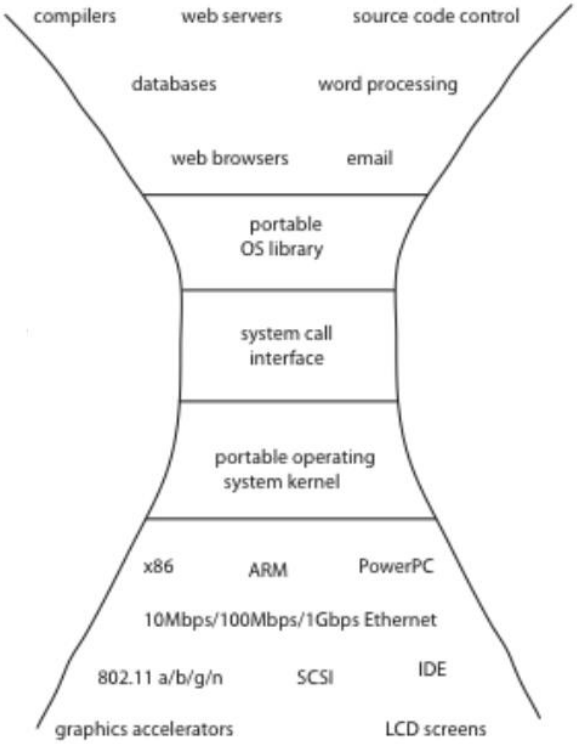
Ci piacerebbe che il S.O avesse tutte queste proprietà, ce ne sono però alcune più importanti delle altre, una è ad esempio l'affidabilità, vogliamo garantire che in situazioni diverse il sistema rimanga affidabile. Fa parte dell'affidabilità anche la disponibilità del sistema, il S.O deve infatti essere disponibile sempre.

La sicurezza è un altro punto critico, vuol dire infatti privacy, ma anche sicurezza nel trattamento dei dati, voglio che non siano alterati e che siano custoditi fino ad un futuro utilizzo.

La portabilità del sistema operativo è importante. Portabilità significa poter usare lo stesso oggetto su macchine di tipo diverso, vogliamo quindi due cose:

Se ho un insieme di programmi questi devono fare riferimento all'interfaccia portabile del sistema operativo, perché se io dovessi cambiare l'hardware, ma continuassi a mettere a disposizione questa interfaccia in modo uniforme potrei continuare ad usare questi programmi in qualsiasi caso.

Vorremmo anche che fosse portabile il sistema operativo, vogliamo lo stesso livello di disaccoppiamento anche tra sistema operativo e parte hardware, vorremmo un'interfaccia che astraesse i livelli hardware, questo implica che vorrei poter prendere il codice di un qualunque S.O ricompilarlo opportunamente e farlo girare su macchine con hardware di tipo diverso.



La cosa altrettanto importante è che tutte queste features garantiscano un alto livello di performances, dobbiamo garantire infatti dei tempi minimi.

Possiamo richiedere una certa latenza, un certo troughput ovvero quanti processi posso mandare in esecuzione in un'unità di tempo, tutte queste operazioni introducono un overhead ovvero un carico di operazioni e quindi di tempo che non avrei se non avessi il S.O. Si deve avere fairness nell'assegnazione delle risorse, ovvero tutti gli utenti devono avere parità nell'accesso alle risorse.

Le cose devono essere predicibili, ovvero devo avere un'idea che certe cose accadano entro un certo limite di tempo.

Struttura del S.O

Il sistema operativo al suo interno può essere fatto in tanti modi diversi, ci sono vari moduli specializzati per la gestione di varie funzionalità.

Questi moduli si devono sincronizzare e devono comunicare tra di loro.

Il file system e la memoria virtuale hanno bisogno di condividere qualcosa, i buffer su cui leggo e su cui scrivo sono buffer della memoria virtuale, il file system può dipendere dai protocolli di rete.

Ci sono diversi modelli:

- Nucleo monolitico: In questo approccio tutto il sistema operativo è un'unica parte con dentro tutto quello che serve, quando modifichiamo anche uno solo di questi moduli di fatto stiamo modificando l'intero sistema operativo.

- Microkernel: In questo caso invece abbiamo poche cose nel nucleo, lo schedulatore dei processi, la memoria virtuale e poco altro, tutto il resto è implementato al di sopra del microkernel.

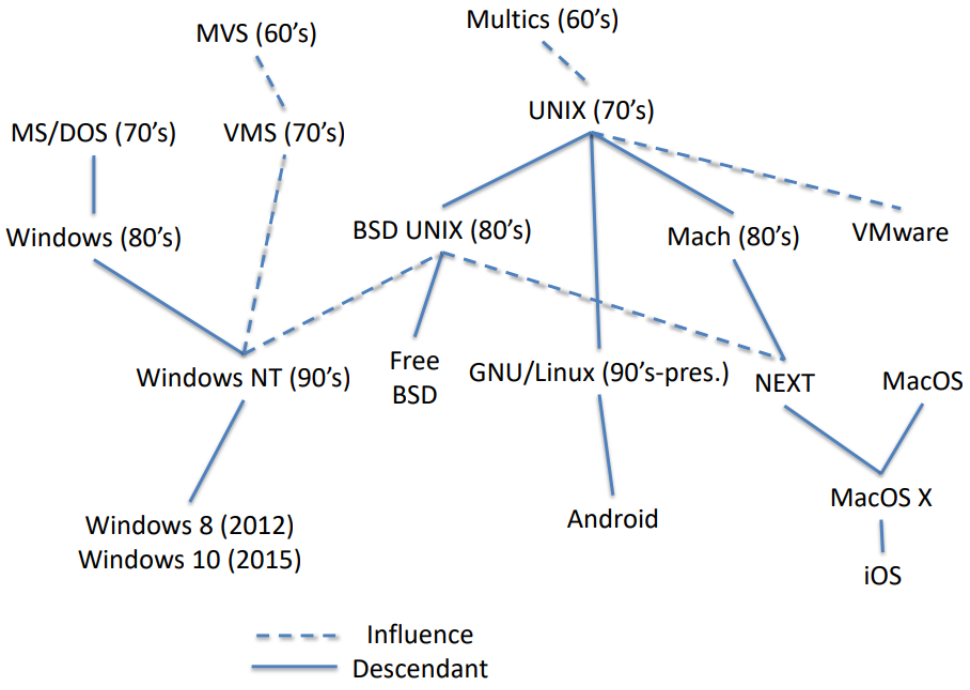
- Modello ibrido: Abbiamo i gestori della memoria e il file system all'interno del kernel il quale però è stratificato e possiamo riconoscere le diverse parti.

All'inizio tutti i S.O erano a nucleo monolitico ma dopo siamo passati a microkernel, infatti se riusciamo a dividere le responsabilità possiamo migliorare la gestione.

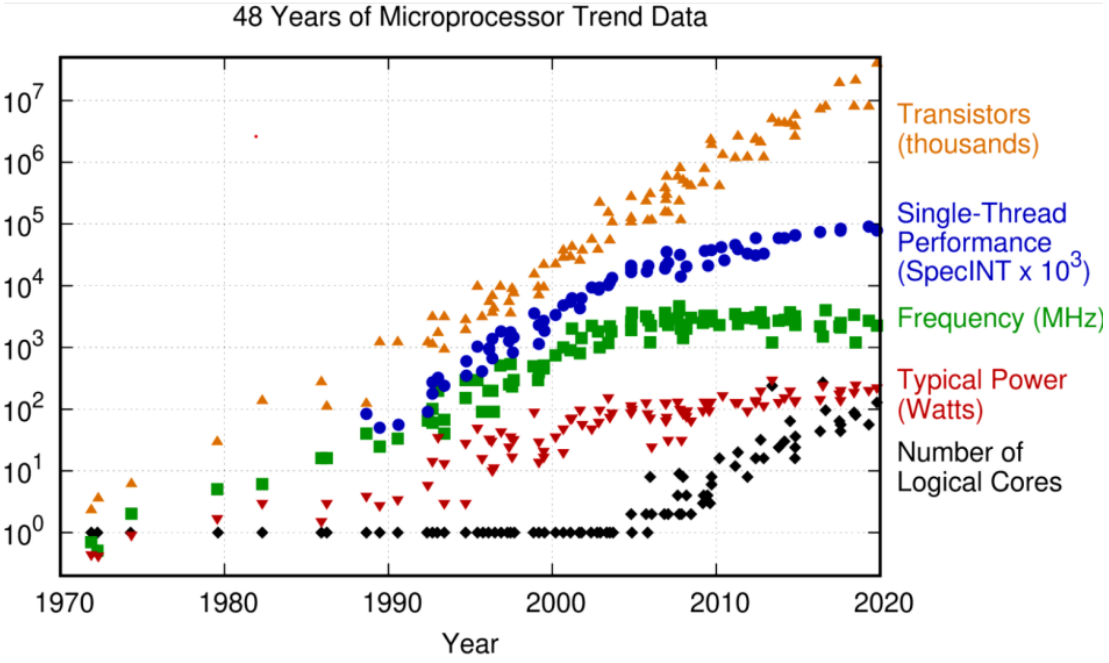
Minix (mini unix) era un sistema operativo di circa 16k linee di codice in C e dentro c'era tutto, era fatto da un solo pezzo che faceva il microkernel, scritto in parte in assembler e sopra aveva due processi soli, il gestore della memoria e il gestore del file system.

I sistemi operativi hanno bisogno di avere una base di applicazioni che ci girano sopra, ci servono tante applicazioni e abbiamo bisogno di supportare tanto hardware, ci serve dunque sia l'interfaccia hardware che software.

Storia dei S.O



Trend dei Microprocessori



Primi sistemi operativi

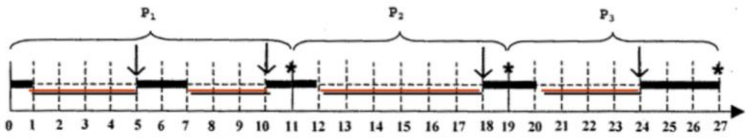
Venivano eseguite un'applicazione alla volta, l'applicazione aveva il pieno controllo del software e il sistema operativo era una libreria a Runtime.

Con i sistemi batch mantenevamo la CPU occupata avendo una coda di processi, il So caricava i vari processi mentre uno di questi era ancora in esecuzione, l'utente sottometteva questi processi/lavori e aspettava che fossero completati.

Sistemi a single task

Si aveva un'esecuzione sequenziale.

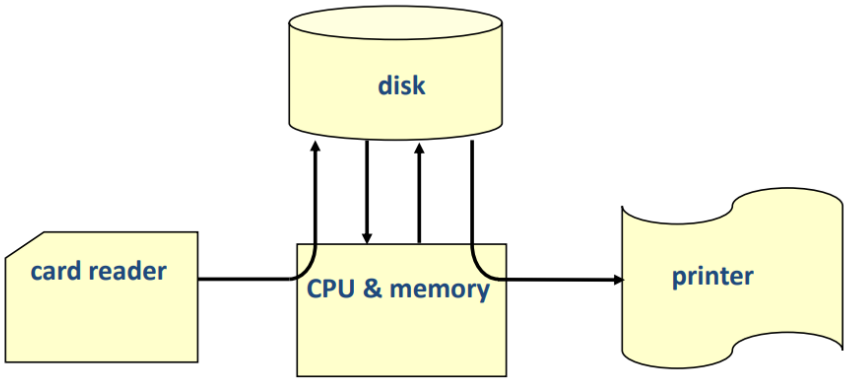
La linea nera nell'immagine rappresenta l'esecuzione del processo, quella rossa le operazioni di I/O.



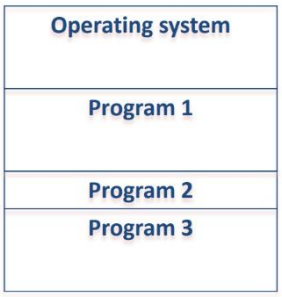
Sistemi batch

Quando dovevamo fare l'I/O siccome era molto lento, l'I/O si faceva su disco, e le operazioni del programma riferivano il file su disco.

Con lo Spooling i dispositivi veri leggevano e scrivevano sul disco, e lo stesso il processore, anche qua si aveva un programma alla volta.



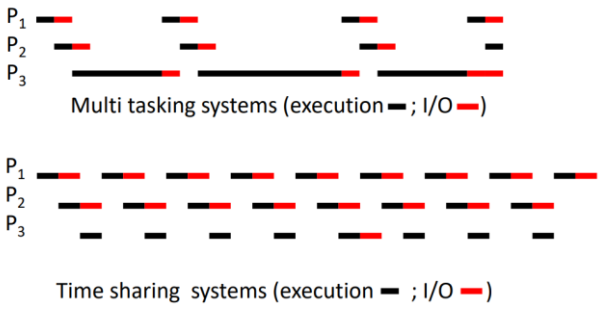
Sistemi batch multi-programma



Stessa struttura dei sistemi batch ma potevano runnare più programmi contemporaneamente, non si aveva interazione, fornivamo i dati in input e il S.O forniva i risultati.

Introduzione del time-sharing

Si introdusse questa tecnica per permettere agli utenti di poter interagire con la programmazione, c'era la possibilità di avere più utenti che interagivano con il terminale sullo stesso computer e il time sharing significa che oltre a fare i task classici avevamo una parte di cpu che era assegnata ad un utente e poi ad altri.



L’astrazione del kernel

Cerchiamo di capire qual è il ruolo del sistema operativo e qual è l’impatto architetturale che i moderni sistemi operativi utilizzano per fornire tutti i vari servizi, uno degli aspetti fondamentali è la protezione.

Dato che oltre al sistema operativo abbiamo altri programmi dobbiamo garantire che questi programmi possano coesistere l’uno con l’altro senza scaturire problemi.

Un aspetto è considerare il fatto che ci possano essere dei programmi volutamente o meno scorretti che possano creare problemi, ad esempio che tentino di scrivere il S.O cercando di modificarlo.

Alcuni esempi sono:

- Script web
- Programmi scaricati
- Programma scritti ma non ancora testati

Un modo potrebbe essere quello di emulare ciascuna istruzione con privilegi limitati, se non genera eccezioni procedo con l’esecuzione vera e propria. Questo è fattibile ma è un procedimento molto lento. Se vogliamo soddisfare almeno uno dei 3 criteri, ovvero quello delle prestazioni, questo metodo non è utilizzabile.

Punti Principali

Per garantire la protezione è stato introdotto il concetto di processo, un processo è un’astrazione del sistema operativo per eseguire programmi, ho un programma che ho scritto e voglio lanciare, il sistema operativo lo esegue sotto forma di astrazione ovvero di processo, questo processo sarà eseguito con privilegi minori rispetto a quelli del S.O stesso.

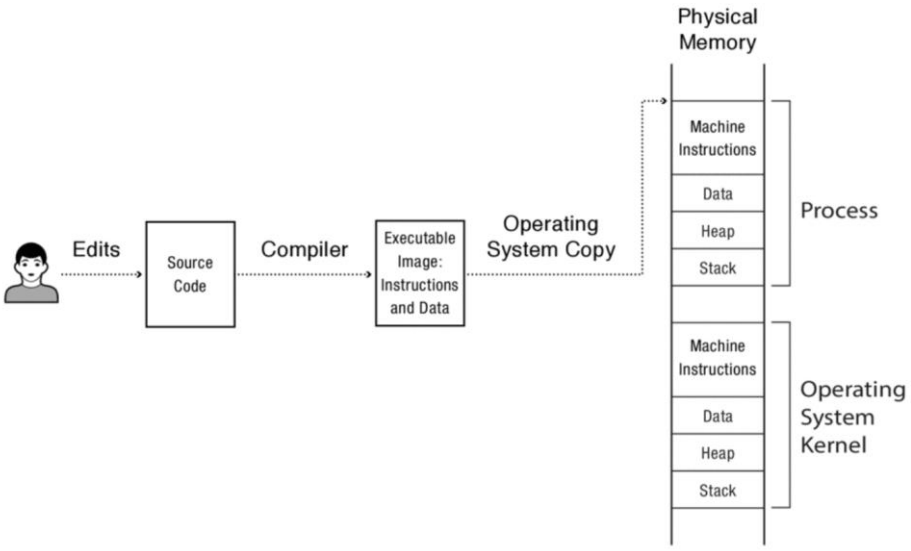
Distinguiamo almeno due modi di operatività:

- Kernel-mode
- User-mode

Queste modalità differiscono per i privilegi di esecuzione, nella kernel mode il codice viene eseguito sulla cpu con pieni privilegi, nella modalità user-mode i processi vengono eseguiti sempre sulla cpu ma con privilegi ridotti.

Processi

Un processo è la rappresentazione in memoria di un programma, il programma è la parte statica. L’eseguibile generato dalla compilazione verrà caricato in una zona di memoria diversa dal S.O, quest’ultimo avrà delle strutture dati tali da poter eseguire le istruzioni del processo. Il processo è caratterizzato da una parte di codice (che sono le istruzioni che eseguirò), una parte di dati statica (es variabili globali), i dati dinamici (che noi chiediamo al sistema operativo) ovvero l’heap e una parte chiamata stack. Quando parleremo di memoria virtuale vedremo che queste parti potranno essere segmentate e sparse in posizioni diverse della memoria.



Un processo all'interno del SO viene rappresentato con una struttura dati che si chiama PCB (Process Control Block).

Il PCB è l'entry di una tabella che si chiama Process Table possiamo immaginare una tabella che ha tante entry per quanti sono i processi.

Ogni processo ha almeno due elementi che lo caratterizzano, uno è il suo flusso di controllo. I flussi di controllo possono però essere anche molteplici, in questo caso si parla di thread.

L'altro aspetto è il suo address space ovvero il suo spazio di indirizzamento che è l'insieme delle parole di memoria che quel processo può riferire. Ovvero le parole di memoria a cui il processo ha diritto di accedere.

Programmi e processi

Un programma è la lista di istruzioni che sono state scritte in un certo linguaggio ed è una struttura statica.

Un processo rispetto ad un programma è un concetto dinamico, ha una sequenza di attività/flussi di controllo che è descritta dal programma statico ma che viene rappresentata all'interno del SO. Il processo è la rappresentazione in memoria del programma, le istruzioni del codice del processo sono eseguite su una o più CPU.

Un processo senza specificare nessun'altra caratteristica è un processo utente.

Process Control Block

All'interno del process control block troviamo un identificatore ovvero un Process ID (PID), tipicamente è l'indice del record all'interno della tabella dei processi che viene assegnato dal kernel.

Abbiamo poi lo stato di esecuzione del processo che è calcolato a seconda di quali istruzioni ha fatto/sta facendo, potrebbe essere ad esempio un processo appena creato.

Si parla di PID e PPID ovvero di parent PID, infatti per ogni processo c'è un processo padre che lo ha creato.

Il processo *init* è il padre di tutti gli altri processi, ogni processo a parte init avrà un parent.

Un processo può essere in stato di attesa, stoppato, in esecuzione ecct...

Un processo contiene anche un insieme di registri, i processi si alternano nell'esecuzione sulla cpu, questo vuol dire che dopo un po' il processo verrà sospeso, salverà i suoi registri dentro il pcb e ripristinerà il contenuto dei registri della pcb nella cpu.

Questi registri definiscono lo stato della cpu per definire il processo.

Se il processo ha più thread nel pcb ci saranno i riferimenti ai thread control block.

Il PCB contiene anche la memoria assegnata a quel processo, in particolare quali limiti.

Vi saranno anche altre informazioni quali: i file aperti, i dispositivi di I/O utilizzati, lo stato dei dispositivi a cui ha acceduto ecct…

Supporto Hardware

Dobbiamo distinguere due modalità:

Kernel Mode:

L'esecuzione avviene con pieni privilegi sull'hardware, ad esempio ci possiamo interfacciare con dispositivi di I/O.

Un processo in kernel mode se gira con diritti kernel può leggere e scrivere la memoria di tutti gli altri processi.

User Mode:

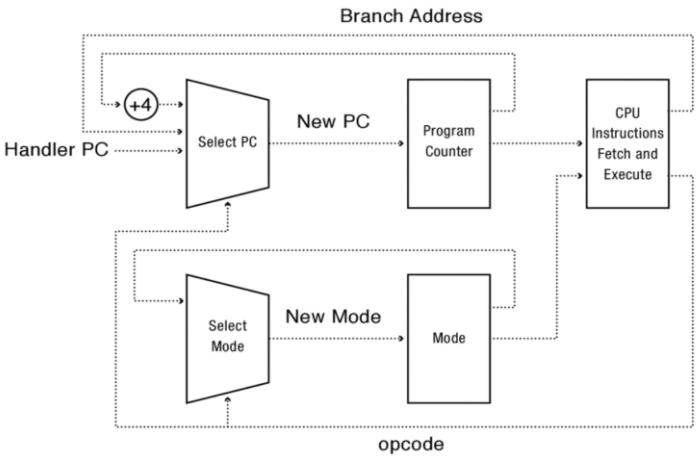
I diritti sono limitati, i diritti di un processo vengono assegnati dal sistema operativo.

Di default l'utente ha diritto di operare sul suo spazio di indirizzamento ma non ha alcun diritto di modificare la memoria di altri processi, tantomeno quella del S.O, inoltre non può interfacciarsi con i device.

Il passaggio tra queste due modalità viene fatto modificando quella che è la program status word, in x86 è il registro EFLAGS mentre in arm è il registro CPSR.

Cambiando un bit in questi due registri switchamo dinamicamente tra kernel mode e user mode, ovviamente questo switch può essere fatto solo dal sistema operativo, lo switch avviene a livello architetturale.

L'hardware mi deve permettere di avere quindi queste due modalità di esecuzione, i vecchi processori non la avevano e lo stesso i primi S.O.



L’hardware per implementare le funzionalità del S.O fornisce la capacità di distinguere le istruzioni in privilegiate e non, quelle privilegiate permettono di guadagnare la modalità di esecuzione kernel e non possono essere invocate in modalità user mode.

La possibilità di accedere a tutta la memoria è garantita solo in kernel mode.

La possibilità di settare il timer è un aspetto molto importante e si può fare solo in kernel mode, il timer è un tempo che una volta scaduto genera un’interruzione.

Istruzioni privilegiate

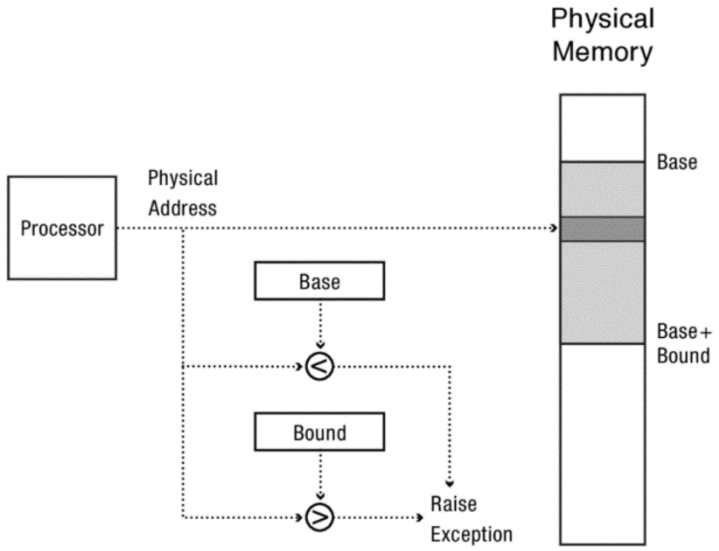
Cosa dovrebbe accadere se il programma in modalità utente tenta di eseguire un’operazione proibita?

Vogliamo che venga generata un’eccezione e che il SO se ne occupi.

Protezione della memoria

Un modo semplice per gestire questa protezione lavorando con indirizzi fisici è quella di richiedere all’hardware di darci la possibilità di avere due indirizzi particolari, *base* che è l’inizio del nostro spazio di indirizzamento e *bound* che è la dimensione massima, nel pcb mettiamo base e bound. Per ogni indirizzo che tentiamo di sfruttare faremo il controllo se rientra tra base e base+bound. È molto semplice e veloce da implementare, ogni volta che mandiamo in esecuzione un nuovo processo carichiamo base e bound.

Lanciamo un’eccezione quando la condizione è violata, l’eccezione è un evento che manda in esecuzione il SO.



Questo modo di operare è eccessivamente rigido perché vorremo avere una gestione della memoria dinamica, ovvero potremmo voler allocare la memoria che ci serve.

Un altro aspetto dinamico non è solo l’heap ma anche uno stack che contiene i record delle funzioni e che anche lui cresce e diminuisce dinamicamente.

Con il base e bound potrei gestire anche le operazioni dinamiche ma rischio di avere dei costi di gestione troppo alti o rischio di sprecare memoria.

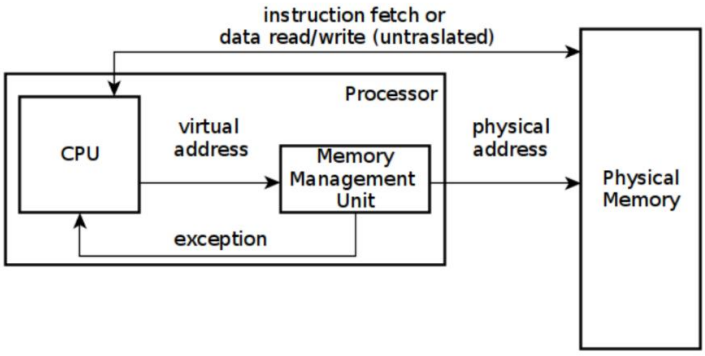
Lavorare con indirizzi fisici è troppo costoso (in termini di flessibilità).

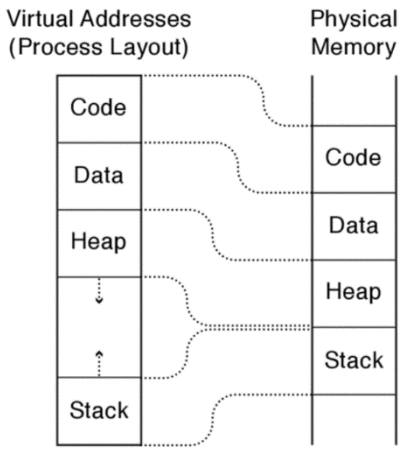
C’è inoltre un problema di frammentazione, ovvero il problema di avere la memoria con dei buchi, ovvero tanti blocchi di piccole dimensioni che non posso sfruttare.

Indirizzi virtuali

Vogliamo utilizzare indirizzi logici ottenuti tramite una funzione di mapping.

La cpu caricherà istruzioni che riferiscono indirizzi logici, (per lei tutto è indirizzo logico), gli indirizzi verranno poi dati all'mmu che si occuperà di capire effettivamente qual è l'indirizzo fisico, se questo indirizzo logico è un indirizzo che non mappa in uno spazio fisico allocato dal sistema operativo per quel processo è proprio l'mmu a lanciare l'eccezione.





Con la memoria virtuale il fatto di avere un dispositivo hardware è fondamentale, se non ci fosse, per ogni indirizzo dovrei fare una traduzione software e sarebbe troppo costosa.

Il fatto che ci sia un dispositivo hardware in grado di fare la traduzione è un requisito che il SO chiede all'architettura sottostante.

Con gli indirizzi logici abbiamo un programma eseguibile che è quindi una sequenza di indirizzi logici.

Abbiamo un'organizzazione come quella in figura, nella memoria fisica dove possono essere tutti attaccati. Si ha un totale disaccoppiamento tra spazio logico e spazio fisico.

Esempio:

• What if we run two instances of this program at the same time?

```
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 10 seconds
        printf ("static address: %p, value: %d\n", &staticVar, staticVar);
        printf ("local address: %p, value: %d\n", &localVar, localVar);
}
```

Produces (*):

```
        static address: 0x100407000, value: 1
        local address: 0xffffcc3c, value: 1
```

(*) Because of the Address Space Layout Randomization (ASLR) the output can be different at each run…….

Lanciare i programmi sempre allo stesso indirizzo logico crea dei problemi di sicurezza, abbiamo quindi l'ASLR (Address Space Layout Randomization) che aggiunge una costante random ad ogni indirizzo, disabilitando questa opzione ogni volta che lanciamo il programma in figura otterremo quell'indirizzo sempre, perché sono indirizzi logici e non c'entrano niente gli indirizzi fisici reali in cui si trovano.

Timer Hardware

Devo avere qualcosa che dopo tot tempo faccia scattare un'interruzione e mandi in esecuzione il SO, se non avessi un sistema del genere potrebbero esserci dei processi che acquisiscono la cpu senza più rilasciarla.

Abbiamo necessariamente qualcosa che dopo un po' manda un evento per stoppare (magari solo temporaneamente) il processo corrente.

Il fatto di avere un timer hardware che ad una certa frequenza genera un'interruzione che permette al kernel di tornare in possesso della cpu può essere visto come un ulteriore sistema di sicurezza.

Il tempo viene settato dal sistema operativo.

Il kernel può decidere di non ascoltare le interruzioni, con delle istruzioni privilegiate può sospendere le interruzioni, questo ci permette di gestire situazioni particolari.

Le interruzioni sono uno dei meccanismi minimali per garantire la mutua esclusione.

Mode switch

Non c'è modo che l'utente disattivi l'interruzione in questa modalità.

Possiamo passare da user mode in kernel mode grazie a:

- Qualunque interruzione.
- Quando lanciamo un'eccezione.

System call o interruzioni software, sono interruzioni particolari che anziché essere generate da dispositivi sono generate da programmi software.

Le system call sono un numero per quanto alto, molto limitato e sono tutte codificate e implementate in modo particolare perché sono l'entry point per passare da una modalità all'altra.

Quando passiamo da kernel mode a user mode?

Quando ad esempio ritorniamo da un'eccezione oppure quando creiamo un nuovo processo o startiamo un nuovo thread.

Possiamo passare su un context switch ovvero abbiamo fatto un cambio tra processi, il context switch è ad opera del kernel.

Upcall

User-level upcall, le upcall sono un meccanismo all'opposto del system call, ci permettono infatti di transire da kernel a user mode, il sistema operativo dice al processo utente di riprendere l'esecuzione, non è un vero e proprio servizio ma sono delle chiamate che vanno nel verso opposto. Sarebbero l'equivalente dei segnali che abbiamo nel mondo unix.

Alla base di questa transizione tra user mode e kernel mode c'è il concetto di interruzione, le interruzioni che sono generate dai dispositivi e le eccezioni che sono lanciate da eventi

sincroni dovute ad istruzioni non legittime sono interpretabili come una sorta di interruzioni, lo stesso sono le system call, le abbiamo infatti chiamate system software, queste interruzioni ci permettono dunque di transire dallo stato user allo stato kernel.

Ora ci poniamo il problema di dire come facciamo a gestire le interruzioni in modo safe essendo un meccanismo molto importante?

Non dobbiamo poter transire da user a kernel se non ne abbiamo il diritto.

Gestione delle interruzioni

Il meccanismo è apparentemente molto semplice, ha però molti dettagli implementativi da tenere in considerazione.

Quando si verifica un'interruzione?

Supponiamo ci sia un processo A che chiede un certo dato ad un dispositivo I/O attraverso una system call, una read o una write, questo processo che ha eseguito la system call viene temporaneamente sospeso in attesa che l'operazione di I/O sia completata e magari nel frattempo viene attivato un altro processo, quando arriva l'interruzione dal dispositivo, arriva un segnale e il SO se ne deve accorgere prontamente, capendo che arriva da un particolare dispositivo a cui aveva precedentemente fatto la richiesta e deve riportare in esecuzione il processo A in modo che questo completi l'operazione, viene quindi interrotta l'esecuzione del processo B.

Questa interruzione necessita di un po' di cose:

- Vettore di interruzioni:

  Lo possiamo immaginare come un vettore di puntatori a funzioni, abbiamo delle funzioni, una per ogni interruzione/handler che abbiamo e che si occuperanno di gestire una particolare interruzione, il vettore ha tante entry per quante sono le interruzioni che il sistema gestisce.

- Stack dedicato:

  Per l'esecuzione dell'handler delle interruzioni, questa funzione che viene chiamata a seguito dell'evento interruzione non usa lo stack del processo attualmente in esecuzione ma ha un suo stack dedicato in spazio kernel.

- Interrupt Masking:

  Un altro meccanismo per la gestione delle interruzioni è la possibilità di mascherarle, questo meccanismo ci permette di ignorarle tutte o solo una parte. È necessario mascherare alcune interruzioni perché quando eseguiamo l'handler magari eseguiamo alcune operazioni molto delicate come ad esempio il salvataggio dei registri nel pcb o sullo stack e voglio che questa operazione non venga interrotta, senza dover gestire un'eventuale altra interruzione.

L’handler viene sempre eseguito a interruzioni disabilitate è infatti lui stesso una volta terminato ad abilitarle.

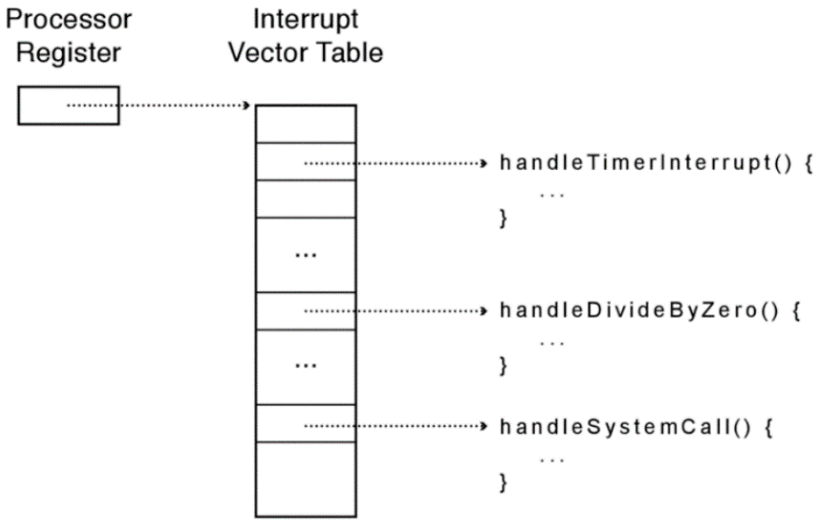
- Atomic transfer control:

  Una cosa molto importante è quello che si chiama atomic transfer control, ovvero la possibilità di salvare alcuni registri e di fare delle operazioni in modo atomico, ovvero senza che possano essere interrotte.
  Alcune di queste operazioni sono ad esempio il salvataggio del program counter il salvataggio dello stack pointer e il salvataggio della program status word, almeno il salvataggio di questi 3 registri unito allo switch tra kernel mode e user mode devono avvenire come se fossero un’operazione sola, cioè quando arriva l’interruzione questa sequenza viene fatta sempre ed in modo indivisibile.

- Transparent restartable execution:

  Un altro meccanismo legato alla gestione delle interruzioni è che per il programma in esecuzione il fatto che sia arrivata un’interruzione e che la stiamo gestendo è totalmente invisibile, ovvero il programma non se ne accorge, che sia una system call, un timer ecct… Alla fine dell’esecuzione del programma sulla cpu è come se il programma fosse stato eseguito senza mai interrompersi, la procedura garantisce che tutto il meccanismo della gestione delle interruzioni sia totalmente trasparente all’esecuzione del programma.

Vettore di interruzioni



Logicamente è un vettore configurato al boot del sistema operativo dove ogni entry contiene un po' di informazioni e un handler ovvero un puntatore logico.

Abbiamo il program counter che punta alla prima istruzione dell’handler del timer.

Questo viene chiamato ogni volta che il timer invia l’interruzione, interruzioni diverse ovviamente hanno handler diversi perché le azioni che bisogna fare saranno diverse, dunque dipendenti dal dispositivo o dall’interruzione ricevuta.

Ad esempio per le system call non essendo legate ad alcun dispositivo si dovranno fare delle azioni diverse dall’esecuzione del timer.

Gli handler sono quindi funzioni software che eseguono azioni diverse.

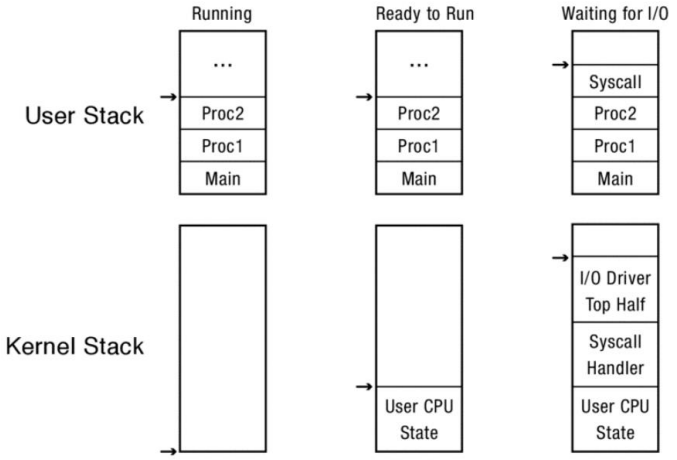
Il program counter è il puntatore alla funzione, ci dice dove fare il salto per andare ad eseguire quella data istruzione, anche l’handler al termine, quando finisce dovrà avere una sorta di return, che in gergo si chiama I_Ret che sta per “instruction return”, o “interrupt return” e permette di fare l’operazione inversa, ovvero permette atomicamente di

ripristinare l'esecuzione di un altro processo, il punto di ingresso e il punto di uscita di un handler richiedono particolare attenzione, questa parte dell'handler è eseguita in kernel mode esisterà quindi un particolare valore della parola di stato.
Ci possiamo immaginare che ogni entry di questo vettore contenga almeno il valore da assegnare al program counter, allo stack pointer del kernel e il valore da assegnare alla program status word.

Perché non possiamo utilizzare la porzione di stack del processo che avevamo in esecuzione fino a quel momento?

Non è possibile farlo per ragioni di sicurezza, immaginiamo un'architettura multi core dove su core diversi ci possono essere flussi di controllo di esecuzione diversi, relativi allo stesso processo (ovvero thread diversi), i flussi di controllo (thread) dello stesso processo vedono l'unico spazio di indirizzamento che è quello del processo, se su un core ricevessi un'interruzione e per la sua gestione utilizzassi lo stack del processo attualmente in esecuzione, se sull'altro core c'è in esecuzione un thread dello stesso processo questo thread potrebbe accedere senza problemi allo stack mentre il kernel sta gestendo l'interruzione e questo non è safe.
Inoltre se ci fosse qualche problema legato al corrompere lo stack da parte dello stack utente potrei danneggiare anche il SO.

Per ogni processo in esecuzione c'è un ulteriore processo in kernel space per la gestione dell'handler, esistono quindi lo user stack ma esiste anche almeno logicamente un kernel stack per lo stesso processo.



L'handler si divide logicamente in due parti, una chiamata top e una bottom, top half e bottom half, di solito la top half è quella che viene eseguita non appena l'handler viene chiamato ed è quella che viene eseguita ad interruzioni disabilitate, la parte bottom half è quella meno delicata che viene tipicamente eseguita ad interruzioni abilitate in alcuni casi è possibile trovarle scritte con termini al contrario, è però solo terminologia.

Mentre è in esecuzione l'handler, il kernel stack è utilizzato e lo user stack è freezato.

Il ready to run è uno stato intermedio e significa che il processo è stato riattivato perché l'interruzione è arrivata, è andato in esecuzione un qualche handler che ha finito di fare la gestione, ha preso il processo wating e lo ha messo in uno stato pronto per eseguire, ovvero si è finito di fare la syscall, dobbiamo solo ripristinare i registri di stato che aveva, che al momento si trovano nel kernel stack, quando andrà in esecuzione questi registri di stato verranno ripristinati sui registri fisici, ovvero verrà ripristinato il program counter e riprenderà l'esecuzione.

Logicamente è uno stack per ogni processo, implementativamente potrebbe essere diverso.

Mascheramento delle interruzioni

Quando arriva l'interruzione l'handler viene gestito sempre ad interruzioni disabilitate, almeno la sua top half, in generale l'handler o comunque qualunque processo giri in kernel mode può "mascherare" o disabilitare alcune interruzioni a suo piacimento, questo si può fare solo in kernel mode, ed è molto importante che ci sia la possibilità di farlo, è un primissimo meccanismo per garantire l'atomicità nell'esecuzione di alcune istruzioni, sarebbe infatti ad esempio critico se mentre sto salvando dei registri dovesse arrivare un'interruzione e dovessi risalvare di nuovo tutti i registri di stato.

In generale la disabilitazione delle interruzioni viene fatta dal kernel quando un'interruzione è ricevuta per procedere a gestirla, la prima parte dell'handler non deve disabilitare le interruzioni in quanto sono già disabilitate.

L'handler potrebbe però riabilitarle e disabilitarle nuovamente, sicuramente quando ritorniamo dall'handler, le interruzioni se erano state disabilitate verranno riabilitate atomicamente dal sistema perché la parte user gira sempre con le interruzioni abilitate.

L'interrupt handler svolge un numero di istruzioni limitato perché non vogliamo tenere le interruzioni disabilitate per troppo tempo, non vorrei ad esempio rischiare di sovrascrivere dati in arrivo da dispositivi di I/O particolarmente veloci, l'handler fa quindi il minimo necessario per permettere al dispositivo di gestire le prossime interruzioni, ogni operazione di wait, di attesa attiva ad interruzioni disabilitate deve essere estremamente limitata a poche istruzioni.

C'è però una parte che viene eseguita in generale da quella che chiamiamo bottom half dell'interrupt handler, che può essere un'altra funzione o addirittura un altro processo a cui demandiamo la responsabilità di svolgere alcuni compiti.

Ci sarà sicuramente una parte software del device driver che verrà eseguita ad interruzioni abilitate in kernel mode e che farà anche operazioni di attesa lunghe, ma essendo le interruzioni abilitate verranno gestite e non ignorate dal sistema.

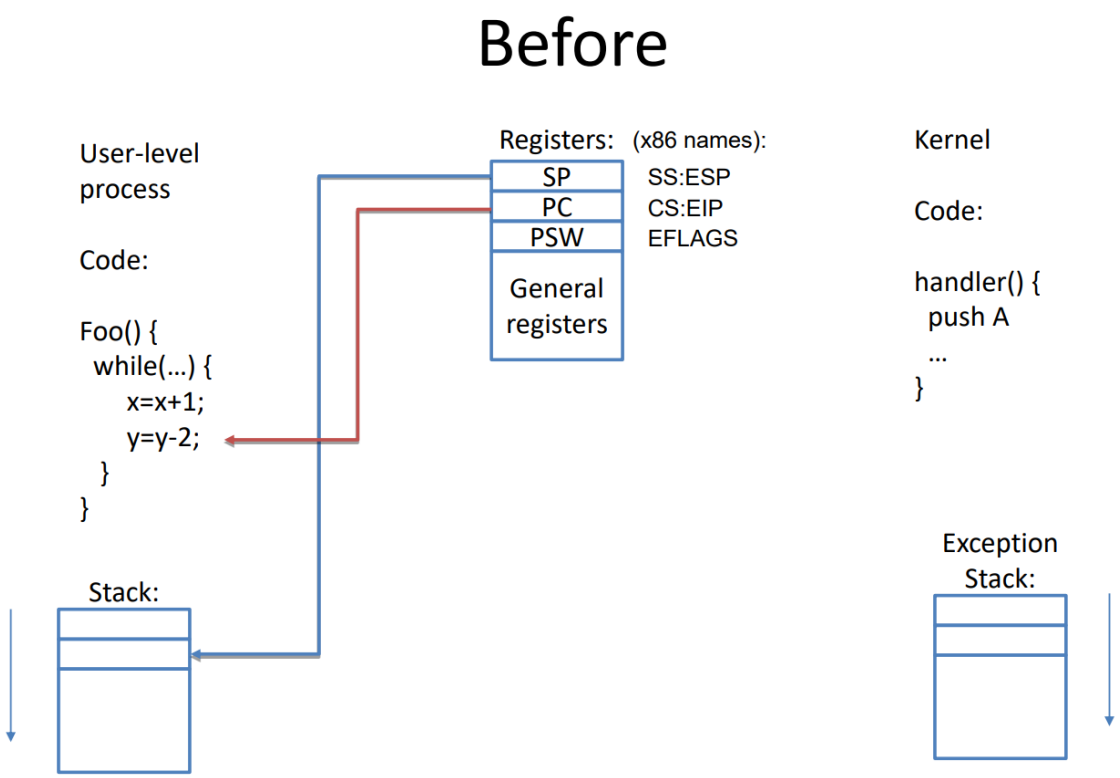
Operazioni atomiche

- Salvare lo stack pointer
- Salvare il Program Counter
- Salvare la Program Status word
- Dobbiamo far puntare lo stack pointer corrente a quello del kernel
- Dobbiamo settare il pc e il psw sullo stack
- Switch tra user mode e kernel mode
- Viene messa in esecuzione il pc e il psw dell'handler sul registro pc e sul psw della cpu

Tutte queste operazioni vengono eseguite in modo atomico come un'unica operazione.

L'hardware con questa operazione atomica ha salvato solo i registri più importanti, tutti gli altri registri non sono stati salvati, ma questo non significa che non debbano esserlo, devono essere infatti salvati anche loro ma la responsabilità è dell'handler vero e proprio, se devono essere salvati sarà suo compito sapere quanti e quali registri deve salvare.

Ctrl+click sull'immagine per vedere esempio



Architettura concreta

Consideriamo inizialmente un'architettura fittizia, con 16 bit, pochi registri e 3 registri di controllo, pc (program counter) sp (stack pointer) e ps (program status), supponiamo che in questa architettura vi sia un unico kernel stack per tutti i processi, che le interruzioni vengano controllate alla fine del ciclo fetch-execute e che ci sia un'operazione particolare IRET che permette di ritornare dall'interrupt handler.

Initial state: interrupt '500' occurs when executing instruction A000

Registri nella CPU

PC	A000
PS	PSW P
SP	FFFC
R1	AAAA
R2	BBBB
…	

Memoria

Programma P

…	…
A000	istr. 1
A004	istr. 2
A008	istr. 3
A00C	istr. 4
A010	istr. 5
…	

Interrupt Handler

100	Store
104	Save
108	Registers
…	
200	Restores
204	Registers
208	IRET

Stack di P

…	…
FFEC	
FFF0	
FFF4	
FFF8	
FFFC	…

Stack del kernel

2FF8	
2FFC	
3000	
3004	
3008	
300C	

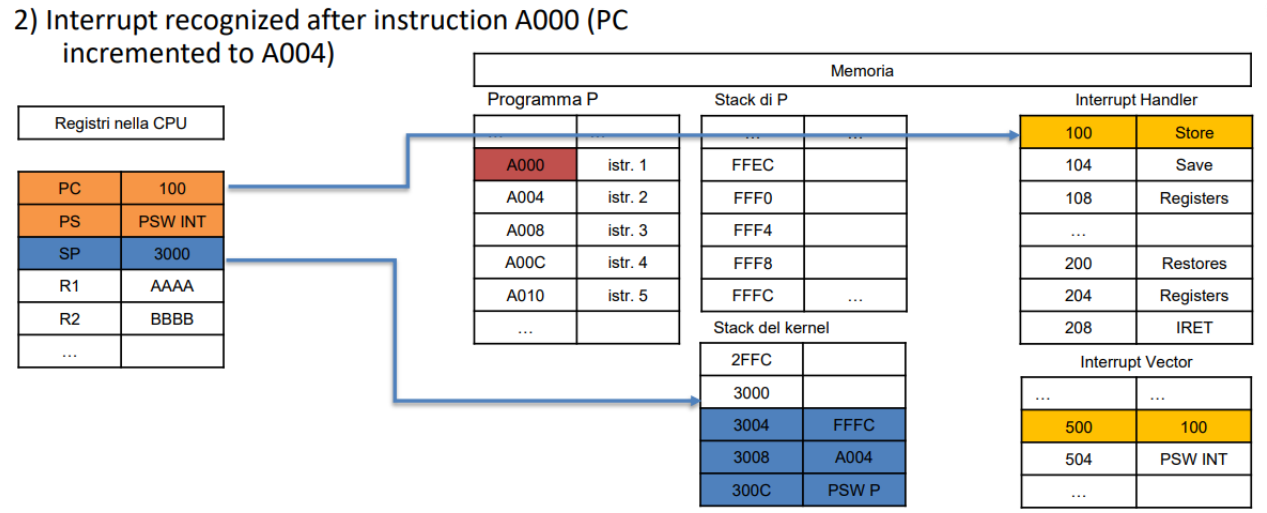
Interrupt Vector

…	…
500	100
504	PSW INT
…	

Lo stato è quello subito dopo l'esecuzione di A000, nel pc c'è l'istruzione corrente, andiamo a controllare le interruzioni (nell'interrupt vector) e vediamo che c'è l'interruzione 500, a sinistra abbiamo i registri principali e pochi registri generali, nella parte destra abbiamo invece le varie parti specifiche, come la sequenza di istruzioni del generico

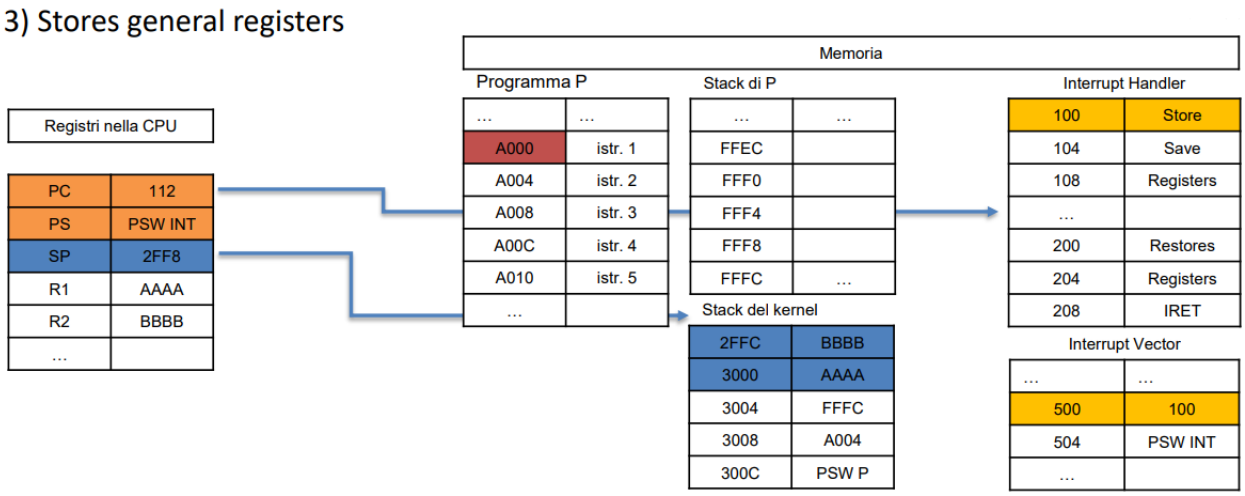
programma con i propri indirizzi, c'è poi lo stack del processo, lo stack del kernel che è vuoto perché stavamo operando in spazio utente, l'interrupt handler e l'interrupt vector.

Nell'interrupt vector per ogni interruzione abbiamo una entry che codifica dove si trova l'interrupt handler corrispondente.



In questa parte abbiamo quello che succede appena viene gestita l'interruzione.

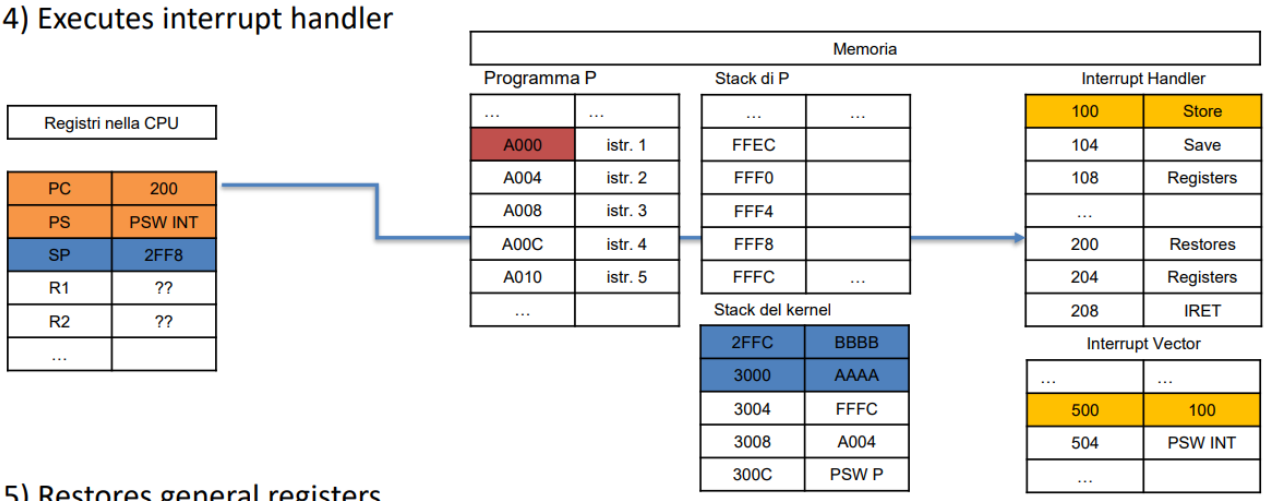
Ad hardware ed in modo atomico vengono salvati i tre registri di stato più importanti pc ps e sp, contestualmente oltre a caricare questi 3 indirizzi sullo stack del kernel, carichiamo nel pc l'indirizzo iniziale dell'interrupt handler (quello trovato nell'interrupt vector).



