SAPIENZA
UNIVERSITÀ DI ROMA

Corso di laurea in Ingegneria dell'Informazione
Indirizzo Informatica

Reti e sistemi operativi

Gestione della memoria

Memoria principale (1/2)

- Generalmente, i linguaggi assembly permettono di accedere direttamente solo ai **registri delle CPU** e alla **memoria principale** (o memoria centrale, solitamente rappresentata dalla RAM, Random Access Memory)
- Il ciclo fetch, decode & execute lavora su indirizzi di memoria principale → i programmi per essere eseguiti devono essere prelevati dalla memoria secondaria (es. HD) e portati in memoria principale.

Memoria principale (2/2)

- (Questione 1) **Velocità**:
  - Lettura/scrittura dei registri: ~1 ciclo di clock
  - Lettura/scrittura RAM: molti cicli di clock

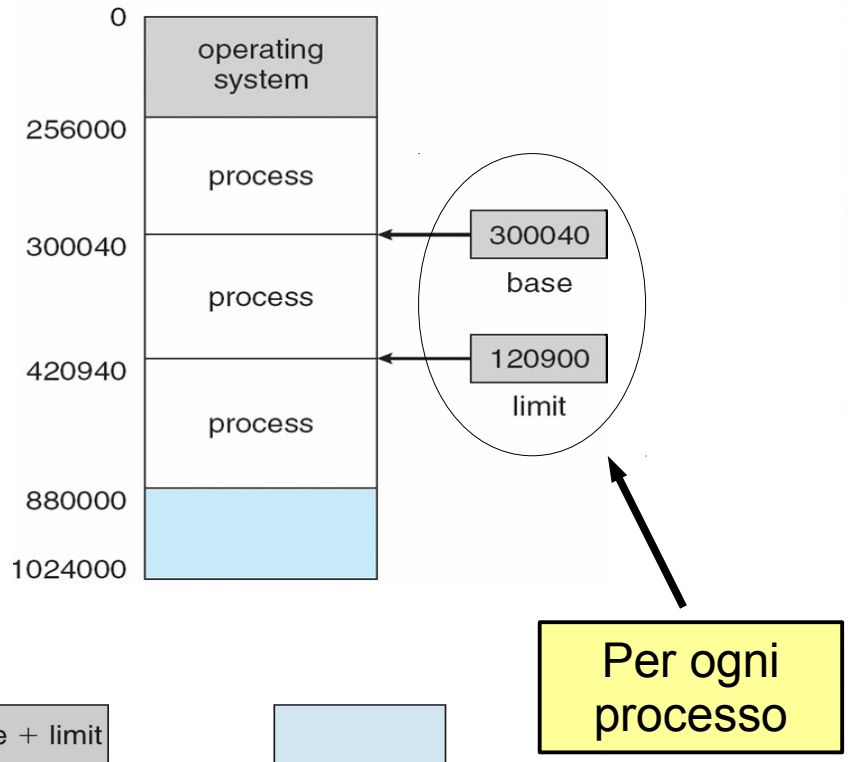
  → utilizzo di buffer temporanei (**memoria cache**), non visibili al programmatore (gestita dall'hardware ed in parte dal sistema operativo), di minor capienza e maggiore velocità, posti tra CPU e memoria principale.
- (Questione 2) **Protezione della memoria** → ad ogni processo è associato uno spazio di indirizzi (ovvero, un'area di memoria) non accessibile da altri processi in user mode (solo il S.O., in kernel mode, può accedervi). Di solito, protezione implementata in hardware con il contributo del S.O..

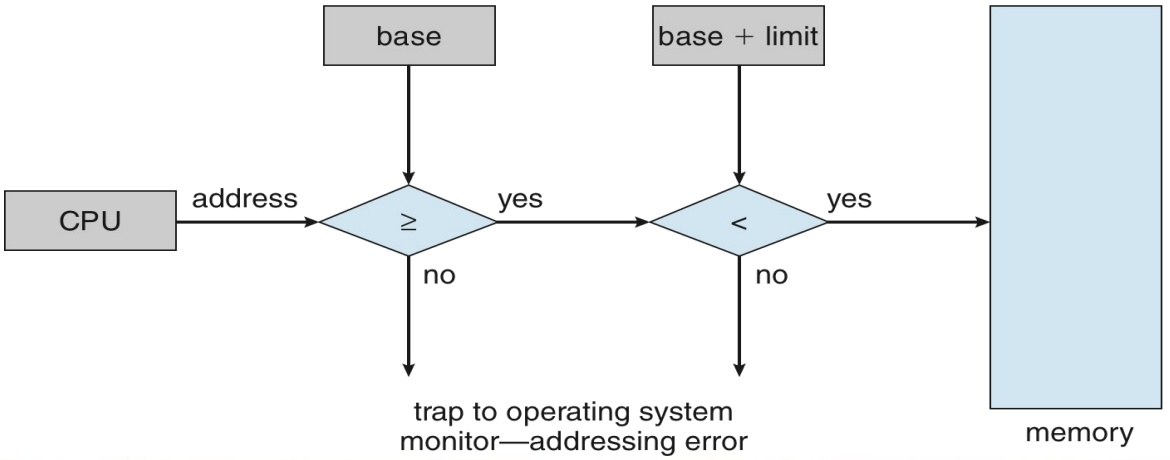
Protezione della memoria

- **Idea di base**
  - Il sistema operativo notifica all'hardware lo spazio di indirizzi associato ad un processo.
  - In user mode, ogni indirizzo di memoria viene controllato dall'HW prima di permettere l'accesso vero e proprio.
  - Se l'indirizzo non appartiene allo spazio di indirizzi associato al processo che ne ha fatto richiesta, l'accesso viene interdetto e viene sollevata un'eccezione (es. TRAP) notificata al S.O..

Registri Base e Limite

- Meccanismo di protezione molto semplice: Ad ogni processo sono associati 2 registri (indirizzo **base** e indirizzo **limite**, caricati dal S.O.), che delimitano lo spazio di memoria associato allo stesso. Controllo di accesso effettuato in HW.





Binding degli indirizzi (1/2)

- **Problema**: Nella soluzione precedente, ai processi possono essere associati spazi di memoria sempre diversi, ovvero per uno stesso programma ad ogni esecuzione quasi mai vengono assegnati gli stessi indirizzi base e limite (al limite per uno stesso programma, rimane fissa la dimensione, es. dimensione = limite – base).
- **Soluzione**: **binding degli indirizzi** (~associazione degli indirizzi). Anziché indirizzi fissi, utilizzare simboli o indirizzi riallocabili. Ad esempio un riferimento a memoria potrebbe essere implementato come un vincolo del tipo “160 byte dall'inizio del programma”. L'indirizzo effettivo verrà associato in seguito. Il binding può essere effettuato:
  - Durante la **compilazione** del programma
  - Durante il **caricamento** del programma
  - Durante l'**esecuzione** del programma.

Binding degli indirizzi (2/2)

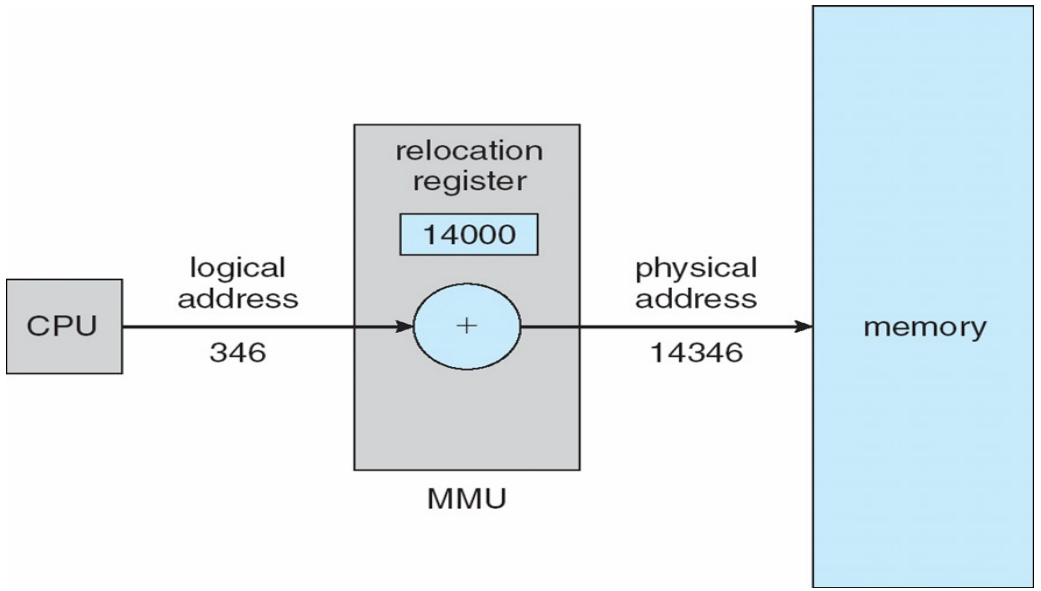
- Binding a tempo di compilazione (**absolute code**)
  - La posizione in memoria del processo è fissa e deve essere nota al tempo della compilazione
  - Se la locazione in cui il processo è caricato in memoria è modificata, è necessaria la ricompilazione del programma.
- Binding a tempo di caricamento (**relocatable code**)
  - La posizione in memoria del processo (indirizzo base del processo) è nota solo nel momento in cui viene eseguito, **dopodiché rimarrà fissa** durante tutta l'esecuzione.
  - Ogni riferimento ad un indirizzo è risolto all'esecuzione tramite un offset sommato all'indirizzo base del processo.
- Binding a tempo di esecuzione (**run-time mapping**)
  - La posizione in memoria può variare durante l'esecuzione
  - Ogni riferimento viene risolto in un indirizzo in **run-time**.

Indirizzi logici e fisici

- Indirizzo logico (o *virtuale*)
  - Generato dalla CPU durante l'esecuzione del programma.
  - Spesso zero-based per ogni processo.
- Indirizzo fisico
  - Posizione reale in memoria
  - Sconosciuto ai programmi in esecuzione: essi generano solo indirizzi logici.
- Indirizzi logici e fisici coincidono per il binding a tempo di compilazione e caricamento, possono invece differire per il binding a tempo di esecuzione → in questo secondo caso, il mapping tra indirizzi logici e indirizzi fisici viene effettuato da un dispositivo hardware dedicato chiamato **Memory-Management Unit** (**MMU**).

