

GESTIONE DELLA MEMORIA

Memoria dal punto di vista di OS

- Un *programma* per essere eseguito deve essere *portato in memoria centrale* ed utilizzare la memoria stessa per trattare con i dati di cui ha bisogno
- Il *tempo di risposta* di un sistema e l'*utilizzazione della CPU* dipendono quindi *anche* dalla gestione della memoria centrale
- Le *strategie* adottabili per la gestione della memoria sono *legate all'hardware* di cui si dispone
- Ai fini di una *gestione di memoria efficiente* non importa come gli *indirizzi* di accesso a memoria siano stati generati, ma *solo la loro sequenza*

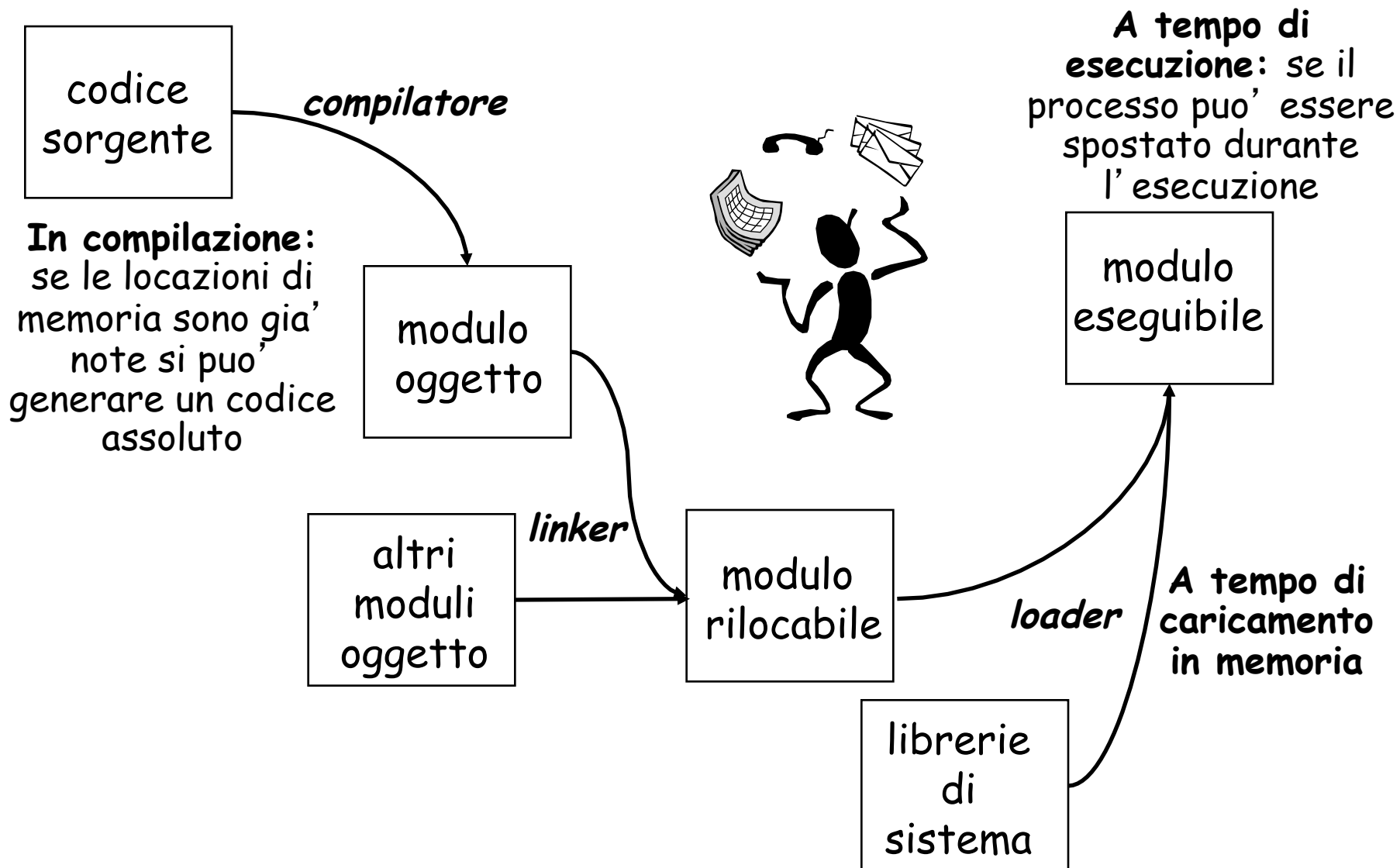
Questione iniziale

Quando vengono *scelti* gli
indirizzi di memoria nei
quali verranno allocati
istruzioni e dati di un
processo???



Associazione (= binding)

istruzioni e dati → indirizzi di memoria



Ulteriori opzioni

- Caricamento dinamico
- Linking dinamico

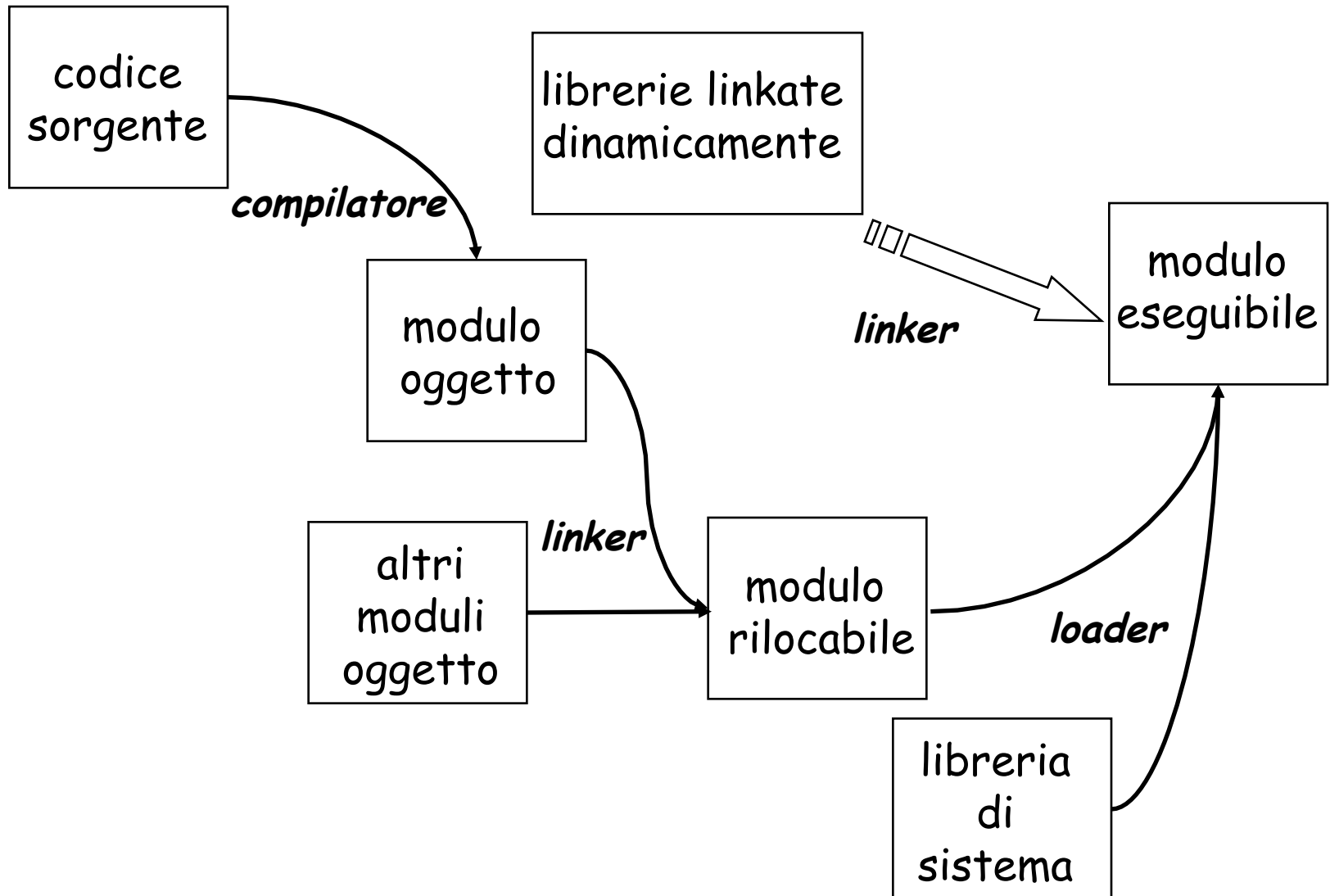
Caricamento dinamico

Una *procedura* non viene *caricata* finche' non viene *invocata*

... vantaggi:

- Migliore *utilizzo della memoria*
- Utile quando *codici di grandi dimensioni* sono necessari per manipolare *casi molto infrequenti*

Linking dinamico ...



... caratteristiche ...

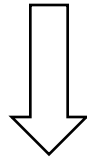
- Il linking viene *postposto a tempo di esecuzione*
- Un piccolo pezzo di codice, *stub*, viene piazzato *al posto della routine* ed utilizzato per localizzare e linkare la routine di libreria
- La routine può *trovarsi già in memoria*, e quindi si deve solo verificare che sia nello spazio degli indirizzi del processo
- La routine può *trovarsi sul disco*, e quindi un'operazione di *caricamento dinamico* deve essere effettuata

... e vantaggi:

- *Non necessaria una copia* della stessa routine in tutti i codici rilocabili che la invocano
- Collegamento automatico a *nuove versioni* della stessa libreria

Che significa virtualizzare la memoria?

Un processo richiede *piu' spazio di quello disponibile* per la sua allocazione



Tenere in memoria solo una parte delle istruzioni e dei dati di un processo : quelli necessari in ogni momento



E su questo ci torneremo ampiamente...

L'idea piu' semplice : *overlays*

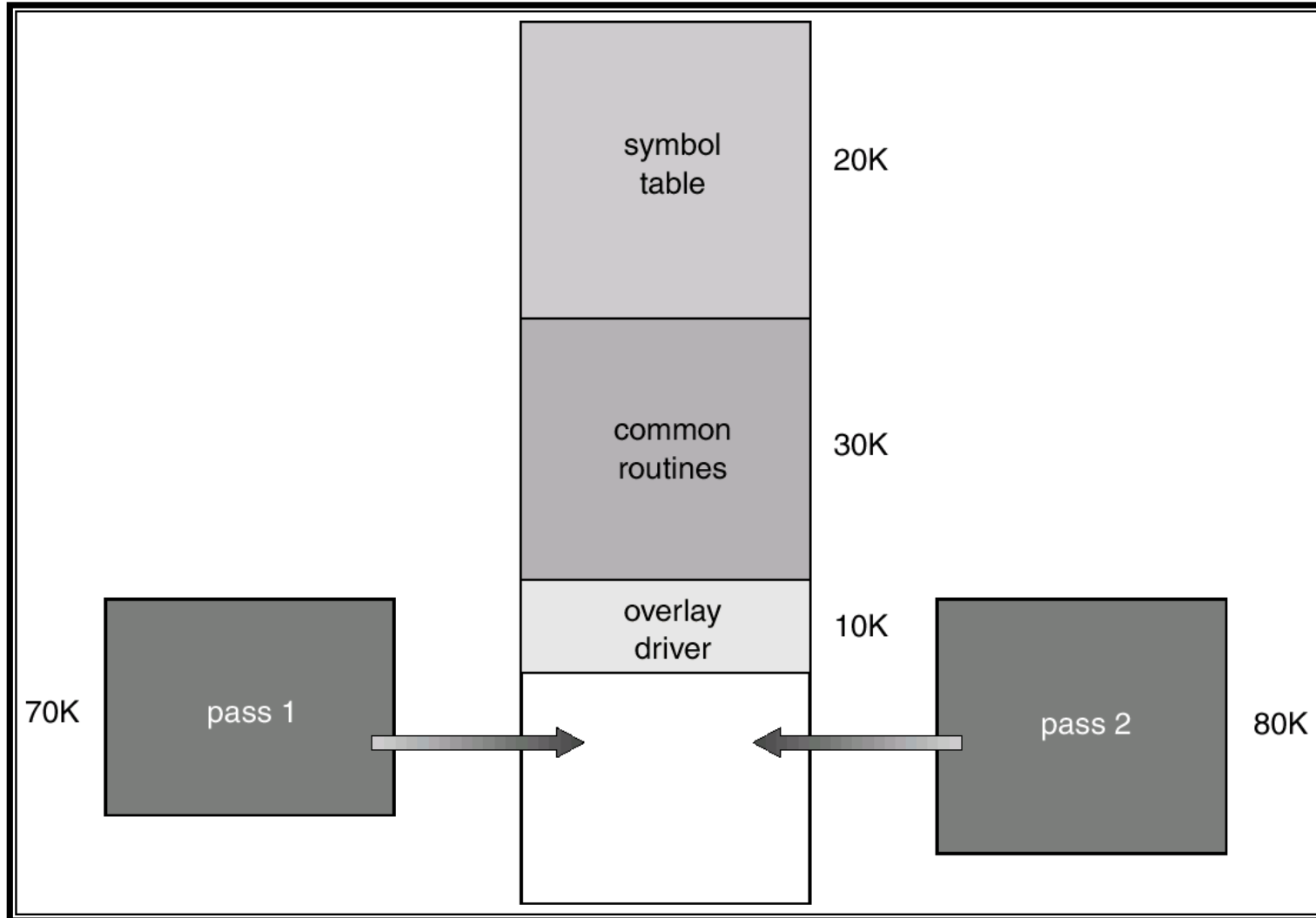
Un programma viene suddiviso in *moduli logicamente distinti*, overlays, che possono essere caricati in memoria in istanti diversi (*overhead di caricamento*)

La *definizione* e la *gestione* degli overlays era *eseguita dall'utente*, che si occupava quindi di implementare anche un gestore di overlays che ne sequenziasse l'esecuzione



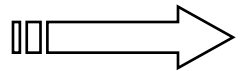
Vedremo tecniche piu' sofisticate...

