Preparazione all'orale

All'orale, come prima domanda, chiederò di commentare uno a caso di questi schemi/algoritmi

02 kernel.pptx

Spiegare la commutazione di contesto

02 kernel.pptx

Spiegare il meccanismo di upcall

07 scheduling.pptx

Spiegare differenze tra processi I/O bound e CPU bound e come devono essere gestiti dallo scheduling

06 advsynch.pptx

Definire lo stallo, dire quali condizioni sono necessarie affinché lo stallo si verifichi e discutere le tecniche per risolvere il problema

lucidi 31…38 – 09 caching.pptx

Spiegare algoritmi di sostituzione ideale, LRU, second chance

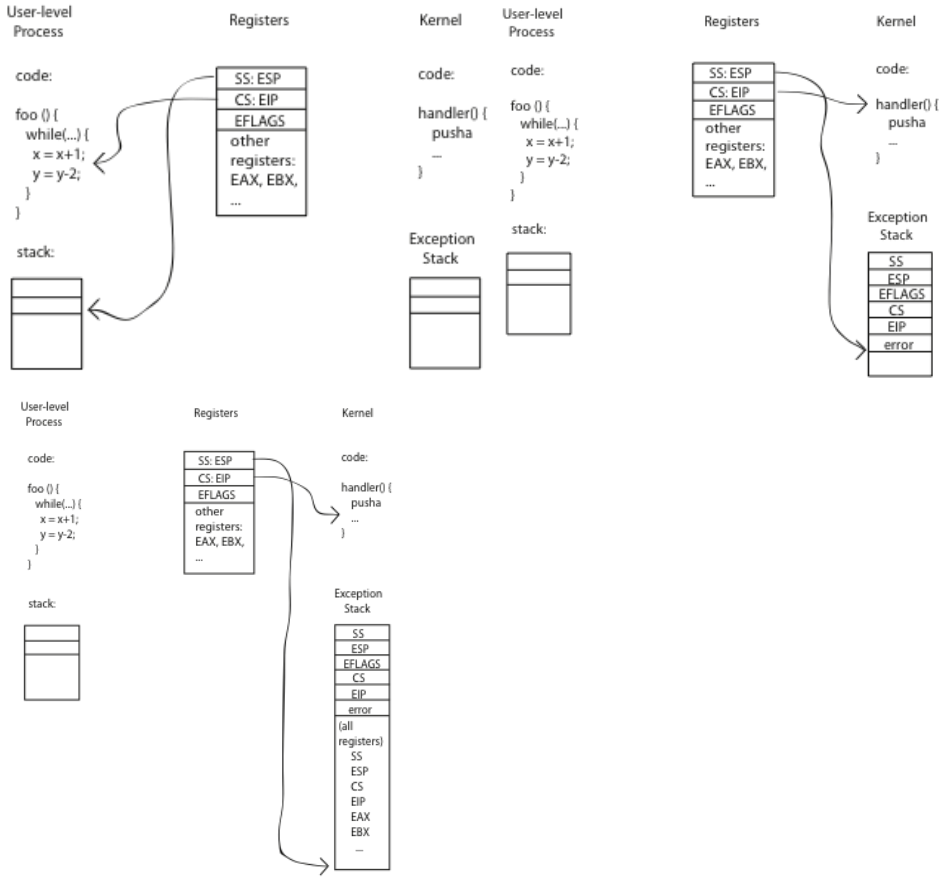
lucido 40 – 09 caching.pptx

Spiegare differenza tra algoritmi di sostituzione locali e globali

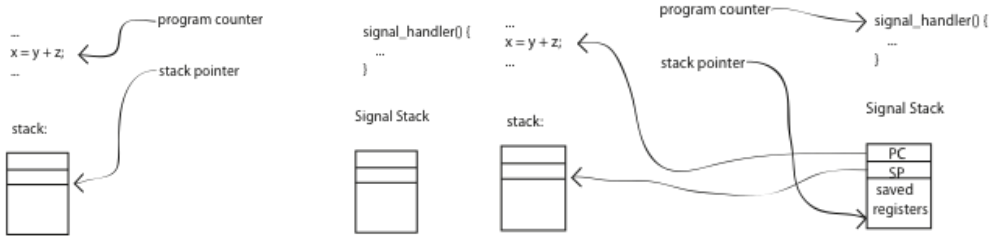
lucido 49 – 09 caching.pptx

Spiegare differenza tra pre-paging e on-demand paging

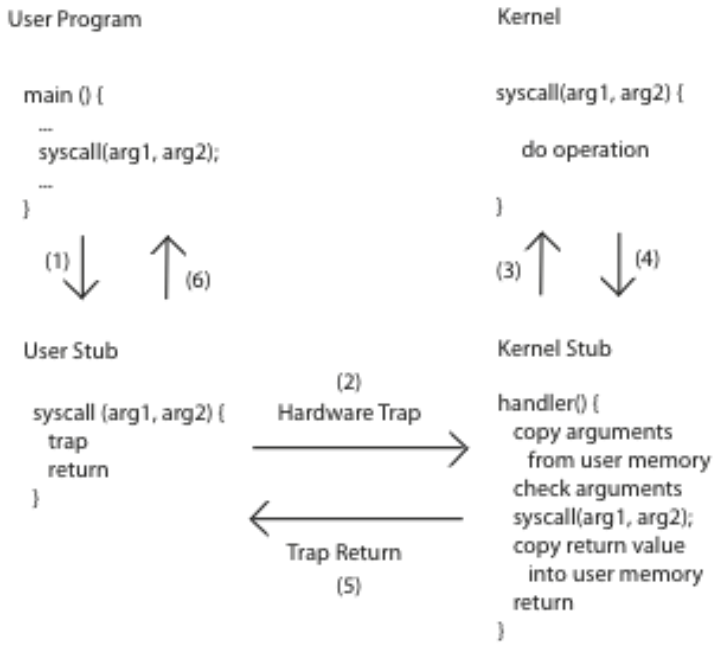
Lucido 37-39 – 02 Kernel.pptx



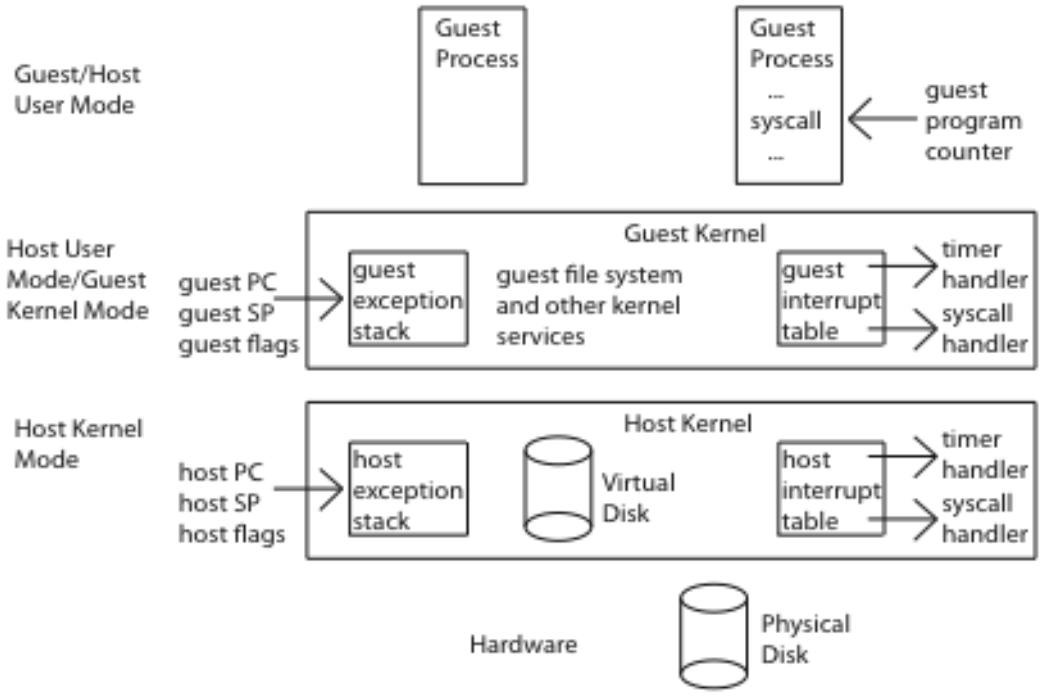
Lucido 50-51 – 02 Kernel.pptx



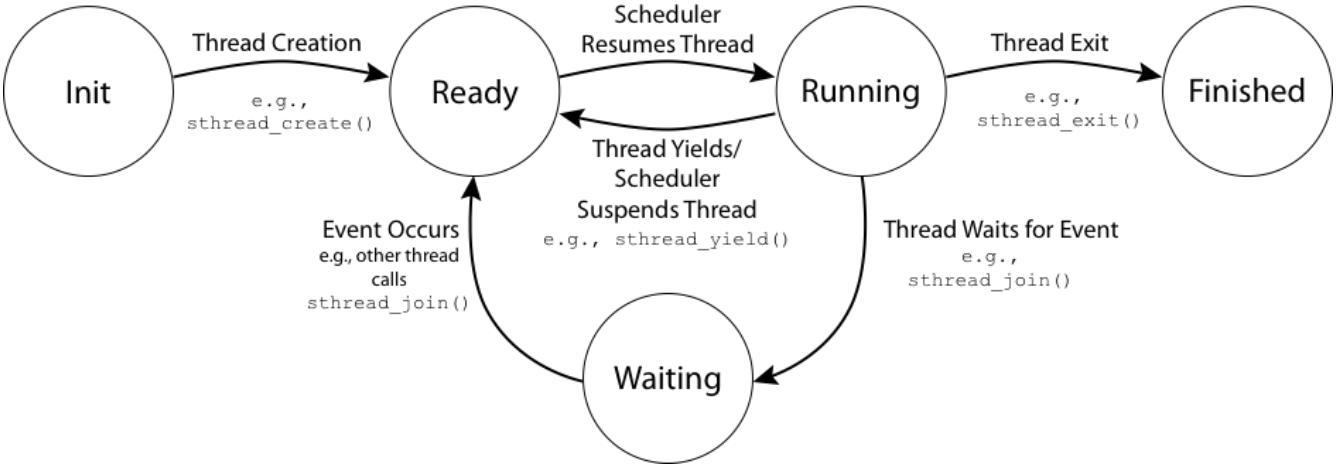
Lucido 46 – 02 kernel.pptx



Lucido 53 – 02 kernel.pptx



Lucido 9 – 04 concurrency.pptx



Lucido 15 – synchronization.pptx

Thread A	Thread B

leave note A
while (note B) // X
  do nothing;
if (!milk)
  buy milk;
remove note A
	leave note B
	if (!noteA){ // Y
	  if (!milk)
	  buy milk
	  }
	remove note B

Lucido 17 – synchronization.pptx

Le variabili Top e Stack sono condivise. Spiegare perché le due sezioni di codice sono sezioni critiche

Thread A	Thread B
… Top++ Stack[top]=y …	… Z=Stack[top] Top-- …

Lucido 25 – synchronization.pptx

```
tryget() {
  item = NULL;
  lock.acquire();
  if (front < last) {
    item = buf[front % size]
    front++;
  }
  lock.release();
  return item;
}
```

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

Lucido 28 – synchronization.pptx

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

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

Lucido 38 – synchronization.pptx

```
get() {
 lock.acquire();
 if (front == last) {
    self = new Condition;
    nextGet.Append(self);
    while (front == last)
      self.wait(lock);
    nextGet.Remove(self);
    delete self;
 }
```

```
 item = buf[front % size]
 front++;
 if (!nextPut.empty())
          nextPut.first()->signal(lock);
 lock.release();
 return item;
}
```

