

# Sistemi Operativi II

Corso di Laurea in Ingegneria Informatica

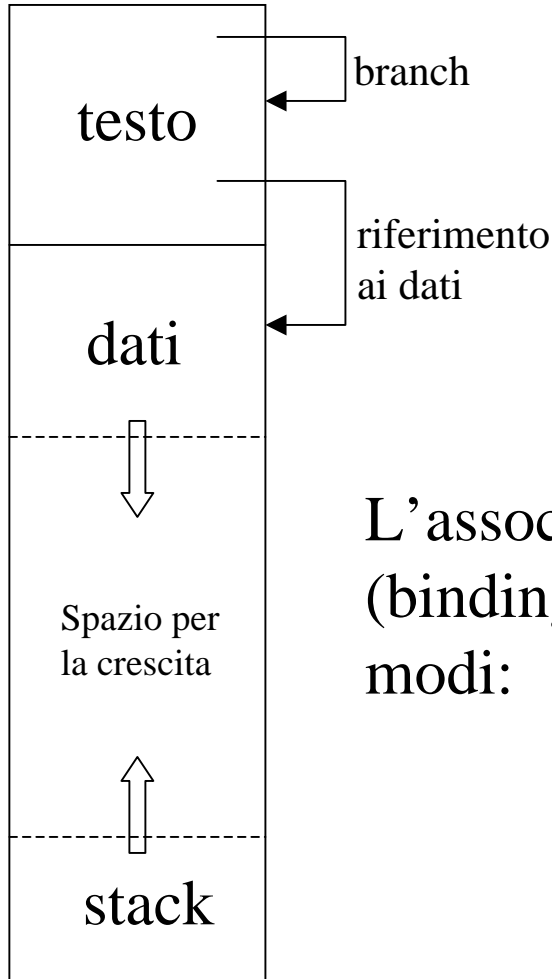
Facolta' di Ingegneria, Universita' "La Sapienza"

Docente: Francesco Quaglia

## **Gestione della memoria:**

1. Binding degli indirizzi
2. Partizionamento statico e dinamico
3. Allocazione non contigua: paginazione e segmentazione
4. Memoria virtuale
5. Gestione della memoria in sistemi operativi attuali  
(NT/UNIX)

# Immagine di memoria e riferimenti

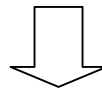


Il testo contiene riferimenti a indirizzi di memoria associati a

1. istruzioni (salti condizionati e non, chiamate di subroutine)
2. dati (indirizzamento assoluto etc..)

L'associazione di istruzioni e dati ad indirizzi di memoria (binding dei riferimenti) può essere effettuata nei seguenti modi:

- a tempo di compilazione
- a tempo di caricamento
- a tempo di esecuzione



Il tipo di binding dipende/condiziona dalla/la modalità di assegnazione della memoria di lavoro ai processi

# Caratteristiche del binding

## Binding a tempo di compilazione

- la posizione in memoria del processo è fissa e nota solo al tempo della compilazione
- il codice è assoluto, ogni riferimento è risolto tramite il corrispondente indirizzo di istruzione o dato
- se la locazione in cui il processo è caricato in memoria dovesse essere modificata sarebbe necessaria la ricompilazione

## Binding a tempo di caricamento

- la posizione in memoria del processo è fissa e nota solo all'atto del lancio della relativa applicazione, non al tempo della compilazione
- il codice è rilocabile, ogni riferimento è risolto tramite un offset a partire dall'inizio dell'immagine del processo
- la base per l'offset è determinata al tempo del lancio

## Binding a tempo di esecuzione

- la posizione in memoria del processo può variare durante l'esecuzione
- un riferimento viene risolto in un indirizzo di memoria solo se richiesto durante l'esecuzione

# Indirizzi logici e fisici

## Indirizzo logico

- riferimento usato nel testo del programma per localizzare istruzioni e/o dati
- indirizzi logici sono generati dalla CPU durante l'esecuzione del programma

## Indirizzo fisico

- posizione reale in memoria di istruzioni e/o dati
- 

Per il binding a tempo di compilazione ed a tempo di caricamento

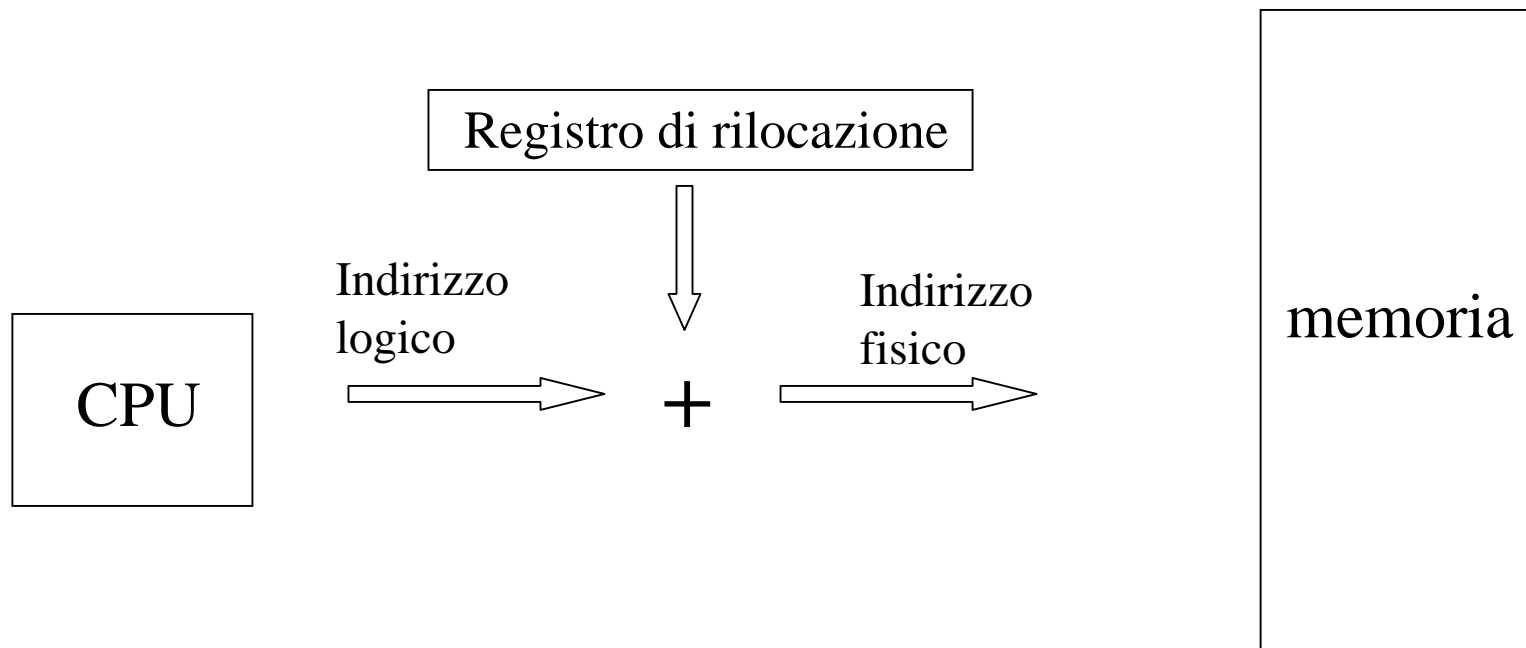
- gli indirizzi logici e fisici coincidono
- ogni indirizzo generato dalla CPU viene direttamente caricato nel memory-address-register

Per il binding a tempo di esecuzione

- gli indirizzi logici e fisici possono non coincidere
- il mapping run-time degli indirizzi logici su indirizzi fisici avviene tramite un apposito dispositivo detto Memory Management Unit (MMU)

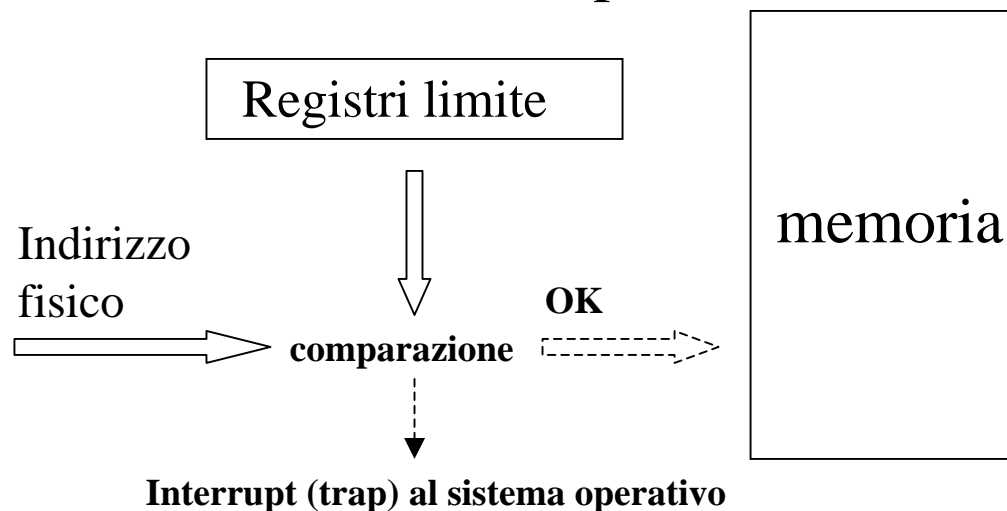
# MMU

- indirizzi logici espressi come offset dall' inizio dell'immagine di memoria
- viene utilizzato un registro di rilocalizzazione caricato con l'indirizzo iniziale della posizione corrente del processo
- il valore del registro di rilocalizzazione indica la base per l'offset



# Protezione

- ogni processo deve essere protetto contro interferenze di altri processi, siano esse accidentali o intenzionali
- i riferimenti di memoria generati da un processo devono essere controllati per accertarsi che cadano nella regione di memoria realmente riservata a quel processo
- il controllo va effettuato run-time poichè la maggioranza dei linguaggi di programmazione supporta il calcolo degli indirizzi tempo di esecuzione (vedi indici di array o puntatori a strutture dati)
- il controllo avviene via hardware per motivi di efficienza

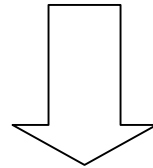


# Partizionamento statico a partizioni multiple (sistemi multiprogrammati)

- spazio di indirizzamento fisico suddiviso in partizioni fisse, di taglia uguale o diversa
- ogni partizione è dedicata ad ospitare un singolo processo

## Problemi

- ogni processo occupa una intera partizione indipendentemente dalla sua taglia (**frammentazione interna**)
- un programma potrebbe essere troppo grande per essere contenuto in una singola partizione

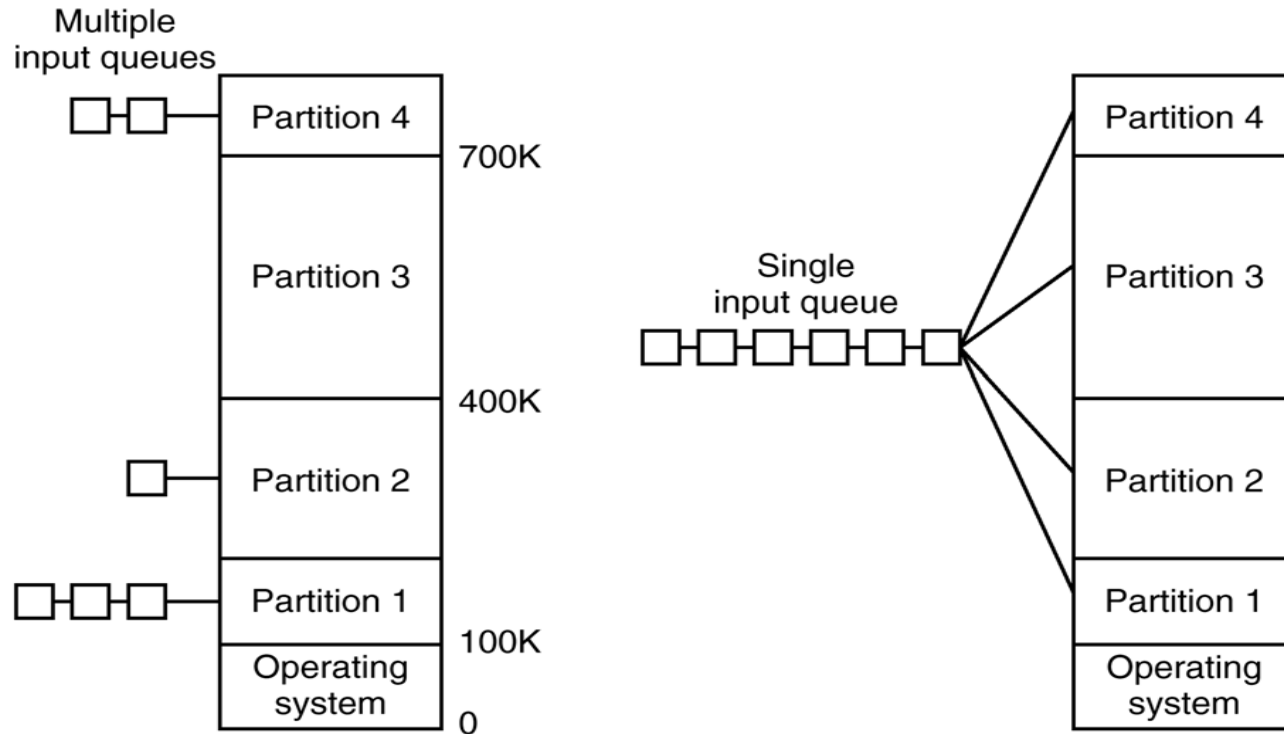


Entrambi i problemi possono essere alleviati nel caso di partizioni di taglia diversa (molte di taglia piccola, poche di taglia grande)

---

Grado di multiprogrammazione limitato dal numero di partizioni

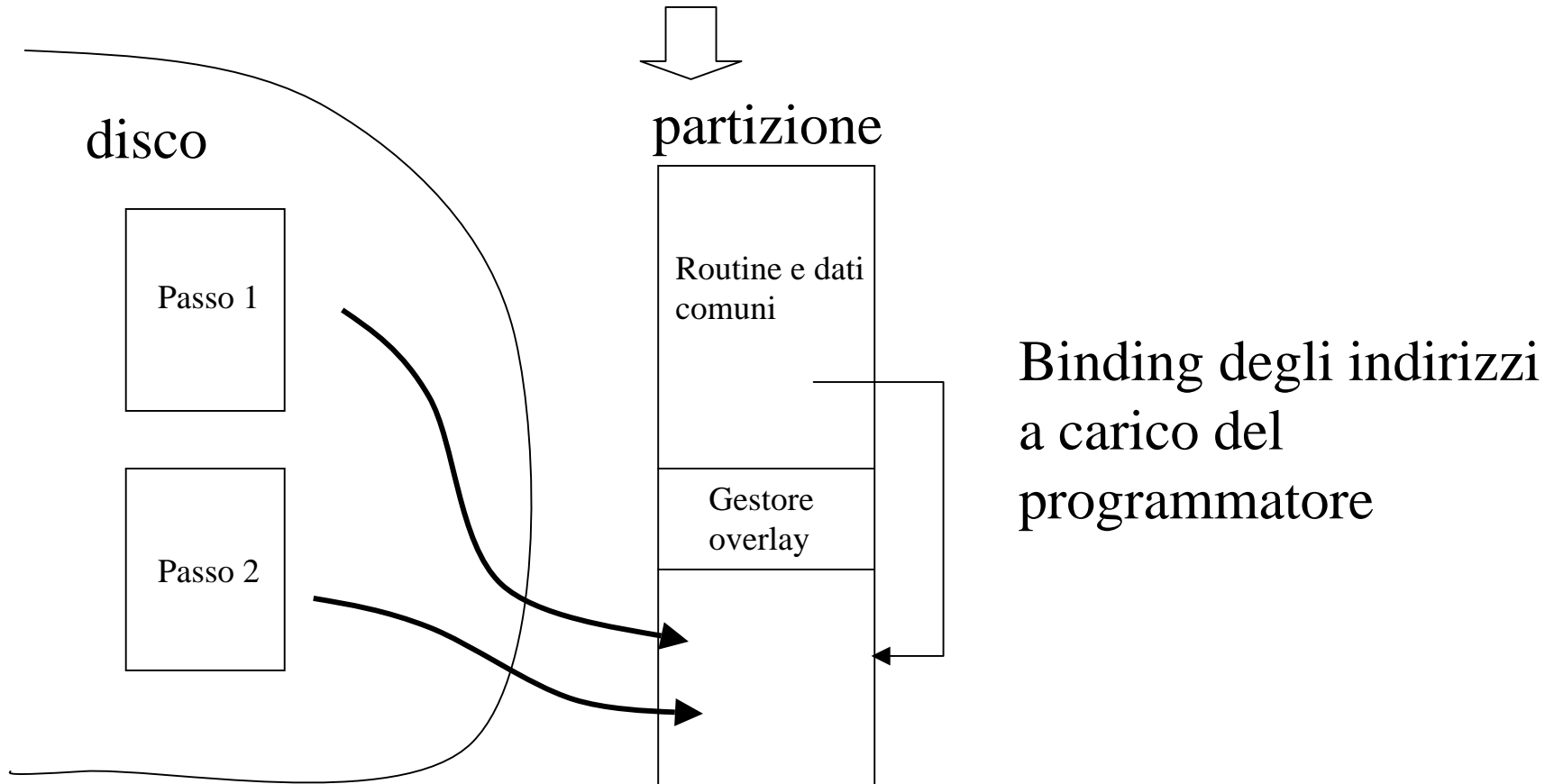
# Allocazione dei processi



- code multiple riducono il problema della frammentazione interna in caso i processi siano assegnati alle code in funzione della loro taglia
- coda singola riduce la probabilità di avere partizioni non utilizzate (**frammentazione esterna**)
- protezione: i registri limite operano sulle singole partizioni

