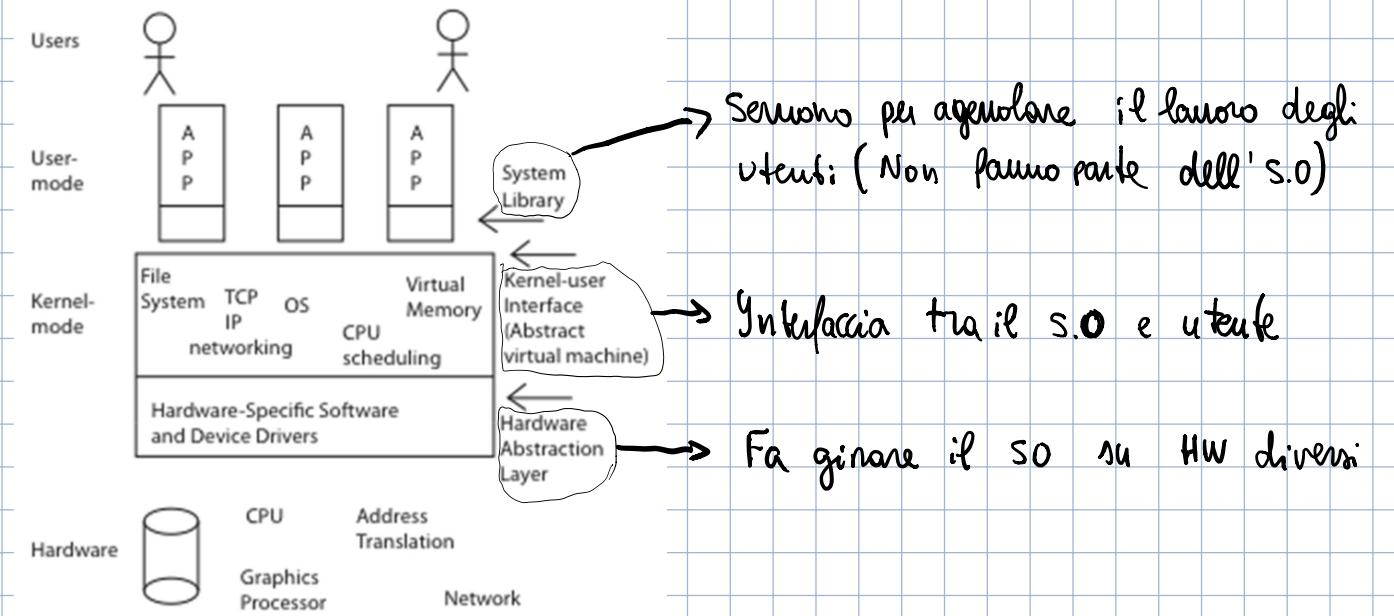


Rianunto SOL-A

Sistema operativo: Software per gestire le risorse del computer per i suoi utenti e applicazioni



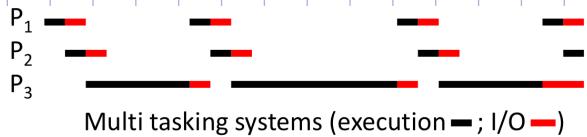
Ruoli del S.O.:

- 1) **Albito:**
 - Gestisce l'allocazione delle risorse tra i vari utenti e applicazioni
 - Usa meccanismi di isolamento tra utenti differenti e applicazioni diverse
 - Gestisce Comunicazione tra utenti / applicazioni
- 2) **Illusorista:** Ogni applicazione sembra avere l'intera macchina per se stessa
(infiniti processori, memoria ecc...)
- 3) **Colla:**
 - Provvede servizi alle applicazioni (library, icons, ...)
 - Semplifica lo sviluppo delle applicazioni

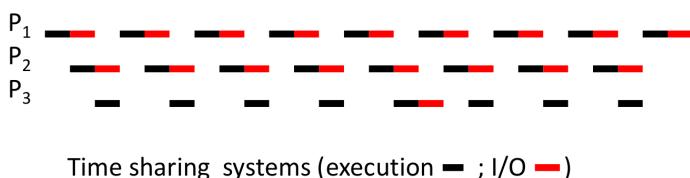
Caratteristiche che un S.O deve assicurare:

- 1) Reliability (affidabilità): Il sistema fa quello che dovrebbe fare?
- 2) Availability (disponibilità): Assicurare il funzionamento del S.O in un certo momento
- 3) Security (sicurezza): Difendersi da possibili attacchi al sistema
- 4) Portability (portabilità): Poder usare gli stessi programmi su S.O diversi e utilizzare S.O su HW diversi
- 5) Performance: Misura delle qualità di un S.O
 - latenza: Quanto tempo un'operazione impiega per essere completata
 - throughput: Quante operazioni possono essere fatte per unità di tempo
 - Overhead: Quantità lavoro extra eseguito dal S.O
 - Equità: Riuscire a dare a ogni app le risorse di calcolo richieste
 - Predictability: Predirne quanto tempo ci vuole ad eseguire un programma

Time Sharing e multitasking



→ Ogni volta che ho un'interruzione passo ad un altro processo (senza limiti di tempo)



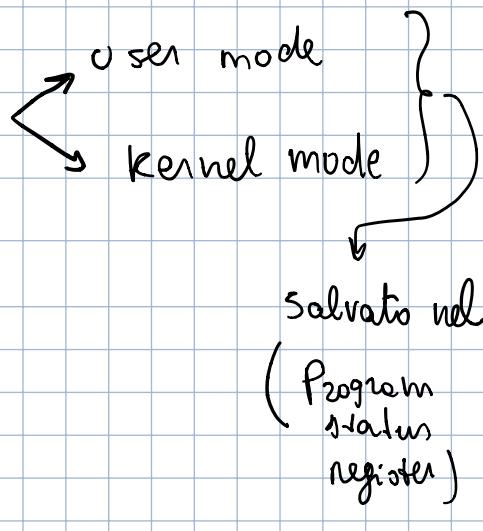
→ Da un max di tempo a ogni processo anche se non finisce (equo)

Time Sharing: Condividere una risorsa del computer tra più utenti allo stesso momento

Multi-tasking: Eseguire più programmi allo stesso momento

Architettura di un computer (Richiamo)

- 1) CPU (1 o più) → CPU Mode :
- 2) Memoria principale (RAM)
- 3) Unità di dispositivi (periferiche)
- 4) Gestore delle Interventi
- 5) Direct memory access (DMA)



Permette alle periferiche di accedere direttamente alla memoria

CPU

- Registri generali
- PC, SP, Program Status Register (PS)
 - contiene lo stato dell'ultima istruzione (FLAGS)
 - CPU MODE (USER e SUPERVISOR) (kernel)
 - Bit interruzioni

Attenzione : Nel ciclo fetch - decode - execute le interruzioni vengono gestite dopo l'esecuzione dell'istruzione.

Kernel Abstraction (S.O.)

I programmi possono essere eseguiti in 2 modi:

- Kernel (supervisor) - Mode : Esegue con privilegi completi
- User - Mode : Esegue con meno privilegi

La modalità del processo

Programma: Sequenza statica di istruzioni



Un programma può attivare più processi

Processo: Sequenza di attività attivate da un programma, con privilegi limitati.

Parti della mem. centrale
↑

Ogni processo ha : il suo codice, i suoi dati, il suo heap e il suo stack

→ **Tipi di processo:**
(Venne dato da un bit del PS)

Kernel : Può eseguire tutte le istruzioni
Utente : Esegue solo istr che l'so permette

→ Vengono tracciati dal **Procen control block (PCB)** (ne abbiamo uno per ogni processo)

→ tutte le PCB sono contenute nella **process table** (come vettore di PCB)

→ **threads:** lightweight process che esegue una sequenza di istruzioni di un processo (multo-processi)
(Posso avere più threads per ogni processo)

→ **Address space:**insieme di diritti di un processo
- Memoria che può essere acceduta
- Altri permessi che un processo ha

PCB: Struttura dati interna al s.o utilizzata per tracciare un processo

↓
perché non devo permettere ad un utente di toccare il PCB di un processo

→ **Struct che Contiene :**

- Nome del processo (Distingue i processi) → Può essere l'indirizzo associato nel vettore della **process table**
- Puntatori ai threads del processo
- Memoria allocata al processo (Mem. che può accedere)
- Altre risorse

Ahi: lo stesso codice può essere eseguito da più processi

Supporto Hardware:

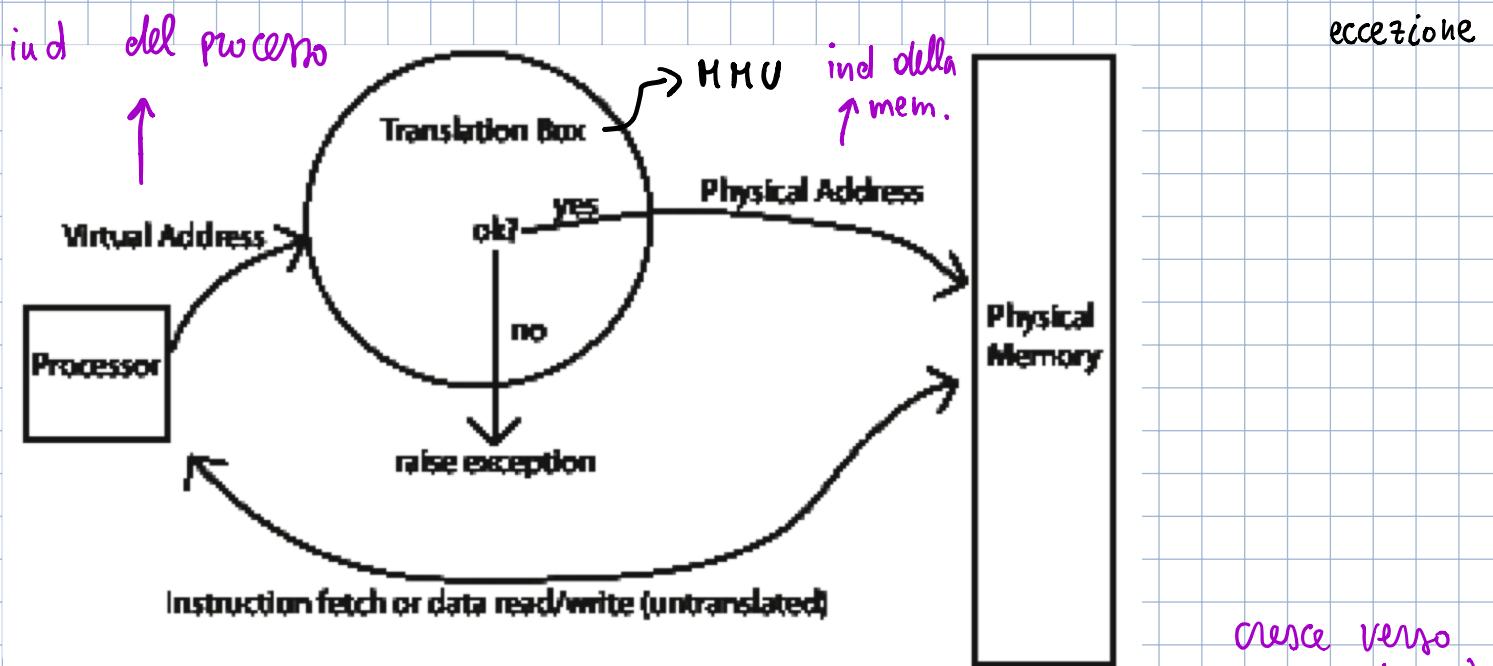
- 1) Istruzioni privilegiate : Istruzioni ASM che possono essere eseguite solo in Kernel-Mode



Se un programma utente prova ad eseguirle viene ucciso il processo / errore di compilazione

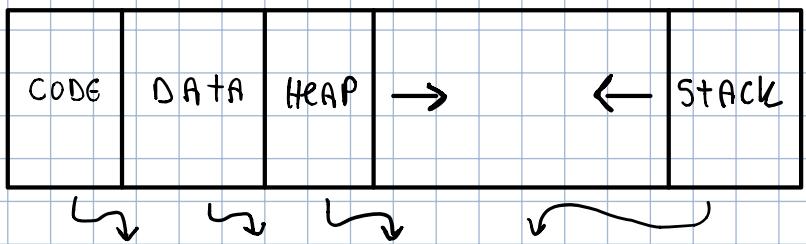
- 2) Memory Protection : IL S.O. da un indirizzo base di mem. al processo e una quantità di mem. su cui può spaziare.

Se prova ad accedere ad un indirizzo sbagliato →



layout processo (iud. virtuale):

più ind logici che fisiici



layout mem. fisica (iud. fisico):



3) Timer: Unita' firmware che interrompe perio di esecuzione il
processore (tempo massimo di esecuzione)

Ha un valore settato che decrementa (bit). Invia
sempre al processore l'OR negato dei bit.

Quando l'OR dei bit negati è 1 (bit tutti a 0)
arriva un'interruzione e il controllo passa all'S.O.

Venne chiamato il programma di gestione del timer
interrupt.

IL TEMPO MAX DI ESECUZIONE VIENE SETTATO DAL S.O

Attenzione: le interruzioni possono essere differenti
dal S.O

Il tempo nel timesharing è dettato dal timer il quale
viene settato dal S.O

es: loop di un programma

4) Mode Switch: Passare da modalità utente a Kernel e viceversa

→ Da user a kernel quando:

- Interruzioni: generate da I/O o timer
- Eccezioni: generate da comportamenti inaspettati di un programma
 - o maligni
- Chiamate di Sistema: operazioni richieste da un programma al S.O

↳ Sono limitate e controllate molto bene

→ Da Kernel a User Quando:

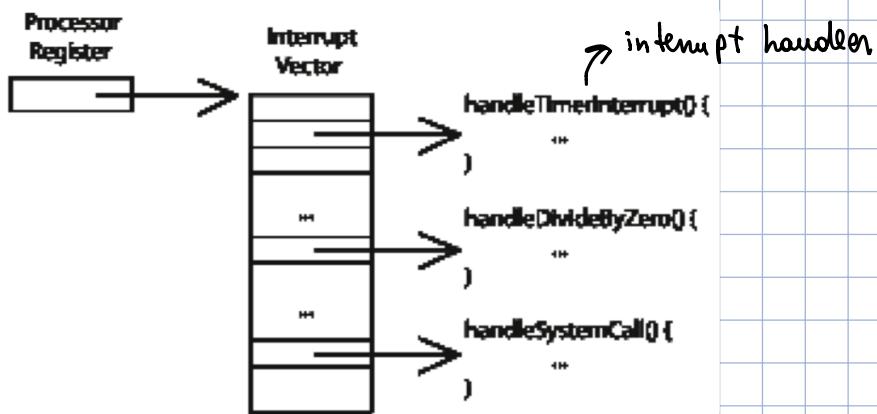
- Ritorno da interruzione/eccezione /System Call → Riprendo l'esec. interrotta
- Nuovo processo/thread → Salto alla prima istruzione
- Cambio di modalità processo/thread → Riprendo qualche altro processo
- User-level upcall → Notifica avvenuta a un prog. utente

Lo switch deve
essere fatto in un
ciclo di clock

Gestione delle interruzioni in modo sicuro

Un interrupt vector : Vettore (tabella) impostata dal S.O che contiene puntatori al codice da eseguire per ogni differente evento

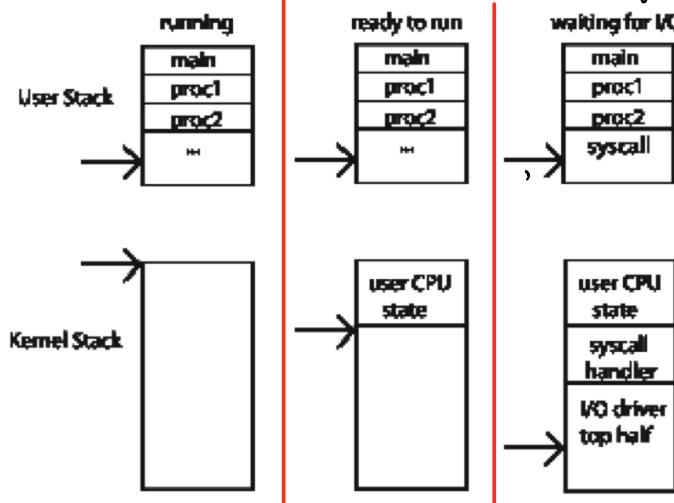
Essendo un punto di accesso l'S.O deve essere molto controllato



Kernel interrupt stack: Stack (Memoria di lavoro) usata dalle interruzioni

↓
→ Uno per ogni processo salvato nel Kernel

Ha locazione nel S.O perché vengono inserite informazioni sensibili



Un interrupt masking : Il S.O può abilitare / disabilitare le interruzioni (gestore non si blocca)

↓
Quando il gestore delle interruzioni è in esecuzione le interruzioni sono disabilitate.

Un interrupt Handler: Non bloccante, non fino a completazione, veloci
↳ Codice per gestire le interruzioni

Atomic transfer control : Singole istn. che cambia: →

- On interrupt (x86)
 - Save current stack pointer
 - Save current program counter
 - Save current processor status word (condition codes)
 - Switch to kernel mode
 - Switch to handler PC & kernel PSW
 - Vector through interrupt table
- Interrupt handler saves registers it might clobber

Gestione delle interruzioni:

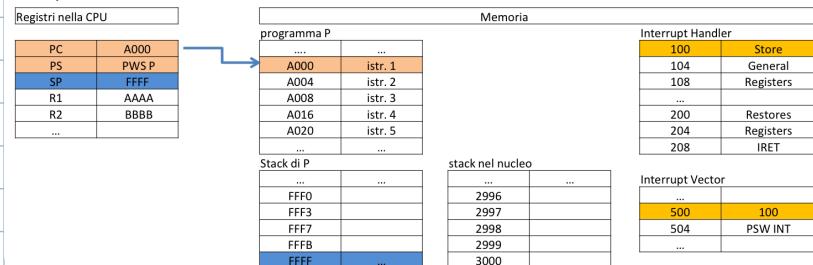
PC punta all'ind dell'istruzione

PS punta allo PSW del processo

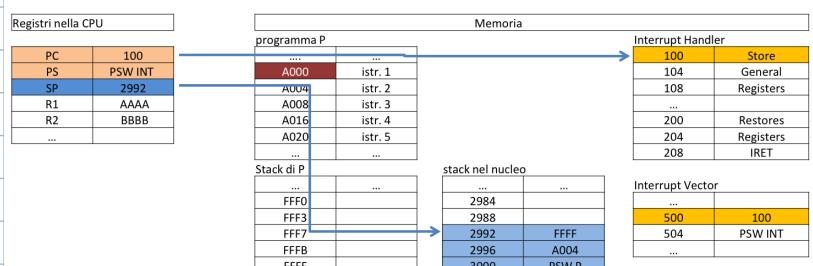
SP punta allo stack del processo

R₁ e R₂ contengono AAAA e BBBB

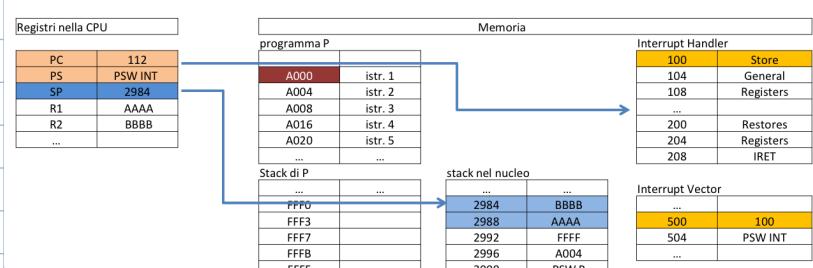
1) Initial state



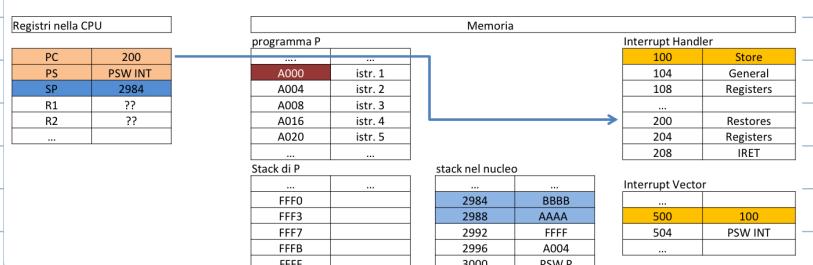
2) Interrupt recognized after instruction A000 (PC incremented to A004)



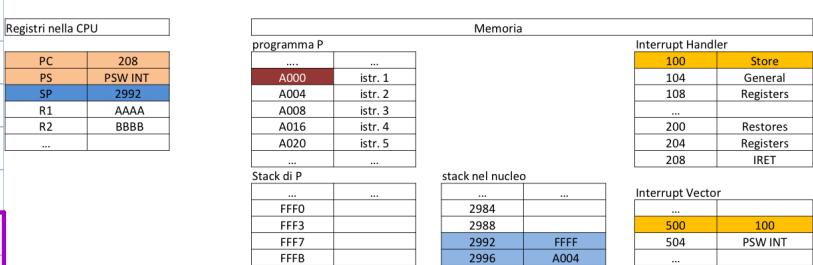
3) Stores general registers



4) Executes interrupt handler



5) Restores general registers



Esegue il codice dell'interrupt handler

PC punta all'istr di restor

Restore dei registri generali dallo stack

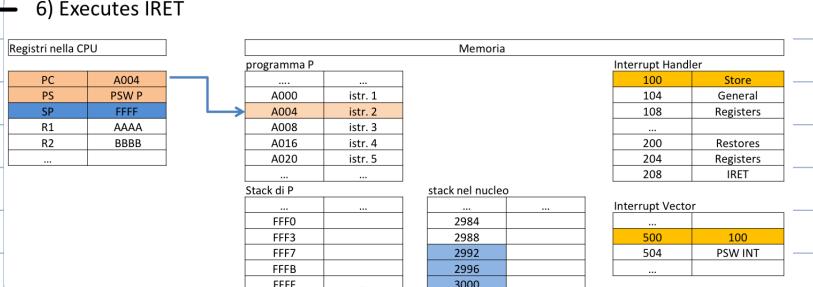
IRET:



- Ripristina PC, SP e PSW

- Switcha da kernel mode a user mode

- Abilita i interrupt



OSSERVAZIONE:

Nel caso dovermi mettere in esecuzione un altro processo, i dati nello stack del nucleo li salvo nel PCB del processo che è stato interrotto dall' interrupt. Poi prendo dal PCB del secondo processo i dati e li inserisco nello stack del nucleo.