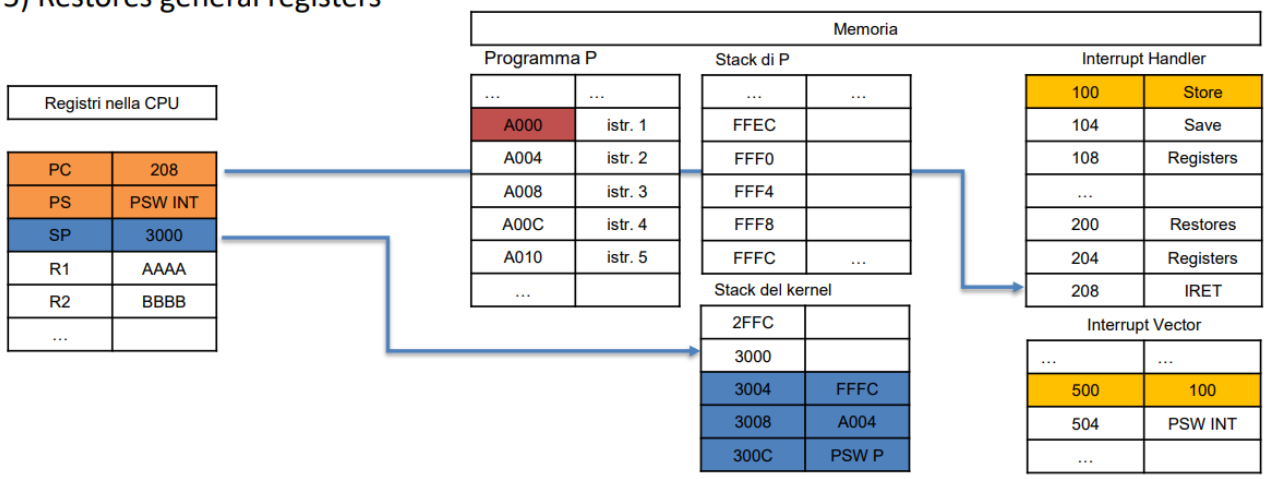
Dopodiché il pc inizierà l'esecuzione dell'handler che si preoccuperà di salvare i registri generali, l'handler viene gestito ad interruzioni disabilitate quindi se anche arrivasse una nuova interruzione non verremmo interrotti, abbiamo tempo per fare il salvataggio dei registri.

Una volta fatto questo le interruzioni vengono riabilitate.

4) Executes interrupt handler

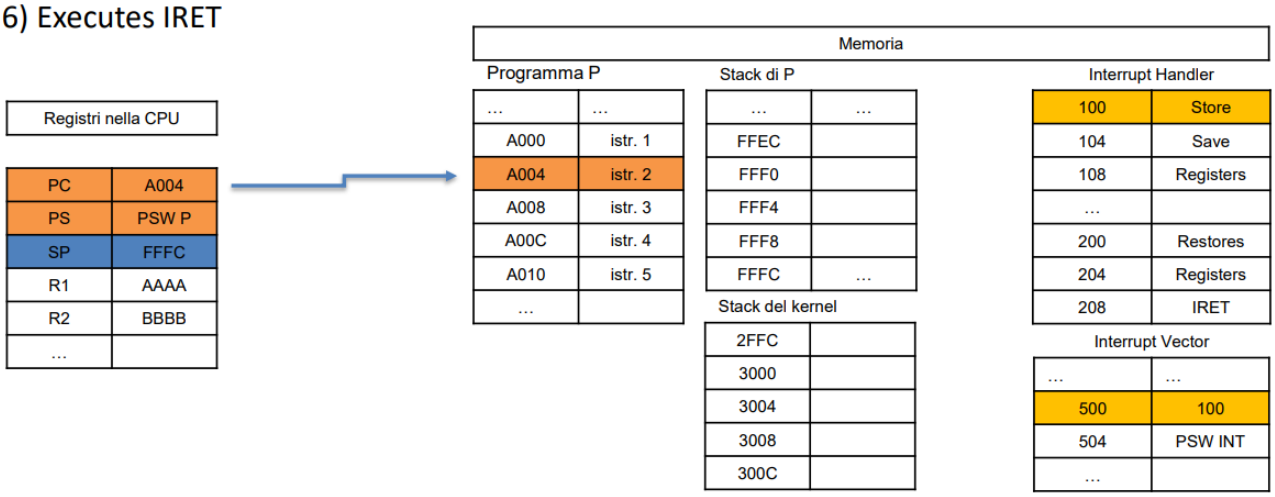


5) Restores general registers



L'handler userà quindi i valori nei registri per gestire l'interruzione e dopo un po' quando ha finito di gestire l'interruzione dovrà ripristinare il processo, disabilitando nuovamente le interruzioni, ovvero ripristinare i registri e fare la IRET.

6) Executes IRET



A questo punto l'istruzione IRET ci permette di ripristinare in modo atomico i 3 registri di stato dallo stack del kernel ai registri della cpu, switchiamo quindi verso la modalità utente cambiamo la parola di stato.

Gestione delle interruzioni in ARM

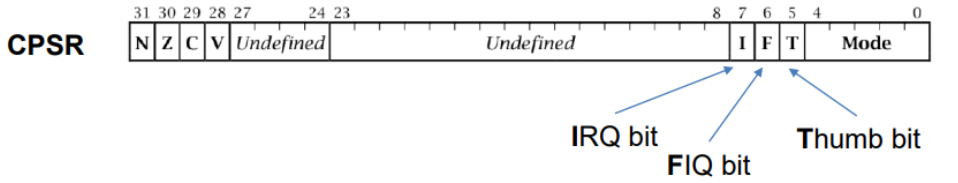
Ci servono almeno due modalità, supervisor o kernel mode e quello utente, l'arm in realtà ha 6 modalità di esecuzione. Il registro psw si chiama cpsr, esistono due categorie di modalità operative, pl0 (non abbiamo diritti) e pl1 (abbiamo diritti), in 5 delle 6 modalità abbiamo la possibilità di accedere a tutte le risorse del sistema e possiamo accedere a dei registri aggiuntivi, il salvataggio dei registri non avviene in kernel stack ma in dei registri aggiuntivi normalmente non visibili in modalità utente.

Modalità:

Mode	$CPSR_{4:0}$
User	10000
Supervisor	10011
Abort	10111
Undefined	11011
Interrupt (IRQ)	10010
Fast Interrupt (FIQ)	10001

Ci sono due modalità per gestire le interruzioni, le interruzioni lunghe che richiedono content switch (ovvero la sospensione del processo) e le interruzioni che non richiedono la sua sospensione. Queste sono le fast interrupt e permettono di avere una gestione molto più veloce.

Il cpsr codifica questi modi di interruzioni in 5 bit da 0:4, seguono poi 3 bit che definiscono la modalità thumb, se le fast interrupt sono abilitate o se le interrupt standard sono abilitate.



In arm si parla al posto di interrupt vector, di exception vector

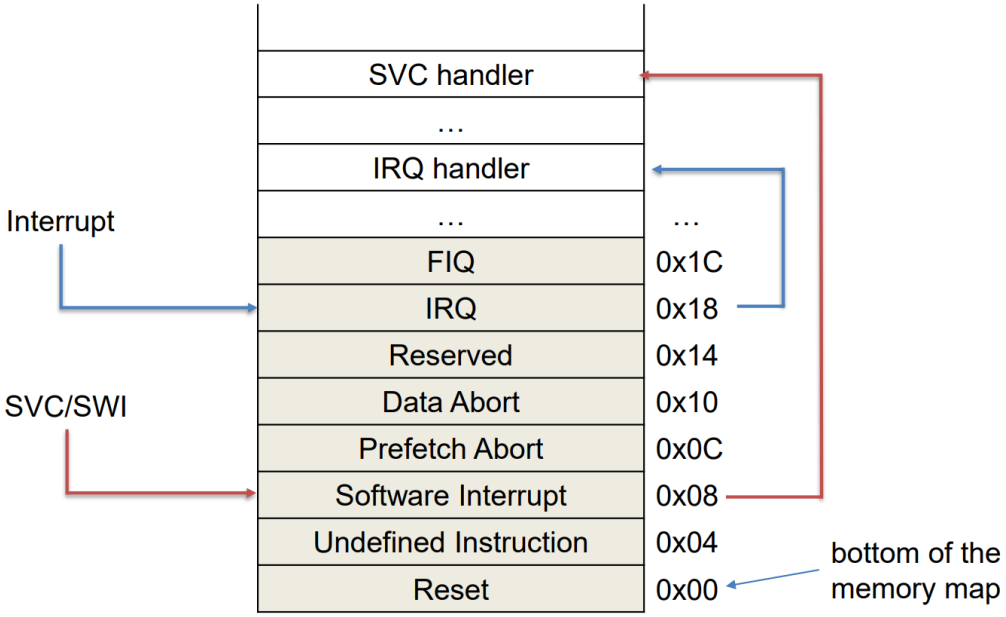
Exception	Address	Mode
Reset	0x00	Supervisor
Undefined Instruction	0x04	Undefined
Supervisor Call	0x08	Supervisor
Prefetch Abort (instruction fetch error)	0x0C	Abort
Data Abort (data load or store error)	0x10	Abort
Reserved	0x14	N/A
Interrupt	0x18	IRQ
Fast Interrupt	0x1C	FIQ

L'exception vector ha indirizzi di dove comincia l'handler di varie interruzioni, esiste una priorità tra le varie modalità, se arrivano contemporaneamente due interruzioni viene selezionata una delle due sulla base della priorità, più basso è il numero più è alta la priorità.

Priority (1 high, 6 low):

- 1 Reset
- 2 Data Abort
- 3 FIQ
- 4 IRQ
- 5 Prefetch Abort
- 6 Undefined Instruction and Software Interrupt (SWI)

Per ogni entry abbiamo un'istruzione che è un salto o un caricamento del program counter in modo tale da poter saltare in zone dove ci sono effettivamente i codici dell'handler delle system call.



Registri

Finora i registri che abbiamo usato erano i registri da r0 a r15 dove r13 era lo sp, l'r14 il link register r15 pc e poi la psw (cpsr).

Abbiamo accesso al cpsr solo per leggere i flag, non possiamo modificare niente.

Le altre 5 modalità hanno dei registri in più, questi sono dei registri aggiuntivi visibili solo nella modalità specificata, ad esempio nella modalità fast interrupt ci sono 8 registri aggiuntivi, in realtà aggiuntivi non è il termine giusto in quanto i registri da R8-R14 non sono più visibili ma vengono abilitati dei corrispondenti registri con cui lavorare così da non dover salvare i dati.

I registri aggiuntivi cambiano a seconda della modalità in cui siamo.

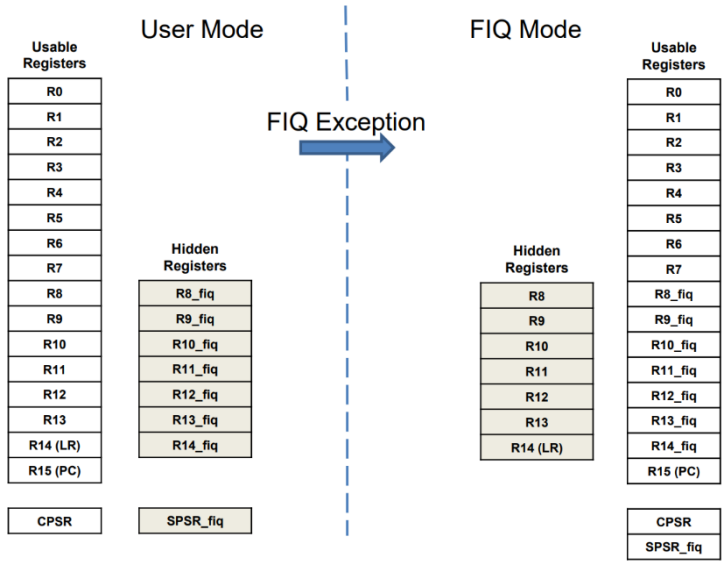
User	System	Fast Interrupt	Interrupt	Supervisor	Abort	Undefined
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	**R8_fiq**	R8	R8	R8	R8
R9	R9	**R9_fiq**	R9	R9	R9	R9
R10	R10	**R10_fiq**	R10	R10	R10	R10
R11	R11	**R11_fiq**	R11	R11	R11	R11
R12	R12	**R12_fiq**	R12	R12	R12	R12
R13 (SP)	R13 (SP)	**R13_fiq**	**R13_irq**	**R13_svc**	**R13_abt**	**R13_und**
R14 (LR)	R14 (LR)	**R14_fiq**	**R14_irq**	**R14_svc**	**R14_abt**	**R14_und**
R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)	R15 (PC)
Program Status Registers						
CPSR	CPSR	CPSR **SPSR_fiq**	CPSR **SPSR_irq**	CPSR **SPSR_svc**	CPSR **SPSR_abt**	CPSR **SPSR_und**

C'è un multiplexer davanti ai due registri duplicati che a seconda della parola di stato sceglie quale registro leggere/scrivere

In totale si hanno 37 registri di cui 31 generali e 6 di controllo.

Risposta ad un'eccezione

Cosa succede nell'arm quando arriva un'interruzione o un'eccezione?



Salviamo nel LR il prossimo valore dell'istruzione in cui dobbiamo tornare, viene copiato il cpsr della user mode dentro il registro spsr (saved program status register), viene cambiata la parola di stato, vengono mappati i registri e vengono disabilitate le interruzioni, in particolare le IRQ (interruzioni standard) vengono sempre disabilitate, le fast interrupt vengono disabilitate solo se questa è una fast interrupt o un reset.

Il pc punterà all'indirizzo dell'handler per la specifica interruzione che abbiamo ricevuto, questi sono 6 passi che vengono eseguiti atomicamente come un'istruzione unica.

Ritornare da un'interruzione

Per ritornare da un'interruzione arm dobbiamo ripristinare la cpsr prendendola dal saved program status register e dobbiamo ripristinare il program counter, l'hardware deve fare almeno queste due azioni atomicamente.

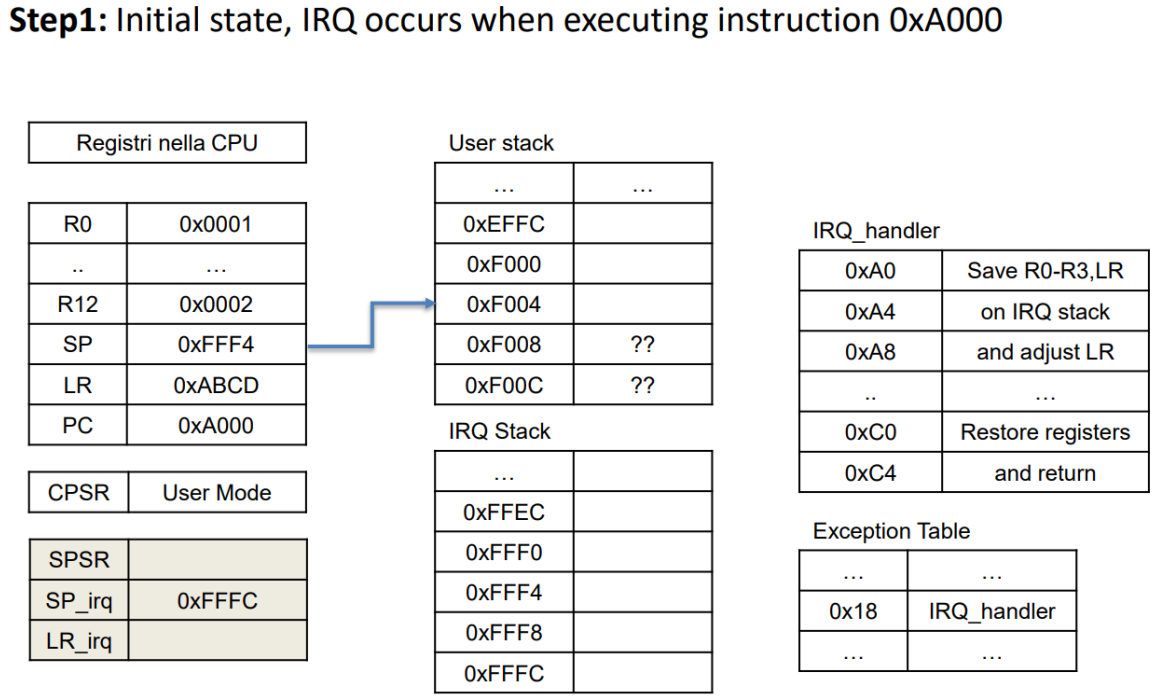
Si usano le istruzioni di tipo movs e subs a seconda se devo ripristinare il link register, il flag S fa sì che la parola di stato possa essere recuperata, possono esserci dei casi in cui

c'è da modificare il valore salvato nel pc, ad esempio si può necessitare di togliere #4 oppure #8, si usa quindi la subs.

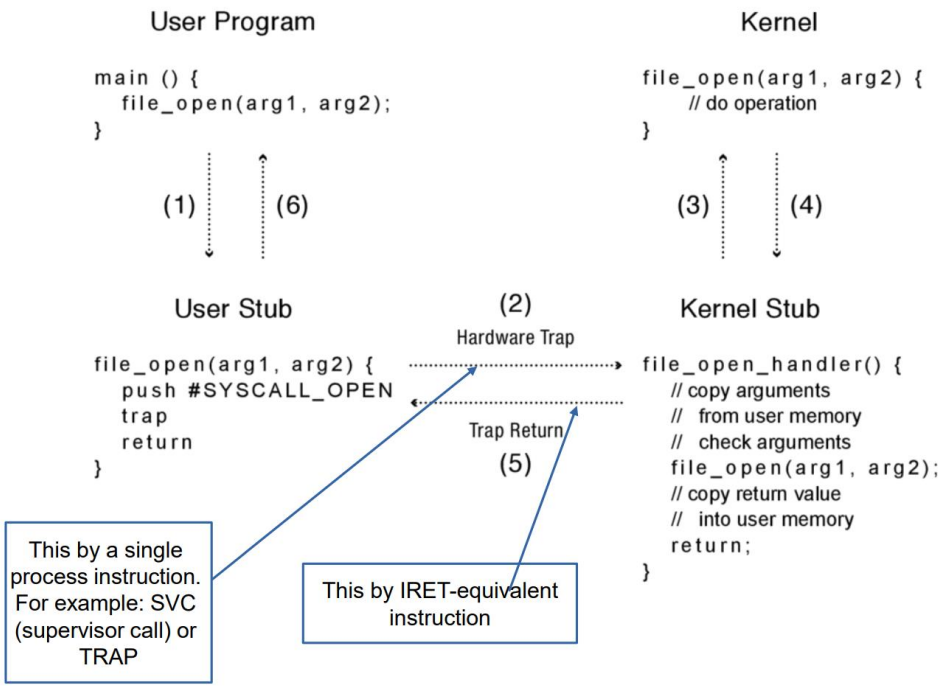
Nel caso della SVC si scriveva nel registro r7 l'indirizzo della system call e poi si faceva svc 0, dopo l'esecuzione della svc vogliamo eseguire l'istruzione successiva, quindi nel link register viene già caricato pc+4, dunque per arrivare all'istruzione successiva basta ripristinare il lr, nell'svc però non salviamo nessun registro generale, per cui sicuramente per la gestione dell'handler sarà stato lui a salvare i registri, ad esempio eseguendo una store multipla (push), è come fare una push nello stack del kernel seguita poi da una pop per ripristinarla. In realtà le vere push e pop non possono essere usate si usano operazioni speciali con la necessità di specificare questo simbolo ^, questo simbolo fa infatti sì che le istruzioni stmfd e ldmfd mi ripristinino anche la program status word, se non avessimo usato lo stack sarebbe bastato fare solo una mov.

```
SUB LR, LR, #4              // handler entry
STMFD SP!, {reglist, LR}
  ....
LDMFD SP!, {reglist, PC}^   // handler exit
```

Cliccare sull'immagine per vedere l'esempio



Software interrupts aka System Call



Con una system call un programma utente chiede un servizio al sistema operativo, di fatto la chiamata che vado a fare non è la system call, è un wrapper, si chiama infatti un'altra funzione in spazio utente a cui vengono passati tutti i parametri, in questa funzione si fanno dei controlli iniziali, esempio controlli sui parametri, viene poi preparato il passaggio dei parametri e l'invocazione della svc, i parametri vengono passati tramite lo stack, si invoca poi la syscall e effettuiamo il passaggio da user a supervisor, quando facciamo questo passaggio, non parte subito l'handler per fare la gestione perché c'è bisogno di recuperare gli argomenti dallo stack o dai registri.

Qui abbiamo subito un problema in quanto l'utente lavora con indirizzi logici e noi al livello kernel vogliamo lavorare con indirizzi fisici, in particolare dobbiamo essere sicuri che gli indirizzi che ci sono stati passati siano indirizzi validi.

(Questo è un passaggio estremamente delicato e servono molti controlli che sono fatti sia lato utente che lato kernel.)

Validati gli indirizzi vengono presi i dati e vengono copiati nello stack del kernel perché vogliamo evitare che vengano modificati da un altro processo, se qualcosa va storto ritorniamo (IRET).

Supponendo che tutti i controlli siano andati a buon fine chiamiamo la syscall ed eseguiamo l'operazione, se il processo in esecuzione è particolarmente lungo potrebbe venire salvato e potrebbe essere fatto content_switch, perché la gestione potrebbe essere particolarmente costosa e richiedere tanto tempo per essere completata, una volta completata torneremo indietro.

In base ai vari sistemi ci sono diverse modalità per le system call di gestione degli argomenti.

SWI handler

```
SWI_Handler                    ; top-level handler
    STMFD   SP!, {R0-R12, LR}  ; pushes registers into the stack
    ADR R8, SYS_CALL_TABLE     ; load syscall table pointer in R8
    ; R7 contains the SC number
    ADD R7, R7, #_SYSCALL_BASE ; OS entry of the sys_* routine
    ; sanity checks
    ;
    LDR PC [R8, R7, LSL #2]    ; call sys_* routing
    ; ...
    ; result of the SVC stored in R0
    ; ...
    ; restore user registers
    ;
    MOVS PC, LR                ; return from handler restoring CPSR
```

All'inizio l'handler copia r0-r12 e lr dopo che è stato fatto lo switch.

Prepara in r8 l'indirizzo del puntatore della tabella delle syscall.

In r7 aggiunge il valore base della tabella delle syscall in modo da prendere l'entry giusta.

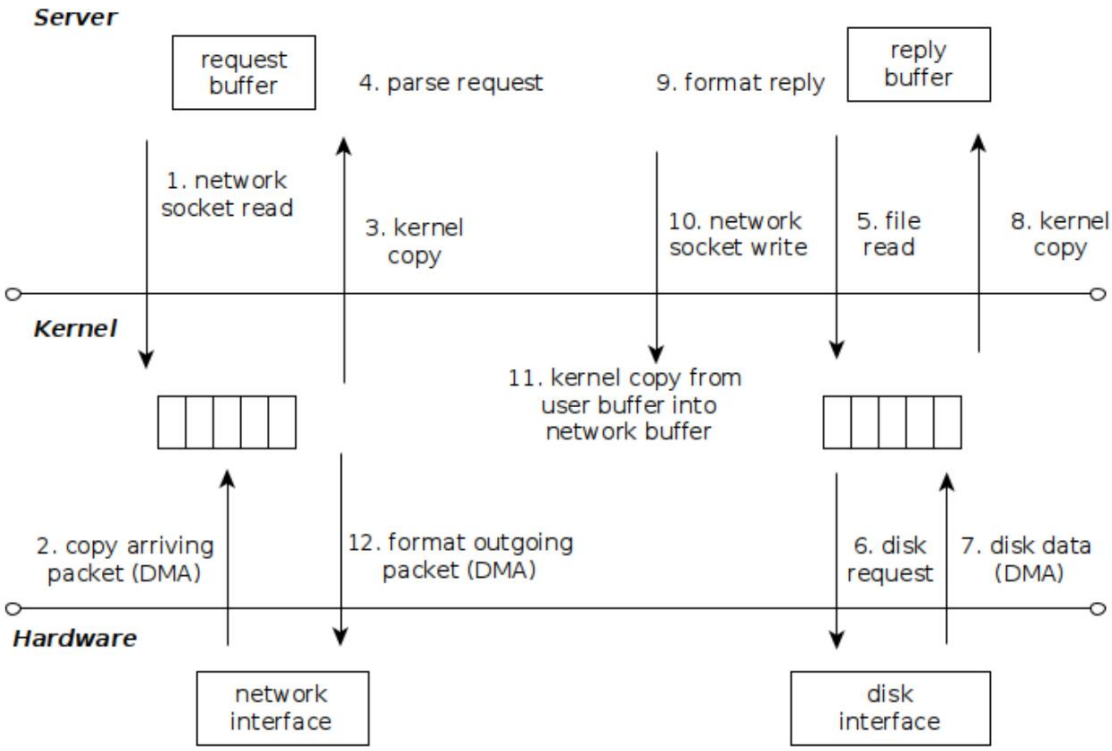
Esegue dei check.

Esegue la chiamata alla syscall mettendo nel pc il valore di r8 con offset r7 shiftato di 2.

Il risultato della svc sarà messo in r0.

Infine, si fa l'IRET ripristinando il LR nel PC in modalità supervisor così da recuperare anche la scpsr.

Esempio:



Upcall

Ci sono dei casi in cui è l'applicazione che ha bisogno di essere notificata dal sistema operativo?

Sì, abbiamo l'opposto delle system call, su questo concetto di upcall si basano tutti gli aspetti di virtualizzazione dell'hardware.

In gergo si chiamano segnali, può capitare ad esempio di dover killare un processo, ovvero mandiamo un segnale dal sistema operativo ad un determinato PID per interrompere un processo.

Come per le interrupt avremo un signal handler, uno stack separato, il cosiddetto signal stack, anche se in questo caso il signal stack si trova in user space come anche il signal handler gira in user space. Ci sarà la possibilità di passare da una modalità all'altra, l'equivalente della IRET con la funzionalità opposta, ovvero vogliamo tornare in stato kernel, così come per le interruzioni possiamo mascherare anche le upcall.

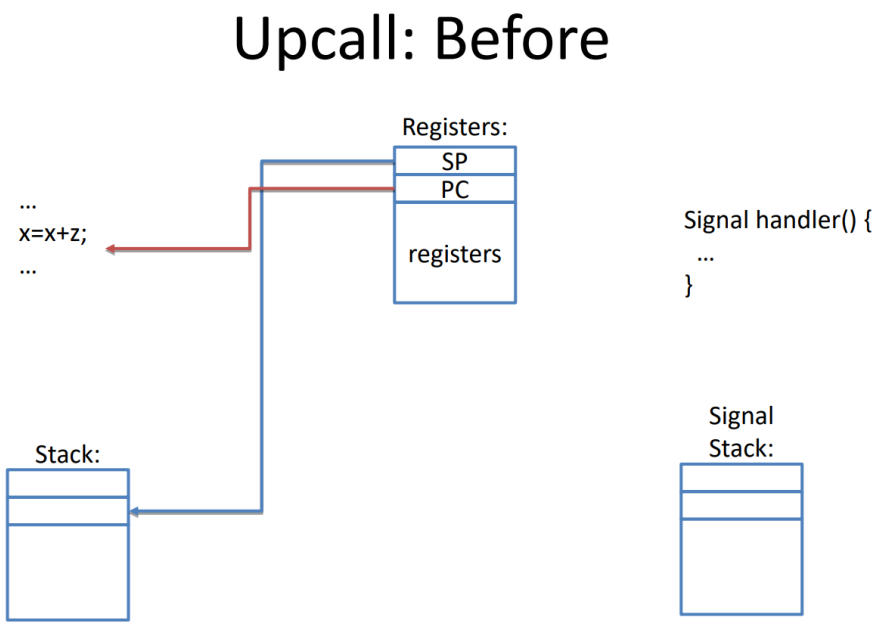
Le interruzioni sono mascherabili, ovvero posso nasconderle per un certo lasso di tempo, alcune interruzioni non possono essere mascherate come ad esempio il reset.

Così come per le interruzioni ci sono dei segnali che non sono mascherabili, immaginiamo di avere un programma che fa while(true) parte e non si ferma più, abbiamo disabilitato tutti i segnali, se questo fosse possibile l'admin di sistema potrebbe provare a fare kill del nostro pid, ma se tutti i segnali sono disabilitati quel processo non sarebbe più interrompibile, alcuni segnali come ad esempio sig_kill non sono mascherabili, l'amministratore potrà sempre fare sig_kill di un certo pid.

Ci sono molti altri segnali; come segnali mandati a processi per eseguire determinate azioni/eventi. Potremmo codificare codice in C in questo modo: se ricevo un determinato segnale faccio questo, altrimenti faccio altro. La comunicazione tra processi essendo isolati avviene anche grazie al kernel e ai segnali.

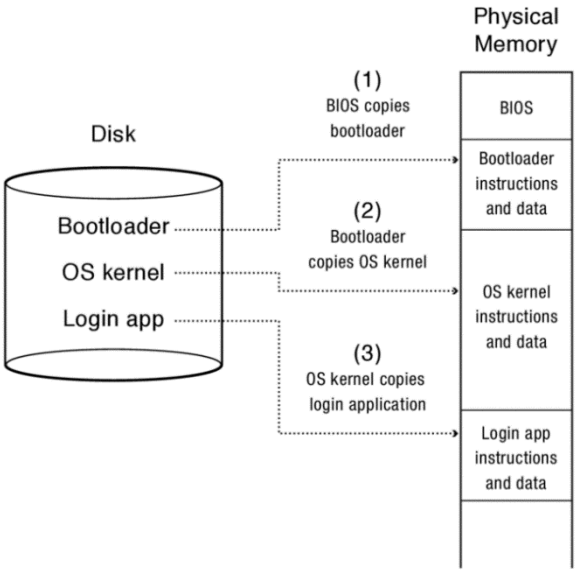
La gestione dei segnali è molto utile in molti casi ed è molto simile alla gestione delle interruzioni, i meccanismi che chiediamo all'hardware sono praticamente gli stessi.

Ctrl+click sull'immagine per vedere esempio



Kernel Booting

Il kernel è un software complesso e grande, sta sul disco e dobbiamo caricarlo, come possiamo fare?



C'è un piccolo software che risiede nella rom, quello che si chiama bios e che ha il minimo indispensabile per poter caricare dal disco un altro software che è il bootloader. Quando accendiamo la macchina dalla rom vengono caricate sul registro della cpu le istruzioni base per poter caricare il bootloader in memoria, il bootloader non fa altro che caricare il kernel in una parte precisa della memoria e inizializzare l'interrupt table e tutto quello che serve per far partire il SO, dopo un po' il bootloader lascia lo spazio ai driver iniziali del kernel che permettono di iniziare a gestire i dispositivi.

Il kernel arriva a far partire il primo processo che è quello di login, è il primo processo che opera in modalità user, il passaggio da kernel a user si fa tramite una IRET.

Summary: HW support for OSs

- **Privilege levels**, at least two: user and kernel level.
- **Privileged instructions**: instructions available only in kernel mode.
- **Memory translation** prevents user programs from accessing kernel data structures and aids in memory management.
- **Processor exceptions** trap to the kernel on a privilege violation or other unexpected event.
- **Timer interrupts** return control to the kernel on time expiration.
- **Device interrupts** return control to the kernel to signal I/O completion.
- **Interprocessor interrupts** cause another processor to return control to the kernel.
- **Interrupt masking** prevents interrupts from being delivered at inopportune times.
- **System calls** trap to the kernel to perform a privileged action on behalf of a user program.
- **Return from interrupt**: switch from kernel mode to user mode, to a specific location in a user process.
- **Boot ROM**: code that loads startup routines from disk into memory.
- **Support for Virtualization:** hypervisor (aka VMM) and additional privilege levels

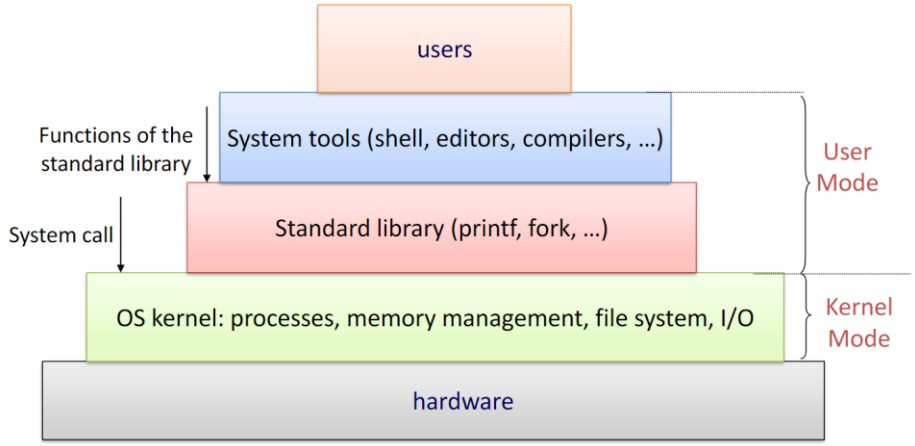
To support threads, we will need one additional mechanism (described later):

- **Atomic read-modify-write instructions** used to implement synchronization in multi-threaded programs.

La parte di supporto per la virtualizzazione o hypervisor, non è stata trattata, serve un minimo di supporto hardware per poter gestire macchine virtuali.

L'ultimo aspetto è un aspetto che vedremo più avanti quando vedremo programmi multi-thread.

Architettura unix



Un processo ha un suo spazio di indirizzamento e una struttura logica ben definita, abbiamo una parte codice, una parte dati, una parte heap (cresce verso l'alto) in cima abbiamo ambiente e stack (che cresce verso il basso).

La chiamata di sistema per creare una entry nella tabella dei processi è la fork(), un processo può fare un certo numero di operazioni, open, read, write, close, ma anche operazioni per interagire con altri processi.

Esistono meccanismi che permettono lo scambio di messaggi tra processi, in ambiente locale sfruttando ad esempio le pipe. La tabella dei descrittori dei file al momento della fork viene copiata dal processo padre al processo figlio, possiamo metterci d'accordo sull'utilizzo dei descrittori della pipe, per scrivere e leggere.

Ci sono anche altri meccanismi come i socket.

Altri meccanismi ancora operano ad ambiente globale ovvero condividono porzioni di memoria, ogni processo ha un suo spazio di indirizzamento privato, è però possibile dedicare un segmento di memoria che magari si mappa diversamente nei due processi ma che in realtà è lo stesso.

Shell

Anche se non ce ne accorgiamo, interagendo con la shell andiamo a fare chiamate di sistema, la shell forka un processo che esegue il comando con i parametri che gli abbiamo passato, siccome il comando scrive in stdout vediamo l'output del comando.

In realtà la shell non fa altro che fare per noi fork, exec, wait, dup ecct…

Creazione dei processi in windows:

Non esiste una chiamata equivalente a fork(), si usano chiamate un po' più complesse, che prendono molti parametri per avere tante funzionalità, le varie funzionalità dipendono dagli argomenti che passo alla funzione.

La createProcess è l'equivalente di una fork.

Unix utilizza un approccio minimalista, ovvero diamo alle funzionalità minime delle singole chiamate.

• UNIX fork – system call to create a copy of the current process, and start it running

– No arguments!

• UNIX exec – system call to change the program being run by the current process

• UNIX wait – system call to wait for a process to finish

• UNIX signal – system call to send notifications among processes

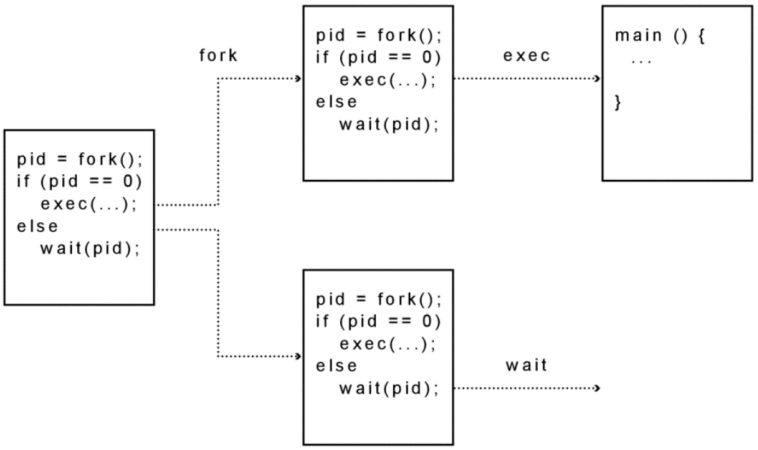
In generale per i processi quelle che utilizziamo sono queste in aggiunta con la chiamata exit, ovvero la chiamata che ci permette di ritornare con il codice di uscita al processo padre, che potrà ricevere il valore di uscita del processo attraverso la wait, più le dup per la comunicazione.

La fork non fa altro che duplicare una copia identica del processo, il pid del figlio sarà 0, per il padre invece si avrà un id positivo.

Ogni volta che c'è un errore, per controllarlo dobbiamo controllare la variabile errno.

Un esempio è readDir, la readDir che è una chiamata di libreria ci ritorna null sia se c'è un errore e sia se abbiamo finito, ovvero se non abbiamo più file, possiamo distinguere i due casi andando a leggere errno.

Unix Process Management



L'exit status è una risorsa del processo ed è scritta nel pcb.

Una volta che i processi sono partiti è compito dello scheduler determinare l'ordine di esecuzione.

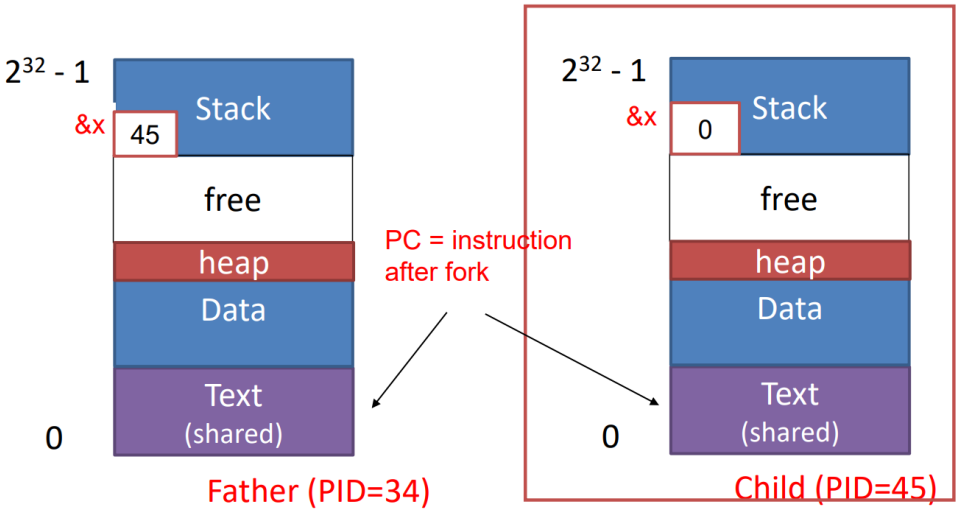
La funzione fork può fallire, un caso in cui ritorna è ad esempio quando ritorna -1 e significa che non ci sono più risorse disponibili.

L'exec può fallire, se dopo l'exec ho un'istruzione e l'exec ha successo quell'istruzione verrà ignorata, se la exec ritorna ed eseguo quell'istruzione allora c'è stato un errore.

Può la wait ritornare immediatamente? Se sì, perche?

Sì se ad esempio il figlio è già terminato.

Implementazione della fork in unix



Unix exec

Ci sono diverse exec che funzionano in vari modi, possiamo chiamare una exec senza aver fatto una fork, il programma corrente verrà buttato via e verrà eseguito il comando assegnatogli.

Se hanno successo non hanno return, se falliscono il codice di errore si trova in errno, dopo l'esecuzione mantiene il pid e il pcb, ma cambia il puntatore al codice e alla data memory, resetta i segnali, ma mantiene il kernel stack e un insieme di risorse che erano già state assegnate, in particolare tutto quello che riguarda i file aperti, questo è molto importante per fare pipelining e ridirezione.

Un processo termina quando chiama la system call exit() oppure quando viene lanciata un'eccezione se si fanno azioni illegali, terminare un processo significa liberare tutte le risorse di quel processo tranne la risorsa in cui si scrive il valore di ritorno, il padre può sapere perché il figlio è terminato.

Si usano ad esempio delle macro per estrarre da una maschera di bit il motivo della terminazione, ad esempio se è stato terminato da un segnale possiamo capire da quale segnale è stato terminato, se è stato terminato da una exit possiamo recuperare il valore di ritorno.

Se il processo termina ma il padre non ha ancora chiamato la wait quel processo prende il nome di zombie, non c'è modo di killarlo finchè non viene chiamata la wait o il processo padre termina.

Se invece il processo padre è già terminato il processo figlio viene adottato dal processo init e si attende la sua terminazione.

Unix I/O

In unix tutto viene visto come un file, anche la comunicazione tra processi, le pipe sono infatti comunicazioni tramite file descriptor.

Le pipe con nome possono essere utilizzate tra processi non parenti, le pipe senza nome invece possono essere usate solamente tra processi che condividono la tabella dei descrittori dei file.

Tutti questi dispositivi, sono tutti accedibili con le chiamate di sistema open, read e write.

Per le directory usiamo un'astrazione che è la opendir che rende l'accesso più facile.

La open è una di quelle poche chiamate che ha molti parametri, o comunque molti flag, in quanto la open è in grado di aprire file di diversa natura, es: directory, file, pipe, socket, disco ecct… deve avere la possibilità di essere chiamata con dei flag.

Inoltre è una chiamata molto potente perché combina di fatto più chiamate insieme, noi possiamo specificare i flag della open per dire ad esempio, fallisci se il file esiste, se il file non esiste crealo.

Perché è stata fatta la scelta di creare questa open?

Se avessimo avuto chiamate separate, non avremmo potuto eseguire l'istruzione in modo atomico.

Questo aspetto dell'atomicità è estremamente importante, sennò rischiamo di avere percorsi nell'esecuzione di questo codice che potrebbero fallire o non fallire se altri processi sono andati in esecuzione (percorsi critici).

Con questo approccio qua o fallisce o va a buon fine indipendentemente dagli altri processi.

Concorrenza

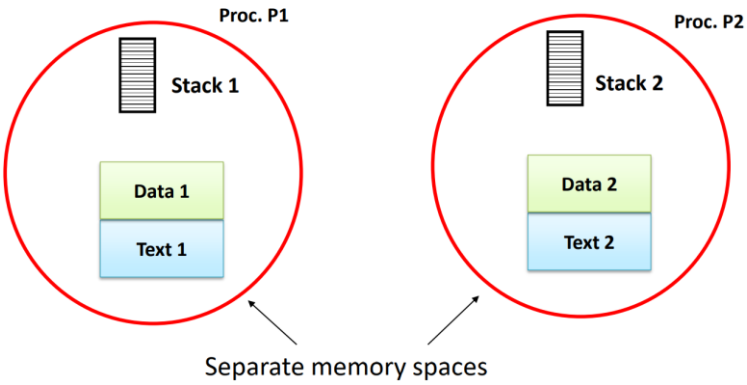
La motivazione per introdurre la concorrenza è che sia nei SO che nelle applicazioni è necessario eseguire più operazioni alla volta, questo è vero non solo per i programmi paralleli, che cercano di sfruttare più core della macchina, vale anche per sistemi concorrenti per la gestione di richieste che arrivano contemporaneamente.

Le richieste devono essere gestite concorrentemente, ovvero portate avanti contemporaneamente, questo può essere fatto sia su una macchina multi-core che su una single-core, nel primo caso si parla di parallelismo su core dedicati, si può avere la concorrenza anche su una macchina single-core, single cpu, dove la macchina darà l'esecuzione al SO o all'app.

MTAO (multiple thing at once) cose necessarie da fare.

Questo concetto di concorrenza è utile anche per attività semplici, come ad esempio la scrittura sullo schermo o la lettura di caratteri da tastiera.

Un modo per implementare la concorrenza può essere quello di implementare queste attività con processi separati, questo è fattibile, però ci sono delle note negative perché ogni processo come abbiamo visto, per garantire la sicurezza implementa un proprio spazio di indirizzamento privato non accessibile da altri processi, queste attività concorrenti però sono spesso correlate tra loro, ovvero magari un'attività ha bisogno di sincronizzarsi o di avere il risultato di un'altra. A seguito del fatto che il processo ha spazio di indirizzamento privato, la comunicazione tra due processi deve essere sempre mediata dal kernel, con vari meccanismi come ad esempio le pipe. Questo genera un costo, in quanto ogni operazione è una system call, questo procedimento, perfettamente lecito e funzionante è usato in molti contesti, ma in altri è troppo costoso, viene infatti introdotto quello che si chiama overhead, ovvero il costo per cambiare da una modalità di stato all'altra.



C'è l'esigenza di avere qualcosa meno costoso, dal punto di vista prettamente funzionale potremmo realizzare tutto con processi.

Con i thread facciamo le stesse cose che faremmo con solo i processi, ma la comunicazione ci costa molto meno.

Thread

Un thread è un'unità di esecuzione, ovvero una sequenza di istruzioni di un processo che può essere gestita dal SO in modo indipendente, come se fosse un processo, abbiamo bisogno di un'entità thread perché ci sono delle esigenze da parte della struttura del programma, a volte conviene programmare con i thread piuttosto che programmare in altri modi, c'è un aspetto anche legato alle prestazioni in quanto riusciamo a diminuire il costo che c'è tra le interazioni di queste entità che vogliamo far proseguire contestualmente.

Sono utili anche per poter interagire con dispositivi di I/O particolarmente lenti.

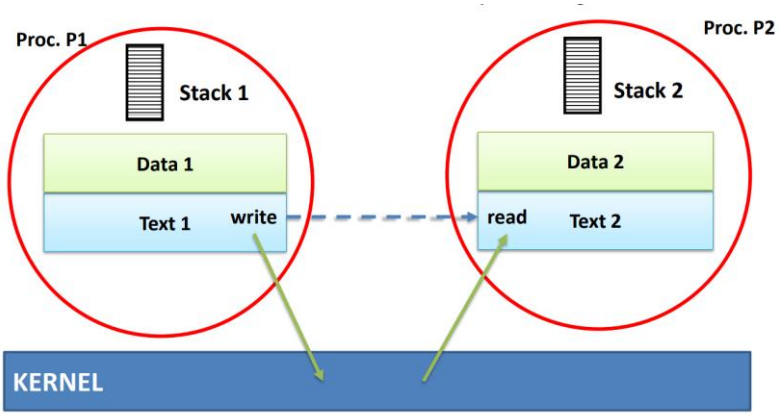
Dal punto di vista della protezione:

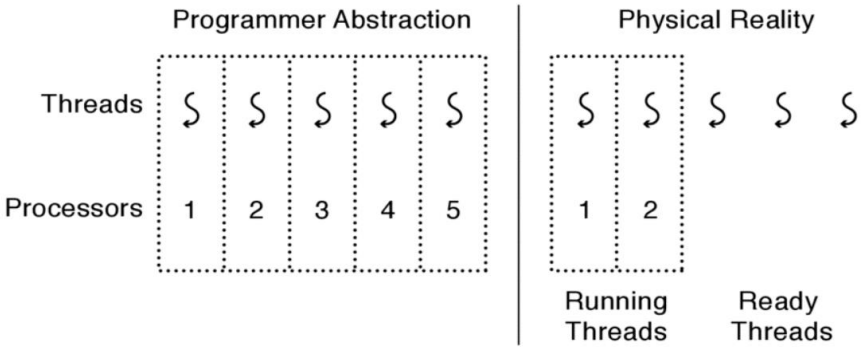
Ci sono diversi modi di implementare i thread, per garantire la sicurezza potremmo pensare di avere come per i processi uno spazio riservato, oppure implementarli senza riservare lo spazio, più rendiamo l'entità thread isolata e più sarà alto il costo di interazione.

I processi sono caratterizzati da un dominio di protezione, i thread invece hanno solo un concetto di modalità di esecuzione e di non esecuzione.

Un thread è una sequenza di istruzioni in un flusso di controllo, per ogni thread possiamo immaginare come un processore logico dedicato a quel thread.

È come avere *n* unità di elaborazione che procedono parallelamente, come se ognuna avesse un core virtuale assegnato che gli permette di avanzare nell'esecuzione indipendentemente dalle altre, anche se la macchina vera ha ad esempio solo due core fisici, in parallelo dunque ne potrò eseguire al più due.





Un processo è un thread in esecuzione, quando c'è un unico flusso non c'è distinzione.

Quando si parla di thread e si guarda alla modalità di esecuzione non possiamo fare affidamento alla velocità del thread stesso in quanto non dipende da lui.

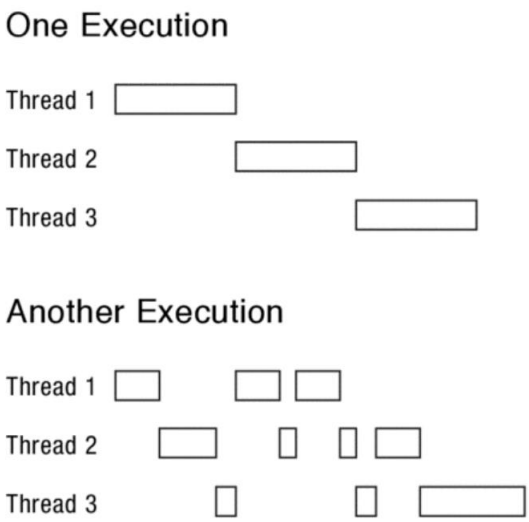
Possibili esecuzioni dei processi a seconda dei core:

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
x = x + 1;	x = x + 1;	x = x + 1;	x = x + 1;
y = y + x;	y = y + x;	.....	y = y + x;
z = x + 5y;	z = x + 5y;	Thread is suspended.	.....
.	.	Other thread(s) run.	Thread is suspended.
.	.	Thread is resumed.	Other thread(s) run.
.	.	.....	Thread is resumed.
		y = y + x;	.....
		z = x + 5y;	z = x + 5y;

Supponiamo di avere un programma che dentro ha 10 thread, ovvero forka 10 thread con pthread_create(), li lancia e a ognuno viene assegnato un id, fissa un valore di ritorno che viene letto dal thread padre (quello che li ha creati).

I thread nonostante siano stati creati uno dopo l'altro la loro esecuzione è del tutto casuale, rilanciando di nuovo il programma otteniamo magari un'esecuzione ancora diversa, tutto questo dipende dal SO, dall'hardware e da molti altri fattori.

Il main è un thread anch'esso.

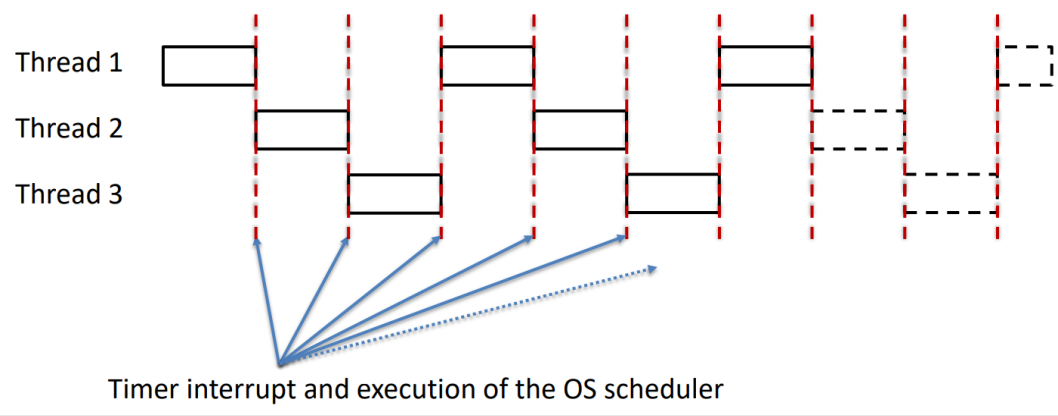


Implementazione dei thread

Si usa il tcb (thread control block) una struttura che memorizza le informazioni sul thread, si definiscono un insieme di operazioni sui thread stessi.

Dal punto di vista del SO il thread può essere schedulato (ovvero gestito) direttamente dal SO.

Lo scheduler è una funzione del SO che ha il compito di assegnare processori fisici, i core, ai thread o ai processi in generale.



Il timer è settato a scadere ad intervalli regolari che prendono il nome di *quanti* di tempo, il SO vede che c'è un processo che ha al suo interno più thread, prende il primo e lo mette in esecuzione sulla cpu per un quanto di tempo, quando arriva l'interruzione del timer, entra in gioco lo scheduler che decide cosa fare, potremmo continuare con il thread_1 o eseguire altri thread, ovviamente se viene cambiato thread si dovrà salvare tutte le informazioni del thread corrente.

Questo schema prende il nome di *preemptive multi-thread* un thread può essere in un dato istante de-schedulato ovvero la cpu fisica può essere tolta ad un thread e assegnata ad un altro, che sia dello stesso processo o di un altro.

Cooperative multithreading

È un modello in cui non c'è una funzione di scheduling che periodicamente stoppa i thread, il thread se è in esecuzione decide lui stesso se rilasciare la cpu ad un altro thread, i thread cooperano tra di loro alternandosi nell'esecuzione.

Questo porta dei vantaggi come ad esempio, nei sistemi uni-core siamo sicuri che se iniziamo una sequenza di istruzioni e la vogliamo eseguire senza essere mai interrotti lo possiamo fare, mentre in quello preemptive dobbiamo regolare la gestione delle interruzioni.