Lucido 41 – synchronization.pptx

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

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

Lucido 43 – synchronization.pptx

```
SpinlockAcquire() {
   while (TestAndSet(&lockValue) == BUSY)
      ;
}
SpinlockRelease() {
   lockValue = FREE;
}
```

Lucido 44 – synchronization.pptx

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

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

Lucido 47 – synchronization.pptx

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

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

Lucido 48 – synchronization.pptx

```
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();
}
```

Lucido 51 – synchronization.pptx (Take 3)

```
wait(lock) {
  sem = new Semaphore;
  queue.Append(sem);   // queue of waiting threads
  lock.release();
  sem.P();
  lock.acquire();
}
signal() {
  if !queue.Empty()
    sem = queue.Remove();
    sem.V();                  // wake up waiter
}
```

Lucidi 9 e 10 – 05-a

```
// 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);
        }
        mutex.Release();
```

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

<scrive>

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

Lucido 36 – advsynch.pptx

```
For each resource Rk:    D: availability vector
                         (Dk number of available units of resource Rk)
For each process Pj : Aj: assignment vector;  Ej: vector of residual requirements;
                         Ej≤D if Ejk ≤ Dk 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
```

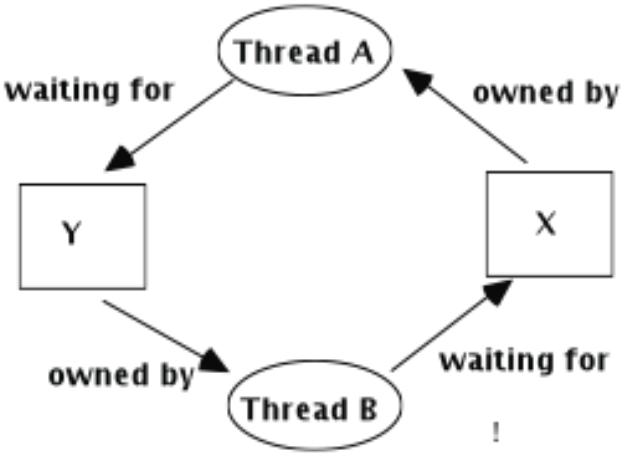
Lucido 5 – 06a.pptx

```
…
while (true) {
                // il filosofo di indice i decide di mangiare: protocollo per mangiare //
        lockBastoncino[i].Acquire();
                //il filosofo i si sospende se non può ottenere il bastoncino situato alla sua sinistra //
         lockBastoncino[(i+ 1) mod N].Acquire();
                // il filosofo i si sospende se non può ottenere il bastoncino situato alla sua destra //
        < il filosofo di indice i mangia >
                //il filosofo di indice i  ha finito di mangiare: protocollo per pensare //
        lockBastoncino[i].Release();
        lockBastoncino[(i+ 1) mod N].Release();
        // il filosofo rilascia i due bastoncini situati alla sua sinistra e alla sua destra //
}
```

