

# SISTEMI OPERATIVI

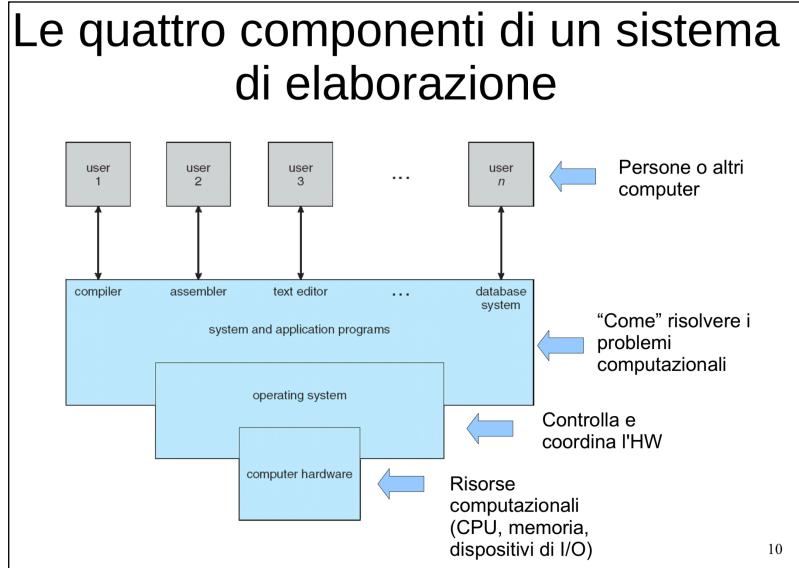
Il sistema operativo serve come interfaccia tra livello applicativo ed hardware, non compie operazioni apparentemente utili ma fornisce un ambiente per il corretto funzionamento dei programmi.

## Funzioni e servizi base

- Controllo flusso ed esecuzione dei programmi
- Allocazione risorse e gestione memoria
- Accesso a dispositivi I/O
- Comunicazione tra processi
- Gestione errori e malfunzionamenti
- Protezione
- Gestione statistiche e raccolta dati

## Componenti base SO

- Gestore dei processi
- Gestore della memoria
- Gestore della memoria secondaria
- Gestore dei sistemi di I/O
- Gestore dei file
- Gestore della comunicazione di rete
- Gestore della sicurezza
- Interprete dei comandi

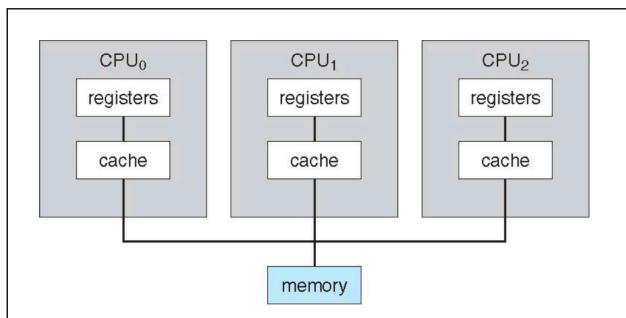


10

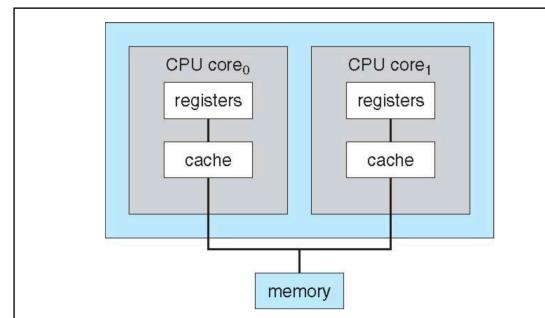
## Tipi di sistema

- **sistemi paralleli** => massimizzano throughput<sup>1</sup> (multiprocessore / multicore / distribuiti) si può simulare con sistemi time-sharing<sup>2</sup>
- **Sistemi real-time** => forniscono risultati entro il tempo prestabilito, possono essere HARD real time o SOFT real time (embedded / sistemi di controllo)

## SISTEMI MULTIPROCESSORE



## SISTEMI MULTICORE

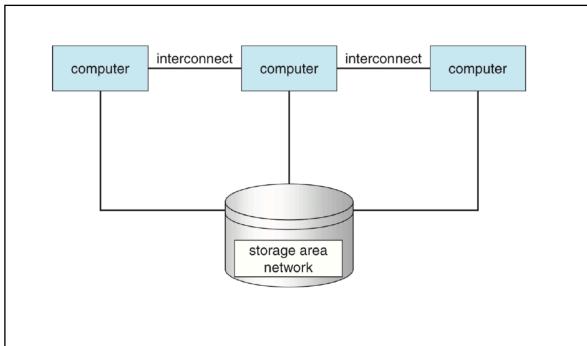


Nel caso di processori con multithread fisico si ha un unico chip con una sola ALU ma 2 set di registri separati che permettono l'esecuzione quasi simultanea.

<sup>1</sup> throughput = (produttività) capacità elaborativa in un tempo determinato

<sup>2</sup> time-sharing = possiedono scheduler che assegna quanti di tempo limitati ad ogni processo

## SISTEMI CLOUSTER

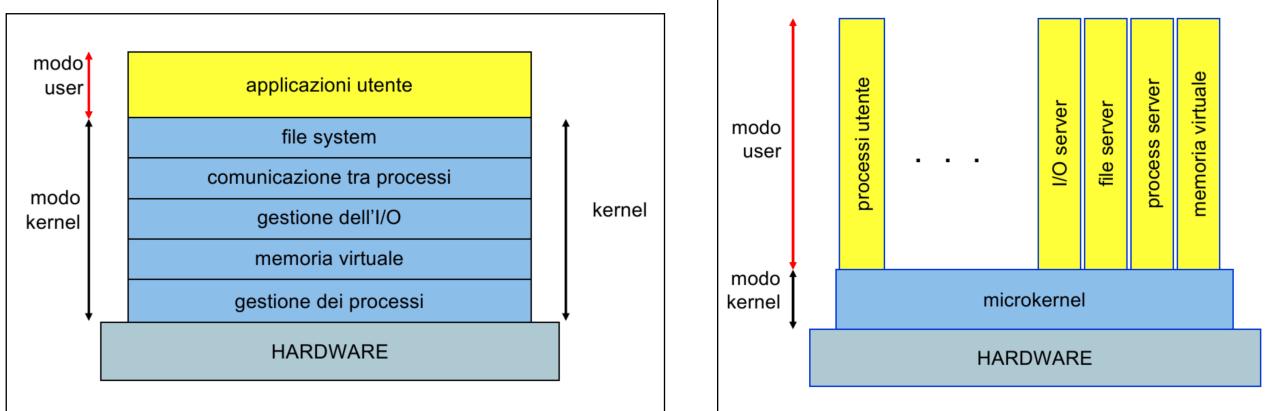


### Tipi di connessione

- **Client-server** => ruoli dispositivi ben definiti, un server ed n clients che si connettono ad esso
- **Peer-to-peer** => ogni dispositivo è sia server che client

### Architetture di base

- **Architettura a strati** => ogni strato fornisce funzionalità allo strato superiore nascondendo quelle del livello inferiore
- **Architettura microkernel** => kernel comprende solo funzioni essenziali (scheduling / gestione interruzioni / gestione memoria / comunicazione tra processi), tutte le altre funzionalità sono trattate come processi



OSS Sia UNIX che windows NT usano architetture intermedie incentrate su un kernel<sup>3</sup> monolitico che implementa gli altri moduli dinamicamente.

Unix ha architettura base a strati.

Windows ha architettura base microkernel, inoltre è multithread nativo.

### Standards

- **Standard di linguaggio (API<sup>4</sup>)** => insieme delle funzioni di libreria che rimangono invariate al variare del sistema, permettono portabilità su piattaforme differenti (ex. ANSI-C)
- **Standard di sistema (ABI<sup>5</sup>)** => insieme di servizi per la programmazione su una determinata famiglia di sistemi (POSIX per UNIX-LIKE<sup>6</sup>, WINAPI per WINDOWS)  
Essendo POSIX gestito da open-group, che vende le licenze, linux usa uno standard simile chiamato LSB (linux standard base)

Gli standard forniscono anche le funzioni preambolo per le System call.

<sup>3</sup> kernel = nucleo del sistema operativo contenente tutte le funzioni essenziali

<sup>4</sup> API = application programming interface, librerie tra app e OS

<sup>5</sup> ABI = application binary interface, convenzioni di sistema, interazione tra OS e Hardware

<sup>6</sup> Certificazione POSIX appartiene a OPEN-GROUP, linux non è certificato posix

## Flusso di controllo eccezionale

Il flusso normale di un programma può essere alterato da:

- **Interrupt** (asincrone) => eventi generati all'esterno della cpu (I/O / segnali )
- **Exception** (sincrone) => eventi generati dalla cpu (trap<sup>7</sup> / fault<sup>8</sup> / abort<sup>9</sup>)

Al verificarsi di uno di questi si utilizza il codice del sistema operativo, quindi cambia anche il modo di esecuzione, si passa da **modalità utente** (mode bit = 1) a **modalità kernel** (mode bit = 0) che permette di usare tutte le istruzioni di cui è dotato il sistema. Per sapere quale programma del kernel bisogna utilizzare il sistema operativo fa riferimento al suo “**interrupt vector**”, un array contenuto nel kernel con tutti i descriptor (linux) o handle (Windows) alle routine di sistema.

Implicano un context switch che utilizza una IRET per il ritorno. (vedere più avanti)

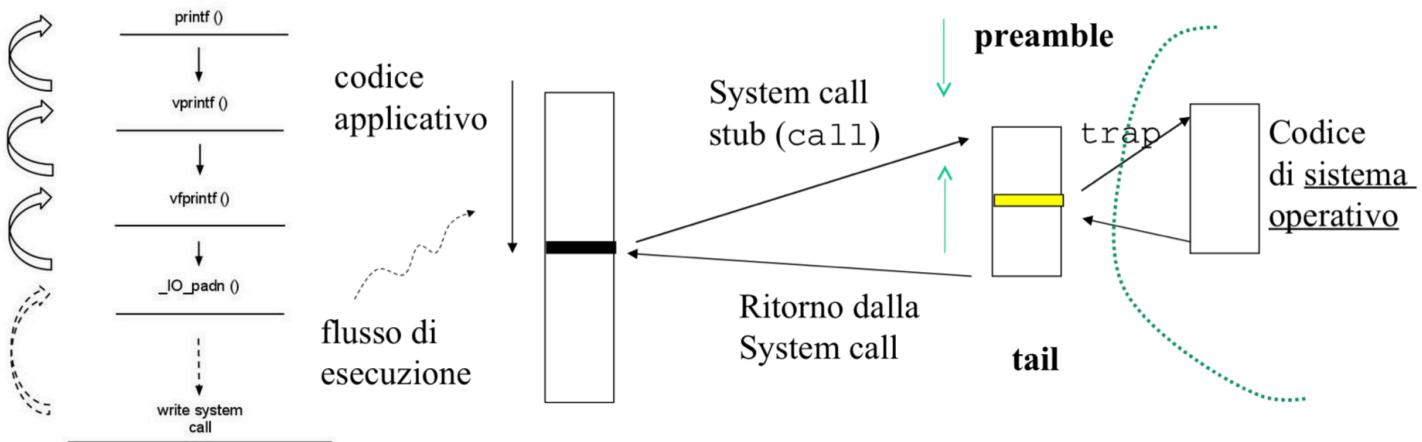
## System call

Sono chiamate ad una routine di sistema operativo tramite il supporto delle trap.

Possono essere utilizzate direttamente tramite programmazione con Assembly mediante l'istruzione “*int \$0x80*” che permette di accedere al “interrupt vector”, verrà eseguita la routine corrispondente al numero contenuto nel registro %a? utilizzando come parametri le informazioni contenute nei registri subito dopo.

Altrimenti possono essere implementate in c (o altro linguaggi di alto livello) tramite uno o più **wrapper** (involucri), ovvero funzioni che avvolgono a loro volta altre funzioni fino ad arrivare al codice Assembly per la syscall. (ex. printf())

I wrapper fanno sembrare le System call normali funzioni di libreria e combinati con l'utilizzo dell'ambiente di esecuzione permettono la portabilità del codice, la syscall in se è però machine dependent.



## Ambiente di esecuzione

Insieme dei moduli aggiuntivi (import librerie e simili) inseriti nel codice dal linker in fase di compilazione del programma, cambia da un sistema operativo all'altro.

Serve come raccordo tra il kernel ed il main del programma, permette la portabilità del codice ricompilando sul nuovo sistema compatibile.

Ex: nei sistemi linux il main richiama la funzione “`_start()`” che effettua il setup e ne permette l'esecuzione.

<sup>7</sup> trap = istruzione macchina per il passaggio del controllo al kernel (usato per system call)

<sup>8</sup> fault = evento generato in condizione di errore, recuperabile (ex. Page fault)

<sup>9</sup> abort = evento raro non recuperabile, provoca terminazione del processo (ex. Prob hardware)

## I processi

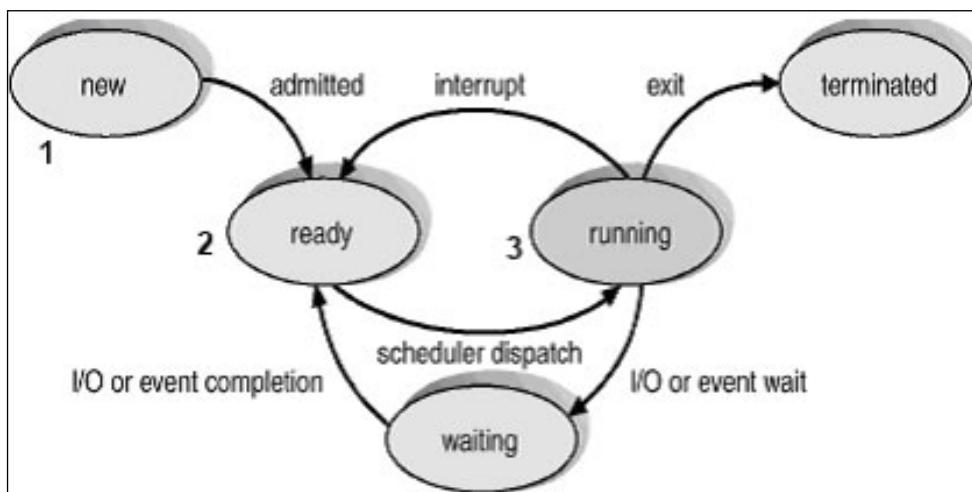
Entità attiva che rappresenta il programma in esecuzione, 2 o più processi possono condividere lo stesso programma.

I processi vengono creati da altri processi come loro copie, il processo che crea il nuovo processo viene detto padre, il processo creato figlio.

Si crea quindi una struttura ad albero dei processi (in linux il primo processo è init).

## Stati di un processo

1. **New** => fase di creazione del processo
2. **Ready** => il processo è pronto ed aspetta di essere assegnato alla CPU dallo scheduler per entrare in esecuzione
3. **Running** (USER / KERNEL) => il processo è in esecuzione nella CPU  
OSS il numero massimo dei processi in stato running è dato dal numero delle CPU
4. **Waiting** => il processo rimane in attesa di un evento di I/O
5. **Terminated** => il processo termina e rilascia le risorse utilizzate (funzione exit())



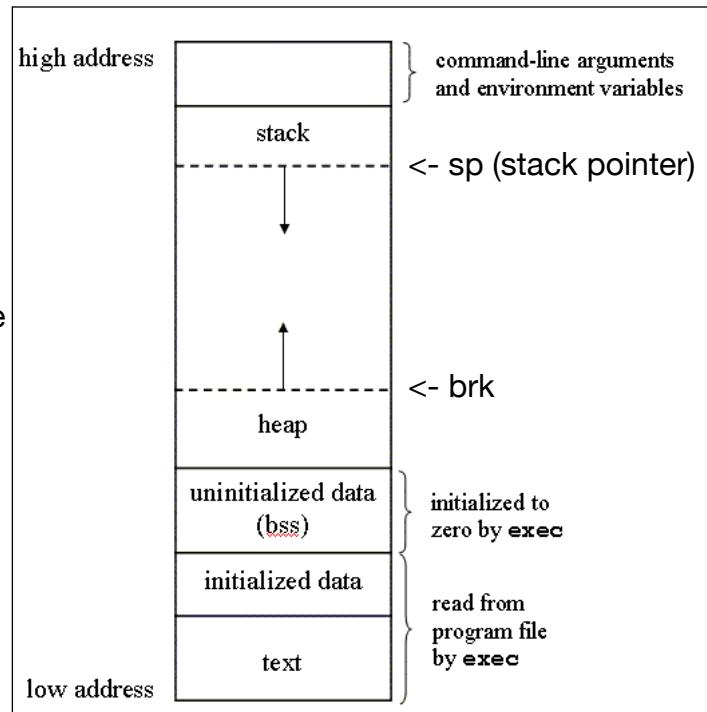
OSS Quando un processo termina, se lo stato di terminazione (dato dalla funzione exit()) non può essere raccolto dal padre, entra nello stato di **Zombie**, il suo stato di terminazione verrà quindi salvato in un record del kernel finché un altro programma lo raccoglierà e chiuderà.

## Immagine in memoria

Ad ogni nuovo processo che entra in esecuzione viene associata una immagine di memoria virtuale corrispondente al suo spazio di indirizzamento.

Lo spazio di memoria si divide in:

- **text** => codice del programma (execute and read only)
- **Data** => variabili globali, static e stringhe (stringhe read only)
- **Heap** => memoria allocata dinamicamente
- **Stack** => dati delle funzioni, ...



## PCB (process control block)

Contiene tutte le informazioni per la gestione del processo contenuta nel kernel, differente dall'immagine di memoria ma contiene riferimenti ad essa.

