

Sistemi operativi

Prof. Avvenuti, 2023-2024

Indice

1. [Introduzione](#)
2. [Gestione dei processi](#)
3. [Sincronizzazione tra processi](#)
4. [Gestione della memoria](#)
5. [Gestione delle periferiche](#)
6. [Il file system](#)
7. [Protezione e sicurezza](#)
8. [Architettura del sistema Unix](#)

Il layout a volte è un pò storiato perchè il pdf è generato direttamente da appunti presi in markdown

1. Introduzione

Storia dei sistemi operativi

Primi sistemi di elaborazione

Inizialmente non c'è nessun SO, solamente il compilatore: il programmatore arriva direttamente con il sorgente su schede perforate, queste vengono inserite, compilate, aggiustate con i dati di ingresso e poi ricaricate per essere eseguite

Successivamente nascono i primi sistemi monoprogrammati: la memoria RAM diventa più grande e permette la nascita di un primo Sistema Operativo, costituito da un "Monitor" (l'equivalente di una shell moderna) più un BIOS contenente le routine per eseguire trasferimenti (anche con dma) dal nastro magnetico, la prima memoria di massa mai utilizzata

Il programmatore arriva con un batch di schede perforate costituite da: sorgente, istruzioni per il Monitor scritte in Job Control Language (ad esempio direttive su come compilare le schede perforate) e dati di input

Una volta raccolti abbastanza batch vengono mandati in esecuzione (uno alla volta, senza un particolare criterio), grazie al DMA (fornisco alla periferica un buffer e il numero di byte da leggere/scrivere) si poteva caricare un nuovo programma o leggere l'output di uno terminato anche quando ne era in esecuzione un altro (tecnica detta spooling: Simultaneous Peripheral Operation On-Line)

Sistemi operativi moderni

- Sistemi batch multiprogrammati

Vengono introdotte le interruzioni, che consentono a più programmi di risiedere concorrentemente in RAM, il sistema operativo, dovendo gestire le interruzioni e lo scheduling, inizia a diventare più complesso, introducendo overhead

- Sistemi time-sharing

Nei sistemi multiprogrammati inizia a diventare importante il tempo di risposta, si introduce dunque la preemption (diritto di revoca) sulla base del tempo di CPU

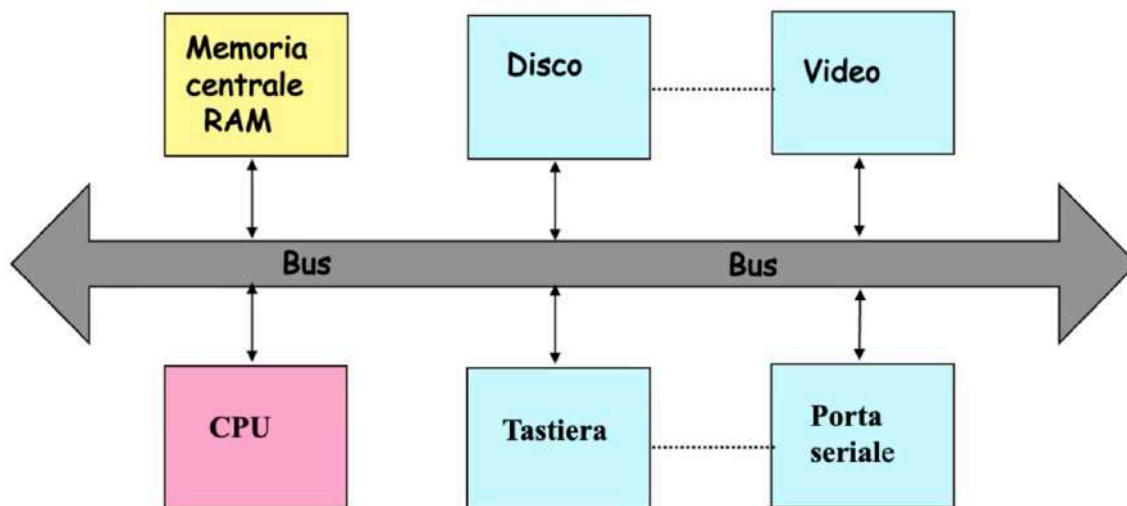
- Sistemi in tempo reale

Sono sistemi in cui la correttezza dei risultati prodotti dipende non solo dalla correttezza logica ma anche da quella temporale, questo è tipico dei sistemi embedded, che devono maneggiare sensori ed attuatori (tipicamente su base prioritaria) rispettando determinati vincoli di tempo: si parla di sistemi a tempo reale **hard** se tutti i task non possono violare la

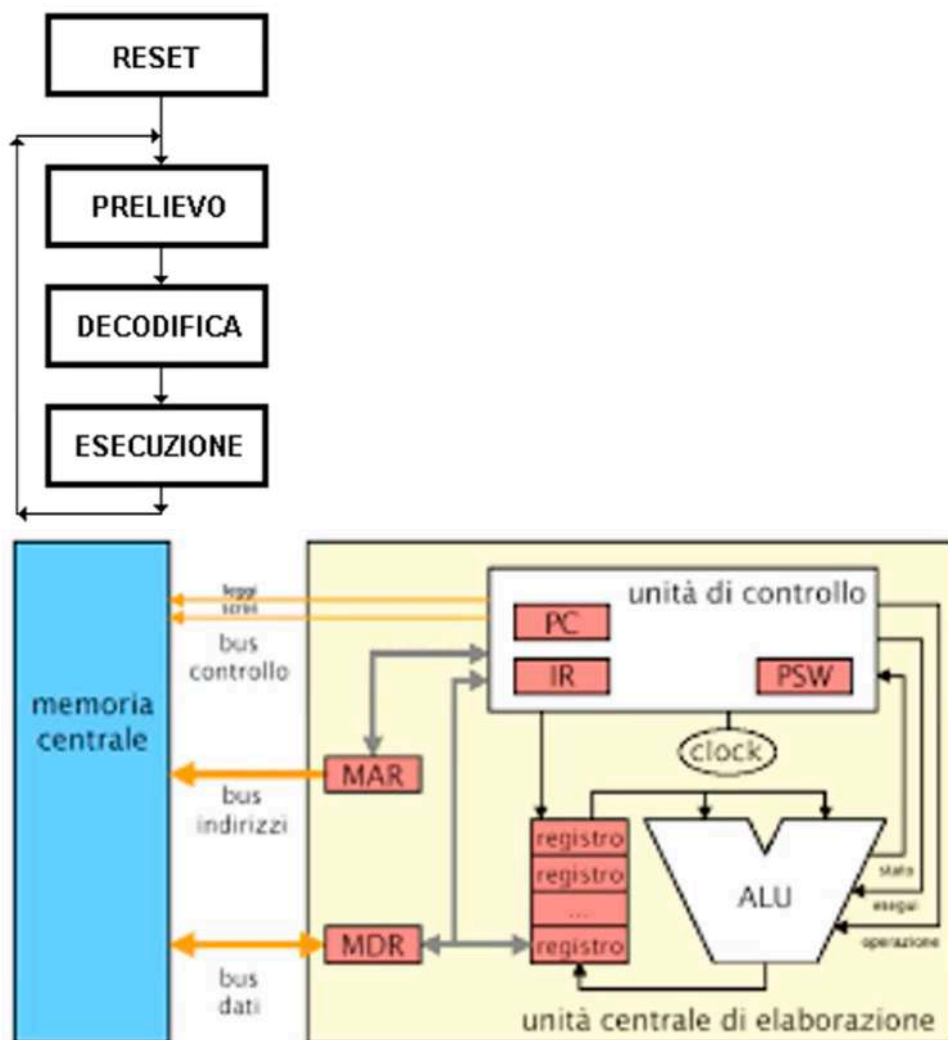
deadline (esempio: controlli di un aereo), **soft** se qualche task può violare la deadline (peggiorando la qualità del servizio, esempio: stazione metereologica)

Architettura di Von Neumann

È l'architettura base di tutti i calcolatori moderni:



- CPU: in quelle più sofisticate è presente una cache, se ad indirizzamento diretto questa è composta da memoria delle etichette (con bit di validità) e memoria dati, e scompone gli indirizzi in:
 - Offset (esclusi i byte enable), grande 3 bit se la cacheline è di 64 Byte
 - Indice, grande a bit se la cacheline è di 2^a cacheline
 - Etichetta, i restanti bit più significativi che identificano univocamente le cacheline
- RAM: ricorda che viene vista come un vettore
- Tastiera, disco, porta seriale... : sono dispositivi di ingresso/uscita collegati a determinate interfacce, ossia hardware dedicato che esegue parallelamente, con i propri registri interni mappati nello spazio di indirizzamento (non per forza di I/O) del processore
- BUS: è un insieme di fili diviso in bus dati e bus indirizzi (con eventuali byte enable)



Esempio di registri di controllo [non accessibili in assembler]

PC: Program Counter (Indirizzo dell'istruzione successiva)

IR: Instruction Register (Istruzione attuale)

PSW: Program Status Word (Flags)

MAR: Memory Address Register (A quale indirizzo leggere/scrivere)

MDR: Memory Data Register (Dati letti/scritti dalla memoria)

Meccanismo delle interruzioni:

Introdotta per evitare le attese attive, funziona nel seguente modo:

1. Nel registro PSW c'è il bit IF che stabilisce se le interruzioni sono attive o meno
2. Se sono attive la cpu, dopo ogni istruzione, controlla tramite un piedino, gestito da un componente hardware esterno, se c'è una nuova interruzione in attesa
3. Se c'è un'interruzione in attesa allora questa viene servita: la cpu recupera il tipo dell'interruzione e con quello accede alla Interrupt Descriptor Table, da cui legge varie informazioni tra cui i registri PSW e PC nuovi, che sovrascrivono i vecchi, salvati in pila
4. In questo modo passa ad eseguire la routine di interruzione, che termina con l'istruzione IRET, la quale riporta i registri PSW e PC al loro posto (gli altri registri o vengono salvati in

pila o ne vengono usate due copie)

Struttura dei sistemi operativi

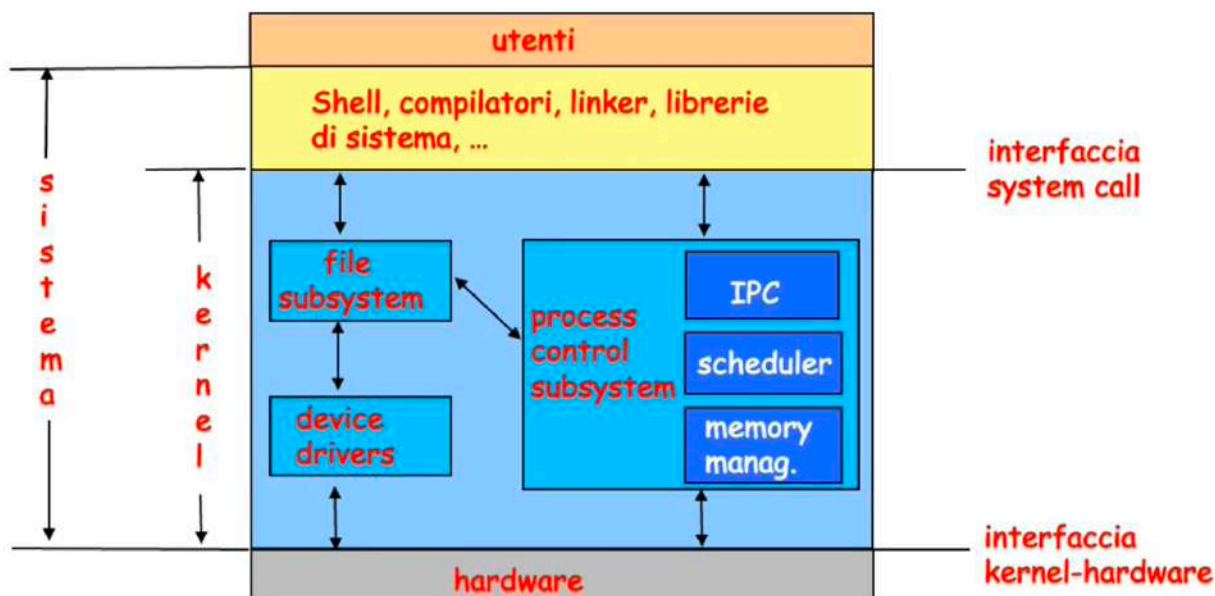
Un sistema operativo è un componente software le cui funzioni principali sono:

- Facilitare lo sviluppo e la portabilità del software, astruendo i dettagli hardware tramite primitive dette system calls (di fatto creando, tramite il cosiddetto kernel, una **macchina "virtuale"**)
- Implementare meccanismi di protezione
- Realizzare politiche di gestione delle risorse hardware

Nota:

- **Meccanismo**: qualcosa di basso livello, cablato in hardware o realizzato in software nelle routine di base
- **Politica**: criterio di scelta realizzato utilizzando i vari meccanismi presenti

Si parla di **microkernel** se i processi di sistema eseguono solamente lo stretto indispensabile e tutto il resto viene delegato a processi utente (più sicuro ma meno veloce), altrimenti si parla di **kernel monolitico**



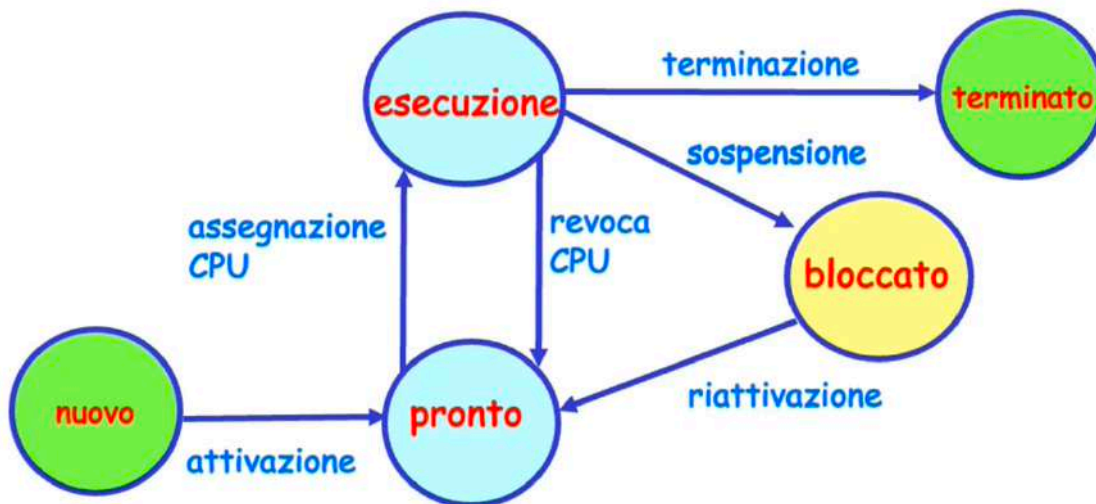
IPC: Inter Process Communication (comunicazione tra gli spazi di indirizzamento privati dei processi)

2. Gestione dei processi

Concetto di processo

Un **processo** può essere definito come la sequenza di eventi generati dal calcolatore durante l'esecuzione di un programma con dei particolari input

Stati di un processo



Descrittore di processo (organizzati in una tabella di processi, possono variare a seconda del SO):

- Nome (es: PID)
- Stato (es: esecuzione, pronto, ...)
- Modalità di servizio (es: priorità, nei SO che la usano)
- Informazioni sulla memoria (es: stack, heap, il registro cr3 visto a calcolatori)
- Contesto (es: registri del processore)
- Utilizzo delle risorse (es: dispositivi di I/O, file aperti, ...)
- Puntatore al processo successivo

Cambio di contesto:

1. Salvataggio del contesto del processo in uscita nel suo descrittore
2. Inserimento del suo descrittore nella coda appropriata
3. Selezione di un altro processo dalla coda pronti (short term scheduling)
4. Caricamento del contesto del processo scelto nei registri della CPU

È possibile avere registri doppi (almeno alcuni, es: non PC e PSW) per kernel e user, così quando viene chiamata una primitiva non bisogna fare sempre il salvataggio del contesto ma solamente se cambia effettivamente il processo

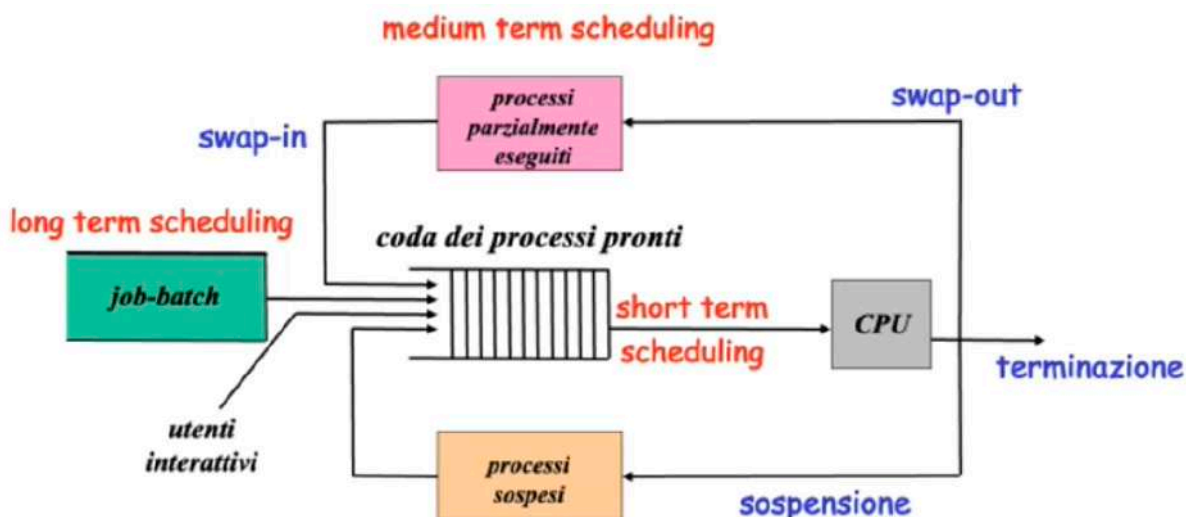
Per sistemi in tempo reale spesso il numero di processi viene deciso a priori, in generale però un processo può crearne un secondo: **ogni processo è figlio di qualche altro processo** e vengono mantenute le informazioni di parentela (il padre può conoscere la terminazione di un figlio e se il padre termina anche i figli terminano)

Scheduling

Lo **scheduling** è l'attività mediante la quale il SO sceglie a quali processi assegnare la CPU e per quanto tempo:

- Breve termine: sceglie tra i processi pronti quello a cui assegnare la CPU quando questa diventa libera, cosa che succede se un altro processo termina, si sospende (I/O, fork, wait for interrupt, ...) o viene fatta preemption (causa errori o termine del tempo assegnato)
- Medio termine: si occupa di compiere scelte nell'ambito dello swap dei processi, ossia il trasferimento dei processi in memoria secondaria quando quella principale risulta piena, e viceversa quando si libera spazio
- Lungo termine: importante solamente nei sistemi multiprogrammati di tipo batch, si occupa, dato un insieme di processi nuovi, di scegliere quali ammettere in memoria principale (deve costruire un mix corretto tra processi io-bound e cpu-bound), più in generale gestisce il grado di multiprogrammazione (il numero dei processi attualmente portati avanti)

Per facilitare gli algoritmi di scheduling è sempre presente un processo pronto fittizio (dummy), con priorità più bassa, che, quando eseguito, non fa nulla



Metriche per la valutazione di algoritmi di scheduling (a breve termine):

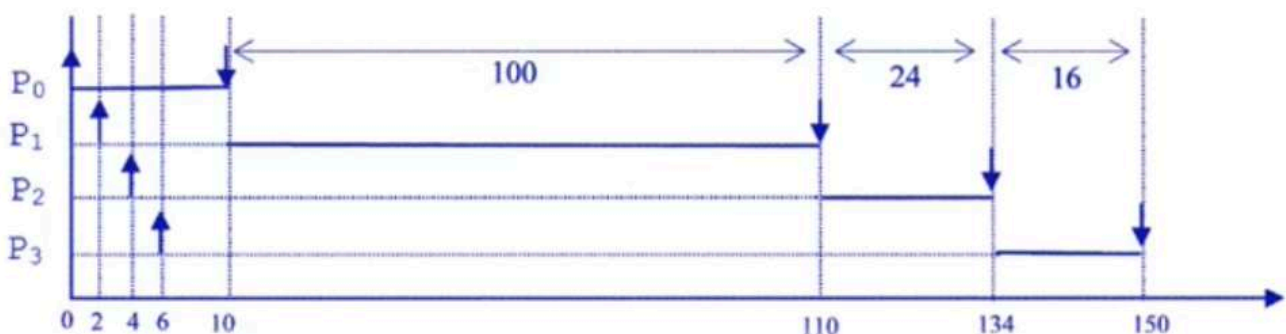
- Per sistemi batch:
 - Utilizzo della CPU in un'unità di tempo (da massimizzare)
 - Tempo medio di completamento (turnaround time): valor medio dei tempi di risposta, ossia degli intervalli temporali che intercorrono tra quando i singoli processi entrano per la prima volta in coda pronti (istante di richiesta) e quando gli stessi terminano le loro esecuzioni
 - Produttività (throughput rate): numero di processi completati in un'unità di tempo (coincide con l'inverso del turnaround time)
- Per sistemi time-sharing:
 - Tempo medio di attesa: valor medio della somma degli intervalli di tempo nei quali i processi attendono in coda pronti
- Per sistemi in tempo reale:
 - Soft: minimizzare il numero di processi che violano la deadline
 - Hard: garantire che tutti i processi rispettino la deadline

Algoritmi di scheduling (a breve termine)

- **FCFS: First Come First Served**

La coda pronti segue logica FIFO, non c'è preemption o priorità

Processo	Istante di arrivo	Durata del CPU burst
P ₀	0	10
P ₁	2	100
P ₂	4	24
P ₃	6	16



- Turnaround time:

$$[(10 - 0) + (110 - 2) + (134 - 4) + (150 - 6)]/4 = 98$$

- Tempo medio di attesa:

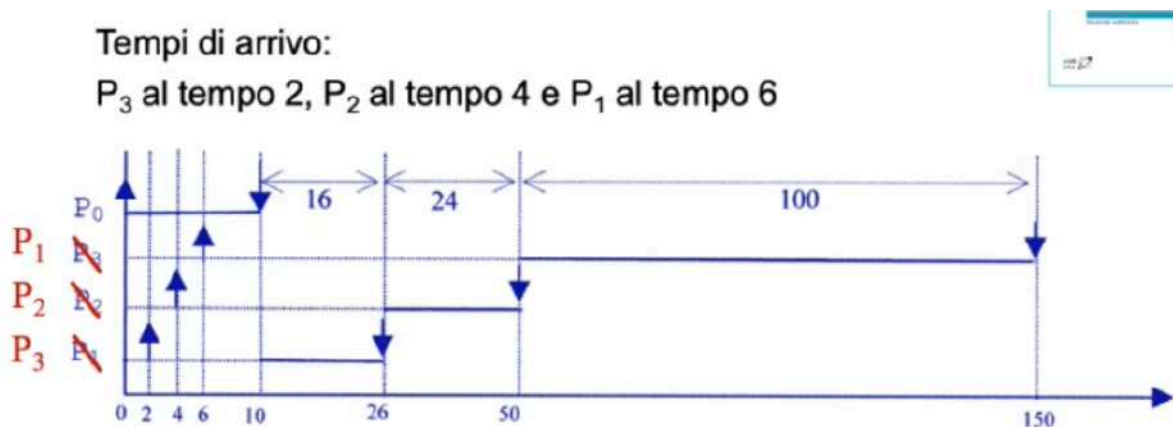
$$[(0) + (10 - 2) + (110 - 4) + (134 - 6)]/4 = 60.5$$

Supponendo P0 fisso, se variava l'ordine di arrivo degli altri processi, ad esempio se P1 arrivava per ultimo, allora diminuivano entrambi i valori misurati, le metriche di valutazione sono dunque del tutto aleatorie ed in pratica l'algoritmo è utile soltanto per sistemi batch

- **SJF: Shortest Job First**

Supponendo che ogni processo abbia un tempo di completamento stimato (fisso) all'interno del suo descrittore di processo, la coda pronti viene gestita usando come priorità l'inverso di tale tempo. L'algoritmo è ottimo (tra i non preemptive) nel senso del tempo medio di attesa, tuttavia stimare il tempo di completamento non è banale e c'è possibilità di **starvation** per processi lunghi se ne arrivano di corti molto frequentemente

Quest'ultimo problema può essere aggiustato tramite il meccanismo dell'**ageing**, ossia mettendo un tempo massimo di attesa che ogni processo può aspettare, dopo il quale la sua priorità diventa massima, in questo modo però le metriche dell'algoritmo tornano a dipendere dal particolare ordine dei processi



- Turnaround time:

$$[(10 - 0) + (150 - 6) + (50 - 4) + (26 - 2)]/4 = 56$$

- Tempo medio di attesa:

$$[(0) + (50 - 6) + (26 - 4) + (10 - 2)]/4 = 18.5$$

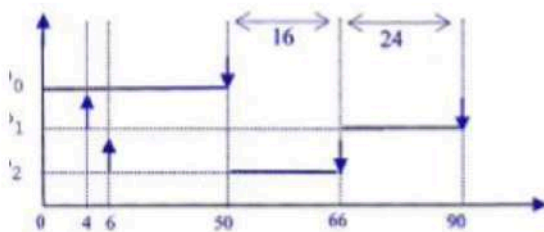
- **SRTF: Shortest Remaining Time First**

Supponendo che ogni processo abbia un tempo di completamento residuo stimato (aggiornato ogni esecuzione) all'interno del suo descrittore di processo, la coda pronti viene gestita, in maniera preemptive, usando come priorità l'inverso di tale tempo.

Nota: Anche questo algoritmo soffre di starvation e si può risolvere come visto per SJF

Processo	Istante di arrivo	Durata del CPU burst
P_0	0	50
P_1	4	24
P_2	6	16

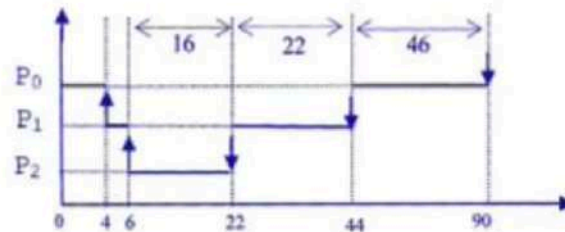
Turnaround medio: 65.3
Tempo medio di attesa: 35.3



SJF

(Fig. 2.15)

Turnaround medio: 48.6
Tempo medio di attesa: 18.6



SRTF

(Fig. 2.16)

La stima del tempo di completamento residuo può essere calcolata come media esponenziale dei tempi in cui il processo è stato in esecuzione (contano tanto meno quanto più sono vecchi):

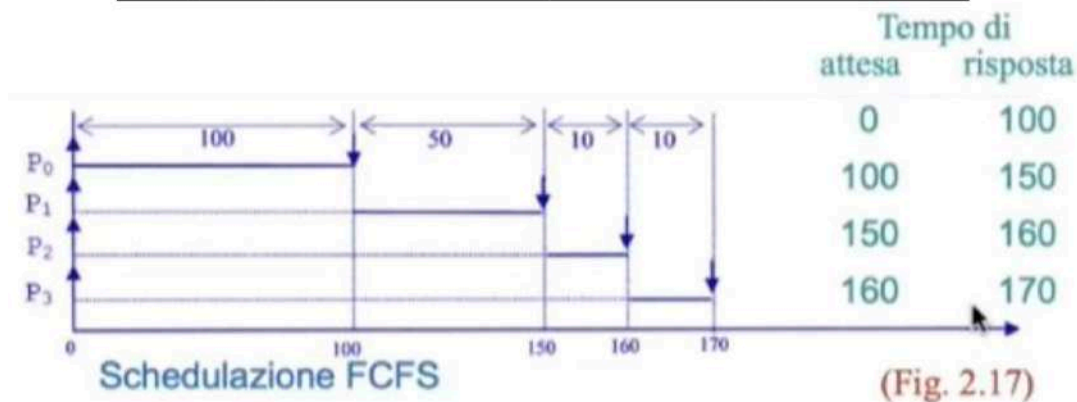
$$S_{n+1} = a \cdot t_n + (1 - a) \cdot S_n$$

- S_0 : Stima iniziale assegnata dal SO
- t_n : Tempo effettivo dell' n -esima esecuzione
- $0 \leq a \leq 1$: (tipicamente $1/2$)
 - Se 0 tengo solo conto della stima iniziale
 - Se 1 tengo solo conto dell'esecuzione precedente

- **RR: Round Robin**

È il classico algoritmo per sistemi time sharing (processi per lo più interattivi), è come FCFS (inserimento in testa) ma preemptive dopo ogni "quanto" di tempo, ha un basso overhead ed è molto potente

Processo	Istante di arrivo	Durata del CPU burst
P_0	0	100
P_1	0	50
P_2	0	10
P_3	0	10



Tempo medio di attesa $\propto N^\circ$ medio di processi pronti \times Quanto

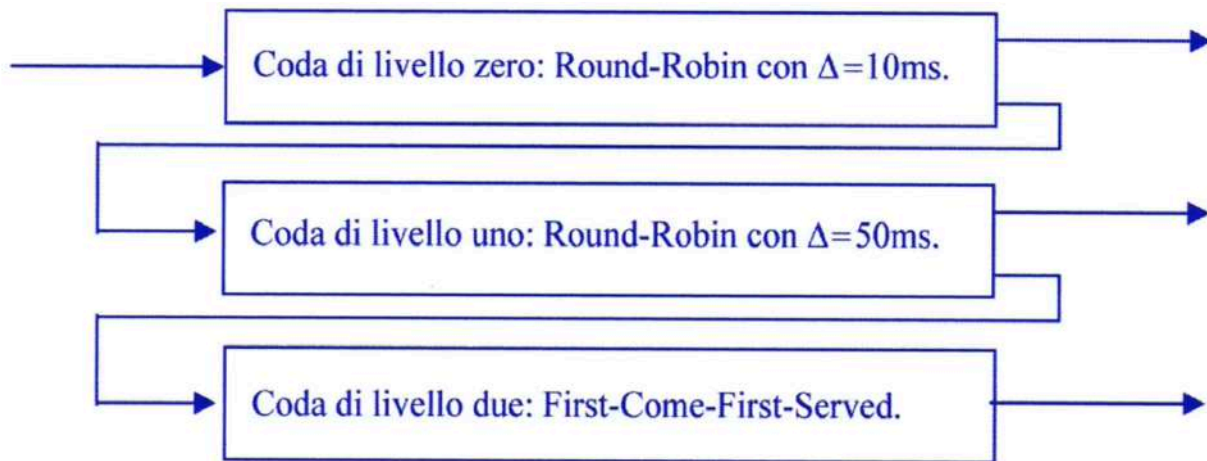
Di conseguenza più il quanto di tempo è piccolo più diminuisce il tempo di attesa medio e il turnaround time, ma più aumenta l'overhead

Schedulazione (a breve termine) a code multiple

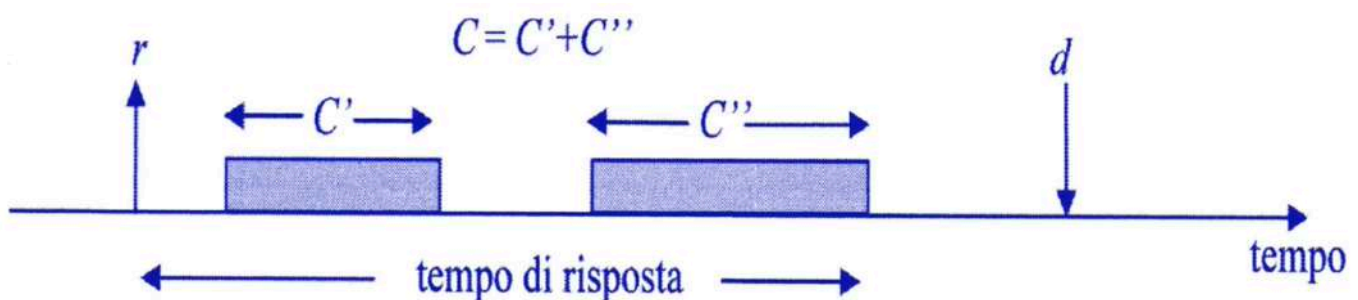
È buona norma avere più code di processi pronti organizzate per priorità di scheduling, in modo da poter usare algoritmi diversi per code diverse: la prima coda (0) ha priorità più alta e la $N + 1_{esima}$ coda può essere schedulata solo se la N_{esima} è vuota, di conseguenza:

- Le code con priorità più basse possono soffrire di starvation, problema che si risolve nuovamente tramite ageing
- Se un processo appartenente alle code più basse è lungo, e la coda non usa un algoritmo di tipo preemptive, si ha comunque un problema, che può essere risolto introducendo una preemption quando arriva un nuovo processo in una coda a priorità più alta (il processo interrotto lo rimetterei in testa alla sua coda)

Tipicamente vorremmo mettere i processi io-bound nelle code a priorità più alta e quelli cpu-bound in quelle a priorità più bassa, facendoli andare avanti quando c'è tempo, tuttavia non sempre è facile distinguere tra queste tipologie di processi, possiamo dunque implementare un feedback: inseriamo tutti i processi nella prima coda, chi non si sospende (o termina) prima di X ms viene messo nella coda successiva e così via...



