

# Sistemi Operativi

Docente: Ugo Erra  
*ugoerr+so@dia.unisa.it*



## 8° LEZIONE MEMORIA CENTRALE – 1° PARTE

*CORSO DI LAUREA TRIENNALE IN INFORMATICA  
UNIVERSITA' DEGLI STUDI DELLA BASILICATA*



# Sommario della lezione



- Introduzione
- Associazione degli indirizzi
- Indirizzi logici e indirizzi fisici
- Uso delle librerie
- Avvicendamento delle librerie
- Allocazione della memoria

# Introduzione



- Nei moderni Sistemi Operativi la gestione della memoria è un aspetto fondamentale per la realizzazione della multiprogrammazione poiché più processi attivi devono essere presenti in memoria
- Lo scopo del Sistema Operativo è di allocare la memoria per i processi da mandare in esecuzione in modo che siano pronti quando la CPU gli viene assegnata
- L'allocazione della memoria ai processi è quindi uno degli scopi di un Sistema Operativo

# La CPU ed i processi

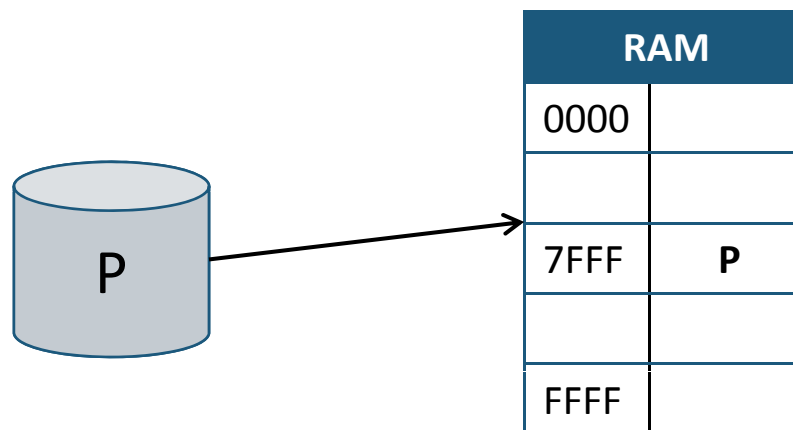


- Un programma per essere eseguito deve trovarsi nella memoria primaria
  - La CPU può accedere ai registri ed alla memoria ma non alle istruzioni o agli indirizzi presenti sul disco
- La CPU è progettata per eseguire le istruzioni o per caricare i dati in memoria nell'arco di uno o più cicli di clock
  - I dati in memoria solitamente possono impiegare più tempo a causa del passaggio sul bus di sistema
  - L'uso della **cache** può migliorare il recupero dei dati dalla memoria

# Avvio di un'applicazione



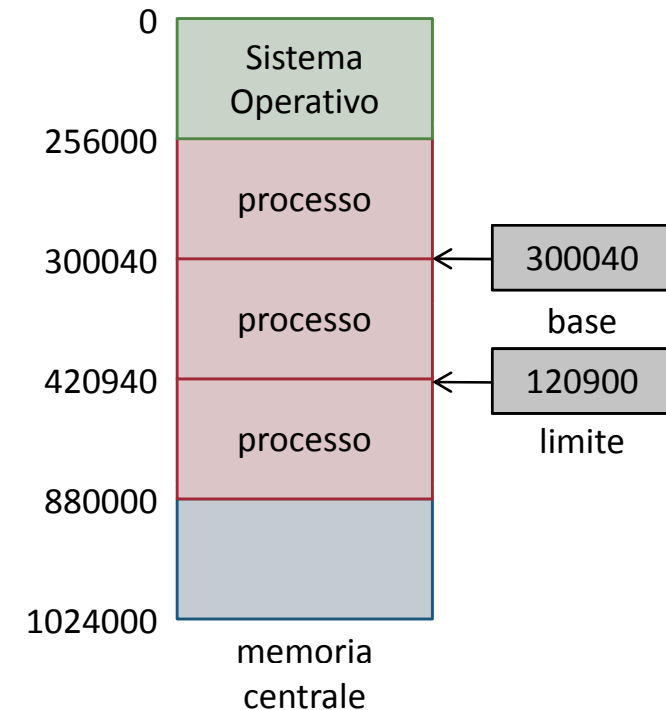
- Nel momento in cui l'utente manda in esecuzione un programma, il Sistema Operativo deve recuperare il codice dalla memoria secondaria e copiarlo in una zona della memoria primaria
- Il Sistema Operativo deve quindi decidere l'indirizzo di RAM dal quale avviare il processo
- Poiché nella RAM potrebbero esserci altri processi l'operazione non è immediata né semplice