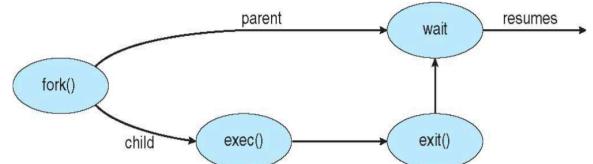
È composto da:

- PID (process ID) e PPID (parent process ID)
- Puntatori ad immagine di memoria
- Privilegi del processo
- Stato del programma
- PC (program counter)
- priorità ed informazioni sullo scheduling
- Tabella file processo ed I/O
- Area salvataggio registri

## Creazione ed esecuzione processo in unix (versione base)

```
#include <unistd.h10>
#include <stdio.h>

int main() {
    pid_t pid = fork()11;
    if (pid < 0) { /* -1 => error occurred */
        printf("Fork Failed");
        exit(1);
    }
    else if (pid == 0) { /* child process */
        execp("/bin/ls", "ls", NULL)12; /* execute program "ls" */
        _exit(0)13;
    }
    else { /* parent process */
        wait(NULL)14;
        printf ("Child Complete");
    }
    return 0; /* in main equivalente a exit() */
}
```



pid\_t getpid() => restituisce pid processo

**WARNING** se si inserisce fork() in ciclo

vfork() => figlio usa stesso Address space del padre

per generare più processi è essenziale

terminare il codice figlio con "break;" o "\_exit( status);" per evitare generazione infinita di processi (data dal "goto" alla fine del ciclo)

<sup>10</sup> <unistd.h> => libreria contenente funzioni fork(), wait(), exec()...

<sup>11</sup> pid\_t fork() => permette di creare nuovo processo come copia del padre ma con pid e immagine di memoria differenti, tutte le operazioni effettuate da adesso in avanti nella sezione del figlio non influenzano il padre (e viceversa)

<sup>12</sup> int exec => famiglia di funzioni che permettono al processo di eseguire un nuovo programma passato come argomento, possibile consultare famiglia exec su OPEN-GROUP

<sup>13</sup> void \_exit( int status) => versione con trattino permette di mantenere canali I/O aperti siccome processo padre e figlio usano gli stessi, status = stato di ritorno

<sup>14</sup> pid\_t wait( int \*stat\_loc) => costringe processo padre ad aspettare terminazione figlio evitando generazione processi zombie. Possibile salvare stato di ritorno del figlio su una variabile passandola come argomento stat\_loc.

Se processo ha più figli si usa funzione "waitpid()" per selezionare processo desiderato

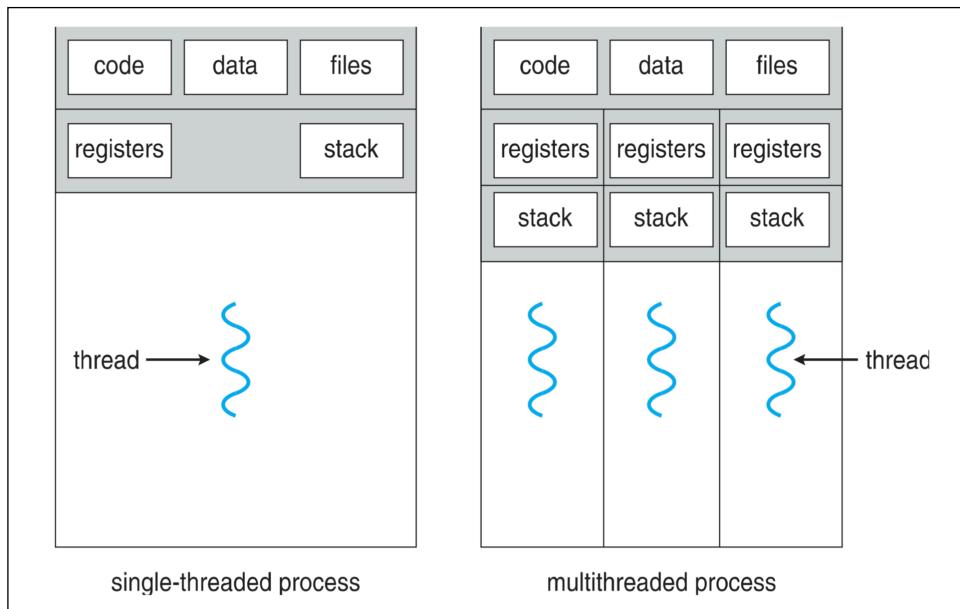
## Threads

Un thread è definito come un processo leggero (**lightweight process**).

I thread appartenenti allo stesso processo, pur eseguendo operazioni differenti, utilizzano lo stesso spazio di memoria.

Rendono quindi possibile:

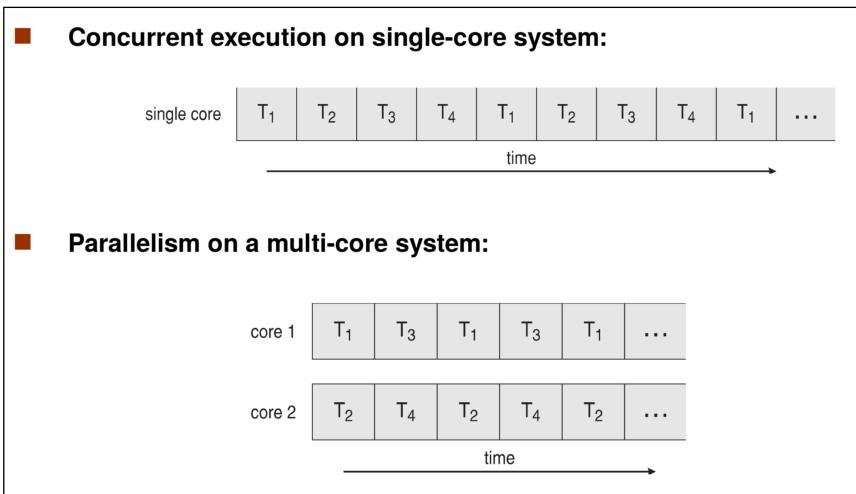
- Condivisione della memoria dentro lo stesso task<sup>15</sup>
- Multiprogrammazione e scalabilità<sup>16</sup>
- Esecuzione indipendente dei singoli thread, un thread può non essere bloccante per altro thread (Ex. grafica separata da elaborazione dati / richieste multiple su server)



## Programmazione singolcore vs multicore

La programmazione multithread permette quindi di sfruttare la concorrenza nei programmi, si può implementare in due modi a seconda del tipo di architettura :

- **Esecuzione concorrente** => si utilizza in sistemi singolo core, illusione di parallelismo data dall'esecuzione alternata dei thread in quanti di tempo (interleaved)
- **Esecuzione parallela** => utilizzabile in sistemi multicore, permette di assegnare ad ogni core uno o più threads ed eseguirli contemporaneamente



## 2 tipi di parallelismo

- **Data parallelism** => suddivisione sottoinsiemi dei dati su più core per eseguire più velocemente stesse operazioni
- **Task parallelism** => suddivisione dei threads su ogni core, ognuno esegue le sue operazioni

<sup>15</sup> task = insieme di threads appartenenti allo stesso processo

<sup>16</sup> scalabilità = esecuzione parallela su più processori (o core)

## Strategie creazione threads

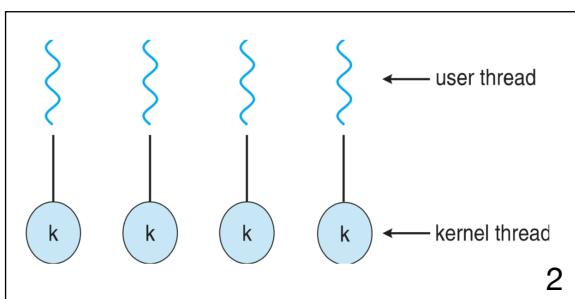
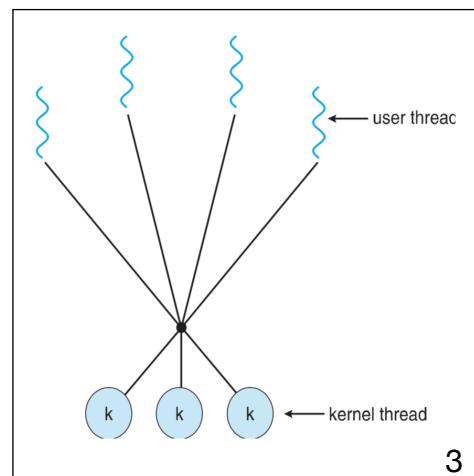
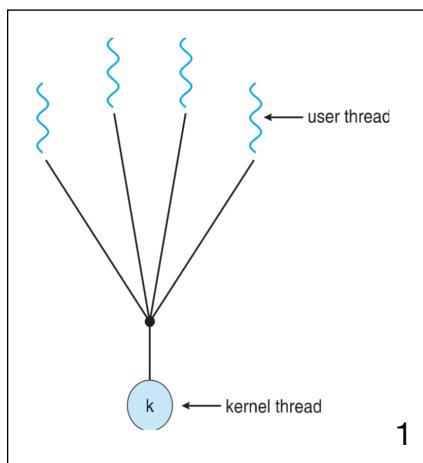
- **Threading sincrono** => (fork-join) genitore dopo aver creato uno o più figli aspetta la loro terminazione e conseguente join prima di continuare la sua esecuzione.  
(Ex. somma vettore)
- **Threading asincrono** => dopo aver creato thread figlio, genitore riprende la sua normale esecuzione e non ha bisogno di conoscere quando termina il figlio.  
(Ex. sever)

## Categorie di thread

- kernel level thread (**KLT**) => implementazione a livello kernel, scheduling più lento a causa del passaggio per il kernel ma i threads sono completamente indipendenti ed il blocco di uno non implica in blocco degli altri.
- User level thread (**ULT**) => implementazione a livello di applicazione, il kernel non è a conoscenza della loro esistenza, quindi scheduling più veloce ma nel caso di syscall bloccante su un thread si bloccano tutti.

## Modelli di multithreading

1. **da uno a molti** => più user threads mappati su un solo kernel thread, gestione efficiente risorse, non è però possibile usare parallelismo ed il blocco di uno provoca il blocco di tutti. Usato su pochi sistemi.
2. **Da uno ad uno** => ogni user thread è mappato su un kernel thread, permette parallelismo e chiamate non sono bloccanti su altri threads, ha lo svantaggio di dover creare un kernel thread per ogni user thread (threads limitati per evitare overhead<sup>17)</sup>). Usato in linux e windows.
3. **Da molti a molti** => più user threads vengono mappati su un numero minore o uguale di kernel threads, permette di creare un numero maggiore di user threads rispetto al modello uno ad uno potendo rimappare gli ULT (necessita di LWP)



<sup>17</sup> overhead = richiesta risorse maggiore di quella realmente necessaria

## Pthreads POSIX

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr ) {
    char *message;
    message = (char *) ptr;
    printf("%s \n", message);
}

main() {
    pthread_t thread1, thread2;
    const char *message1 = "Thread 1";
    const char *message2 = "Thread 2";
    int iret1, iret2; /* permettono di controllare risultato, 0 in caso di successo*/
    iret1 = pthread_create( &thread1, NULL, print_message_function, (void*) message1)18;
    if(iret1)
    {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret1);
        exit(EXIT_FAILURE);
    }
    iret2 = pthread_create( &thread2, NULL, print_message_function, (void*) message2);
    if(iret2)
    {
        fprintf(stderr,"Error - pthread_create() return code: %d\n",iret2);
        exit(EXIT_FAILURE);
    }
    printf("pthread_create() for thread 1 returns: %d\n",iret1);
    printf("pthread_create() for thread 2 returns: %d\n",iret2);
    pthread_join( thread1, NULL)19;
    pthread_join( thread2, NULL);