Schedulazione (a breve termine) dei sistemi in tempo reale hard



I parametri che caratterizzano un processo i di un sistema in tempo reale hard sono:

- t_i = periodo: ogni quanto torna in coda pronti
- d_i = deadline: entro quanto tempo deve terminare (sicuramente $\leq t_i$, possiamo approssimarla con t_i)
- c_i = cpu burst (stima nel caso peggiore)

Il sistema risultante dall'insieme di tutti i processi periodici è a sua volta un sistema periodico con periodo pari a $T = mcm(t_i)$, tale sistema è detto schedulabile se non c'è overflow, ossia se tutti i processi finiscono entro la propria deadline

RM: Rate monotonic

La coda dei processi pronti è organizzata, con preemption, assegnando ad ogni processo priorità P (statica) proporzionale all'inverso del proprio periodo (deadline)

$$\begin{aligned}
 P_a &= t_a = 2 ; c_a = 1 \\
 P_b &= t_b = 5 ; c_b = 2 \\
 T &= \text{mcru}(2, 5) = 10 \\
 p(P_a) &> p(P_b)
 \end{aligned}$$

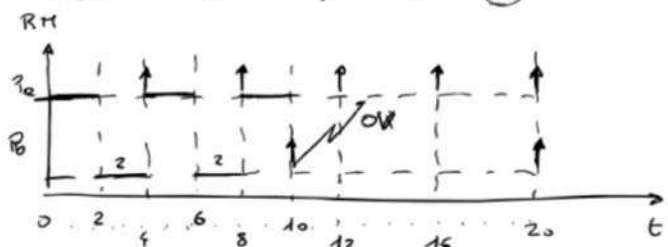


Il sistema risulta schedulabile, di fatto RM è matematicamente ottimo per la classe degli algoritmi a priorità statica dei sistemi a tempo reale hard. È anche presente (in questo caso) un po' di idle, che nei sistemi reali fa bene in quanto, oltre l'overhead dato dal cambio di contesto, ci sono spesso task asincroni da svolgere

EDF: Earliest Deadline First

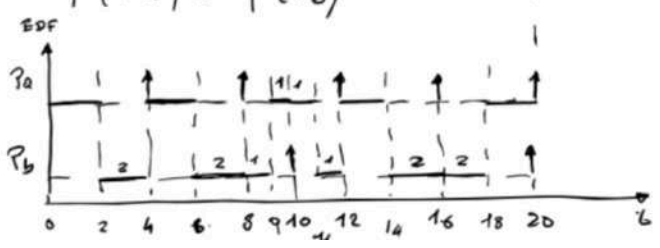
$$\begin{aligned}
 P_a &= t_a = 4 ; c_a = 2 \\
 P_b &= t_b = 10 ; c_b = 5
 \end{aligned}$$

$$U = \frac{2}{4} + \frac{5}{10} = 1 \quad T = 20$$



NON SCHED !!

EARLIEST-DEADLINE-FIRST (EDF)



SCHED !

Il sistema risulta schedulabile, di fatto EDF è matematicamente ottimo per la classe degli algoritmi a priorità dinamica dei sistemi a tempo reale hard (se un sistema non è schedulabile con EDF non lo è con nessun altro algoritmo, nemmeno RM)

In generale in un sistema a tempo reale hard con N processi, ogni processo esegue $k_i = \frac{T}{t_i}$ volte in un periodo (complessivo del sistema), il tempo di cpu totale è dunque:

$$k_1 \cdot c_1 + \dots + k_N \cdot c_N = \frac{T}{t_1} \cdot c_1 + \dots + \frac{T}{t_N} \cdot c_N = T \cdot \sum_{i=1}^N \frac{c_i}{t_i} \leq T$$

$$U = \sum_{i=1}^N \frac{c_i}{t_i} \text{ è definito fattore di ottimizzazione:}$$

- $U \leq 1$ è condizione necessaria e sufficiente perchè il sistema sia schedulabile (tramite EDF)

- $U \leq N \cdot (\sqrt[N]{2} - 1)$ è condizione sufficiente perchè il sistema sia schedulabile tramite RM

Dato che $\lim_{N \rightarrow \infty} N(\sqrt[N]{2} - 1) = \ln 2 \approx 0.693147$, in generale sono (quasi) sicuro di poter applicare RM per $U \leq 0.7$

Thread

I processi si distinguono in:

- Task (processi pesanti): processo standard, dotato di un descrittore di processo, di uno spazio di memoria virtuale privato e di risorse hardware o software
- Thread (processi leggeri): rappresentano un flusso di esecuzione interno ad un processo (pesante), con il quale condividono le risorse principali (facile scambio di informazioni e cambio di contesto), hanno tuttavia un proprio stack ed un proprio descrittore di processo (con meno roba)

A livello utente il **multithreading** viene introdotto come paradigma di programmazione per semplificare la scrittura di software, se poi i thread di uno stesso processo vengono effettivamente visti come entità separate e dunque eseguiti in parallelo, oppure vengono semplicemente visti tutti come lo stesso processo pesante e dunque eseguiti in modo concorrente dipende (oltre a chiaramente dal numero dei processori) dall'architettura del calcolatore

3. Sincronizzazione tra processi

Interazione tra processi

Due processi sono detti **concorrenti** se si sovrappongono nel tempo, ossia se la prima operazione di uno comincia prima che termini l'ultima dell'altro, questi possono essere:

- Processi indipendenti: l'esecuzione di uno non è influenzata da quella dell'altro (non si scambiano dati, non usano le stesse risorse, ...), questo implica la proprietà della **riproducibilità** (a parità delle stesse CI ottengo sempre lo stesso scheduling nel tempo)
- Processi interagenti: l'esecuzione di uno è influenzata da quella dell'altro

Ci interessiamo dei processi interagenti, e in particolare di due problemi che si possono verificare tra di essi:

- Competizione: si ha quando due processi richiedono la stessa risorsa, il tipo di sincronizzazione è detta indiretta o implicita perchè non viene scritta nel codice dei processi, bensì viene garantita dal SO attraverso la mutua esclusione (*primitive wait e signal*)
- Cooperazione: si ha quando due processi eseguono un'attività comune mediante lo scambio di informazioni, il tipo di sincronizzazione è detta diretta o esplicita perchè viene scritta nel codice dei processi → l'ordine è importante!
 - Se i processi hanno **memoria globale**, ossia spazio di indirizzamento con parti a comune, allora comunicano usando tale spazio in mutua esclusione (*primitive wait e signal*)
 - Se i processi hanno **memoria locale**, ossia spazio di indirizzamento completamente privato, allora comunicano tramite messaggi (*primitive send e receive*), meccanismo del SO che però introduce più overhead (banalmente bisogna ricopiare i messaggi in più memorie locali diverse)

Quello che vogliamo evitare in ogni caso è *interferenza*, un tipo di interazione tra processi che porta ad errori dipendenti dell'ordine di esecuzione di questi

Problema della mutua esclusione (competizione)

Per risolvere il problema della mutua esclusione bisogna isolare le **sezioni critiche**, ossia blocchi di istruzioni tramite le quali il codice accede a strutture dati condivise, vediamo dunque alcune possibili soluzioni:

- Stabilire a livello di scheduler un ordine per accedere alla sezione critica (es: utilizzare un algoritmo di scheduling non preemptive con priorità statica)

La soluzione è chiaramente limitante ma perlomeno funziona in sistemi a singolo processore

- Disattivare le interruzioni prima di entrare in una sezione critica, in modo da rendere tale sezione atomica

Anche qui la soluzione non è ottima ma perlomeno funziona in sistemi a singolo processore


- Ciclo while

Soluzione (non corretta) al problema della mutua esclusione

occupato = 1 → risorsa occupata
occupato = 0 → risorsa libera

P ₁	P ₂
prologo: while (occupato==1) ; occupato=1 ; <sezione critica A>; epilogo: occupato=0 ;	prologo: while (occupato==1) ; occupato=1 ; <sezione critica B>; epilogo: occupato=0 ;

Oltre al fatto che, per via dell'attesa attiva, la soluzione non sarebbe ottima, questa non è nemmeno corretta: il prologo infatti non è atomico



- Prologo ed epilogo lock-unlock utilizzando l'istruzione test-and-set

Uso di istruzione tipo *test-and-set* per realizzare le primitive lock e unlock



lock(x):

```
TSL registro, x (copia x nel registro e pone x =1)
CMP registro, 0 (il valore di x era 0 ?)
JNE lock       (se no, ricomincia il ciclo)
RET           (ritorna al chiamante; accesso alla sezione critica)
```

unlock(x):

```
MOVE x, 0      (inserisce 0 in x)
RET           (ritorna al chiamante)
```

Se il processore dispone dell'istruzione TSL, questa di fatto blocca il bus per due cicli di clock, garantendo l'atomicità del prologo, tuttavia rimangono le attese attive, la soluzione non è quindi ottima (va bene lo stesso per sezioni critiche molto brevi), però funziona anche nel caso di più processori

La soluzione corretta è quella di utilizzare i semafori, in particolare i **mutex** (semafori con valori iniziali unitari), insieme ad alcune delle soluzioni precedenti:

```

struct des_sem {
    // Numero di processi che possono eseguire la wait senza sospendersi
    int counter;
    des_proc* queue;
}

void sem_wait(des_sem *sem) {
    if (sem->counter == 0) {
        /// Sospendi il processo ed inseriscilo in coda a queue
    }
    else sem->counter--;
}

void sem_signal(des_sem *sem) {
    if (!queue) {
        /// Estrai in testa da queue ed inseriscilo in coda pronti
    }
    else sem->counter++;
}

// Le funzioni sem_wait e sem_signal devono essere delle primitive di
// sistema perchè devono essere eseguite in modo atomico (e devono
// manipolare le code dei processi), inoltre, in sistemi con più
// processori, queste devono essere racchiuse a loro volta da lock
// ed unlock per garantirne la mutua esclusione

```

Problema della comunicazione (cooperazione)

Abbiamo visto che questo problema dipende dalla modalità attraverso la quale interagiscono i processi, per quanto riguarda processi a memoria privata ci limitiamo a dire che la soluzione è sempre quella di utilizzare le primitive **send** e **receive**:

```

send(destinazione, messaggio)
receive(origine)

```

- Ogni messaggio può essere schematizzato nel modo seguente:



- Se i processi risiedono nello stesso calcolatore destinazione ed origine possono essere il PID, mentre se i processi esistono su calcolatori diversi destinazione ed origine possono essere gli indirizzi ip con le relative porte, destinazione ed origine possono anche essere riferiti ad un insieme di processi
- `send` e `receive` sono dette **simmetriche** se produttore e consumatore si nominano a vicenda, **asimmetriche** se il produttore nomina il consumatore ma non viceversa (al consumatore non interessa l'origine, ossia da quale produttore arriva il messaggio)
- La `send` può essere sia bloccante (sincrona) che non (asincrona), in particolare può sospendere il processo fino a quando il messaggio non arriva a destinazione, oppure eventualmente fino a quando non viene anche consumato, in quest'ultimo caso richiede una sorta di ack con un parametro di ritorno e prende il nome di **Remote Procedure Call**
- Anche la `receive` può essere sia bloccante che non

Per quanto riguarda i processi a memoria condivisa invece analizziamo in dettaglio le soluzioni a diversi scenari comuni:

- **Produttori e consumatori**

```
// Caso in cui esiste un produttore ed un consumatore, che si
//  scambiano messaggi in un buffer unitario
des_sem spazio_disponibile = sem_init(1);
des_sem messaggio_disponibile = sem_init(0);

/* Processo produttore */
/// Produci il messaggio
sem_wait(spazio_disponibile);
/// Deposita il messaggio nel buffer
sem_signal(messaggio_disponibile);

/* Processo consumatore */
sem_wait(messaggio_disponibile);
/// Leggi il messaggio
sem_signal(spazio_disponibile);
/// Consuma il messaggio

// Nel caso di buffer più grande di uno, gestito in modo circolare,
//  cambia solamente lo spazio_disponibile, che viene inizializzato
//  ad N, la mutua esclusione alle singole celle del buffer è
//  garantita per costruzione, se inoltre abbiamo più produttori
//  e più consumatori la mutua esclusione al buffer non è più
//  garantita, va implementata usando un mutex prima di depositare
//  e leggere ogni messaggio
```

- **Lettori e scrittori**

```
// # Prima variante
// Nessun lettore viene fatto attendere se il buffer è già in
// possesso ad un lettore (favorisce i lettori e gli scrittori
// possono andare in starvation)

des_sem buffer_libero = sem_init(1);
des_sem mutex = sem_init(1);
int lettori = 0;

/* Processo scrittore */
sem_wait(buffer_libero);
/// Scrive il buffer
sem_signal(buffer_libero);

/* Processo lettore */
sem_wait(mutex);
lettori++;
if (lettori == 1)
    sem_wait(buffer_libero);
sem_signal(mutex);
/// Legge il buffer
sem_wait(mutex);
lettori--;
if (lettori == 0)
    sem_signal(buffer_libero);
sem_signal(mutex);
```

```

// # Seconda variante
// Nessuno scrittore viene fatto aspettare se il buffer è già in
// possesso ad uno scrittore (favorisce gli scrittori ed i lettori
// possono andare in starvation)

int lettori = 0;
int scrittori = 0;
des_sem rmutex = sem_init(1);
des_sem wmutex = sem_init(1);
des_sem lettura_libera = sem_init(1);
des_sem buffer_libero = sem_init(1);

/* Processo scrittore */
sem_wait(wmutex);
scrittori++;
if (scrittori == 1)
    sem_wait(lettura_libera);
sem_signal(wmutex);

sem_wait(buffer_libero);
/// Scrive il buffer
sem_signal(buffer_libero);

sem_wait(wmutex);
scrittori--;
if (scrittori == 0)
    sem_signal(lettura_libera);
sem_signal(wmutex);

/* Processo lettore */
sem_wait(lettura_libera);
sem_wait(rmutex);
lettori++;
if (lettori == 1)
    sem_wait(buffer_libero);
sem_signal(rmutex);
sem_signal(lettura_libera);
/// Legge il buffer
sem_wait(rmutex);
lettori--;
if (lettori == 0)

```

```
sem_signal(buffer_libero);  
sem_signal(rmutex);
```

- I cinque filosofi



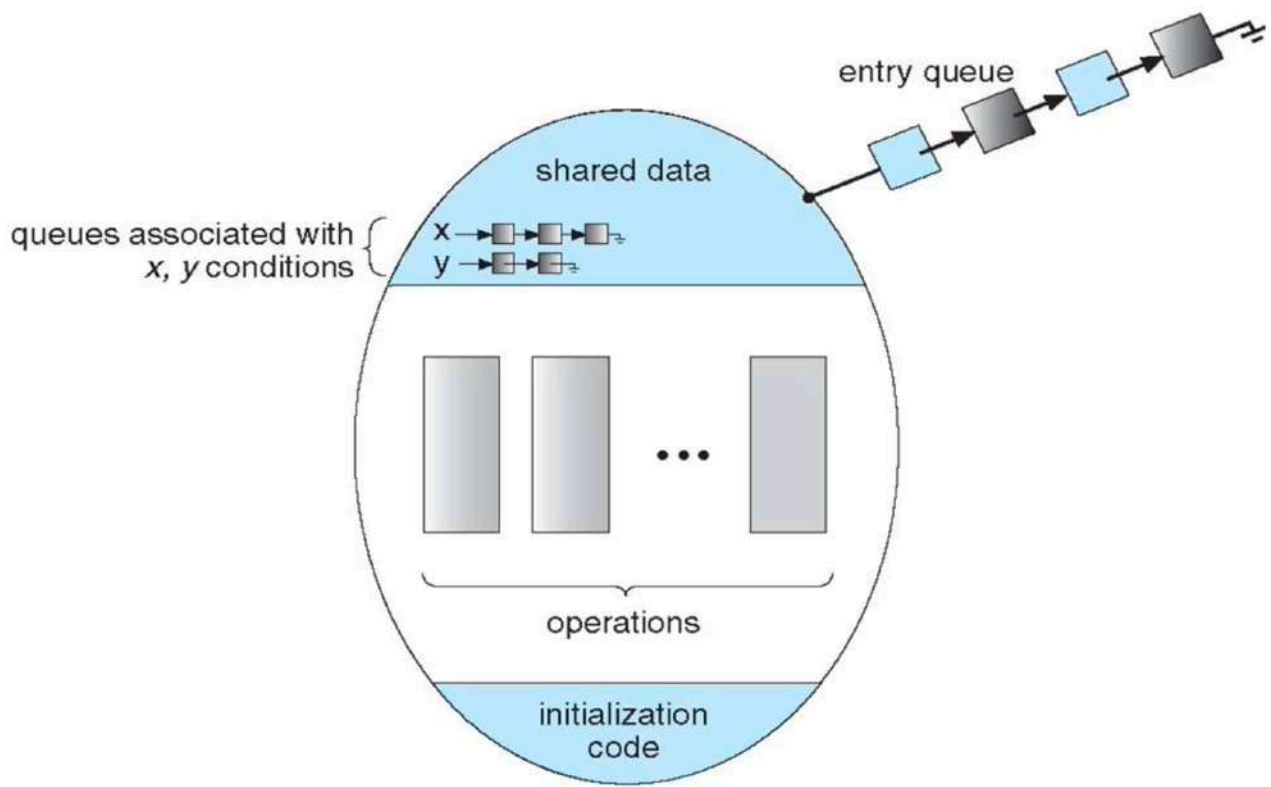
- Philosophers spend their lives thinking and eating
- Don't interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
 - Need both to eat, then release both when done
- In the case of 5 philosophers
 - Shared data
 - ▶ Bowl of rice (data set)
 - ▶ Semaphore **chopstick** [5] initialized to 1

Supponiamo che ogni filosofo si procuri prima la bacchetta a destra e poi quella a sinistra (devo avere un comportamento standard)

- La soluzione tramite semafori in questo caso presenta il problema della possibile attesa ciclica, infatti:

```
/* Filosofo nella posizione i */  
sem_wait(chopstick[i]);  
sem_wait(chopstick[(i+1) % N])  
/// Mangia  
sem_signal(chopstick[i]);  
sem_signal(chopstick[(i+1) % N])  
/// Pensa
```

- Soluzione migliore è quella ottenuta tramite **monitor**, ossia un'astrazione che comprende un set di operazioni predefinite su strutture dati condivise progettate per essere eseguite in mutua esclusione (il programmatore non deve preoccuparsi di nulla):




```

monitor DiningPhilosophers {

    // Strutture dati condivise
    enum {
        THINKING,    // Non ha bacchette e non ne cerca
        HUNGRY,       // Non ha bacchette ma vorrebbe prenderle
        EATING        // Ha due bacchette e sta mangiando
    } state[5];

    // Variabili condizione, dotate di metodi:
    // .wait()      (sempre bloccante, come un sem inizializzato a 0)
    // .signal()
    condition self[5];

    // Inizializzazione delle strutture dati condivise
    void init() {
        for (int i = 0; i < 5; i++)
            state[i] = THINKING;
    }

    // Procedure sulle strutture dati condivise, che supponiamo essere
    // eseguite in mutua esclusione (lo garantisce l'implementazione
    // del monitor stessa, che vediamo dopo)
    void pickup(int i) {
        state[i] = HUNGRY;
        try_to_give_chopsticks(i);
        // Se non ha trovato bacchette disponibili aspetta
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i) {
        state[i] = THINKING;
        // Controlla se liberando la sua bacchetta si risveglia qualcuno
        try_to_give_chopsticks((i + 4) % 5);
        try_to_give_chopsticks((i + 1) % 5);
    }

    void try_to_give_chopsticks(int i) {
        // Se il filosofo i ha fame e quelli a fianco non stanno
        // mangiando, allora gli dà le loro bacchette e lo risveglia
        if (state[i] == HUNGRY
            && state[(i + 4) % 5] != EATING
            && state[(i + 1) % 5] != EATING) {

```

```
        state[i] = EATING;
        self[i].signal();
    }
}

// Problema: possibile starvation dei filosofi "lontani"
```

Vediamo ora come implementare il meccanismo dei monitor utilizzando i semafori e lavorando in **politica signal-and-wait** (se il processo P risveglia Q allora P si blocca):

```

// Semaforo per garantire la mutua esclusione delle procedure
des_sem mutex = sem_init(1);
// Coda di processi pronti dentro al monitor
des_sem next = sem_init(0);
int next_count = 0;

/* Ogni procedura viene wrappata dal seguente codice */
sem_wait(mutex);
/// Procedura
// Se c'è qualche processo pronto dentro al monitor allora lo
// sveglio, altrimenti permetto ad altri fuori di entrare
if (next_count > 0)
    sem_signal(next);
else
    sem_signal(mutex);

/* Per ogni variabile condizione (esempio con una generica x) */
des_sem x_sem = sem_init(0);
// Processi sospesi sulla variabile condizione X
int x_count = 0;

/* x.wait(): */
x_count++;
if (next_count > 0)
    sem_signal(next);
else
    sem_signal(mutex);
sem_wait(x_sem);
// ... Quando mi risvegliano dai processi in attesa su X ...
x_count--;

/* x.signal() */
if (x_count > 0) {
    // Politica signal-and-wait: io mi metto tra i processi
    // pronti e faccio andare uno bloccato su X
    next_count++;
    sem_signal(x_sem);
    sem_wait(next);
    // ... Quando mi risvegliano dai processi pronti ...
    next_count--;
}

```

Se molti processi sono in coda su una variabile condizione X e viene fatta la signal chi va avanti ? FCFS va bene, ma potrebbe farmi piacerebbe avere una priorità:
`x.wait(priority)`

Per implementare invece la **politica signal-and-continue** si potrebbe fare:

```
/* x.wait(): */
x_count++;
if (next_count > 0)
    sem_signal(next);
else
    sem_signal(mutex);
sem_wait(x_sem);
// ... Quando mi risvegliano dai processi in attesa su X ...
x_count--;
next_count++;
sem_wait(next);
// ... Quando mi risvegliano dai processi pronti ...
next_count--;

/* x.signal() */
if (x_count > 0) {
    sem_signal(x_sem);
}
```

Vediamo ora un **monitor per allocare una singola risorsa** (di fatto una `sem_wait()` ma complicandosi la vita):

```

monitor ResourceAllocator {
    bool busy;
    condition x;

    void init() {
        busy = false;
    }

    void acquire(int time) {
        if (busy)
            x.wait(time); // time usato come priorità
        busy = true;
    }
    void release() {
        busy = false;
        x.signal();
    }
}

/* Thread che utilizza la risorsa */
ResourceAllocator.acquire();
/// Sezione critica
ResourceAllocator.release();

```

Stavolta funziona usare un bool busy perchè adesso viene tutto eseguito in mutua esclusione

Deadlock

Rivediamo brevemente il protocollo utilizzato fin'ora per accedere ad una risorsa:

- Richiesta (Prologo: `sem_wait()` o `monitor.acquire()`)
- Sezione critica
- Rilascio (Epilogo: `sem_signal()` o `monitor.release()`)

In un dato sistema con $P = \{P_1, \dots, P_n\}$ processi e $R = \{R_1, \dots, R_m\}$ risorse, ognuna delle quali con W_i istanze, **condizioni necessarie ma non sufficienti per avere un deadlock** sono:

- Mutua esclusione: ogni risorsa può essere utilizzata da al più un processo alla volta
- No preemption: il sistema operativo non può revocare risorse ai processi

- Possesso e attesa: almeno un processo che possiede una risorsa vuole acquisirne un'altra, posseduta da altri processi
- Attesa circolare: un insieme di processi aspettano circolarmente ognuno la risorsa dell'altro (in altre parole ci sono cicli nel grafo)

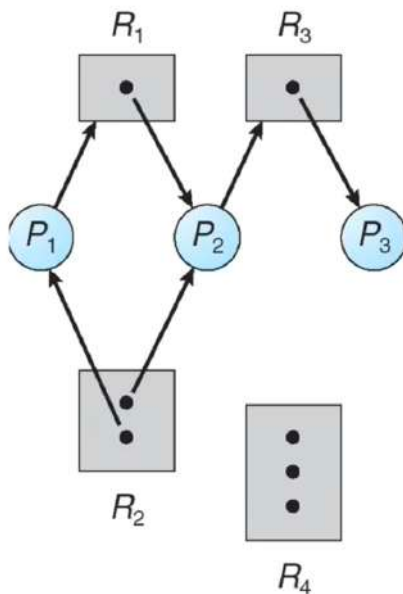
Le prime tre sono praticamente date per scontate

Per studiare il deadlock è necessario definire il **grafo di allocazione delle risorse**, tale per cui:

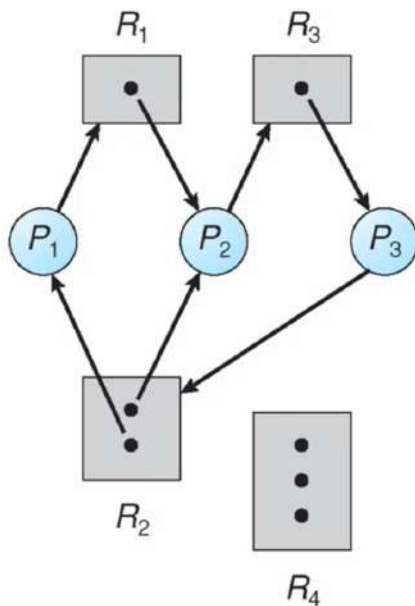
- L'insieme V dei vertici è costituito dagli n processi P e dalle m risorse R
- L'insieme E degli archi è partizionato come segue:
 - Arco di assegnamento $R_j \rightarrow P_i$
 - Arco di richiesta $P_i \rightarrow R_j$

Esempi:

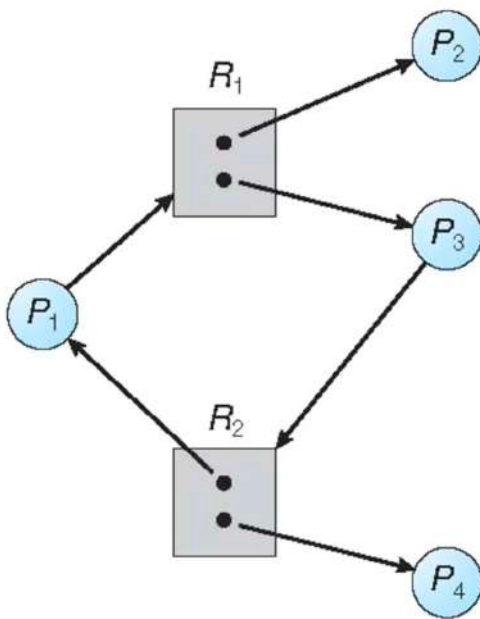
- Non c'è deadlock perchè non ci sono cicli nel grafo



- C'è deadlock perchè sono verificate tutte le condizioni necessarie ed inoltre tutte le istanze delle risorse coinvolte nei due cicli sono possedute da processi appartenenti ai due cicli



- Non c'è un deadlock perchè, sebbene le condizioni necessarie siano verificate, non tutte le istanze delle risorse coinvolte nel ciclo sono possedute da processi appartenenti al ciclo, infatti R_1 ed R_2 sono possedute da processi non in attesa, che prima o poi termineranno



Di conseguenza possiamo affermare che l'ultima delle quattro condizioni necessarie diventa anche sufficiente quando siamo di fronte a risorse a singola istanza!