Esempio di memoria insufficiente



Il concetto di spazio di indirizzi

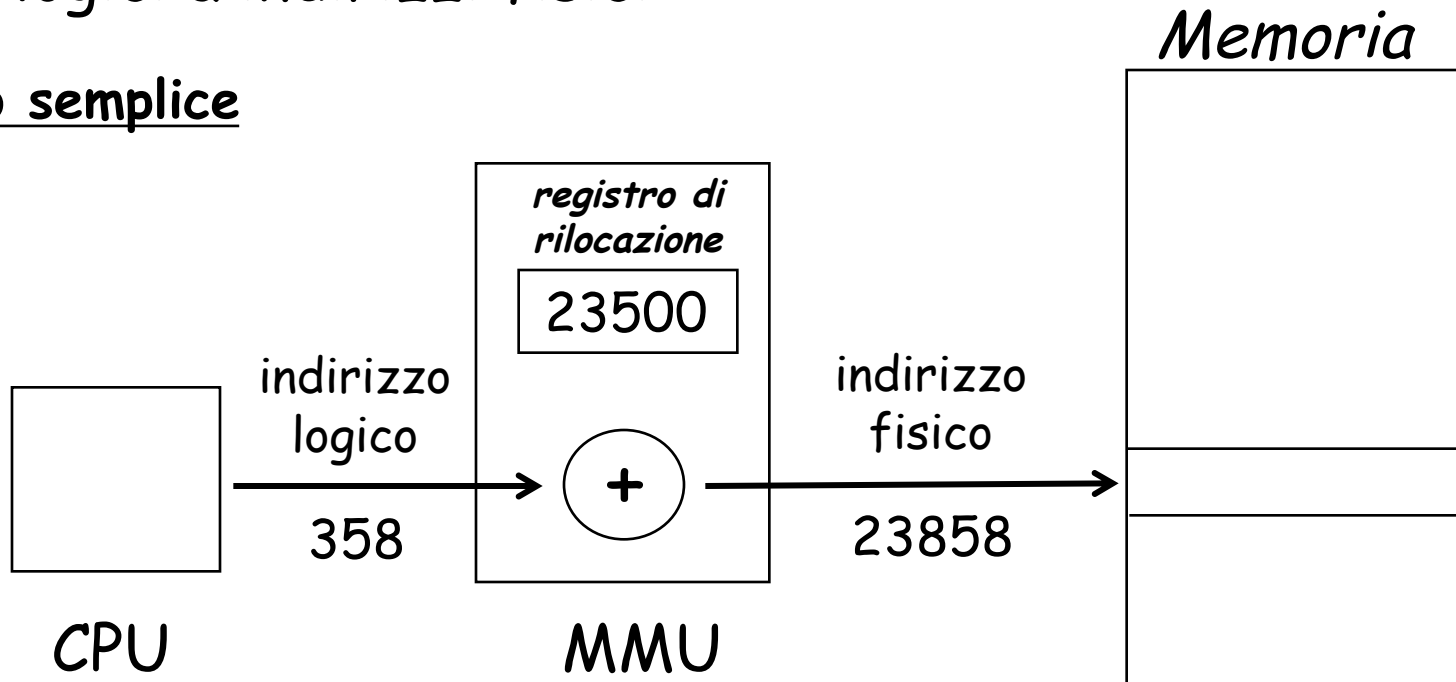
- Un processo si riferisce ad uno spazio di indirizzi logici, che e' associato ad uno spazio di indirizzi fisici
 - **Indirizzo logico (o virtuale)** : generato dalla CPU
 - **Indirizzo fisico** : quello visto e trattato dalla unita' di memoria vera e propria
- Nel **binding a tempo di esecuzione** (e non solo!) indirizzi logici e fisici differiscono...
... chi si occupa di operare la traduzione da logici a fisici?



Memory-Management Unit (MMU)

- I *programmi utente* trattano con *indirizzi logici* e non vedono mai i veri indirizzi fisici
- *MMU* e' il meccanismo che mappa indirizzi logici a indirizzi fisici

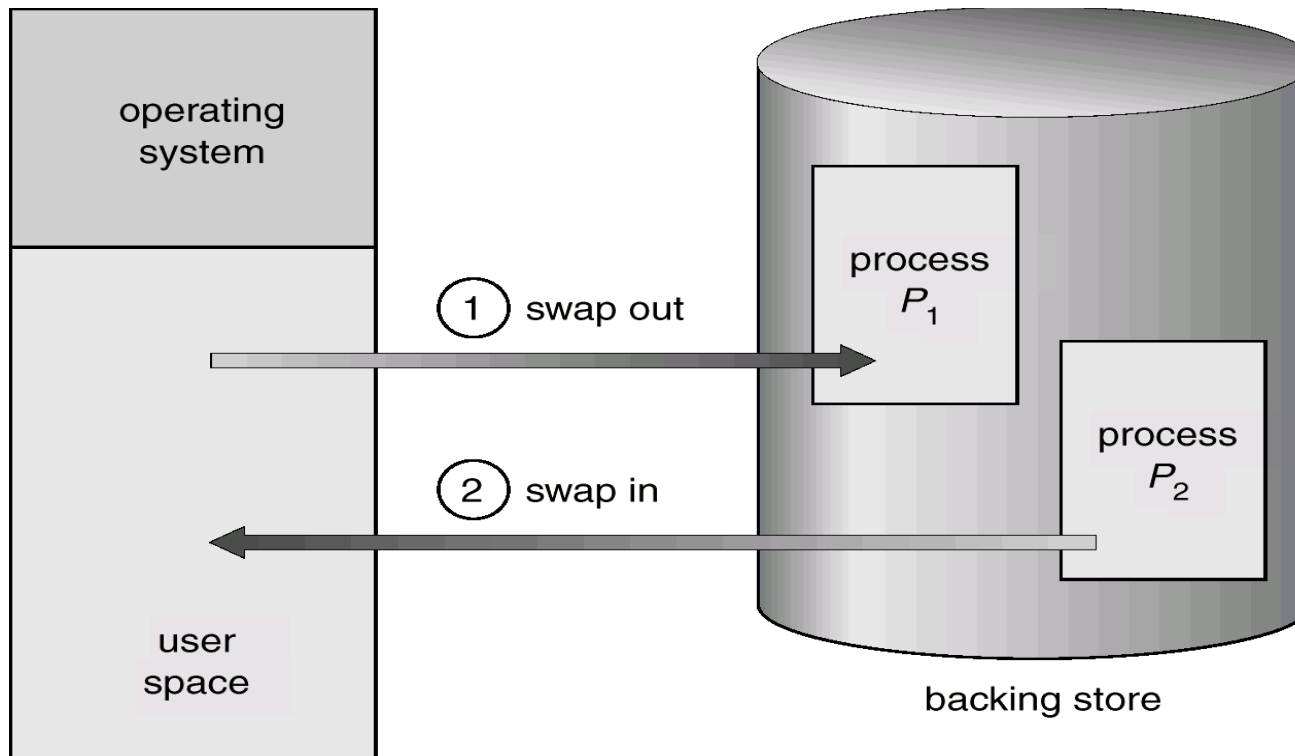
un caso semplice



La strategia interna alla MMU puo' essere ***molto piu' complessa*** di un semplice registro di rilocalizzazione!

Swapping...

Un processo può essere *portato via dalla memoria* (swapped out) ad una memoria secondaria, ed in seguito *riportato in memoria* (swapped in) per proseguire la sua esecuzione



Ricordate il medium-term scheduler?

Lo swapping può essere applicato anche a parti di un processo (p.es. singole pagine), come vedremo....

... e alcune considerazioni

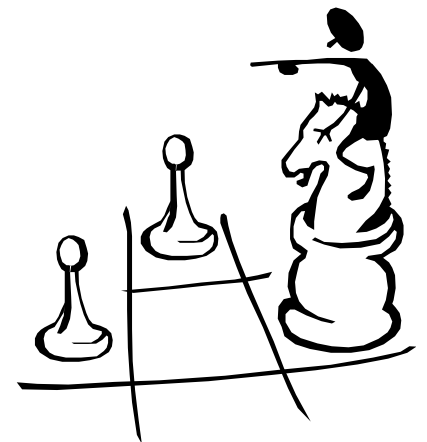
- Il *tempo di swapping* e' proporzionale all'ammontare di memoria swapped
- Ha senso solo se si possono *tenere in memoria piu' processi alla volta* e quindi lo swapping non avviene ad ogni context-switching e non coinvolge tutta la memoria utente
- E' come se la *ready queue* proseguisse pure *su disco* (*coda di input*)

... e alcune considerazioni

- Il criterio di swapping si puo' basare, per esempio, sulla ***priorita' dei processi***
- A causa del suo costo elevato, si puo' pensare ad uno ***swapping non attivo sempre***
- Problemi relativi a ***processi swapped out*** mentre stavano ***su code di dispositivi di I/O*** (scrittura in buffer di I/O non piu' validi)
- Solo con ***binding a tempo di esecuzione*** un processo puo' essere ricaricato in una zona diversa della memoria centrale

Metodologie fondamentali di allocazione della memoria

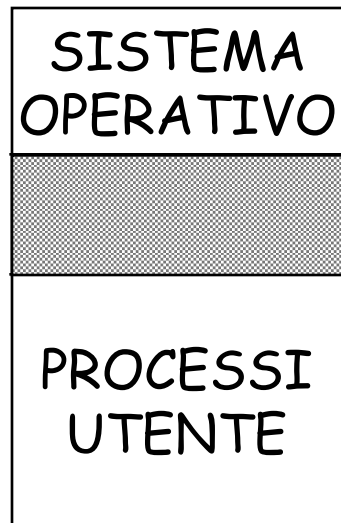
- Allocazione contigua
- Paginazione
- Segmentazione



Allocazione contigua

La memoria e' solitamente suddivisa in due "zone":

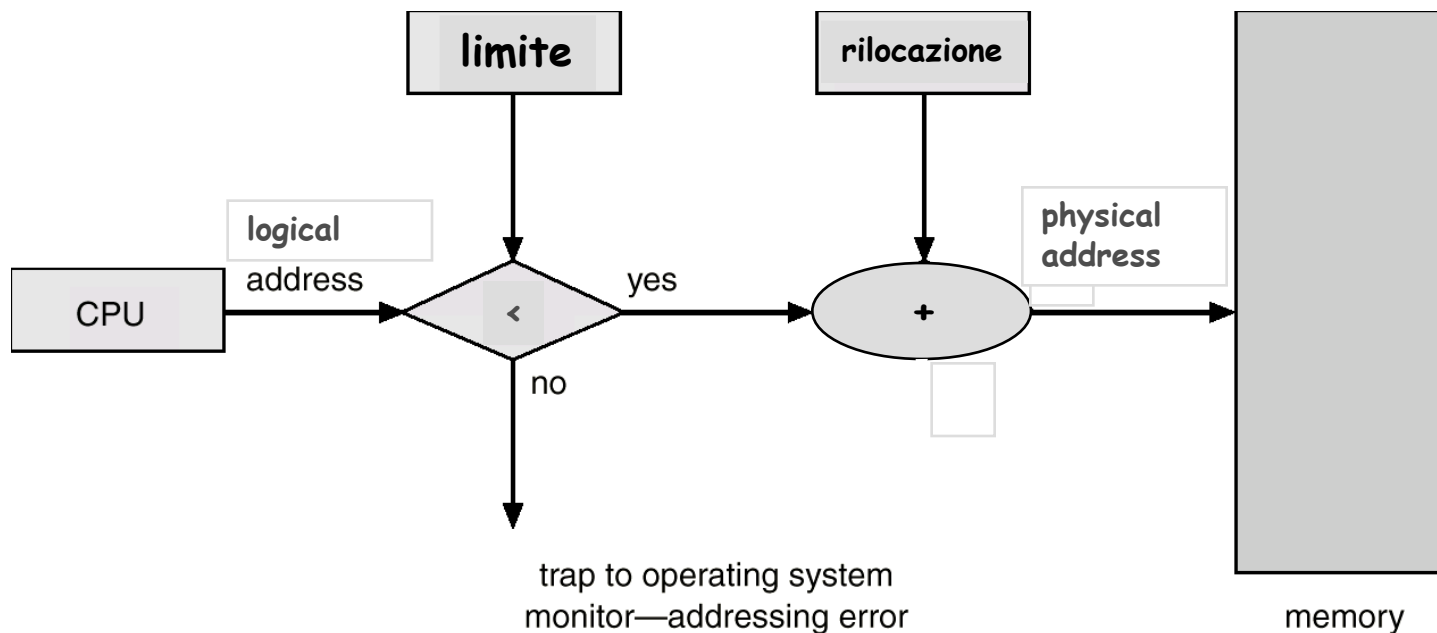
- *Sistema operativo* (se necessario con spazio per l' interrupt vector)
- *Processi utente*



Anche *OS* stesso puo' avere *dimensione variabile* in memoria :
ex. device drivers

Partizioni

Registri di *rilocalizzazione* (base) e *limite* usati per proteggere OS dai processi utente



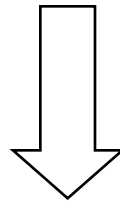
Rilocalizzazione : il piu' piccolo indirizzo fisico