# Registro base e Registro limite



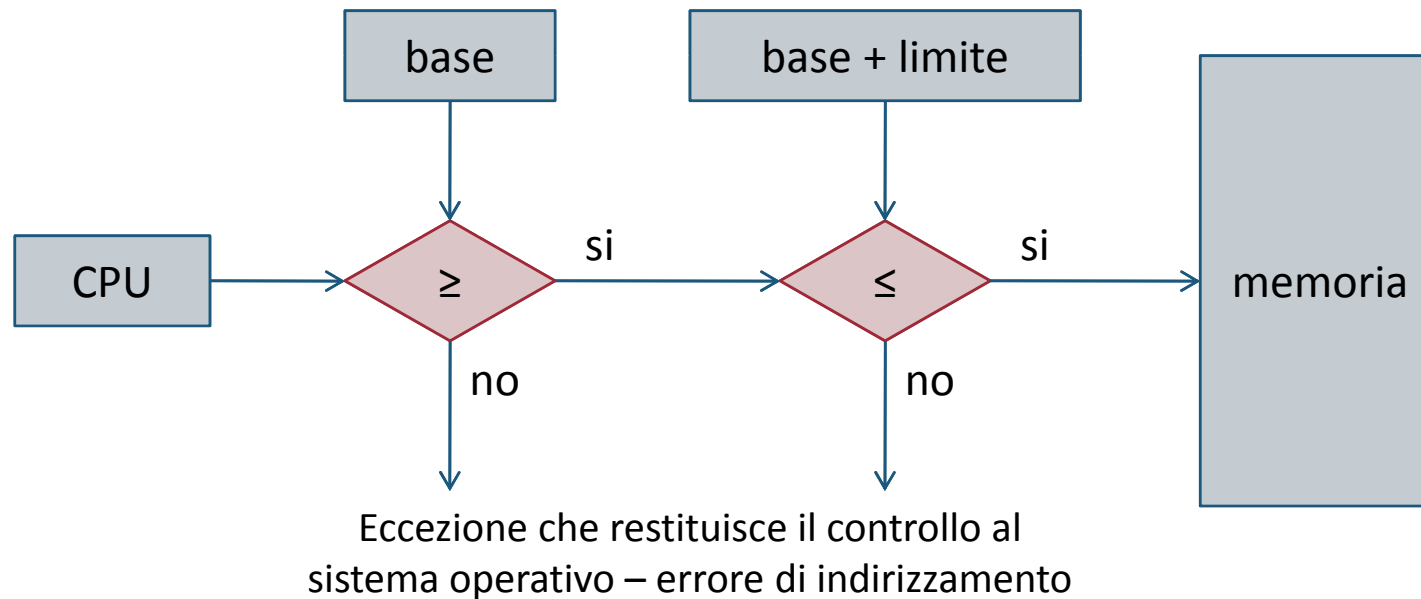
- Ogni processo deve avere uno spazio di memoria proprio e separato dal resto dei processi in modo che nessun altro possa accedervi
- Una soluzione consiste nell'utilizzare due registri
  - **Registro base** : contiene il più piccolo indirizzo fisico ammesso
  - **Registro limite** : contiene la dimensione dell'intervallo ammesso
- Solo il Sistema Operativo può accedere a questi registri ed impedisce ai programmi utenti di modificarli
  - Il S.O. può anche effettuare in questo modo un *dump* (copia) della memoria



# Protezione della memoria



- Per assicurare che non ci siano accessi illegali in memoria la CPU confronta ogni indirizzo generato dal processo con i valori contenuti nel registro base e nel registro limite

