



**La Sapienza**

Università degli Studi di Roma

Dipartimento di Informatica e Sistemistica

# CALCOLATORI ELETTRONICI

## Sistemi operativi

**Emiliano Trevisani**

**[trevisani@dis.uniroma1.it](mailto:trevisani@dis.uniroma1.it)**

**A.A. 2007/2008**

## Overview

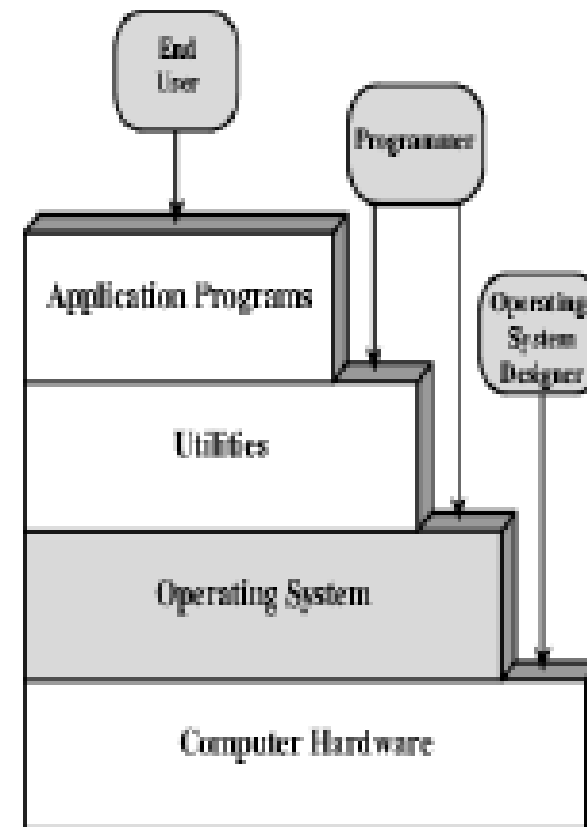


- ❑ Il Sistema Operativo (SO) è il **software** che **controlla** l'esecuzione dei programmi in un sistema di elaborazione **gestendo** le risorse del sistema stesso
- ❑ Obiettivi: un SO deve offrire un punto unico di gestione (efficiente) delle risorse di sistema nascondendo all'utente i dettagli relativi al controllo dell'HW; in altri termini, deve consentire alle applicazioni di utilizzare in modo "conveniente" l'hardware
- ❑ Risorse di sistema: processore (CPU), memoria, periferiche,...
- ❑ In genere l'utente finale interagisce con un sistema di elaborazione a livello "applicativo"
  - l'esecuzione di applicazioni prevede l'utilizzo di un insieme di **processi**
    - processo  $\neq$  programma; un programma è un'entità statica; un processo è un **programma in esecuzione**, ossia un'entità dinamica con uno stato]
    - Processo = <programma eseguibile, stato di esecuzione>
      - stato di esecuzione: contenuto dello spazio di memoria allocata al processo + contenuto dei registri della CPU (in particolare PC)
    - Dato un programma (p.e., Word), in un certo istante ci possono essere più processi che lo eseguono in modo indipendente
  - ciascun processo richiede in generale risorse di sistema per realizzare il suo "task"
  - **il processo è l'unità di lavoro dei moderni SO**

## Overview



- Il Sistema Operativo controlla l'esecuzione dei processi arbitrando le loro richieste di risorse
- Le operazioni di gestione dell'HW e controllo dei processi risultano trasparenti all'utente



## Overview



### □ Servizi del SO

- Creazione di programmi [utilities per lo sviluppo di programmi]
- Esecuzione di programmi
  - il SO avvia e controlla l'esecuzione dei processi necessari all'esecuzione di un programma
- Accesso ai dispositivi di I/O
  - il SO nasconde all'utente i dettagli dell'interazione con l'HW della particolare periferica; quest'ultimo esprime solo la necessità di lettura o scrittura
- Accesso controllato al file system [I/O su file]
- Accesso al sistema [es. login]
- Rilevazione e gestione degli errori [es. overflow, malfunzionamento HW]
- Fornisce meccanismi per garantire
  - l'isolamento dei programmi in esecuzione
  - quando necessario, una corretta cooperazione tra programmi in esecuzione

## Overview



- ❑ Un SO può essere quindi definito come un particolare programma in esecuzione che agisce da **resource manager**
- ❑ Con riferimento alle risorse di sistema, gli aspetti più importanti che un SO affronta:
  - CPU: **se e per quanto tempo** la CPU deve essere assegnata a un dato processo
  - memoria: **quanta memoria e in che modo** essa deve essere allocata per un dato processo
  - periferiche: **arbitra** l'accesso alle periferiche interne ed esterne
- Poiché il SO è un programma in esecuzione e come tale utilizza esso stesso la CPU, deve “rilasciare” quest'ultima per consentire l'esecuzione di altri programmi
- Unità funzionali principali di un SO:
  - Gestione dei processi
  - Gestione della memoria
  - Gestione delle periferiche / File system / Gestione delle comunicazioni di rete
  - Gestione della sicurezza
  - L'insieme di questi moduli costituisce il **nucleo [kernel]** del sistema operativo
  - L'interfaccia utente del SO può essere di tipo grafico o a console

## Overview



### □ Definizioni:

- **Virtualizzazione delle risorse:** il SO realizza un ambiente a risorse virtuali [CPU, memoria, periferiche] per l'esecuzione dei processi; ciascun processo:
  - ignora i dettagli relativi ai servizi forniti dalle risorse
  - ignora l'eventuale presenza di altri programmi in esecuzione [es. "vede" le risorse come dedicate]
  - "vede" uno spazio di memoria autonomo
  - "vede" le periferiche attraverso i servizi del SO
  - In particolare:
    - **CPU virtualizzata** per offrire supporto alla multiprogrammazione
    - **Memoria virtualizzata** per offrire un ambiente isolato per l'esecuzione di processi
    - **I/O virtualizzato** per offrire il servizio di accesso a periferiche eterogenee attraverso un'unica interfaccia software
- **Ciclo CPU – I/O Burst** : l'esecuzione di un processo consiste di un ciclo di esecuzione nella CPU e attesa per operazioni di I/O.

# Multiprogrammazione



## □ Definizioni:

- **Multiprogrammazione:** prevede l'esecuzione simultanea di più processi su un sistema
  - Motivazioni: aumento di prestazioni e risparmio di risorse; il parallelismo di calcolo insito in alcuni problemi può essere sfruttato
  - Tipologie:
    - **batch:** non è prevista l'interattività con l'utente. L'obiettivo di questo tipo di multiprogrammazione è la massimizzazione delle prestazioni.
    - **time-sharing:** la definizione classica si riferisce a più utenti che utilizzano contemporaneamente ed interattivamente lo stesso sistema, massimizzando la praticità d'uso a scapito delle prestazioni.
      - La CPU del sistema viene utilizzata per servire le richieste dei singoli utenti, commutando rapidamente da uno all'altro (context switch) dando l'impressione che ognuno abbia a disposizione completa il sistema
      - si estende al caso di sistemi mono-utente riferendosi ad un approccio interattivo a “divisione di tempo” per utilizzo della CPU da parte dei diversi processi

# Multiprogrammazione



## □ Definizioni:

- **Grado di multiprogrammazione:** numero di processi contemporaneamente presenti nel sistema
- **Processi I/O-bound.** Processi che fanno un uso intensivo dei dispositivi periferici, e la cui esecuzione è caratterizzata da frequenti ma brevi periodi (bursts) di utilizzo della CPU
- **Processi CPU-bound.** Sono processi che spendono più tempo in elaborazioni che non nello svolgimento di operazioni di I/O; la loro esecuzione è caratterizzata da pochi e prolungati periodi di utilizzo della CPU.
- **Classificazione Sistemi Operativi:**
  - **Multi-user:** supporta più utenti contemporaneamente attivi
  - **Multi-processing:** supporta l'esecuzione di un programma utilizzando più CPU simultaneamente
  - **Multi-tasking:** supporta l'esecuzione simultanea di più programmi
  - **Multi-threading:** supporta l'esecuzione simultanea di più componenti di uno stesso programma [1 processo = insieme di thread]
  - **Real-time:** offre un ambiente per l'esecuzione di applicazioni con stringenti requisiti temporali

## Multiprogrammazione



### □ Un modello per la multiprogrammazione:

- Obiettivo: valutare come la multiprogrammazione incrementa l'efficienza d'utilizzo della CPU
- Supponiamo attivi nel sistema  $n$  processi indipendenti [grado di multiprogrammazione =  $n$ ]
- In generale un processo spende una frazione del suo tempo di esecuzione in attesa del completamento di operazioni di I/O; indicheremo con  $p$  la probabilità che, istante per istante, il processo sia bloccato in attesa di I/O
- Poiché abbiamo supposto gli  $n$  processi indipendenti, la probabilità che la CPU resti inutilizzata è esattamente quella per cui tutti i processi sono impegnati in operazioni di I/O, ossia  $p^n$ ; la probabilità che la CPU risulti occupata è invece pari a  $1 - p^n$

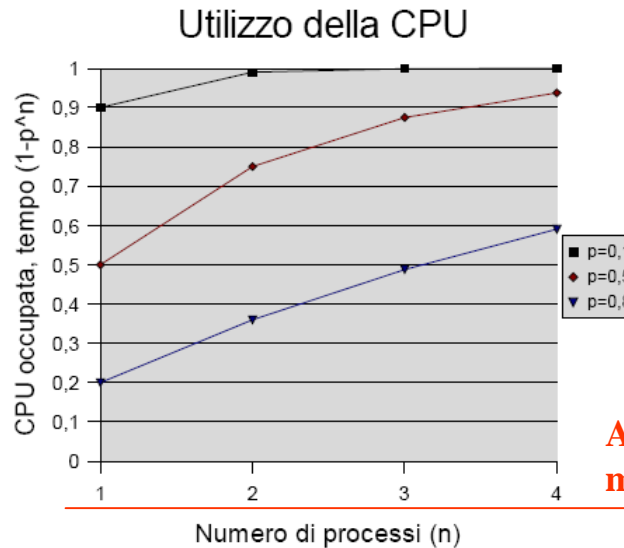
# Multiprogrammazione



□ Un modello per la multiprogrammazione:

- Il grafico seguente mostra l'andamento della probabilità di utilizzo della CPU [ $\cong$  frazione del tempo per cui la CPU è utilizzata ossia il grado di utilizzo del processore] per diversi valori dei parametri  $n$  e  $p$ .
- In particolare, esso mette in luce il diverso grado di utilizzo della CPU per le due diverse categorie di processi I/O bound e CPU bound

**Aumenta il grado di utilizzo della CPU**



**Aumenta il grado di multiprogrammazione**

CPU time ( $1-p^n$ )	0,9	0,990	0,999	1,000	CPU bound
	0,5	0,750	0,875	0,938	
	0,2	0,360	0,488	0,590	I/O bound
# processi	1	2	3	4	

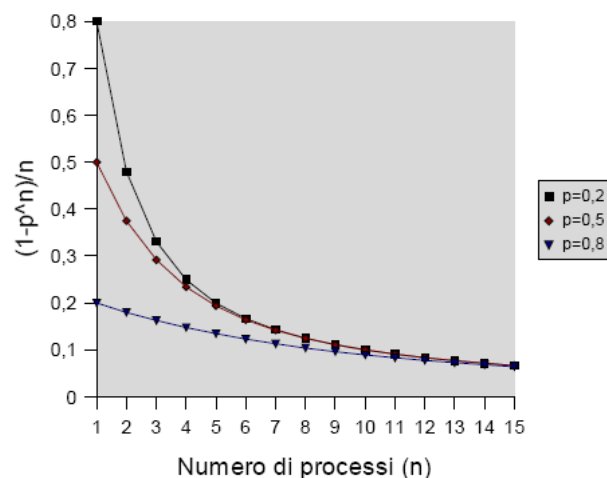
# Multiprogrammazione



## □ Un modello per la multiprogrammazione:

- Il modello illustrato è un modello ideale; i processi in genere non sono tra loro indipendenti [competono per l'utilizzo di risorse]
- Anche con questo modello semplificato si può cogliere il vantaggio della multiprogrammazione: al crescere del grado di multiprogrammazione cresce l'utilizzo del processore
  - vedremo che questo è vero finché c'è memoria sufficiente per tutti i processi
- Osservazione: al crescere del grado di multiprogrammazione  $n$  la frazione di tempo di CPU dedicata a ciascun processo diminuisce; infatti, essa è pari a  $\frac{(1-p^n)}{n}$

Uso della CPU per singolo processo



## Multiprogrammazione



□ Esempio: Il seguente esempio mostra l'incremento di prestazioni di un ambiente multiprogrammato rispetto ad uno uniprogrammato