# Overlay

- processi con taglia superiore a quella della massima partizione non possono essere caricati dal sistema operativo
- al programmatore è lasciato il compito di gestire tale situazione

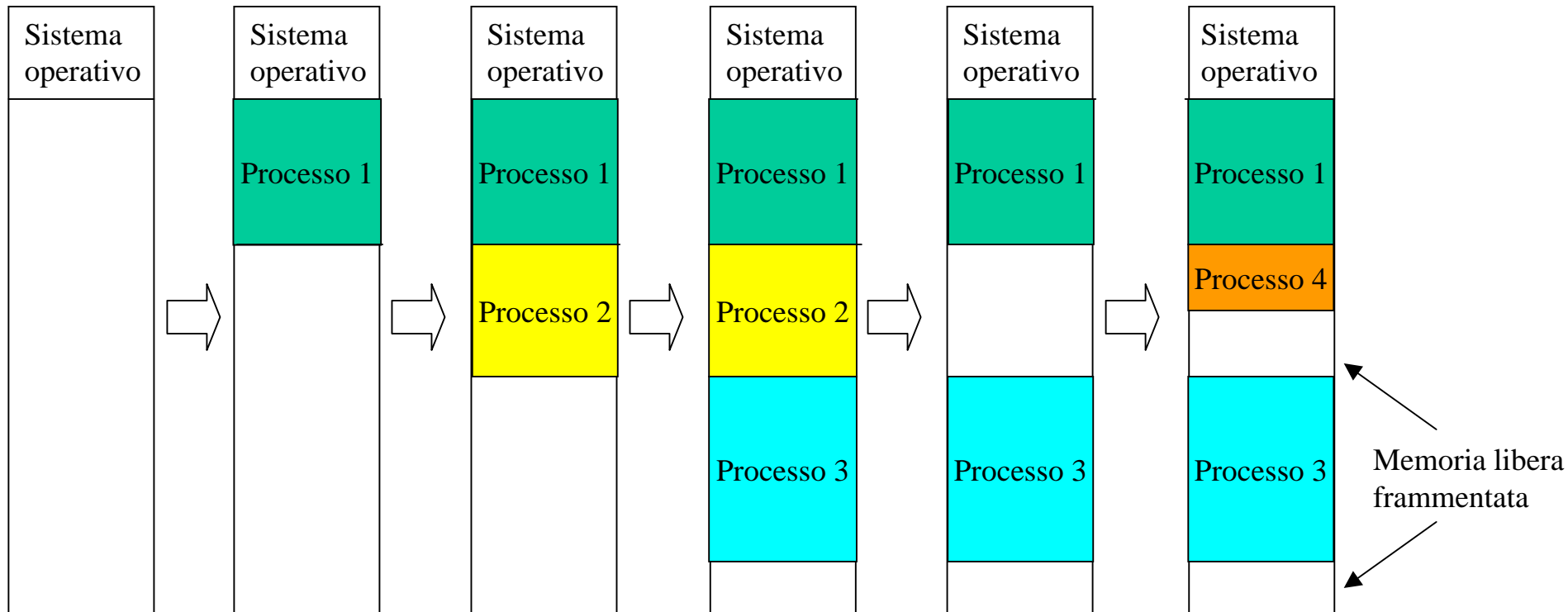


routine e dati comuni + gestore overlay + passo 1 = **overlay A**

routine e dati comuni + gestore overlay + passo 2 = **overlay B**

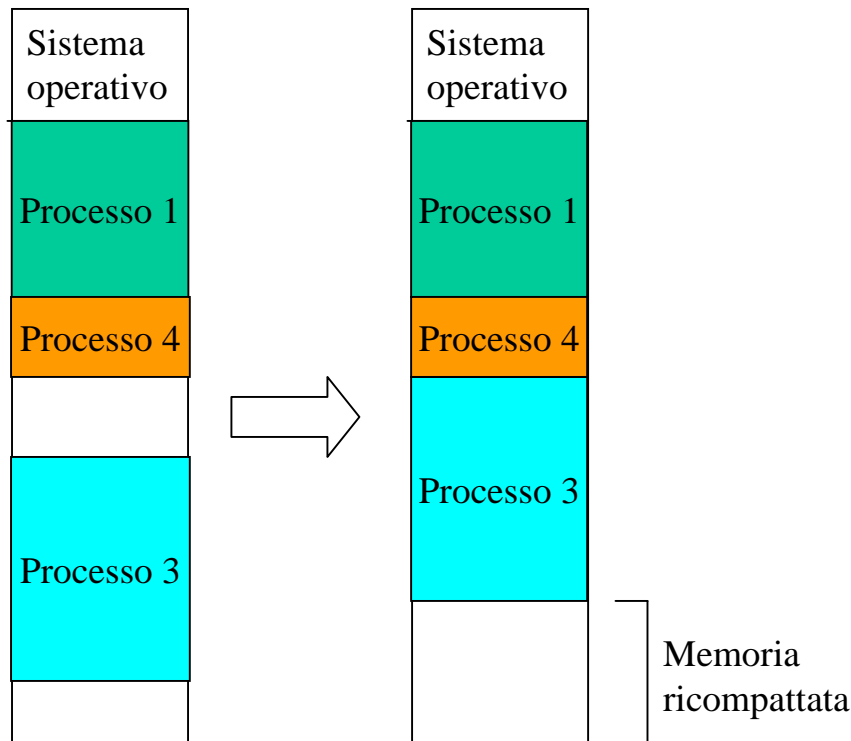
# Partizioni dinamiche

- si usano partizioni in numero e lunghezza variabile
- quando un processo è caricato in memoria principale, gli viene allocata tanta memoria quanta richiesta e mai di più'
- la memoria allocata per un processo costituisce una nuova partizione
- quando in processo libera la memoria, una partizione di taglia pari a quella del processo viene resa nuovamente disponibile



# Frammentazione esterna e ricompattazione

- il partizionamento dinamico è soggetto a frammentazione esterna (la memoria esterna alle partizioni diviene sempre più frammentata, e quindi meno utilizzabile)
- per ovviare a tale problema il sistema operativo sposta periodicamente i processi nella memoria in modo da ricompattare le aree di memoria libere



## Vincolo

- supporti per il binding a tempo di esecuzione

## Svantaggi

- costo elevato

# Allocazione dei processi

## First fit

- Il processo viene allocato nel primo “buco” disponibile sufficientemente grande
- La ricerca può iniziare sia dall’inizio dell’insieme dei buchi liberi che dal punto in cui era terminata la ricerca precedente

## Best fit

- Il processo viene allocato nel buco più piccolo che può accoglierlo
- La ricerca deve essere effettuata su tutto l’insieme dei buchi disponibili, a meno che questo non sia ordinato in base alla taglia dei buchi
- Questa strategia tende a produrre frammenti di memoria di dimensioni minori, lasciando non frammentate parti di porzioni di memoria più grandi

## Worst fit

- Il processo viene allocato nel buco più grande in grado di accoglierlo
- Anche in questo caso la ricerca deve essere effettuata su tutto l’insieme dei buchi disponibili, a meno che questo non sia ordinato in base alla taglia
- Questa strategia tende a produrre frammenti relativamente grandi, utili quindi ad accogliere una quantità di nuovi processi di taglia ragionevole

# Swapping

- processi in stato di blocco (attesa di evento) possono essere riversati fuori della memoria (**Swap-out**) per far posto ad altri processi
- lo swapping è in genere utilizzato in caso il sistema supporti binding a tempo di esecuzione (flessibilità sullo **Swap-in** di processi precedentemente riversati fuori dalla memoria)
- in caso di binding a tempo di compilazione o caricamento, lo Swap-in deve afferire alla stessa partizione di Swap-out

## Vincoli

- per poter effettuare Swap-out è necessario che ci sia completa inattività sullo spazio di indirizzamnto (I/O asincrono non permesso)
- i buffer per l'I/O asincrono vengono quindi allocati nella memoria riservata per il sistema operativo

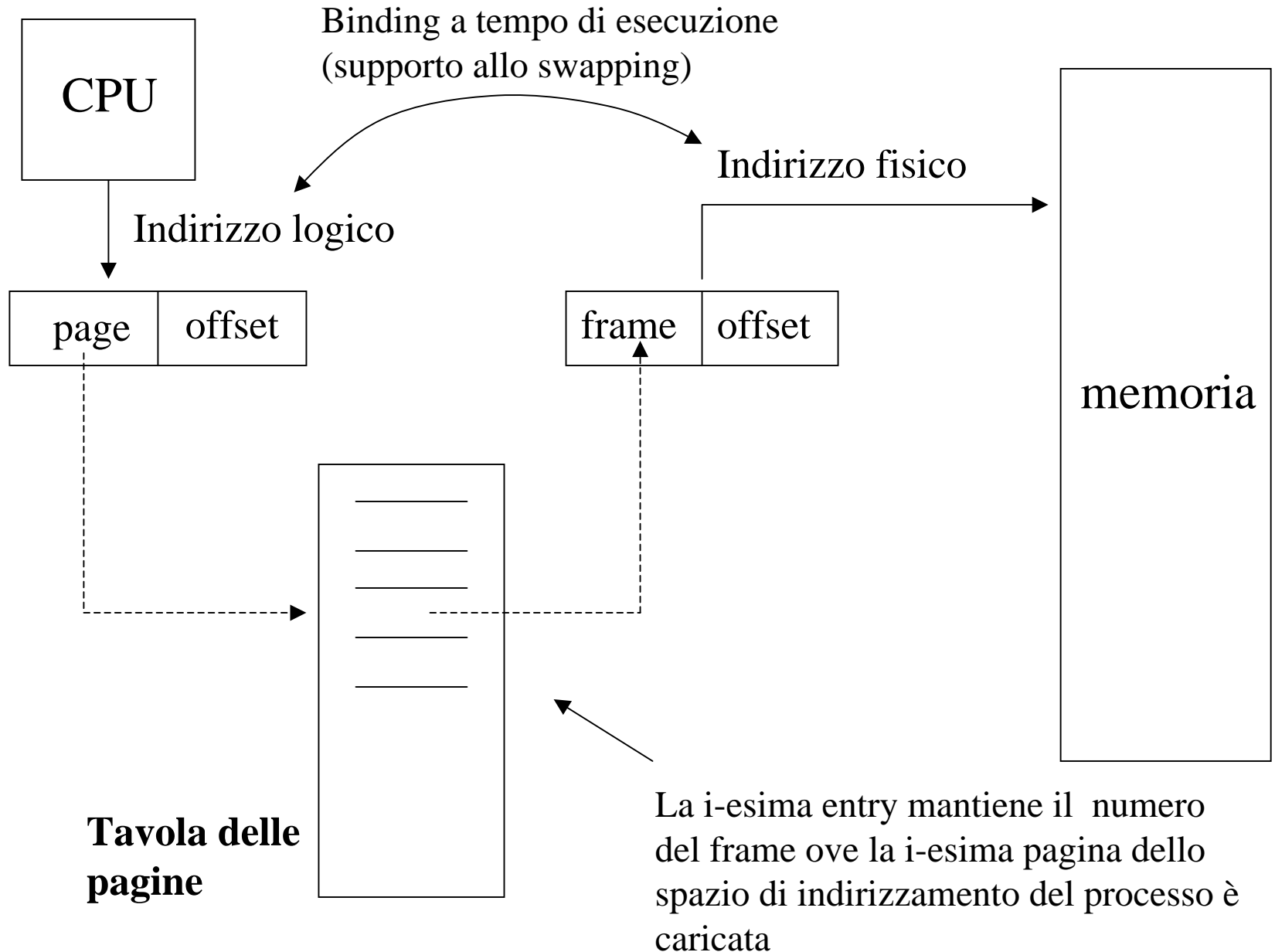
# Paginazione

- lo spazio di indirizzamento di un processo viene visto come un insieme di pagine di taglia fissa  $X$  bytes
- la memoria di lavoro viene vista come partizionata in un insieme di pagine della stessa taglia ( $X$  bytes), denominate frames
- ogni pagina dello spazio di indirizzamento viene caricata in uno specifico frame di memoria
- i frame allocati per un dato processo possono anche essere non contigui (allocazione di processo non contigua in memoria)
- la frammentazione interna alle pagine è in media di  $X/2$  bytes per ogni processo attivo

## Vantaggio

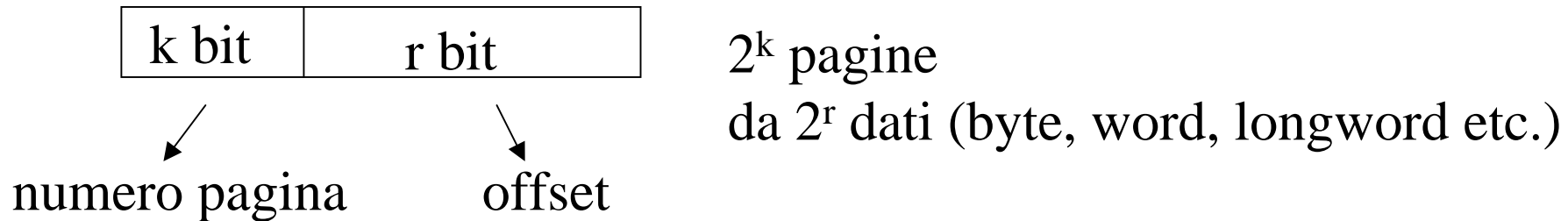
- l'allocazione non contigua aumenta la flessibilità di allocazione dei processi in memoria
- con  $N$  bytes liberi in memoria (distribuiti su  $\lceil N/X \rceil$  frames) è sempre possibile caricare in memoria un processo con taglia fino ad  $N$  byte