Limite : il piu' grande indirizzo logico

Partizioni multiple : piu' processi in memoria allo stesso tempo

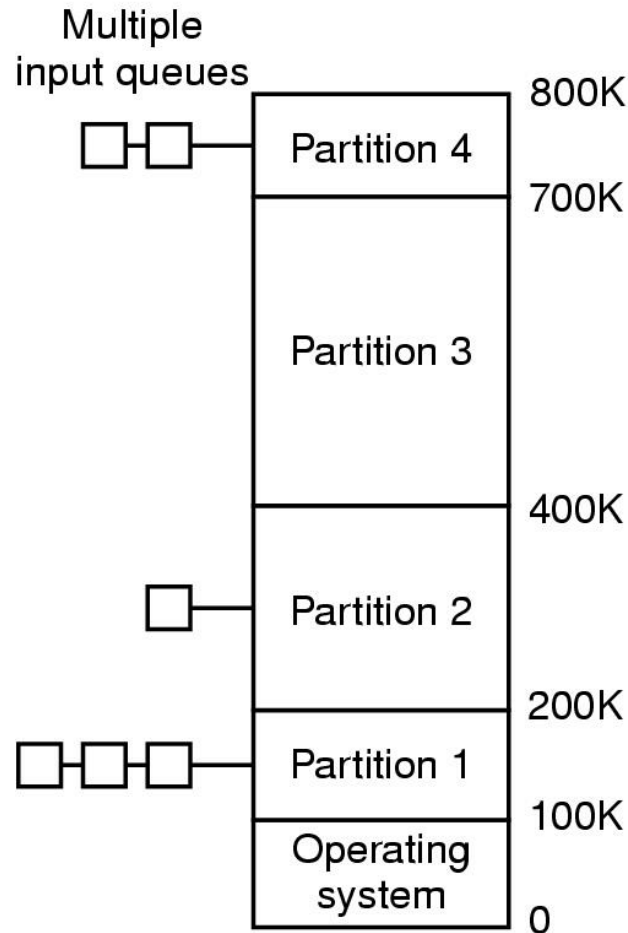
IDEA BANALE:

Suddivisione della memoria in un numero fisso di *partizioni* tutte *aventi la stessa dimensione*

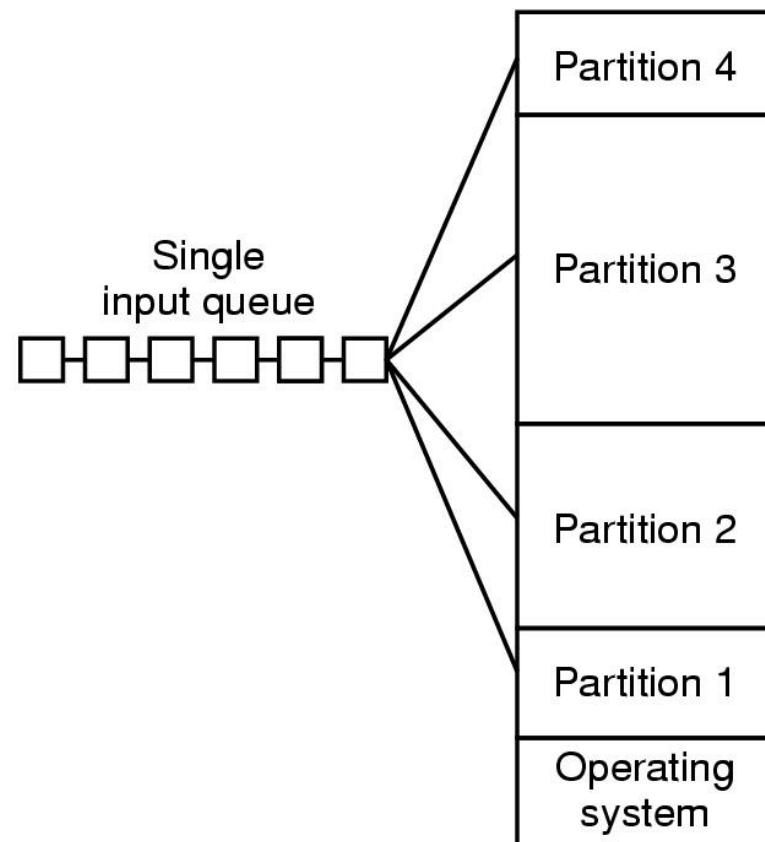


- *Grado di multiprogrammazione* prefissato
- *Taglia massima* di processo che puo' essere eseguito

... ma si puo' anche pensare a partizione fisse di dimensioni diverse

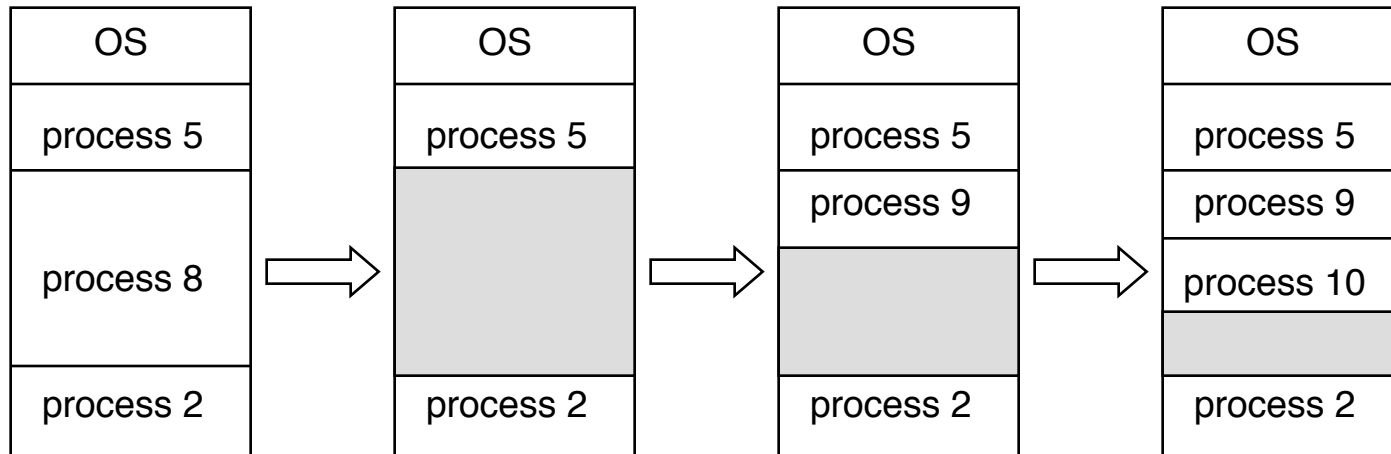


(a)



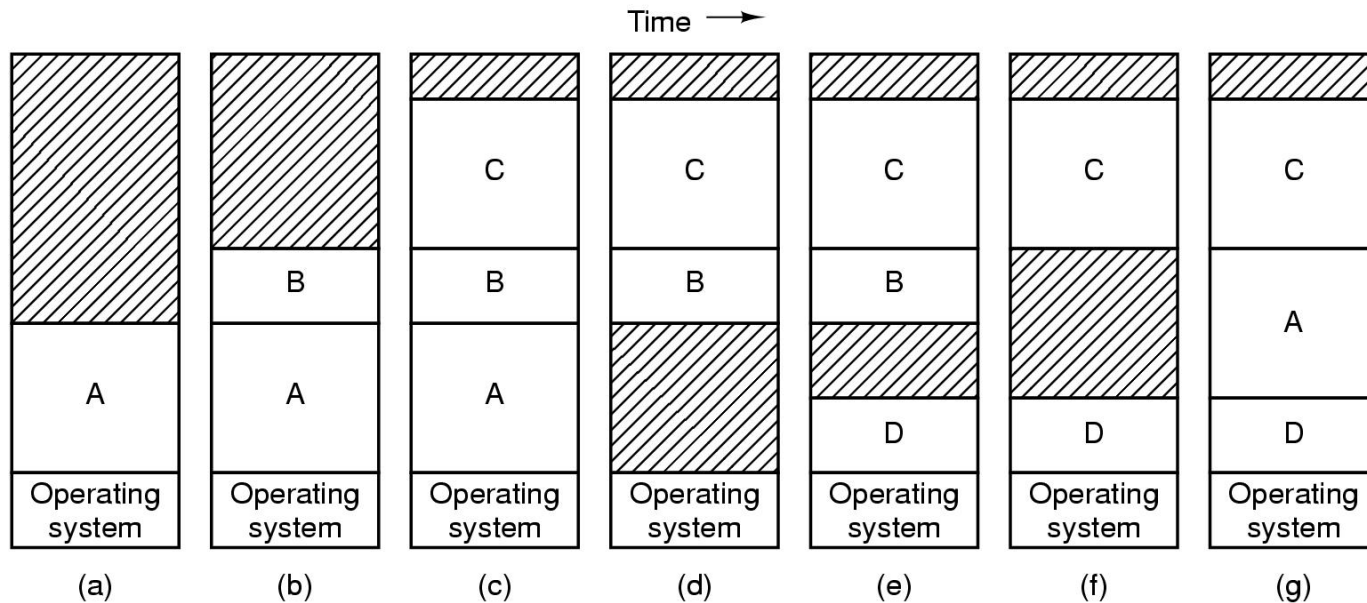
(b)

Partizioni multiple a dimensione variabile (in pratica, assenza di partizioni predefinite)



- **Buco** - blocco di memoria disponibile
- Quando un processo arriva, gli viene **allocato un buco di memoria grande abbastanza** per sistemare il processo

Partizioni multiple a dimensione variabile (in pratica, assenza di partizioni predefinite)



- *Buchi di dimensione varia* vengono originati lungo la memoria
- OS deve mantenere *traccia* delle *partizioni allocate* e dei *buchi*

Tabella dei buchi : collocazione e taglia di ogni buco
Coda di input : processi pronti che non sono in memoria

... e qualche considerazione ...

- Se il prossimo processo candidato ad entrare in memoria non trova un buco di dimensione adeguata alla sua taglia, la *strategia di ingresso in memoria* puo' anche considerare il processo seguente, e cosi' via...
- La *tabella dei buchi* viene *aggiornata* ogni volta che un processo entra o esce dalla memoria
- Quando un processo esce dalla memoria puo' essere un momento opportuno per *dare uno sguardo alla coda di input*

Un problema generale

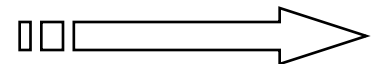
Allocazione dinamica della memoria : come soddisfare una richiesta di *taglia n* da una tabella/lista di buchi?!

- **First-fit**: Alloca il *primo* buco che e' grande abbastanza
- **Best-fit**: Alloca il *piu' piccolo* buco che e' grande abbastanza
- **Worst-fit**: Alloca il *piu' grande* buco che e' grande abbastanza



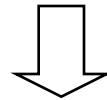
Parametri per la valutazione di una strategia :

- *velocita'* di allocazione (*first-fit*)
- *utilizzazione* della memoria



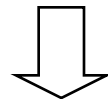
Un problema di utilizzazione della memoria: la frammentazione

Quando la differenza tra memoria richiesta e buco disponibile e' molto piccola vengono allocati blocchi leggermente piu' grandi perche' sarebbe oneroso (e inutile?) tenere traccia di tutti

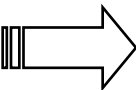


Frammentazione interna - distribuzione di spazio libero *all'interno di blocchi occupati*

Esiste spazio totale in memoria per soddisfare una richiesta, ma tale spazio non e' contiguo



Frammentazione esterna - distribuzione di spazio libero *tra blocchi occupati*



La *frammentazione esterna* e' piu' comune e *piu' grave* in termini di *utilizzazione di memoria*

In media viene *sprecato un terzo dello spazio*

First-fit e *best-fit* sono migliori di *worst-fit* in quanto a velocita' di allocazione e utilizzazione di memoria

Una soluzione immediata : *la compattazione*

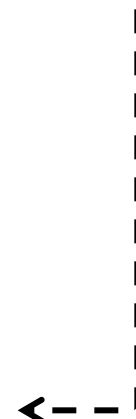
Spostare i contenuti della memoria per piazzare tutta la memoria disponibile in *un unico buco grande*

(Naturalmente e' possibile solo se *l'associazione degli indirizzi e' a tempo di esecuzione*)

Compattazione

Varie strategie di compattazione (esempi)

SISTEMA OPERATIVO	
P1	200K
	100K
P2	500K
	50K
P3	300K
	400K



Quanto spesso compattare? Se lo swapping e' gia' parte di OS, il codice per la compattazione e' minimo

Soluzioni piu' raffinate alla frammentazione : allocazione non contigua della memoria