    exit(EXIT_SUCCESS);
}
```

funzioni ritornano come  
valore 0 in caso di  
successo, altrimenti la  
causa di errore.

I sistemi unix non fanno  
distinzione tra processi e threads.  
Infatti posso anche usare syscall  
clone() invece di pthread\_create()

int pthread\_detach( phtread\_t thread) => utilizzabile al posto  
di join, notifica che non verranno eseguite  
operazioni di join sul thread

void pthread\_exit (void\* value\_ptr) => termina thread corrente

OSS se uso funzione famiglia fork in processo multithread in molti sistemi unix posso scegliere se replicare tutti i thread nel nuovo processore oppure no, se uso exec elimina tutti i thread già presenti. DA EVITARE ENTRAMBE

<sup>18</sup> int pthread\_create( phtread\_t\* thread, const pthread\_attr\_t\* attr, void\* (\*start\_routine)(void\*), void\* arg);  
- thread => puntatore variabile pthread\_t dove memorizzare id  
- Attr => attributo di creazione (spesso NULL)  
- start\_routine => funzione da eseguire (deve essere void)  
- Arg => puntatore ad argomenti della funzione, spesso vettore allocato dinamicamente

<sup>19</sup> int pthread\_join(phtread\_t thread, void\*\* value\_ptr) => attende terminazione esplicita  
thread con id passato come argomento

## Extra threads

### **Threading implicito**

Consiste nel delegare creazione e gestione dei thread a compilatori e librerie di routine.

- **thread pools** (gruppi) => Si crea un gruppo di threads durante la creazione del processo che attendono di eseguire un determinato servizio.  
Finito il lavoro il thread torna in attesa, se non ci sono threads disponibili il servizio aspetta.
  - Non bisogna attendere creazione threads durante esecuzione processo e si evita overhead dato dalla creazione di troppi threads
  - OSS numero threads potrebbe cambiare durante esecuzione
- **OpenMP** => insieme di direttive per il compilatore che si inseriscono nel codice e permettono esecuzione parallela tramite librerie runtime openMP
  - (Ex. #pragma omp parallel => crea più thread possibili su architettura ed esegue)
- **grand central dispatch** => versione proprietaria per sistemi Apple simile a openMP e basata su pthread POSIX...

### **Thread-local storage (TLS)**

Permette di allocare variabili visibili solo al thread, uniche per ogni thread.

Differiscono da variabili locali poiché visibili in tutto il thread e non solo nella funzione in cui sono invocate.

### **Processori virtuali (LWP) e scheduling threads**

Necessari per implementazione da molti a molti dinamica, il sistema operativo crea processori virtuali intermedi che dialogano tra kernel threads e user threads.

Ad ogni LWP sono associati vari threads, al blocco di uno si bloccano anche gli altri .

Sono essenziali per lo scheduling in sistemi multithread che può avvenire in 2 modi:

- **process-contention scope** (PCS) => ambito della cortesia rispetto al processo, ovvero la mappatura su LWP libero viene fatta in ambito utente tra i threads dello stesso processo.
- **System-contention scope** (SCS) => ambito della cortesia allargato al sistema, ovvero la mappatura su LWP viene fatta dal kernel e comprende tutti i threads presenti nel sistema

OSS È possibile personalizzare le priorità di scheduling a livello utente

OSS linux e mac OS X permettono solo SCS

## Lo scheduler

Si occupa della commutazione della CPU da un processo all'altro, funzione alla base dei sistemi operativi multiprogrammati, questi permettono infatti di mantenere in memoria più processi rispetto al numero di core disponibili.

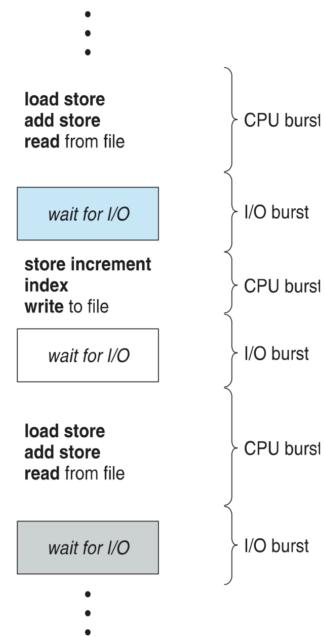
Quando un processo entra in stato di wait (I/O) lo scheduler gli sottrae il controllo della CPU e la assegna ad un altro processo in stato ready.

### Fasi cicliche di un processo

- **CPU burst** => fasi di elaborazione della CPU
- **I/O burst** => fasi di attesa dell'I/O

I programmi si possono dividere in due tipologie a seconda del tempo impiegato in ogni burst, è importante osservare la durata dei cicli per la scelta del corretto algoritmo di scheduling

- **CPU bound** => programma con prevalenza di CPU burst
- **I/O bound** => programma con prevalenza di I/O burst



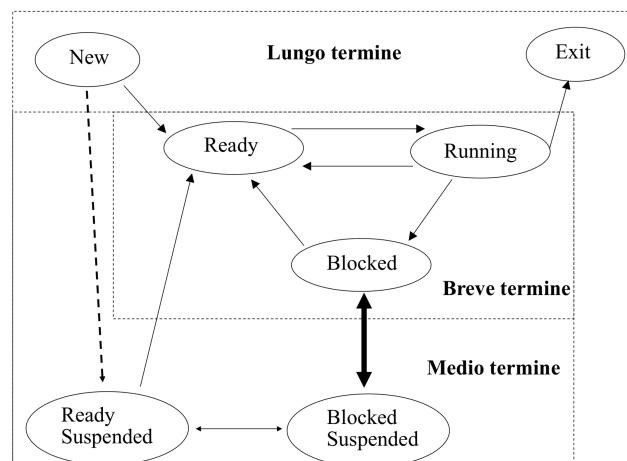
### Operazioni decisionali dello scheduler su processo

1. Passaggio da **running** a **waiting** (Ex. Richiesta I/O)
2. Passaggio da **running** a **ready** (Ex. interrupt)
3. Passaggio da **waiting** a **ready** (Ex. Fine richiesta I/O)
4. **Terminazione** processo

Gli scheduler con **prelazione**<sup>20</sup> (preemptive) possono intervenire in tutte le fasi interrompendo altri processi ma possono portare a situazioni di "**race condition**"<sup>21</sup>. Quando lo scheduler può intervenire solo nelle condizioni 1 e 4 si dice senza prelazione (nonpreemptive), questo implica che una volta entrato in esecuzione un processo rilascia la CPU solo in caso di terminazione o wait.

### Categorie di scheduling

- **A lungo termine** => prende decisioni sull'aggiunta dei processi, inutilizzabile su sistemi interattivi, veniva utilizzato in sistemi batch
- **A medio termine** => decisioni su inserimento parziale o totale dei processi attivi in memoria
- **A breve termine** => (dispatching) decide quale processo deve impegnare la CPU
- **I/O scheduling** => decisioni su sequenzializzazione richieste dei dispositivi (logici e fisici)



### Dispatcher

Modulo che permette di effettuare il context switch ed operazioni correlate sul processo scelto dallo scheduler a breve termine.

La **latenza di dispatch** è il tempo tempo richiesto per effettuare le operazioni di scambio.

<sup>20</sup> prelazione = rimozione "forzata" del processo dalla CPU data da evento esterno allo stesso

<sup>21</sup> race condition = più processi cercano di accedere alla stessa risorsa nello stesso momento

## Criteri dello scheduling

I criteri per la scelta del giusto algoritmo di scheduling sono:

- **CPU utilization** => utilizzo della CPU, deve essere impegnata il più possibile
- **Throughput** (produttività) => numero di processi completati in nell'unità di tempo
- **Turnaround time** => tempo necessario per eseguire il processo (tutti gli stati compresi)
- **Waiting time** => tempo totale passato nello stato di ready
- **Response time** => tempo tra il primo input ed il primo output

## Scheduling queues

Lo scheduler usa delle strutture del kernel (generalmente liste collegate) contenenti i PCB per gestire i vari processi attivi, le più importanti sono:

- **Job queue** => mantiene tutti i processi in esecuzione
- **Ready queue** => insieme dei processi in stato di attesa ma pronti per essere eseguiti contenuti in memoria
- **Device queue** => insieme di processi in stato di attesa di I/O

## Context switch

Si effettua ogni volta che il processore passa da un processo ad un altro, consiste nel salvare in memoria tutte le informazioni in possesso alla CPU sullo stato corrente del processo (PCB) in modo da poterlo ripristinare in seguito.

I registri vengono salvati in cima allo stack del processo, verranno ripristinati da **IRET**. Il context switch è fonte di overhead, il suo tempo di esecuzione è influenzato dal tempo per il salvataggio in memoria, ripristino della pipeline e ripopolamento della cache.

OSS Ad ogni context switch si passa sempre per la modalità kernel.

---

## Algoritmi di scheduling

### First-come, first-served (FCFS)

Basato su coda **FIFO**<sup>22</sup>, la CPU viene assegnata al primo processo che ha inserito il suo PCB nella ready queue, implementazione semplice ma non ottimale poiché non tiene conto della lunghezza dei CPU burst e I/O burst.

Svantaggioso per programmi con CPU burst brevi, inadatto sistemi interattivi.

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

■ Suppose that the processes arrive in the order:  $P_1, P_2, P_3$   
The Gantt Chart for the schedule is:

The Gantt chart illustrates the execution of three processes,  $P_1$ ,  $P_2$ , and  $P_3$ , over a timeline from 0 to 30. The chart is divided into three main segments: a long segment for  $P_1$  from 0 to 24, a short segment for  $P_2$  from 24 to 27, and a short segment for  $P_3$  from 27 to 30. The x-axis is labeled with 0, 24, 27, and 30.

- Waiting time for  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$
- Average waiting time:  $(0 + 24 + 27)/3 = 17$

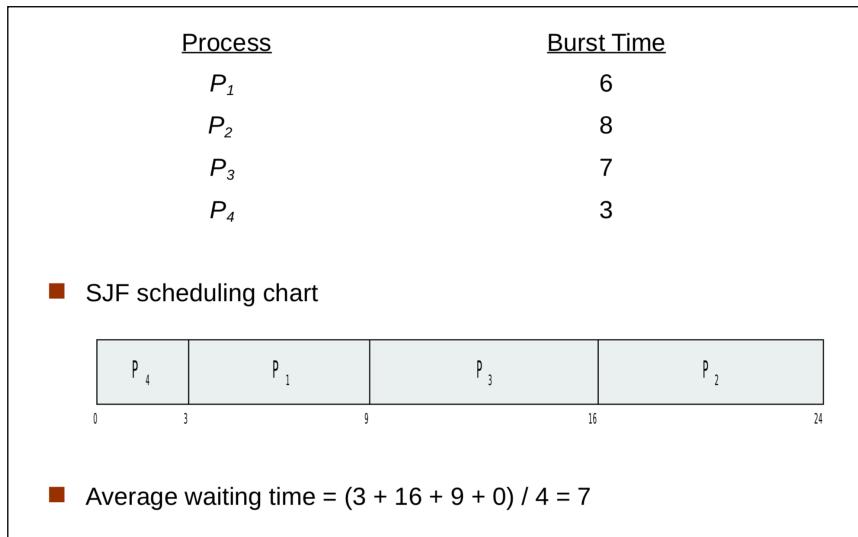
---

<sup>22</sup> FIFO = first in first out

## Shortest-job-first (SJF)

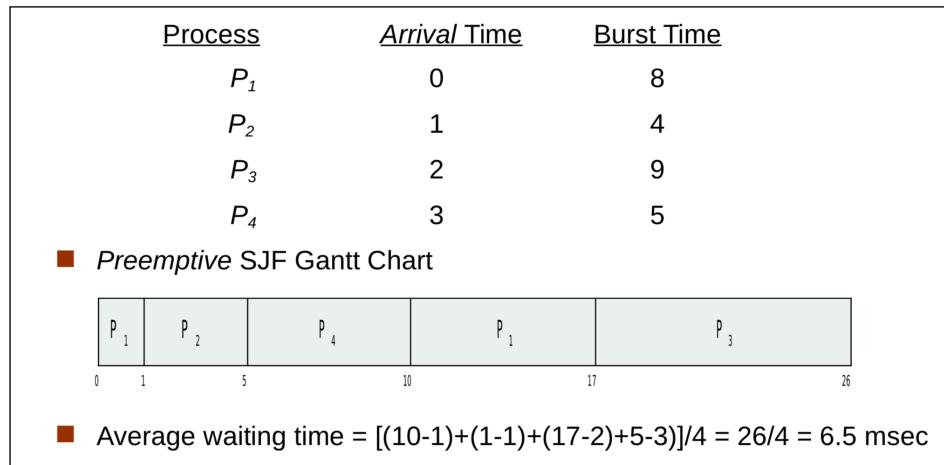
Eseguito per primo il processo con il prossimo CPU burst più veloce, è l'algoritmo ottimale per ridurre il tempo di attesa medio dei processi.

Non è realmente implementabile in scheduling a breve termine poiché non è possibile conoscere in anticipo la lunghezza dei burst, si può basare solo su previsioni date dalla media esponenziale delle tempistiche misurate in precedenza.



## Shortest-remainning-time-first (SRTF)

Versione con prelazione dello SJF, ovvero nel caso dovesse arrivare un nuovo processo con tempo di CPU burst più breve durante l'esecuzione di un processo, questi si scambierebbero.



OSS tutti gli algoritmi si possono dividere in 2 tipologie generali:

- Fairness => tutti i processi hanno le stesse priorità e vengono eseguiti con stessi tempi
- Balancing => in base alla priorità del processo

## Priority Scheduling

Ad ogni processo viene associata una priorità che può dipendere dai fattori interni al sistema o fattori esterni (ex. importanza data dai programmatori al processo).

Possono essere implementati con prelazione, possono avere problemi di “**starvation**<sup>23</sup>” risolvibili tramite “**aging**<sup>24</sup>” (SJF è un algoritmo con priorità).

Process	Burst Time	Priority
$P_1$	11	3
$P_2$	5	1
$P_3$	2	4
$P_4$	1	5

- Priority scheduling Gantt Chart

■ Average waiting time = 8.2 msec

## Round-robin (RR)

l'algoritmo di scheduling circolare è un evoluzione del FCFS a cui viene aggiunto il meccanismo di prelazione, progettato appositamente per sistemi time-sharing.

Ad ogni processo viene assegnato un **time slice** (quanto di tempo) che varia generalmente da 10 a 100 millisecondi, se durante questo periodo il processo non termina viene inserito nuovamente nella ready queue.

Questo poiché context switch generalmente non supera i 10 millisecondi e deve occupare circa 10% dello slice, quanti troppo piccoli implicano più context switch.

Process	Burst Time
$P_1$	24
$P_2$	3
$P_3$	3

- The Gantt chart is:

- Typically, higher average turnaround than SJF, but better **response**
- q should be large compared to context switch time
- q usually 10ms to 100ms, context switch < 10 usec

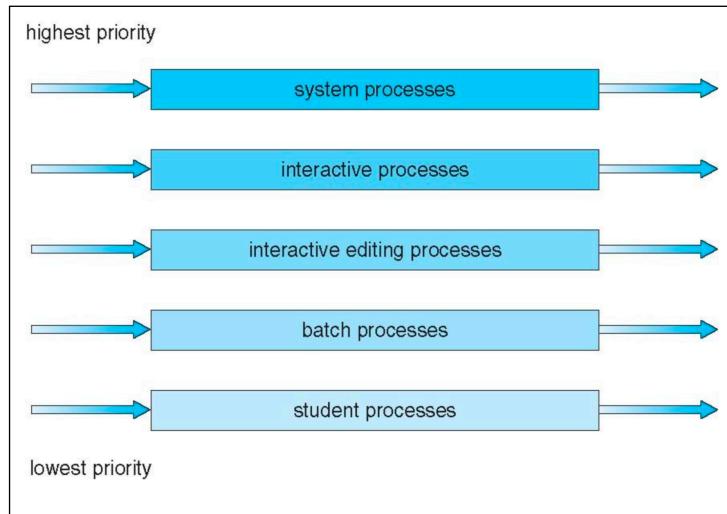
<sup>23</sup> starvation => il processo non viene mai scelto dallo scheduler e quindi non può evolvere

<sup>24</sup> aging => la priorità di un processo aumenta con il tempo

## Multilevel queue

l'algoritmo di scheduling a code multilivello divide la ready queue in più code distinte, ogni processo viene assegnato permanentemente ad una coda a seconda delle sue caratteristiche.

Ogni coda ha il suo algoritmo di scheduling che si ricollega ad un algoritmo generale per lo scheduling tra tutte le code (si può usare qualunque algoritmo).



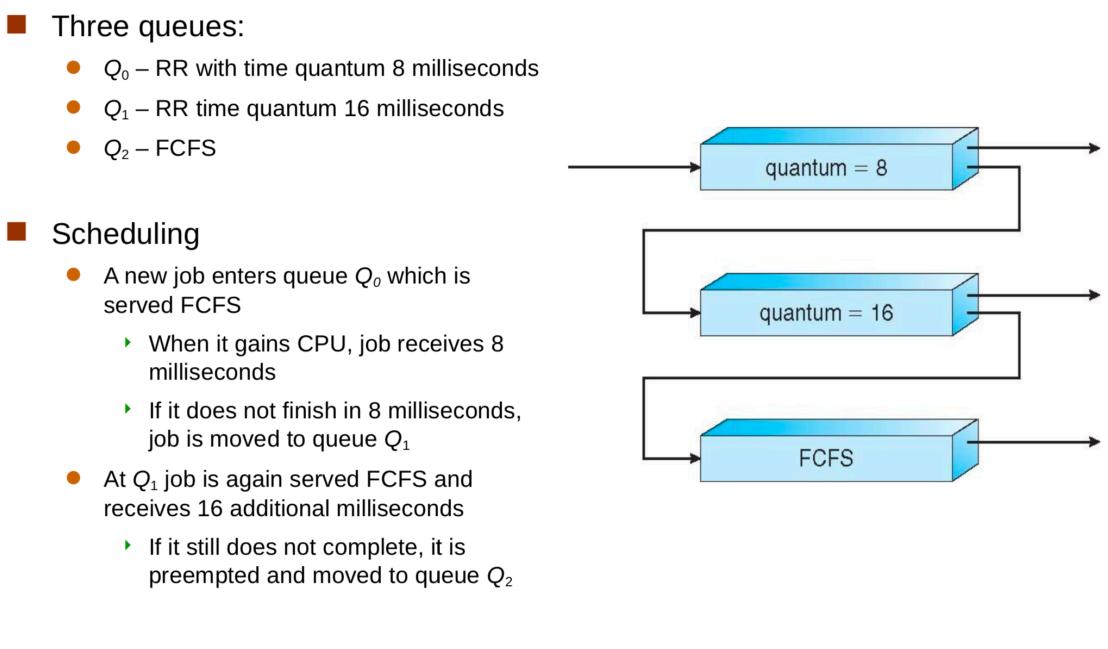
Importante distinzione tra processi foreground (primo piano / interattivi) e background (sottofondo) che si posizionano in code differenti

## Multilevel feedback queue

Lo scheduling a code multilivello con retroazione, variante di quello senza retroazione, permette inoltre ad i processi di cambiare la coda di appartenenza basandosi sul tempo di CPU burst.

Solitamente questo algoritmo predilige i processi con CPU burst brevi, se il processo non termina nel quanto di tempo assegnato la prima volta passa in una coda con quanti di tempo maggiori ma priorità più bassa.

È il sistema è il più versatile ma anche il più complesso da implementare.



## Scheduling real-time

Si possono dividere in 2 categorie:

- **Hard-real time** => richiedono completamento dei processi critici entro un intervallo di tempo predeterminato e garantito
- **Soft-real time** => viene solo assegnata priorità più alta a processi critici

I sistemi real time sono guidati dagli eventi, ovvero il sistema rimane in attesa fino al verificarsi di eventi da gestire.

Con **latenza d'evento** si indica il tempo che intercorre tra l'occorrenza dell'evento ed il momento in cui il sistema effettua la gestione, è influenzata da:

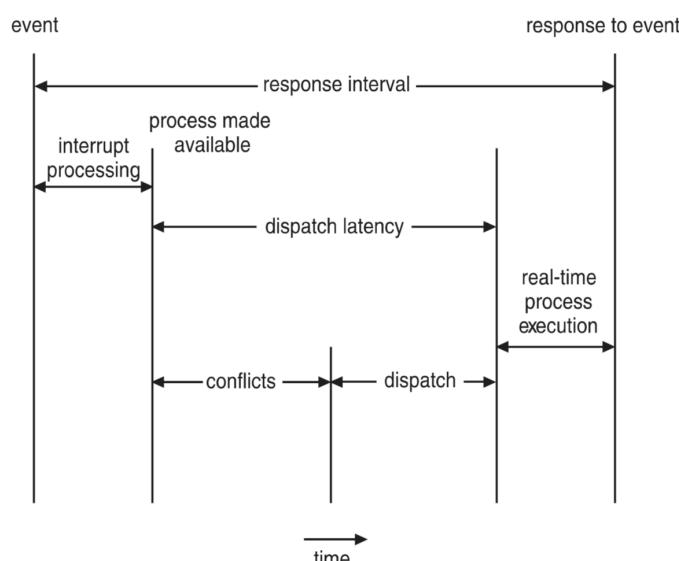
- **Latenza relativa alle interruzioni** => tempo compreso tra la notifica dell'interruzione alla CPU e l'avvio della routine che la gestisce.

Per un corretto funzionamento questi sistemi devono limitare al massimo periodi di disattivazione delle interruzioni per l'aggiornamento dei dati del kernel.

- **Latenza relativa al dispatch** => tempo necessario per passare da un processo ad un altro, per ridurne i tempi si necessita utilizzo di kernel con prelazione.

Bisogna comunque aggiungere un tempo per la “**fase di conflitto**”, formata da:

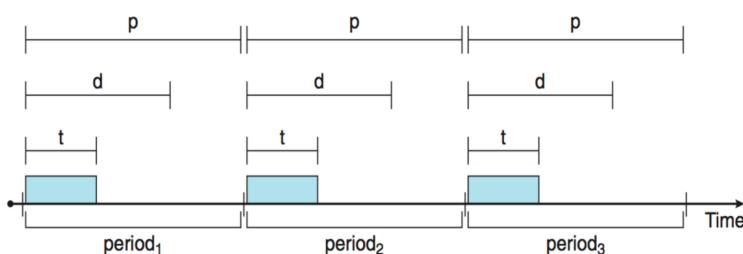
1. Prelazione di ogni altro processo nel kernel
2. Cessione delle risorse da parte dei processi con basse priorità



## Scheduling con priorità

Di base, lo scheduling per sistemi soft-real time, si può implementare usando uno scheduler che implementa un algoritmo con prelazione basato su priorità.

Nel caso di sistemi hard-real time con processi periodici può essere implementato garantendo anche che le attività in tempo reale vengano servite rispettando le scadenze.



P = periodo  
D = scadenza  
T = tempo fisso d'esecuzione

OSS Un sistema real-time si dice schedulabile, ovvero può gestire i processi senza sfornare i periodi, se rispetta:

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

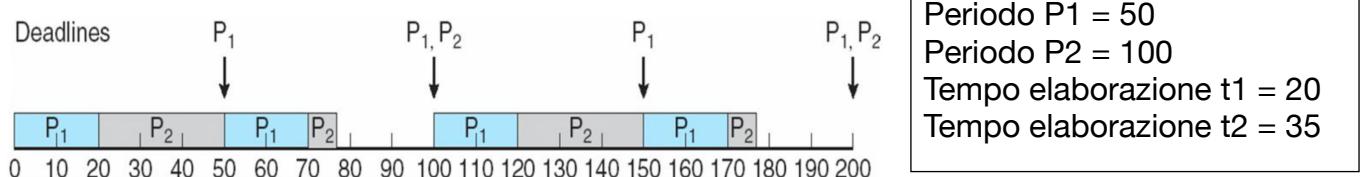
dove:

- m = numero di eventi
- $C_i$  = tempo richiesto dal processo i
- $P_i$  = periodo processo i

### Scheduling con priorità proporzionale alla frequenza

Assegna staticamente ad ogni processo una priorità inversamente proporzionale al periodo, ovvero periodo più corto implica priorità più elevata, serve per privilegiare i processi con utilizzo più frequente della CPU.

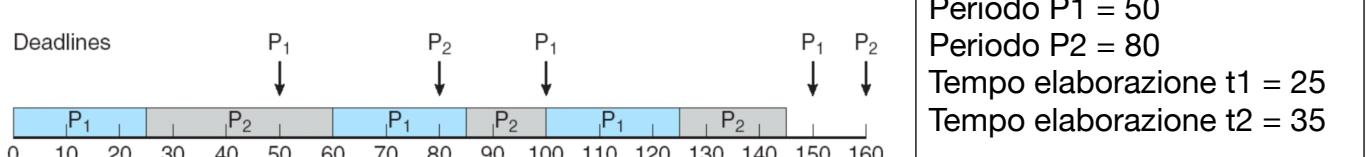
I tempi di elaborazione sono considerati fissi.



Prendendo prima il processo con il periodo più lungo si rischia di non rispettare il periodo del processo con periodo più corto.

### Earliest-Deadline-First (EDF)

Algoritmo ottimale, assegna dinamicamente la priorità ai processi con la scadenza più vicina, applicabile anche su processi non periodici e con tempo di elaborazione variabile. Ogni processo pronto, prima di poter entrare in esecuzione, deve dichiarare la sua scadenza, in modo che lo scheduler possa calcolare la priorità e modificare di conseguenza quella degli altri processi.



### Scheduling a quote proporzionali

Lo scheduler distribuisce un certo numero di quote T tra i processi, ogni processo può ricevere un certo numero di quote N dove, avendo così N/T del tempo totale di CPU.

Se un altro processo richiede quote arrivando ad un totale maggiore di T, questo viene scartato dal meccanismo di controllo dell'ammissione.

ex. T = 100;

si assegnano A = 50; B = 15; C = 20, se un processo richiede un numero di quote maggiore di 15, viene scartato.

## Scheduling sistemi multiprocessore

### Strategie di scheduling

- **Asymmetric multiprocessing** => Multielaborazione asimmetrica: le altre attività di sistema vengono assegnate tutte ad un processore, detto **master**, gli altri processori, detti **slave**, eseguono solo il codice utente
- **Symmetric multiprocessing (SMP)** => Multielaborazione simmetrica: utilizzata in quasi tutti i sistemi moderni, i processori condividono una ready queue comune o vi è una coda apposita per ogni processore, bisogna fare attenzione al problema della concorrenza ed a mantenere la **predilezione**<sup>25</sup> nella programmazione dello scheduler

OSS Nei sistemi SMP con code separate è importante tener conto del bilanciamento del carico che deve essere uniforme su tutti i processori, si effettua tramite:

- **Migrazione push** => un processo apposito controlla periodicamente stato dei processori
- **Migrazione pull** => processore inattivo prende un processo dalla coda di uno saturo Il bilanciamento va a scapito della predilezione, generalmente infatti nei sistemi reali la migrazione avviene solo se la disparità supera una certa soglia

### Processori multicore e multithread

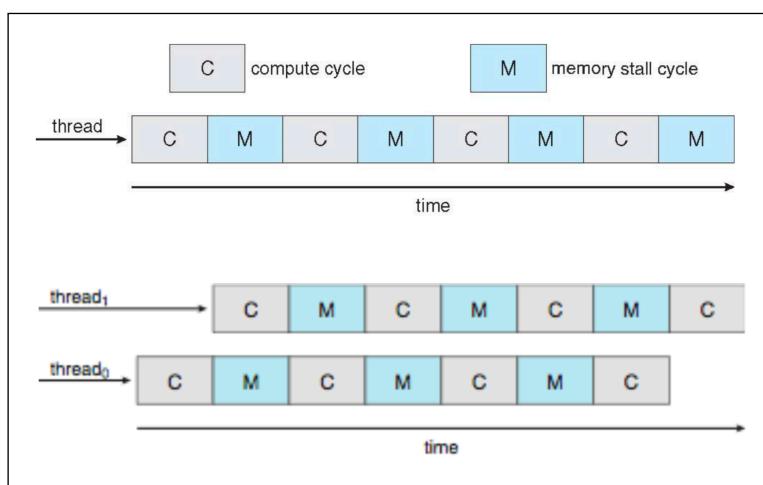
Nel caso di sistemi multicore ogni core mantiene il proprio stato ed appare al sistema operativo come fosse un processore fisico separato, questo permette ai sistemi SMP di essere più veloci e consumare meno energia.

Durante l'esecuzione di un processo possono verificarsi **stalli della memoria**, ovvero tempi morti molto lunghi in cui il processore accede alla memoria, questo ha portato alla progettazione di processori con multithread hardware che pur potendo eseguire operazioni solo in modo alternato vengono visti dal sistema come processori separati.

Possono essere di 2 tipi:

- **Multithreading a grana grossa** (coarse-grained) => switch avviene solo quando si ha evento con latenza lunga (ex. Accesso a memoria)
- **Multithreading a grana fine** (interleaved multithreading) => switch alla fine di ogni ciclo istruzione

OSS processore multicore multithread richiede 2 livelli di scheduling, generalmente il livello dei thread è gestito direttamente dall'hardware



<sup>25</sup> predilezione = (affinity) tendenza del processo a rimanere sempre sullo stesso processore, serve per evitare scambio di dati tra cache, necessari a causa dell'invalidamento dei dati

## Sincronizzazione dei processi

Serve per evitare la **race condition**, ovvero più processi paralleli che cercano di modificare gli stessi dati nello stesso momento provocando risultati casuali.

È quindi necessaria per mantenere la **coerenza dei dati condivisi**.

### La sezione critica (SC)

Con **sezione critica** si intende la parte esatta del codice di programma che accede ai dati in comune e necessita di protocolli di accesso che permettano la **mutua esclusione (ME)**, ovvero se un processo in esecuzione è entrato in SC nessun altro processo deve poter accedere alla SC corrispondente.

Nei sistemi multiprogrammati la sezione critica più importante è proprio quella relativa alle strutture dati del kernel, poiché ogni processo che entra in modalità kernel cerca di accedervi. Si possono avere quindi 2 tipi di kernel:

- **kernel con diritto di prelazione** => implementa gestione autonoma dei processi in modalità kernel, può decidere chi può usare le sue strutture dati in ogni momento, necessita di gestione della SC
- **kernel senza diritto di prelazione** => non ha il controllo sul processo già presente, questo rilascerà la modalità kernel in modo autonomo solo alla fine delle sue operazioni