# Supporti per la paginazione: tavola delle pagine



# Struttura degli indirizzi

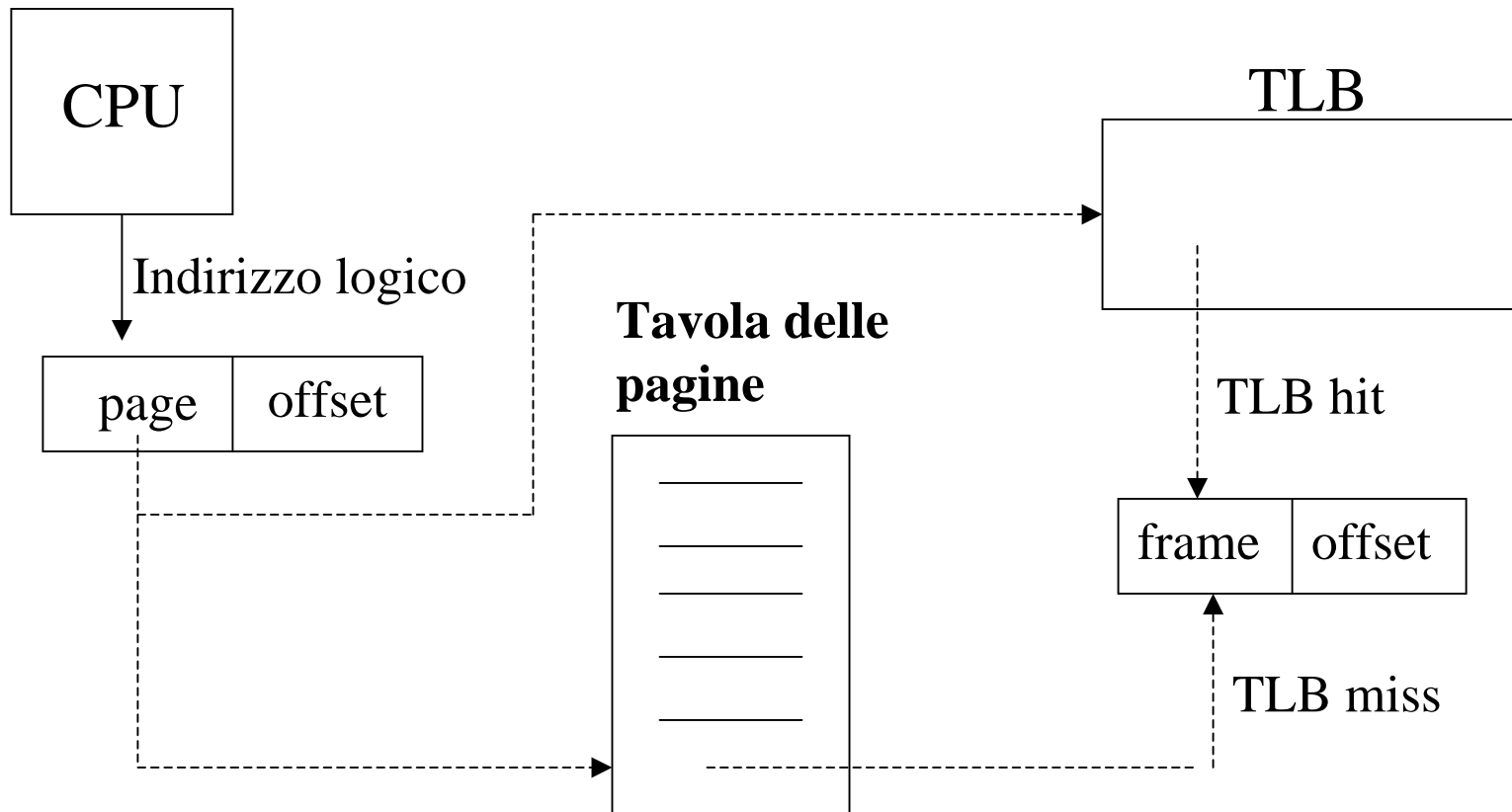
Indirizzi logici a due livelli



- l'indirizzo logico coincide con un indirizzo relativo
- il binding mantiene l'offset di pagina
- la protezione della memoria è garantita dal fatto che l'offset permette di muoversi solo ed esclusivamente all'interno del frame allocato per quella data pagina (frames destinati a pagine di altri processi non vengono coinvolti nell'accesso)

# TLB

- Il sistema operativo mantiene le tabelle delle pagine via software
- Per accelerare il binding viene utilizzato un traduttore hardware tra numero di pagina e frame denominato TLB (Translation-Lookaside-Buffer)
- Il TLB è una cache che mantiene associazioni tra numero di pagina e frame per il processo correntemente in esecuzione

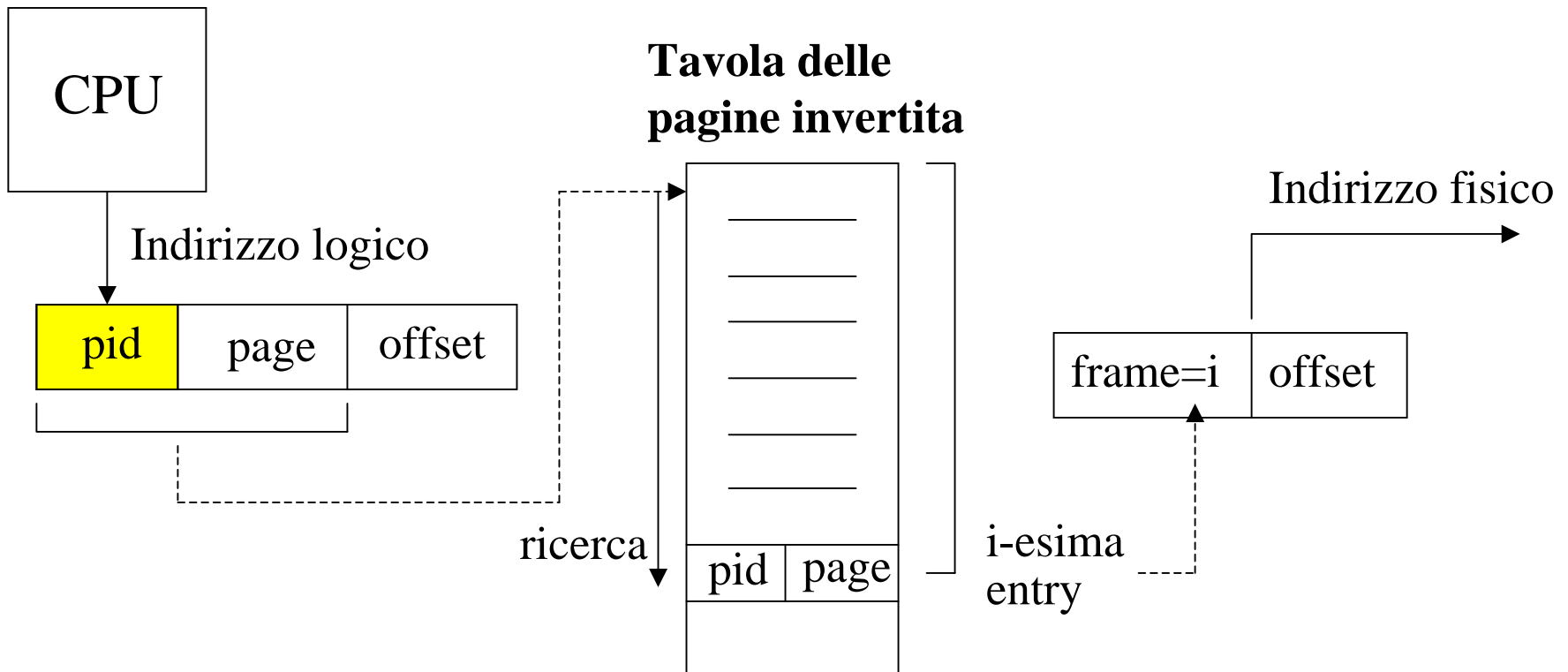


# Paginazione e dipendenza dall'hardware

- Mantenimento della tabella delle pagine via software non significa indipendenza dalla specifica architettura hardware
- La risoluzione di un miss sul TLB viene infatti effettuata dal microcodice della specifica CPU
- Questo effettua uno o più accessi alla memoria di lavoro per consultare la tabella delle pagine del processo corrente
- La struttura della tabella delle pagine (e delle relative informazioni) dipende quindi dalla logica del microcodice
- **NOTA BENE:**
  - dato che mantenere la tabella delle pagine via software significa che essa è accessibile tramite indirizzi logici, sarà necessario avere il supporto hardware per mantenere ad ogni istante la corrispondente traduzione fisica
  - questo supporto è tipicamente esterno al TLB (ad esempio su i386 è un registro dedicato denominato CR3)

# Tavola delle pagine invertita

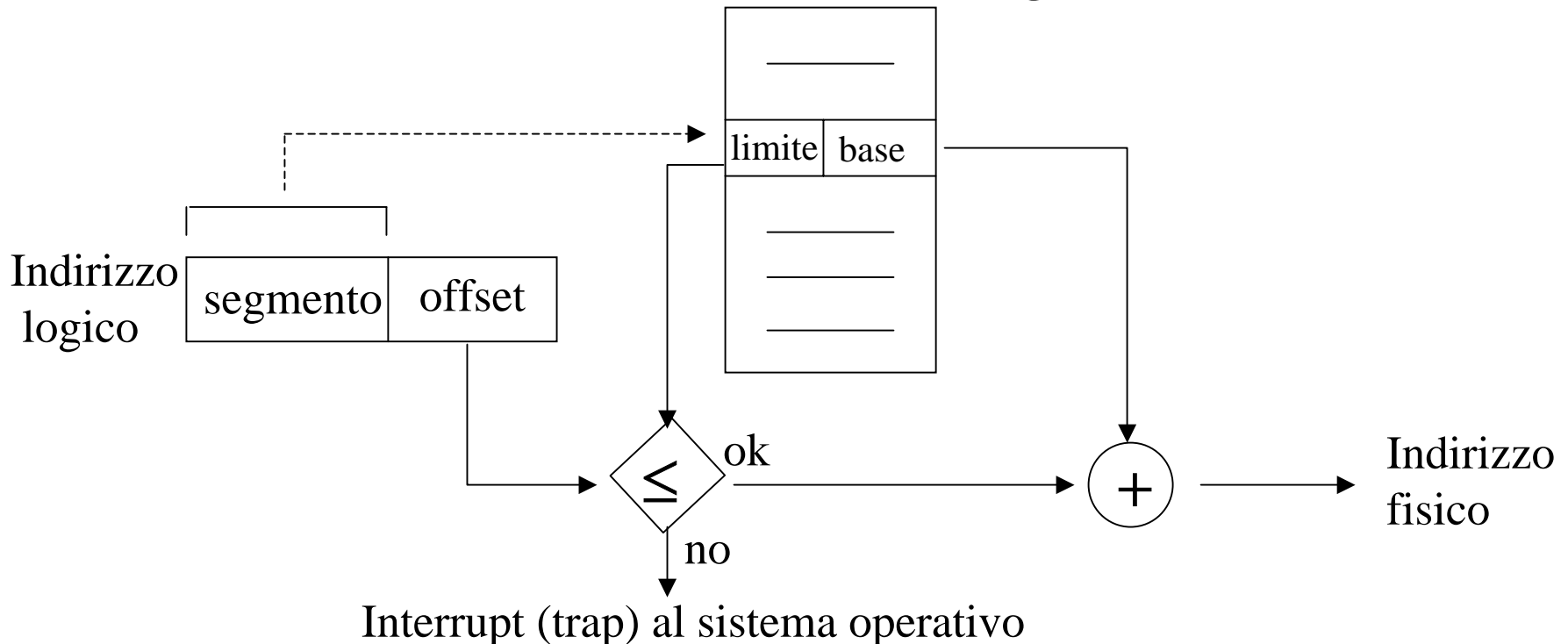
- le tabelle delle pagine vengono mantenute in memoria centrale dal sistema operativo
- in caso di spazio di indirizzi logici molto ampio, l'occupazione di memoria dovuta alle tabelle delle pagine può essere ridotta tramite la tabella delle pagine invertita, che è unica e mantiene una entry per ciascun frame di memoria



# Segmentazione

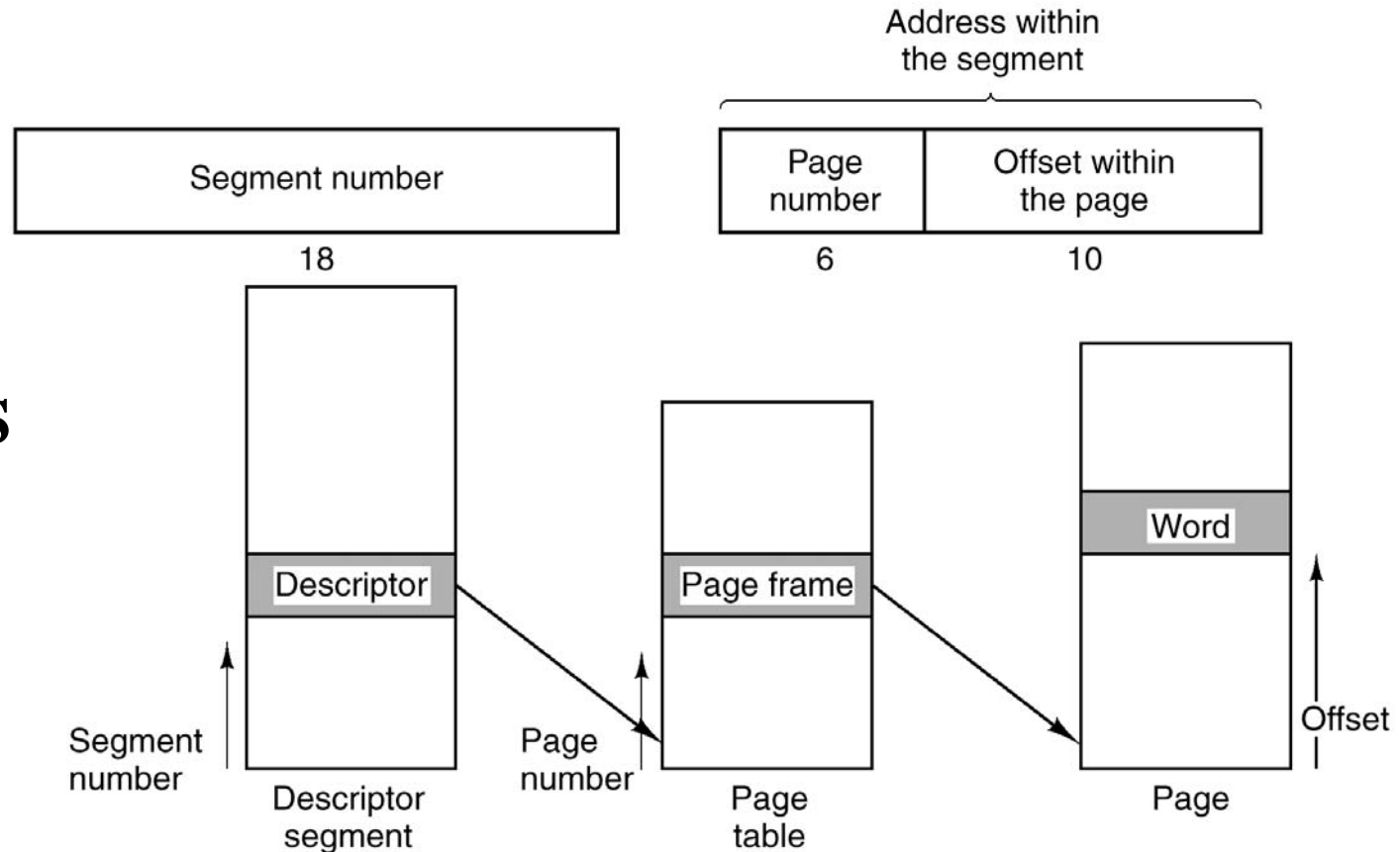
- lo spazio di indirizzamento viene visto come un insieme di segmenti distinti (es. testo, stack, dati globali)
- indirizzi logici sono formati da numero di segmento e spiazzamento all'interno del segmento
- i segmenti possono essere caricati in partizioni non contigue
- in caso di partizionamento statico, la segmentazione può permettere il caricamento di un processo di taglia superiore a quella della massima partizione (in ogni caso non della memoria fisica)

## Tavola dei descrittori segmenti



# Segmentazione e paginazione

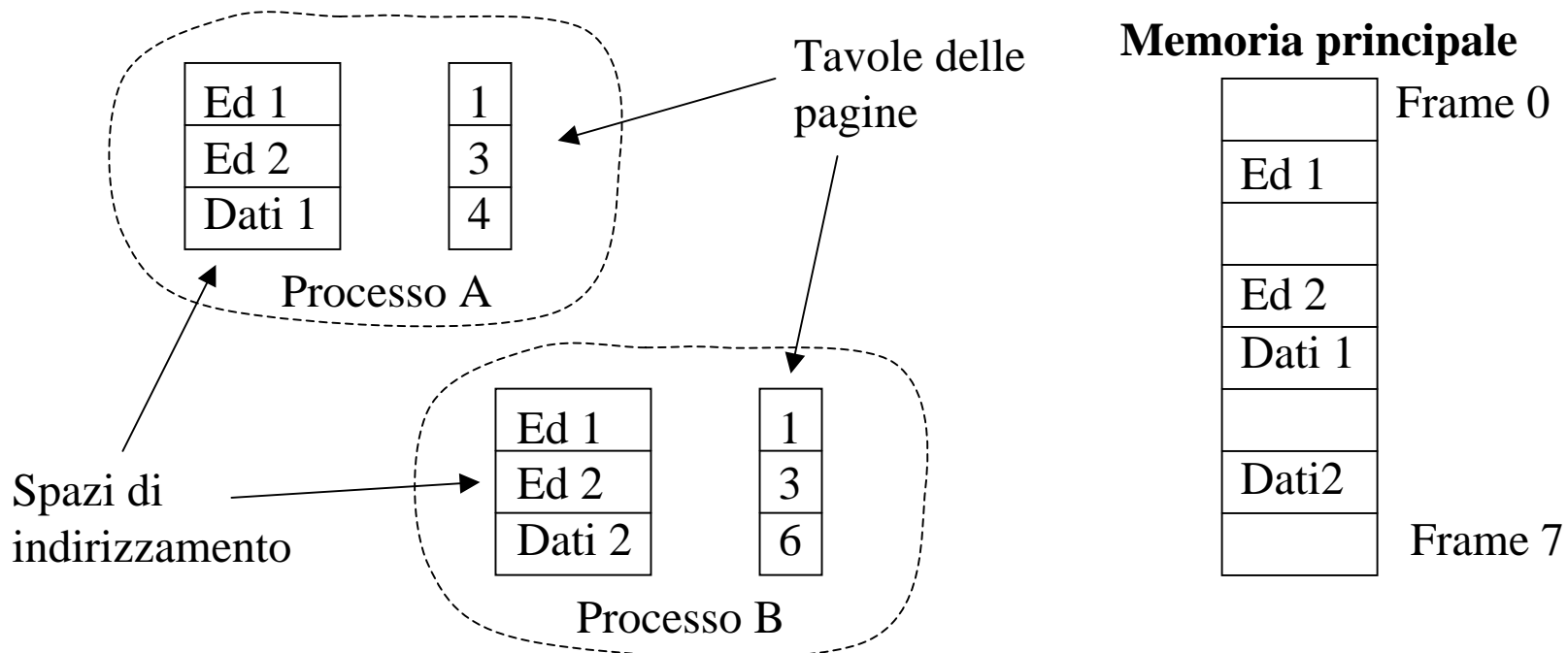
- la segmentazione, a differenza della paginazione, può aiutare il programmatore nell'organizzazione del software (modularizzazione)
- la segmentazione presenta il problema della frammentazione esterna, che può essere ridotto tramite segmentazione paginata