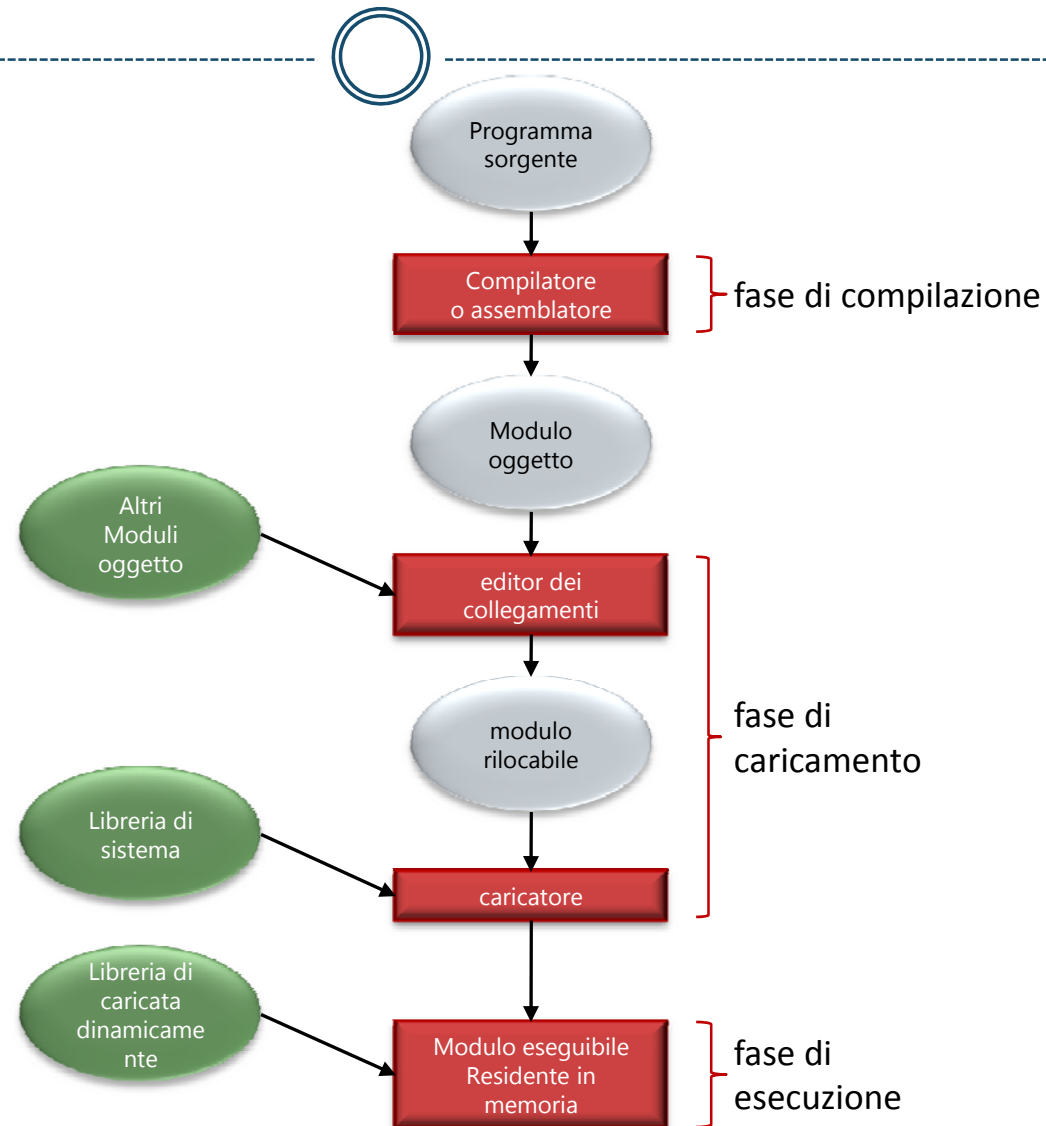


# Il problema dell'associazione degli indirizzi



- Supponiamo che tre processi  $P_1, P_2$  e  $P_3$  siano presenti in memoria
- In questo scenario sono possibili due eventi in un certo istante:
  1. Un nuovo processo  $P_4$  deve essere mandato in esecuzione (dove?)
  2. Il processo  $P_1$  in attesa di I/O viene momentaneamente spostato sull'hard disk per liberare memoria ad un processo da mandare in esecuzione. Successivamente  $P_1$  sarà nuovamente mandato in esecuzione (dove?)
- In entrambi i casi il Sistema Operativo deve essere in grado di mandare sempre in esecuzione il processo indipendentemente dall'indirizzo di partenza

# Fasi di elaborazioni di un programma utente



## Generazione del codice - 2



- Supponiamo di avere una istruzione in C del genere

```
counter = counter + 1;
```

in assembly avremo qualcosa del genere:

```
20500 0x00003
...
20600 load(R1, 20500)
20604 add(R1, #1);
20608 store(R1, 20500)
```

- La cella di memoria con indirizzo 20500 contiene il valore della variabile counter da incrementare

## Generazione del codice - 2

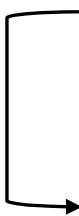


- Supponiamo di avere una istruzione in C del genere

```
if(counter!=100) counter=0
```

in assembly avremo qualcosa del genere:

```
20500 0x00003
...
300FC ... ..
30100 load(R1, 20500)
30104 bne(R1, #100, 30110);
30108 zero(R1)
3010C store(R1, 20500)
30110 ... ..
```



- La cella di memoria con indirizzo 20500 contiene il valore della variabile counter da incrementare

# Associazione degli indirizzi (binding)



- L'associazione di istruzioni e dati a indirizzi di memoria si può compiere in diverse fasi:
  - Compilazione
  - Caricamento
  - Esecuzione

# Compilazione



- Se in fase di compilazione è noto a priori dove risiederà il processo in memoria è possibile generare **codice assoluto**
  - Ad esempio se il processo deve iniziare dalla locazione  $r$  allora il compilatore genera codice a partire dalla locazione  $r$
- Il Sistema Operativo MS-DOS utilizzava questo meccanismo di associazione
- Quale è lo svantaggio principale di questo meccanismo?
  - Se la locazione di memoria iniziale cambia oppure non è disponibile non è possibile mandare in esecuzione il processo anche se è disponibile memoria libera da qualche altra parte

# Caricamento



- Se a priori non è possibile stabilire dove allocare il processo allora è necessario generare **codice rilocabile**
- Finché il processo non è caricato in memoria non è possibile associare al programma gli indirizzi di memoria
- L'associazione non può essere cambiata nel caso in cui programma viene momentaneamente spostato dal disco
- Nel caso in cui il programma deve essere spostato in memoria è necessario caricarlo di nuovo dal disco

# Esecuzione



- Se il programma durante la sua esecuzione può essere spostato da una parte all'altra della memoria allora l'associazione deve essere fatta a tempo di esecuzione
- Per poter utilizzare questo tipo di associazione è necessario un supporto dell'hardware
- I vantaggi principali sono:
  - Assoluta indipendenza da una locazione di memoria
  - Possibilità di cambiare il blocco di memoria occupato immediatamente
- I Sistemi Operativi moderni supportano questa modalità di associazione