In un sistema multi-core il cooperative multi-threading crea solo problemi.

Simple thread API

• thread_create(thread, func, args)

– Creates a new thread, storing information about it in thread, to run func(args)

• thread_yield()

– The calling thread voluntarily gives up the processor to let some other thread(s) run

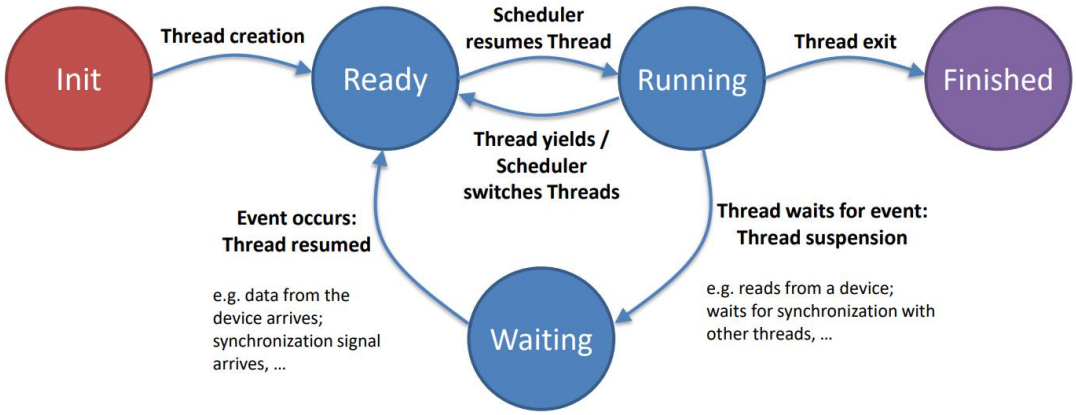
• thread_join(thread)

– Waits for thread to finish if it has not already done so, then returns the exit status of the thread. It can be called only once for each thread

• thread_exit(exit_status)

– Finishes the current thread. Stores the exit_status in the threads's data structure. If another thread is waiting in the thread_join, resumes it.

Ciclo di vita



È lo stesso più o meno di quello di un processo.

Le parti blu sono implementate con una coda, la parte rossa e la parte viola sono il punto di ingresso e il punto di uscita.

Dallo stato ready, posso andare in esecuzione se lo scheduler me lo permette.

Ci pensa il SO a far transire dalla coda di attesa alla coda ready, una volta fatta una pthread_exit entriamo nello stato di terminazione ed è possibile leggere il suo valore di uscita.

Lo stato del thread può essere:

- Init
- Ready
- Running
- Waiting
- Finished

I thread si trovano tutti nel thread control block tranne quando siamo nello stato di running, in quel caso siamo nel processore, lo stato è dato dai registri della cpu.

I thread possono essere implementati in diversi modi, potremmo implementare single thread-process ovvero non averli, in questo caso si hanno processi formati da un singolo thread.

Potremmo avere che i thread li abbiamo ma sono solo a livello utente, ovvero all'interno del processo, ma il SO non li vede, il SO vede solo il processo e i domini di protezione, questa modalità si chiama multiple user-level thread.

Possiamo avere (ed è il sistema attuale) un misto tra processi single-thread e processi multi-thread, in cui i thread sono visibili al livello del sistema operativo e anche quest'ultimo usa i thread, il SO non solo li vede, li usa ma è anche in grado di schedularli.

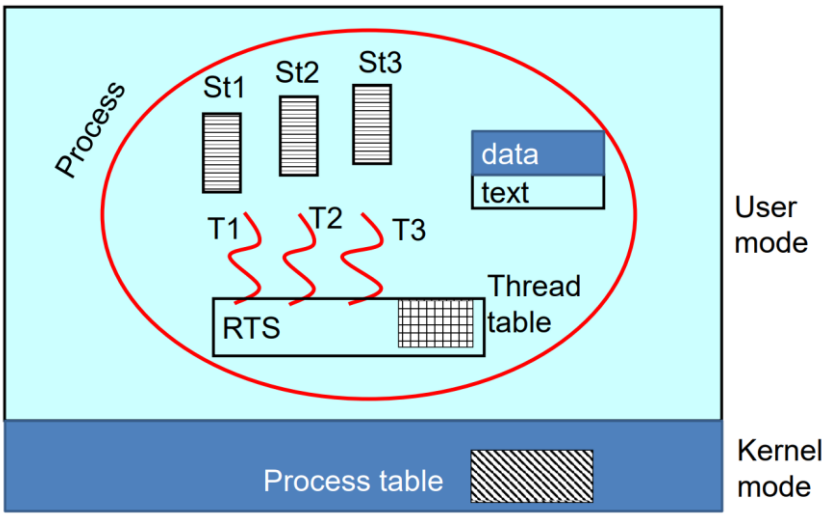
User Level thread

Sono thread che esistono in modalità utente, ma che il SO non è in grado di vedere.

Esiste una libreria che implementa le chiamate viste prima che però non sono chiamate di sistema, ma sono chiamate di libreria e servono per gestire i thread a livello utente.

Tutto quello che rappresenta un thread (tcb) non risiede in kernel space, bensì in user space, all'interno della libreria, questo vale anche per lo scheduling che non è fatto dal SO ma da una funzione di libreria.

Esisterà quello che si chiama runtime support che ha il compito di fare da scheduler per i thread di un determinato processo.



Abbiamo un dominio di protezione che è un processo, il processo ha al suo interno *n* thread, e una libreria che è in grado di gestirli. Tutto quello che rappresenta un thread esiste solamente all'interno di questo dominio di protezione.

Nella modalità user level thread quando chiediamo servizi al SO, quest'ultimo vede che è stato chiesto dal processo e non dal singolo thread, quindi viene sospeso l'intero processo, sospendendo magari anche thread al suo interno che non necessitavano del servizio richiesto e che avrebbero potuto continuare l'esecuzione.

Il vantaggio è che lo scheduling dei thread in questo modo è molto meno costoso in quanto non è ad opera del sistema operativo.

In caso di multi-core con questo sistema non sfruttiamo il parallelismo in quanto il processo è assegnato ad un singolo core e dunque i vari thread di quel processo si alternano su quel core.

Implementazione degli user-level threads

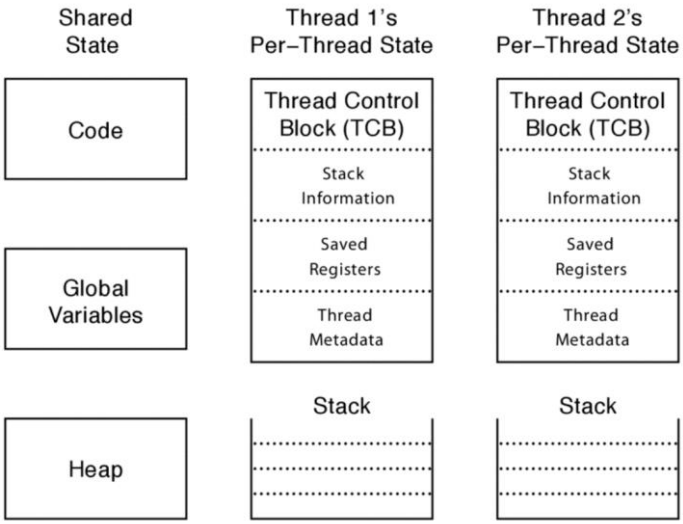
Si creano con la thread-create() in cui viene allocato uno stack, ogni thread infatti ha un proprio stack dedicato, essendo nello stesso dominio di protezione però thread diversi posso accedere e sporcare lo stack di un altro thread.

Generalmente quando si crea un thread viene allocata un'area sull'heap per quel thread specifico che viene gestita come uno stack.

Quando viene fatto partire un thread di fatto quello che succede è che non viene eseguita direttamente la funzione function che passo, esiste infatti uno *stab* che è una funzione di libreria, questa chiama la funzione function con gli argomenti, e si assicura che la funzione termini sempre con un dato exit status, questa funzione stab ha senso solo nello user-level.

Abbiamo poi la ready-list che è gestita dal run-time system.

Dal punto di vista della memoria sebbene i thread si trovino all'interno dello stesso processo, non condividono tutti lo stack principale, ma ogni thread ha uno stack dedicato, tutte le informazioni sono contenute nel tcb.



Cosa succede se un thread alloca troppi dati sullo stack?

Lo stack è allocato in una zona di memoria, se supero la dimensione che mi è stata assegnata, può succedere qualunque cosa, potrei sporcare lo stack di altri thread o scrivere in una zona di memoria che non era stata allocata, questa cosa è imprevedibile. L'unica cosa che si può fare e che alcune librerie fanno è mettere dei flag, ovvero una determinata parola in delle righe dello stack che ci dicono se abbiamo superato la dimensione massima, tutto questo viene fatto a software (ovvero da librerie) e non dal SO.

I pro dello user-level sono che la creazione, la terminazione e il content switch sono molto efficienti, posso inoltre implementare le librerie in un qualunque SO.

I contro sono che le syscall bloccanti richiedono una gestione complessa e non ho vantaggi dai sistemi multi-core.

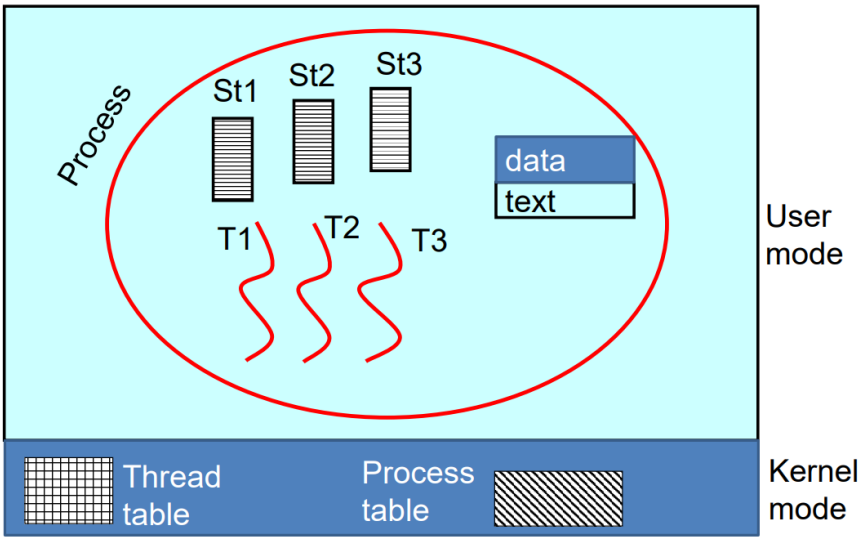
Kernel level threads

I thread lanciati da un processo sono visibili anche a livello kernel, la tabella dei thread è visibile in kernel space ed è accessibile da tutti i processi.

Questi thread vengono trattati come se fossero processi, le system call gestiscono l'attivazione, la terminazione e il content switch e lo scheduling è a carico del kernel.

Thread diversi di uno stesso processo possono essere eseguiti in parallelo su core diversi.

Il pcb è all'interno del kernel insieme alla process table, in kernel space abbiamo anche il tcb, e la thread table che è unica per tutti i thread, inclusi quelli del sistema operativo.



TCB e PCB

Nel PCB abbiamo il pid, il ppid e tutte le informazioni relative alla memoria assegnata a quel processo come ad esempio dove comincia l'heap, dove comincia lo stack ecct...

Ci sono le informazioni riguardo le risorse condivise, due processi diversi possono operare su uno stesso file a patto di dovute precauzioni. Tutte le risorse che sono uguali per tutti i thread stanno nel pcb, quelle che sono relative e specifiche per thread sono contenute nel tcb.

Nel tcb abbiamo il thread che è un entità che determina il flusso di esecuzione, abbiamo uno stato, che ci determina cosa sta facendo il thread, contiene infatti quello che il thread sta facendo e lo scheduling, oltre ai metadati, stack e variabili di errore come *errno*.

Lo scheduling è attivato da system call che possono essere bloccanti oppure dal timing ovvero dalle interruzioni, se il thread esegue syscall bloccanti, all'interno della gestione della chiamata di sistema, l'handler si renderà conto che dovrà sospendere quel thread dicendo allo scheduler di mandare in esecuzione un altro thread anche se di un altro processo.

Thread nei processi

Sono utili per molti motivi, possono essere schedulati indipendentemente e posso anche eseguirli in parallelo, ci permettono di nascondere la latenza di I/O, in quanto nel frattempo posso usare un thread per eseguire qualcos'altro, se fossi in un sistema single thread rimarrei in attesa del dato I/O.

Gli user level thread sono utili nonostante gli svantaggi, in quanto in alcuni contesti ha senso non demandare lo scheduling dei thread al sistema operativo, un esempio è quando

vogliamo dedicare i core soprattutto all'esecuzione dell'applicazione usando pochi core per le operazioni di I/O tramite chiamate di libreria con gli user-level thread.

L'altra opzione è quella utilizzata da tutti i sistemi correnti, in cui le syscall creano e gestiscono i thread, lo scheduling è di tipo preemptive.

Esistono altre opzioni tra cui la scheduler activations, questa è utilizzata solamente nei sistemi windows, e cerca di togliere gli svantaggi degli user-level threads, il SO fornisce un supporto esplicito che aiuta a fare lo scheduling a livello utente.

Questo permette di avere un minore overhead anche se la gestione diventa un po' più complicata.

Esistono altre due opzioni che sono l'asynchronous I/O in cui i thread non vengono usati, magari in sistemi multi-core, ovvero sistemi in cui non vogliamo mai bloccarci, rendiamo tutta la gestione non bloccante, e chiediamo al SO di mandare un evento, un upcall quando c'è un qualcosa pronto su un certo descrittore di file. Dal punto di vista della programmazione la gestione è molto più difficile in quanto non ci possiamo mai bloccare.

L'altro è l'asynchronous I/O + thread, abbiamo pochi thread e al loro interno portiamo avanti attività non bloccanti asincrone e in modalità standard.

Thread Switch

Che sia una chiamata o lo scadere del timer io devo switchare da un thread all'altro, abbiamo due casi principali per lo switch tra due thread, il primo è che un thread potrebbe decidere volontariamente di cedere la modalità d'esecuzione, questo lo possiamo fare con la thread_yield, la seconda modalità è quella legata alla gestione di eventi interrotti da interruzioni.

Le interruzioni sono visibili solo dal kernel, mentre le eccezioni quando vengono lanciate possono essere catturate dall'applicazione e gestite in un certo modo.

In entrambi i casi la modalità di gestione del content-switch è pressoché identica sia in kernel-level che user-level.

Content switch volontario

È la modalità di gestione tipica degli user-level thread, molto spesso la gestione del content switch è tramite cooperazione esplicita, un thread con un'esecuzione molto lunga attiva la callback e passa l'esecuzione ad un altro thread, in modo da non monopolizzare l'utilizzo della cpu.

Quello che avviene ogni volta che facciamo una thread_yield è quello di salvare i registri nel tcb, passare l'esecuzione ad un altro thread switchando il nuovo stack e ripristinando i registri di stato del nuovo thread dal tcb.

Per i kernel-level thread i passi sono esattamente gli stessi solo che vengono fatti in modalità kernel, e la return non è semplicemente il return di una funzione di libreria ma è la I_RET, ovvero l'istruzione che ripristina i registri di stato in modo atomico.

La thread_yield è implementata nello stesso identico modo nel kernel-level e nello user-level solo che nel primo caso alcuni passi vengono eseguiti in modalità kernel.

Fare thread_yield non significa sospendere il thread, significa metterlo in una particolare modalità che si chiama pronto, ovvero è pronto per essere eseguito anche se è stato lui a cedere l'esecuzione.

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
...	...	...

Content switch with interruption

I passi sono identici a quando abbiamo un'interruzione software.

Supponiamo arrivi un'interruzione dal timer, passa in esecuzione l'handler del timer, in questo caso entrerà in gioco il kernel, ci sono due modalità fast simple e una fast, quella fast è un'ottimizzazione dei passi della simple.

Nella fast simple dopo aver mandato in esecuzione l'interrupt handler, vengono salvati i registri sullo stack del kernel, l'handler deve decidere, siccome è un'operazione, di salvare tutti i registri sul kernel stack, a questo punto mando in esecuzione una funzione del kernel che mi permette di attivare lo scheduler e switchare thread, prima di mandare in esecuzione un altro thread deve salvare dallo stack del kernel dentro il tcb i registri del thread corrente, dovrà poi caricare i registri del nuovo thread nel kernel stack e da lì nei registri (fisici) per poi mandare in esecuzione il thread.

La versione fast anziché passare dai registri (fisici) al kernel stack e dal kernel stack al tcb, sa che si tratta di un'interruzione e quindi uno switch, dunque copia direttamente i registri dall'hardware nel thread control block senza passare dallo stack, questo vuol dire che dovrò avere almeno un registro che mi indichi con un puntatore il tcb del thread in esecuzione.

Analogamente dato che devo mandare in esecuzione un altro thread copio dal tcb ai registri della macchina hardware quelli necessari per il nuovo thread da mandare in esecuzione.

Questa è un'ottimizzazione, non cambiano i passi da fare.

switch_threads()

• Save registers (context) of the old thread from kernel stack in the TCB
• Move old thread's TCB to Ready List or to a Waiting List
• Select a new thread from the Ready List
• scheduling problem – discussed later
• 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…)

Overhead del thread switch

Il content switch è un'operazione costosa, in alcuni contesti dove le prestazioni sono un fattore critico si tende ad utilizzare gli user-level thread o degli accorgimenti particolari per ridurre le operazioni di content-switch.

Dobbiamo gestire i registri, salvarli e ripristinarli, dobbiamo gestire le code dei tcb, dobbiamo gestire le cache, in quanto essendo thread eseguiti magari su core diversi, le cache sono diverse e avremmo dunque molti miss.

C'è tutta una parte di gestione della memoria perché in qualche modo se mandiamo in esecuzione un thread di un altro processo, tutto quello che riguarda la gestione della memoria come stack heap ecct.. deve essere gestito.

Le azioni da fare nel content switch sono tante e richiedono tempo.

Let us consider again a simplified processor with special registers PC and PS, the user-level stack pointer SP and just two general registers R1, R2. The interrupt vector is in memory. The system uses a single kernel stack (shared for all threads). IRET to return. When it receives an interrupt, the processor:

- Sets kernel mode;
- Disables interrupts;
- Saves PC & PS & SP on the kernel stack
- Loads the new PC & PS from the interrupt vector
- Consequently it jumps to the interrupt handler in the kernel

Hardware

The IRET instruction:

- Enables interrupts;
- Sets user mode;
- Restores PC, PS & SP from the kernel stack; (consequently jumps back to the address at which the RUNNING thread had been interrupted in the past)

The interrupt handler:

- First saves the general registers on the kernel stack
- At the end, it restores the general registers and then executes IRET

Software

Ctrl+click sull'immagine per vedere il primo esempio

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	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

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

address	5000
PS	AA45
Interrupt vector	

base kernel SP	0FFF

Ctrl+click sull'immagine per vedere il secondo esempio

Hyp. B): thread T1 invokes a system call that blocks T1 and 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	
0FFE	
0FFD	
0FFC	
0FFB	
0FFA	

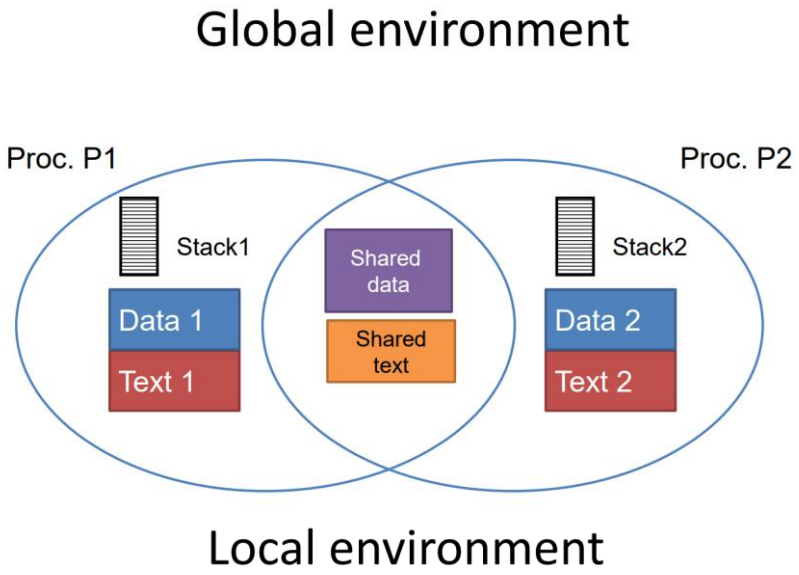
Registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

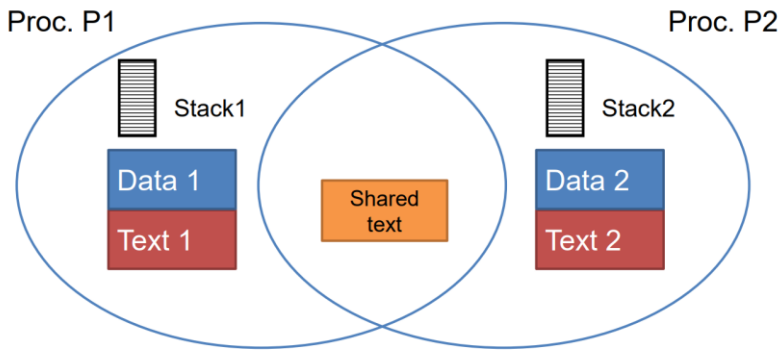
address	5000
PS	AA45
Interrupt vector	

base kernel SP	0FFF

Cooperation Model

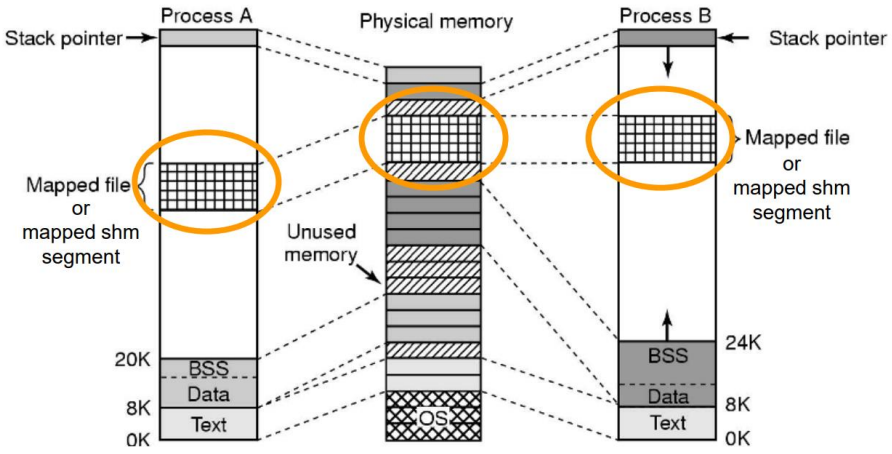
Esistono due modelli di sincronizzazione tra thread/processi, uno è tipico dei thread ovvero quello della shared memory, e l'altro è quello tipico dei processi in cui essendoci un livello di protezione si opera tramite cooperazione.





È facile definire un file condiviso tra processi distinti, è però un'operazione che noi facciamo esplicitamente, di base i processi non condividerebbero niente.

L'ambiente locale è quello tipico, dove la parte dati è contenuta all'interno del dominio di protezione e al più può essere condiviso il codice, la cooperazione avviene attraverso una comunicazione esplicita di messaggi.



Tutte le interazioni avvengono o tramite processi server, in cui vengono incapsulate delle variabili, o tramite un processo client che può chiedere il valore al server mediante astrazione dei canali di comunicazione come ad esempio pipe e socket, serve però sempre l'interazione del SO.

Ci occuperemo delle comunicazioni con variabili globali.

I SO che utilizzano thread e processi utilizzano di default il modello globale, all'interno il sistema operativo per cooperare utilizza delle variabili condivise.

Sincronizzazione

Come gestiamo la sincronizzazione tra l'utilizzo di variabili condivise?

Parliamo di thread che eseguono operazioni di read e write che operano su variabili condivise e che necessitano di un vero e proprio sistema di arbitraggio, altrimenti potremmo avere un comportamento indefinito che dipende dalla velocità con cui i processi vengono eseguiti.

Ci servono dei meccanismi per gestire la sincronizzazione.

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

Abbiamo due thread, i thread sono ad ambiente globale quindi condividono lo stesso spazio di indirizzamento, le funzioni sono stateless ovvero non hanno memoria.

Il thread1 esegue una certa funzione deterministica e memorizza il risultato nella variabile p, dopo setta una variabile globale isInit a true, l'altro thread esegue un ciclo true, fintanto che isInit non diventa false, quando questo ciclo viene superato, il thread2 chiama una funzione e gli passa p, il risultato viene memorizzato in q, aggiungiamo anche un controllo per verificare effettivamente che il calcolo eseguito sia corretto.

Questo codice può fallire per due motivi:

- Sia il compilatore che l'hardware potrebbero cambiare l'ordine ed eseguire istruzioni non nell'ordine specificato dal programma.
- Il compilatore potrebbe riordinare le istruzioni in base alle dipendenze, per sfruttare al meglio l'architettura sottostante.

Le istruzioni potrebbero inoltre essere riordinate dall'hardware, nelle cache avevamo ad esempio un buffer per cercare di velocizzare i cache hit, molte architetture per ragioni di ottimizzazione possono prendere alcune di queste scritture sospese nel write buffer e magari quelle che vanno nello stesso blocco di memoria accoppiarle per farle tutto insieme, in questo modo l'ordine delle istruzioni potrebbe venir modificato.

Se questi thread girassero su un unico core grandi problemi a livello architetturale non ci sarebbero.

Le architetture intel x86 hanno un meccanismo di scrittura che si chiama total store order, garantiscono che due scritture fatte da uno stesso thread avvengano sempre nello stesso ordine, l'arm ha un sistema più rilassato che si chiama weak memory ordering.

Come risolviamo questo problema?

Questo tipo di architetture con questo approccio aggressivo hanno delle istruzioni particolari a livello assembler, queste istruzioni si chiamano memory barrier, tutte le operazioni prima dell'istruzione di barriera devono essere completate prima di iniziare ad eseguire le istruzioni dopo l'istruzione di barriera.

I meccanismi di sincronizzazione che vedremo, inducono automaticamente una memory barrier.

In realtà le memory barrier non bastano avremmo anche infatti dei side-effect.

Definizioni:

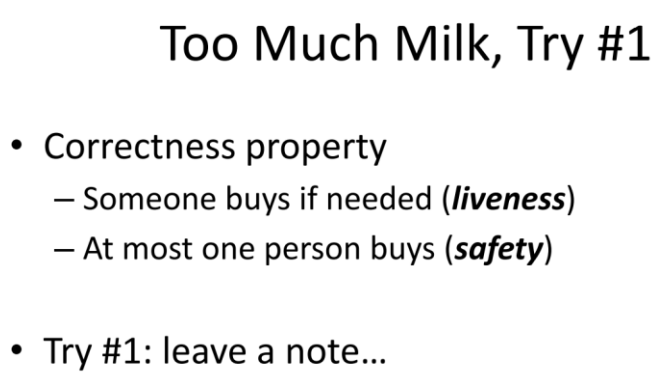
Race condition (corsa critica): Il risultato di un'operazione o di un programma dipende dall'ordine con cui vengono fatte le istruzioni.

Mutual exclusion: è quella proprietà che permette ad un thread di fare una particolare cosa in mutua esclusione, o la fa uno o la fa l'altro.

Lock: Si parla di variabili di sincronizzazione, sono dei meccanismi che ci permettono di realizzare la mutua esclusione.

Vogliamo garantire la correttezza dell'algoritmo e in particolare due proprietà, *liveness* e *safety*, la liveness ci dice che il problema che voglio risolvere viene risolto da almeno un thread, la safety vuol dire che il programma non entra in uno stato di inconsistenza, ovvero che non entra in uno stato diverso da quello che ci aspettiamo, vogliamo che almeno uno dei due thread completi il compito.

Cliccare su immagine per vedere l'esempio



Attesa Attiva

La prima soluzione è che i due thread si mettano d'accordo con un meccanismo di comunicazione per avvertire l'altro dell'inizio dell'esecuzione dell'operazione.

Esiste almeno un caso in cui questo non funzioni, i due thread potrebbero infatti entrambi iniziare ad eseguire lo stesso processo prima di lasciare il messaggio.

Può succede poi che nessuno dei due thread esegua il processo, se i messaggi vengono lasciati in interleaving ognuno dei due thread vede il messaggio dell'altro e nessuno esegue le operazioni.

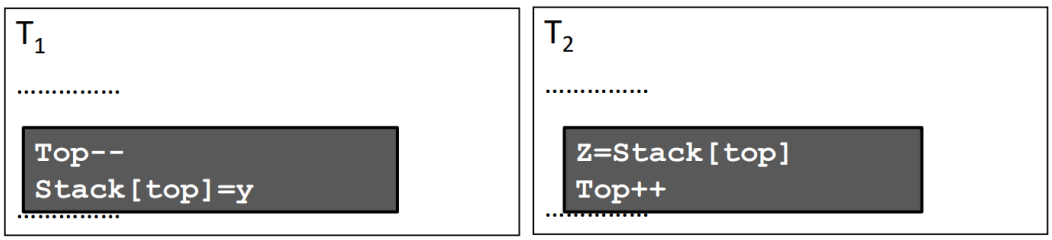
La soluzione vede una simmetria tra il codice dei due thread, inoltre il primo thread rimane in attesa senza fare niente, questa si chiama attesa attiva.

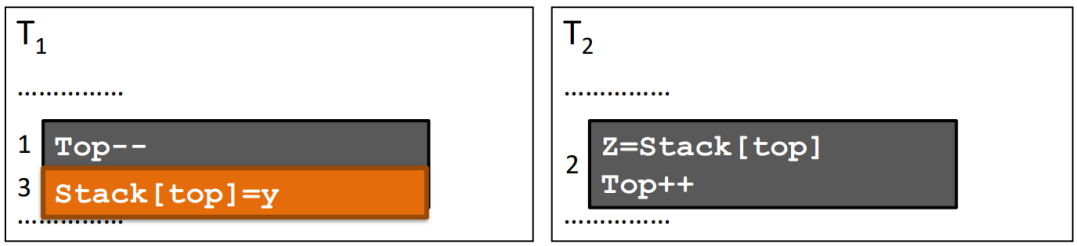
L'attesa attiva è quello che ho aggiunto per differenziarmi dalle soluzioni precedenti, è ciò che mi rende il codice asimmetrico e che mi potrebbe far perdere cicli di clock inutilmente, risolvendo però il problema. Questi svantaggi in generale non sono desiderati.

Si parla di *algoritmo di peterson* (la generalizzazione con n thread) e non funziona con architetture che hanno un modello di gestione della memoria rilassato.

Altro esempio:

Immaginiamo di avere due thread che vogliono gestire uno stack, vorremmo decrementare il puntatore alla testa dello stack e scriverci. Nell'altro thread vorremmo leggere il dato appena scritto e incrementare top, queste operazioni se eseguite in interleaving portano ad un risultato sbagliato.





Il problema della concorrenza nell'ambiente globale può essere realizzato sulla carta con particolari istruzioni di load/store, memory barrier e attesa attiva, questo però come abbiamo visto genera asimmetria del codice in quanto dobbiamo gestire l'attesa attiva.

Possiamo infatti evitare lo spreco di risorse usando altri metodi di sincronizzazione quali le variabili di lock.

Variabili di Lock

Queste variabili hanno almeno due operazioni una è acquire() e l'altra release(), le due operazioni sono chiusura del lucchetto e apertura del lucchetto, l'acquire se il lucchetto è chiuso ci fa attendere che il lucchetto venga aperto, quando qualcuno lo apre, ovvero fa una release se c'era qualcuno in attesa gli viene passato il controllo del lucchetto.

Se le utilizziamo nel modo corretto, ovvero la lock la chiamiamo all'inizio della sezione critica e la rilasciamo alla fine della sezione critica, le implementazioni di queste speciali operazioni ci garantiscono che al più ci sarà uno solo thread che ha il lucchetto chiuso ovvero che detiene la lock.

Se nessuno detiene la lock se chiamiamo la acquire chiudiamo il lucchetto, se qualcuno invece detiene il lucchetto facendo l'acquire ci mettiamo in attesa, prima o poi sperabilmente la release verrà fatta.

```
char *malloc (n) {
  heaplock.acquire();
  p = allocate memory
  heaplock.release();
  return p;
}
```

```
void free(char *p) {
  heaplock.acquire();
  put p back on free list
  heaplock.release();
}
```

6

Regole di utilizzo della lock

Le lock sono variabili e dunque devono essere inizializzate, di solito sono inizializzate a free (lucchetto aperto). Devo sempre fare l'acquire prima di accedere ad un dato che è condiviso da altri, l'accesso può essere sia di lettura che scrittura, bisogna sempre riaprire il lucchetto quando abbiamo finito con il dato condiviso.

Buon stile di programmazione vuole che non si faccia la lock su un thread e la unlock su un altro thread di uno stesso lucchetto.

```
[…]
if (p == NULL) {        // No! This check is mistaken! Why?
  lock.acquire();
  if (p == NULL) {
    p = newP();
  }
  lock.release();
}
use p->field1
```

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

[6] Nota questa è un'implementazione semplificata delle funzioni malloc e free, non corrispondente alla realtà.

Questo codice è sbagliato perché P viene letto prima di fare l'acquire, potrei trovare p == NULL ed eseguire il codice, nel frattempo, un altro thread potrebbe leggere il valore di p e trovarlo non più a null, per poi usarlo, questo non significa però che sia stato inizializzato, il primo thread potrebbe infatti venir de-schedulato subito dopo la malloc.

Le lock risolvono il problema di avere codice simmetrico, proteggendo la parte di codice dove si usano variabili condivise.

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

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

Buffer circolare, con un indice di testa e un indice di coda, entrambi all'inizio 0, c'è una variabile salvata come FREE, quando faccio tryget() se non ci sono elementi nel buffer ritorno NULL, subito dopo quindi riproveremo, se due thread fanno contemporaneamente tryget, il primo acquisirà mutua esclusione, il secondo sarà in attesa, per cui la lock mi garantisce che per la sezione critica non faccio attesa attiva, non me lo garantisce però su tutto il codice.

Variabili di condizione

Le variabili di condizione o condition variables ci permettono di attendere all'interno di una sezione critica rilasciando la lock, dando la possibilità quindi di acquisire il lucchetto e di svegliarci quando il lucchetto è stato rilasciato.

È fondamentale associare sempre ad una variabile di condizione una variabile di mutua esclusione, le variabili di condizione non hanno senso senza le variabili di lock.

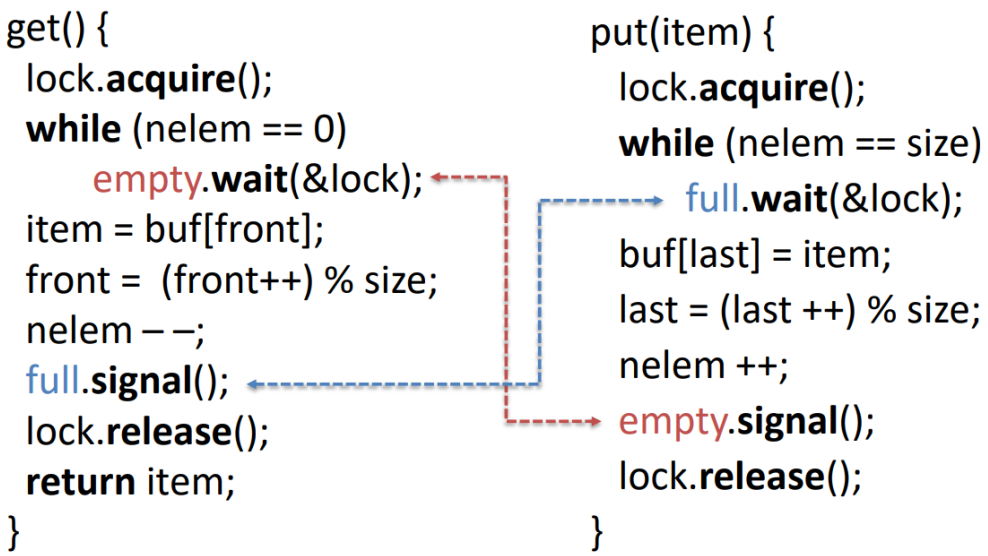
Le operazioni definite sulle variabili di condizione sono:

Wait: Atomicamente rilascia la lock associata e sospende il thread sulla variabile di condizione, serve quindi una variabile di lock chiusa, altrimenti non funziona.

Signal: Sveglia, se ce ne sono, uno tra i thread in attesa

Broadcast: Sveglia, se ce ne sono, tutti i thread in attesa

Example: Bounded Buffer



Ho una risorsa condivisa il buffer, associato come front e last, la risorsa è il bounded buffer ovvero un buffer di capacita limitata di *nelem* posizioni, il buffer può essere vuoto o pieno.

Il problema produttore consumatore, prevede che i produttori depositino i dati in un buffer condiviso, i consumatori prelevino i dati dallo stesso buffer, i prodotti possono essere estratti solo una volta.

Possono esistere delle varianti in base alla capacità del buffer.

Riusciamo ad implementare questo modello utilizzando una variabile di lock, e due variabili di condizione perché abbiamo due eventi nel caso di capacità limitata, buffer pieno e buffer vuoto.

Posso sospendermi sia in attesa che ci sia un dato nel buffer, oppure sospendermi se sono il produttore e ho trovato il buffer pieno in attesa che un consumatore utilizzi un elemento.

È necessario avere un while per la gestione di errori di schedulazione, questo in generale vale per tutte le variabili di condizione, vanno infatti sempre testate con un while in quanto se avessi solo un if, potrei magari non superare la condizione e passare all'istruzione successiva al corpo del blocco if.

Tra le operazioni di variabili di condizione oltre alla wait e alla signal, c'è la broadcast che sveglia tutti i thread che sono in attesa di quella particolare condizione.

La wait prende la variabile di lock per riferimento, in quanto modifica atomicamente la lock, rilasciandola e mettendo il processo in attesa.

Come si dimostra che questa implementazione rispetta i criteri di safety e liveness?

Per fare questa dimostrazione dovremmo considerare tutti i possibili casi.

Le variabili di condizione sono sempre accoppiate ad una variabile di lock, esempio la wait.

La wait e la signal devono essere invocate sempre con la lock acquisita, con la signal potremmo obiettare dicendo che potrebbe essere messa fuori, se disaccoppiamo la segnalazione e il rilascio della lock però potrebbero esserci dei casi in cui non funziona.

Le variabili di condizione sono senza memoria, non hanno uno stato interno, non sappiamo quante signal sono state fatte, se le signal non svegliano nessuno si perdono, è come se non fossero mai avvenute. I semafori invece hanno memoria, hanno infatti un contatore interno.

Questa è una delle differenze principali tra le variabili di condizione e i semafori.

Il rilascio della lock e la sospensione è atomico.

Se ci sospendiamo senza rilasciare la lock nessuno potrà più accedere alla sezione critica, se rilasciassimo la lock e ci sospendessimo con due operazioni separate potrebbe succedere che, dopo aver rilasciato la lock potrei venir de-schedulato e potrebbe arrivare un signal che io non sento perché non ho ancora fatto la wait, questo (signal) dunque si perderebbe e quando arriverei a fare la wait potrei non svegliarmi più perché quello sarebbe potuto essere l'unico signal, è necessario quindi che le operazioni siano atomiche.

Quando un thread viene svegliato da una signal o da una broadcast il thread deve acquisire la lock, quando viene svegliato non la acquisisce subito, ma viene messo in coda-pronto per essere eseguito, quando risveglio un thread quello non andrà subito in esecuzione, questo vuol dire però che quando il produttore rilascia la lock, quello che è arrivato dopo di me potrebbe passare prima in esecuzione, il fatto che io venga svegliato non significa che io sarò il prossimo a consumare il dato del buffer.

Dobbiamo, è obbligatorio, sempre testare la wait in un while.

Spurius wait

È possibile che sotto certe condizioni ci siano dei wake up spuri, ovvero che il thread venga risvegliato anche se sarebbe dovuto rimanere silente, se non ritesto la condizione potrei assumere erroneamente che la condizione sia vera e il programma potrebbe fallire.

Regole di utilizzo

Acquisire la lock sempre prima di entrare nella sezione critica e rilasciarla alla fine.

Acquisire le lock quando usiamo variabili di condizione.

Usare sempre un while loop per testare la variabile di condizione.

Evitare di fare attesa passiva con le sleep per sincronizzare approssimativamente i processi.

Mesa vs. Hoare semantics

Esistono due semantiche diverse per le variabili di condizione: la semantica Mesa e Hoare

Quella descritta fino ad adesso è la semantica Mesa che è quella più usata.

Si usa un'altra semantica perché è una piccola variante con un impatto però significativo, con le variabili di condizione quando risvegliamo un processo in attesa, dato che vogliamo ri-acquisire la lock, potrebbe succedere che qualcuno ci scavalchi, è come se io fossi in coda e arrivasse qualcuno che ci passasse davanti, il problema non c'è se a passarci d'avanti fosse uno che era in attesa, è un problema magari se ci passa davanti qualcuno che si è appena svegliato. Le variabili di condizione per come le abbiamo descritte ora non c'è garanzia che rispettino l'ordine fifo, per implementare un consumo fifo del buffer dobbiamo usare la semantica di Hoare

Cambiando la semantica della signal riusciamo a fare un'implementazione fifo senza cambiare il codice, quando facciamo la signal invece di segnalare e basta segnaliamo

passando la lock, così che chi viene risvegliato ha già la lock, e non la deve riacquisire, sicuramente se gli passo la lock non potrà essere scavalcato da un altro.

FIFO Bounded Buffer (Hoare semantics)

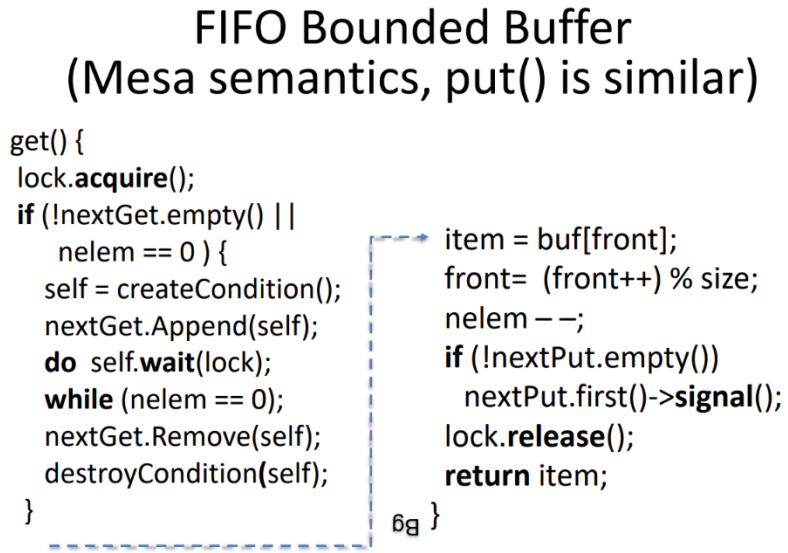
```
get() {
  lock.acquire();
  while (nelem == 0) // if also works
        empty.wait(&lock);
  item = buf[front];
  front=  (front++) % size;
  nelem – –;
  full.signal(&lock);
  lock.release();
  return item;
}
```

```
put(item) {
  lock.acquire();
  while (nelem == size)
        full.wait(&lock);
  buf[last] = item;
  last = (last ++) % size;
  nelem ++;
  empty.signal(&lock);
  // CAREFUL: someone else ran
  lock.release();
}
```

La semantica è uguale all'altro caso che abbiamo visto, anche se il segnalante che prima non prendeva la lock, ora la prende e la passa logicamente a chi verrà svegliato, non solo quindi lo metto in coda pronti, lo metto in coda pronti dicendogli tu sei il detentore della lock, tutto questo viene fatto in modo atomico. Quando il processo svegliato ha finito, farà lock release che ritornerà il controllo al chiamante, il chiamante eseguirà quindi un ultimo pezzo di codice che si troverà tra la signal e la release.

Dobbiamo stare attenti che quando torniamo sicuramente qualcuno avrà cambiato lo stato.

Come implementare la fifo con il Mesa?



Vediamo la get, usiamo una coda (nextGet) su cui andrò a mettere variabili di condizione.

Coda implementata in modo fifo, se la coda è vuota oppure non ci sono elementi nel buffer, anziché sospendermi, creo una variabile di condizione privata, su questa variabile di condizione ci sospendiamo, Verrò risvegliato da qualcuno che fa la signal, quando deve fare la signal anziché farla controlla la coda.

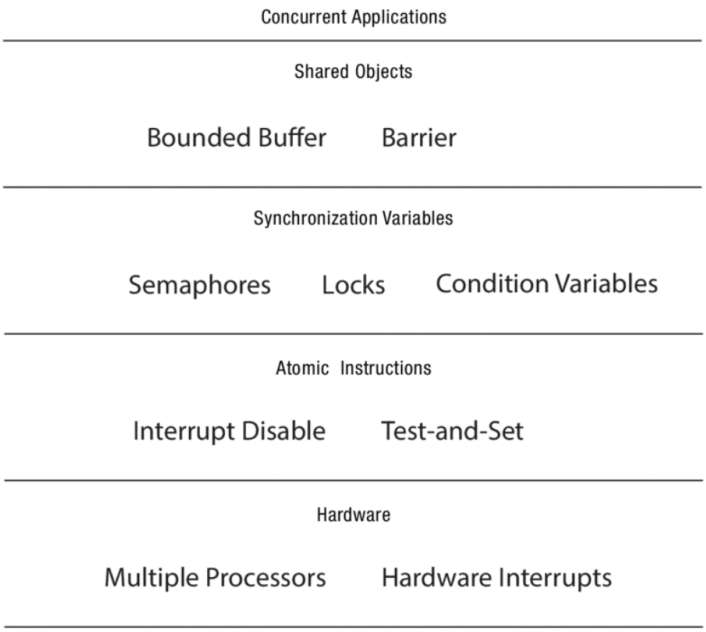
Se c'è qualcuno in attesa estrae dalla coda la variabile di condizione e farà la signal su quella variabile di condizione, sono quindi sicuro che verrà estratto il primo che aveva fatto la wait.

Dov'è l'intoppo? Anziché utilizzare una sola variabile di condizione ne abbiamo N, N per i consumatori e M per i produttori.

Questo schema è uno schema generale, se volessi cambiare schema basta cambiare il sistema di gestione della coda.

Si parla di self-trick in quanto creiamo una variabile di condizione privata.

Sincronizzazione nel SO



Anche il So al suo interno è concorrente e deve usare dei sistemi di sincronizzazione.

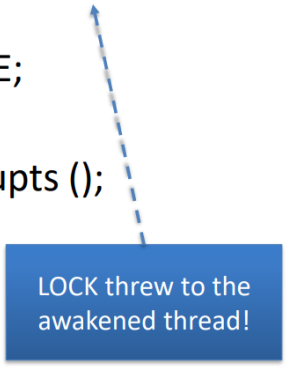
Abbiamo diverse possibilità:

- Totalmente con load/store
- Tramite abilitazione/disabilitazione delle interruzioni (a patto che io sia su un sistema uni-core)
- Implementazione che utilizza le spin-lock

Uniprocessor

```
LockAcquire() {
  disableInterrupts ();
  if (value == BUSY) {
    waiting.add(myTCB);
    suspend(); *
  } else {
    value = BUSY;
  }
  enableInterrupts ();
}
* Invokes the scheduler,
  context switch & enables
  interrupts
```

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



Nella lock acquire disabilitiamo le interruzioni, così non saremo interrotti da niente, guardiamo il valore della lock (0 libera e 1 occupata o viceversa) se è occupato io che ho invocato la lock mi devo sospendere. Prendo il contesto, lo metto in stato di attesa nel tcb salvando i registri e chiamando la chiamata suspend, che chiamerà eventualmente lo scheduler che a sua volta chiamerà il content-switch per mettere il contesto di un altro thread in esecuzione. Eseguendo la I_ret verranno riabilitate le interruzioni.

A questo punto quando qualcuno mi sveglia riabilito le interruzioni prima di uscire, se la lock era libera, metto la lock come occupata e abilito le interruzioni.

Multiprocessor

Abbiamo bisogno di operazioni atomiche di tipo readModifyWrite, dobbiamo capire come poter combinare un'operazione di lettura e di scrittura in modo atomico, stiamo chiedendo supporto architetturale, noi di questa classe di istruzioni ne vedremo una sola, la test and set ma ne esistono diverse, sono istruzioni assembler non privilegiate, possono essere chiamate in user mode o in kernel mode, il SO utilizza queste op per implementare la spin lock che è il meccanismo che viene utilizzato per garantire mutua esclusione sul multi processore.

Spin Lock

È una lock senza attesa passiva, spin significa ciclare sostanzialmente, il concetto è analogo a quello delle lock, solo che nella lock c'è una coda in cui chi chiama se trova la lock acquisita si sospende e si mette in coda di attesa sulla coda associata alla lock.

La spin lock non ha una coda ma solo una locazione che viene testata fino a quando il valore di quella locazione cambia, stiamo re-introducendo attesa attiva perché è necessaria.

L'implementazione è semplice, chiamo in un ciclo while testAndset e finchè è occupata continuo a ciclare, quando la trovo libera entro e la acquisisco, per rilasciarla la setto a free e chiamo una memory barrier.

Il trucco per capire come funziona una spin lock è capire come funziona un'istruzione readModifyWrite, come la testAndset.

```
TSL REG, &ADDR    // REG ← MEMORY[ADDR]
                  // MEMORY[ADDR] ← #BUSY
```

La test and set è un'operazione nativa assembler, in questo caso tsl (test and set lock) concettualmente prima scrive nel registro il valore contenuto in memoria all'indirizzo fornito dal secondo argomento e poi nell'indirizzo individuato dal secondo argomento scrive busy.

Se c'era busy, lo leggo e sovrascrivo busy quindi continuerò a ciclare, se c'era free significa che qualcuno ha fatto una release quindi la test and set passerà.

```
// &spinLockValue is a memory cell containing a binary value: FREE (0) or BUSY
// TSL R, &spinLockValue :
    writes the content of &spinLockValue in R and writes BUSY (1) in
    &spinLockValue


spinlockAcquire(&spinLockValue) {
        Loop:   TSL R, &spinLockValue
                CMP R, #BUSY
                BEQ Loop          // jump if last comparison was successful
                END               // at this point &spinLockValue == BUSY!!!!
}
spinlockRelease(&spinLockValue) {
    MOV #FREE, &spinLockValue     // this unlocks a thread in the loop, if any
    MFENCE                        // memory barrier
}
```

La spinLock non ha nulla a che fare con la lock del livello applicativo.

Se cicliamo, cicliamo per pochissimo tempo, la spinlock serve per proteggere la struttura dati waiting e per fare sched_suspend e basta.

Se poi chi ha acquisito la lock la detiene per un quarto d'ora è un problema dell'applicazione, non del SO.

Vogliamo accodare il tcb ad una coda di attesa che è associata ad una variabile di lock, la variabile di lock in C potrebbe essere una struttura con due campi:
Un value e una coda waiting, su questa coda ci mettiamo i thread in attesa, in particolare mettiamo i tcb dei thread che hanno fatto la lock e che devono essere sospesi.

Implementazione delle spinlock in ARM

```
static inline
void arch_spin_lock(arch_spin_lock *lock){
  unsigned long tmp;
  __asm__ __volatile__(
      "1:     ldrex   %0, [%1]\n"
      "       teq     %0, #0\n"
      WFE("ne")
      "       strexeq %0, %2, [%1]\n"
      "       teqeq   %0, #0\n"
      "       bne     1b"
      : "=&r" (tmp)
      : "r" (&lock->lock), "r" (1)
      : "cc");
  smp_mb();
}
```

```
static inline
void arch_spin_unlock(arch_spinlock_t *lock){
  smp_mb();
  __asm__ __volatile__(
      "       str     %1, [%0]\n"
      :
      : "r" (&lock->lock), "r" (0)
      : "cc");
  dsb_sev();
}
```

Dopo la ldr e str ex sta per exclusive, queste operazioni infatti fanno la load/store di quello che devono fare però il fatto che sia ex significa che blocca l'arbitro di memoria e dice guarda che finchè non ti arriva un'altra ex non devi ascoltare nessuna altra richiesta ne da altri core ne da nient'altro, le prime 4 istruzioni implementano la mutua esclusione, la prima load la avvia la str la termina, questa garanzia di atomicità è data sulla memoria e non sul processore. Questa cosa è fatta ovviamente ad interruzioni disabilitate.

Nella spin_unlock invece abbiamo solamente una store di quanto di per se è atomica.

La sospensione delle lock costa, implica content switch. In linux, ma in molti SO si utilizza una tecnica che non approfondiremo, è una tecnica di ottimizzazione.

Dato che il sistema sa che la lock è costosa mentre la spinlock se dura poco no, le implementazioni reali utilizzano una tecnica che si chiama fastpath-slowpath.

Prima implementano la lock facendo una spinlock, eseguendo quest'ultima per pochi cicli, se sono fortunato entro quei pochi cicli acquisiremo la lock, se siamo sfortunati cadiamo nello slowpath e andiamo nella lock classica dove dobbiamo sospenderci.

È in realtà molto difficile da implementare, i sistemi utilizzano un sistema adattivo.