**Es. MULTICS**

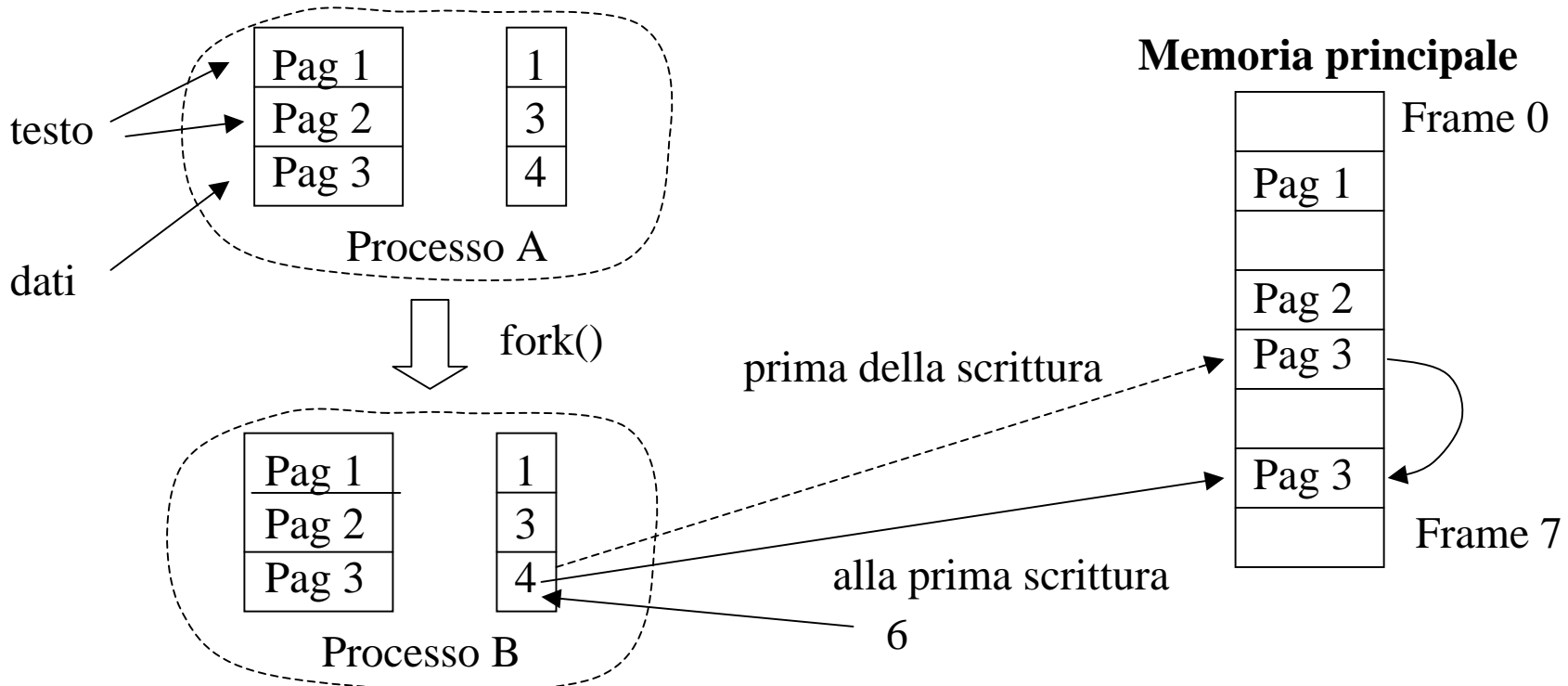
# Condivisione

- ulteriori vantaggi della paginazione e della segmentazione consistono nella possibilità di condividere codice di uso comune (es. text editors)
- ciascun processo possiede dati privati, caricati su frame o partizioni di memoria proprie
- il testo viene invece caricato su pagine o partizioni comuni
- il binding a tempo di esecuzione supporta la condivisione in modo automatico senza costi aggiuntivi



# Controllo sulla modalità di accesso

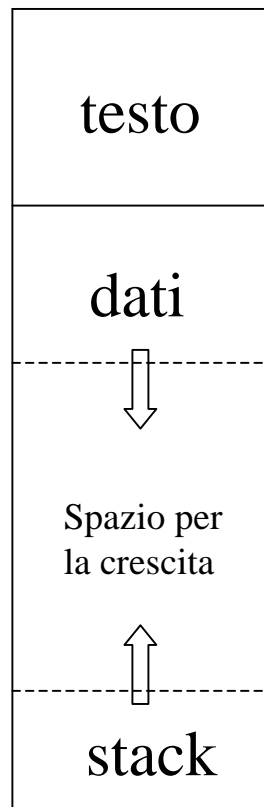
- singole pagine o segmenti possono essere protetti rispetto a specifiche modalità di accesso (lettura/scrittura/esecuzione)
- la tavola delle pagine contiene bit addizionali per determinare il tipo di protezione
- protezione **copy on write**: la pagina o il segmento è condiviso fino alla prima scrittura (supporto efficiente per i meccanismi duplicazione di processo, es. *fork()* UNIX)



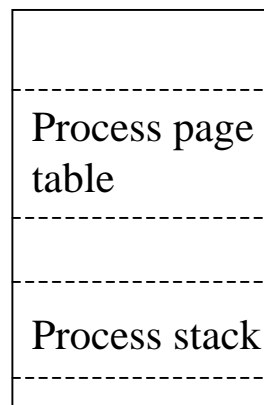
# Condivisione e moduli kernel

- la condivisione tramite segmentazione e paginazione e' anche un modo per rendere accessibile a processi distinti l'unica istanza di codice/dati del kernel
- il kernel e' infatti tipicamente raggiungibile in un determinato set di indirizzi logici
- questi vengono "mappati" su indirizzi fisici identici per tutti i processi attivi in modo da ottenere una visione univoca e coerente dello stato del kernel stesso

# Tipico layout per sistemi UNIX



- **User level** – indirizzi logici da 0 a 3 GB
- Condivisione governata dallo schema “copy on write” (possibilità di istanze multiple in memoria di lavoro)

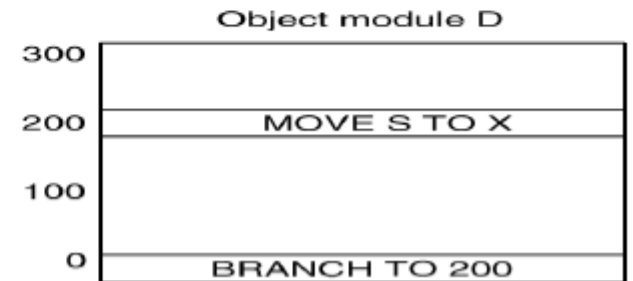
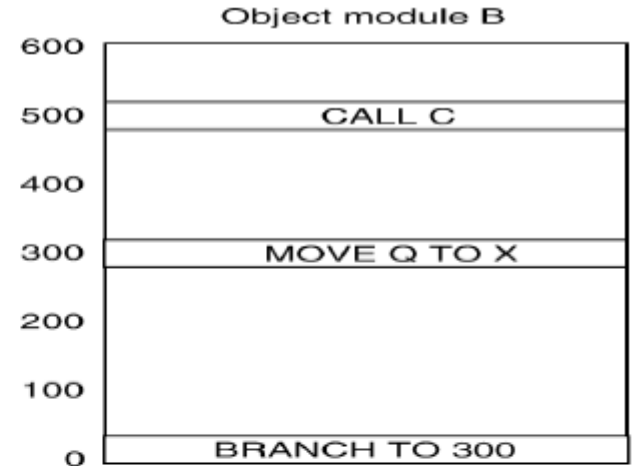
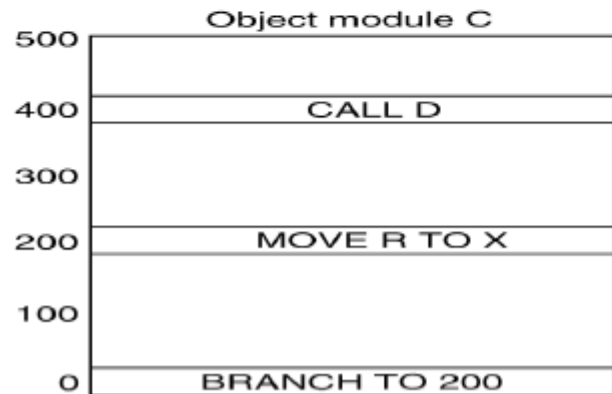
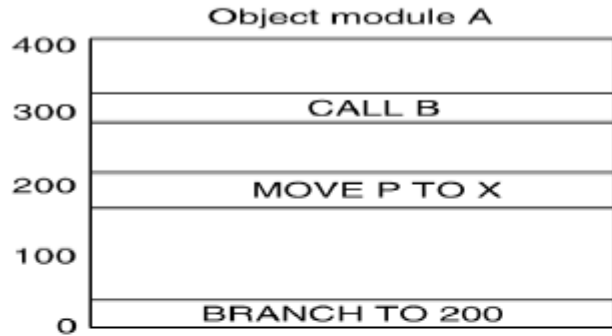


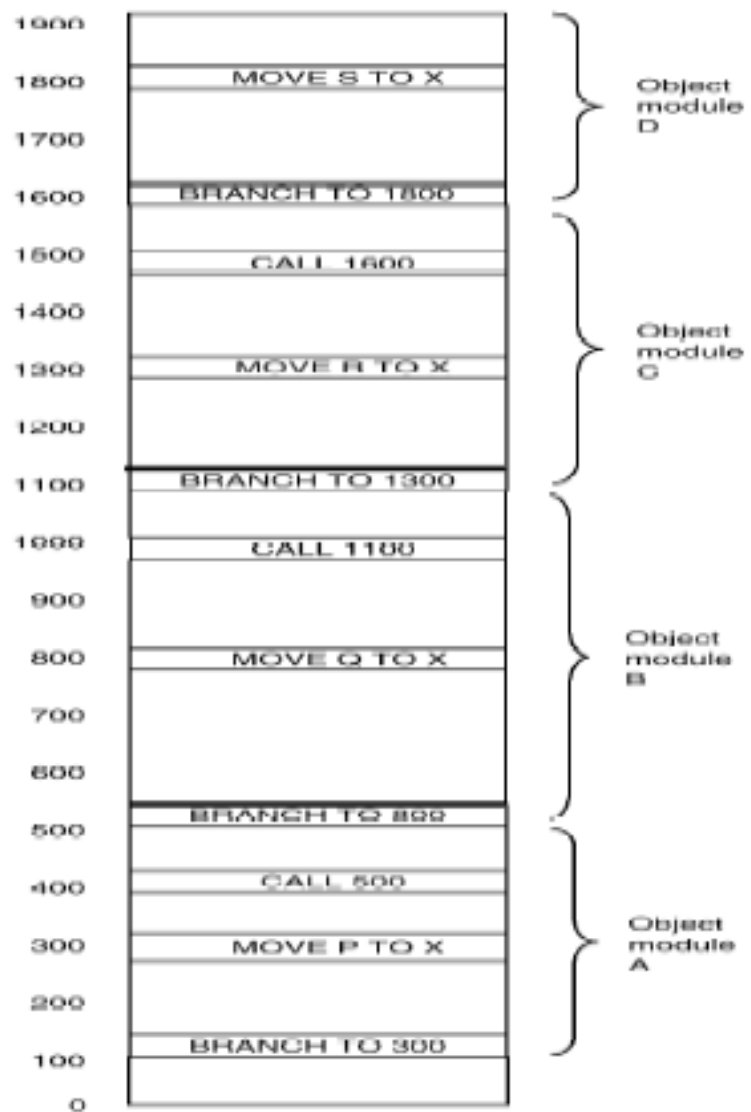
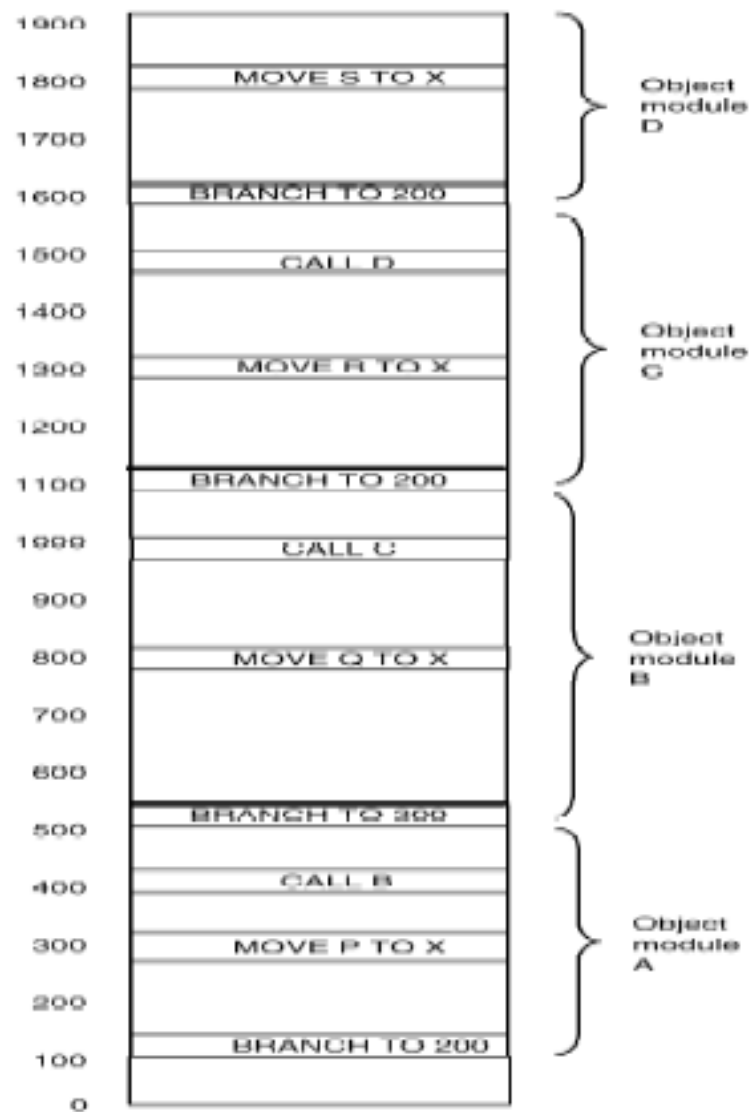
- **Kernel level** – indirizzi logici da 3 a 4 GB
- Condivisione esplicita tra tutti i processi (istanza singola in memoria di lavoro)