Questo perché se devo mandare in exec. il 2° processo, mi serve il suo PCB, Stack Pointer, ecc...

System Calls (SVC = supervisor calls) => È una interruzione anche essa che è stata programmata

Gestore delle system calls:

- 1) Prende gli argomenti delle chiamate di sistema che ponono entro nel Reg. o in memoria dell'utente (tipicamente sullo stack), copiandoli dalla mem utente alla mem kernel.
- 2) Controlla se ci sono errori negli argomenti (es. apertura file non esistente)
- 3) Validare gli argomenti: Proteggere il Kernel da errore nel user code
- 4) Copiare i risultati nella memoria utente

Il procedimento per gestire il cambio di contesto è lo stesso delle interrupt hardware

Bisogna far attenzione a come si scrivono le SVC perché sono punti di accesso al S.O.

Upcalls (User-level interrupt) - Unix Signal

Sono chiamate che effettua il S.O verso il processo utente.

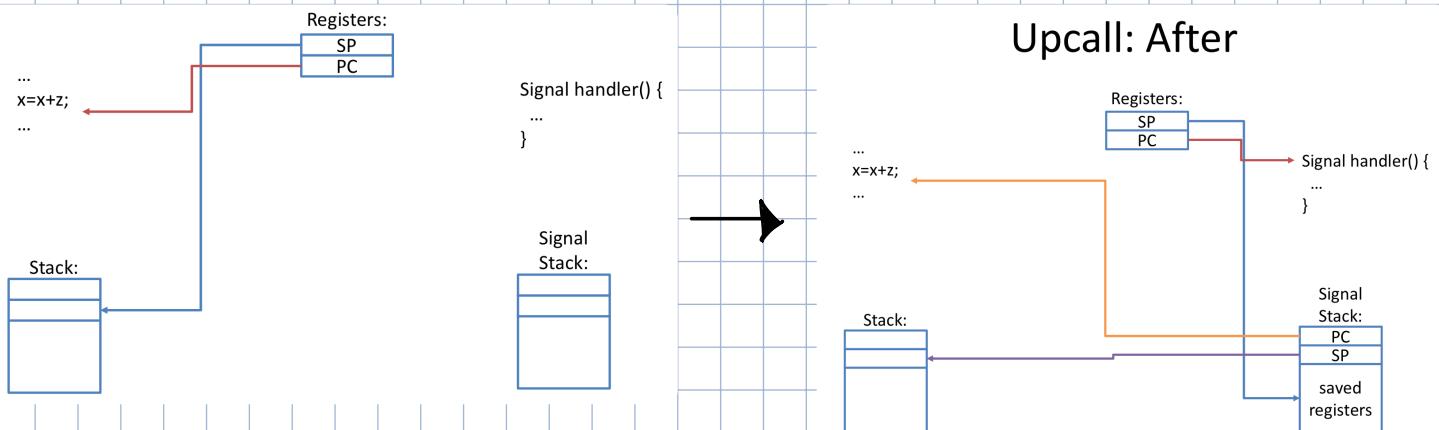
Si gestiscono allo stesso modo delle interruzioni (Metà procedimento).

Prendono le informazioni dal PCB del processo.

Indicano al processo utente che c'è succ. qualcosa che deve gestire (il processo utente deve avere il codice per gestirlo)

Duale
delle
interruzioni

- Interruzione che l'SO manda al processo utente
- Dopo il processo utente continua a fare il suo lavoro
- Ogni Unix signal ha uno stack
- I segnali possono essere mascherati



Booting

Bios \rightarrow Carica in memoria RAM ciò che serve per il Booting

\hookrightarrow Sta nella ROM

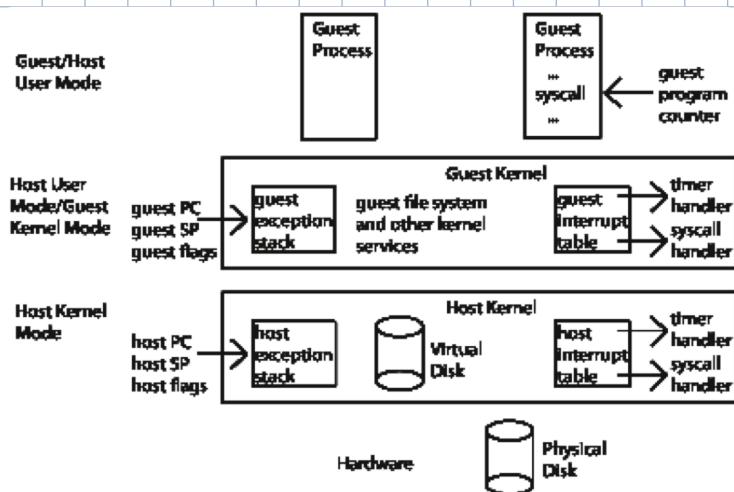
- 1) Carica Bootloader (istruzioni e dati) \rightarrow Informazioni per caricare il S.O
- 2) O.S (istruzioni e dati)
- 3) App di login (istruzioni e dati)

Virtual Machines

Consistono in macchine che ospitano dei sistemi operativi guest.

Abbiamo una macchina ospite che ha il suo S.O e poi un S.O ospite quindi tra l'HW e un processo utente ci sono 2 livelli di kernel.

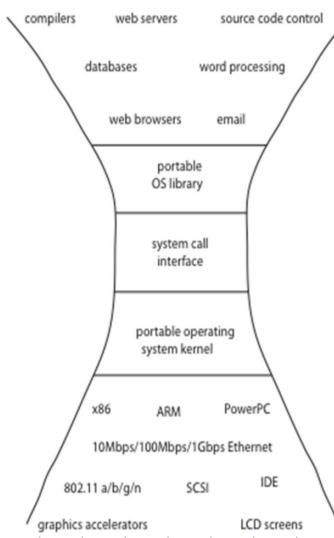
Aumenta la ricchezza perché ogni kernel fa un controllo quindi abbiamo un doppio controllo.



Quindi ho tutto duplicato: stack interrupt guest e host
interrupt vector guest e host

- Se interrupt è per VM → upcall
- Se interrupt è per un altro processo, rinstalla interrupt vector e ripristina kernel

Struttura S.O. :



→ Programmi a livello utente

→ librerie di sistema

→ Interfaccia tra S.O e utente

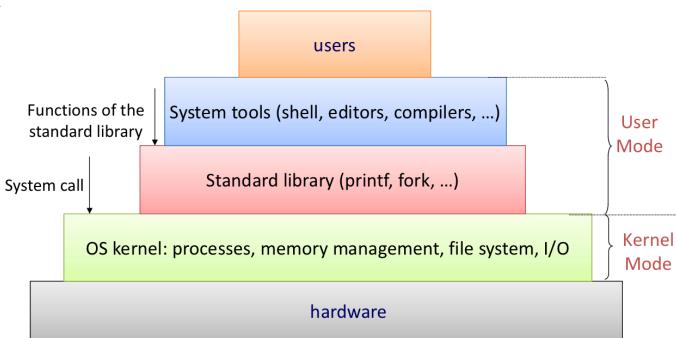
→ Kernel S.O portabile

→ HW

Architettura Unix: → S.O a nucleo grande (massimo)

tra system tools e standard library ci sono le funzioni della libreria standard

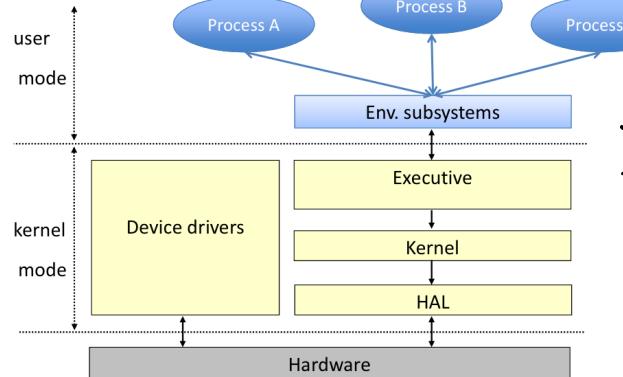
tra standard library e kernel ci sono le system call



- Della utente
- Strumenti utente
- libreria
- kernel

Architettura Windows → S.O a nucleo piccolo (minimo)

Windows architecture



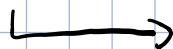
Ogni utente ha il suo (user mode)

executive : prende i dati dall'ambiente

S.O → Kernel : Contiene le fun. + importanti

Hardware abstraction : Usato dal kernel per comunicare con l' HW

Programming interface (esempio: Shell)



Job control System



permette agli utenti di creare e gestire un insieme di programmi per fare qualche task

Funzioni:

- Creare e gestire i processi (Fork, exec, wait)
- Fare I/O (open, read, write, close)
- Comunicazione fra processi (pipe, dup, select, connect)

Gestione dei processi UNIX

1) Unix Fork() → Crea una copia "Figlio" del processo Padre con lo stesso codice, lo copia del Kernel e user data ma con un PID diverso, quindi viene creato un PCB nel vettore dei PCB

- Ritorna 0 per il codice figlio
- Ritorna il PID del figlio al codice padre (sempre un numero negativo se ci sono errori)

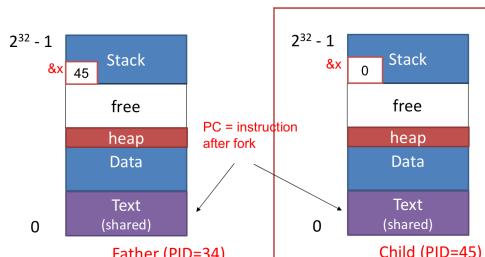
vettore PCB pieno

Piani del Fork():

- Crea e initializza il PCB nel kernel.
- Crea spazio di indirizzamento e lo inizializza con la copia dei dati dello spazio del padre
- Copia gli argomenti nello spazio di mem. dello spazio di indirizzamento
- Eredita il contesto di esecuzione del padre (es: Oggi file aperto)
- Informa lo scheduler che il nuovo processo è pronto per essere avviato

Implementing UNIX fork

Addressing spaces of the father and the child after a successful fork



Può tornare un errore se la process table è piena.

2) Unix Exec → Rimpiazza il codice eseguito dal processo
↓ Rimpiazza i dati

→ Eredita PCB e cambia lo spazio di indirizzamento

Example:

```
int execl(char *pathname, char *arg0,  
         ... , char *argN, (char*) 0)  
- Pathname is the name of an executable file  
- argN[1]...argN [...] are the arguments passed to  
the program  
- list terminated with (char *) 0
```

Se eseguito con successo il processo esegue un altro programma senza ritornare un codice di errore (File non eseguibile)

Dopo l'esecuzione:

- Mantiene pid
- Mantiene PCB
- Resetta i segnali
- Mantiene lo stack kernel
- Mantiene le risorse assegnate

Un processo può terminare:

- Eccezione per azioni non consentite
- Invocando la System Call Exit
 - Il processo ritorna un Exit value al padre
 - Valore della system call wait
 - Se il padre non ha ancora invocato la wait il processo terminato va nello stato "zombie"
 - Se il padre è già terminato, il processo iniziale aspetta la terminazione del figlio

3) Exit e Wait

Void exit (int status); →

- status = codice di terminazione
- Exit non ritorna mai
- libera la memoria, rilascia le risorse
- Se switcha allo "zombie mode", tiene il PCB fin quando il padre non invoca la wait

int wait (int *status); → Status = PID del processo terminato o di un codice di errore

processo
padre di:
tutti

4) Unix I/O

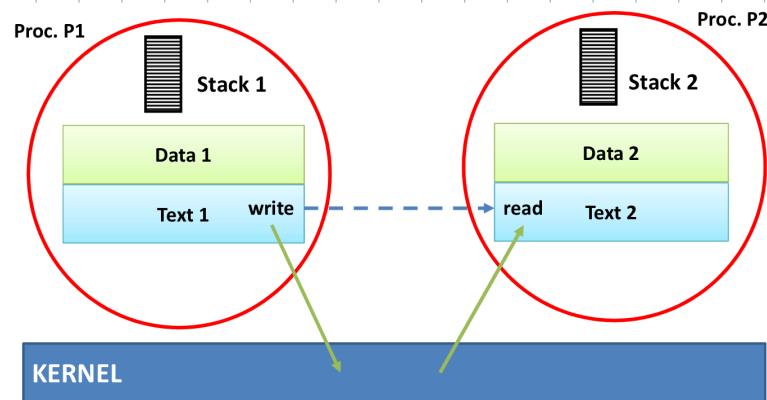
- Uniformity: tutte le operazioni su tutti i file usano lo stesso insieme di system calls: open, close, read, write
- Aprire il file prima di usarlo: open ritorna una descrizione del file per l'uso successivo di esso
- Byte oriented
- Kernel - buffered read / write
- Chiusura esplicita per "garbage collect": i descrittori dei file aperti

Concurrenzia → Capacità di gestire più cose alla volta \neq Parallelismo
"Multiple things at once (MTAO)"



Implementazione: Ogni funzione in un processo separato dove:

- Ogni processo ha il suo spazio di memoria privato \rightarrow Non lo condividono
- Gli processi possono cooperare usando meccanismi offerti dall'OS



Threads: Singole esecuzioni sequenziali che rappresenta un task schedulabile separatamente

Single thread user program: Un thread

Multi thread user program: Molti threads, che condividono strutture dati, separati dagli altri processi utente

Multi thread Kernel: Molti threads, che condividono strutture dati del kernel, capaci di utilizzare istruzioni privilegiate

Attenzione: Ogni processo può eseguire un solo thread simultaneamente.

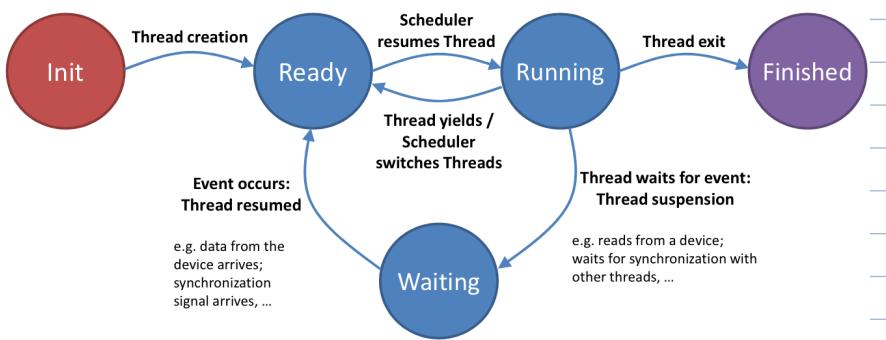
↓
I thread in esecuzione vengono decisi dallo scheduler.

Lo scheduler può decidere quando eseguirli, quando fermarli e in quale ordine.

Implementare i threads:

- thread control block (TCB) → struttura dati che contiene info sul thread
- insieme di operazioni sui thread
- uno scheduler: funzione dell'OS che assegna ai thread il processor/i

Ciclo di vita dei threads:



Location of thread's per thread state

State of thread	Location of TCB	Location of registers
INIT	Being created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List, then Deleted	TCB

Modi in cui possono essere implementati i threads

- Multiple user-level threads, dentro un processo Unix (Early Java)
- Multiple single-threaded processes (Early Unix)
- Mix di single/multi threaded process e kernel threads (Linux, Mac OS, Windows)
- Scheduler Activations (Windows)

User-level threads → threads implementati tramite una libreria a livello utente

- S.O non consapevole dei threads a livel utente
- thread table fra ogni processo
- Scheduling dei threads implementato dal run-time support del processo

↳ threads possono usare `thread_yield()` per rilasciare il processo

- Una invocazione a una system call bloccante, blocca tutti i thread del processo

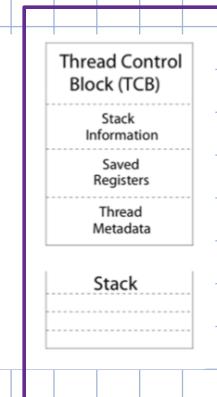
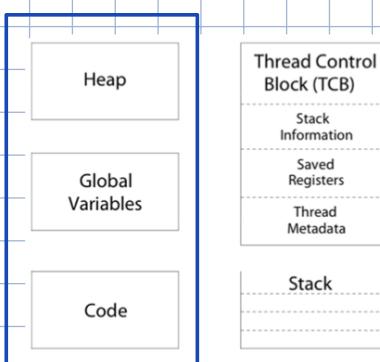
Implementazione:

- 1) `thread_fork(func, args)` :
 - Alloca TCB
 - Alloca lo stack
 - Costruisce lo stack frame per la base dello stack (stub)
 - Mette func, args sullo stack

- 2) `stub(func, args)` :
 - Chiama `(*func)(args)`
 - Chiama `thread_exit()`

Stato dei threads → thread table → uno per ogni processo

Stato condiviso da tutti i threads



Stato per ogni thread

Pro:

- 1) Creazione, terminazione e cambio di contesto molto efficienti perché viene fatto tutto a livello user (Non necessita delle system call, solo la threads library in caso di cambio di contesto lo spazio di indirizzamento resta lo stesso)
- 2) Può essere implementato su ogni SO che non supporta il multi-threading

Contro:

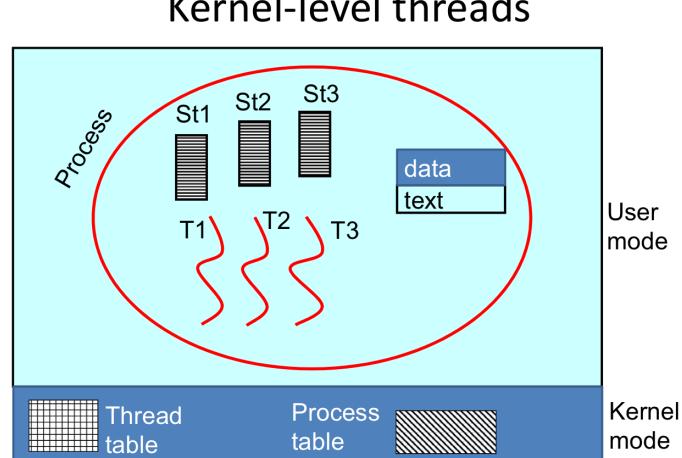
- 1) Le system calls bloccanti bloccano tutti i threads a livello utente di un processo
- 2) Non prende vantaggio dalle architetture multiprocessore

Kernel level - threads → threads implementati nel Kernel

- tabella dei threads nel Kernel
- Creazione, terminazione e cambio di contesto attivano le system calls
- thread differenti dello stesso processo può eseguire in parallelo su processori differenti
- thread scheduling implementato dal SO
- threads possono invocare system call bloccanti → solo chi le invoca viene bloccato
- Il cambio di contesto è effettuato dal Kernel

TCB e PCB:

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



Rappresentazione dei thread e processi:

- PCB → strutture dati associate a ogni processo
- Process table → contiene tutti i PCB, sta nel Kernel ed è uno per l'intero sistema
- TCB → uno per threads
- thread table → nel Kernel, uno per l'intero sistema

Cambio di thread \rightarrow Motivi: Volontario, A causa di un'interruzione/eccezione

Operazioni:

- 1) Salvo i registri del vecchio thread nel TCB
- 2) Muovo il TCB del vecchio thread nel lista ready o waiting
- 3) Seleziona un nuovo thread dalla lista dei Ready
- 4) Ripristina il TCB del nuovo thread nel processore
- 5) Mette il TCB del nuovo thread nella lista dei running
- 6) Restituisce il controllo al nuovo thread (IRET)

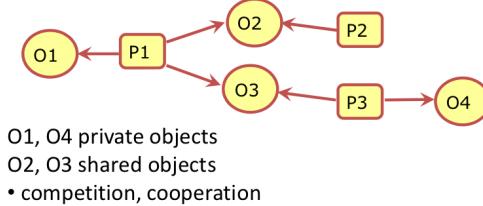
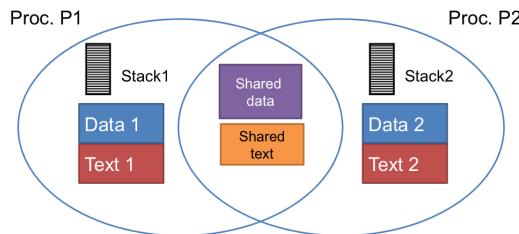
Cambio di contesto

- 1) Imposta la modalità Kernel
 - 2) Disabilita gli interrupt
 - 3) Salva PC, PS e SP nello stack del Kernel
 - 4) Carica il PC e PS dal vettore delle interruzioni
 - 5) Dopo salta al gestore delle interruzioni del Kernel
 - 6) Esegue la IRET
 - 1) Abilita gli interrupt
 - 2) Imposta la modalità utente
 - 3) Ripristina PC, SP e PS dallo stack del Kernel
 - 7) torna ad eseguire il thread che è in stato "Running" (Salta all'indirizzo)
-
- ```
graph LR; A[Salva i reg. generali nello stack del Kernel] --> B[Alla fine ripristina i reg. generali dallo stack del kernel e fa la IRET]
```

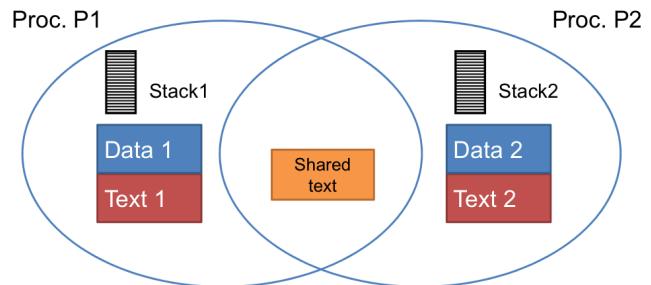
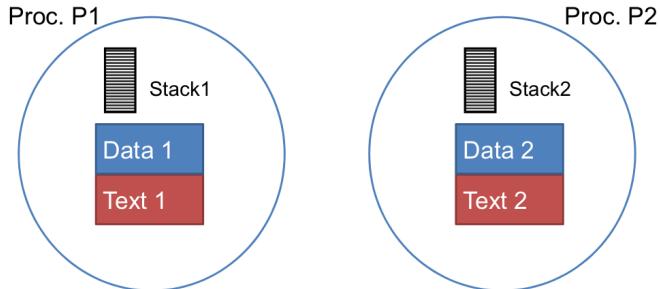
Ambiente globale: Processi e threads possono condividere i dati e hanno la mem. condivisa

Ambiente locale: Processi e threads non condividono i dati e non hanno la mem. condivisa.

