

Gestione della Memoria

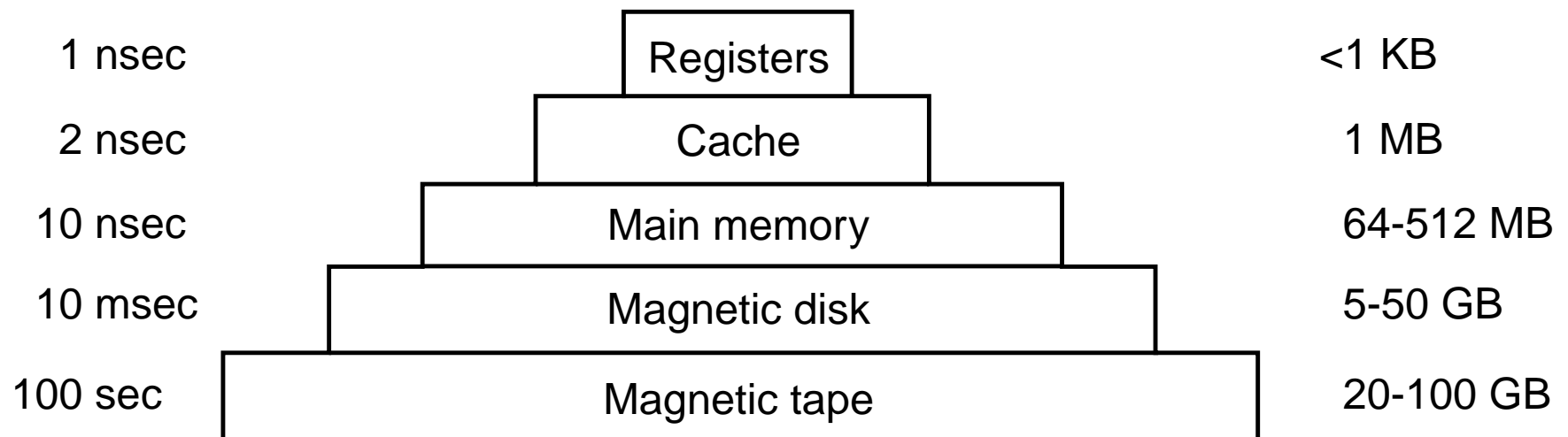
- Fondamenti
- Associazione degli indirizzi alla memoria fisica
- Spazio indirizzi logico vs. fisico
- Allocazione contigua
 - partizionamento fisso
 - partizionamento dinamico
- Allocazione non contigua
 - Paginazione
 - Segmentazione
 - Segmentazione con paginazione
- Implementazione

Gestione della Memoria

- La memoria è una risorsa importante, e limitata.
- “I programmi sono come i gas reali: si espandono fino a riempire la memoria disponibile”
- Memoria illimitata, infinitamente veloce, economica: non esiste.
- Esiste la *gerarchia della memoria*, gestita dal *gestore della memoria*

Typical access time

Typical capacity

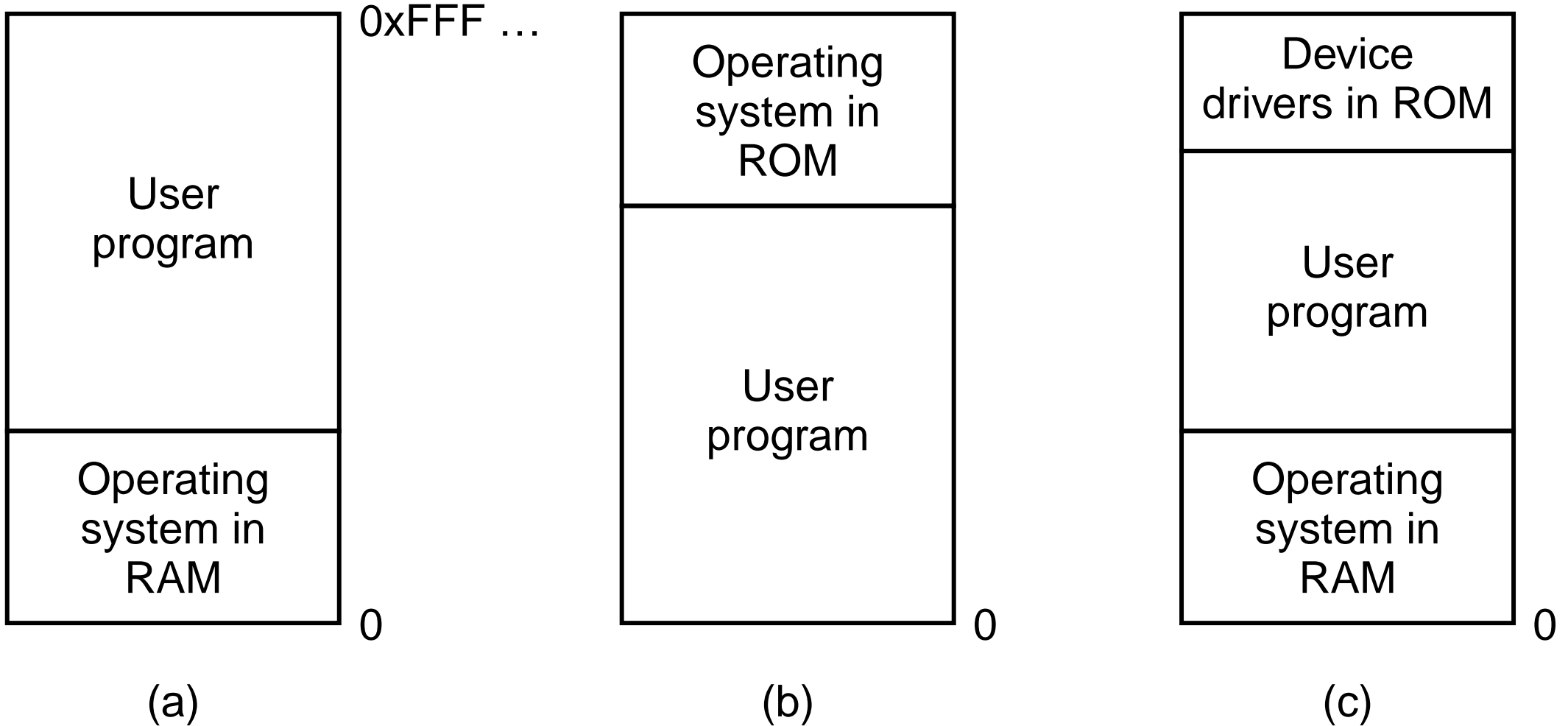


Gestione della memoria: Fondamenti

La gestione della memoria mira a soddisfare questi requisiti:

- Organizzazione logica: offrire una visione astratta della gerarchia della memoria: allocare e deallocare memoria ai processi su richiesta
- Organizzazione fisica: tener conto a chi è allocato cosa, e effettuare gli scambi con il disco.
- Rilocazione
- Protezione: tra i processi, e per il sistema operativo
- Condivisione: aumentare l'efficienza

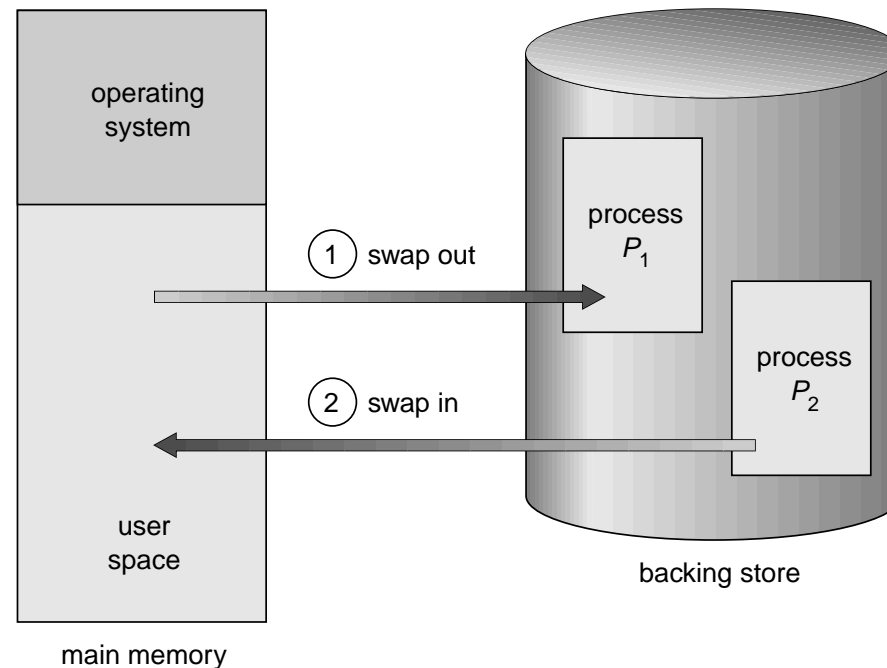
Monoprogrammazione



Un solo programma per volta (oltre al sistema operativo). (a) vecchi mainframe e minicomputer; (b) palmari e sistemi embedded; (c) BIOS (Basic Input Output System) in MS-DOS.

Swapping

- Un processo in esecuzione può essere temporaneamente rimosso dalla memoria e riversato (*swapped*) in una memoria secondaria (detta *backing store* o *swap area*); in seguito può essere riportato in memoria per continuare l'esecuzione.
- Lo spazio indirizzi di interi processi viene spostato
- *Backing store*: dischi veloci e abbastanza larghi da tenere copia delle immagini delle memorie dei processi che si intende swappare.



Swapping (Cont.)

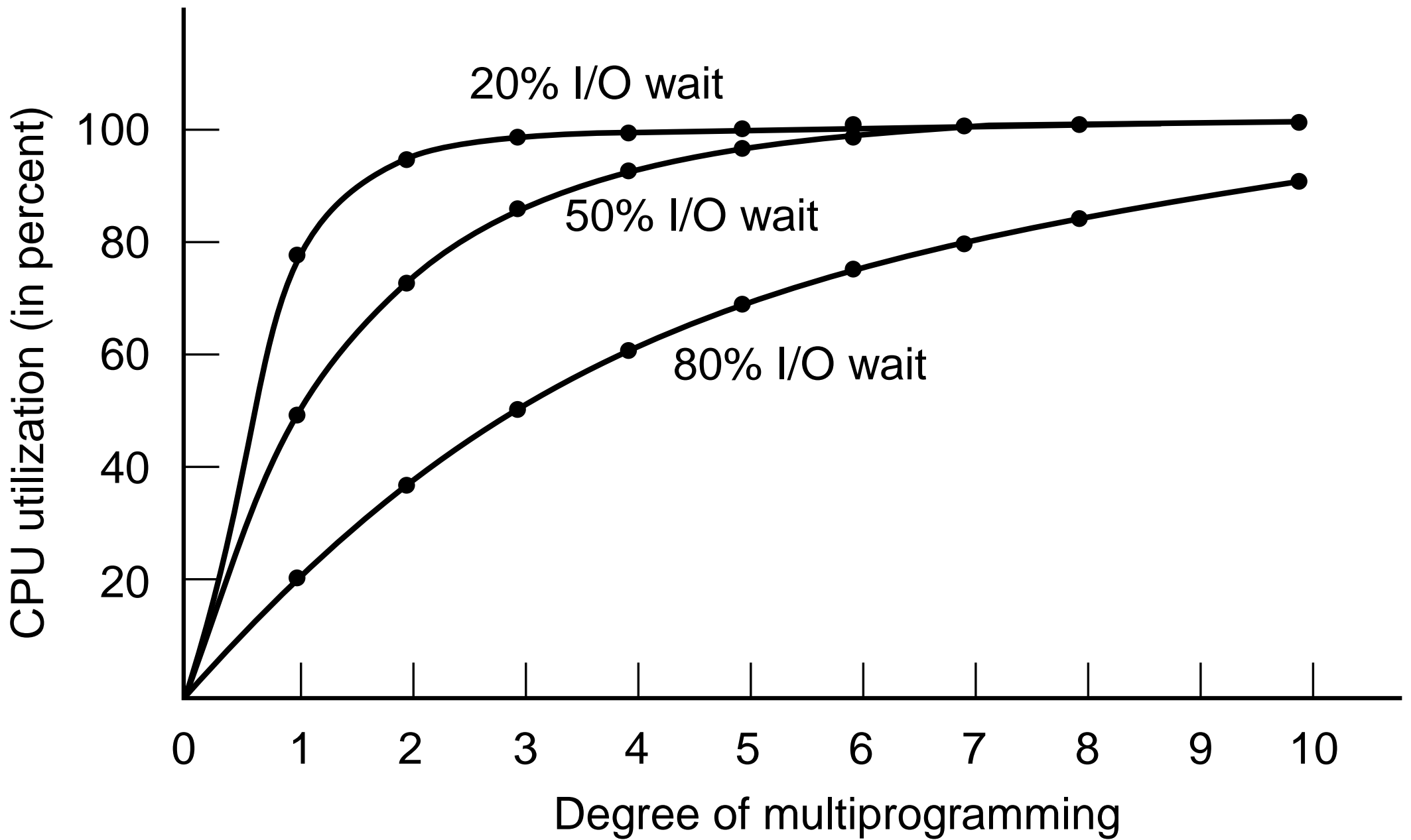
- È gestito dallo scheduler di medio termine
- Allo swap-in, il processo deve essere ricaricato esattamente nelle stesse regioni di memoria, a meno che non ci sia un binding dinamico
- *Roll out, roll in*: variante dello swapping usata per algoritmi di scheduling (a medio termine) a priorità: processi a bassa priorità vengono riversati per permettere il ripristino dei processi a priorità maggiore.
- La maggior parte del tempo di swap è nel trasferimento da/per il disco, che è proporzionale alla dimensione della memoria swappata.
- Per essere swappabile, un processo deve essere “inattivo”: buffer di I/O asincrono devono rimanere in memoria, strutture in kernel devono essere rilasciate, etc.
- Attualmente, lo swapping standard non viene impiegato—troppo costoso.
- Versioni modificate di swapping erano implementate in molti sistemi, es. primi Unix, Windows 3.x.