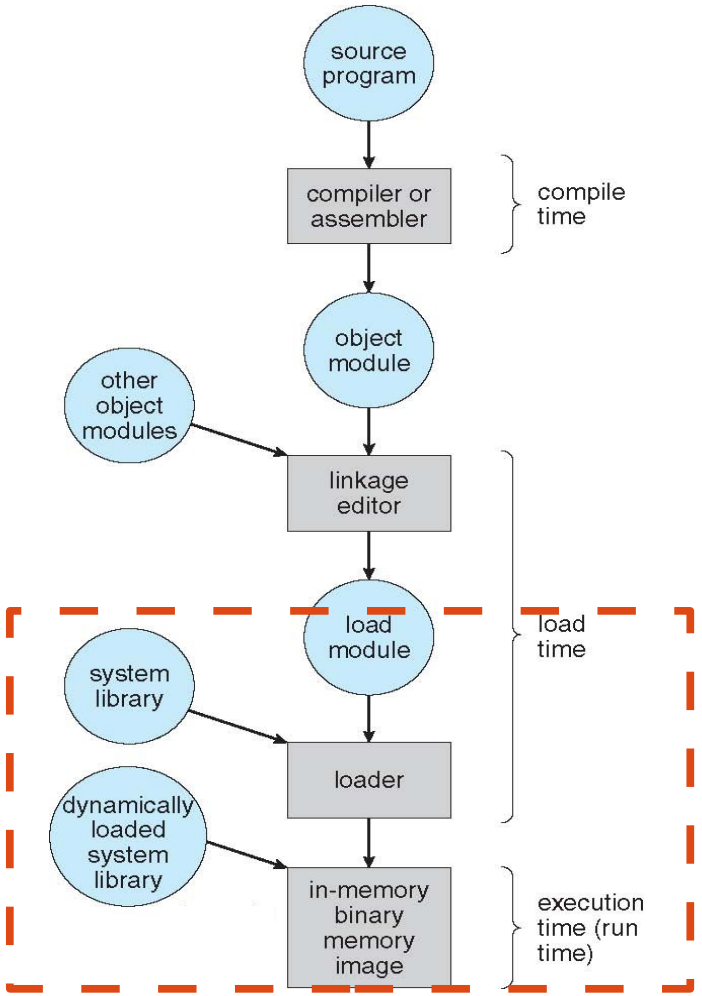
Una semplice MMU

- Utilizza solo l'indirizzo di base (**indirizzo di riallocazione**, salvato in un apposito registro) dello spazio di memoria associato ad un processo, caricato dal sistema operativo ad ogni spostamento in memoria dello stesso.
- Ogni processo utente *i* genera indirizzi logici che vanno da 0 a *max_i*, ove *max_i* è la quantità di memoria richiesta dal processo stesso per poter essere eseguito.



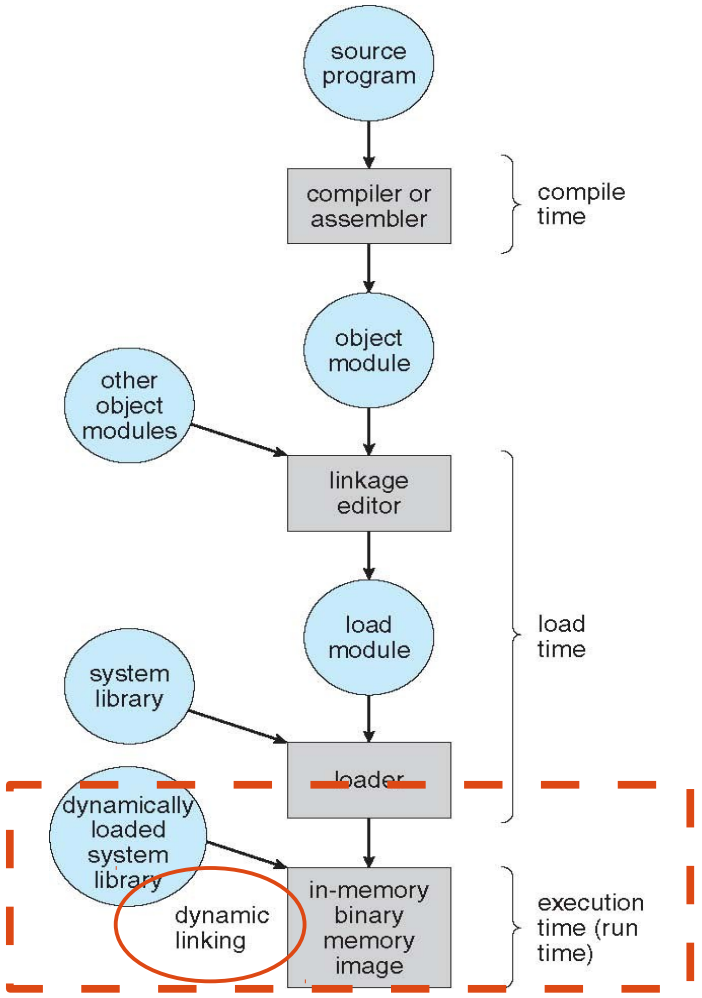
Caricamento dinamico

- Non sempre è necessario caricare tutto il programma in memoria.
- **Dynamic Loading**: ogni routine è **caricata** (**disco → memoria**) solo quando viene chiamata, assieme a tutte le altre routine da lei chiamate.
- Le routine utilizzano ovviamente **indirizzi riallocabili** (base + offset).
- E' una funzionalità richiesta **esplicitamente** dal programmatore, attraverso funzioni messe a disposizione dal sistema operativo.
- Spesso usato nei **software plug-in** dei programmi



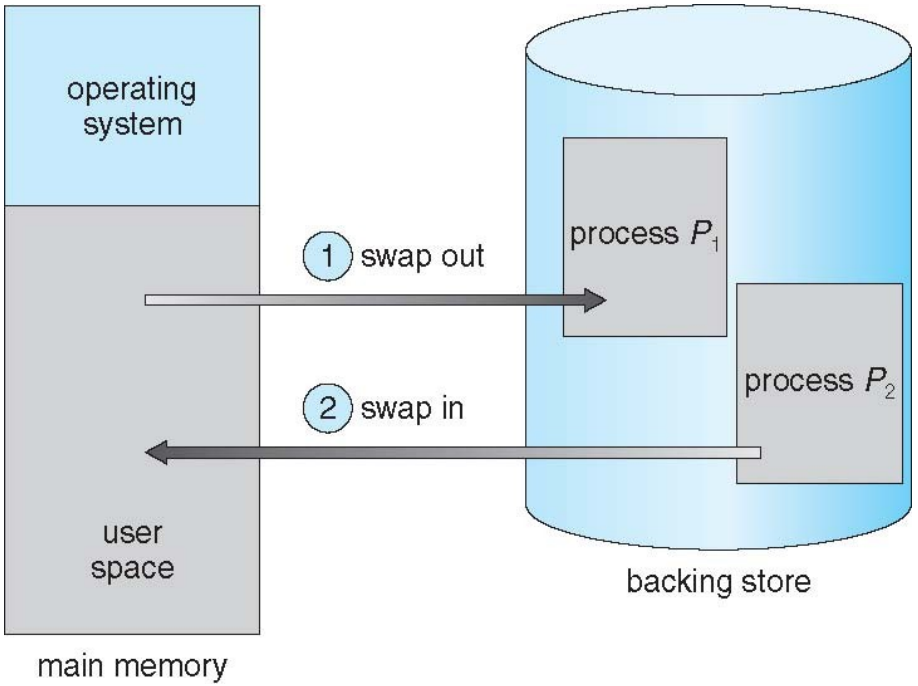
Linking dinamico

- **Static linking** → ogni funzione (routine) di libreria esterna viene inglobata (copiata) nel codice del programma.
- **Dynamic linking** → simile a dynamic loading, ma trasparente al programmatore. Il codice rimane lo stesso, è di solito **un'opzione di linking,** ovvero: si linkano **librerie dinamiche** anziché statiche.
- Al posto delle funzioni è inserita una porzione di codice (**stub**) in grado di:
  (a) utilizzare subito la routine se già in memoria (as. usata da altri programmi)
  (b) caricare la routine se non presente in memoria.
- **E' possibile sostituire le librerie dinamiche (es. con nuove versioni più performanti) lasciando inalterati i programmi che ne fanno uso.**



Swapping (1/3)

- **Problema**: La memoria richiesta dalla totalità dei processi in esecuzione potrebbe eccedere il totale di memoria principale presente nel sistema.
- Soluzione: **swapping** → lo spazio di memoria associato a qualche processo viene temporaneamente salvato a disco (**swap out**) liberando spazio per la memoria associata altri processi, eventualmente recuperata da disco (**swap in**) a seguito di un precedente swap-out.



•Ricorda: **Spazio di memoria utilizzato da un processo in esecuzione ("spazio di indirizzamento")**: programma (sezione text) + dati globali (sezione data) + stack + heap + ecc...

Swapping (2/3)

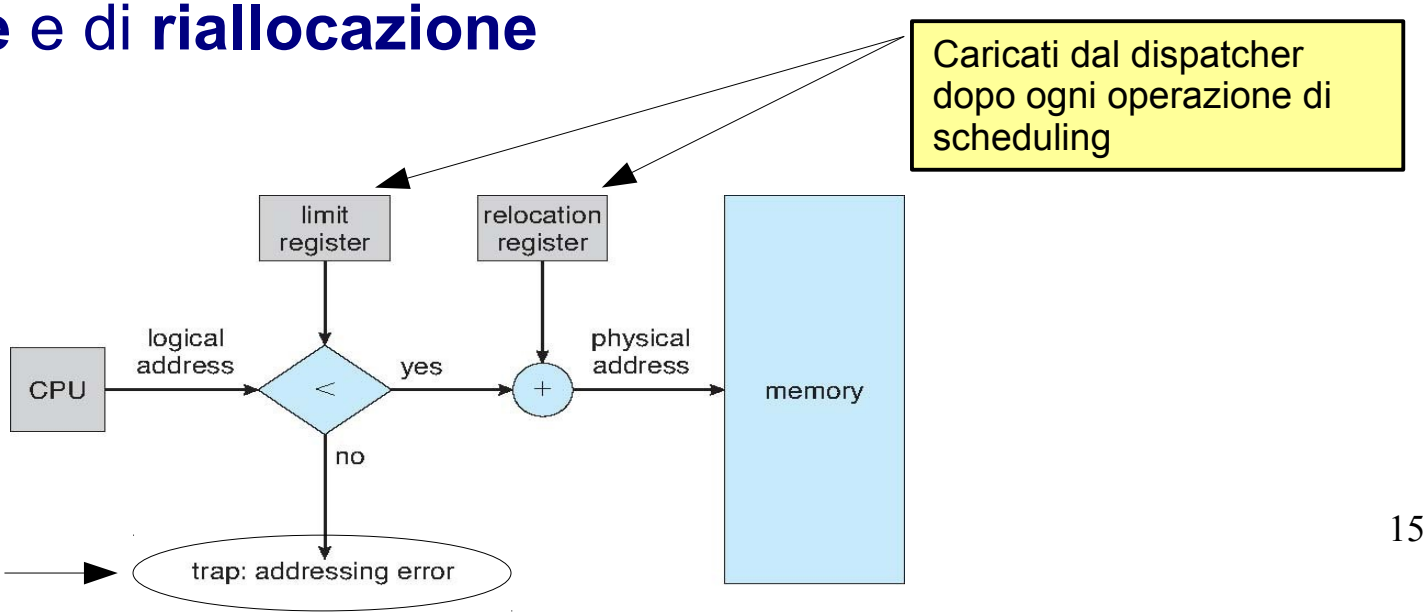
- **Idea di base**: se non c'è sufficiente spazio in memoria, swap out del processo appena schedulato, nel mentre swap in dei prossimi processi da schedulare
- **Roll out, roll in**: variante dello swapping basato sulla priorità dei processi in esecuzione: swap out di processi a bassa priorità così i processi ad alta priorità possono essere eseguiti.
- Problema in caso di swapping: **altissimo overhead** sui context switch dovuto ai tempi di trasferimento da e per il disco. Esempio per un context switch con swapping:
  - Dimensione tipica spazio di memoria occupata da processi “pesanti” (es. browser): ~50-100 MB
  - Velocità tipica HD (non cached read): ~50-100 MB/sec
  - Swap-out: 0.5-2 sec. + Swap-in: 0.5-2 sec. = 1-4 sec.
  - Normalmente vi sono centinaia di context switch al secondo: **il sistema perde totalmente di reattività**

Swapping (3/3)

- Altri problemi dello swapping:
  - In caso di swap in su una posizione di memoria diversa da quella occupata prima dello swap out, la mappa tra indirizzi logici e fisici va aggiornata → ovvero deve essere supportato il **binding a tempo di esecuzione.**
  - Durante le operazioni di I/O la memoria usata per attendere dati in input potrebbe appartenere ad un nuovo processo nel frattempo subentrato in memoria al posto del processo in attesa dei dati.
- Nei S.O. moderni, lo swap è attivato solo se la percentuale di memoria principale occupata dai processi supera una certa soglia (es. in Linux di default circa ~60 %).