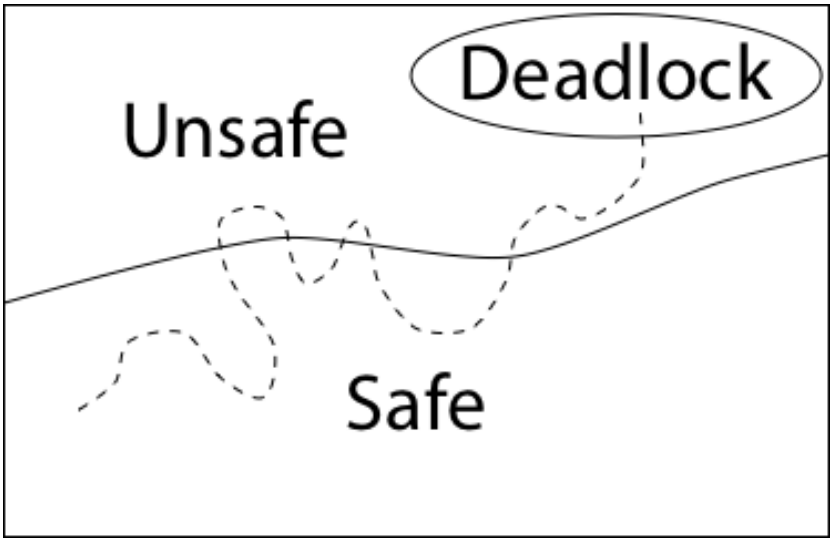
Lucidi 9 e 10 – 06a.pptx

```
…
while (true) {
                    // il filosofo di indice i decide di mangiare: protocollo per mangiare //
          mutex.Acquire();
          stato[i]= HaFame;
          while  (stato[(i- 1) mod N] == mangia) || (stato[(i+ 1) mod N] == mangia )          {
                                        attesaFilosofo[i].Wait(&mutex);
          }
          stato[i]= mangia;          //ha ottenuto ambedue i bastoncini //
          mutex.Release();
          < il filosofo di indice i mangia >
                    // il filosofo di indice i  ha finito di mangiare: protocollo per pensare //
          mutex.Acquire();
          stato[i]=pensa;
          if (stato[(i - 1) mod N]== HaFame) && (stato[(i - 2) mod N]<> mangia) {
                              // riattiva il filosofo (i-1) mod N se può ottenere entrambi i bastoncini //
                    stato[(i - 1) mod N] = mangia;
                     attesaFilosofo[(i- 1) mod N].Signal(&mutex);
          }
          if (stato[(i + 1) mod N]== HaFame) && (stato[(i + 2) mod N]<> mangia) {
                              // riattiva il filosofo (i+1) mod N se può ottenere entrambi i bastoncini //
                    stato[(i + 1) mod N]= mangia;
                     attesaFilosofo[(i + 1) mod N].Signal(&mutex);
                    }
          mutex.Release();
          }
```

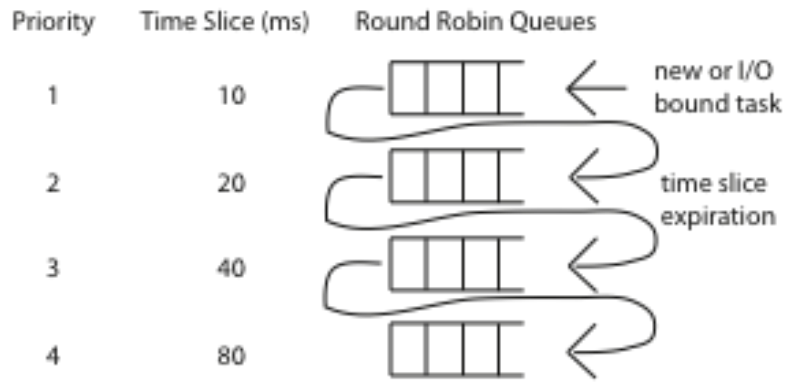
Lucido 15 – 06 adv synchronization.pptx – discutere le condizioni dello stallo



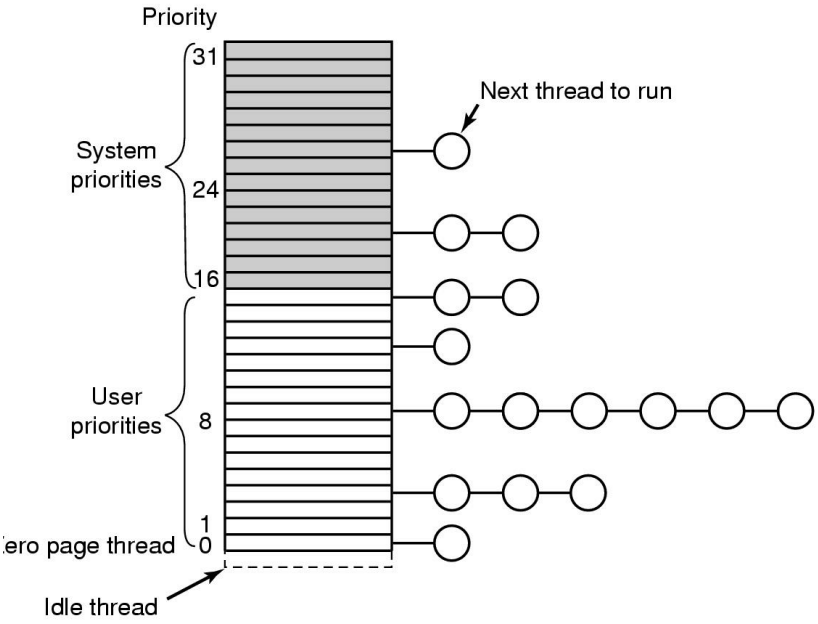
Lucido 30 – 06 adv synchronization.pptx – spiegare concetti di stato sicuro/non sicuro e algoritmo del banchiere