Vediamo adesso i possibili metodi per affrontare il deadlock:

- **Ignorare il fatto che esista**, come avviene nella maggior parte dei sistemi operativi, e delegare la gestione del deadlock ai particolari linguaggi di programmazione
- Garantire che non si arrivi mai ad una situazione di deadlock
 - **Prevenzione statica** (prevention)
 - **Prevenzione dinamica** (avoidance)
- **Tollerare il deadlock ed in caso gestirlo** (detection)

In particolare ci soffermiamo sugli ultimi tre:

1. Prevenzione statica

Assicuro a priori che il deadlock non avvenga mai restringendo le modalità con cui ogni processo può richiedere una risorsa:

- Mutua esclusione: non è possibile allentarla
- Possesso e attesa: dovremmo garantire che un processo non possa richiedere una nuova risorsa se ne possiede già una, ma allora il programmatore dovrebbe cambiare il codice, oppure che richieda tutte le risorse di cui ha bisogno all'inizio, tuttavia, anche se le conoscessi, sarebbe non ottimale dal punto di vista dell'efficienza, eliminerei di fatto la concorrenza tra processi che vogliono la stessa risorsa
- No preemption: Potrei introdurla, ad esempio, se un processo richiedesse una risorsa che non gli può essere assegnata immediatamente, allora potrei sospenderlo e liberare tutte le risorse da esso possedute, per poi riassegnargliele una volta terminata l'attesa, questo però implicherebbe di dover mantenere lo stato di tali risorse (perlomeno questo meccanismo sarebbe trasparente al programmatore)
- Attesa circolare: Potremmo evitare a priori la formazione di cicli ordinando le risorse ed imponendo che il processo che già possiede la risorsa R_i non possa richiedere la risorsa R_j dove $j < i$, anche questo però cambierebbe il modo in cui il programmatore deve scrivere il codice

In generale nessuna di queste soluzioni è ottima, però sono tutte fattibili su sistemi embedded, dove sono spesso noti a priori sia i processi che le risorse

2. Prevenzione dinamica

Ogni volta che un processo richiede una risorsa, oltre a controllare il rispettivo semaforo, simulo, **sapendo quante e quali risorse chiederà al massimo ciascun processo**, l'evoluzione del grafo di allocazione delle risorse, se esiste almeno un ordine di esecuzione dei processi che evita con certezza una situazione di deadlock, allora lo stato attuale è detto sicuro e la risorsa è concessa al processo richiedente

Più formalmente, un sistema si trova in uno **stato sicuro** se esiste una sequenza di esecuzione tale per cui ogni processo per terminare abbia bisogno delle risorse che già possiede più quelle possedute dai soli processi precedenti in sequenza di esecuzione (ossia quelli che termineranno prima di lui)

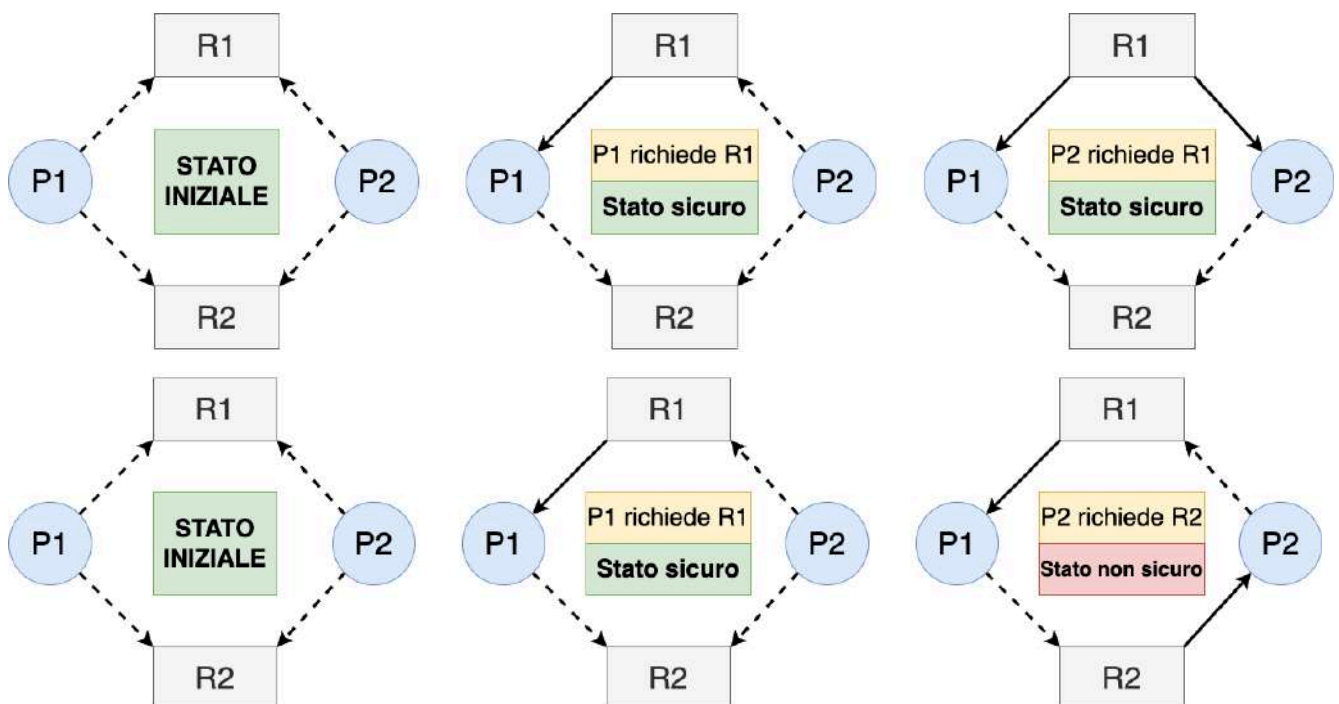
Questa definizione risulterà utile per applicare l'algoritmo del banchiere

Se un sistema si trova in uno **stato non sicuro** non per forza accadrà con certezza una situazione di deadlock, dipende da come interagiscono le risorse ed i processi

Come si controlla di arrivare in uno stato sicuro ?

- Risorse ad istanza singola: ricordando che, per risorse ad istanza singola, condizione sufficiente perchè si verifichi il deadlock è che ci sia un ciclo nel grafo di allocazione delle risorse, allora, perchè una determinata richiesta porti ad uno stato sicuro, basta controllare che, convertendo l'arco di richiesta futura in assegnamento, non si formino cicli

Le richieste future sono modellate da un arco tratteggiato $P_i \dashrightarrow R_j$



Questo algoritmo si può implementare nella primitiva wait dei semafori, ed è interessante perchè per il programmatore non cambia nulla

- Risorse ad istanza multipla: uso l'**algoritmo del banchiere**

Supponiamo di avere N processi ed M risorse, ognuna con una o più istanze, sono necessarie le seguenti strutture dati:

- $Available (M) : Available[j] = k \iff R_j$ ha k istanze libere
- $Allocation (N \times M) : Allocation[i, j] = k \iff P_i$ possiede attualmente k istanze di R_j
- $Max (N \times M) : Max[i, j] = k \iff P_i$ richiede al massimo k istanze di R_j

- $Need (N \times M) : Need[i, j] = k \iff P_i$ richiede ancora k istanze di R_j per terminare

Di conseguenza $need[i, j] = max[i, j] - allocation[i, j]$

Allora l'algoritmo del banchiere è il seguente:

Request = request vector for process P_i . If $Request_i[j] = k$ then process P_i wants k instances of resource type R_j

1. If $Request_i \leq Need_i$ go to step 2. Otherwise, raise error condition, since process has exceeded its maximum claim
2. If $Request_i \leq Available$, go to step 3. Otherwise P_i must wait, since resources are not available
3. Pretend to allocate requested resources to P_i by modifying the state as follows:

$Available = Available - Request_i;$

$Allocation_i = Allocation_i + Request_i;$

$Need_i = Need_i - Request_i;$

- If safe \Rightarrow the resources are allocated to P_i
- If unsafe $\Rightarrow P_i$ must wait, and the old resource-allocation state is restored

Questa implementazione è un caso generale in cui non ho una richiesta alla volta ma un insieme di richieste per ogni tipo di risorsa alla volta (non cambia nulla)

Dove la verifica dello stato sicuro avviene nel seguente modo:

1. Let **Work** and **Finish** be vectors of length m and n , respectively. Initialize:

$Work = Available$

$Finish[i] = false$ for $i = 0, 1, \dots, n-1$

2. Find an i such that both:

(a) $Finish[i] = false$

(b) $Need_i \leq Work$

If no such i exists, go to step 4

3. $Work = Work + Allocation_i$

$Finish[i] = true$

go to step 2

4. If $Finish[i] == true$ for all i , then the system is in a safe state

In realtà per i processi con $Need_i = \vec{0}$ si può mettere direttamente $Finish[i] = true$

L'algoritmo risulta di difficile applicazione pratica poichè assume che siano noti a priori e restino costanti: il numero massimo di risorse che ogni processo intende utilizzare, il numero dei processi, il numero delle risorse di cui il sistema dispone

Prima di vedere la deadlock detection, il terzo ed ultimo argomento sulla quale ci soffermiamo per quanto riguarda il deadlock, vediamo un esempio di applicazione dell'algoritmo del banchiere:

- *Situazione iniziale*

■ 5 processes P_0 through P_4 ;

3 resource types:

A (10 instances), B (5 instances), and C (7 instances)

Snapshot at time T_0 :

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

- *Controllo dello stato attuale*

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	3 3 2, 5 3 2, 7 4 3, 7 4 5, 7 5 5, 10 5 7
P_1	2 0 0	1 2 2	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

P1, P3, P4, P0, P2

- *Richiesta in entrata*

$Request_1 = (1, 0, 2)$

Controllo che $Request_1 \leq Need_1 : (1, 0, 2) \leq (1, 2, 2)$

Controllo che $Request_1 \leq Available_1 : (1, 0, 2) \leq (3, 3, 2)$

$Need_1 = Need_1 - Request_1 = (1, 2, 2) - (1, 0, 2) = (0, 2, 0)$

$Allocation_1 = Allocation_1 + Request_1 = (2, 0, 0) + (1, 0, 2) = (3, 0, 2)$

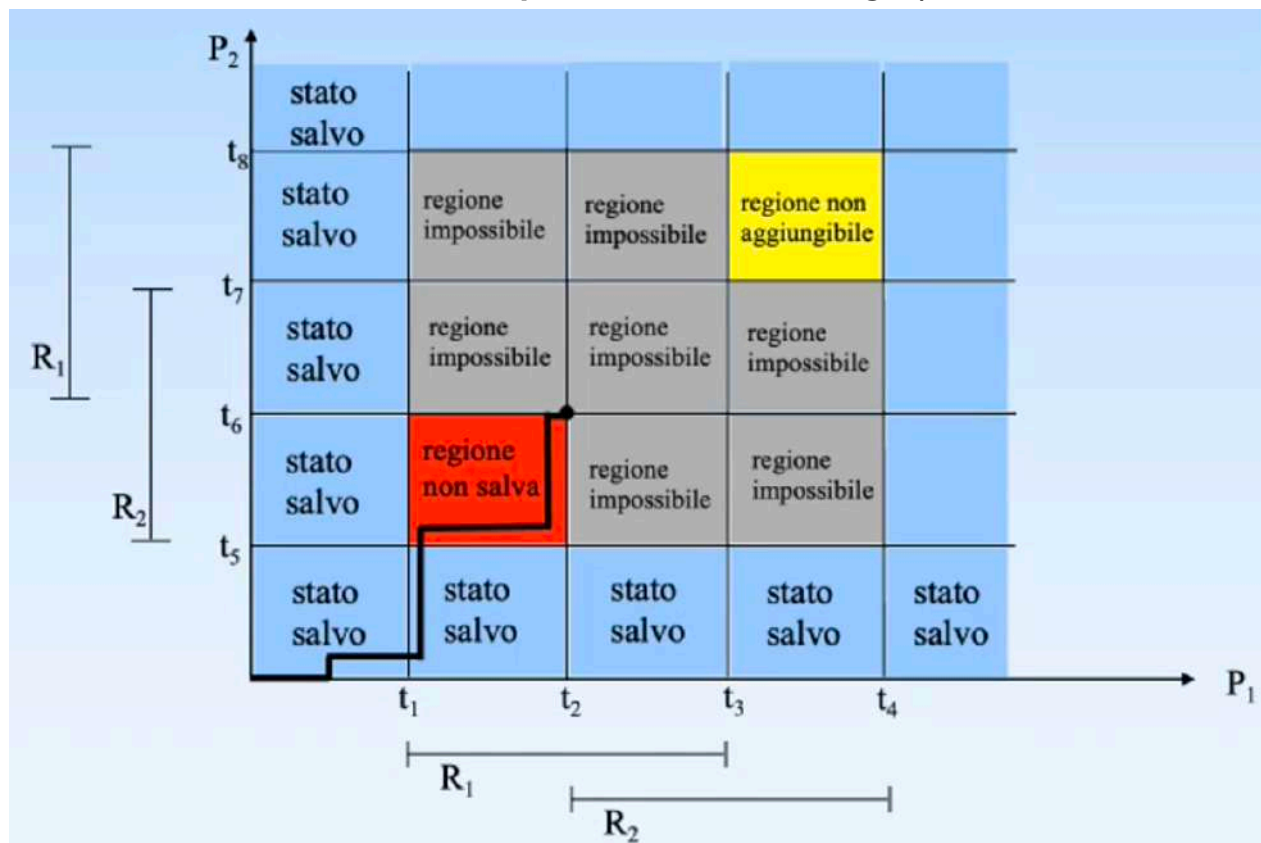
$Available = Available - Request_1 = (3, 3, 2) - (1, 0, 2) = (2, 3, 0)$

- Controllo dello stato ipotetico

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- Executing safety algorithm shows that sequence $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfies safety requirement

Vediamo infine un esempio tratto dal libro di un ulteriore algoritmo di prevenzione dinamica basato sulla conoscenza della **sequenza** di richieste di ogni processo:



Ad esempio il grafico ci dice che P_1 al tempo t_1 richiederà R_1 e la utilizzerà fino al tempo t_3 , la linea nera invece va in orizzontale se in quel momento esegue P_1 ed in verticale se esegue P_2

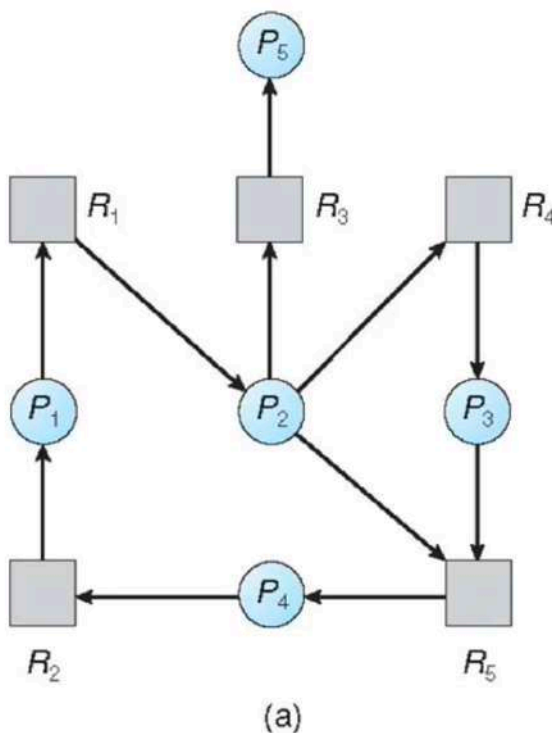
3. Tollerare il deadlock ed in caso gestirlo (detection)

Le risorse vengono sempre concesse ai processi, tuttavia ogni tanto un algoritmo controlla se è avvenuto un deadlock, ed in caso tenta di uscirne. Vediamo per prima cosa due algoritmi per la rilevazione del deadlock, a seconda del tipo di risorse utilizzate dai processi:

- Risorse a singola istanza

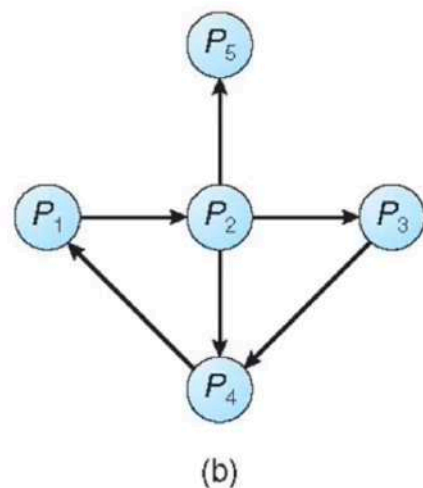
Per questioni di efficienza si usa una versione ridotta del grafo di allocazione detta **grafo di attesa** (*wait-for-graph*), che ha come nodi i processi del sistema e come archi le attese tra i vari processi ($i \rightarrow j \iff P_i$ deve aspettare P_j)

Dato che trovare un ciclo in un grafo è quadratico col numero di nodi, il secondo grafo è più efficiente perchè li dimezza



(a)

Resource-Allocation Graph



(b)

Corresponding wait-for graph

- Risorse a multipla istanza

Supponiamo di avere N processi ed M risorse, ognuna con una o più istanze, per l'algoritmo di controllo del deadlock sono necessarie le seguenti strutture dati:

- *Available* (M) : $Available[j] = k \iff R_j$ ha k istanze libere
- *Allocation* ($N \times M$) : $Allocation[i, j] = k \iff P_i$ possiede attualmente k istanze di R_j
- *Request* ($N \times M$) : $Request[i, j] = k \iff P_i$ richiede attualmente k istanze di R_j

Quest'ultima è una differenza con l'algoritmo del banchiere, che utilizzava *Need* e *Max*

L'algoritmo, molto simile a quello di controllo per uno stato sicuro, è dunque il seguente:

1. Let *Work* and *Finish* be vectors of length m and n , respectively initialize:
 - (a) *Work* = *Available*
 - (b) For $i = 1, 2, \dots, n$, if $Allocation_i \neq 0$, then *Finish*[i] = false; otherwise, *Finish*[i] = true
2. Find an index i such that both:
 - (a) *Finish*[i] == false
 - (b) $Request_i \leq Work$
 If no such i exists, go to step 4
3. *Work* = *Work* + $Allocation_i$
Finish[i] = true
 go to step 2
4. If *Finish*[i] == false, for some i , $1 \leq i \leq n$, then the system is : deadlock state. Moreover, if *Finish*[i] == false, then P_i is deadlocked

Algorithm requires an order of $O(m \times n^2)$ operations to detect whether the system is in deadlocked state

Il punto 1 assume che i processi rilascino le proprie risorse una volta terminati (potrebbe anche essere un processo appena "nato", ma non è comunque un problema)

In realtà al punto 4, come per gli algoritmi di prevenzione dinamica, non si guarda la certezza del deadlock ma la sola possibilità

Rimane da discutere quando mandare in esecuzione l'algoritmo:

- Ogni volta che un processo si sospende: porterebbe, considerando anche le operazioni di recover, ad un overhead troppo grande

- Ogni tot tempo: bisogna farsi un'idea della frequenza con la quale in un dato sistema si verificano deadlock e controllarli di conseguenza
- Quando la CPU viene utilizzata poco

Gli ultimi due, combinati, sono la soluzione migliore

Vediamo brevemente un esempio di questo algoritmo:

■ Snapshot at time T_0 :

	<u>Allocation</u>	<u>Request</u>	<u>Available</u>
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

■ Sequence $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ will result in $Finish[i] = \text{true}$ for all i

Vediamo infine, ed in maniera molto sintetica, come rimediare ad una situazione di deadlock presunto (**deadlock recover**):

- Termino i processi, in particolare:
 - Abortisco tutti i processi appartenenti al deadlock in un colpo solo
 - Abortisco un processo alla volta tra quelli appartenenti al deadlock, ri eseguendo l'algoritmo di detection fino a quando non si risolve, in questo caso i processi possono essere terminati seguendo determinate politiche:
 - Prima quelli a più bassa priorità
 - Prima quelli più recenti
 - Prima quelli che hanno utilizzato meno risorse
 - Prima quelli a cui mancano più risorse per completare
 - Prima quelli io-bound
- Revoco le risorse: seleziono uno o più processi vittima (attenzione alla starvation!) ai quali revocare risorse a seconda della complessità che richiede ripristinare lo stato dei processi pre acquisizione di tali risorse (c'è molto overhead)

È più conveniente la prima opzione (terminare i processi)

4. Gestione della memoria

Introduzione

Perchè serve un meccanismo di gestione della memoria? Secondo il modello di Von Neumann tutti i processi attivi devono stare in memoria principale, questa tuttavia è di dimensioni limitate, e per accomodare tutti i processi può dover essere artificialmente estesa con porzioni di memoria di massa (zone di **swap**)

Come nel caso della CPU, il risultato che vogliamo ottenere è quello di allocare a ciascun processo una **memoria virtuale**, per fare questo ci servono:

- Strutture dati: da definire (descrittore di processo nel caso della CPU virtuale)
- Meccanismi: swap in e swap out (cambio di contesto nel caso della CPU virtuale)
- Politiche: da definire (scheduling nel caso della CPU virtuale)

La differenza principale tra la CPU virtuale e la memoria virtuale è che la CPU è atomica (o la assegno tutta o niente), mentre la memoria è partizionabile, questo rende tra l'altro necessari meccanismi di protezione e di condivisione della memoria tra processi

Il concetto di memoria virtuale rende inoltre possibile l'esecuzione di processi la cui **immagine virtuale** (l'effettiva memoria necessaria al processo per eseguire, derivante dal processo di caricamento) è più grande della memoria fisica del sistema

Tecniche base di gestione della memoria

Rilocazione:

- Statica

L'immagine del processo viene caricata in memoria fisica modificando tutti gli indirizzi a livello di bytecode, il programma genera dunque indirizzi fisici e non c'è overhead di traduzione a runtime

Il problema principale di questo approccio è che, una volta caricato, il programma deve rimanere sempre agli stessi indirizzi fisici, rendendo praticamente impossibili le operazioni di swap, infatti, anche se venissero ricalcolati gli indirizzi nel bytecode, eventuali indirizzi di ritorno nello stack rimarrebbero inalterati!

In questo caso l'allocazione può essere soltanto contigua (la funzione di rilocalizzazione non è disponibile a runtime) e il caricamento soltanto unico (non si può fare swap)

- Dinamica

L'immagine del processo viene caricata in memoria fisica senza modificare gli indirizzi a livello di bytecode, questi rimangono dunque virtuali e permettono lo swap dei processi, il prezzo da pagare è quello di dover effettuare la traduzione degli indirizzi a runtime

Dato che la traduzione deve avvenire praticamente sempre (basti pensare al fetch delle istruzioni), per essere eseguita in modo efficiente serve un componente hardware aggiuntivo, detto **Memory Management Unit**, spesso dotato di una cache, detta **Translation Lookaside Buffer**

Nel caso più semplice la funzione di traduzione è semplicemente una somma (registro base) e può essere anche arricchita con un controllo sulla validità dell'indirizzo tradotto (registro limite)

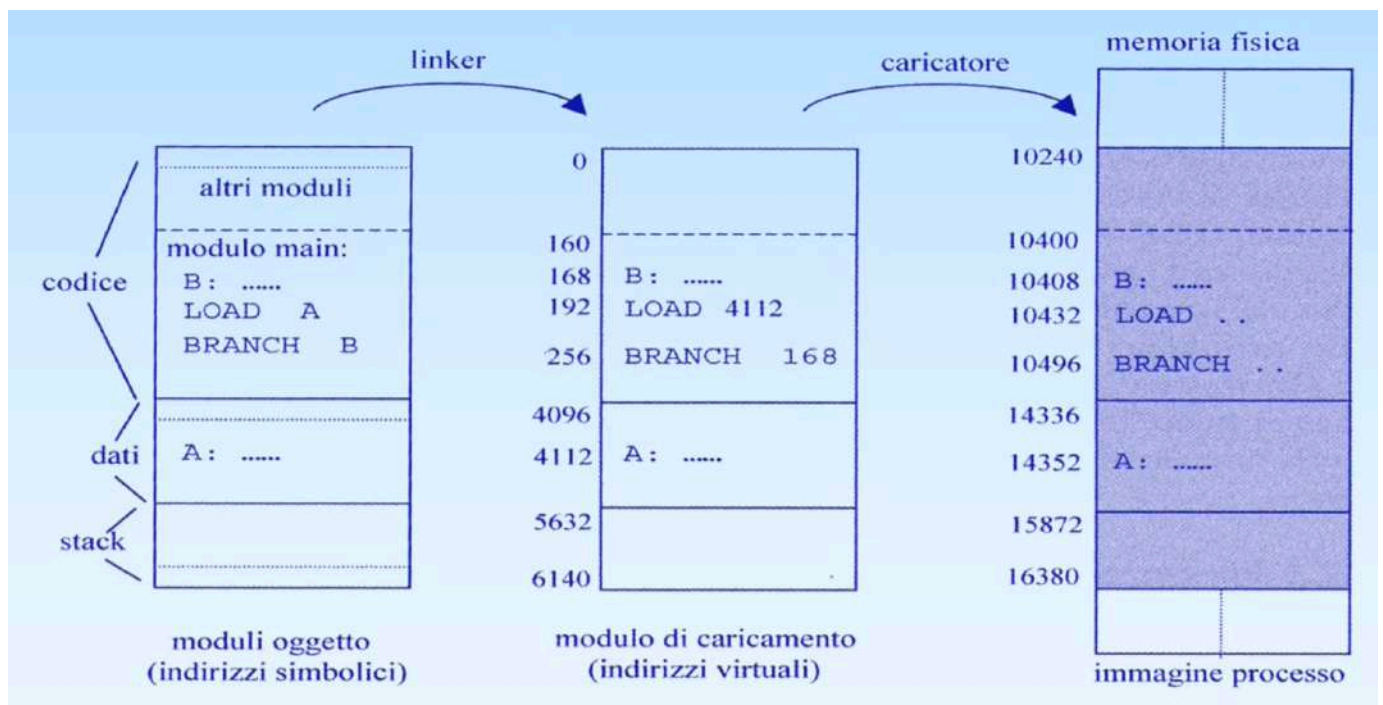
Gestione dello spazio virtuale:

- Unico

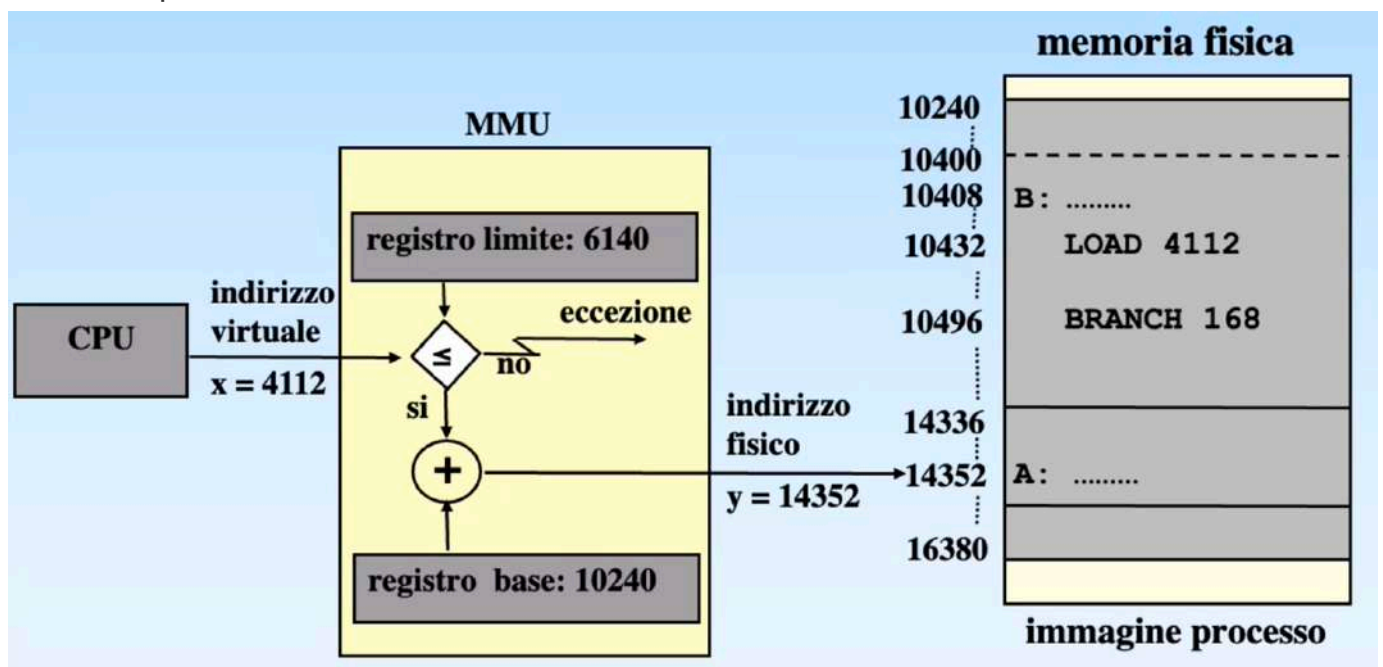
L'immagine del processo viene considerata nella sua interezza, questo rende le cose più semplici ma non consente di condividere zone di memoria tra processi

Questo approccio diventa inoltre problematico se l'allocazione è contigua (ed il caricamento è unico): se non si trova un blocco contiguo di memoria fisica abbastanza grande da contenere l'intera immagine virtuale del processo allora non si può procedere!

- Esempio con *rilocalizzazione statica*



- Esempio con *rilocalizzazione dinamica*



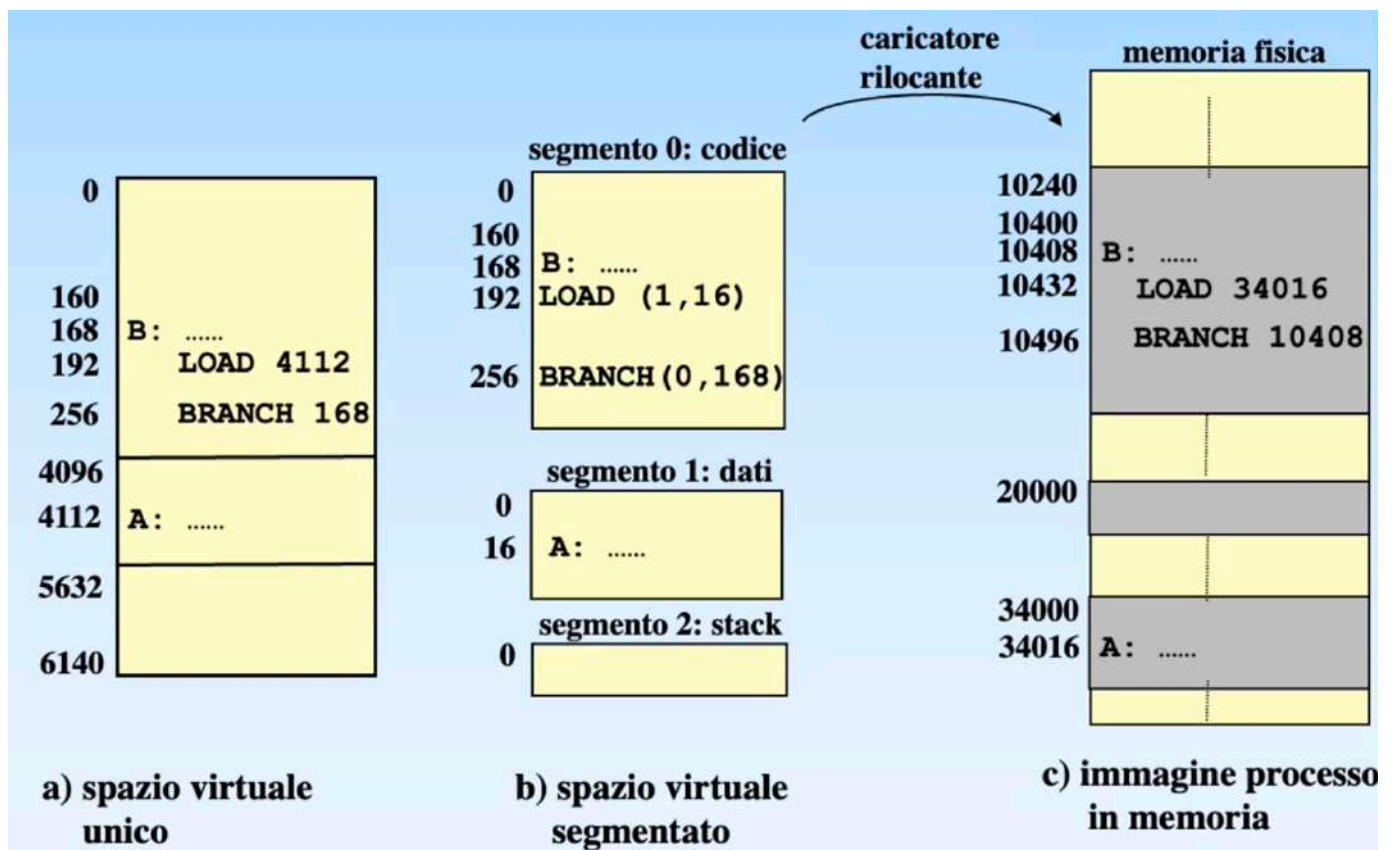
- Segmentato

L'immagine del processo viene divisa in segmenti (tipicamente i tre in figura), questo aggiunge complessità ma permette di condividere zone di memoria tra più processi

In questo caso la MMU contiene una tabella di segmenti (TLB), dove per ciascuno è indicato almeno l'indirizzo base e l'indirizzo limite

Questo approccio può essere comunque problematico in caso di allocazione contigua e caricamento unico se i segmenti sono particolarmente grandi