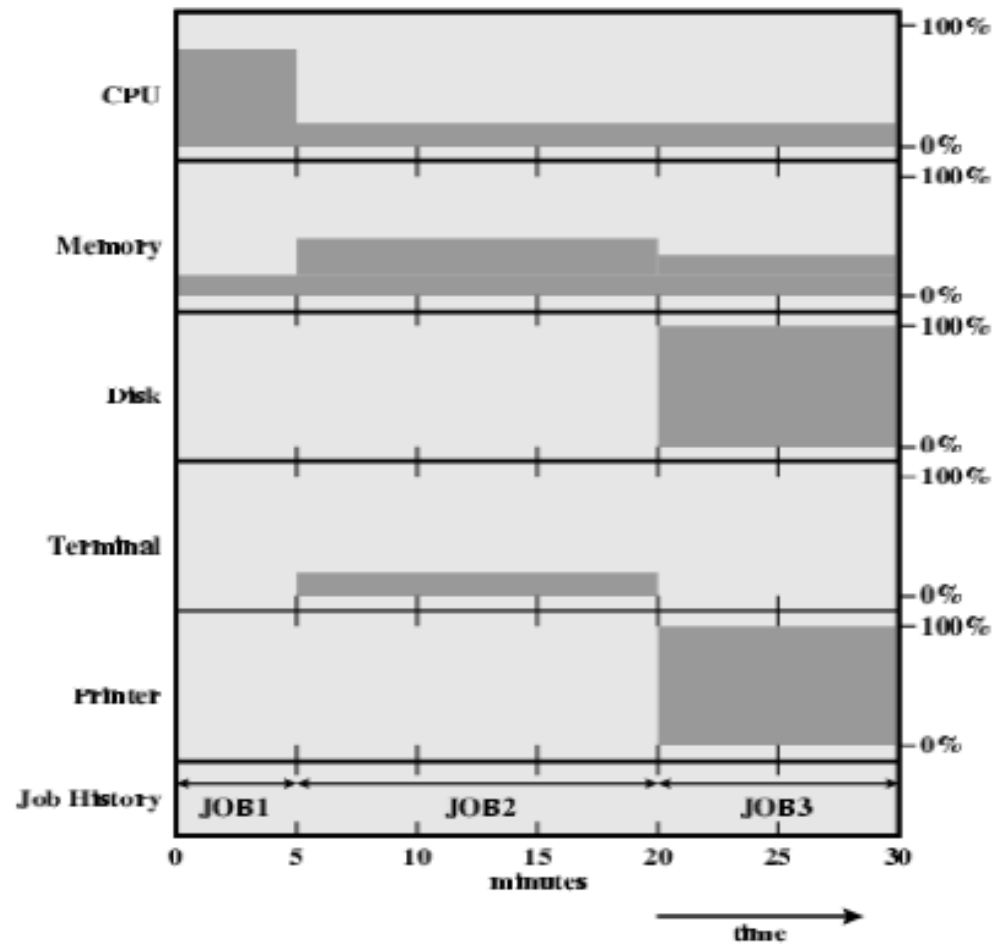
- Scenario: 3 Processi [nel seguito JOB=Processo] con diverse esigenze in termini di memoria, CPU, accesso a periferiche
  - JOB1 richiede un elevato uso della CPU
  - JOB2 richiede un elevato uso del terminale
  - JOB3 richiede un elevato uso di disco e stampante
- Confronteremo l'esecuzione "sequenziale" dei 3 processi [ambiente uniprogrammato] con quella "parallela" [ambiente multiprogrammato]; vedremo che la seconda incrementa l'utilizzo delle risorse richieste

	JOB1	JOB2	JOB3
Type of job	Heavy compute	Heavy I/O	Heavy I/O
Duration	5 min	15 min	10 min
Memory required	50 K	100 K	80 K
Need disk?	No	No	Yes
Need terminal?	No	Yes	No
Need printer?	No	No	Yes

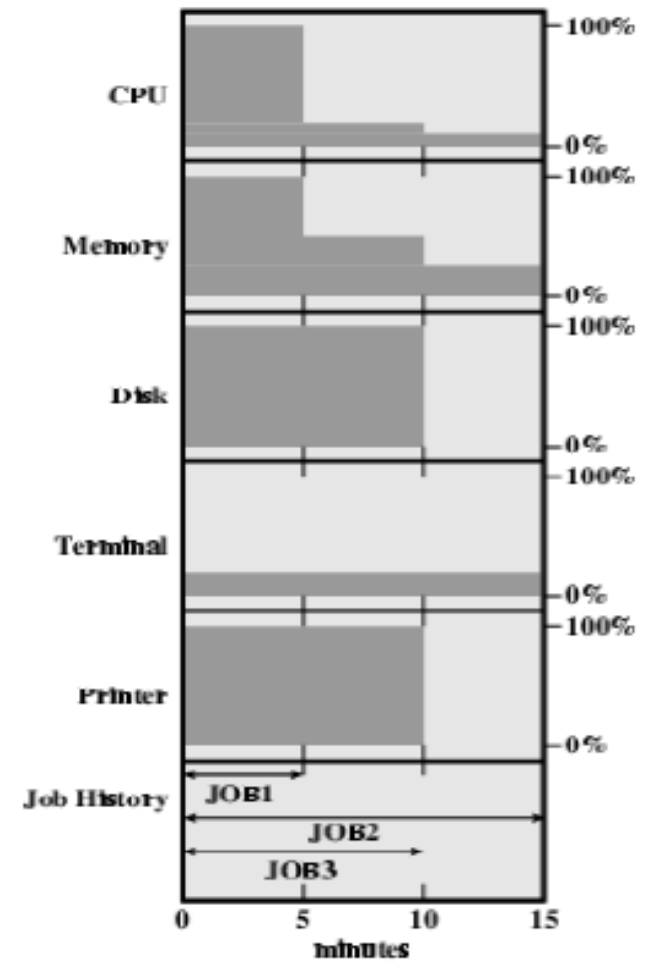
# Multiprogrammazione



□ Esempio:



(a) Unprogramming



(b) Multiprogramming

## Scheduling dei processi



- ❑ Un SO multi-tasking deve gestire l'esecuzione simultanea di una molteplicità di processi [più programmi in esecuzione]
- ❑ Lo **scheduler** è l'unità funzionale del SO che si occupa della gestione dei processi; in particolare:
  - obiettivo: massimizzare l'utilizzo della CPU
    - arbitraggio dei processi
    - assegnazione della CPU ai processi
      - in un dato ordine
      - per un certo intervallo di tempo
- ❑ In genere, in relazione alla frequenza di attivazione, esistono almeno tre livelli di scheduling in SO:
  - Lungo termine
  - Medio termine
  - Breve termine

## Scheduling dei processi



### □ Scheduler a lungo termine [admission scheduler]

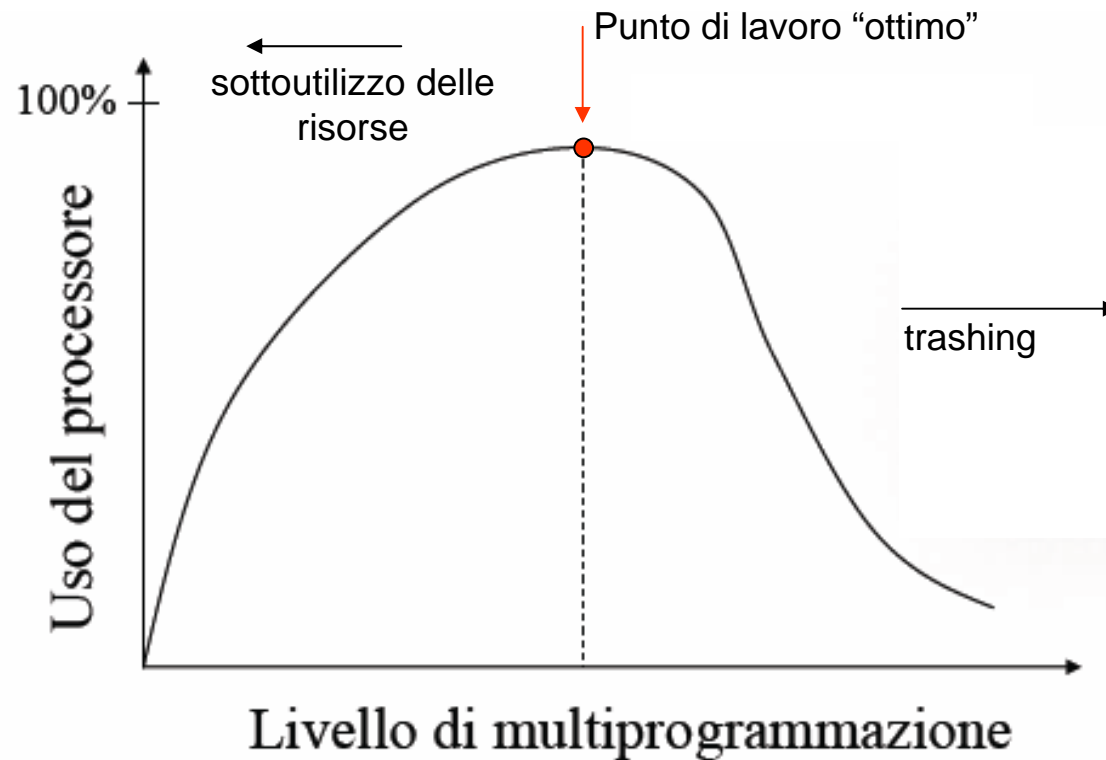
- Lo scheduling di lungo termine realizza uno “scheduling di alto livello”: decide se un’applicazione (e quindi i processi corrispondenti ) possa essere o meno avviata
- Seleziona da una coda di “applicazioni ready to start” quale avviare trasformandola in un processo [o in più processi]
  - In particolare, decide anche se inserire o meno un applicazione nella coda
- Questo scheduler decide il grado di multiprogrammazione del sistema, ossia quanti processi saranno al più contemporaneamente attivi in ogni istante [competeranno per l’accesso a risorse, es: CPU]
  - Su questo parametro può agire anche lo scheduler a medio termine [tuning del grado di multiprogrammazione]
  - Tende ad ottenere un certo rapporto tra processi I/O-bound e processi CPU-bound
- Nei sistemi “time sharing”, questo scheduler ammette un certo numero di utenti fino a un definito livello di “saturazione”

## Scheduling dei processi



### □ Scheduler a lungo termine [admission scheduler]

- Obiettivo: regolare il grado di multiprogrammazione per portare il sistema sul punto di lavoro “ottimo” [massimo utilizzo della CPU]





### □ Scheduler a medio termine

- Lo scheduling di medio termine è legato ad attività di gestione della memoria relativamente ai processi correntemente attivi (“swapping” in|out)
- Introduce un ulteriore livello di tuning relativamente al grado di multiprogrammazione: decide infatti quali, fra i processi ammessi dallo scheduler di lungo termine:
  - possono restare in memoria (restare attivi) continuando a competere per l’accesso a risorse
  - possono essere temporaneamente rimossi dalla memoria (swap out) per ridurre il grado multiprogrammazione [può essere necessario in una condizione di “trashing” come vedremo]
  - possono essere riportati in memoria (swap in) a seguito di un precedente swap out per aumentare il grado multiprogrammazione

# Scheduling dei processi



## □ Scheduler a medio termine

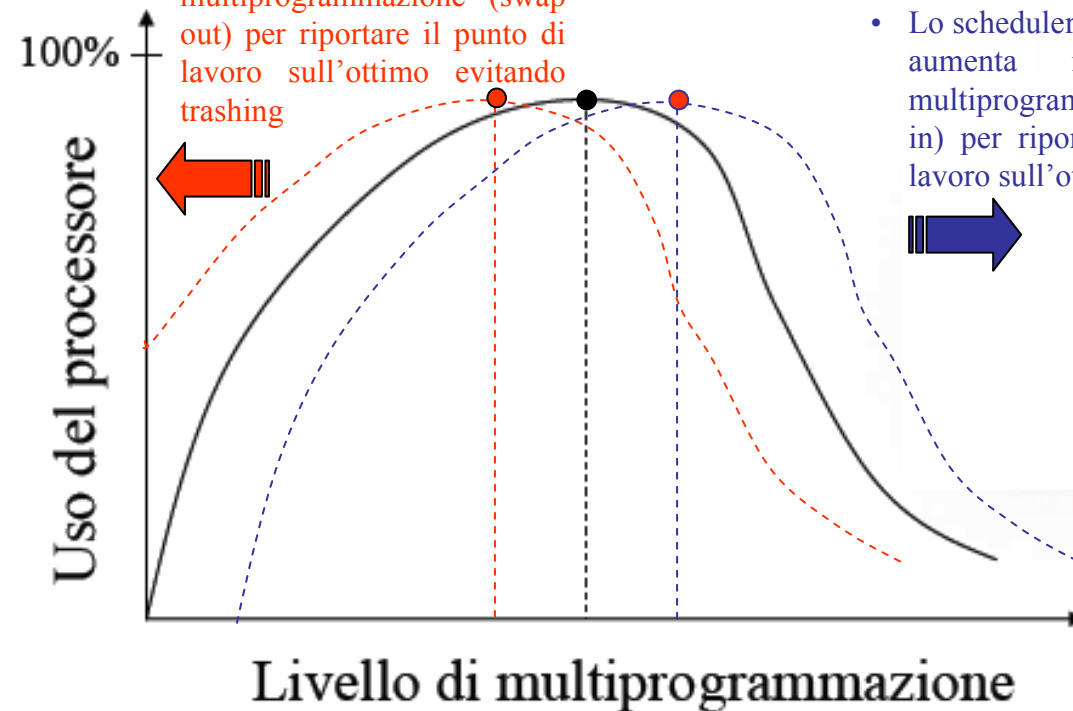
- Obiettivo: **mantenere** il sistema sul punto di lavoro “ottimo” agendo sul grado di multiprogrammazione a seguito di cambiamenti nell’esigenze di memoria dei processi attivi

- Aumentano le esigenze di memoria medie dei processi

- Lo scheduler a medio termine diminuisce il grado di multiprogrammazione (swap out) per riportare il punto di lavoro sull’ottimo evitando trashing

- Diminuiscono le esigenze di memoria medie dei processi

- Lo scheduler a medio termine aumenta il grado di multiprogrammazione (swap in) per riportare il punto di lavoro sull’ottimo