1. PAGINAZIONE

2. SEGMENTAZIONE



1. Paginazione : idea di base

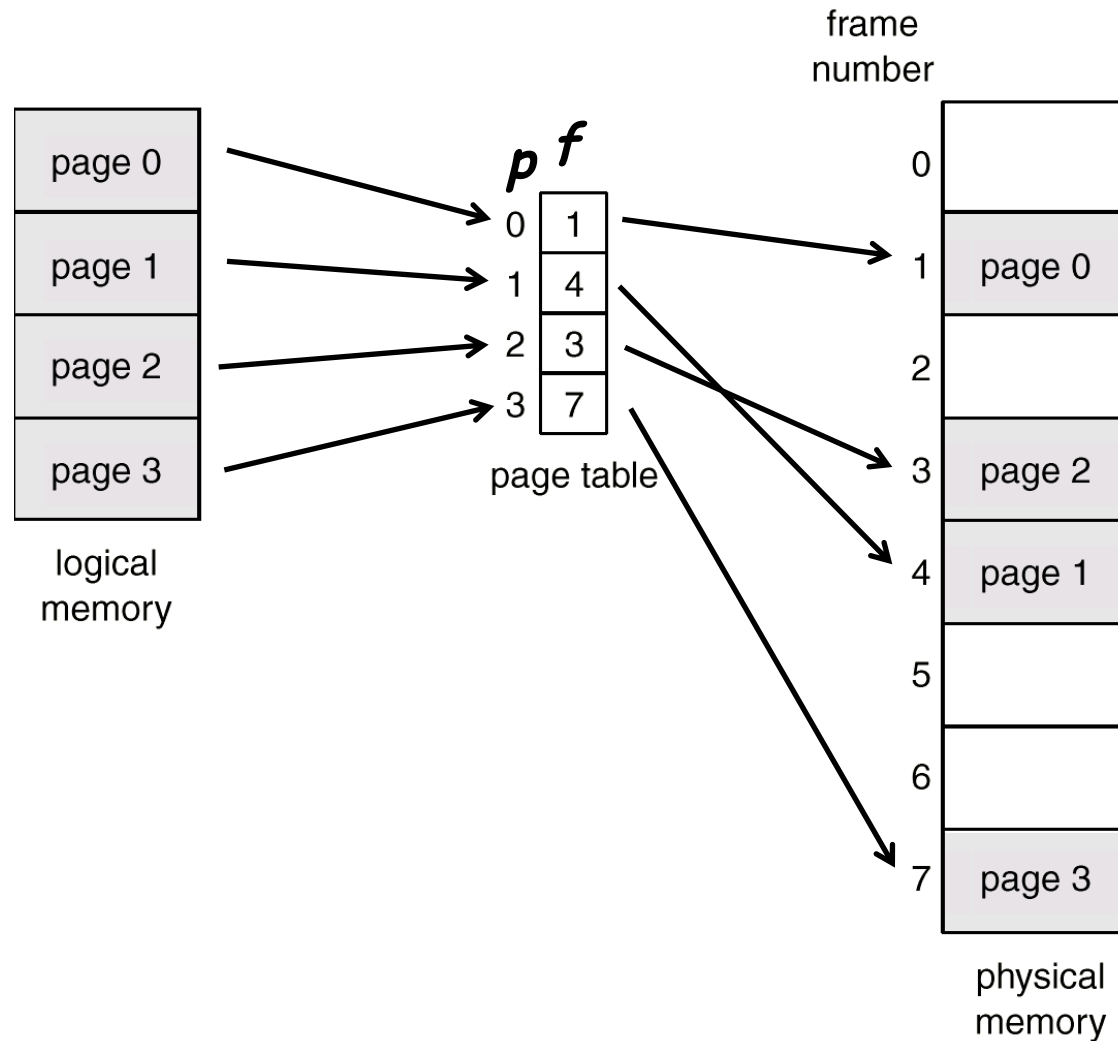
Lo spazio di indirizzi logici di un processo puo' essere non contiguo

- Si suddivide la *memoria fisica* in blocchi di taglia fissa chiamati *frame*
- Si suddivide la *memoria logica* in blocchi della stessa taglia chiamati *pagine*

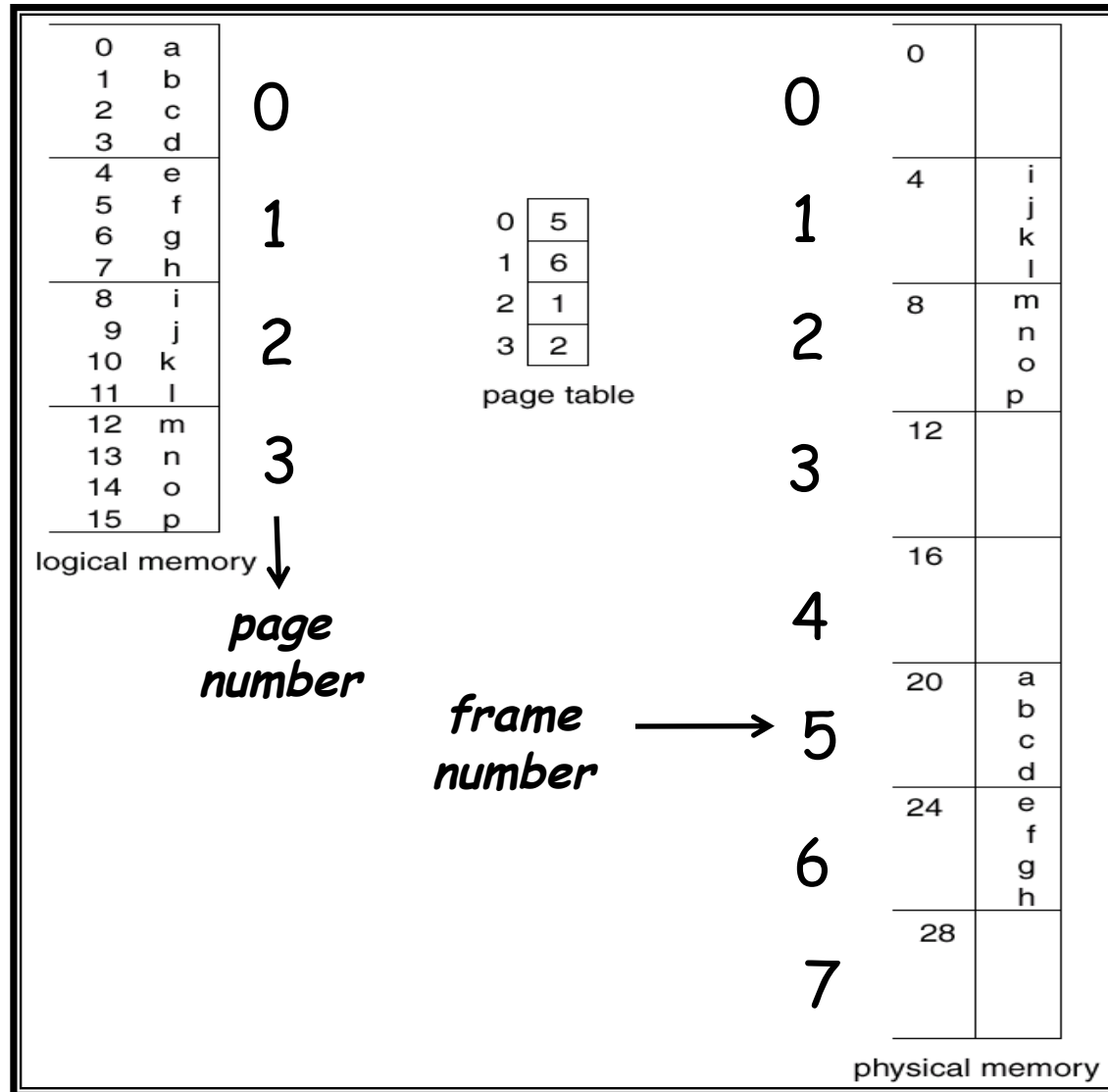
Per eseguire un *processo di n pagine*, si devono trovare *n frame liberi*

Per gestire le pagine di memoria, per ogni processo viene introdotta una *page table*

Un esempio di page table



... ed un altro esempio ...



... e qualche considerazione

- La *dimensione fissa* dei frame evita anche il problema di *allocare blocchi* di dimensioni diverse *nel disco* quando vengono deallocati
- La *frammentazione interna* e' limitata solo all'ultima pagina allocata ad un processo
- *Dimensione* di una pagina (pagine piccole comunque generano page table con molte entries)
- *Tabella/lista dei frame* (liberi e occupati) gestita da OS

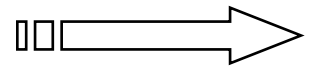
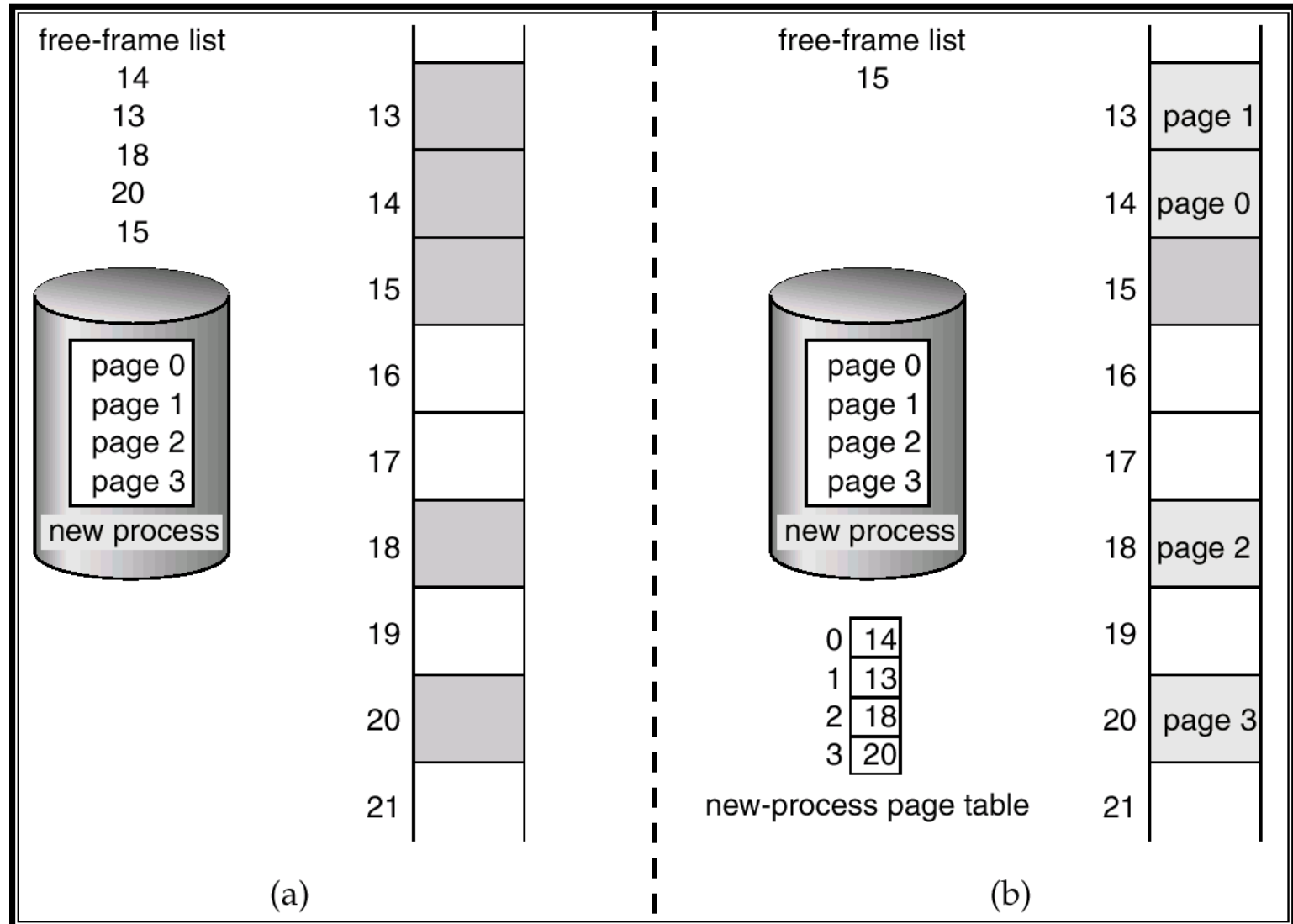


Tabella frame liberi



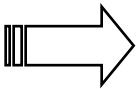
Come cambia la gestione di un processo ?!



- Una page table per ogni processo
- *Un campo in piu' nel PCB* : (il puntatore alla) page table
- *Aggiornamento dei frame* quando il processo viene scaricaricato o ricaricato
- *Context switching piu' oneroso* in quanto coinvolge anche la gestione delle pagine

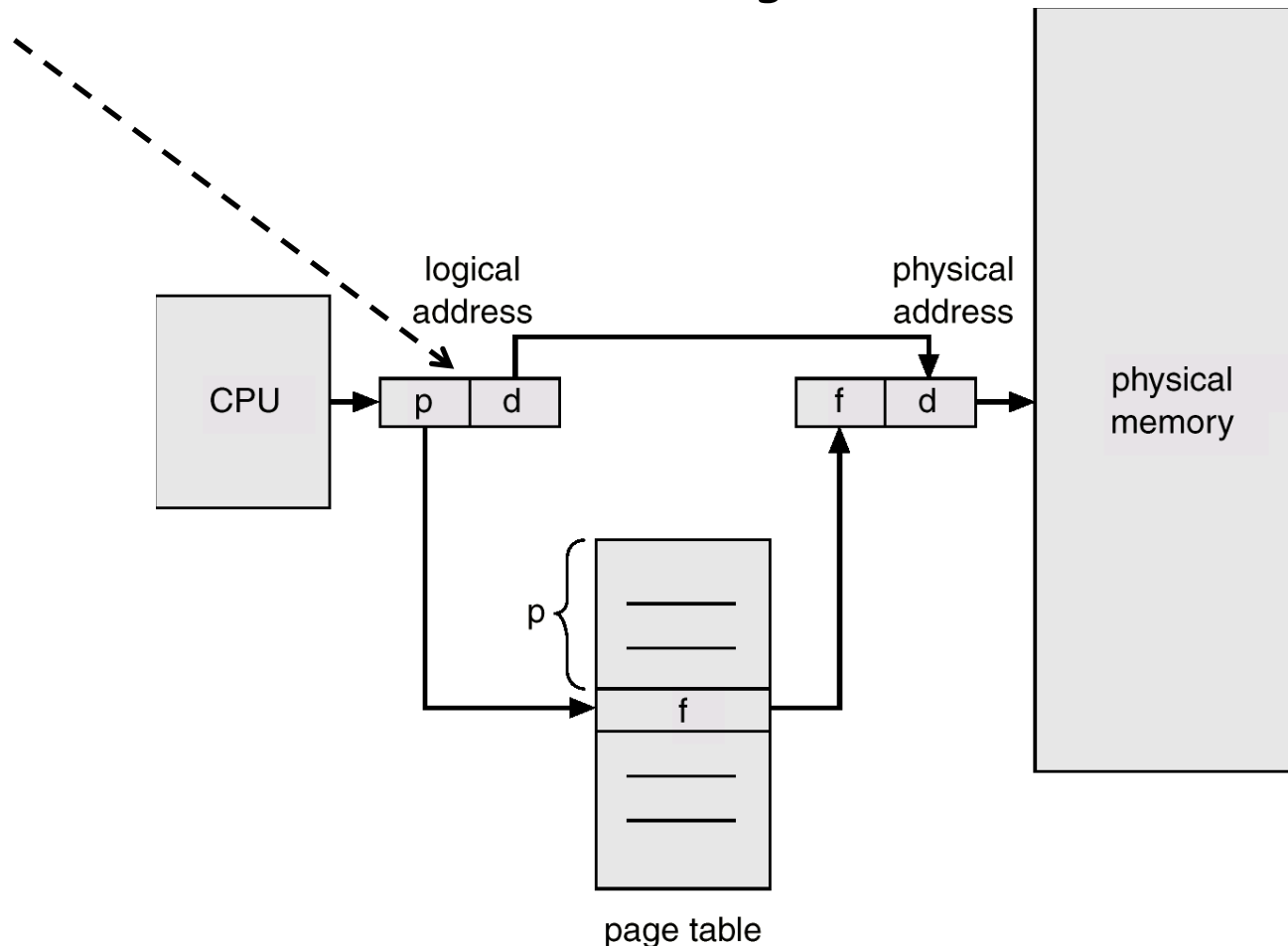
Supporto alla paginazione ...