```
do {  
    entry section  
    critical section  
    exit section  
    remainder section  
} while (true);
```

### Soluzione di Peterson

Soluzione software classica al problema nel caso di 2 processi, non implementabile su calcolatori moderni.

I processi condividono i dati:

- int turno;
- Boolean flag[2];

Se turn == i, il processo Pi può accedere alla SC, flag invece serve per dire se il processo è pronto ad entrare.

Protocollo:

1. il processo i per entrare imposta il suo flag su TRUE
2. assegna il turno all'altro processo j
3. finito l'accesso j imposterà il flag su FALSE conferendo il passaggio al processo i.

```
while (1) {  
    flag[i]=TRUE;  
    turno=j;  
    while (flag[j] && turno==j);  
        SEZIONE CRITICA  
    flag[i]=FALSE;  
        SEZIONE NON CRITICA  
}
```

### Lock e sincronizzazione hardware

Le soluzioni al problema della sezione critica si basano tutte su uno strumento chiamato **lock**; a livello hardware, nei sistemi monoprocessoare è sufficiente disabilitare gli interrupt, nelle architetture attuali si usano le istruzioni atomiche<sup>26</sup>.

La **test\_and\_set** e la **compare\_and\_swap**, protocollo:

1. Si dichiara una variabile globale lock, inizialmente lock = 0
2. Quando un processo entra in sezione critica imposta lock = 1
3. Gli altri processi trovando lock = 1 non possono accedere alla SC e si mettono in attesa della terminazione del processo in SC

OSS questo metodo comporta Busy waiting da parte di altri processi e **starvation**<sup>27</sup>

OSS generalmente non accessibili ai programmatori →

<sup>26</sup> istruzione atomica = non si può interrompere la sua esecuzione, una volta iniziata deve essere completata

<sup>27</sup> starvation = processo non viene mai scelto dallo scheluder e quindi non può evolvere

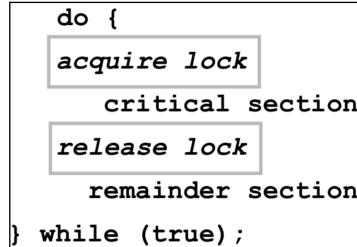
<pre> boolean test_and_set (boolean *target) {     boolean rv = *target;     *target = TRUE;     return rv; } </pre>	<pre> int compare_and_swap(int *value, int expected, int new_value) {     int temp = *value;      if (*value == expected)         *value = new_value;     return temp; } </pre>
<pre> do {     while (test_and_set(&amp;lock))         ; /* do nothing */     /* critical section */     lock = false;     /* remainder section */ } while (true); </pre>	<pre> do {     while (compare_and_swap(&amp;lock, 0, 1) != 0)         ; /* do nothing */     /* critical section */     lock = 0;     /* remainder section */ } while (true); </pre>

## Mutex lock

Soluzione più semplice per la gestione della SC utilizzabile dai programmati.

Prima di poter accedere alla SC il processo deve acquisire il possesso del mutex lock tramite la funzione **acquire()** per poi rilasciarlo alla fine delle operazioni tramite **release()**.

Solo un processo alla volta può acquisire il lock e quindi avere accesso alla SC, gli altri rimangono in attesa



```

acquire() {
    while (!available)
        ; /* busy wait */
    available = false;
}

release() {
    available = true;
}

```

## Semafori

Un semaforo è una variabile intera alla quale si può accedere (escludendo inizializzazione) solo tramite le operazioni atomiche predefinite:

- **wait()**, che decrementa il valore del semaforo
- **signal()**, che invece lo incrementa

Si può accedere alla SC solo quando il valore del semaforo è positivo ( $>0$ ).

I semafori possono essere di 2 tipi:

- **semafori binari** => valore 1 oppure 0, equivalente ai mutex lock
- **semafori contatore** => utilizzabile su risorse multiple; inizializzato con il numero di risorse disponibili, ogni volta che un processo accede ad una di esse ne decremente il valore tramite **wait()** per poi incrementarlo nuovamente con **signal()**, quando il semaforo arriva a 0 blocca tutti gli altri processi.

OSS nei semafori contatori i processi che tentano di entrare in SC senza successo non rimangono in Busy waiting ma il loro PCB viene salvato in una coda associata al semaforo così che altri processi possano essere eseguiti nel frattempo.

**block()** e **mackup()** sono le funzioni che bloccano e riattivano i processi.

<pre> wait(semaphore *S) {     S-&gt;value--;     if (S-&gt;value &lt; 0) {         add this process to S-&gt;list;         block();     } } </pre>	<pre> signal(semaphore *S) {     S-&gt;value++;     if (S-&gt;value &lt;= 0) {         remove a process P from S-&gt;list;         wakeup(P);     } } </pre>	<pre> typedef struct{     int value;     struct process *list; } semaphore; </pre>
---	--	--

## Deadlock

Il deadlock è una condizione nella quale 2 o più processi non possono evolvere poiché ognuno attende la terminazione dell'altro.  
Ex. essendo la funzione wait() bloccante per il processo, si ha che se un processo  $P_0$  invoca wait(S) e  $P_1$  invoca wait(Q), se  $P_0$  dovesse invocare wait(Q) l'attesa diventerebbe infinita.

Questo poiché il semaforo Q già posseduto da  $P_1$  che non può terminare fintanto che il semaforo S non viene rilasciato.

$P_0$	$P_1$
<b>wait(S);</b>	<b>wait(Q);</b>
<b>wait(Q);</b>	<b>wait(S);</b>
...	...
<b>signal(S);</b>	<b>signal(Q);</b>
<b>signal(Q);</b>	<b>signal(S);</b>

## ex. Produttore / consumatore

Prima implementazione base dei semafori, serve per la gestione di un buffer di n dati.  
Il semaforo binario mutex permette l'accesso alla SC ad un solo processo alla volta.  
I semafori contatori empty e full tengono conto di quanti processi stanno scrivendo o prelevando dati dal buffer.

```
Semaphore mutex initialized to the value 1
Semaphore full initialized to the value 0
Semaphore empty initialized to the value n
```

```
do {
    ...
    /* produce an item in next_produced */
    ...
    wait(empty);
    wait(mutex);
    ...
    /* add next produced to the buffer */
    ...
    signal(mutex);
    signal(full);
} while (true);
```

Produttore

```
Do {
    wait(full);
    wait(mutex);
    ...
    /* remove an item from buffer to next_consumed */
    ...
    signal(mutex);
    signal(empty);
    ...
    /* consume the item in next_consumed */
    ...
} while (true);
```

Consumatore

## Monitor

Evoluzione dei semafori, più semplice da controllare ed implementato in più linguaggi.

È un **tipo di dato astratto** (ADT) che incapsula:

- Variabili condivise
- Procedure che operano sui dati condivisi

Il costrutto monitor quindi incapsulando le variabili condivise ne rende possibile l'accesso solo tramite le sue procedure interne che implementano la mutua esclusione tramite variabili di condizione, wait() e signal()/post().

In questo modo un solo processo alla volta può avere accesso alle procedure del monitor e modificarne le variabili

```
monitor monitor-name
{
    // shared variable declarations

    procedure P1 (...){ .... }

    ...
    procedure Pn (...){.....}

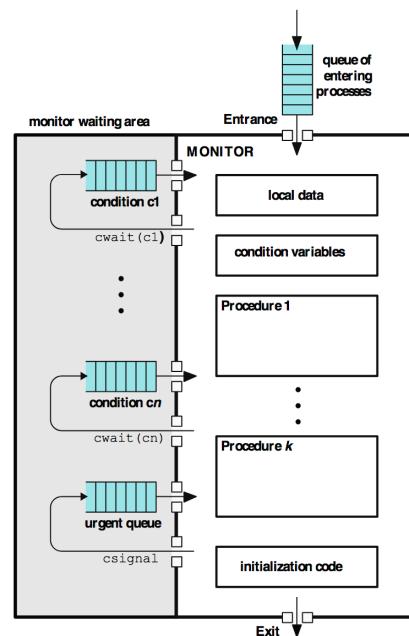
    Initialization code ( ....) { ... }

    ...
}
```

```
/* program producerconsumer */
monitor boundedbuffer;
char buffer [N];
int nextin, nextout;
int count;                                /* space for N items */
                                                /* buffer pointers */
                                                /* number of items in buffer */
cond notfull, notempty;                  /* condition variables for synchronization */

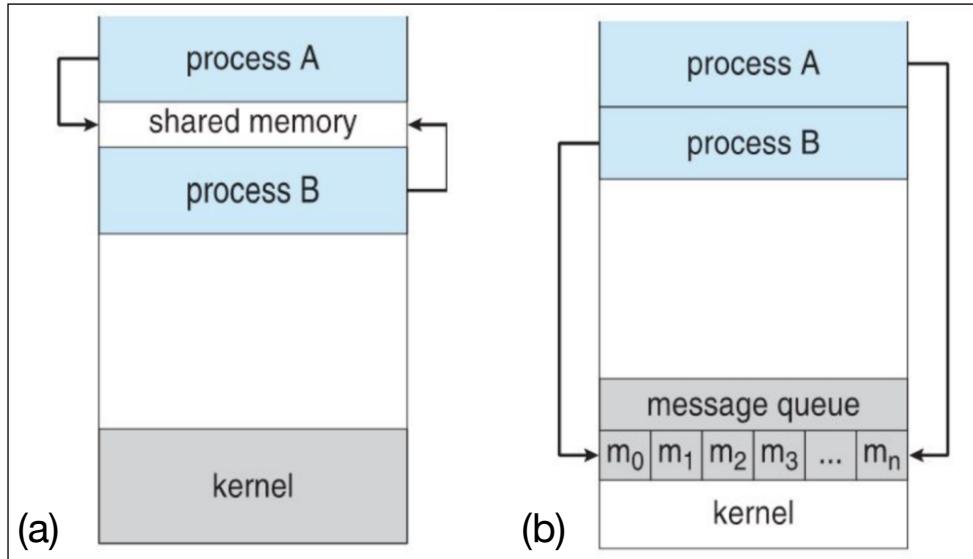
void append (char x)
{
    if (count == N) cwait(notfull);        /* buffer is full; avoid overflow */
    buffer[nextin] = x;
    nextin = (nextin + 1) % N;
    count++;
    /* one more item in buffer */
    csignal(notempty);                   /* resume any waiting consumer */
}
void take (char x)
{
    if (count == 0) cwait(notempty);      /* buffer is empty; avoid underflow */
    x = buffer[nextout];
    nextout = (nextout + 1) % N;
    count--;
    /* one fewer item in buffer */
    csignal(notfull);                   /* resume any waiting producer */
}
{
    nextin = 0; nextout = 0; count = 0;    /* monitor body */
                                         /* buffer initially empty */
}
```