Multiprogrammazione

- La monoprogrammazione non sfrutta la CPU
- Idea: se un processo usa la CPU al 20%, 5 processi la usano al 100%
- Più precisamente, sia p la percentuale di tempo in attesa di I/O di un processo (probabilità che il processo sia in attesa). Con n processi (indipendenti):

$$\text{utilizzo CPU} = 1 - p^n$$

- Maggiore il *grado di multiprogrammazione*, maggiore l'utilizzo della CPU



Se attesa=80%, almeno 10 processi per ottenere il 90% di uso della CPU

Uso del modello

- Il modello precedente è impreciso (i processi non sono indipendenti)
- Può essere utile tuttavia per stimare l'opportunità di upgrade.
- Esempio: Memoria=32MB, sistema operativo=16MB, processi utenti=4MB, attesa media=80
 - Memoria per utenti = 16MB: grado = 4, utilizzo CPU =60%
 - Se aggiungo 16MB agli utenti = 32MB: grado = 8, utilizzo CPU =83% (+38%)
 - Se aggiungo altri 12MB= 48MB: grado = 12, utilizzo CPU =93% (+12%)
 - La seconda upgrade non conviene

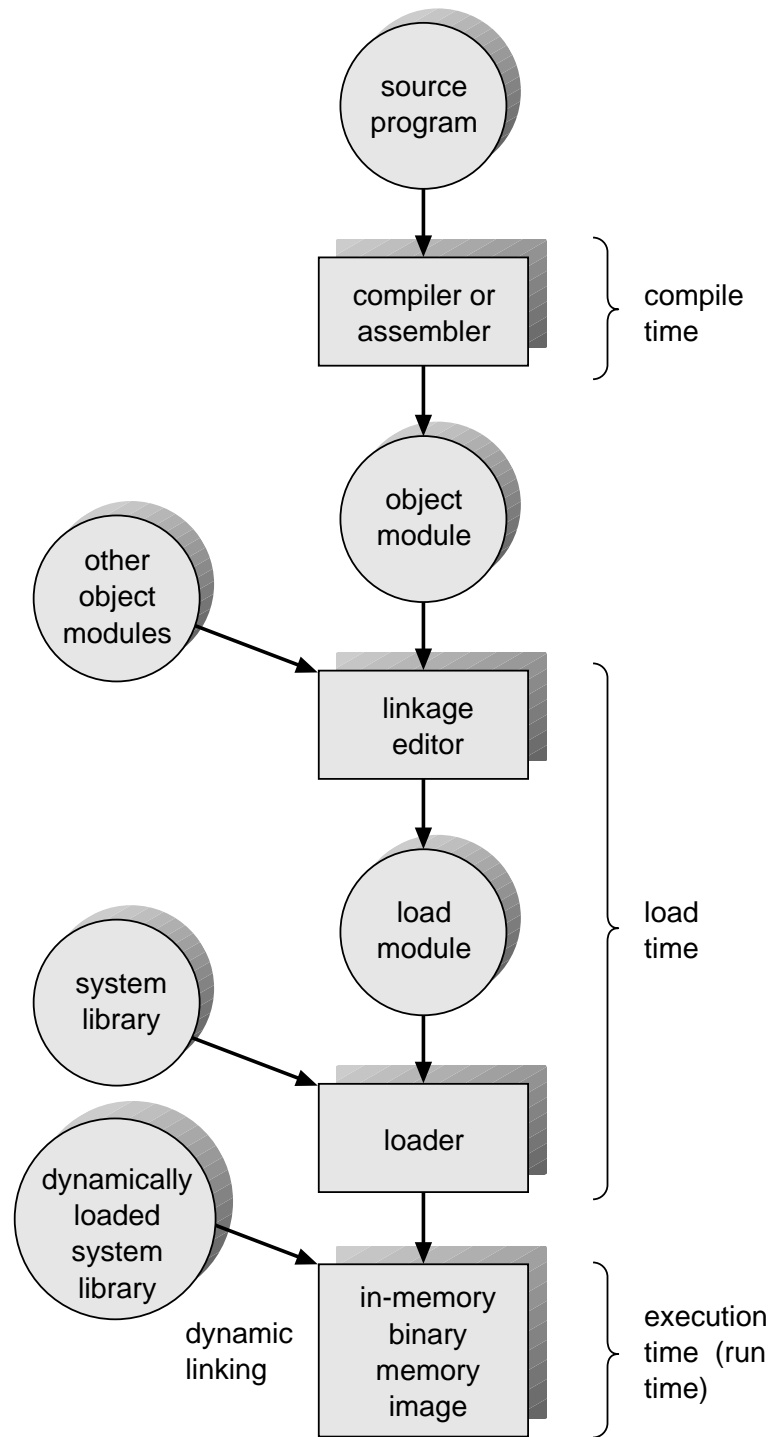
Multiprogrammazione (cont)

- Ogni programma deve essere portato in memoria e posto nello spazio indirizzi di un processo, per poter essere eseguito.
- *Coda in input*: l'insieme dei programmi su disco in attesa di essere portati in memoria per essere eseguiti.
- La selezione è fatta dallo scheduler di lungo termine (se c'è).
- Sorgono problemi di *rilocazione* e *protezione*

Binding degli indirizzi

L'associazione di istruzioni e dati a indirizzi di memoria può avvenire al

- **Compile time:** Se le locazioni di memoria sono note a priori, si può produrre del codice *assoluto*. Deve essere ricompilato ogni volta che si cambia locazione di esecuzione.
- **Load time:** La locazione di esecuzione non è nota a priori; il compilatore genera codice *rilocabile* la cui posizione in memoria viene decisa al momento del caricamento. Non può essere cambiata durante l'esecuzione.
- **Execution time:** L'associazione è fatta durante l'esecuzione. Il programma può essere spostato da una zona all'altra durante l'esecuzione. Necessita di un supporto hardware speciale per gestire questa rilocazione (es. registri *base e limite*).



Caricamento dinamico

- Un segmento di codice (eg. routine) non viene caricato finché non serve (la routine viene chiamata).
- Migliore utilizzo della memoria: il codice mai usato non viene caricato.
- Vantaggioso quando grosse parti di codice servono per gestire casi infrequenti (e.g., errori)
- Non serve un supporto specifico dal sistema operativo: può essere realizzato completamente a livello di linguaggio o di programma.
- Il sistema operativo può tuttavia fornire delle librerie per facilitare il caricamento dinamico.

Collegamento dinamico

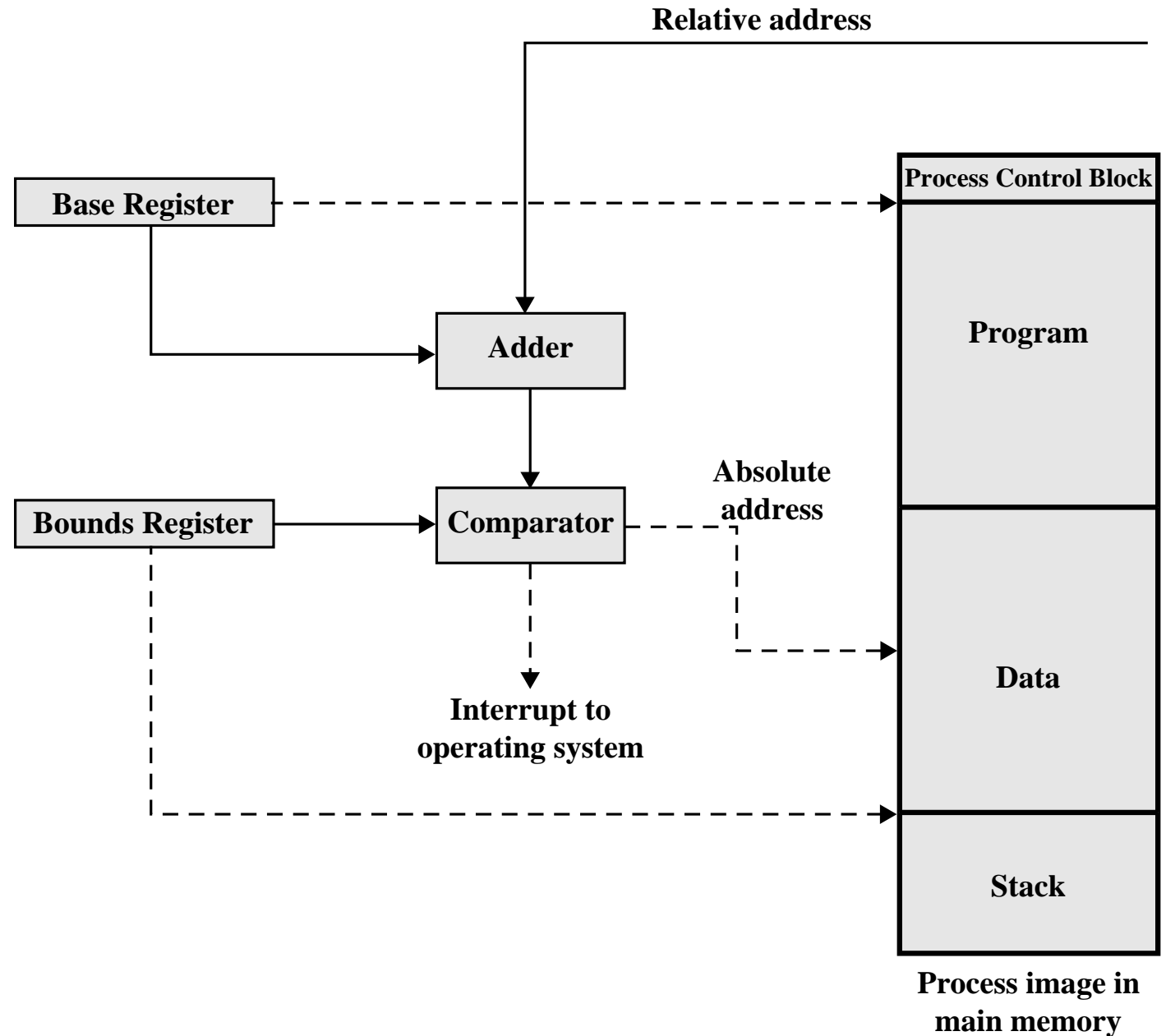
- **Linking dinamico:** le librerie vengono collegate all'esecuzione. Esempi: le .so su Unix, le .DLL su Windows.
- Nell'eseguibile si inseriscono piccole porzioni di codice, dette *stub*, che servono per localizzare la routine.
- Alla prima esecuzione, si carica il segmento se non è presente in memoria, e lo stub viene rimpiazzato dall'indirizzo della routine e si salta alla routine stessa.
- Migliore sfruttamento della memoria: il segmento di una libreria può essere condiviso tra più processi.
- Utili negli aggiornamenti delle librerie (ma bisogna fare attenzione a tener traccia delle versioni!)
- Richiede un supporto da parte del sistema operativo per far condividere segmenti di codice tra più processi.

Spazi di indirizzi logici e fisici

- Il concetto di *spazio indirizzi logico* che viene legato ad uno *spazio indirizzi fisico* diverso e separato è fondamentale nella gestione della memoria.
 - *Indirizzo logico*: generato dalla CPU. Detto anche *indirizzo virtuale*.
 - *Indirizzo fisico*: indirizzo visto dalla memoria.
- Indirizzi logici e fisici coincidono nel caso di binding al compile time o load time
- Possono essere differenti nel caso di binding al tempo di esecuzione. Necessita di un hardware di traduzione.

Memory-Management Unit (MMU)

- È un dispositivo hardware che associa al run time gli indirizzi logici a quelli fisici.
- Nel caso più semplice, il valore del registro di rilocazione viene sommato ad ogni indirizzo richiesto da un processo.
- Il programma utente vede solamente gli indirizzi logici; non vede *mai* gli indirizzi reali, fisici.

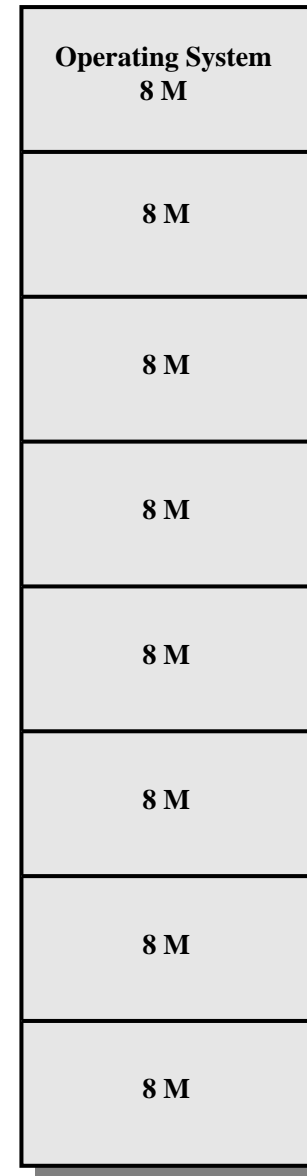


Allocazione contigua

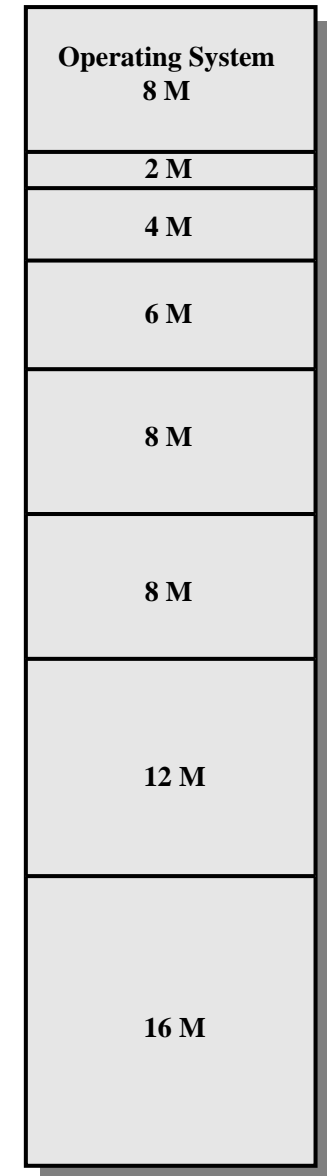
- La memoria è divisa in (almeno) due partizioni:
 - Sistema operativo residente, normalmente nella zona bassa degli indirizzi assieme al vettore delle interruzioni.
 - Spazio per i processi utente — tutta la memoria rimanente.
- Allocazione a partizione singola
 - Un processo è contenuto tutto in una sola partizione
 - Schema di protezione con *registri di rilocalizzazione e limite*, per proteggere i processi l'uno dall'altro e il kernel da tutti.
 - Il registro di rilocalizzazione contiene il valore del primo indirizzo fisico del processo; il registro limite contiene il range degli indirizzi logici.
 - Questi registri sono contenuti nella MMU e vengono caricati dal kernel ad ogni context-switch.