## Scheduling dei processi



### □ Scheduler a breve termine [dispatcher]

- Realizza uno “scheduling di basso livello”: decide quali fra i processi attivi in attesa di CPU eseguire ed in quale sequenza
  - La sequenza di esecuzione viene decisa in accordo a una data strategia (es. FIFO, SJF, Round Robin,..)
    - La strategia mira ad ottimizzare uno o più parametri; alcuni esempi:
      - priorità tra processi
      - grado di utilizzo della CPU
      - tempo di turnaround – tempo totale per eseguire un processo
      - tempo di waiting – tempo totale di attesa sulla ready queue
      - tempo di risposta dei processi: tempo fra richiesta e prima risposta
      - overall throughput (numero di processi completati per unità di tempo)
      - fairness (suddivisione equilibrata delle risorse di calcolo tra i processi; il suo obiettivo è evitare la condizione di *starvation*)
  - Lo scheduler di basso livello assegna uno stato a ciascun processo
  - Gli stati di un processo evolvono e vengono via via modificati fino all’eventuale conclusione o interruzione

## Scheduling dei processi



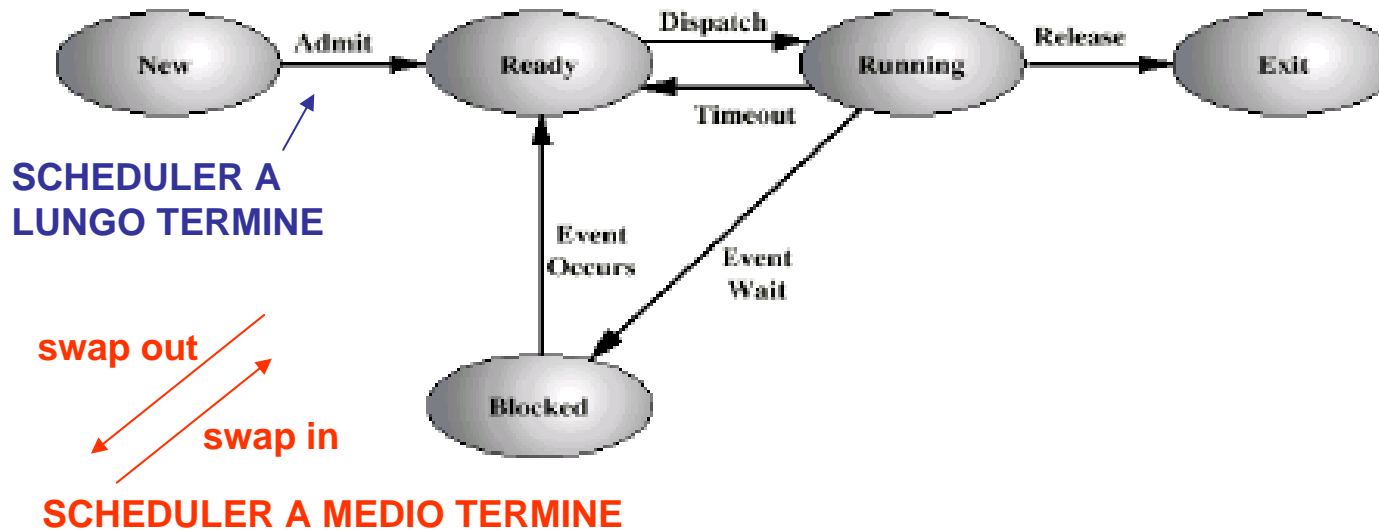
### □ Scheduler a breve termine [dispatcher]

- Criteri di ottimizzazione:
  - Massimizzare l'utilizzo della CPU
  - Massimizzare il throughput
  - Minimizzare il tempo di turnaround
  - Minimizzare il tempo di waiting
  - Minimizzare il tempo di risposta
- Generalmente si tende ad ottimizzare i valori medi.

# Scheduling dei processi



## □ Scheduler a breve termine [dispatcher]



- Nuovo (New): un programma è convertito in processo; ancora non compete per l'accesso a risorse
- Pronto (Ready): il processo è pronto per l'esecuzione e attende l'accesso al processore
- Esecuzione (Running): il processo viene eseguito dal processore
- Sospeso (Waiting/Blocked): il processo è sospeso perché ha richiesto una risorsa (es. l'accesso ad una periferica)
- Terminato (Halted/Exit): il processo è concluso [regolarmente o terminato forzatamente dal SO]

## Scheduling dei processi



### □ Scheduler a breve termine [dispatcher]

- Il meccanismo che consente la commutazione della CPU da un processo ad un altro è definito **context switch**.
- La struttura dati di supporto al context switch e associata ad ogni processo è definita **process control block (PCB)** che contiene le seguenti informazioni:
  - puntatore all'area di memoria contenete il processo (istruzioni + dati)
  - stato del processo (ready, running, waiting, etc.)
  - process identifier (PID)
  - contenuto del PC [registro program counter]
  - contenuto dello status register [registro di stato della CPU]
  - contenuto dei registri generici
  - lista di file aperti
  - dati per la protezione e la gestione dei diritti



### □ Scheduler a breve termine [dispatcher]

- Strategie di dispatching
  - **First In First Out (FIFO)**
    - I processi vengono serviti nell'ordine di inserimento nella coda di processi "ready"
    - Implementazione semplice: coda senza priorità
    - I processi non vengono prelazionati: un processo resta nello stato running fino a quando termina o va in attesa di eventi.
    - Non ottimizza alcun parametro in particolare
    - Poco adatto per processi interattivi

## Scheduling dei processi



### □ Scheduler a breve termine [dispatcher]

- Strategie di dispatching
  - **First In First Out (FIFO)**
    - Esempio 1

Primo arrivato, primo servito (gestito con coda FIFO).

<u>Processo</u>	<u>Burst Time</u>
$P_1$	24
$P_2$	3
$P_3$	3

Supponiamo che i processi arrivino nell'ordine:  $P_1$ ,  $P_2$ ,  $P_3$   
Lo schema di Gantt è:



Tempo di waiting per:  $P_1 = 0$ ;  $P_2 = 24$ ;  $P_3 = 27$   
Tempo di waiting medio:  $(0 + 24 + 27)/3 = 17$

## Scheduling dei processi



### □ Scheduler a breve termine [dispatcher]

- Strategie di dispatching
  - **First In First Out (FIFO)**
    - Esempio 2

Supponiamo che i processi arrivino nell'ordine

$P_2, P_3, P_1$ .

Lo schema di Gantt è:



Tempo di waiting per  $P_1 = 6; P_2 = 0; P_3 = 3$

Tempo di waiting medio:  $(6 + 0 + 3)/3 = 3$

Molto meglio che nel caso precedente.

*Effetto convoglio:* i processi "brevi" attendono i processi "lunghi".

## Scheduling dei processi



### □ Scheduler a breve termine [dispatcher]

- Strategie di dispatching
  - **Shortest Job First (SJF)**
    - I processi in coda vengono serviti a partire da quello il cui tempo di esecuzione **stimato** è minimo
    - Associa ad ogni processo la lunghezza del prossimo **CPU burst**. Usa questi tempi per schedulare il processo con la lunghezza minima.
    - Due schemi:
      - **nonpreemptive** – il processo assegnato non può essere sospeso prima di completare il suo CPU burst.
      - **preemptive** – se arriva un nuovo processo con un CPU burst più breve del tempo rimanente per il processo corrente, viene servito. Questo schema è conosciuto come Shortest-Remaining-Time-First (SRTF).
    - SJF è ottimale (rispetto al waiting time): offre il minimo tempo medio di attesa per un insieme di processi.

## Scheduling dei processi

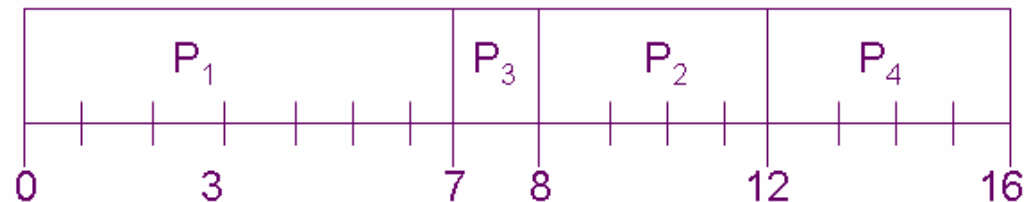


### □ Scheduler a breve termine [dispatcher]

- Strategie di dispatching
  - **Shortest Job First (SJF)**
    - Esempio: nonpreemptive SJF

<u>Processo</u>	<u>Tempo Arrivo</u>	<u>Tempo di Burst</u>
$P_1$	0.0	7
$P_2$	2.0	4
$P_3$	4.0	1
$P_4$	5.0	4

SJF (non-preemptive)



$$\text{Tempo di attesa medio} = (0 + 6 + 3 + 7)/4 = 4$$

## Scheduling dei processi



### □ Scheduler a breve termine [dispatcher]

- Strategie di dispatching
  - **Shortest Job First (SJF)**
    - Esempio: preemptive SJF

Processo	Tempo di Arrivo	Tempo di Burst
$P_1$	0.0	7
$P_2$	2.0	4
$P_3$	4.0	1
$P_4$	5.0	4

SJF (preemptive)



$$\text{Tempo di attesa medio} = (9 + 1 + 0 + 2)/4 = 3$$

## Scheduling dei processi



### □ Scheduler a breve termine [dispatcher]

- Strategie di dispatching
  - **Scheduling a priorità**
    - Una priorità (numero intero) è assegnata ad ogni processo.
    - La CPU è assegnata al processo con la priorità più alta in accordo ad uno schema
      - **preemptive**
      - **nonpreemptive**
    - SJF può essere visto come uno scheduling a priorità stabilita dal valore del tempo del prossimo CPU burst.
    - Problema: **Starvation**: i processi a più bassa priorità potrebbero non essere mai eseguiti
      - Possibile soluzione: **Aging**: al trascorrere del tempo di attesa si incrementa la priorità di un processo che attende.

## Scheduling dei processi



### □ Scheduler a breve termine [dispatcher]

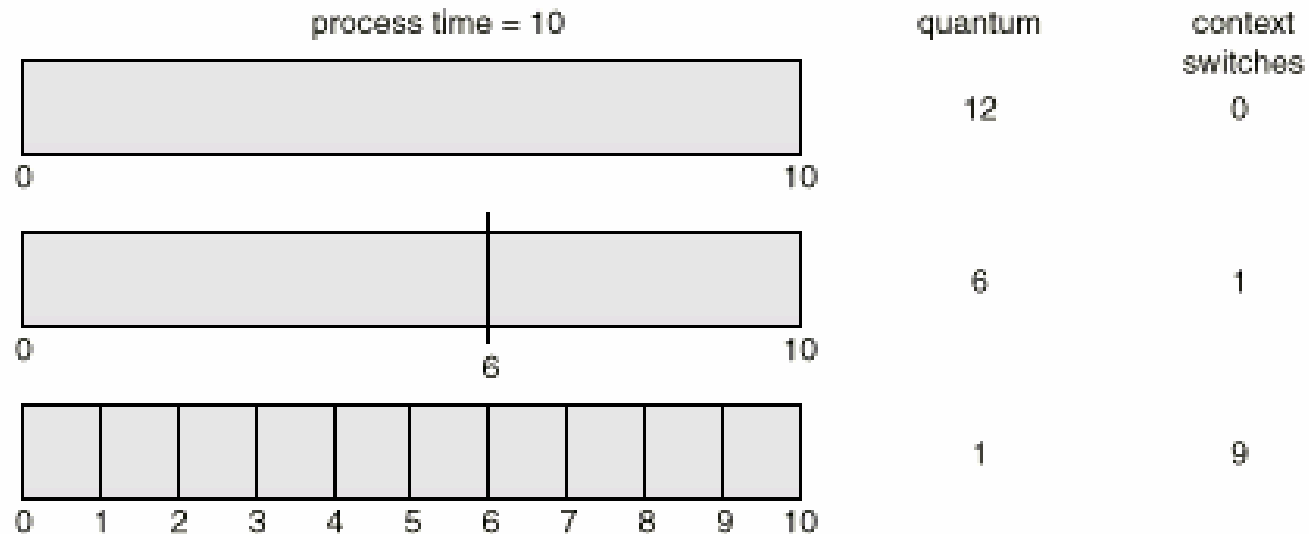
- Strategie di dispatching
  - **Scheduling Round Robin [RR]**
    - Ogni processo è assegnato alla CPU per un intervallo temporale fissato [**quanto di tempo** o **time slice**; es: 10-100 ms]
    - Esaurito il quanto di tempo al processo viene sottratta la CPU e questo viene inserito nella ready queue.
    - Se ci sono N processi nella ready queue e il quanto di tempo è Q, ogni processo ottiene  $1/N$  del tempo della CPU a blocchi di lunghezza Q. Nessun processo attende più di  $(N-1)Q$  unità di tempo.
  - Prestazioni
    - Q elevato: RR → FIFO
    - Q ridotto: Q deve essere molto più grande del tempo di **context switch**, altrimenti il costo è troppo alto.

# Scheduling dei processi



## □ Scheduler a breve termine [dispatcher]

- Strategie di dispatching
  - **Scheduling Round Robin [RR]**
    - Relazione tra quanto di tempo e tempo di context switch: il quanto di tempo deve essere molto più grande del tempo di context switch, altrimenti il costo è troppo alto.