# Dynamic loading & linking

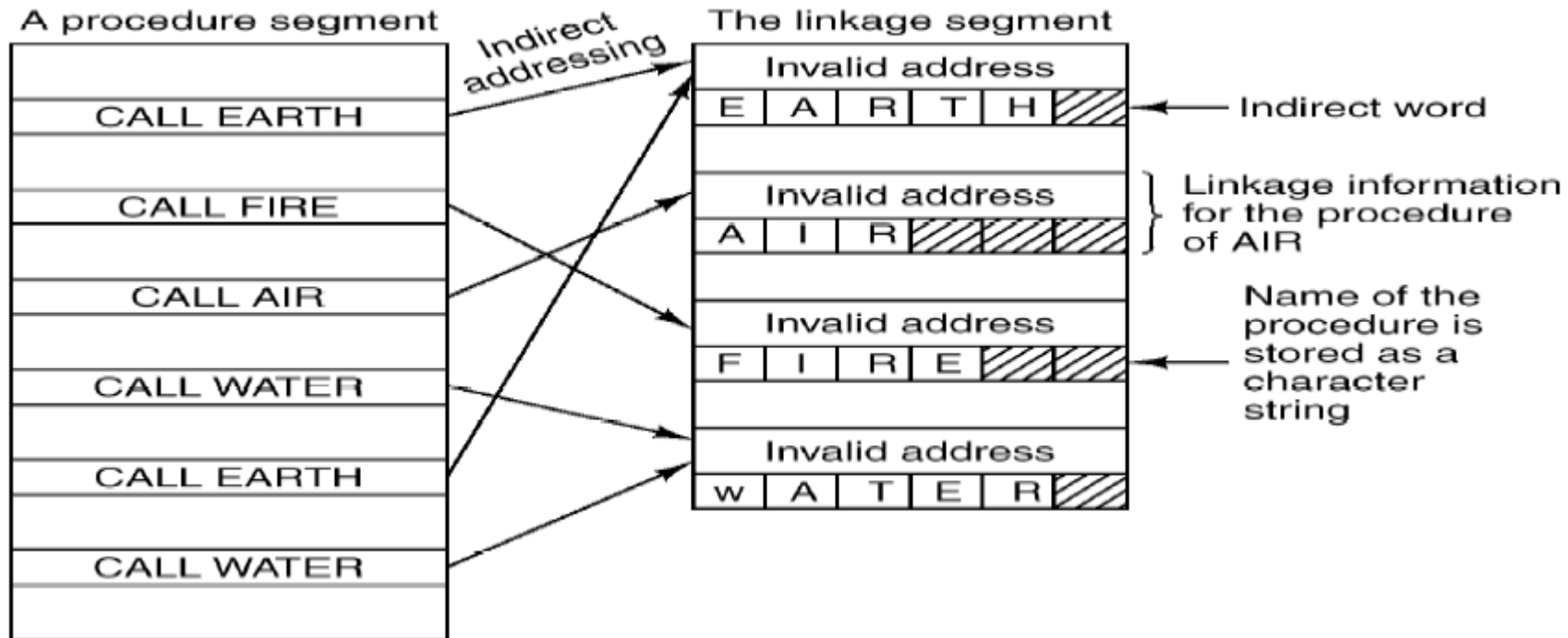
- Il dynamic loading e' una tecnica che permette di caricare del codice nella memoria riservata per un processo solo quando questo serve
- E' una tecnica simile all'overlay, ma con la differenza che la gestione del caricamento e' a carico di un modulo denominato dynamic loader, ed e' quindi trasparente al programmatore
- La definizione dei riferimenti di memoria per il codice caricato dinamicamente avviene run-time in modo da supportare dinamicamente il collegamento al resto del codice in esecuzione

# Esempio di collegamento statico

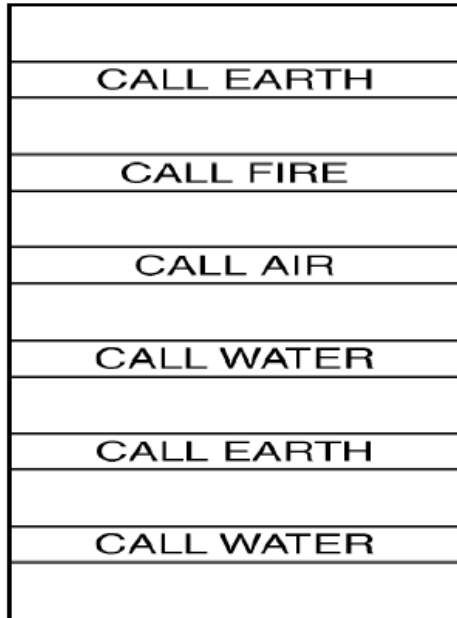




# Esempio di collegamento dinamico

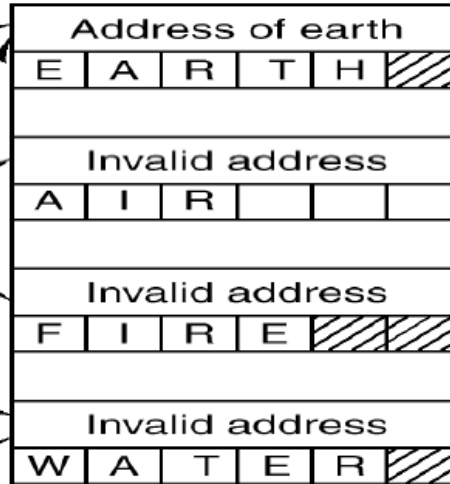


A procedure segment

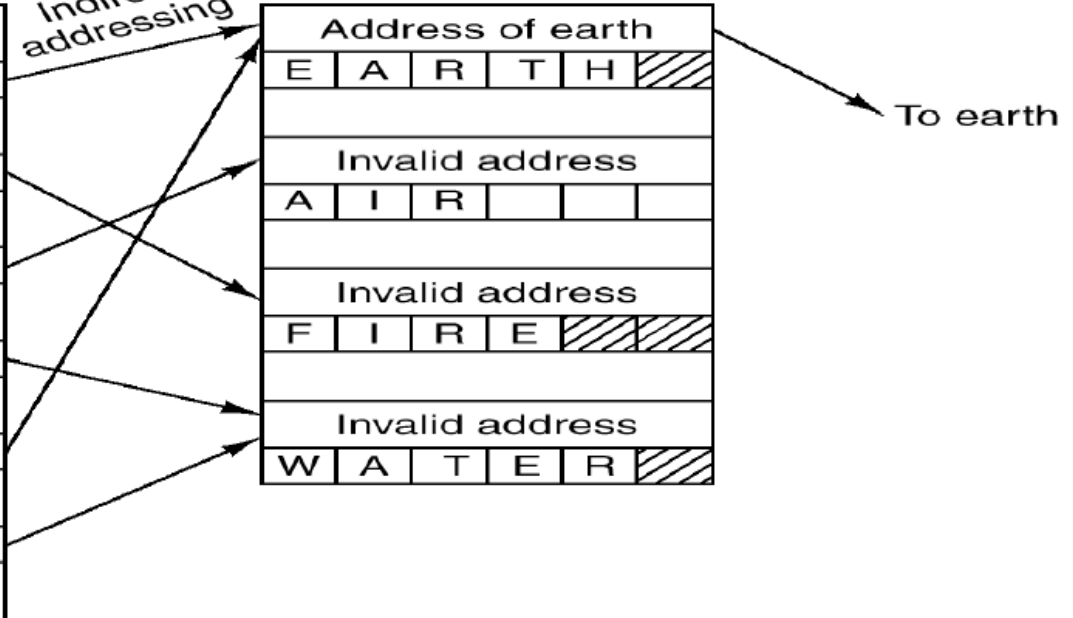


Indirect  
addressing

The linkage segment



To earth



# Memoria virtuale

- partizionamento, paginazione e segmentazione hanno lo scopo di permettere la condivisione della risorsa memoria tra più processi in modo da permettere multiprogrammazione
  - la loro caratteristica è che lo spazio di indirizzamento di un processo deve risiedere completamente in memoria centrale (con allocazione contigua o non)
  - unica eccezione è il caso dell'overlay, che comunque deve essere interamente gestito dal programmatore
- 

La memoria virtuale è una tecnica di gestione della memoria atta a permettere l'esecuzione di un processo il cui spazio di indirizzamento è caricato solo parzialmente in memoria centrale

# Vantaggi della memoria virtuale

- permette l'esecuzione di processi il cui spazio di indirizzamento eccede le dimensioni della memoria di lavoro
  - permette di aumentare il numero di processi che possono essere mantenuti contemporaneamente in memoria (aumento del grado di multiprogrammazione)
  - permette la riduzione del tempo necessario alle operazioni di swapping
- 

## Motivazioni per l'efficacia

- principi di località spaziale e temporale (l'accesso ad un indirizzo logico implica, con elevata probabilità, accesso allo stesso indirizzo o ad indirizzi adiacenti nell'immediato futuro)
- esistenza di porzioni di codice per la gestione di condizioni di errore o opzioni di programma di cui rarissimamente viene richiesta l'esecuzione
- sovradimensionamento di strutture dati (array, tavole ...) per le quali viene allocata più memoria di quella necessaria per la specifica istanza del problema

# Memoria virtuale in ambiente paginato

## Paginazione su richiesta (on demand)

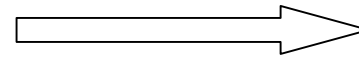
- all'attivazione di un processo, il suo spazio di indirizzamento non viene caricato in memoria
- una pagina dello spazio di indirizzamento viene caricata in un frame di memoria solo quando un indirizzo logico in quella pagina viene riferito dalla CPU (**page fault**)

## Prepaginazione (prepaging)

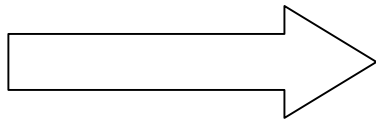
- un insieme di pagine viene caricato in memoria all'occorrenza di un page fault (l'insieme contiene la pagina per cui il page fault si verifica)

## Determinazione del page fault

- necessità di un meccanismo per verificare se la pagina dello spazio di indirizzamento a cui fa capo l'indirizzo correntemente riferito è già caricata in memoria centrale oppure no



Estensione della tavola  
delle pagine ...

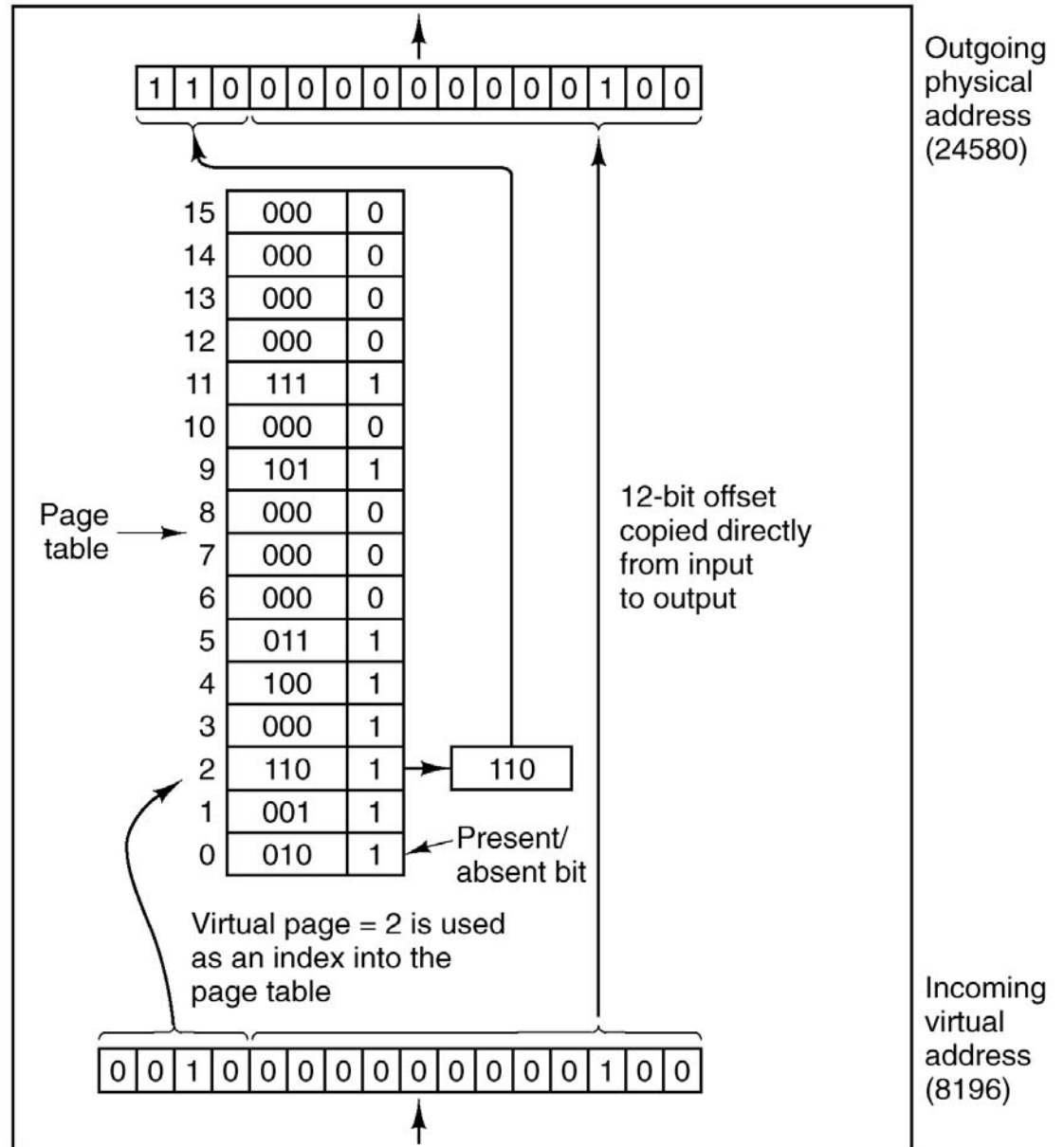


... tramite il  
bit di presenza

---

Se il bit di presenza  
vale 0 si ha un **page fault**  
e la pagina va caricata dal  
disco in memoria

---



# Prestazioni della memoria virtuale

$ma$  = tempo di accesso alla memoria

$pft$  = tempo medio di caricamento della pagina da disco  
(tempo di page fault)

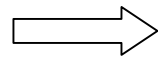
$f$  = frequenza di page fault

Tempo di accesso effettivo:  $ma \times (1 - f) + pft \times f$

**Supponendo:**

$ma = 5 \mu\text{sec}$

$pft = 10 \text{ msec}$



Tempo di accesso effettivo:  $5 + f9995 \mu\text{sec}$

Per un rallentamento inferiore al 10%,  $f$  deve essere dell'ordine di  $10^{-5}$

---

Criticità delle prestazioni in caso di frequenza di page fault non minimale

# Strategie di sostituzione delle pagine

Si tratta di stabilire quale pagina presente in memoria debba essere sostituita per far posto ad una nuova pagina da caricare

## Aspetti coinvolti

- resident set management:
  1. se l'insieme delle pagine da considerare per la sostituzione deve essere limitato alle pagine del processo che causa il page fault oppure si devono considerare pagine in un qualsiasi frame
  2. il numero di frame che devono essere allocati per ciascun processo
- nell'insieme di pagine considerate, quale “**vittima**” scegliere
- tener traccia di eventuali modifiche apportate alla pagina attualmente presente nel frame da sovrascrivere (gestione della coerenza della copia della pagina presente su disco)

# Restrizioni

## Blocco di frame

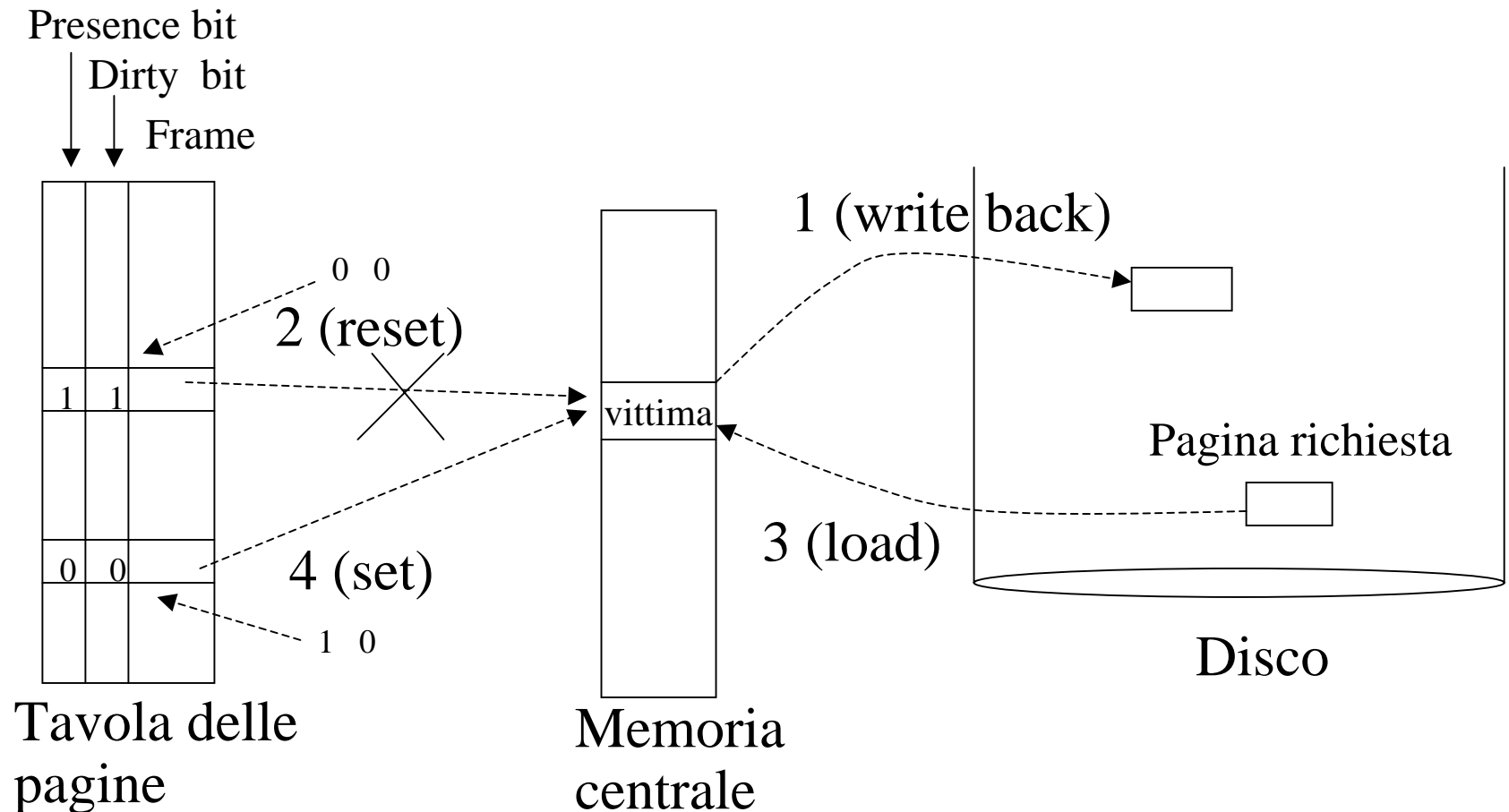
- alcuni frame di memoria possono essere bloccati, quindi la relativa pagina non può essere sostituita
- per motivi di efficienza ed implementativi, buona parte del kernel del sistema operativo è mantenuta in memoria in frame bloccati
- I/O asincrono effettuato direttamente su spazio di indirizzamento di un processo richiede il blocco in memoria dei frame associati ai buffer di I/O