Allocazione contigua: partizionamento statico

- La memoria disponibile è divisa in partizioni fisse (uguali o diverse)
- Il sistema operativo mantiene informazioni sulle partizioni allocate e quelle libere
- Quando arriva un processo, viene scelta una partizione tra quelle libere e completamente allocatagli
- Porta a **frammentazione interna**: la memoria allocata ad un processo è superiore a quella necessaria, e quindi parte non è usata.
- Oggi usato solo su hardware povero

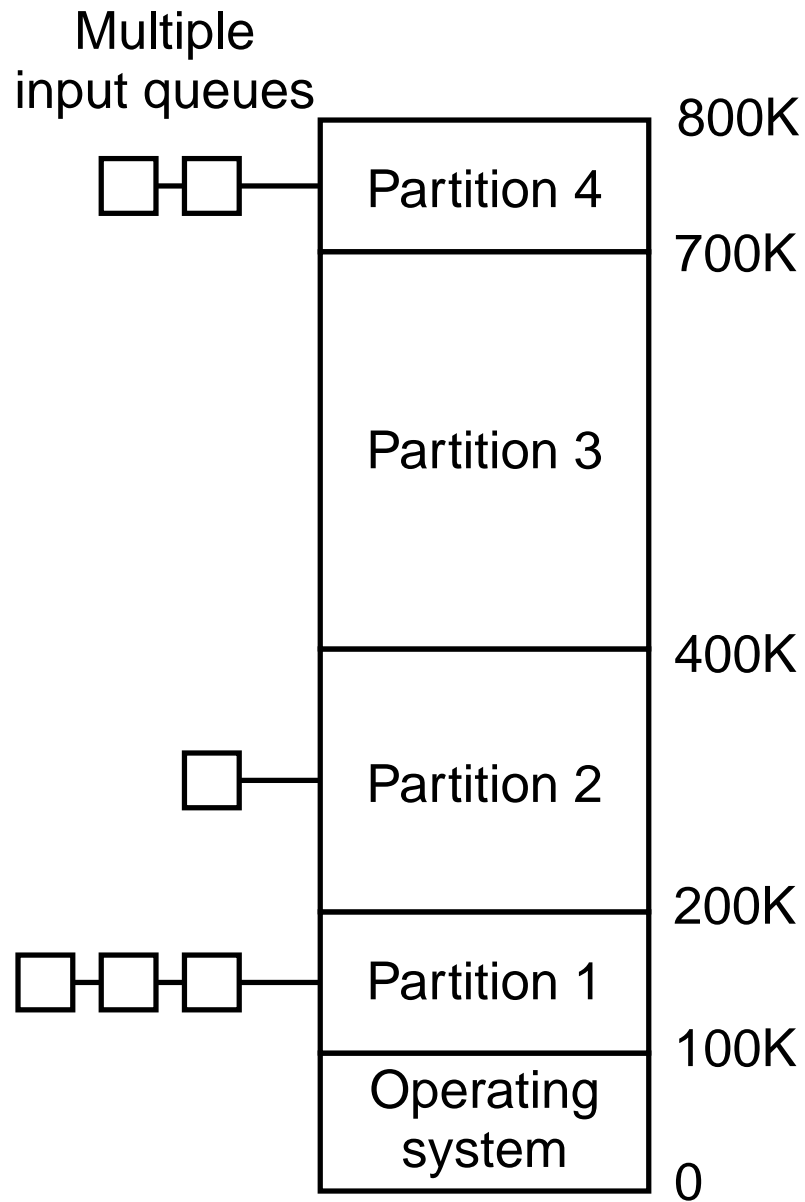


(a) Equal-size partitions

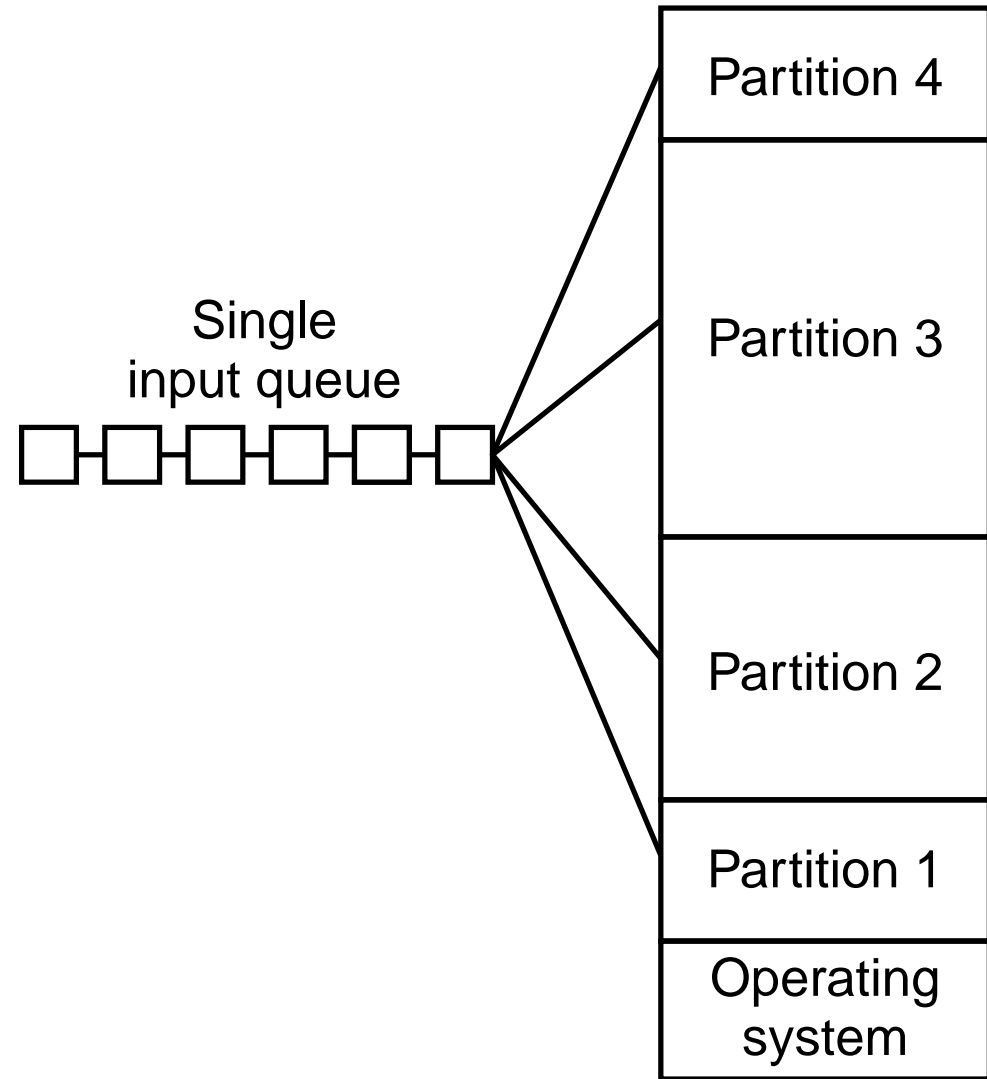


(b) Unequal-size partitions

Allocazione contigua: code di input



(a)

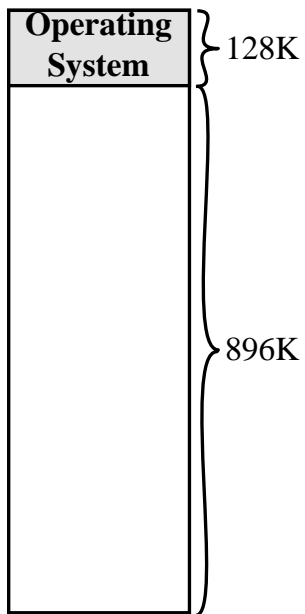


(b)

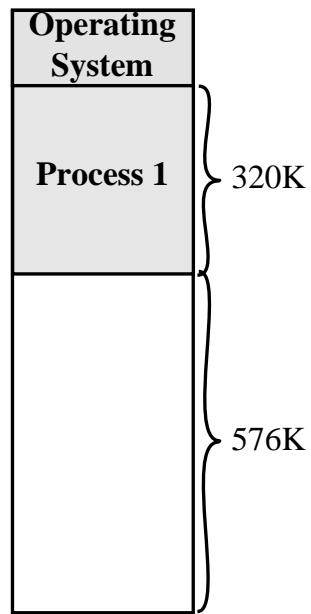
- Quando arriva un processo, viene scelta una partizione tra quelle libere e completamente allocatagli
- Una coda per ogni partizione: possibilità di inutilizzo di memoria
- Una coda per tutte le partizioni: come scegliere il job da allocare?
 - first-fit: per ogni buco, il primo che ci entra
 - best-fit: il più grande che ci entra. Penalizza i job piccoli (che magari sono interattivi. . .)

Allocazione contigua: partizionamento dinamico

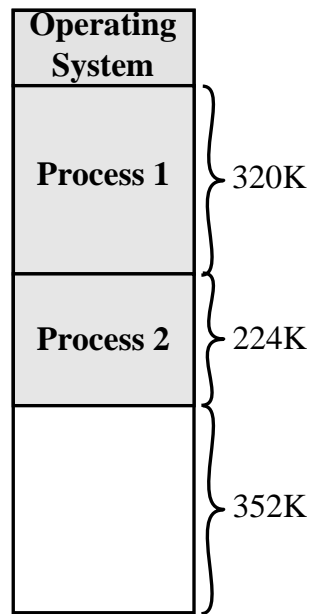
- Le partizioni vengono decise al runtime
- *Hole*: blocco di memoria libera. Buchi di dimensione variabile sono sparpagliati lungo la memoria.
- Il sistema operativo mantiene informazioni sulle partizioni allocate e i buchi
- Quando arriva un processo, gli viene allocato una partizione all'interno di un buco sufficientemente largo.



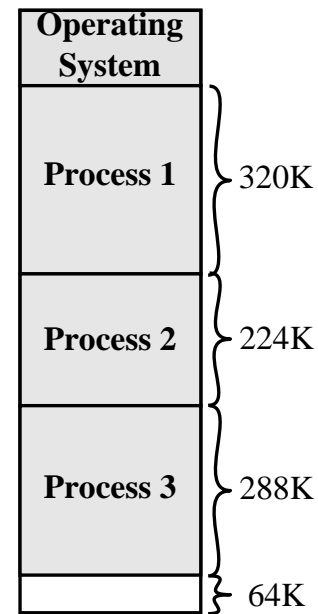
(a)



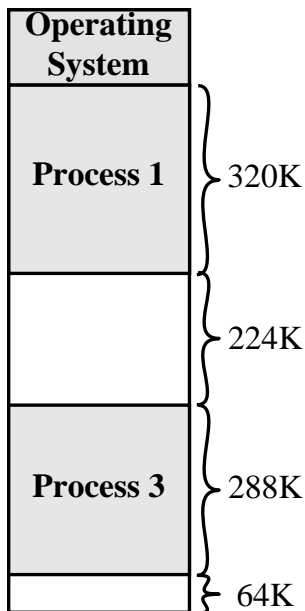
(b)



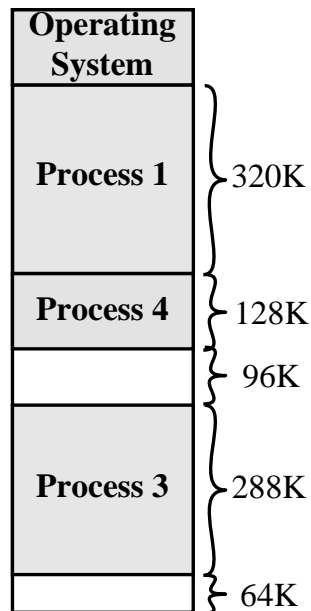
(c)



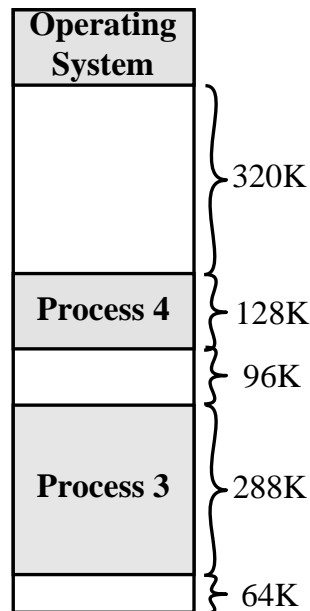
(d)



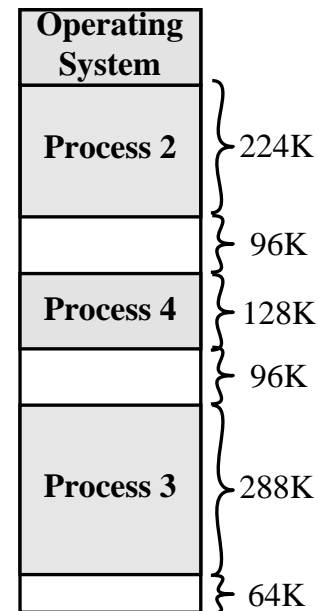
(e)



(f)



(g)



(h)

Allocazione contigua: partizionamento dinamico (cont.)