## Scheduling dei processi

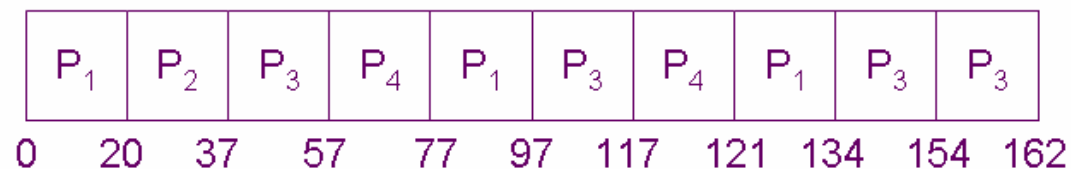


### □ Scheduler a breve termine [dispatcher]

- Strategie di dispatching
  - **Scheduling Round Robin [RR]**
    - Esempio: RR con  $Q=20$

<u>Processi</u>	<u>tempo di burst</u>
$P_1$	53
$P_2$	17
$P_3$	68
$P_4$	24

Gantt:



Tempo di *turnaround* maggiore di SJF, ma migliore *tempo di risposta*.

## Memoria virtuale



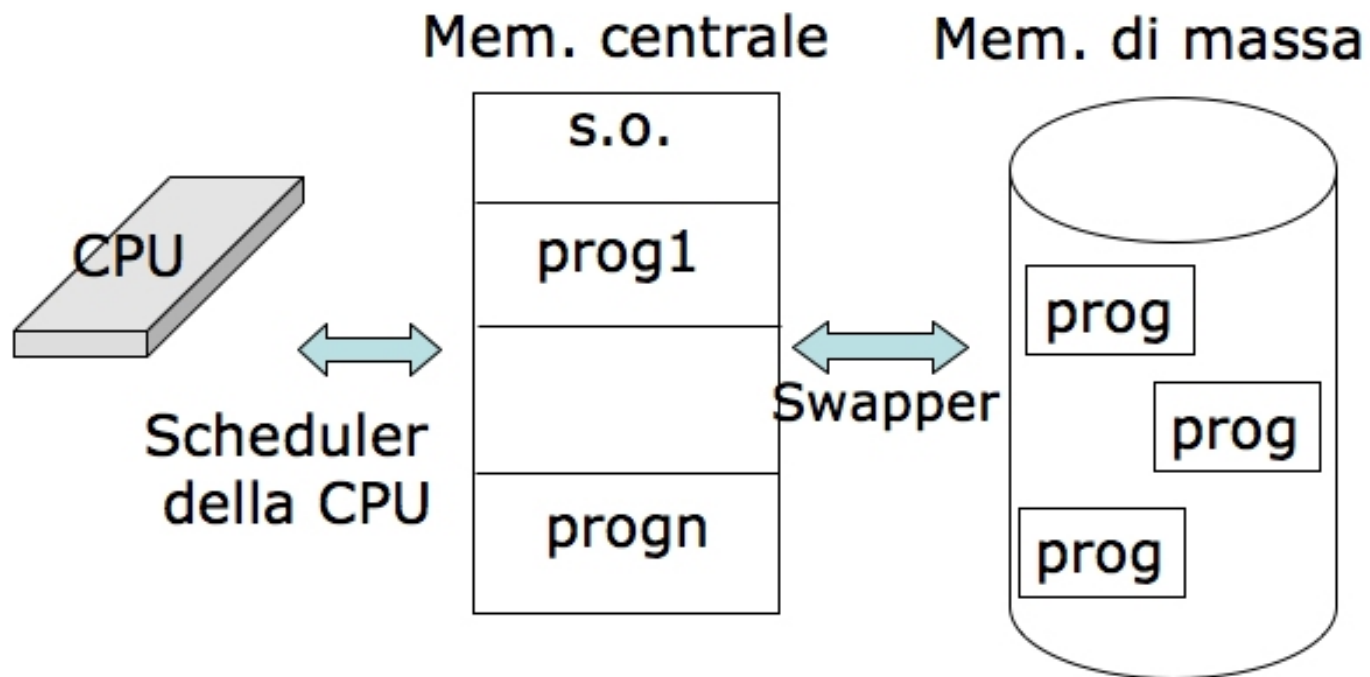
### □ Problemi:

- Memoria fisica **limitata e condivisa tra SO e processi attivi**.
  - Come eseguire un processo che richiede uno spazio di memoria fisica superiore a quello disponibile?
- Durante la fase di sviluppo del software in generale non è nota a priori l'area di memoria nella quale sarà eseguito: è necessario utilizzare indirizzi **simbolici** ad alto livello e **rilocabili** [virtuali] a livello di programma compilato; a tempo di esecuzione, gli indirizzi virtuali devono essere mappati in indirizzi di memoria fisica

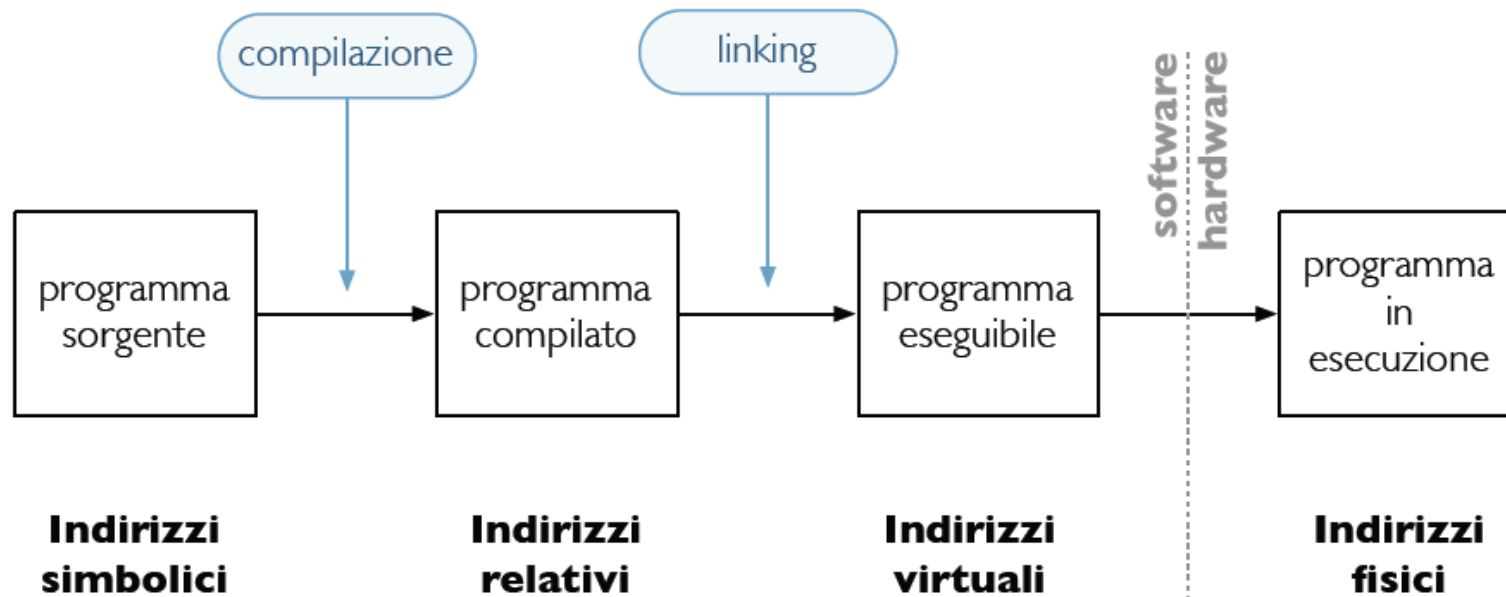
### □ Approccio a **memoria virtuale**: un processo può essere eseguito anche se il suo spazio di indirizzamento è solo **parzialmente** presente in memoria.

- Il meccanismo di memoria virtuale consente a più processi di risiedere in memoria [grado di multiprogrammazione....]
- Il meccanismo di memoria virtuale può essere inquadrato secondo il modello di gerarchie di memoria:
  - è giustificato dalla località temporale e spaziale degli accessi in memoria

# Memoria virtuale

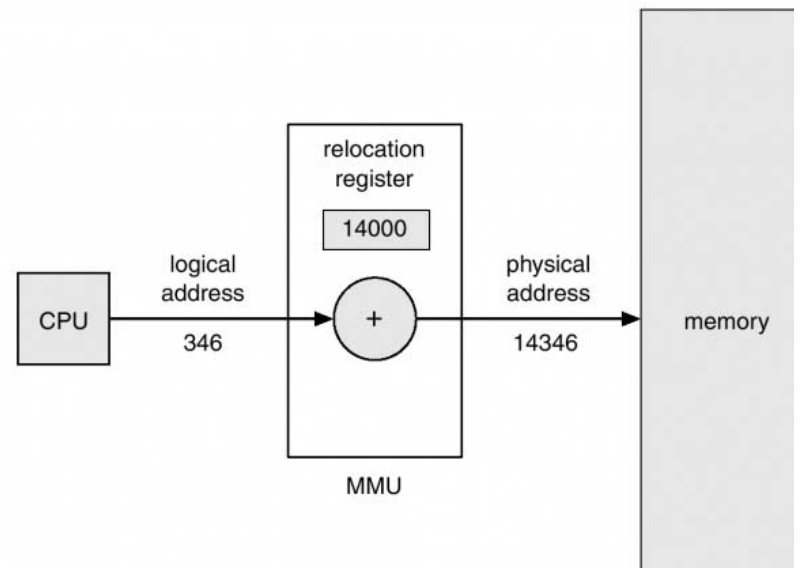


# Memoria virtuale



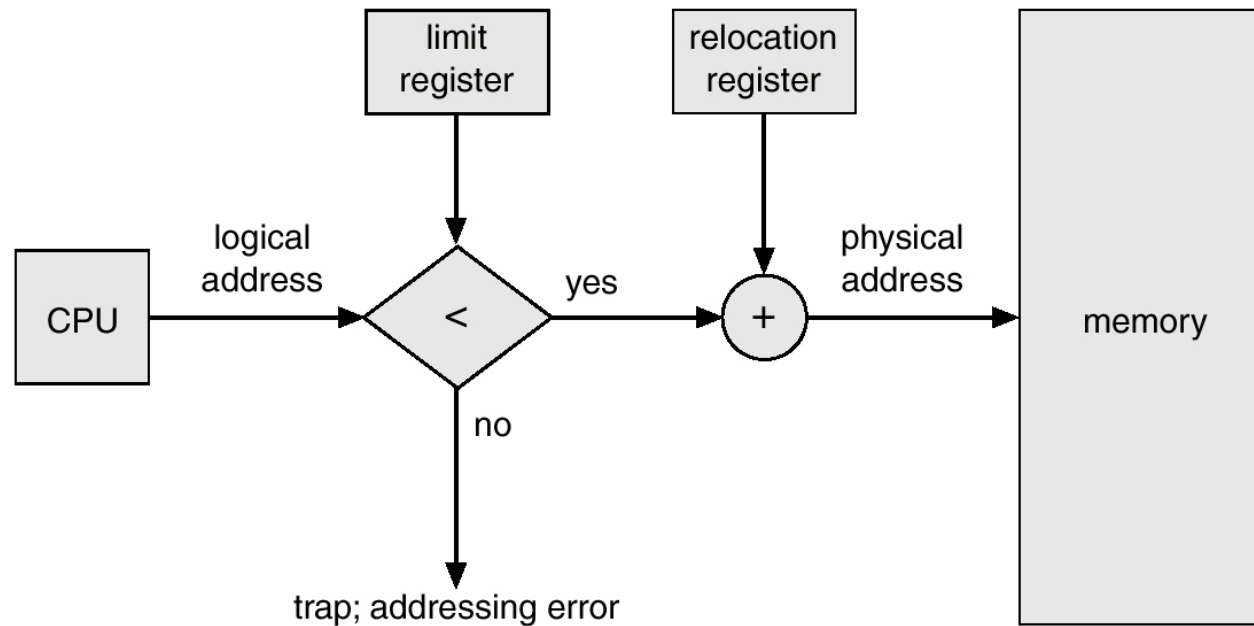
### □ Traduzione di indirizzi virtuali in indirizzi fisici

- Un processo effettua accessi in memoria in lettura / scrittura specificando indirizzi **virtuali** (o **logici**)
- L'indirizzo logico viene mappato nel corrispondente indirizzo fisico da HW dedicato: **Memory Management Unit** [supporto HW alle funzionalità del SO]
  - Registro base [registro di rilocazione]



## □ Traduzione di indirizzi virtuali in indirizzi fisici

- Registro base [registro di rilocazione] + Registro limite [meccanismo di protezione]





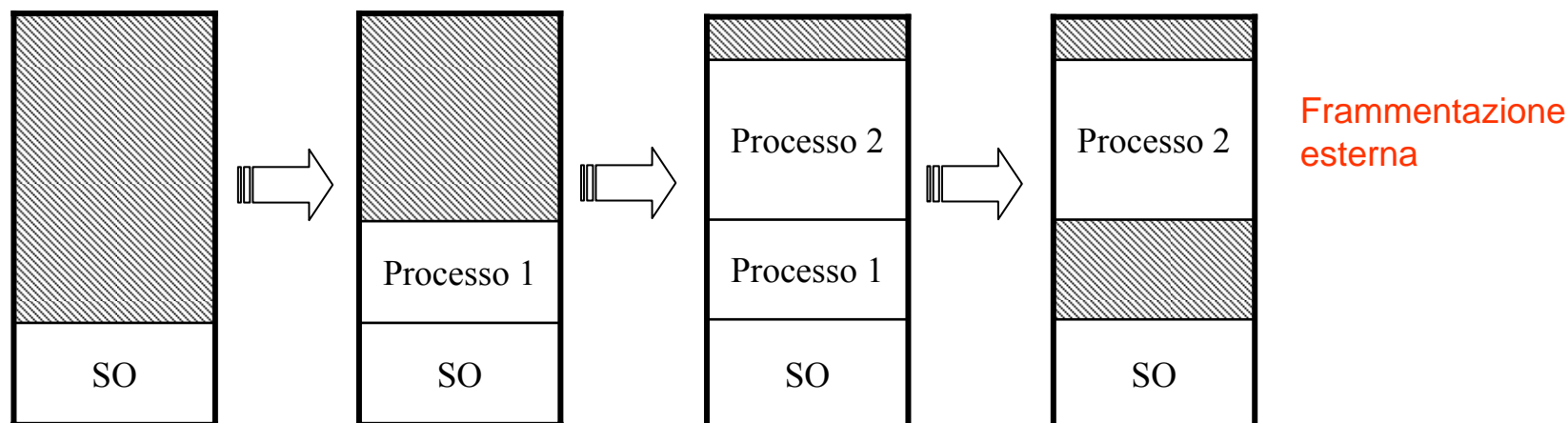
## □ Partizionamento della memoria

- Partizionamento statico
  - Memoria suddivisa in frammenti (partizioni) di dimensione **fissa**.
  - Ad un processo può essere assegnata una singola partizione
    - processi che necessitano una partizione più grande non possono essere eseguiti.
    - processi per i quali sarebbe sufficiente una partizione più piccola lasciano memoria inutilizzata generando **frammentazione interna**.