- Esempio con *rilocalizzazione statica*

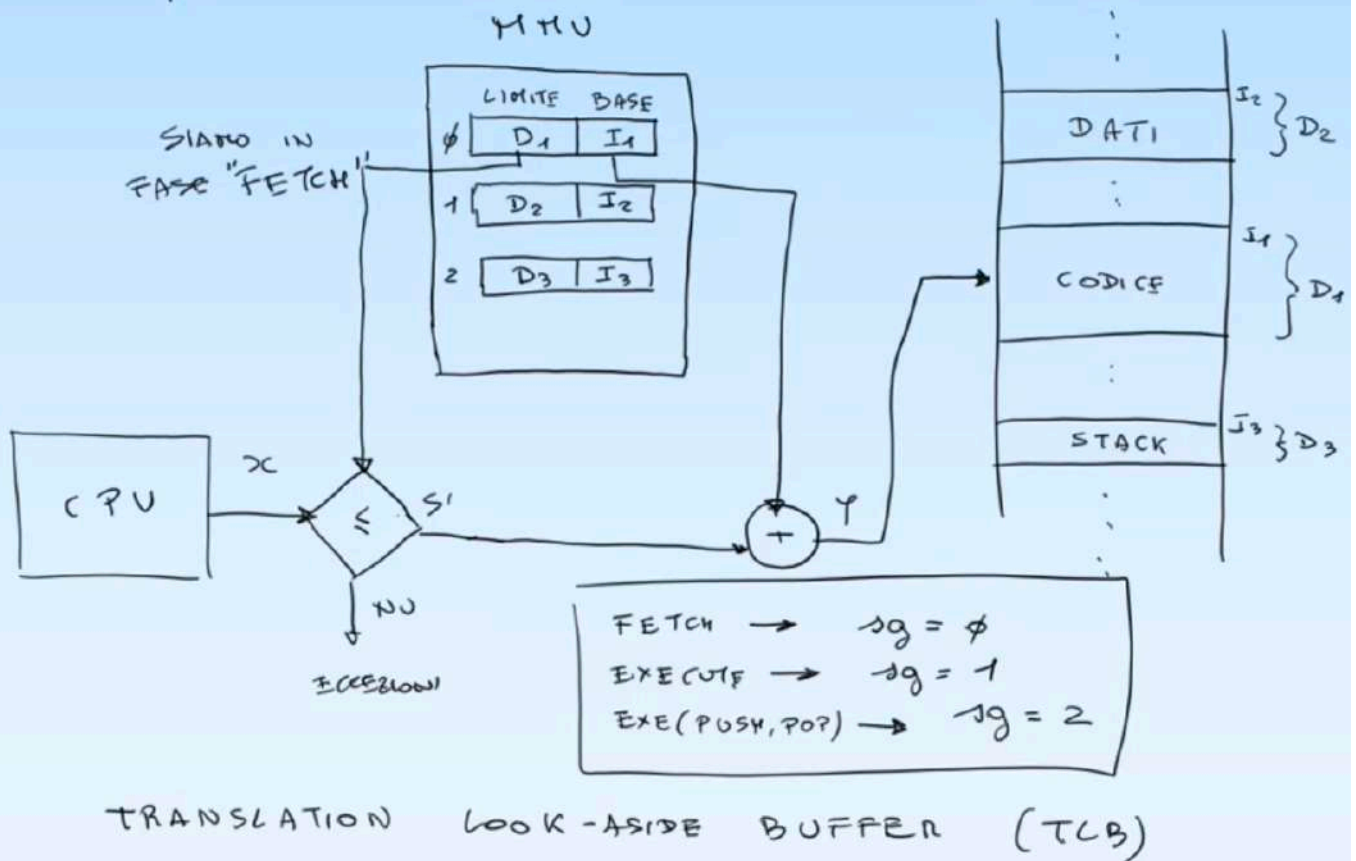


In caso di rilocazione statica gli indirizzi virtuali devono essere strutturati in (segmento, offset) e i segmenti devono essere caricati tutti insieme, altrimenti non si conoscono gli indirizzi

- Esempio con *rilocazione dinamica*

$$x = \langle sg, of \rangle$$

↑



Almeno per la segmentazione a 3 illustrata in figura, non è necessario avere indirizzi virtuali strutturati, in quanto il segmento è implicito dal tipo di istruzione

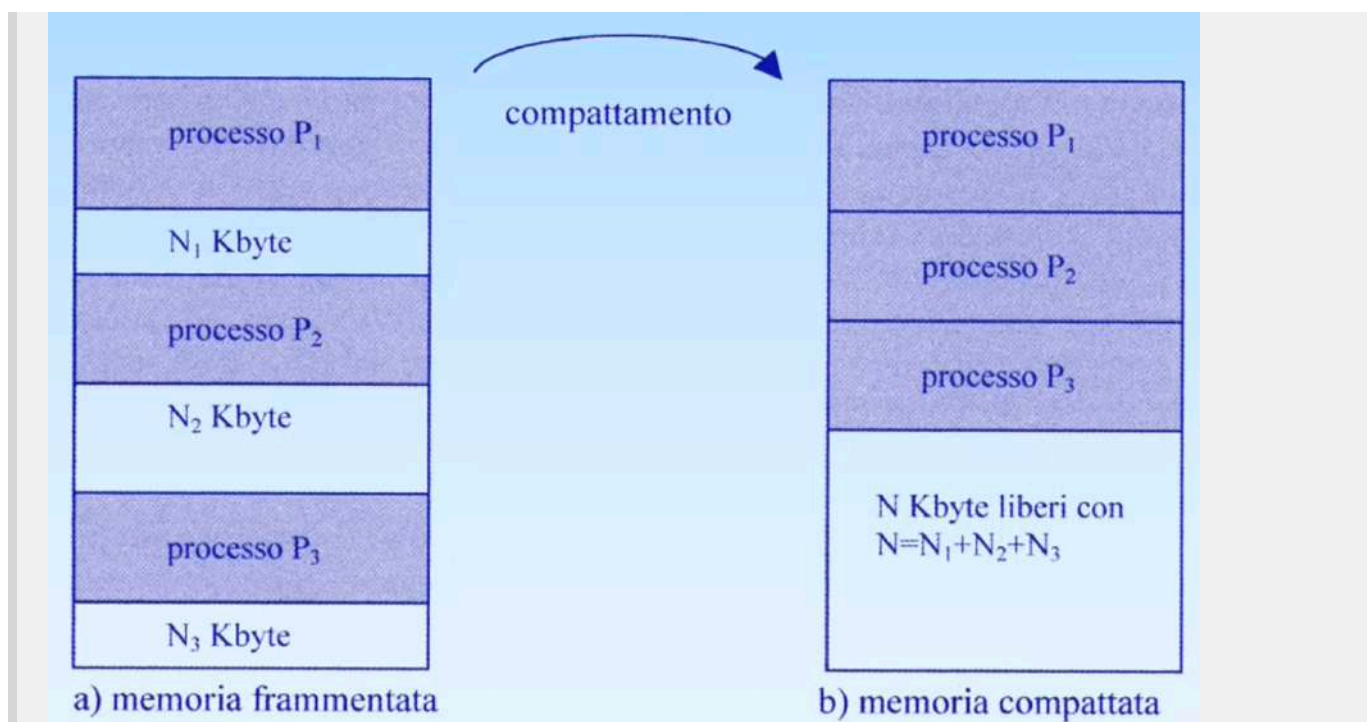
Allocazione:

- Contigua

L'immagine del processo, segmentata o meno, viene caricata in zone contigue di memoria fisica

Il problema principale di questo approccio è che non sfrutta pienamente la memoria: se non c'è un blocco contiguo di memoria fisica abbastanza grande da contenere l'intera immagine virtuale del processo, o l'intero segmento, allora non si può procedere

In caso di rilocalizzazione dinamica si può rimediare al problema della **frammentazione esterna** con la compattazione periodica della memoria



- Non contigua

Valido solo per rilocalizzazione dinamica!

L'immagine del processo viene divisa in pagine, che vengono caricate in corrispondenti frame di memoria fisica

Il problema principale di questo approccio è che la funzione di rilocalizzazione diventa complessa, rendendo necessario l'utilizzo di strutture dati ad hoc

Caricamento:

- Unico

Lo spazio virtuale, unico o segmentato ed eventualmente paginato, viene caricato una sola volta ed interamente in memoria principale

Il problema principale di questo approccio è che non è possibile eseguire processi con spazio di indirizzamento virtuale maggiore della memoria fisica disponibile

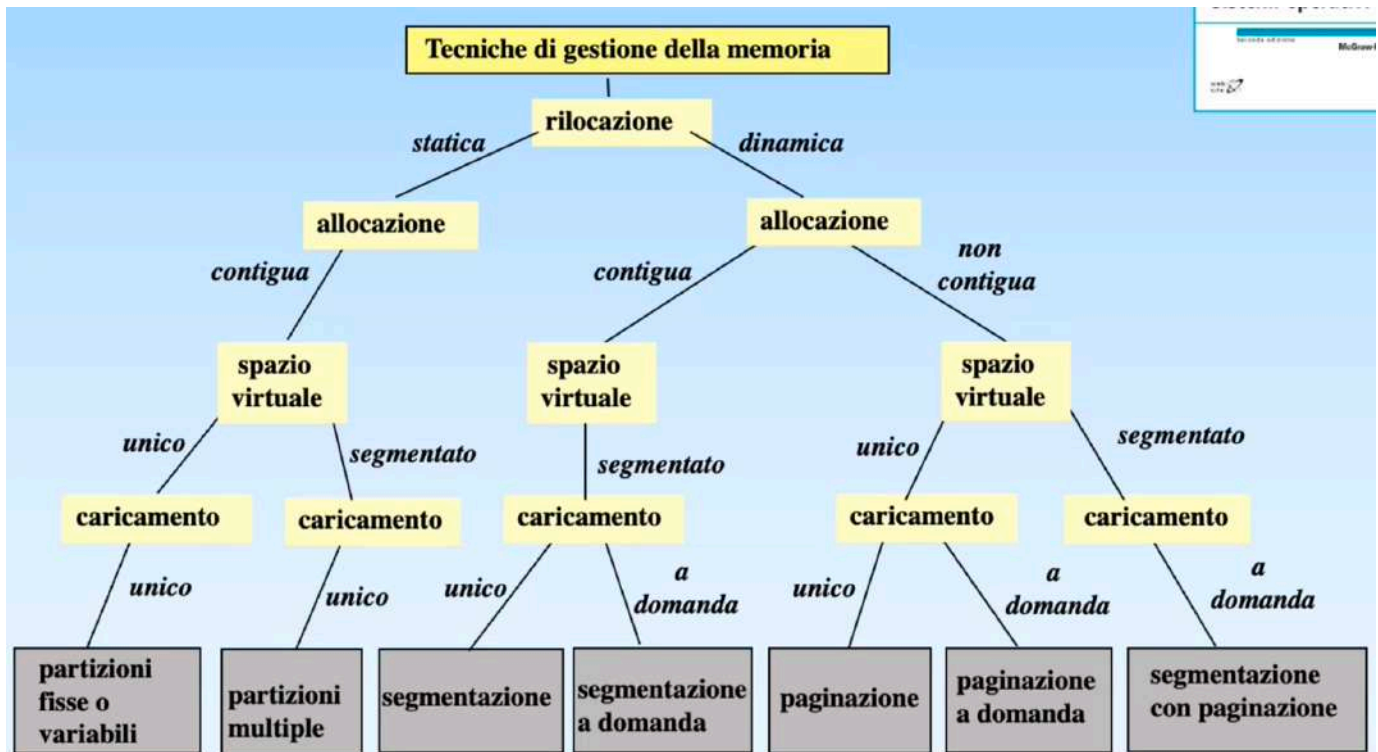
- Su domanda

Valido solo per rilocalizzazione dinamica!

Dello spazio virtuale, unico e paginato o segmentato ed eventualmente paginato, ne vengono caricate in memoria principale solamente le pagine o i segmenti richiesti (*tramite eccezioni su segmenti/pagine non esistenti*)

Questo consente tra l'altro di eseguire processi con immagine più grande della memoria fisicamente disponibile!

Riassumendo:



Le tecniche di gestione della memoria discendenti dalla rilocalizzazione dinamica con allocazione contigua possono avere spazio virtuale unico, ma è estremamente svantaggioso e dunque non viene riportato nel grafo

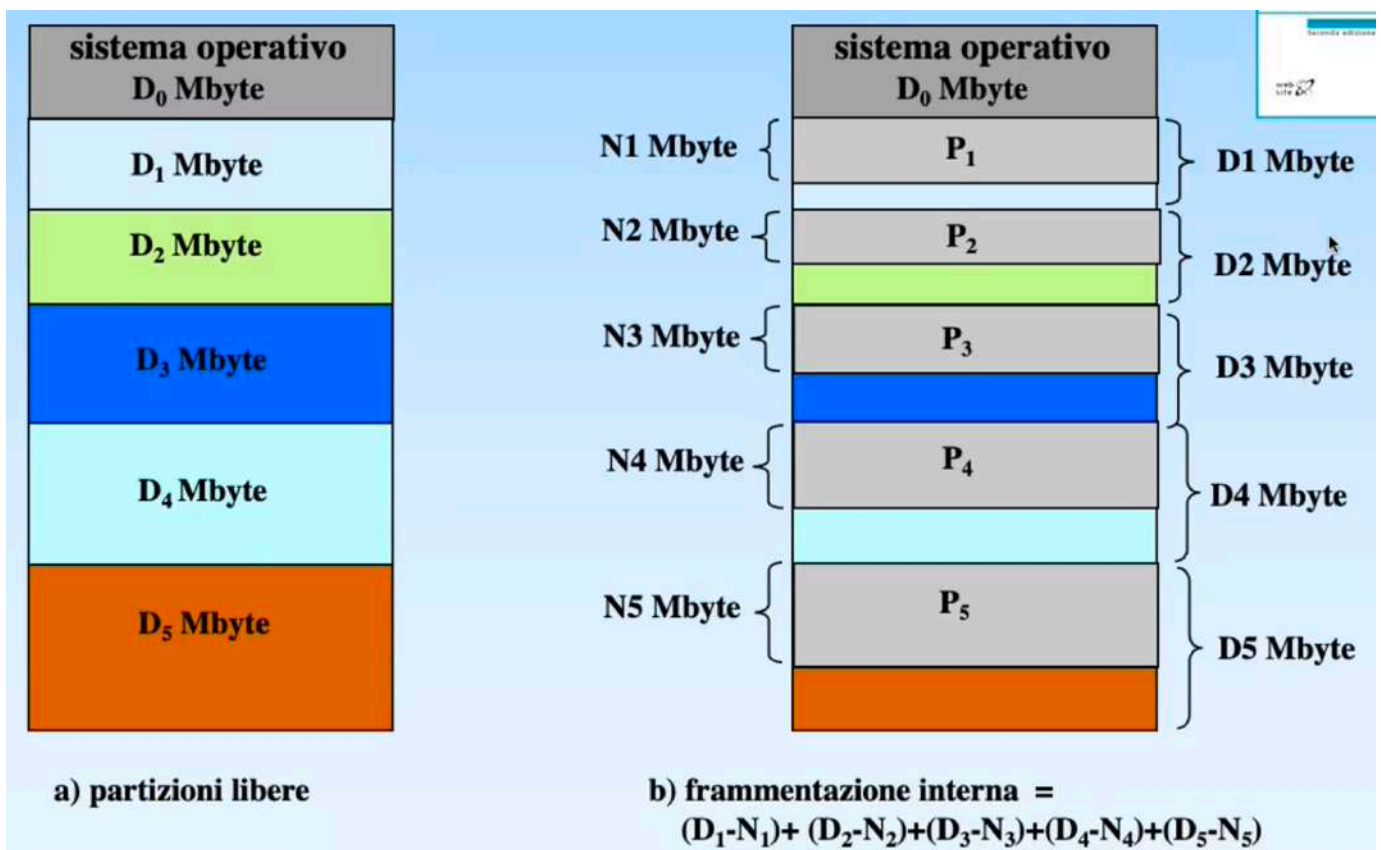
Politiche di gestione della memoria

Adesso che conosciamo tutte le tecniche base di gestione della memoria vediamo come queste (le foglie dell'albero sopra) possono essere implementate in un sistema operativo:

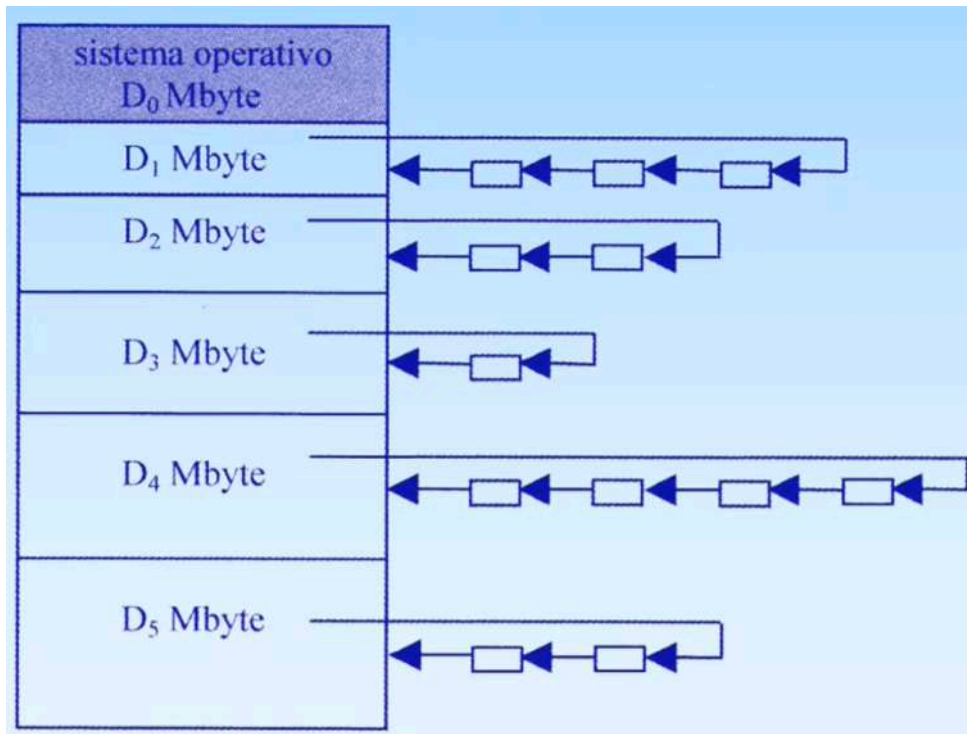
Rilocalizzazione *statica* -> Allocazione *contigua* -> Spazio virtuale *unico* -> Caricamento *unico*

- **Partizioni fisse**

È la soluzione più semplice, la memoria viene preventivamente divisa in partizioni fisse, all'interno delle quali vengono caricati, uno per partizione, i vari processi (e il sistema stesso)



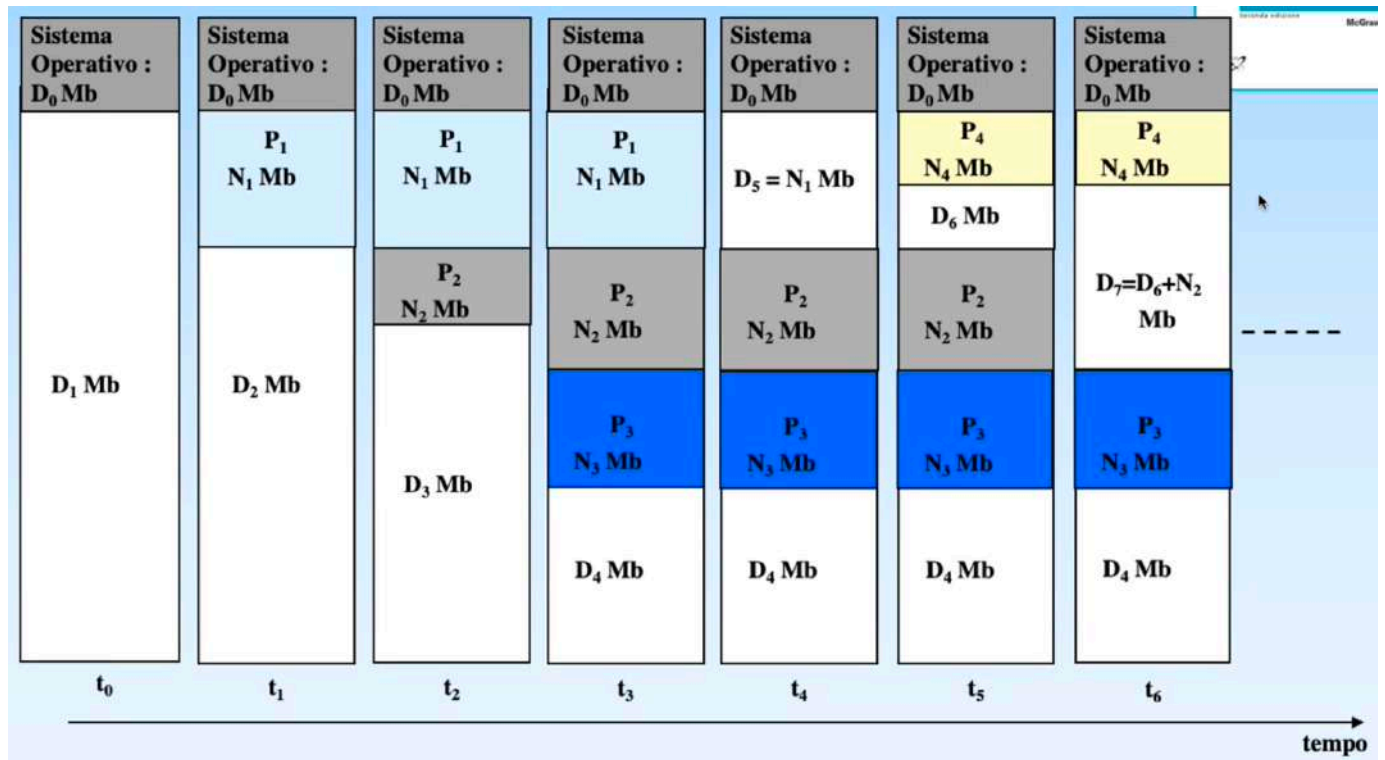
Lo svantaggio principale di questo approccio è chiaramente quello della **frammentazione interna** delle partizioni, c'è tuttavia un grande vantaggio: è possibile effettuare swap (a livello di partizione) anche nel caso di rilocalizzazione statica!



- **Partizioni variabili**

Risolve il problema della frammentazione interna introducendo **frammentazione esterna**, che però, a differenza di quella interna, può essere mitigata, non c'è tuttavia nessuna

possibilità di swap



Rilocalizzazione *statica* -> Allocazione *contigua* -> Spazio virtuale *segmentato* -> Caricamento *unico*

- **Partizioni multiple**

È il nome che prendono le partizioni fisse o variabili nel caso di spazio virtuale segmentato, tendenzialmente in questo caso si preferiscono quelle variabili in quanto la segmentazione aiuta a mitigare la frammentazione esterna

Un dettaglio importante di questi approcci è il modo in cui viene mantenuta ordinata la lista delle partizioni libere (fisse o variabili)

- Best fit: *ordine crescente di dimensione*

Ogni processo (o segmento) viene caricato nella partizione più piccola che lo contiene interamente, se le partizioni sono variabili allora non è una soluzione furba perchè la nuova partizione libera sarà spesso molto piccola (per lo meno la alloco tra le prime posizioni della lista)

Il problema principale di questo approccio è la fase di rilascio, nel caso di partizioni variabili infatti è cruciale fondere due partizioni adiacenti quando una di queste viene liberata, tuttavia per fare questo dovrei ogni volta scorrere praticamente l'intera lista

Di conseguenza questo approccio è preferibile per sistemi a partizioni fisse

- First fit: *ordine crescente di indirizzo base*

Ogni processo (o segmento) viene caricato nella prima partizione abbastanza grande che lo contiene, di conseguenza se le partizioni sono variabili la soluzione è buona perchè la nuova partizione che si crea non per forza ha dimensione piccola ed inoltre viene posizionata subito dopo in lista

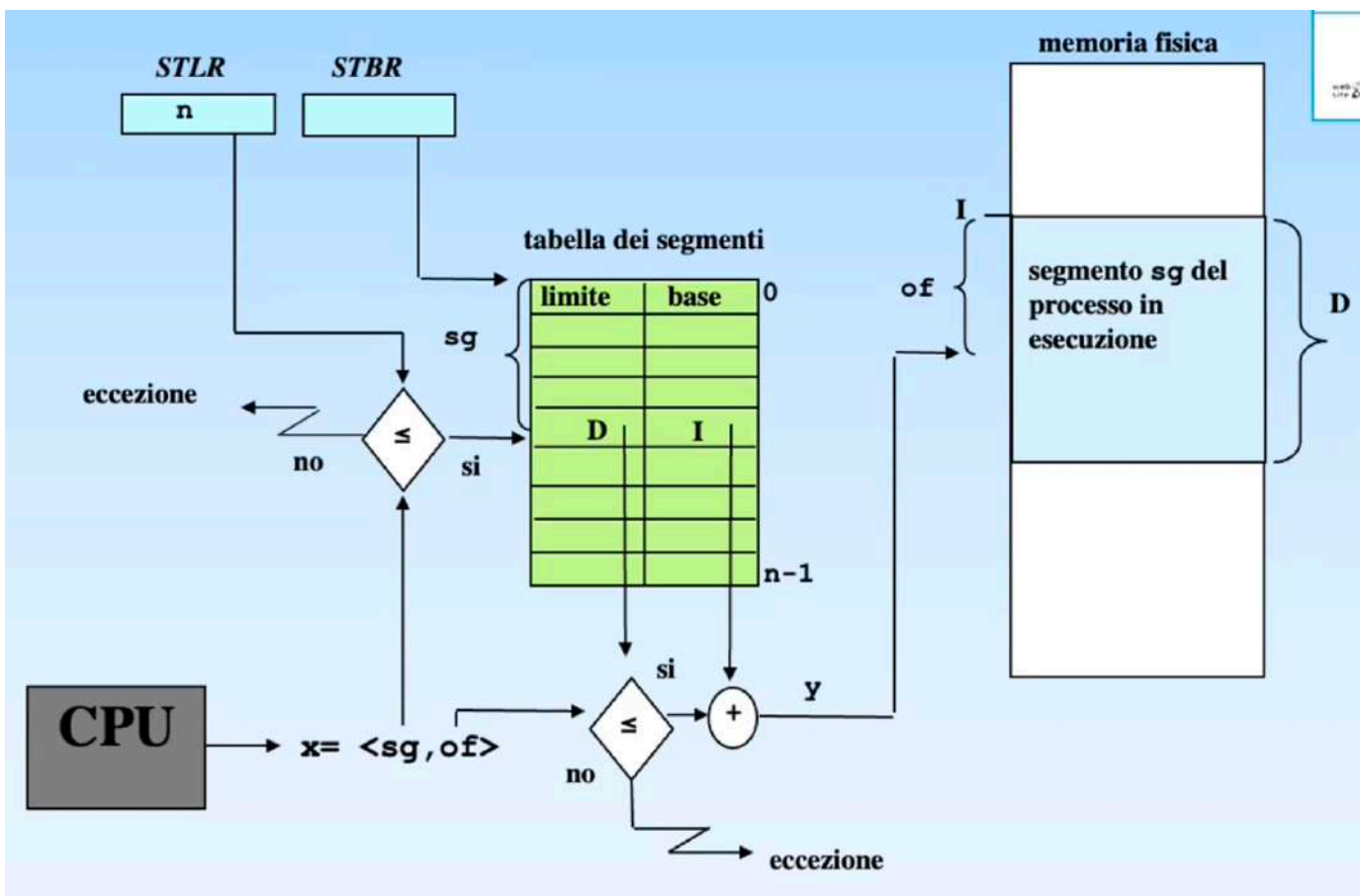
Sempre nel caso di partizioni variabili anche la fase di rilascio è avviene in modo efficiente, per controllare se due partizioni sono adiacenti mi basta controllare l'elemento precedente e successivo in lista

Di conseguenza questo approccio è preferibile per sistemi a partizioni variabili

Rilocazione *dinamica* -> Allocazione *contigua* -> Spazio virtuale *segmentato* ->
Caricamento *unico*

- **Segmentazione**

Gli indirizzi virtuali sono strutturati (poco importa, si occupano di tutto compilatore e linker) e si riferiscono ad una tabella dei segmenti, puntata da *Segment Table Base Register* e grande *Segment Table Length Register*, l'overhead di questo approccio è quello di accedere alla tabella dei segmenti, che, se abbastanza piccola, si trova direttamente nella MMU, altrimenti si trova in RAM e la memoria della MMU (TLB) funge da cache

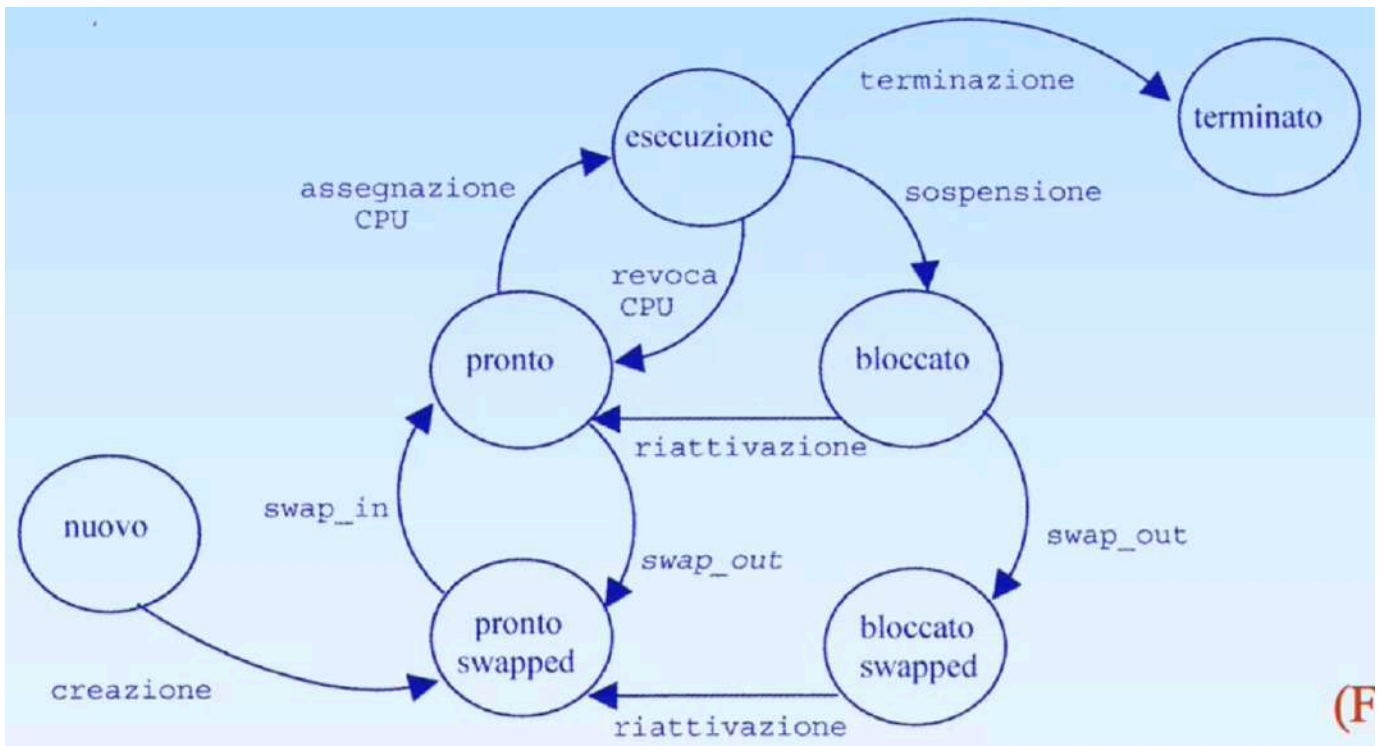


Il descrittore di segmento può anche contenere informazioni di controllo, ad esempio con (Base, Limite, R, W) si può limitare la lettura e la scrittura di certi segmenti

STBR è l'indirizzo fisico della tabella dei segmenti, che deve stare in memoria protetta (heap del sistema operativo)

In presenza di rilocalizzazione dinamica si introduce inoltre il concetto di swap, in caso di caricamento unico bisogna però rivedere gli stati dei processi:

Un processo nuovo finisce per prima cosa in pronto swapped, tutto il suo spazio virtuale rimane nel disco (non per forza nella zona di swap) e di conseguenza anche gli indirizzi base e limite della tabella dei segmenti si riferiscono al disco. Sarà poi lo scheduler di medio termine a stabilire quando caricare il processo in memoria principale e quando eventualmente swapparlo, stavolta nella zona di swap (almeno per i segmenti modificati), per fare spazio ad altri processi