- Hardware necessario: niente se la rilocazione non è dinamica; base-register se la rilocazione è dinamica.
- Non c'è frammentazione interna
- Porta a **frammentazione esterna**: può darsi che ci sia memoria libera sufficiente per un processo, ma non è contigua.
- La frammentazione esterna si riduce con la *compattazione*
 - riordinare la memoria per agglomerare tutti i buchi in un unico buco
 - la compactazione è possibile solo se la rilocazione è dinamica
 - Problemi con I/O: non si possono spostare i buffer durante operazioni di DMA. Due possibilità:
 - * Mantenere fissi i processi coinvolti in I/O
 - * Eseguire I/O solo in buffer del kernel (che non si sposta mai)

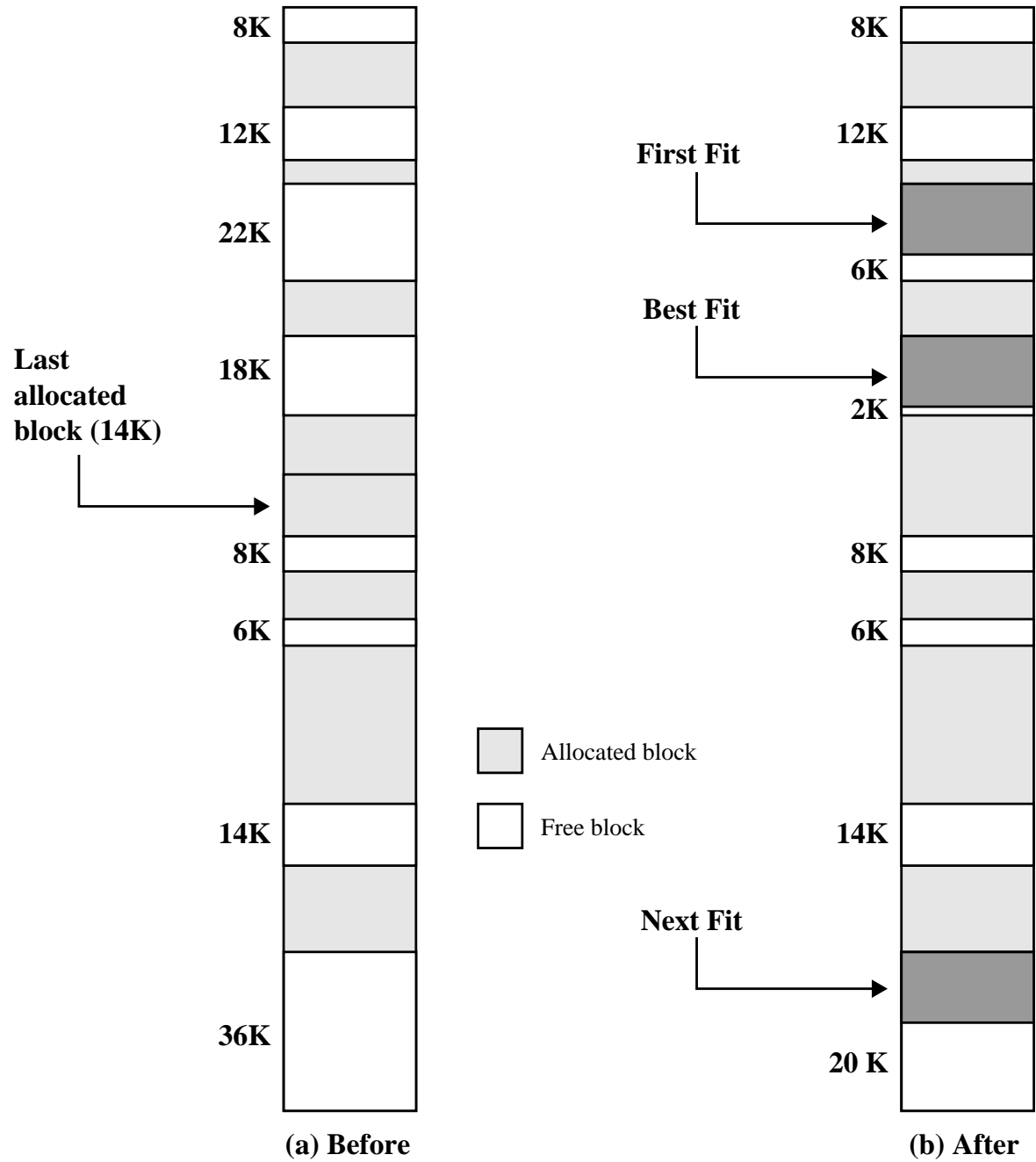
Allocazione contigua: partizionamento dinamico (cont.)

Come soddisfare una richiesta di dimensione n ?

- **First-fit:** Alloca il *primo* buco sufficientemente grande
- **Next-fit:** Alloca il *primo* buco sufficientemente grande a partire dall'ultimo usato.
- **Best-fit:** Alloca il *più piccolo* buco sufficientemente grande. Deve scandire l'intera lista (a meno che non sia ordinata). Produce il più piccolo buco di scarto.
- **Worst-fit:** Alloca il *più grande* buco sufficientemente grande. Deve scandire l'intera lista (a meno che non sia ordinata). Produce il più grande buco di scarto.

In generale, gli algoritmi migliori sono il first-fit e il next-fit. Best-fit tende a frammentare molto. Worst-fit è più lento.

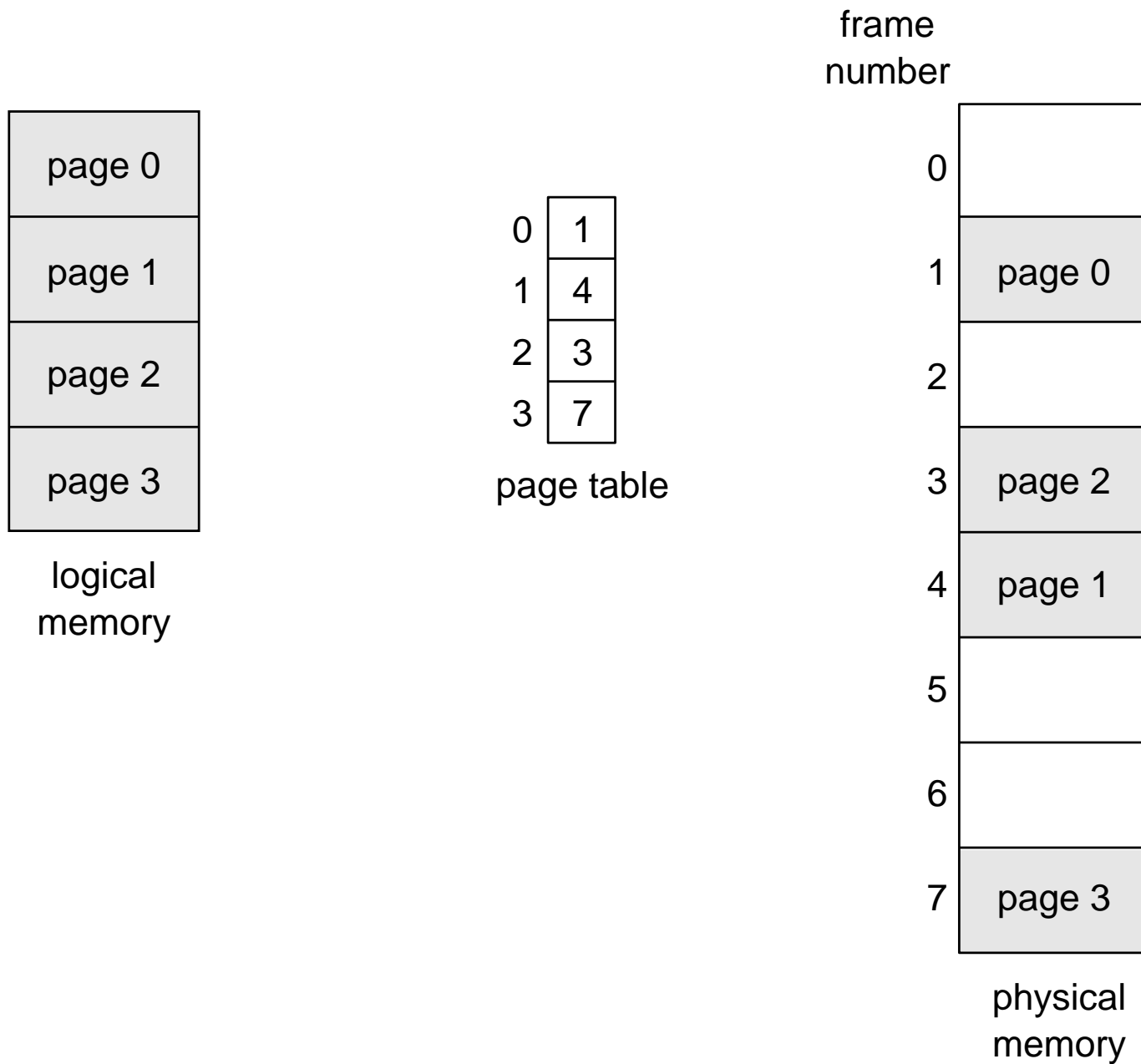
Allocazione contigua: esempi di allocazione



Allocazione non contigua: Paginazione

- Lo spazio logico di un processo può essere non contiguo: ad un processo viene allocata memoria fisica dovunque essa si trovi.
- Si divide la memoria fisica in *frame*, blocchi di dimensione fissa (una potenza di 2, tra 512 e 8192 byte)
- Si divide la memoria logica in *pagine*, della stessa dimensione
- Il sistema operativo tiene traccia dei frame liberi
- Per eseguire un programma di n pagine, servono n frame liberi in cui caricare il programma.
- Si imposta una *page table* per tradurre indirizzi logici in indirizzi fisici.
- Non esiste frammentazione esterna
- Ridotta frammentazione interna

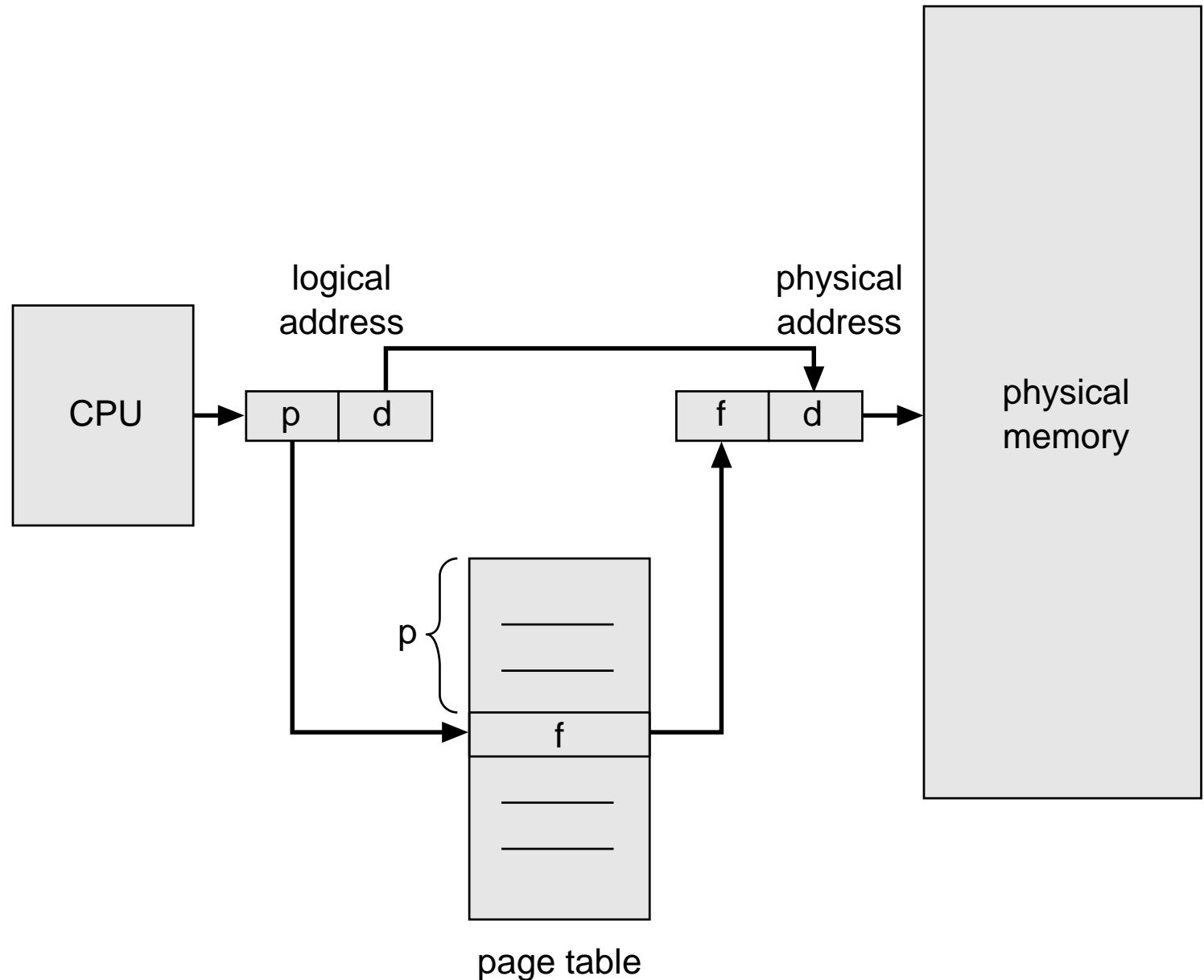
Esempio di paginazione



Schema di traduzione degli indirizzi

L'indirizzo generato dalla CPU viene diviso in

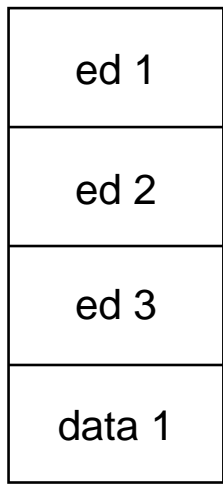
- *Numero di pagina p*: usato come indice in una *page table* che contiene il numero del frame contenente la pagina *p*.
- *Offset di pagina d*: combinato con il numero di frame fornisce l'indirizzo fisico da inviare alla memoria.