- Tenere traccia di tutti i *frame liberi*
- Organizzare una *page table* per tradurre indirizzi logici in fisici
- Un indirizzo generato dalla CPU (logico) viene quindi suddiviso in :
 - *Numero di pagina (p)* - usato come *indice in una page table* che contiene l'indirizzo di base di ogni pagina nella memoria fisica
 - *Spiazzamento (d)* - combinato con l'indirizzo di base della pagina definisce l'indirizzo fisico di memoria



... uno schema illustrativo ...

Una *taglia* di pagina *potenza di 2* serve per la corretta suddivisione dell'indirizzo logico

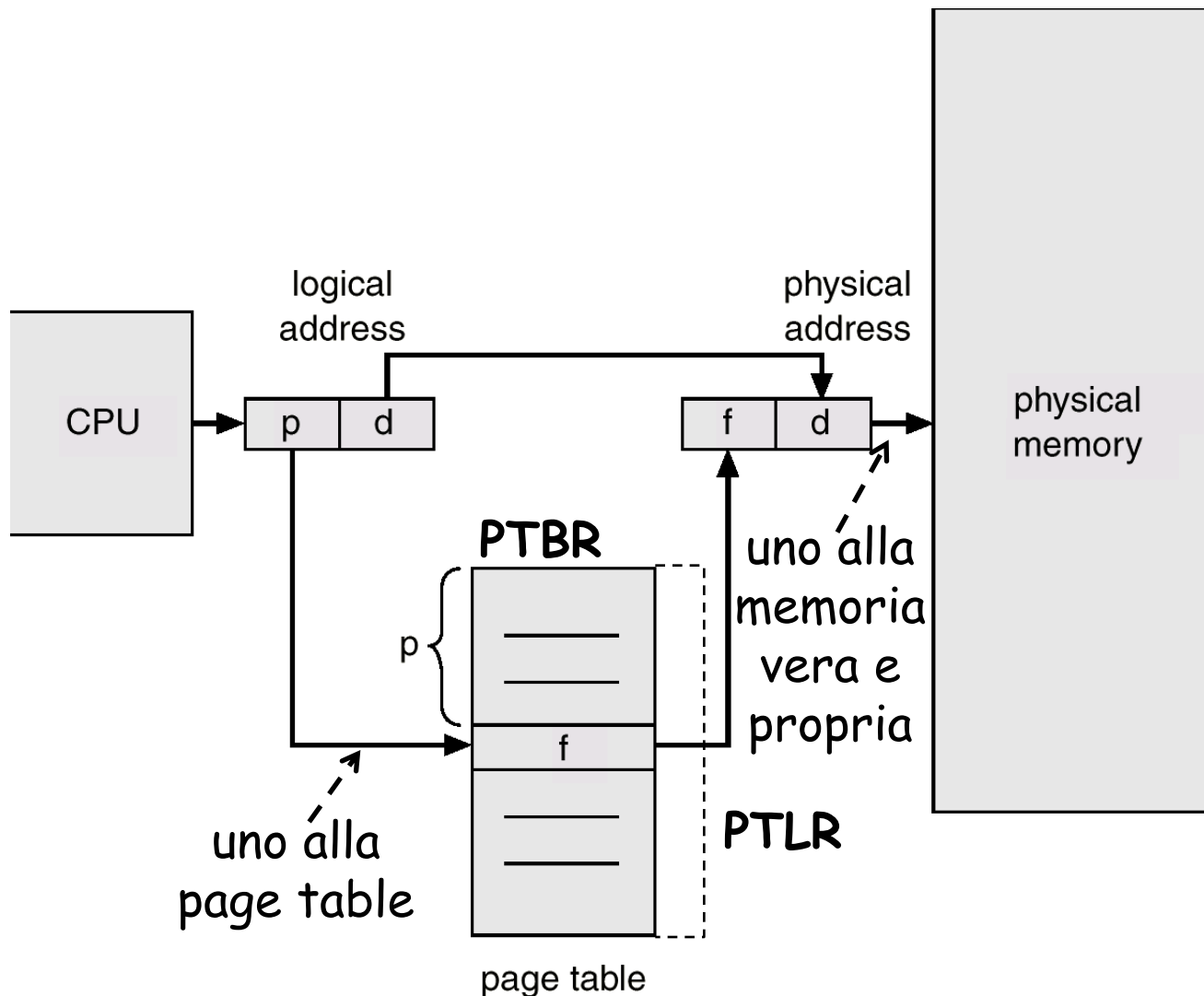


Implementazione di una page table

La page table puo' essere *tenuta in memoria* con il supporto di:

- *Page-table base register* (PTBR) punta all' *inizio* della page table
- *Page-table length register* (PTLR) indica la *taglia* della page table

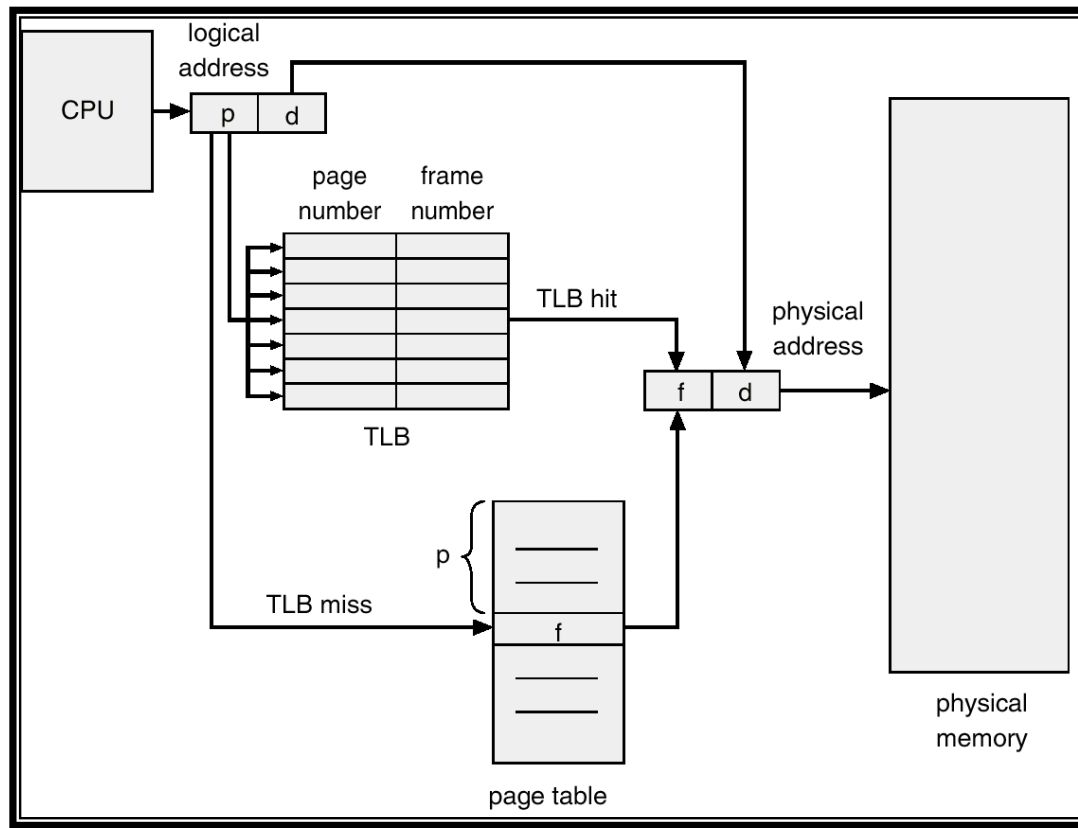
... e una raffigurazione



Ogni accesso a un'informazione e richiede quindi ***due accessi a memoria***

Soluzione al doppio accesso: memoria associativa

Una speciale cache chiamata *translation look-aside buffers* (TLBs)



Traduzione dell'indirizzo (P, D)

Se P e' in TLBs prendi
il numero di frame F

Altrimenti recupera il
numero di frame F
dalla page table in
memoria e riportalo
anche in TLBs

In caso di context switching TLBs va ripulita

Tempo di accesso effettivo (EAT) con memoria associativa

- *Accesso alla cache* : ε microsecondi
- *Accesso alla memoria* : 1 microsecondo
- *Tasso di hit* : α - percentuale di volte che un numero di pagina viene trovato in TLBs
(*proporzionale al numero di registri*)

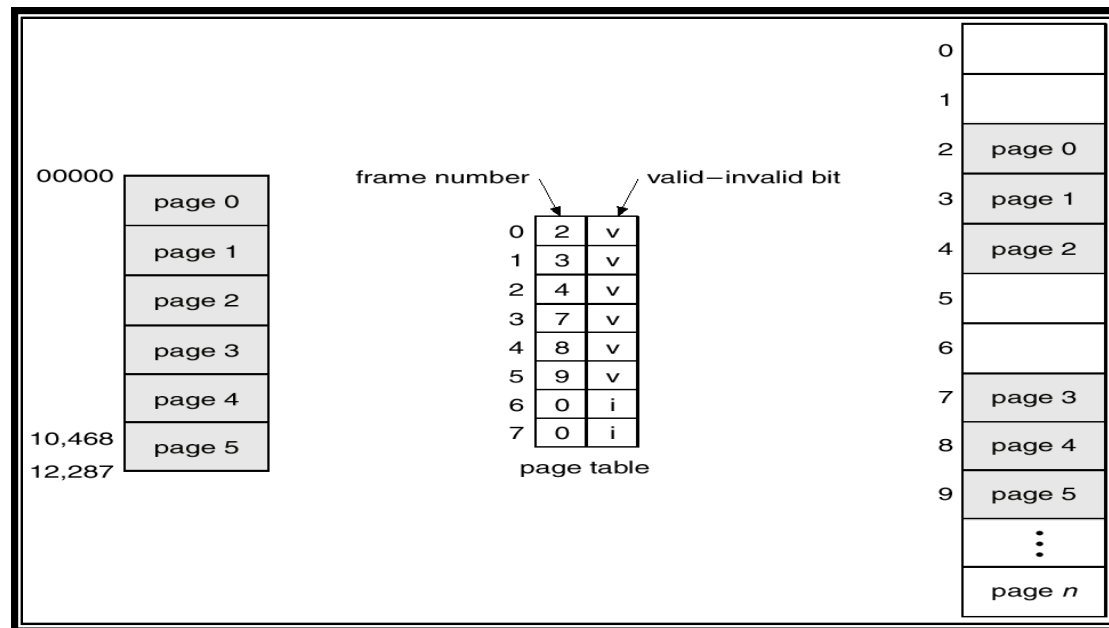
$$EAT = (1 + \varepsilon) \alpha + (2 + \varepsilon)(1 - \alpha) = 2 + \varepsilon - \alpha$$

*Provate ad assegnare dei valori per
comprendere l'andamento di questa funzione!*

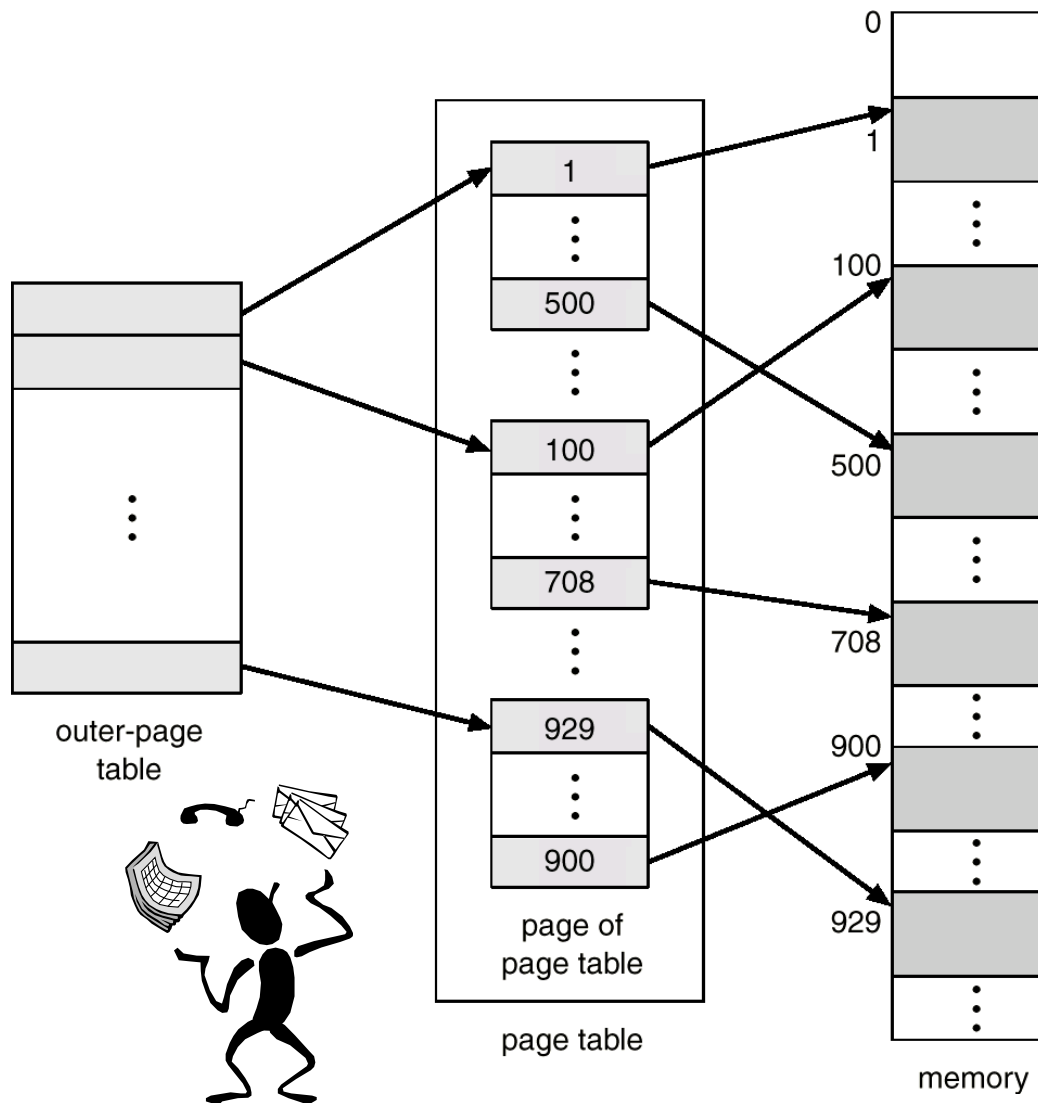


Protezione della memoria paginata

- Bit di protezione possono essere associati ad ogni frame:
lettura, scrittura, esecuzione
- Un bit *di validita'* puo' essere associato ad ogni entry di una page table:
valid indica che la pagina e' nello spazio di indirizzi logici del processo



E per page table troppo lunghe...