---

Il sistema operativo mantiene un **lock bit** per ciascun frame, il cui valore indica se la pagina contenuta nel frame può essere considerata per una sostituzione

# Tener traccia delle modifiche

- la tavola delle pagine viene estesa con un bit per ogni entry, denominato **dirty bit**
- il dirty bit viene posto ad 1 all'atto di una scrittura sulla corrispondente pagina
- se il dirty bit associato alla vittima è pari a 1, la vittima viene ricopiata su disco



# Algoritmi di selezione della vittima

## Metrica di valutazione

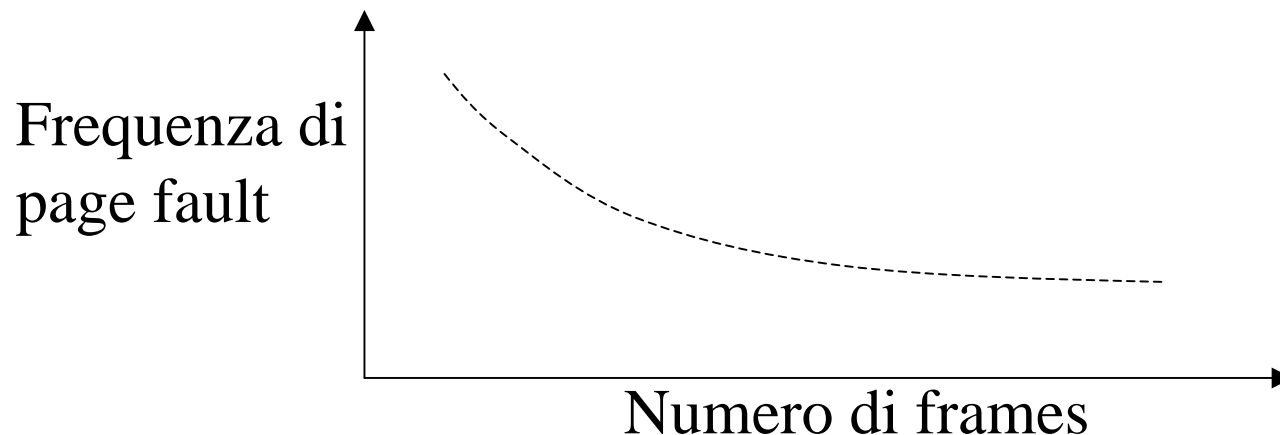
- frequenza di page fault

## Metodo di valutazione

- si utilizzano particolari sequenze di riferimenti ad indirizzi logici, generate in modo casuale oppure in base a tracce reali di esecuzione

## Aspettativa

- all'aumentare del numero di frames della memoria centrale, per una data sequenza di riferimenti ad indirizzi logici, la frequenza di page fault dovrebbe decrescere monotonamente



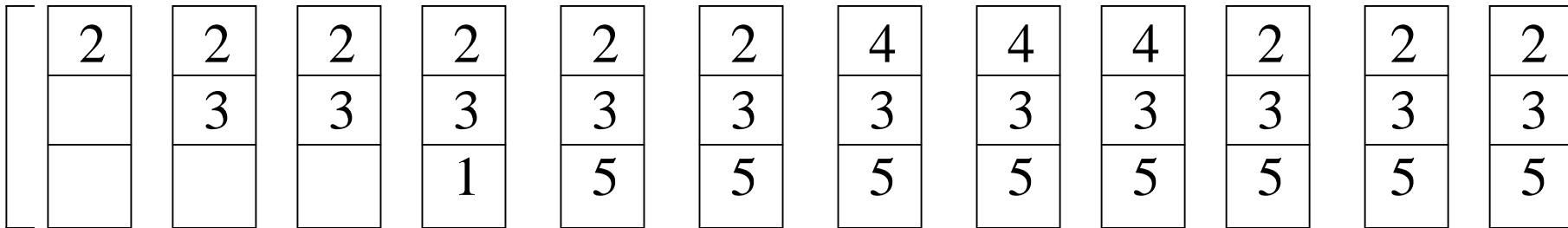
# Algoritmo ottimo

- seleziona per la sostituzione la pagina alla quale ci si riferirà dopo il più lungo tempo
- impossibile da implementare perchè richiede che il sistema operativo abbia conoscenza esatta degli eventi futuri (ovvero dei riferimenti alle pagine)
- si può usare come termine di paragone per valutare altri algoritmi

## Un esempio

Pagine riferite →

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



P

P

P

P

P

P

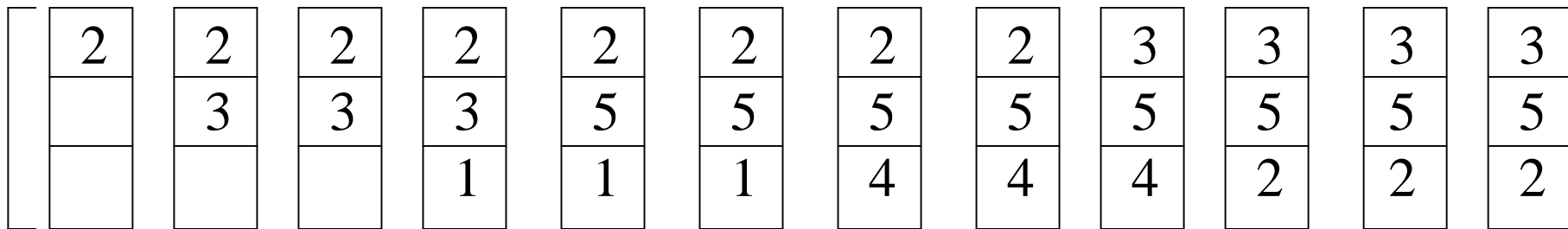
# Algoritmo Least-Recently-Used (LRU)

- seleziona per la sostituzione la pagina alla quale non ci si riferisce da più tempo (predizione della località in base al comportamento passato)
- difficile da implementare (necessità di marcare le pagine col tempo di riferimento o di mantenere uno stack di riferimenti alle pagine)

## Un esempio

Pagine riferite →

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



P

P

P

P

P

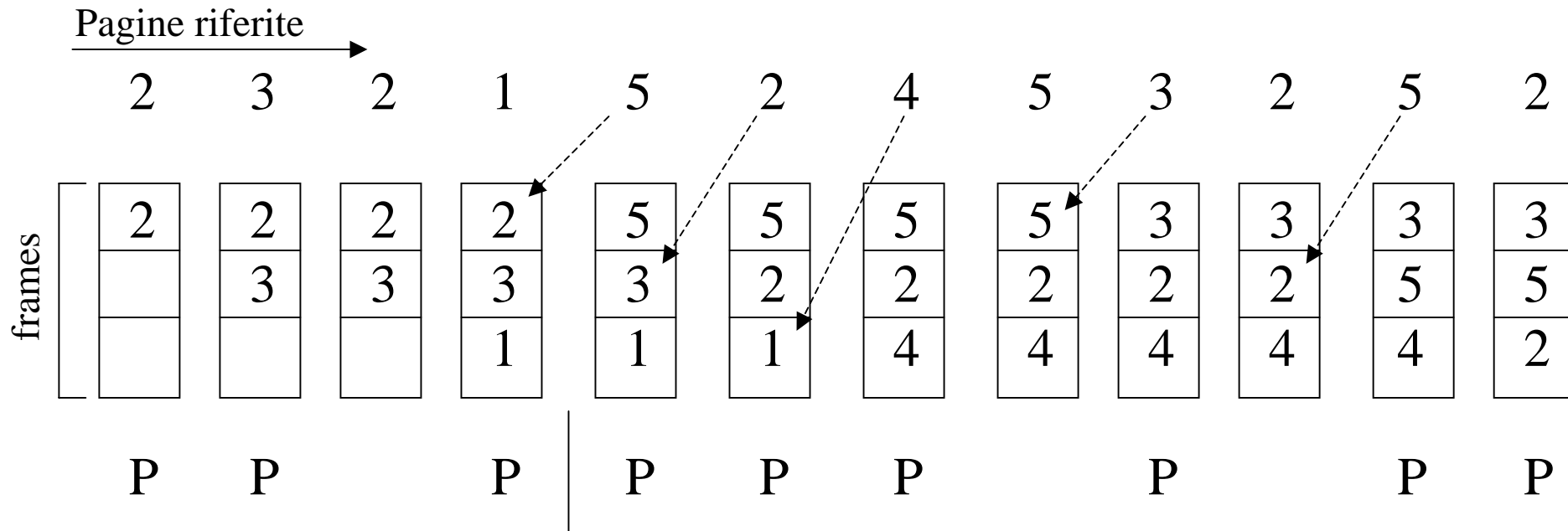
P

P

# Algoritmo First-In-First-Out (FIFO)

- seleziona per la sostituzione la pagina presente in memoria da più lungo tempo
- semplice da implementare (basta mantenere una lista dei frame organizzata in base all'ordine di caricamento delle pagine)
- non sfrutta a pieno il comportamento del programma in termini di località

## Un esempio



# Anomalia di Belady

Per una data dequenza di riferimenti, all'aumentare dei frames di memoria, il numero di page faults prodotti dall'algoritmo di sostituzione aumenta

**FIFO** presenta questa anomalia

0	1	2	3	0	1	4	0	1	2	3	4
0	1	2	3	0	1	4	4	4	2	3	3
	0	1	2	3	0	1	1	1	4	2	2
		0	1	2	3	0	0	0	1	4	4
P	P	P	P	P	P	P			P	P	

9 Page Faults

0	1	2	3	0	1	4	0	1	2	3	4
0	1	2	3	3	3	4	0	1	2	3	4
	0	1	2	2	2	3	4	0	1	2	3
		0	1	1	1	2	3	4	0	1	2
			0	0	0	1	2	3	4	0	1
P	P	P	P			P	P	P	P	P	P

10 Page Faults

# Algoritmi a stack

## Caratteristiche

- per qualsiasi sequenza di riferimenti l'insieme delle pagine in memoria per  $n$  frames è sempre un sottoinsieme dell'insieme di pagine in memoria per  $n+1$  frames

$$\forall \text{ sequenza } r : M(n+1, r) \supseteq M(n, r)$$

## Proprietà

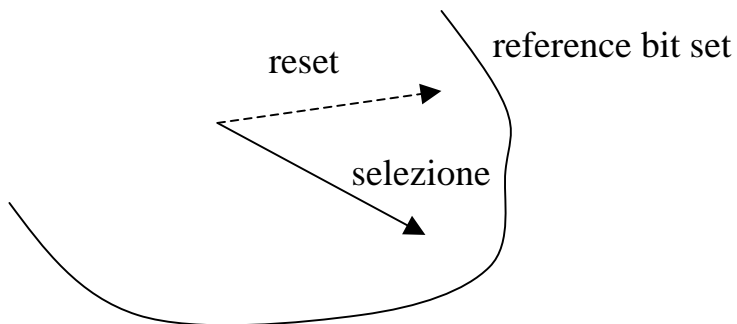
- non mostrano anomalia di Belady (la frequenza di page fault decresce all'aumentare del numero di frames)

---

**LRU è un algoritmo a stack**

# Algoritmo dell'orologio (NRU)

- un **reference bit** è associato a ciascun frame
- il reference bit è impostato ad 1 ogni volta che la pagina in quel frame è referenziata in lettura o scrittura
- periodicamene , il sistema operativo scorre parte dell'insieme dei reference bit e li imposta a 0 (lancetta del reset)
- quando una sostituzione di pagina è necessaria allora il sistema operativo scorre i reference bit fino a che non ne trova uno impostato a 0 (lancetta di selezione), la pagina corrispondente viene selezionata per la sostituzione
- durante quest'ultimo scorrimento, reference bit ad 1 vengono resettati



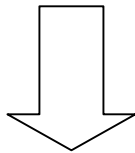
La lancetta di selezione fa un giro completo in caso lo use bit di tutti i frame sia reimpostato ad 1 dall'ultimo reset

# Algoritmo dell'orologio con bit aggiuntivi

- il **reference bit** viene manipolato come nel caso dell'algoritmo dell'orologio classico
- il **dirty bit** viene usato in combinazione al **reference bit** per determinare quale pagina sostituire

## Possibili combinazioni

(rb=0, db=0)   (rb=1, db=0)   (rb=0, db=1)   (rb=1, db=1)



Pagina preferita per la sostituzione perchè, oltre a non essere stata usata di recente, non necessita di essere ricopiata su disco

**Es. sistemi Macintosh**

# Resident set

- minore è la taglia del resident set, maggiore è la quantità di processi mantenuti in memoria (e quindi il grado di multiprogrammazione), inoltre il tempo di swapping viene ridotto
  - se il resident set è troppo piccolo, la frequenza di page fault potrebbe essere inaccettabilmente alta
- 

## Allocazione fissa

- si assegnano al processo un numero di frames fisso, deciso alla sua attivazione (criticità della scelta della taglia)
- la sostituzione di pagina coinvolge solo frames del processo

## Allocazione variabile

- il numero di frames per un dato processo può variare durante la sua esecuzione (migliore utilizzo della memoria, ma complessità di gestione superiore)
- la sostituzione di pagina può coinvolgere frames di altri processi

# Allocazione mista

- quando un processo è attivato, si alloca un dato numero di frames, caricati tramite paginazione su richiesta o prepaginazione
- quando avviene un page fault, la pagina da sostituire viene selezionata tra quelle del resident set del processo
- periodicamente si rivaluta la taglia del resident set dei processi attivi, per favorire l'aumento di quella dei processi con meno località

## Strategie di rivalutazione della taglia

- Working Set
- Frequenza di page fault

# Working Set

Si definisce Working set di un processo al tempo  $t$ , l'insieme  $W(t, \Delta)$  costituito dagli ultimi  $\Delta$  riferimenti di pagina per quel processo

- se una pagina è stata usata di recente, allora probabilmente appartiene al Working set
- assumendo il comportamento nel recente passato come rappresentativo dell'immediato futuro, il Working set approssima la località del processo
- la precisione del Working set dipende dal parametro  $\Delta$ 
  1. Se troppo grande, può venir catturata anche la variazione di località
  2. Se troppo piccolo, la località può non venir catturata completamente