Allocazione contigua (1/2)

- Semplice organizzazione della memoria in cui lo spazio di memoria di ogni processo è rappresentato da un'**unica area contigua**.
- Il S.O. possiede anch'esso uno spazio di memoria, di solito allocato a partire dalle prime locazioni di memoria (*low memory*), di seguito gli altri processi (*high memory*).
- Mapping tra indirizzi logici e fisici basato sull'utilizzo dei registri **limite** e di **riallocazione**

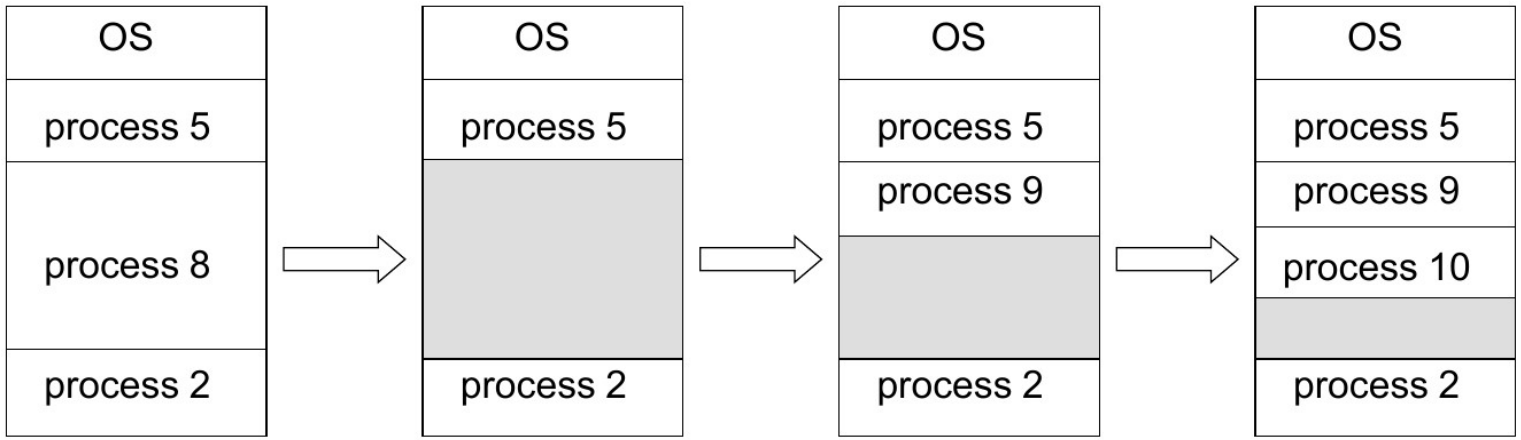


Allocazione contigua (2/2)

- Come viene suddivisa la memoria tra i vari processi? In caso di allocazione contigua un'idea di base è di adottare un **partizionamento statico**:
  - Memoria suddivisa in un **numero fisso di partizioni**, di taglia uguale o diversa.
  - Ogni partizione è dedicata ad ospitare un singolo processo
  - **Problema**: grado di multiprogrammazione (multitasking) limitato dal numero di partizioni
- **Partizionamento dinamico (variabile)**: Partizioni di numero e dimensione variabile. Il sistema operativo tiene traccia delle partizioni allocate (occupate da qualche processo) e delle partizioni libere (buchi, *hole*) sparse in memoria.

Partizionamento dinamico (1/2)

- Quando un processo entra in memoria, è allocato in un hole abbastanza capiente per contenerlo
- Quando termina, viene creato un hole, eventualmente unendo hole contigui
- Un esempio:



Partizionamento dinamico (2/2)

Quale area libera tra le disponibili viene allocata?

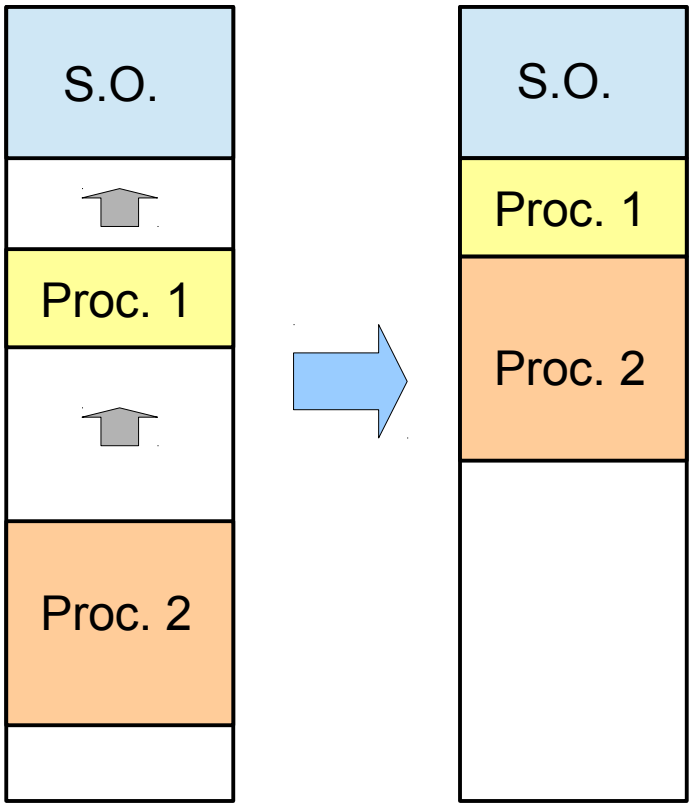
- **First fit**: Il processo viene allocato nel primo hole disponibile sufficientemente grande (partendo dall'inizio o dal punto in cui era terminata la ricerca precedente)
- **Best fit**: Il processo viene allocato nell'hole più piccolo che può accoglierlo
  - Questa strategia tende a produrre frammenti di memoria di dimensioni minori
- **Worst fit**: Il processo viene allocato nell'hole più grande in grado di accoglierlo
  - Questa strategia tende a produrre frammenti relativamente grandi, utili quindi ad accogliere una quantità di nuovi processi di taglia ragionevole
- **First fit** e **best fit** sperimentalmente superiori a **worst** fit per quel che riguarda l'utilizzazione delle risorse

Frammentazione (1/2)

- **Problema** partizionamento dinamico → frammentazione esterna, ovvero **la memoria esterna alle partizioni diviene sempre più frammentata**, e quindi di più difficile utilizzo.
- **Frammentazione esterna**: esiste abbastanza memoria per contenere un nuovo processo, ma tale memoria NON è contigua.
- **Frammentazione interna**: la memoria è allocata a blocchi di dimensione fissa (block_size), per un processo che richiede una quantità di memoria pari proc_size si alloca un numero N di blocchi sufficiente a contenerlo → facilmente N*block_size > proc_size

Frammentazione (2/2)

- **Soluzione di base**: ricompattazione
- Necessario il supporto per il binding a tempo di esecuzione
- Costo elevato → overhead non trascurabile
- Soluzione più efficiente: **non contiguità**



Paginazione (1/3)

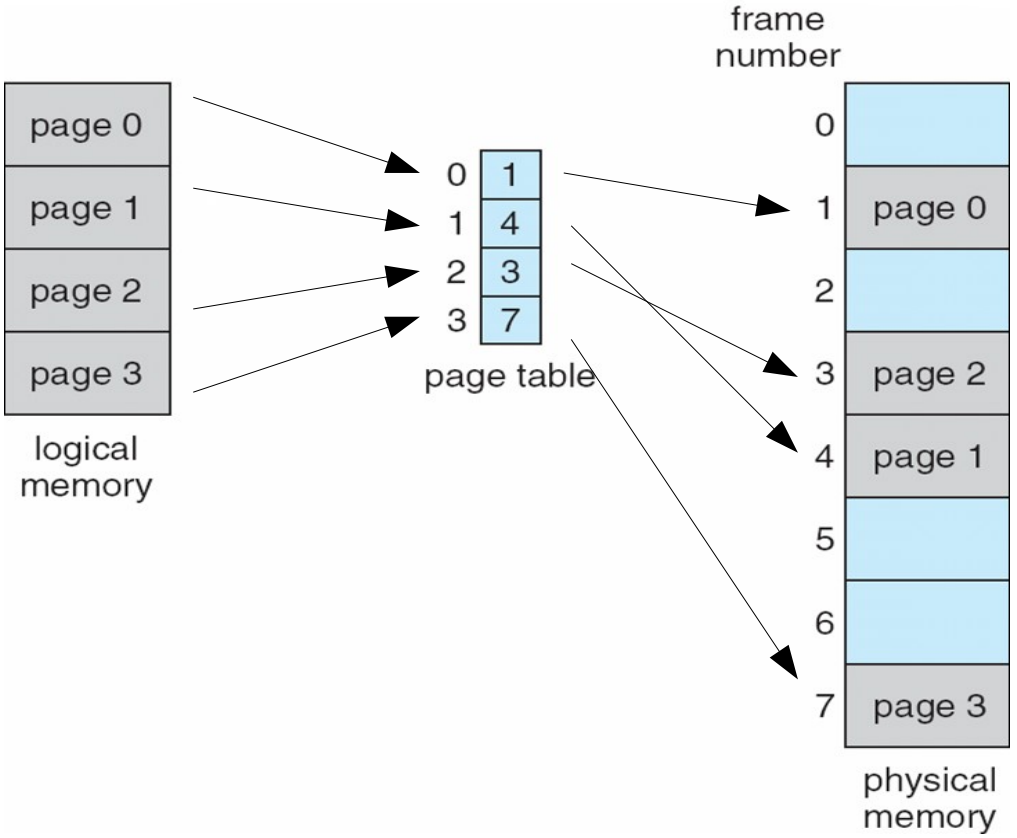
- Soluzione di allocazione non contigua della memoria, evita la frammentazione esterna e la necessità di ricompattare la memoria.
- Lo spazio di indirizzamento (indirizzi logici) di un processo viene diviso in un insieme di blocchi di taglia fissa di X bytes, detti **pagine** (X potenza di due, es. 4096 byte)
- Equivalentemente, la memoria principale (indirizzi fisici) viene divisa in un insieme di blocchi **non contigui** detti **frames**, di taglia identica alle pagine (X bytes),
- **Anche la memoria secondaria (es. l'HD) è suddivisa in frames**.
- Ogni pagina viene effettivamente caricata in uno specifico frame di memoria.