... allocazione
non contigua
della page table

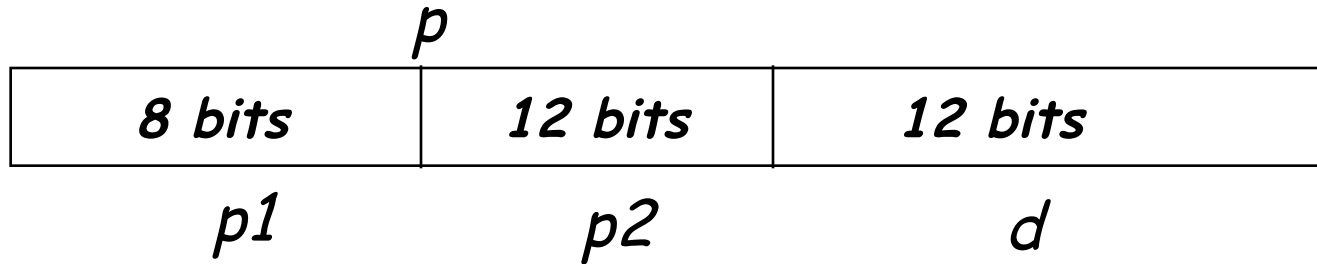
Paginazione a
due livelli:

***paginazione
della page
table***

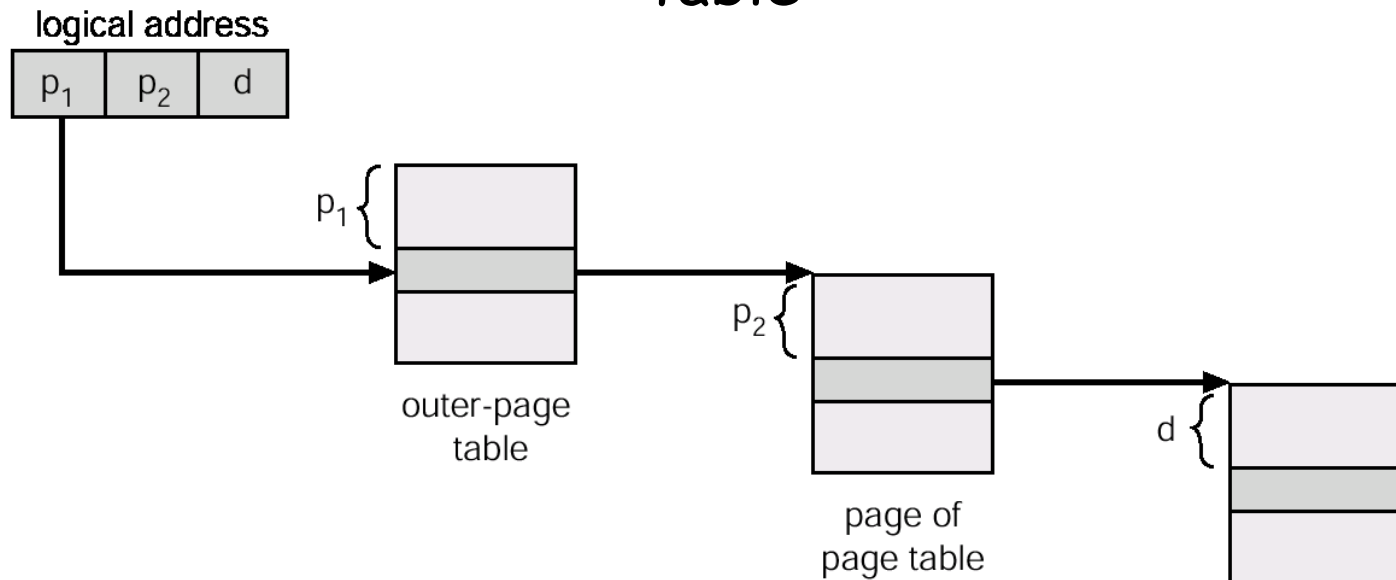
... un esempio ...

- Un indirizzo logico (su una macchina a 32-bit con una pagina di taglia 4K) viene suddiviso in:
 - *un numero di pagina* di 20 bits (p)
 - *un offset* di 12 bits (d)
- Poiche' *la page table e' paginata* (p.es. con pagine di taglia 1K), il numero di pagina viene a sua volta suddiviso in:
 - *un numero di pagina* di 8 bits (p1)
 - *un offset all'interno della pagina della page table* di 12 bits (p2)

Un indirizzo logico ha quindi il seguente formato:



dove $p1$ e' l' indice della outer page table, e $p2$ e' lo spiazzamento all' interno della pagina della page table



... e tempo di accesso

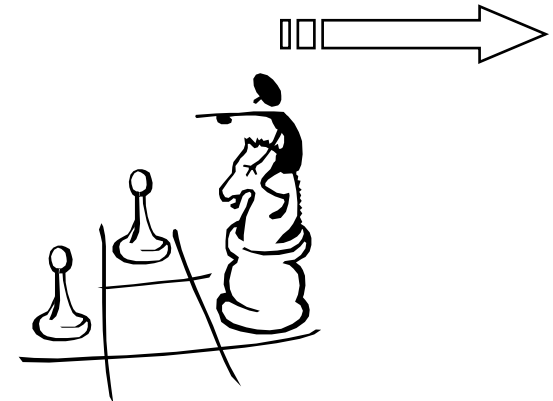
- Poiche' ogni livello e' memorizzato come una separata tabella in memoria, *la traduzione da logico a fisico puo' implicare 3 accessi a memoria* nel caso di paginazione della page table (i livelli possono essere anche di piu' !)
- Il *caching* potrebbe permettere di tenere un tempo di accesso ragionevole
- Un *hit ratio* del 98% porta a :

$$EAT = 0.98 \times (1 + \varepsilon) + 0.02 \times (3 + \varepsilon) = 1 + (\varepsilon + 0.04) \mu s$$

che e' un rallentamento di $(\varepsilon + 0.04)$ del tempo di accesso a memoria

Condivisione di pagine

- **IDEA** : Mappare una stessa pagina fisica su piu' pagine logiche, in modo da condividere applicazioni e/o dati
- (((Non facilmente praticabile con una *inverted page table*)))
- Se si tratta di **codice** :
 - non deve essere in grado di modificare se' stesso
 - deve apparire nella stessa locazione dello spazio logico di tutti i processi
- **Casi di uso** : editor, compilatori, etc.



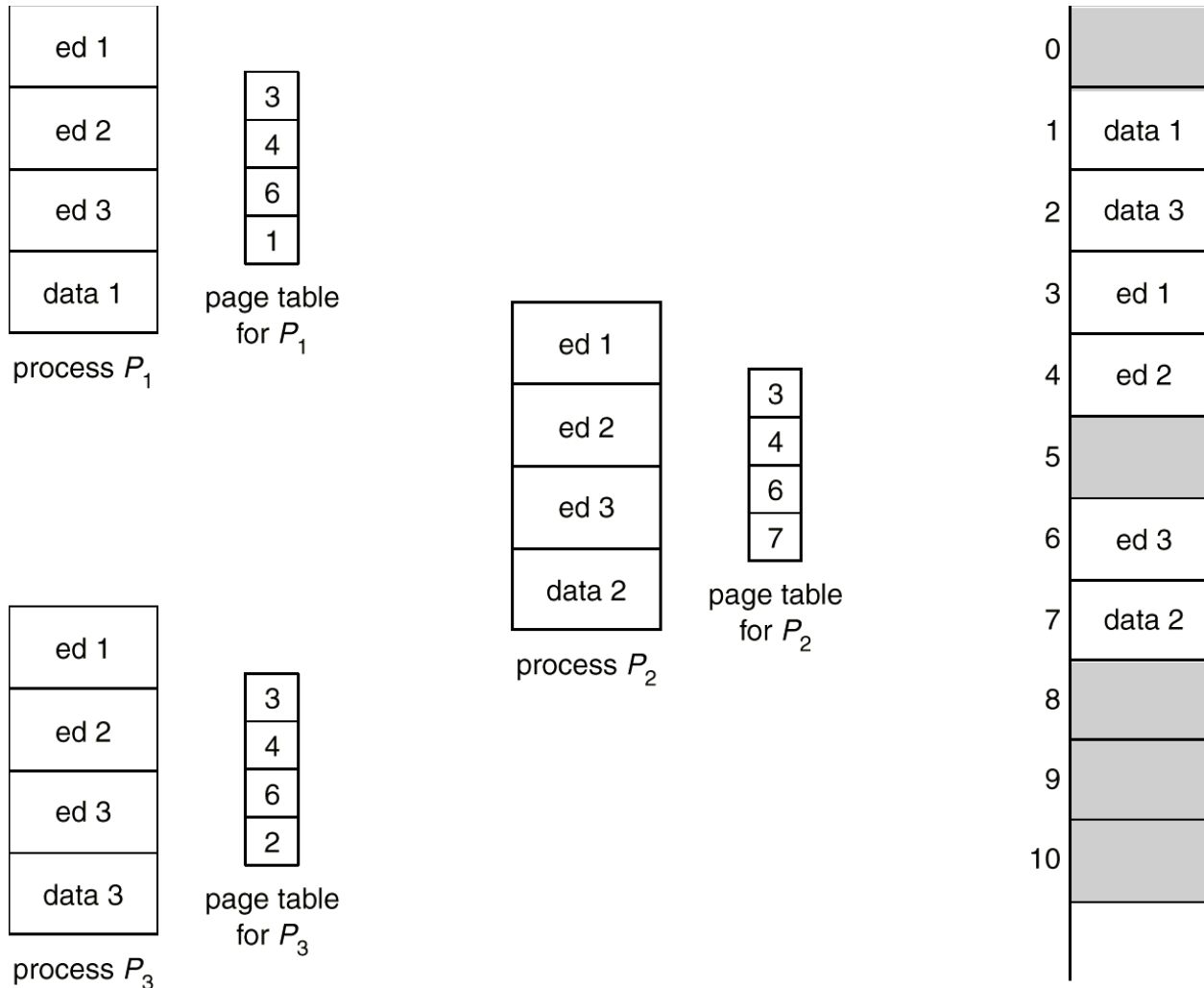
E perche' la pagina condivisa deve apparire
nella stessa locazione dello spazio logico di
tutti i processi ???