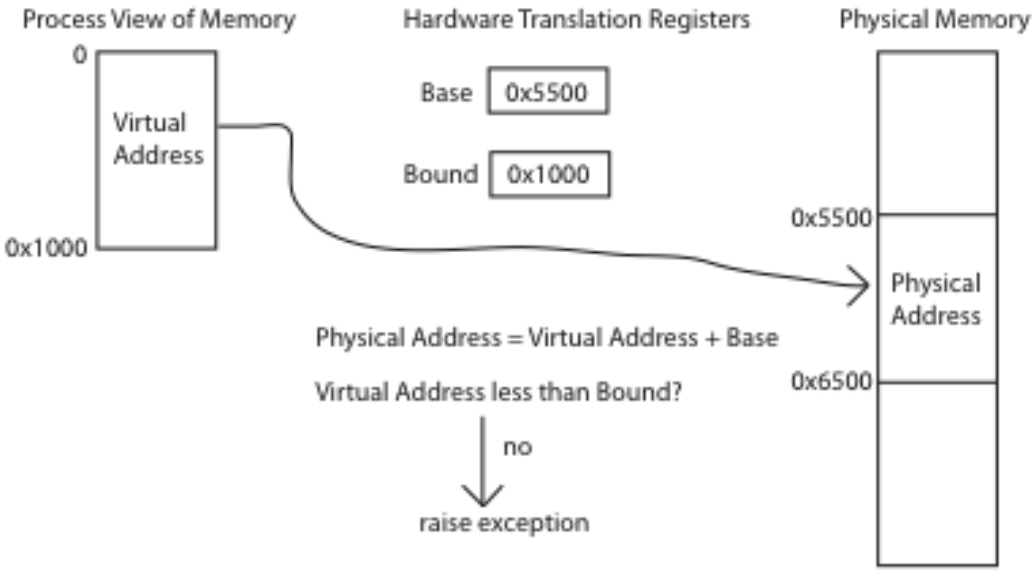
Lucido 26 – 07 scheduling



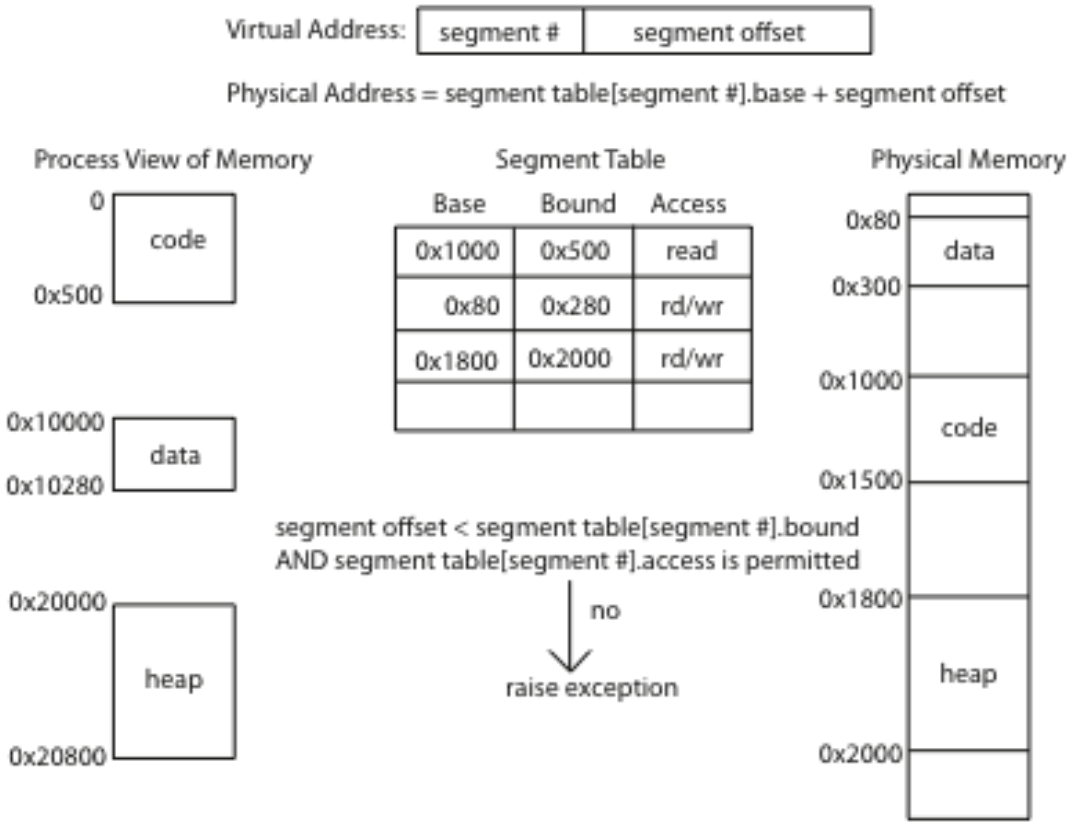
Lucido 28 – 07 scheduling.pptx



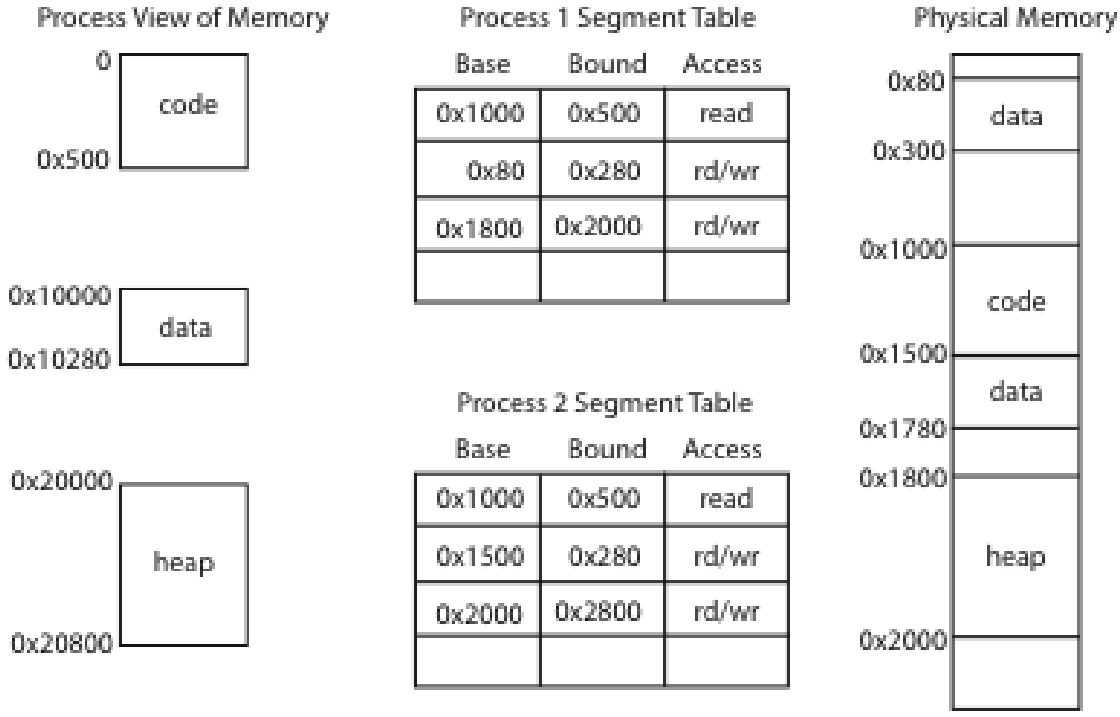
Lucido 13 – 08 address.pptx



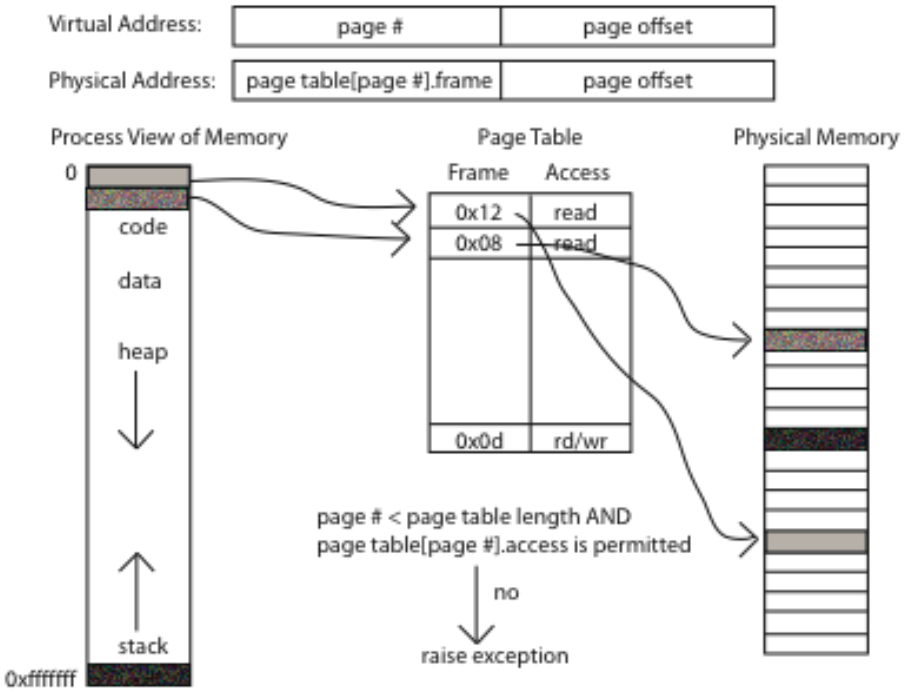