## □ Partizionamento della memoria

- Partizionamento dinamico
  - Memoria suddivisa in frammenti (partizioni) di **dimensione e posizione variabili** sulla base di:
    - esigenze dei processi: ad un nuovo processo viene allocato un frammento di dimensioni pari alla memoria richiesta
    - memoria disponibile: a seguito di continue allocazioni / deallocazioni di frammenti di dimensione variabile può generare **frammentazione esterna**.





### □ Partizionamento della memoria

- Partizionamento dinamico
  - A fronte della richiesta di memoria di un processo, il SO può disporre in generale di diverse aree contigue di memoria disponibile e di dimensione sufficiente; quale fra queste allocare al processo?  
Esempi di strategie:
    - **First Fit**: viene selezionato il primo spazio di memoria disponibile di dimensione compatibile con la richiesta
    - **Best Fit**: viene selezionato lo spazio di memoria disponibile che meglio approssima [naturalmente in eccesso] la richiesta
      - Frammentazione esterna caratterizzata da frammenti di dimensioni ridotte
    - **Worst Fit**: viene selezionato lo spazio di memoria disponibile più grande fra quelli di dimensione compatibile con la richiesta
      - Frammentazione esterna caratterizzata da frammenti di dimensioni più elevate

# Memoria virtuale



## □ Partizionamento della memoria

### ▪ Paginazione

- Obiettivo: combinare i vantaggi del partizionamento statico [no frammentazione esterna] con quelli del partizionamento dinamico [no frammentazione interna]
- Da un punto di vista logico si utilizzano **pagine di memoria** di dimensione fissa [S]:
  - S è “più piccola” rispetto a quella utilizzata per il partizionamento statico
  - Ad un processo viene allocato un insieme di pagine di memoria
- Analogamente, anche la memoria fisica è partizionata in unità di dimensione S definite **frames**
  - Una pagina di memoria allocata ad un processo **può** essere caricata in un frame disponibile
  - Una pagina di memoria allocata ad un processo può **non** essere caricata in un frame di memoria ma risiedere su disco
  - I frame che contengono le pagine di memoria allocate ad un processo, non necessariamente occupano un'area di memoria fisica contigua
- La tecnica di paginazione non elimina completamente la frammentazione interna ma la riduce fortemente offrendo maggiore flessibilità (nell'allocazione della memoria a processi con esigenze diverse) rispetto al partizionamento statico

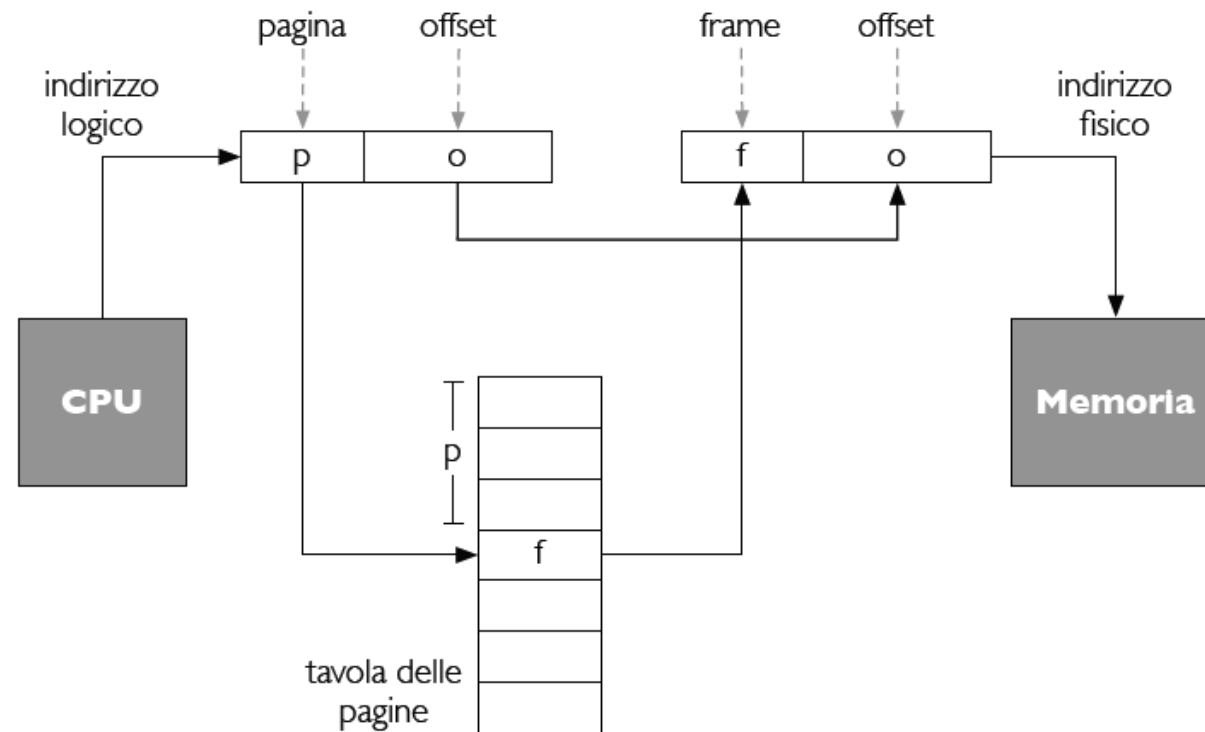
# Memoria virtuale



## □ Partizionamento della memoria

### ▪ Paginazione

- Il mapping **pagina di memoria**  $\leftrightarrow$  **frame** è affidato ad una struttura dati del SO detta **tabella delle pagine** [page table], indirizzata dalla componente “indirizzo di pagina” dell’indirizzo logico



# Memoria virtuale



## □ Partizionamento della memoria

### ▪ Paginazione

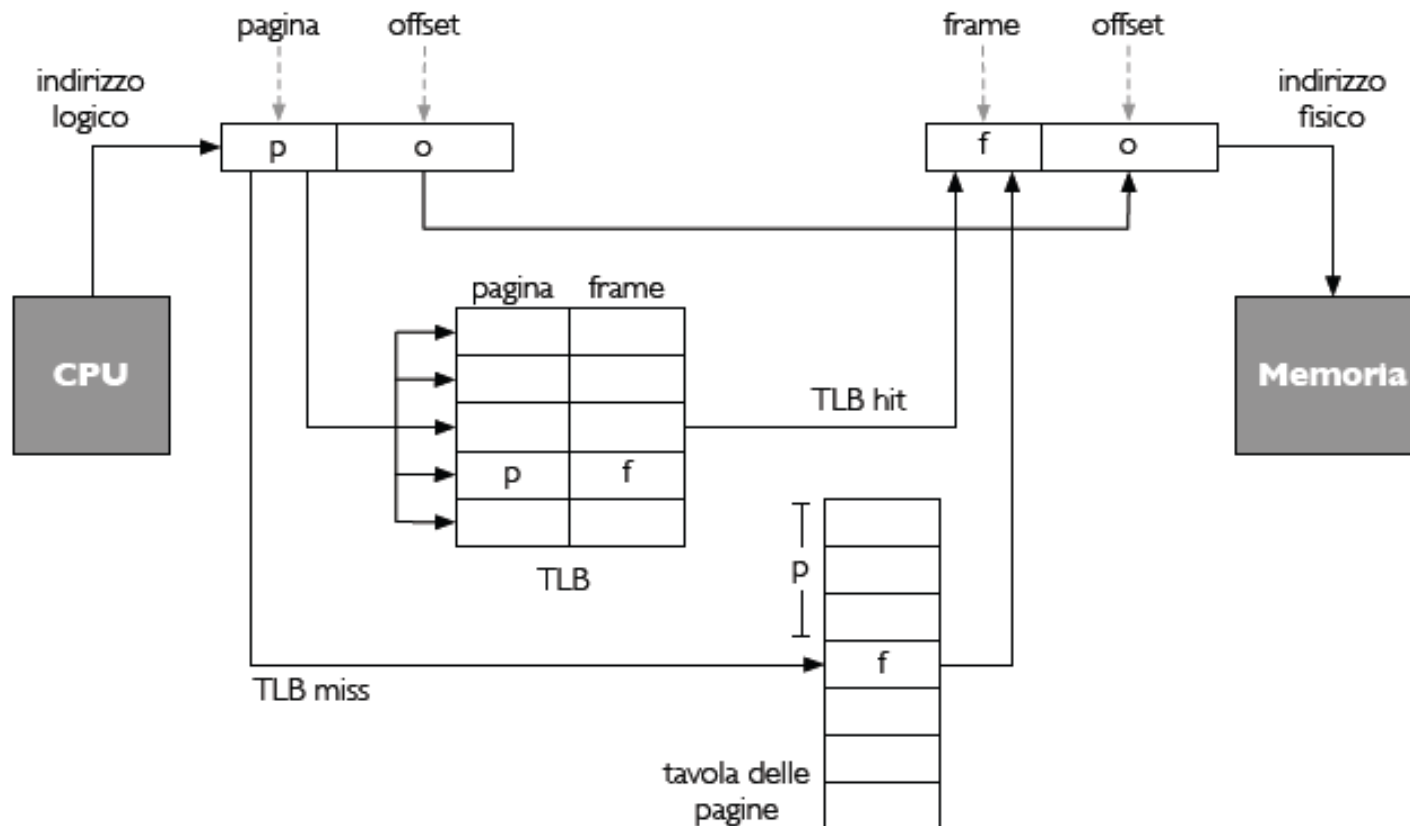
- Per questioni di efficienza [incrementare la velocità del mapping], la traduzione pagina di memoria  $\leftrightarrow$  frame è affidata ad una particolare memoria cache detta **TLB** [**translation lookaside buffer**]
  - Contiene una porzione della tabella delle pagine; [TLB = cache per la tabella delle pagine]
  - È tipicamente una memoria associativa [CAM - Content Addressable Memory]: le parole di memoria sono indirizzate attraverso una chiave di ricerca e non attraverso l'indirizzo.
  - Se la traduzione cercata non è presente nel TLB occorre risalire alla tabella delle pagine completa con i meccanismi analoghi alle cache
  - Il TLB può risiedere:
    - tra la CPU e la cache: la cache è indirizzata utilizzando indirizzamento fisico
    - tra la cache e la memoria primaria: la cache è indirizzata utilizzando indirizzamento virtuale.