# Spazi di indirizzi logici e fisici



- Gli indirizzi utilizzati da una applicazione sono diversi dagli indirizzi *realmente* utilizzati dall'applicazione in memoria
  - **Indirizzo logico** (o **indirizzi virtuali**): è generato dalla CPU nel momento della compilazione
  - **Indirizzo fisico** : è l'indirizzo effettivamente utilizzato dall'applicazione in memoria e caricato nel **registro di indirizzamento della memoria** (*memory address register, MAR*)
- L'associazione degli indirizzi basati sulla compilazione e sul caricamento producono indirizzi logici e fisici identici
- L'associazione degli indirizzi basati sull'esecuzione genera indirizzi logici e fisici diversi

# Unità di gestione della memoria

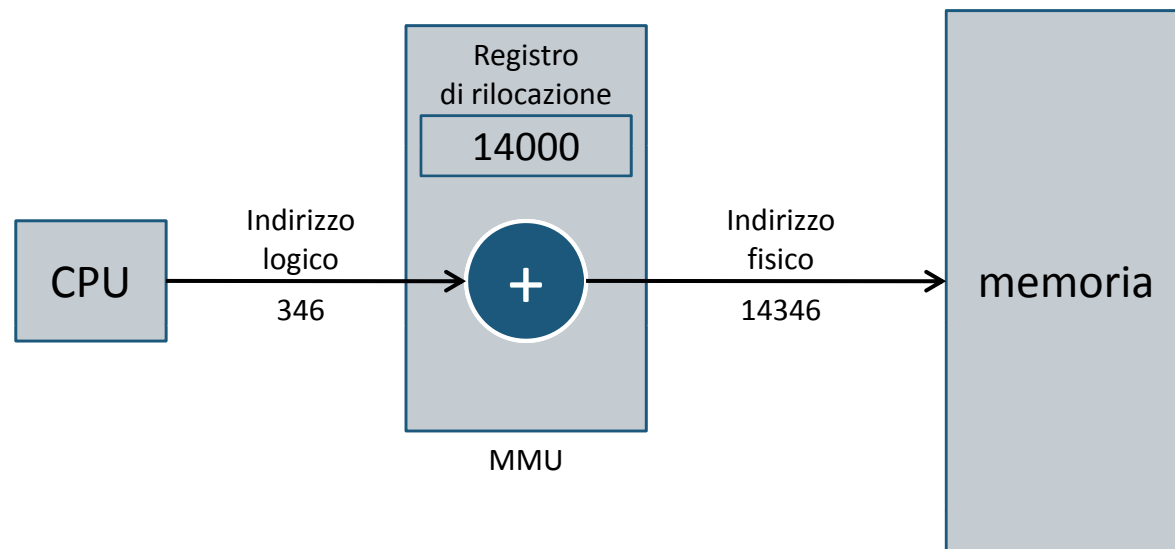


- L'associazione in fase di esecuzione tra lo spazio degli indirizzi logici e lo spazio di indirizzi virtuali è svolta da **unità di gestione della memoria** (*memory-management unit*, MMU)
- La MMU possiede un **registro di rilocazione** che permette immediatamente di convertire un indirizzo logico in indirizzo fisico
- Il programma utente tratta sempre indirizzi logici mentre l'architettura di sistema tratta indirizzi fisici
  - Esempio: se l'applicazione utilizza un range di indirizzi logici da 0 a  $max$  ed il registro di locazione contiene il valore  $r$  allora il range di indirizzi fisici sarà da  $0+r$  a  $r+max$

# Es. di rilocazione dinamica con registro di rilocazione



- Supponiamo che il registro di rilocazione contenga il valore 14000
- Qualunque tentativo da parte dell'applicazione di accedere alle locazioni di memoria 0, 346 o 1200 si traduce in
  - $0 + 14000 = 14000$
  - $346 + 14000 = 14346$



# Caricamento dinamico



- Un applicazione durante il suo ciclo di esecuzione utilizza delle procedure esterne (matematiche, database, disegno, etc...)
- Il **caricamento dinamico** permette di caricare una procedura in memoria solo quando viene richiamata
- La memoria viene utilizzata meglio poiché la procedura è caricata solo se effettivamente utilizzata
- Anche se l'applicazione è enorme la memoria occupata è effettivamente solo quella necessaria
- Il programmatore deve prestare attenzione progettando l'applicazione in modo da trarre vantaggio dal caricamento dinamico

# Librerie di procedure



- Le librerie di procedure possono essere fondamentalmente di due tipi:
  - Librerie statiche
  - Librerie dinamiche
- Il programmatore solitamente può accedervi utilizzando un file .h in C/C++

# Librerie statiche



- Le **librerie statiche** contengono procedure che vengono collegate al codice del programma principale dal compilatore o dal loader e diventano parte dell'eseguibile
- Lo svantaggio principale è che se  $n$  programmi utilizzano la stessa libreria statica questa sarà incorporata da ogni eseguibile e quindi lo stesso codice sarà caricato più volte in memoria
  - Anche l'occupazione su disco aumenta inutilmente
- Oltretutto se l'applicazione una volta mandata in esecuzione non utilizzerà nessuna procedura della libreria occuperà inutilmente spazio in memoria