## Uso

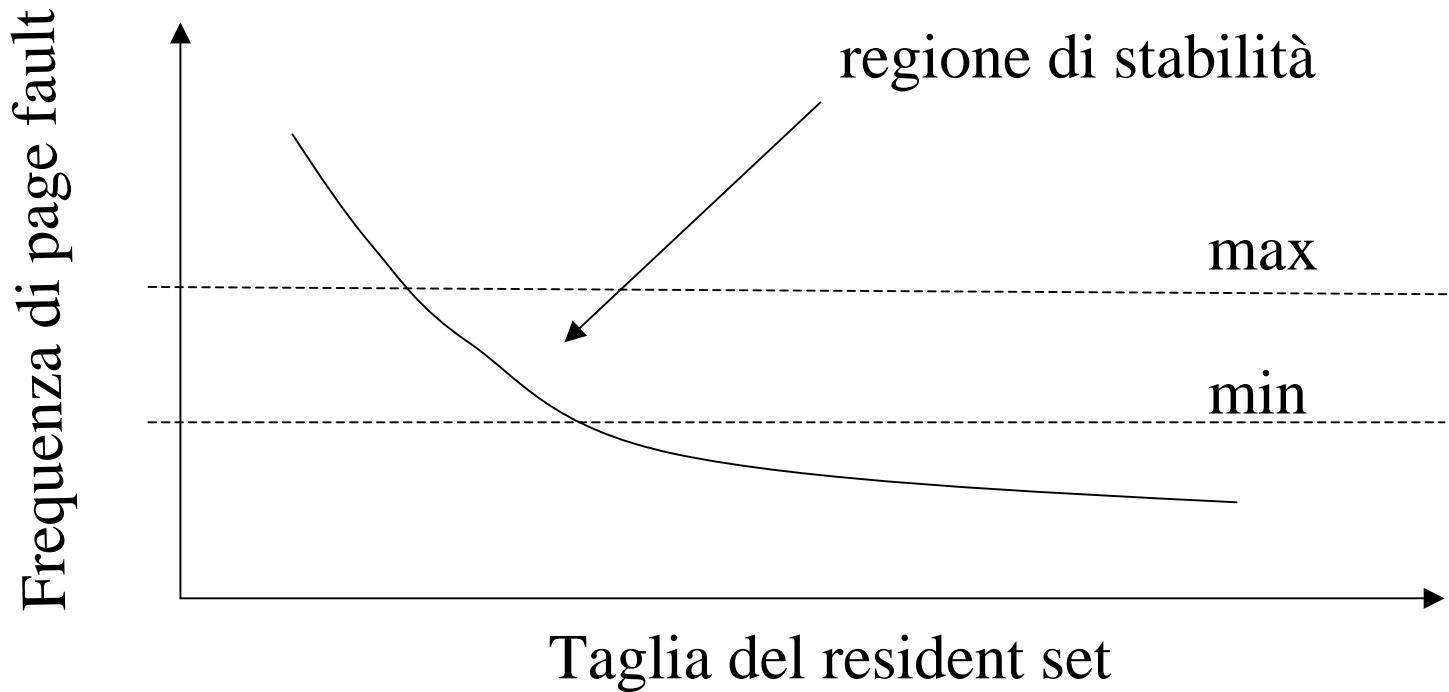
- il sistema operativo dovrebbe ingrandire o ridurre il resident set in modo da ottenere una taglia pari al Working set

## Problemi

- difficoltà pratica di scegliere il valore di  $\Delta$  e di valutare dove cadano gli ultimi  $\Delta$  riferimenti a pagine di un dato processo

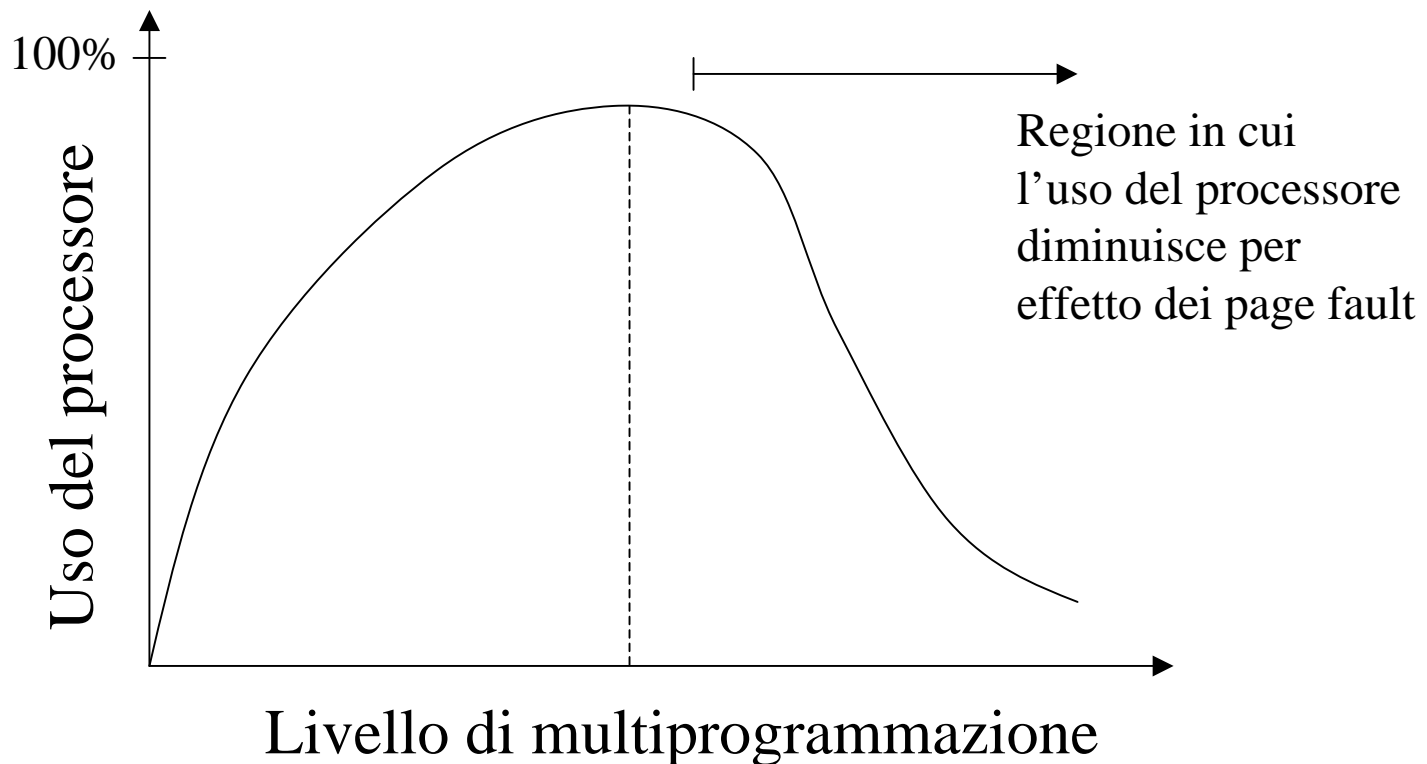
# Frequenza di page fault

Se la frequenza di page fault di un dato processo è sotto una soglia minima è conveniente ridurre il suo resident set a favore di processi aventi frequenza di page fault sopra una soglia massima



# Il thrashing

- tipicamente un aumento del livello di multiprogrammazione dovrebbe favorire l'uso del processore e dei dispositivi
- ma un grado di multiprogrammazione eccessivo tende a creare resident sets troppo piccoli e l'uso del processore diminuisce (la frequenza di page fault diviene eccessiva per tutti i processi attivi i quali rimangono bloccati in attesa di caricamento delle loro pagine)



# Recupero dal thrashing

## Swap-out del processo

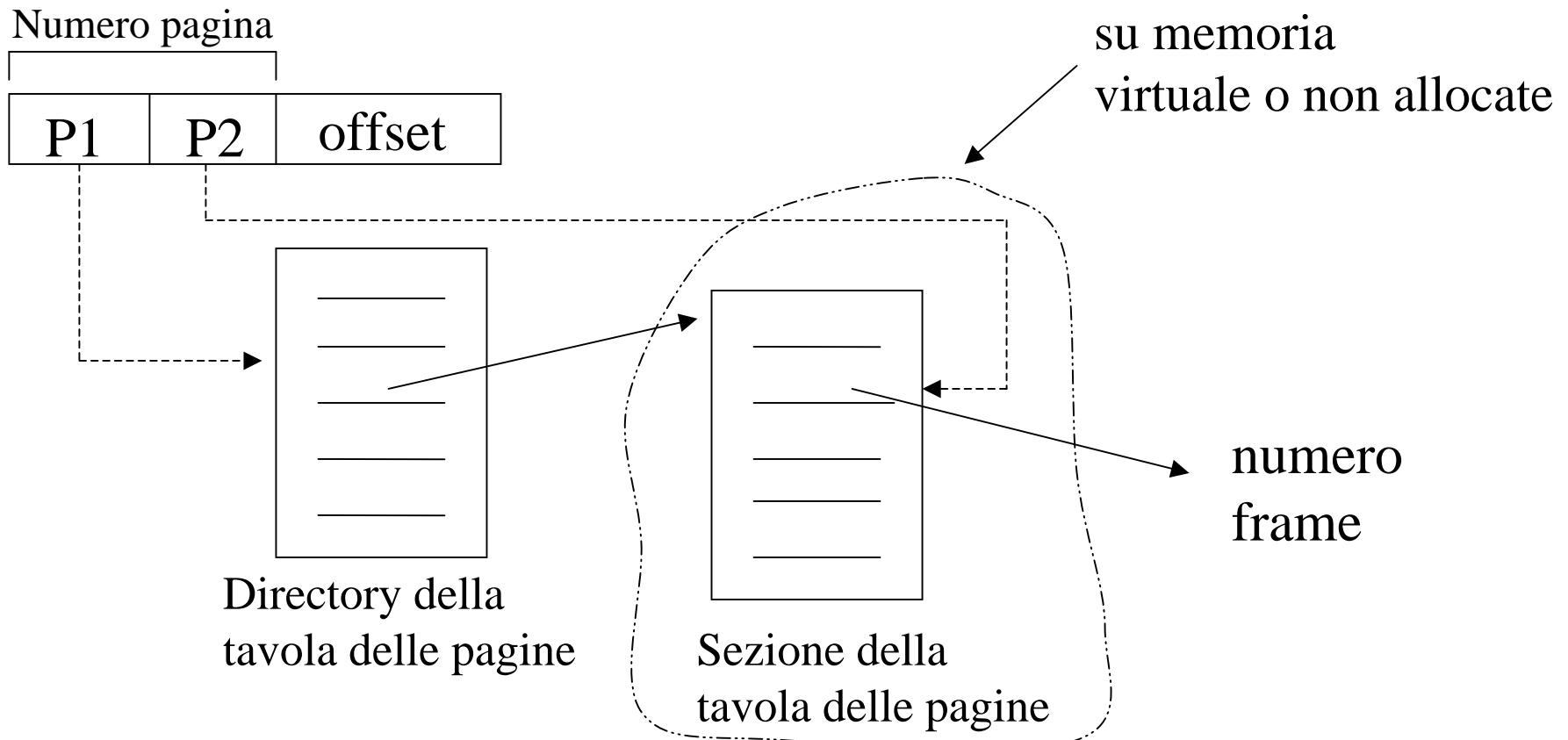
- a più bassa priorità
  - che ha più alta frequenza di page fault (taglia del resident set non adeguata)
  - ultimo attivato (working set non ancora completo)
  - di taglia minore (ridotto costo di swapping)
  - di taglia maggiore (maggior numero di frames liberati)
- 

Non supportato a causa di eccessiva complessità di gestione (monitoraggio e mantenimento di strutture dati)

Si è tipicamente ottimisti sul fatto che il thrashing sia un evento molto raro

# Paginazione a livelli multipli

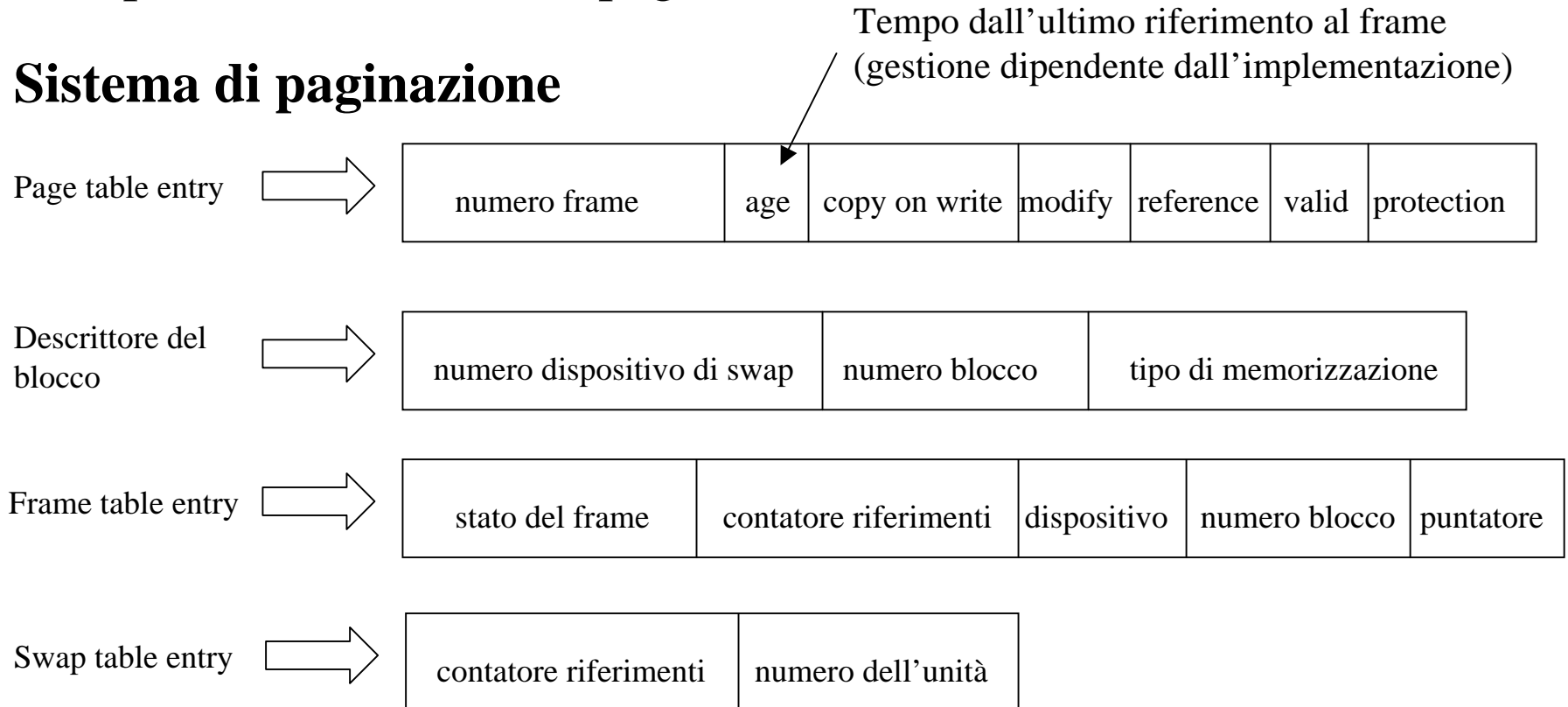
- le tabelle delle pagine vengono mantenute nello spazio riservato al sistema operativo
- l'occupazione di memoria del sistema operativo può essere ridotta utilizzando la memoria virtuale per la stessa tabella delle pagine o allocando sezioni della tabella solo qualora necessarie



# Gestione della memoria in UNIX

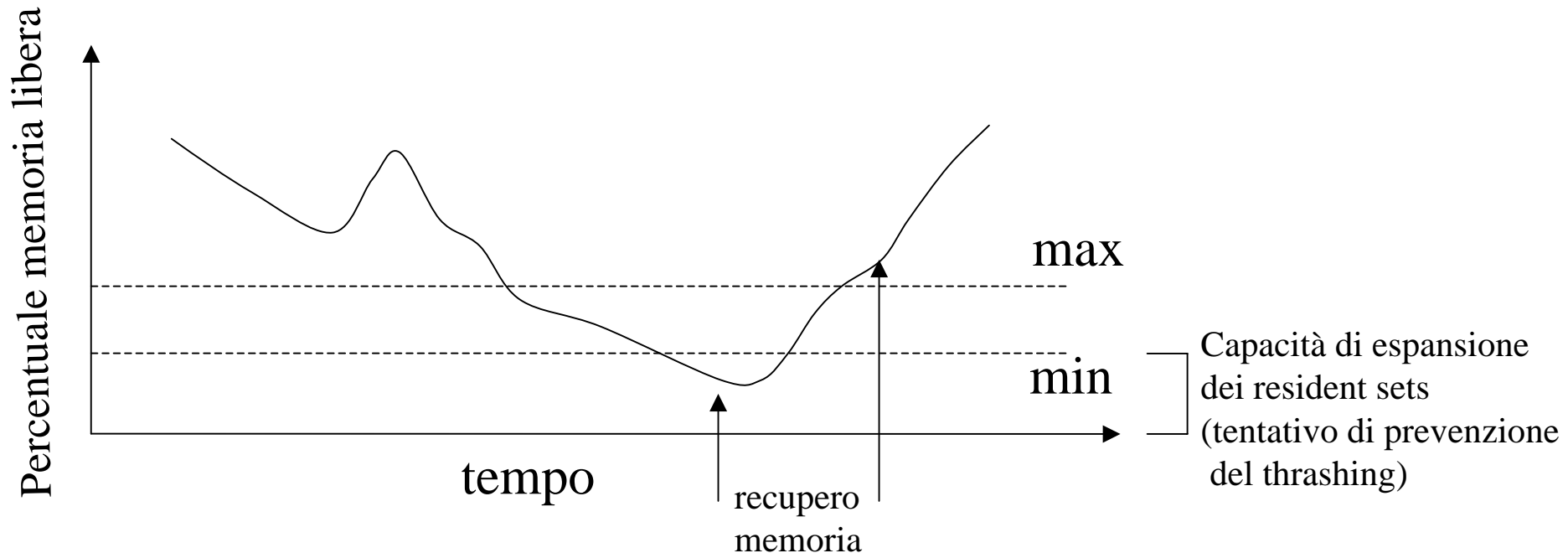
- le prime versioni usavano partizionamento dinamico senza uso di memoria virtuale
- da SVR4 e Solaris2.x in poi si ha un sistema di memoria virtuale paginata
- il kernel ha un allocatore di memoria apposito (Buddy System), separato dal sistema di paginazione