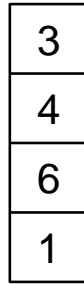
Paginazione: condivisione

La paginazione permette la condivisione del codice

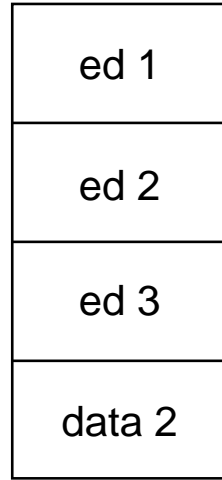
- Una sola copia di codice read-only può essere condivisa tra più processi. Il codice deve essere *rientrante* (separare codice eseguibile da record di attivazione). Es.: editors, shell, compilatori, . . .
- Il codice condiviso appare nelle stesse locazioni logiche per tutti i processi che vi accedono
- Ogni processo mantiene una copia separata dei propri dati



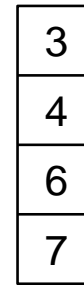
process P_1



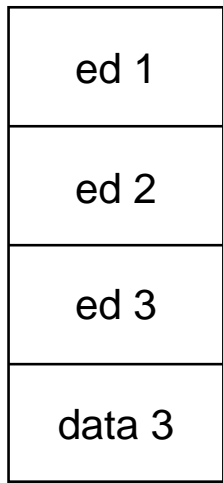
page table
for P_1



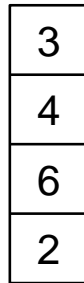
process P_2



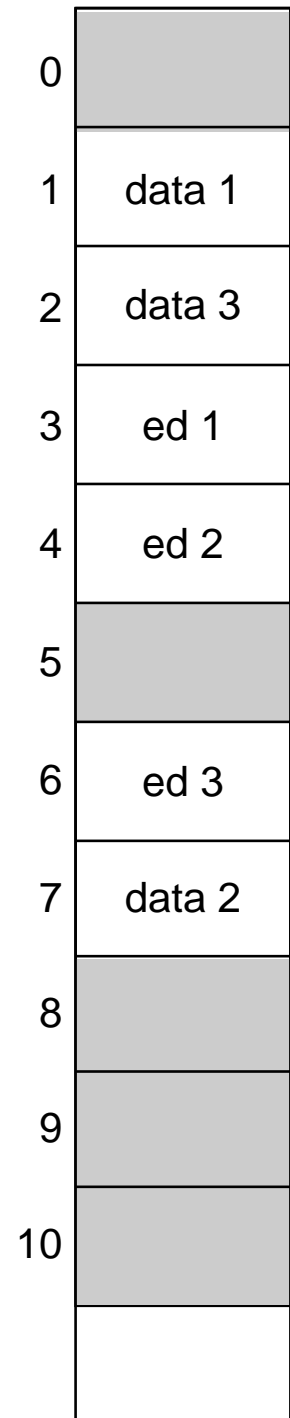
page table
for P_2



process P_3



page table
for P_3

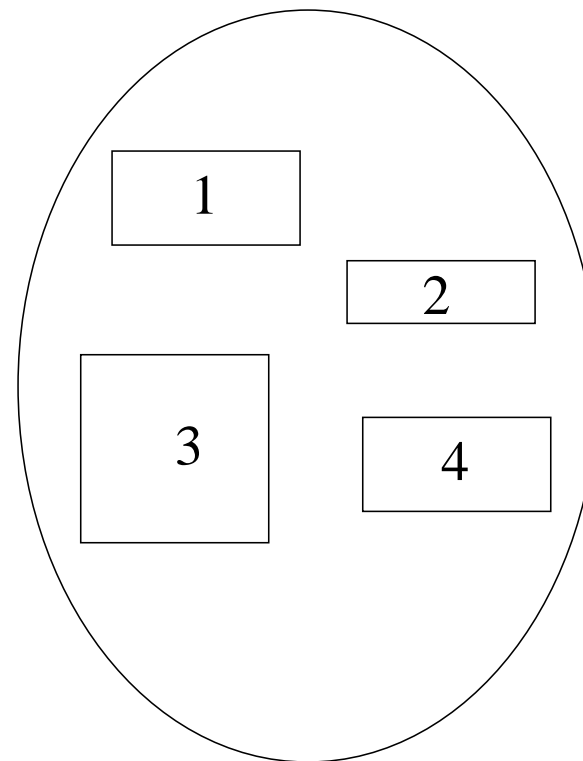


Paginazione: protezione

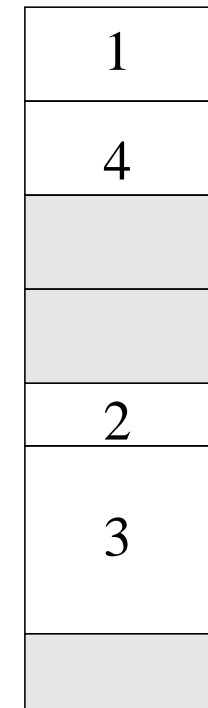
- La protezione della memoria è implementata associando bit di protezione ad ogni frame.
- *Valid* bit collegato ad ogni entry nella page table
 - “valid” = indica che la pagina associata è nello spazio logico del processo, e quindi è legale accedervi
 - “invalid” = indica che la pagina non è nello spazio logico del processo
⇒ violazione di indirizzi (Segment violation)

Allocazione non contigua: Segmentazione

- È uno schema di MM che supporta la visione *utente* della memoria
- Un programma è una collezione di segmenti. Un segmento è una unità logica di memoria; ad esempio: programma principale, procedure, funzioni, variabili locali, variabili globali stack, tabella dei simboli memoria condivisa, ...



user space

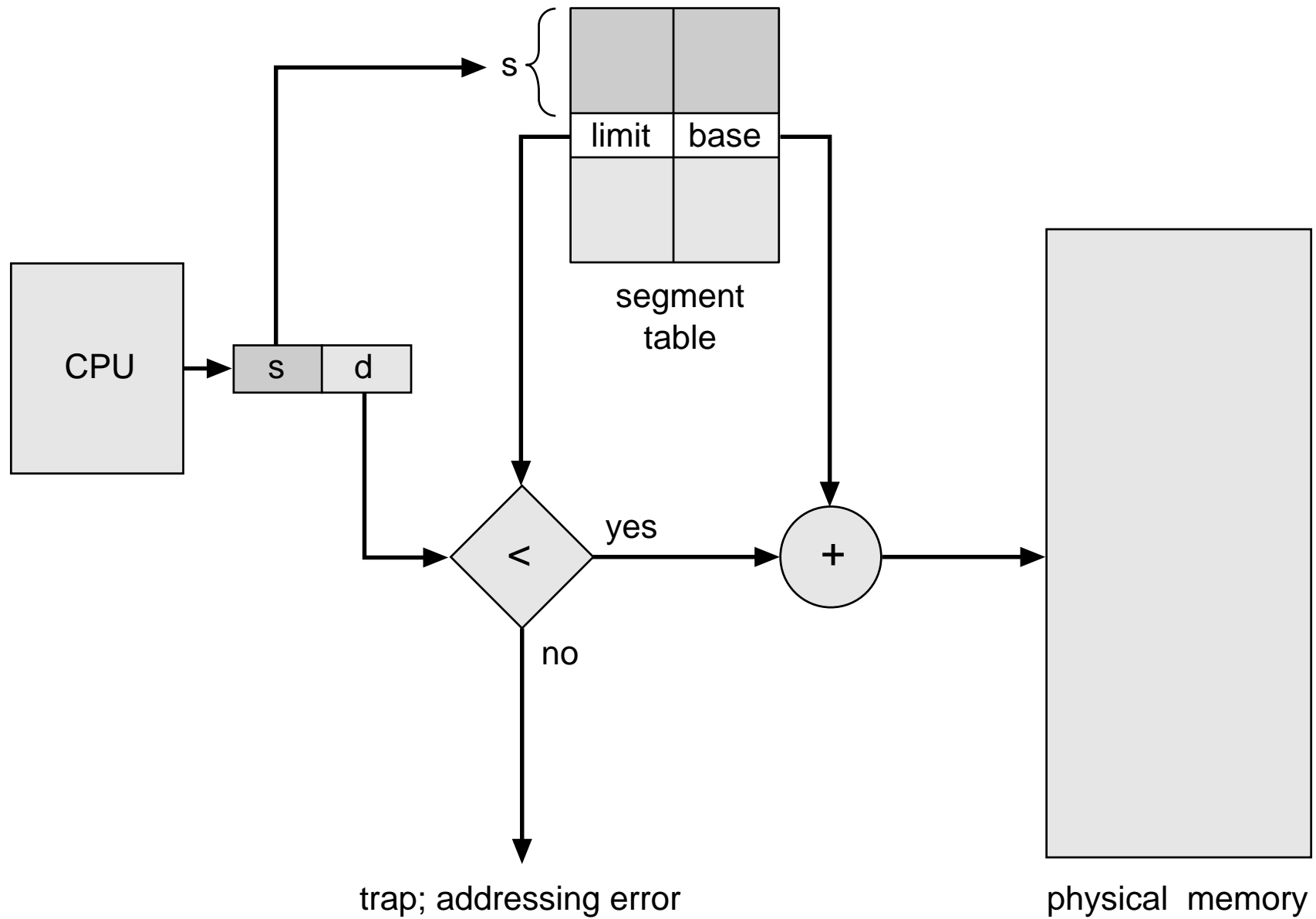


physical memory

Architettura della Segmentazione

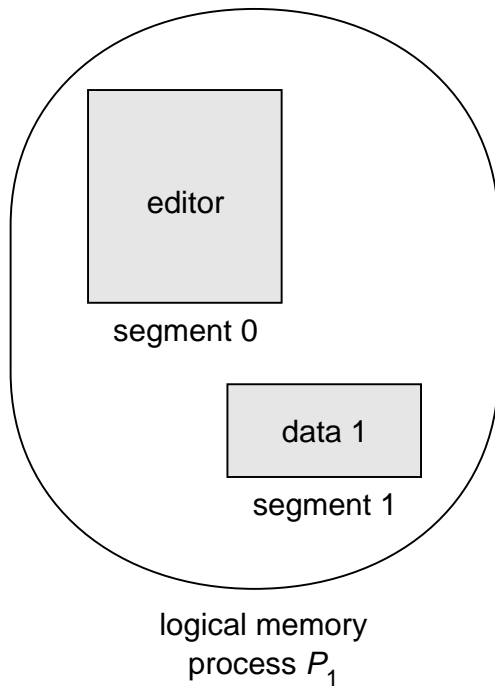
- L'indirizzo logico consiste in un coppia
 <segment-number, offset>.
- La *segment table* mappa gli indirizzi bidimensionali dell'utente negli indirizzi fisici unidimensionali. Ogni entry ha
 - *base*: indirizzo fisico di inizio del segmento
 - *limit*: lunghezza del segmento
- *Segment-table base register (STBR)* punta all'inizio della tabella dei segmenti
- *Segment-table length register (STLR)* indica il numero di segmenti usati dal programma
 segment number s è legale se $s < STLR$.

Hardware per la segmentazione



Architettura della Segmentazione (cont.)

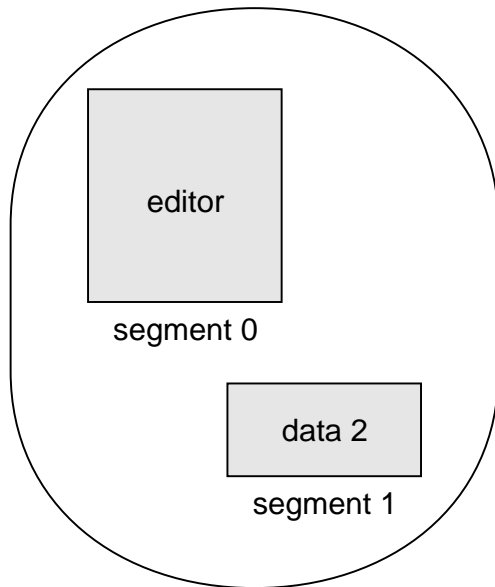
- Rilocazione
 - dinamica, attraverso tabella dei segmenti
- Condivisione
 - interi segmenti possono essere condivisi
- Allocazione
 - gli stessi algoritmi dell'allocazione contigua
 - frammentazione esterna; non c'è frammentazione interna
- Protezione: ad ogni entry nella segment table si associa
 - bit di validità: 0 \Rightarrow segmento illegale
 - privilegi di read/write/execute
- I segmenti possono cambiare di lunghezza durante l'esecuzione (es. lo stack): problema di allocazione dinamica di memoria.



logical memory
process P_1

	limit	base
0	25286	43062
1	4425	68348

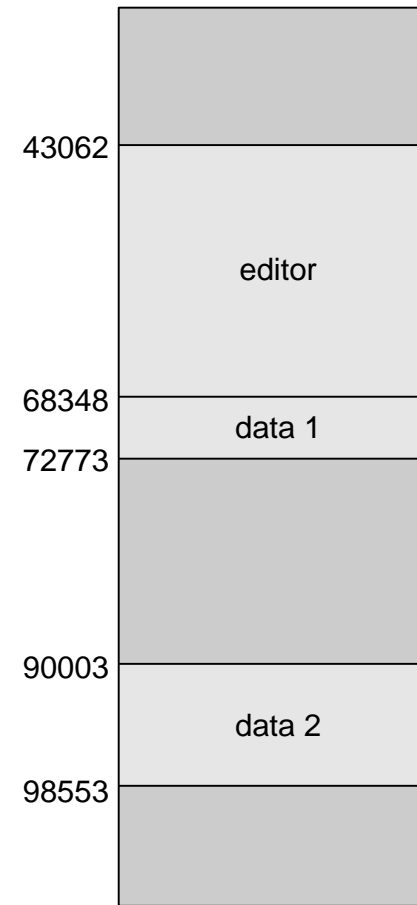
segment table
process P_1



logical memory
process P_2

	limit	base
0	25286	43062
1	8850	90003

segment table
process P_2

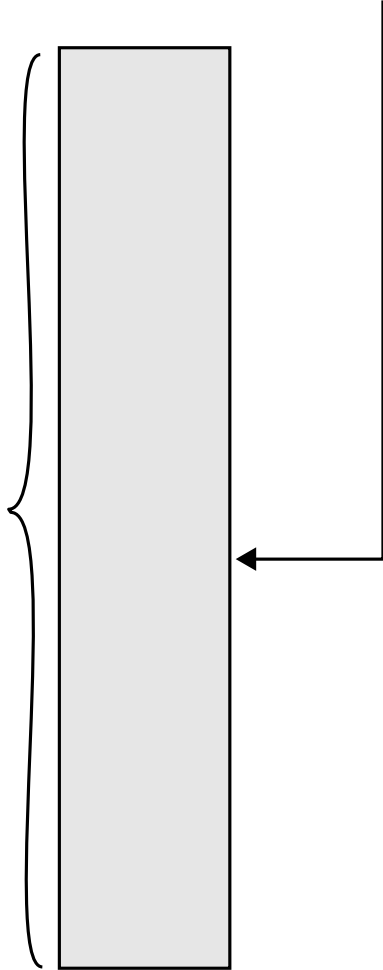


physical memory

Relative address = 1502

0000010111011110

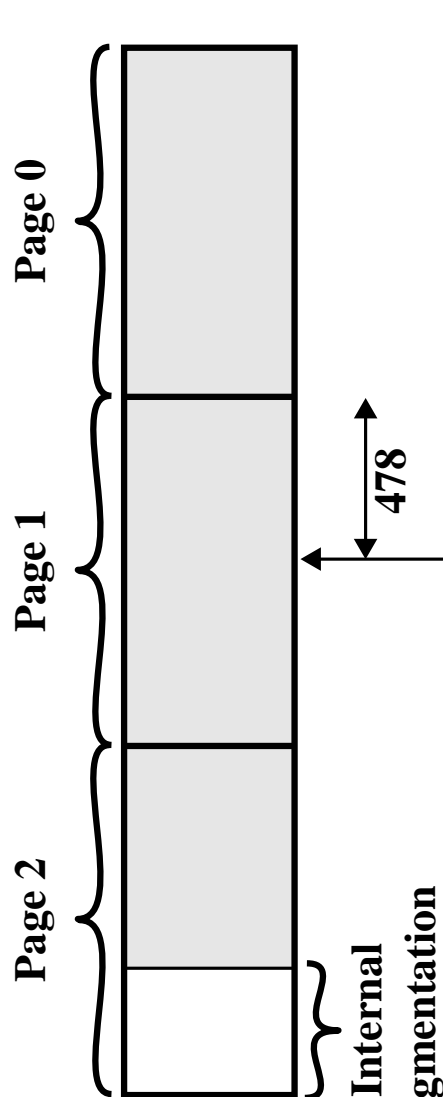
User process
(2700 bytes)