Ha senso però fare un'implementazione del genere perché di solito le lock proteggono sezioni di programma brevi.

Le spinLock a livello applicativo sono utili solo se le operazioni sono molto brevi.

Semafori

Un semaforo è un meccanismo di sincronizzazione, ha memoria, combina insieme una lock e una variabile di condizione, tanto è vero che i semafori si usano sia per la mutua esclusione che per l'attesa passiva, è un contatore non negativo e una coda di attesa, una lock era una variabile booleana e una coda di attesa.

Sul contatore possono essere effettuate due operazioni P e V, P per decrementare il contatore del semaforo a patto che il contatore sia >0, se il contatore è 0 il thread si sospende sulla coda del semaforo, V per incrementarlo.

Le operazioni avvengono dal punto di vista logico in modo atomico.

Implementazione nel SO

```
P(sem){
  disableInterrupts();
  spinLockAcquire(&spinLock);
  if (sem.value == 0){
    waiting.add(myTCB);
    suspend(&spinLock);
  } else {
     sem.value --;
     spinLockRelease(&spinLock);
  }
  enableInterrupts();
}
```

Invokes scheduler; context switch; enables interrupts

```
V(sem) {
  disableInterrupts();
  spinLockAcquire(&spinLock);
  if (!waiting.Empty()){
     thTCB = waiting.Remove();
     readyList.Append(thTCB);
   }
  else sem.value ++;
  spinLockRelease(&spinLock);
  enableInterrupts();
}
```

Throws semaphore to the awakened thread!

Concettualmente non è molto diversa dall'implementazione delle lock, tranne per il fatto che in questo caso dobbiamo gestire uno stato, rappresentato da un contatore intero.

Il meccanismo è il solito, disabilito le interruzioni per il singolo core e utilizzo una spinlock per proteggere il dato del SO del semaforo.

Semaforo bounded buffer

Servono 3 semafori, un semaforo per fare la lock, che chiamiamo mutex, un semaforo per sospenderci nella condizione se è vuota e un semaforo per sospenderci nella condizione se è piena.

I semafori vanno inizializzati:

La empty va inizializzato a 0 (indica gli elementi presenti)

Il Full va inizializzato a n (indica i posti liberi)

Il mutex va inizializzato a 1 (variabile di lock libera)

```
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();
}
```

Get()

Se il buffer non è vuoto, qualcuno avrà prodotto qualcosa quindi il contatore del sem è > 0, devo acquisire la lock (tramite mutex), consumo, faccio la V metto la lock a libera e incremento il valore del semaforo full segnalando eventualmente che un altro produttore può inserire valori nel buffer.

Posso scambiare empty.P e mutex.P?

No perché se mi sospendo prima di rilasciare la lock e magari il buffer è vuoto non potrò mai riempirlo.

Implementare una variabile di condizione usando i semafori

Bisogna implementare la wait, la wait deve rilasciare la lock e sospendere il processo/thread che la chiama, quando verrà svegliata riacquisirà la lock ecct…

Potremmo usare un solo semaforo, facendo la P nella wait e mettendomi in attesa, prima o poi qualcuno farà una signal su questa var di condizione, verrà fatta la V e verrò svegliato, questa implementazione però non è corretta in quanto non rispetta la semantica delle variabili di condizione.

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

```
signal() {
  sem.V();
}
```

Che succede se si fa prima la signal che la wait?

Dovrebbe perdersi ma qua non si perde perché i semafori hanno memoria, le variabili di condizione non dovrebbero avere memoria.

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

```
signal() {
  if semaphore not empty
    sem.V();
}
```

Nella signal prima testo il semaforo, se è non vuoto significa che c'è qualcuno sospeso, dunque lo sveglio altrimenti vado avanti, anche questo però non funziona in quanto non rispetta la semantica delle var di condizione.

Noi vogliamo che la wait atomicamente rilasci la lock e sospenda il thread che l'ha chiamata. In questo caso la release e la sem.P sono due operazioni distinte, potrei avere una de-schedulazione tra queste due istruzioni.

```
wait(lock) {
  selfsem =
    createSemaphore();
  // queue of waiting threads
  queue.Append(selfsem);
  lock.release();
  sem.P();
  destroySemaphore(selfsem);
  lock.acquire();
}
```

```
signal() {
  if !queue.Empty() {
     seflsem = queue.Remove();
     selfsem.V();    // wake up waiter
  }
}
```

In questa soluzione usiamo il self-trick per le variabili di condizione, in quel caso era per garantire l'ordinamento fifo, in questo caso invece lo possiamo usare per eseguire le operazioni atomiche.

Quando eseguo la wait sono in mutua esclusione, le prime operazioni vengono quindi eseguite con una lock presa, mi creo un semaforo privato, questo semaforo lo metto in una coda che serve ad implementare una variabile di condizione, finito questo rilascio la lock e quindi termino la mutua esclusione e mi metto in attesa sul semaforo condiviso.

La signal verrà chiamata con la lock presa, dunque il controllo della coda verrà eseguito in mutua esclusione, la coda sicuramente non sarà empty, estraggo il semaforo e lo incremento.

Una volta finita la wait distruggo il semaforo privato e acquisisco la lock.

Sincronizzazione tramite monitor

È il concetto dei synchronized object di java, è un costrutto primitivo dei linguaggi di programmazione, un monitor è un tipo di dato astratto fornito al programmatore per effettuare sincronizzazioni in modo più semplice rispetto alle lock e alle condition variables, è un meccanismo più semplice per la sincronizzazione.

Un monitor è una struttura dati che incapsula i dati che voglio proteggere, però l'accesso a questi dati è mediato sempre tramite una chiamata di un metodo, ovvero da una funzione, dobbiamo per forza invocare un metodo del monitor e non possiamo accedere ai dati direttamente.

Dal nostro punto di vista un monitor non è altro che zucchero sintattico, aggiunge della sintassi per rendere l'implementazione della sincronizzazione più semplice, i problemi rimangono ma sono interni al monitor ed è dunque compito di chi sviluppa la libreria risolverli.

Se più thread chiamano lo stesso metodo ci pensa il monitor a far passare l'uno o l'altro noi non ci dobbiamo preoccupare di dover aggiungere ulteriore sincronizzazione.

Lettori e scrittori

Come nel caso produttore consumatore, il problema dei lettori e scrittori è la modifica della struttura dati a cui accediamo concorrentemente.

Possiamo avere in sezione critica più lettori contemporaneamente dato che non la modificano, ma non più scrittori.

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

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

Il lettore ce lo possiamo immaginare che esegue un ciclo infinito in cui prima di entrare nella sezione critica lo dichiara, fa quello che deve fare e prima di uscire lo dichiara.

La stessa cosa la fa lo scrittore, dichiariamo il ruolo invocando o startRead o startWrite.

Il codice tra start e done sarà la nostra sezione critica.

Prima Soluzione (non fair per scrittori)

Vogliamo garantire l'accesso alla sezione critica in ordine di arrivo, questa soluzione non garantisce la proprietà di fairness, la priorità viene infatti data ai lettori, finchè ci sono lettori li fa entrare, se nella sezione critica c'è uno scrittore i lettori aspettano, altrimenti hanno la priorità, un potenziale scrittore in coda se arrivano sempre lettori in continuazione non verrà mai servito, questa situazione si chiama starvation.

Per quanto riguarda il rilascio, quando uno scrittore rilascia la mutua esclusione tutti i lettori in attesa vengono risvegliati, se non ci sono lettori ma c'è almeno uno scrittore questo viene fatto iniziare, se non ci sono né lettori né scrittori torniamo al caso 1, quando un lettore rilascia la struttura dati se ci sono altri lettori che la stanno utilizzando continuano, se non ce ne sono altri, ed esiste almeno uno scrittore in attesa il primo di questi ottiene l'accesso.

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: utilizzata per la sospensione dei Lettori;
- Cond writeGo: utilizzata per la sospensione degli Scrittori;

Lettore:

```
// startRead()
    mutex.Acquire();
    waitingReaders++;
    while (activeWriters > 0 ) {
          readGo.Wait(&mutex);
    }
    waitingReaders --;
    activeReaders++;
    mutex.Release();

<legge>
```

```
// doneRead()

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

Scrittore:

```
// startWrite()

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

<scrive>
```

```
// doneWrite()

    mutex.Acquire();
    activeWriters--;
    if (waitingReaders > 0)
      readGo.Broadcast();
    else
       writeGo.Signal();
    mutex.Release();
```

Nel metodo startRead quando entra un lettore vuole prendere la mutua esclusione, questa ci serve per acquisire le strutture dati per implementare le nostre soluzioni, incrementiamo il contatore lettori in attesa e se ci sono scrittori in attesa mi sospendo, anche qua serve un ciclo while perché qualora venissi svegliato devo controllare che non ci siano scrittori in sezione critica, se sono nella sezione critica decremento il numero di lettori sospesi e aumento quello di lettori attivi, per uscire rilascio poi la mutex.

Quando uno dei lettori esce dopo aver acquisito la mutua esclusione, diminuisco il numero di lettori attivi, se sono l'ultimo lancio un segnale e poi rilascio il mutex.

Vediamo adesso una variante non fair per i lettori, è una soluzione simmetrica molto utilizzata, se ci sono scrittori quando escono lo scrittore da precedenza agli altri scrittori in attesa, se continuano sempre ad arrivare scrittori, i lettori potrebbero starvare (rimanere in attesa indefinita).

Seconda soluzione (non fair per i lettori)

Lettore:

```
// startRead()
    mutex.Acquire();
    waitingReaders++;
    while (activeWriters > 0 ||
            waitingWriters > 0)
      readGo.Wait(&mutex);
    waitingReaders --;
    activeReaders++;
    mutex.Release();

<legge>
```

```
// doneRead()
    mutex.Acquire();
    activeReaders--;
    if (activeReaders ==0 &&
       waitingWriters > 0)
          writeGo.Signal();
    mutex.Release();
```

Scrittore:

```
// startWrite()

    mutex.Acquire();
    waitingWriters++;
    while (activeWriters > 0 ||
            activeReaders > 0)
       writeGo.Wait(&mutex);
    waitingWriters --;
    activeWriters++;
    mutex.Release();

<scrive>
```

```
// doneWrite()

    mutex.Acquire();
    activeWriters--;
    if (waitingWriters > 0)
      writerGo.Signal();
    else  readerGo.Broadcast();
    mutex.Release();
```

Vediamo adesso l'implementazione fair, non necessariamente fifo, questa soluzione fornisce un ordinamento nell'accesso ma non è detto che sia fifo, l'approccio è il solito, startRead doneRead, startWrite doneWrite.

Il problema viene risolto garantendo una politica fair di accesso alla struttura dati condivisa consentendo ai lettori di accedere concorrentemente.

Rispetto alle due soluzioni precedenti:

• Se un lettore richiede l'accesso ma vi è almeno uno scrittore in attesa di accedere, il lettore richiedente viene sospeso (questa clausola evita l'attesa indefinita per gli scrittori).

• L'ultimo dei lettori che rilascia la struttura dati fa entrare l'eventuale scrittore in attesa.

• Quando uno scrittore rilascia la struttura dati, fa entrare il prossimo in attesa (sia esso lettore e/o scrittore).

• Il lettore in attesa sbloccato dallo scrittore sblocca a suo volta l'ingresso all'eventuale altro lettore in attesa dopo di lui.

Fair significa che non privilegiamo né i lettori ne gli scrittori, utilizziamo una variabile extra di mutua esclusione che non usiamo per proteggere una sezione critica bensì per dare un ordinamento, prima dobbiamo passare da ordering-Acquire(), quando arrivo in un lettore ordering-Acquire() all'inizio è libera, la prima volta che acquisisco questa ordering è tutto libero, non ci sono scrittori, infine rilascio la ordering così che chi eventualmente si era

bloccato nella coda cerca di fare progresso e di andare avanti, il caso interessante è il passaggio a ordering, se sono un lettore e c'è uno scrittore devo sospendermi senza rilasciare però ordering, quando invece la mutex viene rilasciata, ordering e mutex sono diverse, la prima rimane linkata, questo fa si che tutti quelli che arriveranno saranno bloccati in questa ordering, tentando di fare ordering acquire, la coda non si sa come è gestita, dunque non sappiamo il reale ordinamento, in quanto è dato dall'implementazione della coda.

```
// startRead()

    ordering.Acquire();   // uno per volta
    mutex.Acquire();
    while (activeWriters > 0)
          Go.Wait(&mutex);  // (*)
    activeReaders++;
    ordering.Release(); // avanti il prossimo
    mutex.Release();
```

```
// doneRead()

    mutex.Acquire();
    activeReaders--;
    if (activeReaders ==0)
         Go.Signal();
    mutex.Release();
```

<legge>
(*) il lettore che si sospende non rilascia ordering ma solo mutex

```
// startWrite()

    ordering.Acquire(); // uno per volta
    mutex.Acquire();
    while (activeReaders > 0 ||
           activeWriters > 0)
       Go.Wait(&mutex);   // (*)
    activeWriters++;
    ordering.Release(); // avanti il prossimo
    mutex.Release();
```

```
// doneWrite()

    mutex.Acquire();
    activeWriters--;
    Go.Signal();
    mutex.Release();
```

<scrive>
(*) lo Scrittore che si sospende non rilascia ordering ma solo mutex

Multi-Object Synchronization

Nei sistemi operativi abbiamo risorse multiple, e necessitiamo di acquisirle contemporaneamente, abbiamo dei processi che richiedono l'accesso a più risorse magari in più copie, quello che succede è che abbiamo bisogno di organizzare gli accessi alle risorse in modo che si evitino situazioni di stallo, ovvero situazioni in cui più entità/processi richiedono risorse e si induce una situazione di attesa reciproca tipicamente circolare, che può portare al blocco del sistema.

Come ci sincronizziamo quando abbiamo oggetti multipli?

Vogliamo evitare il deadlock ovvero l'attesa circolare di risorse.

Le soluzioni possono essere azioni preventive, ovvero cerchiamo di evitare di trovarci in una situazione di stallo, oppure azione correttive, per risolvere uno stallo.

Questi problemi si hanno soprattutto in grandi programmi, in cui ogni oggetto ha una sua lock e una sua variabile di condizione.

Dobbiamo cercare di evitare situazioni di deadlock senza influire né la semantica né la correttezza dei nostri programmi.

Definizioni

Come risorsa definiamo un qualcosa che possiamo utilizzare nel nostro sistema, come ad esempio un core della cpu, la memoria, oggetti condivisi, tutto quello che possiamo pensare di usare nel SO è una risorsa. Noi vogliamo controllare se una risorsa è rilasciabile o no, ovvero vogliamo sapere se la risorsa può essere reclamata dal SO anche se è in uso da un certo processo/thread (preemptable) oppure no (non-preemptable).

Per come visti sino ad ora, un thread prende una risorsa e se la tiene finché non la rilascia.

Per starvation intendiamo quando abbiamo dei thread che attendono una risorsa che non arriverà mai, nessuno riesce ad accedere alla risorsa, sono tutti in attesa.

Abbiamo la situazione di deadlock in cui abbiamo un'attesa circolare delle risorse, vorremmo andare avanti, ma necessitiamo di una risorsa che è detenuta da qualcun altro e colui che detiene la risorsa è esso stesso in attesa di un'altra risorsa.

Il deadlock è una situazione di starvation, ma la starvation non necessariamente è un deadlock.

Assunzioni

Consideriamo che le risorse siano riutilizzabili, ovvero se sto usando una sezione critica, nel momento in cui mollo la risorsa questa è riutilizzabile da altri thread.

Dobbiamo distinguere nell'uso delle risorse: la richiesta, l'assegnamento e il rilascio della risorsa.

La richiesta è l'acquire, se passo ad avere la risorsa l'uso delle risorse è assegnato, dopo averla utilizzata faccio la release rilasciando la risorsa.

Vogliamo avere dei meccanismi che ci permettano se la risorsa non è disponibile di bloccarci, questo vogliamo realizzarlo tramite i meccanismi visti sino ad ora.

Esempio

Il thread_A prende correttamente la lock1, il thread_B prende correttamente la lock2, si crea però una situazione di deadlock quando il thread_A prova ad acquisire la lock2, che però è usata dal thread_B, lo stesso il thread_B tenta di acquisire la lock1.

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

Anche con lock e variabili di condizione la situazione non è molto diversa, entrambi fanno la lock acquire di 1 e poi 2, quando fanno la wait e controllano la condizione, dobbiamo aspettare che la lock2 venga rilasciata, cosa che però non accadrà, in quanto non riesco mai a fare la lock1 acquire.

```
Thread A                          Thread B

lock1.acquire();                  lock1.acquire();
…                                 …
lock2.acquire();                  lock2.acquire();
while (need to wait)              ….
  condition.wait(lock2);          condition.signal();
lock2.release();                  lock2.release();
…                                 …
lock1.release();                  lock1.release();
```

Condizioni per deadlock

Le condizioni che portano ad un deadlock sono delle condizioni ben precise:

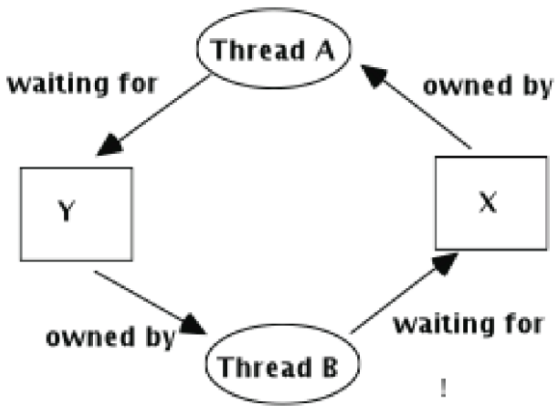
Abbiamo infatti un accesso limitato alle risorse, un numero limitato di thread utilizza una risorsa a cui tipicamente è assegnato un mutex, se avessimo infinite risorse non avremmo situazioni di deadlock.

Il secondo problema è il rilascio, su una cosa che è stata assegnata posso dire (io SO) che la prendo e la riassegno. Se avessi la possibilità di fare preemption lo stallo non si creerebbe.

Wait while holding, durante l'attesa ci teniamo le risorse che abbiamo e non le rilasciamo, se ho una situazione di stallo vuol dire che chi ha acquisito delle risorse non le rilascia mentre tenta di acquisirne delle altre.

Devo avere un sistema circolare di richieste, ovvero chi richiede delle risorse le richiede a qualcuno che a sua volta sta richiedendo delle risorse.

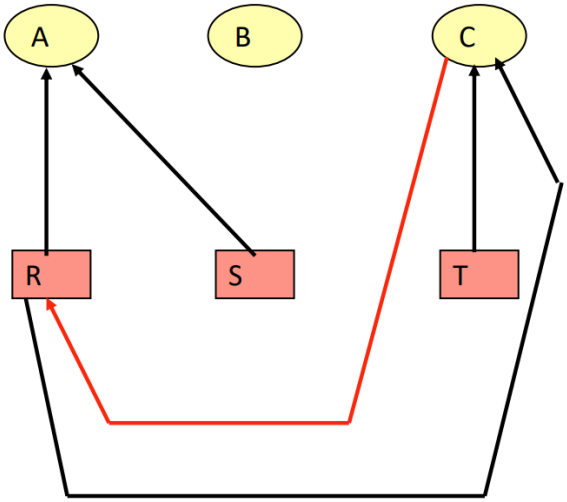
Attesa circolare



Esempio che non porta a deadlock

A requests R
C requests T
A requests S
C requests R
A releases R
A releases S
…

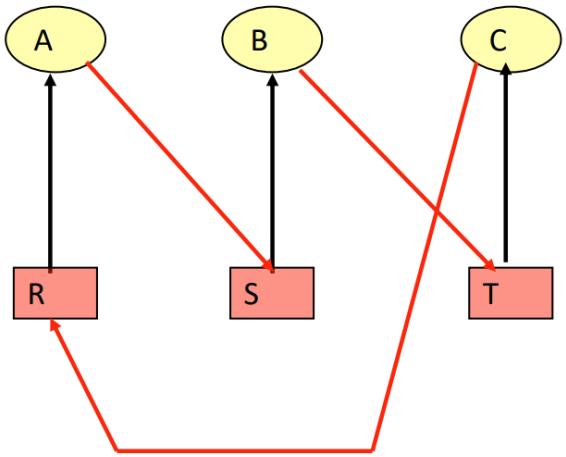
hyp: A needs only R and S
C needs only T and R



Esempio che porta a deadlock

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

Deadlock!



Gestione dei Deadlock

Metodi per la gestione di Deadlock

3 tipi:

- Cerchiamo di accorgerci di una situazione di stallo e in qualche modo cerchiamo di risolverla.
- Prevenzione statica, evitiamo che una situazione di stallo si verifichi, ci serve sapere in che ordine i thread eseguono le operazioni.
- Prevenzione dinamica, mentre stiamo eseguendo delle azioni non eseguiamo quelle che potrebbero portare ad uno stallo.

In molti casi questo il SO non lo fa, in quanto ognuna di queste operazioni ha un costo, si fa affidamento a chi scrive il programma concorrente.

L'unica azione che il SO unix fa è quella di:

Se andiamo ad esaurire le risorse di sistema ne rimane almeno una in mano al superutente che è in grado di terminare i processi.

Soluzioni

1° Soluzione: Detect and fix

Cerchiamo di scoprire se c'è una situazione di deadlock e la risolviamo a posteriori, potremmo avere un algoritmo che utilizza un grafo di richieste e risorse, e controlliamo se esiste un ciclo nel grafo, troviamo poi un modo per risolvere questo ciclo, due modi generali per fare questo sono: quello di prendere due entità concorrenti, eliminarne una riassegnando le risorse, oppure fare un roll back del thread facendogli rilasciare le risorse.

Soppressione di un thread: è molto semplice anche se aggressiva per il sistema, ci servono dei criteri per la selezione del thread da eliminare, vogliamo infatti avere il minimo impatto sul sistema.

Roll-Back: facciamo tornare sui suoi passi un thread, ripristiniamo uno stato precedente del processo, non modifichiamo il codice ma rallentiamo solamente la sua esecuzione sperando che arrivando al punto di deadlock precedente ora non si verifichi più. Questo richiede che esistano dei checkpoint ovvero dei punti in cui salvare lo stato del processo per poi ripristinarlo, questa è però un'operazione costosa.

2° Soluzione: Deadlock prevention

La cosa migliore sarebbe prevenire che un deadlock si verifichi, per la prevenzione possiamo cercare di fare delle azioni che rimuovano una delle condizioni che portano alla creazione di deadlock. Per eliminare una di queste condizioni potremmo fare un ordinamento tra richieste e risorse, se ponessimo un ordinamento sul numero di lock, magari il programma sarebbe meno efficiente, ma si eviterebbero situazioni di attesa incrociata.

Questo meccanismo è molto utilizzato nell'implementazione degli OS kernel.

Stiamo in qualche modo allungando il tempo di esecuzione, in quanto richiediamo una risorsa prima ancora di utilizzarla, in questo lasso di tempo la risorsa potrebbe essere usata da qualcun altro che però potrebbe condurci in deadlock.

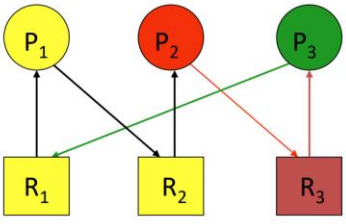
Potrei fare, che, se cerco di acquisire una risorsa che non è disponibile in quel momento, rilascio tutte le risorse e riprovo più tardi, dobbiamo prestare attenzione a come rilasciamo le risorse nel caso in cui siano state modificate.

Possiamo sennò intervenire su quelle risorse che sono limitate.

Proviamo ad eliminare la condizione di attesa, potremmo fare in modo che il processo o thread dichiari tutte le risorse di cui ha bisogno, se non posso riservare tutte le risorse allora il processo rimane in attesa.

La gestione dell'attesa può avvenire in modi diversi a seconda delle necessità, fifo, priorità, oppure in base al minor numero di risorse necessarie.

Il drawback richiede molte cose, chiediamo infatti al processo di dichiarare tutte le cose di cui ha bisogno.



R3: Resource with maximum index
P_3 violates the constraint of sorted requests
P_3 causes circular waiting

Alcune risorse possono essere gestite come se fossero infinite, queste risorse sono quelle prive di stato, un esempio sono le stampanti, potremmo mettere in piedi uno spool, ovvero mettiamo in coda i task della stampa per eseguirli.

Lo spooler è una coda dove ci metto tutti i task di stampa e la stampante pesca gli elementi in ordine fifo.

Lo spooler è un dispositivo virtuale e fa spooling su disco, due processi quindi possono sfruttarlo e terminare senza problemi.

Questa è una buona soluzione ma è utilizzata solo in casi particolari, se la risorsa utilizzata fosse un disco, fare un disco virtuale con spooling di dischi crea dei problemi.

Ci possono ancora essere dei deadlock nei buffer utilizzati dal gestore di spool.

Il lock ordering è una delle tecniche che mi permettono di risolvere il lock semplicemente, possiamo infatti ordinare i 3 processi in modo da evitare l'attesa circolare.

Condition	Method
Limited access to resources:	Spool / virtualization
"wait while holding":	Request all resources in advance
Circular waiting:	Lock ordering

Prevenzione dei Deadlock

Se vogliamo aumentare il numero di risorse con la virtualizzazione, invece di dare il processore al processo possiamo usare delle macchine virtuali con delle risorse e uno schedulatore che manda in esecuzione le macchine virtuali, non aumentiamo veramente il numero di risorse ma ci permette di disaccoppiare l'esecuzione.

L'attesa circolare può essere rimossa se imponiamo una richiesta delle risorse in ordine.

Dobbiamo considerare le risorse come molteplici e non singole, abbiamo un certo numero di risorse per ogni tipo, abbiamo anche una certa disponibilità.

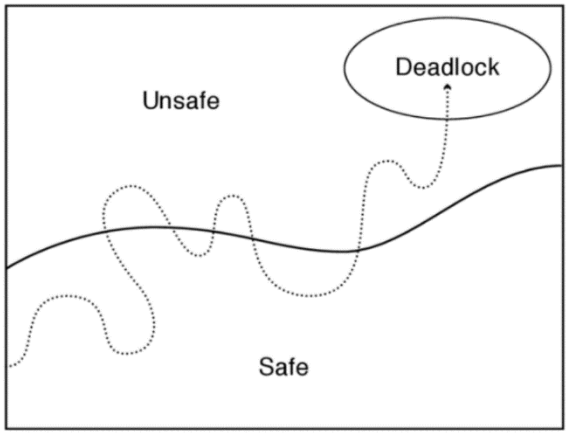
Questo va considerato con un metodo di richiesta che tiene conto del fatto se ho delle risorse disponibili in quel momento oppure no.

Algoritmo del banchiere

Nella pratica viene usato poco, abbiamo delle risorse con una molteplicità e richieste con necessità di varie risorse, accettiamo che parta un processo che fa un determinato numero di richieste se e solo se l'assegnazione di quelle risorse mi lascia il sistema in uno stato tale per cui esiste un cammino di tutti gli altri processi che partecipano all'acquisizione di risorse per cui qualcuno arriverà in fondo e rilascerà le risorse, quindi le risorse aumenteranno (Fatto dovuto alla terminazione del processo) .

Si chiama così perché questo dovrebbe essere il comportamento di un banchiere che presta i soldi in modo prudente, ovvero li presta a chi è in grado di restituirli, esiste un manager (un gestore) che valuta se assegnare le risorse o meno a seguito di una richiesta. Potrebbe non assegnare delle risorse perché soddisfando una particolare richiesta potremmo entrare in uno stato che non è detto essere sicuro, se il banchiere

vede che c'è rischio di deadlock ovvero di non riprendere i soldi non li presta, oppure aspetta di avere maggiori risorse per poterli prestare.



Se ho abbastanza risorse per dire che sono in uno stato sicuro posso soddisfare tutte le potenziali richieste, è come se avessi risorse infinite, non ho quindi una delle condizioni che ci porta al deadlock, se sono sotto la striscia nera vuol dire che ho abbastanza risorse, e sono nello stato sicuro, se oltrepasso la striscia nera entro in uno stato non sicuro il che vuol dire che potrei, anzi sicuramente, in un dato istante non avrò tutte le risorse per poter soddisfare tutte le potenziali richieste, non è detto però che entrare in uno stato non sicuro porti sicuramente al deadlock, potrebbe non portarlo, la striscia tratteggiata fa vedere un potenziale comportamento in cui potrei entrare in uno stato non sicuro per poi uscirne e magari rientrare, solo al termine potrei entrare in uno stato così detto compromesso, in cui non riesco più a rientrare in uno stato sicuro e sicuramente prima o poi andrò in uno stato di deadlock.

Transire in uno stato non sicuro non significa arrivare necessariamente ad un deadlock.

Finché rimango in uno stato sicuro, sicuramente non posso raggiungere deadlock.

Se voglio evitare deadlock, serve un algoritmo costoso dal punto di vista della complessità computazionale, pagando il costo di far gestire le risorse dal banchiere sono sicuro di essere sempre in uno stato safe e quindi non arriverò mai in uno stato di deadlock.

Definizioni:

• Safe state:

– For any possible sequence of future resource requests, it is possible to eventually grant all requests (not necessarily in the same order they are requested…)

• And thus to make all the processes end correctly

– May require waiting even when resources are available!

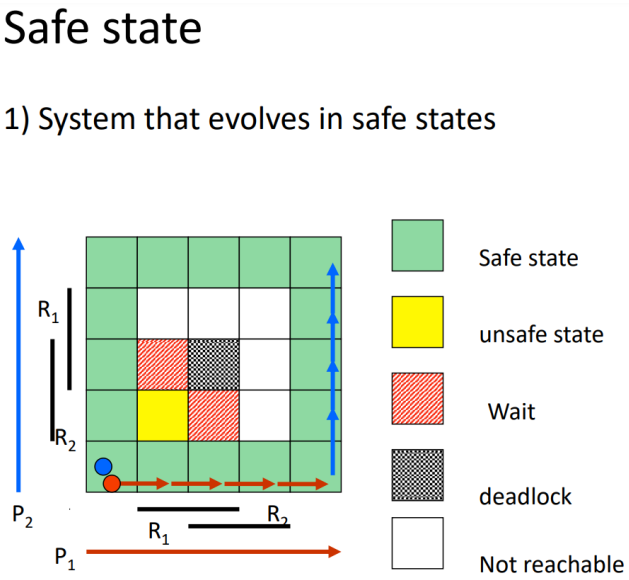
• Unsafe state:

– Some sequence of resource requests can result in deadlock

• Doomed state:

– All possible computations lead to deadlock

Cliccare sull'immagine per vedere esempio animato



Il banchiere evita di raggiungere uno stato unsafe, che non necessariamente implicherebbe deadlock, ma per sicurezza rimaniamo solo negli stati safe.

Il banchiere assegna la risorsa se e solo se lo stato raggiunto dopo tale richiesta è uno stato sicuro.

Algoritmo

Per far funzionare l'algoritmo del banchiere, e per avere soluzioni così forti che garantiscono la non esistenza del deadlock si deve pagare un prezzo, uno di questi è il costo computazionale dell'algoritmo, ma anche il fatto che ogni processo, ogni entità concorrente deve dichiarare il numero e il tipo delle richieste che avrà bisogno di effettuare.

Questo non vuol dire che gli vengano assegnate istantaneamente, gli verranno assegnate nel tempo quando ne avrà bisogno.

Il banchiere deve sapere l'esigenza, ovvero quante risorse dovranno essere assegnate a quel processo durante tutta l'esecuzione del programma.

Durante l'esecuzione di una richiesta da un generico processo P il banchiere controlla se le risorse che gli assegnerebbe permettano di rimanere in uno stato sicuro.

L'algoritmo del banchiere è un simulatore, esamina quello che succederebbe allo stato attuale se assegnasse la risorsa al processo che gli ha fatto richiesta.

Questo può farlo grazie al fatto che ogni processo ha dichiarato le sue esigenze, il banchiere infatti controlla che al momento attuale, soddisfando questa richiesta possano venir soddisfatte anche tutte le richieste di tutti gli altri processi.

L'algoritmo è concettualmente molto semplice, è un ciclo. Il banchiere utilizza un insieme di strutture dati, il vettore delle disponibilità o la matrice delle disponibilità, sa quante risorse lui gestisce R e conosce le disponibilità residue di ogni risorsa.

D: availability vector
For each resource R_k : D_k *number of available units of* R_k
For each process P_j :

- A_j: assignment vector;
- E_j: 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 (∃ a non-marked Pj that satisfies Ej≤D) {
    mark Pj;
    D = D + Aj ;
   } else ends while, the state is not safe;
}
success: the initial state is safe
```

Ogni volta che riceve una richiesta esegue questo pseudocodice, facciamo una simulazione, marchiamo un processo come soddisfatto se e solo se riusciamo a soddisfare tutte le richieste, finché esiste un processo non soddisfatto ovvero che non ho analizzato, vado avanti.

È un algoritmo che è iterativo ma continua ad andare avanti esaminando anche più volte lo stesso processo, ci chiediamo se l'esigenza del processo P_j è minore della capacità effettiva delle risorse.

Se ne esiste almeno uno che non riesco a soddisfare allora lo stato non è safe e non assegnerò la risorsa al processo che me l'ha richiesta.

Cliccare qui per vedere esempi.

Sistema con 4 processi p1-p4 e risorse r1-r4, ci serve conoscere l'esigenza iniziale di ciascun processo e la molteplicità, ovvero quante risorse per ogni tipo abbiamo.

Dobbiamo valutare se con queste esigenze e questo assegnamento attuale ai processi possiamo dire che questo è uno stato sicuro.

Per verificare che uno stato è sicuro devo verificare se riesco a soddisfare tutte le esigenze residue dei processi, quale scegliamo per primo?

Tipicamente si può pensare di utilizzare quello che ha esigenza minore, sostanzialmente noi dobbiamo continuare a valutare le esigenze di ogni processo sino a quando o li abbiamo marcati tutti oppure non riusciamo a far avanzare nessuno, a quel punto ci dobbiamo fermare, partiamo dal processo p2 per capire se possiamo soddisfare la richiesta.

A p2 gli manca una sola risorsa, r2, il banchiere si chiede se può assegnargli quella unità, la disponibilità di r2 è uno quindi possiamo soddisfare la richiesta, p2 ci restituirà tutte le risorse che aveva sino a quel momento, so che una volta che assegno l'ultima risorsa il processo termina, marchiamo infatti il processo come soddisfatto, e mettiamo nella disponibilità attuale tutte le risorse che aveva.

Passiamo poi a p3 ecct…

Questo algoritmo è molto costoso, nei SO abbiamo tantissimi processi e tantissime risorse, questo algoritmo nella pratica non si applica.

Nei SO moderni, questo algoritmo anche se in grado di evitare deadlock non viene utilizzato per diverse ragioni:

- Molto costoso quindi ha senso utilizzarlo solo in situazioni molto critiche.
- Dobbiamo sapere in anticipo quello di cui ogni processo ha bisogno.

I SO operativi moderni utilizzando la tecnica dello struzzo, siccome il deadlock è talmente poco probabile non facciamo nulla per evitarlo, le risorse sono tantissime.

I sistemi in unix anche in caso di deadlock garantiscono il login per l'amministratore, nella peggiore delle ipotesi l'amministratore può loggarsi e controllare la lista di processi, in modo killare quelli che utilizzano più risorse.

Filosofi a cena

N filosofi si incontrano per cena e ogni filosofo ha bisogno di due bacchette, quella a dx e sx, il problema è che le bacchette sono N, una volta che il filosofo ha acquisito le bacchette inizia a mangiare, dopo aver mangiato medita per un po', per poi riprendere a mangiare, se riesce ad acquisire solo una forchetta attende di acquisirle entrambe.

Le bacchette sono le risorse, noi vogliamo massimizzare la concorrenza ed evitare il deadlock.

```
while (true) {
    penso();   // il filosofo pensa
    // il filosofo di indice i decide di mangiare
    lockBastoncino[i].Acquire();   // acquisisce il bastoncino di destra
    // il filosofo si sospende se non può acquisire il bastoncino alla sua sinistra
    lockBastoncino[(i+ 1) mod N].Acquire();
    mangia();  // il filosofo di indice i mangia
    // rilascia i bastoncini
    lockBastoncino[(i+ 1) mod N].Release();
    lockBastoncino[i].Release();
}
```

Si potrebbe creare deadlock in quanto tutti tentano di acquisire la bacchetta alla loro destra e rimanere in attesa infinita.

Come possiamo evitare lo stallo?

Cerchiamo di agire su una delle condizioni che portano allo stallo.

3 possibili soluzioni:

Proviamo ad imporre un ordinamento, in questo particolare problema imporre un ordinamento significa far eseguire la funzione in modo asimmetrico, possiamo immaginarci che i filosofi di indice pari acquisiscano la bacchetta di destra e quelli di indice dispari la bacchetta di sinistra.

Un'altra soluzione si riferisce all'evitare una condizione di blocco, ovvero il wait while holding, evitiamo di tentare di acquisire una risorsa se non possiamo, rilasciando quella acquisita.

La terza soluzione usa il concetto di «monitor» per cui, per l'acquisizione dei bastoncini, si tiene conto dello stato dei filosofi vicini e si agisce di conseguenza.

Quando un filosofo smette di mangiare assegna le bacchette a chi era in attesa con una sola forchetta, non vengono rilasciate ma assegnate.

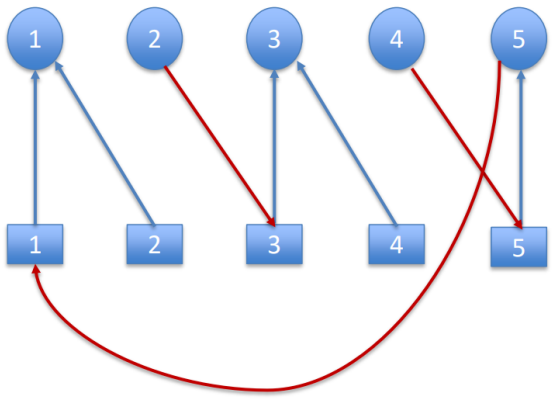
Prima soluzione

Questa soluzione è una soluzione che tende a differenziare il codice, tende ad etichettare il comportamento dei thread di indice pari e di thread di indice dispari.

Inoltre, questa soluzione se la andiamo a guardare in dettaglio, per 5 processi garantisce il massimo parallelismo che è 2, se proviamo con 4 filosofi non è detto che garantisca il massimo parallelismo, c'è almeno un caso in cui mangia uno solo quando potrebbero mangiare in 2.

```
while (true) {
    penso();
    if (i % 2) { // filosofo con indice dispari
       // prima acquisisce la bacchetta di sinistra e poi quella di destra
       lockBastoncino[i].Acquire();    lockBastoncino[(i+ 1) mod N].Acquire();
       mangia();  // il filosofo di indice i mangia
       lockBastoncino[(i+ 1) mod N].Release();    lockBastoncino[i].Release();
    } else  { // filosofo con indice pari
       // prima acquisisce la bacchetta di destra e poi quella di sinistra
       lockBastoncino[(i+ 1) mod N].Acquire();    lockBastoncino[i].Acquire();
      mangia();  // il filosofo di indice i mangia
      lockBastoncino[i].Release();    lockBastoncino[(i+ 1) mod N].Release();
   }
}
```

Esempio di un possibile grafo delle attese:



I filosofi di indice pari prendono prima la bacchetta di indice i+1 e poi i, i filosofi di indice dispari al contrario.

Nell'esempio di figura, riescono a mangiare i filosofi 1 e 3 mentre gli altri aspettano.

Appena 1 e 3 smettono di mangiare, mangiano (probabilmente!) 5 e 2….

Seconda soluzione

La seconda soluzione agisce su un'altra condizione in cui implementiamo l'attesa attiva, questa soluzione non va in stallo perché abbiamo rotto la condizione di attendere tenendosi la risorsa.

Con questa soluzione uno dei filosofi, potrebbe tentare di prendere una delle due all'infinito senza mai mangiare. (Problema di fairness)

```
while (true) {
    penso();   // il filosofo pensa
    // il filosofo di indice «i» decide di mangiare
    PrendiBastoncini(&lockBastoncino[i], &lockBastoncino[[(i+ 1) mod N);
    mangia();  // il filosofo di indice «i» mangia
    RilasciaBastoncini(&lockBastoncino[i], &lockBastoncino[[(i+ 1) mod N);
}
```

```
PrendiBasoncini(lock1, lock2) {
  while(true) {
    lock1.Acquire();
    if (lock2.tryAcquire()) return;  // successo
    lock1.Release();      // rilascio il bastoncino e ….
    swap(lock1, lock2); // …. provo nell'ordine contrario
  }
}
```

```
RilasciaBastoncini(lock1, lock2) {
      lock1.Release();
      lock2.Release();  }
```

Terza soluzione

L'ultima soluzione che utilizza attesa passiva sfrutta il concetto di monitor, definiamo una classe, un oggetto che espone dei metodi, in questo caso: inizializza, prendi e rilascia, il blocco è ad opera del monitor, cerchiamo di astrarre e non lasciar decidere al filosofo quale delle due prendere prima, ci pensa l'implementazione del monitor che in base alla sua politica cerca di essere fair.

```
while (true) {
    penso();   // il filosofo pensa
    // il filosofo di indice i decide di mangiare
    PrendiBastoncini(i);
    mangia();  // il filosofo di indice i mangia
    RilasciaBastoncini(i);
}
```

L'assegnamento ai filosofi avverrà se entrambi i bastoncini sono disponibili, in caso contrario il thread si sospenderà senza nessuna risorsa, verrà riattivato quando un thread rilascerà una risorsa, il monitor tiene traccia di uno stato interno, sa infatti cosa sta facendo il filosofo di sinistra e quello di destra, sfrutta questa conoscenza per prendere una decisione, utilizziamo un array di stato con tante posizioni per quanti sono i filosofi, in questo array ci sono gli stati dei vari filosofi, se sta mangiando, se sta pensando, se è in attesa con uno dei due bastoncini, per proteggere queste strutture dati utilizziamo una variabile mutex e un array di variabili di condizione che ci permette di far svegliare solo il thread che ci interessa.

Scheduling

Definizioni:

- Task/job: è una richiesta che deve essere schedulata (movimento del mouse o in generale qualunque cosa possiamo richiedere al so).

- Latency/Response: Sono due metriche che indicano quanto tempo ci metto a completare quel dato task.

- Throughupt: mi dice quanti task riesco a completare in un'unità di tempo, alcune tecniche ottimizzano il throughput e altre il response time.

- Overhead: per applicare l'algoritmo di scheduling dobbiamo effettuare dei calcoli, del lavoro, avere overhead grandi significa sprecare tempo per eseguire processi di sistema che non mandano avanti l'esecuzione del programma.

- Fairness: Dobbiamo stare attenti all'equità, in generale vogliamo che l'algoritmo di scheduling permetta di fare progresso a tutti.

- Predictability: è interessante studiare gli algoritmi di scheduling che sono predicibili ovvero che sotto determinate condizioni il risultato sia deterministico e non random.

L'altro aspetto è quello delle prestazioni nel tempo, vogliamo essere sempre efficienti e non in alcuni casi sì e in altri no.

Workload: è il carico di lavoro ovvero l'insieme di task che un certo sistema deve gestire.

Preemptive scheduler, politica che permette il prerilascio, togliamo la risorsa precedentemente assegnata, in alcuni casi se in coda pronti compare un processo che ha una priorità maggiore del thread in esecuzione e l'algoritmo è con prerilascio il thread in esecuzione viene de-schedulato e viene messo in esecuzione quello con priorità maggiore.

Algoritmi work-conserving: Si distinguono in work conserving e resource conserving, nel work conserving finché abbiamo task da schedulare vogliamo utilizzare tutte le risorse che abbiamo per farli.

In alcuni contesti, dove è importante conservare una determinata risorsa, diamo priorità al salvataggio della risorsa.

Un algoritmo di scheduling è un algoritmo che prende in input un workload e decide con una qualche politica quali task eseguire prima. È un algoritmo che magari privilegia certe metriche rispetto ad altre, noi parleremo di algoritmi work conserving.

Algoritmi di scheduling

Primo algoritmo: FIFO

Primo algoritmo di scheduling banale: FIFO.

Il primo task che arriva da eseguire lo eseguiamo, eseguiamo i task in ordine di arrivo, eseguo il task fino alla fine. Non prevediamo prerilascio.

Lo utilizziamo come benchmark, ovvero come algoritmo di riferimento.

Secondo Algoritmo: SJF

Secondo algoritmo: Shortest job first

Eseguo prima i task più brevi, passando avanti ai task lunghi, eseguiamo i task che possiamo finire più velocemente.

Ci sono due versioni con e senza prerilascio:

La prima è quella senza prerilascio, io potrei star eseguendo un certo task che è lungo 100, se siamo senza prerilascio e arriva un task lungo 10 io devo aspettare che quello lungo 100 termini, se sono con prerilascio quando arriva quello con lunghezza minore passa lui in esecuzione.

Con prerilascio dobbiamo salvare le informazioni del task in esecuzione per riprendere da dove eravamo rimasti con l'esecuzione.

Esistono versioni con prerilascio dette shortest remaining time first, in questi casi rimettiamo in esecuzione quelli che hanno meno tempo rimasto per la fine dell'esecuzione.

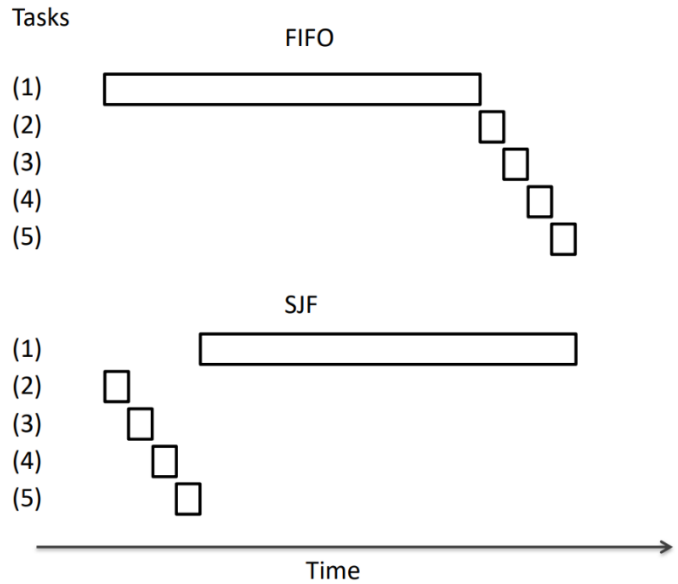
Il problema è che molto spesso non sappiamo un task quanto tempo ci metterà ad eseguire, dunque possiamo fare solo delle stime.

Esempio:

Supponiamo di avere al momento 5 task da schedulare

Per scegliere quale algoritmo va meglio dobbiamo scegliere una metrica da utilizzare, il tempo medio di risposta nel caso di fifo sarebbe molto più alto di sjf.

La media a volte non è tutto, potrei voler considerare altri aspetti, se consideriamo la varianza, nel fifo finiscono quasi tutti uno dopo l'altro, nel secondo invece c'è una grossa varianza ovvero il processo più lungo terminerà molto tempo dopo rispetto alla terminazione dell'ultimo processo piccolo.



SJF non è equo, se mi arrivassero sempre task minori di cento quello rimarrebbe sempre lì in attesa di essere eseguito, ci potrebbe essere starvation.

SJF non è fair ma è ottimo nel response time, fifo è ottimo quando i task hanno tutti la stessa lunghezza, infatti mi dà il troughput migliore.

Possiamo provare formalmente a dimostrare che sjf è un algoritmo ottimo se si considera il tempo medio di risposta.

SJF funziona particolarmente bene quando ho sistemi batch.

Terzo Algoritmo: Round Robin

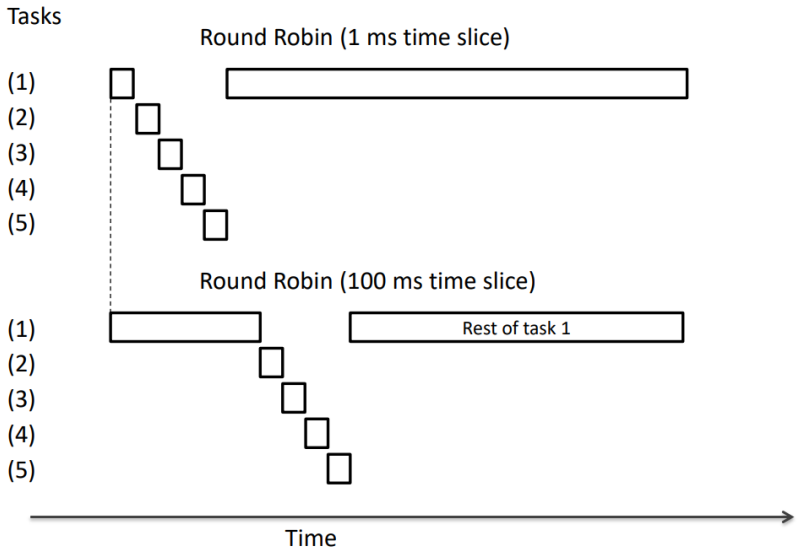
Algoritmo di round robin, ogni task acquisisce la cpu per un certo periodo di tempo, chiamato quanto di tempo, questo quanto di tempo è fisso, il round robin fa il prerilascio, vengo messo in coda pronti, e la cpu viene assegnata ad un altro thread che verrà eseguito per uno stesso quanto di tempo.

Round Robin necessita di una coda di thread e di determinare un quanto di tempo.

Questo algoritmo è sicuramente fair, però pone un problema:

Quanto facciamo lungo il quanto di tempo?

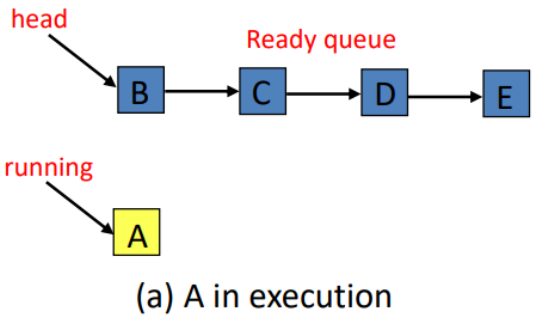
Se utilizziamo un quanto di tempo troppo corto dobbiamo fare content switch spesso, questo induce un grande overhead, se lo facciamo troppo lungo rischiamo di cadere nel tempo del fifo, se arriva un job lungo che non si sospende mai rischiamo di ritardare gli altri.



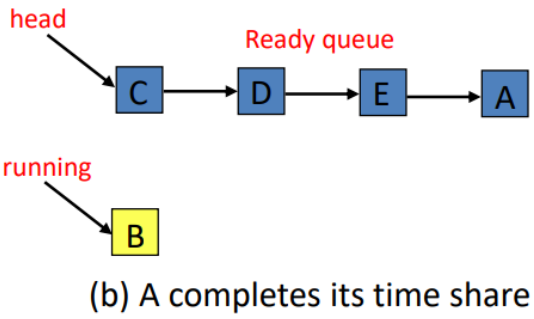
Il round robin è deterministico possiamo sapere sin da subito quanto tempo ci vorrà prima di finire quei task.

Funziona bene se alcuni thread si sospendono e non completano nel quanto di tempo.

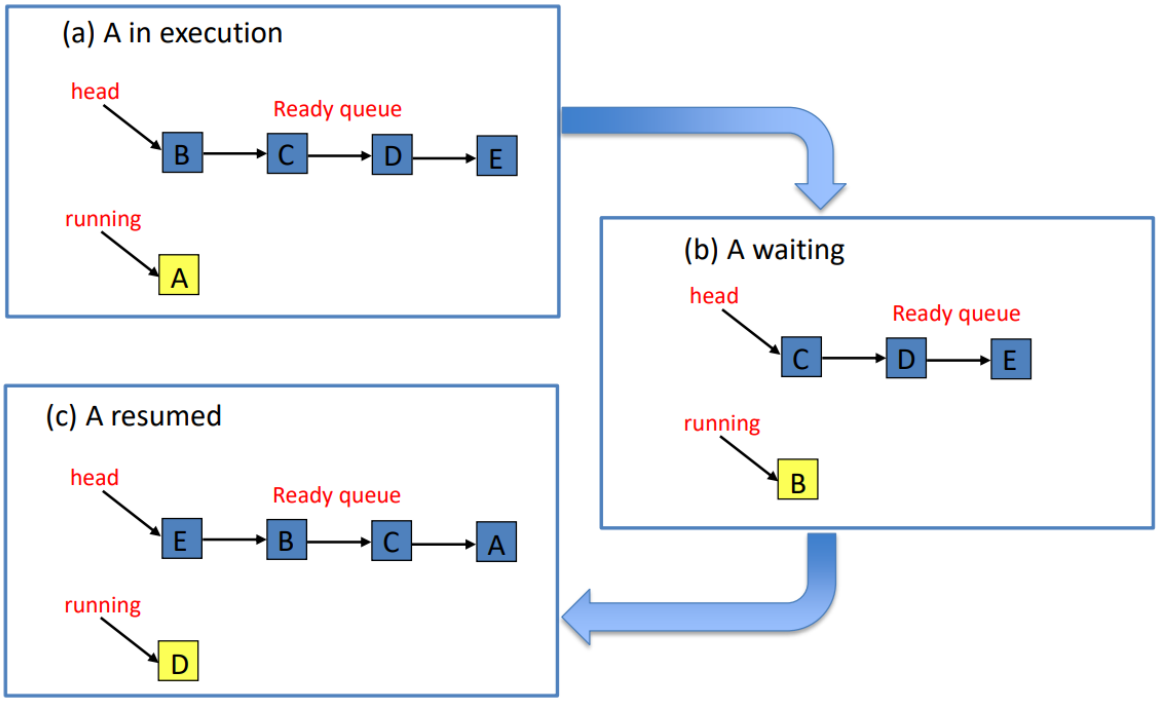
Il quanto di tempo viene segnalato dal timer, abbiamo la gestione delle interruzioni, in questo caso sarà compito dello scheduler riattivare il timer, in linea di principio il quanto di tempo potrebbe essere variabile.