# Librerie dinamiche

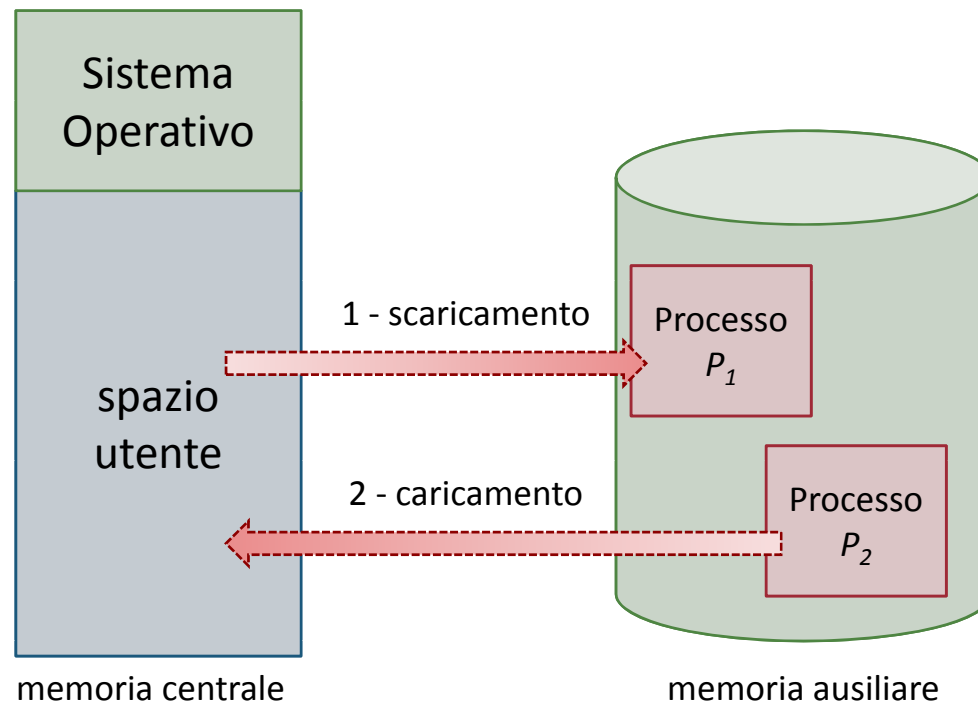


- Una **libreria dinamica** viene caricata in memoria solo quando l'applicazione chiama una delle sue procedure. La libreria è quindi caricata in memoria a run-time utilizzando un caricamento dinamico
- L'applicazione che utilizza una libreria dinamica conterrà all'interno del suo codice il nome della procedura che vuole chiamare
- Il vantaggio principale è che il codice di una libreria dinamica può essere condivisa tra più processi evitando inutili duplicazioni in memoria di codice identico
  - Sotto linux le librerie dinamiche hanno estensione .so
  - Sotto Windows le librerie dinamiche hanno la "ben nota" estensione .dll

# Avvicendamento dei processi



- L'idea dell'**avvicendamento dei processi** (o *swapping*) è salvare nella memoria secondaria un processo non in esecuzione (**swap out**) e ricaricarlo (**swap in**) appena prima di dargli la CPU



# Swapping



- Lo swapping permette di avere più processi attivi di quanti effettivamente possa contenere la RAM
- I processi attivi sono mantenuti temporaneamente su un'area del disco fisso ad uso esclusivo del Sistema Operativo chiamata **area di swap**
- Poiché un processo può essere ricaricato in una diversa area di memoria principale è necessario utilizzare codice dinamicamente rilocabile in fase di esecuzione

# Il cambio di contesto nello swapping



- La fase di context switch utilizzando lo swapping può aumentare significativamente
- Ad esempio supponiamo di avere in memoria un processo di 10MB e supponiamo che il disco fisso abbia una velocità di trasferimento di 40MB/sec
- Il tempo di trasferimento è pari a:  
$$10.000\text{KB} / 40.000\text{KB/sec} = \frac{1}{4} \text{ sec} = 250 \text{ millisecondi}$$
- A questo tempo va aggiunto il tempo per posizionare le testine (circa 8 millisecondi) quindi passiamo a:  
$$258 \text{ millisecondi}$$
- Infine va considerato che l'avvicendamento include una fase di scrittura (swap out) ed una di lettura (swap in) quindi in totale avremo:  
$$516 \text{ millisecondi}$$
- Quindi un quanto di tempo dovrebbe essere maggiore di 0,516 secondi

# Swapping

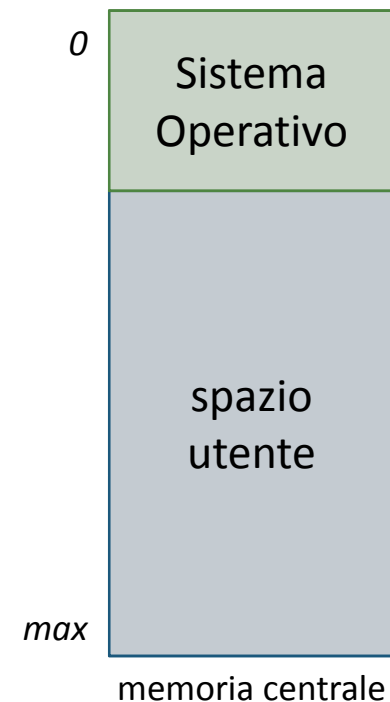


- I calcoli precedenti valgono nel caso di un processo di 10MB ma le conclusioni sono differenti se il processo occupa più memoria
- Normalmente i processi soggetti a swapping sono quelli in attesa di I/O ma...
  - Un processo  $P_1$  con I/O asincrono potrebbero avere dei problemi
  - Se il processo  $P_1$  viene scaricato dalla memoria un altro processo  $P_2$  potrebbe utilizzare la memoria di  $P_1$  dedicata alle operazioni di I/O
- L'avvicendamento dei processi nella sua forma originale è utilizzato raramente nei sistemi operativi
- Il primo sistema operativo di massa ad usare swapping è stato Windows 3.1
  - L'utente in maniera inconsapevole decideva quali applicazioni scaricare/caricare da disco