```
void producer()
{
    char x;
    while (true) {
        produce(x);
        append(x);
    }
}
void consumer()
{
    char x;
    while (true) {
        take(x);
        consume(x);
    }
}
void main()
{
    parbegin (producer, consumer);
}
```



## IPC (inter process communication)

Meccanismo di comunicazione tra processi per lo scambio dei dati che permette di usare processi **cooperativi**, si implementa in 2 modi:



### **(a) shared memory**

Un processo alloca una o più zone di memoria e permette l'accesso in lettura e scrittura ad altri processi, questo permette una sola chiamata al kernel durante la creazione, tutte le altre interazioni avvengono in modalità user, richiede mutua esclusione.

Usata per processi su stesso elaboratore, può provocare problemi coerenza cache su elaboratori con molti processori o core

### **(b) Message passing:**

Permette lo scambio di messaggi tra due processi tramite un communication link (canale di comunicazione) e le primitive **send** e **receive**, non necessita la condivisione dello stesso spazio di indirizzi e non ha problemi di sincronizzazione ma implica passaggio per il kernel ad ogni interazione. (il che provoca un overhead maggiore)  
Ottimale in sistemi distribuiti o processori con più core.

I componenti necessari per implementare questo metodo sono:

- **naming** => riferimento al processo a cui indirizzare messaggi, si può avere:
  - **Comunicazione diretta** => il destinatario/mittente viene esplicitato direttamente nella funzione, semplice ma non modulare
  - **Comunicazione indiretta** => messaggi vengono scambiati tramite una porta o mailbox nel sistema alla quale accedono entrambi i processi.  
*OSS* la mailbox può anche essere associata direttamente ad un processo, in questo caso però può solo ricevere i messaggi da altri processi
- **Sincronizzazione** => definisce se chiamate sono bloccanti:
  - **Invio sincrono** => processo che invia messaggio si blocca finché non viene ricevuto
  - **Invio asincrono** => processo invia messaggio e riprende normale esecuzione
  - **Ricezione sincrona** => processo si blocca fino a ricezione messaggio
  - **Ricezione asincrona** => ricezione di messaggio durante normale esecuzione  
*OSS* è possibile implementare qualsiasi combinazione

## Possibile implementazione shared memory POSIX

```
#include <unistd.h>
#include <sys/mman.h28>

#define MAX_LEN 10000

struct region {      /* Defines "structure" of shared memory */
    int len;
    char buf[MAX_LEN];
};

struct region *rptr;
int fd;

/* open or create shared memory object and set its size */
fd = shm_open("/myregion", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR)29;

if (fd == -1)      /* Handle error */;

/* set object size (function of unistd.h)*/
if (ftruncate(fd, sizeof(struct region)) == -1) /* Handle error */;

/* Map shared memory object */
rptr = mmap(NULL, sizeof(struct region), PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0)30;

if (rptr == MAP_FAILED)      /* Handle error */;