## Global environment



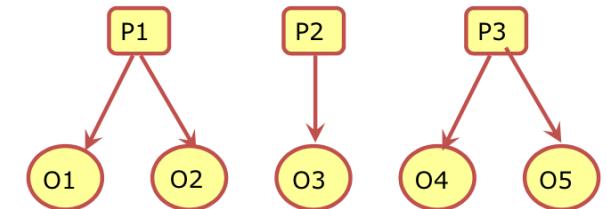
## Local environment



## Local environment



Cooperation (communication, synchronization) by means of message passing



O1-O5 are private objects  
Competition through server processes  
Cooperation through communication

## Sincronizzazione (Ambiente globale)

→ Serve per poter far cooperare due threads

Definizioni:

- 1) Race condition: l'output di un programma concorrente dipende nello ordine delle operazioni tra i threads
- 2) Mutual exclusion: procedimenti di sincronizzazione che impedisce a più task paralleli di accedere contemporaneamente a dati condivisi o a risorse soggette a race condition

3) Critical selection: Pezzi di codice che solo un thread puo' eseguire alla volta perché accede a una risorsa condivisa tra più flussi di esecuzione di un sistema concorrente.

4) lock: Guardare definizione sotto \*

✓ thread possono avere problemi di interferenza quando devono cooperare.

Potrebbero svolgere un'azione che rende inconsistente l'azione e i dati su cui opera un altro thread.

Per evitare questi casi, bisogna "sincronizzare" i thread.

\*locks: Meccanismo di sincronizzazione per limitare l'accesso ad una risorsa condivisa in un ambiente multitasking o ad un solo tipo di thread alla volta

Metodi:

- 1) lock.acquire → Aspetta fin quando il "lock" non è libero, acquisiscilo quando è libero.

- 2) lock.release → Rilascia il lock e sveglia tutti i processi che aspettano il lock

Regole per utilizzare i lock: Chiamate solo quando è ottenuto il lock

- 1) Il lock è inizialmente libero
- 2) Prima di accedere a dati condivisi, bisogna acquisire il lock  
(Inizio della procedura)
- 3) Rilasciare il lock dopo aver finito di usare dati condivisi  
(Fine della procedura)
- 4) Non accedere a una risorsa condivisa senza lock!

## Variabili condizione

- 1) wait: Rilascia automaticamente il lock e rinuncia al processo fin quando non è segnalato (signaled)
- 2) Signal: Sveglia uno che sta aspettando (se ce ne sono)
- 3) Broadcast: Sveglia tutti quelli che stanno aspettando, se ce ne sono

Esempio:

Bounded buffer  
↓  
coda FIFO

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

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

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

### Example: Bounded Buffer

Front: Posizione del primo elemento

last: Posizione ultimo elemento

Stato al lock.acquire del buffer  
e al lock.release

- nelem ≥ 0 e front ≤ last

- nelem < size e front + size ≥ last

## Regole per le variabili condizione:

- 1) Quando vengono chiamate bisogna sempre avere il lock (Dopo lock.acquire() e prima di lock.release())
  - Forniscono la sincronizzazione per lo stato condiviso
  - Sempre avere il lock quando si accede a uno stato condiviso
- 2) Sono senza memoria :
  - Se faccio lo signal quando nessuno aspetta, non succede nulla
  - Se faccio lo wait prima di fare lo signal uno che sta aspettando viene svegliato
- 3) La wait rilascia in modo atomico lo lock → 1 solo ciclo di clock

4) Quando un thread viene risvegliato dallo stato di wait, viene aggiunto alla lista dei pronti, quindi potrebbe non andare in esecuzione immediatamente perché qualcun altro potrebbe acquistare il lock

5) Il metodo wait deve essere in loop

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

## Sincronizzazione strutturata

1) Identificare oggetti o strutture dati che possono essere acceduti da thread multipli concorrenti.

2) Aggiungere locks a moduli e oggetti:

- Prendere il lock all'inizio di ogni metodo/procedura
- Rilasciare il lock alla fine

3) Il wait va messo in loop e non annuncia che un processo appena svegliato inizi subito

4) Se bisogna svegliare qualcuno: Signal or Broadcast

5) Lasciare le variabili di stato condizionare in uno stato corretto

## Semantica di sincronizzazione:

1) Mesa:

- Signal mettono un waiter nella lista dei pronti

$\downarrow$   
Nessuna  
annuncio

- Quelli che lanciano il signal tengono lock e proveranno

2) Hoane:

- Signal danno processo e lock a un waiter

$\downarrow$   
FIFO

- Quando un waiter finisce, processo/lock vengono restituiti a colui che lanciò il signal
- Signal intrecciati sono possibili

Produttore/consumatore  
con semantica  
di Hoane

FIFO Bounded Buffer  
(Hoare semantics)

```
get() {
 lock.acquire();
 while (nelem == 0)
 empty.wait(lock);
 item = buff[front];
 front = (front++) % size;
 nelem -= 1;
 full.signal(lock);
 lock.release();
 return item;
}
```

put(item) {
 lock.acquire();
 while (nelem == size)
 full.wait(lock);
 buff[last] = item;
 last = (last++) % size;
 nelem += 1;
 empty.signal(lock);
 // CAREFUL: someone else ran
 // nelem++ here
}

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

Produttore/consumatore  
con semantica  
di Mesa

FIFO Bounded Buffer  
(Mesa semantics, put() is similar)

```
get() {
 lock.acquire();
 if (nelem == 0 || !nextGet.empty())
 self = new Condition();
 nelem -= 1;
 nextGet.append(self);
 while (nelem == 0)
 self.wait(lock);
 nextGet.remove(self);
 delete self;
}

item = buff[front];
front = (front++) % size;
nelem -= 1;
if (!nextPut.empty())
 nextPut.first->signal(lock);
lock.release();
return item;
```

Initially: nelem = front = last = 0; size is buffer capacity  
nextGet, nextPut are queues of Condition Variables

## Implementazione Van. Condizioni MCSA

### wait:

```

varcond.wait(L)
{lock.release(L); //rilascio della lock: uscita dalla sezione critica
SVC(insert(run,varcond);scheduler(ready)); // questa SVC inserisce il thread che
ha eseguito la wait (che è run), nella coda della variabile condizione, ed attiva lo
scheduler per scegliere un nuovo thread da eseguire scelto tra quelli della coda ready,
quindi un nuovo run, un nuovo thread da mettere in esecuzione
lock.acquire(L); //acquisizione della lock: il thread che ha fatto la wait, quando
viene riattivato, cioè quando è di nuovo messo in esecuzione, run, deve rientrare nella
sezione critica
}

```

### broadcast:

```

varcond.broadcast()
{SVC(broadcast(varcond),ready); }// questa SVC può essere implementata nel
modo seguente:
x=extract(varcond);
while (x!=NULL)
{insert(x,ready);
x=extract(varcond);
}

```

oppure semplicemente appendendo tutta la coda varcond in fondo alla coda ready, a seconda dello scheduler utilizzato

## Implementazione Van. Condizioni Hoane

### wait:

```

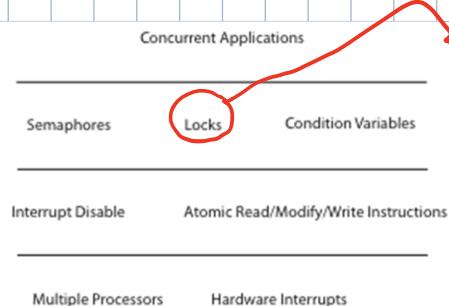
varcond.wait(L)
{lock.release(L); //rilascio della lock: uscita dalla sezione critica
SVC(insert(run,varcond);scheduler(ready)); // questa SVC inserisce il thread che
ha eseguito la wait (che è run), nella coda della variabile condizione, ed attiva lo
scheduler per scegliere un nuovo thread da eseguire scelto tra quelli della coda ready,
quindi un nuovo run, un nuovo thread da mettere in esecuzione
}

```

### Broadcast:

Nou e' prevista

## Implementare la sincronizzazione:



### Signal:

```

varcond.signal()
{SVC(insert(extract(varcond),ready); }// questa SVC può essere implementata nel
modo seguente:
x=extract(varcond);
if x!=NULL
{insert(x,ready)}

```

**Signal:** Para la lock L al maggiore  
e estare con politica FIFO  
(la van. cond. e' gestita FIFO)

### Lock Implementation, Uniprocessor

```

LockAcquire() {
 disableInterrupts ();
 if (value == BUSY) {
 waiting.add(current TCB);
 suspend(); *
 } else {
 value = BUSY;
 }
 enableInterrupts ();
}
* Invokes the scheduler,
context switch & enable
interrupts

```

```

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

```

## Nel caso dei multiprocessori:

### Read-modify-write instructions

- Atomically read a value from memory, operate on it, and then write it back to memory
- Intervening instructions prevented in hardware

**Spinlocks:** Il processore aspetta in loop che il lock diventi libero:

- Viene attuato che il lock viene tenuto per poco tempo
- Usato per proteggere la ready list per implementare i lock

```
SpinlockAcquire() {
 while (testAndSet(&spinLockValue) == BUSY)
 ;
}
```

```
SpinlockRelease() {
 spinLockValue = FREE;
}
```

### Lock Implementation, Multiprocessor

```
LockAcquire(){
 spinLock.Acquire();
 if (value == BUSY){
 waiting.add(current TCB);
 sched.suspend(&spinLock); *
 } else {
 value = BUSY;
 spinLock.Release(); *
 }
} * scheduler: marks thread as waiting; release spinlock; schedules next thread;
```

```
LockRelease() {
 spinLock.Acquire();
 if (!waiting.Empty()){
 thread = waiting.Remove();
 sched.makeReady(thread, &spinLock); *
 } else {
 value = FREE;
 }
 spinLock.Release(); *
} * scheduler: marks thread as ready, put it in the ready list.
```

**Semafori** → Struttura dati:  $\begin{cases} \text{int value} \geq 0 \\ \text{coda} \end{cases}$

Metodi: 1) P(sem) → Se value > 0 ⇒ risorsa condiziona libera ⇒ Entra e value--  
Se value == 0 ⇒ si mette in coda

2) V(sem) → Incrementa atomicamente il valore di uno ( se nessuno waiter è presente )  
sempre sveglia un waiter. ( se faccio 2 v() a filo senza waiter ⇒ value = 2 )

**Attenzione:** Il valore è 0 o è > 0 → 0 se il semaforo è nono  
> 0 se " è vero  
e non ci sono waiter

} Se il valore è uno  
e eseguo due volte  
P(), il risultato è:  
Value=0 e un waiter

### Implementazione P e V

se multi:  
processe

```
P(sem){
 spinLock.Acquire();
 disableInterrupts();
 if (sem.value == 0){
 waiting.add(current TCB);
 suspend(&spinLock); *
 } else {
 sem.value--;
 spinLock.Release();
 enableInterrupts();
 }
} * suspends, invokes scheduler,
context switch & enable
interrupts
```

sempre single  
process

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

se multi:  
processe

sempre single  
process

## Produttore / consumatore con semafori:

### Semaphore Bounded Buffer

```

é importante
l'ordine
delle P()
 get() {
 empty.P();
 mutex.P();
 item = buf[front];
 front = (front+1) % size;
 mutex.V();
 full.V();
 return item;
 }
Initially: front = last = 0; size is buffer capacity
empty/full are semaphores (initialized to 0 and size)
Mutex is a semaphore initialized to 1
 put(item) {
 full.P();
 mutex.P();
 buf[last] = item;
 last = (last +1) % size;
 mutex.V();
 empty.V();
 }

```

## Implementazione Variabili condizioni

```

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
 }
}

```

↓ P e V con var.  
condizione

```

P(sem) {
 disabilita interruzioni;
 if (sem.val==0) {
 self = new Condition;
 Append(self,sem.coda); // appende in fondo
 yield; // ipotesi: la yield è una SVC che verrà
 sentita quando le interruzioni verranno
 riabilitate
 }
 else sem.val--;
 abilita interruzioni;
}

V(sem) {
 disabilita interruzioni;
 if (!empty(sem.coda)) {
 first(sem.coda) → codapronti;
 }
 else sem.val++;
 abilita interruzioni;
}

```

## Sincronizzazione di oggetti multipli

→ Problema: deadlock = Due o più processi che si bloccano a vicenda.  
Attesa circolare delle risorse

Risposta: Ogni cosa pensava che è necessaria un thread per fare un lavoro

→ Può essere:  
- Previsibilmente: Può essere rimossa dall'OS a un thread senza perdere il lavoro compiuto

- Non Previsibilmente: deve essere lasciata quando il thread se ne va

Stavolta: thread aspetta un tempo indefinito.

→ Deadlock → sfavillante ma non il viceversa.

## Esempi di lock:

1) Thread A

```
lock1.acquire();
lock2.acquire();
lock2.release();
lock1.release();
```

Thread B

```
lock2.acquire();
lock1.acquire();
lock1.release();
lock2.release();
```

2)

Thread A

```
lock1.acquire();
...
lock2.acquire();
while (need to wait)
 condition.wait(lock2);
lock2.release();
...
lock1.release();
```

Thread B

```
lock1.acquire();
...
lock2.acquire();
...
condition.signal(lock2);
lock2.release();
...
lock1.release();
```

3)

Thread A

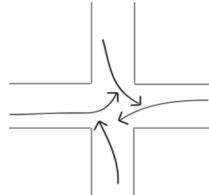
```
buffer1.put(data);
buffer1.put(data);
```

Thread B

```
buffer2.put(data);
buffer2.put(data);
```

Dining Lawyers

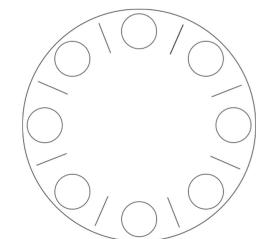
4) Yet another Example



5)



6)



Each lawyer needs two chopsticks to eat.  
Each grabs chopstick on the right first.

## Condizioni che causano il deadlock:

Sono tutte condizioni necessarie  $\rightarrow$  Il sistema va in Deadlock se si verificano tutte e 4

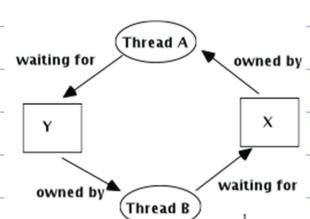
1) Accesso limitato alle risorse e mutua esclusione

2) Risorse non prelasciabili (esempio: lock)

3) Multiple richieste indipendenti (Richiedo nuove risorse, e tengo quelle già ottenute)

4) Catenza di richieste circolari

### Circular Waiting



## Metodi per gestire il deadlock

1) Detect and fix : Algoritmo: Scavonano il grafo, rileva i cicli e risolvili

Risolvere i cicli

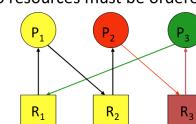
1) Uccidere il thread  $\rightarrow$  Necesita criteri per la selezione dei thread con minimo impatto

2) Azioni di rollback  $\rightarrow$  ripristinare lo stato precedente se avviene un errore  
(costoso perché bisogna salvare lo stato ogni tot.)

2) Prevenzione del deadlock: Eliminare una delle 4 condizioni del deadlock

- Lock ordering (risolve l'altera circolare): Ottenere le risorse sempre nello stesso ordine

- Requests to resources must be ordered



R3: Resource with maximum index  
P<sub>3</sub> violates the constraint of sorted requests  
P<sub>3</sub> causes circular waiting

(es: lock)

$\Rightarrow$  Avere le richieste tutte da Ax verso dx

- Disegnare sistemi che rilasciano le risorse e riprovano se devono aspettare ("elimina le wait while holding")

- Avere risorse infinite  $\rightarrow$  spool gestisce le richieste  $\rightarrow$  Virtualizzazione infinita risorse
  - Acquisire tutte le risorse necessarie all'inizio (elimina i.e. "wait while holding")  
↳ Problema: O prende tutte le risorse o nessuna, in più potrebbe prendere risorse che serviscono ad un altro processo.

Molti piccola- delle risone : risone classificate in accordo con il loro tipo

Moltiplicata  $M$ : Numero di rigore di un certo tipo

**Disponibilità D:** Numero di risorse di un certo tipo ancora disponibili.

## Metodo di nichiesta:

- Richiesta singola
  - Richieste multiple:
    - se  $K \leq D$  allora tutte le risorse di  $K$  vengono assegnate
    - se  $K > D$  allora nessuna risorsa viene assegnata e il processo aspetta.

3) Prevenzione dinamica ( Algoritmo del banchiere ) :

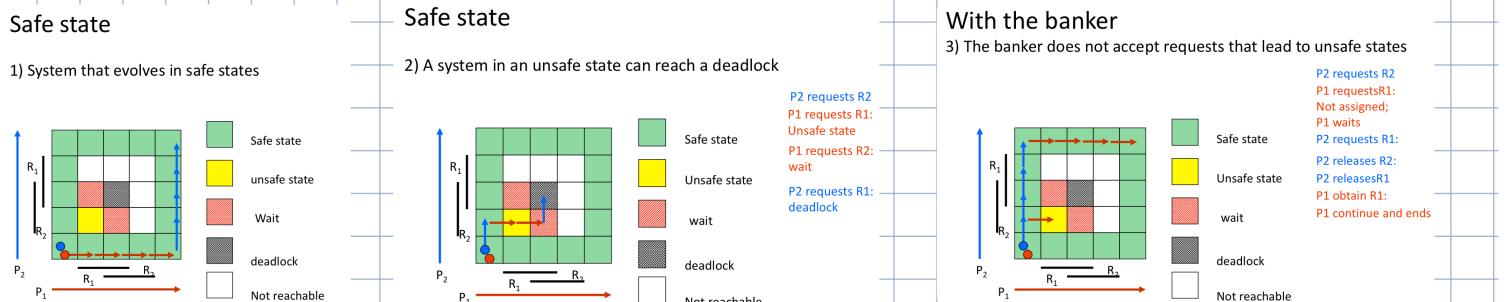
- Stabilisce il massimo delle risorse necessarie, in anticipo.
  - Alloca le risorse dinamicamente quando sono richieste e aspetta a dare le risorse se produrrebbero un deadlock
  - Le richieste possono essere soddisfatte se qualche sequenza di ordinamento dei thread non genera un deadlock.

Possibili stati:

1) Safe state: per ogni possibile sequenza delle richieste future è possibile soddisfare tutte le richieste

2) Unsafe state: Alcune sequenze di richieste possono generare un deadlock

3) Doomed state: tutte le possibili computazioni generano un deadlock.