# Allocazione contigua della memoria



- La memoria centrale deve contenere sia il Sistema Operativo che i processi degli utenti
- Normalmente il Sistema Operativo è caricato nella parte bassa della memoria (a partire dall'indirizzo 0)
- Per le applicazioni che devono essere caricate in memoria è necessario adottare una politica di allocazione in memoria considerando la coda di ingresso dei processi



# Allocazione della memoria

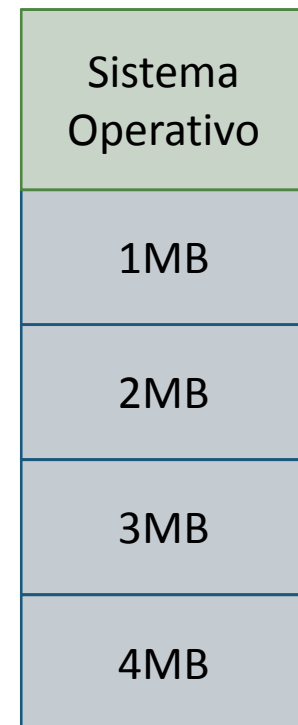


- La memoria può essere allocata utilizzando diverse strategie tra le quali le più note sono:
  - Partizioni multiple fisse
  - Partizioni multiple variabile

# Partizioni multiple fisse



- Nello schema a **partizioni multiple fisse** la memoria è suddivisa in blocchi (non necessariamente tutti uguali) di dimensione fissa
- Ogni partizione può contenere un processo
- Il numero di partizioni decide il grado di multiprogrammazione
- Al termine di un processo, la partizione è libera per contenere un altro processo



memoria centrale

# Contex switch nelle partizioni multiple fisse



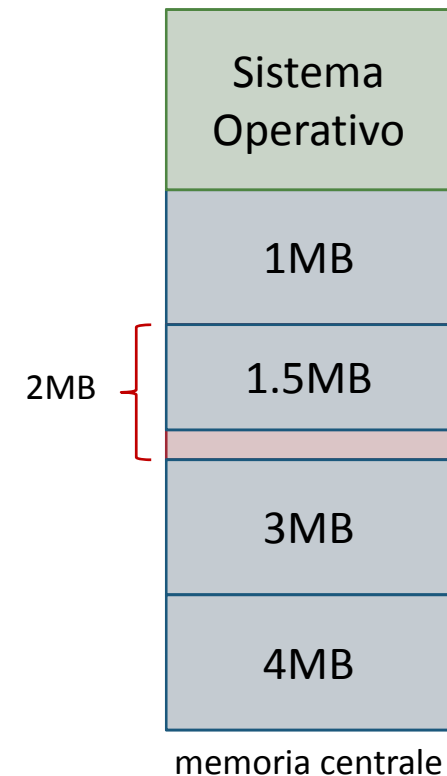
memoria centrale

- Il meccanismo dei registri limite e di rilocalizzazione può essere usato per proteggere le varie partizioni
- In fase di contex switch il Sistema Operativo carica:
  - Nel registro di rilocalizzazione l'indirizzo iniziale della partizione
  - Nel registro limite la dimensione del processo

# Frammentazione interna



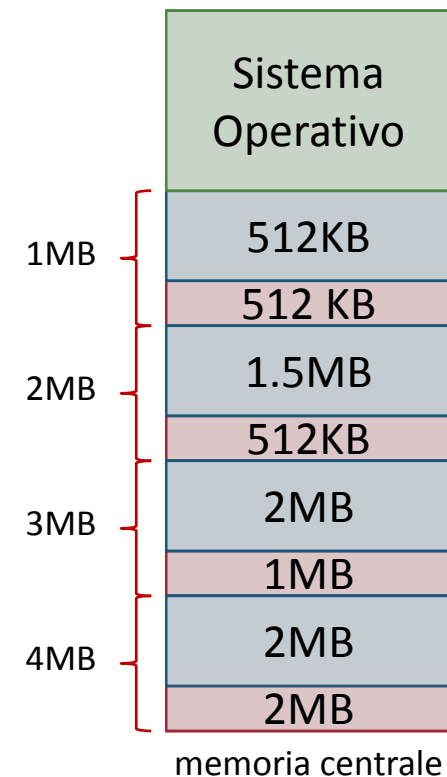
- Un processo allocato in una partizione difficilmente ha esattamente la stessa dimensione di quella partizione
- La parte che rimane viene sprecata
  - Ad esempio, nella partizione da 2Mbyte, 512KB vengono sprecati
- Questo problema è noto come **frammentazione interna**