Paginazione (2/3)

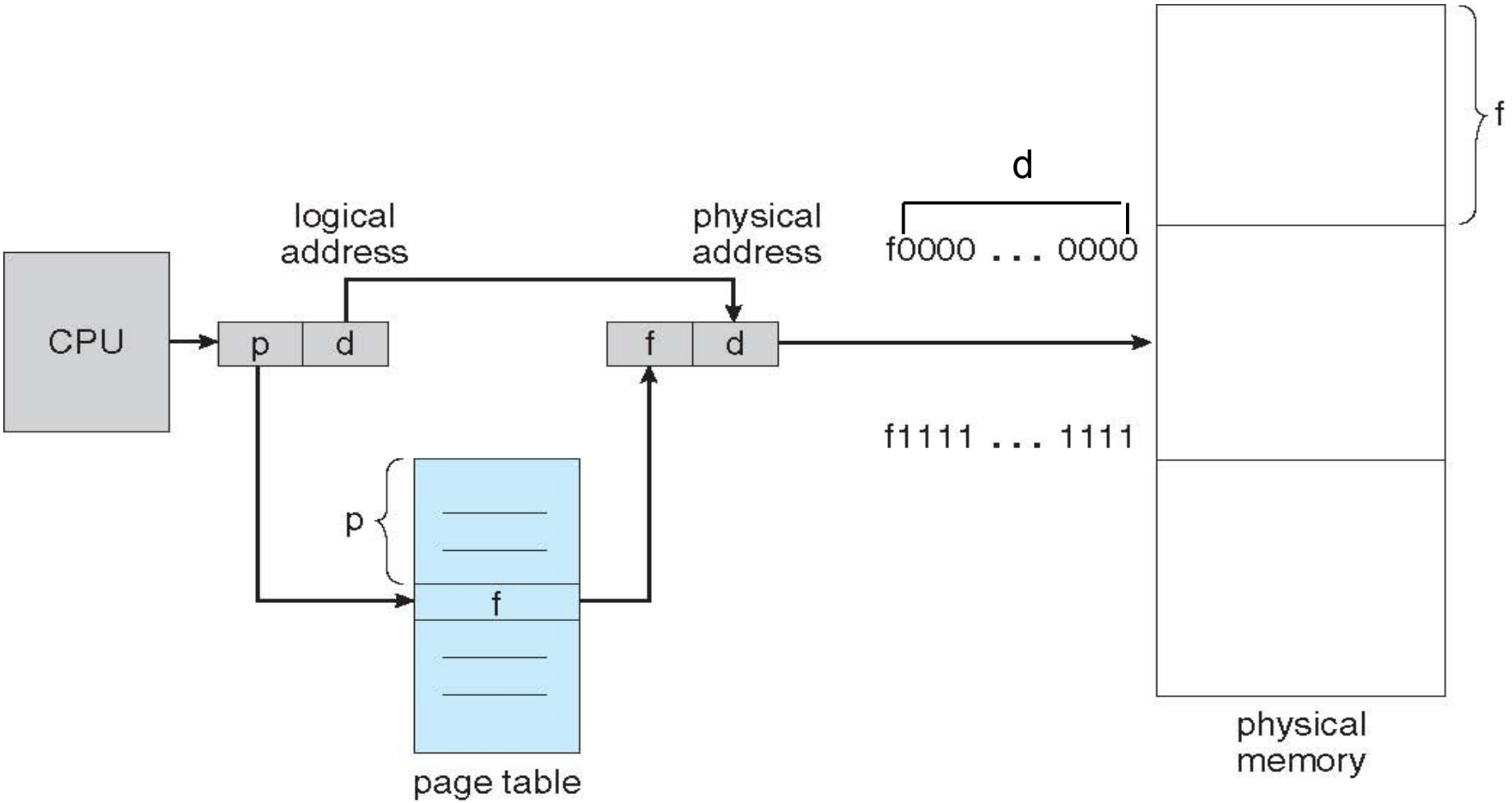
- Per caricare in memoria un programma che richiede memoria pari a N pagine, basta trovare in memoria principale N frames liberi sparsi per la memoria
- E' necessaria una mappa (tabella della pagine o **page table**) per convertire indirizzi logici in indirizzi fisici.
- **Usata in praticamente tutti i sistemi attuali.**

Paginazione (3/3)

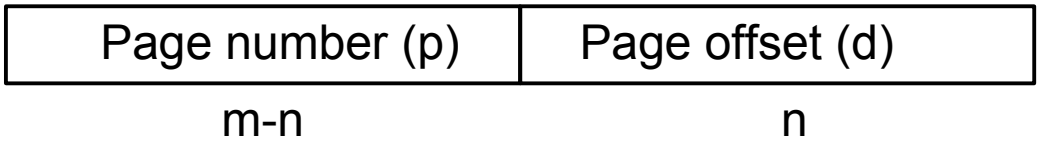
- Esempio:



Supporto hardware per la paginazione



Paginazione: indirizzamento



- Indirizzi logici a due livelli:
  - **Page number (p)**: rappresenta l'indice dell'elemento della page table che contiene l'indirizzo di base del frame associato in memoria centrale.
  - **Page offset (d)**: posizione all'interno della pagina che corrisponde alla posiziona all'interno del frame
- Per indirizzi logici di dimensione m:
  - 2^m possibili indirizzi logici
  - $2^{(m-n)}$ pagine al più
  - 2^n dimensione pagine

Frammentazione interna

- **Problema** paginazione: frammentazione interna. Ad esempio:
  - Dimensione pagine = 2048 bytes
  - Dimensione processo = 72766 bytes
  - 35 pages + 1086 bytes
  - **Frammentazione interna di 2048 - 1086 = 962 bytes**
- Caso peggiore: dimensione pagine – 1 byte
- Caso medio: dimensione pagine / 2
- Pagine di dimensione minore generano meno frammentazione interna ma richiedono più memoria per memorizzare page table di dimensioni maggiori
- Possibile soluzione: dimensione delle pagine variabile, modificata in run-time.

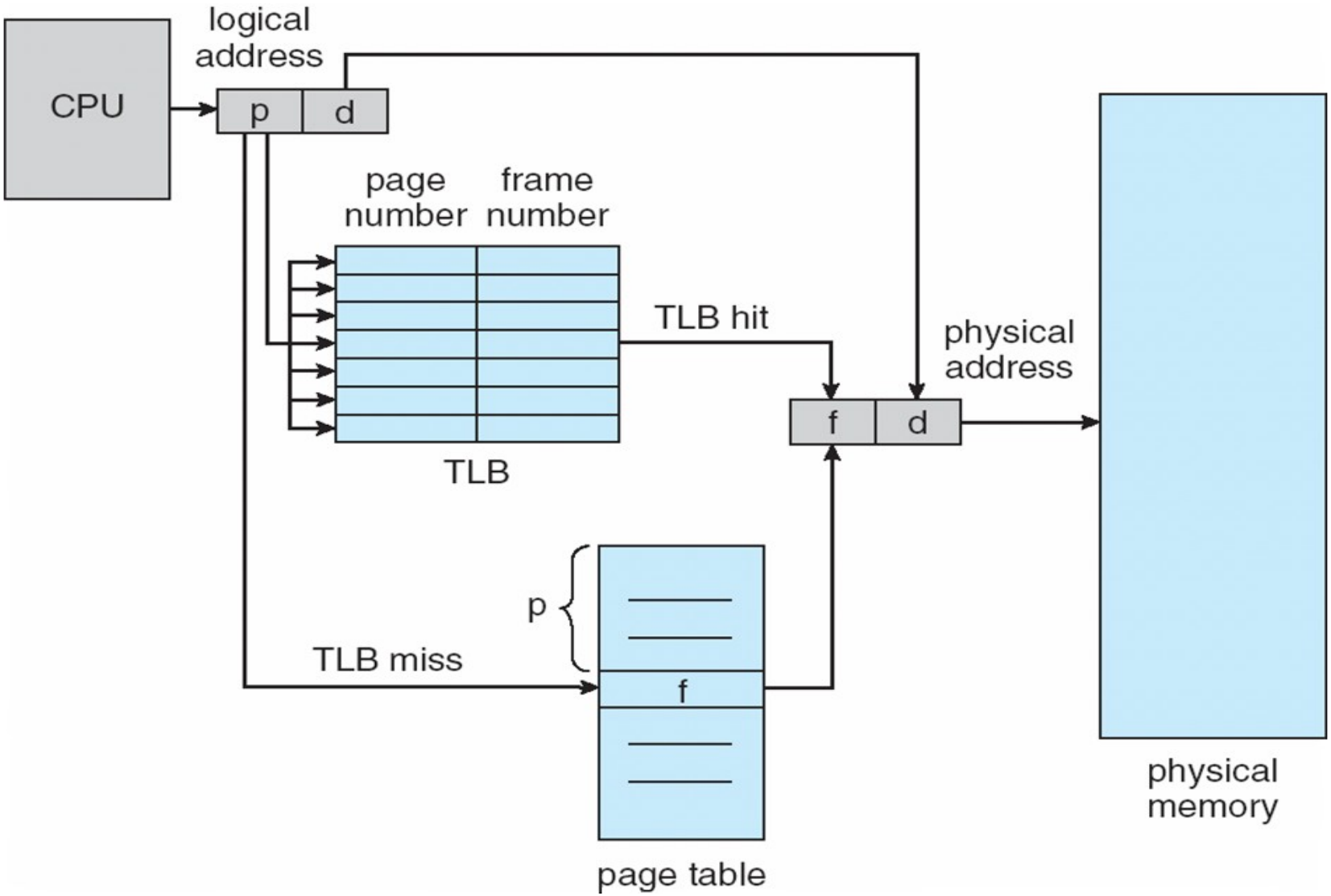
Page table: implementazione (1/2)

- La page table è mantenuta in memoria principale e gestita dal sistema operativo.
- L'hardware mette a disposizione 2 registri:
  - Page-table base register (**PTBR**): conterrà l'indirizzo ove si trova la page table
  - Page-table length register (**PTLR**): conterrà la dimensione della page table
- **Problema**: con questo schema per ogni dato letto o scritto in memoria sono necessari 2 accessi in memoria,
  - Il primo sulla page table, il secondo per il dato

Page table: implementazione (2/2)

- **Soluzione**: 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 (memoria associativa, che permette rapidissime ricerche in parallelo) che mantiene associazioni tra numero di pagina e frame per il processo correntemente in esecuzione.
- Dimensione TLB tipicamente ridotte (<=1024 bye)
  - Se non viene trovato l'entry cercata in TLB (TLB miss) il valore viene caricato accedendo direttamente alla page table completa in memoria centrale.