/* Now we can refer to mapped region using fields of rptr */
```

Vedere dispense programmazione per altre implementazioni e message passing

int shm\_unlink(const char \*name) => rimuove segmento di memoria condivisa dal /dev/shm ma non dealloca spazio mappato in processo e filedescriptor, bisogna usare munmap e close

<sup>28</sup> <sys/mman.h> = contiene syscall per creazione e gestione zone shared memory

<sup>29</sup> int shm\_open(const char \*name, int oflag, mode\_t mode) => apre zona di memoria condivisa, ritorna file descriptor che si riferisce alla zona, -1 in caso di errore  
- name = nome univoco della zona nel filesystem visibile ad i processi (in /dev/shm)  
- oflag = O\_RDWR specifica apertura in lettura scrittura, O\_CREAT se la zona con nome name non è già presente ne crea una... posso utilizzare tutti i flag di open()  
- mode = attributi di creazione per O\_CREAT

<sup>30</sup> void \*mmap(void \*addr, size\_t len, int prot, int flags, int fd, off\_t off) => consente di mappare il filedescriptor (qui segmento di memoria condivisa) nella memoria del processo, ritorna indirizzo in caso di successo, MAP\_FAILED in caso di errore  
- addr =  
- len = grandezza blocco da mappare  
- prot = permessi di accesso alla zona  
- flags = informazioni su handling zona (ex. MAP\_SHARED => cambiamenti condivisi, MAP\_PRIVATE => cambiamenti privati)  
- fd = filedescriptor da mappare  
- off = offset

## Deadlock

Ogni sistema è composto da un numero limitato di risorse che possono essere richieste ed in caso utilizzate dai processi, il deadlock è una condizione di stallo nella quale 2 o più processi non possono evolvere poiché ognuno attende il rilascio delle risorse da parte di un altro.

Condizioni necessarie per deadlock:

1. **Mutua esclusione** => almeno una delle risorse condivise dai processi deve essere non condivisibile e quindi richiedere la mutua esclusione
2. **Possesso e attesa** => un processo deve possedere una risorsa ed allo stesso tempo richiederne un'altra
3. **Assenza di prelazione** => le risorse non possono essere momentaneamente rilasciate
4. **Attesa circolare** =>  $P_0$  attende  $P_1$ ,  $P_1$  attende  $P_2$ , ...

## Grafo di assegnazione

Rappresenta situazioni di stallo, formato da vertici  $V$  e archi  $E$ , con  $V$  diviso in:

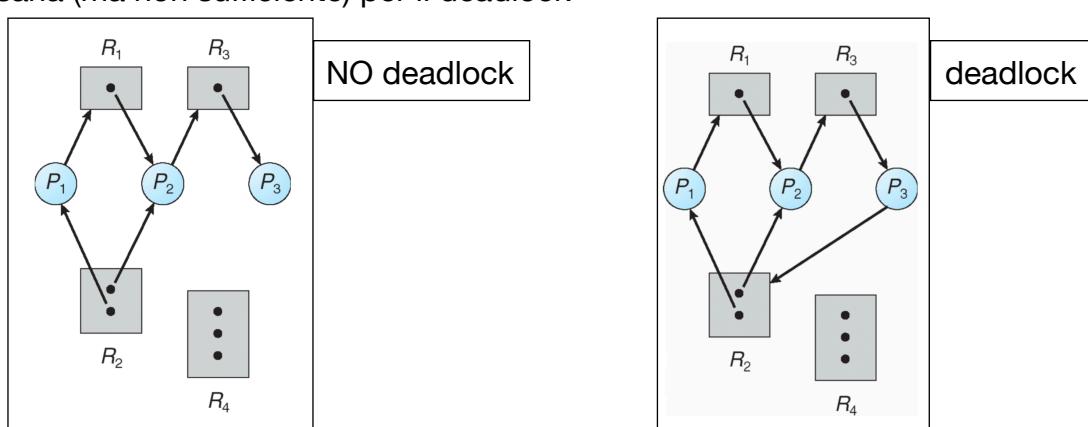
- $P = \{P_0, P_1, \dots, P_n\}$  => tutti i processi del sistema
- $R = \{R_0, R_1, \dots, R_n\}$  => tutte le risorse del sistema

E invece è composto da:

$P_i \rightarrow R_j$  indica che il processo  $i$  ha richiesto utilizzo risorsa  $j$ , **arco di richiesta**

$R_j \rightarrow P_i$  indica che la risorsa  $j$  è stata assegnata al processo  $i$ , **arco di assegnazione**

Il grafo permette di controllare immediatamente la presenza di cicli che sono condizione necessaria (ma non sufficiente) per il deadlock



## Metodi gestione stallo

Il problema del deadlock si può affrontare a livello di sistema in 3 modi:

- Si usa protocollo che previene oppure evita situazioni di deadlock:
  - Prevenire stalli => si evita a monte il verificarsi di almeno una delle 4 condizioni necessarie per il deadlock
  - Evitare stalli => si forniscono al sistema le informazioni sulle risorse richieste dai processi in anticipo, in modo che possa decidere quali soddisfare e quali devono invece rimanere in attesa.
- Si permette al sistema di entrare in stallo, individuarlo, e quindi eseguire ripristino tramite specifici algoritmi evitando il degrado delle prestazioni
- Si ignora il problema, i processi possono entrare in stallo e provocare un degrado delle prestazioni se non se ne occupa il programmatore applicativo.

Utilizzato dalla maggior parte dei sistemi operativi, compresi Linux e Windows poiché l'implementazione è più economica e flessibile

## Prevenire stalli

Per evitare il verificarsi di una delle 4 condizioni:

1. Mutua esclusione => in caso di risorse non condivisibili deve valere condizione di mutua esclusione, non necessaria invece per risorse condivisibili sulle quali possono lavorare entrambi i processi (ex. File aperto in sola lettura).
2. Possesso e attesa => si permette solo a processi non aventi risorse in possesso di richiederne altre, i processi sono quindi obbligati ad esplicitare tutte le risorse di cui necessitano per il completamento.
3. Assenza di prelazione => se un processo richiede una risorsa che non gli si può assegnare allora rilascia tutte le altre in suo possesso fino a quando non potrà avere tutte le risorse necessarie, altrimenti il processo impone il rilascio delle risorse agli altri processi per poi rilasciare tutto una volta terminato.
4. Attesa circolare => si impone un ordinamento totale all'insieme di tutti i tipi di risorse e si impone che ciascun processo le richieda in ordine crescente.

## Evitare stalli

Le tecniche per evitare stalli si basano su una conoscenza approfondita da parte del sistema delle risorse necessarie al processo durante tutto il suo ciclo di vita, evitando possibili sprechi di risorse come può accadere nei metodi di prevenzione.

Ogni algoritmo differisce per la quantità ed il tipo di informazioni richieste, la versione più semplice richiede al processo solo il numero massimo per ogni tipo di risorsa che utilizza, garantisce quindi l'assenza di stalli confrontando tutte le risorse e le richieste presenti.

Un sistema si dice in **stato sicuro** se è in grado di assegnare risorse a ciascun processo in un certo ordine impedendo situazioni di stallo, ovvero esiste una sequenza sicura.

Una sequenza di processi  $\langle P_0, P_1, \dots, P_n \rangle$  è una **sequenza sicura** per lo stato di assegnazione attuale se, per ogni  $P_i$ , le richieste che  $P_i$  può ancora fare si possono soddisfare impiegando le risorse attualmente disponibili più le risorse possedute da tutti i processi  $P_j$  con  $j \leq i$ .

È quindi possibile evitare stalli mantenendo sempre il sistema in stato sicuro.

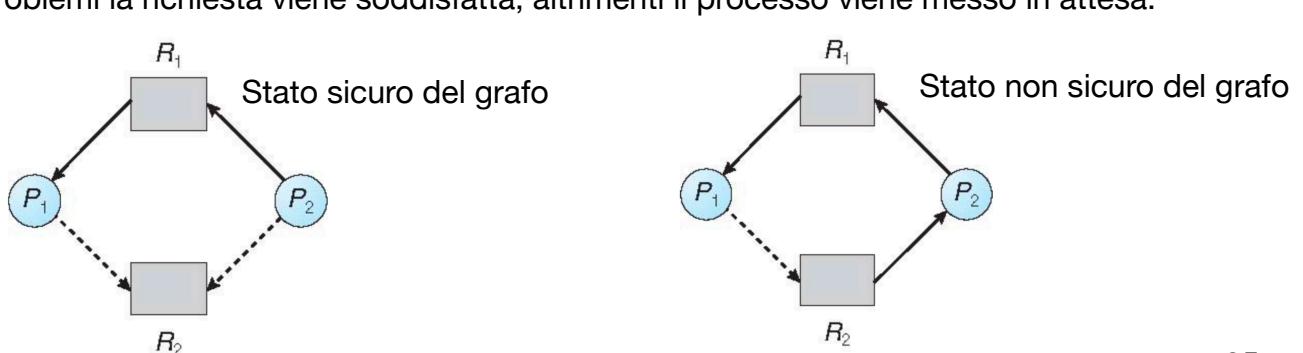
## Algoritmo con grafo di assegnazione risorse

Utilizzabile solo quando si ha una sola risorsa per ogni tipo.

Basato sui grafi di assegnazione, oltre agli archi di richiesta e assegnazione introduce gli **archi di rivendicazione** che indicano quali risorse può richiedere il processo in futuro (come arco di richiesta  $P_i \rightarrow R_j$  ma tratteggiato).

Quando un processo entra in esecuzione esplicita tutti i suoi archi di rivendicazione che ad ogni richiesta di una risorsa diventano archi di richiesta.

L'algoritmo (costo  $n^2$  con  $n = \text{numero processi}$ ) ad ogni richiesta controlla se questa provoca la formazione di cicli e conseguente passaggio a stato non sicuro, se non ci sono problemi la richiesta viene soddisfatta, altrimenti il processo viene messo in attesa.



## Algoritmo del banchiere

Utilizzabile con più risorse dello stesso tipo.

Ogni nuovo processo per entrare in esecuzione deve esplicitare il numero massimo di risorse di cui necessita per ogni tipo di risorsa in modo che il sistema possa verificare che si mantenga lo stato sicuro in caso di assegnazione.

Dati  $n$  processi ed  $m$  tipi di risorsa, l'algoritmo necessita delle seguenti strutture dati:

- **Disponibili** => vettore lunghezza  $m$  che indica numero di istanze disponibili;  
disponibili $[j] = k \Rightarrow k$  indica il numero di istanze disponibili per il tipo di risorsa  $R_j$
- **Massimo** => matrice  $n \times m$  che definisce la richiesta massima di ogni processo;  
massimo $[i,j] = k \Rightarrow k$  indica il numero massimo di  $R_j$  che può richiedere  $P_i$
- **Assegnate** => matrice  $n \times m$  che definisce numero di istanze di ogni tipo di risorsa già assegnate ad ogni processo;  
assegnate $[i,j] = k \Rightarrow k$  indica il numero di istanze di  $R_j$  già assegnate a  $P_i$
- **Necessità** => matrice  $n \times m$  che indica risorse ancora necessarie ad ogni processo;  
necessità $[i,j] = k \Rightarrow k$  indica il numero di istanze di  $R_j$  mancanti a  $P_i$   
OSS necessità $[i,j] = \text{massimo}[i,j] - \text{assegnate}[i,j]$

Dati due vettori  $X$  e  $Y$  (lunghezza  $n$ ), si dice che  $X \leq Y$  se  $X[i] \leq Y[i]$  per ogni  $i = 1, \dots, n$

Inoltre  $X < Y$  se vale anche la condizione  $X \neq Y$

### I. Algoritmo di verifica della sicurezza

Serve per determinare se il sistema è in uno stato sicuro controllando le risorse disponibili per ogni tipo, una ad una.

1. Siano Lavoro e fine vettori di lunghezza  $m$  e  $n$ , con:
  - lavoro = disponibili
  - fine $[i] = \text{falso}$ , per  $i = 1, 2, \dots, n-1$
2. Si cerca un indice  $i$  per cui valgono le relazioni:
  - fine $[i] == \text{falso}$
  - necessità $_i \leq \text{lavoro}$
3. lavoro = lavoro + assegnate $_i$   
fine $[i] = \text{vero}$   
goto II
4. If (fine $[i] == \text{vero}$ ) per ogni  $i \Rightarrow$  sistema è al sicuro

notazione: ogni riga delle matrici assegnate e necessità si indica con assegnate $_i$  e necessità $_i$

Il costo asintotico dell'algoritmo è  $m \times n^2$

### II. Algoritmo di richiesta delle risorse

Serve per determinare se le richieste possono essere soddisfatte.

Dato richieste $_i$ , vettore delle richieste di  $P_i$ .

Si ha che richieste $[j] == k$ , indica il numero di istanze della risorsa tipo  $R_j$  richieste da  $P_i$ .

Se  $P_i$  effettua una richiesta si effettuano le seguenti azioni:

1. If ( $\text{richieste}_i \leq \text{necessita}_i$ ) goto 2;  
else condizione di errore, superato numero massimo di richieste
2. If ( $\text{richieste}_i \leq \text{disponibili}_i$ ) goto 3;  
else  $P_i$  deve aspettare risorse diventino disponibili
3. Si simula l'assegnazione delle risorse modificando le strutture dati:  
$$\text{disponibili} = \text{disponibili} - \text{richieste}_i$$
$$\text{assegnate}_i = \text{assegnate}_i + \text{richieste}_i$$
$$\text{necessita}_i = \text{necessita}_i - \text{richieste}_i$$
4. If (simulazione ritorna stato sicuro) vengono assegnate le risorse al processo  $P_i$   
else il processo  $P_i$  deve aspettare, si ripristina assegnazione risorse precedente

## Rilevamento e ripristino situazioni di stallo

### Rilevamento

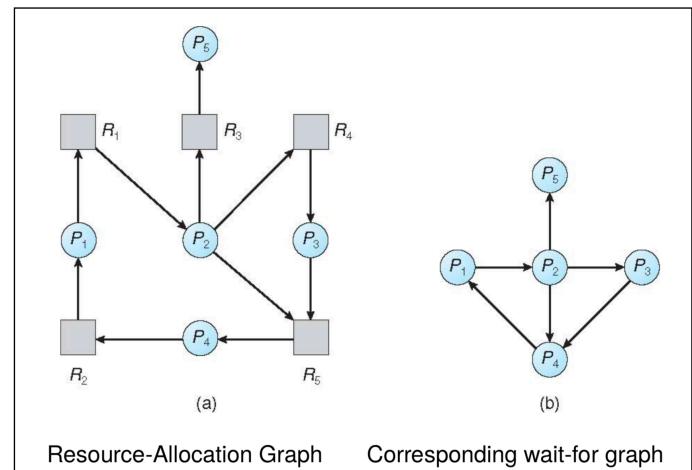
#### I. Istanza singola per ogni tipo di risorsa

Si effettua tramite una variante del grafo di assegnazione detta grafo d'attesa.

Connessione diretta tra i processi, ad ogni arco d'attesa tra i processi  $P_i \rightarrow P_j$

corrispondono 2 archi  $P_i \rightarrow R_q$  e  $R_q \rightarrow P_j$ , questo infatti indica che il processo  $P_i$  attende la terminazione di  $P_j$  per essere eseguito.

Anche in questo caso i cicli indicano la presenza di stallo.



#### II. Più istanze per ogni tipo di risorsa

Simile ad algoritmo del banchiere, dati  $n$  processi ed  $m$  tipi di risorsa, l'algoritmo necessita delle seguenti strutture dati:

- **Disponibili** => vettore lunghezza  $m$  che indica numero di istanze disponibili;  
disponibili[j] = k => k indica il numero di istanze disponibili per il tipo di risorsa  $R_j$
- **Assegnate** => matrice  $n \times m$  che definisce numero di istanze di ogni tipo di risorsa già assegnate ad ogni processo;  
assegnate[i,j] = k => k indica il numero di istanze di  $R_j$  già assegnate a  $P_i$
- **Richieste** => matrice  $n \times m$  che indica richiesta attuale di ogni processo;  
necessità[i,j] = k => k indica il numero di istanze di  $R_j$  che sta richiedendo  $P_i$   
OSS necessità[i,j] = massimo[i,j] - assegnate[i,j]

Algoritmo di rilevamento:

1. Siano Lavoro e fine vettori di lunghezza  $m$  e  $n$ , con:
  - lavoro = disponibili
  - For (  $i=0$ ;  $i < n$ ;  $i++$  )
    - If (assegnate<sub>i</sub> != 0) fine[i] = falso
    - else fine[i] = true
2. Si cerca un indice  $i$  per cui valgono le relazioni:
  - fine[i] == falso
  - richieste<sub>i</sub>  $\leq$  lavoro
  - IF (non esiste nessun  $i$ ) goto 4;
3. lavoro = lavoro + assegnate<sub>i</sub>
  - fine[i] = vero
  - goto 1
4. If (fine[i] == falso) per qualche  $i$  => sistema è in stallo, inoltre è proprio  $i$  in stallo

Il costo asintotico dell'algoritmo è  $m \times n^2$

L'utilizzo dell'algoritmo per la rivelazione degli stalli comporta un notevole aumento del carico computazionale, quindi, anche se si potrebbe usare ogni volta che un processo effettua una richiesta che non si può soddisfare immediatamente, solitamente l'algoritmo viene utilizzato ad intervalli definiti oppure quando il carico computazionale scende sotto una determinata soglia.

## Ripristino

Si può effettuare in 2 modi differenti:

### I. Terminazione dei processi

Si possono usare 2 metodi:

- Terminazione di tutti i processi in stallo => operazione onerosa, inoltre non potendo salvare lo stato dei processi i risultati dovranno essere scartati e ricalcolati
- Terminazione di un processo alla volta fino all'eliminazione dello stallo => comporta overhead notevole dovendo richiamare l'algoritmo di rilevazione dopo ogni terminazione

### II. Prelazione delle risorse

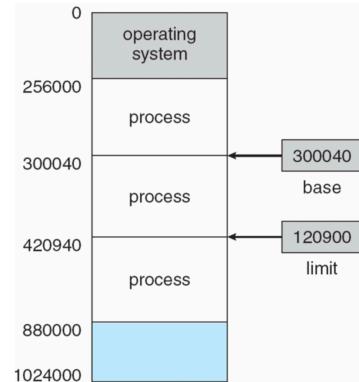
Si prelevano risorse ai processi, considerando i seguenti problemi:

1. Selezione vittima => bisogna stabilire un ordine con il quale togliere determinate risorse a determinati processi allo scopo di minimizzare il costo dell'operazione
2. Ristabilimento stato precedente => prelevando una risorsa si altera lo stato del processo che deve quindi essere ricondotto ad uno stato sicuro precedente (rollback), oppure più semplicemente si esegue un riavvio dello stesso
3. starvation => non si devono verificare situazioni di attesa indefinita su un processo, ovvero bisogna variare i processi ai quali si sottraggono risorse

## La memoria centrale

A livello logico la memoria è come un grande vettore di byte, ognuno con il suo indirizzo, usati tramite le primitive load e store. Ogni processo possiede un **registro base** ed un **registro limite** che permettono di dividere la memoria in intervalli.

Ad ogni accesso in memoria da parte di un processo la cpu confronta l'indirizzo generato con i valori contenuti nei 2 registri, effettuabile solo dal kernel in modo da mantenere sicurezza.



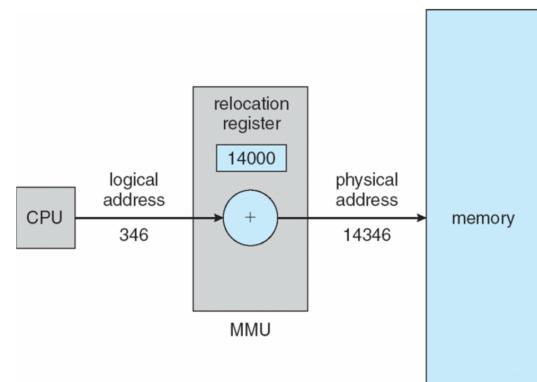
## Associazione indirizzi

I programmi sono salvati sul disco come file binari eseguibili da caricare in memoria durante l'esecuzione, l'insieme dei processi in attesa di esecuzione è detta **input queue**. In fase di compilazione non è possibile conoscere in quale porzione di memoria verrà salvato il programma, viene quindi generato codice assoluto basato su un indirizzo di partenza chiamato indirizzo di rilocazione.

Quando il programma viene lanciato il loader effettua una fase di linking che permette di associare (bind) gli indirizzi simbolici (ex.  $x = &x + 4$ ) del codice sorgente con gli indirizzi logici corrispondenti nell'immagine di memoria.

Durante la fase di esecuzione invece tutti gli indirizzi logici del programma vengono mappati dalla **MMU**<sup>31</sup> sugli indirizzi fisici, visibili solo dalla memoria ed in comune per ogni programma che risiede in essa.

Nella versione più semplice l'associazione avviene sommando l'indirizzo logico al registro di rilocazione ed effettuando un controllo per verificare che l'indirizzo fisico non vada oltre il registro limite.



OSS Per ottimizzare l'utilizzo della memoria è possibile utilizzare il **dynamic loading** (caricamento dinamico) che permette di caricare solo porzioni del programma in memoria durante l'esecuzione, quando necessarie. Il linking dinamico si utilizza anche per sfruttare le librerie di sistema delle quali si salva solo un riferimento e non tutto il codice.

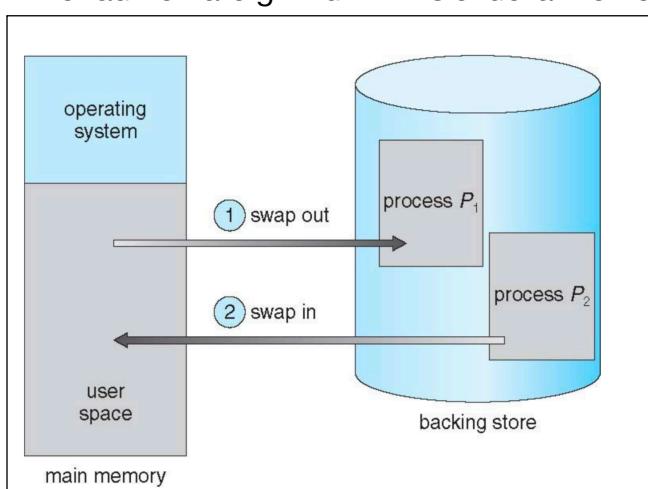
## Swapping (avvicendamento processi)

Per aumentare gli indirizzi fisici della memoria è possibile creare aree di swapping sul disco nelle quali spostare temporaneamente processi momentaneamente inattivi.