(a) Partitioning

Logical address =
Page# = 1, Offset = 478

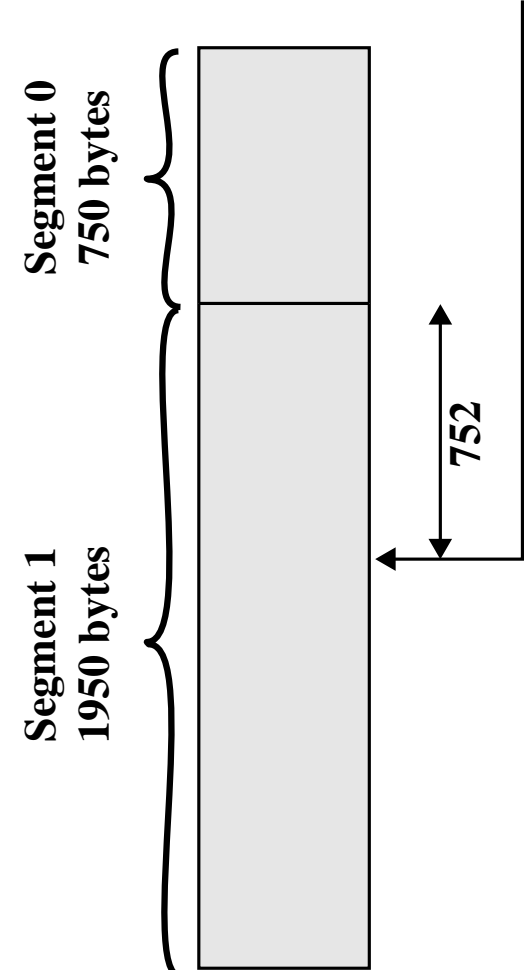
000001|0111011110



(b) Paging
(page size = 1K)

Logical address =
Segment# = 1, Offset = 752

0001|001011110000



(c) Segmentation

- Indirizzi usati nella paginazione

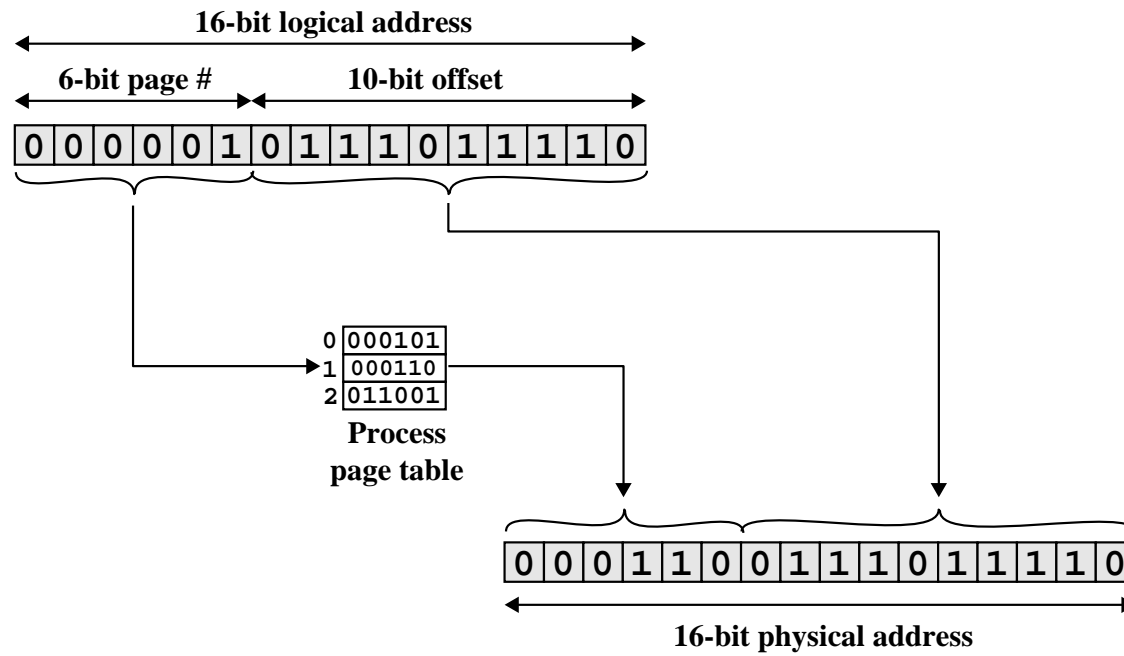
- Spazio di indirizzi logici = 2^m

- Dimensione di pagina = 2^n

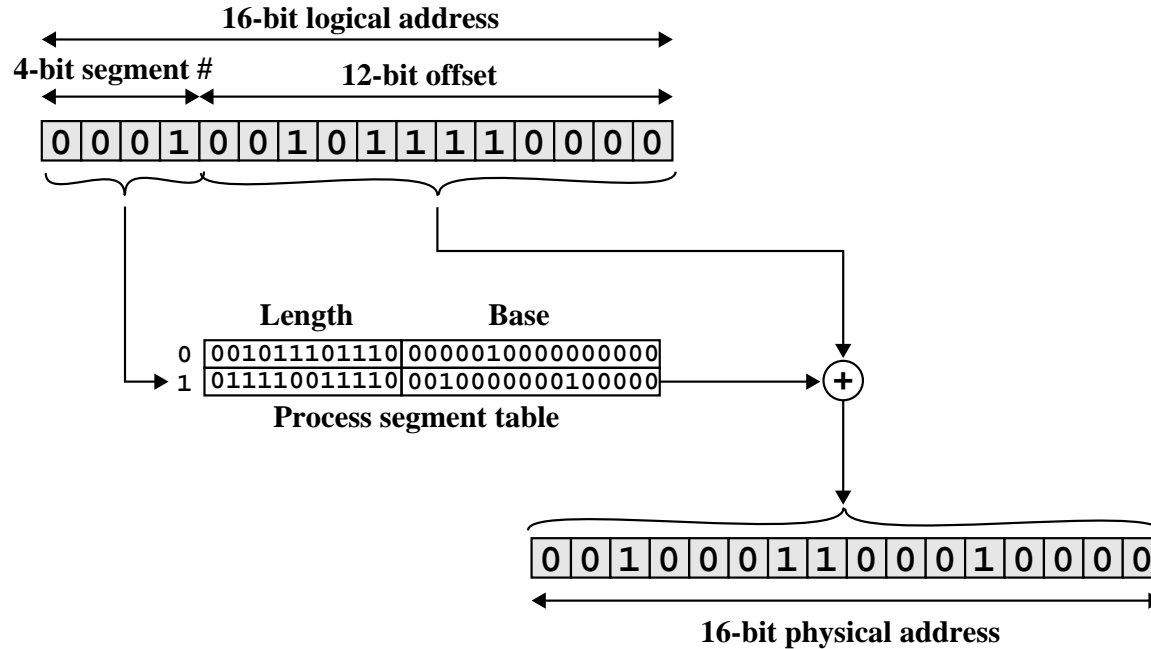
- Numero di pagine = 2^{m-n}

- $m - n$ bit più significativi di un indirizzo logico per il numero di pagina

- n bit meno significativi per offset



(a) Paging

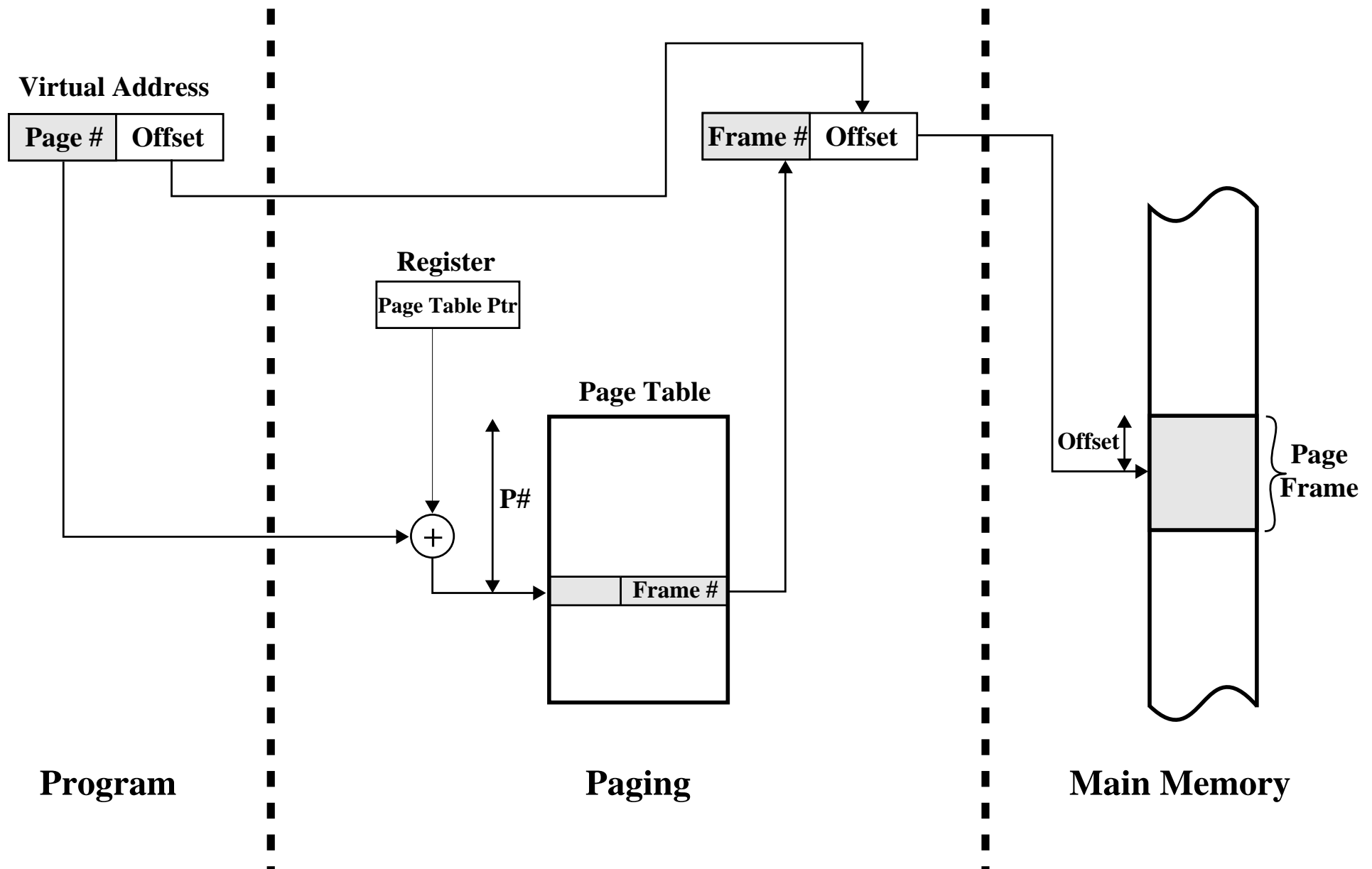


(b) Segmentation

Implementazione della Page Table

- Idealmente, la page table dovrebbe stare in registri veloci della MMU.
 - Costoso al context switch (carico/ricarico di tutta la tabella)
 - Improponibile se il numero delle pagine è elevato. Es: indirizzi virtuali a 32 bit, pagine di 4K (2^{12}): ci sono $2^{20} > 10^6$ entry.
 - Se usiamo 2byte per ogni entry (max RAM = 256M) abbiamo bisogno di $2^{21} = 2M$ in registri.
- La page table viene tenuta in memoria principale
 - *Page-table base register (PTBR)* punta all'inizio della page table
 - *Page-table length register (PTLR)* indica il numero di entry della page table

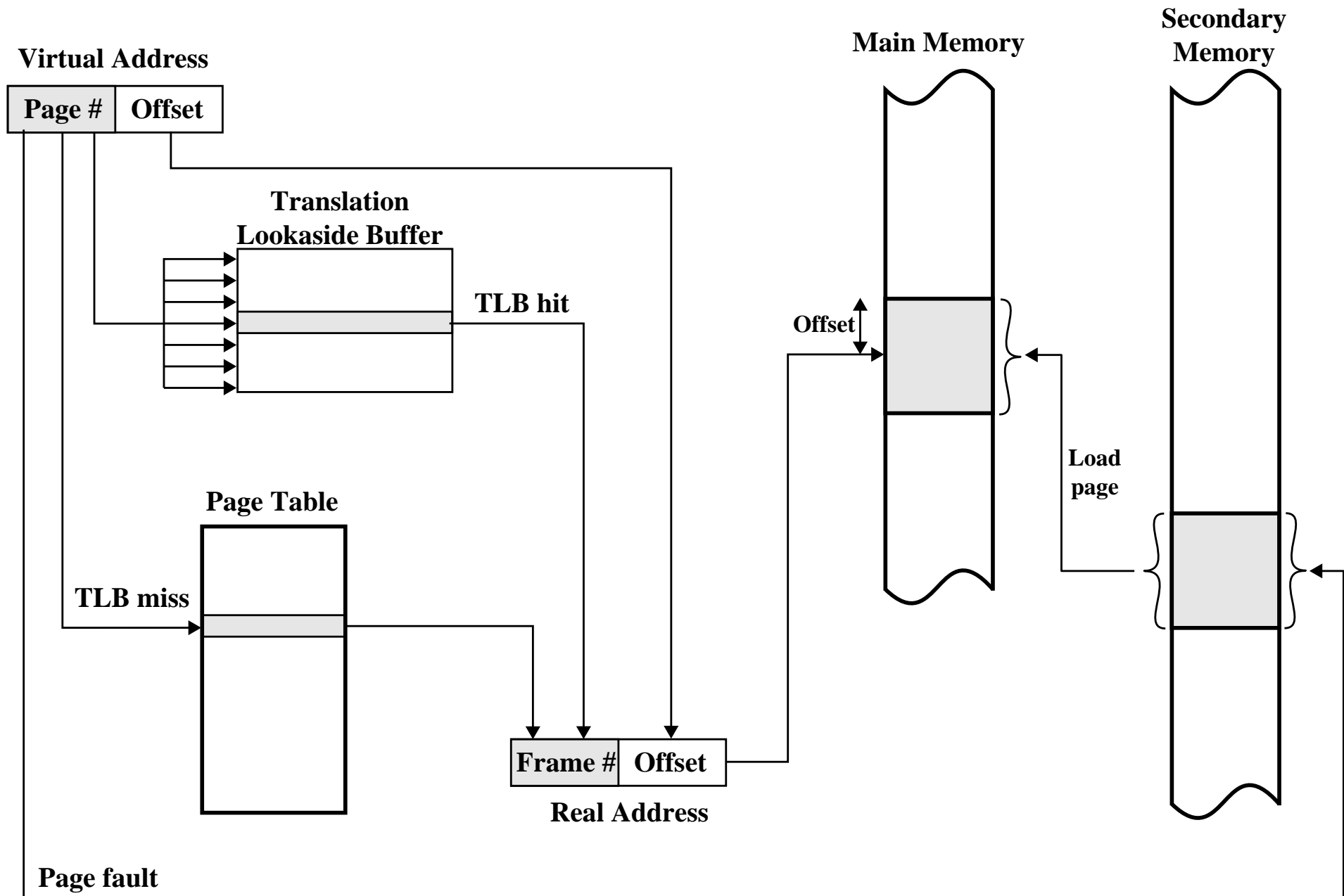
Paginazione con page table in memoria



Paginazione con page table in memoria (cont.)