(a) A in execution



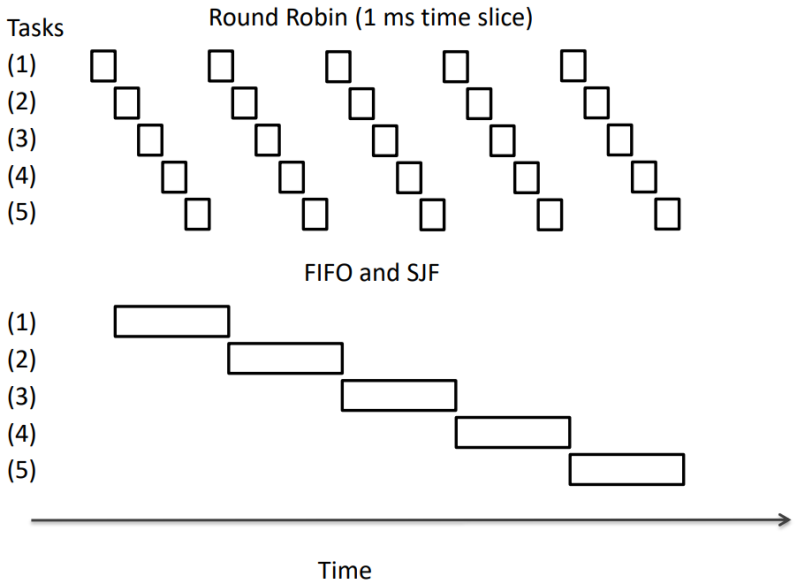
(b) A completes its time share



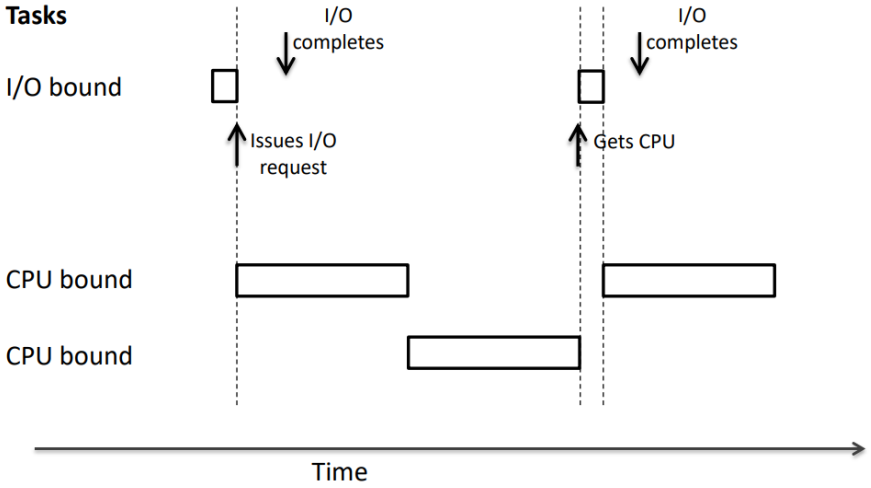
Nel caso di quando ci svegliamo ovvero quando facciamo una V su un semaforo, veniamo messi in coda pronti e lo scheduler deciderà chi mettere in esecuzione in base alla priorità ecct, nei sistemi correnti il quanto di tempo è di circa 20-120 msec.

Supponiamo che il content switch ci costi 0, il round robin è meglio di fifo?

No, il migliore è fifo, infatti è il suo caso ottimo, tutto sommato round robin non è così male, con il round robin riusciamo a far fare progresso a tutti è più reattivo, quindi in tutti quei casi in cui abbiamo bisogno di reattività è migliore di fifo, se però consideriamo il tempo di risposta è peggio di fifo.



Noi supponiamo che il round robin sia fair, ma in realtà non è così in quanto i task non sono tutti uguali, i task che eseguiamo possono essere divisi in due categorie cpu bound e I/O bound, nei cpu bound sfruttiamo tanto la cpu, ci sono però algoritmi che sono più I/O bound, accediamo tanto al disco e alla rete, facciamo tante syscall e abbiamo tante interruzioni.



Se abbiamo quanti di tempo lunghi le applicazioni di I/O bound rischiano di non sfruttarlo mai.

In realtà il round robin non è equo se consideriamo workload misti.

Potremmo risolvere modificando il quanto di tempo, ma diminuendolo faremmo molti cambi di contesto e quindi tanto overhead.

Se invece lo allunghiamo tanto, tendiamo a privilegiare i task cpu bound, perché quelli I/O bound richiedono di essere de-schedulati per la lettura I/O e finiscono in fondo alla coda pronti una volta finita.

Quarto Algoritmo: Max-Min Fairness

Abbiamo una variante che è il Max-Min fairness, questo algoritmo cerca di assegnare il quanto di tempo in base al reale utilizzo che ogni processo fa, cerchiamo di massimizzare l'assegnamento dell'allocazione minima del tempo assegnato ad un task, dobbiamo immaginarcelo nel contesto di workload misti, in cui abbiamo sia cpu che I/O bound.

Esempio:

Se ho 4 task da eseguire e ho uno slot di esecuzione di 100 inizialmente potrei dire che divido lo slot di 100 e do 25 ad ognuno, questo sarebbe il round robin, magari però il primo ha bisogno solo di 5 unità, le rimanenti 20 unità di tempo che non vengono usate vengono ripartite tra gli altri thread in modo proporzionale, cerchiamo di eseguire i thread che richiedono meno istanti di tempo in modo da poter ripartire meglio il tempo rimanente.

Chi ha bisogno di poco tempo probabilmente lo avrà tutto, chi ne ha bisogno di tanto glielo assegno proporzionalmente.

Dal punto di vista pratico questo algoritmo da gestire sarebbe troppo costoso, dovrei gestire la coda pronti con una priorità.

Quando si parla di workload misti non vuol dire che una applicazione che è I/O bound sarà sempre I/O bound, tendenzialmente un'app in istanti di tempo diversi si può comportare come cpu o I/O bound.

L'approccio che tendenzialmente usiamo è un ulteriore algoritmo di scheduling che è l'MFQ, non è ottimo in nessun caso ma si comporta bene al caso medio.

Quinto Algoritmo: MFQ

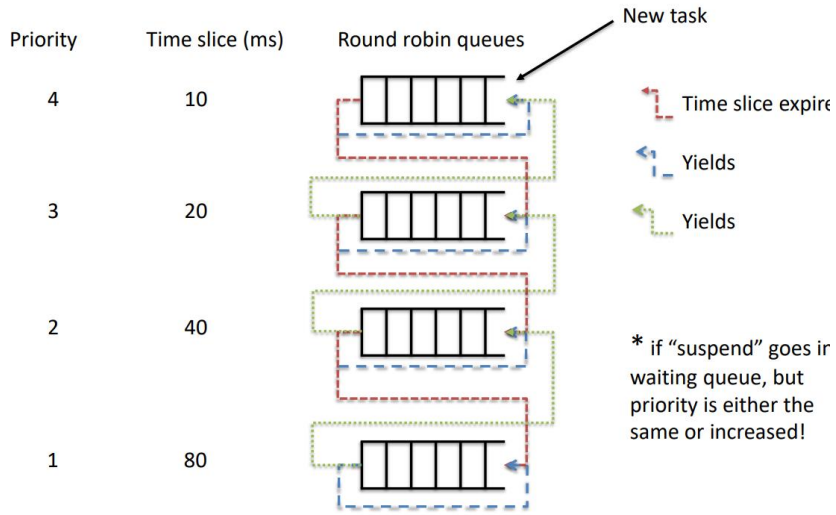
(Multi level Feedback Queue)

Garantiamo che non ci sia starvation, abbiamo un basso overhead, gestisce bene workload misti, ed è equo, gestisce i task in base alla priorità.

Funzionamento:

Invece di utilizzare una sola coda di tipo round robin usiamo un set di code di tipo round robin, ogni coda ha una certa priorità e ha associato uno specifico quanto di tempo, abbiamo dunque n quanti di tempo diversi.

La particolarità è che le code che hanno maggiore priorità hanno un quanto di tempo più basso, quelle a più bassa priorità hanno un quanto di tempo più lungo, lo scheduler cerca di scorrere questa lista di code passando da quella a più alta priorità a quella con meno priorità.



Ogni volta che scade il quanto di tempo il task viene spostato nella coda di livello inferiore.

È come se supponessimo che fosse cpu bound quindi potrebbe beneficiare di un quanto di tempo più lungo, se rilasciamo il processore volontariamente invece rimaniamo nella coda corrente.

Se il processo si sospende potrei essere un task I/O bound, quindi il processo sale di un livello di priorità.

In questo modo avremo che i thread I/O bound dopo un po' di tempo tenderanno a stare nelle code più alte con quanto di tempo più basso e i thread cpu bound nelle code più basse con quanti di tempo maggiore.

Cosa succederebbe se continuassero ad arrivare I/O bound?

Sembrerebbe non essere fair, la fairness deriva però dall'implementazione di questi algoritmi.

Ci sono un sacco di correttivi per far sì che non ci siano penalità nei confronti di qualche task.

Ogni tanto se un task non va in esecuzione per un po' di tempo gli viene dato un boost di priorità, lo scheduler deve quindi tenere traccia dell'esecuzione dei thread.

Priority inversion

Problema:

Il thread A che ha priorità 12 esegue una P su un semaforo e si sospende, passa in esecuzione il thread/task che ha priorità 8, chi avrebbe potuto fare però la V su questo semaforo è il task B che ha però priorità 5, in questo modo A che ha priorità maggior viene ritardato da un task che ha priorità minore e che non va in esecuzione.

Per questo motivo se A è un po' che non esegue, quando viene svegliato gli verrà dato un boost di priorità, cerchiamo di ribilanciare il comportamento cercando di minimizzare queste situazioni che non è possibile sapere a priori.

Queste situazioni non dipendono solo dalla struttura statica del programma ma magari anche dai dati in ingresso.

Uniprocessor Summary:

Le politiche che abbiamo visto fino ad esso le abbiamo immaginate su single core.

Fifo semplice ha 0 overhead, non ha supporto hardware però non si comporta bene se i task hanno tempi diversi, se i task hanno la stessa durata fifo è ottima.

Considerando solo il tempo di esecuzione sjf è ottimo quando abbiamo task con tempi di esecuzione variabili, sjf ha però una varianza limitata, non è dunque fair.

Round Robin se i task sono di lunghezza variabile approssima sjf, se i task hanno tutti la stessa durata round robin tende a fifo anche se si comporta peggio perché c'è overhead per il cambio di contesto, round robin è fair.

Per workload variabili round robin non va bene, utilizziamo quindi il max/min fairness, sia round robin che max/min fairness non hanno il problema della starvation.

Ci serve però un comportamento dinamico e abbiamo visto mfq, con un po' di accorgimenti riusciamo a non essere ottimi in nessun caso ma ci comportiamo bene nei casi medi.

MultiProcessore

Le cose si complicano molto, noi non lo tratteremo in modo esaustivo, è molto comune avere server di fascia alta con 100-128 core, dobbiamo sfruttare bene tutti i core.

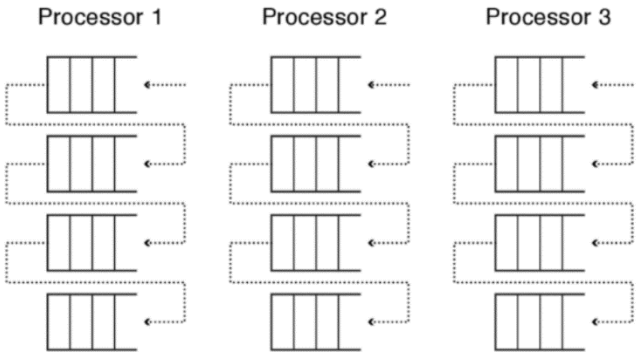
Il primo approccio potrebbe essere quello di usare l'algoritmo mfq per tutti i core, utilizziamo una struttura mfq protetta da spinlock. Siccome l'algoritmo di scheduling parte su tutti i core modificando la tabella, ci serve la mutua esclusione.

Tutti i core prendono da questa tabella i task da eseguire.

Abbiamo due problemi:

Con tanti core la contesa sulla spinlock è molto alta, inoltre non sfrutto al meglio le cache, un task che ha eseguito su un core la prossima volta potrebbe eseguire su un core diverso avendo però i dati nella cache dell'altro core e inducendo quindi cache miss, dunque questo algoritmo non funziona bene.

Cerchiamo quindi di ridurre la contesa delle spinlock e soprattutto ottimizzare i livelli di cache che ho sui singoli core privati. Utilizziamo un mfq con affinity scheduling, utilizzo una coda di tipo mfq per ogni processore, devo comunque utilizzare una spinlock su ognuna di loro ma in questo modo gli accessi si distribuiscono su n strutture dati anziché una sola, ogni scheduler lavora sulla coda assegnata a quel processore, quando la sua coda è vuota cerca di andare a prendere dei task dalle altre code.



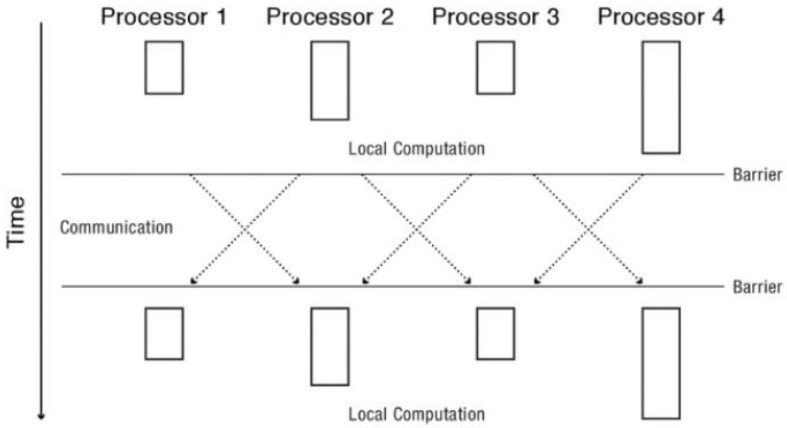
Il vantaggio delle cache lo abbiamo perché finché rimetto in qualche posizione di queste code un task rimango ad utilizzare la cache locale, il cache transfering lo abbiamo solo quando c'è work stealing ovvero quando un core prende il lavoro dalla coda di un altro, si necessita quindi anche del trasferimento dei dati.

Come al solito abbiamo il problema di quanti task rubare, se ne rubo pochi potrei rischiare di dover fare stealing tante volte, se ne rubo tanti quel core potrebbe rimanere senza.

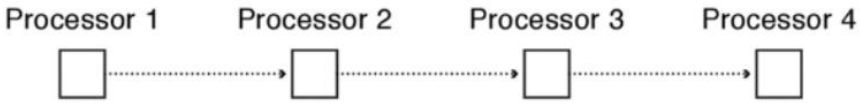
Ci sono però altre problematiche, finora abbiamo considerato i programmi applicativi multi-threaded ma non veramente paralleli, nei multicore attuali eseguiamo sia processi single core oppure thread che fanno cose diverse tra di loro.

Ci sono dei casi in cui l'applicazione è intrinsecamente parallela, i cui thread sono sì paralleli ma a volte necessitano di sincronizzarsi tra loro.

Bulk Synchronous Parallel (BSP)



Producer-Consumer pipeline



Nella bsp ci sono thread di lunghezza variabile che vengono eseguiti, prima di passare alla prossima esecuzione c'è però una barriera, i thread devono infatti sincronizzarsi.

Il tempo di esecuzione è dettato quindi dal thread più lungo.

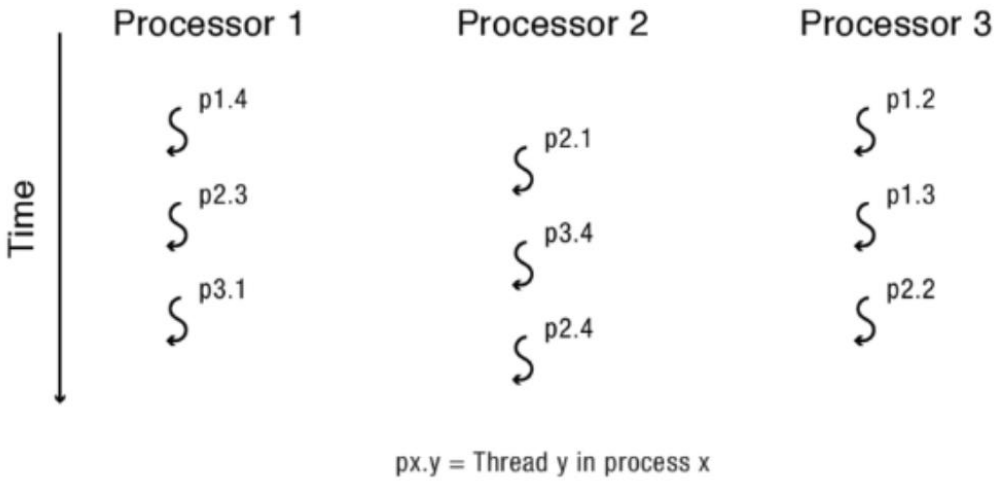
Un problema simile si ha nel classico produttore e consumatore dove ho tanti consumatori e tanti produttori.

Ci sono dei thread che lavorano in parallelo ma alcuni thread hanno bisogno del lavoro del thread precedente, il tempo è dettato quindi da quello più lento.

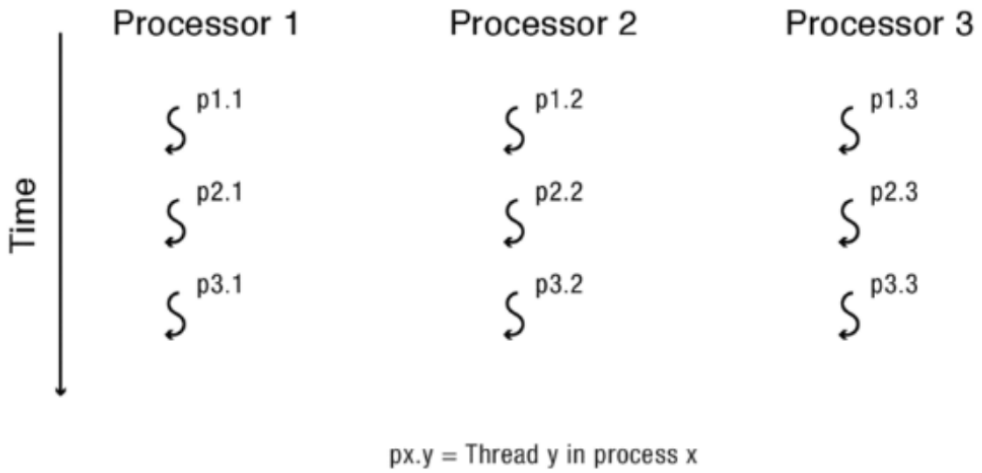
Ci conviene schedulare questi thread insieme, infatti se un thread si bloccasse per motivi di scheduling si bloccherebbe l'intera esecuzione.

Oblivius scheduling

In generale è lo scheduling mfq con affinity, ma con tipo oblivius generalizza i programmi paralleli.



Utilizziamo un altro tipo di scheduling, il gang scheduling, sapendo che ci sono task paralleli tutti i thread della stessa applicazione li scheduliamo o tutti insieme o nessuno, i thread di un'applicazione vanno quindi in esecuzione parallelamente sui processori.

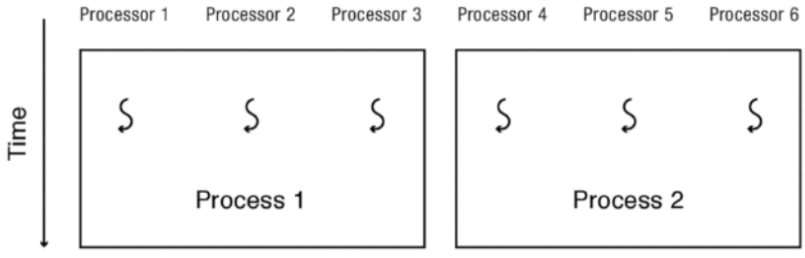


Questa soluzione non è ottima, se abbiamo infatti task sequenziali andremmo ad eseguirli solo su un core lasciando gli altri liberi.

Su sistemi general purpose la scelta non è banale in quanto molte volte eseguiamo programmi sequenziali altre volte paralleli, non esiste UN algoritmo di scheduling.

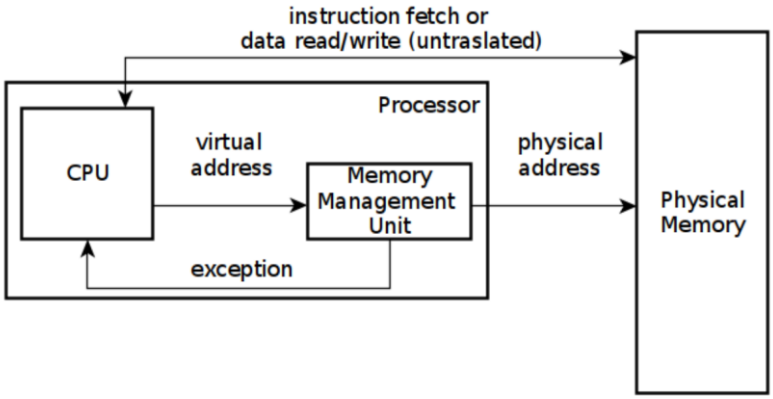
Space sharing

Un altro algoritmo è questo, potremmo partizionare il processore, invece di utilizzare tutti i processori ne uso un sottoinsieme e mando in esecuzione i thread dell'app1 sul primo set, i secondi sull'altra partizione.



Address Translation

Ci serve un meccanismo a carico del SO che permette di fare la traduzione da indirizzo logico a indirizzo fisico, vogliamo mappare le zone senza tener conto di alcun vincolo di continuità.



Obiettivi:

Vogliamo realizzare diversi aspetti:

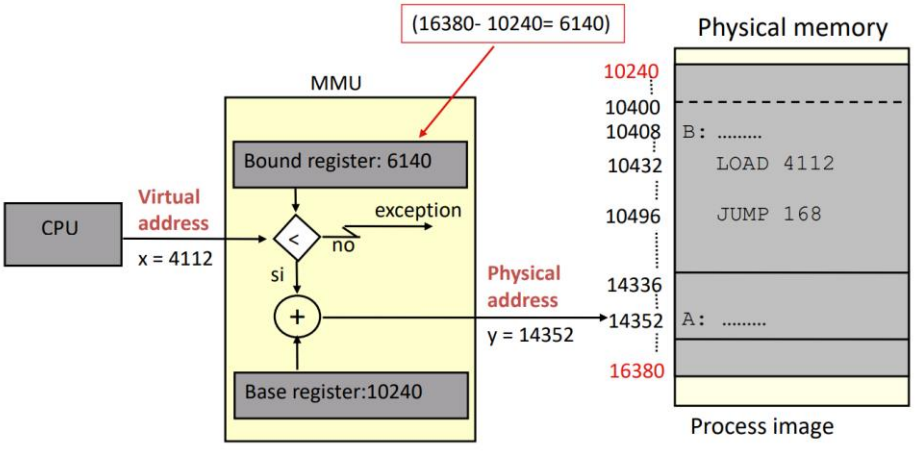
- Protezione della memoria
- Condivisione della memoria
- Vogliamo piazzare varie parti del nostro programma in varie parti della memoria fisica
- Vogliamo che il meccanismo di traduzione degli indirizzi sia efficiente
- Portabilità

Tra i vari usi che possiamo avere di questa astrazione uno è quello di poter condividere pezzi di codice e dati tra processi diversi, possiamo gestire la memoria dinamicamente, fino a cose un po' più complesse come ad esempio la zero-copy I/O, in cui copiamo direttamente i dati attraverso il dma, senza interrompere l'esecuzione sulla cpu.

Questo ci permette di fare molte cose, come ad esempio process migration, potrei spostare un processo da una macchina ad un'altra macchina senza dover interrompere l'esecuzione del processo.

Virtual base and bound

Lo avevamo accennato con indirizzi fisici, ora lo rivediamo con indirizzi logici, la cpu riferisce sempre indirizzi logici.



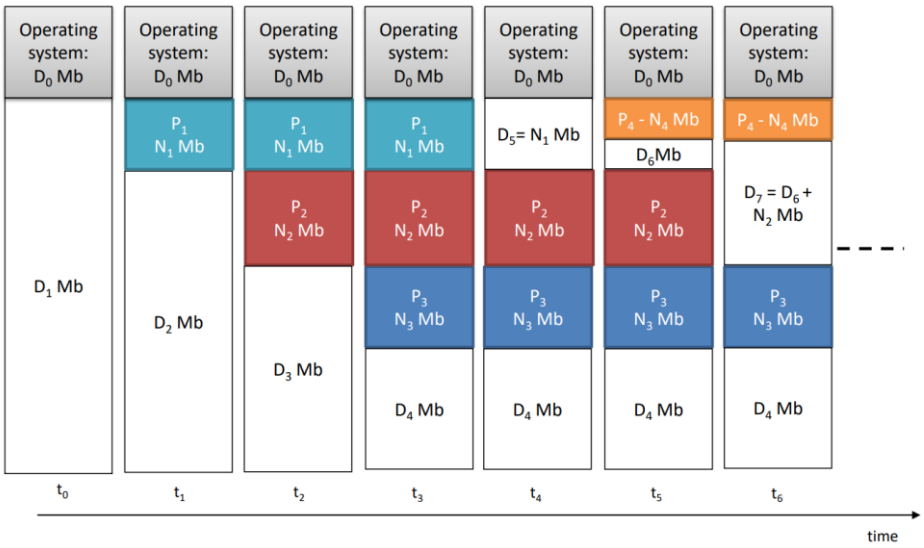
La memoria del processo è allocata come un unico segmento di memoria fisica contiguo che contiene tutti i dati del processo, il programma ad un certo punto chiede di caricare un certo indirizzo, questo indirizzo viene tradotto dall'mmu utilizzando due registri, base e bound, controlliamo che l'indirizzo a cui stiamo tentando di accedere stia tra base e base+bound, l'indirizzo virtuale viene trasformato in indirizzo fisico come base + ind_virt.

Questo meccanismo è estremamente efficiente in quanto ogni volta che un processo va in esecuzione devo cambiare solo i valori di base e bound.

Questo metodo però ha due problemi:

La memoria fisica è contigua, viene allocato un solo segmento, se volessi condividere una parte in lettura tra due processi non potrei farlo perché ognuno deve avere il proprio segmento, ci permette di avere un'astrazione ma è troppo rigido, non possiamo fare la condivisione di pezzi di memoria tra processi diversi, per condividerla dovrei replicarla.

L'altro problema è che questi segmenti contigui sono di dimensione variabile, per cui dopo un po' quello che succede è che potremmo avere molto spazio libero ma frammentato in porzioni piccole, in questo modo un processo con uno spazio di indirizzamento necessario più grande della dimensione maggiore tra i pezzi frammentati non può andare in esecuzione.



Come vengono allocate le nuove partizioni?

Abbiamo due metodi:

First-fit e Best-fit.

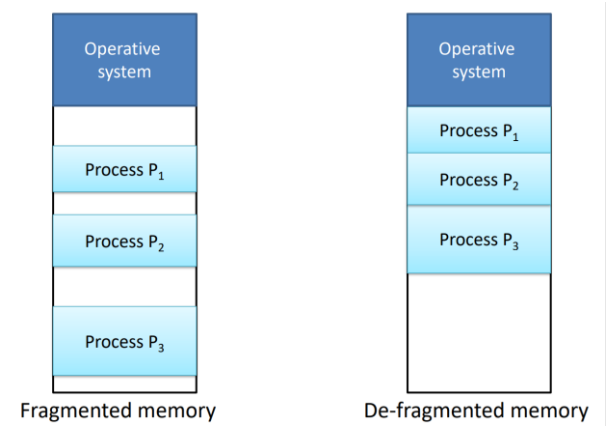
Possiamo scegliere il primo che troviamo libero (che abbia dimensione sufficiente), oppure possiamo scegliere tra i buchi esistenti che saranno memorizzati in una struttura.

Scegliendo il best fit necessitiamo di una lista ordinata che ha però un costo di gestione, in generale questa politica di base e bound virtuale, crea questo problema che è la frammentazione.

Esistono due tipi di frammentazione, la frammentazione esterna che è quella che abbiamo visto sino ad ora, sono buchi inutilizzati che non appartengono a nessun segmento allocato e si ha quando creiamo segmenti di dimensione variabile.

C'è un problema diverso che è la frammentazione interna che abbiamo quando i segmenti sono tutti grandi uguali, a questo punto però rischiamo di sprecare tanto spazio allocato se abbiamo un processo che usa poca memoria.

La frammentazione è un problema serio, in particolare con i segmenti variabili abbiamo solo frammentazione esterna ed è il caso più tipico, ogni tanto a seguito di questo problema il SO deve compattare de-frammentando la memoria, in modo tale da creare un unico segmento vuoto in fondo.

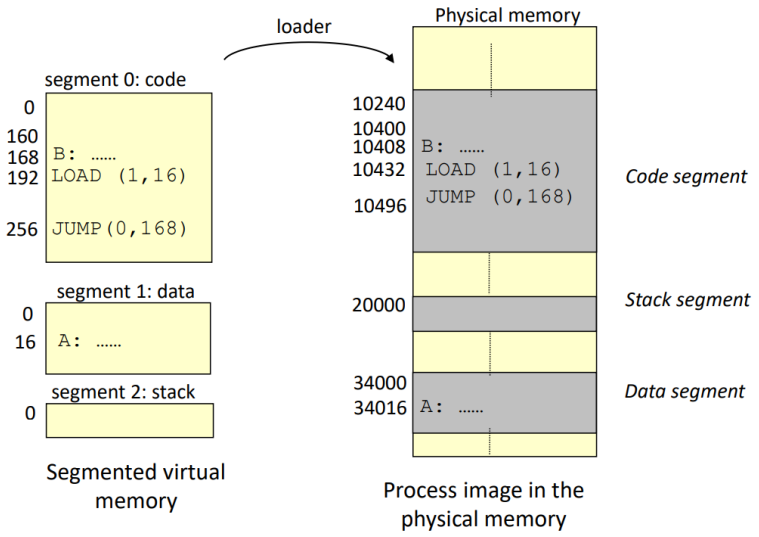


I pro del virtual base and bound sono che è semplice ed efficiente perché necessitiamo di poca logica di controllo, ha però il problema della frammentazione e non è flessibile come vorremmo.

Segmentazione

L'idea del segmento è buona perché è semplice da realizzare e ci permette di avere una base ed una lunghezza, anziché avere un segmento solo per tutto il processo aumento la possibilità di avere segmenti, tutti di una dimensione diversa, non mi bastano più dunque due registri, bensì mi serve una tabella, un segmento è sempre un'area di memoria contigua, però in questo caso, in questi segmenti che possono essere tanti, di solito si parla di decine o centinaia di segmenti, non abbiamo vincolo di continuità. I vincoli di continuità li abbiamo solamente all'interno del segmento, li possiamo dunque piazzare dove vogliamo all'interno della memoria. Possiamo inoltre non solo utilizzare base+length ma possiamo assegnare dei diritti ad ogni segmento, potremmo voler non modificare mai la parte codice dando permessi di accesso differenti per ogni segmento che creo, ad esempio il segmento codice potrebbe essere solo read only.

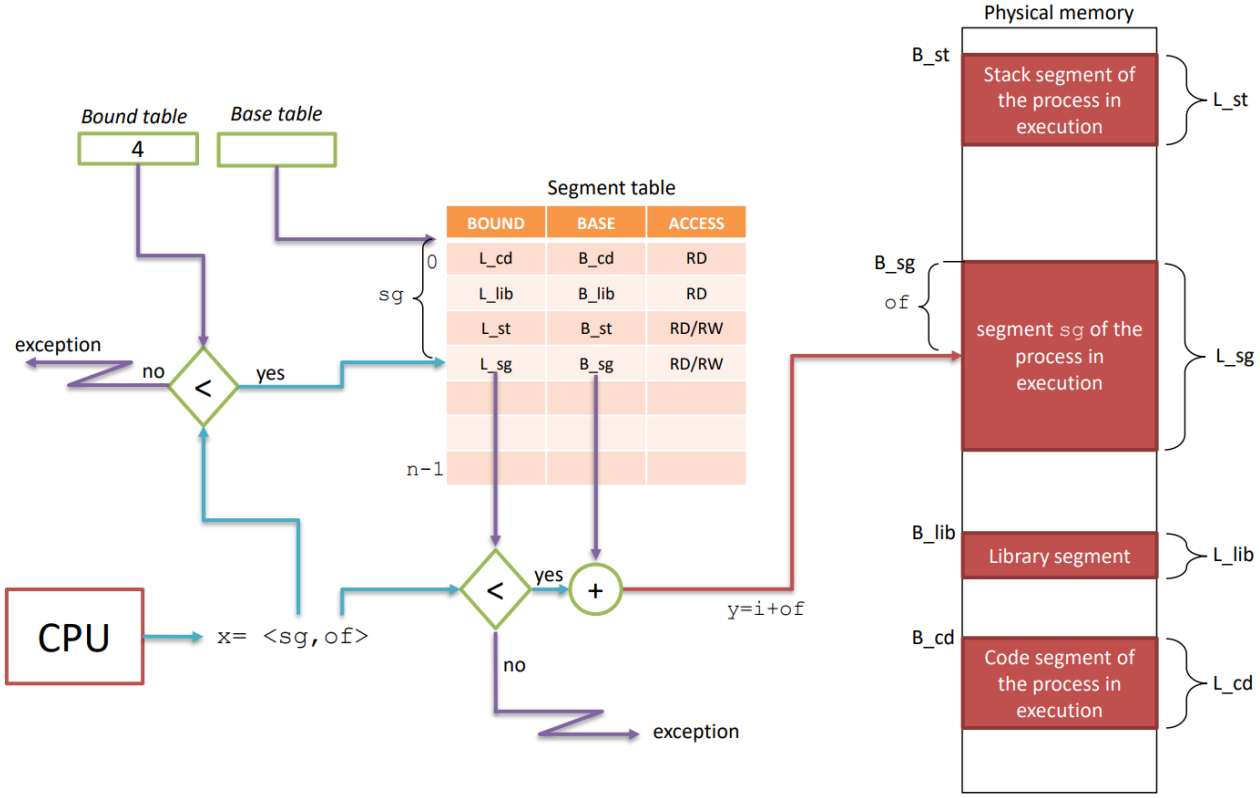
I processi possono condividere i segmenti.



Nel nostro programma con indirizzi logici i segmenti partono da 0 per ogni segmento e arrivano a length, in questo esempio abbiamo 3 segmenti, uno per il codice, uno per la parte dati e uno per lo stack. L'espansione dei segmenti in questo caso potrebbe essere fatta più facilmente in quanto dovrei espandere solo un segmento ed eventualmente avrei la necessità di spostare un segmento in un'area di memoria che permetta di fare l'estensione. Questi segmenti sono memorizzati in memoria fisica non necessariamente in modo contiguo.

Dal punto di vista architetturale quello che devo fare è sostanzialmente avere una segment table che può essere vista come un insieme di registri, dove una riga di questa segment table è l'indirizzo base ovvero la seconda colonna, abbiamo poi la lunghezza e i diritti di accesso, ci servono dei registri aggiuntivi, essendo il numero di segmenti non troppo grande è ragionevole pensare che la tabella dei segmenti sia tenuta ad hardware dentro la mmu, può essere vista come una sorta di registro con qualche centinaio di righe.

La implementiamo ad hardware in quanto è molto veloce e la traduzione viene fatta dall'mmu accedendo a questa tabella.



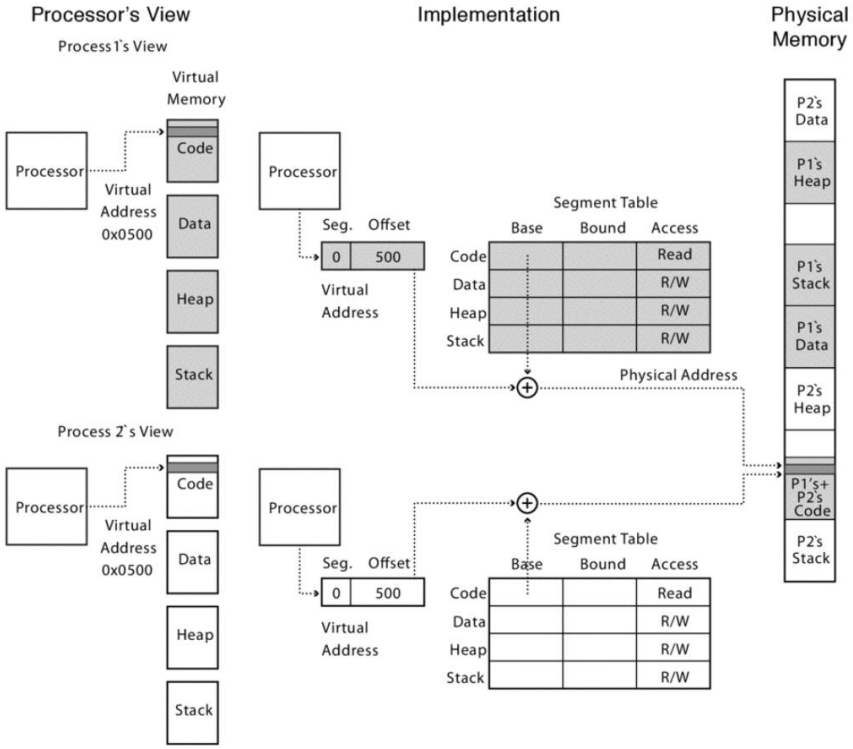
Un generico indirizzo logico che è sempre un indirizzo a 32 bit, lo vediamo diviso in due parti: una parte indica l'indice di segmento, l'altra parte ci dice l'offset del segmento, quello che fa l'mmu è scorporare la parte dell'indirizzo logico che corrisponde all'indice del segmento, controlla in base agli altri registri, in particolare a quanto è lungo se magari stiamo sforando, ovvero se sto creando un indirizzo che sta sforando in un segmento non esistente. Se invece va bene e sta indirizzando un segmento corretto, controlla se con l'offset rientro dentro la lunghezza, se dovesi sforare, avrei segmentation_fault.

Se va tutto bene prendiamo l'indirizzo base aggiungiamo un offset e otteniamo l'indirizzo destinazione.

Necessitiamo di una quantità maggiore di risorse hardware però dal punto di vista della logica aggiuntiva non è molto costoso.

Segment sharing

È facile fare segment sharing, basta dare la stessa entry allo stesso segmento nelle due tabelle dei due processi



Copy on write

Un altro aspetto che possiamo realizzare con la segmentazione è la copy on write, questa è una feature molto usata nei SO.

Esempio: Il processo padre crea un processo figlio con fork, facciamo una copia dello spazio di indirizzamento, se dovessi fare questa copia dovrei pagare il tempo di duplicare tutti i segmenti del processo padre nel processo figlio, dopo la fork potrei fare una exec, buttando via tutto quello che ho copiato ripartendo con un nuovo segmento codice, un segmento stack azzerato, sprecando un sacco di tempo per fare la copia inutilmente, in passato veniva usata la vfork() è una syscall ormai inutilizzata in quanto adesso esiste la copy on write, la vfork() evitava di fare la copia, faceva semplicemente la copia logica.

Ora con la copy on write è il SO a farlo per noi, egli non fa infatti una copia vera, bit a bit, ma crea una nuova tabella dei segmenti per il nuovo processo, nella tabella dei segmenti del padre e del figlio scrive gli stessi valori e segna tutto come readonly, ogni volta che il padre o il figlio tentano di scrivere il segmento si genera un'eccezione di violazione di protezione, però il SO sa che aveva fatto una fork e duplica solo in quel momento il segmento marcandolo come read-write, si fa quindi la copia on write, il segmento viene effettivamente copiato solo se tentiamo di scriverlo.

Un'altra cosa che si può fare è la zero-on-reference, potremmo chiederci quanto sono lunghi i segmenti, potremmo indipendentemente non allocare niente o poco e poi ri-allocare on-demand, su richiesta.

Quello che dobbiamo fare infatti sarebbe solamente cambiare la lunghezza nella tabella dei segmenti, se ci dovesse andare male dovremmo ri-allocare il segmento anche nella memoria fisica.

In alcuni casi però quando spostiamo un segmento la memoria potrebbe dover essere azzerata altrimenti potremmo avere dei leak di informazioni di altri processi.

Ripulire la memoria è però un'operazione costosa che viene fatta solo su richiesta.

Pro e contro della segmentazione:

Abbiamo tanti pro, permette di condividere codice e dati tra processi, permette di avere livelli di protezione individuali per ogni singolo segmento, posso far crescere e decrescere dinamicamente solo alcune parti.

I contro sono che se la paragoniamo alle partizioni fisse è più costosa, ha un costo hardware maggiore e non ci risolve del tutto il problema della frammentazione esterna.

A volte potrebbe richiedere di fare de-frammentazione.

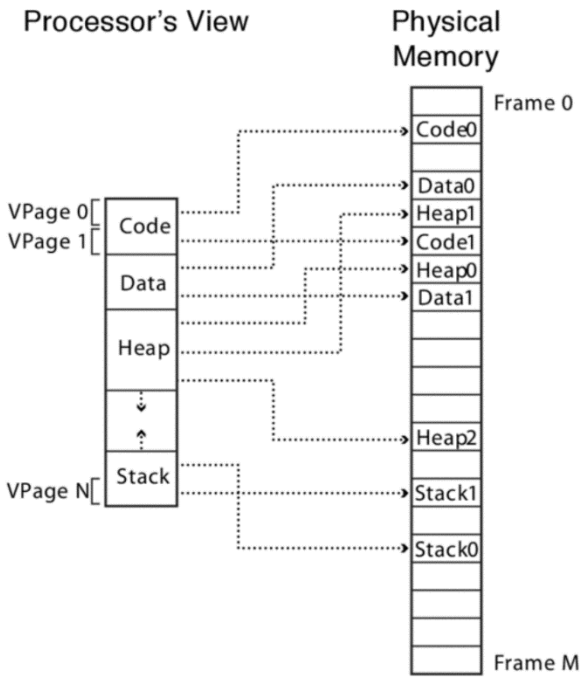
Si tende a scegliere un'altra tecnica che è quella della paginazione.

Paginazione

Le memorie logica e fisica vengono divise in segmenti, nel caso della memoria fisica si parla di page frames, questi frame vengono creati tutti della solita dimensione, tipicamente piccola, le dimensioni di una pagina sono dell'ordine di 4-8k, stiamo tornando alle partizioni fisse, in questo caso però sono tutte della stessa dimensione dunque non posso avere frammentazione esterna, potrei avere frammentazione interna, dato che però la dimensione è molto piccola la frammentazione interna è molto piccola, spreco nel caso O(k) e ce lo facciamo quindi andare bene.

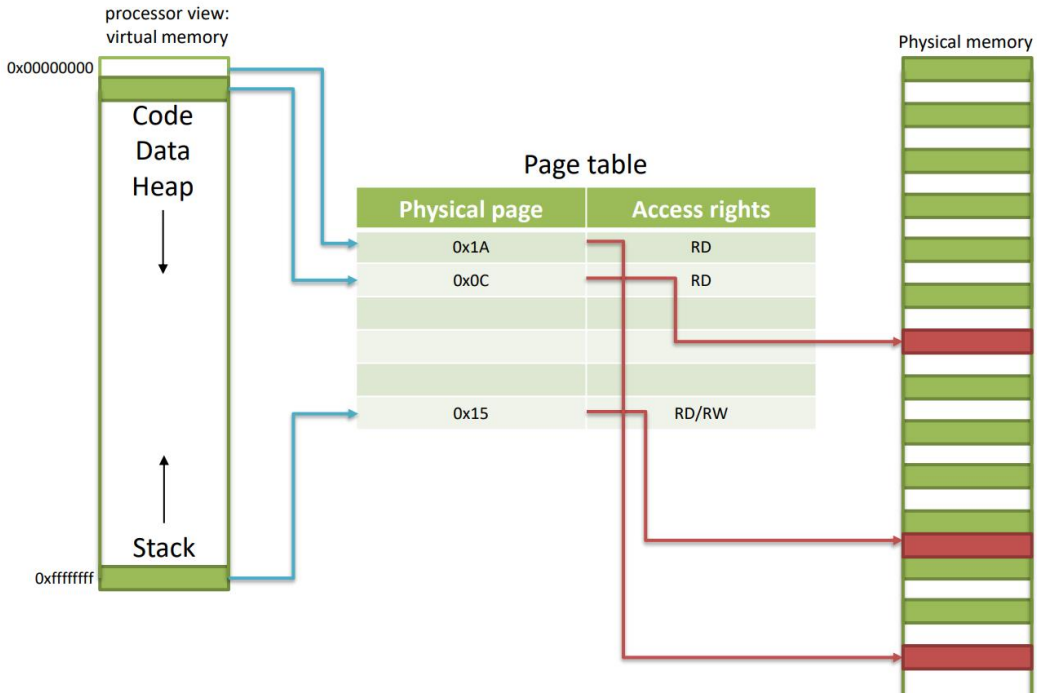
Ogni processo ha una propria tabella che si chiama tabella delle pagine, questa tabella delle pagine però non può essere caricata nella mmu, in quanto se dividiamo in piccoli pezzi tutto lo spazio logico avremo una tabella delle pagine grandissima, non possiamo quindi memorizzarla ad hardware, la teniamo in memoria.

Per risolvere il problema delle prestazioni utilizzeremo una cache nella mmu per evitare di andare tutte le volte in memoria a caricare pezzi della page table. Il problema principale riguarda la dimensione della tabella, dal punto di vista della mmu, dobbiamo tenerci solamente due valori, un indirizzo di dove risiede la tabella delle pagine del processo e la lunghezza della tabella.



Prendiamo la memoria fisica e la dividiamo in tante parti che corrispondono a pagine logiche, tutte di 4k, abbiamo creato tanti segmenti tutti uguali, non dobbiamo però mantenere la corrispondenza 1 a 1 per quanto riguarda le posizioni delle pagine logiche e di quelle fisiche.

Lo stack che prima vedevamo come un unico segmento adesso possiamo vederlo come tanti piccoli segmenti che sono le pagine che lo compongono.

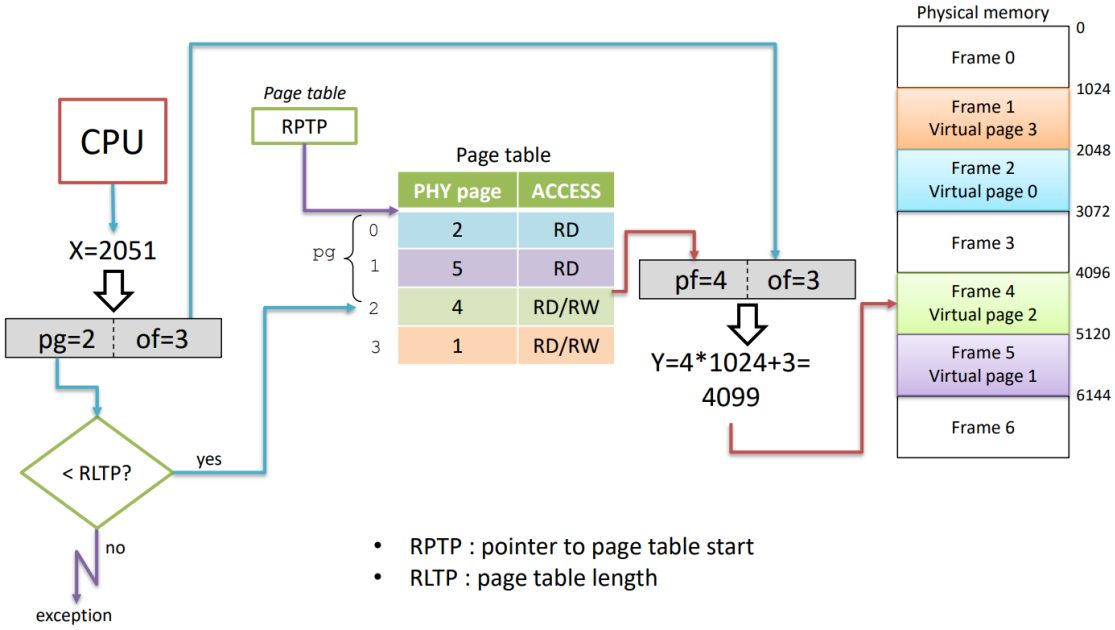


Nella tabella abbiamo una corrispondenza tra pagina logica e indirizzo del frame della pagina fisica, nella tabella oltre agli indirizzi delle pagine logiche abbiamo i diritti di accesso.

Traduzione degli indirizzi

Il processo sulla cpu emette l'indirizzo logico 2051, devo capire su quale pagina fisica va a finire questo indirizzo, assumo di avere pagine da 1kb, questo indirizzo sarà mappato nella 3° pagina e avrà offset 3, 2048+3, nella tabella delle pagine prendo la prima parte dell'indirizzo e controlliamo se esiste la entry, prendiamo la entry 2, ovvero la 3° pagina. (equivale a (0, 1, 2)) Nella entry 2 abbiamo l'indirizzo della pagina fisica che in questo caso è 4, a questo punto l'indirizzo fisico è ottenuto da 4*1024+offset, della 3° pagina devo prendere la 3° parola.

L'indirizzo fisico sarà quindi 4*1024+3 = 4099, il dato nella memoria fisica si troverà quindi nel frame 4.



Con 20 bit la tabella delle pagine è grande 2^{20} ovvero ha circa1 milione di entry, questa non è però la dimensione in byte.

Se avessimo pagine piccole da 128 byte, avremmo 2^{25}.

Se le pagine sono piccole ne abbiamo un numero maggiore, riduciamo quindi il numero di bit per l'offset ma avremo una tabella enorme.

Se le pagine sono troppo grandi rischiamo di sprecare spazio, ovvero abbiamo frammentazione interna.

Nella segmentazione dovevamo salvare la tabella dei segmenti, quella essendo piccola potevamo salvarla nella mmu. In questo caso ci serve però ricordare solo due valori, dove si trova la tabella in memoria e la sua dimensione.

Possiamo fare tutto quello che facevamo con la segmentazione, con la paginazione possiamo far partire un programma anche se non lo abbiamo caricato in memoria, possiamo iniziare a caricare le prime parti del codice, una parte dello stack e una parte dello heap, le altre parti le vado a caricare solo se vado ad eseguire un certo ramo del codice, è come se fornissi memoria on demand.

La tabella dovrà essere caricata per forza tutta in memoria, l'address space è sparpagliato.

I segmenti condivisi vengono caricati nel gap ovvero nello spazio vuoto tra heap e stack, abbiamo nella tabella indirizzi sparsi, non possiamo caricarne solo una porzione.

Il caricamento della tabella costa ordine di milioni di miliardi (varia a seconda della grandezza della parola di memoria)

La paginazione ci permette di realizzare protezione per singola pagina, questa però ha un costo che è lo spazio, *come riusciamo a risolvere questo problema?*

Utilizziamo una struttura dati anziché fissa come la tabella, con un albero, utilizziamo delle tecniche di segmentazione con paginazione, abbiamo dei segmenti e ogni segmento avrà una tabella, i segmenti possono essere creati dinamicamente in questo modo evitiamo la necessità di allocare la tabella subito, possiamo usare la paginazione multilivello combinata alla segmentazione.

Abbiamo sempre dei segmenti di memoria fissi che sono le pagine, anche chiamati page frames, cambiamo la struttura con cui modelliamo queste pagine utilizzando un approccio ad albero.

Questo approccio risolve il problema dello spazio però peggiora il costo di accesso; infatti, l'accesso ad un vettore avviene in O(1) quello ad un albero in O(logn).

Dal punto di vista delle prestazioni queste tecniche sono peggiori della paginazione pura.

Con una cache, chiamata tlb, ci memorizziamo gli indirizzi già tradotti, migliorando così le prestazioni.

Paged segmentation

L'entry della tabella dei segmenti avrà un puntatore ad una tabella delle pagine, la sua lunghezza ed i permessi di accesso.