# Frammentazione esterna



- Un altro svantaggio sta nel numero di buchi che restano liberi in ogni partizione
  - Ad esempio: allocando diversi processi in ogni partizione otteniamo uno spreco di  
 $512\text{KB} + 512\text{KB} + 1\text{MB} + 2\text{MB} = 3\text{MB}$
- Questo problema è noto come **frammentazione esterna**
- Un processo potrebbe essere all'allocato nel blocco ottenuto sommando i diversi buchi sprecati



# Svantaggi delle partizioni multiple fisse

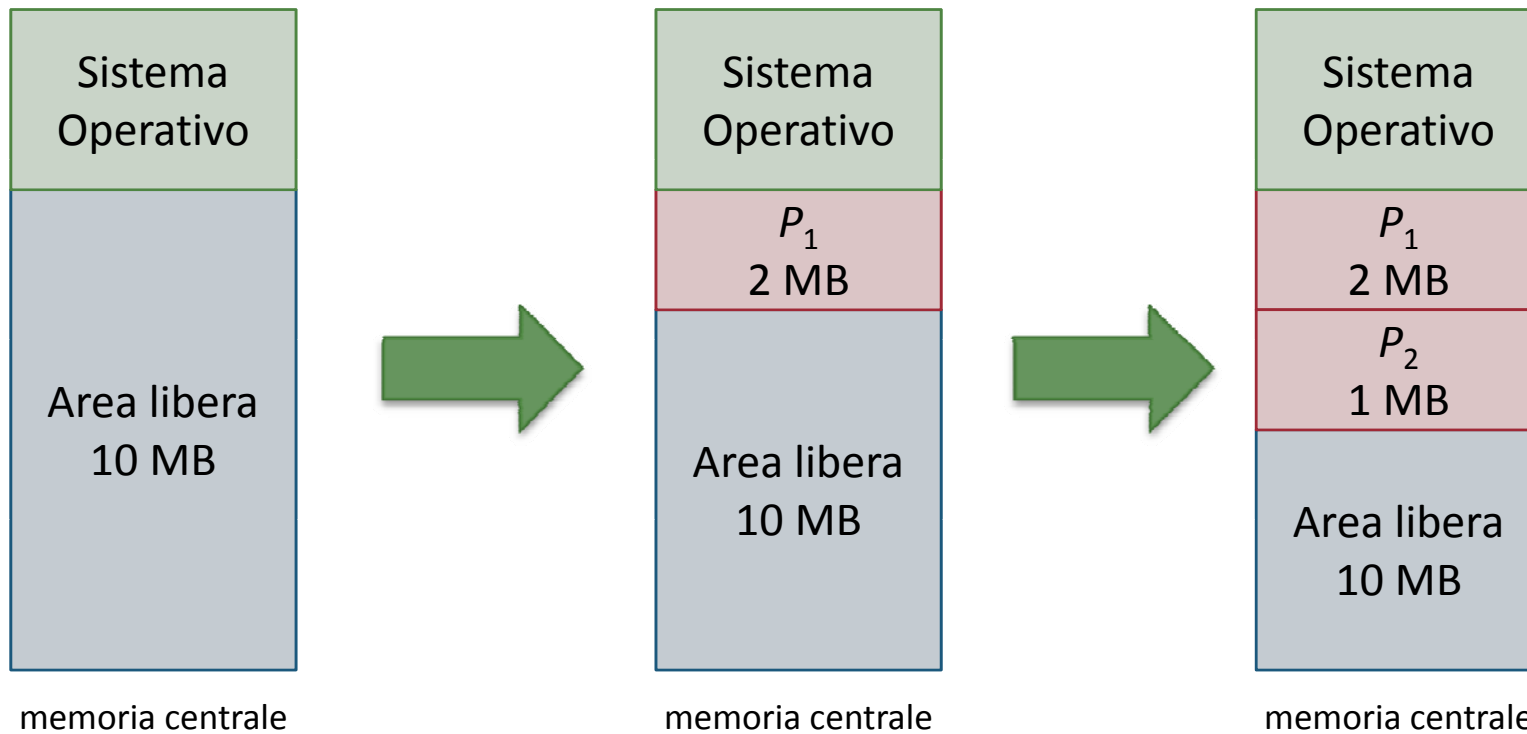


- **Frammentazione interna**
  - In un blocco di dimensione fissa la memoria interna ad un partizione è assegnata ma non utilizzata completamente
- **Frammentazione esterna**
  - Lo spazio di memoria lasciato dai buchi è sufficiente per contenere un altro processo ma non è contigua
- **Se un processo è grande ma non abbastanza da essere contenuto nella partizione più grande non può essere caricato**
  - Aumentando la dimensione delle partizioni diminuisce il grado di multiprogrammazione ma tende anche ad aumentare la frammentazione interna