Per limitare l'overhead si potrebbe rinunciare allo swap out dei processi pronti

Rilocazione *dinamica* -> Allocazione *contigua* -> Spazio virtuale *segmentato* -> Caricamento *su domanda*

• Segmentazione su domanda

I segmenti vengono caricati in memoria (previa lancio di segmentation fault) solamente quando la cpu genera un indirizzo virtuale che ricade al loro interno, se non c'è abbastanza memoria libera allora ne viene liberata facendo swap out di altri segmenti

Il descrittore di segmento diventa più complesso (*Base, Limite* | *R, W* | *U, M, P*):

- **Base**: se il segmento non è caricato in memoria indica l'indirizzo di memoria secondaria (swap) dove è memorizzato il segmento
- **Limite**: se l'indirizzo virtuale eccede limite allora si abortisce il processo oppure l'eccezione viene interpretata come la volontà del processo di espandere il segmento
- **Presenza**: indica se il segmento è già caricato in memoria, se non lo è lo si carica passando base e limite al DMA, liberando preventivamente, e se necessario, memoria tramite swap out di altri segmenti
- **Utilizzo**: indica se la cpu ha acceduto (in lettura o scrittura) al segmento
- **Modifica**: indica se la cpu ha modificato il segmento, questo serve per evitare di ricopiare inutilmente segmenti in fase di swap out

All'inizio, quando il processo è nuovo, si può fare pre-paging, ossia caricare in memoria un sottoinsieme di segmenti prescelti, oppure non caricare alcun segmento

In generale lo schema della segmentazione è ancora valido modificando però il descrittore di processo ed aggiungendo i vari test sui bit

Rilocazione *dinamica* -> Allocazione *non contigua* -> Spazio virtuale *unico* -> Caricamento *unico*

- **Paginazione**

Sia lo spazio virtuale che quello fisico sono suddivisi in parti uguali dette rispettivamente **pagine** e **frame** (anche detti pagine fisiche), in corrispondenza biunivoca

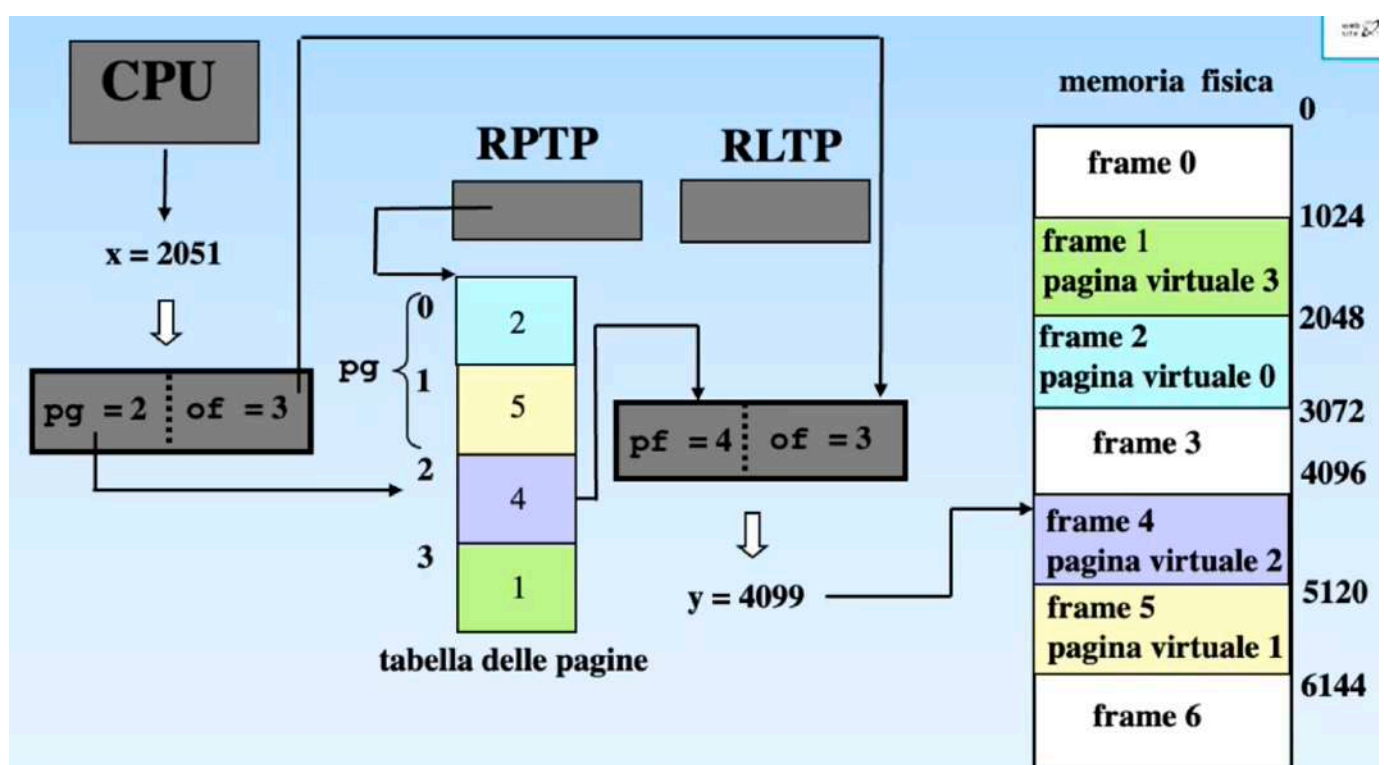
Gli indirizzi virtuali sono "strutturati" nel senso che i bit più significativi vengono presi come indice di pagina e quelli meno significativi come offset

È necessaria una tabella delle pagine, lunga *Registro Lunghezza Tabella Pagine* e puntata da *Registro Puntatore Tabella Pagine* (in questo caso il TLB è fondamentale) con i bit di controllo R e W, ed una tabella dei frame (o tabella delle pagine fisiche)

Come minimo il descrittore di pagina dovrà contenere l'indice del frame corrispondente (risparmio bit rispetto al memorizzare l'indirizzo), mentre il descrittore di frame dovrà contenere se è libero o meno

Con questo approccio si elimina la frammentazione esterna reintroducendo però lieve frammentazione interna (nell'ultima pagina di ogni processo)

Essendo il caricamento unico sono di nuovo necessari gli stati swapped!



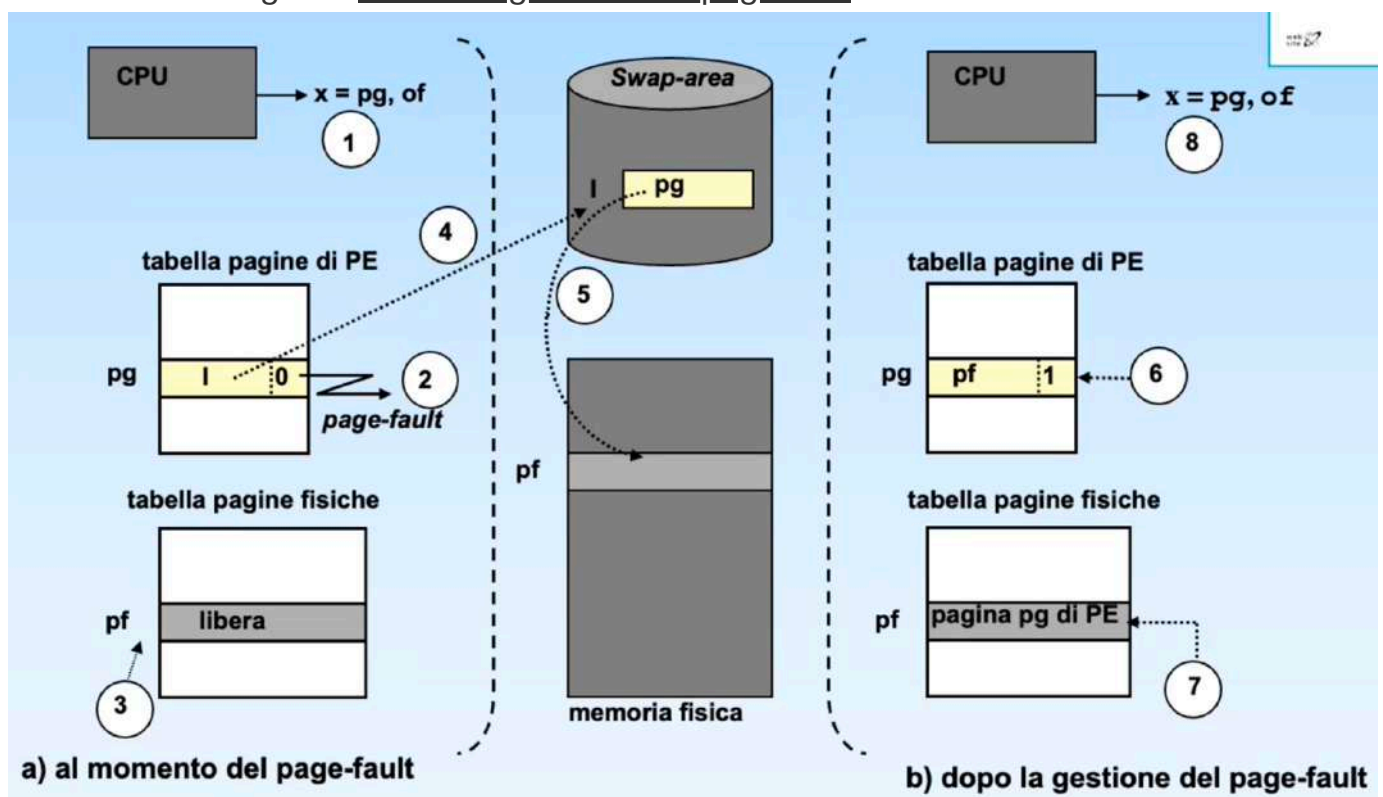
Rilocazione *dinamica* -> Allocazione *non contigua* -> Spazio virtuale *unico* -> Caricamento *su domanda*

- **Paginazione su domanda**

Il descrittore di pagina diventa più complesso (*Frame | R, W | U, M, P*), in particolare, come avveniva per la segmentazione su domanda, se la pagina non è caricata in memoria allora **Frame** indica l'indirizzo di memoria secondaria dove è memorizzata

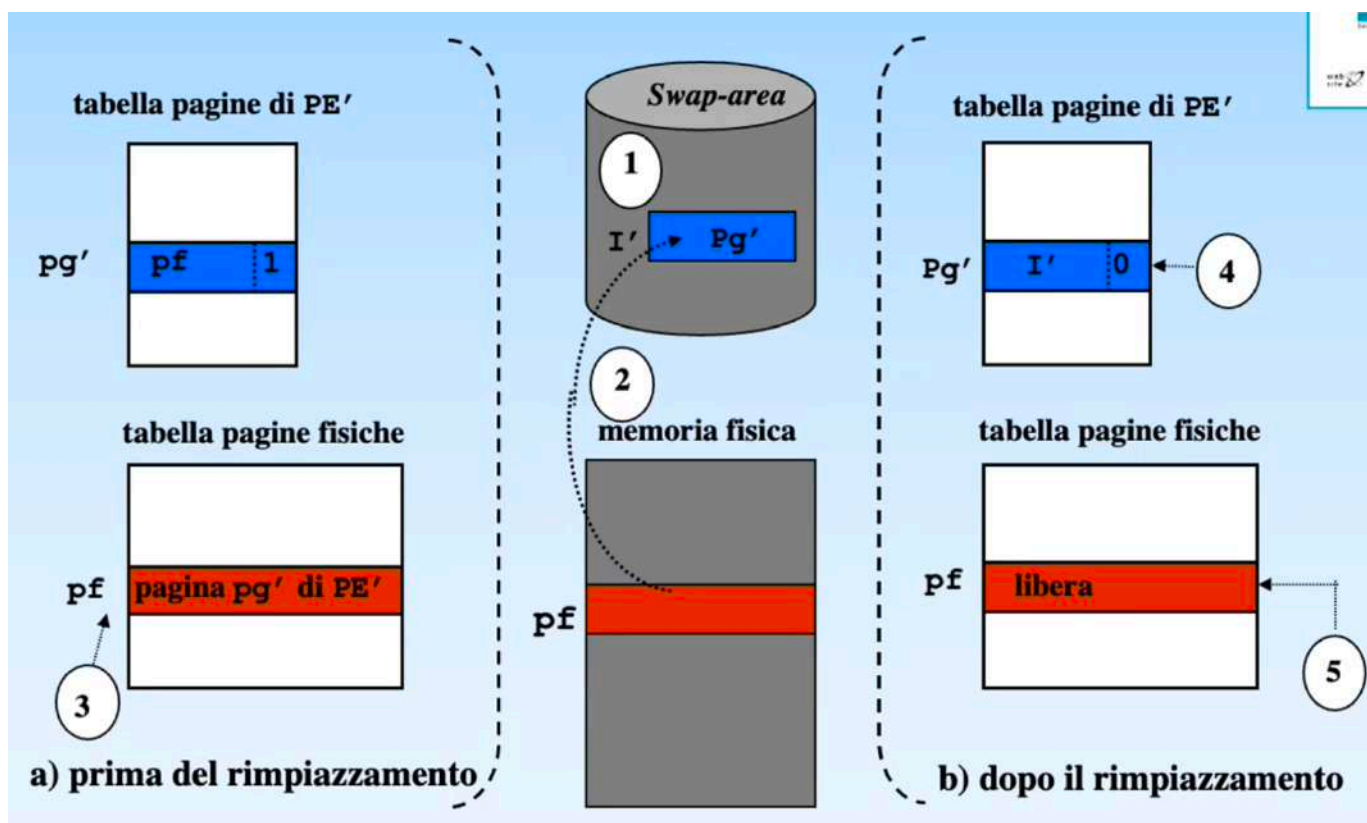
Anche il descrittore di frame si complica: per farne lo swap out diventa infatti necessario risalire direttamente a quale pagina di quale processo traduce

Vediamo in dettaglio la routine di gestione del page fault:



- Per minimizzare i cambi di contesto potremmo far bloccare il processo generante page fault all'istante 4 anziché all'istante 2, facendogli dunque svolgere con privilegi di sistema le operazioni 3 e 4, in ogni caso prima di cambiare contesto bisogna ricordarsi di cambiare il program counter, riportandolo all'istruzione che ha generato il page fault
- Al punto 5 arriva l'interruzione del DMA che sancisce la fine del trasferimento della pagina, come prima, per minimizzare i cambi di contesto, possiamo pensare di far eseguire le operazioni 6 e 7 al processo (casuale) in esecuzione durante il trasferimento del DMA

Se al punto 3 non ci sono frame liberi bisogna chiamare la routine di rimpiazzamento:



- Come prima cosa si cerca uno spazio libero nell'area di swap per memorizzare il frame in uscita
- In secondo luogo si sceglie, secondo politiche che vedremo dopo, un frame da rimpiazzare, e si delega al DMA il suo trasferimento nello swap (se necessario, ossia se il bit di modifica è 1)
- Infine, dalla tabella dei frame si legge quale pagina traduceva il frame rimpiazzato e si aggiorna la tabella delle pagine del processo coinvolto

Tutto ciò ha un costo, per evitare che questo succeda unix ad esempio fa in modo di mantenere sempre un po' di pagine libere

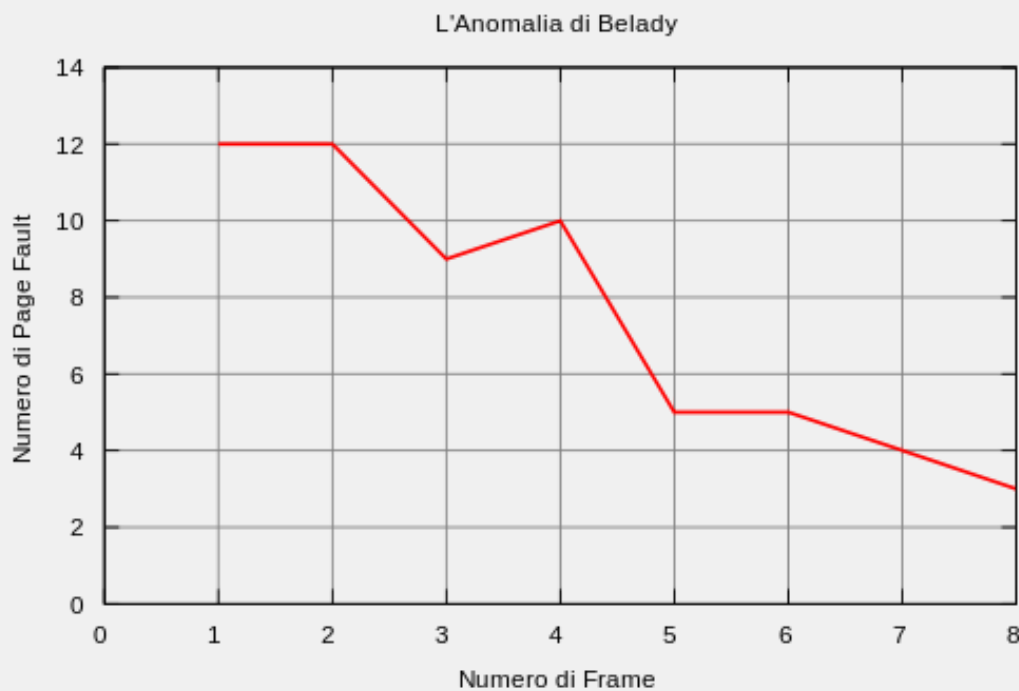
Come si sceglie il frame da rimpiazzare ?

Si definisce **working set** di un processo le pagine con le quali sta attualmente lavorando, questo, se il codice è scritto bene, inizialmente si espande, poi satura, ed infine cambia lentamente

L'algoritmo ottimo è quello che rimpiazza solamente le pagine che non verranno più usate, questo è chiaramente impossibile da implementare, però ci fornisce una metrica per confrontare altri algoritmi, infatti, il numero di page fault generati da un processo nel corso della sua vita (*page reference list*), in presenza dell'algoritmo ottimo, è uguale alla dimensione (in pagine) dello spazio virtuale del processo, o, in modo equivalente, il numero di page fault generati da un processo in un particolare momento della sua esecuzione per ottenere un determinato working set è sempre uguale alla dimensione del working set

Il più semplice algoritmo di rimpiazzamento è quello a politica **FIFO**: il frame da rimpiazzare è quello da più tempo in memoria, questo è facile da implementare se la tabella dei frame viene gestita come un array circolare, tuttavia non sempre è una buona scelta (e neppure strettamente FIFO, immagina quando termina un processo e si liberano pagine in posti a caso)

Gli algoritmi di rimpiazzamento di questa tipologia sono affetti dalla cosiddetta **Anomalia di Belady**: aumentando il numero di frame disponibili in genere ci aspettiamo che il numero di page fault diminuisca, e questo è vero, con una piccola eccezione:



Un'altra politica di rimpiazzamento è quella **LRU**: il frame da rimpiazzare è quello meno recentemente utilizzato, implementare questa politica in modo preciso è estremamente costoso, dovrei infatti modificare il descrittore di pagina inserendo un timestamp al posto del bit di uso, aggiornarlo ad ogni accesso, ed inoltre scorrere tutta la tabella delle pagine per cercare quella LRU (ed ottenere di conseguenza il frame)

In ogni caso una volta scelto l'algoritmo ci sono due strategie possibili:

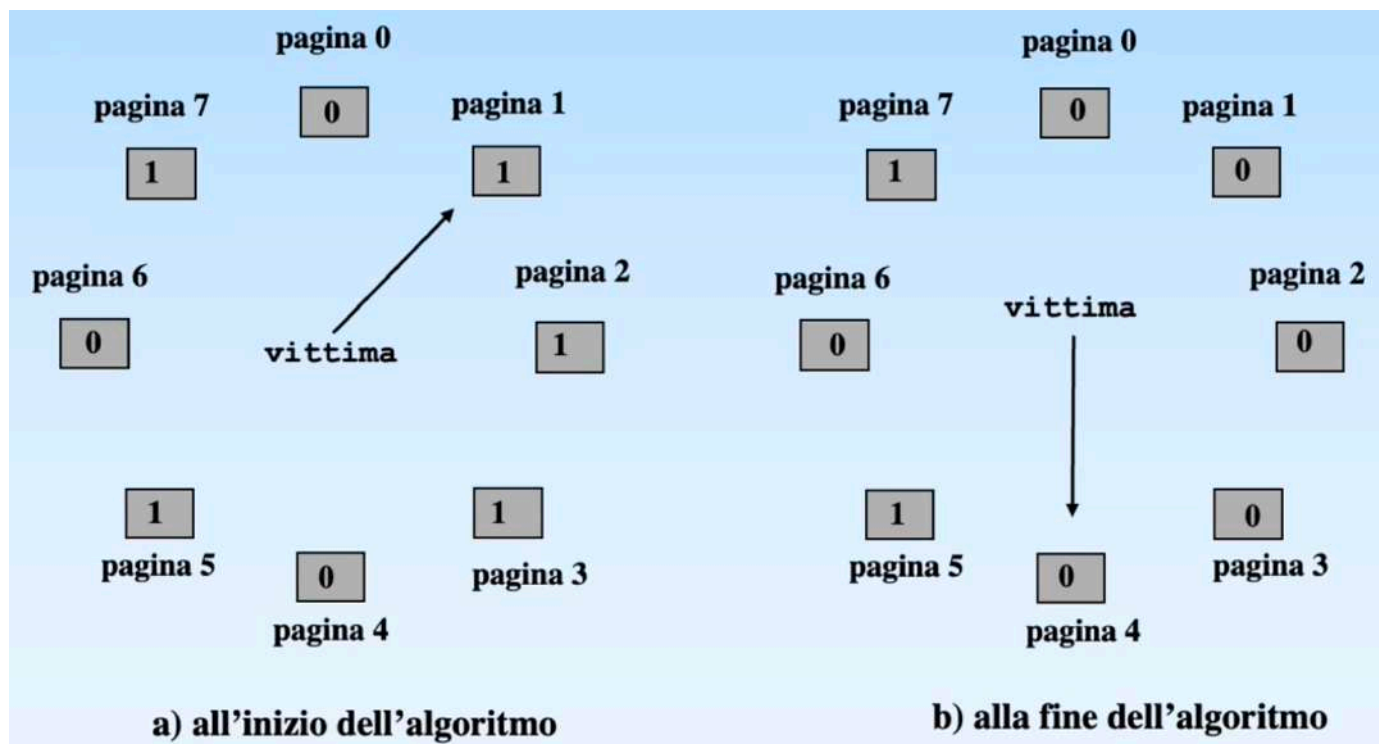
- Rimpiazzamento locale: la pagina da rimpiazzare viene cercata tra le pagine del processo che ha generato il page fault (nel caso di algoritmo FIFO non solo è sbagliato ma implicherebbe anche avere una tabella dei frame per ogni processo)
- Rimpiazzamento globale: la pagina da rimpiazzare viene cercata tra tutte le pagine di tutti i processi (nel caso di algoritmo FIFO se un processo è scritto male e genera tanti page fault gli altri processi pagano)

Una delle politiche di rimpiazzamento utilizzate in sistemi reali è quella di **Second Chance**: supponendo un rimpiazzamento locale, la pagina da rimpiazzare è la prima, a partire

dall'indice vittima (pagine gestite come array circolare), con bit di uso a 0, alle pagine percorse viene resettato il bit di uso ed infine l'indice vittima viene spostato alla pagina successiva a quella rimpiazzata

Un'ottimizzazione di questa politica può essere la seguente: tra le possibili vittime (pagine con bit di uso a 0), viene preferita la prima con bit di modifica a 0 (così non la devo ricopiare in swap), se non ce n'è allora si procede come al solito

Questo algoritmo si può eseguire "in linea", ossia ogni qualvolta viene generato un page fault con memoria fisica piena, oppure, come in unix, eseguire ogni tot tempo o ogni volta che si supera soglia di memoria per fare in modo di avere sempre memoria libera in caso di page fault



Un sistema si dice in situazione di **trashing** se, in seguito ad una politica errata di rimpiazzamento, utilizza la maggior parte della CPU per gestire page fault

Rilocazione *dinamica* -> Allocazione *non contigua* -> Spazio virtuale *segmentato* -> Caricamento *su domanda*

- **Paginazione con segmentazione**

La strategia di gestione della memoria più avanzata è quella che combina la paginazione (zero frammentazione esterna) alla segmentazione (facilità di protezione e condivisione della memoria), il tutto con caricamento su domanda

Utilizzando la segmentazione, gli indirizzi virtuali devono essere strutturati (la paginazione è trasparente al programma, la segmentazione no)

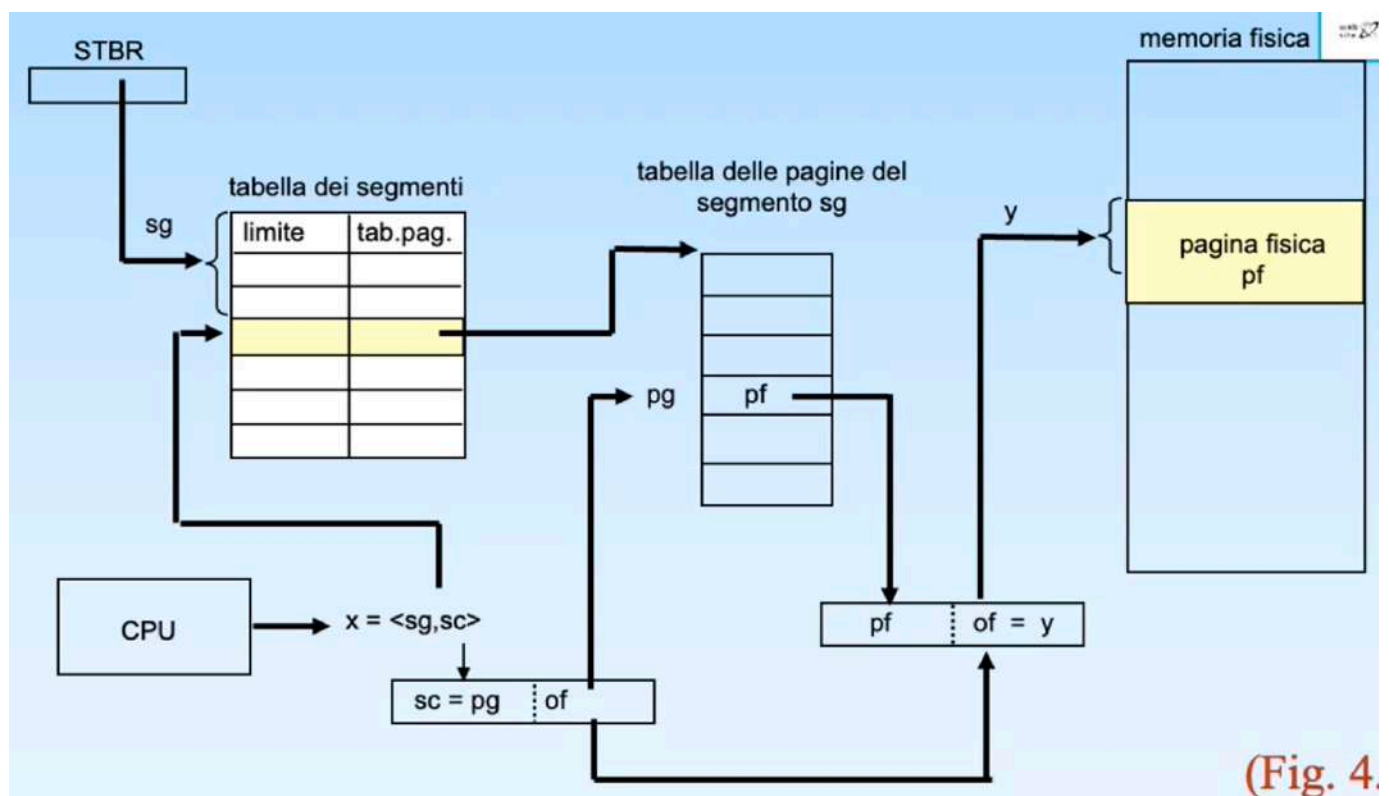
Il descrittore di segmento deve inoltre cambiare: al posto dell'indirizzo di base è necessario mantenere il puntatore (fisico) alla tabella delle pagine che traducono il segmento stesso (la memoria non è più contigua)

Con questa configurazione il segmentation fault avviene prima del page fault, i bit di controllo dunque vengono controllati prima sul segmento e poi, opzionalmente, anche sulle singole pagine

Essendo il caricamento su domanda, se il bit di presenza per un determinato segmento è 0, allora significa che la tabella che traduce quelle determinate pagine va creata da zero, oppure esiste già ma si trova nello swap (in questo caso però deve essere protetta in qualche modo)

Le tabelle dei segmenti e delle pagine quando sono caricate in memoria stanno nello heap di sistema!

Il TLB diventa ancora più importante poichè per ogni accesso in memoria ne servono due ulteriori: uno alla tabella dei segmenti ed uno alla tabella delle pagine del particolare segmento (sapendo che il TLB si flusha dopo ogni cambio contesto comprendiamo adesso l'importanza dei thread)



(Fig. 4.

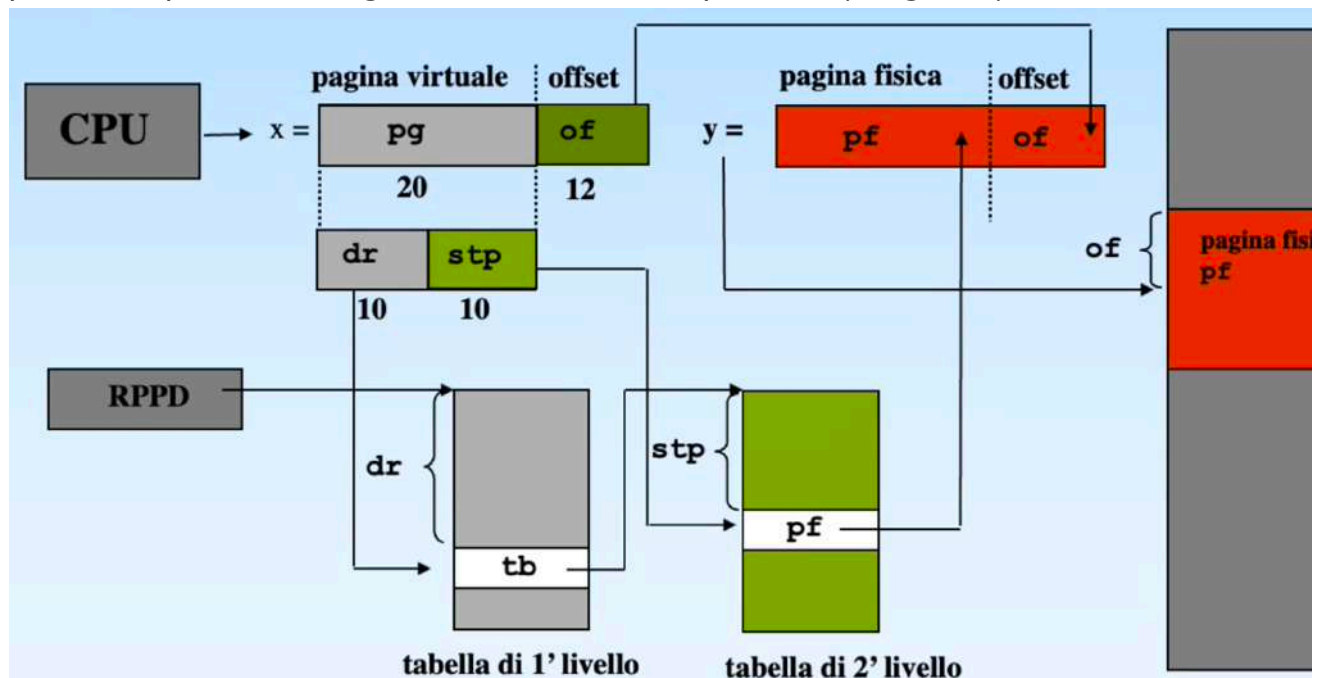
Il controllo $pg < n^{\circ} \text{ pagine}$ è implicito dal controllo $sc < limite$

Vediamo infine due dettagli riguardanti questo approccio (e più in generale quello della paginazione):

- Se gli spazi virtuali del SO e dei processi sono separati (come avveniva in origine), allora, quando un processo chiama una primitiva di sistema bisognerebbe cambiare funzione di traduzione ed invalidare il TLB, e tutto questo costa (per non parlare della difficoltà nel passaggio dei parametri), la soluzione è tuttavia semplice: basta mettere sempre lo spazio virtuale del SO (protetto con un ulteriore bit di controllo magari) dentro a quello di ogni processo

Non serve ripetere la traduzione del SO per ogni processo, possiamo condividere il segmento sistema!