Lucido 24 – 08 address.pptx



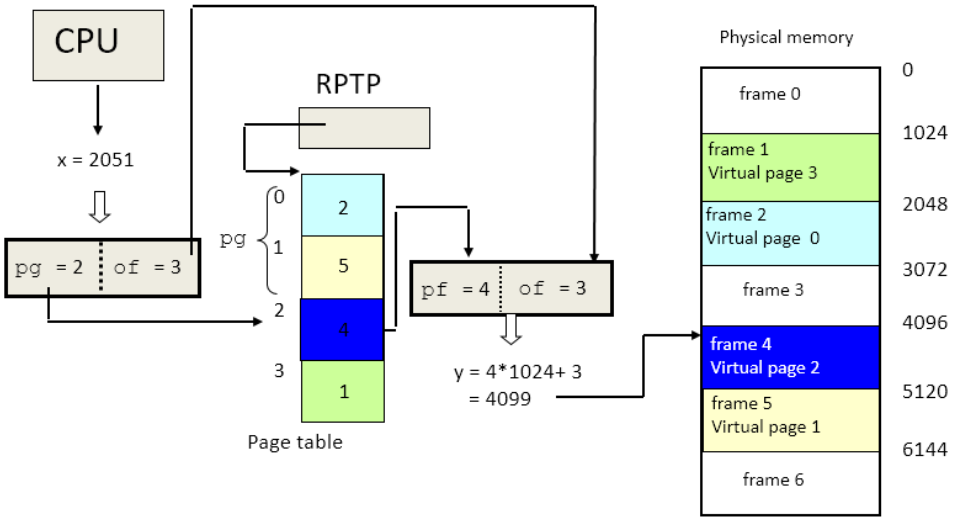
Lucido 28 – 08 address.pptx



Lucido 31 – 08 address.pptx

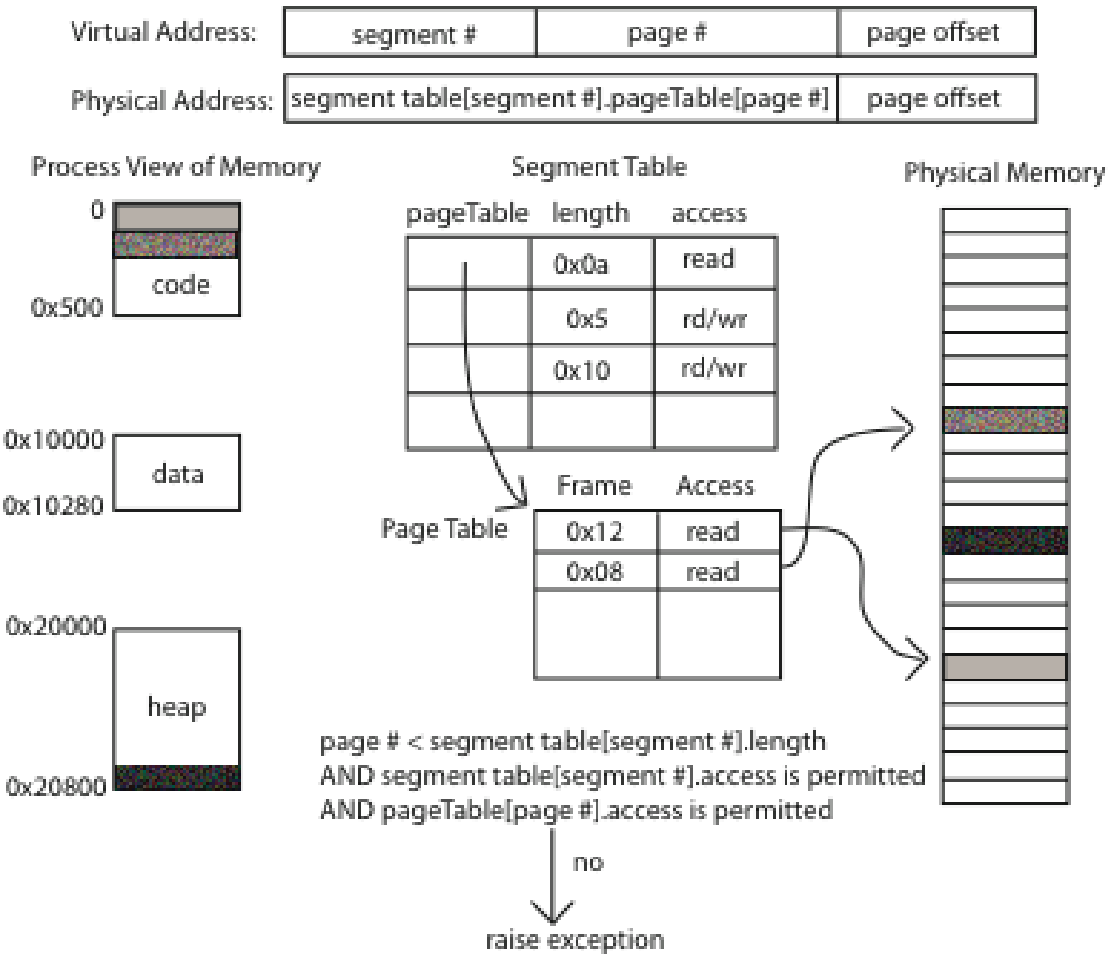


Lucido 32 – 08 address.pptx

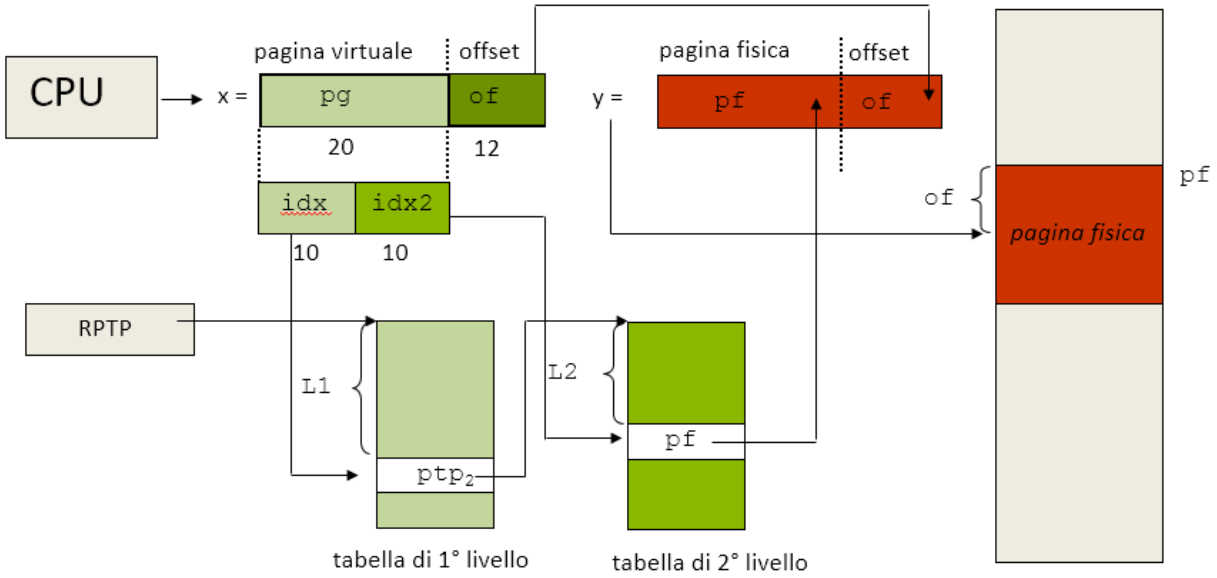


- RPTP: pointer to page table start