- Rimane comunque un grande consumo di memoria (1 page table per ogni processo). Nell'es. di prima: 100 processi \Rightarrow 200M in page tables (su 256MB RAM complessivi).
- Ogni accesso a dati/istruzioni richiede 2 accessi alla memoria: uno per la page table e uno per i dati/istruzioni \Rightarrow degrado del 100%.
- Il doppio accesso alla memoria si riduce con una cache dedicata per le entry delle page tables: *registri associativi* detti anche *translation look-aside buffer (TLB)*.

Registri Associativi (TLB)



Traduzione indirizzo logico (A' , A'') con TLB

- Il virtual page number A' viene confrontato con tutte le entry contemporaneamente.
- Se A' è nel TLB (TLB hit), si usa il frame # nel TLB
- Altrimenti, la MMU esegue un normale lookup nelle page table in memoria, e sostituisce una entry della TLB con quella appena trovata
- Il S.O. viene informato solo nel caso di un page fault

Variante: software TLB

I TLB miss vengono gestiti direttamente dal S.O.

- nel caso di una TLB miss, la MMU manda un interrupt al processore (*TLB fault*)
- si attiva una apposita routine del S.O., che gestisce le page table e la TLB esplicitamente

Abbastanza efficiente con TLB suff. grandi (≥ 64 entries)

MMU estremamente semplice \Rightarrow lascia spazio sul chip per ulteriori cache

Molto usato (SPARC, MIPS, Alpha, PowerPC, HP-PA, Itanium...)

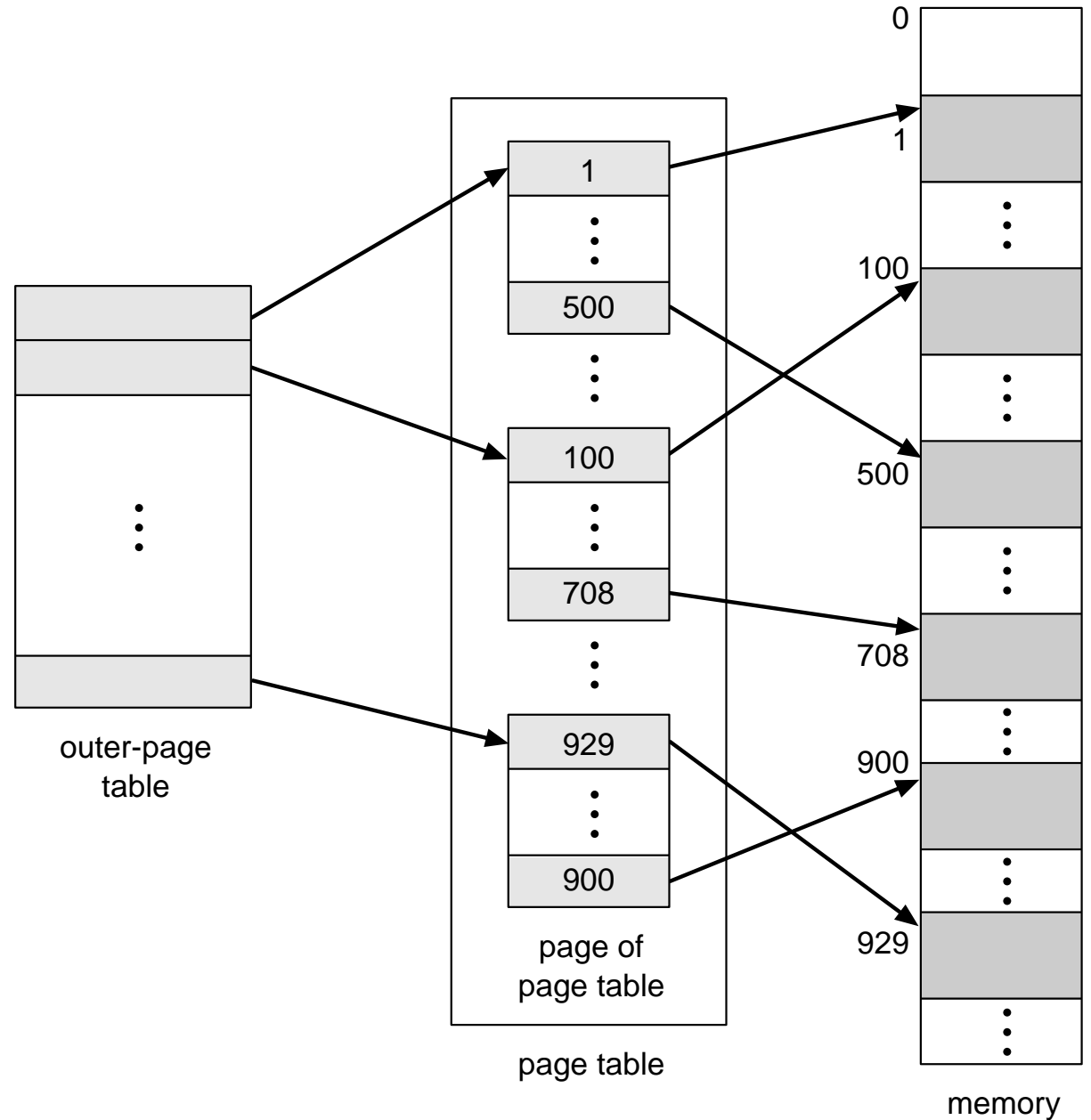
Tempo effettivo di accesso con TLB

- ϵ = tempo del lookup associativo
- t = tempo della memoria
- α = *Hit ratio*: percentuale dei page # reperiti nel TLB (dipende dalla grandezza del TLB, dalla natura del programma...)

$$EAT = (t + \epsilon)\alpha + (2t + \epsilon)(1 - \alpha) = (2 - \alpha)t + \epsilon$$

- In virtù del *principio di località*, l'hit ratio è solitamente alto
- Con $t = 50ns$, $\epsilon = 1ns$, $\alpha = 0.98$ si ha $EAT/t = 1.04$

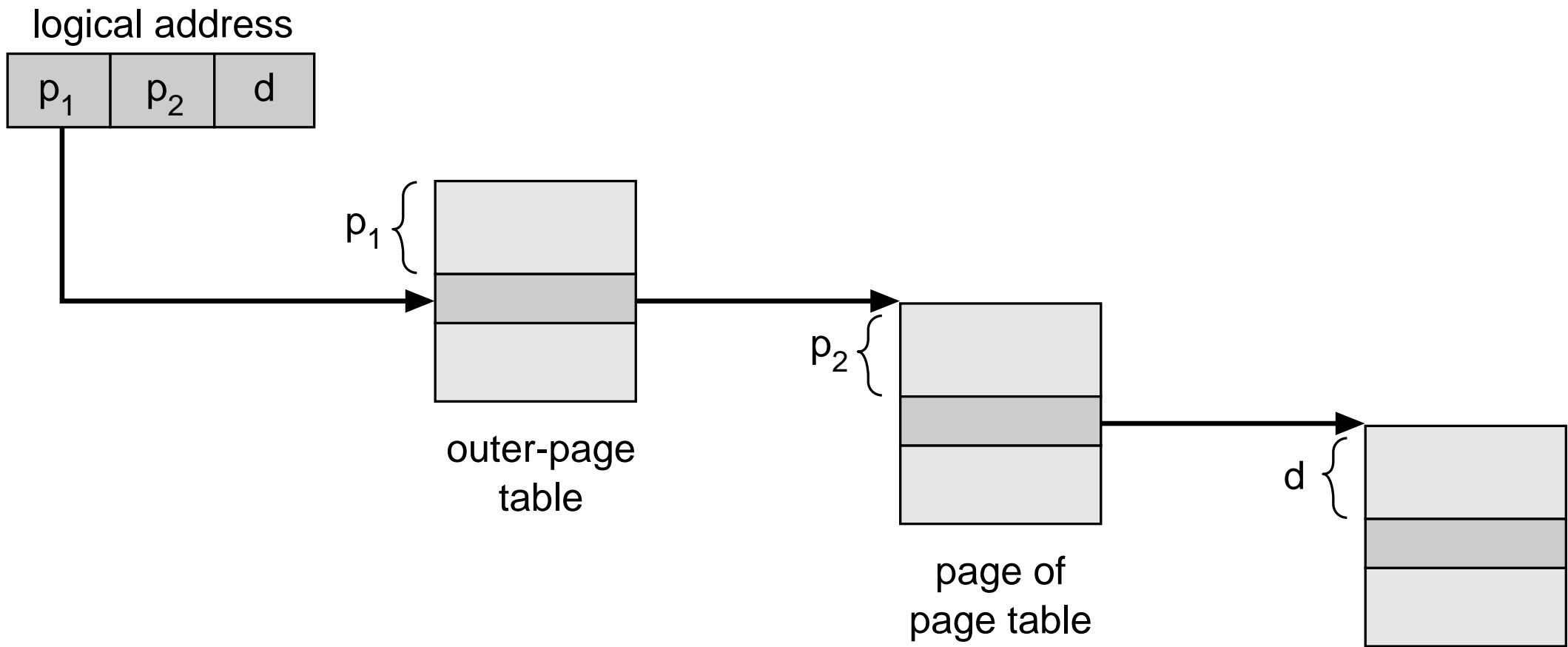
Paginazione a più livelli



Per ridurre l'occupazione della page table, si pagina la page table stessa. Solo le pagine effettivamente usate sono allocate in memoria RAM.

Esempio di paginazione a due livelli

- Un indirizzo logico (a 32 bit con pagine da 4K) è diviso in
 - un numero di pagina consistente in 20 bit
 - un offset di 12 bit
- La page table è paginata, quindi il numero di pagina è diviso in
 - un *directory number* di 10 bit
 - un *page offset* di 10 bit.



Performance della paginazione a più livelli

- Dato che ogni livello è memorizzato in RAM, la conversione dell'indirizzo logico in indirizzo fisico può necessitare di 4 accessi alla memoria (in caso di TLB miss: lookup in TLB, 2 accessi alla RAM per la rilocalizzazione, 1 accesso per leggere la casella di memoria)
- Il caching degli indirizzi di pagina permette di ridurre drasticamente l'impatto degli accessi multipli. Ad esempio se in caso di TLB miss servono 5 accessi alla memoria (ad es. 4 livelli di page tabling)

$$EAT = \alpha(t + \epsilon) + (1 - \alpha)(5t + \epsilon) = \epsilon + (5 - 4\alpha)t$$

- Nell'esempio di prima, con un hit rate del 98%: $EAT/t = 1.1$: 10% di degrado
- Schema molto adottato da CPU a 32 bit (IA32 (Pentium), 68000, SPARC a 32 bit, ...)

Tabella delle pagine invertita

- Una tabella con una entry per ogni *frame*, non per ogni page.
- Ogni entry consiste nel numero della pagina (virtuale) memorizzata in quel frame, con informazioni riguardo il processo che possiede la pagina.
- Diminuisce la memoria necessaria per memorizzare le page table, ma aumenta il tempo di accesso alla tabella.
- Questo schema è usato su diversi RISC a 32 bit (PowerPC), e tutti quelli a 64 bit (UltraSPARC, Alpha, HPPA, ...), ove una page table occuperebbe petabytes (es: a pagine da 4k: $8 \times 2^{52} = 32\text{PB}$ per ogni page table)