# Memoria virtuale



## □ Partizionamento della memoria

- Paginazione
  - TLB





### □ Memoria virtuale e partizionamento della memoria

- Un processo può essere eseguito anche se il suo spazio di indirizzamento è solo **parzialmente** presente in memoria
  - Per aumentare il grado di multiprogrammazione, solo una frazione della memoria allocata al processo è presente in memoria
  - La restante parte risiede in un'area di memoria secondaria [**backing store**]
    - In alcune circostanze [swap out], tutta la memoria allocata ad un processo può risiedere in memoria secondaria

# Memoria virtuale



## □ Memoria virtuale e partizionamento della memoria

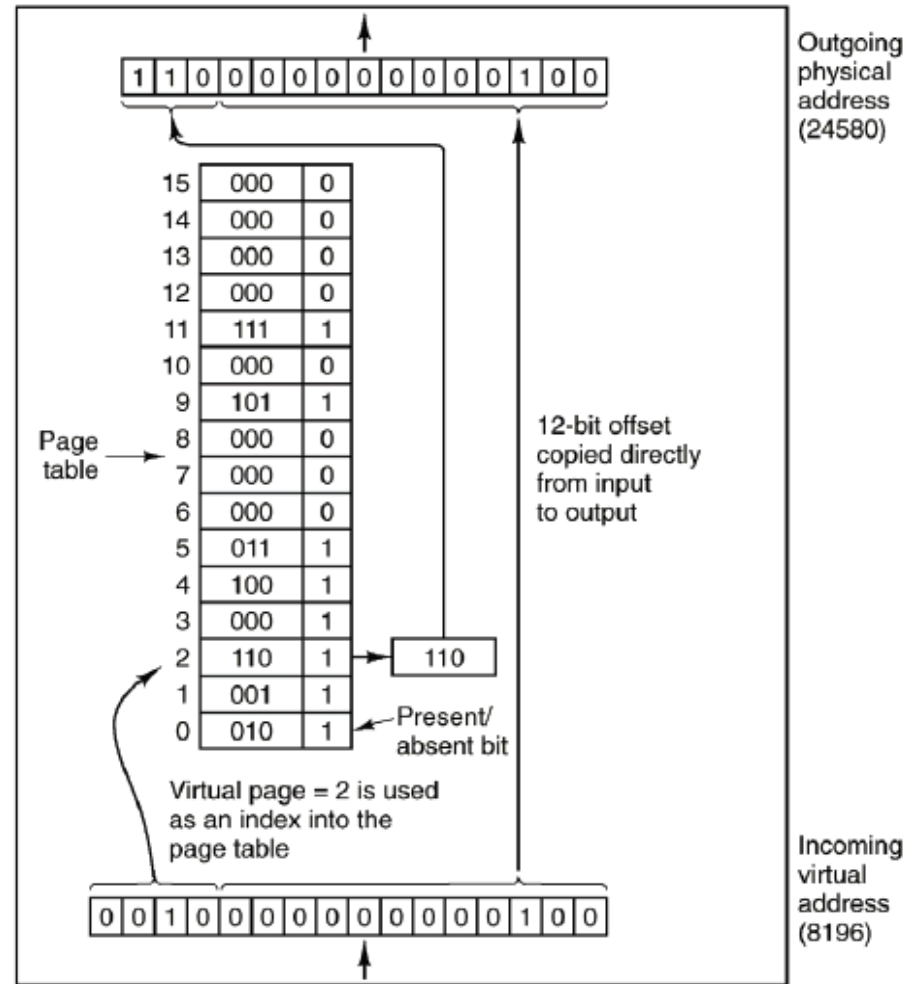
- Memoria virtuale paginata
  - Solo una frazione delle pagine di memoria allocate al processo è contenuta nei frames di memoria fisica
  - Quando caricare una pagina di memoria dalla backing store in un frame di memoria fisica? Due strategie:
    - **Paginazione su richiesta**
      - Una pagina viene caricata in un frame solo all'occorrenza ossia quando la CPU richiede un indirizzo logico che fa riferimento ad una pagina che non si trova in un frame di memoria [⇒**page fault**: accesso a memoria secondaria per recuperare la pagina]
      - Un page fault prevede un accesso a memoria secondaria che è **costosissimo** dal punto di vista del tempo d'accesso
    - **Prepaginazione**
      - A seguito di page fault, più pagine di memoria vengono portate in memoria fisica [inclusa quella richiesta]
      - Tende ad assorbire il costo dell'accesso a memoria secondaria sulla base del principio di località

# Memoria virtuale



## □ Memoria virtuale e partizionamento della memoria

- Memoria virtuale paginata
  - Per stabilire l'occorrenza di un page fault si utilizza un **presence bit** [bit di presenza] nella tabella delle pagine



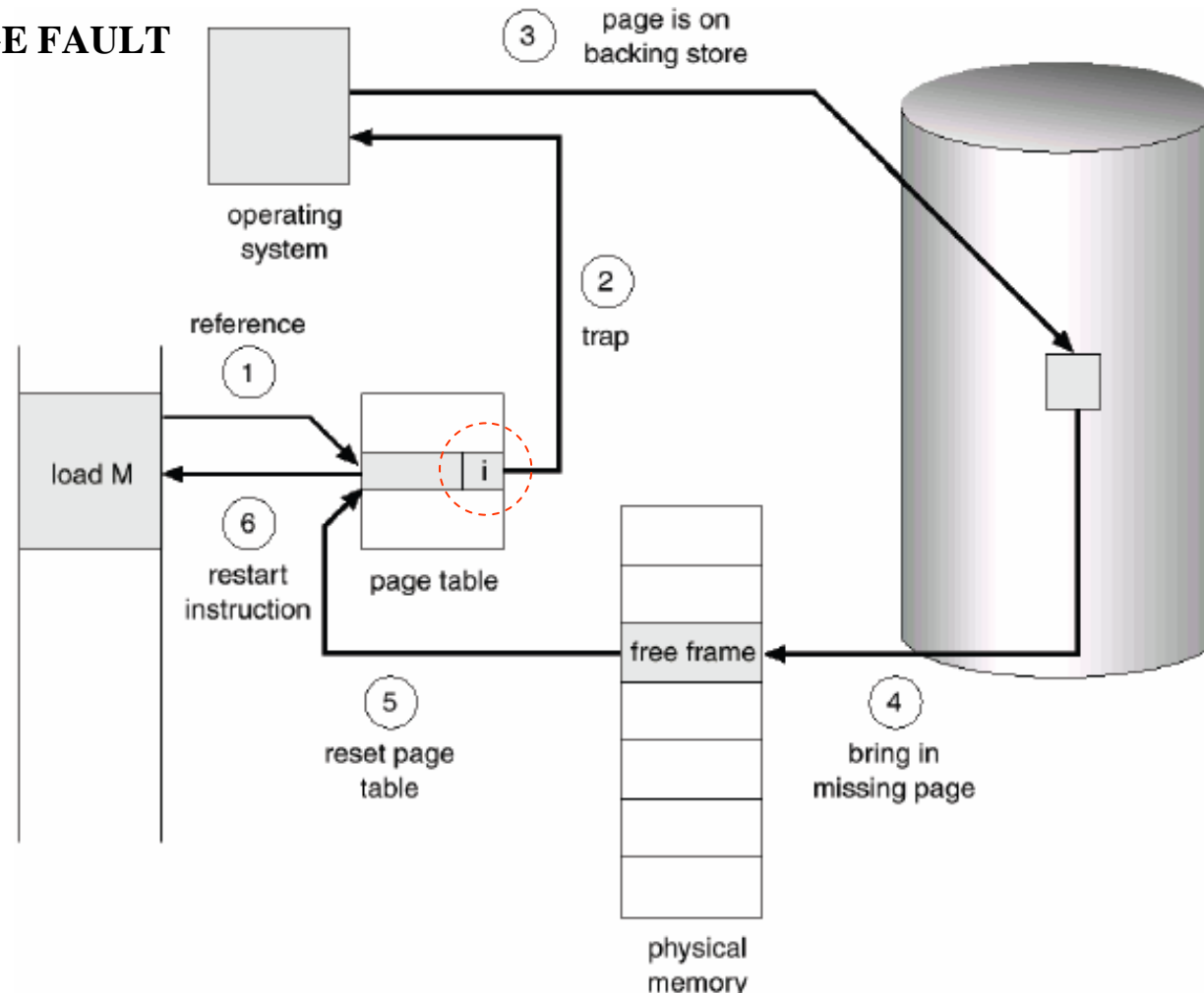
# Memoria virtuale



## □ Memoria virtuale e partizionamento della memoria

- Memoria virtuale paginata

- **PAGE FAULT**

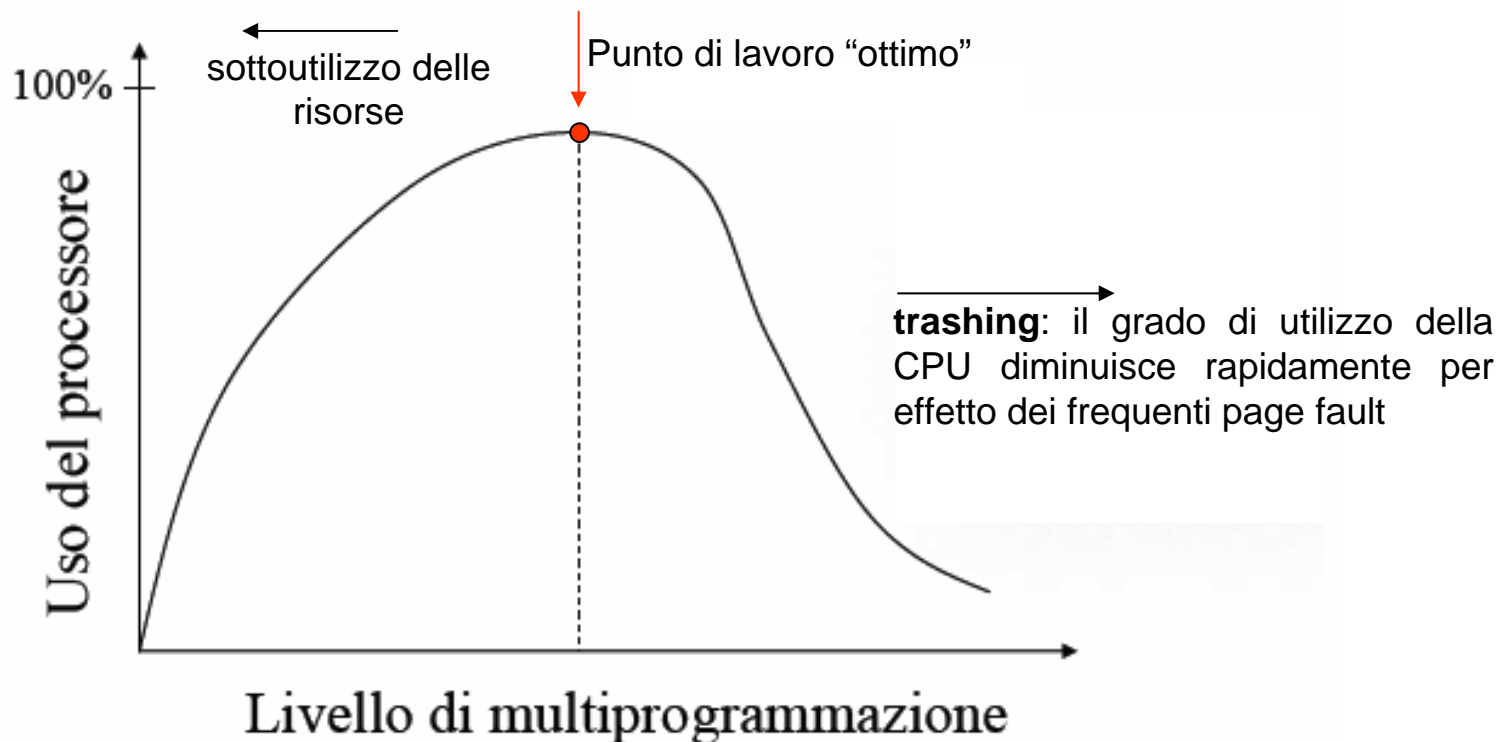


## Memoria virtuale



### □ Memoria virtuale e partizionamento della memoria

- Il concetto di memoria virtuale e la penalty elevata introdotta da un page fault, consentono di motivare la zona di **trashing** nel grafico analizzato in precedenza
- Infatti, aumentando il grado di multiprogrammazione oltre un certo limite, il numero di pagine contenute in memoria fisica per ogni processo si riduce eccessivamente  $\Rightarrow$  questo genera un numero eccesso di page fault  $\Rightarrow$  **degrado delle prestazioni**



# Memoria virtuale



## □ Memoria virtuale e partizionamento della memoria

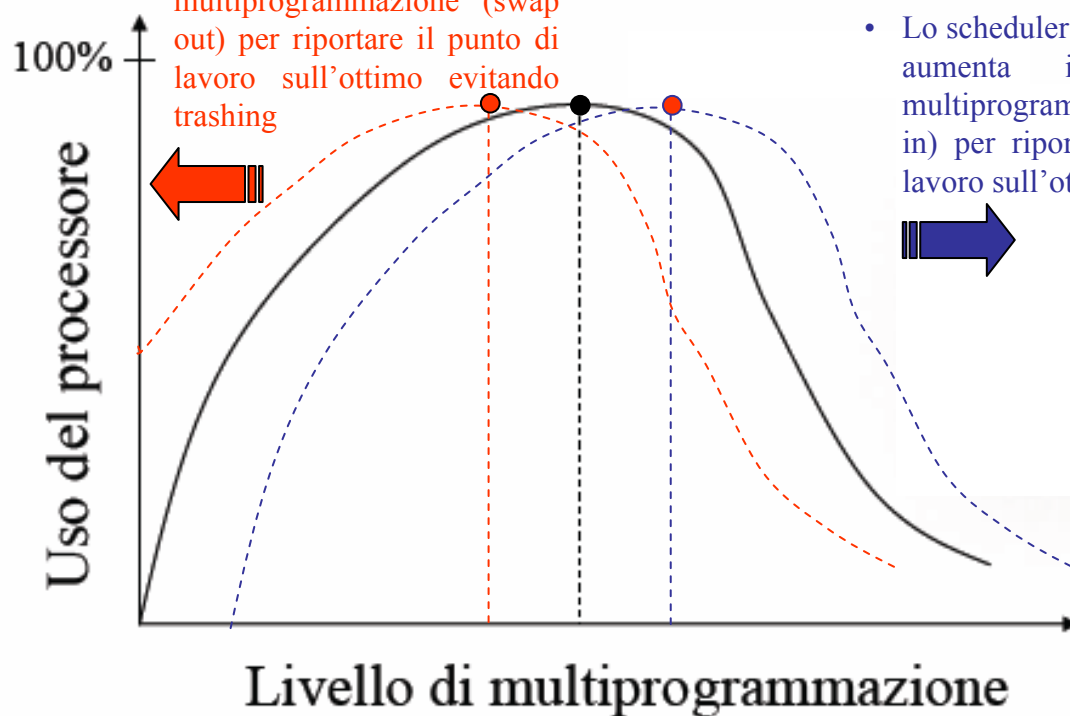
- Come già visto, questa curva è dinamica in quanto risulta funzione delle esigenze correnti di memoria dei processi [più o meno pagine in memoria]

- Aumentano le esigenze di memoria medie dei processi

- Lo scheduler a medio termine diminuisce il grado di multiprogrammazione (swap out) per riportare il punto di lavoro sull'ottimo evitando trashing