- Il secondo problema riguarda la grandezza delle tabelle di traduzione, per limitarla possiamo pensare di organizzare le tabelle su più livelli (in figura 2)



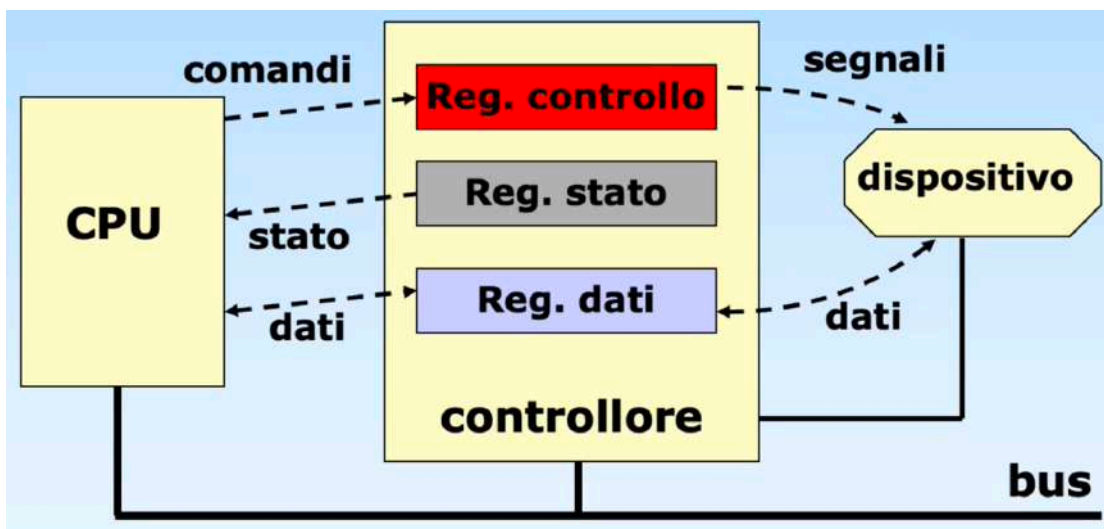
Se poi le sottotabelle le facciamo grandi esattamente quanto una pagina (e in questo caso lo sono, supponendo 4 byte per descrittore di pagina), allora ci aiuta con il caricamento su domanda e con il rimpiazzamento

5. Gestione delle periferiche

Introduzione

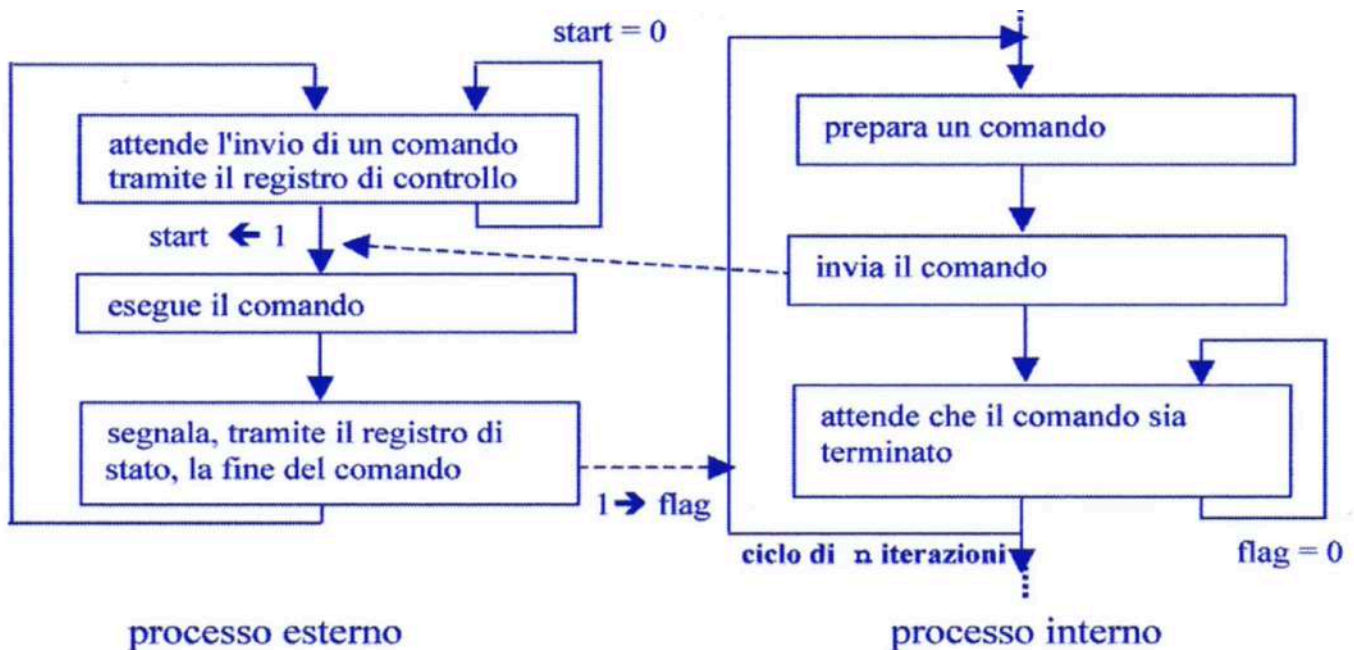
Il compito principale del sottosistema di I/O è quello di **nascondere i dettagli hardware** dei controllori (interfacce) delle varie periferiche

Noi studieremo una versione semplificata in cui lo spazio di indirizzamento di I/O è diviso da quello di memoria e contiene tre registri per ogni controllore:

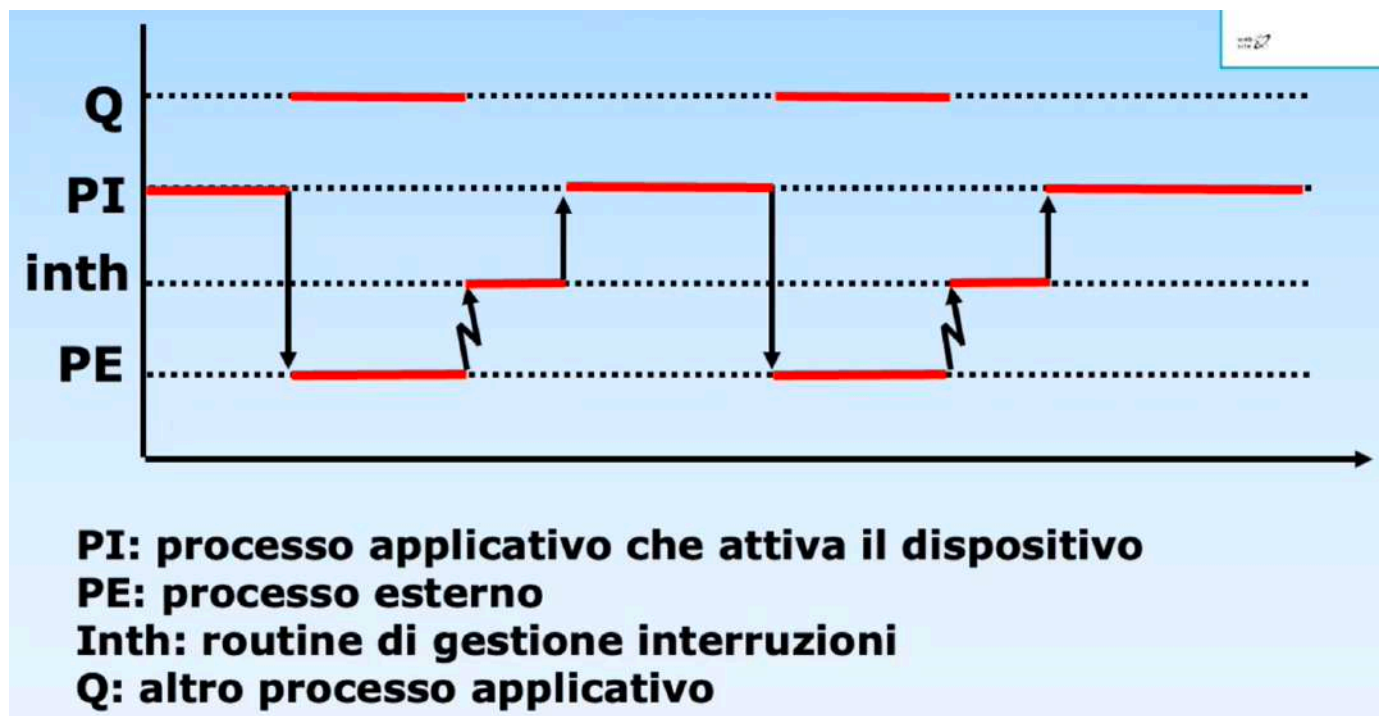


Il registro di controllo ha sicuramente il bit di start e quello di abilitazione alle interruzioni, mentre il registro di stato ha ad esempio il bit che segnala condizioni di errore

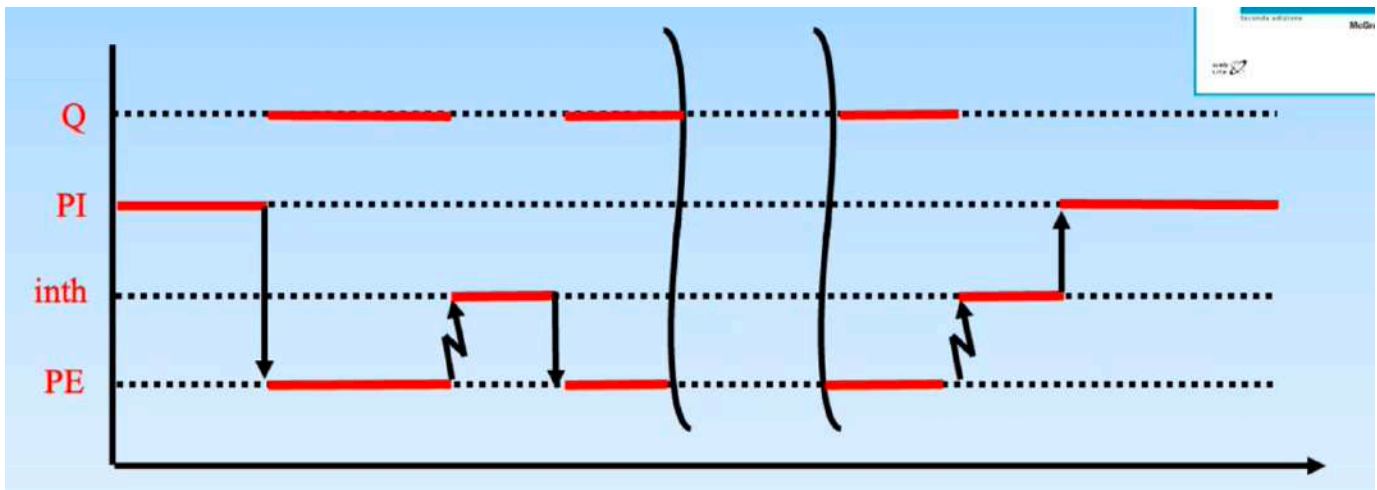
Chiamiamo **processo esterno** le operazioni eseguite dal controllore (CPU dedicata) sulla periferica, l'interazione che ci aspettiamo di ottenere tra il processo chiamante (interno) e quello esterno è del seguente tipo:



In realtà processo interno non può fare attesa attiva, ma si sospenderà su un semaforo in attesa della fine dell'operazione richiesta, un diagramma di temporizzazione che mostra questo comportamento è il seguente:

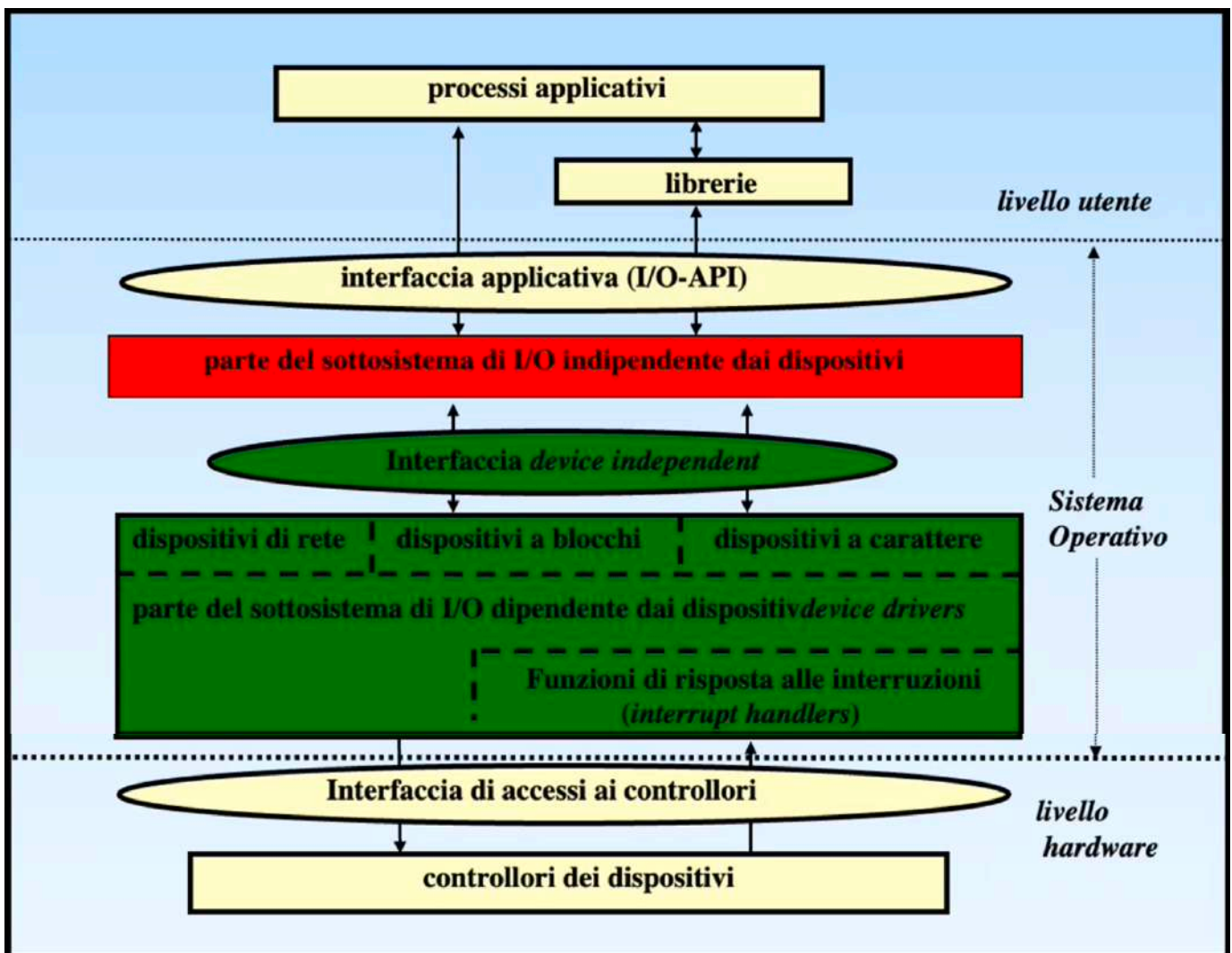


In particolare se l'operazione coinvolge più byte non ha senso risvegliare sempre il processo esterno, possiamo fare in modo che inth si occupi di (quasi) tutto:



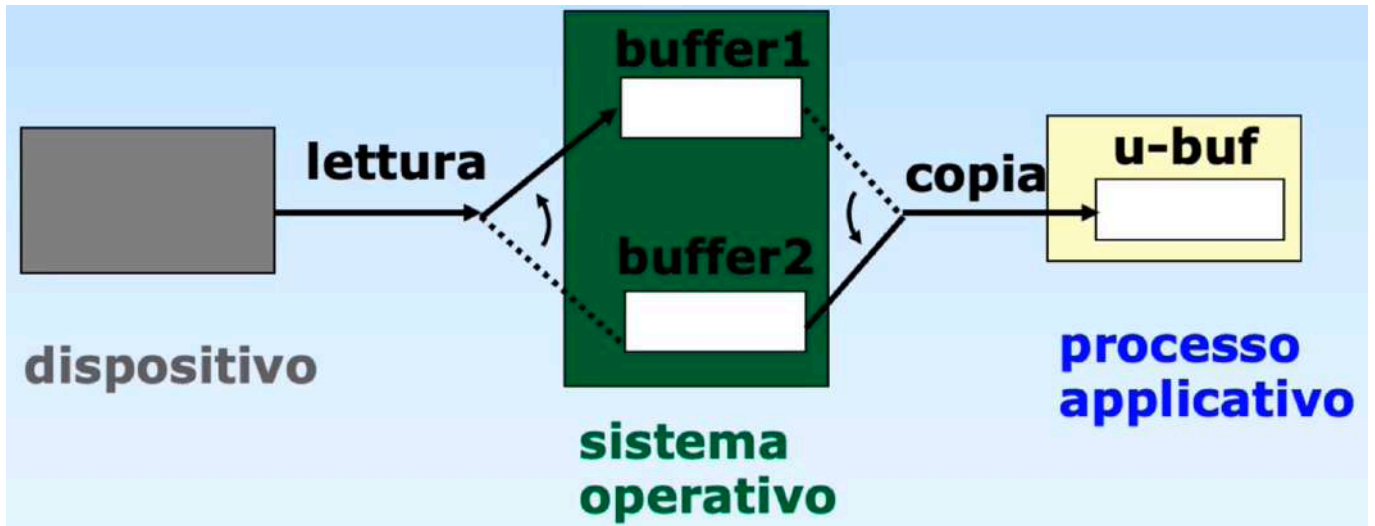
La routine di gestione delle interruzioni *inth* fa eseguire il processo Q interrotto cambiando privilegi

Organizzazione logica del sottosistema di I/O



Approfondiamo la parte del sottosistema di I/O indipendente dai dispositivi (rettangolo rosso):

- Implementa il **buffering**: per problemi di differenze di velocità/funzionalità è bene disaccoppiare nel tempo e nello spazio processi e periferiche, aggiungendo un passaggio intermedio nel sistema operativo, se questo non fosse presente dovremmo inoltre garantire che lo spazio virtuale del processo chiamante non faccia swap out, nonostante questo sia sospeso, ed inoltre non è detto che il buffer fornito dall'utente sia ottimizzato per i trasferimenti dalla determinata periferica



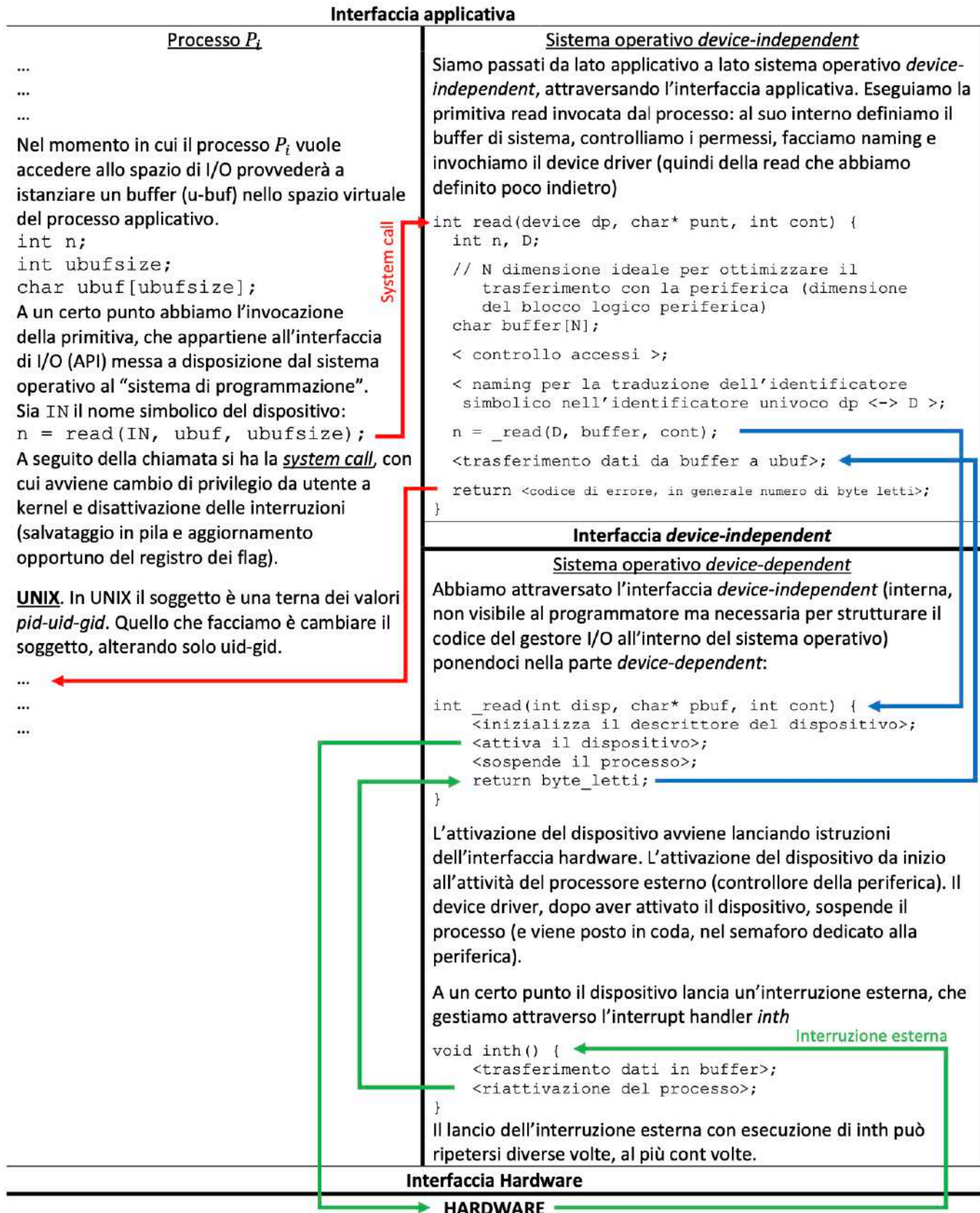
- Definisce lo **spazio dei nomi** con cui identificare i dispositivi (in Unix sono file in /dev con un nome simbolico a cui viene associato un id unico)
- Gestisce i **malfunzionamenti**, almeno quelli che ha senso sollevare fino al livello di astrazione dell'utente
- **Alloca i dispositivi** ai processi applicativi, ossia definisce le richieste di accesso ai dispositivi, poi il modo in cui sono gestite le code dipende dal demone che gestisce la particolare periferica
- Implementa i meccanismi di **protezione**, ossia si assicura che chi accede al dispositivo ne abbia il diritto (ad esempio in Unix essendo i dispositivi file, ne si può controllare utente e gruppo)

Vediamo adesso la parte del sottosistema di I/O dipendente dai dispositivi (rettangolo verde, anche detta **device drivers**):

- **Sincronizza** una periferica con il processo che l'ha attivata (definisce i semafori), principalmente per problemi di differenza di velocità
- **Dialoga con i controllori** dei particolari dispositivi, a partire dai dati in ingresso della parte indipendente del sottosistema di I/O
- **Definisce il descrittore di dispositivo**, il quale contiene:
 - Indirizzi dei registri nello spazio di I/O
 - Semaforo *dato_disponibile* inizializzato a 0

- Contatore dei byte trasferiti o da trasferire
- Puntatore al buffer in memoria fisica
- Esito dell'operazione

Il flusso di esecuzione di un processo che vuole utilizzare una periferica è dunque il seguente:



In particolare la `_read(...)` del device driver può essere implementata nel seguente modo:

```
int _read(int dispositivo, char *buffer, int contatore) {
    des_dispositivi[dispositivo].contatore = contatore;
    des_dispositivi[dispositivo].buffer = buffer;

    <Attivazione del dispositivo>
    // Scrittura del bit start nel registro di controllo

    des_dispositivi[dispositivo].dato_disponibile.wait();
    // Il processo applicativo che ha continuato ad eseguire
    // la primitiva (senza cambiare contesto) si sospende

    if (des_dispositivi[dispositivo].esito == ERRORE)
        return -1;
    return contatore - des_dispositivi[dispositivo].contatore;
}
```

Mentre una routine di gestione delle interruzioni può essere la seguente:


```

void inth(...) {
    registro_stato = <Lettura del registro di stato del dispositivo>;
    if (registro_stato & bit_errore) {
        <Routine di gestione errore>
        if (<Errore non recuperabile>)
            des_dispositivi[dispositivo].esito = ERRORE;
        des_dispositivi[dispositivo].dato_disponibile.signal();
        return;
    }

    registro_dati = <Lettura registro dati>;
    *des_dispositivi[dispositivo].buffer = registro_dati;
    des_dispositivi[dispositivo].contatore--;
    des_dispositivi[dispositivo].buffer++;

    if (des_dispositivi[dispositivo].contatore) {
        <Riattiva dispositivo>
    }
    else {
        <Disattiva il dispositivo>
        des_dispositivi[dispositivo].esito = NO_ERRORE;
        des_dispositivi[dispositivo].dato_disponibile.signal();
    }
}

```

Esempi di periferiche

Timer

Prima abbiamo visto in modo generico un descrittore di dispositivo, vediamo adesso in particolare come può essere quello del timer:

- Indirizzi dei registri nello spazio di I/O (in particolare il registro dati posso anche chiamarlo contatore)
- Array di N semafori *fine_attesa* inizializzati a 0
- Array di N interi *ritardo* inizializzati con i tempi che ogni processo vuole attendere

Vediamo ora il device driver per un generico timer:

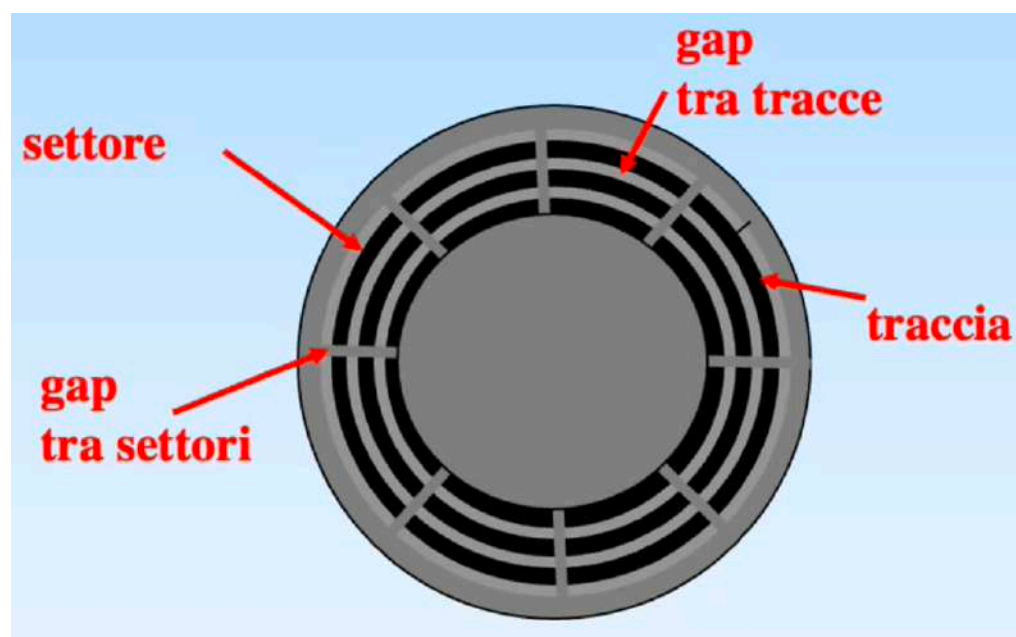
```
void _delay(int periodi) {
    int processo = <PID del processo in esecuzione>;
    // Il tempo di sospensione viene espresso in periodi del timer
    des_timer.ritardo[processo] = periodi;
    des_timer.fine_attesa[processo].wait();
}
```

e la corrispondente routine di gestione delle interruzioni:

```
void inth() {
    for (int i = 0; i < N; i++) {
        if (!des_timer.ritardo[i])
            continue;
        des_timer.ritardo[i]--;
        if (!des_timer.ritardo[i])
            des_timer.fine_attesa[i].signal();
    }
}
```

Disco

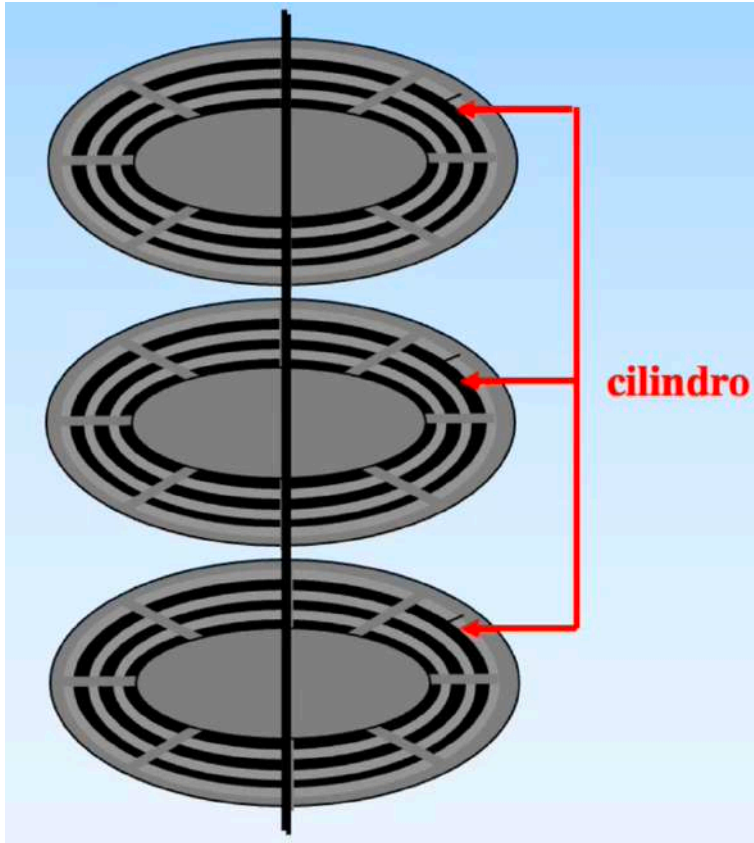
Il disco è una di quelle periferiche in cui l'ordine di accesso è molto importante e dipende dalla sua struttura fisica, che richiamiamo brevemente adesso:



Il settore è la minima unità di trasferimento del disco, anche se questi hanno lunghezze diverse, quelli più esterni sono meno densi, per cui la loro dimensione in byte rimane

Non raffigurata c'è una testina che si sposta tra le varie tracce per leggere o scrivere i vari settori, che, per effetto della rotazione del disco, prima o poi finiscono sotto di essa

Nel caso di più dischi concentrici, si definisce cilindro l'insieme delle tracce sui vari dischi alla stessa distanza dall'asse di rotazione