Gli *autoriferimenti alla pagina condivisa* implicano
che essa abbia lo stesso numero logico nei diversi
programmi perche' *gli indirizzi che essa contiene*
possono essere del tipo *<pagina logica, offset>*

Invece *indirizzi/riferimenti che sono espressi in
termini di offset* dal program counter attuale o da un
registro che contiene il corrente numero di pagina
sono *indiretti* e quindi vanno bene

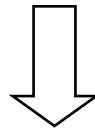


... ed un esempio di condivisione



Inverted Page Table

La page table di ogni processo deve avere *una entry per ogni pagina logica*, a prescindere dal fatto che questa sia *realmente allocata in memoria o meno*

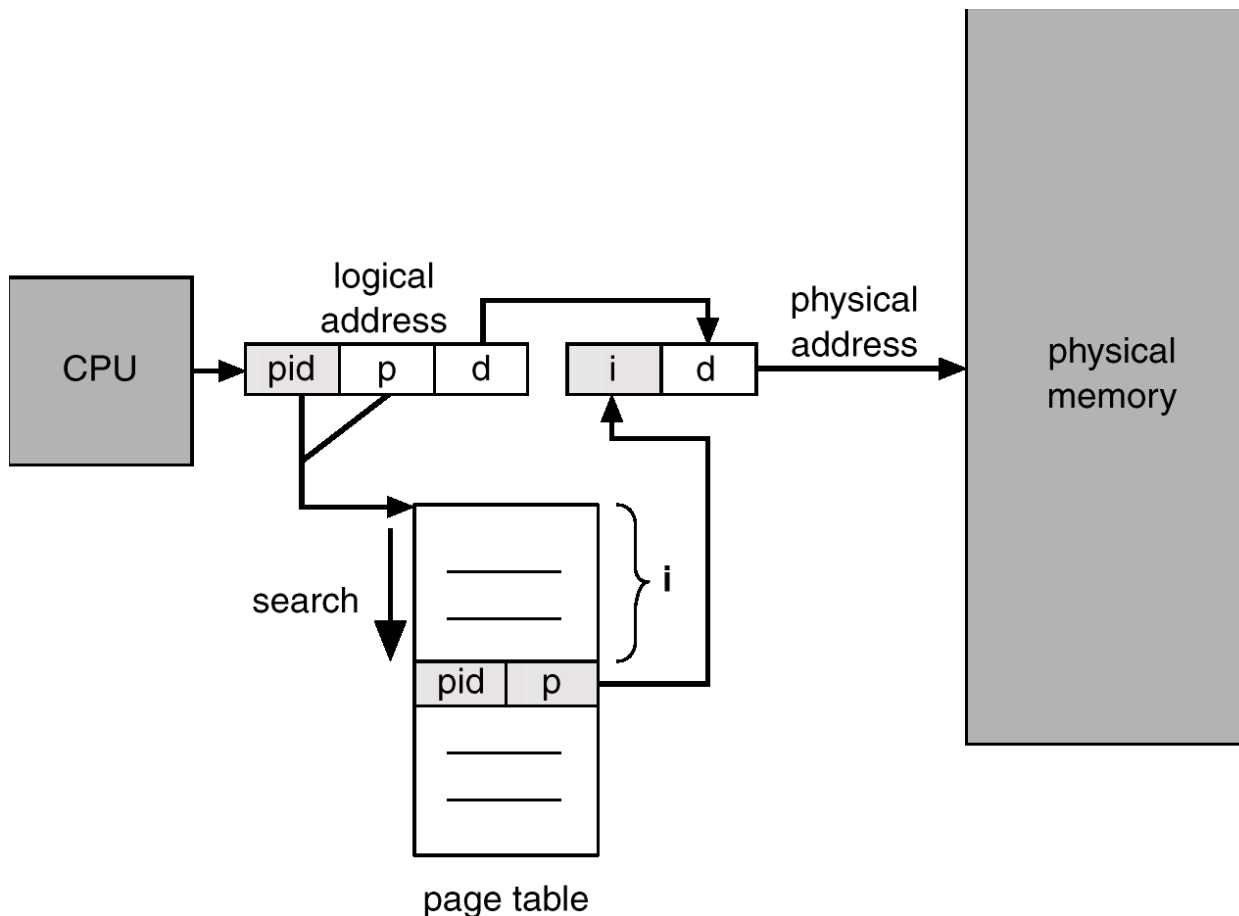


Il *numero totale di entry* puo' essere infinitamente maggiore del numero di frame disponibili

Inverted Page Table : tiene traccia *solo delle pagine fisiche realmente utilizzate*

Perche' "inverted" ?

- Una entry per ogni *pagina fisica* in memoria
- *Dal numero di processo e di pagina logica si puo' risalire alla pagina fisica*



Meno *spazio di memoria*

Maggiore *tempo necessario per ritrovare una pagina*: ricerca sequenziale su tutti i frame della coppia (*processo, pagina logica*)

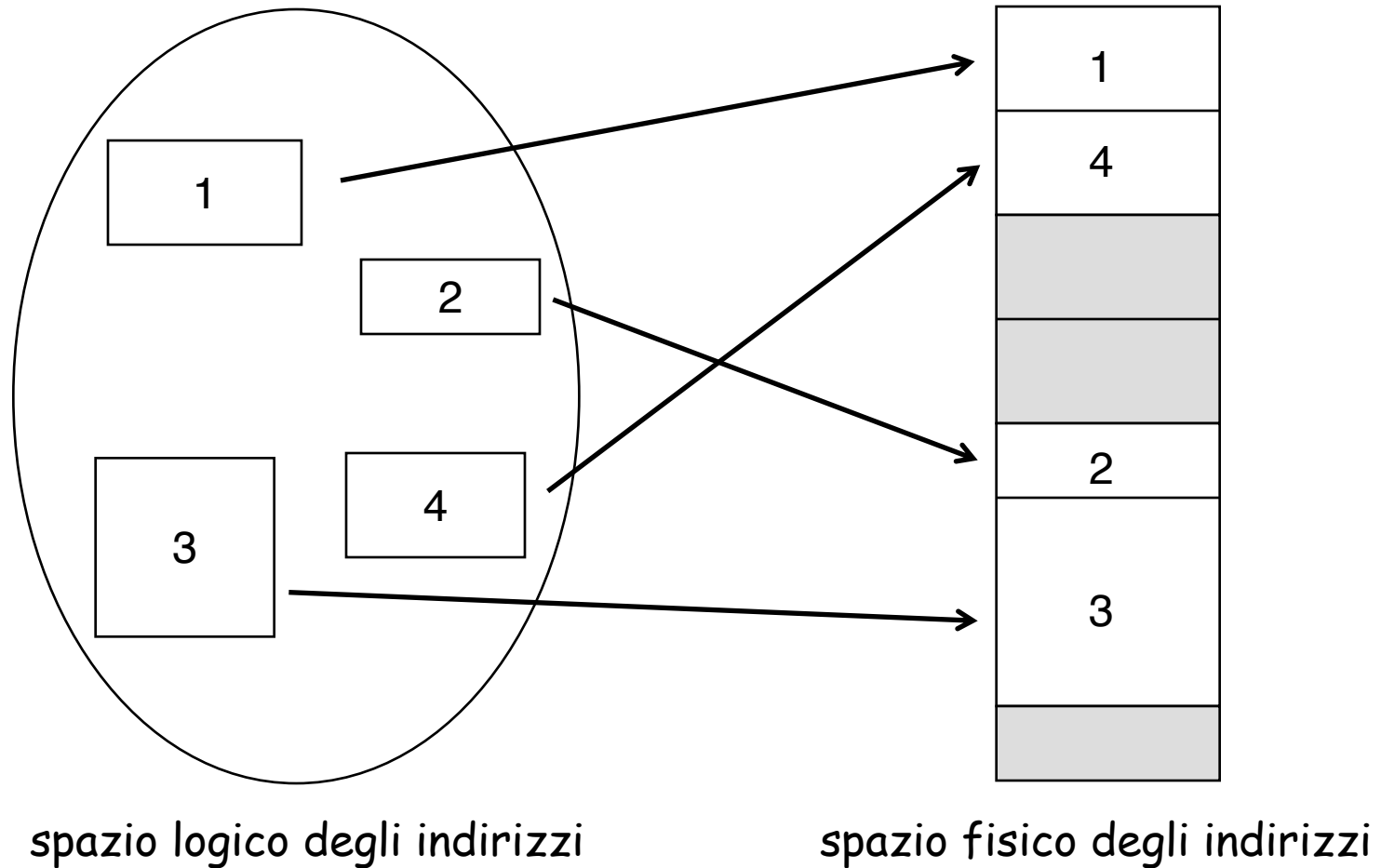
Si puo' usare una *hash table* per limitare la ricerca

2. Segmentazione : idea di base

Suddivisione di un processo in *parti logicamente differenti* (segmenti), *di dimensioni diverse*, per esempio:

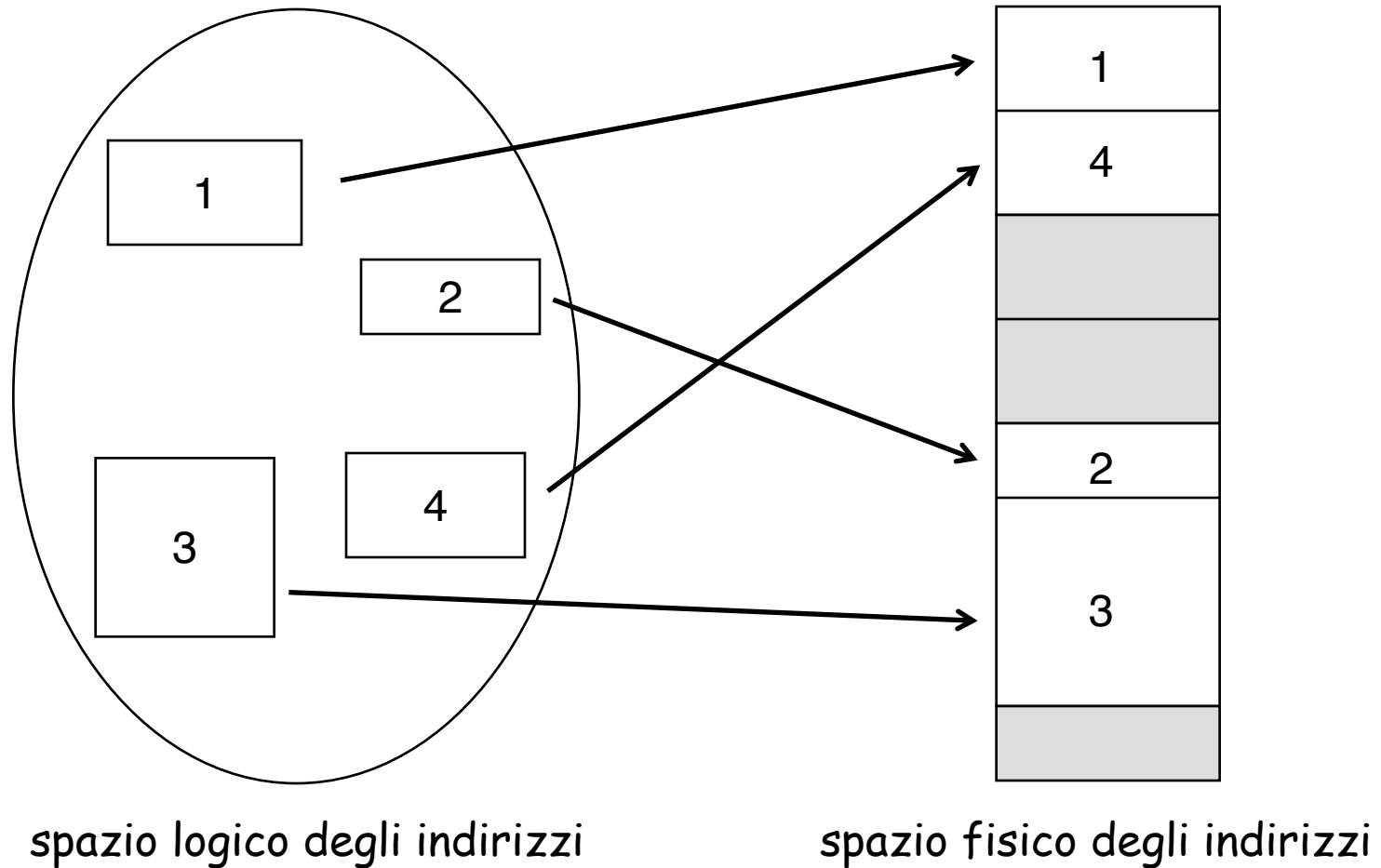
1. il programma principale
2. le procedure
3. le funzioni
4. le variabili globali

... come funziona



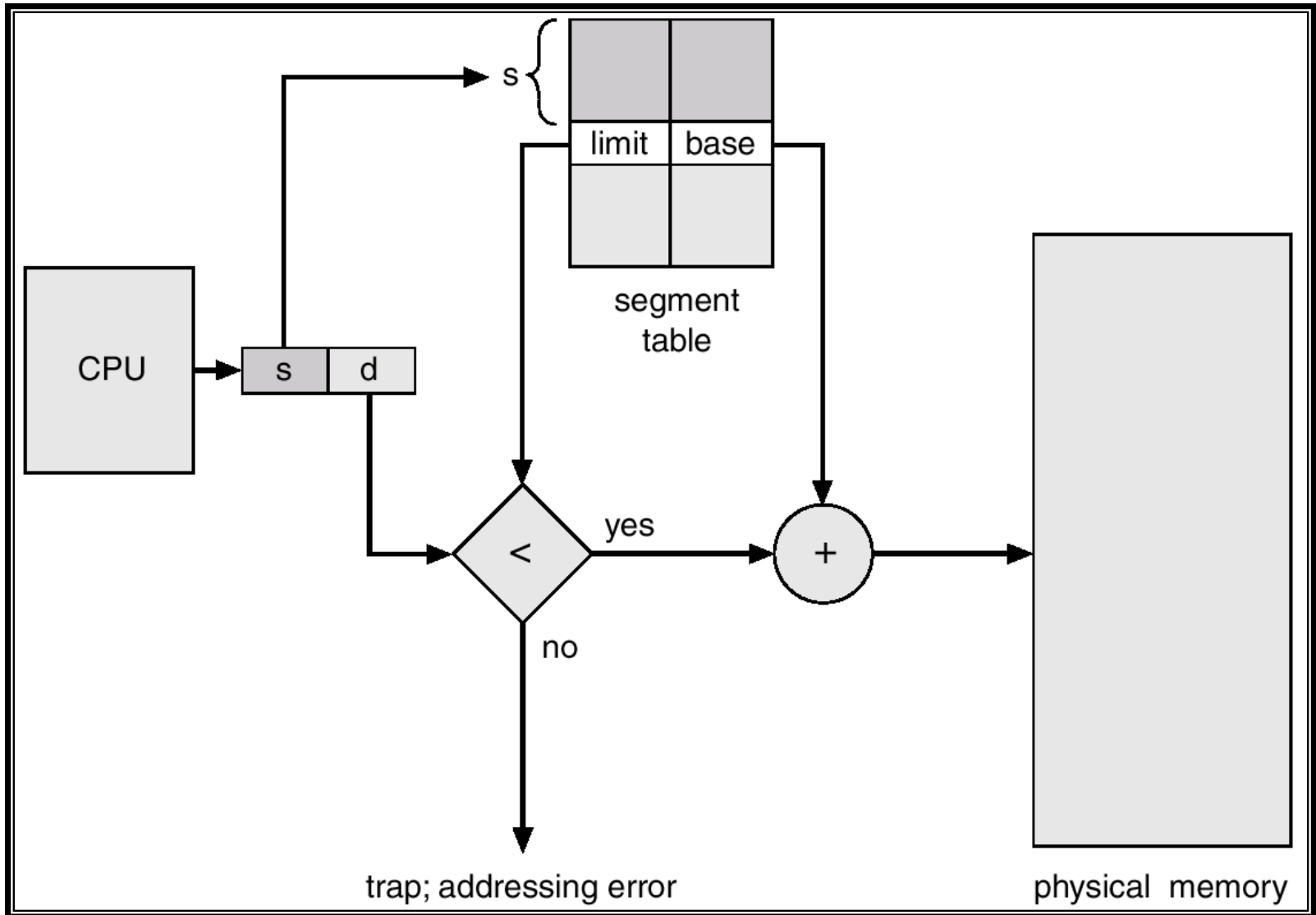
Possibilita' di ***identificazione di un elemento*** che si trova all'interno di un segmento (ex. seconda istruzione di un programma, quinto elemento di un vettore globale, etc.)