Supporto hardware per la paginazione con TLB



Tempo di accesso effettivo con TLB

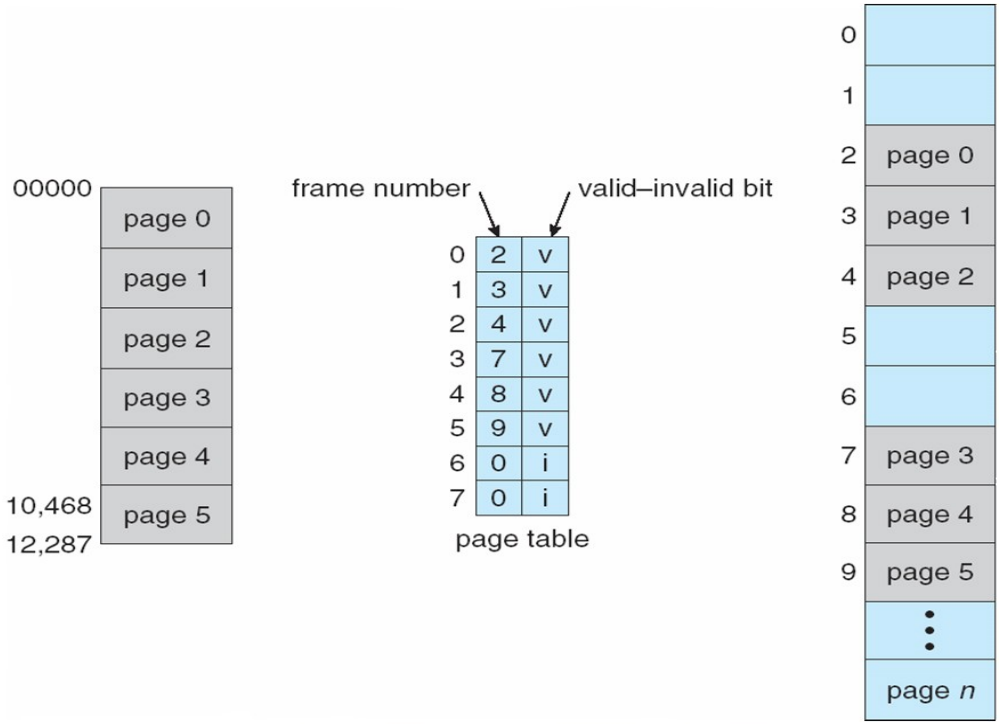
- Ricerca (***lookup***) in TLB di una pagina = t_{TLB} unità di tempo
  - Può essere meno del 10% del tempo t_{MEM} richiesto per effettuare la stessa operazione in memoria centrale
- **Hit ratio:** percentuale α delle volte che una pagina è trovata in TLB
- **Effective Access Time (EAT)**
  - $EAT = (t_{MEM} + t_{TLB})\alpha + (2*t_{MEM} + t_{TLB})(1 - \alpha)$
- Si consideri ad esempio α = 80%, t_{TLB} = 20ns e t_{MEM} = 100ns
  - EAT = 0.80 x 120 + 0.20 x 220 = 140ns
- Con α = 98%, t_{TLB} = 20ns e t_{MEM} = 140ns
  - EAT = 0.98 x 160 + 0.02 x 300 = 162.8ns

Protezione della memoria (1/2)

- Protezione per accesso di **processi non autorizzati** fatta attraverso (ASID) "**access-space identifier**"
- Ogni riga della TLB contiene un ASID che identifica univocamente il processo proprietario di una pagina.
- Solo il processo con il corretto ASID potrà accedere alle informazioni di quella riga della TLB.
- Se il sistema operativo non supporta l'ASID, per proteggere la memoria **ogni processo carica la propria page table quando entra in esecuzione e la cancella quando esce dall'esecuzione** → la page table contiene solo pagine associate al processo

Protezione della memoria (2/2)

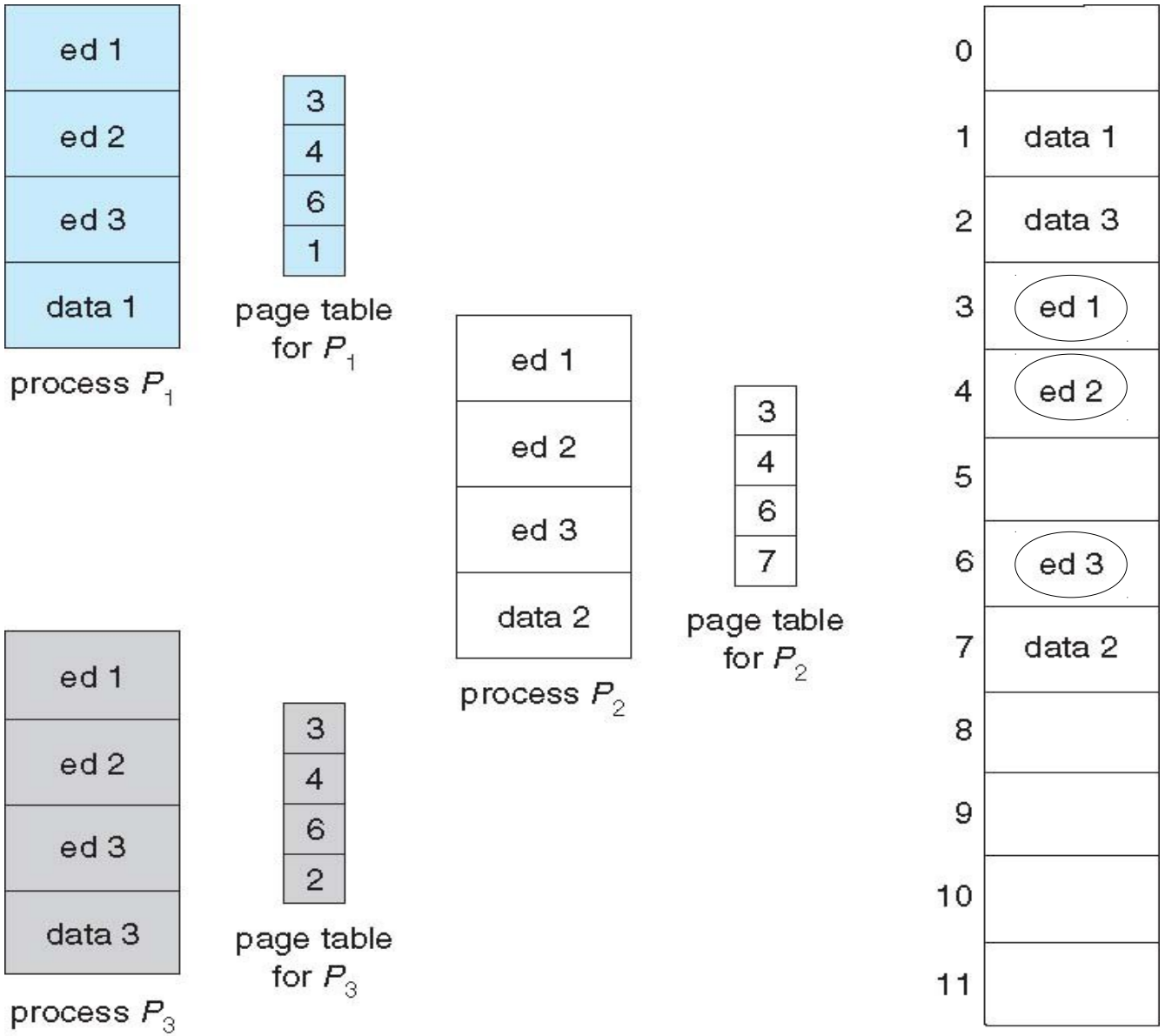
- Metodo alternativo: bit di protezione associati ad ogni entry della page table
  - Bit read-only
  - Bit di **validità**: se true tale pagina appartiene al processo corrente



Pagine condivise (1/2)

- E' spesso necessario condividere memoria tra vari processi, per risparmiare memoria e per permettere ai processi di comunicare tra di loro:
  - Librerie (dinamica) condivisa tra più programmi;
  - Programma utilizzato da più utenti dello stesso sistema;
  - Memoria condivisa per IPC
- **Idea**: frame di memoria fisica condivisa si troveranno in qualche entry di tutte le page table dei processi che condividono tale memoria.
- Il sistema operativo potrà utilizzare dei bit di protezione per regolare l'accesso a tali pagine

Pagine condivise (2/2)

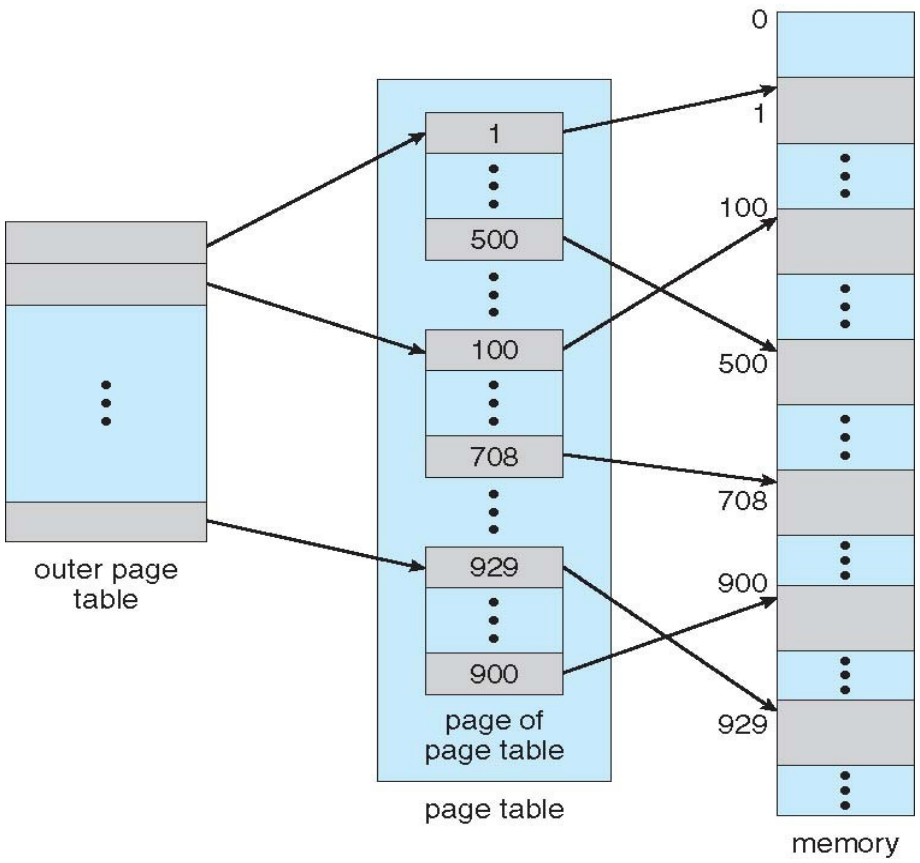


Dimensione della page table

- **Problema**: la paginazione può costare moltissimo sia in termini **quantità di memoria** sia in termini di **contiguità**, es.:
  - Architettura 32 bit: indirizzi logici (e fisici) di 32 bit
  - Dimensione pagine 4 KB (12 bit)
  - → La page table potrebbe avere **2^{20} entry contigue**
- E' necessario estendere l'architettura base della paginazione

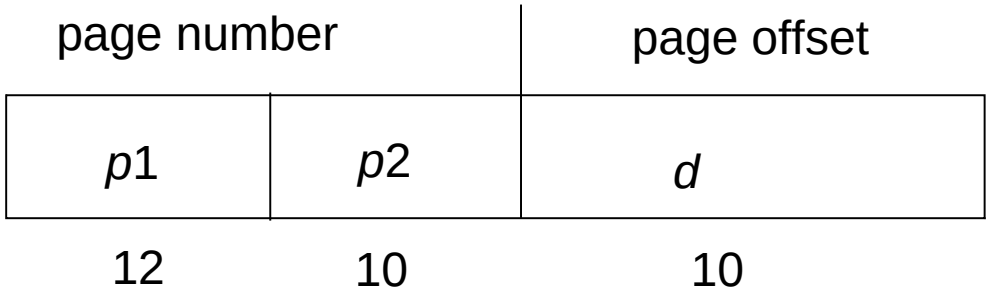
Page table gerarchiche (1/3)

- Idea di base: **“paginare” la page table**
- L'indirizzo logico è diviso in n porzioni, n-1 delle quali sono page number di n-1 distinte page table.
- Risolve il problema della contiguità della memoria