Una traccia può essere magnetizzata sia sopra che sotto

Dal punto di vista logico un disco può essere visto, indipendentemente dalla sua struttura fisica, come array di settori, indicizzati dalla terna (faccia, traccia, settore), la primitiva di lettura/scrittura dovrà dunque avere modo di tradurre la posizione di un file sul filesystem nella terna illustrata

Dal punto di vista delle prestazioni ci può interessare il tempo medio di trasferimento di un singolo settore $T = ST + RL + TT$:

- $ST = \text{Seek Time}$: tempo medio che impiega la testina per arrivare sulla traccia richiesta
- $RL = \text{Rotational Latency}$: tempo medio che impiega il settore per fare il giro e arrivare sotto la testina
- $TT = \text{Transfer Time}$: tempo medio che impiega il settore per scorrere interamente sotto la testina

Per dare un senso a questi tempi vediamo un esempio di datasheet con i relativi conti:

Esempio di datasheet	#
Facce per cilindro	8
Tracce per faccia	13614
Settori per traccia	320
Byte per settore	512
Seek time minimo	0.6 ms
Seek time medio	5.2 ms
Rotational latency	6.0 ms
Tempo lettura settore	19 μ s

Vogliamo calcolare il tempo medio di trasferimento per un file da 320kB allocato su settori contigui del disco, innanzitutto possiamo notare che questo non sta su una sola traccia ma due, infatti $320'000/512 = 625 < 640 = 320 * 2$, allora:

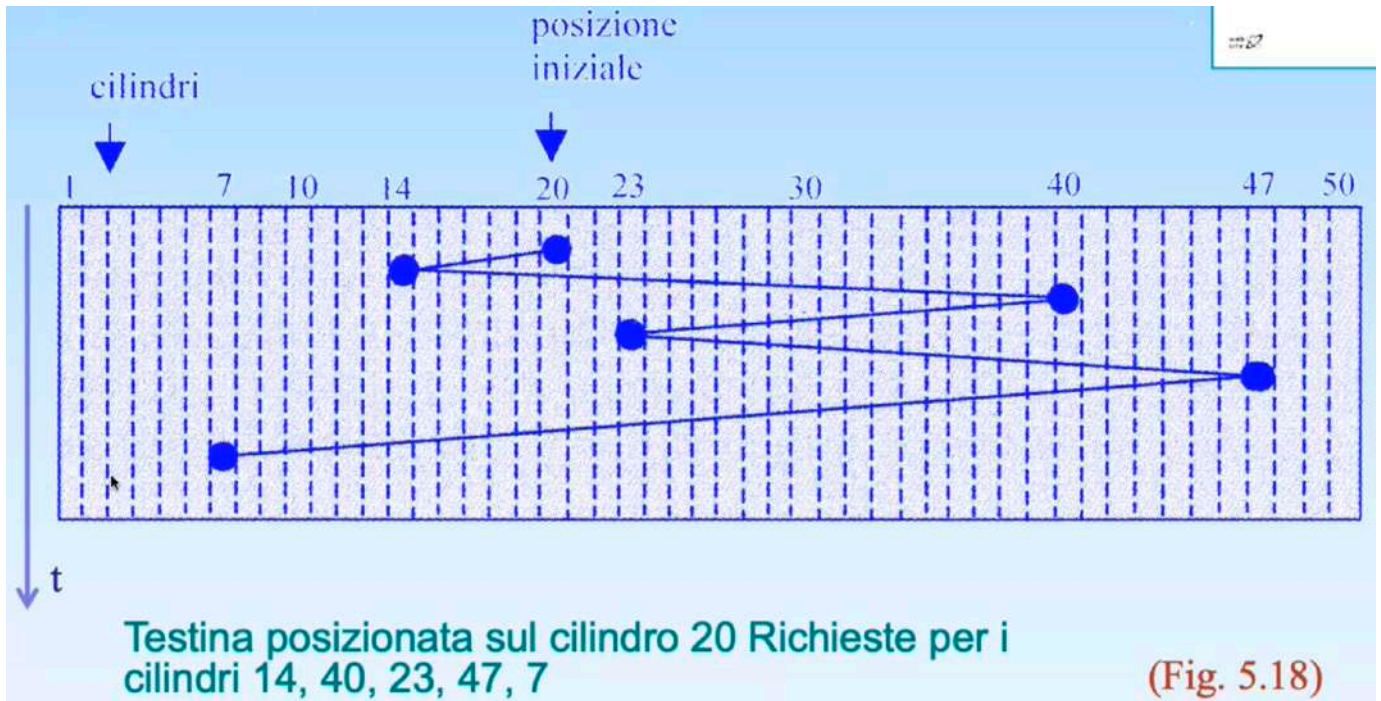
```
5.2      Sposta la testina sulla prima traccia
+ 6.0/2   Aspetta che il primo settore utile arrivi
+ 6.0     Aspetta un giro completo
+ 0.6     Sposta la testina sulla traccia adiacente
+ 6.0/2   Aspetta che il primo settore utile arrivi
+ 6.0 * 625/640 Aspetta (quasi) un giro completo
= 23.659375 ms
```

Se invece il file fosse allocato in settori casuali e non contigui?

```
( 5.2      Sposta la testina sulla prima traccia
+ 6.0/2    Aspetta che il primo settore utile arrivi
+ 0.019    Trasferisci il singolo settore
) * 625    Numero di settori da leggere
= 5136.875 ms
```

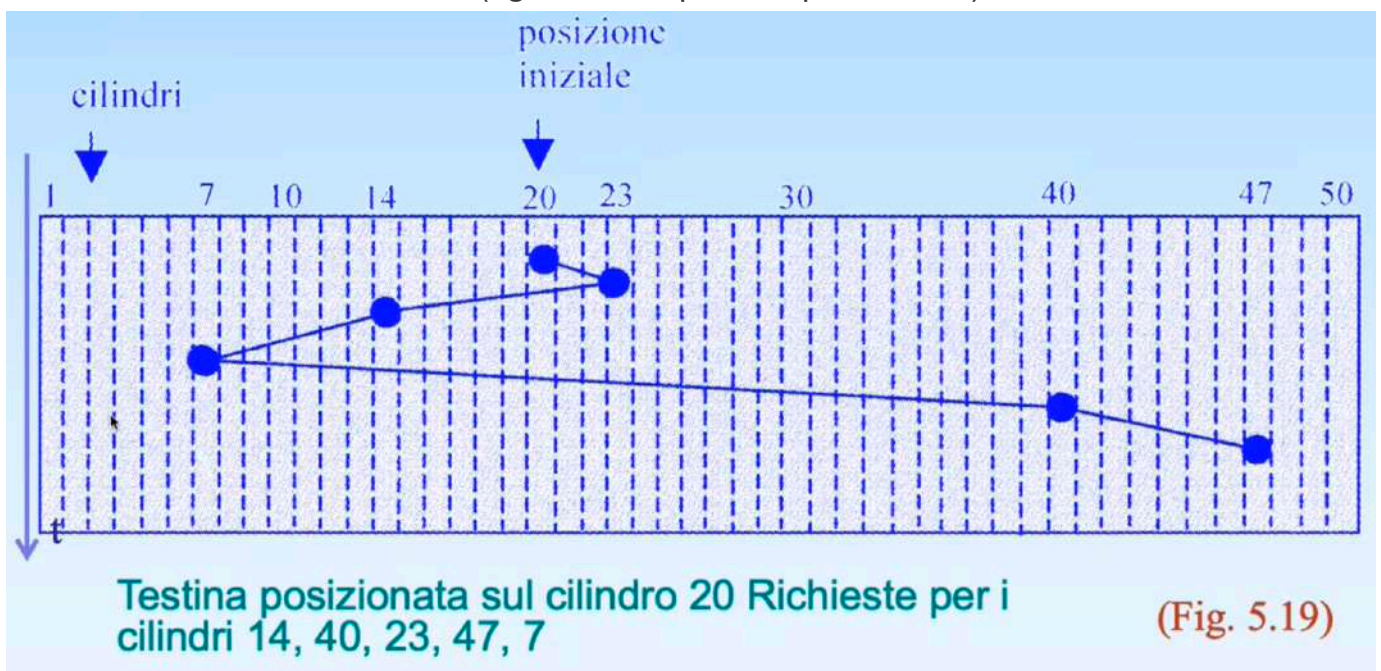
Osserviamo dunque che leggere in modo contiguo è estremamente importante, in particolare la variabile da minimizzare è il *seek time*, vediamo dunque alcuni algoritmi di scheduling possibili, tenendo solo conto della posizione delle tracce da leggere:

- **FCFS**



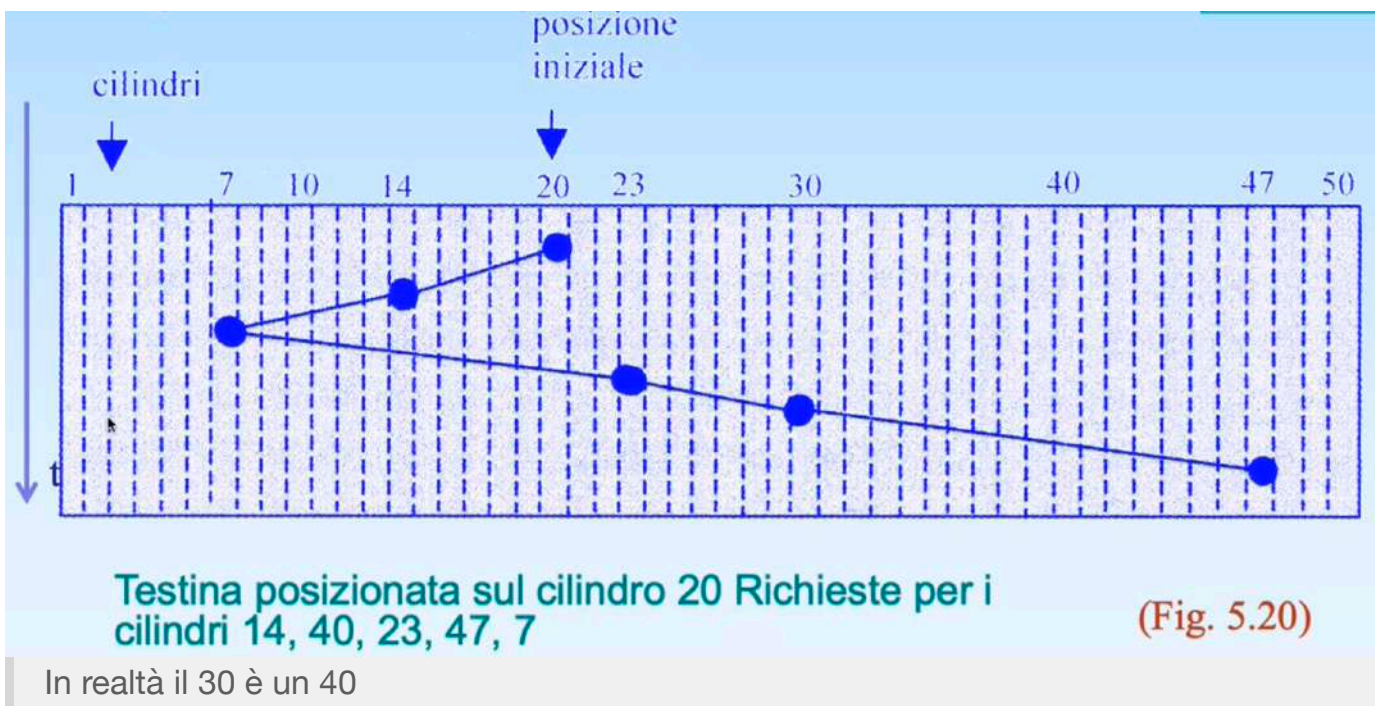
Il tempo di attesa per questa configurazione è $113 \cdot \text{Seek time minimo}$, il problema è che è aleatorio rispetto all'ordine delle richieste

- **SSTF** Shortest Seek Time First (ogni volta rispetto al precedente)



Il tempo di attesa per questa configurazione è $59 \cdot \text{Seek time minimo}$, tuttavia c'è tanto overhead ed è possibile, per le tracce più "lontane", entrare in starvation (servirebbe ageing con timestamp)

- **SCAN**



Come gli ascensori, una volta che la testina si muove in una direzione accetta solo fermate lungo tale direzione, serve dunque memorizzare un bit per il verso della testina

Il tempo di attesa per questa configurazione è $53 \cdot \text{Seek time minimo}$ ed è il tempo ottimo

Spesso serve più tempo a fermare la testina al centro del disco rispetto che agli estremi, una possibile variante di questo approccio è dunque far oscillare sempre la testina tra i due estremi, anche se non ci sono richieste da servire (probabilmente ne arriveranno)

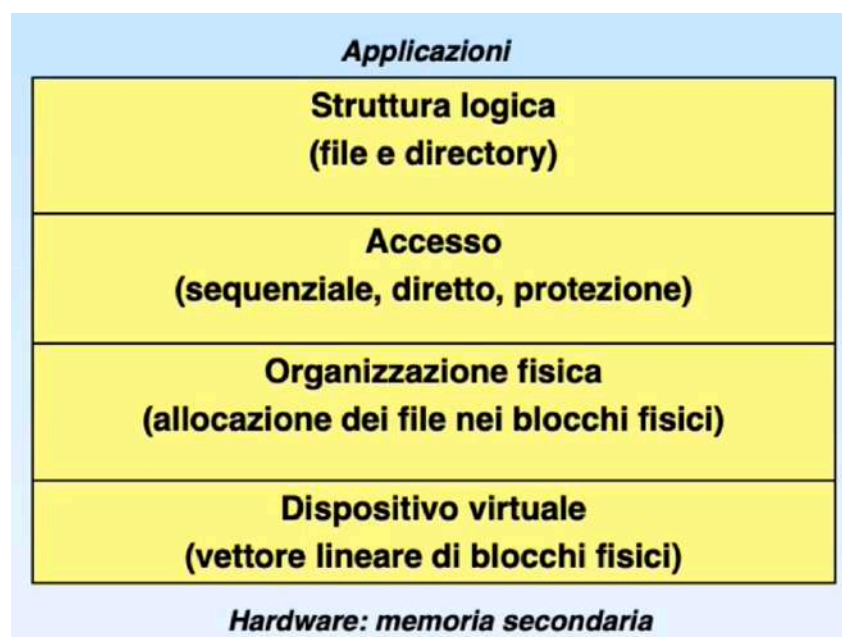
6. Il file system

Il file system è quella parte del SO che fornisce i meccanismi necessari per l'accesso e l'archiviazione delle informazioni in memoria secondaria

Realizza i concetti astratti di:

- **File:** insieme di informazioni memorizzate mediante record logici, caratterizzate, ad esempio, dai seguenti attributi:
 - Tipo: ad esempio file ordinario o directory
 - Nome: nome simbolico la cui sintassi è regolata dal file system
 - Estensione: interpretazione da dare al contenuto del file
 - Indirizzo: qualche tipo di puntatore alla memoria secondaria
 - Dimensione, Proprietarietà, protezione, data di creazione e di modifica...
- **Directory:** file speciale che rappresenta un elenco di altri file in relazione di contenimento essa
- **Partizione:** disco logico mappato su un disco fisico (o parte di esso)

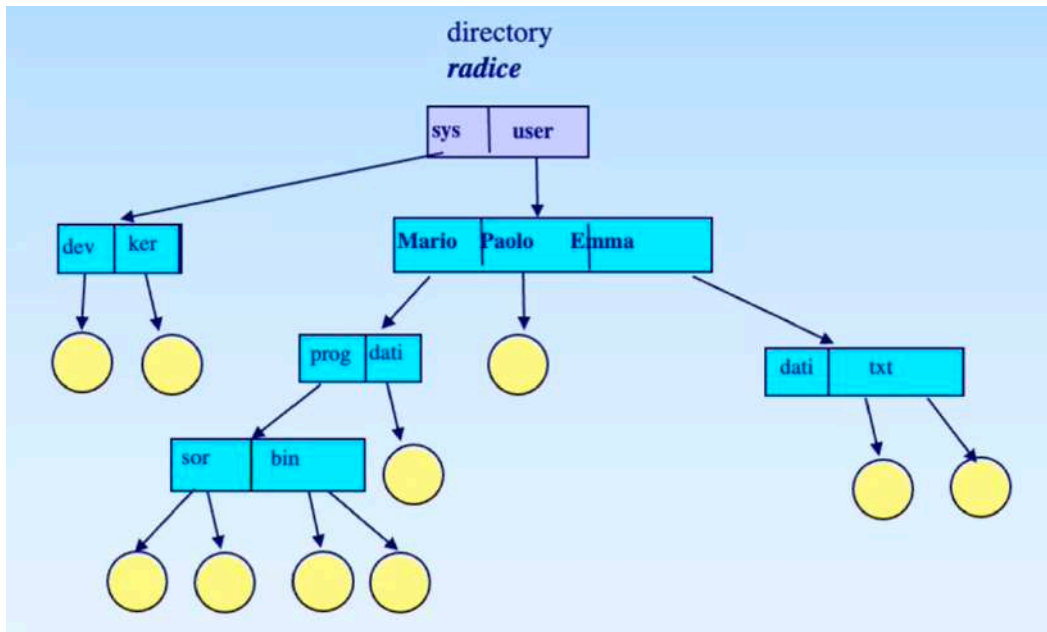
La struttura di un file system può generalmente essere rappresentata da un insieme di componenti organizzate su più livelli:



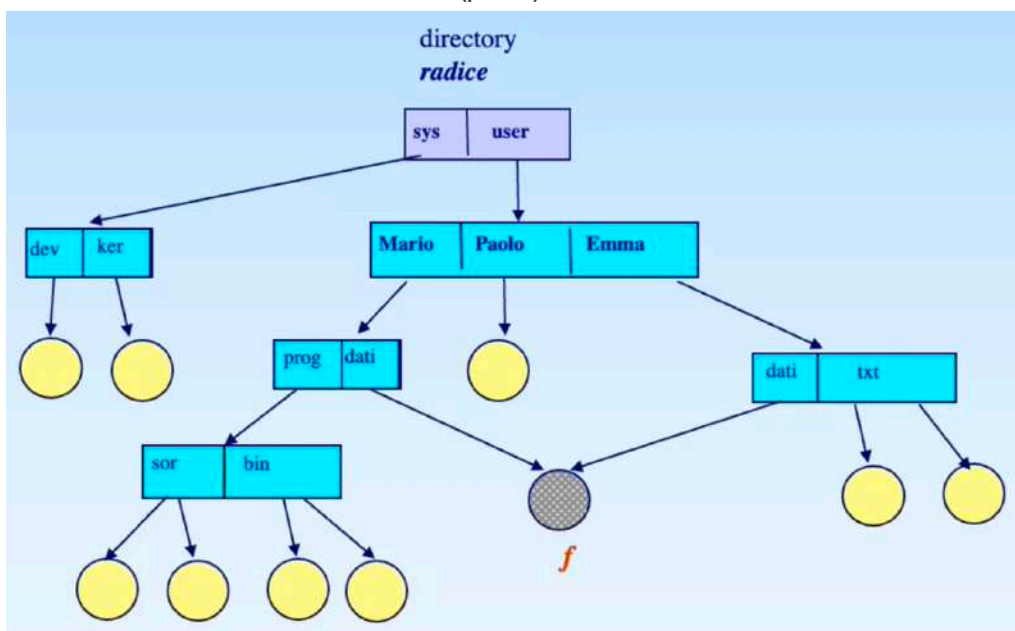
Struttura logica

Il livello della struttura logica mette a disposizione dell'utente le primitive che permettono di interagire col file system (ad esempio creazione, cancellazione e navigazione di file e directory) e ne definisce l'organizzazione logica, questa può essere:

- Ad albero



- A grafo diretto aciclico: permette, tramite il meccanismo del linking, di poter raggiungere lo stesso file con nomi simbolici (path) diversi



Accesso

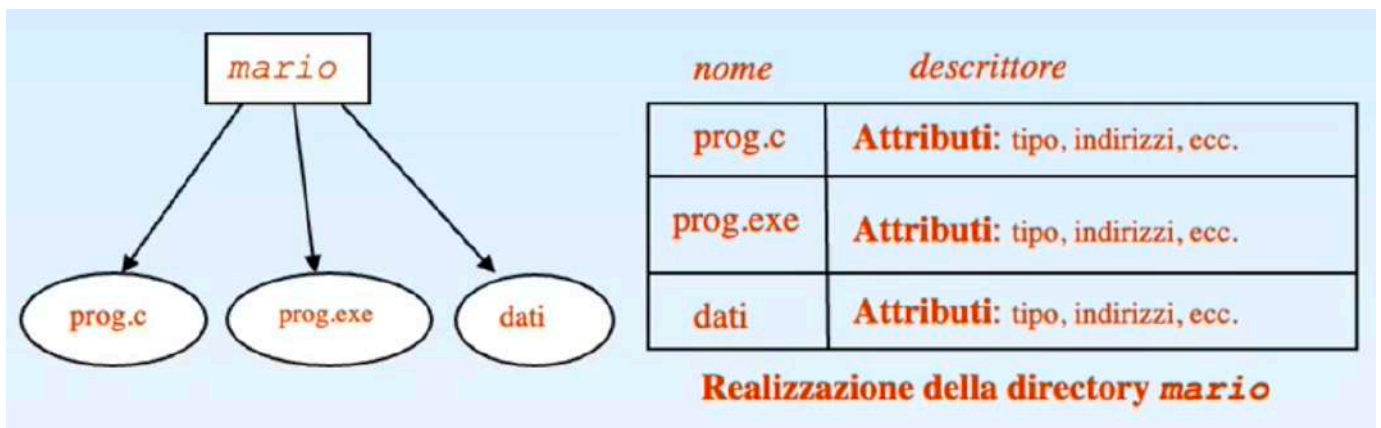
Il livello dell'accesso definisce come i processi, tramite system call o funzioni di libreria, accedono al file system, rispettando le regole di protezione (che vedremo più avanti).

Abbiamo visto che un disco (ma anche ogni altro elemento di memoria secondaria) è organizzato in **settori**, grazie al livello più basso, quello del **dispositivo virtuale**, sappiamo inoltre che tali settori vengono rappresentati all'interno del SO come un vettore di **blocchi**, per motivi di ottimizzazione dei trasferimenti la dimensione di un blocco è solitamente nell'ordine dei kilobyte (sicuramente più grande di quella di un singolo settore), per questo motivo c'è bisogno di scomporre ulteriormente un blocco in **record** (o blocchi logici), di dimensione molto minore (in Unix un singolo byte)

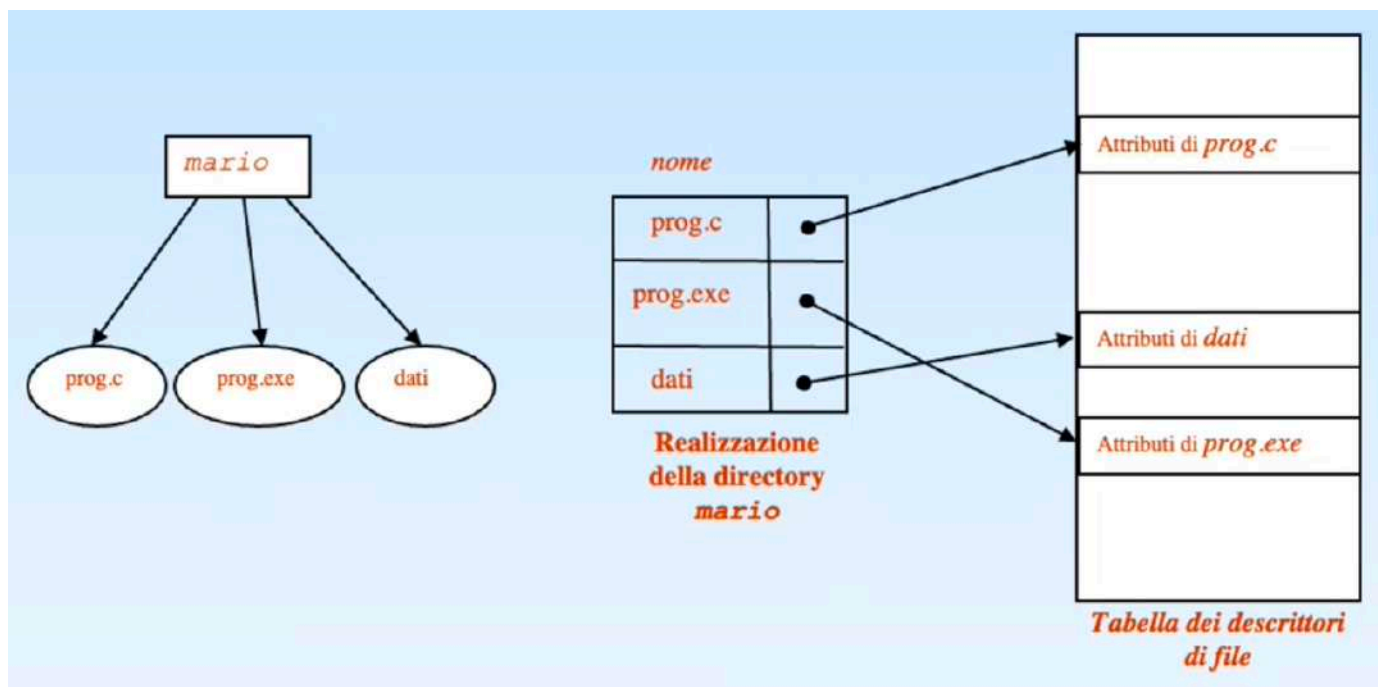
Ogni file viene definito da un descrittore, ed il suo contenuto viene interpretato come array di record, il livello dell'accesso si occupa dunque di definire il modo in cui vengono memorizzati i descrittori dei file e quello in cui si accede ai record di un file:

I descrittori dei file devono essere memorizzati in modo persistente, mediante apposite strutture in memoria secondaria, per ottenere questo si possono seguire due approcci:

- Approccio di windows: i descrittori dei file sono distribuiti nel file system stesso, in altre parole il contenuto di una directory è una tabella contenete i descrittori dei file logicamente al suo interno



- Approccio di Unix: i descrittori dei file sono centralizzati in una tabella, in altre parole il contenuto di una directory è una tabella contenete i puntatori ai descrittori dei file logicamente al suo interno



L'approccio di Unix comporta una lettura aggiuntiva del disco per raggiungere un determinato file, però garantisce una ricerca più efficiente grazie alla centralizzazione dei descrittori

Supponiamo adesso di voler accedere ad un file, quest'operazione ha un costo non trascurabile perchè bisogna percorrere tutto l'albero del file system partendo dalla radice, fino ad arrivare al suo descrittore, inoltre, anche una volta ottenuto un descrittore di file, accedere ad un suo record comporta sicuramente letture aggiuntive sul disco (come vedremo nel livello successivo), per questo motivo si subordina l'accesso ai file ad operazioni di apertura, la quale carica in memoria principale i descrittori dei file ed altre informazioni utili su record e processi (come vedremo in Unix), informazioni che poi vengono rimosse con l'operazione di chiusura del file, alcuni sistemi fanno inoltre **memory mapping**, ossia copiano parte o tutto il contenuto di certi file in memoria principale

Una volta aperto un file, bisogna comunque trovare un modo per accedere ai suoi record, le opzioni principali sono le seguenti:

- **Sequenziale:** una volta aperto un file viene letto sempre a partire dal primo record, sequenzialmente, fino ad arrivare a quello desiderato, in altre parole per accedere all' i -esimo record devo aver acceduto a tutti gli $i - 1$ precedenti

```
// Le system call saranno del tipo:
readnext(file, &buffer)
writenext(file, buffer)
```

- **Diretto:** il record da leggere è specificato a piacere dall'utente, questa soluzione sembra la più intuitiva ma vedremo nel prossimo livello che a volte è meno efficiente dell'accesso sequenziale

```
// Le system call saranno del tipo: (d sta per directed)
readd(file, i, &buffer)
writed(file, i, buffer)
```

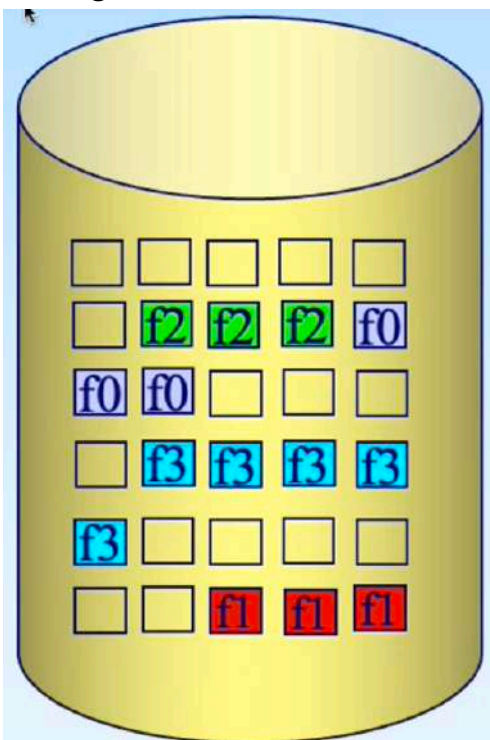
- **Ad indice:** ogni file contiene una struttura dati che ad ogni record associa una chiave (ha origine nei database)

```
// Le system call saranno del tipo: (k sta per key)
readk(file, key, &buffer)
writek(file, key, buffer)
```

Organizzazione fisica

Il livello dell'organizzazione fisica definisce le tecniche di allocazione dei file all'interno del dispositivo virtuale (file e directory sono indistinguibili a questo livello), in altre parole questo livello mappa i record sui blocchi, supponiamo per semplicità che ogni blocco contenga al più record contigui dello stesso file, abbiamo allora tre tecniche di allocazione principali:

- **Contigua**



Il grande punto a favore di questo approccio è che, una volta aperto un file, il tempo impiegato per calcolare il blocco in cui si trova un suo qualsiasi record è nullo (supponendo

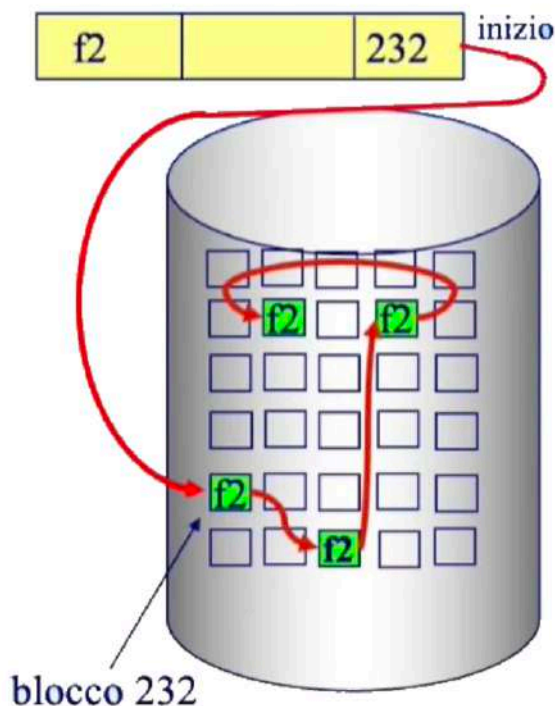
che nel descrittore di file sia scritto il primo blocco occupato), le tre strategie di accesso viste prima sono dunque equivalenti dal punto di vista della velocità di calcolo del blocco contenente il record richiesto

I problemi di questo approccio sono però molteplici:

- La frammentazione esterna è notevole (quella interna è inevitabile ed è data dal fatto che l'ultimo blocco conterrà quasi certamente record vuoti)
- L'allocazione di un nuovo file richiede una lunga ricerca di spazio libero contiguo
- Bisogna gestire la crescita dei file

La frammentazione esterna è mitigabile tramite compattazione e gestione *worst-fit* della lista dei blocchi liberi

- **A lista**



Ogni blocco contiene l'informazione necessaria per puntare a quello successivo, i vantaggi di questo approccio sono che non c'è frammentazione esterna ed allocare un nuovo file è un'operazione veloce, tuttavia, oltre al fatto che i puntatori occupano spazio, l'unico vero svantaggio è che se anche solo un bit di un collegamento viene danneggiato allora praticamente perdo il file

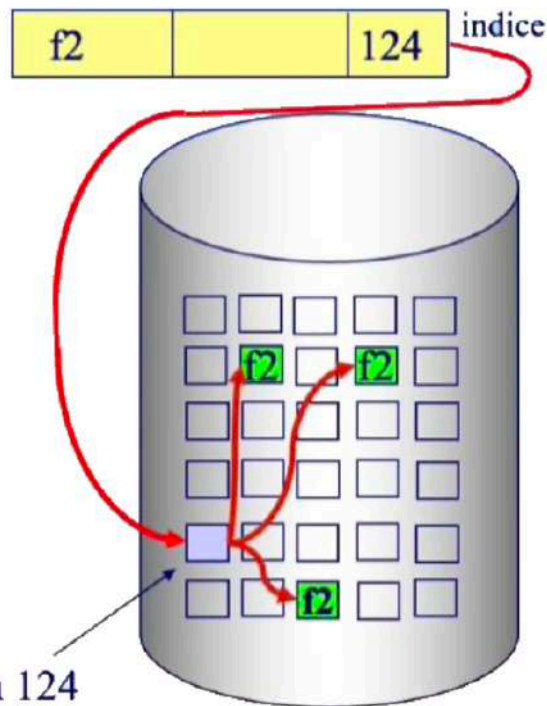
Se come criterio di costo adottiamo il numero di accessi al disco per capire il blocco in cui si trova un determinato record di un file, allora:

- L'accesso sequenziale è da preferire perchè ogni volta si può scrivere in memoria principale l'indice del blocco a cui accedere successivamente
- L'accesso diretto è oneroso, in particolare il numero di accessi al disco necessari equivale alla parte intera di $\frac{\text{Record desiderato}}{\text{Numero di record per blocco}}$

Per aumentare la fault tolerance è possibile utilizzare una lista doppiamente concatenata, ed inserire nel descrittore del file anche l'indice dell'ultimo blocco occupato, un'altra possibile soluzione è quella adottata da windows: viene introdotta una tabella per file, detta FAT, che riporta per ogni blocco quale è il successivo in lista

La FAT è di dimensioni ragionevoli per essere copiata in memoria centrale, di conseguenza anche l'accesso diretto non è più così lento

- **Ad indice**



Blocco indice n 124

Nel descrittore di ogni file viene memorizzato l'indice di un blocco il cui contenuto è una tabella con la posizione dei blocchi che compongono effettivamente il contenuto del file

Il vantaggio principale di questo approccio è sicuramente la maggiore velocità di accesso (posso copiare in memoria secondaria il blocco indice in fase di apertura del file), l'unico contro è invece la limitata scalabilità, che però può essere risolta introducendo indici multilivello (come vedremo in Unix)

Con questo approccio inoltre le strategie di accesso diretto e sequenziale sono equivalenti!