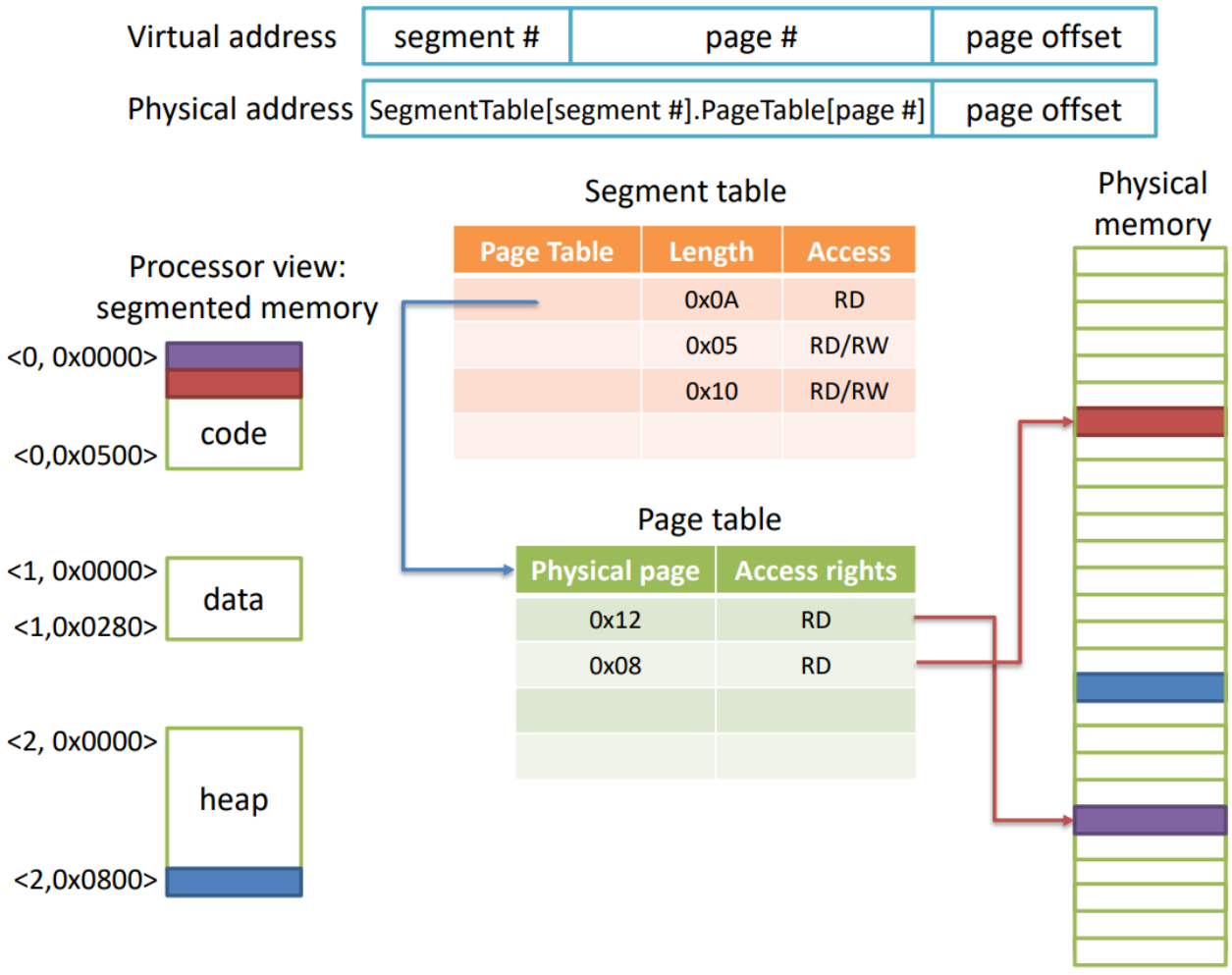
La tabella delle pagine conterrà a sua volta vari indirizzi ai frame della memoria fisica e i permessi di accesso.

Tutto quello che facevamo prima possiamo continuare a farlo.

Supponiamo una struttura divisa in 3 segmenti, ogni entry della tabella dei segmenti mi punta a una tabella di pagine differente, e ogni entry della tabella delle pagine ad un frame diverso.

L'indirizzo virtuale è fatto dunque da 3 pezzi, l'indirizzo del segmento, l'indirizzo della entry nella tabella delle pagine, e l'offset.

Devo assicurarmi che il numero del segmento sia minore del numero di segmenti totale, lo stesso per il numero della tabella delle pagine, devo controllare di avere i diritti di accesso sia sul segmento che sulla pagina.

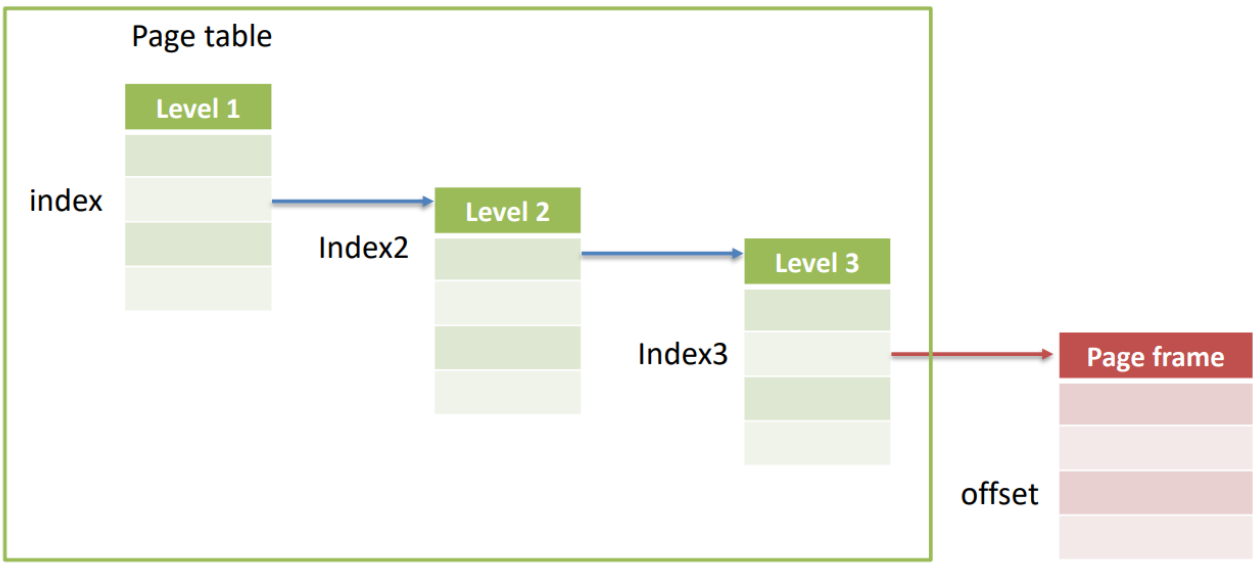


Necessitiamo di salvare la tabella dei segmenti ogni volta che facciamo content-switch.

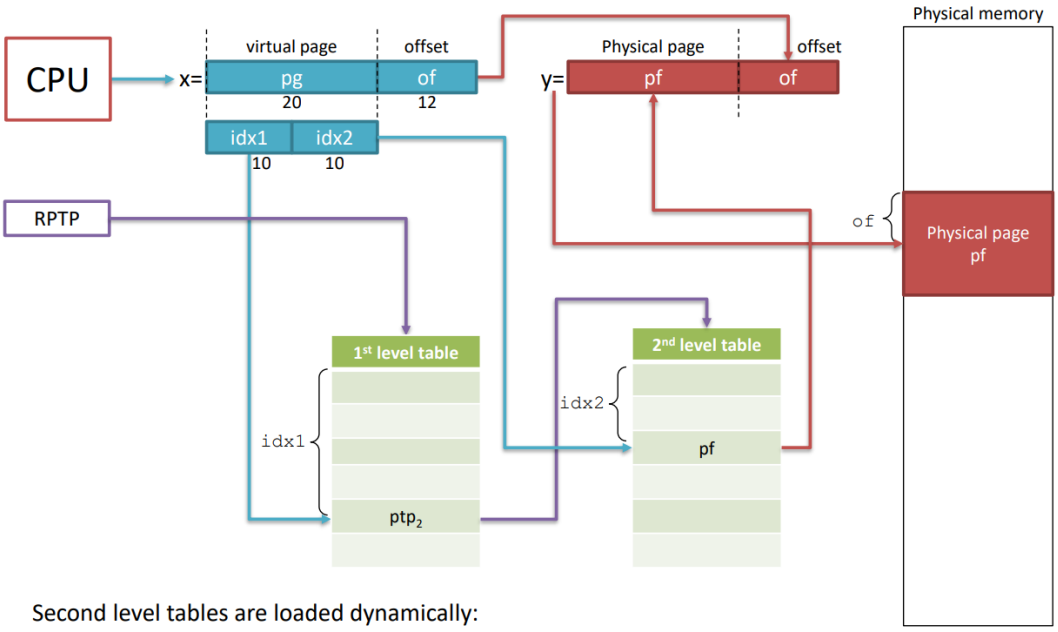
Multilevel Paging

Tecnica molto usata, in questo caso l'idea è di avere una struttura ad albero, prima avevamo una struttura ad albero a due livelli (segmenti-tabelle-frame fisici), in questo caso prendiamo l'indirizzo e lo spezziamo in più indici di tabelle delle pagine, abbiamo tanti livelli di tabelle delle pagine quanto il numero di fette in cui ho splittato l'indirizzo.

physical address=pageTable[index].pageTable[index2].pageTable[index3]|page offset



Supponiamo un indirizzo da 32 bit lo spezzo in gruppi da 8 bit, le pagine sono dunque da 256 byte, ogni tabella ha 256 entry, nel primo livello troviamo l'indirizzo base della tabella di secondo livello, prendo il bit del secondo gruppo e questo sarà l'offset da utilizzare in questa tabella delle pagine per trovare il pointer alla terza tabella a cui accederò al campo dettato dall'offset del terzo gruppo e così via fino a trovare il page frame.



Second level tables are loaded dynamically:

- reduce memory occupation

Le tabelle non ci servono tutte, ce ne servono solamente alcune, tranne ovviamente la prima che dovrà essere allocata per forza.

Concettualmente abbiamo ridotto tantissimo la dimensione in quanto rispetto alla tabella delle pagine che devo tenere tutta allocata, qua dovrò allocare solo una porzione di questo albero.

Dal punto di vista prestazionale costa di più, ci siamo però preoccupati di risolvere solo il problema dello spazio e lo abbiamo risolto, facciamo l'ipotesi di avere indirizzi da 32 bit e pagine logiche e fisiche di 4k, con 12 bit per l'offset supponiamo che ogni entry fosse 4 byte, ci sarebbero serviti 4mb.

Il vantaggio è che possiamo, a partire dalla tabella del secondo livello utilizzarne solo un sottoinsieme, il risparmio è legato al fatto che invece di caricare tutte le tabelle ne carichiamo probabilmente molte meno.

Comparing page table size

- Hypothesis:
  - 32 bit address;
  - logical and physical pages of 4K (→ 12 bit for the offset)
  - Page table entry of 4 byte
    - 3 bytes for block address
    - 1 byte for flags
- Page table size one single page level:
  - Number of table entries: 2^{20}
  - Space in bytes: $2^{20} * 4 = 2^{22}$ (4MB)
- Max physical memory size:
  - 3 bytes for the block address means 2^{24} pages → 64GB
- Page table size two-level paging (same hypothesis):
  - Of the 20 bits, 10 bits are for index1 and 10 bits for index2
  - We have 2^{10} second level tables
  - First and second level page table size is $2^{10} * 4 = 2^{12}$ (4K)
  - Best case: only 1 table in the second level → size = 8 K
  - Worst case: all tables in the second level → size = 4K + 4M
  - In general 4K + n*4K where n is the number of second level tables
- Max physical memory size:
  - 3 bytes for the block address means 2^{24} pages → 64GB

La segmentazione con paginazione multilivello è quella storicamente usata da x86, in particolare abbiamo dei valori tipici in questa architettura di pagine di 4k, in caso di architetture a 32 bit per ogni segmento si utilizzano due livelli di paginazione, in caso di architetture a 64 bit abbiamo 4 livelli per ogni segmento.

I pro e contro della traduzione multilivello:

I pro sono che ci permette di poterla implementare in modo poco costoso, si ha poca logica hardware aggiuntiva da mettere nella mmu.

I contro di questo è che ci servono due o più accessi.

Portabilità:

Per quello che riguarda la traduzione degli indirizzi facciamo affidamento al fatto che esista questo meccanismo nella mmu, ci serve però sapere cosa caricare dentro questa mmu, più siamo a basso livello più c'è il problema della portabilità, non vogliamo una traduzione degli indirizzi specializzata per ogni architettura.

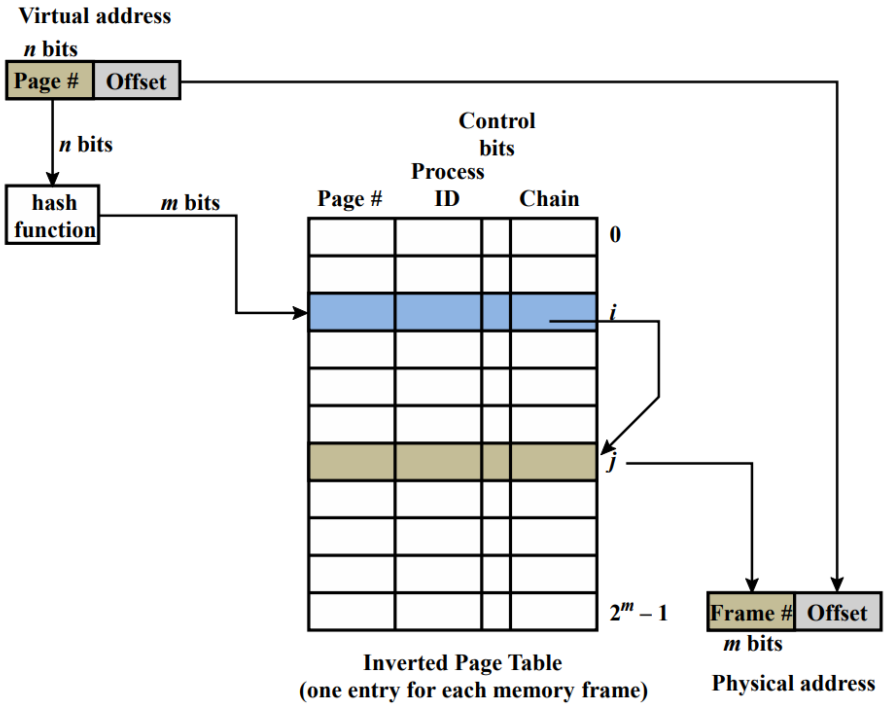
Il SO di per sé ha anche lui necessità di fare traduzione di indirizzi, un esempio è quando dobbiamo gestire le syscall. Durante il passaggio da user a kernel, infatti, si fa la copia dei dati da user space a kernel space, abbiamo dunque una sorta di traduzione dell'indirizzo a software.

Sicuramente il SO deve mantenersi delle strutture dati per fare la traduzione da indirizzo virtuale a fisico.

Ci sono un certo numero di strutture dati lato SO che permettono di fare la traduzione degli indirizzi, abbiamo ad esempio la lista dei segmenti.

Il SO tiene una struttura dati che si chiama inverted page table che è una tabella hash all'interno del kernel che permette di fare la traduzione facendo l'hashing su una parte dell'indirizzo virtuale, se il SO ha bisogno di fare una traduzione a software dell'indirizzo da logico a fisico utilizza questa tabella, si chiama inverted perché è possibile fare anche l'operazione al contrario, possiamo passare dall'indirizzo del frame fisico all'indirizzo del frame logico.

In generale questa tabella hash ha tante entry, ogni entry è proporzionale al numero delle pagine fisiche, si ha un numero fisso.



Quando il SO deve trovare l'indirizzo logico o fisico prende un certo numero di bit che rappresentano il numero di pagina, vi applica una funzione di hash e trova la entry corrispondente.

Perché necessitiamo anche della tabella delle pagine?

Potremmo fare tutto a software tramite questa inverted page table, i problemi sono che ogni volta che dovrei trovare un nuovo indirizzo dovrei chiedere al SO un servizio, cosa che implica content switch, la gestione a software per quanto efficiente richiede la gestione delle liste di trabocco che richiede tempo.

Traduzione efficiente:

Come facciamo a velocizzare i lookup multipli che dobbiamo fare?

Per come è fatta la traduzione degli indirizzi è facile rendersi conto che per quanto riguarda l'accesso alle pagine c'è molta probabilità di avere località spaziale, in quanto tutti gli indirizzi all'interno di una pagina si traducono ad uno stesso modo.

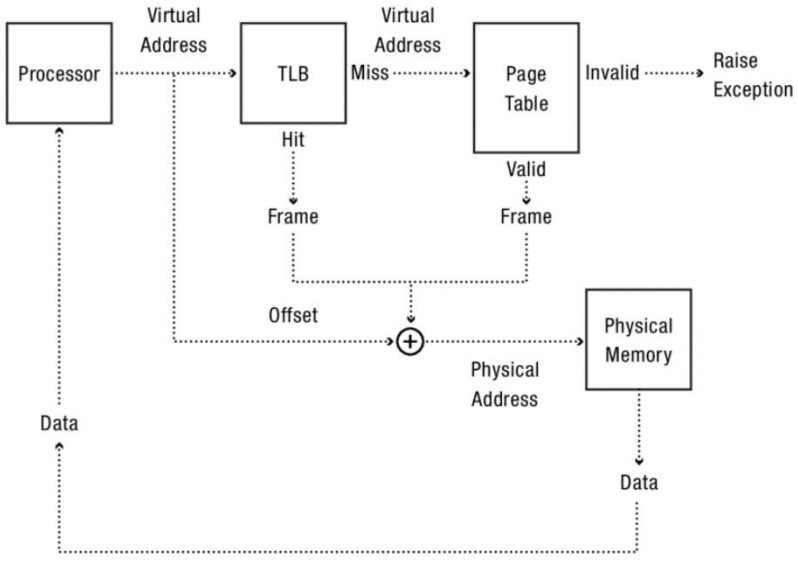
Se conservo in una cache i risultati della traduzione dato che vale il principio di località sappiamo di avere un grande vantaggio, solo in caso di miss dovrò fare nuovamente la traduzione, in realtà c'è anche località temporale, perché spesso tendiamo a riaccedere a dati che si trovano nello stesso insieme delle pagine.

Sfruttiamo quindi una cache che anziché contenere i dati, contiene l'associazione indirizzo virtuale – indirizzo fisico, abbiamo una cache di indirizzi anziché una cache di dati, questa cache si chiama Translation Lookaside Buffer (TLB).

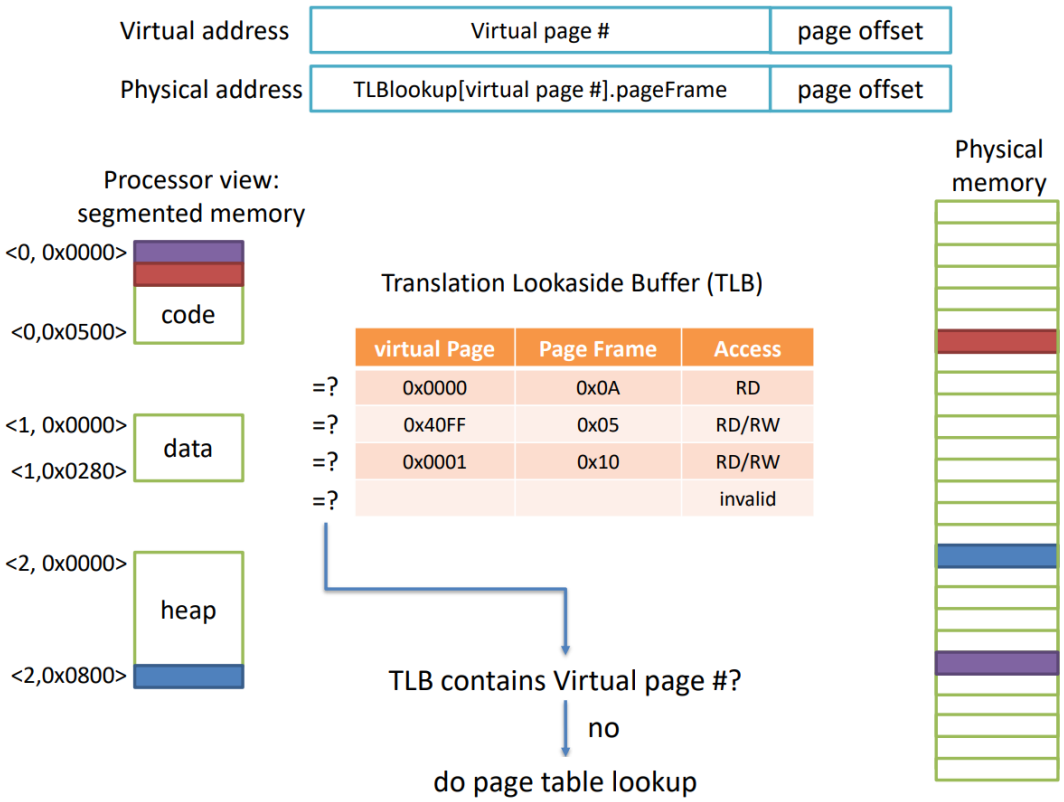
Si tiene traccia di tutti attraverso una cache, di solito è realizzata in modo totalmente associativo, non ha tantissime entry, 16-512, all'interno di un blocco abbiamo 1-2 page table entries.

Tipicamente il valore di miss rate è basso, sotto all'1%

Il problema è il costo, ci costa logica di spazio nel chip per implementare questa cache.



$$Cost\ of\ Address\ Translation = TLB_{hit} + TLB_{miss} * CostFullTranslation$$



Spesso l'mmu viene chiamato tlb, in quanto gran parte della mmu serve per la realizzazione della tlb.

Come viene caricata questa cache?

Visto che i cache hit sono alti potrei usare la tlb e quando ho un miss usare l'inverted table, anziché passare per le page table. Questa tecnica veniva utilizzata molto tempo fa.

Dal punto di vista del costo non è banale, si ha un costo più alto, perché anche per quell'1% di fault che abbiamo dobbiamo fare content-switch e gestire la nostra tabella hash, questa tecnica non è conveniente.

Si preferisce oggi avere la tlb e la mmu che fa la traduzione totalmente ad hardware, quando è il So ad averne bisogno la fa a software.

Il tlb come tutte le cache funziona bene solo se c'è località spaziale e/o temporale, meglio se tutte e 2.

Esempio:

Immaginiamo di dover fare una computazione su un frame video molto grande, ogni riga buffer è grande quanto o più di una pagina, supponiamo di fare un accesso per colonna, stiamo accedendo sempre a pagine diverse, non stiamo sfruttando né località spaziale né temporale, in questo caso il TLB non funziona, cadiamo sempre nel caso di fault.

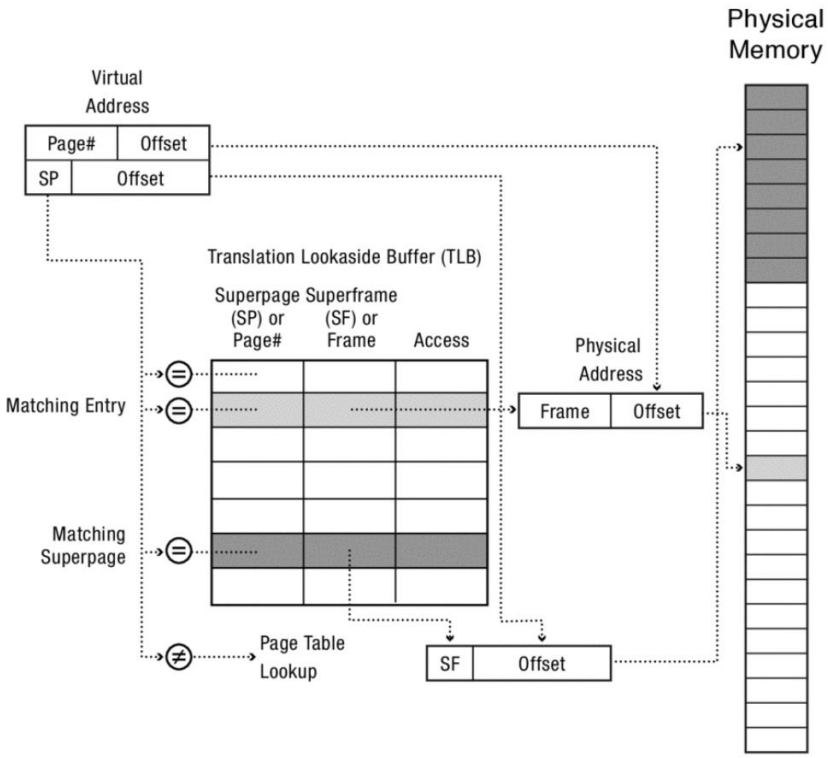
Superpagine

Questo ha però una soluzione, le superpagine (o Huge pages), anziché allocare pagine da 4k allochiamo un insieme contiguo di pagine, nell'x86 possiamo chiedere superpagine di 4k, 2mb, 1G, questo risolve il problema in quanto abbiamo meno entry e riusciamo a metterle tutte nel tlb.

Quando accedo ai singoli elementi potrei avere anche località spaziale grazie al fatto che si trovino in un'unica superpagina.

Questo implica però una TLB più sofisticata e devo in qualche modo distinguere la dimensione della pagina.

Queste pagine vengono settate tramite delle chiamate di sistema, questo fa sì che il SO informi la mmu e il TLB in modo tale che al suo interno vengano scritte superpagine della dim voluta.

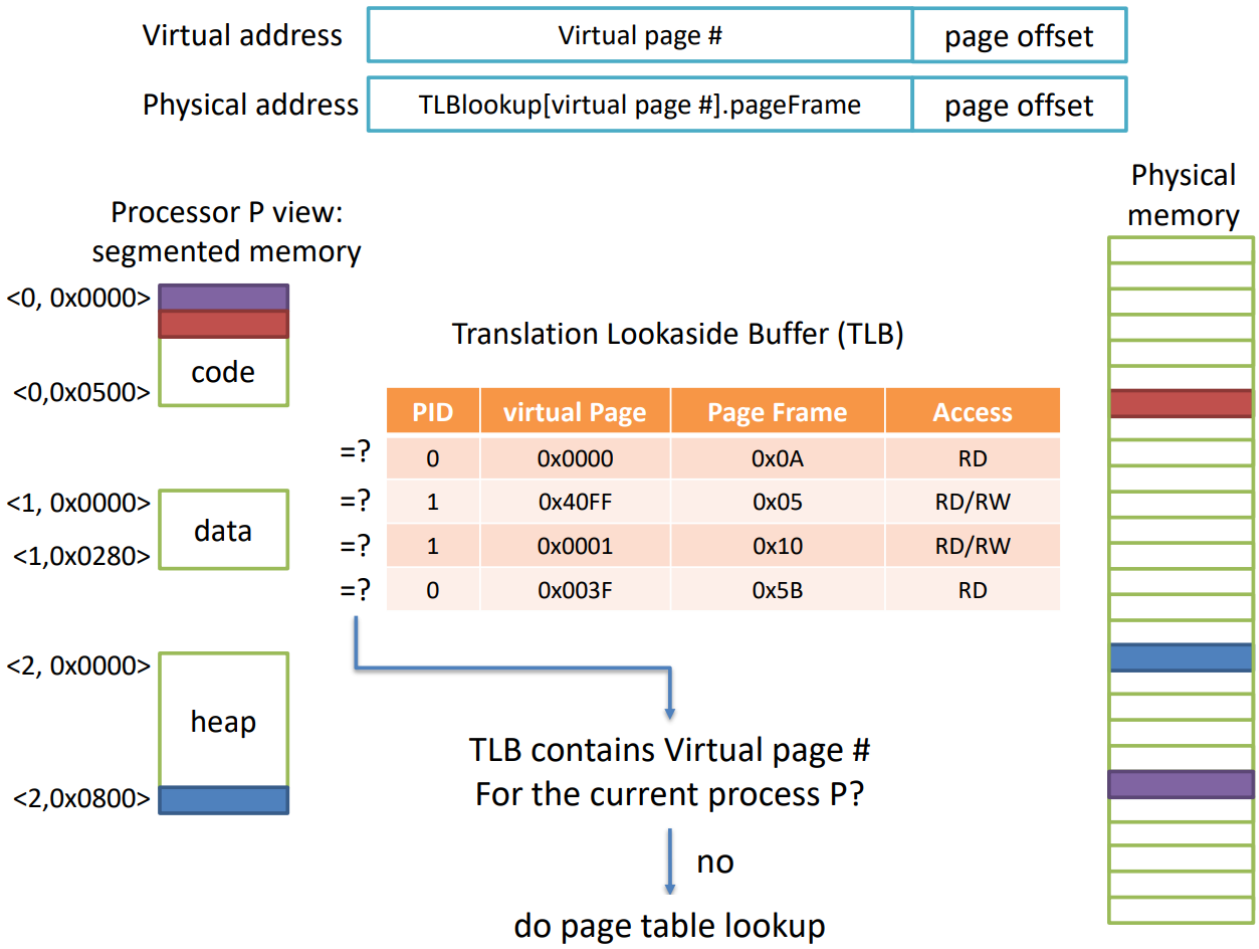


L'idea è di compattare le informazioni che salvo nel TLB.

Spesso accade che in alcuni programmi che si comportano in questo modo si hanno tanti miss nel tlb.

Due processi diversi possono indirizzare lo stesso indirizzo virtuale ma in spazi di indirizzamento totalmente disgiunti.

Potrebbe accadere che nella tlb abbiamo un indirizzo logico che apparteneva al processo precedente prima di fare content switch, non vogliamo invalidare tutta la tlb, vogliamo taggare le entry, ad ogni entry associamo il pid del processo, in questo modo possiamo utilizzarlo per capire se la nostra entry è valida oppure no.



Che cosa accade quanto il sistema operativo cambia i diritti di una pagina?

Cosa succede quando questo avviene in un sistema multicore?

In un sistema multicore ho TLB distinte per ogni core, in quanto la traduzione viene fatta per core, dobbiamo però tenere consistenti le entry della cache, ogni volta che cambio un permesso in una tabella, lo devo riportare nella tlb che deve aggiornare le altre tlb sugli altri core, entriamo in quello che viene detto TLB shootdown.

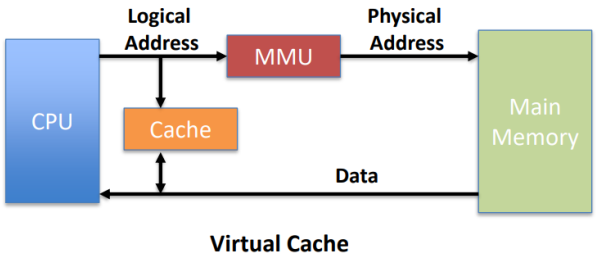
Per mantenere consistente la cache dobbiamo aggiornare tutte le TLB degli altri core, se questo avviene di frequente ci costa molto, se è frequente può avere un impatto prestazionale.

Il fatto di avere un TLB ci dà un guadagno, ma ci aggiunge anche degli overhead di gestione.

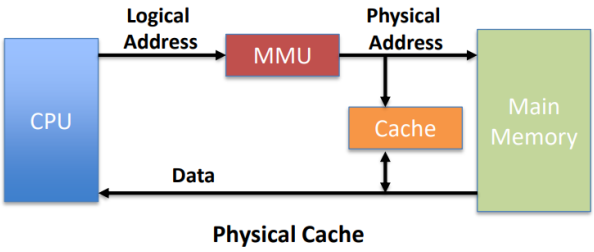
L'architettura con TLB possiamo immaginarcela con due tipi diversi di cache, possiamo avere cache che indirizziamo con indirizzi logici, ovvero in cui indirizziamo la cache prima di aver fatto la traduzione degli indirizzi. Necessitiamo di taggare però le entry della cache perché potremmo avere stessi indirizzi su thread diversi, è più veloce perché non faccio la traduzione dell'indirizzo da logico a fisico.

Oppure potrei avere una cache fisica, in cui memorizzo indirizzi fisici, lo faccio dopo aver fatto la traduzione degli indirizzi, in questo caso la cache che opera con indirizzi fisici, è più semplice in quanto non necessitiamo di taggare le entry però dobbiamo fare per forza la traduzione degli indirizzi, che eventualmente farò con la TLB che si trova dentro la mmu.

- Virtual cache faster because the cache can respond before the MMU performs an address translation
- But it requires to tag cache entries (same virtual address space for processes), thus more complexity and more space needed

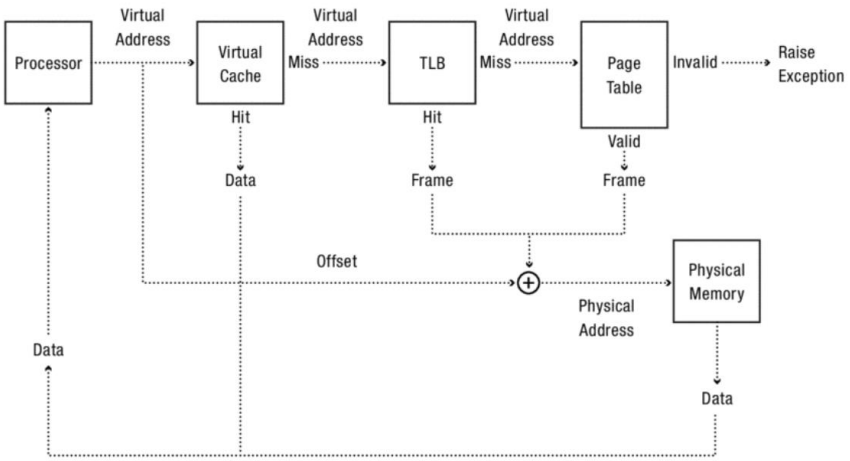


Virtual Cache

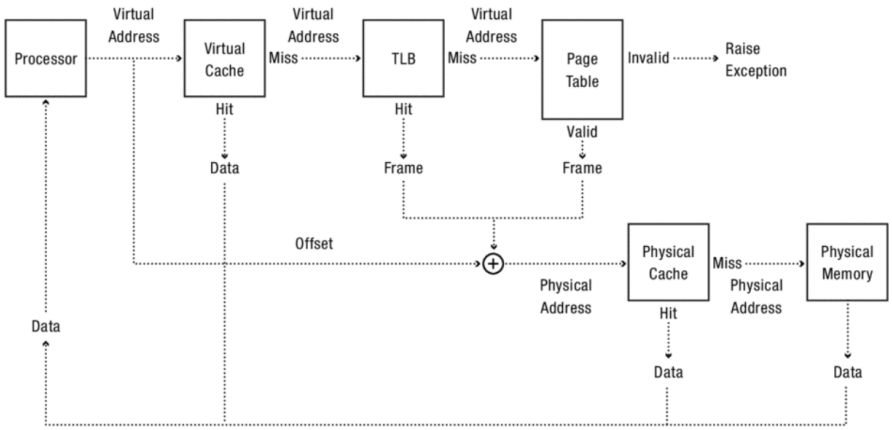


Physical Cache

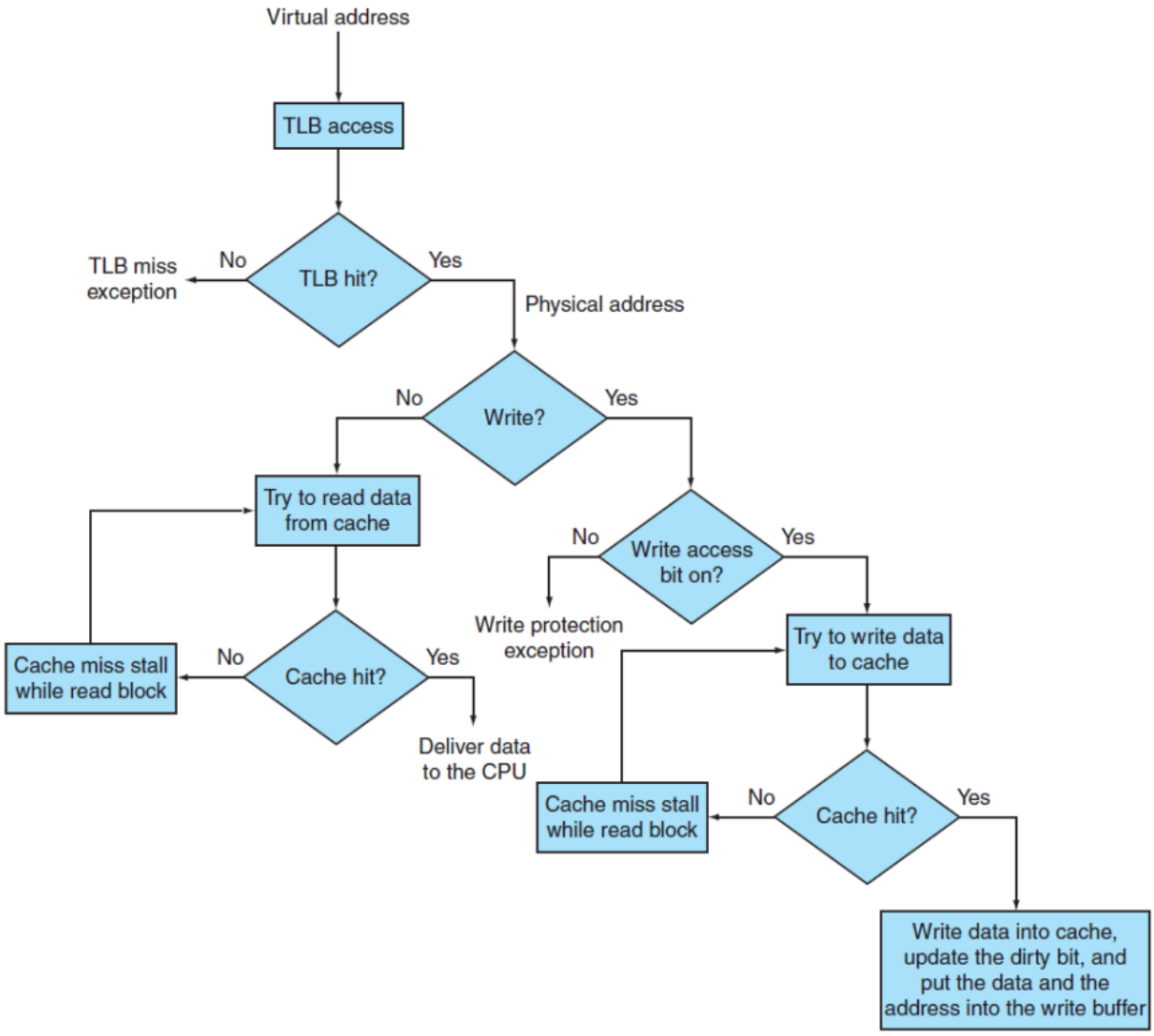
Virtually addressed Cache



Physically addressed Cache



TLB and Cache Interaction



Memoria virtuale

Per quanto riguarda la memoria il So fornisce l'astrazione di avere una sorta di memoria più grande rispetto a quella effettiva, la memoria fisica può essere più piccola della memoria logica.

Questa illusione che il So dà ai processi si chiama memoria virtuale, in generale la memoria virtuale è pari almeno alla memoria fisica più il livello disponibile sul livello della gerarchia di memoria più alta, il disco.

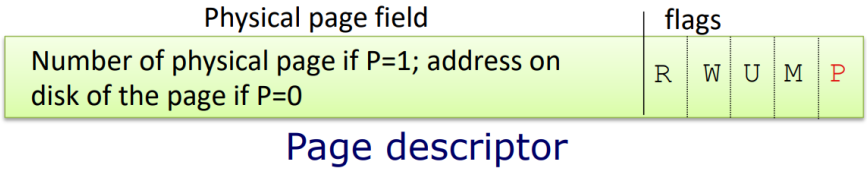
La memoria effettivamente disponibile non è solo la memoria fisica ma anche l'area di swap. (Approfondimento qui)

In qualche modo la memoria principale è solo uno dei livelli della gerarchia di memoria, la possiamo vedere come una sorta di cache i cui blocchi sono le pagine che abbiamo sul disco.

Tutte le pagine dei processi che vogliamo eseguire non ci stanno, buona parte starà sul disco, il So operativo le caricherà e scaricherà on-demand.

In questo caso la gestione è fatta totalmente lato software dal sistema operativo.

Per implementare questo meccanismo di caricamento su domanda, riconsideriamo una entry della tabella delle pagine.

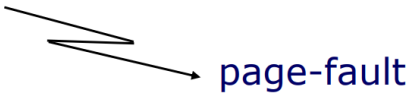


Page descriptor

The page table contains a page descriptor for each page
Beyond the information for translation of the address, the descriptor contains some flags:

- R, W: read/write access rights
- M, U: modified/use bits (for the page replacement algorithms)
- P: presence bit (page is invalid if not present in main memory)

- P = 1: page in main memory
- P = 0: page not in main memory



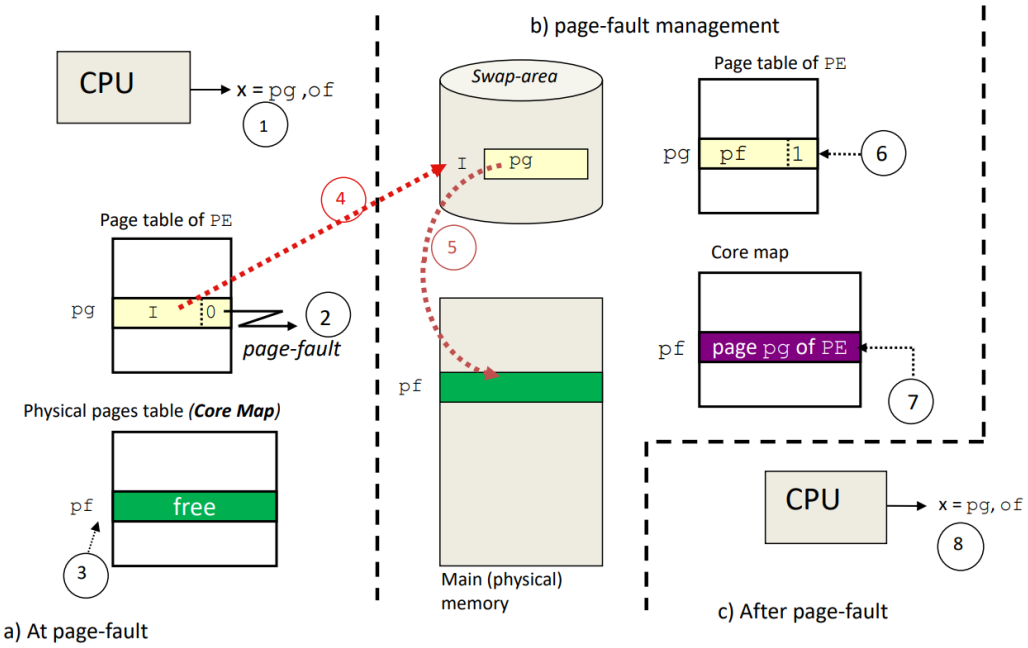
In una entry, supponendo per semplicità una tecnologia di paginazione ad un livello, ogni entry è un certo numero di byte in cui un sottoinsieme di bit è utilizzato per i flag, i restanti byte/bit sono utilizzati per codificare l'indirizzo della pagina fisica. Se la pagina non è in memoria possiamo immaginarci che i byte che codificano la pagina codifichino invece la posizione della pagina su disco.

Si distingue tra indirizzo in memoria e indirizzo sul disco grazie ad un bit, questo bit flag, se è a 1 indica che la pagina è presente in memoria, se invece è 0 è presente sul disco.

Gli altri flag sono i bit di protezione, anche se ce ne sono altri come ad esempio M, U dove M sta per modified, e U sta per used se la pagina è in uso / è stata usata da almeno un processo.

Se abbiamo una memoria fisica piccola, può capitare che non ci sia spazio per caricare dal disco altre pagine (che stiamo cercando di riferire). Esistono algoritmi per scegliere quale pagina sostituire. Nelle cache avevamo random e LRU (Last recently used).

Alcuni bit mi potrebbero essere utili per decidere quale pagina rimuovere quando ne dobbiamo rimuovere una (pagina vittima).



La cpu emette un indirizzo logico, formato da indx pagina logica e offset, andiamo a vedere nella tabella delle pagine e troviamo il bit di presenza a 0, l'indirizzo che stiamo riferendo è i, la pagina non è dunque presente, andiamo a cercare un page frame libero ovvero una pagina fisica libera dove andrà caricata la pagina logica che si trova sul disco all'indirizzo i, supponiamo ci siano pagine fisiche libere, a questo punto visto il bit di presenza a 0 viene generato un page fault, che sarebbe una sorta di eccezione/interruzione che manda in esecuzione il SO, il quale invocherà un handler che si occuperà del caricamento della pagina. Recupererà in qualche modo dal disco la pagina P_g all'indirizzo i, chiederà al driver del disco di caricare la pagina all'indirizzo i, magari attraverso dma, la procedura partirà, il processo verrà sospeso e verrà risvegliato l'handler quando il dma controller lancerà l'interruzione di pagina caricata in memoria, a questo punto l'handler potrà settare il bit di presenza ad uno, avremo nel page frame l'indirizzo giusto e potremmo risolvere l'indirizzo iniziale da indirizzo logico a fisico. Perché abbiamo caricato nella pagina fisica la pagina logica che stavamo riferendo.

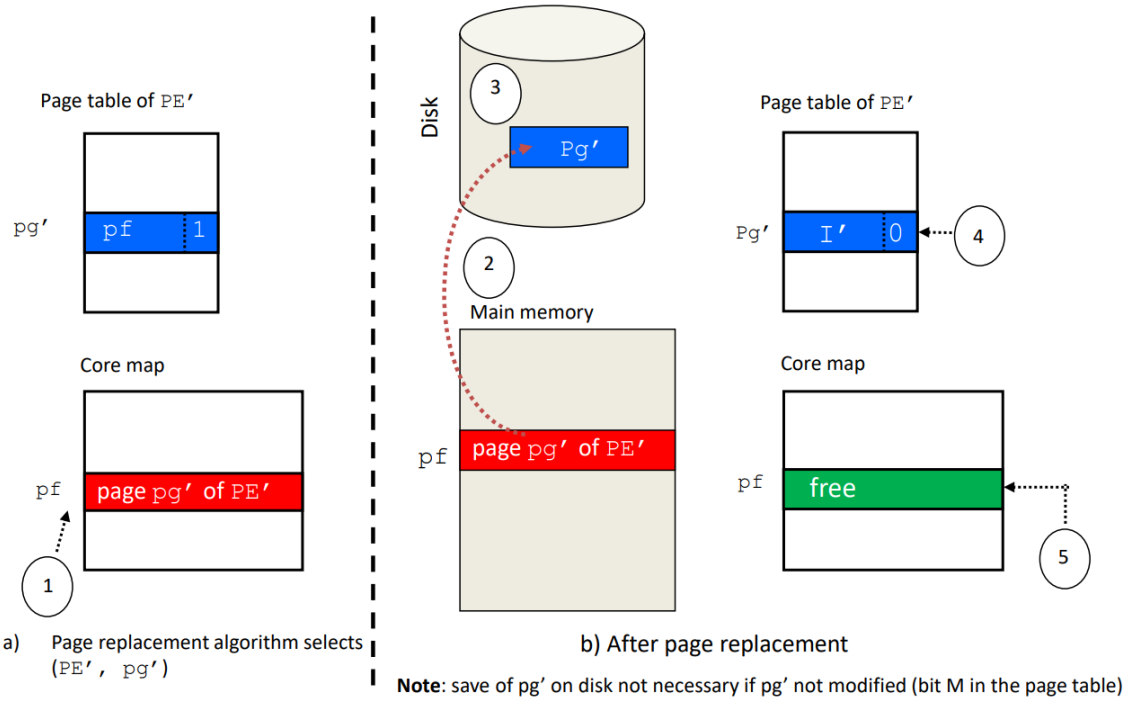
Si parla di *core map*, noi l'abbiamo chiamata *inverted paged table*, sono in realtà la stessa struttura, la chiamiamo inverted page table quando facciamo la traduzione degli indirizzi a software, se invece parliamo di traduzione per indirizzi per i sistemi moderni che avviene ad hardware attraverso il tlb e in generale l'mmu, allora la struttura dati che utilizza il kernel la chiameremo core map, questa core map è come l'inverted page table, ovvero tiene traccia per ogni pagina fisica di quali processi riferiscono la data pagina ed eventualmente contiene altre informazioni come ad esempio: se la pagina è libera o occupata, se è readOnly o no ecct…

In questo schema logico la core map viene utilizzata per trovare una pagina fisica disponibile, avevamo visto che l'indirizzo P_f era libero perché c'era il bit del flag settato ad 1, quando lo abbiamo caricato in memoria abbiamo settato il bit a 0, e ci segniamo anche quale processo lo sta utilizzando.

Se ci fosse il tlb nello schema che abbiamo visto non cambierebbe molto.

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

Cosa succede se non abbiamo più pagine fisiche disponibili?



Dobbiamo trovare una pagina vittima da rimuovere, ci serve una procedura che ci permetta di selezionare una pagina opportuna tra quelle occupate.

Se ne ho già una copia sul disco e non l'ho modificata posso rimuoverla dalla core map e lasciare inalterata la copia, altrimenti se la pagina è stata modificata devo ricaricarla sul disco, dopo questo posso considerare la pagina libera.

La pagina potrebbe essere condivisa tra più processi, allora in ogni tabella delle pagine di ogni processo che la riferisce devo settare il bit di presenza a 0, tutta questa procedura è un'operazione molto costosa, dobbiamo fare in modo, in generale di avere le pagine che riferiremo presenti in memoria il più possibile.

Se ci troviamo su multi core abbiamo molti tlb, su core diversi potremmo avere entry della tlb che riferiscono la pagina che stiamo rendendo invalida, dobbiamo dunque invalidare anche tutti i tlb che contengono quella entry, visto che la traduzione degli indirizzi passa sempre per il tlb se non la invalidiamo potrebbe pensare che ci sia ancora.

Come facciamo a sapere se una pagina è stata modificata?

Nei sistemi moderni il bit M (modified) viene settato ad hardware dall'mmu, ogni volta che facciamo una store, l'mmu lo setta nel tlb e nella page table, ci deve essere consistenza tra le entry del tlb e le entry delle page table, dobbiamo avere un algoritmo che garantisca che non andiamo a prendere cose che non sono ancora consistenti.

Se la pagina è usata, usarla è diverso da modificarla, se la leggo la sto usando, quindi non devo avere consistenza solo per ogni store ma anche per ogni load. Facendo una load se leggo un byte di quella pagina devo settare il bit U ad 1, anche questo nei sistemi moderni viene fatto ad hardware sia nel tlb che nella tabella delle pagine.

Nei sistemi meno recenti la gestione veniva lasciata a software da So che dunque deve avere la possibilità di modificare questi bit U ed M, cosa che deve essere fatta ad ogni accesso alla pagina, magari non per singolo accesso ma dovrà tenerne traccia.

Ovviamente siccome se ne occupava il SO avevamo content switch ogni tot accessi e quindi overhead.

Un modo per emulare i bit M ed U è ad esempio per il bit M settare tutte le pagine che non abbiamo modificato a read only, non appena tentiamo un accesso in scrittura verrà generato un fault, partirà il SO, nella core map ci siamo riportati che quella pagina fosse stata forzatamente settata a read only, e riconoscendola setteremo il bit M ad 1.

Per il bit U segniamo tutte le pagine come invalide, non appena tentiamo un accesso verrà scatenato un fault.

Il problema del trashing lo avevamo già incontrato per le cache ad hardware, potremmo andare a caricare su disco una pagina che ci servirà dopo poco, potremmo innescare un circolo in cui stiamo andando a scaricare e caricare continuamente sul disco e dal disco le pagine, questo fenomeno è il trashing in cui il SO spende più tempo a fare questa gestione che a permette l'esecuzione delle applicazioni.

Diventa critico scegliere le pagine giuste, il punto è cercare di tenere in cache il working set, in questo caso l'insieme delle pagine riferite di più, vogliamo massimizzare il cache hit rate.

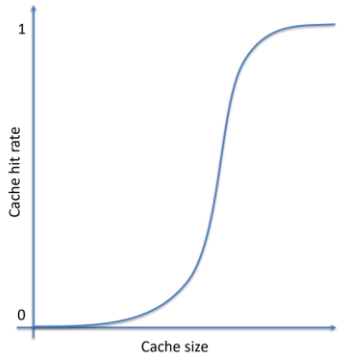
Se teniamo una gran parte del working set in memoria abbiamo un altissimo cache hit e questa operazione di swap la facciamo raramente.

Sappiamo anche che in qualche modo il working set dipende anche dalla dim della cache, se abbiamo cache molto piccole è probabile che avremo molti miss.

Aumentando la dimensione della memoria fisica posso attenuare questo problema, il problema principale è però che il working set non è costante, esso varia infatti in base all'applicazione e ai campi di contesto, quando un processo si sospende ne va in esecuzione un altro che magari ha bisogno di meno pagine o semplicemente di pagine diverse.

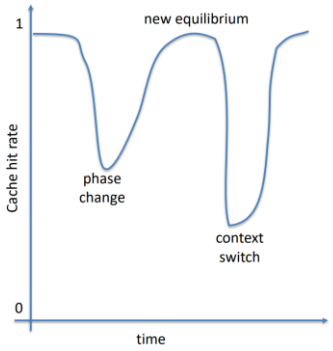
Working set

- Working Set: set of memory locations that need to be cached for reasonable cache hit rate
- Thrashing happens when system has a too small cache



Phase Change Behavior

- Programs may change their working set over time
- Context switches also change working set



Possiamo vedere qui come l'hit rate diminuisca a causa di un cambiamento del working set e come crolli in seguito a un content switch.

Replacement Policy

Cosa succede una volta che abbiamo scelto quale pagina vittima rimpiazzare?