Quando il sistema schedula un processo dalla ready queue, controlla prima se questo è presente nella memoria principale, altrimenti lo carica dall'area di swap.

Sa la memoria è piena esegue uno **swap out** di un processo già presente ed uno **swap in** del processo richiesto, questo però aumenta di molto il tempo di context switch.

Lo swapping non è presente su dispositivi mobili, i quali chiudono direttamente altri processi attivi.



<sup>31</sup> MMU = memory-management unit, componente hardware per la mappatura degli indirizzi

## Allocazione contigua della memoria

La prima parte della memoria fisica è riservata per il sistema operativo, il resto viene lasciato ai processi, ognuno dei quali possiede una sezione contigua di memoria.

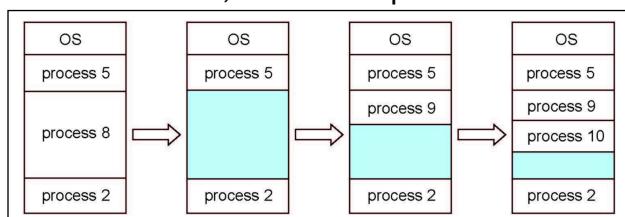
L'utilizzo degli indirizzi di rilocazione permette al sistema operativo di cambiare le sue dimensioni semplicemente alterando il loro valore.

Esistono diversi modi per effettuare l'allocazione:

- **Schema a partizioni fisse** => implementazione più semplice (non più in uso), consiste nel dividere la memoria in sezioni di dimensione fissa ad ognuna delle quali si associa un processo, limita multiprogrammazione.
- **Schema a partizione variabile** => la grandezza della sezione di memoria viene calcolata al momento di avvio del processo aumentando la flessibilità, inoltre il SO mantiene una tabella con posizione e grandezza degli spazi vuoti.

Avendo che processi diversi necessitano di una quantità diversa di memoria nasce il **problema di allocazione dinamica della memoria**, ovvero in quale buco libero allocare i dati dei nuovi processi, si possono usare 3 diversi criteri:

- **First-fit** => il primo spazio libero abbastanza grande
- **Best-fit** => il più piccolo buco abbastanza grande
- **Worst-fit** => il più grande tra quelli disponibili



OSS si crea inoltre il problema della **frammentazione** che implica un utilizzo inefficiente dello spazio libero

## La frammentazione

Ne esistono 2 tipi:

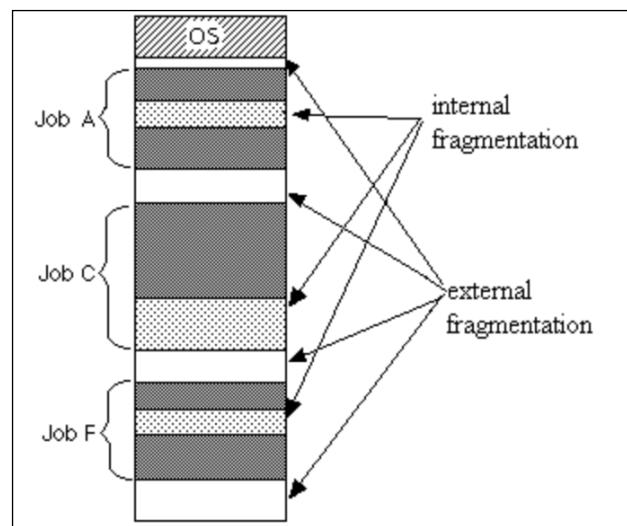
- **Frammentazione esterna** => Ne soffrono first-fit e best-fit, si verifica quando, pur avendo spazio sufficiente in memoria, non è possibile allocare dati poiché le sezioni libere non sono contigue.

Infatti caricando e rimuovendo dalla memoria dati di diverse dimensioni, si possono creare tanti piccoli buchi frammentati.

Una soluzione a questo problema è data dalla compattazione che permette di spostare le locazioni di memoria utilizzando solamente l'indirizzo di rilocazione

- **Frammentazione interna** => si verifica quando la memoria allocata è superiore a quella realmente necessaria a causata dall'allineamento dei dati.

L'allineamento è utile poiché permette di ridurre le strutture necessarie per la memorizzazione di posizioni e grandezze, inoltre semplifica molte operazioni.



## Segmentazione

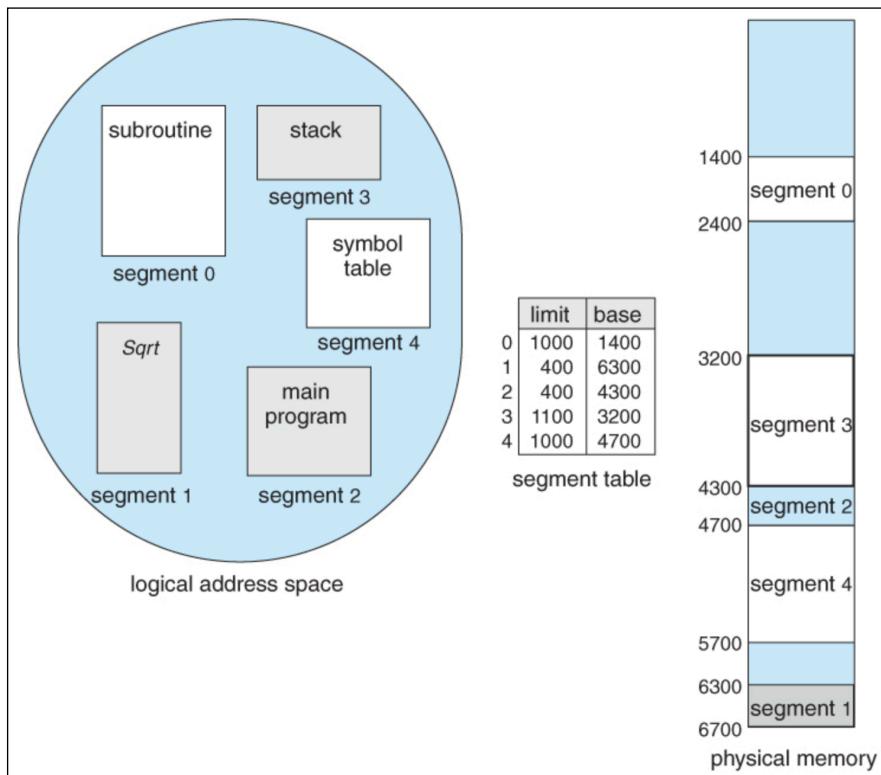
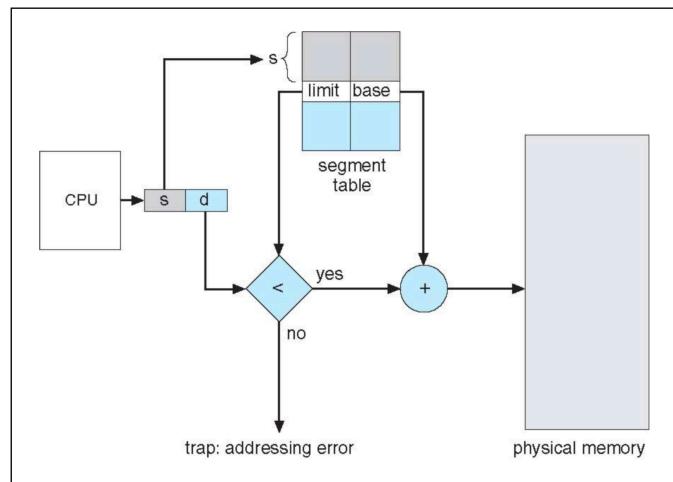
Con questo approccio non si vede più la memoria come un grande array lineare ma si associa un segmento di memoria ad ogni componente del programma (ex. stack, heap). A livello logico si usano indirizzi bidimensionali del tipo *<numero segmento, offset>*, dove numero segmento specifica il componente del programma e offset specifica l'indirizzo logico all'interno dell'elemento.

La traduzione da indirizzi logici bidimensionali a indirizzi fisici unidimensionali si effettua tramite la **tabella dei segmenti**, ogni suo elemento è una coppia ordinata *base segmento* e *limite segmento*, la base contiene l'indirizzo iniziale del segmento e il limite la fine.

Quindi riprendendo la coppia *<numero segmento, offset>*:

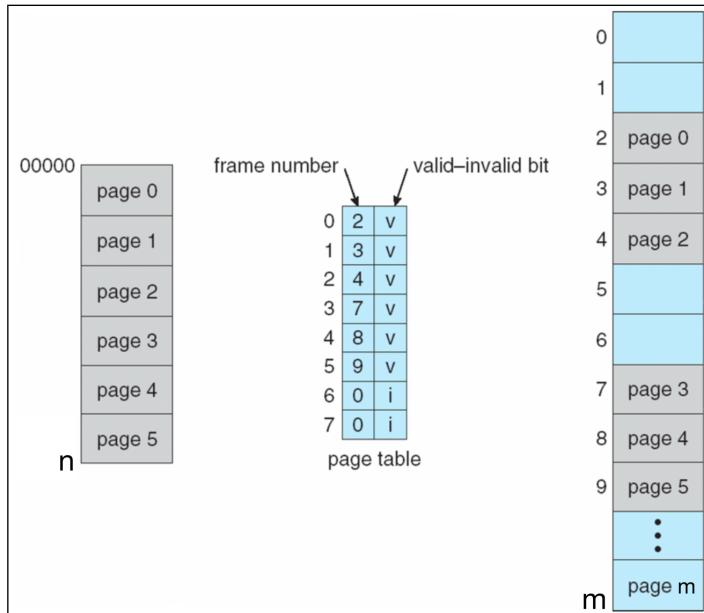
- Numero segmento estrae l'elemento base con l'indirizzo
- Offset viene confrontato con il limite corrispondente al componente  
IF (offset < limite) offset si somma a base e genera indirizzo fisico  
ELSE si genera errore

OSS soffre del problema della frammentazione esterna



## Paginazione

Consiste nel suddividere la memoria fisica in blocchi di dimensione fissa, detti **frame**, su cui mappare (tramite la **tabella delle pagine**<sup>32</sup>) blocchi di memoria logica, detti **pagine**. Ogni processo possiede una propria tabella delle pagine che si ricollega ad una tabella principale, detta **tabella di frame**, visibile solo al kernel, e nella quale sono indicati tutti i frame già occupati e il/i processo/i a cui sono assegnati.



Questa completa separazione tra indirizzi fisici ed indirizzi logici è ciò che permette ad ogni processo di avere un'immagine di memoria (memoria virtuale) con spazio degli indirizzi pari all'architettura (ex. 64 bit) anche se la memoria reale è nettamente inferiore.

La dimensione dei frame, e quindi delle pagine, dipende dall'hardware e dal sistema operativo, generalmente nei sistemi di Linux è di 4KB.

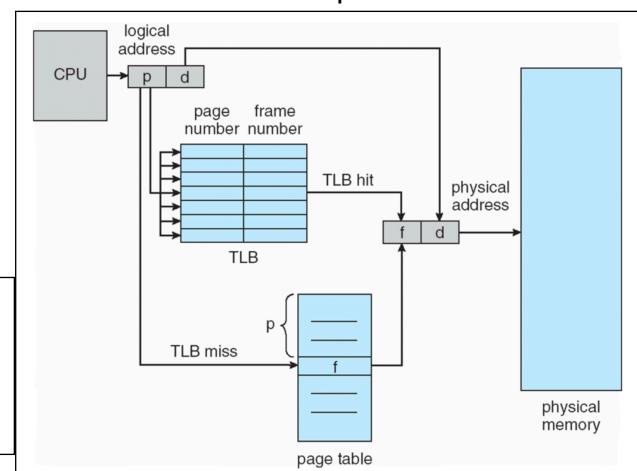
OSS Frame fissi evitano il problema della frammentazione esterna ma possono portare a problemi di frammentazione interna.

## Indirizzamento

Dato uno spazio degli indirizzi logici di  $2^n$  bit (dove  $n$  è pari all'architettura) con pagine di dimensione  $2^k$  bit, si ha che ogni indirizzo logico viene diviso in 2 componenti:

- Numero della pagina => contiene un indice della tabella delle pagine tramite il quale è possibile ricavare l'indirizzo base del frame, dato dagli  $n-k$  bit più significativi
- Offset di pagina => rimane invariato, determina l'offset all'interno del frame, dato dai  $k$  bit meno significativi

Ex. Avendo un'architettura a 32 bit, si ha:  
Spazio indirizzi logici =  $2^{32}$  bit ( $n = 32$ )  
Pagina = 4KB =  $2^{12}$  bit ( $k = 12$ )  
=> tabella delle pagine =  $2^{32}/2^{12} = 2^{20}$  pagine



Nei sistemi moderni avendo che la quantità di pagine disponibili è molto grande, per velocizzare l'accesso si utilizza una piccola cache associativa detta **TLB** (translation look-aside buffer) che contiene solo una parte della tabella delle pagine.

Così come per le normali cache ad ogni interazione si controlla prima la presenza della pagina nella TLB, altrimenti in caso di TLB miss si accede in memoria e si controlla la page table.

Avendo più processi che la utilizzano contemporaneamente, si associa ad ogni elemento un ASID (address-space identifier) che indica univocamente il processo a cui si riferisce.

<sup>32</sup> page table = array lineare con i riferimenti alle pagine del processo caricate in memoria, contiene inoltre informazioni su validità riferimento e permessi di lettura scrittura su frame

## La memoria virtuale

La memoria virtuale si basa sulla distinzione tra indirizzi logici e fisici, permette di astrarre dalla quantità reale di memoria del calcolatore ed associare ad ogni processo uno spazio degli indirizzi virtuale (immagine di memoria del processo) pari a  $2^n$ , dove n dipende dall'architettura (ex. 32, 64, ...).

Inoltre isola ogni blocco permettendo l'accesso ai dati solo a processi con permesso esplicito, aumentando la protezione e permettendo condivisione della memoria.

### Paginazione su richiesta

Estende il concetto di paginazione anche all'utilizzo in memoria secondaria, permette di caricare in memoria solo le pagine realmente utili, le altre rimangono su disco.

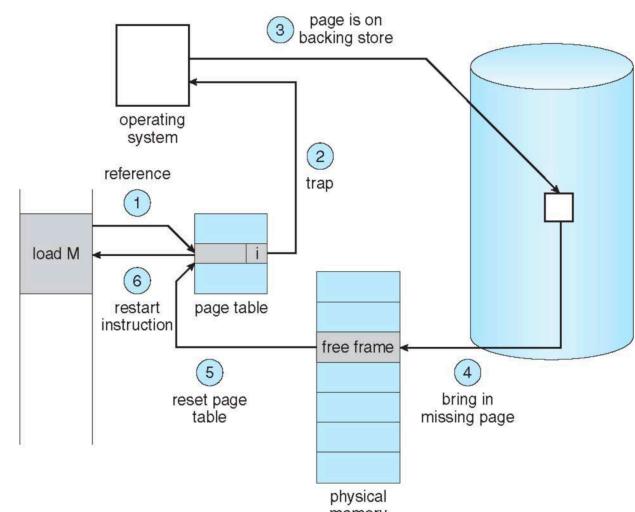
Ad ogni accesso a memoria le operazioni sono:

1. Si controlla nella tabella delle pagine il bit di validità del riferimento
2. IF (riferimento valido) carica da memoria la pagina corrispondente, exit  
ELSE controlla l'esistenza del riferimento nella tabella interna al processo (in PCB):  
IF (riferimento esiste) **page fault**, goto 3  
ELSE errore, segmentation fault

3. Si individua un frame libero e la pagina in memoria secondaria
4. Si trasferisce la pagina dal disco alla memoria principale
5. Completato il trasferimento si modifica la tabella delle pagine
6. Si riavvia l'istruzione interrotta

Teoricamente un processo potrebbe iniziare l'esecuzione caricando solo la prima pagina e poi caricare le altre, una alla volta.

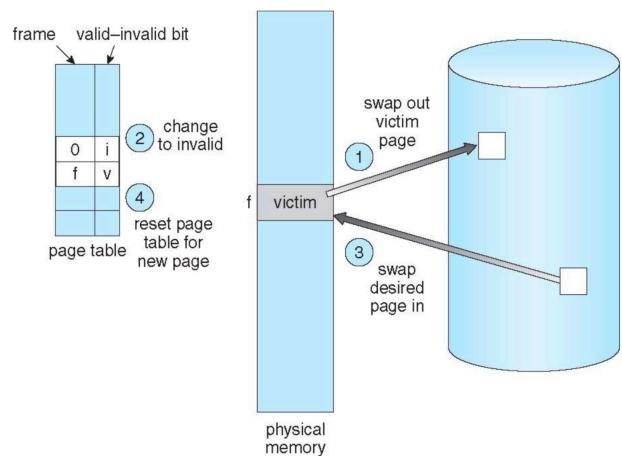
Sebbene questo modo permetta di risparmiare molto spazio in memoria, ogni caricamento di pagina è molto costoso, ed avendo anche solo un page fault ogni 1000 accessi l'esecuzione del programma può rallentare di 40 volte.



OSS la generazione di nuovi processi avviene duplicando il genitore, ma poiché gran parte dei processi figli non necessita di tutte le pagine del padre, è possibile utilizzare il sistema di copy-on-write che duplica solo le pagine che vengono modificate dal figlio. In linux e diverse versioni UNIX questo si effettua tramite la `vfork()`.

La continua allocazione di nuove pagine da parte di più processi comporta il riempimento della memoria, si necessita quindi di un sistema di sovrallocazione che vada a modificare il punto 3.

- I. Se la memoria è piena si sceglie un frame vittima tra quelli inutilizzati.
- II. Se il **dirty bit**<sup>33</sup> corrispondente è posto a 1 si esegue un swap out del frame nel disco e swap in del nuovo frame, altrimenti direttamente lo swap in.



<sup>33</sup> dirty bit = (bit di modifica) indica se frame è stato sovrascritto dopo inserimento in memoria

## Gestione della paginazione su richiesta

Per il funzionamento ottimale di questo sistema è richiesta la presenza di 2 algoritmi:

- **Sostituzione delle pagine** => scelgono quali frame sostituire, minimizzano la frequenza di page fault
- **Allocazione dei frame** => determinano numero di frame da associare al processo

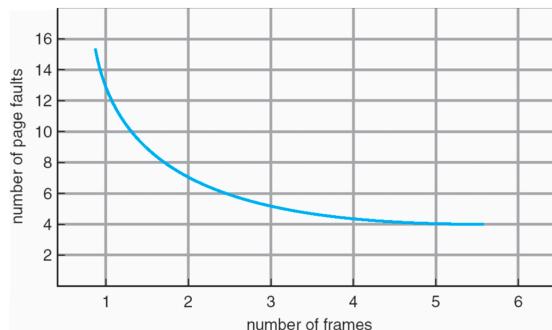
## Algoritmi di sostituzione delle pagine

La valutazione degli algoritmi si effettua tramite una reference string, ovvero una stringa composta dai soli riferimenti alle pagine.

Si considera solo il numero della pagina e non tutto l'indirizzo, poiché l'accesso alla stessa pagina, già contenuta in memoria, non provoca page fault.

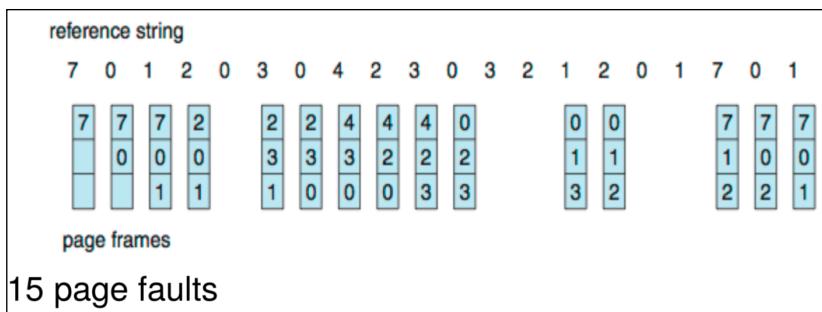
Negli esempi:

Reference string => 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1  
numero di frame = 3



## First in, first out (FIFO)

Si sostituisce la pagina presente in memoria da più tempo, non è necessario salvare l'istante di memorizzazione di ogni pagina, si usa una coda FIFO delle pagine in memoria. Questo algoritmo è semplice da implementare ma non ha buone prestazioni poiché non considera la frequenza di utilizzo delle pagine.



Inoltre questo algoritmo soffre dell'**anomalia di belady**, ovvero aumentando i frame le prestazioni invece di migliorare potrebbero peggiorare

ex. Reference string =>

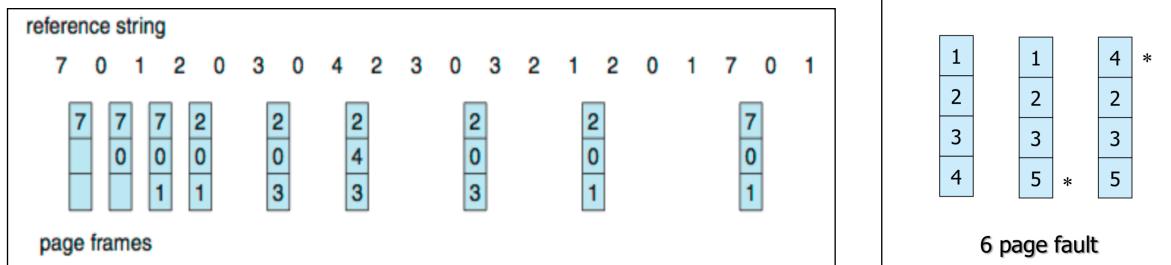
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

3 frame			
1	1	4	5
2	2	1	3
3	3	2	4
9 page fault			
4 frame			
1	1	5	4
2	2	1	5
3	3	2	
4	4	3	
10 page fault			

## Algoritmo ottimale (OPT o MIN)

Si sostituisce la pagina che non verrà usata per il periodo di tempo più lungo, non potendo conoscere questa informazione questo algoritmo rimane teorico ed è utilizzato solo come metro di misura per gli altri algoritmi.

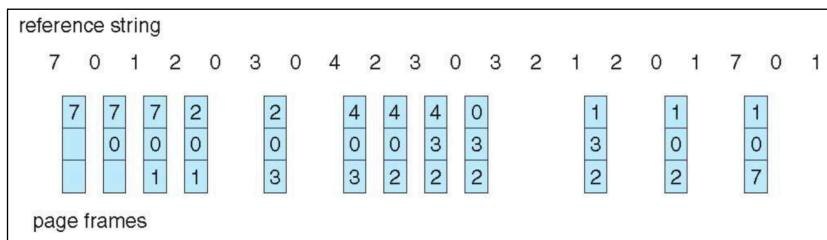
I page fault con la stessa reference string diventano 9.



## Last Recently Used (LRU)

Si sostituisce la pagina che non è stata utilizzata per più tempo, concettualmente simile a OPT ma guarda al passato invece che al futuro, algoritmo con prestazioni migliori rispetto a FIFO ma ovviamente peggiori rispetto a OPT.

I page fault con la stessa reference string diventano 12.



Implementabile in 2 modi:

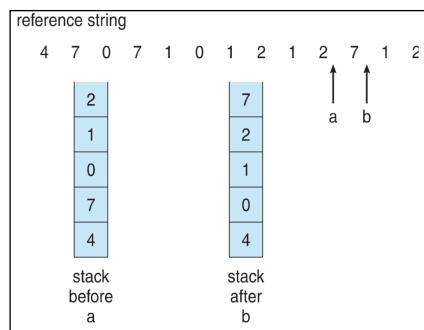
- **Contatori** => si associa un contatore generale alla CPU e dei contatori specifici ad ogni pagina, ad ogni riferimento si incrementa il valore del contatore generale e si copia il valore nel registro contatore associato alla pagina.

Ad ogni sostituzione si prende il frame con il valore del contatore più basso.

Richiede una ricerca lineare sulla tabella delle pagine ad ogni caricamento e può avere problemi di overflow sul contatore.

- **Stack** => si mantengono i numeri di pagina in uno stack implementato tramite lista doppiamente collegata, ad ogni riferimento a una pagina il numero corrispondente viene posizionato in cima, in modo che in basso si trovi sempre il più vecchio.

L'aggiornamento della pila è più costoso (max. cambio 6 puntatori) ma non richiede ricerche sulla tabella delle pagine.



OSS LRU e OPT sono detti algoritmi a pila e non soffrono dell'anomalia di belady. È quindi possibile dimostrare che l'insieme delle pagine in memoria per n frame è sottoinsieme dell'insieme con n+1 frame.

## LRU approximation algorithms

l'algoritmo LRU è lento e richiede un hardware dedicato.

Nella realtà si usano delle varianti basate su un **reference Bit** (bit di riferimento), ovvero un bit inizializzato a 0 che si associa ad ogni pagina e diventa 1 ogni volta che si fa riferimento ad essa.

Di base solo questo sistema non permette di conoscere l'ordine di accesso alle pagine, si usano quindi i seguenti algoritmi:

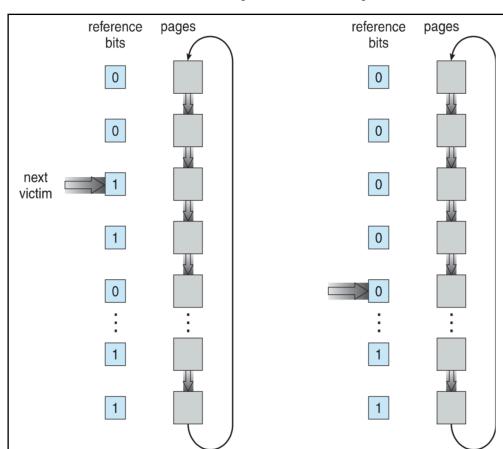
- **Bit supplementari di riferimento** => si salva in memoria una tabella contenente un byte per ogni pagina, a intervalli regolari si passa il controllo al sistema operativo che shifta ogni byte e salva il reference bit attuale della pagina nel bit più significativo.

Quindi si ha che la pagina da sostituire è quella con il valore del byte più piccolo.

- **Seconda chance** => implementato tramite una coda circolare, si basa sull'algoritmo FIFO al quale si aggiunge il reference bit, ad ogni richiesta di sostituzione si esegue ricorsivamente il seguente controllo:

IF (reference bit = 0) si sostituisce pagina

ELSE si pone reference bit = 0 e si passa a prossimo elemento della pila



- **Seconda chance migliorato** => miglioramento dell'algoritmo utilizzando una coppia ordinata formata del reference bit e del dirty bit:
  1. (0,0) => non utilizzato o modificato di recente
  2. (0,1) => non usato ma modificato recentemente, necessita swap out
  3. (1,0) => usato recentemente ma non modificato, probabilmente verrà usato presto
  4. (1,1) => usato e modificato recentemente, probabilmente verrà riutilizzato presto ed inoltre necessita di swap out

**ATTENZIONE** la ricerca della classe minore può richiedere di scandire la coda più volte

## Algoritmi basati sul conteggio

Si mantiene un contatore del numero di riferimenti fatti alla pagina, 2 tipi:

- **LFU** (least frequently used) => si rimpiazza pagina con valore più basso
  - **MFU** (most frequently used) => si rimpiazza valore più alto, assumendo che pagine con valore basso sono appena state caricate in memoria e quindi deve ancora essere usate
- Implementazione costosa e poco efficienti

OSS per diminuire i costi di swap out è possibile mantenere una pool di frame liberi in cui eseguire lo swap in delle nuove pagine rimandando lo swap out.

Inoltre si mantiene una lista delle pagine modificate e ogni volta che il dispositivo di paginazione è inattivo si esegue lo swap out di una di queste evitando di dover attendere al momento di selezione della vittima

## Algoritmi di allocazione dei frame

Basati sul fatto che ogni processo richiede un numero minimo di frame che dipende dall'architettura, con un massimo che dipende dalla disponibilità di memoria.

Gli algoritmi base sono:

- **Allocazione uniforme** => Dati  $m$  frame ed  $n$  processi, il numero di frame per ogni processo è dato da  $m/n$ , ovvero si assegna un numero uguale di frame per processo.  
Solitamente viene comunque lasciata una pool di frame liberi
- **Allocazione proporzionale** => il numero di frame assegnati al processo dipende dalla dimensione massima del processo, secondo la formula:  $a_i = s_1/Sxm$

$a_i$  = frame allocati per il processo

$s_i$  = numero di pagine del processo

$S$  = numero di pagine per tutti i processi

$m$  = numero complessivo di frame

ex.	$m = 64$
	$s_1 = 10$
	$s_2 = 127$
	$a_1 = (10/137) \times 64 = 5$