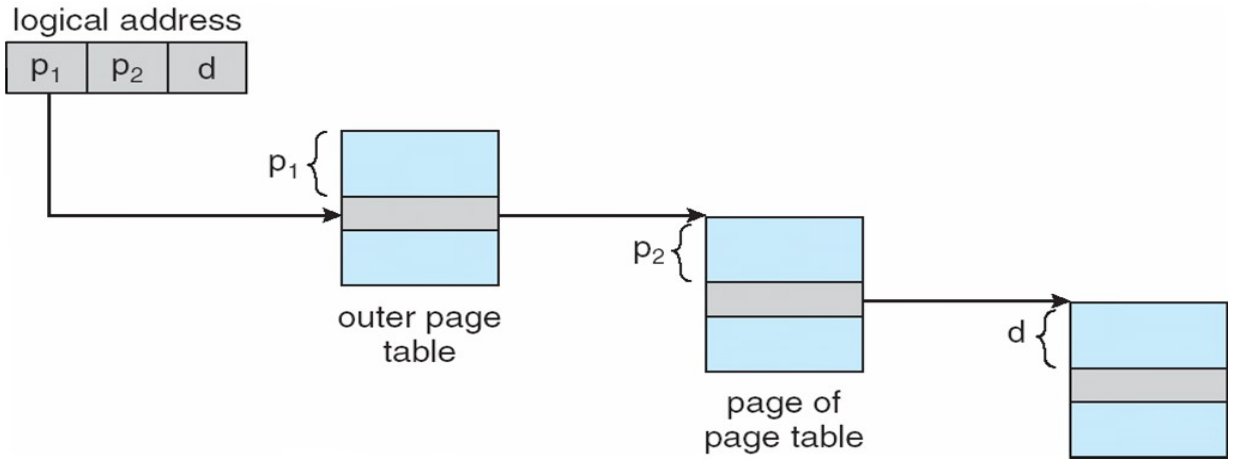


Page table gerarchiche (2/3)

- Esempio a due livelli:

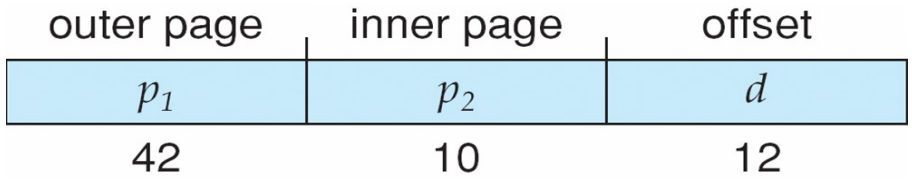


- p1 è un indice nella page table “esterna”, p2 è l'indice nella page table “interna”

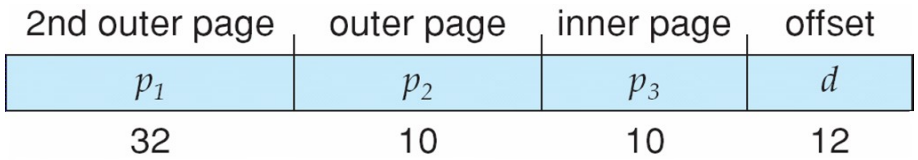


Page table gerarchiche (3/3)

- In caso di architettura 64 bit 2 livelli non bastano, es:



- Possibile soluzione → aggiungere uno o più livelli:

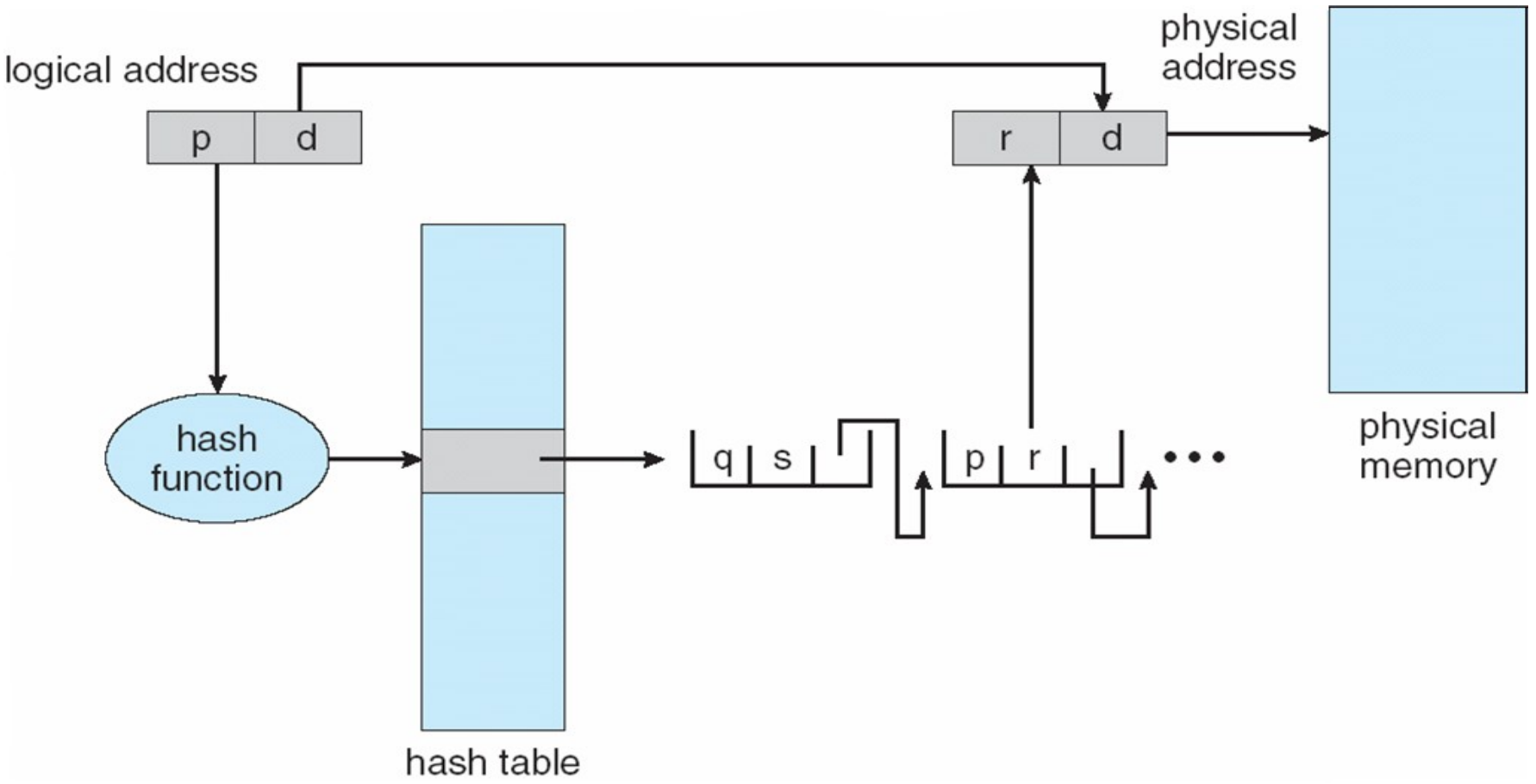


- **Problema**: elevato numero di accessi alla memoria per singole operazione → page table gerarchiche non adeguate per architetture 64 bit

Page table con hash table (1/2)

- Page table implementata come un hash table:
  - Il (virtual) page number viene trasformato nel corrispondente codice hash
  - Ogni entry dell'hash table contiene una **lista concatenata di triple**:
    - Page number
    - Page frame associato
    - Puntatore al prossimo elemento della lista
- Una volta selezionata la lista corrispondente, essa viene scandita alla ricerca del corrispondente page number.
  - Se presente, si calcola l'indirizzo fisico
  - Se non presente?

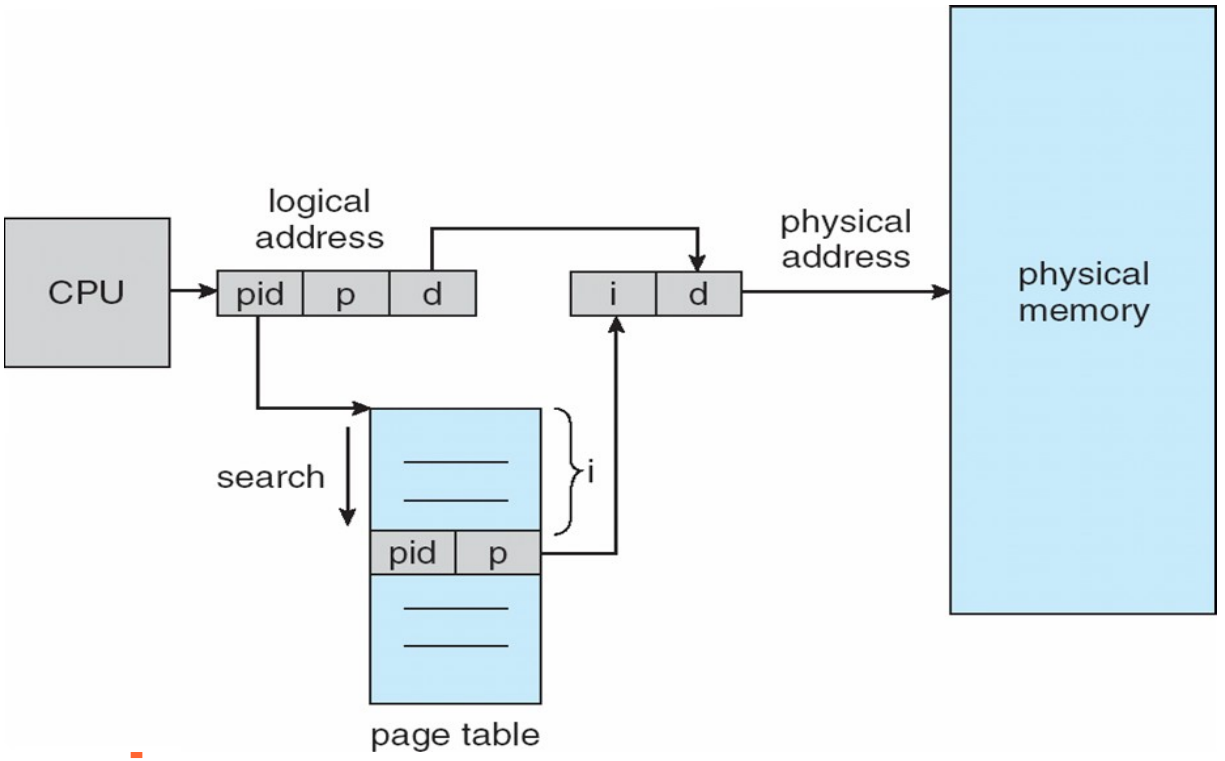
Page table con hash table (2/2)



Page table invertita (1/2)

- Page table → ogni processo tiene traccia di tutti i possibili **indirizzi logici**
  - Scope: **processo**
- Page table invertita → il sistema operativo tiene traccia di tutte i possibili **indirizzi fisici** (ovvero **frame**)
  - Scope: **sistema**
  - Un'entry per ogni frame, che include:
    - PID (process ID) → protezione della memoria
    - Page number → utilizzato per la ricerca
- Un indirizzo logico è formato da PID, page number e offset: la paga table invertita viene scandita sino a trovare PID e page number corrispondenti → **l'offset è il frame number**