# Algoritmo del banchiere:

## Banker's algorithm for resources of the same type

- First each process declares the number of resources it needs
- At a request of process P the banker checks whether the resource assignment keeps a safe state. To this purpose:
  - Considers the state S reached if request is granted
  - For each process computes the residual requirement R (the number of resources it still needs)
  - Sorts the processes for an increasing value of R
  - Executes the algorithm (see next slide)
  - If at all processes are marked at the end of algorithm the state is safe
- If the state S is safe then the request can be granted
- Otherwise the process P waits until there are enough resources to let it proceed

## Esempio:

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

Stato raggiunto dal sistema al tempo  $t$ :

|                                    |   | MOLTEPLICITA' $\rightarrow$ |    |    |    |
|------------------------------------|---|-----------------------------|----|----|----|
| ASSEGNAZIONE ATTUALE $\rightarrow$ |   | R1                          | R2 | R3 | R4 |
| P1                                 | 2 | 1                           | 1  | 1  |    |
| P2                                 | 2 | 0                           | 1  | 2  |    |
| P3                                 | 0 | 1                           | 0  | 0  |    |
| P4                                 | 0 | 2                           | 2  | 2  |    |

|                  |   | DISPONIBILITA' ATTUALE |    |    |    |
|------------------|---|------------------------|----|----|----|
| ESIGENZA RESIDUA |   | R1                     | R2 | R3 | R4 |
| P1               | 4 | 5                      | 5  | 5  |    |
| P2               | - | -                      | -  | -  |    |
| P3               | 0 | 1                      | 0  | 0  |    |
| P4               | 0 | 2                      | 2  | 2  |    |

Verifica dello stato sicuro:

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

|                                    |   | MOLTEPLICITA' $\rightarrow$ |    |    |    |
|------------------------------------|---|-----------------------------|----|----|----|
| ASSEGNAZIONE ATTUALE $\rightarrow$ |   | R1                          | R2 | R3 | R4 |
| P1                                 | 2 | 2                           | 1  | 1  |    |
| P2                                 | - | -                           | -  | -  |    |
| P3                                 | 0 | 1                           | 0  | 0  |    |
| P4                                 | 0 | 2                           | 2  | 2  |    |

|                  |   | DISPONIBILITA' ATTUALE |    |    |    |
|------------------|---|------------------------|----|----|----|
| ESIGENZA RESIDUA |   | R1                     | R2 | R3 | R4 |
| P1               | 4 | 5                      | 5  | 5  |    |
| P2               | - | -                      | -  | -  |    |
| P3               | 0 | 2                      | 0  | 0  |    |
| P4               | 0 | 0                      | 3  | 0  |    |

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

Verifica dello stato sicuro:

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

|                                    |   | MOLTEPLICITA' $\rightarrow$ |    |    |    |
|------------------------------------|---|-----------------------------|----|----|----|
| ASSEGNAZIONE ATTUALE $\rightarrow$ |   | R1                          | R2 | R3 | R4 |
| P1                                 | 2 | 1                           | 1  | 1  |    |
| P2                                 | - | -                           | -  | -  |    |
| P3                                 | - | -                           | -  | -  |    |
| P4                                 | 0 | 2                           | 2  | 2  |    |

|                  |   | DISPONIBILITA' ATTUALE |    |    |    |
|------------------|---|------------------------|----|----|----|
| ESIGENZA RESIDUA |   | R1                     | R2 | R3 | R4 |
| P1               | 4 | 5                      | 5  | 5  |    |
| P2               | - | -                      | -  | -  |    |
| P3               | - | -                      | -  | -  |    |
| P4               | 0 | 0                      | 3  | 0  |    |

Verifica dello stato sicuro:

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

|                                    |   | MOLTEPLICITA' $\rightarrow$ |    |    |    |
|------------------------------------|---|-----------------------------|----|----|----|
| ASSEGNAZIONE ATTUALE $\rightarrow$ |   | R1                          | R2 | R3 | R4 |
| P1                                 | - | -                           | -  | -  |    |
| P2                                 | - | -                           | -  | -  |    |
| P3                                 | - | -                           | -  | -  |    |
| P4                                 | 0 | 2                           | 2  | 2  |    |

|                  |   | DISPONIBILITA' ATTUALE |    |    |    |
|------------------|---|------------------------|----|----|----|
| ESIGENZA RESIDUA |   | R1                     | R2 | R3 | R4 |
| P1               | 4 | 5                      | 5  | 5  |    |
| P2               | - | -                      | -  | -  |    |
| P3               | - | -                      | -  | -  |    |
| P4               | 0 | 0                      | 3  | 0  |    |

## Banker's algorithm for multiple resources of multiple types

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; ( $E_{jk} \leq D_k$  if  $E_{jk} \leq D_k$  for each k)

Initially each process  $P_j$  is not marked

```
while (exists a non-marked process Pj that satisfies Ej <= D) {
 mark Pj;
 D = D + Aj;
}
```

success: the initial state is safe

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

Verifica dello stato sicuro:

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

|                                    |   | MOLTEPLICITA' $\rightarrow$ |    |    |    |
|------------------------------------|---|-----------------------------|----|----|----|
| ASSEGNAZIONE ATTUALE $\rightarrow$ |   | R1                          | R2 | R3 | R4 |
| P1                                 | 2 | 2                           | 1  | 1  |    |
| P2                                 | - | -                           | -  | -  |    |
| P3                                 | 0 | 1                           | 0  | 0  |    |
| P4                                 | 0 | 2                           | 2  | 2  |    |

|                  |   | DISPONIBILITA' ATTUALE |    |    |    |
|------------------|---|------------------------|----|----|----|
| ESIGENZA RESIDUA |   | R1                     | R2 | R3 | R4 |
| P1               | 4 | 5                      | 5  | 5  |    |
| P2               | - | -                      | -  | -  |    |
| P3               | 0 | 0                      | 0  | 2  |    |
| P4               | 0 | 0                      | 3  | 0  |    |

Verifica dello stato sicuro:

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

|                                    |   | MOLTEPLICITA' $\rightarrow$ |    |    |    |
|------------------------------------|---|-----------------------------|----|----|----|
| ASSEGNAZIONE ATTUALE $\rightarrow$ |   | R1                          | R2 | R3 | R4 |
| P1                                 | - | -                           | -  | -  |    |
| P2                                 | - | -                           | -  | -  |    |
| P3                                 | - | -                           | -  | -  |    |
| P4                                 | 0 | 2                           | 2  | 2  |    |

|                  |   | DISPONIBILITA' ATTUALE |    |    |    |
|------------------|---|------------------------|----|----|----|
| ESIGENZA RESIDUA |   | R1                     | R2 | R3 | R4 |
| P1               | 4 | 3                      | 3  | 3  |    |
| P2               | - | -                      | -  | -  |    |
| P3               | - | -                      | -  | -  |    |
| P4               | 0 | 0                      | 3  | 0  |    |

ESIGENZA RESIDUA

STATO SICURO !

**Attenzione:** Se arriviamo a un punto dove il banchiere non può completare nemmeno un processo con le risorse rimaste, abbiamo uno stato non sicuro!

**Scheduling**: Decide quale thread mandare in esecuzione, quando ci sono thread multipli pronti a essere eseguiti

**Definizioni:**

- Task / job: Richieste utenti
- Latenza: Quanto tempo richiede un task per essere completato
- Throughput: Quanti task possono essere finiti in una unità di tempo
- Overhead: Quanto lavoro extra deve fare lo scheduler per calcolare il nuovo scheduling
- Fairness: Quanto è equa la performance ricevuta dai vari utenti
- Predictabilità: Quanto è costante la performance nel tempo.

**Altre definizioni:**

- Workload:insieme di task che il sistema deve fare
- Scheduler prelasciabile: Se può togliere le risorse a un task in running
- Work-covering: Una risorsa che è usata ogni volta che un task la vuole utilizzare

**Algoritmi di scheduling:**

- 1) Prende un insieme di thread/processi in input
- 2) Decide quale task eseguire prima
- 3) Restituisce in output una metrica di performance (throughput, latenza)
- 4) Solo gli scheduler prelasciabili e workcovering devono essere considerati

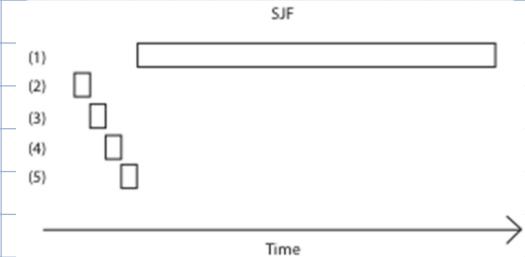
**Algoritmi di scheduling:**

- 1) FIFO (First in First out) → Schedula i task nell'ordine di arrivo



Ha un basso tempo medio di risposta se i task sono di grandezza variabili, dunque è ottimo.  
Mi limita l'overhead

2) Shortest Job First  $\rightarrow$  Schedula i task in ordine di lunghezza, dal più piccolo al più grande



S J F  
minimizza il tempo medio di risposta (latenza media)

4 jobs A,B,C,D with execution time:  $a, b, c, d$

Scheduling sequence:  $a \rightarrow b \rightarrow c \rightarrow d$

- turnaround(A) --  $a$
- turnaround(B) --  $a + b$
- turnaround(C) --  $a + b + c$
- turnaround(D) --  $a + b + c + d$

Total turnaround:  $4a + 3b + 2c + d$

Minimized iff  $a, b, c, d$  are sorted in increasing order

Problema: Sapere il tempo rimanente di ogni task.

Difficile da sapere

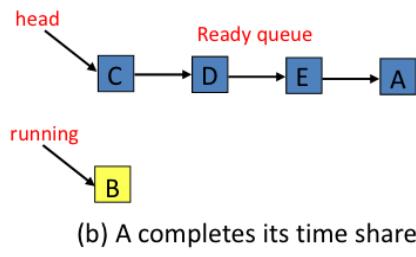
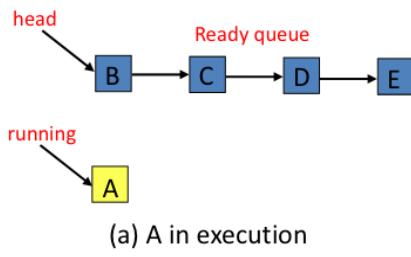
$\hookrightarrow$  Poco ha il problema dello Starvation cioè un processo lungo potrebbe aspettare troppo tempo. (Ferrina Fairness)

Pensiamo in termini di Variance response time.

3) Round Robin  $\rightarrow$  Ogni task tiene una risorsa solo per una determinata quantità di tempo (quanto di tempo)  $\Rightarrow$  Timer

$\downarrow$   
Ottima Fairness

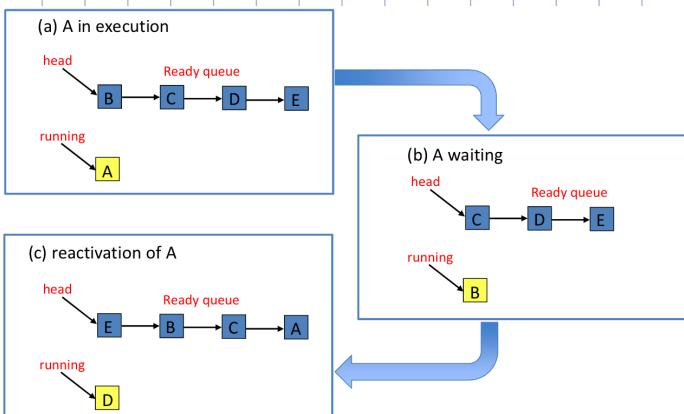
Se un task ha finito di eseguire, torna nella lista dei task ready.



tempo di risposta, cioè tempo che un processo deve aspettare per entrare meno in esecuzione è limitato superiormente dal numero di processi

# ready process \* time share

Gestione dei thread in attesa nel Round Robin:



## OSSERVAZIONI SU ROUND-ROBIN

- la quantità di tempo che un thread può usare una risorsa è dato dal timer
  - un timer interrupt causa l'attivazione dello scheduler
  - lo scheduler fa ripartire il timer dopo aver cambiato il thread in esecuzione
- lo scheduler continua ad andare anche in caso di una sospensione del processo in running, riassegna la CPU e riavvia il timer

## Algoritmi di Scheduling in caso di Mixed Workload



### 1) Multi-Level feedback Queue (MLFQ)

- 1) Preferenza a short jobs
- 2) Preferenza a I/O bounded process
- 3) Separa i processi in base a cosa necessitano del processore

Algoritmo che non eccelle in nulla ma cerca di far bene tutto

- CPU Bound  $\rightarrow$  lungo utilizzo di CPU con scarso I/O
- I/O Bound  $\rightarrow$  corto utilizzo di CPU con I/O frequente

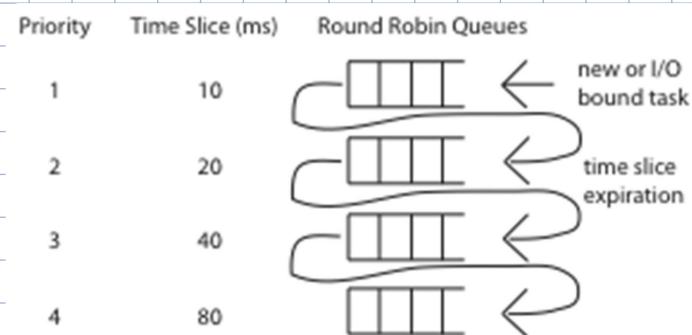
- Imposta delle code Round-Robin ognuna con diversa priorità

- le code con priorità elevata hanno piccole fetta di tempo, mentre le code con priorità bassa hanno lunghe fetta di tempo.

- lo scheduler prende il primo thread dalla coda con priorità più alta

- se task inizia nella coda di priorità più alta, se la fetta di tempo scade, il task

scende di un livello di priorità



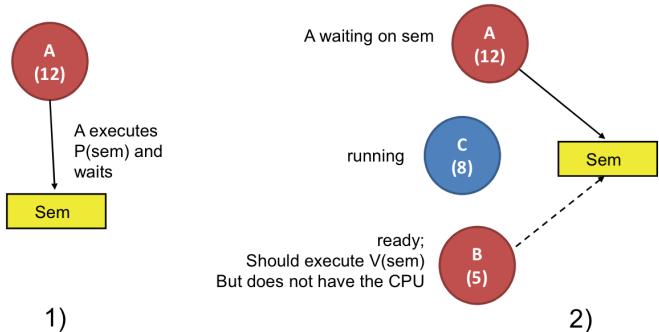
## Problemi con l'MLFQ

- 1) Starvation  $\rightarrow$  Avviene se tutte le code superiori sono sempre premi di I/O Bounded Process
- 2) Necesita di politiche per alzare la priorità di:
  - I/O Bounded processes
  - CPU-Bound Processes that are Starving
- 3) MFQ sono combinate di solito con priorità dinamiche

## Scheduling in windows:

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

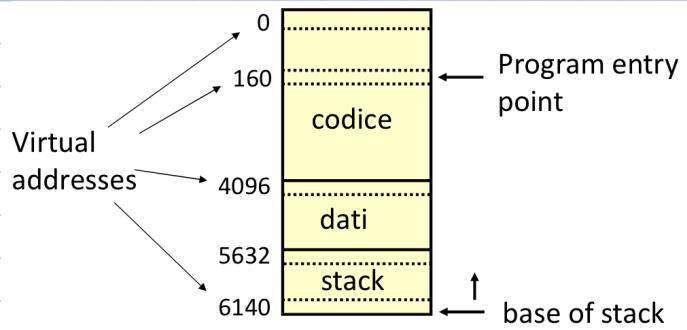
## Inversion of priority



## Seconda Parte del programma

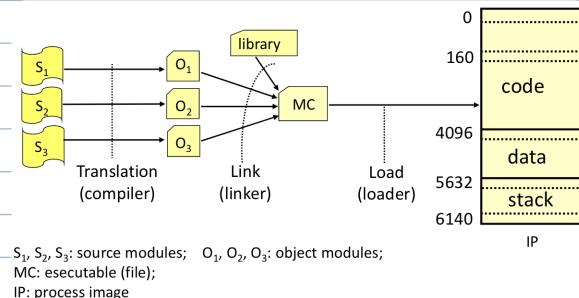
### Traduzione degli indirizzi:

Processo in memoria:



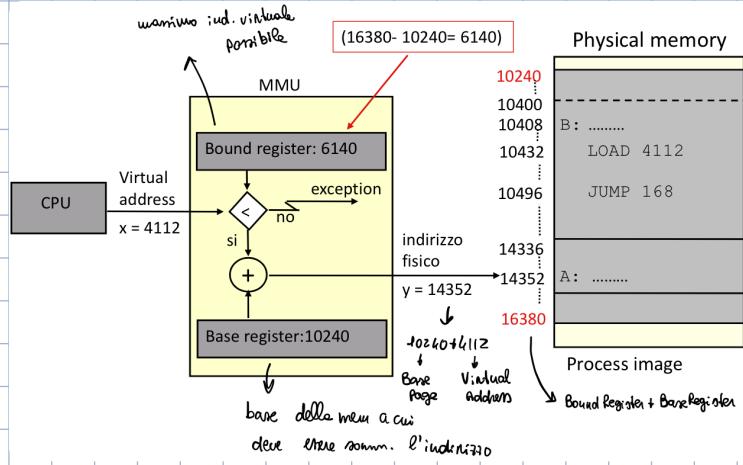
tutti i programmi hanno l'indirizzo logico che parte da 0 per comodità

Dalla compilazione all'esecuzione:



### Metodi di traduzione:

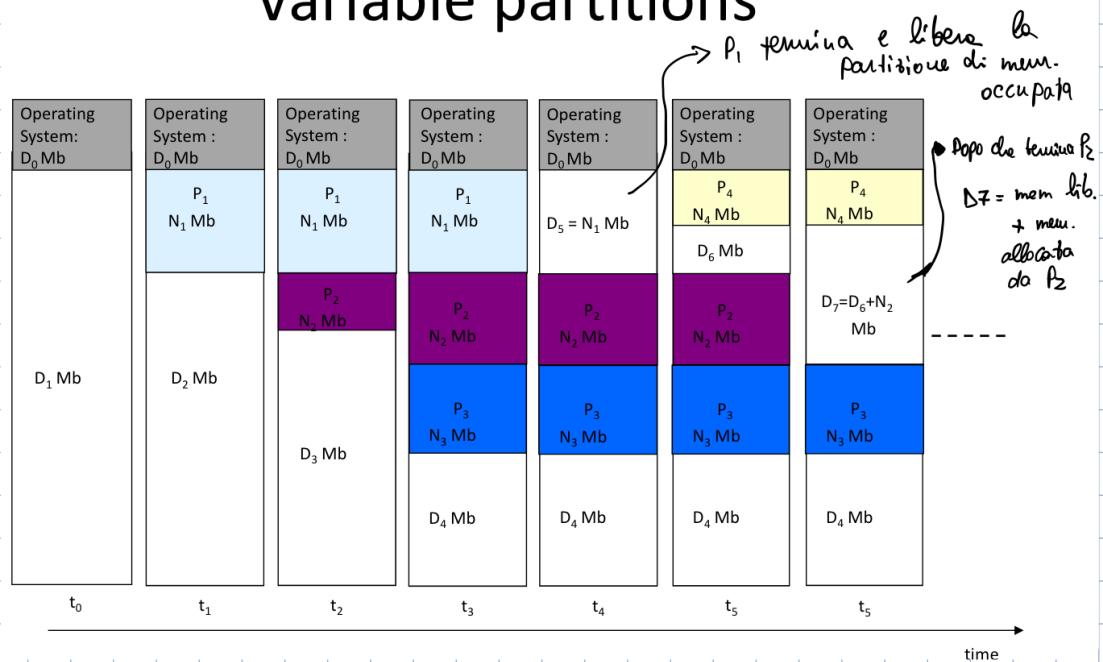
#### 1) Virtual base and Bounds



2) Partizioni variabili: Ogni volta che un processo va in esecuzione, alloca una partizione di mem. per quel processo.

**Problema:** Rischio di trovarmi ad un punto dove ho tante zone libere ma piccole, quindi il processo non riesce ad allocare memoria.