	$a_2 = (127/137) \times 64 = 59$

OSS può essere modificato basandolo sulla priorità invece che sulla dimensione

## Allocazione globale e locale

Avendo più processi che competono per l'utilizzo della memoria bisogna introdurre un altro metodo di classificazione per la selezione e sostituzione dei frame:

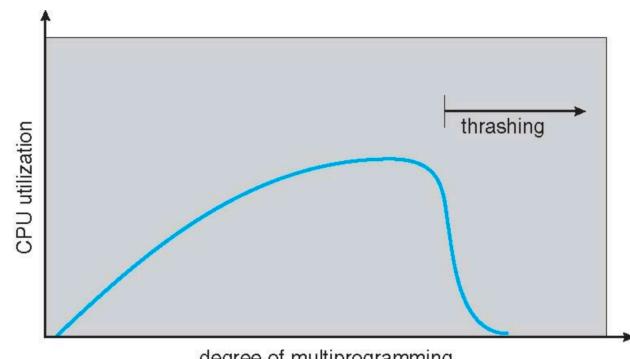
- **Sostituzione globale** => al momento della selezione del frame il processo può scegliere tra tutti i frame presenti in memoria, anche appartenenti ad altri processi.  
Il tempo di esecuzione dei processi può variare in modo significativo ma questo sistema è comunque generalmente più produttivo ed quindi usato su più sistemi
- **Sostituzione locale** => al momento della selezione del frame il processo può scegliere solo tra i frame in suo possesso, può comportare sottoutilizzo della memoria

## Trashing

Situazione in cui non si hanno abbastanza frame liberi in memoria e quindi il processo è costantemente occupato a spostare pagine dal disco alla memoria e viceversa.

Era frequente nei primi sistemi di paginazione, continui page Fault causano un aumento dell'attività di I/O e di conseguenza portano ad uno scarso utilizzo della CPU, il sistema operativo quindi, credendo di avere più risorse a disposizione aumentava il grado di multiprogrammazione fino a raggiungere il collasso del sistema.

Può essere parzialmente migliorato utilizzando i sistemi di sostituzione locale, che permettono di limitare multiprogrammazione, e algoritmi di sostituzione con priorità.



Nei sistemi moderni questo problema è risolto dal **modello di località**, una località è un insieme di pagine del processo che devono essere usate insieme, il processo quindi si sposta da una località all'altra impedendo una frequenza di page fault troppo alta.  
(ex. Un passaggio di località corrisponde ad un passaggio da una procedura all'altra)

Per evitare il trashing basta quindi impostare un numero massimo e minimo per la frequenza di page fault, se aumenta troppo il processo viene sospeso.

## File system

Componente del sistema operativo che fornisce i meccanismi di accesso e memorizzazione dei dati, consente una visione logica uniforme di tutte le informazioni memorizzate in modo permanente sul dispositivo.

Permette di astrarre dalle caratteristiche fisiche dei supporti di memorizzazione.

### **Il file**

Un file è un insieme di informazioni correlate che formano uno spazio logico, vengono salvate nella memoria secondaria e vengono strutturate a seconda del tipo del file.

Ogni file è caratterizzato da un insieme di attributi:

- **nome** => univoco nella directory (255 caratteri), leggibile da utente
- **Identificatore** => etichetta unica che lo identifica nel file system, leggibile da sistema
- **Tipo** => identifica la struttura interna del file e consente di capire come trattarlo, visibile istantaneamente come estensione nel nome
- **Locazione** => puntatore alla locazione del file nel dispositivo di memorizzazione
- **Dimensione** => dimensione corrente
- **Permessi** => permessi read / write / execute
- Ora / data / id utente => informazioni su creazione ed ultime modifiche

Tutti i file sono contenuti in **directory**<sup>34</sup>, salvate anch'esse in memoria secondaria, mantengono tutte le informazioni su di essi. (ogni file è un elemento della directory)  
Alcuni sistemi recenti supportano anche attributi estesi.

### **Operazioni sui file**

Il file è un **tipo di dato astratto** sul quale si possono definire le operazioni fondamentali:

- **Creazione** => trova ed alloca spazio nel file system, crea un nuovo FCB in memoria e un nuovo elemento nella directory
- **Scrittura** => write(), puntatore in FCB
- **Lettura** => read(), puntatore in FCB (stesso di write())
- **Riposizionamento** => ricerca elemento e cambia puntatore a locazione corrente
- **Cancellazione** => rilascia spazio associato al file
- **Troncamento** => cancella contenuto file mantenendo attributi

Tutte le altre operazioni sono combinazioni di queste 6.

Siccome gran parte di queste richiederebbe la continua ricerca dei file sul disco nel file system il sistema prima di eseguirne una effettua sempre **open()** sul file in modo da mantenerlo nella tabella dei file aperti contenuta in memoria.

**OSS** Avendo che più processi possono accedere agli stessi file si ha una doppio livello di tabelle, uno con tutti i file aperti nel sistema, uno con tutte le tabelle di ogni processo.  
La tabella dei file del processo contiene semplicemente riferimenti alla tabella del kernel.

### **Tabella dei file aperti (sistema)**

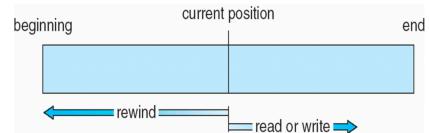
Ad ogni file nella tabella sono associate:

- **Puntatore al file** => ultima posizione di lettura/scrittura, unico per ogni processo
- **Contatore file aperti** => numero di open()/close() effettuate dai processi
- **Posizione nel disco** => evita ricerca ad ogni operazione
- **Privilegi** => differenti per ogni processo

È possibile inoltre applicare lock al file per proteggere accesso concorrente.

---

<sup>34</sup> Directory = specifica entità del file system che elenca altre entità (file/directory).



## Metodi di accesso ai file

- **Accesso sequenziale** => usato da editor e compilatori, informazioni si elaborano ordinatamente avanti e indietro.
- **Accesso diretto** => file diviso in record di lunghezza fissa, utile per accesso immediato a grandi quantità di dati come database  
È possibile inoltre utilizzare varianti di questi due metodi che usano una tabella index contenente i puntatori a vari blocchi del file.

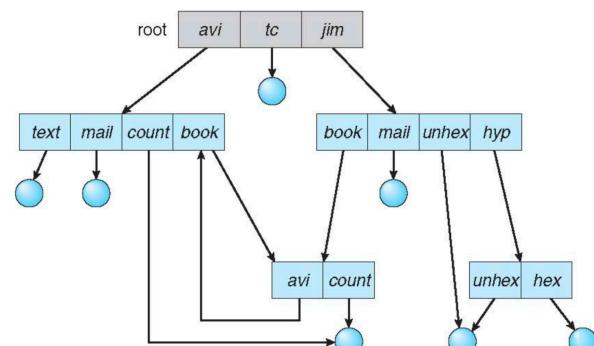
In un disco ci possono essere più partizioni con file system e sistemi operativi differenti

## Strutture delle directory

Esistono diverse definizioni della struttura logica delle directory, una estensione dell'altra:

- **Directory a 1 livello** => tutti i file sono contenuti nella stessa directory, presenta limiti all'aumentare dei file e degli utenti poiché impossibile mantenere unicità nomi
- **Directory a 2 livelli** => ogni utente dispone della sua user file directory (UFD) contenuta a sua volta nella master file directory (MFD), risolve collisione sui nomi poiché le ricerche avvengono prima nella directory dell'utente.  
Complica l'accesso ai dati condivisi tra utenti poiché per accedervi bisogna riferirsi prima alla MFD ed oltre al nome bisogna specificare anche l'utente.  
Nel caso dei file di sistema esiste una directory condivisa che viene acceduta automaticamente tramite un **search path**<sup>35</sup> predefinito se la ricerca iniziale fallisce.
- **Directory ad albero** => ogni utente può creare altre sottodirectory a partire dalla sua **root directory** (radice).  
introduce il concetto di directory corrente nella quale vengono eseguite le ricerche, se i file non si trovano in questa directory bisognerà specificarne il path.
- **Directory a grafo aciclico** => permette alle directory di avere sottodirectory e file in comune, anche tra utenti diversi, realizzabile tramite puntatori o copia completa.  
**WARNING** Nel caso dei puntatori è importante porre attenzione al problema della cancellazione, poiché deallocando un file tramite un percorso rimangono comunque i puntatori al file da altri percorsi.  
Generalmente quindi l'eliminazione provoca la cancellazione dei dati ma si mantiene comunque una zona che una volta puntata genererà un errore.  
Altrimenti si associa ad ogni file un contatore dei suoi collegamenti, se il contatore arriva a 0 si elimina il file, altrimenti si elimina solo il collegamento ed il file rimane.
- **directory a grafo generale** => permette di usare collegamenti a file e directory in qualunque punto del file system, questo implica complessità maggiore degli algoritmi di ricerca che, se mal progettati, potrebbero cadere in loop infiniti.

OSS con directory a grafo generale in caso di cancellazione di una directory o file è consigliabile usare il metodo di **garbage collection** che esamina tutto il file system alla ricerca dei collegamenti per eliminarli definitivamente



<sup>35</sup> search path (percorso di ricerca) = sequenza di directory sulla quale è stata effettuata la ricerca

- percorso assoluto => a partire da root directory
- percorso relativo => dalla directory corrente

## Montaggio file system

Operazione necessaria per aprire e rendere un file system accessibile ai processi di un sistema, permette di usare più file system differenti su stesso OS.

La prima operazione da compiere per il montaggio consiste nel fornire il nome del dispositivo ed il **punto di montaggio**<sup>36</sup>, dopodiché si chiede al driver del dispositivo di leggere le directory in modo da annotarne la struttura.

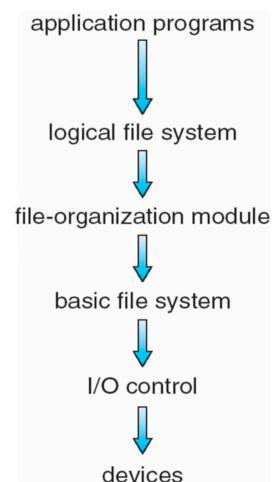
## Realizzazione del file system

### Struttura generale del file system

I trasferimenti tra memoria centrale e secondaria si eseguono a blocchi.  
(generalmente compresi tra 512 byte e 4096 byte)

Il file system è stratificato, ovvero composto da livelli distinti:

- **I/O control** => (microcodice del controllore) driver dei dispositivi, sono come traduttori che interpretano i comandi ad alto livello ed emettono istruzioni a basso livello per la gestione dei blocchi.
- **Basic file system** => invia comandi al driver apposito per lettura e scrittura su blocchi fisici del disco, inoltre gestisce cache e buffer.
- **file-organization module** => consente traduzione da blocchi logici a fisici, inoltre comprende bitmap per individuazione spazi vuoti.
- **Logical file system** => gestisce i metadati e comprende tutte le strutture del file system. (eccetto contenuto file)



### Strutture dati nei dischi

Contengono informazioni su come avviare il sistema operativo e su struttura generale:

- **Boot control block** => (blocco di controllo dell'avviamento) primo blocco del volume, se presente contiene informazioni necessarie per l'avviamento di un sistema operativo.
- **Volume control block** => (blocco di controllo del volume) contiene dettagli sulla partizione (numero blocchi, contatore blocchi liberi, FCB liberi e puntatori).
- **Struttura della directory** => organizzazione dei file, una per file system
- **File control block (FCB)** per ogni file => contiene dettagli sul file

### Strutture dati in memoria

Permettono gestione del file system:

- **Tabella di montaggio** => informazioni su ogni volume montato
- **Struttura della directory** => informazioni su directory accedute di recente
- **Tabella file aperti sistema** => copia FCB di ogni file aperto, il del file in questa tabella è detto **file descriptor** (UNIX) o **handle** (windows)
- **Tabella file aperti per ogni processo** => puntatore a tabella sistema più altre informazioni relative al processo
- **Buffer** => conserva blocchi durante lettura / scrittura

OSS i sistemi UNIX trattano le directory esattamente come file distinguendole solo tramite un campo che indica in tipo.

OSS i è possibile utilizzare dischi privi di OS e file system che immagazzinano dati secondo una struttura data dal programma che li utilizza (ex. database)

<sup>36</sup> punto di montaggio = locazione di partenza del file system, è una directory vuota (ex. /home)

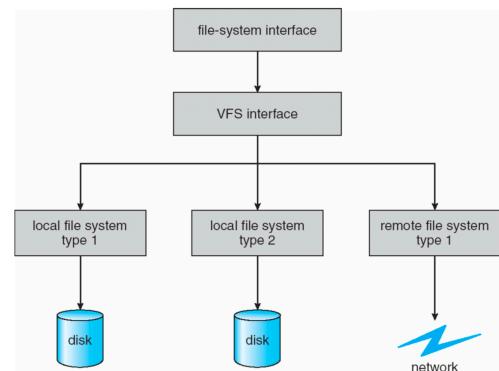
## File system virtuali (VFS)

Strato intermedio utilizzato per astrarre l'implementazione delle procedure del file system permettendo così di interfacciare diverse strutture sullo stesso sistema.

Sarà quindi uno strato sottostante locale a contenere il codice implementativo del file system.

Permettono inoltre rappresentazione univoca di un file su tutta la rete tramite una rappresentazione detta **vnode** che contiene un indicatore numerico identificativo, consentendo di distinguere file locali da file remoti.

Nei VFS LINUX un file viene detto **inode**.



## Realizzazione delle directory

L'implementazione più semplice è tramite lista lineare contenente nomi dei file e puntatori ad i blocchi dei dati, poco efficiente (solitamente lista ordinata o albero).

Per migliorarne le prestazioni è possibile usare una **tavella hash**, generalmente la gestione delle collisioni è effettuata tramite il metodo delle liste collegate.

## Metodi di allocazione

- **Allocazione contigua** => ogni file occupa un insieme di blocchi contigui sul disco, non ho quindi ulteriori spostamenti della testina e permette accesso sequenziale ai dati.  
Teoricamente ottimale ma difficile da realizzare poiché a causa del problema della frammentazione è difficile individuare lo spazio per un nuovo file.
- **Allocazione concatenata** => memorizza il file tramite una lista doppiamente concatenata di blocchi, la directory contiene puntatore a primo ed ultimo.  
Anche se attenuabile tramite l'allocazione di cluster (insiemi di blocchi) questo metodo implica la perdita di spazio a causa del salvataggio dei puntatori in ogni blocco, i quali essendo sparsi costringono inoltre ad una lettura sequenziale.
- **File allocation table (FAT)** => variante dell'allocazione concatenata, salva tabella all'inizio di ogni volume, ogni elemento della tabella rappresenta un blocco.  
Gli elementi directory contengono numero del primo blocco del file i quali a loro volta contengono numero corrispondente al prossimo blocco o un valore speciale nel caso di terminazione.  
Utilizza una **bitmap**<sup>37</sup> per segnare quali blocchi sono inutilizzati. (inutilizzato = 0)
- **Allocazione indicizzata** => raggruppa tutti i puntatori ad i blocchi in una sola locazione chiamata blocco indice (array di puntatori), uno per ogni file.  
Avendo bisogno di un indice per ogni file ho un overhead maggiore e si introduce la questione sulla grandezza del blocco, risoluzioni possibili:
  - **Schema concatenato** => formato da normali blocchi del disco contenenti unicamente puntatori ai blocchi, ultimo è null o altro blocco.
  - **Indice a più livelli** => il sistema utilizza l'indice di primo livello per individuare un secondo blocco indice di secondo livello che individua blocco dati richiesto.  
OSS possono essere anche più livelli, ogni livello incrementa numero di blocchi indirizzabili in modo esponenziale
  - **Schema combinato** => usata nei sistemi UNIX, in questo caso i primi 12 di 15 puntatori puntano direttamente ai file, gli altri 3 usano schema a livelli.  
Primo usa indirizzamento indiretto singolo, secondo doppio e terzo triplo.

<sup>37</sup> bitmap => vettore di bit, ottimizza ricerca primo blocco vuoto

## EXTRA

### **Legge di Amdahl**

Serve per calcolare le prestazioni guadagnate aggiungendo core ad un applicazione avente sia componenti seriali che paralleli:

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

dove:

S = posizione seriale

N = numero di processori

### **Misurare tempo esecuzione processo**

Comando => **time ./nomeprogramma**

In output si hanno i seguenti parametri:

Real => tempo reale di esecuzione considerando kernel ed altri processi

User => tempo passato solo in modalità user (non considera waiting)

System => tempo passato nel kernel

### **Legge di Amdahl (sistemi di calcolo)**

Permette di calcolare l'ottimizzazione massima di un programma modificando una sola componente (funzione).

$$speedup = \frac{1}{\frac{\alpha}{k} + (1 - \alpha)}$$

$\alpha \in [0,1]$  = percentuale di tempo nella funzione ottimizzata

K = ottimizzazione della funzione, si esprime tramite moltiplicatore (ex. 2X)

Per profilare un programma e calcolare i suddetti valori si usa gprof, nel seguente modo

1. Si compila tramite: **gcc -pg programma.c -o programma**
2. Si esegue **./programma**, in questo modo si genera il file **gmon.out**
3. Si esegue il programma sotto gprof con: **gprof ./programma > testo.txt**

In questo modo tutta la profilazione verrà inserita nel file testo.txt

### **Canali standard POSIX**

0 = input

1 = output terminale

2 = error

3, 4, ... = file

## Struttura generale tabelle controllo di un OS