- Diminuiscono le esigenze di memoria medie dei processi

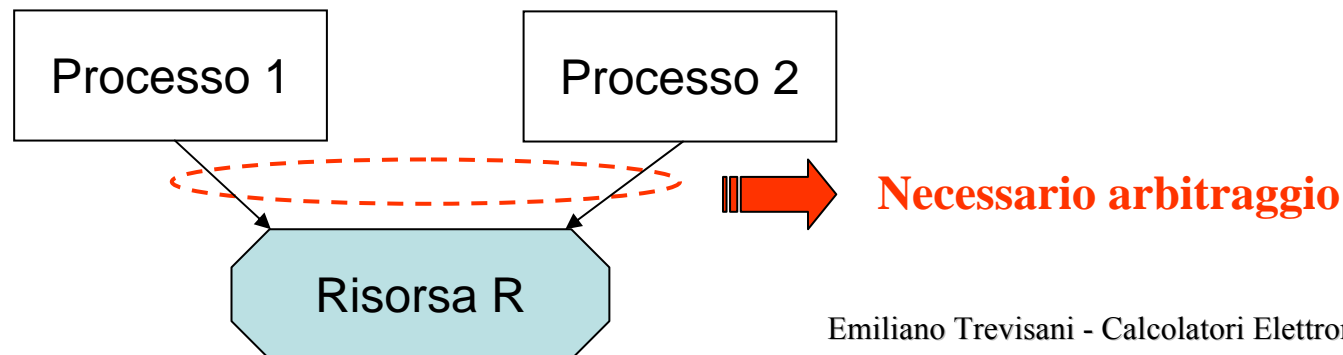
- Lo scheduler a medio termine aumenta il grado di multiprogrammazione (swap in) per riportare il punto di lavoro sull'ottimo



## Allocazione di risorse e deadlock



- ❑ Abbiamo già visto come il SO agisca da **resource manager** arbitrando le richieste di accesso dei processi attivi alle risorse di sistema
- ❑ Il termine risorsa si riferisce genericamente a quanto occorre ad un processo per proseguire la sua esecuzione
  - entità fisiche [es. periferiche]
  - entità logiche [risorse di calcolo]
- ❑ Le considerazioni che faremo nel seguito non si riferiscono alla risorsa logica CPU (per la quale lo scheduler a breve termine agisce già da arbitro) ma alla condizione di attesa di eventi esterni per un processo (es. stato “blocked” nel diagramma degli stati dei processi)
- ❑ Poiché la quantità di risorse è in generale limitata: come arbitrare le richieste conflittuali di processi concorrenti? [Problema dell'**allocazione delle risorse**]



## Allocazione di risorse e deadlock



- ❑ In generale possono esistere istanze multiple della stessa risorsa [unità di risorsa]
  - Con riferimento all'esigenza di una risorsa da parte di un processo, ciascuna delle istanze è equivalente
- ❑ Accesso di un processo ad una risorsa:
  - Richiesta al SO; se non esistono unità di risorsa disponibili il processo resta bloccato in attesa di disponibilità
  - Allocazione da parte del SO: un'unità di risorsa disponibile viene allocata al processo che può, pertanto, utilizzarla
  - Rilascio da parte del processo: il processo rilascia l'unità di risorsa terminato il suo utilizzo; questa ritorna nella disponibilità del SO
  - Per implementare un meccanismo di questo tipo il SO deve tenere traccia:
    - **dello stato di allocazione corrente delle risorse** [per ciascuna unità di risorsa: disponibile o allocata a quale processo]
    - **delle richieste dei processi non ancora servite**

## Allocazione di risorse e deadlock



□ Per rappresentare, in un certo istante, entrambe queste informazioni si ricorre al **grafo di allocazione delle risorse**:

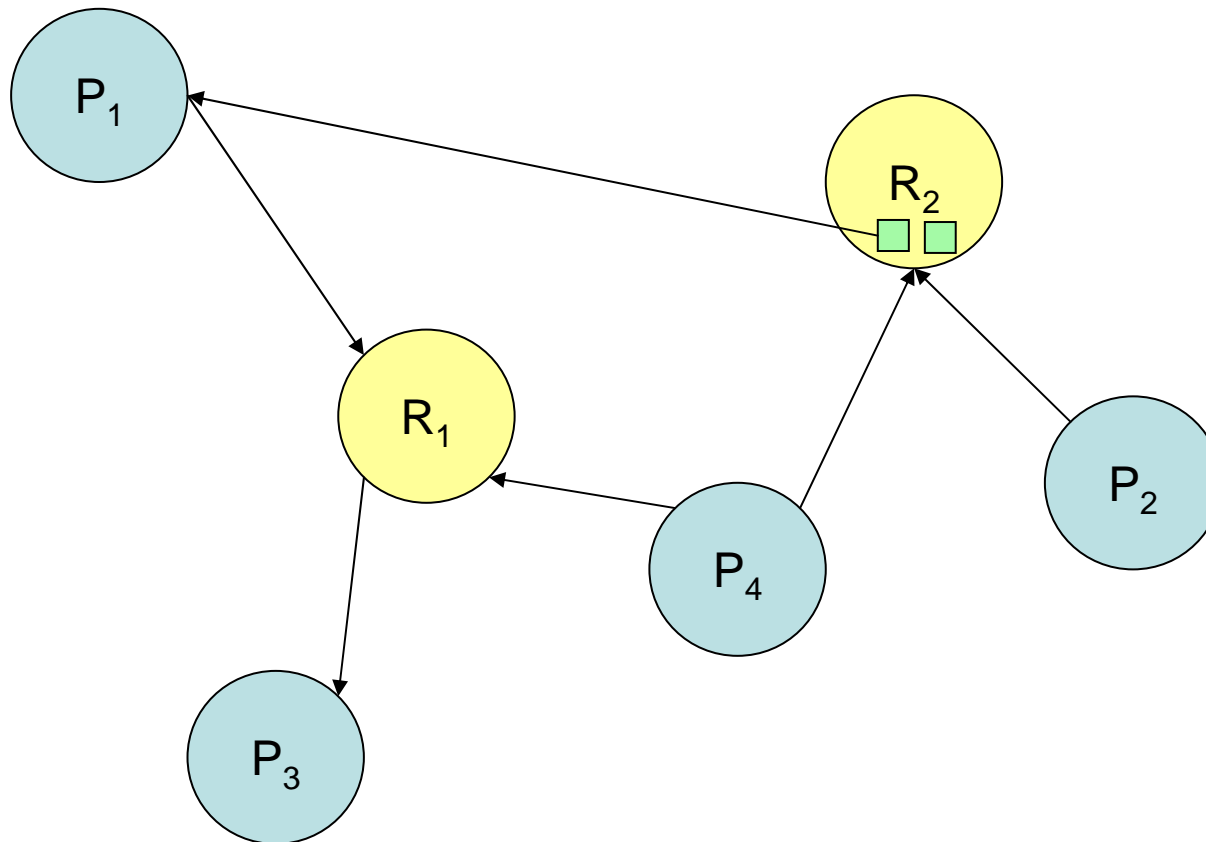
- Siano:
  - $P = \{P_1, P_2, \dots, P_n\}$  l'insieme dei processi [n processi]
  - $R = \{R_1, R_2, \dots, R_k\}$  l'insieme delle risorse [k risorse, ciascuna con un numero generico di unità di risorsa]
- I vertici del grafo di allocazione delle risorse appartengono all'insieme  $P \cup R$
- Gli archi del grafo di allocazione delle risorse sono così definiti:
  - l'arco  $R_i \rightarrow P_j$  indica lo stato di allocazione corrente della risorsa  $R_i$  al processo  $P_j$ ;  $i \in \{1, \dots, k\}$ ,  $j \in \{1, \dots, n\}$ 
    - Se  $R_i$  possiede istanze multiple, si potrà avere un arco uscente da ciascuna di queste
  - l'arco  $P_j \rightarrow R_i$  indica la richiesta corrente [non ancora servita] della risorsa  $R_i$  da parte del processo  $P_j$ ;  $i \in \{1, \dots, k\}$ ,  $j \in \{1, \dots, n\}$
- Il grafo è di natura **dinamica**: modella lo stato di assegnazione delle risorse e le richieste pendenti

## Allocazione di risorse e deadlock



□ Grafo di allocazione delle risorse:

▪ Esempio 1:

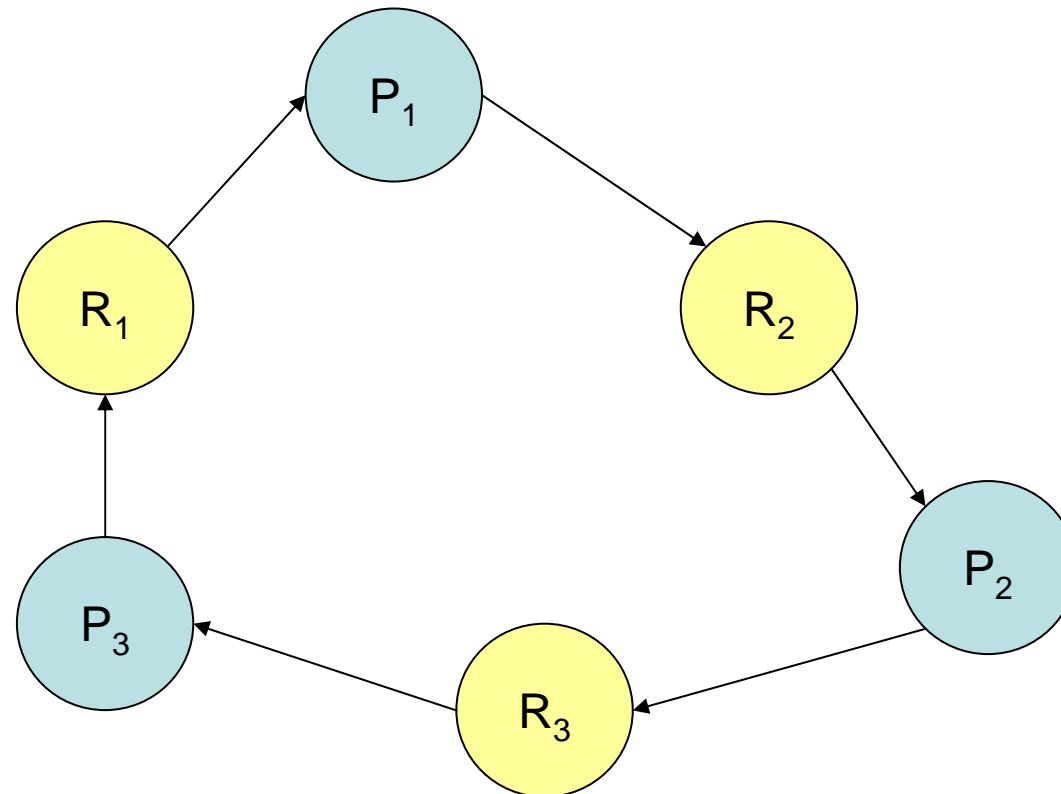


## Allocazione di risorse e deadlock



□ Grafo di allocazione delle risorse:

▪ Esempio 2:



**Condizione di STALLO**  
**[deadlock]**

## Allocazione di risorse e deadlock



### □ Condizione di stallo [deadlock]

- Si riferisce ad una condizione per la quale due o più processi risultano in attesa indefinita di risorse la cui disponibilità [rilascio] dipende dal proseguimento della loro esecuzione
  - L'assenza di cicli nel grafo di allocazione delle risorse implica l'assenza di deadlock
  - La presenza di un ciclo nel grafo di allocazione delle risorse implica la condizione di deadlock se il ciclo coinvolge solo risorse ad istanza singola; nel caso in cui almeno una risorsa abbia istanze multiple la condizione di deadlock **può** verificarsi
- La condizione di stallo non prevede vie d'uscita a parte la terminazione dei processi coinvolti

## Allocazione di risorse e deadlock



### □ Condizione di stallo [deadlock]

- Si può provare che esistono 4 **condizioni necessarie** affinché la condizione di stallo possa verificarsi
  - A. Mutua esclusione:** almeno una delle risorse coinvolte deve risultare non condivisibile (es: risorsa a singola istanza)
  - B. Possesso ed attesa:** processi che detengono risorse possono richiederne altre.
  - C. Impossibilità di prelazione:** solo il processo che detiene la risorsa può rilasciarla.
  - D. Attesa circolare:** esiste un sottoinsieme S dell'insieme dei processi tale per cui: il primo elemento di S è in attesa di una risorsa assegnata al secondo, quest'ultimo è in attesa di una risorsa assegnata al terzo, e così via fino a chiudere il ciclo
- Osservazione: è sufficiente **negare** una delle 4 condizioni per evitare la condizione di stallo

$$STALLO \Rightarrow (A \wedge B \wedge C \wedge D) \quad \longleftrightarrow \quad \overline{STALLO} \Leftarrow (\bar{A} \vee \bar{B} \vee \bar{C} \vee \bar{D})$$

$$Oss : \alpha \Rightarrow \beta \Leftrightarrow \bar{\alpha} \Leftarrow \bar{\beta}$$



### □ Condizione di stallo [deadlock]

- Approcci al problema:
  - **Prevenzione dello stallo:** negare una delle 4 condizioni necessarie
  - **Evitare lo stallo:** introdurre un meccanismo di contabilizzazione delle risorse che consenta di verificare, preliminarmente, se l'assegnazione di risorse ad un processo porta il sistema in uno stato a rischio di stallo
  - **Risolvere lo stallo:** introdurre meccanismi di diagnosi dello stallo e terminare i processi per risolvere il deadlock
  - **Ignorare lo stallo:** assumendo la condizione di stallo poco probabile si ignora; fault del sistema in caso di stallo