- RLTP: page table length

Lucido 40 – 08 address.pptx

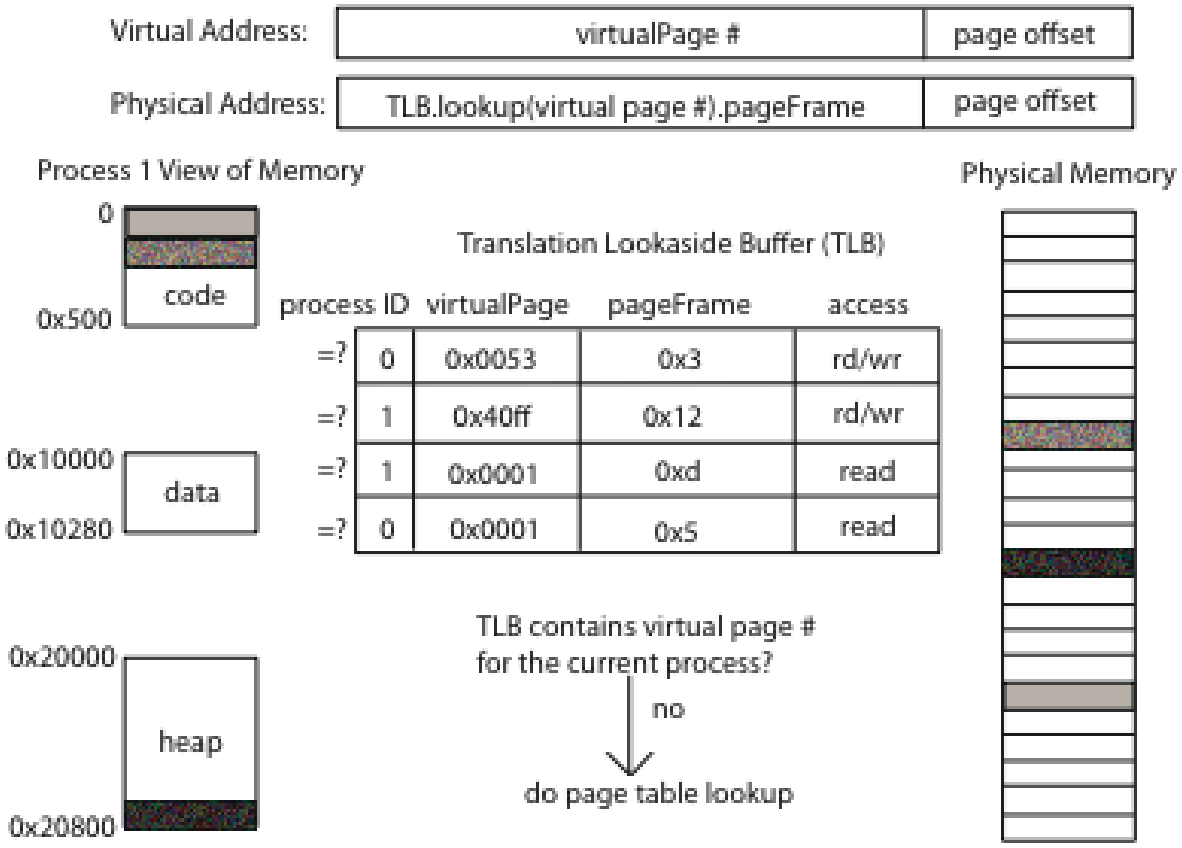


Lucido 42 – 08 address.pptx

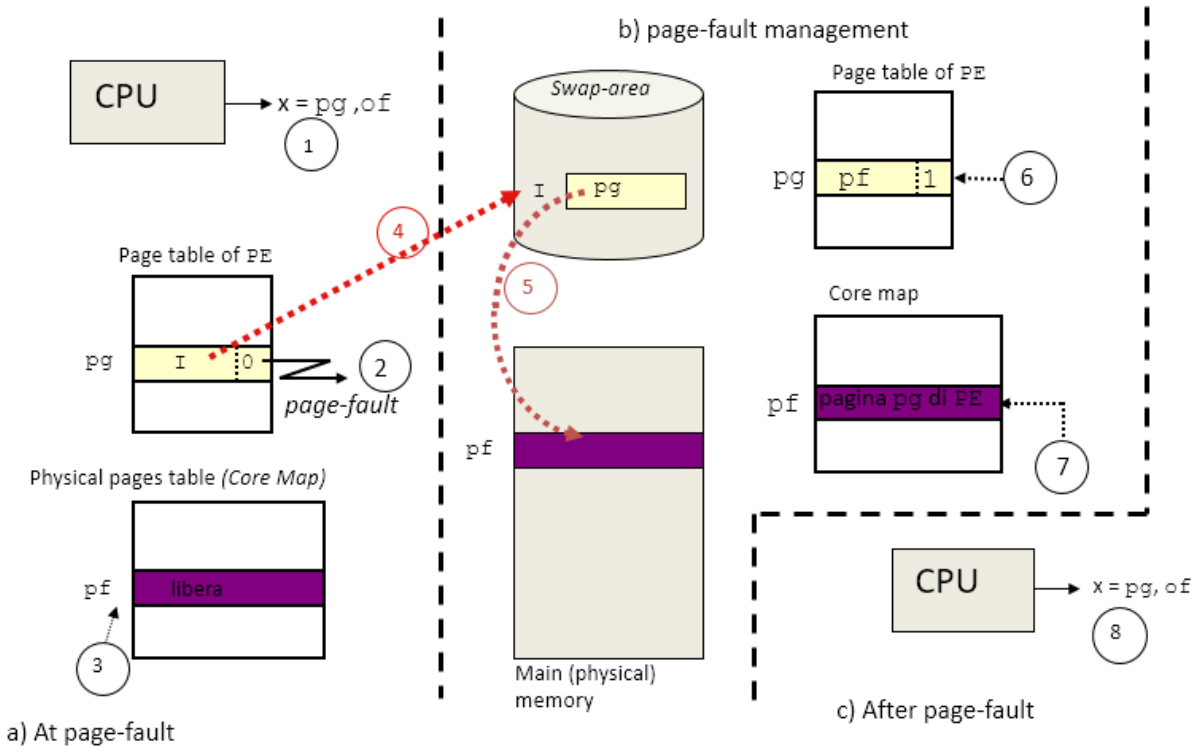


Caricamento dinamico delle tabelle delle pagine di secondo livello
==> minore occupazione di memoria