## Variable partitions

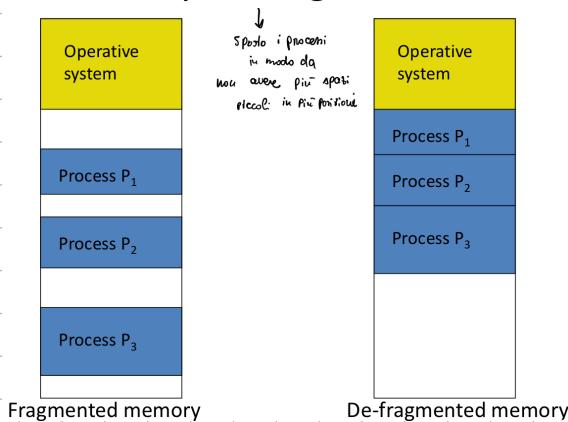


Modi di allocare le partizioni nuove :

- First-Fit : Prima partizione libera di mem. che puo' contenere la nuova partizione
- Best-Fit : Prende la piu' piccola partizione libera che puo' contenere la nuova partizione

## Memory de-fragmentation

3) Fragmentazione :



## a) Segmentazione:

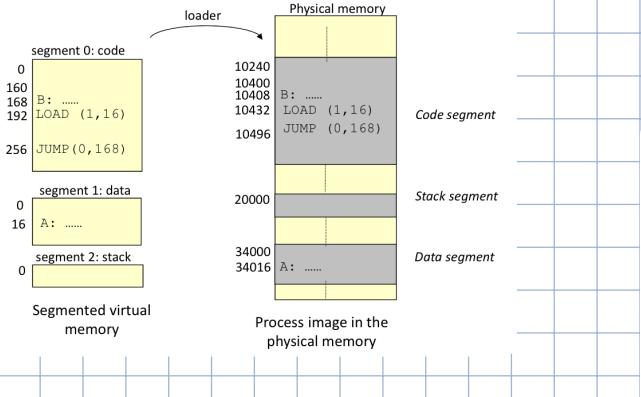
**Segmento:** è una regione contigua di memoria

↳ possono essere allocati ovunque nella memoria

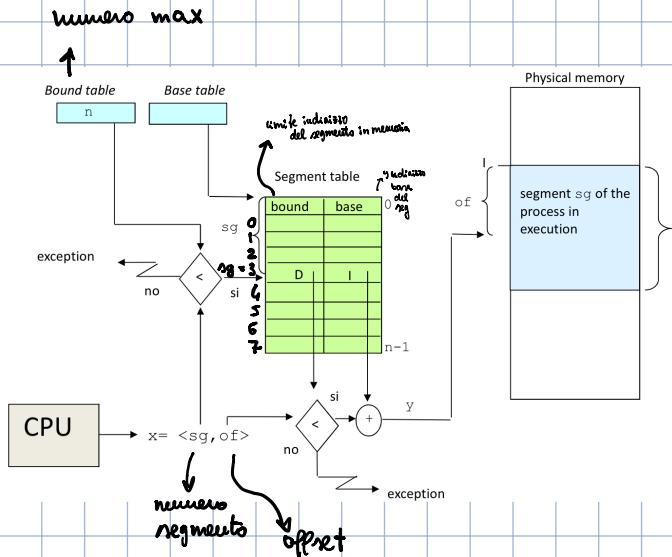
↳ i processi possono condividere i segmenti

**IMPORTANTE**

**OSS:** Ogni processo ha una segment table dove ogni entry nella tabella contiene base e bound di un segmento



**traduzione degli indirizzi:**



Quando avviene una Fork()  $\Rightarrow$  copy or write

- Copia la segment table nel processo figlio
- Imposta i segmenti del padre e figlio in read-only
- Esegua il processo figlio; return al genitore
- Se il figlio/genitore scrive in un segmento: viene catturata dal kernel e viene fatta una copia del segmento.  
Poi riparte.

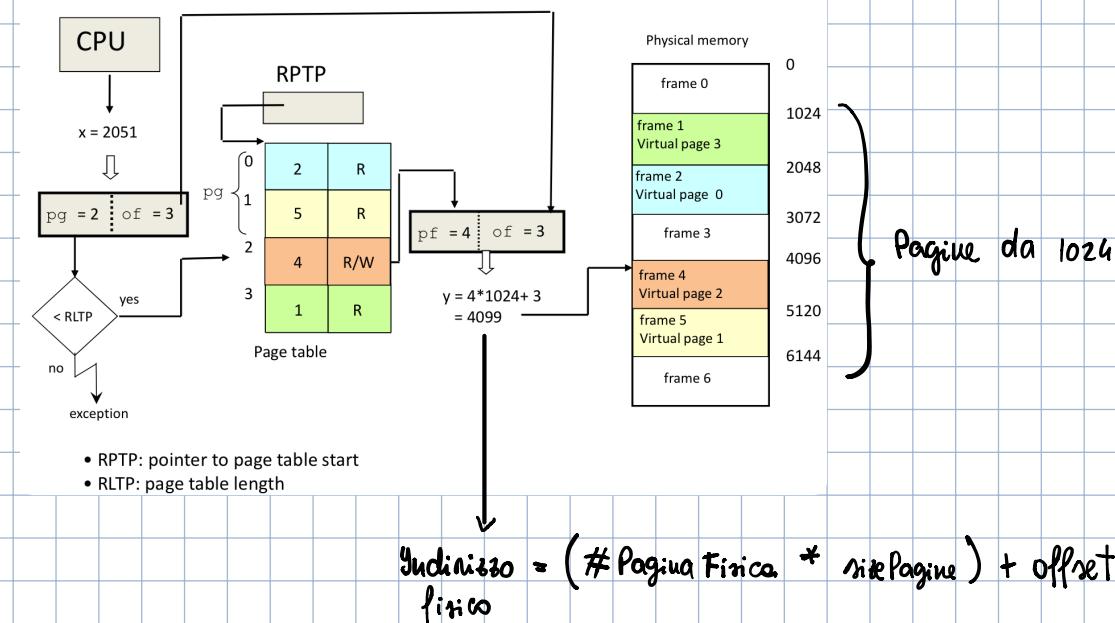
## Zero-on-Reference

- How much physical memory do we need to allocate for the stack or heap?
  - Zero bytes!
- When program touches the heap
  - Segmentation fault into OS kernel
  - Kernel allocates some memory
    - How much?
  - Zeros the memory  $\rightarrow$  Prima di allocare la mem, viene riportata tutta a 0
    - avoids accidentally leaking information!
  - Restart process

5) Paged translation: la memoria viene divisa in pagine di ugual  
grandezza.

Per cercare una pagina si usa il bitmap allocation: 1001100110000 dove  
ogni bit rappresenta una pagina fisica.

## Paged Translation



### Osservazioni:

- 1) Ogni processo ha una page table di un processo
- 2) Un cambio di contesto <sup>v</sup> bisogna salvare / ripristinare: Puntatore alla tabella delle pagine e alla sua grandezza
- 3) le page tables sono nella memoria principale
- 4) Se le pagine sono più piccole  $\Rightarrow$  richiede più pagine  $\Rightarrow$  page table che occupa più spazio
- 5) Se le pagine sono grandi  $\Rightarrow$  frammentazione interna
- 6) Paging and Copy on Write

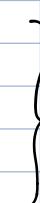
- Can we share memory between processes?
  - Set entries in both page tables to point to the same page frames
  - Need core map of page frames to track which processes are pointing to which page frames
- UNIX fork with copy on write at page granularity
  - Copy page table entries to new process
  - Mark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

Quando eseguo un programma:

- 1) Imposto tutte le entry della page table su invalid
- 2) Quando una pagina viene riferita per la prima volta:
  - Chiama il S.O
  - Il S.O "porta" la pagina
  - Riprende l'esecuzione
- 3) Le pagine rimanenti vengono trasferite durante il prog. in esecuzione

traduzione multi - livello (multi - level translation)  $\Rightarrow$

- Paged Segmentation
- Multi - lvl page tables
- Multi - lvl paged Segmentation

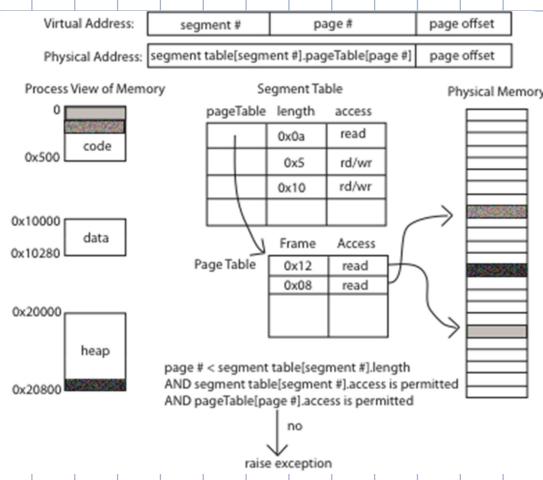


Imposta la grandezza della pagina  
come lowest level unit

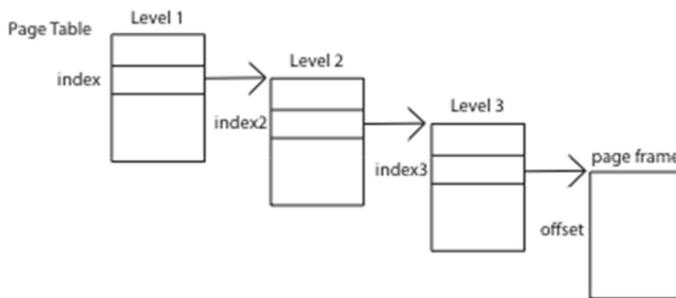
- Pros:
  - Allocate/fill only as many page tables as used
  - Simple memory allocation
  - Share at segment or page level
- Cons:
  - Two or more lookups per memory reference

## 1) Paged Segmentation

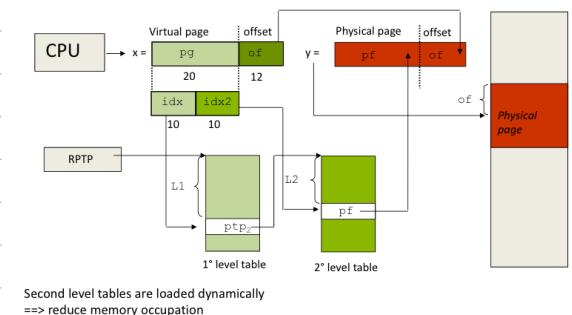
- La memoria del processo è segmentata
- le entry della segment table contengono:
  - 1) Puntatore alla page table
  - 2) la lunghezza della page table
  - 3) permessi d'accesso
- le entry della page Table:
  - 1) Page Frame
  - 2) permessi d'accesso



Muti: eve paging  $\Rightarrow$  Page table a più livelli



Two level paging



Couplando tra tabelle delle pag. a 1 o 2 lvl

IPOTESI:

- Indirizzi logici di 32 bit; pagine logiche e fisiche di 4 kByte.  
==> lunghezza del campo offset : 12 bit; indice di pagina codificato con 20 bit
  - Descrittori di pagina (elementi della tabella delle pagine) codificati con 4 byte, di cui:
    - 3 byte (24 bit) per la codifica dell'indice di blocco;
    - 1 byte riservato agli indicatori
- TABELLA DELLE PAGINE A 1 LIVELLO:
- Numero di elementi della tabella delle pagine:  $2^{20}$
  - Spazio occupato dalla tabella delle pagine:  $2^{20} * 4 = 2^{22}$  byte = 4 Mbyte
  - Massima dimensione della memoria fisica:  $2^{24}$  blocchi  
==>  $2^{24} * 2^{12} = 2^{36}$  byte = 64 Gbyte

TABELLA DELLE PAGINE A 2 LIVELLI:

- Ipotesi:  $2^{10}$  tabelle delle pagine di secondo livello;  
==> La tabella di primo livello ha  $2^{10}$  elementi  
==> ripartizione dell'indirizzo logico:  
  - 12 bit per offset;
  - 10 bit per indirizzare la tabella di primo livello
  - 10 bit per indirizzare la tabella di secondo livello selezionata;
- ==> ogni elemento di tabella di primo livello corrisponde a una tabella di secondo livello
- 3 byte: indice di blocco nel quale risiede la tabella di secondo livello (se presente)
  - 1 byte: indicatori (tra cui indicatore di presenza).
- ==> ogni elemento di tabella di secondo livello corrisponde a una pagina
- 3 byte: indice di blocco nel quale risiede la pagina (se presente)
  - 1 byte: indicatori (tra cui indicatore di presenza).
- lunghezza di ogni tabella di primo o secondo livello:  $2^{10}$  elementi ==>  $2^{10} * 4$  = 4 Kbyte
  - massima dimensione della memoria fisica :  $2^{24}$  blocchi ==>  $2^{24} * 2^{12} = 2^{36}$  byte = 64 Gbyte.

Muti: eve Paged Segmentation

Global descriptor table (segment table):

- Puntatore alla page table per ogni segmento
- lunghezza segmento
- permette d'accesso del segmento
- Context switch: Cambia il registro GTDR

Multilevel page table

- 4Kb pages (ogni livello della page table fits in una pagina)
- 32 bit: 2 lvl page table (per segmento)
- 64 bit: 4 lvl page table (per segmento)

Traduzione degli indirizzi efficiente

Uso della TLB (translation lookaside buffer)  $\Rightarrow$  Sta nella MMU

- TLB:
- 1) Cache che contiene le recenti traduzioni degli indirizzi fisici-logici
  - 2) Se abbiamo un cache-hit, si usa la traduzione  
Se abbiamo un cache-miss, si "commuta" la multi-level page table

Costo delle traduzioni: Costo lookup TLB +  $(\% \text{ TLB miss})^*$  costo lookup page table

TLB entry possono avere:

- Pagina

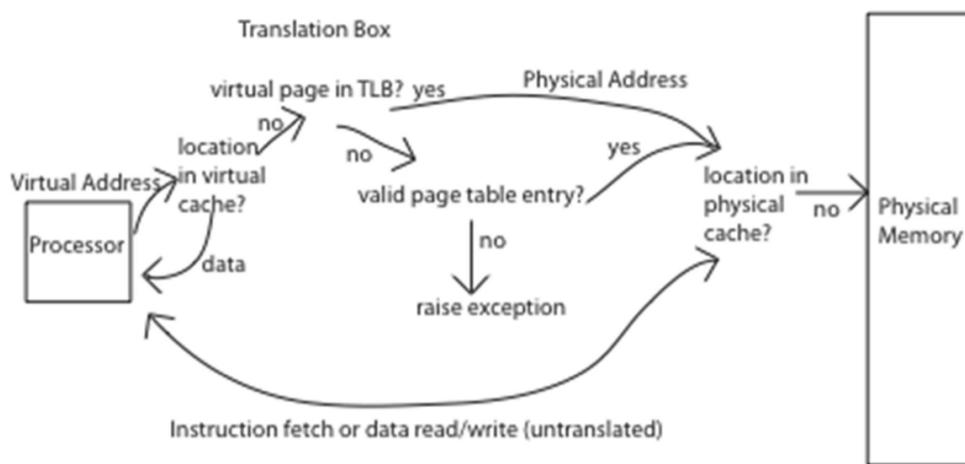


- Superpagina: Gruppo di pagine contigue

x86:

- superpage: Gruppo di page in una page table

- TLB entries:
  - 4 kB
  - 2 MB
  - 16 B



Attention: Alla TLB posso aggiungere l'id del processo, in modo da non smontarla completamente se cambia processo in esecuzione

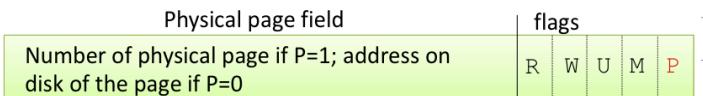
Posso anche aggiungere un bit che ci dice se l'informazione è aggiornata o no.

## On-demand Paging

On demand Paging:



1. TLB miss
2. Page table walk
3. Page fault (page invalid in page table)
4. Trap to kernel
5. Convert address to file + offset
6. Allocate page frame
  - Evict page if needed
7. Initiate disk block read into page frame
8. Disk interrupt when DMA complete
9. Mark page as valid
10. Resume process at faulting instruction
11. TLB miss
12. Page table walk to fetch translation
13. Execute instruction



la page table contiene un descrittore di pagina per ogni pagina

Oltre alle informazioni per la traduzione degli indirizzi, il descrittore contiene dei flags:

R, W : Read / write permission

M, U : Modified / Use bits

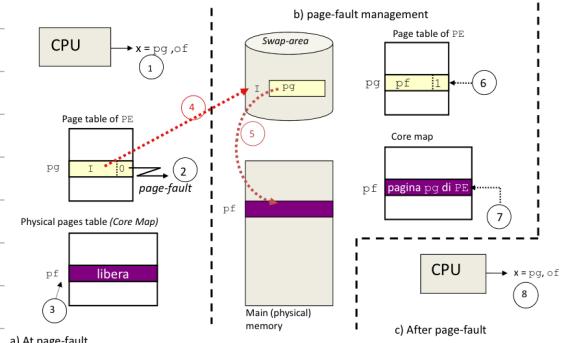
P : bit di Presenza

$\rightarrow$  P=0 : Pagina non nella memoria principale

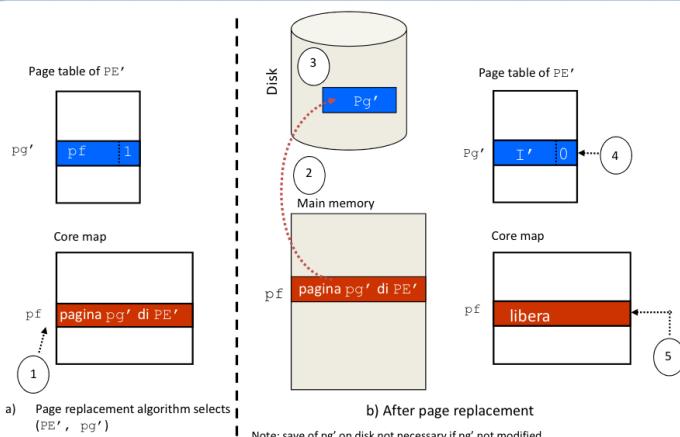
$\rightarrow$  Page fault

P=1 : Pagina nella memoria principale

## Page fault management



## Rimpiazzamento di pagina



## Allocating a Page Frame

- Select old page to evict
- Find all page table entries that refer to old page
  - If page frame is shared
- Set each page table entry to invalid
- Remove any TLB entries
  - Copies of now invalid page table entry
- Write changes to page to disk, if necessary
  - i.e. if the page had been modified

Come si ricava se una pagina è stata modificata / usata?

Setta il bit M o U a 1

- Questi bit possono essere necessari dell'OS:
- Quando le pagine sono inviate al disco
  - Per tracciare quali pagine sono state usate recentemente

### Emulating a Modified Bit

- Some processor architectures do not keep a modified bit in the page table entry
  - Extra bookkeeping and complexity
- OS can emulate a modified bit:
  - Set all clean pages as read-only
  - On first write, take page fault to kernel
  - Kernel sets modified bit, marks page as read-write

### Emulating a Use Bit

- Some processor architectures do not keep a use bit in the page table entry
  - Extra bookkeeping and complexity
- OS can emulate a use bit:
  - Set all unused pages as invalid
  - On first read/write, take page fault to kernel
  - Kernel sets use bit, marks page as read or read/write

## CACHING e Memoria Virtuale

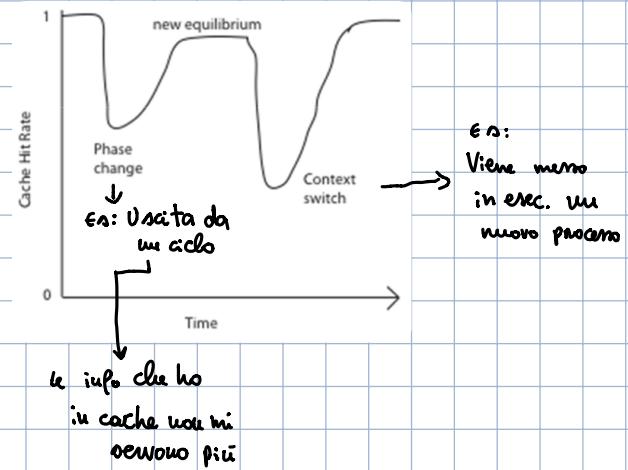
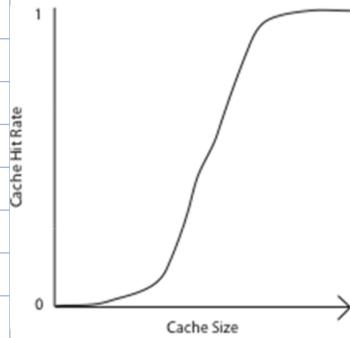
Definizioni:

- Cache: Copia dei dati più veloce da accedere rispetto agli originali
  - miss: Se la cache non ha una copia
  - hit: Se la cache ha una copia
- Cache block: Unità dello storage delle cache
- Località temporale: Un programma tende a fare riferimenti allo stesso spazio di memoria in un certo periodo di tempo.
- Località spaziale: Un programma tende a fare riferimenti a locazioni di memoria vicine

- Working set:insieme di locazioni di memoria che devono essere in cache per avere una ragionevole cache hit rate  
(Vengono usate consistentemente in futuro)