... come funziona



- I **segmenti** sono **identificati a priori**
- La **programmazione object-oriented** favorisce l'identificazione di segmenti

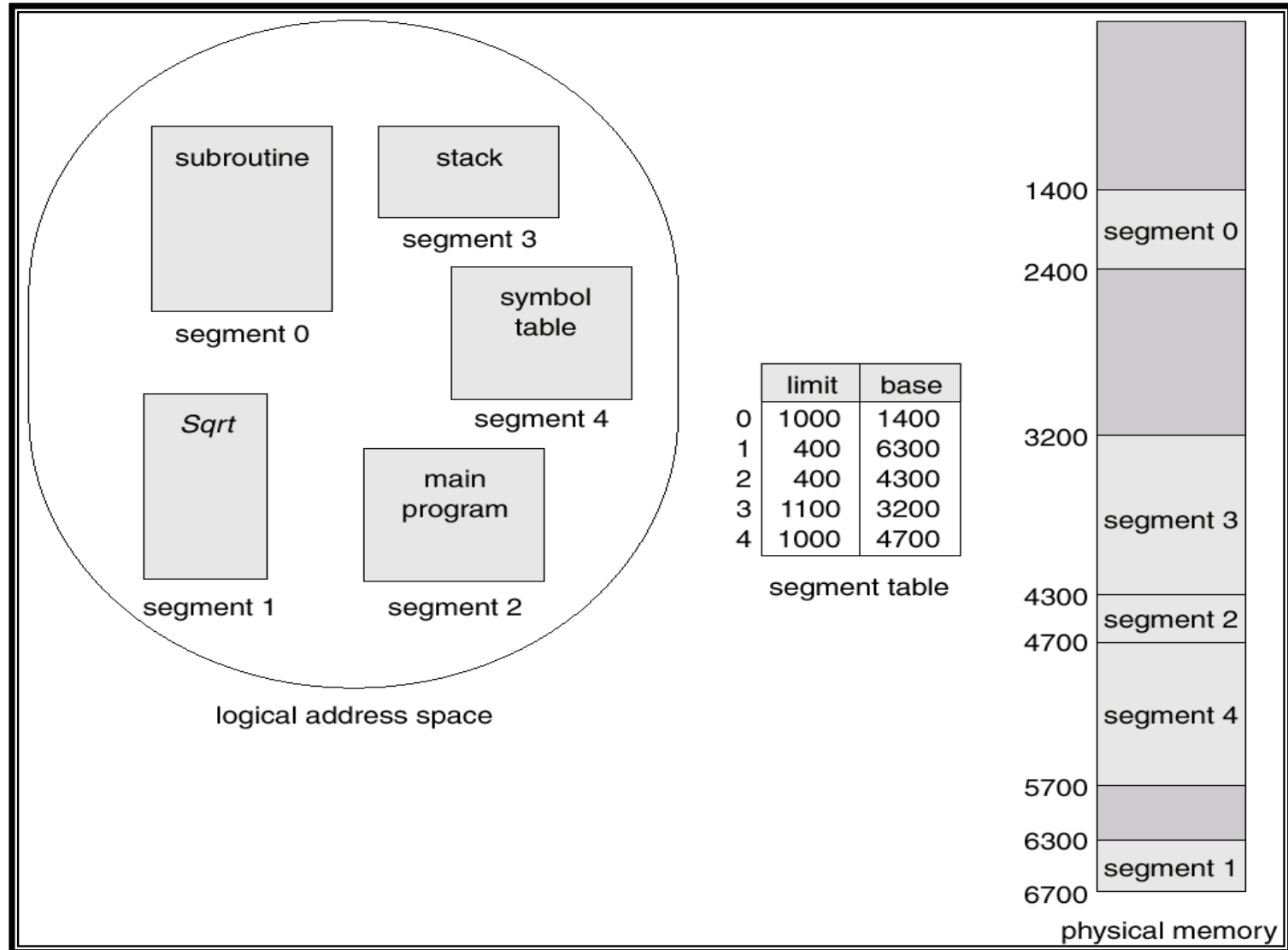
Da indirizzo logico a indirizzo fisico



... e sua implementazione

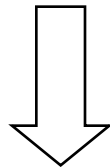
- L'indirizzo logico consiste nella coppia:
⟨segment-number, offset⟩
- **Segment table** - ogni entry corrispondente ad un segment number contiene:
 - **base** : l'indirizzo fisico di inizio del segmento
 - **limit** : la lunghezza del segmento
- Analogamente al caso della paginazione:
 - **Segment-table base register (STBR)**
 - **Segment-table length register (STLR)**
- ... e tutte le considerazioni a proposito del **doppio accesso** a memoria fatte per la paginazione possono essere riproposte

Un esempio di segmentazione



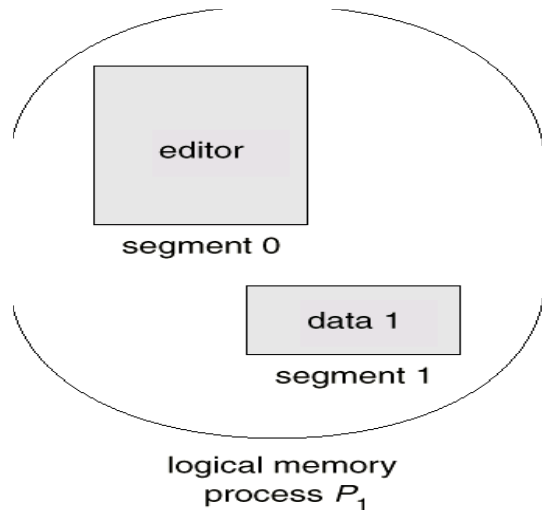
Fondamentale differenza tra paginazione e segmentazione

Poiche' i segmenti possono essere *di differente dimensione*, ritrovare spazio in memoria per essi e' un problema di *allocazione dinamica della memoria*



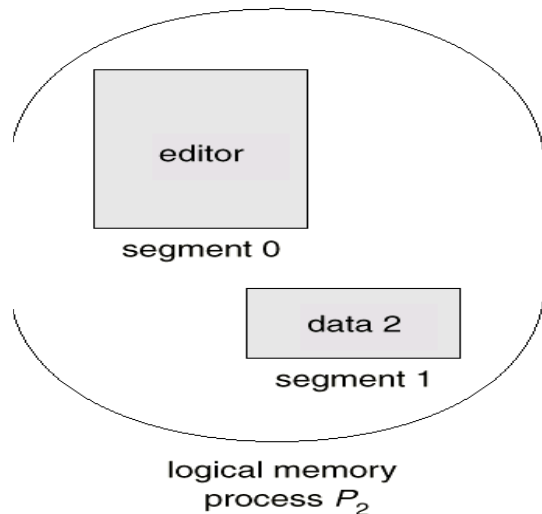
Si ritorna alle *First-fit, Best-fit, Worst-fit*,
e alla *compattazione*

Una maggiore facilità' nella *protezione* di parti di codice logicamente a se' stanti, così' come nella *condivisione*



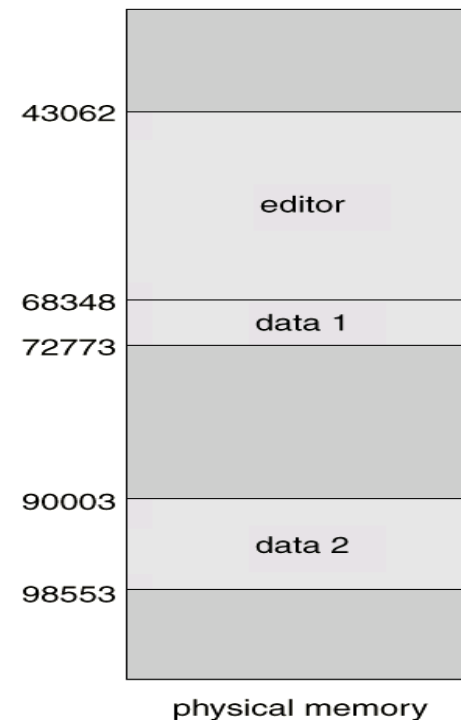
	limit	base
0	25286	43062
1	4425	68348

segment table
process P_1



	limit	base
0	25286	43062
1	8850	90003

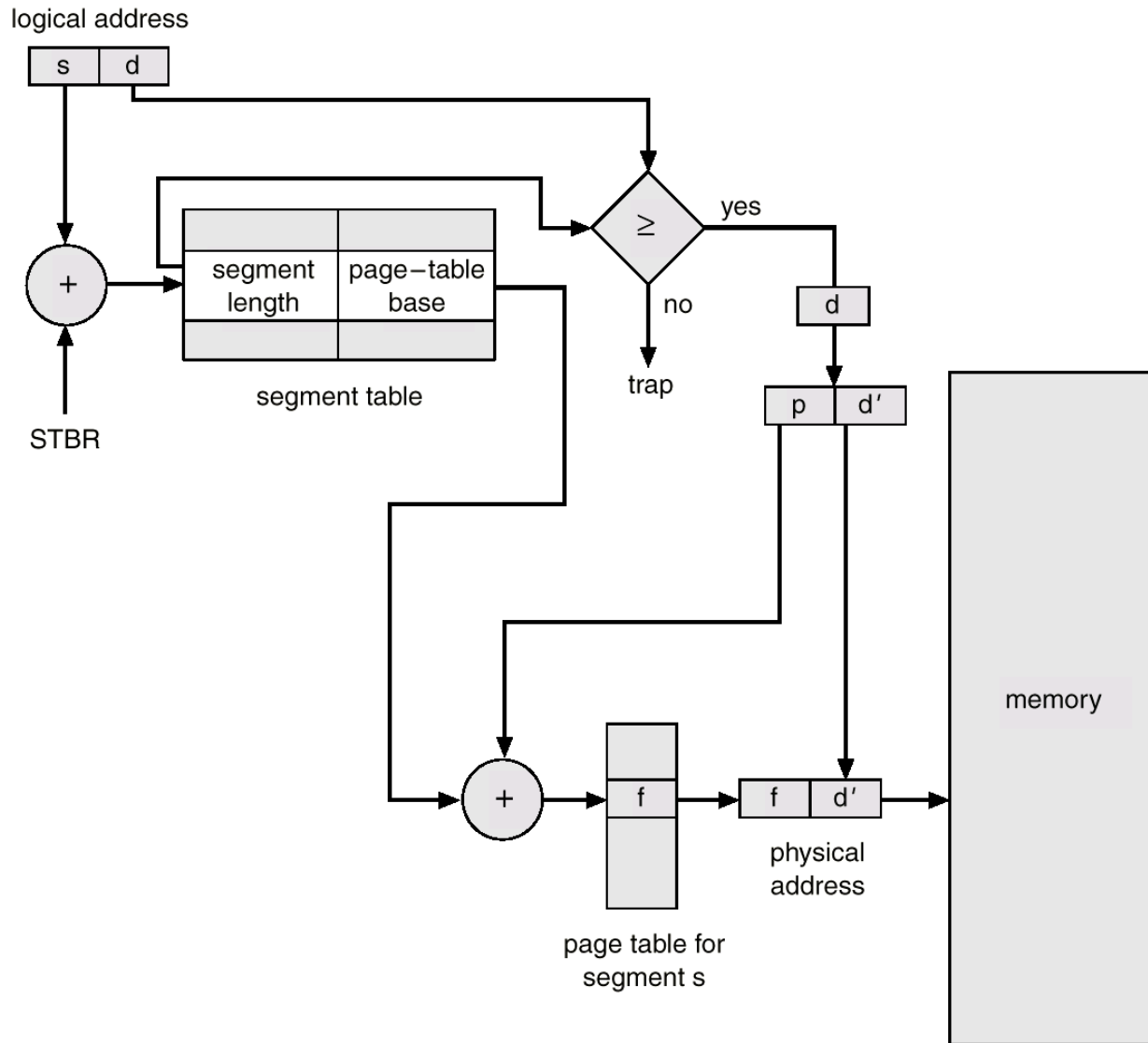
segment table
process P_2



Ed infine una soluzione ibrida: Segmentazione paginata...

- Elimina il problema di frammentazione esterna per *segmenti troppo grandi* : essi vengono *suddivisi in pagine* e quindi allocate in frame equivalenti non necessariamente contigui
- OS MULTICS - *una page table distinta per ogni segmento*
- La soluzione differisce dalla segmentazione pura in quanto *una entry della segment table non contiene l'indirizzo base di un segmento, ma l'indirizzo base della page table di quel segmento*

... ed una sua illustrazione



Si potrebbe perfino pensare di ***paginare a sua volta la segment table***, in quanto troppo grande!