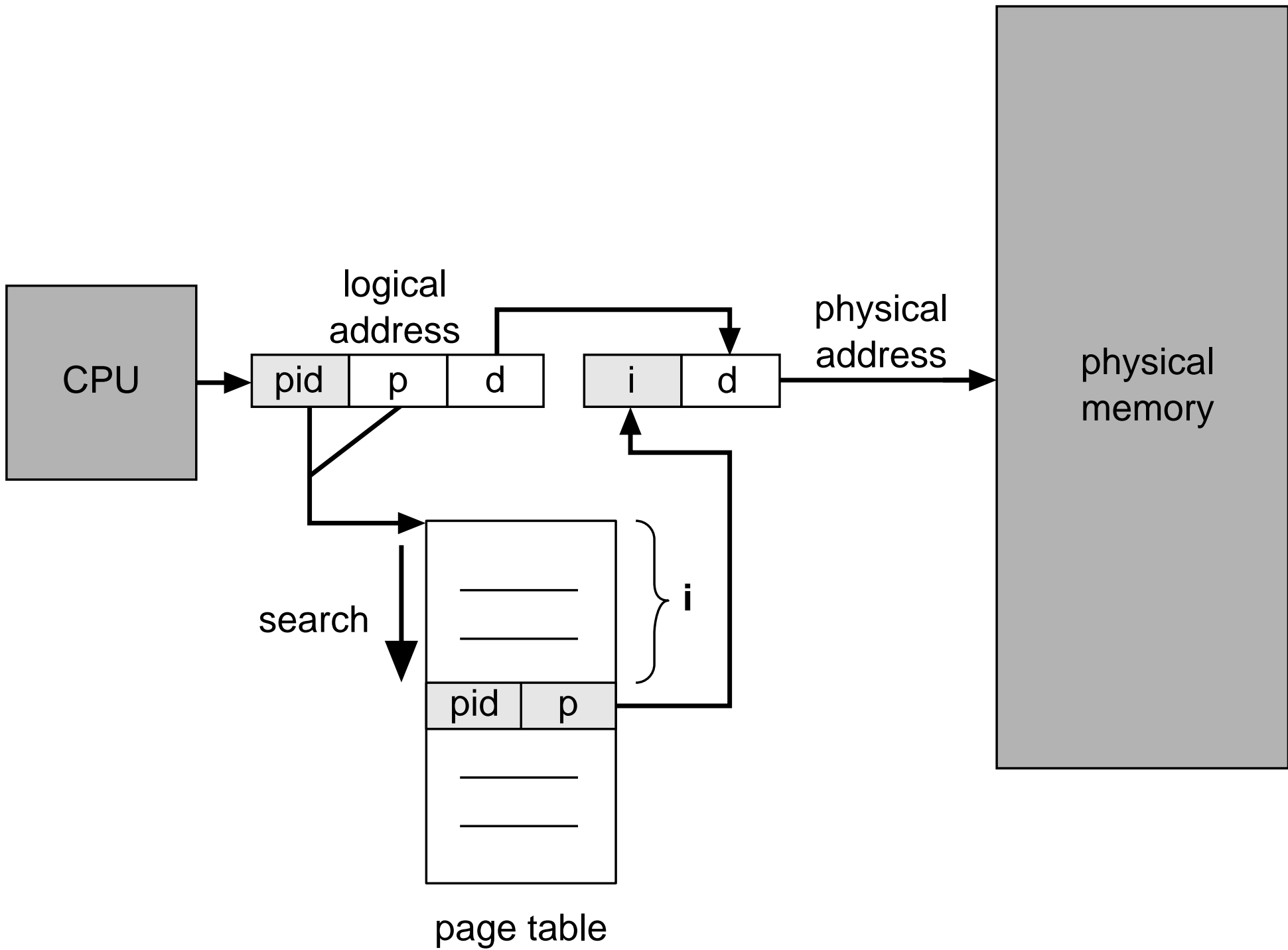
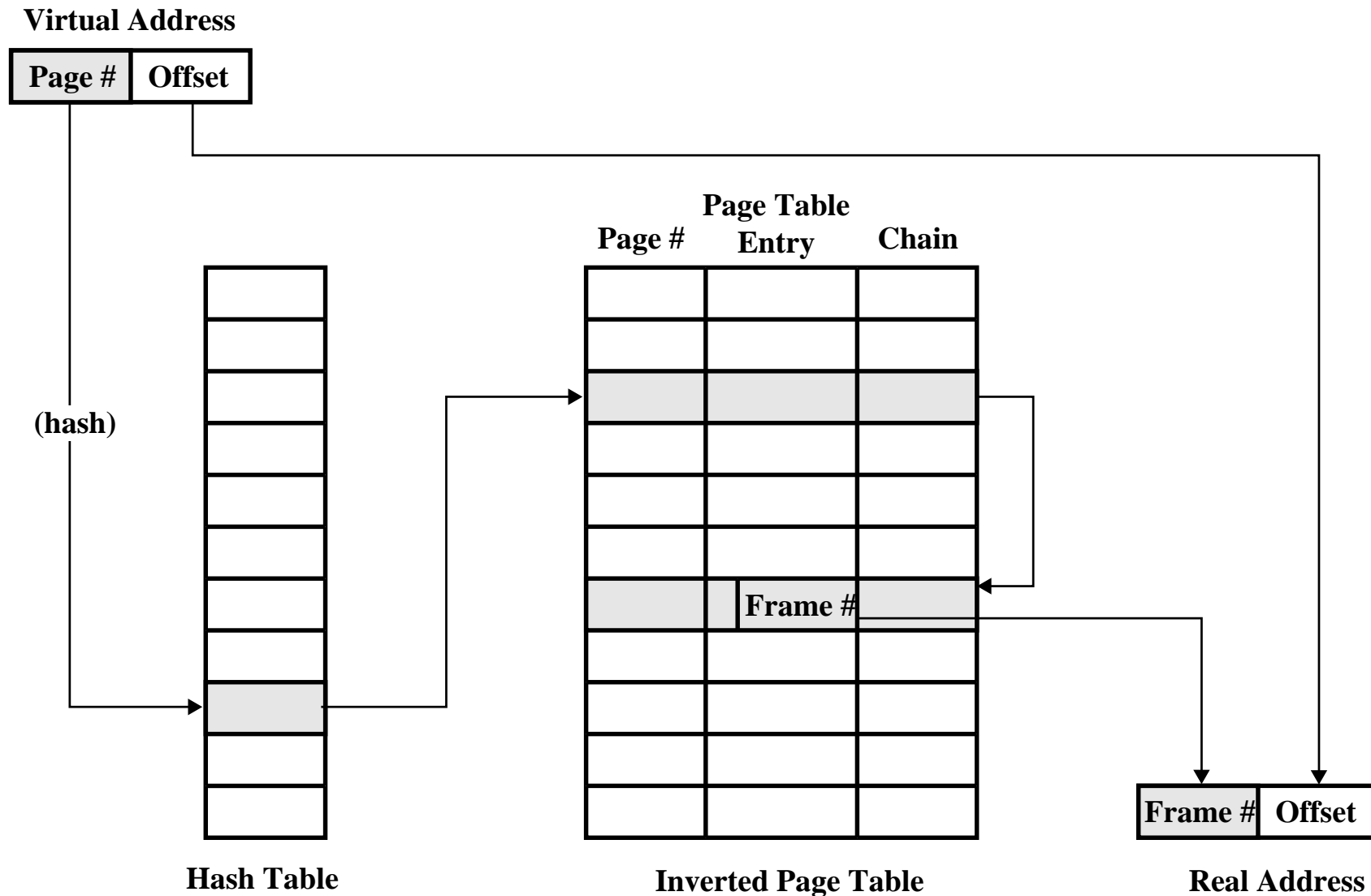


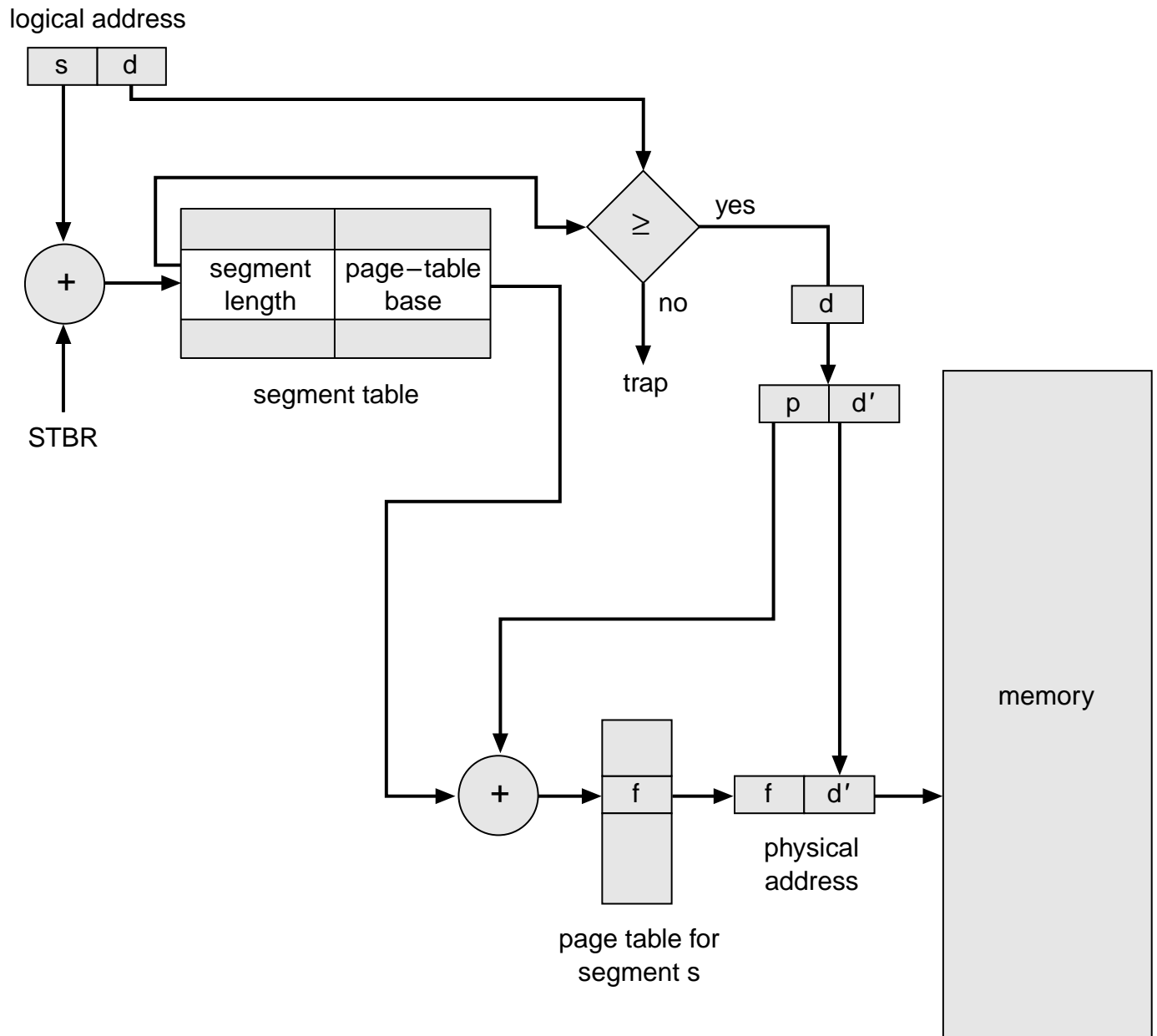
Tabella delle pagine invertita con hashing

Per ridurre i tempi di ricerca nella tabella invertita, si usa una funzione di hash (hash table) per limitare l'accesso a poche entry (1 o 2, solitamente).

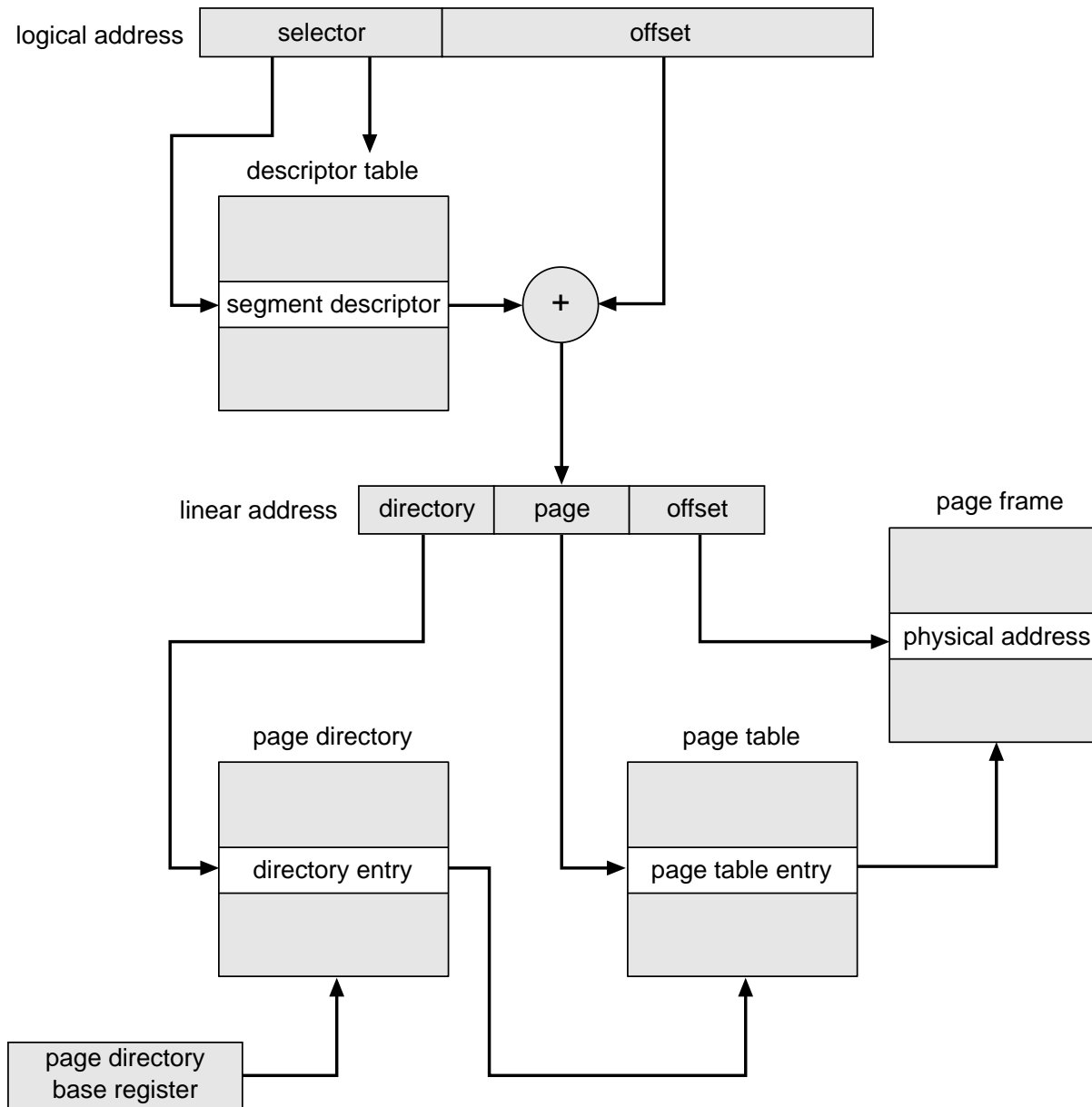


Segmentazione con paginazione: MULTICS

- Il MULTICS ha risolto il problema della frammentazione esterna paginando i segmenti
- Permette di combinare i vantaggi di entrambi gli approcci
- A differenza della pura segmentazione, nella segment table ci sono gli indirizzi base delle page table dei segmenti



Segmentazione con paginazione a 2 livelli: la IA32



Sommario sulle strategie della Gestione della Memoria

- Supporto Hardware: da registri per base-limite a tabelle di mappatura per segmentazione e paginazione
- Performance: maggiore compessità del sistema, maggiore tempo di traduzione. Un TLB può ridurre sensibilmente l'overhead.
- Frammentazione: la multiprogrammazione aumenta l'efficienza temporale. Massimizzare il num. di processi in memoria richiede ridurre spreco di memoria non allocabile. Due tipi di frammentazione.
- Rilocazione: la compattazione è impossibile con binding statico/al load time; serve la rilocazione dinamica.

Sommario sulle strategie della Gestione della Memoria (Cont)

- Swapping: applicabile a qualsiasi algoritmo. Legato alla politica di scheduling a medio termine della CPU.
- Condivisione: permette di ridurre lo spreco di memoria e quindi aumentare la multiprogrammazione. Generalmente, richiede paginazione e/o segmentazione. Altamente efficiente, ma complesso da gestire (dipendenze sulle versioni).
- Protezione: modalità di accesso associate a singole sezioni dello spazio del processo, in caso di segmentazione/paginazione. Permette la condivisione e l'identificazione di errori di programmazione.