## Allocazione di risorse e deadlock



### ❑ **Prevenzione dello stallo**

- Evitare mutua esclusione:
  - Aumentare il numero di istanze disponibili per risorsa
- Evitare possesso ed attesa
  - I processi potrebbero richiedere preliminarmente tutte le risorse delle quali necessitano
  - I processi potrebbero essere costretti a rilasciare le risorse assegnate prima di richiederne altre
- Evitare impossibilità di prelazione
  - Consentire la prelazione: questo potrebbe lasciare i processi prelati in uno stato inconsistente.
- Evitare attesa circolare:

## Allocazione di risorse e deadlock



### ❑ Evitare lo stallo

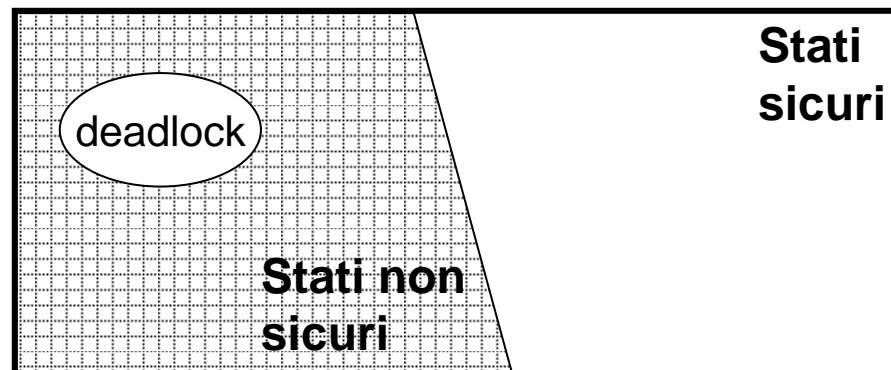
- Concetto di **sequenza sicura**:
  - corrisponde ad un ordinamento dei processi attivi tale per cui
    - il primo processo dell'ordinamento può terminare
    - il secondo processo dell'ordinamento può terminare dopo la terminazione del primo
    - in generale, l'*i*-esimo processo dell'ordinamento può terminare dopo la terminazione dei processi precedenti
  - Oss: “processo può terminare” deve essere inteso “il processo ha risorse sufficienti per terminare la sua esecuzione”
- Concetto di **stato sicuro**:
  - uno stato di allocazione delle risorse è sicuro se esiste almeno una sequenza sicura a partire da quello stato
  - uno stato sicuro corrisponde ad uno stato di allocazione delle risorse ai processi **non a rischio di stallo**

## Allocazione di risorse e deadlock



### ❑ Evitare lo stallo

- Idea: introdurre un meccanismo di **contabilizzazione delle risorse** che consenta di verificare, preliminarmente, se l'assegnazione di risorse ad un processo porta il sistema in uno stato non sicuro
  - Occorre tener traccia:
    - delle risorse correntemente disponibili
    - delle risorse correntemente allocate
    - del numero massimo di risorse che ogni processo potrà richiedere.



## Allocazione di risorse e deadlock



### ❑ Evitare lo stallo

- **Algoritmo del banchiere:** consente di verificare se uno stato di allocazione delle risorse è sicuro.
  - Utilizza le seguenti strutture dati:
    - **Vettore AVAILABLE**
      - $AVAILABLE[j]$  indica il numero di unità di risorsa disponibili per la risorsa  $j$ .
    - **Matrice MAX**
      - $MAX[i,j]$  indica il massimo numero di unità di risorsa della risorsa  $j$  che possono essere richieste dal processo  $i$  durante la sua esecuzione.
    - **Matrice ALLOCATION**
      - $ALLOCATION[i,j]$  indica il numero di unità di risorsa della risorsa  $j$  attualmente allocate al processo  $i$ .
    - **Matrice NEED**
      - $NEED[i,j]$  indica il numero di unità di risorsa della risorsa  $j$  che potranno essere richieste in futuro dal processo  $i$
  - Si osservi che  $NEED[i,j] = MAX[i,j] - ALLOCATION[i,j]$

## Allocazione di risorse e deadlock



- **Algoritmo del banchiere**
  - Sia TEMP un vettore di dimensione  $m$  [numero di risorse]
  - Sia TERMINATED un vettore di dimensione  $n$  [numero di processi]
  - Fase di inizializzazione:
    - $\forall j$  TEMP[j]=AVAILABLE[j]
    - $\forall i$  TERMINATED[i]=FALSE
  - Step di computazione:
    - Step1
      - Cerca un indice  $i$  tale che  $TERMINATED[i] == FALSE$  and  $\forall j$   $NEED[i,j] \leq TEMP[j]$
      - Se tale  $i$  non esiste allora vai al passo 3
    - Step2
      - $\forall j$  TEMP[j]=TEMP[j]+ALLOCATION[i,j]
      - TERMINATED[i]=TRUE
      - Torna allo Step1
    - Step3
      - Se  $\forall i$  TERMINATED[i]=TRUE allora  $\Rightarrow$  STATO SICURO

## Allocazione di risorse e deadlock



- **Algoritmo del banchiere:**
  - Osservazioni:
    - L'idea dell'algoritmo è quella di verificare se lo stato è sicuro cercando di individuare una sequenza sicura
    - Lo Step2 simula la terminazione del processo  $i$  [ $TERMINATED[i]==TRUE$ ] rilasciando tutte le risorse ad esso assegnate; queste potranno servire per consentire la terminazione dei processi successivi nell'ordinamento associato alla sequenza sicura
    - Il confronto eseguito nello Step1 tiene conto del caso peggiore: devo assicurarmi l'impossibilità di andare in stallo [stato sicuro] nel caso peggiore ossia quando tutti i processi richiedono ciascuno il massimo delle risorse residue richiedibili [la matrice NEED]
    - In uno stato non sicuro non è detto che i processi vadano in stallo; se in uno stato non sicuro tutti i processi richiedessero ciascuno il massimo delle risorse residue richiedibili si avrebbe certamente una condizione di stallo
    - Svantaggi:
      - Un processo deve pre-dichiarare staticamente MAX
      - Basso grado di utilizzo delle risorse dovuto all'approccio fortemente cautelativo
      - Alto costo computazionale [devo eseguire l'algoritmo ad ogni richiesta]

## Allocazione di risorse e deadlock



### □ Esempio: applicazione dell'algoritmo del banchiere

- Considerando il seguente scenario e lo stato di allocazione delle risorse specificato da ALLOCATION, stabilire se lo stato è sicuro.
  - 3 processi  $P_1, P_2, P_3$
  - 2 risorse:  $R_1$ [3 unità],  $R_2$ [2 unità]
  - MAX, AVAILABLE, ALLOCATION

MAX		
	$R_1$	$R_2$
$P_1$	2	1
$P_2$	2	2
$P_3$	1	1

ALLOCATION		
	$R_1$	$R_2$
$P_1$	1	0
$P_2$	0	1
$P_3$	0	1

AVAILABLE	
$R_1$	2
$R_2$	0

## Allocazione di risorse e deadlock



### □ Esempio: applicazione dell'algoritmo del banchiere

- Inizializzazione:

TEMP	
R <sub>1</sub>	2
R <sub>2</sub>	0

TERMINATED	
P <sub>1</sub>	FALSE
P <sub>2</sub>	FALSE
P <sub>3</sub>	FALSE

NEED		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	1
P <sub>2</sub>	2	1
P <sub>3</sub>	1	0

MAX		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	2	1
P <sub>2</sub>	2	2
P <sub>3</sub>	1	1

ALLOCATION		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	0
P <sub>2</sub>	0	1
P <sub>3</sub>	0	1

AVAILABLE	
R <sub>1</sub>	2
R <sub>2</sub>	0


# Allocazione di risorse e deadlock



## □ Esempio: applicazione dell'algoritmo del banchiere

- Step1 [ $i=3$ ]

TEMP	
R <sub>1</sub>	2
R <sub>2</sub>	0



TERMINATED	
P <sub>1</sub>	FALSE
P <sub>2</sub>	FALSE
P <sub>3</sub>	FALSE

NEED		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	1
P <sub>2</sub>	2	1
P <sub>3</sub>	1	0

MAX		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	2	1
P <sub>2</sub>	2	2
P <sub>3</sub>	1	1

ALLOCATION		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	0
P <sub>2</sub>	0	1
P <sub>3</sub>	0	1

AVAILABLE	
R <sub>1</sub>	2
R <sub>2</sub>	0

## Allocazione di risorse e deadlock



### □ Esempio: applicazione dell'algoritmo del banchiere

- Step2 [ $i=3$ ]

TEMP	
R <sub>1</sub>	2
R <sub>2</sub>	<b>1</b>

TERMINATED	
P <sub>1</sub>	FALSE
P <sub>2</sub>	FALSE
<b>P<sub>3</sub></b>	<b>TRUE</b>

NEED		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	1
P <sub>2</sub>	2	1
P <sub>3</sub>	1	0

MAX		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	2	1
P <sub>2</sub>	2	2
P <sub>3</sub>	1	1

ALLOCATION		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	0
P <sub>2</sub>	0	1
P <sub>3</sub>	0	1

AVAILABLE	
R <sub>1</sub>	2
R <sub>2</sub>	0

## Allocazione di risorse e deadlock



### □ Esempio: applicazione dell'algoritmo del banchiere

- Step1 [ $i=1$ ]

TEMP	
R <sub>1</sub>	2
R <sub>2</sub>	1



TERMINATED	
P <sub>1</sub>	FALSE
P <sub>2</sub>	FALSE
P <sub>3</sub>	TRUE

NEED		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	1
P <sub>2</sub>	2	1
P <sub>3</sub>	1	0

MAX		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	2	1
P <sub>2</sub>	2	2
P <sub>3</sub>	1	1

ALLOCATION		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	0
P <sub>2</sub>	0	1
P <sub>3</sub>	0	1

AVAILABLE	
R <sub>1</sub>	2
R <sub>2</sub>	0

## Allocazione di risorse e deadlock



### □ Esempio: applicazione dell'algoritmo del banchiere

- Step2 [ $i=1$ ]

TEMP	
R <sub>1</sub>	3
R <sub>2</sub>	1

TERMINATED	
P <sub>1</sub>	TRUE
P <sub>2</sub>	FALSE
P <sub>3</sub>	TRUE

NEED		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	1
P <sub>2</sub>	2	1
P <sub>3</sub>	1	0

MAX		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	2	1
P <sub>2</sub>	2	2
P <sub>3</sub>	1	1

ALLOCATION		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	0
P <sub>2</sub>	0	1
P <sub>3</sub>	0	1

AVAILABLE	
R <sub>1</sub>	2
R <sub>2</sub>	0


## Allocazione di risorse e deadlock



### □ Esempio: applicazione dell'algoritmo del banchiere

- Step1 [ $i=2$ ]

TEMP	
R <sub>1</sub>	3
R <sub>2</sub>	1



TERMINATED	
P <sub>1</sub>	TRUE
P <sub>2</sub>	FALSE
P <sub>3</sub>	TRUE

NEED		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	1
P <sub>2</sub>	2	1
P <sub>3</sub>	1	0

MAX		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	2	1
P <sub>2</sub>	2	2
P <sub>3</sub>	1	1

ALLOCATION		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	0
P <sub>2</sub>	0	1
P <sub>3</sub>	0	1

AVAILABLE	
R <sub>1</sub>	2
R <sub>2</sub>	0

## Allocazione di risorse e deadlock



### □ Esempio: applicazione dell'algoritmo del banchiere

- Step2 [ $i=2$ ]

TEMP	
R <sub>1</sub>	3
R <sub>2</sub>	2

TERMINATED	
P <sub>1</sub>	TRUE
P <sub>2</sub>	TRUE
P <sub>3</sub>	TRUE

NEED		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	1
P <sub>2</sub>	2	1
P <sub>3</sub>	1	0

MAX		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	2	1
P <sub>2</sub>	2	2
P <sub>3</sub>	1	1

ALLOCATION		
	R <sub>1</sub>	R <sub>2</sub>
P <sub>1</sub>	1	0
P <sub>2</sub>	0	1
P <sub>3</sub>	0	1

AVAILABLE	
R <sub>1</sub>	2
R <sub>2</sub>	0

# Allocazione di risorse e deadlock



## □ Esempio: applicazione dell'algoritmo del banchiere

- Step3: STATO SICURO [Sequenza sicura  $P_3 P_1 P_2$ ]

TEMP	
$R_1$	3
$R_2$	2

TERMINATED	
$P_1$	TRUE
$P_2$	TRUE
$P_3$	TRUE

	NEED	
	$R_1$	$R_2$
$P_1$	1	1
$P_2$	2	1
$P_3$	1	0

Osservazioni:

- Lo stato è sicuro: anche se in questo stato i processi richiedessero ciascuno il massimo delle risorse residue richiedibili [NEED], per come è costruito l'algoritmo, non si potrebbe raggiungere la condizione di stallo
- Questa circostanza corrisponde alla condizione più sfavorevole, per la quale abbiamo visto esiste la sequenza sicura  $P_3 P_1 P_2$  che evita lo stallo

	MAX	
	$R_1$	$R_2$
$P_1$	2	1
$P_2$	2	2
$P_3$	1	1

	ALLOCATION	
	$R_1$	$R_2$
$P_1$	1	0
$P_2$	0	1
$P_3$	0	1

AVAILABLE	
$R_1$	2
$R_2$	0