- Thrashing: Quando il sistema ha una cache troppo piccole

**Attention:** I programmi possono cambiare il loro working set  
I cambi di contesto cambiano il working set

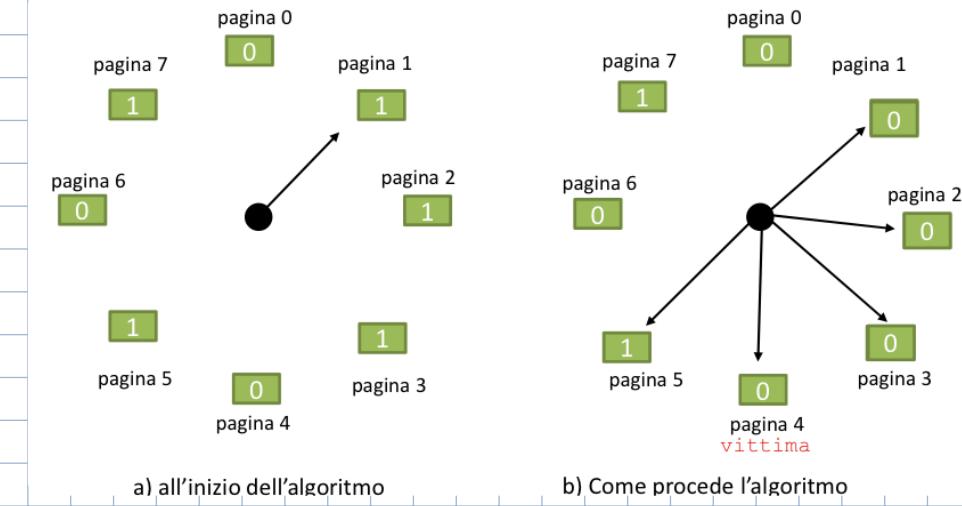


### Cache replacement policy

- 1) Random: Rimpiazza una pagina a caso
- 2) FIFO: Rimpiazza la pagina che è stata in cache (Memoria Principale) per più tempo
- 3) MIN: Rimpiazza la pagina che non verrà usata per il più lungo tempo in futuro.  
Ottimale ma bisogna prevedere quale pagina uscirà.
- 4) LRU (Last recently used): Rimpiazza la pagina che non è stata usata per più tempo.
- 5) NRU (Not Recently Used): Rimpiazza una delle pagine che non sono state recentemente usate

## Algoritmo dell' orologio:

- 1) Periodicamente, parsa da tutte le pagine
- 2) Se una pagina non è utilizzata, è la vittima
- 3) Se una pagina è utilizzata, imposta il bit a 0



## N<sup>th</sup> cache algorithm:

- 1) Periodicamente, parsa da tutti i pochi frames
- 2) Se una pagina non è stata usata in passato per n pagaggi, è la vittima
- 3) se una pagina è stata utilizzata, netta a 0 e settale attiva nel panaggio corrente

## Local and Global page replacement

Algoritmi globali: - la pagina è scelta tra tutte le pagine di tutti i processi

- Page distance viene basata sul tempo globale

Algoritmi locali: - la pagina viene selezionata tra le pagine del processo

- Page distance viene basata sul tempo relativo del processo

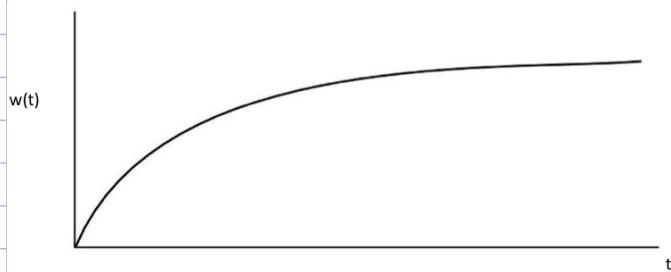
## Algoritmo basato sul working set

tempo in memoria il working set del processo

Il working set può essere definito come:

- Unione di pagine riferite negli ultimi  $k$  accessi

- Unione delle pagine riferite nel periodo  $T$



- working set: The set of pages referred in the last  $k$  memory accesses
- $w(t)$  is the size of the working set as function of time

1) Ogni processo ha un numero di pagine finché riservate per uplodare il suo working set  $\Rightarrow$  WS replacement policy è locale

2) Resident set:

- Unione attuale delle virtual page in memoria
- Alcune di esse possono non essere del WS
- $\neq$  working set

# Working set algorithm

- WS defined as the set of pages referred in the last period P
  - P is a parameter of the algorithm
- For each page:
  - R bit (called "referred" or "use" bit) indicates whether the page had been referred in the last time tick
  - Keep an approximation of the time of last reference to the page
    - At the end of each time tick resets bit R for each page and updates the approximation of time of last reference
  - Age of a page defined as the difference between current time and time of last reference
- At page fault:
  - For each page checks bit R and time of last reference
    - If R=1: set last reference time to current time and resets R
  - The pages referred in the last period P are in the working set and (if possible) are not removed

# Working set algorithm

Current virtual time: 2204

| Page table             |       |     |
|------------------------|-------|-----|
| Time of last reference | Bit R | ... |
| 2084                   | 1     |     |
| 2003                   | 0     |     |
| 1980                   | 1     |     |
| 1213                   | 0     |     |
| 2014                   | 1     |     |
| 2020                   | 1     |     |
| 1604                   | 0     |     |

```

For each page:
 if (R==1)
 time of last reference = current virtual time; R=0

 else if (R==0) && (age>P)
 removes the page
}

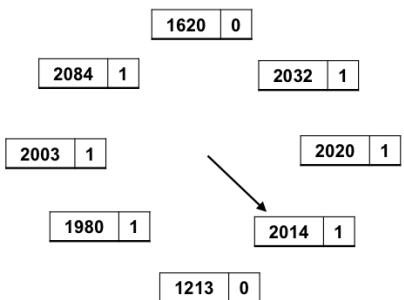
if (age<=P for each page)
 removes the page with smaller time of last reference

```

Age: current virtual time - time of last reference

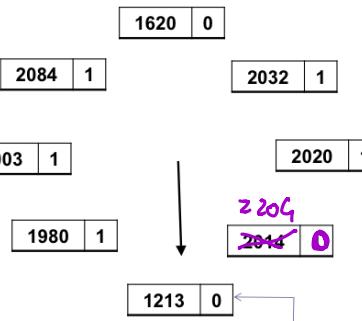
## working - Set + Clock

Current virtual time : 2204



- Considers only the pages in main memory
  - More efficient than scanning the page table
- Pages in a circular list
- At page fault looks for a page out of the WS
  - Better if not "dirty"
  - If selects a dirty page, the page is saved before its actual removal

Current virtual time : 2204

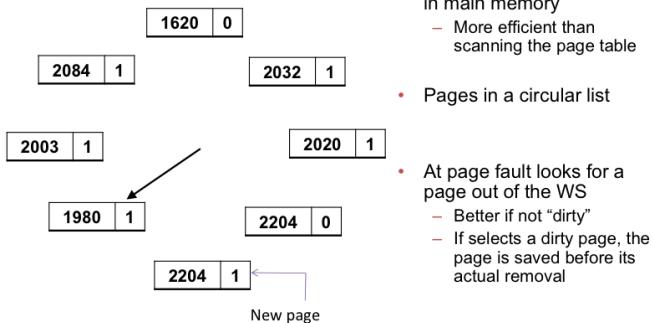


- Considers only the pages in main memory
  - More efficient than scanning the page table
- Pages in a circular list
- At page fault looks for a page out of the WS
  - Better if not "dirty"
  - If selects a dirty page, the page is saved before its actual removal

Removes this page

## WSClock (working set + clock)

Current virtual time : 2204



- Considers only the pages in main memory
  - More efficient than scanning the page table
- Pages in a circular list
- At page fault looks for a page out of the WS
  - Better if not "dirty"
  - If selects a dirty page, the page is saved before its actual removal

## Working set algorithm

- In practice, WS and all page replacement algorithms are executed in advance
- Guarantees free physical pages in case of page fault
  - To speed up the page fault

### On demand paging:

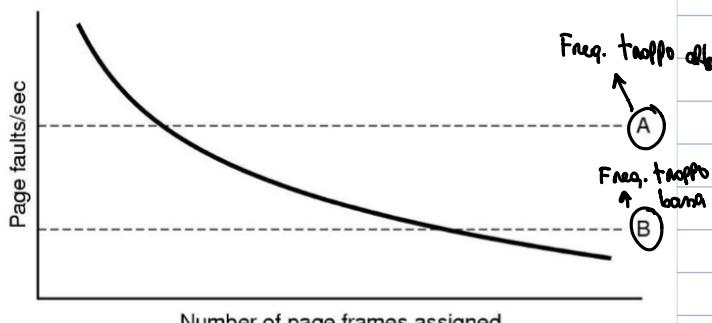
- Initially no page of the process is loaded in memory
- Pages loaded by the process by generating page faults
  - Initially the number of page faults is high
- When the working set had been loaded the number of page faults reduces

### Prepaging

- A new process becomes ready when all its pages in the working set are loaded in main memory
- Need to know (or to predict) what pages will be in the working set initially
  - not easy, can be done for some pages

now visit  
each  
page  
differ-

## Page fault frequency



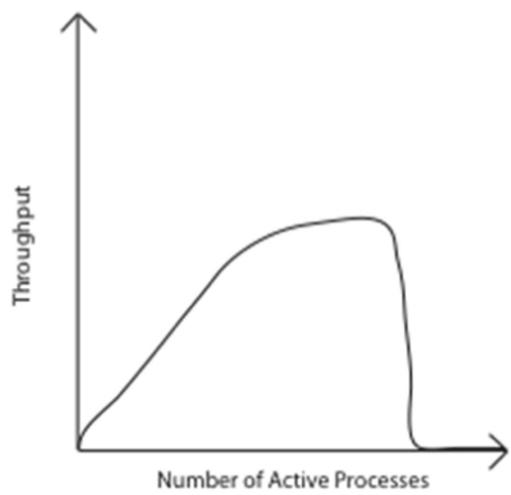
Number of page faults per unit of time, depending on the number of physical pages assigned to the process

## Page Fault Frequency (PFF)

- Determina dinamicamente il numero di pagine assegnate al processo
- Quando il numero di page faults (>A) è molto più grande alla "Freq. Normale" si aumenta lo size del resident set
- Viceversa, lo diminuisce. (se <B)

Cosa succede se il numero dei working sets diventa più grande della memoria finita?

## Thrashing



Succede quando qualche processo richiede più memoria e nessun processo ne richiede di meno. Troppi processi che richiedono troppa memoria.

Soluzione:

- 1) Ridurre il grado di multi-programmazione (competizione per la memoria)
- 2) Scambiare fuori qualche processo al disco

Dove sono salvate le pagine:

- Every process segment backed by a file on disk
  - Code segment -> code portion of executable
  - Data, heap, stack segments -> temp files
  - Shared libraries -> code file and temp data file
  - Memory-mapped files -> memory-mapped files
  - When process ends, delete temp files
- Provides the illusion of an infinite amount of memory to programs

→ Ne ha sempre una copia sul disco

# Gestione della memoria in linux

## Paged segmentation

Virtual memory basata sull' on-demand Paging

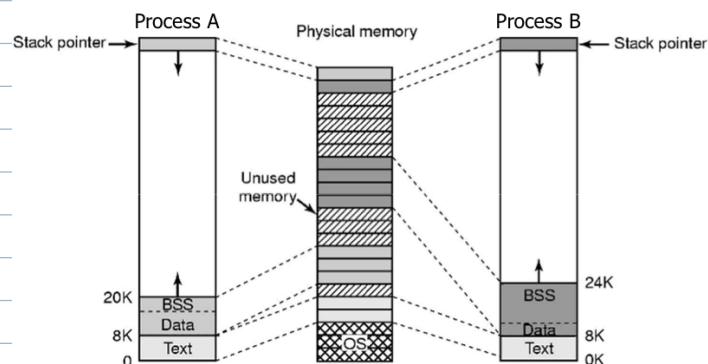
core map:

Page Replacement Algorithm

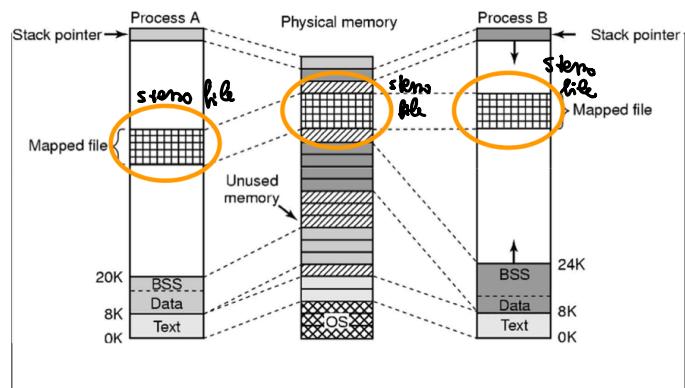
Second chance

- Struttura dati kernel che contiene l'allocazione dei blocchi fisici
- Usata in caso di page fault

## Unix: memory organization



## Unix: sharing memory mapped files



## Modello per l'applicazione file I/O

### 1) Esplicite system calls read/write

- Dati copiati al processo utente da SC
- Applicazioni operano sui dati
- Dati copiati indiretto al kernel usando SC

### 2) Memory - Mapped files

#### Advantages to Memory-mapped Files

- Programming simplicity, especially for large file
  - Operate directly on file, instead of copy in/copy out
- Zero-copy I/O
  - Data brought from disk directly into page frame
- Pipelining
  - Process can start working before all the pages are populated
- Interprocess communication
  - Shared memory segment vs. temporary file

- Aprire il file come un segmento di memoria
- Il programma usa load/store istruzioni sul segmento lavorando implicitamente sul file
- Page fault se una posizione di file non è in mem. ancora
- Kernel porta in memoria il blocco e fa il resume

## 1) Core Map + Tabelle delle pagine

| SO     | SO     | SO | SO | SO | SO |   | A,1<br>2 | B,0<br>10 | C,1<br>3 |    | B,6<br>5 | C,7<br>8 |    | C,3<br>6 | A,5<br>9 | C,5<br>7 | B,2<br>1 | A,7<br>4 |
|--------|--------|----|----|----|----|---|----------|-----------|----------|----|----------|----------|----|----------|----------|----------|----------|----------|
| 0      | 1      | 2  | 3  | 4  | 5  | 6 | 7        | 8         | 9        | 10 | 11       | 12       | 13 | 14       | 15       | 16       | 17       | 18       |
| Pagina | Blocco |    |    |    |    |   | Pagina   | Blocco    |          |    |          |          |    | Pagina   | Blocco   |          |          |          |
| 0      | -      |    |    |    |    |   | 0        | 8         |          |    |          |          |    | 0        | -        |          |          |          |
| 1      | 7      |    |    |    |    |   | 1        | -         |          |    |          |          |    | 1        | 9        |          |          |          |
| 2      | -      |    |    |    |    |   | 2        | 17        |          |    |          |          |    | 2        | -        |          |          |          |
| 3      | -      |    |    |    |    |   | 3        | -         |          |    |          |          |    | 3        | 14       |          |          |          |
| 4      | -      |    |    |    |    |   | 4        | -         |          |    |          |          |    | 4        | -        |          |          |          |
| 5      | 15     |    |    |    |    |   | 5        | -         |          |    |          |          |    | 5        | 16       |          |          |          |
| 6      | -      |    |    |    |    |   | 6        | 11        |          |    |          |          |    | 6        | -        |          |          |          |
| 7      | 18     |    |    |    |    |   | 7        | -         |          |    |          |          |    | 7        | 12       |          |          |          |

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<br>2 | B,0<br>10 | C,1<br>3 |    | B,6<br>5 | C,7<br>8 |    | C,3<br>6 | A,5<br>9 | C,5<br>7 | B,2<br>1 | A,7<br>4 |
|----|----|----|----|----|----|---|----------|-----------|----------|----|----------|----------|----|----------|----------|----------|----------|----------|
| 0  | 1  | 2  | 3  | 4  | 5  | 6 | 7        | 8         | 9        | 10 | 11       | 12       | 13 | 14       | 15       | 16       | 17       | 18       |

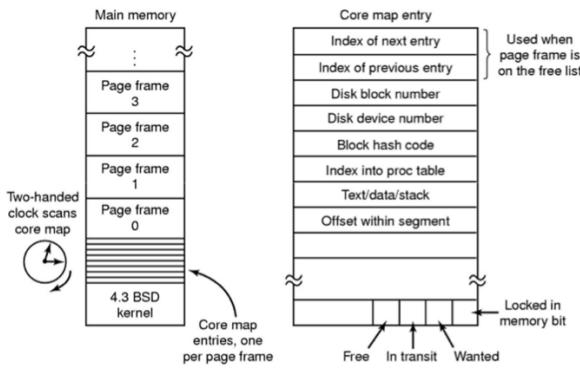
Core Map indicizzata da indice di blocco

--> 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

## Paging in Unix BSD

with an inverse page table (core map):



## Swap dei problemi:

Swapout:

```
if (#freeblocks < minfree) and
(Average[#freeblocks, Δt] < desfree)
// The average is computed over a given time frame Δt
```

The Page Daemon selects «victim» processes based on :

- Priority
- Elapsed time without being executed
- Amount of memory required
- ...

Swaps out one or more victim process until  $\#freeblocks \geq lotsfree + k$  (with  $k > 0$ )

Swapin:

If the number of free blocks is large enough

The Page Daemon selects one or more processes based on:

- Time spent in swapped-out state
- Amount of memory required
- ...

Swapin of one or more processes provided  $\#freeblocks \geq lotsfree + k$  (with  $k > 0$ )

## Page replacement algorithm:

- Second chance (global)
- or variants (e.g. the two-handed clock algorithm)

Page replacement executed periodically by the *Page Daemon*:

- Uses parameters: *lotsfree*, *desfree*, *minfree*, with:

$$lotsfree > desfree > minfree$$

*PageDaemon* algorithm (sketch):

- if ( $\#freeblocks \geq lotsfree$ ) return //no operation required
- if ( $minfree \leq \#freeblocks < lotsfree$ ) or ( $\#freeblocks < minfree$  and Average[ $\#freeblocks, \Delta t$ ]  $> desfree$ ) replage pages until  $\#freeblocks = lotsfree + k$  (with  $k > 0$ )
- if ( $\#freeblocks < minfree$  and Average[ $\#freeblocks, \Delta t$ ]  $< desfree$ ) swapout processes

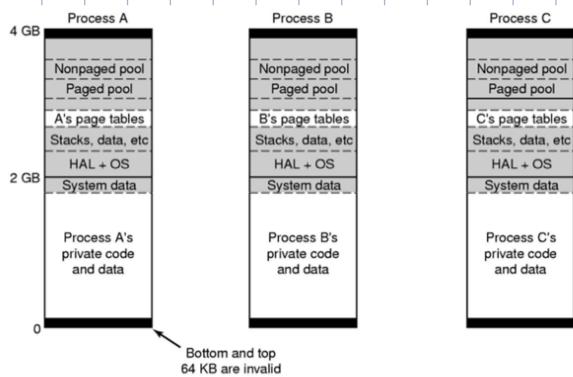
Relationship with the working set theory:

- If  $\#freeblocks < minfree$  :
  - High number of page faults from the last execution of *Page Daemon*
  - There exist processes with  $\#RS < \#WS$  that cause thrashing
    - RS: resident set, i.e. set of pages resident in memory
    - WS: working set
- If Average[ $\#freeblocks, \Delta t$ ]  $< desfree$  :
  - The problem has been there for a while
  - The swapout of some processes will free resources and avoid thrashing
    - The processes with  $\#RS < \#WS$  will expand their resident set, as consequence of their future page faults

# Gestione della memoria in Windows (32 bit)

- Dimensione della memoria virtuale: 4 Gbyte (indirizzo virtuale di 32 bit).
- Memoria virtuale paginata (paginazione a domanda) con pagine di dimensioni fisse (le dimensioni della pagina dipendono dalla particolare macchina fisica).
- Spazio virtuale suddiviso in due sottospazi di 2 Gbyte ciascuno
  - il sottospazio virtuale inferiore è privato di ogni processo
  - il sottospazio virtuale superiore è condiviso tra tutti i processi e mappa il sistema operativo.