Scegliamo prima in qualche modo la pagina da rimpiazzare, poi cerchiamo questa pagina in tutte le tabelle delle pagine, in seguito andrà settata come invalida, conseguentemente vanno rimosse tutte le sue occorrenze nella TLB ed infine, solo se necessario, dobbiamo scrivere le modifiche su disco.

Ecco un riassunto dei passi:

- 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
  – i.e. if the page had been modified

L'obbiettivo è ridurre i cache misses, tenendo quello che ci serve in memoria centrale e non sul disco.

Come scegliamo la nuova pagina da rimpiazzare?

Scegliamo quella che sarà la più probabile che non verrà utilizzata.

Avevamo accennato per le cache totalmente associative, che uno tra gli algoritmi possibili era quello random, in questo caso non è una buona scelta anche se possiamo utilizzarlo come benchmark.

Questo è dovuto soprattutto al fatto che l'overhead è 0, non è una buona idea usarlo nella pratica perché non tiene conto né della località e né del working set.

Una valida alternativa potrebbe essere fifo, possiamo selezionare come vittima la pagina che è in memoria da più tempo.

Può essere una possibile soluzione, ma non è detto che la pagina che è lì da tanto tempo sia quella meno riferita.

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 when program strides through memory that is larger than the main memory

- working set larger than cache capacity

Qui possiamo osservare il fenomeno del trashing con il FIFO.

Supponiamo con un algoritmo fifo una cache di capacità 4, con 4 pagine, e di riferire le pagine abcde, ovvero di riferire 5 pagine, abbiamo un numero di pagine logiche maggiore di quelle fisiche.

Algoritmi di rimpiazzamento

Discuteremo 3 algoritmi, il primo *min* è un algoritmo ottimo, ideale, rimpiazza la pagina che non sarà usata per il tempo maggiore, cerca di stimare il futuro, non è un algoritmo implementabile, stiamo rimpiazzando la pagina che useremo più lontano nel tempo tra quelle che ho, si può provare che questo algoritmo è ottimale.

Se riusciamo a predire quali saranno gli accessi posso organizzarli per minimizzare i fault, questo ci serve come algoritmo di riferimento per stimare la bontà degli algoritmi.

Random lo utilizziamo come benchmark perché è un algoritmo che ci costa poco dal punto di vista della complessità computazionale, quante operazioni abbiamo bisogno di fare rispetto all'algoritmo più semplice possibile, min lo utilizziamo dal punto di vista della capacità di minimizzare i page fault.

LRU è un algoritmo molto buono, rimuoviamo la pagina che è stata usata meno di recente, questo approssima min.

NRU

Abbiamo poi i *not recently used* che sono una classe di algoritmi, lru è un po' costoso da implementare perché necessitiamo di avere un concetto di tempo per ogni accesso, se faccio più accessi a parole di una stessa pagina devo ricordami il tempo dell'ultimo accesso che ho fatto, rilassiamo quindi ancora di più l'approssimazione di min utilizzando il concetto di not recently used, ovvero una pagina non recentemente usata, non sarà la migliore ma sappiamo che non è stata utilizzata di recente, questa classe tenta di approssimare lru che a sua volta approssima min, alcuni di questi sono *second chance* e *working set algorithms*.

Semplificando, il concetto dietro gli nru è che azzeriamo tutti i bit di riferimento, poi mano a mano che usiamo le pagine li settiamo ad 1 e dopo un intervallo di tempo guardiamo quali sono ancora a 0, cioè non sono state più usate e quindi le scelgo come vittime.

Vediamo un esempio di lru/min

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

reference	A	B	C	D	E	A	B	C	D	E	A	B	C	D	E
1	A					+					+				
2		B					+					+	C		
3			C					+	D					+	
4				D	E					+					+

Il concetto è che se c'è località temporale lru approssima min, mentre fifo è peggiore di lru.

È stato dimostrato che in realtà anche aumentando la dimensione della cache, per fifo i cache hit non aumentano, non è detto che fifo si comporti bene anche se ho tutto il working set, in quanto non tiene conto di nessuna località.

Vediamo questo esempio costruito ad hoc dove abbiamo località temporale.

Se c'è località di riferimenti (i + sono gli hit della sequenza).

Facciamo vedere che LRU e min coincidono mentre fifo con questa sequenza ha un numero di cache hit più basso, fifo non si comporta bene mentre LRU tende a min.

Sebbene questo sia un esempio costruito ad arte LRU approssima veramente min, mentre fifo mi dà meno cache hit di LRU.

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

reference	A	B	A	C	B	D	A	D	E	D	A	E	B	A	C
1	A		+				+		E			+			
2		B			+						A			+	
3				C									B		
4						D		+		+					C

MIN

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

Cosa succederebbe se aumentassimo la dimensione della cache?

Fifo potrebbe funzionare meglio, anche se ovviamente se riuscissimo a mettere tutto in cache tutti funzionerebbero bene.

In realtà però è stata dimostrata la Belady's Anomaly

Fifo non è detto che si comporti bene anche aumentando la cache ovvero avendo abbastanza spazio per tutto il working set.

Perché fifo non tiene conto di nessun tipo di località.

Belady's Anomaly

FIFO (3 slots)

reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					+
2		B			A			+		C		
3			C			B			+		D	

FIFO (4 slots)

reference	A	B	C	D	A	B	E	A	B	C	D	E
1	A				+		E				D	
2		B				+		A				E
3			C						B			
4				D						C		

Il take away message è che nessuno algoritmo implementabile è perfetto, ma per ognuno c'è almeno un caso in cui non performa bene.

Altra classe di algoritmi

Second Chance

Dobbiamo utilizzare gli algoritmi che stimano lru, uno di questi è second chance, è una variante di fifo, utilizza un bit che dice se la pagina è stata riferita o no. Ordiniamo le pagine nella memoria fisica in una lista linkata e teniamo conto di questo bit riferito R, teniamo traccia in questa lista del tempo di vita di una pagina, le pagine che abbiamo però sappiamo se sono state usate di recente oppure no.

Se il bit è a 0 la prendiamo come vittima, altrimenti lo mettiamo a 0 e la spostiamo alla fine della coda.

Il problema di questo algoritmo è che necessita di un sacco di spostamenti su questa lista linkata.

Clock Algorithm

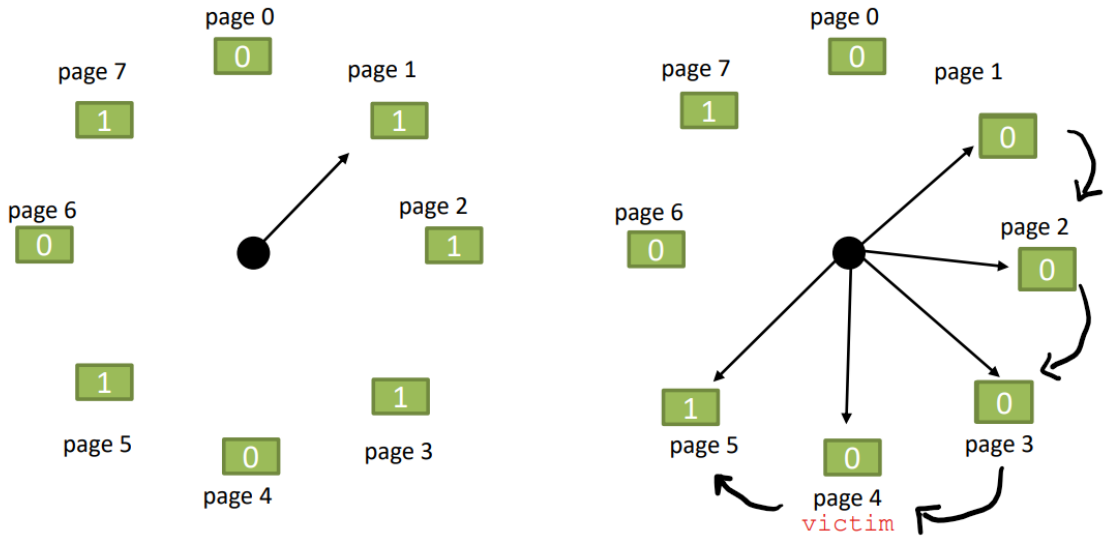
In realtà utilizziamo una variante di lru e di second chance detto clock algorithm in cui la lista linkata è una lista circolare, carichiamo le pagine che abbiamo in memoria e abbiamo una sorta di puntatore che ci indica l'ultima pagina visitata, abbiamo ancora il bit R che ci dice se la pagina è stata riferita oppure no.

L'obbiettivo di questo algoritmo non è tanto scegliere una singola vittima, ma mantenere un pool di pagine libere. Il principio alla base è sempre quello di sacrificare la pagine che non sono state usate.

I bit R saranno aggiornati all'ultima esecuzione dell'algoritmo.

Quando parte?

Potrebbe partire quando non ho più pagine libere, un altro modo potrebbe essere quello di avere questo algoritmo in esecuzione su un thread in modo da poterlo eseguire ogni tot.



Questo algoritmo cerca la pagina che ha il bit riferito a 0, ogni volta che trova una pagina con il bit R a 1 lo setta a 0, quando troviamo la pagina con il bit del riferito a 0 la seleziona come vittima e sposta il cursore alla pagina successiva per partire da lì alla prossima esecuzione.

Questo algoritmo approssima LRU perché tiene conto di un valore temporale che è tutte le volte che mandiamo in esecuzione l'algoritmo, essendo un'approssimazione può essere più o meno preciso ma è molto semplice da implementare, utilizza infatti una lista circolare, un cursore e un bit di riferito per ogni pagina.

Se dovessero essere tutti ad 1 utilizziamo una politica random.

I bit vengono messi ad 1 dall'mmu quando la pagina viene riferita.

Algoritmo Nth chance

In generale si parla anche di algoritmi Nth chance, anziché utilizzare bit che hanno valore binario utilizziamo un intero ovvero un certo numero di bit che mi dice per quante volte quella pagina non è stata riferita.

È un'evoluzione del clock algorithm, ma al posto di un singolo bit, teniamo un contatore che incrementiamo ad ogni giro e se raggiunge un valore prefissato la pagina viene scelta come vittima.

È come dire do N chance ad una pagina prima di essere rimossa, se la pagina è usata settiamo il puntatore a 0, altrimenti se la pagina non è stata usata ma è minore di un N fissato continuo ad incrementarlo, altrimenti se sono arrivato al valore massimo e incontro una pagina che ha questo contatore a quel valore quella è la vittima.

```
if (page is used) {
   notInUseSince = 0;
} else if (notInUseSince < N) {
   notInUseSince++;
} else {
    reclaim page;
}
```

Prima di scaricare una pagina aspettiamo di non usarla per un bel po', aspettiamo di fare il giro dell'orologio un po' di volte, quando arriviamo all'ultimo giro e la pagina non è stata mai usata, allora non è stata usata per tanto tempo ed è quindi una buona vittima.

Anche qua potrebbe succedere che tutte sono usate e noi prenderemo quella meno usata.

Quali pagine considero per l'eliminazione? Quelle globali o del processo in esecuzione?

Nell'algoritmo globale consideriamo la pagina da rimuovere dalla memoria fisica dove abbiamo le pagine di tutti i processi, nel secondo caso parliamo di algoritmo locale e consideriamo il processo in esecuzione, la vittima la scegliamo tra le pagine del processo, prenderemo quella meno riferita o non recentemente riferita.

Nel caso di algoritmi globali abbiamo un problema legato ad i processi lenti, immaginiamoci un processo con molti I/O, riferisce pagine ma le riferisce lentamente rispetto ad un processo cpu bound, i processi lenti avranno sempre un numero minore di pagine riferite e quindi tenderemo a scartare le pagine riferite da quel processo, è stato dimostrato che gli algoritmi globali non funzionano bene in quanto tenderei a swappare le pagine dei processi lenti.

Quelli globali sono meno fair ma sono più facili da implementare in quanto ho una visione più ampia, possiamo tenere traccia delle pagine riferite globalmente, ma abbiamo lo svantaggio che i processi lenti che riferiscono poche pagine, le loro pagine tendono ad essere selezionate come vittime.

Gli algoritmi globali lavorano sulla core Map.

La core map per ogni frame della memoria fisica tiene traccia di quale processo la sta usando.

Gli algoritmi locali invece sono più fair per i processi lenti, questi algoritmi lavorano sulla tabella delle pagine.

T: time of last reference

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**

a) Initial configuration
b) Page replacement with a local policy (WS, LRU, sec. chance)
c) Page replacement with a global policy (LRU, sec. chance)

Supponiamo di avere il tempo dell'ultimo riferimento (è un contatore), e di avere 3 processi, se il valore T è basso la pagina è più vecchia. Un tempo di ultimo riferimento più alto significa che ho riferito la pagina più di recente.

Supponiamo sia in esecuzione A, l'algoritmo in esecuzione è locale e rimuovo la pagina A2, mi sarebbe però convenuto rimuovere C2 perché tra tutte era quello con il tempo di riferimento più basso, ovvero quello non riferito da più tempo.

In sostanza c'è una differenza nella considerazione dei tempi in base alla scelta di applicare l'algoritmo globalmente oppure localmente.

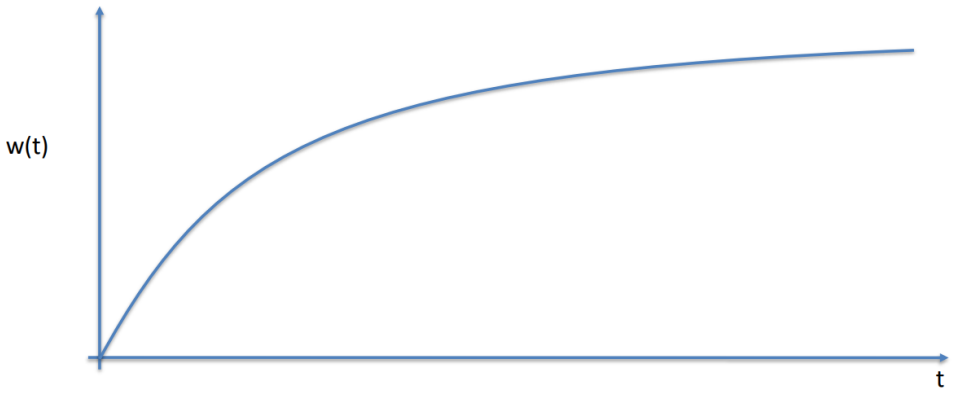
Working set algorithm

Per le cache un concetto fondamentale è quello di working set, ovvero l'insieme degli oggetti che ci conviene tenere in cache al fine di avere l'hit rate più alto possibile, in generale il working set si può definire in due modi diversi, possiamo considerare come working set l'insieme delle pagine riferite negli ultimi k accessi in memoria, dove k è un valore fissato, consideriamo che il ws è l'insieme delle pagine riferite negli ultimi k accessi, implementare questo è complicato in quanto per ogni processo dovremmo tener traccia delle pagine riferite.

In alternativa possiamo considerare un valore temporale ovvero consideriamo il working set come l'insieme delle pagine riferito nell'ultimo tempo definito come Δt.

Utilizzando la seconda definizione è più facile da implementare per questo tendiamo ad utilizzare questa definizione come working set per il processo.

Working set



- Working Set: **The set of pages referred in the last period T**
- w(*t*) is the size of the working set as function of time

Il working set ha una andamento come quello in figura, inizialmente è 0 e cresce per poi assestarsi, in una delle prime fasi tende a salire abbastanza rapidamente per poi stabilizzarsi.

Abbiamo una fase di caricamento e poi riferiamo poche altre pagine aggiuntive.

Un algoritmo che veniva usato (molto in passato) come algoritmo di rimpiazzamento è il Ws (working set) algorithm un algoritmo globale che tende a rimpiazzare pagine che non stanno nel working set o che si suppone non stiano nel working set, dato che il working set è un insieme di pagine che è difficile stimare per ogni singolo processo, quello che si fa in realtà è quello di stimare non il working set ma il resident set, il resident set in generale è molto più piccolo del working set.

Questo resident set mediamente approssima il working set dei processi ma è in generale un sottoinsieme, corrisponde all'insieme delle pagine della memoria virtuale di tutti i processi che sono attualmente caricate in memoria.

Il resident set potrebbe contenere anche delle pagine che non sono più riferite, perché magari è cambiato il working set del processo a cui quelle pagine appartenevano per cui quel processo non le riferirà per un po', questo però noi non possiamo saperlo.

Se teniamo in memoria le pagine che appartengono al working set minimizziamo il numero di page fault che è molto costoso, in particolare se dobbiamo riportare sul disco pagine che sono state modificate e che dovranno essere quindi riscritte.

Questo algoritmo del working set spesso si applica non al singolo processo ma globalmente al resident set, che è l'insieme delle pagine che in un certo istante sono in memoria, non è detto che questo resident set approssimi il working set, si cerca però di stimare il working set dei processi lavorando su questo insieme.

Il resident set in generale non ha niente a che fare con il working set, lavorando però su questo insieme si riesce ad avere una buona approssimazione del working set globale ovvero delle pagine riferite da tutti i processi in esecuzione, abbiamo quindi che il resident set tende al working set.

In memoria potremmo avere pagine che riferiamo molto poco ovvero che di fatto non fanno parte del working set, esiste un algoritmo che cerca di bilanciare le pagine poco riferite e quelle molto riferite, cercando di scaricare sul disco le prime.

Questo algoritmo cerca di stimare il working set globale e per farlo utilizza per ogni pagina:

Il bit R che sta per referred o U che sta per used bit e ci indica se la pagina è stata riferita nell'ultimo periodo di tempo t.

Un'altra informazione è il TLR che sta per *time of last reference*, ovvero il tempo dall'ultimo riferimento, ed è un'approssimazione del tempo passato da quando abbiamo riferito la pagina l'ultima volta.

Abbiamo poi l'età che è un contatore che ci dice da quanto tempo la pagina risiede in memoria, è definita come la differenza tra il tempo corrente e il tempo dell'ultimo riferimento.

Current virtual time: 2204

Page table

age	R	…
2084	1	
2003	0	
1980	1	
1213	0	
2014	1	
2020	1	
1604	0	

```
For each page: {
  if (R==0)
     age = current_time – TLR;
  else if (R==1) {
     TLR= current_time; R=0; age=0;
  }
  if ( (age>T)
     removes the page
}
-----------------------------------

if (age<=T for each page)
  removes the page with higher age
```

[age: current_time – TLR]

Dobbiamo immaginare che l'algoritmo vada in esecuzione periodicamente, oppure quando c'è una pagina da scartare.

Quando parte questo algoritmo per ogni pagina considera i 3 campi:

Se troviamo il bit riferito a 0 settiamo l'età al tempo corrente - tlr, se il bit era ad 1 settiamo il tempo dell'ultimo riferimento al tempo corrente e mettiamo l'età e R a 0.

Le pagine che trovo che hanno un età < T dove T è il valore fissato dall'algoritmo vuol dire che la considero nel working set, altrimenti se non la riferisco per troppo tempo vuol dire che potrebbe essere una pagina che probabilmente non è nel working set e quindi una pagina che potenzialmente potrebbe venir rimossa.

Notiamo che questo algoritmo viene applicato per ogni pagina e cerca di capire per ogni pagina quale ha un età maggiore di T per essere rimossa.

Questo algoritmo è in grado di dare una buona approssimazione del working set del processo, o del resident set che approssima il working set se applicato sulla core map.

Siccome applicare questo algoritmo localmente sulla tabella delle pagine è molto costoso, utilizziamo una variante applicabile alla core map, questo si chiama working set clock.

Working set clock

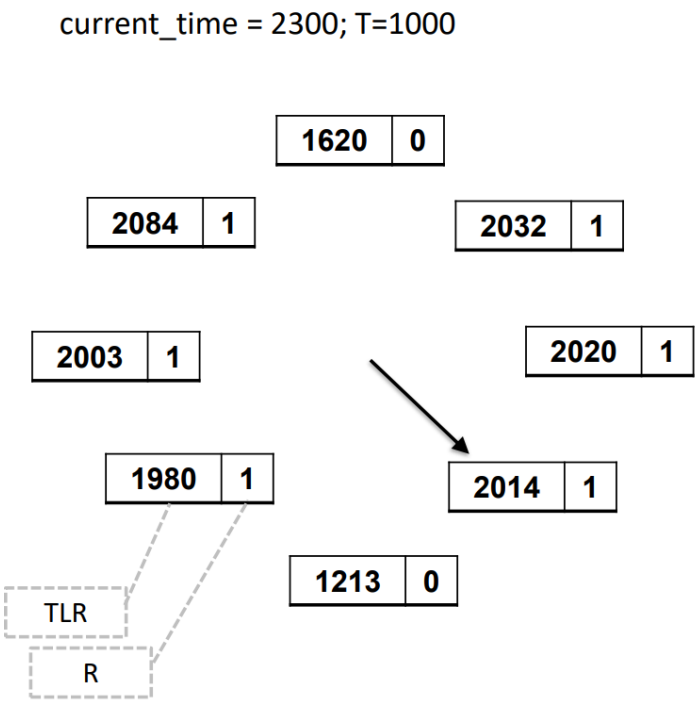
Concateniamo in una lista circolare le pagine usate in memoria, riferiamo tutte le pagine usate, presenti in memoria nella core map. Con questo algoritmo consideriamo solo quelle nella core map, in modo da non dover scansionare tutta la page table.

In questo esempio abbiamo un tempo corrente a 2300, l'algoritmo è stato impostato a considerare un Δt di 1000.

Questo t è un valore dato, il valore che vediamo nei rettangolini è il tlr, ovvero il tempo dell'ultimo riferimento, cioè quando ci eravamo passati l'ultima volta. L'altro è il bit R, 1 vuol dire riferita 0 non riferita.

L'algoritmo opera in questo modo, se la pagina è stata riferita, ovvero il bit è ad 1 quello che fa è quello di scrivere il tempo attuale nel tlr, e resettiamo il bit, spostiamo poi il cursore alla pagina successiva.

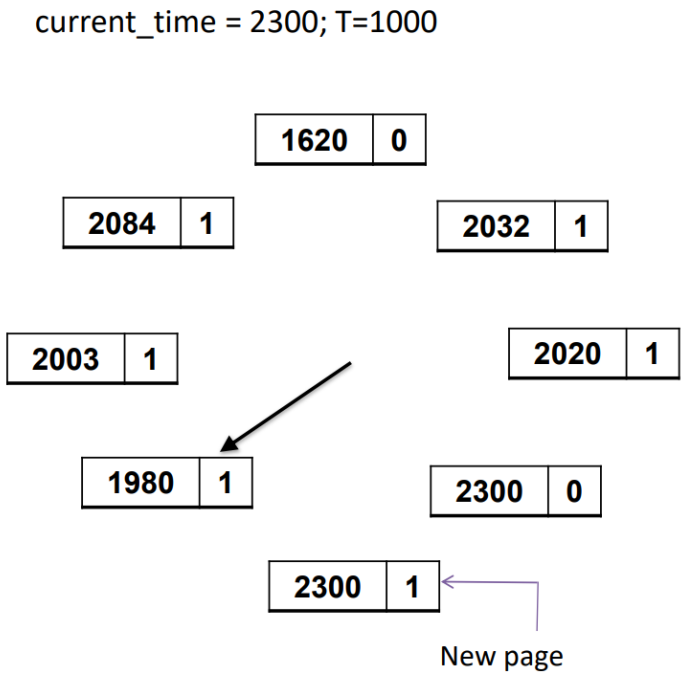
Andiamo poi alla pagina successiva, il bit è a 0, non è stata riferita, quello che facciamo è quindi sottrarre al current_time il tlr e verificare se il valore ottento è maggiore o no di T, in questo caso lo è quindi questa pagina è una pagina che può essere rimpiazzata.



Carichiamo quindi una nuova pagina fisica al suo posto, il tempo viene settato a current_time e il bit a 1.

Spostiamo poi il cursore alla prossima pagina.

Se l'algoritmo esegue periodicamente e non abbiamo al momento bisogno della pagina, questa pagina potrebbe essere messa in una lista di pagine pronte per essere rimosse.



Quello che spesso fanno questo tipo di algoritmi è quello di selezionare pagine vittime che non sono state modificate in modo da non doverle scrivere sul disco, potremmo avere quindi che l'algoritmo parte, seleziona un certo numero di pagine vittime, e le ordina sulla base del bit M, in modo tale che se poi mi servirà spazio prenderò la prima che non devo scaricare sul disco.

In questo modo applicando l'algoritmo working set clock riesco ad approssimare il working set del processo.

Cosa carichiamo all'avvio di un processo?

In generale quando carichiamo un programma in memoria abbiamo due possibilità, o cerchiamo di anticipare, facendo prepaging, ovvero cerchiamo di caricare in memoria pagine che sappiamo che verranno sicuramente riferite.

Questo può essere facile da fare in alcuni casi ma molto difficile in altri, per pagine contenenti la zona relativa allo stack del processo, ci serve caricarlo.

In generale quello che si fa è quello di utilizzare l'on-demand paging ovvero cerco di caricare il numero minimo indispensabile di pagine che un processo usa.

Quante pagine tengo in memoria nel processo?

In un sistema con tantissimi processi in esecuzione qual è il numero di pagine di memoria fisica per un processo che è in esecuzione?

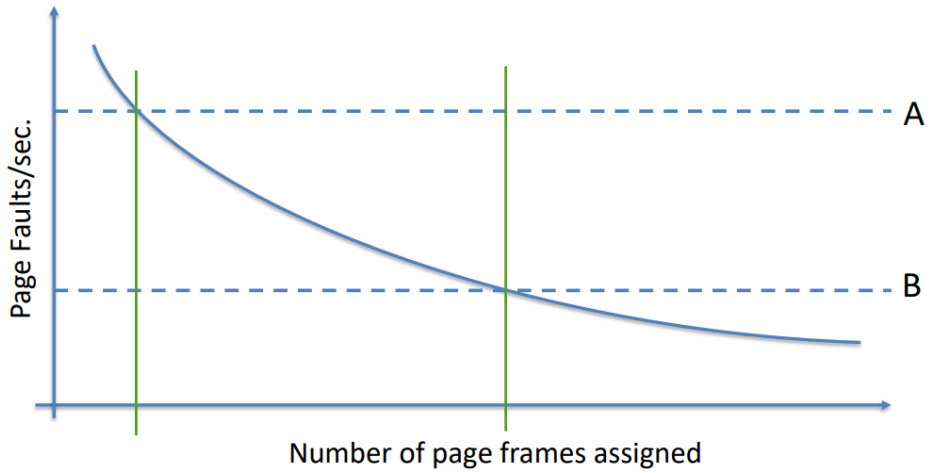
Quanti processi devo portare avanti in un certo momento?

Questo ha un impatto sul grado di multiprogrammazione, ovvero su quanti processi riusciamo a portare avanti concorrentemente in modo tale da dare effettivamente l'illusione che ogni processo abbia la CPU e il sistema a sua totale disposizione.

In generale non riusciamo a determinare questo valore in modo statico.

Page Fault Frequency

Quello che si fa è di utilizzare algoritmi dinamici ovvero a runtime, noi vedremo il page fault frequency. Prima di vedere l'idea sotto l'algoritmo osserviamo il grafico:



Number of page faults per unit of time, depending on the number of physical pages assigned to the process

Sulle ascisse abbiamo il numero di pagine fisiche assegnate ad un processo e sulle ordinate i fault per unità di tempo, utilizzando dei benchmark è stato possibile individuare due valori soglia tali per cui se allochiamo poche pagine abbiamo tantissimi fault, e un

valore per cui se allochiamo più di tot pagine non riusciamo ad impattare sui page fault, questo perché stiamo tenendo in memoria delle pagine non riferite.

Se ne allochiamo troppe inoltre rischiamo di non avere pagine libere da allocare per altri processi.

Questo algoritmo cerca di trovare dinamicamente i due valori soglia e funziona in questo modo:

Cerca di determinare il numero di pagine allocate per processo all'interno di questo intervallo e questo garantisce che il resident set sia leggermente più grande del working set soprattutto se allochiamo un numero di pagine vicino al valore B. È stato visto che quando la frequenza di fault è maggiore della natural frequency ovvero quando ci troviamo al di fuori di questo intervallo ci conviene incrementare il numero di pagine, se invece la frequenza di fault è minore o molto minore vuol dire che probabilmente ci conviene selezionare alcune pagine di quel processo da scaricare, che verranno individuate dal working set clock. Questi due algoritmi servono per cose diverse, working set clock serve per selezionare la pagina da scartare, questo (il pff) ci dice, ci dà una stima di quante pagine tenere in memoria **globalmente**.

Il pff serve a trovare due punti:

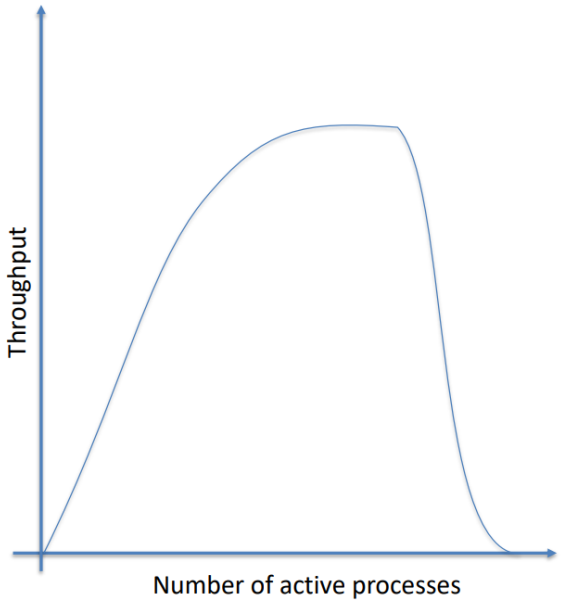
- Il primo dove se ci fermassimo prima avremmo troppi fault perché si starebbero utilizzando troppe poche pagine.
- Il secondo dove se ci si fermasse dopo sprecheremmo troppe pagine per avere un fault che diminuisce di poco.

L'obbiettivo è individuare questi due punti e fermarci nel mezzo ad essi, ossi individuare il numero buono di pagine da mantenere in memoria, che dovrebbe essere almeno uguale al working set.

Come facciamo a sapere quanti processi eseguire contemporaneamente?

Per ogni processo che parte devo allocare delle pagine.

Se il numero di processi attivi in un istante è molto alto potremmo arrivare ad una soglia in cui il So spende più tempo a caricare e scaricare pagine per cercare di portare avanti tutti i processi rispetto al tempo dedicato al calcolo effettivo, ricadiamo quindi nel fenomeno del trashing.



Quello che possiamo fare è utilizzare l'algoritmo page fault frequency per aiutarci ad evitare il trashing, se applicando l'algoritmo vediamo che tanti processi richiedono molta memoria e nessuno dei processi in esecuzione la rilascia vuol dire che stiamo rischiando di andare in trashing, quindi quello che possiamo fare sono due cose:

Killiamo un processo che utilizza tanta memoria, come possiamo scegliere però quale processo killare e quando?

In realtà si usa un altro metodo, ovvero riduciamo il grado di multiprogrammazione, il So si accorge che sta andando in trashing, riduce quindi il numero di processi che vengono schedulati contemporaneamente, ne prendiamo un certo numero e li scarichiamo sul disco, facciamo *swap out*, liberiamo le pagine che stanno utilizzando e speriamo che liberando questa memoria alcuni degli altri processi possano terminare così che io possa riprendere ad eseguire quelli che ho swappato.

I processi non vengono killati vengono solamente messi in pausa.

Si tende a fare swap out soprattutto per i processi che sono eseguiti in background.

Non sempre questa tecnica è risolutiva perché se continuano ad arrivare nuovi processi prima o poi andremo comunque in trashing.

Le pagine vengono scaricate sul disco, le parti di codice vengono semplicemente rimosse, verranno poi ricaricate dall'eseguibile, per quanto riguarda i segmenti data, heap e stack, vengono scaricate in file temporanei che di solito sono allocati in una swap area.

Stiamo allungando la memoria del sistema, se finisce la memoria fisica e tutta l'area di swap la memoria è finita.

Le shared libraries vengono messe nei file temporanei o nei file codice.

I file mappati in memoria vengono messi in memory mapped files.

Quando un processo finisce vengono eliminati i file temporanei.

Zipf Model

In realtà ci sono sistemi in cui il modello working set non funziona, un esempio sono i web server, là si ha infatti una cache (su disco o in memoria) delle pagine (web) maggiormente richieste, qua il modello working set non funziona, utilizziamo quindi altri modelli, uno di questi è lo zipf model. Tutte le volte che abbiamo a che fare con una sorta di popolarità il modello non è quello del working set, utilizziamo quindi altri modelli con le stesse prestazioni, abbiamo pochissime pagine web riferite tantissimo, e un'infinità di pagine web riferite molto poco.

Ha una coda molto lunga e un picco su un rage molto basso, viene fatto un ordinamento sulla base del ranking della pagina web.

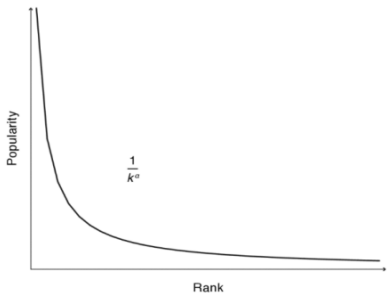
L'obbiettivo dei web server non è tenere un po' di ogni sito web, ma è quello di tenere l'insieme delle pagine più riferite.

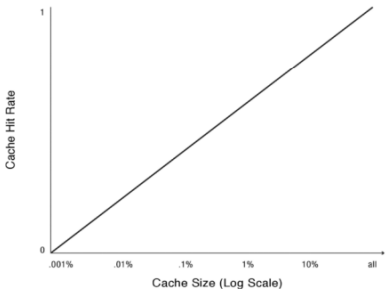
Il modello working set è specifico per il caching dei programmi e per le cache hardware, esistono molti altri modelli.

Caching behavior of many systems are not well characterized by the working set model (e.g., web server)

An alternative is the Zipf distribution model

- Popularity ~ $\frac{1}{k^c}$ for k-th most popular item (1<c<2)
- Examples: web pages, movies, words in text, etc..





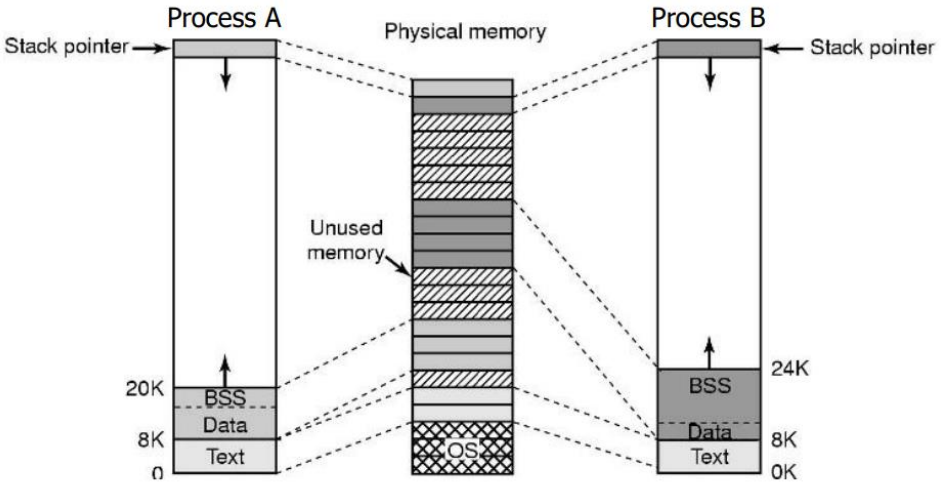
Memory Management Unix

Vediamo un po' di esempi basati su sistemi operativi un po' vecchi che però utilizzano le tecniche viste sino ad adesso.

BSD v.3:

Usava la segmentazione con paginazione, la memoria virtuale era basata sull'on demand paging.

La paginazione veniva implementata con la core map nel kernel che teneva traccia dei frame fisici a cui i processi venivano associati e utilizzava l'algoritmo di rimpiazzamento second chance.



Il sistema supportava memorymapped file e segmenti di memoria condivisa, era possibile creare segmenti di memoria condivisa oppure mappare un singolo file in memoria fisica e mapparlo nello spazio di indirizzamento logico dei due processi che lo condividono.

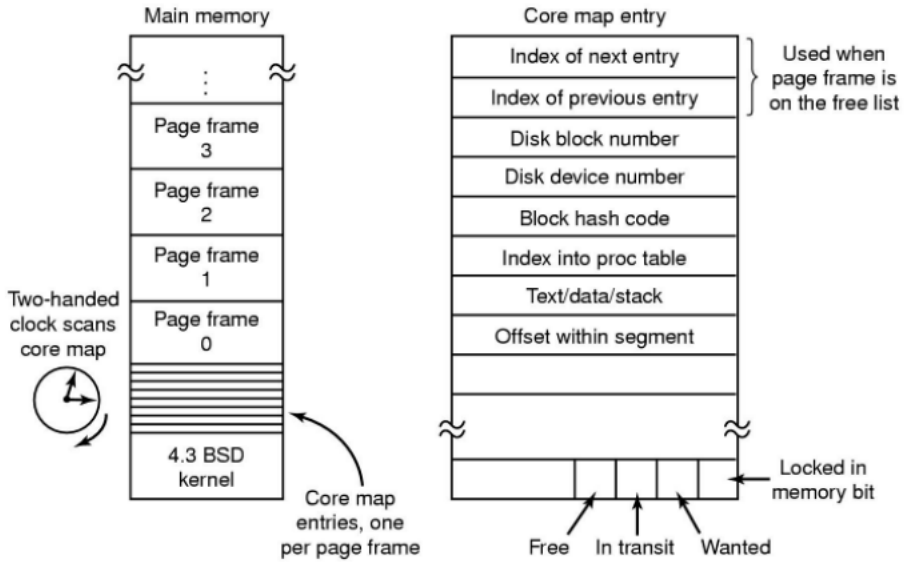
Una cosa interessante del memory mapping è la zero copy I/O

Se facciamo I/O con chiamate di sistema stiamo caricando una porzione del file in un buffer in user space, il kernel oltre a fare content switch, copierà i dati richiesti nel buffer di destinazione del processo in user space.

Se invece facciamo memory mapping, paghiamo il costo di una chiamata di sistema che è il costo della mmap ma avremo poi tutti i dati nel nostro buffer di indirizzamento, non faremo più una copia da kernel space in user space o viceversa.

Inoltre riusciamo a sovrapporre le operazioni che magari facciamo sul file con il caricamento o scaricamento delle porzioni di file che magari ho già letto/scritto (pipelining).

Il sistema BSD utilizzava una core map



L'algoritmo second chance utilizzava due lancette anziché una, in ogni entry della core map abbiamo i flag, e molte altre informazioni relativamente alla posizione sul disco di questa pagina, la core map per implementare second chance è implementata come lista circolare, servono quindi dei puntatori alla pagina successiva e alla precedente.

1) Core Map + Tabelle delle pagine

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
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

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

- l'indice di pagina virtuale non è contenuto nella tabella

2) Core Map = Tabella delle pagine inversa

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
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

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

Page replacement (BSD)

È una variante degli algoritmi visti, il sistema utilizzava un deamon chiamato page deamon che utilizzava 3 parametri: lotsFree, minFree e desFree.

MinFree sta per minime pagine libere, DesFree per desiderate e lotsFree per la quantità che abbiamo di pagine libere.

Dove vale la relazione:

Le pagine che ho libere > quelle che desidero avere libere > minime che devo avere libere

$$LotsFree > DesFree > MinFree$$

PageDaemon algorithm (**sketch**):

- ***if*** (*#freeblocks* ≥ *lotsfree*) return *//no operation required*
- ***if*** (*minfree* ≤ *#freeblocks* < *lotsfree*) ***or***
  (*#freeblocks* < *minfree* ***and***
  Average[*#freeblocks,* Δ*t*] ≥ *desfree*))
  replace pages until *#freeblocks* = *lotsfree* + *k* (with *k*>0)
- ***if*** ((*#freeblocks* < *minfree* ***and***
  Average[*#freeblocks,* Δ*t*] < *desfree*))
  swapout processes

Quando avveniva lo swapping?

Quando vengono poi ricaricate le pagine che avevo tolto con lo swap out?

Ovvero quando faccio swap in?

In questo BSD unix lo swap-in avveniva se il numero di blocchi liberi era abbastanza grande allora venivano selezionati uno o più processi tra quelli che avevo interrotto considerando o il tempo per cui sono stati messi in pausa oppure per la memoria che richiedono per l'esecuzione.

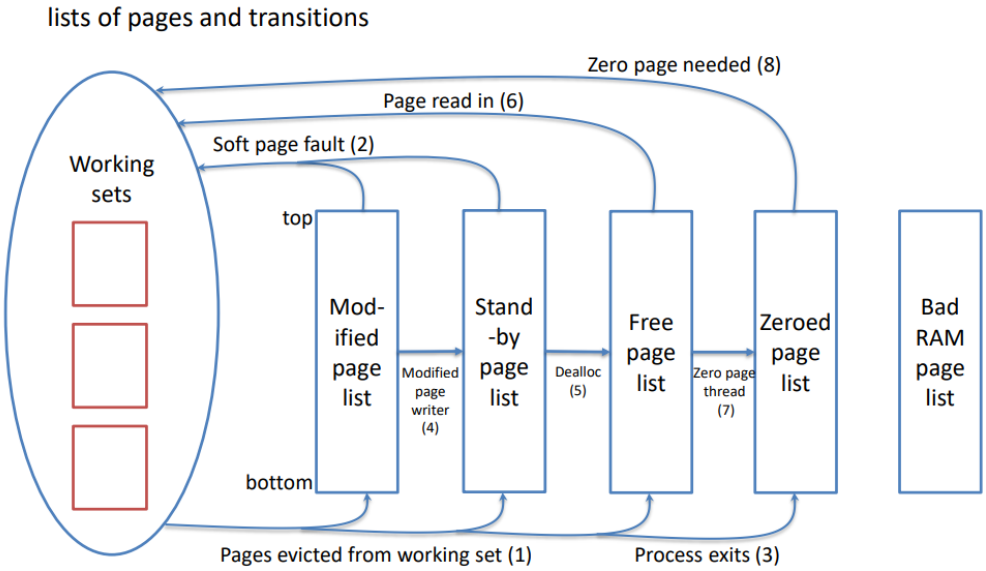
Vecchi Windows

I vecchi windows:

Windows ha utilizzato algoritmi di tipo locale, usava l'algoritmo working set applicato localmente e per ogni processo in esecuzione si stabiliva un valore di pagine residenti che dovevano essere compresi in un intervallo [min, max], questi valori venivano aggiornati dinamicamente in base agli utilizzi della memoria.

A seguito di un page fault la pagina veniva caricata in un blocco libero della memoria e la size del resident set per il processo p veniva incrementata, se il numero delle pagine era maggiore del valore massimo stabilito venivano rimosse alcune pagine dal working set in modo random.

Inoltre si cercava sempre di mantenere un certo numero di pagine libere disponibile all'uso; per farlo venivano utilizzati due deamon: il working set manager utilizzato per rimpiazzare le pagine e un balance set manager che cercava di mantenere la memoria fisica, se questa era scarsa eliminava le pagine identificate dal working set manager.



Dato che la scrittura sul disco a qui tempi costava tantissimo la politca era la seguente, quando dobbiamo scaricare una pagina non scriviamo sul disco, ma le teniamo in memoria in un'insieme di liste ordinate, tenendo traccia del processo a cui apprtenevano.

Dopo un po' le pagine non richieste vengono spostate in una lista di stand by dove anche qua si tiene traccia del processo, queste pagine possono essere assegnate solo al processo a cui vengono tolte.

In seguito utilizziamo una lista di pagine che abbiamo azzerato, il processo poteva infatti aver bisogno di pagine nuove.

Fyle System

Noi abbiamo visto HDD e SSD ed abbiamo detto che il SO ci fa avere l'astrazione di vedere un vettore di blocchi, in ogni blocco però devono esserci delle informazioni che ci aiutino ad interpretare quei dati, il modo in cui decidiamo di organizzare tali informazioni genera il file system.

Il file system è un'astrazione al di sopra di uno storage che garantisce persistenza, potrebbe essere un ssd o un hard disk, ma anche un disco di rete, potrebbe non essere fisicamente installato nella macchina ma potrebbe trovarsi ad esempio su un altro computer.

Per storage intendiamo un dispositivo in grado di memorizzare dati in modo permanente.

Il file system fornisce l'illusione di avere tutto lo spazio di storage a disposizione.

Quando parliamo di dispositivi un aspetto critico non sono solo le prestazioni, vogliamo che i dati che manteniamo permanentemente sullo storage li ritroviamo quando li riaccediamo in futuro, quindi che lo storage sia affidabile, che se ci dovesse essere trashing i file non vengano compromessi ecct..

Spazio dei nomi, non vogliamo indirizzare direttamente il disco, vogliamo avere nomi di lunghezza qualunque senza avere limitazioni.

Un altro aspetto critico sono le prestazioni, compito del file system è cachare in memoria i dati che si trovano sul disco. I fs e in generale le librerie che operano su file fanno caching sia lato utente che lato kernel.

Per svuotare il buffer in user space si usa fflush, per fare il flushing dei dati cachati in lato kernel si usa fsync.

Il file system è molto aggressivo dal punto di vista del caching.

Vogliamo avere sicurezza, vogliamo un controllo degli accessi su dati condivisi, nei file system unix esiste il concetto di owner che permette di allargare i permessi di accesso in lettura e scrittura agli altri utenti del gruppo.

Un file è una collezione di dati con un nome, possiamo immaginarcela come una sequenza di byte/record, accedibile mediante opportune chiamate di sistema.

Ci possono essere chiamate read/write, anche se, per evitare di usare chiamate di sistema possiamo mappare file in memoria nello spazio di indirizzamento del processo, in modo da evitare syscall e poter accedere con normali load e store.

Una directory è la possibilità di raggruppare sotto un unico nome un insieme di file e directory.

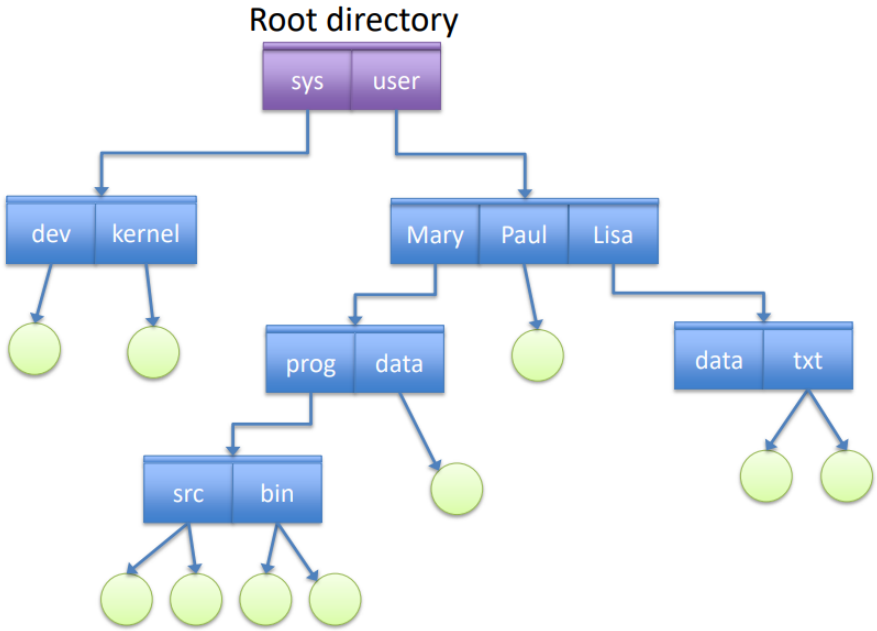
Esistono due tipi di path, path assoluto e path realtivo, se il path inizia con uno / è un path assoluto, lo / indica la directory root.

Abbiamo poi i Link, nel modo unix si distinguono link fisici e link logici, che sono degli alias, ovvero la possibilità di dare un altro nome ad un file per poter fare riferimenti a file che si trovano in directory diverse.

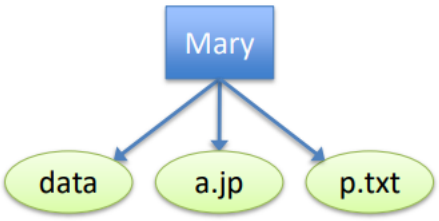
Possiamo montare sul computer dischi esterni dandogli una posizione relativa al file system corrente, in quella root inizia però la nuova root.

Struttura del file system

Il file system è un albero, tipicamente sbilanciato, sono aciclici, in modo tale da avere una gestione più semplice.



Una directory è un file con particolari caratteristiche che raggruppa altri file, una directory è una struttura dati che raggruppa i file e associa ogni file ai suoi metadati, tipicamente una entry di una directory dal punto di vista logico associa un nome di file o di directory ad un descrittore associato a quel file, questo descrittore contiene degli attributi, dei metadati con dei puntatori a dove si trovano i dati relativi a quel file.



name	descriptor
data	Attribute: type, address, etc.
a.jp	Attribute: type, address, etc.
p.txt	Attribute: type, address, etc.

Implementation of directory Mary

Implementazione delle directory: Unix

Al nome è associato un puntatore ad una struttura dati di tipo I_NODE.

Accesso ai file

Un file è una sequenza contigua di record, un record è un insieme di byte, le operazioni tipiche sono quelle di leggere e scrivere record all'interno di un file, abbiamo anche operazioni quali stat o fstat che permettono di recuperare gli attributi, ovvero quali sono i metadati del file, la size è un metadato che si trova nel descrittore di file che è possibile ottenere grazie ad una opportuna chiamata di sistema.

In generale una struttura record, la cui implementazione dipende dal file system viene cachata in memoria, e ce la teniamo fino a quando non viene chiesto di chiudere il file.

Tra i metadati abbiamo ad esempio la data dell'ultima modifica.

Gli accessi al file possono essere sequenziali o diretti, con l'accesso sequenziale ogni volta che leggiamo o scriviamo, il puntatore si sposta al record successivo, se so qual'è la posizione dell'informazione che mi interessa all'interno del file posso posizionarmi direttamente sul record di interesse.

Unix file system API

Vediamo alcune delle funzioni di libreria che operano sui file system:

- create, link, unlink, mkdir, rmdir
  - Create file, link to file, remove link
  - Create directory, remove directory
- open, close, read, write, seek (mmap , munmap)
  - 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)

Nel file system unix esistono 8 tipi di file, e sono: file regolari, file standard, le diirectory, i file a blocchi (dischi), file a caratteri, i link, le pipe e i socket.

File System design

Per realizzare un file system dobbiamo porci delle domande come ad esempio:

Abbiamo file piccoli o grandi?

Pesano di più i file piccoli o quelli grandi?

Di solito le dimensioni della maggior parte dei file sono medio/piccole.

Quali sono mediamente i file più acceduti?

Solitamente quelli di piccole dimensioni, anche se i file grandi ci permettono di sfruttare bene il dispositivo, se facciamo accessi sequenziali riusciamo a fare un accesso efficiente in quanto leggiamo blocchi grandi.

Il file system deve gestire bene sia file di piccole dimensioni che file di grandi dimensioni.

Tipicamente i file vengono acceduti prevalentemente in modo sequenziale, anche se in alcuni casi potremmo andare a scrivere in posizioni specifiche o accedere in posizioni casuali.

Per i file piccoli ci converebbe avere blocchi di disco piccoli, l'unità minima di accesso al disco è il settore, non potremo mai caricare un byte dal disco. Anche il file system per quanto ci dia un'astrazione al byte non legge un solo byte, ma un blocco logico che a sua volta è mappato su un blocco fisico che è il settore.

Per file piccoli ci converebbe avere blocchi di dimensione piccola per non sprecare spazio.

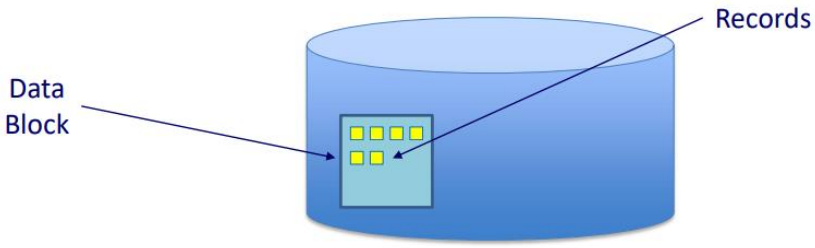
D'altra parte per file grandi ci converebbe avere blocchi grandi, in quanto sicuramente li riempiamo e li possiamo leggere in modo efficiente.

Potremmo ad esempio raggruppare tutti i file di una stessa directory se sono piccoli, così da metterli magari in uno stesso blocco di memoria fisica in modo da avere poi una sorta di località spaziale.

Dal punto di vista del design di un file system abbiamo a che fare con 3 strutture dati, la prima è quella che mappa le directory e quindi i filename in metadati, la seconda è quella che ci permette di memorizzare i metadati ovvero ci permette di dare i puntatori ai blocchi dati, l'ultima struttura è la freemap che ci dice quali blocchi sul disco sono ancora liberi.

Ogni fs utilizza implementazioni di queste strutture dati diverse.

Su disco l'accesso è al blocco, un blocco è un insieme di record, i file sono organizzati a record, per il fs unix un record è un byte, ovvero è l'unità minima.



Noi leggeremo un blocco che contiene più record appartenenti allo stesso file.

La dimensione del blocco solitamente è molto maggiore rispetto a quella del record.