## Sistema di paginazione



# Sostituzione delle pagine

- algoritmo dell'orologio
- allocazione variabile (numero di frames variabile per processo)
- il sistema libera periodicamente i frame di memoria quando la percentuale di memoria libera scende sotto una soglia minima
- sono liberati i frames necessari a riportare la memoria libera sopra una soglia massima

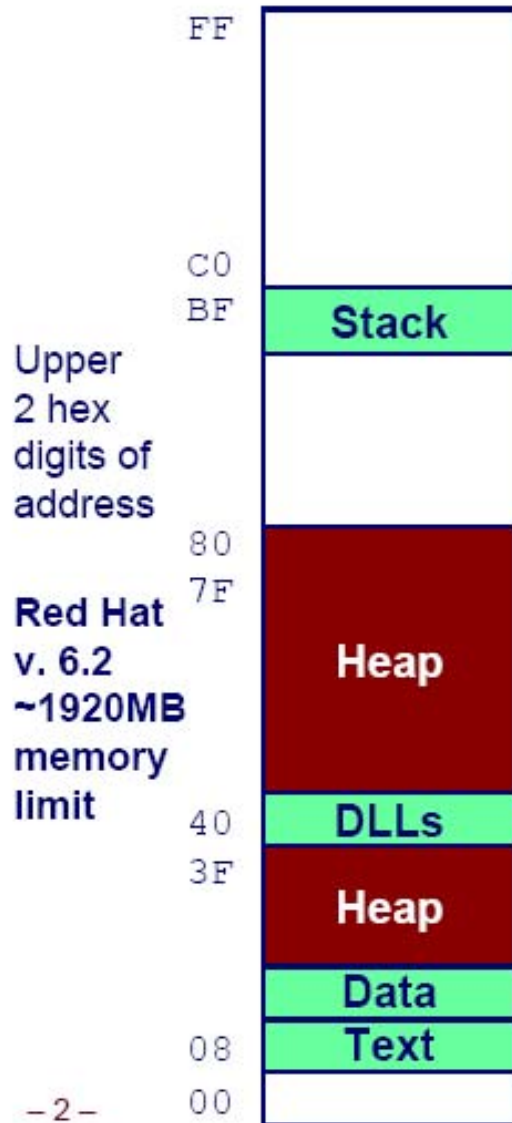


# Gestione memoria kernel: Buddy System (sistema dei compagni)

- i blocchi di memoria allocati/deallocati hanno dimensione  $2^k$
- max blocco è di taglia  $2^u$ , corrispondente alla memoria totale disponibile per il kernel
- per ogni richiesta di dimensione  $s$  tale che  $2^{k-1} < s \leq 2^k$  il blocco di taglia  $2^k$  viene allocato per la richiesta
- se la precedente condizione non è soddisfatta, allora il blocco di taglia  $2^k$  viene diviso in due e la condizione viene rivalutata sulle taglie  $2^{k-1}$  e  $2^{k-2}$
- per tenere traccia di blocchi liberi/occupati viene usata una struttura ad albero
- blocchi adiacenti vengono ricompattati secondo una politica lazy (pigra), ovvero quando il loro numero supera una certa soglia



# Linux Memory Layout



## Stack

- Runtime stack (8MB limit)

## Heap

- Dynamically allocated storage
- When call `malloc`, `calloc`, `new`

## DLLs

- Dynamically Linked Libraries
- Library routines (e.g., `printf`, `malloc`)
- Linked into object code when first executed

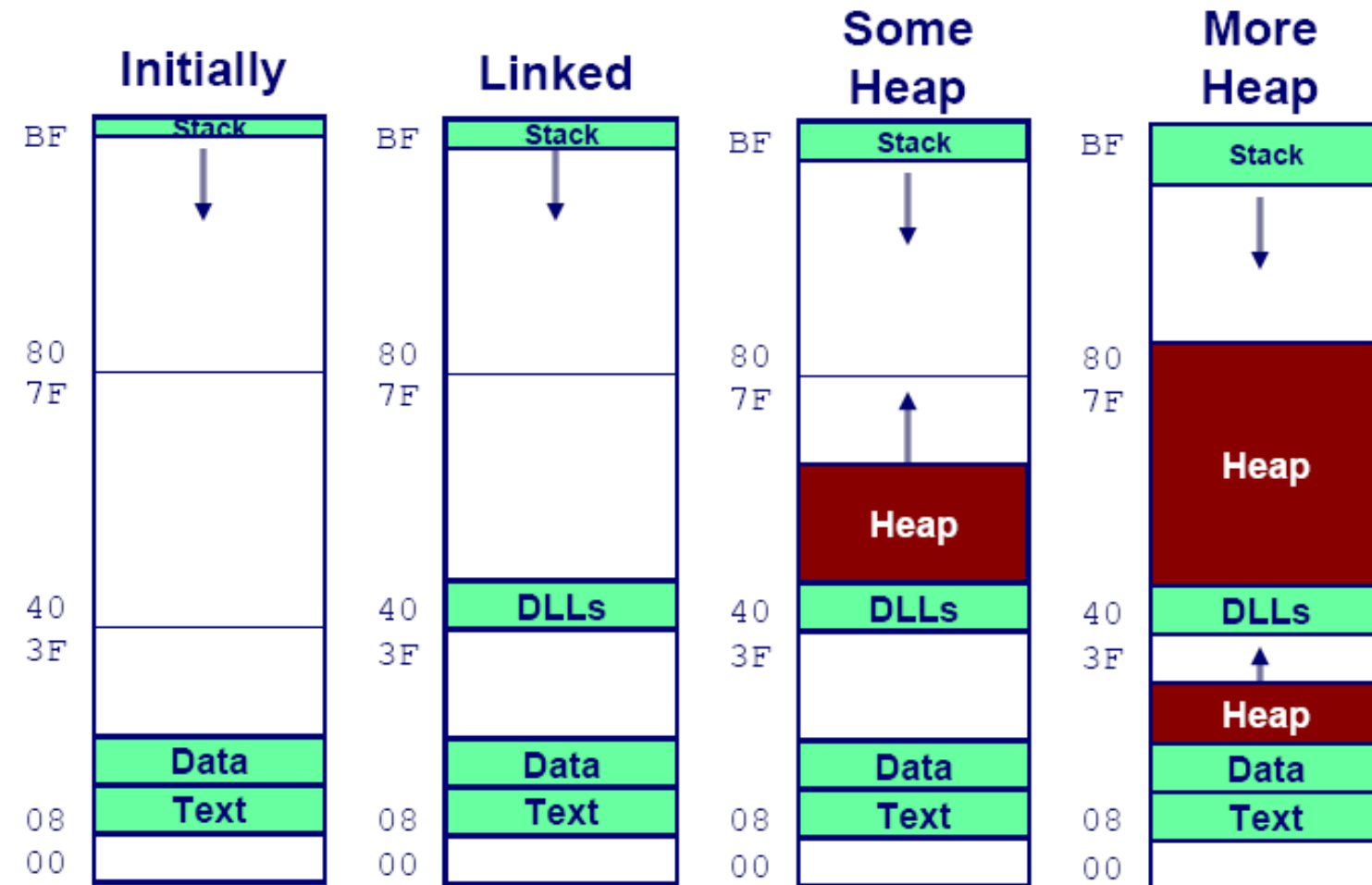
## Data

- Statically allocated data
- E.g., arrays & strings declared in code

## Text

- Executable machine instructions
- Read-only

# Linux Memory Allocation



# System call UNIX per l'allocazione e deallocazione di memoria virtuale

## SYNOPSIS

```
#include <unistd.h>

int brk(void *end data segment);

void *sbrk(ptrdiff_t increment);
```

## DESCRIPTION

**brk** sets the end of the data segment to the value specified by end data segment, when that value is reasonable, the system does have enough memory and the process does not exceed its max data size (see **setrlimit(2)**).

**sbrk** increments the program's data space by increment bytes. **sbrk** isn't a system call, it is just a C library wrapper. Calling **sbrk** with an increment of 0 can be used to find the current location of the program break.

## RETURN VALUE

On success, **brk** returns zero, and **sbrk** returns a pointer to the start of the new area. On error, -1 is returned, and errno is set to **ENOMEM**.

## SYNOPSIS

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

#ifdef _POSIX_MAPPED_FILES

void * mmap(void *start, size_t length, int prot , int flags, int
fd, off_t offset);

int munmap(void *start, size_t length);

#endif
```

## DESCRIPTION

The `mmap` function asks to map `length` bytes starting at offset `offset` from the file (or other object) specified by the file descriptor `fd` into memory, preferably at address `start`. This latter address is a hint only, and is usually specified as 0. The actual place where the object is mapped is returned by `mmap`, and is never 0.

The `prot` argument describes the desired memory protection (and must not conflict with the open mode of the file). It is either `PROT_NONE` or is the bitwise OR of one or more of the other `PROT_*` flags.

# Memoria virtuale in NT/2000

- il gestore della memoria usa dimensioni di pagina variabili tra 4 KB e 64 KB
- spazio di indirizzamento virtuale di 4 GB
- 2 GB privati
- 2 GB condivisi e riservati al sistema operativo (NT executive, microkernel, driver dei dispositivi)
- area di separazione tra le due (di taglia 1 pagina) come guardia per l'uso improprio dell'indirizzamento (indirizzamento fuori limite)

## Stati delle pagine (spazio di indirizzamento parte privata)

- **disponibile:** correntemente non usata dal processo
- **riservata:** messa da parte per un processo ma non in quota al processo fino a che esso non accede in scrittura
- **committed:** pagina per la quale il gestore ha realmente riservato memoria per la paginazione da gestire in modalità write-back

# Gestione del resident set

- la sostituzione di una pagina avviene nel resident set del processo (ambito locale)
- il resident set può essere espanso in caso di page fault qualora la memoria libera sia al di sopra di una data soglia
- se la memoria libera scende sotto una data soglia, il gestore libera i frame seguendo una politica di tipo not-recently-used
- i frame vengono liberati nell'ambito dei resident set di tutti i processi attivi

# System call Windows per l'allocazione e deallocazione di memoria virtuale

## VirtualAlloc

The **VirtualAlloc** function reserves or commits a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero, unless `MEM_RESET` is specified.

To allocate memory in the address space of another process, use the [VirtualAllocEx](#) function.

```
LPVOID VirtualAlloc(  
    LPVOID lpAddress,  
    SIZE_T dwSize,  
    DWORD flAllocationType,  
    DWORD flProtect  
);
```

## Parameters

### *lpAddress*

[in] The starting address of the region to allocate. If the memory is being reserved, the specified address is rounded down to the nearest multiple of the allocation granularity. If the memory is already reserved and is being committed, the address is rounded down to the next page boundary. To determine the size of a page and the allocation granularity on the host computer, use the [GetSystemInfo](#) function. If this parameter is NULL, the system determines where to allocate the region.

### *dwSize*

[in] The size of the region, in bytes. If the *lpAddress* parameter is NULL, this value is rounded up to the next page boundary. Otherwise, the allocated pages include all pages containing one or more bytes in the range from *lpAddress* to *lpAddress+dwSize*. This means that a 2-byte range straddling a page boundary causes both pages to be included in the allocated region.

### *flAllocationType*

[in] Type of memory allocation. This parameter must contain one of the following values.

### *flProtect*

[in] Memory protection for the region of pages to be allocated. If the pages are being committed, you can specify any one of the [memory protection constants](#).

Protection attributes specified when protecting a page cannot conflict with those specified when allocating a page.

Value	Meaning
MEM_COMMIT 0x1000	<p>Allocates physical storage in memory or in the paging file on disk for the specified reserved memory pages. The function initializes the memory to zero.</p> <p>To reserve and commit pages in one step, call <b>VirtualAlloc</b> with MEM_COMMIT   MEM_RESERVED.</p> <p>The function fails if you attempt to commit a page that has not been reserved. The resulting error code is ERROR_INVALID_ADDRESS.</p> <p>An attempt to commit a page that is already committed does not cause the function to fail. This means that you can commit pages without first determining the current commitment state of each page.</p>
MEM_RESERVE 0x2000	<p>Reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk.</p> <p>You can commit reserved pages in subsequent calls to the <b>VirtualAlloc</b> function. To reserve and commit pages in one step, call <b>VirtualAlloc</b> with MEM_COMMIT   MEM_RESERVED.</p> <p>Other memory allocation functions, such as <b>malloc</b> and <a href="#">LocalAlloc</a>, cannot use a reserved range of memory until it is released.</p>
MEM_RESET 0x80000	<p>Indicates that data in the memory range specified by <i>lpAddress</i> and <i>dwSize</i> is no longer of interest. The pages should not be read from or written to the paging file. However, the memory block will be used again later, so it should not be decommitted. This value cannot be used with any other value.</p> <p>Using this value does not guarantee that the range operated on with MEM_RESET will contain zeroes. If you want the range to contain zeroes, decommit the memory and then recommit it.</p> <p>When you specify MEM_RESET, the <b>VirtualAlloc</b> function ignores the value of <i>fProtect</i>. However, you must still set <i>fProtect</i> to a valid protection value, such as PAGE_NOACCESS.</p> <p><b>VirtualAlloc</b> returns an error if you use MEM_RESET and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.</p> <p><b>Windows Me/98/95:</b> This flag is not supported.</p>

# VirtualFree

The **VirtualFree** function releases, decommits, or releases and decommits a region of pages within the virtual address space of the calling process.

To free memory allocated in another process by the [VirtualAllocEx](#) function, use the [VirtualFreeEx](#) function.

```
BOOL VirtualFree(  
    LPVOID lpAddress,  
    SIZE_T dwSize,  
    DWORD dwFreeType  
);
```

## Parameters

*lpAddress*

[in] A pointer to the base address of the region of pages to be freed.

If the *dwFreeType* parameter is MEM\_RELEASE, this parameter must be the base address returned by the [VirtualAlloc](#) function when the region of pages is reserved.

*dwSize*

[in] The size of the region of memory to be freed, in bytes.

If the *dwFreeType* parameter is MEM\_RELEASE, this parameter must be 0 (zero). The function frees the entire region that is reserved in the initial allocation call to **VirtualAlloc**.

If the *dwFreeType* parameter is MEM\_DECOMMIT, the function decommits all memory pages that contain one or more bytes in the range from the *lpAddress* parameter to (*lpAddress+dwSize*). This means, for example, that a 2-byte region of memory that straddles a page boundary causes both pages to be decommitted. If *lpAddress* is the base address returned by **VirtualAlloc** and *dwSize* is 0 (zero), the function decommits the entire region that is allocated by **VirtualAlloc**. After that, the entire region is in the reserved state.

*dwFreeType*

[in] The type of free operation. This parameter can be one of the following values.

Value	Meaning
MEM_DECOMMIT 0x4000	<p>Decommits the specified region of committed pages. After the operation, the pages are in the reserved state.</p> <p>The function does not fail if you attempt to decommit an uncommitted page. This means that you can decommit a range of pages without first determining the current commitment state.</p> <p>Do not use this value with MEM_RELEASE.</p>
MEM_RELEASE 0x8000	<p>Releases the specified region of pages. After this operation, the pages are in the free state.</p> <p>If you specify this value, <i>dwSize</i> must be 0 (zero), and <i>lpAddress</i> must point to the base address returned by the <a href="#">VirtualAlloc</a> function when the region is reserved. The function fails if even of the conditions is not met.</p> <p>If any pages in the region are committed currently, the function first decommits, and then releases them.</p> <p>The function does not fail if you attempt to release pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining the current commitment state.</p> <p>Do not use this value with MEM_DECOMMIT.</p>