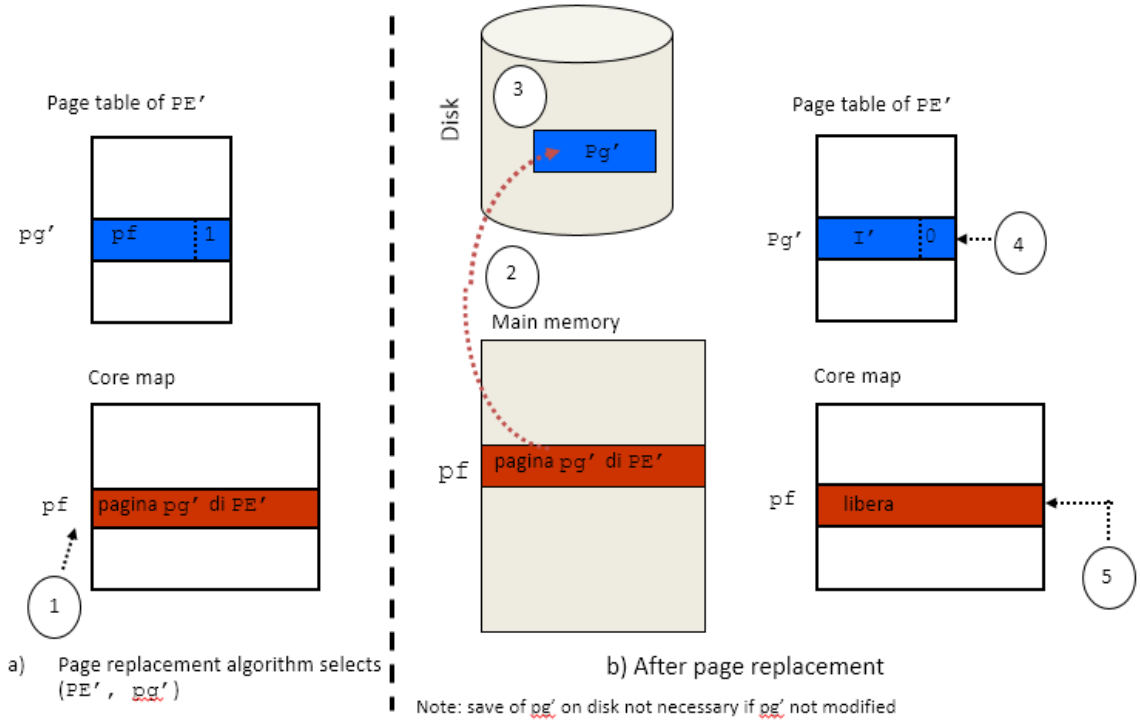
Lucido 58 – 08 address.pptx



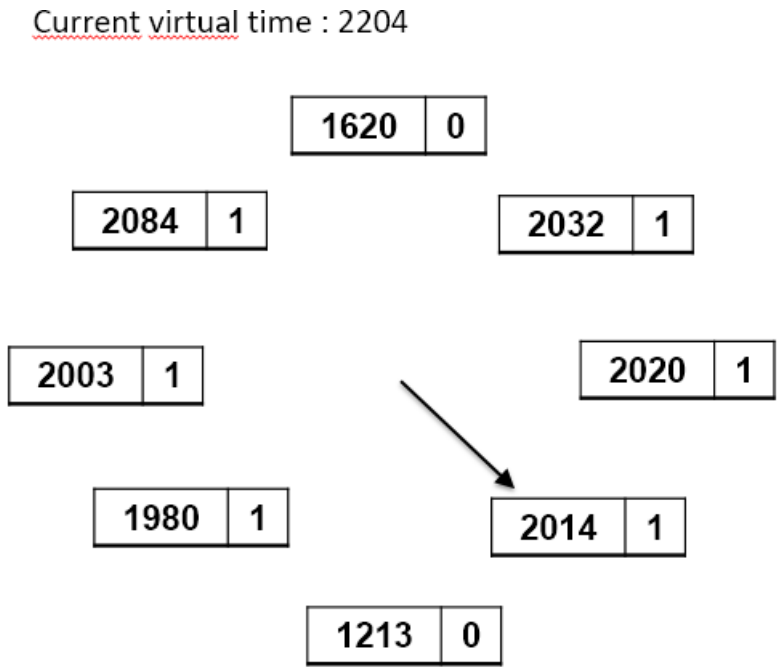
Lucido 3 – 09 caching.pptx



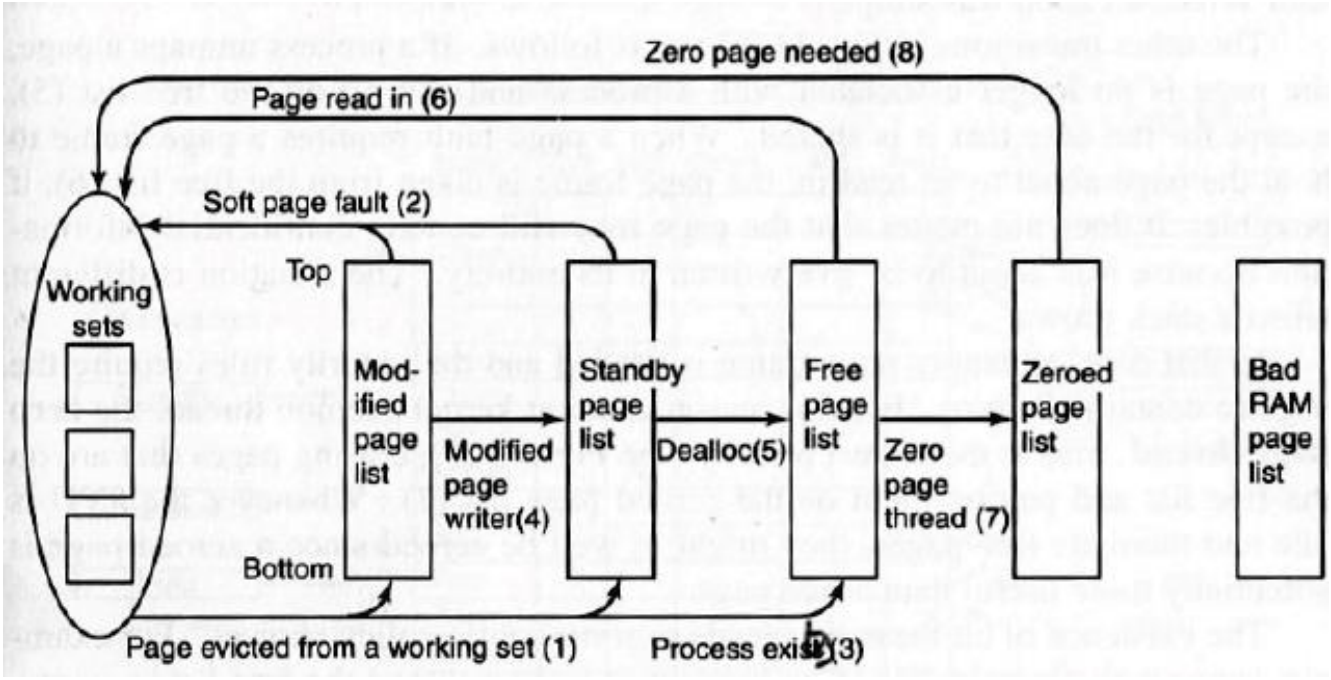
Lucido 4 – 09 caching.pptx



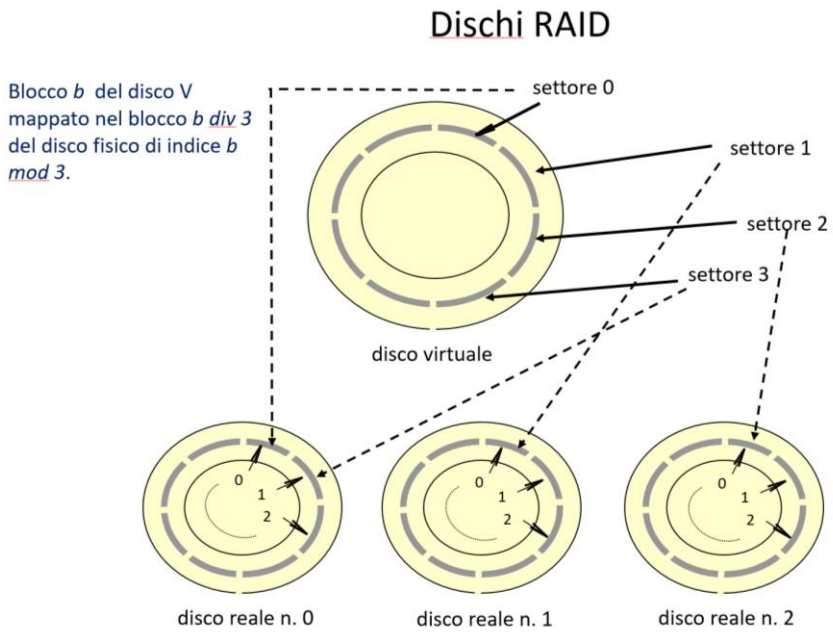
Lucido 46 – 09 caching.pptx



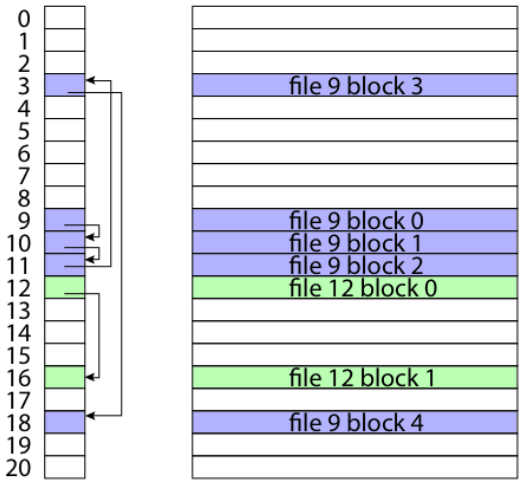
Lucido 75 – 09 caching.pptx



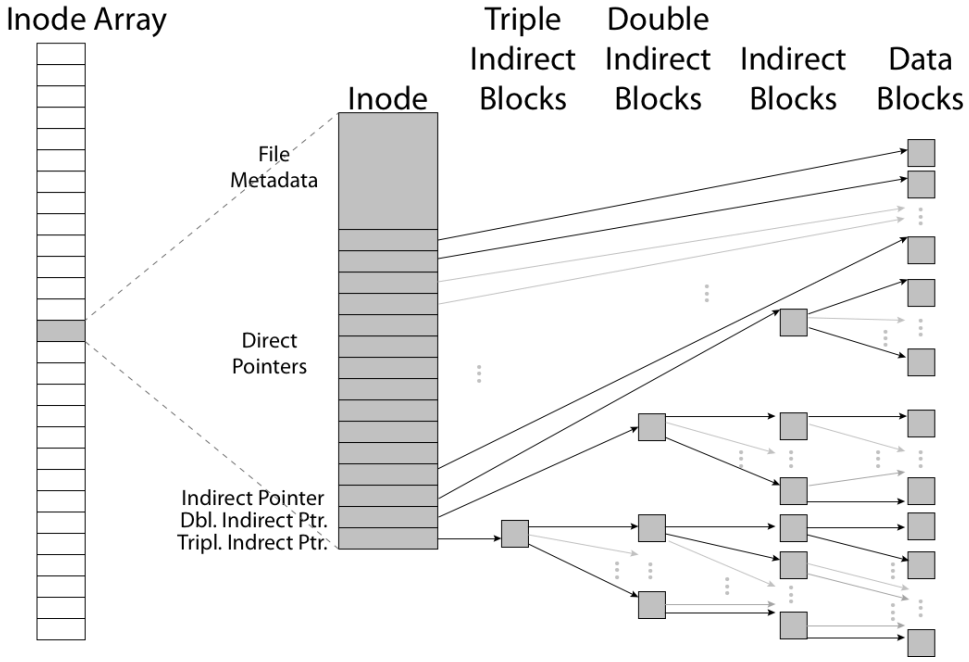