## struttura mem. Virtuale



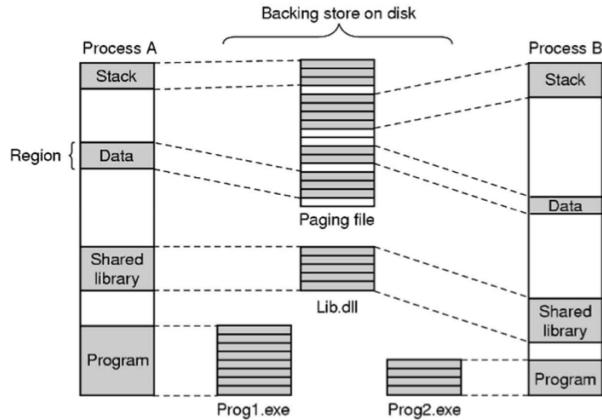
Le aree bianche sono private; le aree scure sono condivise

Spazio virtuale unico, suddiviso in *regioni*

Ogni pagina logica può essere :

- *free* : se non è assegnata a nessuna regione
  - un accesso a una pagina free determina page fault non gestibile
- *reserved* : è una pagina non ancora in uso ma che è stata riservata per espandere una regione
  - ==> non mappata nella tabella delle pagine
  - esempio: riservata per l'espansione dello stack
  - non utilizzabile per mappare nuove regioni
  - un accesso a una pagina reserved determina page fault gestibile
- *committed* : se appartiene a una regione già mappata nella tabella delle pagine
  - un accesso a una pagina committed non presente in memoria risulta in un page fault, che determina il caricamento della pagina solo se questa non si trova in una lista di pagine eliminata dal working set

## Windows: backing store



# Page Fault Management

Windows adopts a working set algorithm

- Local algorithm
- However, here working set is defined as the set of *resident pages (RS)*
- For a given process,  $x = \#RS$  ranges in  $[min, max]$ 
  - $min$  and  $max$  are initially set to default values...
  - ... but they vary during the life of a process, to adapt to its memory need

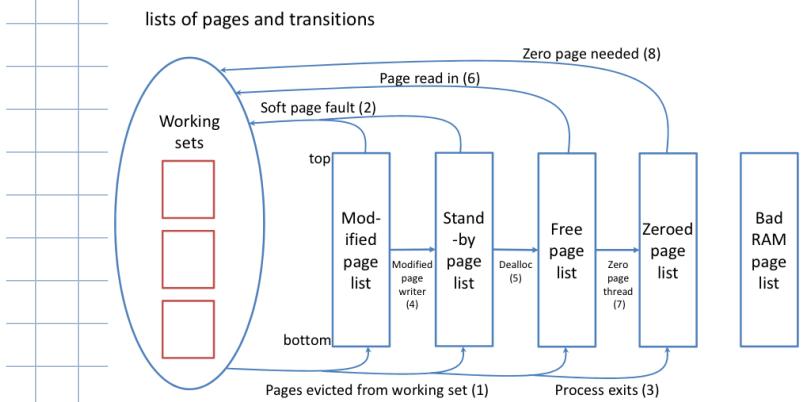
- At page fault of process  $P$ , the requested page is always loaded in a free block in memory
- Hence the size of the resident set of  $P$  increases ( $x = x + 1$ )
- If  $x \geq max$  the pages of  $P$  in excess will be removed by the working set manager

## Windows : Working set Manager

Adopts a working set page replacement policy

- For each process with  $x > min$ :
  - For each page  $p$  with *reference bit R* = 1: reset  $R$  and  $count(p)$
  - For each page  $p$  with  $R$  bit = 0: increase  $count(p)$ 
    - $count(p)$  is an approximation of the time of last reference (past distance) of a page
  - Set for removal  $x - max$  pages in decreasing order of  $count(p)$
- If the number of free blocks is still low: remove pages also of processes with  $x > min$

## Management of pages



## Storage System e File System

Storage device:

- Dischi Magnetici
- Memorie flash (USB)

File system:

- 1) Assegnazione degli storage device
- 2) Persistenza dei dati salvati nel file system
- 3) Naming: Da un nome ai dati nel disco
- 4) Performance: Cached data e Data placement and data organization
- 5) Accesso controllato ai dati condivisi

# Astrazione dei file system

1) Organizzazione gerarchica e accesso controllato

2) Unione di dati con nome (File)

3) Tolleranza a crash e errori

4) Performance

5) Directory

6) Path : Stringa che identifica file e directory

7) links ↗ Hand

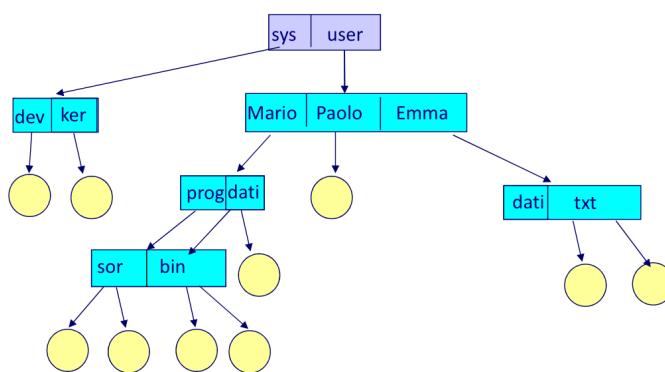
soit (collegamento)

8) Mount

## Struttura di un file system

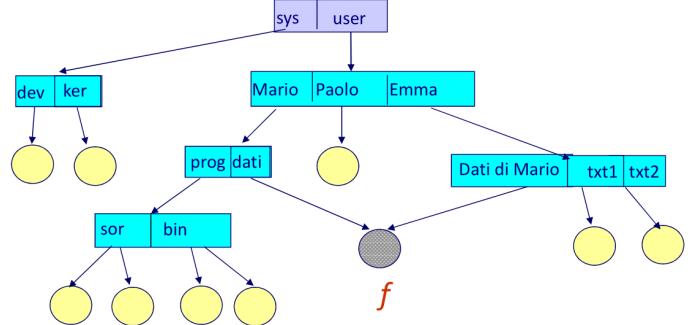
1) Albero

Root directory



2) Grafo aciclico

Root directory

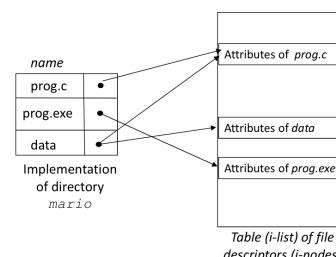
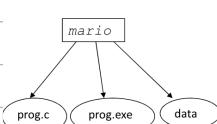


Directories: Struttura dati che collega un file name a un file attributes

→ In Unix:

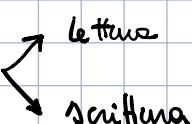
Implementation of directories in Unix:

- The directory is a table that includes the references to the file descriptors (i-nodes), which are stored in a separate data structure in the disk



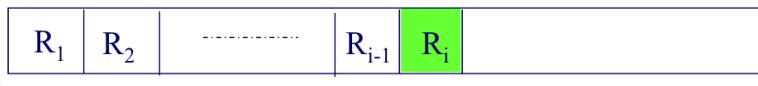
- File size
- Address nel disco
- :
- :

## Accesso a un file



1) Accesso sequenziale: se il file è una sequenza di record logici  $\{R_1, \dots, R_n\}$ ,

Per accedere a ogni record logico  $R_i$  è necessario accedere  
prima agli  $(i-1)$  records



Access operations:

- *readnext*: read next logical record
- *writenext*: write next logical record

Each operation (read/write) positions an access pointer on  
the next logical record

2) Accesso diretto: se il file è un insieme  $\{R_1, \dots, R_n\}$  di record logici  
l'utente può accedere direttamente al record  $R_i$

Access operations:

- *read(f, i, &V)*: read logical record  $i$  of file  $f$ ; the read data is stored in buffer  $V$ ;
- *write(f, i, &V)*: write the content of buffer  $V$  on the logical record  $i$  of file  $f$ .

## File system interface

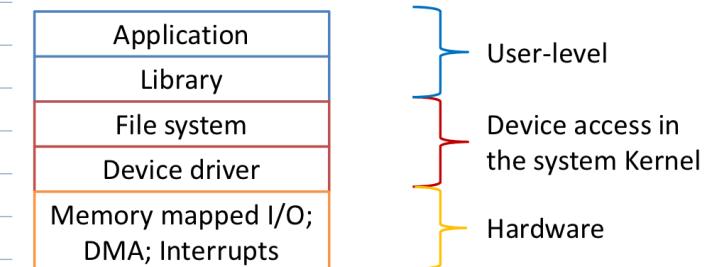
UNIX file open is a Swiss Army knife:

- Open the file, return file descriptor
- Options:
  - if file doesn't exist, return an error
  - If file doesn't exist, create file and open it
  - If file does exist, return an error
  - If file does exist, open file
  - If file exists but isn't empty, nix it then open
  - If file exists but isn't empty, return an error
  - ...

## File system API

- create, link, unlink, createdir, rmdir
  - Create file, link to file, remove link
  - Create directory, remove directory
- open, close, read, write, seek
  - Open/close a file for reading/writing
  - Seek resets current position
- fsync
  - File modifications can be cached
  - fsync forces modifications to disk (like a memory barrier)

Acceso Ai device  $\Rightarrow$  Un posix è equivalente ad accedere ad un file



## Device drivers

- 1) Punte superiore : Hw - Indipendent
- 2) Punte inferiore : Hw - dipendent

Per interagire:

File system

- 1) Definisce uno spazio con nome
- 2) Implementa politiche di caching
- 3) È Hw indipendente

Device driver

- 1) Hw - dependent
- 2) Gestisce la sincronizzazione con il device
- 3) Gestisce il transf. dati
- 4) Gestisce darta failurs

Memory mapped I/O

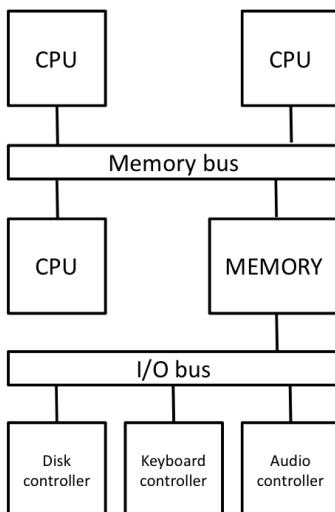
(per gestire op. di  
input/output)



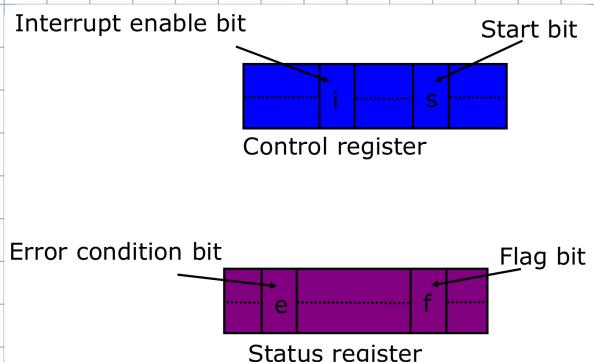
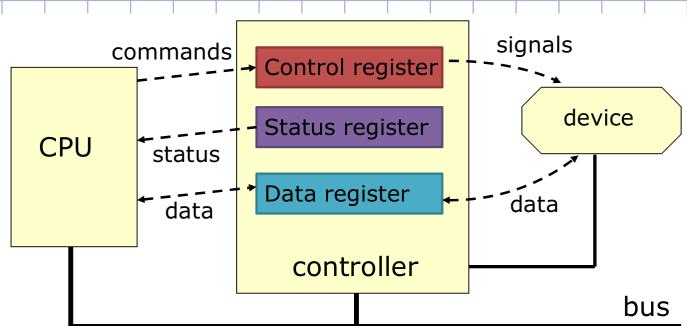
Le registri interni del device sono  
mappati con locazioni di memoria



R/W op. a quelle loc. sono redinrette  
al device



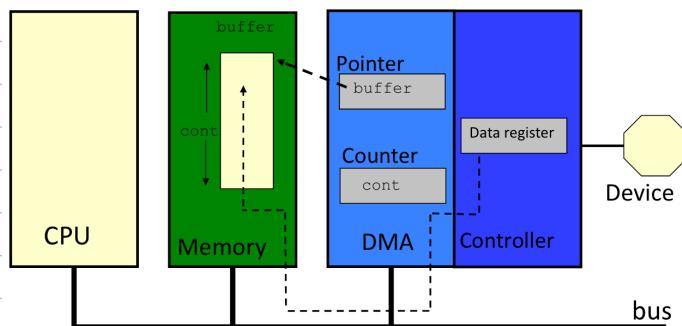
Device Controller: Usato per i software input. Prende i dati  
(sta nel device) dalla CPU e li mette sul device.



# Direct Memory Access (DMA)

Gestisce i trasferimenti dati dal Device alle memorie senza l'uso della CPU

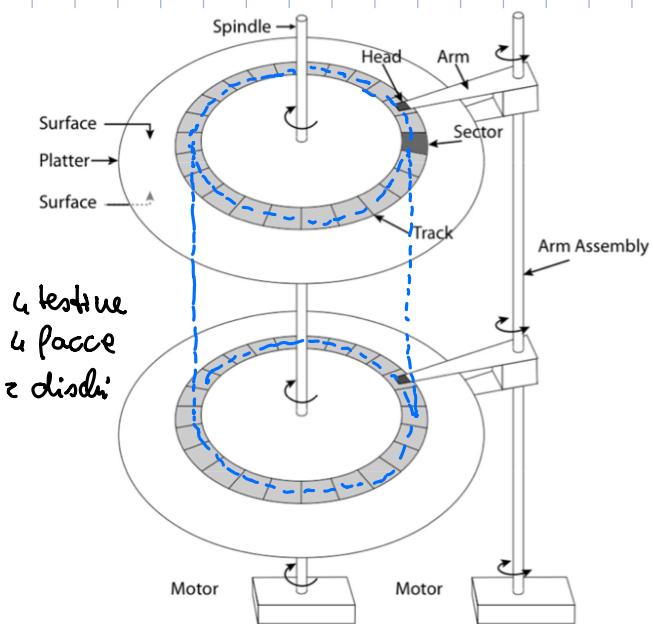
- 1) Device driver programma il DMA
- 2) Quando il transf. è completo, il DMA lancia un interrupt
- 3) L'interrupt riportina il device driver che ha concluso l'interrupt



## STORAGE DEVICE

### 1) Dischi magnetici

Composizione:



- ↔ Proprietà:
- 1) Raramente corrotti
  - 2) Alta capacità e poco costo
  - 3) Block lvl Random access
  - 4) Performance buone dovute all'accesso random
  - 5) Performance buone per streaming access
  - 6) Scrittura lenta, lettura non troppo



tracce (tracks) :

↓  
Anelli concentrici

1) lunghezze  $\approx$  1 micron

2) separate da regioni quadra inutilizzate

(Riduce le possibilità di connessione tra scritte di traccia vicine)

3) lunghezza delle tracce varia nel disco

- outside: più settori per ogni traccia, più lunghezza delle bande

- Il disco è organizzato in regioni composte da tracce con lo stesso # di settori/tracce

- Solo la metà più esterna del disco viene usata

Settori: Contengono codice sofisticato per la correzione degli errori

- la testina del disco è più grande di una traccia

- Nasconde la connessione tra due scritte di traccia vicine.

1) Sector spanning: Rimappa bad sectors a settori di riserva sulla stessa superficie

2) Split spanning: Rimappa tutti i settori (Quando c'è un bad sector) per preservare il comportamento sequenziale

3) Track skewing: Il setore numero l'offre tra una traccia e la prossima, per permettere alla testina del disco movimenti per salti sequenziali.

Settori e blocchi:

1) A low level il controller accede a settori individuali

- Typical sector size is 256/512 bytes
- Identified by a triple: <#cylinder, #face, #sector>

2) Il gruppo di driver del disco raggruppa settori contigui in un blocco

- Typical block size is 2/4/8 Kbytes

- identified by a pointer on a contiguous address space (from 0 to max-blocks)

Dato a sector number  $b$ , e una tripla  $\langle c, f, s \rangle$

$$b = c (\# \text{ faces} \cdot \# \text{ sectors}) + f \cdot (\# \text{ sectors}) + s$$

$\downarrow$                      $\downarrow$                      $\downarrow$                      $\downarrow$   
numero cilindro      numero di facce del disco      numero di settori per traccia      numero settore

Di Conseguenza:

$$c = b \text{ div } (\# \text{ faces} \cdot \# \text{ sectors})$$
$$f = (b \text{ mod } (\# \text{ faces} \cdot \# \text{ sectors})) \text{ div } \# \text{ sectors}$$
$$s = (b \text{ mod } (\# \text{ faces} \cdot \# \text{ sectors})) \text{ mod } \# \text{ sectors}$$

Performance di un disco:

$$\text{Latenza} = \text{Seek time} + \text{Rotation Time} + \text{Transfer time}$$

Seek time: tempo per muovere il disk arm sulla traccia (1-20ms)

Rotation time: tempo da aspettare per ruotare il disco sotto la testina  
(4-15ms)

Transfer time: tempo di trasf. dei dati in/out dal disk alla memoria  
transfer rate: 50-100 KB/s (5-10 usec/sector)

Esempio:

1) Quanto tempo impiega per 500 random lettura del disco?

- Seek: average 10.5 msec
- Rotation: average 4.15 msec
- Transfer: 5-10 usec
- $500 * (10.5 + 4.15 + 0.01) / 1000 = 7.3 \text{ seconds}$

2) E per 500 sequenziali?

- Seek Time: 10.5 ms (to reach first sector)
- Rotation Time: 4.15 ms (to reach first sector)
- Transfer Time: (outer track)  
 $500 \text{ sectors} * 512 \text{ bytes} / 128 \text{ MB/sec} = 2 \text{ ms}$

$$\text{Total: } 10.5 + 4.15 + 2 = 16.7 \text{ ms}$$

3) Quanto grande è richiesto un transf. per arrivare l'80% del massimo transf. netto del disco?

Assume y rotations are needed, then solve for y:

$$0.8(10.5 \text{ ms} + 8.4 \text{ ms } y + 1 \text{ ms } (y-1)) = 8.4 \text{ ms } y$$

seek iniziale      y rotazioni      y-1 extra seeks

80% of time required to read y tracks with overhead =

= time to read y tracks without overhead

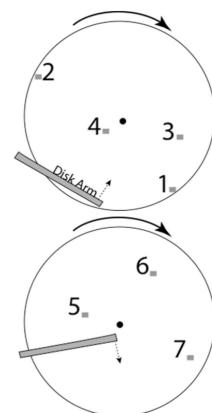
Total:  $y \sim 9$  rotations,  $\sim 10\text{MB}$

$\Rightarrow$  Shortest seek time first  $\Rightarrow$  Problema: altera incertezza per salti ai lati del disco

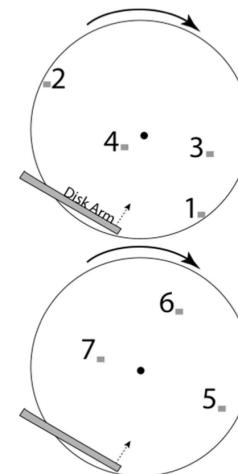
Disk scheduling  $\Rightarrow$  Ottimizzare il tempo di spostamento della testina (seek-time) per la n/w dei rotori

SCAN: Muovi il disk arm in una direzione fin quando tutte le richieste sono sotto soddisfatte, poi vai nella direzione opposta.

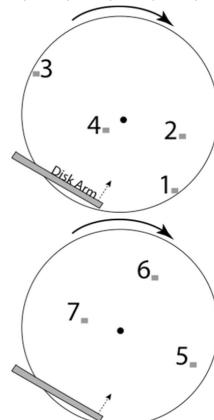
Direzioni: 1) verso centro 2) verso l'esterno)



C SCAN: Muove il disk arm in una direzione fin quando tutte le richieste sono sotto soddisfatte. Poi inizia nuovamente dalla più lontana.



R-CSCAN: CSCAN ma tiene di conto che la track switch è minore del rotational delay



Esempio:

1) Quanto impegno per leggere 500 random lettura, in un qualunque ordine?