# Partizioni multiple variabili



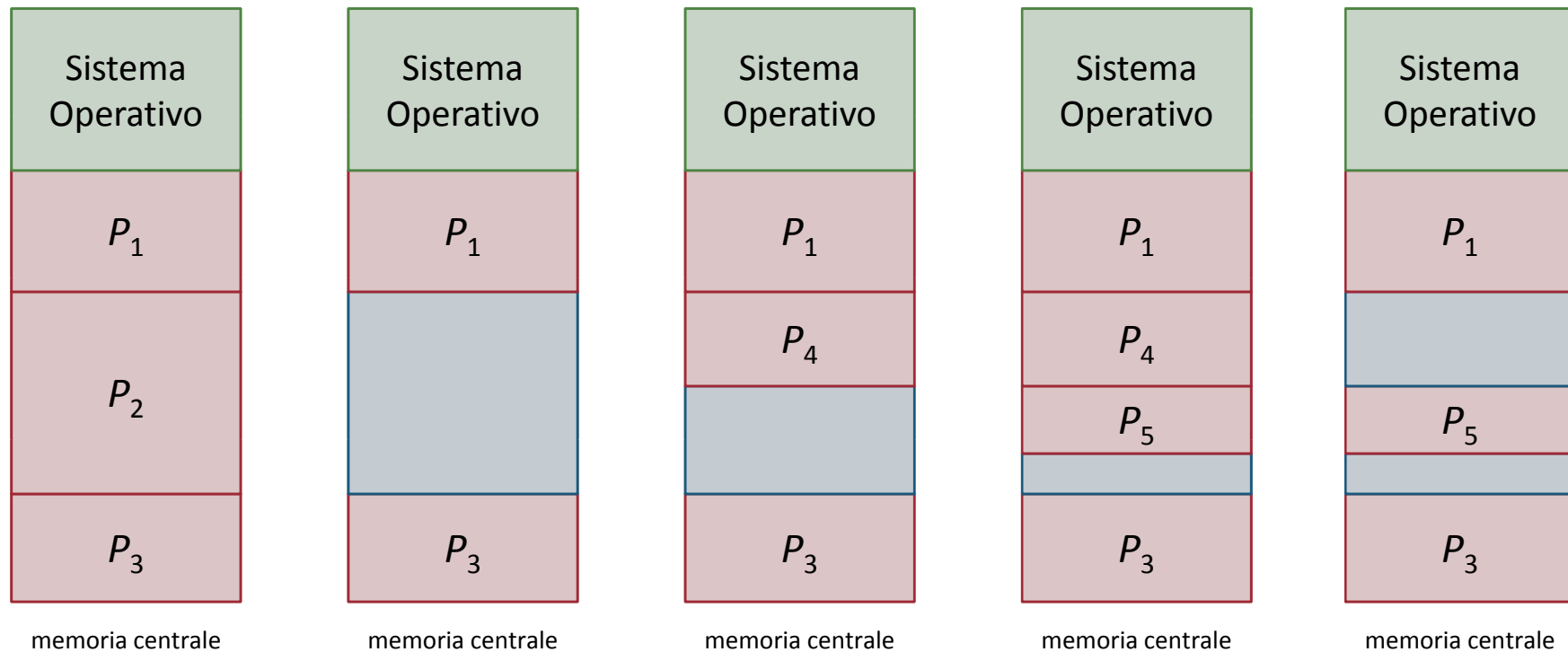
- Nella schema a **partizioni multiple variabili** un processo riceve un blocco di memoria pari alla sua dimensione
  - Non è più presente la frammentazione interna



# Partizioni multiple variabili: creazioni di buchi



- Dopo che diversi processi si sono avvicendati in memoria vengono lasciati diversi buchi
  - I buchi con il tempo diventano sempre più sparsi e non utilizzabili



# Metodi di allocazione



- Il Sistema Operativo tiene traccia di tutti i buchi liberi e della loro dimensione
- Quando un processo deve essere caricato in memoria il Sistema Operativo deve cercare un buco sufficientemente grande per contenerlo
- Le strategie per scegliere un buco libero sono:
  - **First-fit** : scegli *la prima partizione abbastanza grande per poter caricare il processo*
  - **Best-fit**: scegli *la più piccola partizione abbastanza grande per poter caricare il processo*
  - **Worst-fit**: scegli *la partizione più grande*
- Sperimentalmente il metodo migliore di allocazione risulta essere il First-fit ed il Best-fit

# Svantaggi delle partizioni multiple variabili



- **Frammentazione esterna**
  - Con il tempo si formano diversi buchi non contigui che non sono in grado di ospitare un processo
  - La regola del 50 per cento stabilisce che per  $n$  blocchi allocati  $0,5n$  blocchi sono persi nella frammentazione
- **Frammentazione interna**
  - Il carico necessario per tenere traccia dei buchi lasciati liberi è più grande del buco stesso

# Compattazione



- La **compattazione** è la soluzione che si adopera per recuperare i buchi di memoria inutilizzata
- L'idea è di spostare le immagini dei processi in maniera da ricavare un buco di memoria contigua sufficientemente grande da permettere di caricare altri processi
- Per poter utilizzare la compactazione è necessario che i processi siano rilocabili
- Essendo una operazione onerosa che può richiedere molto tempo durante la compactazione il sistema è inutilizzabile

# Strategie di compattazione



- La compattazione può essere effettuata in diversi modi:
  - Spostare tutti i processi verso un estremo della memoria (molto costoso)
  - Spostare i processi in buchi già esistenti verso gli estremi della memoria (statisticamente si sposta meno memoria)
  - Spostare solo i processi necessari per far entrare un nuovo processo che altrimenti non avrebbe spazio sufficiente

# Esempi di compattazione