Lucido 70 – 11-12 storage



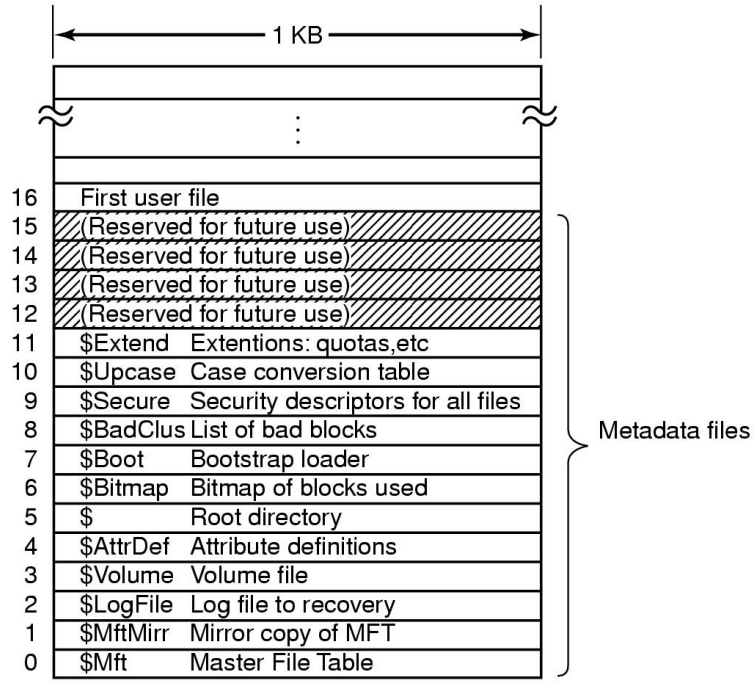
Lucido 16 – 13-filesys.pptx



Lucido 24 – 13-filesys.pptx


Inode Array
Inode
Triple Indirect Blocks
Double Indirect Blocks
Indirect Blocks
Data Blocks
File Metadata
Direct Pointers
Indirect Pointer
Dbl. Indirect Ptr.
Tripl. Indrect Ptr.

Lucido 38 – 13-filesys.pptx



Lucido 41 – 13-filesys.pptx