Dove si trovano però i blocchi?

Ci serve una struttura che indicizziamo in modo efficiente e che ci permette di sapere dove si trova il blocco, come entry troveremo poi l'offset giusto.

Esiste anche un concetto di località che è molto importante soprattuto per i dischi non a stato solito, per i dischi a stato solido infatti sappiamo che non ci interessa dove andiamo a scrivere i dati, abbiamo simmetria tra lettura e scrittura.

Per gli hard disk sparpagliare i i blocchi tra le varie tracce non è una buona idea in quanto la lettura del disco avviene in modo sequenziale con lo spostamento delle testine.

File System a confronto

Guardiamo adesso le caratteristiche di 3 file system:

	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

FAT (File allocation Table)

Creato da microsoft prevalentemente per i primi pc, è molto semplice e ancora usato da alcune chiavette usb, ha però alcuni problemi.

- Utilizza una linked list.
- La granularità è il blocco.
- La struttura dati che ci permette di capire dove sono blocchi liberi è implementata come un vettore.
- Fat usa il concetto di deframmentazione, ogni tanto decompatta la struttura dati per togliere i buchi che si creano

FFS (Fast file system)

È il nonno dei moderni fs unix.

Risolve una parte delle problematiche di fat ma ne lascia alcune.

- Utilizza un albero fisso.
- La granularità è il blocco.
- La struttura dati che ci permette di capire dove sono blocchi liberi è implementata come una bitmap e viene fissata sul disco.

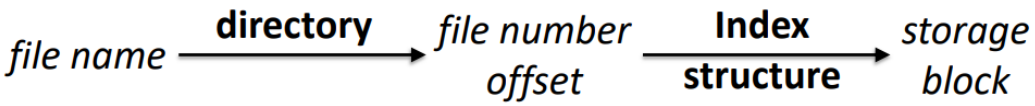
NTFS (New technology file system)

Ideato da microsoft per i server

Risolve le problematiche di ffs

- Utilizza un albero dinamico.
- La granularità è l'extent.
- La struttura dati che ci permette di capire dove sono blocchi liberi è implementata come una bitmap ma sotto forma di file.

In generale quando chiediamo di aprire un file; attraverso la directory si recupera tramite metadati il file number offset che ci permette di indicizzare la struttura che ora vedremo e che è diversa per ogni file system per recuperare i blocchi dove risiedono i dati che vogliamo leggere/scrivere, cercheremo di capire com'è fatta questa index structure per i vari file system.



FAT

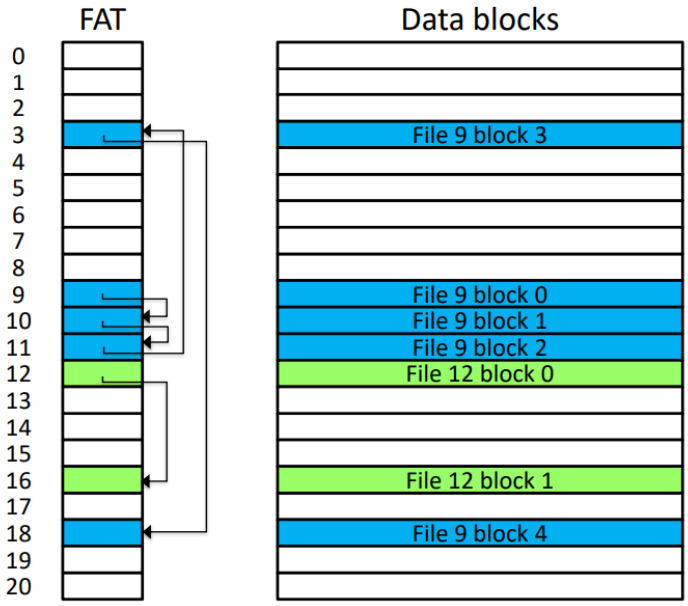
Fat è il file system più semplice nasce negli anni '70, molto diffuso negli anni '80 ed è tuttora usato, la struttura indice che ci permette di recuperare i blocchi è una lista linkata ovvero molto semplice da implementare, viene infatti implementata come un array. Questa struttura ha tante entry quanti sono i blocchi del disco, questo è fondamentale e caratterizza fortemente la FAT, ogni file è di fatto una lista di blocchi all'interno di questa struttura dati array, supponendo di avere un disco con 21 blocchi avremmo una fat di 21 entry.

Esiste un'associazione 1:1 tra le entry della fat e i blocchi del disco.

All'interno del blocco 18 il valore del puntatore al blocco successivo sarà un puntatore ad un valore speciale che ci dirà che il file è finito.

I blocchi liberi si trovano tutti in una lista linkata in modo che sia facile trovarli.

L'inizio del file lo recuperiamo dalla directory, nella directory in qualche modo riusciamo a trovare il file Number offset che per la fat ci dice l'entry iniziale.



Pro:

L'implementazione è molto semplice, è facile trovare blocchi liberi e fare operazioni di append e lettura sequenziale di un file, se devo appendere, in tutte le implementazioni della fat ho anche il puntatore alla coda o alla testa per accedere in fondo.

Non è facile fare accessi random, se dovessimo accedere al blocco 11 non sappiamo dove si trova, dobbiamo scorrerla tutta fino a trovare il puntatore al blocco 11, questo non era tanto un problema prima, in quanto l'accesso era sequenziale.

L'informazione dei metadati che possiamo contenere nella directory per quanto riguarda fat è limitata, in particolare fat non gestisce i permessi dei file e i bit di protezione.

Gli aspetti di sicurezza non erano prevalenti quando è stato progettato, in quanto è stato creato per utilizzi su personal computer con un solo utente.

Un problema grande è la dimensione della struttura dati fat, essendo un'associazione 1:1 con i blocchi disco la struttura dati è tipicamente grossa, anche se dipende da quanto è grande il file system.

Dobbiamo per forza tenerla tutta in memoria, ci occuperà quindi diverse pagine di memoria togliendo spazio.

Abbiamo una forte limitazione su quanto grande può essere il file system.

È soggetta inoltre al problema della frammentazione, dopo un po' visto che i file vengono creati e cancellati si creano molti buchi sparsi, non riusciamo a mettere blocchi dello stesso file vicini, quando si usava questo fs veniva fatta deframmentazione per cercare di compattare la tabella, e mettere i file il più possibile contigui uno dietro l'altro.

Questa implementazione per quanto molto semplice ha delle limitazioni forti.

Quanto è grande?

Data la lunghezza in bit degli elementi della fat: L

Data la dimensione in byte dei blocchi del disco: B

Il numero di blocchi indirizzabili è: 2^L

La massima dimensione del file system è: $B \cdot 2^L$

Se ogni elemento occupa N byte la fat occupa complessivamente: $N \cdot 2^L \ byte$

Esempio: con N=2 ($\rightarrow$ FAT 16) e B= 2^{10} (blocchi di 1 Kbyte)

La massima estensione del file system è 2^{16} blocchi (2^{26} byte = 64MB)

La FAT occupa complessivamente 2* 2^{16} byte= 128 Kbyte

- Sul disco la FAT occupa 128 KB, cioè 128 blocchi.
- con una memoria paginata e pagine di 1Kbyte, la FAT occupa 128 pagine in memoria principale.

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.

$\rightarrow$ Per realizzare file systems più estesi si usano blocchi di dimensioni maggiori.

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

FFS

Fast file system o unix file system, è il progenitore dei moderni file system unix, in questo caso non si utilizza più una tabella con un numero di entry fisse come per la fat, oggi si utilizza una tabella degli I_Node.

Un I_Node è il descrittore di un file che contiene tutti i metadati.
La directory conterrà il nome del file e l'indice dell'I_Node, o meglio l'offset che mi permette di recuperare l'I_Node come entry di una I_Node table.

L'I_Node è diviso in due parti, una parte sono i metadati veri e propri: size, permessi, owner, i timestamp, lo user Id, il group Id, i bit di protezione, ecct…

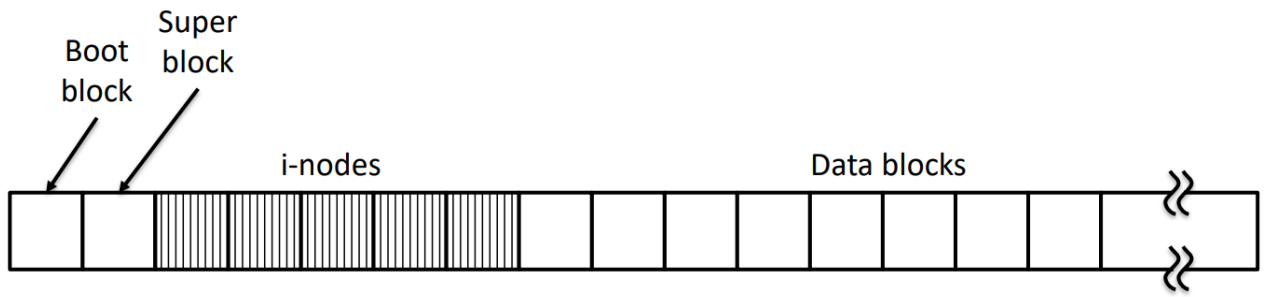
Non è presente dentro all'I_node il nome del file, l'I-Node ha infatti una dimensione fissa mentre il nome del file può avere una dimensione variabile.

La seconda parte dell'I_Node è un insieme (fisso) di puntatori che ci permettono di recuperare i blocchi del disco che contengono i dati relativi a quel file.

L'organizzazione logica di questo file system è la seguente, abbiamo dei blocchi dedicati, come il boot block che contiene le informazioni per fare il boot, abbiamo poi un super block che è specifico del file system e che contiene tutte le caratteristiche del file system come ad esempio qual è la dimensione dei blocchi, qual è l'informazione della directory radice, contiene un insieme di dati che caratterizzano dove si trovano le bitmap dei blocchi liberi, dove comincia l'I_Node table.

L'I_Node table è distribuita su vari settori ma occupa un certo numero di blocchi del disco.

Infine poi abbiamo i blocchi dati veri e propri in base a quanto è grande il file system.



I metadati di un I_Node sono tutti quelli che recuperiamo con una chiamata stat o fstat posix, sono codificati in modo particolare ma quelli sono tutti e soli i metadati.

A seconda della versione del file system abbiamo un certo numero di puntatori a blocchi diretti.

Abbiamo poi un certo numero di puntatori a blocchi indiretti singoli, doppi e tripli.

Un file con blocchi del disco da 4k, può essere grande al più 4T(+4G+4M+48K), questo dipende infatti da come è costruito l'albero.

Supponiamo che associato ad un certo file nella directory trovo questo indice di I_Node me lo carico in memoria e avrò la prima parte dei metadati e questa sequenza di puntatori, i primi puntatori (10-12) sono puntatori a blocchi del disco diretti, cioè ci troviamo direttamente i dati di quel file, con questi 12 puntatori possiamo immaginare di indirizzare blocchi di memoria da 4k, per un totale di 48k (12 * 4k), se il file ha dimensione minore di 48k riusciamo ad indirizzarlo tutto, altrimenti non mi bastano e dovrò utilizzare un puntatore a blocco indiretto, avremo un puntatore (ad esempio il 13°) che punta ad un blocco che non contiene dati, ma che contiene puntatori a blocchi dati, è una indirezione

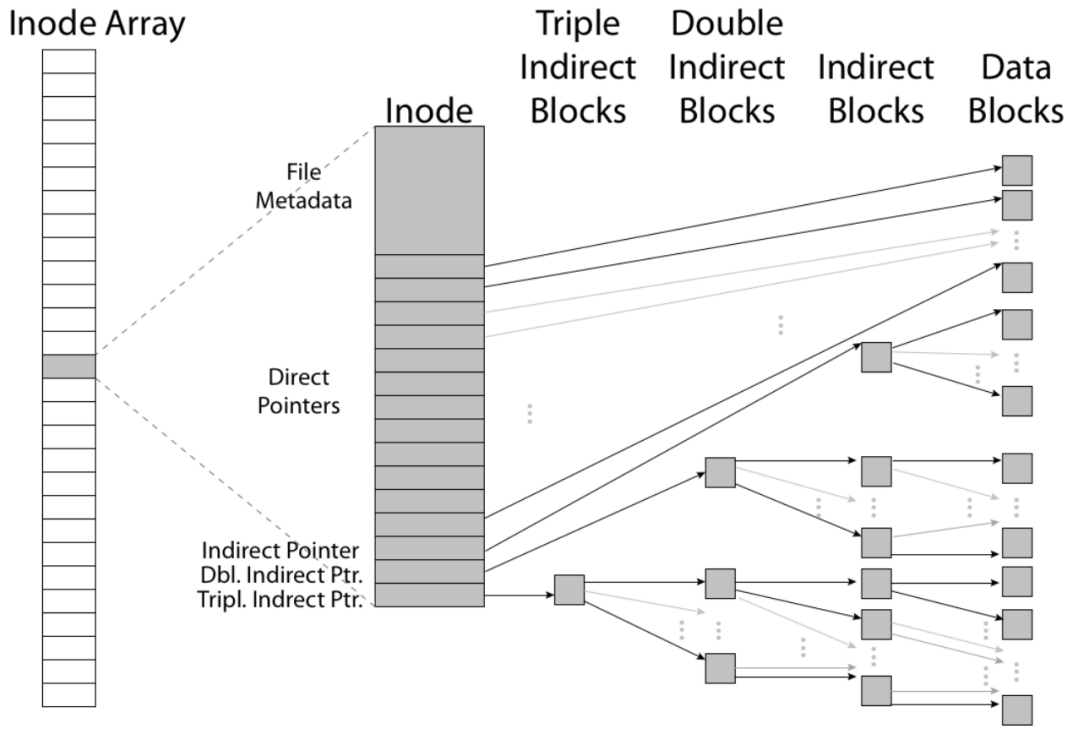
semplice, conterrà *n* puntatori a seconda della dimensione del blocco, se il blocco era 4k e supponendo di avere puntatori da 4byte (dimensione standard di un puntatore), ci saranno 1024 (4k / 4Byte) puntatori a blocchi in questo blocco puntato, avremo quindi 1024 nuovi blocchi dati ognuno da 4k.

Con questi puntatori indiretti singoli riusciamo ad indicizzare file grandi fino a 4M+48k.

Se sono più grandi dobbiamo passare al puntatore indiretto doppo, abbiamo un puntatore che punta ad un blocco dati, dove ci sono 1024 puntatori a blocchi indiretti singoli, in cui ognuno di essi contiene 1024 puntatori a blocchi dati.

E così via fino ad arrivare al puntatore indiretto triplo, dove ho 3 livelli di indirezione, e alla fine ho blocchi dati.

Se facciamo il conto con questi numeri riusciamo ad indicizzare un file grande 4T+4G+4M+48K.



Se il file è piccolo bastano i puntatori iniziali, riusciamo a gestire bene sia file piccoli che file grandi. In particolare, per file grandi tutto sommato riusciamo tramite un accesso a più blocchi del disco ad accedervi abbastanza velocemente, riusciamo a calcolare bene a quale blocco dobbiamo accedere, in quanto la struttura è una struttura fissa, una volta fissato la dimensione del blocco dati (4k, 2k, 8k) so quanti puntatori singoli ho.

Riusciamo bene a navigare l'albero.

Con il file system ffs i file piccoli hanno un albero piccolo (poco profondo) i file molto grandi hanno un albero molto profondo, riusciamo comunque a fare un lookup abbastanza efficiente dovendo fare pochi salti.

Inoltre riusciamo a gestire i file così detti sparsi, potrei creare file enormi da 4T non occupando 4T sul disco, potremmo creare un file allocando 2 blocchi dati, il file system non alloca tutti i blocchi intermedi necessari per l'indirizzamento, in quanto sappiamo di dover fare un file da 4T ma scriveremo piano piano su quei due blocchi dati (8k), quando li abbiamo esauriti il FS mi crea la struttura on-demand.

Dal punto di vista della località, delle prestazioni, l'accesso random è facile, non è inefficiente, però come sfruttiamo la località?

Chi ci garantisce che questi blocchi siano vicini?

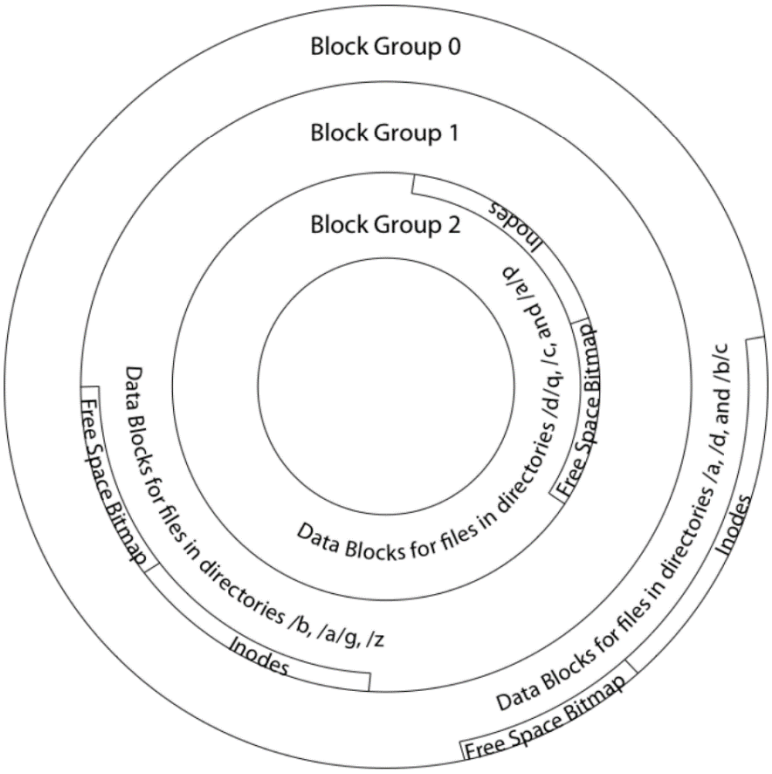
Non lo garantisce nessuno, quello che si può fare però è di raggruppare i blocchi appartenenti ad uno stesso file in un gruppo. Un block group è un insieme di tracce del disco occupati dai dati di uno stesso file.

Questo ci permette avendoli vicini di spostarci, visto che il file lo accederò sequenzialmente raggruppando porzioni dell'albero tutte su un insieme di uno stesso gruppo di un settore riesco ad ottimizzare la località che ho.

Quest'organizzazione a blocchi e gruppi ha senso solo per hard disk, per dischi ssd non c'è questo tipo di allocazione.

Tendiamo a memorizzare blocchi di un file il più possibile sullo stesso gruppo che è un insieme di tracce, ma non solo, si cerca di memorizzare i file di una stessa directory per quanto possibile sullo stesso gruppo ma anche directory annidate in gruppi adiacenti.

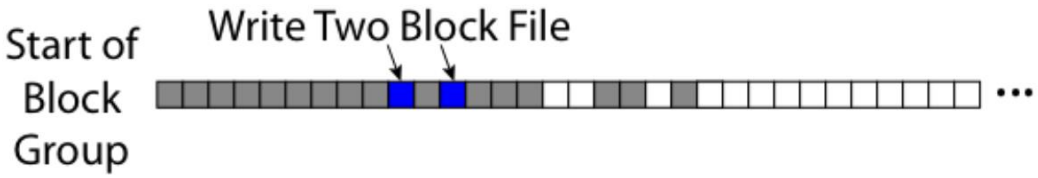
Anche la bitmap e l'I_Node list sono sparpagliate tra i vari gruppi in modo da trovare tutto quello che ci serve quando la testina si trova in una determinata sezione del disco.



Questo non ci dà la garanzia di assoluta efficienza, dopo un po' anche con questo meccanismo potremmo avere frammentazione, dopo un po' soprattutto se il disco è molto pieno potrei non avere più spazio per memorizzare file di una stessa directory nello stesso gruppo, passeremo dunque ad un altro gruppo, finchè nel disco c'è abbastanza spazio riusciamo ad essere abbastanza efficienti.

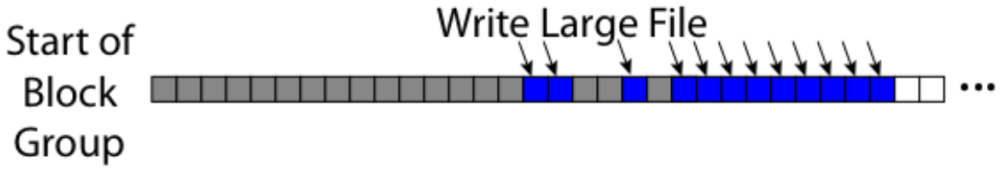
Come si allocano i blocchi liberi?

La politica che funziona meglio in questo caso è la first fit, il primo che trovo libero lo occupo, il primo bit a 1 che mi dice che il blocco è libero lo riservo, è come una bitmap, supponiamo un file che necessiti di due blocchi, se ho due buchi scorrendo la lista dei blocchi del gruppo li trovo e li seleziono.



Stiamo sacrificando un po' la località spaziale tra blocchi in quanto non stiamo considerando se questi due blocchi trovati sono vicini tra loro.

Questo non è un problema in quanto accade tendenzialmente con file piccoli, con file molto grandi andrò ad occupare la parte finale della bitmap e avrò località spaziale.



Questa politica funziona ragionevolmente bene per file piccoli e abbastanza bene per file grandi, c'è un però; è stato notato che questa politica funziona particolarmente bene solo se abbiamo abbastanza blocchi liberi alla fine della bitmap, per evitare di saltare tra un gruppo ed un altro troppo frequentemente quello che il file system fa è quello di riservare più blocchi per gruppo rispetto a quelli reali, quando viene formattato il disco, circa il 10% dei blocchi di quel disco, viene riservato, non viene messo a disposizione dell'utente ma viene tenuto per avere la garanzia di avere abbastanza blocchi liberi per ogni gruppo, con questa strategia si è visto che effettivamente la frammentazione è molto bassa, abbiamo però perso il 10% del disco, non possiamo più allocarlo.

Se il disco è pieno l'amministratore non riesce a loggarsi per liberare le risorse, allora l'idea è stata che l'amministratore può sfruttare questo spazio extra che è stato allocato in anticipo per garantire che il superutente si possa loggare.

L'algoritmo è stato visto che funziona sacrificando un po' di spazio disco.

Pro:

Questo file system funziona bene sia per file piccoli che grandi, riusciamo con la politica first fit a garantire una buona località per i file grandi, riusciamo ad avere località per i metadati, l'I_Node viene caricato in memoria e ce lo teniamo fin quando il file non viene chiuso.

Gli svantaggi sono che per file di un byte comunque dobbiamo allocare un I_Node e un blocco di disco, lo spazio che sprechiamo è enorme, per file minuscoli questo tipo di approccio non è efficiente.

Per garantire buona località inoltre dobbiamo perdere una parte dello spazio disco 10~15%, inoltre l'altro aspetto è che se ho un file che accedo sequenzialmente e che magari scrivo in un colpo solo con una write su disco, quel file lo vorrei contiguo, ma non ho la garanzia che tutti i blocchi siano allocati contigui.

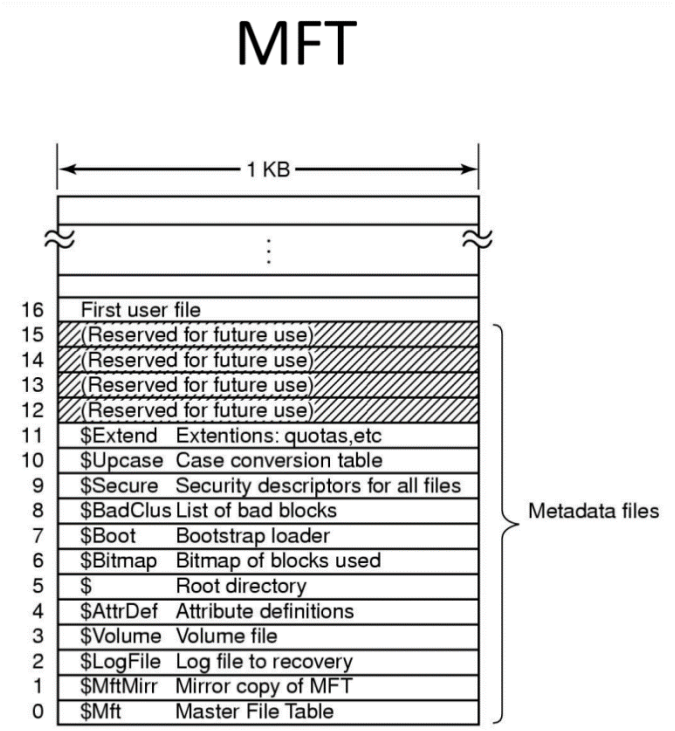
NTFS

Le limitazioni di FFS sono state superate da NTFS (new technology file system)

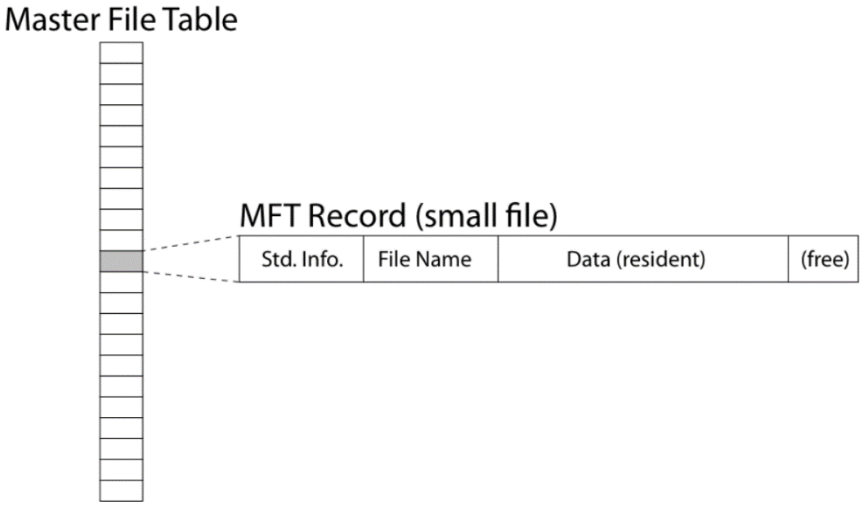
In questo caso non c'è più il concetto di I_Node ma c'è un'unica tabella che si chiama master file table, in cui ogni entry della tabella è un master file record.

Ogni record ha una dimensione fissa, tipicamente 1-2k e su un record della tabella mft ci possiamo memorizzare sia dati che metadati in un colpo solo. Non teniamo separati come in ffs la struttura dei metadati (l'I_Node) dai dati.

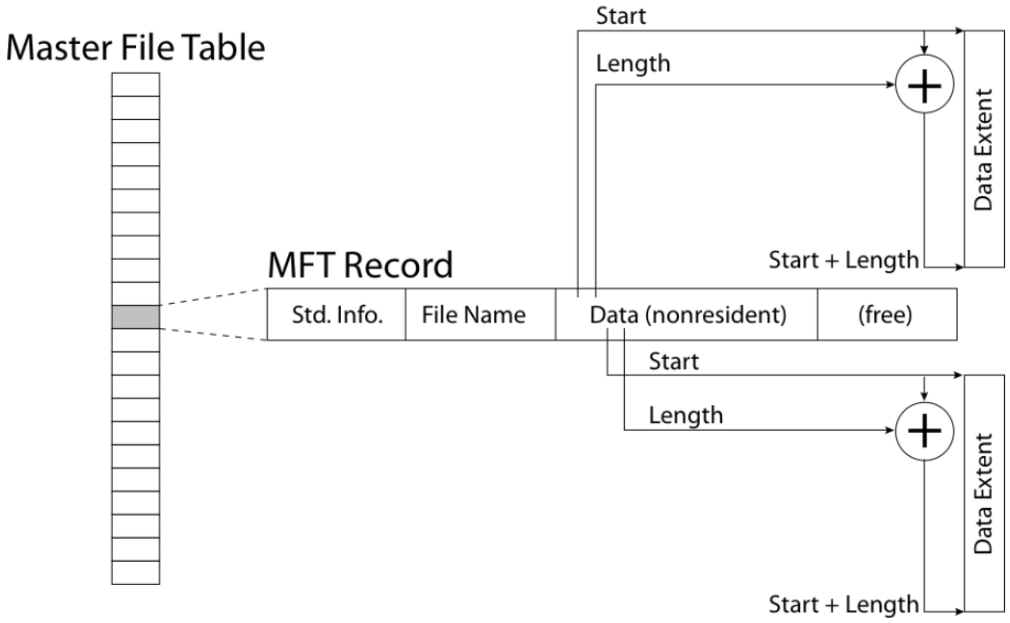
Ntfs fonde i due concetti in un'unica struttura dati chiamata mft, inoltre non utilizza più il concetto di blocco con dimensione fissa, ma utilizza il concetto di extents, che sarebbe un blocco con dimensione variabile.



L'mft è una tabella dove ogni entry è larga almeno 1kb, alcune entry della tabella sono riservate, ad esempio è riservata l'entry 5 che contiene le informazioni della root directory, l'entry 6 è per la bitmap dei blocchi liberi e altre informazioni, in questo esempio l'entry di indice 16 è la prima utilizzabile dai file veri e propri.



Concettualmente funziona che quando si apre/crea un file tramite la directory troviamo l'offset da usare all'interno della master file table, individuiamo poi l'mft record che dobbiamo utilizzare, dentro questo record troviamo tutto quello che ci serve: i metadati, in questo caso anche il nome, siccome un file può avere nomi diversi tramite link, nella directory memorizziamo il nome iniziale, nella mft invece tutti i suoi alias se li abbiamo.



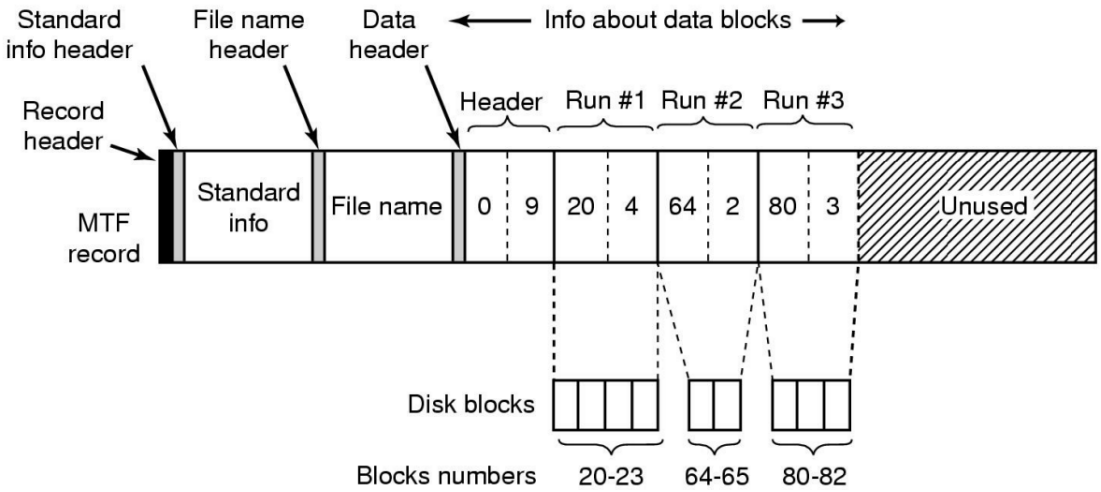
Se il file è molto piccolo i dati li scriviamo direttamente nell'mft record.

Se il file cresce e non entra più allochiamo un altro extent.

Nella parte dati anziché mettere i dati veri e propri mettiamo un puntatore e una lunghezza del blocco dati.

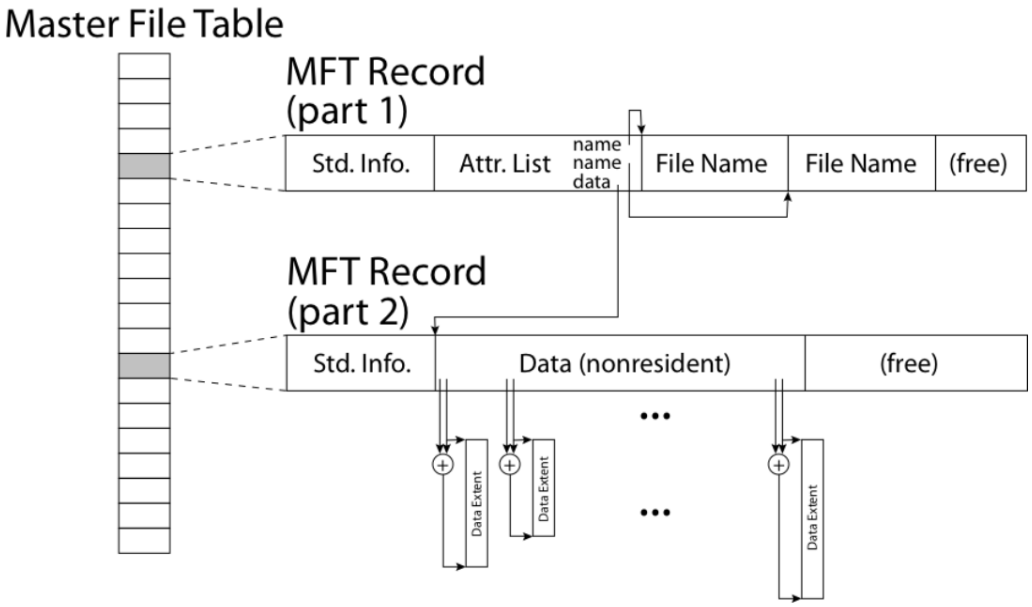
È una struttura ad albero che può crescere dinamicamente.

Risparmiamo tutto lo spazio per gli I_Node, per la I_Node list ecct, risparmiamo un sacco di overhead che è pesante per file molto piccoli.



Nell'mft record troviamo un reacord Header, le informazioni di sistema, i metadati, il nome del file, ogni volta c'è un header, l'header ci dice dove comincia il filename e quanto è lungo il filename, codifichiamo il punto di partenza e la lunghezza.

Poi per ogni blocco dati codifichiamo il blocco iniziale e quanti blocchi è grande.

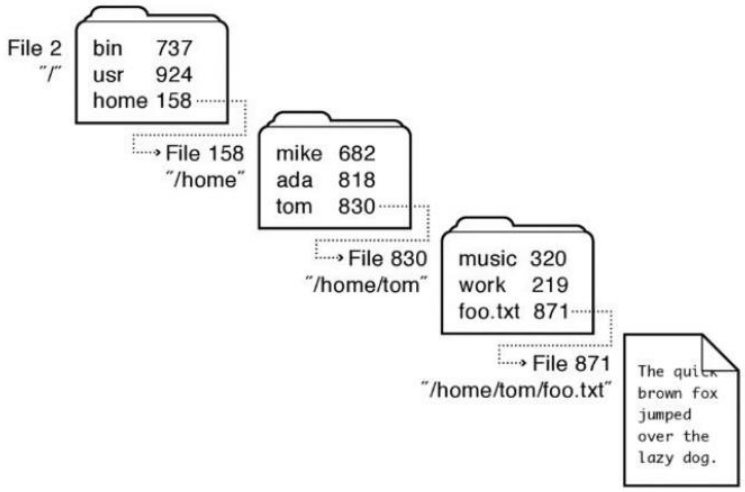


Anche in questo caso c'è rischio di frammentazione anche se con molte meno probabilità, infatti per file molto grandi potrebbero crearsi dei buchi, NTFS richiede ogni tanto di fare deframmentazione per poter allocare extent più grandi.

Directory

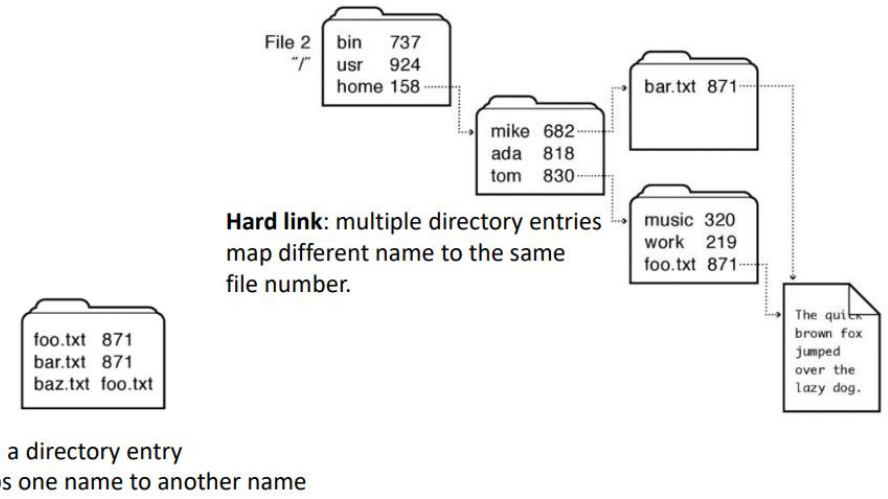
Per i 3 file system che abbiamo analizzato abbiamo visto la struttura dati che viene indicizzata a partire dai dati contenuti nella directory, utilizzando l'identificatore del file che è l'indice all'interno o della master file table in caso di NTFS oppure potrebbe essere l'indice all'interno della fat, o l'indice dell'I_Node nell'ffs.

Una directory in caso di file system ffs ha un path del tipo:



Partiamo dalla directory radice che è "/" e viene caricata al momento del boot in memoria e i cui metadati sono contenuti nel master boot record.

Accediamo poi alla directory home caricandola, prendiamo un'altra directory tom, e in fine il file foo.txt



Abbiamo un file che si chiama bar.txt, e ha come I_Node il nodo 871, in un'altra directory c'è un alias di foo.txt che punta allo stesso I_Node, abbiamo due link allo stesso I_Node questo è il concetto di hard link, questo è poco costoso in quanto è solo una nuova entry di una nuova directory. Abbiamo però alcune limitazioni, non si possono fare link a partizioni diverse del file system e non si possono fare link a file system diversi, possiamo fare link solamente a file e non a directory.

Se cancelliamo uno dei due link, non succede niente, rimuoviamo la entry e il file rimane in vita perché c'è un altro link che punta a quell'I_Node, verrà cancellato il file quando non ci saranno più link fisici all'I_Node.

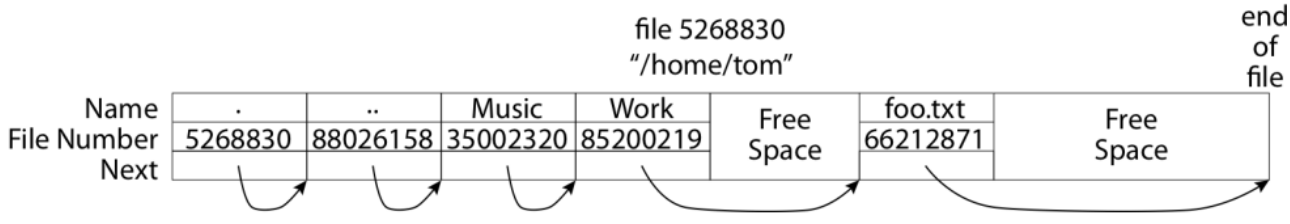
Abbiamo un campo detto hard-link count che tiene traccia di quante volte quel file è stato riferito, quando questo contatore arriva a 0 il file viene eliminato.

Le problematiche di questa tecnologia portano all'utilizzo di soft link, è anch'esso una directory entry in cui però non abbiamo come nel caso precedente la coppia nomefile I_Node ma un'associazione nome1 nome2.

In questo caso abbiamo due hard link e il terzo è un soft link, abbiamo un riferimento non al file (I_NODE) ma ad un altro nome. È allocato dunque un I-NODE per baz.txt dove però al posto del puntatore ad un file c'è il puntatore al nome.

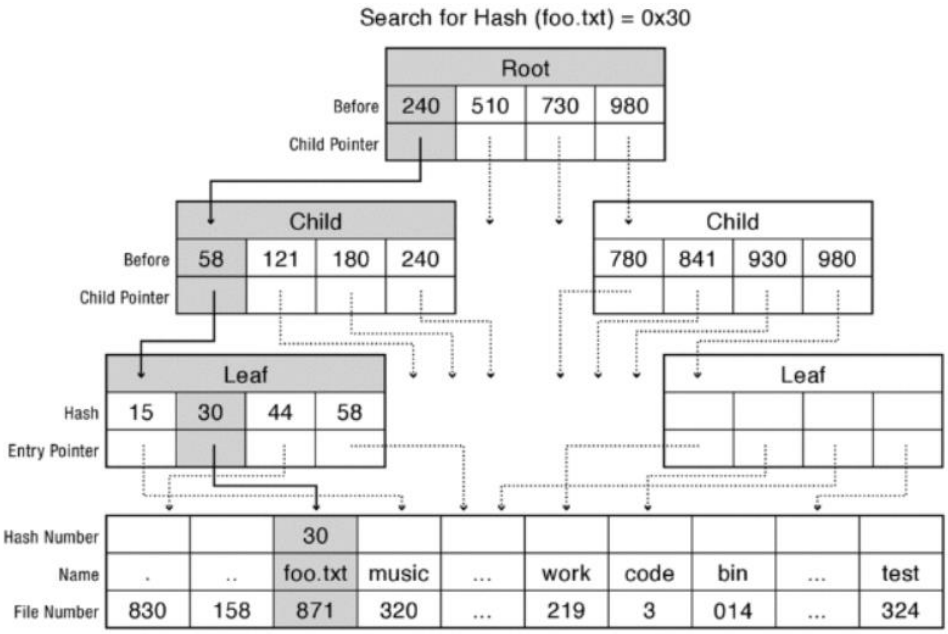
In questo caso se cancello il file originale questo link si rompe, in quanto baz.txt non punta più a niente.

I soft link sono molto più flessibili in quanto è un'associazione nome – nome, lo svantaggio è che occupiamo più spazio, consumiamo un ulteriore I_Node per memorizzare il nome del file.



La struttura dati directory soprattutto in file system vecchi era implementata come una lista linkata, il problema di questa struttura è che se nella directory memorizzo tantissimi file accedere ai file della directory è costoso, dobbiamo infatti scorrere tutta la lista.

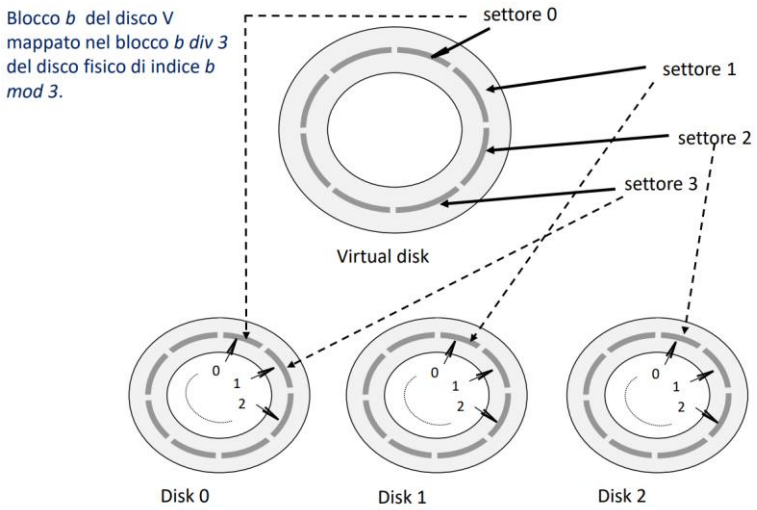
Non usiamo più una struttura a lista bensì una struttura ad albero, in ffs la struttura dati della directory è un btree (binary search tree?), viene fatta una funzione hash per la ricerca, la funzione produce un indirizzo esadecimale che partiremo a cercare dalla radice.



Storage Raid

I dischi, soprattuto i dischi magnetici, gli hard disk, sono soggetti per via delle parti meccaniche a rompersi, il sistema raid si usa anche per gli ssd, è usato in tutti i casi in cui ho bisogno di affidabiità, si basa sul concetto di usare più dischi.

Consiste in un set, un array di dischi ridondanti, dal punto di vista dell'utente però non si vuole sapere a quale disco accedere, noi vediamo un unico disco virtuale che rappresenta un aggregato dei dischi fisici.



Il driver che gestisce il raid si preoccuperà di recuperare l'informazione sulla base del settore vero in cui si trova, abbiamo un'astrazione, noi utenti vedremo il sistema come se ci fosse un unico disco.

Ci sono diversi livelli di raid, alcuni hanno solo rilevanza storica, i livelli 2 e 3 e trattano dischi sincroni, in cui le testine dei vari dischi sono sincronizzate e non è possibile fare accessi in parallelo.

Noi vedremo i livelli 0, 1 4 (anche se poco usato), 5 e 6.

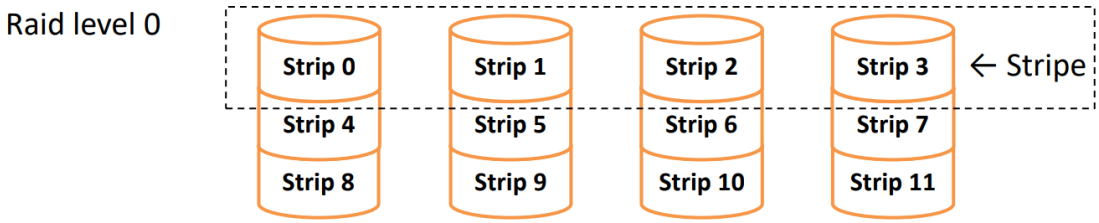
Raid 0

Un raid di livello 0 non ha ridondanza, spesso si chiama JBOD (è solamente un insieme di dischi) non miglioriamo quindi l'affidabilità, si usa infatti per migliorare le prestazioni, se io faccio il raid_0 sui due dischi, i dischi sono indipendenti e non c'è ridondanza.

Quello che succede è che i settori, i blocchi che si chiamano stripe, sono memorizzati in modo interallacciato sui due dischi, se abbiamo un file grande 100, e uno stripe di 500, il primo pezzo del file lo salvo sul primo disco, il secondo pezzo sul secondo, se accedo a parti del file diverse posso fare letture e scritture in parallelo.

Se si rompe un disco siamo fregati perché perdiamo tutti i dati che stanno su quel disco.

Nei raid di livello 0 si parla di stripe che sono o singoli settori oppure settori contigui sul disco.



Raid di livello 1

I dischi sono sempre asincroni. Il raid di livello 1 ha un minimo di due dischi, il primo disco contiene i dati, il secondo disco è un mirror del primo, ovvero contiene una copia identica del disco 1, le scritture avvengono contemporaneamente, se ne ho 3, uno sarà il master e 2 i mirror. Se ne ho 4, uno master e 3 mirror.

Nel raid di livello 1 facciamo una copia dei dischi su dischi diversi.

Raid di livello 2

Sono dischi sincroni con le testine mantenute tutte nella stessa posizione in modo da fare tutti le stesse operazioni.

Si utilizzano dischi ridondanti e la particolarità è che esistono codici di correzione degli errori, vengono scritti blocchi dati di dati aggiuntivi per poter correggere errori se si generano, ovvero se qualche parte del disco dovesse corrompersi, codici di correzione degli errori ne esistono di tanti tipi e si utilizzavano prevalentemente nei sistemi dove c'erano un'altissima probabilità di guasti e di rotture.

Raid di livello 3

È simile al raid di livello 2, ci sono sempre dischi sincroni, non si usano codici di gestione degli errori ma codici di parità non permettono di gestire errori ma solo di riconoscerli e recuperare errori molto semplici.

Questi due raid non si utilizzano più, soprattutto per il fatto che usano dischi sincroni.

Raid di livello 4

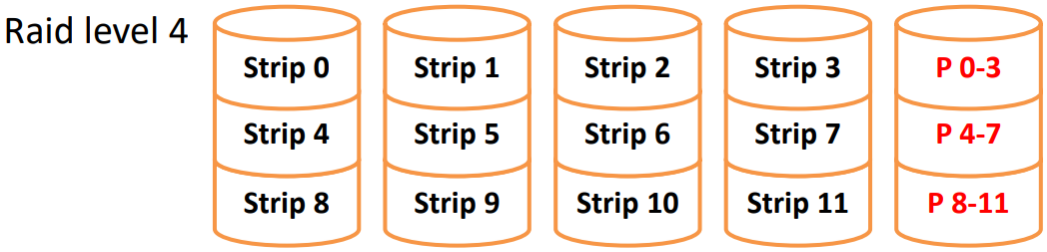
È stato ormai superato dal raid di livello 5

Abbiamo dischi asincroni come per 0 e 1, abbiamo un disco ridondante dedicato a contenere codici di parità, un disco è utilizzato solo per codici di parità, per gestire eventuali guasti di altri dischi dove ci possiamo scrivere e leggere contemporaneamente.

Non si usa perché se si rompe il disco dove si ha la parità il sistema diventa molto fragile.

Se si rompe un disco e il disco della parità siamo fregati.

Il raid 4 è sostanzialmente un raid 0 per un certo numero di dischi, e un disco che contiene la parità che ci permette di sapere se una parte di un file è stata corrotta, e tramite questo codice di parità riusciamo a capire e a riscostruire i dati presenti in questo disco.



Raid di livello 5

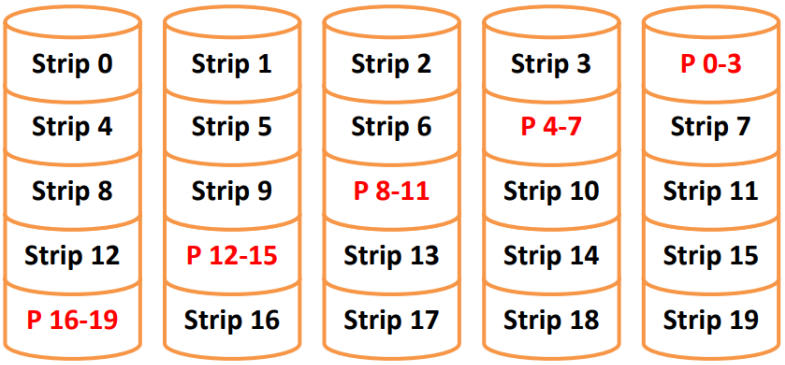
Come nel 4 viene mantenuta la parità di ogni disco che però non è centralizzata ma viene distribuita tra tutti i dischi, il vantaggio è che in questo caso tutti i dischi sono equivalenti, si richiede che esistano almeno 3 dischi.

Se ho dischi da 100 giga avrò capacità di 200 giga e i 100 giga restanti verranno distribuiti tra i dischi per i codici di parità.

Se si rompe un disco ne aggiungiamo un altro e il controller raid ricostruisce i dati persi, se però se ne rompono due perdo i dati.

Nel raid 5 la parità che nel 4 era concentrata in un disco solo viene distribuita su tutti i dischi.

Raid level 5



Raid di livello 6

È un'evoluzione del raid 5, richiede almeno 4 dischi la ridondanza invece di essere distribuita in modo interallacciato è data grazie a gruppi di parità fatti da coppie, posso tollerare con 4 dischi la rottura di 2 dischi contemporaneamente e il sistema sopravvive.

Un controller raid è una scheda che gestisce e si occupa di fare i calcoli e di capire dove recupare i dati (su quale disco), dove scrivere la parità. Tipicamente questi controller sono associati agli storage, in uno storage c'è un controller con attaccati *n* dischi, il controller ha un suo mini sistema operativo che serve per controllare e gestire le repliche per garantire consistenza.

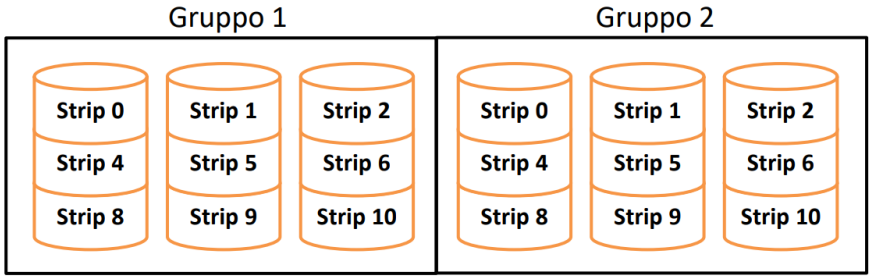
Il raid può essere fatto anche a software, tipicamente si fa solo per i raid_0 e 1 dove l'affidabilità non è la priorità.

Combinazione di raid

Questi livelli di raid possono essere combinati in vari modi, quelli più usati sono 1_0 e 0_1.

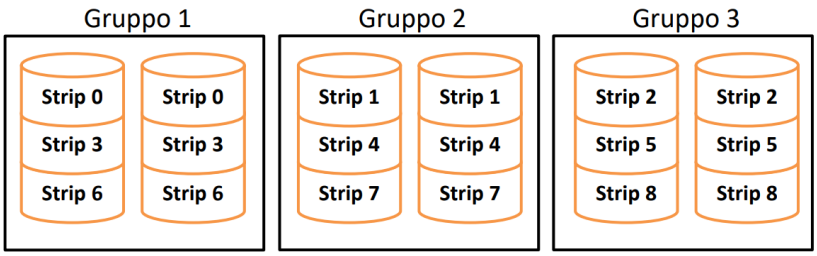
Nel raid 0+1 abbiamo il mirror di stripe, facciamo due gruppi di dischi, uno è il mirror dell'altro, all'interno di un gruppo faccio raid 0 ovvero faccio lo stripe.

Raid level 01



Nel raid 1+0 creiamo n gruppi da due dischi in ogni gruppo abbiamo un disco e il suo mirror, i gruppi sono gestiti come stripe.

Raid level 10



Esempio dischi Raid

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.

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