7. Protezione e sicurezza

Mentre la **sicurezza** garantisce l'**autenticazione** degli utenti, impedendo operazioni non autorizzate, la **protezione** riguarda il **controllo dell'accesso** alle risorse logiche e fisiche da parte degli utenti, noi ci concentreremo su quest'ultima, è divisa in tre livelli concettuali:

- **Modelli**

Definiscono i **soggetti** (la parte attiva del sistema), gli **oggetti** (la parte passiva del sistema, ad esempio le risorse) e i **diritti di accesso** tra questi ultimi (ossia le operazioni che i soggetti possono compiere sugli oggetti)

In particolare un soggetto è una coppia (processo, dominio), dove il **dominio** (o ambiente di protezione) è un'insieme di diritti di accesso (praticamente una lista oggetti → diritti di accesso)

Tale associazione tra processo e dominio può essere:

- Statica: un processo rimane sempre uno stesso soggetto, non adatta al **principio del privilegio minimo** (ad un soggetto sono garantiti i diritti di accesso ai solo oggetti strettamente necessari per la sua esecuzione), inoltre potremmo non sapere neanche gli oggetti che un processo richiederà prima di eseguirlo
- Dinamica: un processo può cambiare a runtime il soggetto per modificare i suoi diritti di accesso, ad esempio in Unix questo avviene tramite il meccanismo delle system call e quello della exec su file con `suid = 1`

- **Politiche**

Definiscono cosa viene fatto, ossia le regole con le quali si possono attribuire i diritti di accesso tra soggetti ed oggetti, alcuni esempi possono essere:

- DAC (Discretionary Access Control): il creatore di un oggetto ne controlla pienamente i diritti (es Unix), sconsigliato per sistemi che contengono informazioni sensibili
- MAC (Mandatory Access Control): i diritti di accesso vengono gestiti centralmente
- RBAC (Role-Based Access Control): i diritti di accesso sono assegnati ad un ruolo e non ad un particolare utente

- **Meccanismi**

Definiscono come le cose vengono fatte, ossia gli strumenti messi a disposizione per

imporre determinate politiche

Vediamo il modello di protezione più diffuso nei SO, ossia il **modello a matrice degli accessi**, modello ad associazione dinamica che permette oltretutto di definire diritti di accesso tra due soggetti

	X_1	X_2	X_3	S_1	S_2	S_3
S_1	read*	read	execute		terminate	receive
S_2		owner write		control receive		terminate
S_3	write execute		read	send	send receive	

In colonna ci sono gli oggetti, in riga i soggetti (in Unix terne ProcessID, UserID, GroupID), il contenuto di una riga rappresenta il dominio di un particolare soggetto

Il meccanismo associato al modello dovrà dunque garantire le seguenti operazioni:

- Ogni volta che arriva una nuova richiesta di accesso da parte di un soggetto verificare se è consentita o meno
- Modificare dinamicamente dominio di un processo
- Modificare dinamicamente numero di soggetti ed oggetti
- Modificare dinamicamente, in modo controllato lo stato di protezione

Per quanto riguarda i diritti di accesso, Graham e Denning hanno definito un set di regole che permettono di modificare dinamicamente lo stato di protezione in modo controllato:

- Aggiunta dei diritti di accesso

Il soggetto S_i può aggiungere il diritto di accesso α sull'oggetto O al soggetto S_j solamente se è **owner** di O

- Rimozione dei diritti di accesso

Il soggetto S_i può rimuovere il diritto di accesso α sull'oggetto O al soggetto S_j se è owner di O o se ha il diritto **control** su S_j

- Propagazione dei diritti di accesso

Il soggetto S_i può trasferire il diritto di accesso α sull'oggetto O al soggetto S_j se α ha il **copy flag**

Può essere trasferito anche il *copy flag* stesso, oppure può anche essere prevista la rimozione del diritto trasferito al soggetto trasferente

Il modello è stato dunque definito, per quanto riguarda i meccanismi che lo implementano ci sono però alcuni problemi:

- La matrice ha dimensione enorme, soprattutto il numero di colonne (basti pensare che c'è almeno una colonna per file nel sistema)
- La matrice è sparsa, soprattutto se viene rispettato il principio del privilegio minimo
- Se un'operazione può essere eseguita su tutti gli oggetti, deve essere replicata in tutti i domini

Sorge dunque la necessità di dover implementare la matrice degli accessi in un'altra forma più efficiente, per fare questo ci sono due possibilità:

- **Access Control List:** la matrice viene memorizzata per colonne, ad ogni oggetto è dunque associata una lista di soggetti che hanno diritti su di esso, in realtà può anche essere usata per gruppi di utenti, introducendo il gruppo nel generico elemento della lista: UID, GID → Insieme di diritti (*anche vuoto volendo*), con possibilità di wildcard *

Ha senso immaginare che esistano ACL di default applicabili ad ogni oggetto

- **Capability List:** la matrice viene memorizzata per righe, ad ogni soggetto sono dunque associati i diritti di accesso che ha sui vari oggetti

Unix utilizza entrambe le soluzioni, le ACL sono i vettori dei bit di protezione, sono distribuite nei vari i-node in memoria secondaria e vengono controllate solo in fase di apertura del file, le CL sono invece le tabelle dei file aperti di processo, e vengono controllate per gli accessi a runtime

Unix fa in questo modo perchè controllare una CL è più efficiente, mentre le ACL sono comunque necessarie per mantenere la persistenza dei dati

Vediamo infine come potrebbe essere implementato il meccanismo di revoca dei diritti a runtime, innanzitutto definiamo che tipologia di revoca vogliamo fare:

- Selettiva o generale: tutti gli utenti, un gruppo o un utente singolo
- Parziale o totale: tutti i diritti oppure un sottoinsieme di diritti
- Temporanea o permanente: indica se il diritto può essere successivamente riottenuto o meno

Se la matrice degli accessi viene implementata come ACL allora è facile, basta prendere la lista dell'oggetto desiderato e modificarla, se invece la matrice degli accessi è implementata come capability list non è tutto così semplice, per ogni soggetto devo scorrere la sua CL, vedere se contiene l'oggetto target, ed infine applicare le modifiche desiderate

La matrice degli accessi da sola non basta per evitare il problema del **cavallo di troia**: un soggetto che ha diritto di leggere un particolare oggetto può copiarlo su un secondo oggetto creato da lui stesso e poi distribuirne i diritti di accesso a chiunque

Per risolvere questo problema abbiamo bisogno di un modello con regole aggiuntive e compatibili con la matrice degli accessi, questo è ad esempio il **modello di Bell-La Padula**

Tale modello presuppone un sistema dove sia i documenti che i soggetti sono classificati secondo classi di sicurezza, e definisce le seguenti regole di interazione:

- Proprietà semplice sicurezza: un soggetto in esecuzione ad un certo livello di sicurezza può leggere solamente oggetti che si trovano ad un livello di sicurezza uguale o inferiore
- Proprietà *: un soggetto in esecuzione ad un certo livello di sicurezza può scrivere solamente oggetti che si trovano ad un livello di sicurezza uguale o superiore

Questo modello, impiegato da solo, risolve il problema del controllo del flusso di informazione, ma non si preoccupa dell'integrità di tale flusso, per questo motivo fa buona coppia con il modello a matrice degli accessi, nel seguente esempio infatti il cavallo di troia O_2 installato da S_2 viene bloccato:

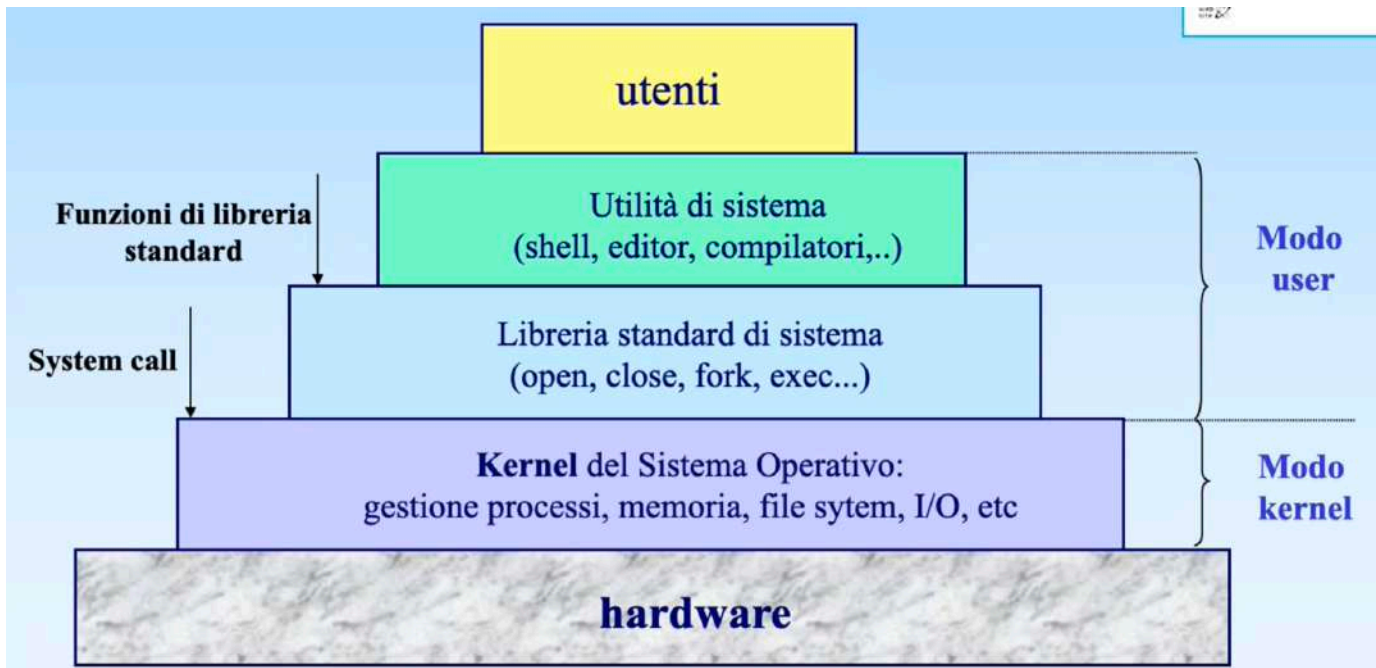
	O_1 (Segreto)	O_2 (Pubblico)	O_3 (Pubblico)
S_1 (Segreto)	Read	Execute	Write
S_2 (Pubblico)		Execute, Owner	Read, Owner

Menzioniamo per finire un secondo modello, chiamato **BIBA**, che applica le regole di Bell-La Padula esattamente al contrario, garantendo dunque l'integrità del flusso di informazione ma non il suo controllo

8. Architettura del sistema Unix

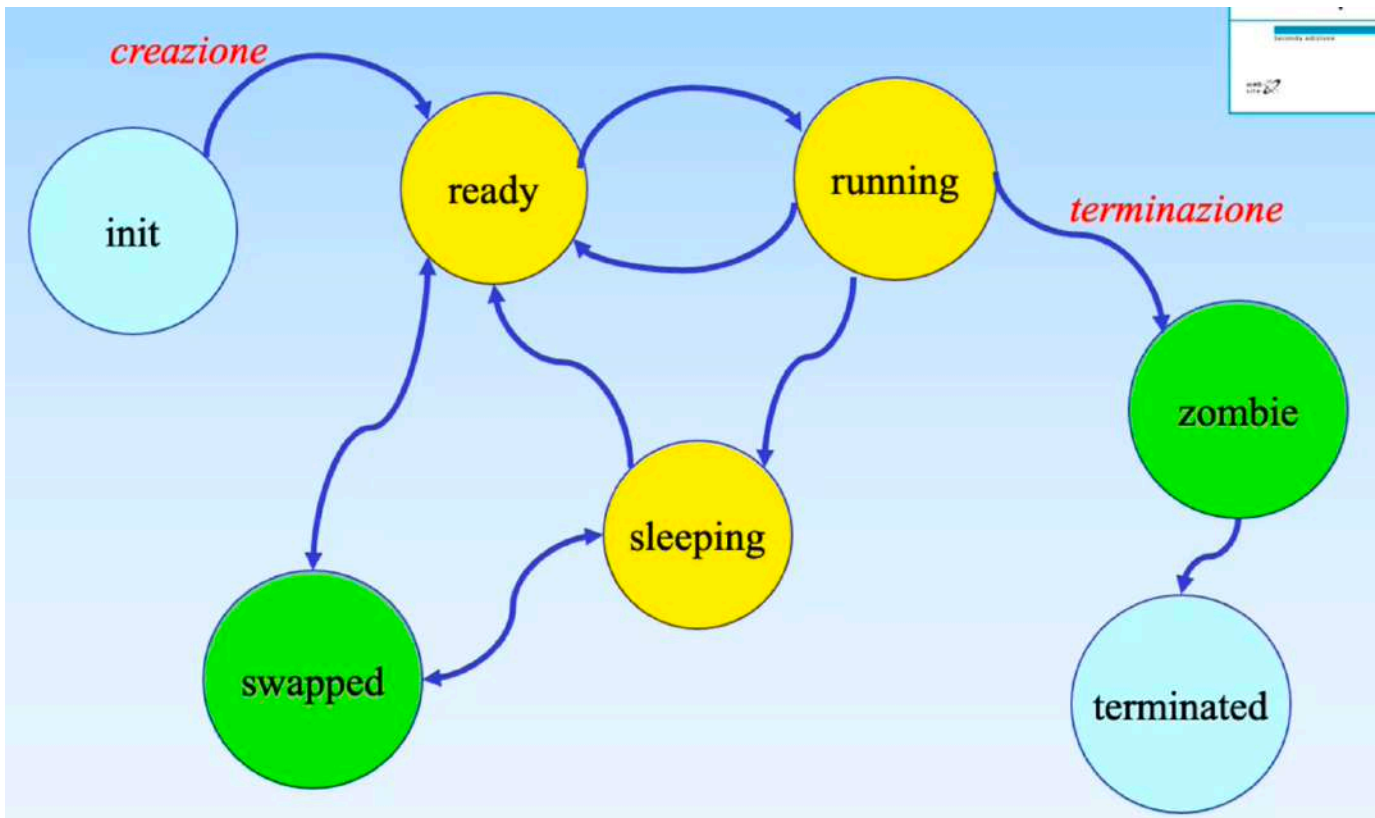
Scheduling, processi e gestione della memoria

Unix nasce come sistema multiprogrammato **modulare** (non monolitico), la sua struttura è infatti organizzata con i seguenti moduli:



Unix opera nella cosiddetta "**dual mode**", ossia modo utente o modo sistema, a seconda dei flag di privilegio contenuti nel registro di stato

Unix gestisce la memoria tramite la tecnica della **segmentazione con paginazione** (la tabella dei frame è chiamata *core map*), dunque con caricamento su domanda, nonostante questo è presente uno **stato swapped** per conseguenza del particolare modo in cui viene fatta l'operazione di lo swap out delle pagine (vedremo che in certe occasioni viene swappato un intero processo anzichè singole pagine)



I processi *zombie* sono ad esempio quei processi figli che aspettano solamente che il padre legga il proprio stato di terminazione

L'algoritmo di rimpiazzamento delle pagine è **second chance**, tuttavia, come anticipato prima, viene usato in modo particolare:

- *lotsfree* è il numero minimo di frame liberi necessari per evitare di dover eseguire rimpiazzamento di pagine in occasione di page fault: un processo in background detto *page daemon* esegue l'algoritmo di *second chance* ogni volta che i frame liberi sono minori di *lotsfree*
- *minfree* è il numero minimo di frame liberi necessari per evitare di dover effettuare swap dei processi in caso di page fault, potrebbe infatti succedere che processi detti *page hungry* provochino page fault più veloce di quanto *second chance* riesca a gestire, in questi casi ci pensa ancora un processo in background detto *swapper daemon* che esegue lo swap di un processo, liberando molte pagine in un colpo solo (ecco perchè esiste lo stato *swapped*)

Tuttavia, se *swapper* eseguisse subito quando il numero di frame liberi è minore di *minfree*, allora si potrebbe andare in trashing, per questo motivo è stato definito *desfree* come numero minimo di frame desiderabili per evitare di andare in trashing con *minfree*: se lo *swapper daemon* esegue quando il numero di frame liberi è minore di *minfree* e allo stesso tempo il numero medio di frame liberi per unità di tempo è minore di *desfree* allora il sistema non finisce in trashing

$lotsfree > desfree > minfree$

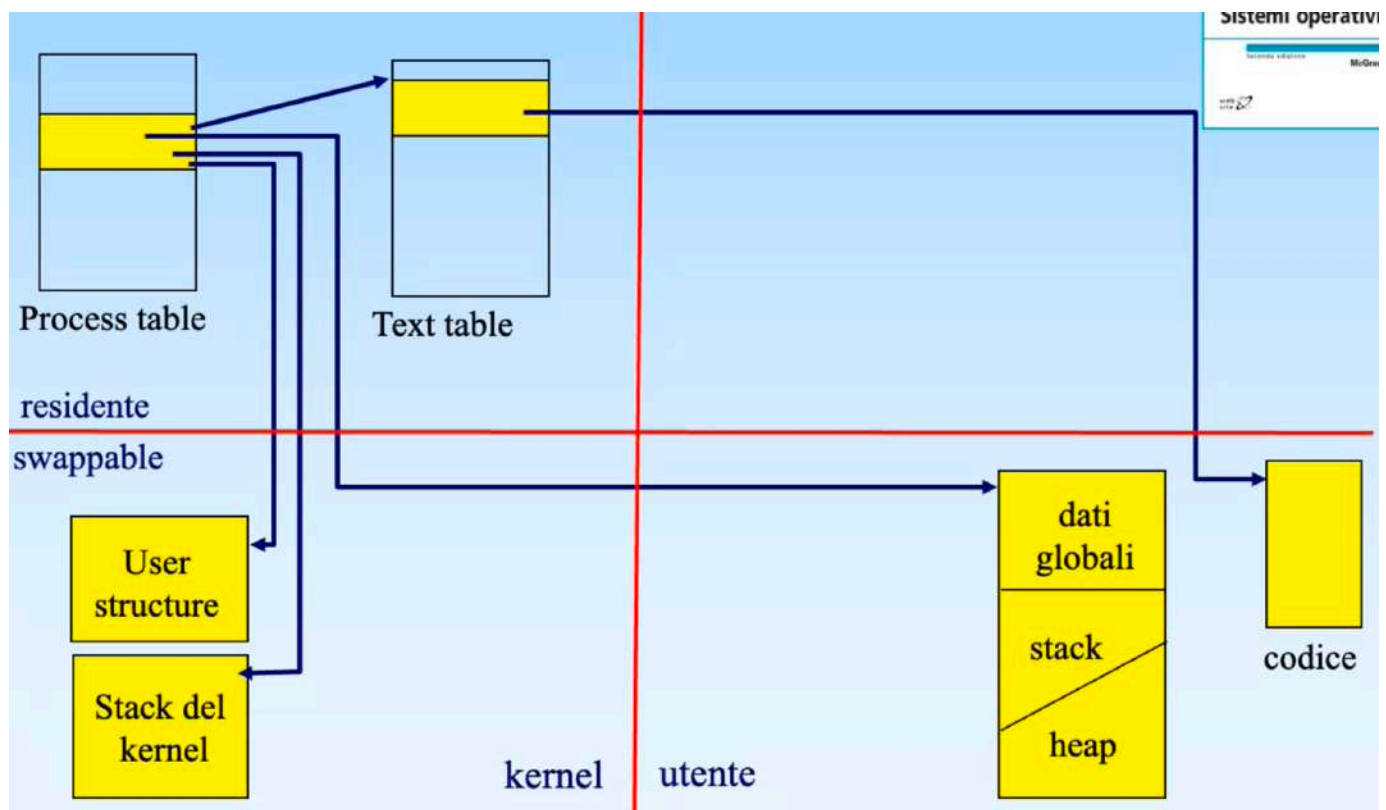
In Unix i processi hanno codice detto *rientrante*, ossia condivisibile (a differenza dei dati, per i quali si usano messaggi), questo è possibile grazie alla **text table**, una tabella che, tramite *text structure*, memorizza puntatori ai segmenti di codice (eventualmente in memoria secondaria se swapped) e contatori che tengono conto dei processi che utilizzano attualmente tali segmenti

Il descrittore dei processi di Unix si chiama **process control block** ed è organizzato in due parti:

- Process structure, residente sempre in memoria principale, mantiene informazioni sul processo a prescindere dal suo stato, ad esempio:
 - PID
 - Stato
 - Puntatori ai segmenti dati, stack e text-structure (la segmentazione utilizzata è a 3)
 - Informazioni di scheduling (es: priorità, tempo di cpu)
 - PID del processo padre
 - Informazioni di gestione dei segnali (es: quelli inviati ma non ancora gestiti)
 - Puntatore ad altre *process structure*
 - Puntatore a *user structure*
- User structure, è presente in memoria principale solo quando il processo è "residente", ossia è anch'esso si trova attualmente in memoria principale, ad esempio:
 - Contesto (copia dei registri della CPU)
 - Informazioni sulle risorse allocate (es: file aperti)
 - Informazioni di gestione dei segnali
 - Ambiente del processo (es: utente che lo esegue, gruppo che lo esegue, argc, argv, path, ...)

Le varie process structure sono organizzate in una tabella detta **process table**, e ogni processo può accedere alla sua utilizzando il PID come indice

In Unix non tutta l'immagine virtuale del processo è accessibile in modo utente (es: la process structure è protetta, o almeno alcune sue parti), ed inoltre non tutta l'immagine dei processi è swappabile (es: la process structure, come detto anche prima, non lo è)



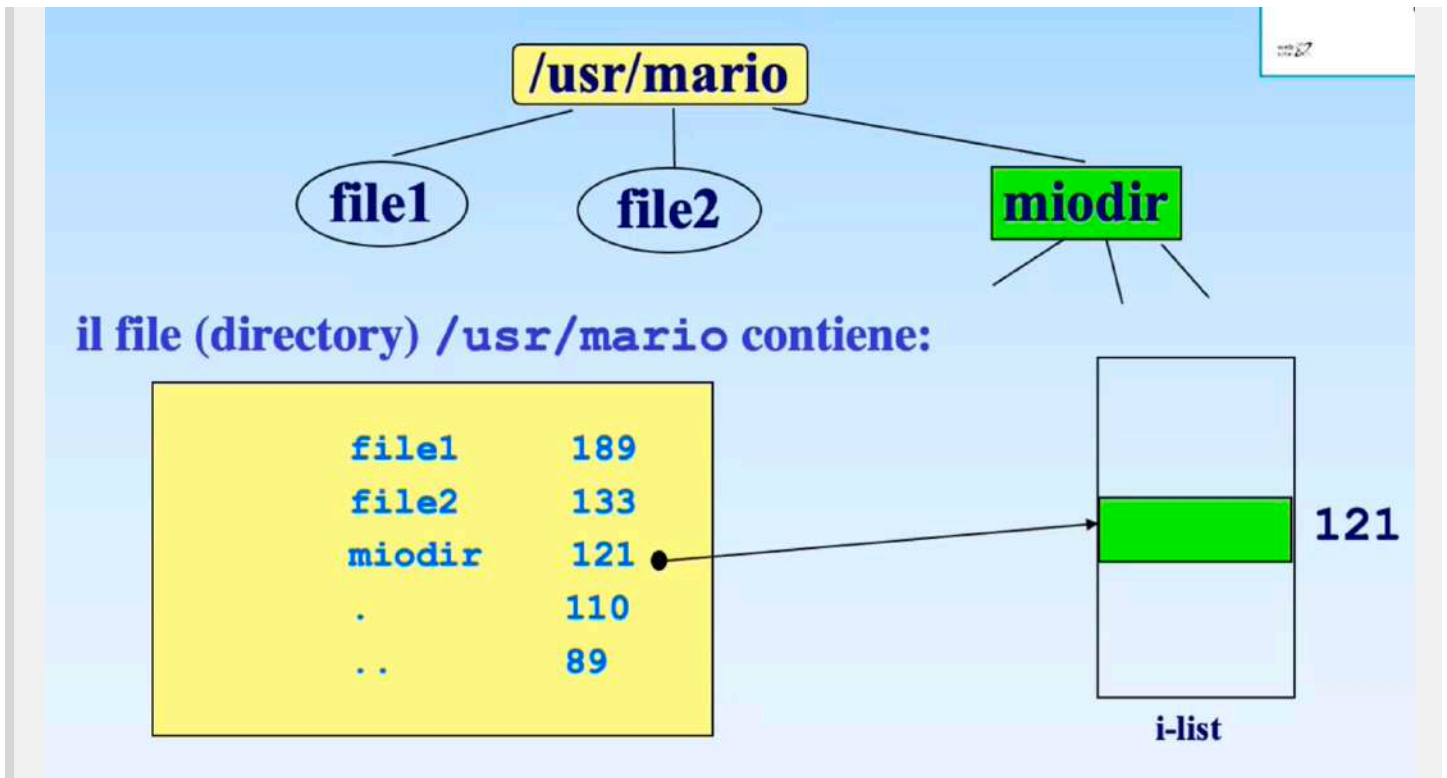
Ogni processo (anche thread!) ha il suo stack nel kernel

Per quanto riguarda lo scheduling Unix privilegia i processi interattivi, utilizzando **round robin su code multilivello**, le code sono definite in base a priorità dinamiche, che vengono ricalcolate ogni secondo: la priorità di un processo decresce (in Unix vuol dire maggiore importanza) al crescere del tempo di cpu utilizzato

File system

Per quanto riguarda il **file system** questo è organizzato come grafo diretto aciclico: lo stesso file può avere più path (nomi simbolici) ma un unico descrittore detto **i-node**, allocato all'indice **i-number** della **i-list**, in memoria secondaria

Le directory sono file contenenti coppie *nome* <-> *i-number*



Il descrittore di file *i-node* contiene i seguenti campi:

- Tipo: file ordinario, directory o file speciale (ad esempio i dispositivi)
- Proprietario e gruppo: user id e group id
- Dimensione: espressa in **record**, che **in Unix sono grandi 1 byte**
- Date di creazione e di modifica
- 12 bit di protezione
- Numero di link simbolici che riferiscono il file
- Vettore lungo da 13 a 15 indici di blocchi (del disco virtuale) tramite i quali viene memorizzato il file:

Dimensione del blocco: 512 byte=0,5 KB

indirizzi di 32 bit (4 byte)

→ 1 blocco contiene 128 indirizzi

Quindi:

- **10 blocchi** di dati sono accessibili *direttamente*
- **128 blocchi** di dati sono accessibili con *indirizzazione singola* (mediante il puntatore 11): $(128 \cdot 512) \text{ byte} = 65536 \text{ byte} = 64 \text{ KB}$
- **128*128 blocchi** di dati sono accessibili con *indirizzazione doppia* (mediante il puntatore 12): $128 \cdot 128 \cdot 512 \text{ byte} = 8 \text{ MB}$
- **128*128*128 blocchi** di dati sono accessibili con *indirizzazione tripla* (mediante il puntatore 13): $128 \cdot 128 \cdot 128 \cdot 512 \text{ byte} = 1 \text{ GB}$

→ la dimensione massima del file è dell'ordine del Gigabyte=1GB+8MB+64KB+5KB

L'allocazione fisica è dunque un indice multilivello (in realtà è anche ibrida: file piccoli sono allocati in modo contiguo)

In Unix il disco virtuale è diviso in quattro sezioni:

- Boot block: estensione delle ROM in memoria secondaria, utilizzata in fase di bootstrap
- Super block: definisce le quattro sezioni che stiamo elencando, mantiene inoltre un puntatore alla lista dei blocchi liberi nel *Data block* ed uno alla lista degli *i-number* liberi nella *i-list*
- i-list
- Data block

La dimensione dei blocchi è configurabile da 512 a 4096 byte (non può scendere in ogni caso sotto alla dimensione del settore del disco)

L'accesso ad un file è strettamente subordinato all'operazione di apertura, che prevede l'impiego delle seguenti tabelle:

- Tabella dei file aperti di processo: è puntata dalla user structure e contiene, per ogni file aperto dal processo, un puntatore ad un'entrata della tabella dei file aperti di sistema (detto *file descriptor*)

I primi tre file descriptor sono sempre *stdin*, *stdout* e *stderr*

- Tabella dei file aperti di sistema: presenta un'entrata per ogni singola operazione di apertura (di file non ancora chiusi) effettuata da qualsiasi processo, ognuna di queste entrate contiene un **I/O pointer** ed un puntatore alla tabella dei file attivi

Dato che in Unix l'accesso ai file è sequenziale, l'I/O pointer non è altro che l'ultimo record acceduto

- Tabella dei file attivi: contiene una copia degli *i-node* dei file aperti (e non ancora chiusi) da qualsiasi processo (senza duplicati)