- Disk seek: 1ms (most will be short)
- Rotation: 4.15ms
- Transfer: 5-10usec
- Total:  $500 * (1 + 4.15 + 0.01) = 2.2$  seconds
  - Would be a bit shorter with R-CSCAN
  - vs. 7.3 seconds if FIFO order

2) Quanto impegno per leggere tutti i bytes fuori dal disco

- Disk capacity: 320GB
- Disk bandwidth: 54-128MB/s
- Transfer time =  
Disk capacity / average disk bandwidth  
 $\sim 3500$  seconds (1 hour)

## Memorie flash

$\Rightarrow$  proprietà:

- 1) Random write costato
- 2) Capacità a costo intermedio (2x disk)

- le scritture devono operare su celle "pulite"

Nessun aggiornamento in place

Quindi:

1) richiede la cancellazione di grandi blocchi  
prima delle scritture

$\downarrow$   
 $128 - 512 \text{ kb}$

3) Accesso random a livello block

4) Buone performance in read,  
peggiori in write

2) tempo di cancellazione: Alcuni millisecondi

- Write/Read page (2-4kB)  $\Rightarrow$  50-100 usec

- lettura più veloci delle scritture

## Flash translation layer:

1) Pagine cancellate in anticipo  $\Rightarrow$  Ci sono sempre delle pagine disponibili per le scritte

2) Per conoscere dove sono le pagine del mio file:

Flash device Firmware  $\Rightarrow$  Mappa le pagine logiche a una location fisica



- Move live pages as needed for erasure
  - Garbage collect empty erasure block by copying live pages to new location
- Wear-leveling
  - Can only write each physical page a limited number of times
- Avoid pages that no longer work

3) Flash translation layer viene occupato sul garbage collection

- I live blocks devono essere rimappati a una nuova location per compattare le pagine libere in ordine per procedere con le block erasure

Succede anche con molto spazio libero

## TRIM COMMAND

- File system dice al device quando le pagine non sono in uso
- Aiuta il File translation layer a ottimizzare le garbage collection

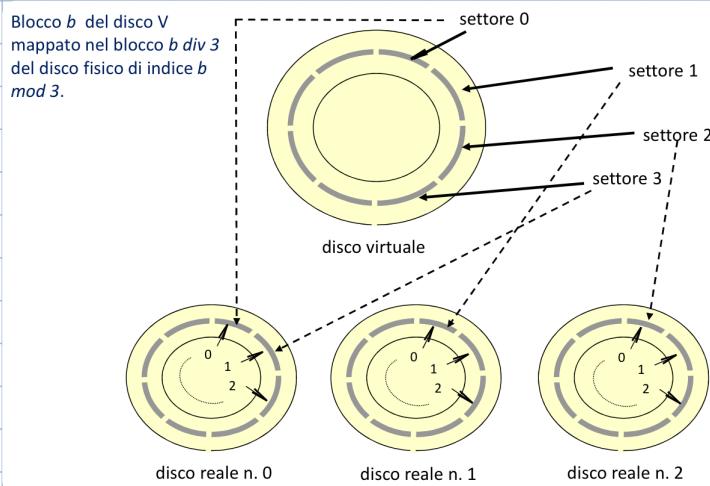
# Dischi RAID (Redundant array of Independent Disks)

Architettura RAID:

- Realizza un *disco virtuale* di capacità superiore a quella dei singoli dischi  
l'interfaccia è quella di un unico disco
- sfrutta il parallelismo per ottenere un accesso più veloce  
i blocchi consecutivi di uno stesso file sono distribuiti sui dischi dell'array in modo da permettere operazioni contemporanee
- sfrutta la ridondanza per accrescere l'affidabilità  
la ridondanza permette di correggere gli errori di certe classi

Diversi livelli di architettura RAID, con diversi livelli di ridondanza

Un disco virtuale viene suddiviso in più dischi reali (horizontal scale)



Livelli dischi raid:

Livello 0: Dischi asincroni, nessuna ridondanza

- Si possono effettuare contemporaneamente operazioni indipendenti
- Anche detto JBOD (just a bunch of disks)

Livello 1: Dischi asincroni, disco con copie ridondanti (*mirror*)

- Si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Livello 2: Dischi sincroni, i dischi ridondanti contengono codici per la correzione degli errori

- non si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Livello 3: Dischi sincroni, un solo disco ridondante

- contiene la parità del contenuto degli altri dischi
- non si possono effettuare contemporaneamente operazioni indipendenti e correggere errori

Livello 4: Dischi asincroni; un disco ridondante

- contiene la parità del contenuto degli altri dischi
- si possono effettuare contemporaneamente operazioni indipendenti e correggere errori
- Il disco ridondante è sovraccarico nei piccoli aggiornamenti

Livello 5: Come livello precedente, ma parità distribuita tra tutti i dischi

- permette un miglior bilanciamento del carico tra i dischi

Livello 1+0: Dischi asincroni, Mirror di stripes

Livello 0+1: Dischi asincroni, Stripe di mirror

Per la correzione di errori: ipotesi di *errori singoli* e *crash faults*

1)

Dischi sincroni: vengono distribuiti i bit

Livello 2: Ridondanza con codice correttore di errori, stripe bit-level

Esempio bit 0, 1, 2, 3 e 4 di informazione, bit 5, 6 e 7 ridondanti

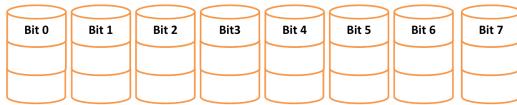
- Lettura o scrittura contemporanea di tutti i bit della strip; *rilevazione errori doppi e correzione di errori singoli (Hamming ECC Code)*

Livello 3: Ridondanza con parità, stripe bit-level (può anche essere byte-level)

Esempio: bit 5 parità dei bit 0, 1, 2, 3, 4

- Lettura o scrittura contemporanea di tutti i byte della strip e *correzione di crash faults singoli*

Raid level 2



Raid level 3



2)

Dischi asincroni, vengono distribuite le *stripe*

Livello 4: Ridondanza con *strip* di parità

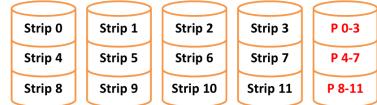
Esempio: disco 4 contiene le strip di parità delle strips omologhe dei dischi 0, 1, 2, 3

- esecuzione contemporanea di operazioni indipendenti e *correzione di crash faults singoli*

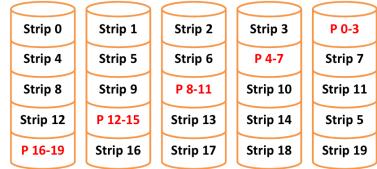
Livello 5: Come livello 4, ma strip di parità distribuite nei vari dischi

- migliore bilanciamento del carico tra i dischi

Raid level 4



Raid level 5



## Esempio:

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  $b \text{ mod } 4$  del disco fisico di indice  $b \text{ mod } 4$ . 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.

3)

Dischi asincroni, organizzazione in cluster

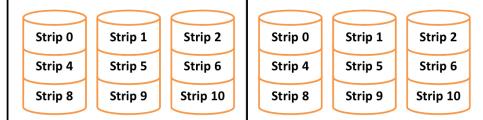
Livello 0+1 (o 01): Mirror di stripes

- Organizzato in gruppi: ogni gruppo è un mirror di un RAID 0.

Livello 1+0 (o 10): Stripe di mirror

- Organizzato in gruppi, ogni gruppo è un RAID 1. I gruppi realizzano un RAID 0
- Miglior tolleranza ai guasti del 01: se si guastano due dischi in gruppi diversi il sistema può ancora funzionare (i dischi in un gruppo non sono indipendenti)

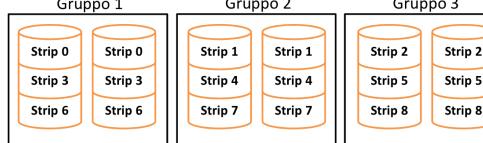
Raid level 01



Gruppo 1

Gruppo 2

Raid level 10



Gruppo 1

Gruppo 2

Gruppo 3

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.

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 | 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`, 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 | 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 |

**File System**  $\Rightarrow$  Scopo: traduzione ind. da logico a fisico

- la maggior parte dei file ha grandezza piccola, ma il totale dello storage viene occupato dai file grandi;
- Vengono accediti per la maggior parte file piccoli: ma contano di più per i bytes I/O i file grandi;

**FS Design:**

File piccoli

- 1) Blochi piccoli
- 2) Operazioni concorrenti e non seq.
- 3) File usati insieme salvati insieme

File grandi

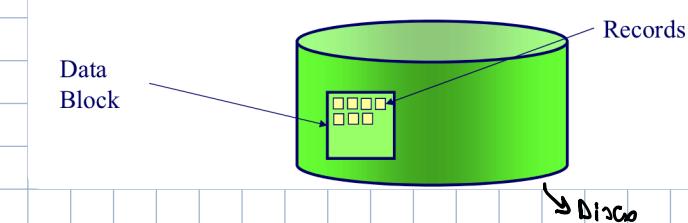
- 1) Blochi grandi
- 2) Allocazione contigua per accesi sequenziali
- 3) lookup efficienti per accesi random

**Strutture dati:**

- **Directory:** Collegano un file name a un file metadata
- **File Metadata:** Describe come trovare il file + altre proprietà di esso
- **Free Map:** lista dei dischi liberi  $\Rightarrow$  Per sapere i blocchi liberi sul disco

**Data block e records**

- Data e file sono accessibili in records (organizzati in records)
- Data (records) sono salvati ed accessibili in blocchi (Fisicamente)
- Di solito block size  $\gg$  record size



# Design Challenges

- Index structure
  - How do we locate the blocks of a file?
- Index granularity
  - What block size do we use?
- Free space
  - How do we find unused blocks on disk?
- Locality
  - How do we preserve spatial locality?
- Reliability
  - What if machine crashes in middle of a file system op?

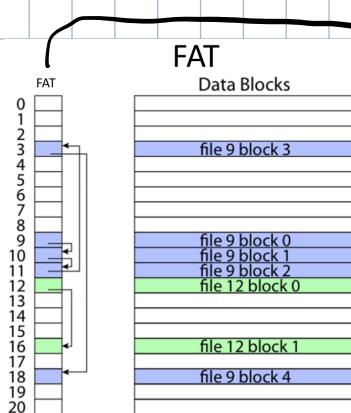
# File System Design Options

|                       | FAT             | FFS                          | NTFS                    |
|-----------------------|-----------------|------------------------------|-------------------------|
| Index structure       | Linked list     | Tree (fixed, asym.)          | Tree (dynamic)          |
| granularity           | block           | block                        | extent                  |
| free space allocation | FAT array       | Bitmap (fixed location)      | Bitmap (file)           |
| Locality              | defragmentation | Block groups + reserve space | Extents Best fit defrag |

req di blocchi  
adiacenti

Microsoft file allocation table (FAT)  $\Rightarrow$  più antico

- linked list index structure
- File table:
  - Mappa lineare di tutti i blocchi sul disco
  - Ogni file è una linked list di blocchi



Nella fat individuiamo il puntatore al blocco libero  
successivo

FAT

- Pros:
  - Easy to find free block
  - Easy to append to a file
  - Easy to delete a file
- Cons:
  - FAT size
    - Should be uploaded in main memory (**tutta la FAT**)
    - Limitation to file system size
  - Limited metadata and no protection
  - Fragmentation
    - File blocks for a given file may be scattered
    - Files in the same directory may be scattered
    - Problem becomes worse as disk fills

| 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

Limitazioni di FAT  $\Rightarrow$

Capacità disco = Numero di blocchi indirizzabili =  $2^L$

Mai massima estensione FS =  $B \cdot 2^L$  byte

Fat occupa compl. =  $N \cdot 2^L$  byte



n byte occupati da un elemento della FAT

Posto:

$L$  = lunghezza in bit degli elementi della FAT

$B$  = Dimensione in byte dei blocchi del disco

# Berkeley UNIX FFS (Fast File System)

## - i-node table

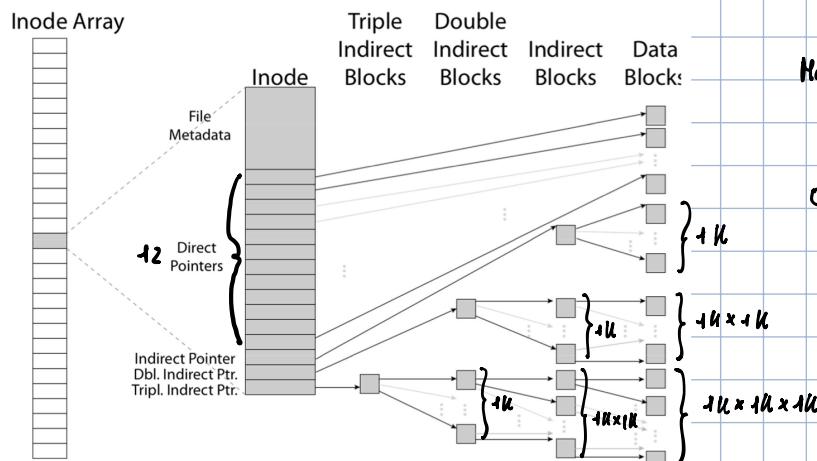


- Metadata = File owner, access permissions ...
- Unione di puntatori a blocchi di dati

## Organizzazione fisica del disco in UNIX



- FFS INODE:**
- Metadata = File owner, access permissions ...
  - Unione di 12 puntatori a dati (blocchi) (con 4 KB blocks  $\Rightarrow$  40 KB max size)
  - **Indirect block pointer** = Puntatore al blocco del disco di data pointers (+1K puntatori a blocchi di dati (4 KB))
  - **Doubly Indirect block pointer** = 1K indirect blocks 4 GB ( $+4\text{GB} + 48\text{KB}$ )
  - **Triply Indirect block pointer** = 1K doubly indirect blocks 4 TB ( $+4\text{GB} + 4\text{MB} + 48\text{KB}$ )



Maxima capacità del disco:

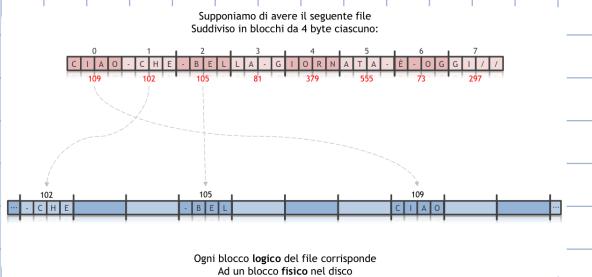
Ogni blocco indiretto contiene: byte blocco / lunghezza byte indirizzi

Ogni blocco indiretto doppio:  $2(\text{byte blocco} / \text{lunghezza byte indirizzi})$

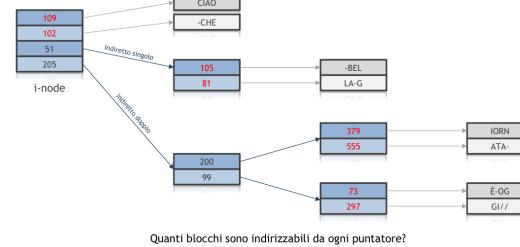
File ind. da un i-node: ind direz. ind indiretti + ind indiretti doppi + i + t

## Esempio:

- Blocco di 4 byte
- Puntatori di 2 byte
- File di 26 byte (quindi 7 blocchi)
- I node con:
  - 2 puntatore diretti
  - 1 puntatore indiretto singolo
  - 1 puntatore indiretto doppio



Supponiamo adesso che gli indirizzi siano a 2 byte



Quanti blocchi sono indirizzabili da ogni puntatore?  
4Byte/2Byte = 2

## Pros

- Efficient storage for both small and large files
- Locality for both small and large files
- Locality for metadata and data

## Cons

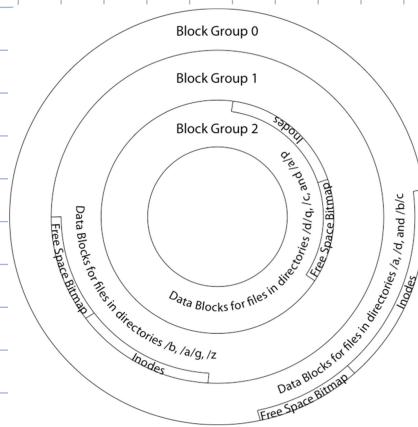
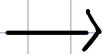
- Inefficient for tiny files (a 1 byte file requires both an i-node and a data block)
- Inefficient encoding when file is mostly contiguous on disk (no equivalent to superpages)
- Need to reserve 10-20% of free space to prevent fragmentation

## FFS Asymmetric tree

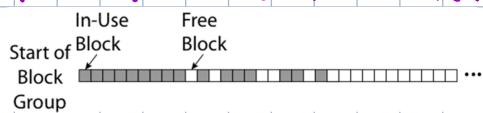
- File piccoli  $\Rightarrow$  Shallow tree  $\Rightarrow$  Efficient storage for small file
- File grandi  $\Rightarrow$  Deep tree  $\Rightarrow$  efficient for random access

## FFS locality:

- Block group allocation
  - Block group is a set of nearby cylinders
  - Files in same directory located in same group
  - Subdirectories located in different block groups
- i-node table spread throughout disk
  - i-nodes, bitmap near file blocks
- First fit allocation
  - Small files fragmented, large files contiguous



## FFS first block allocation

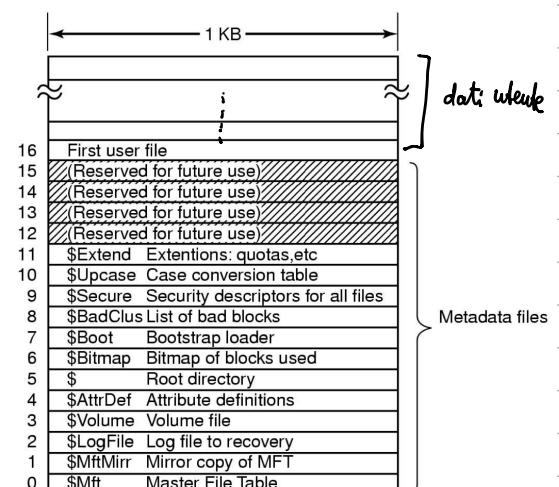


## NTFS (New technology file system) $\Rightarrow$ Microsoft $\Rightarrow$ migliore attualmente

- Master file table:
  - tabella di records (uno per file)
  - 1KB flexible storage per metadata file e data
  - la tabella è una stessa un file

- Extents: Sequenza di blocchi contigui dello stesso file

## MFT



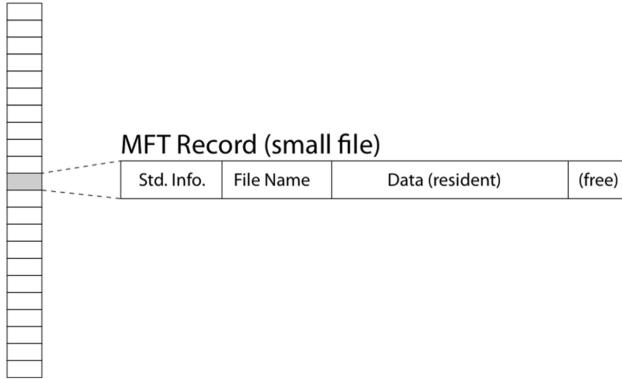
Per descrivere questa seq. non mi servono tutti

i puntatori, ma solo quelli che coprono tutti i

blocchi. Ese: Parte da 1 ed arriva a 100

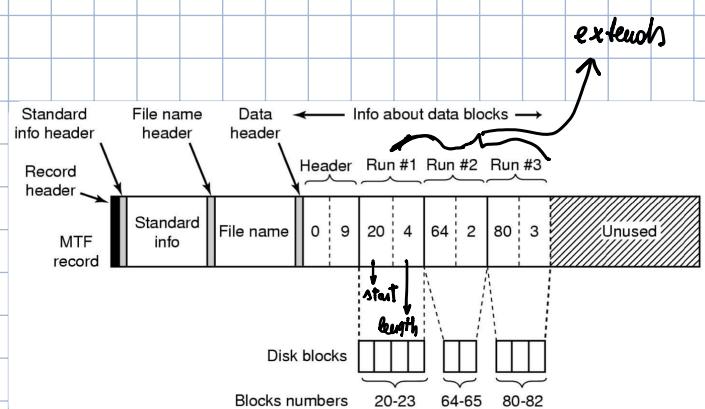
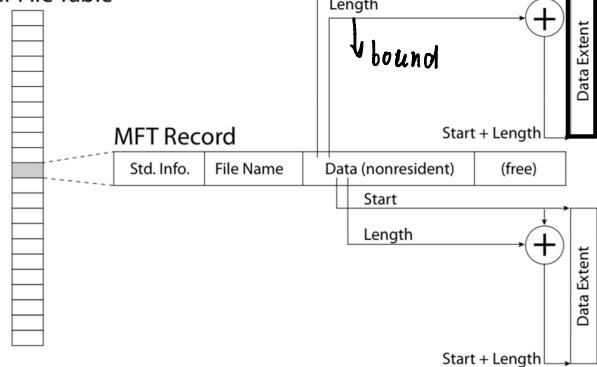
## NTFS File piccoli

Master File Table



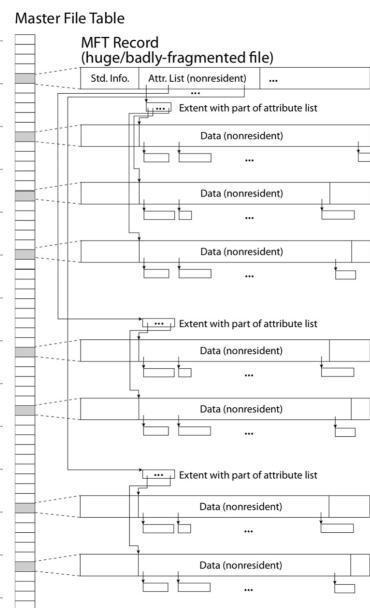
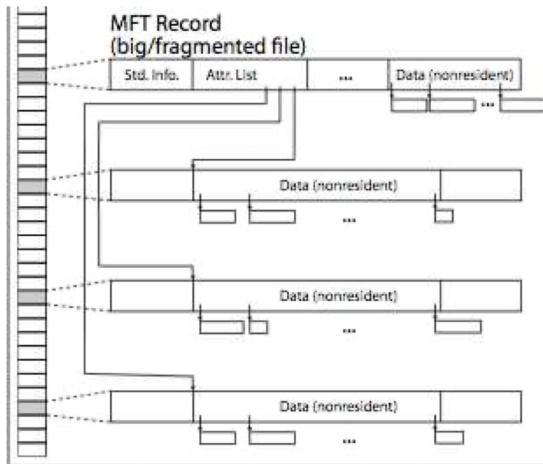
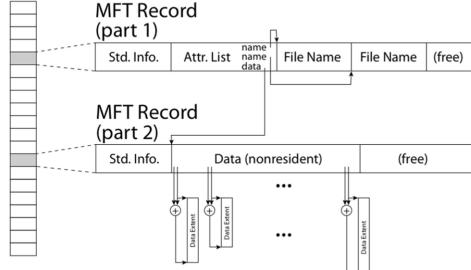
## NTFS file medi

Master File Table



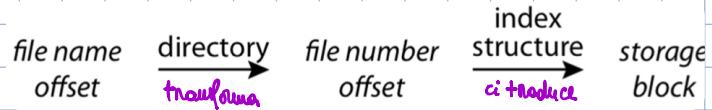
## NTFS iudirect blocks

Master File Table

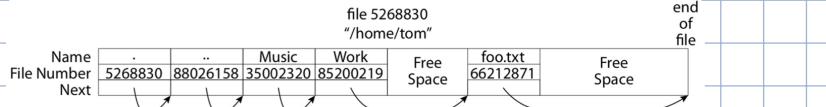


Usato nel caso non avrai abbastanza spazio nel singolo record MFT

## Named data in un FS



### Directories:



- Directories are files
  - Map file name to file number (MFT #, inode num)
- Table of file name → file number
  - Small directories: linear search