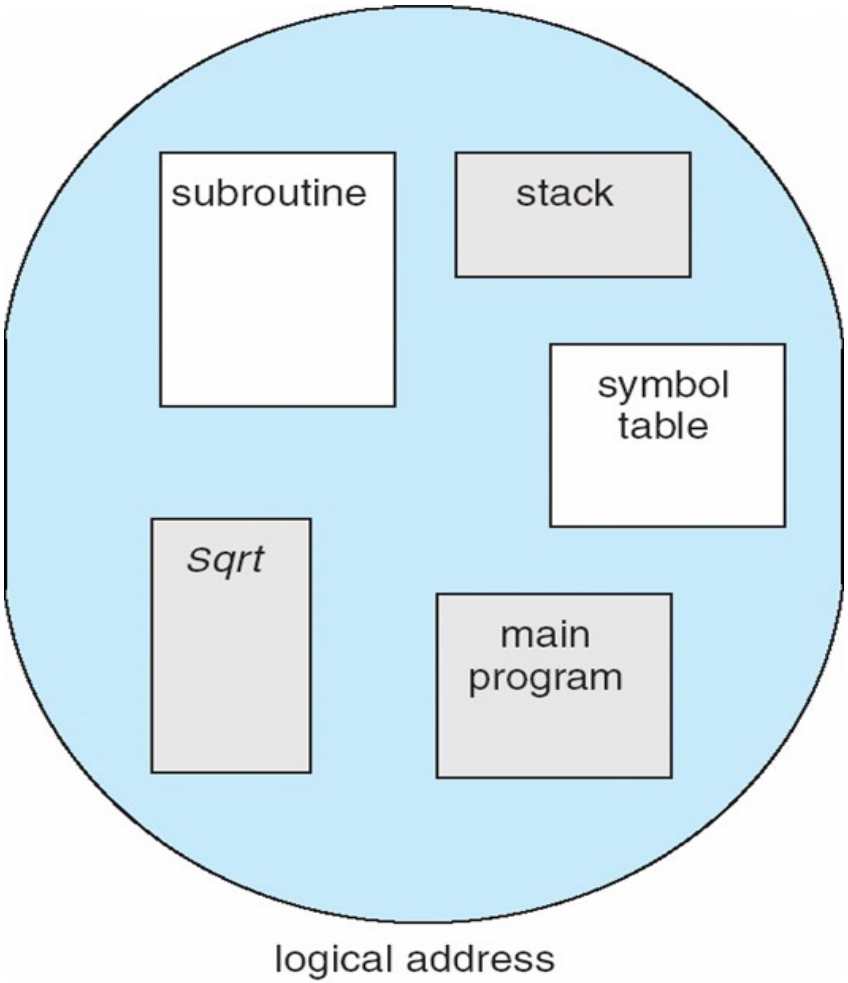
Page table invertita (2/2)



- **Problemi**:
  - la ricerca potrebbe coinvolgere l'intera p.t. invertita → **overhead**
  - Shared memory non implementabile

Segmentazione (1/4)

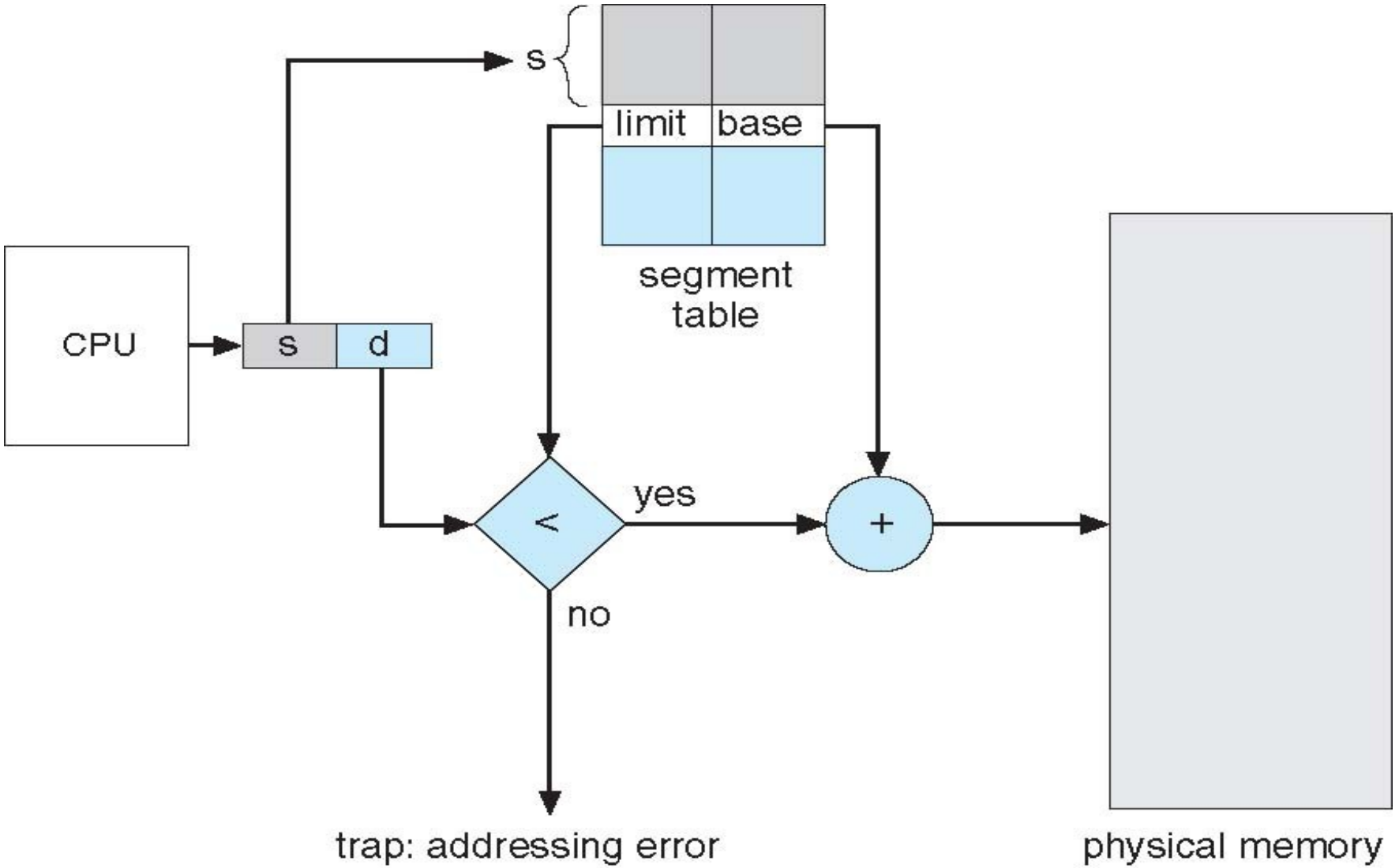
- Gestione della memoria che asseconda la visione dell'utente, il quale vede la memoria come un insieme di segmenti, ognuno con la propria tipologia di contenuto:
  - Programma principale
  - Stack
  - Hash
  - Librerie
  - ...



Segmentazione (2/4)

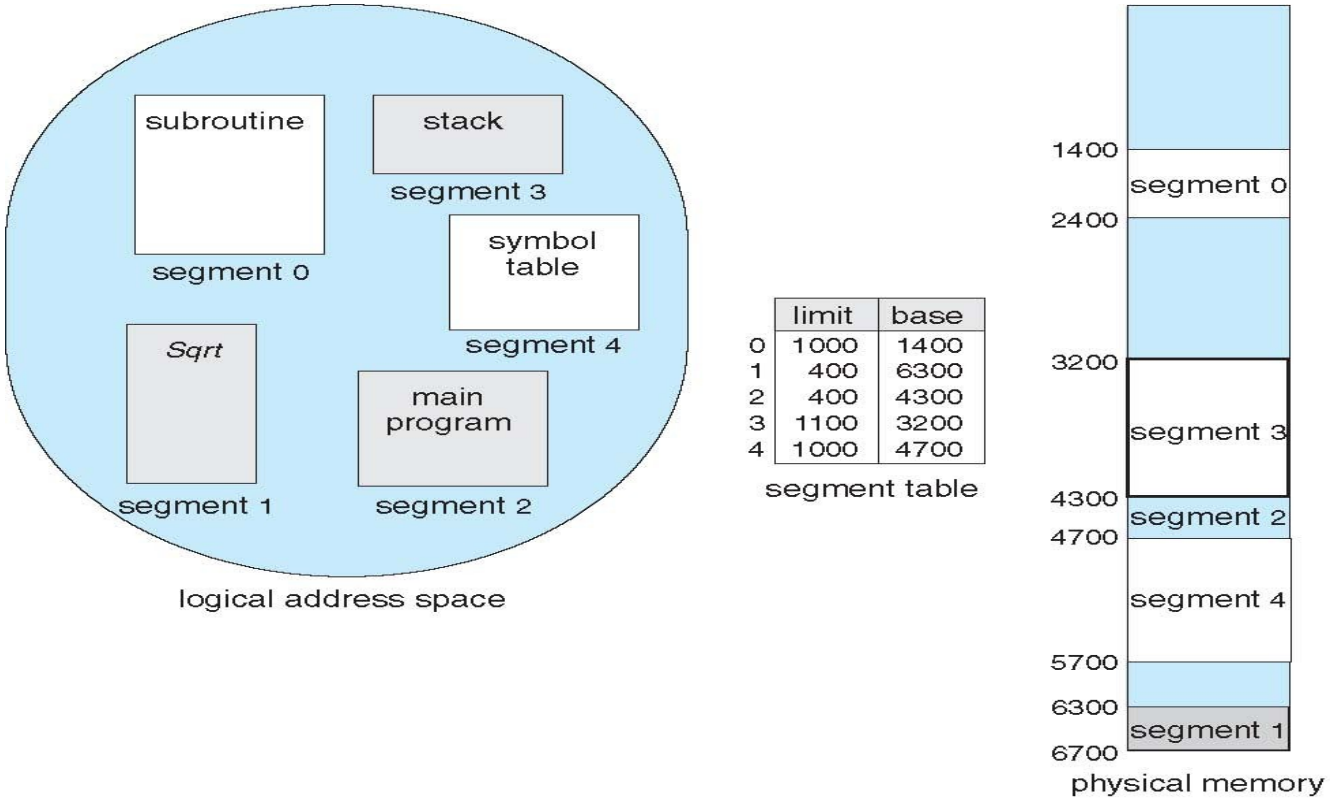
- Indirizzi logici:
  - Tuple <numero segmento, offset>
- **Segment table** → per ogni segmento:
  - **Base**: indirizzo **fisico** dove inizia il segmento
  - **Limite**: Lunghezza del segmento
- Segment-table base register (**STBR**): punta alla segment table in memoria
- Segment-table length register (**STLR**): lunghezza della segment table (l'indirizzo è valido solo se numero segmento < STLR
- Protezione, ogni entry delle segment table:
  - Validation bit, se = 0 segmento illegale
  - Read/write bit

Segmentazione (3/4)



Segmentazione (4/4)

- Un esempio:

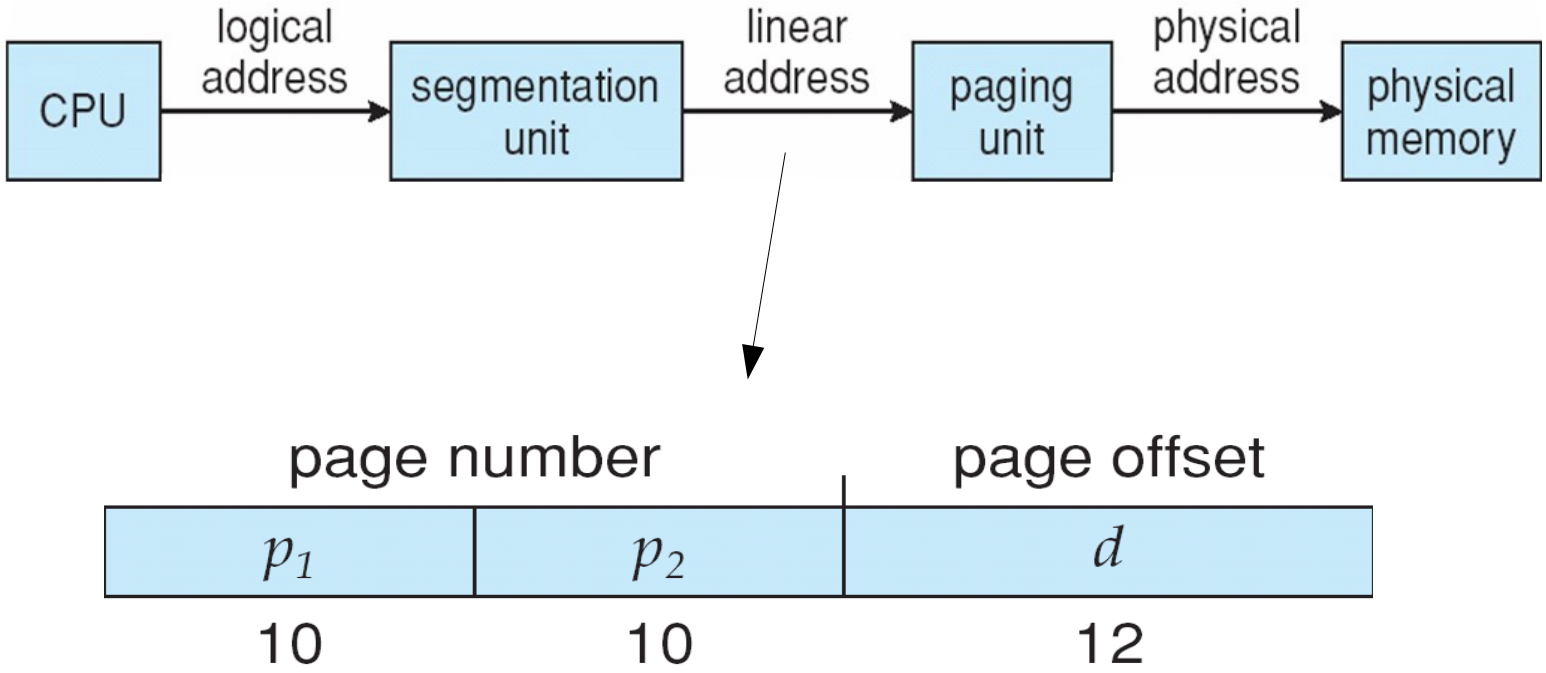


- **Problema**: contiguità della memoria
- Spesso usato assieme alla paginazione

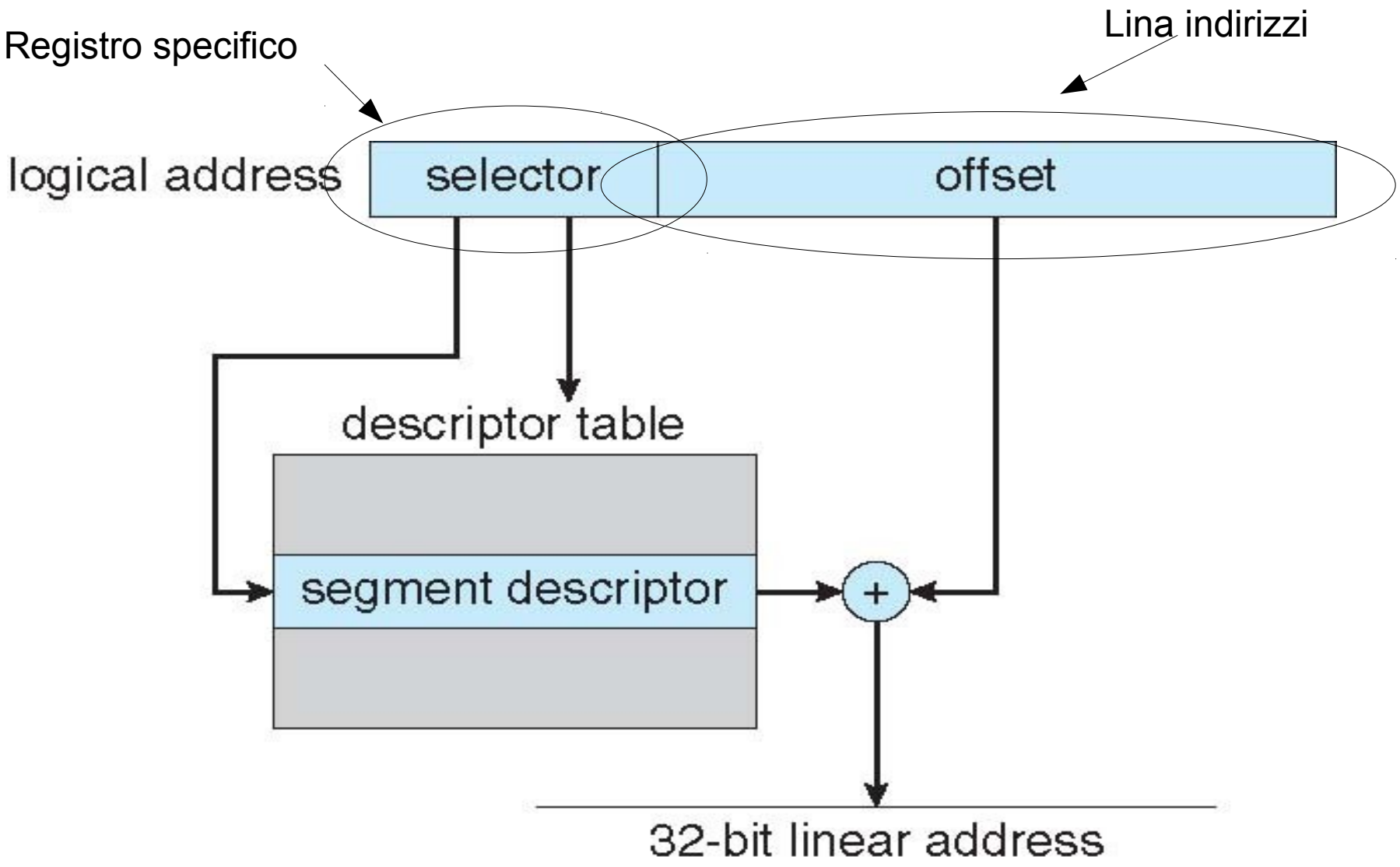
Esempio: Intel Pentium (1/2)

- Supporta paginazione e segmentazione, anche in contemporanea:
  - Al più 16K segmenti, di dimensione al più 4GB l'uno
  - Segmenti divisi in due partizioni:
    - 8K privati ai processi (local descriptor table **LDT**)
    - 8K condivisi tra i processi (global descriptor table GDT)
  - Pagine (e frame) di dimensione 4KB oppure 4MB, con 4KB paginazione gerarchica a 2 livelli
- Segmentazione e paginazione in serie

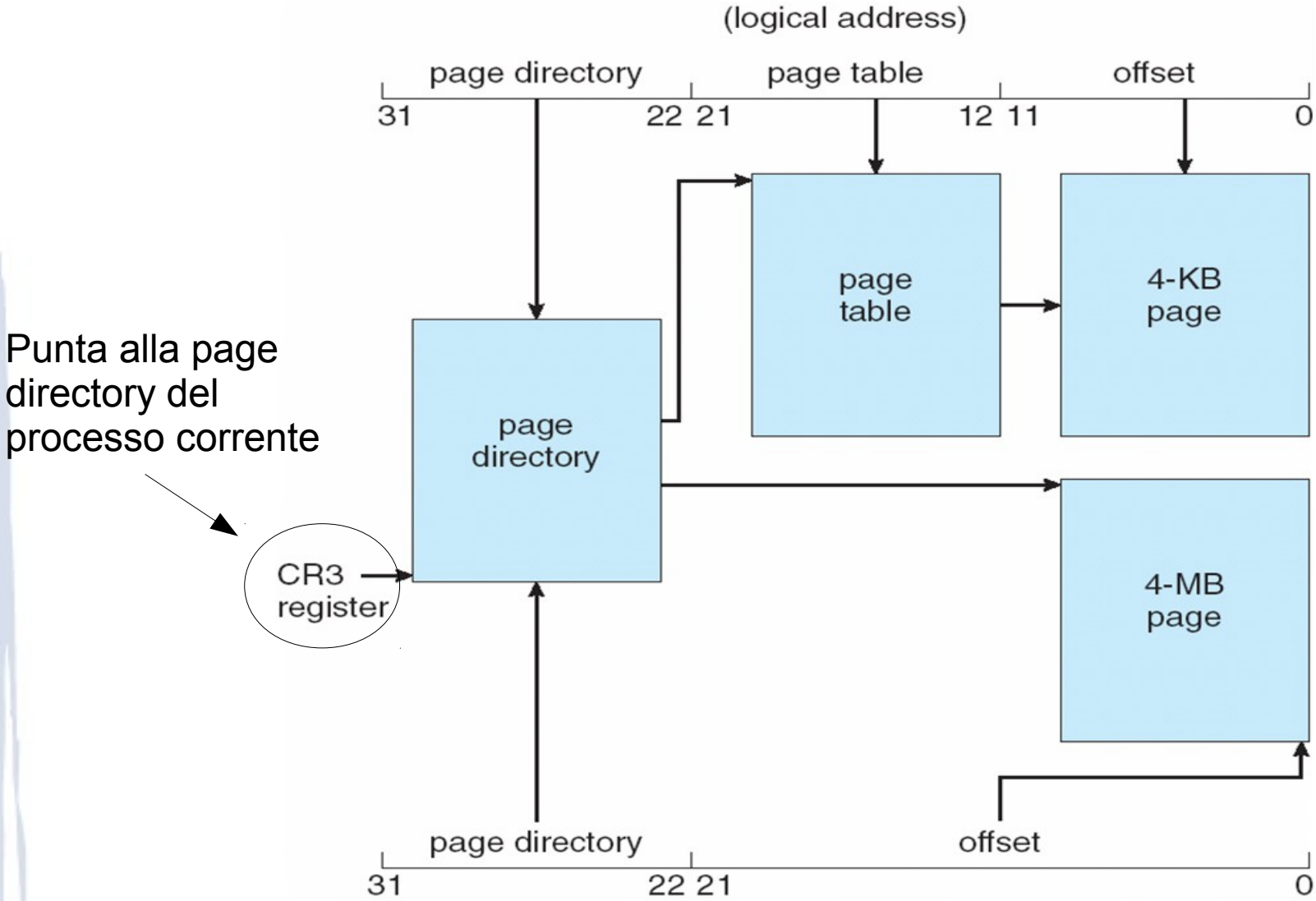
Esempio: Intel Pentium (2/2)



Segmentazione nell'Intel Pentium



Paginazione nell'Intel Pentium



Punta alla page directory del processo corrente