

# Gestione della memoria

# Gestione della Memoria

- Introduzione
- Spazi di indirizzamento
- Allocazione contigua
- Paginazione
- Segmentazione
- Segmentazione con paginazione

# Introduzione

- La condivisione della memoria da parte di più processi è essenziale per l'efficienza del sistema
- Problematiche:
  - Allocazione della memoria ai singoli job
  - Protezione dello spazio di indirizzamento
  - Condivisione dello spazio di indirizzamento
  - Gestione dello swap
- Nei sistemi moderni:
  - Gestione della memoria inseparabile dal concetto di *memoria virtuale*

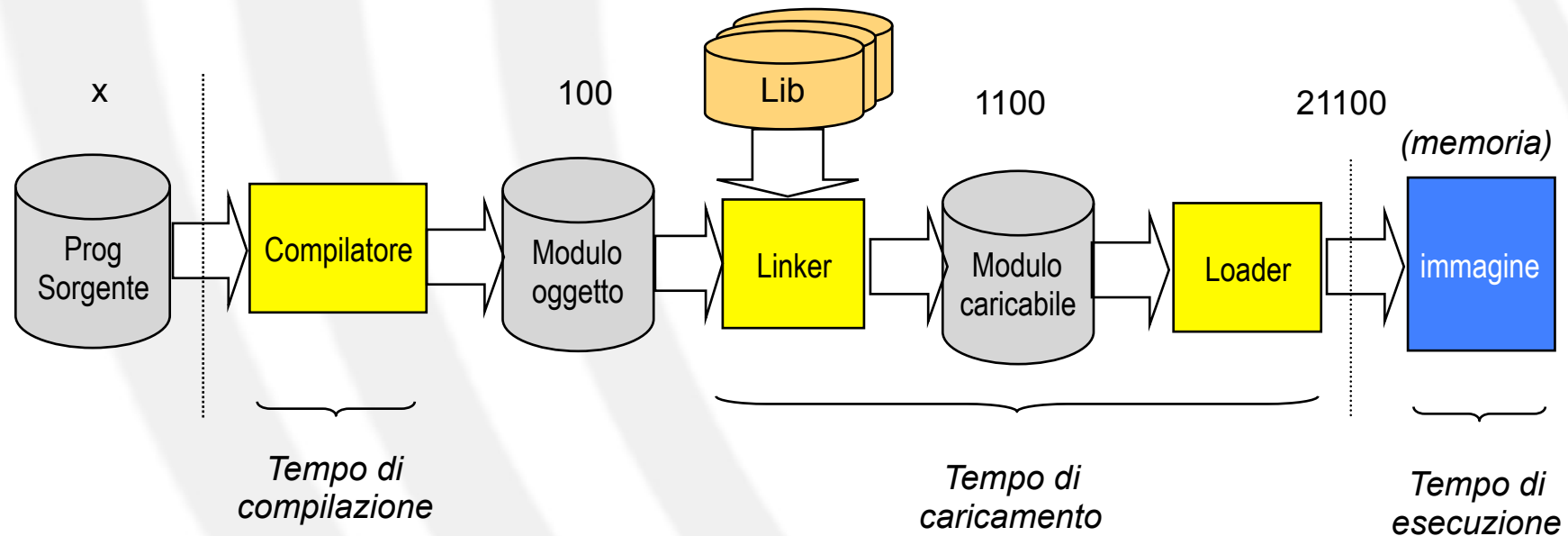
# Introduzione

- Vincolo: ogni programma deve essere portato in memoria e trasformato in processo per essere eseguito
  - La CPU preleva le istruzioni da eseguire dalla memoria in base al valore del program counter
  - L'istruzione viene codificata e può prevedere il prelievo di operandi dalla memoria
  - Al termine dell'esecuzione dell'istruzione, il risultato può essere scritto in memoria
  - Quando il processo termina, la sua memoria viene rilasciata

# Introduzione

- La trasformazione da programma a processo avviene attraverso varie fasi precedenti all'esecuzione
  - In ogni fase si ha una diversa semantica degli indirizzi
    - Spazio logico vs. spazio fisico
  - Gli indirizzi del programma sorgente sono *simbolici*
  - Come diventano *indirizzi fisici*?
    - Il compilatore associa agli indirizzi simbolici indirizzi rilocabili
    - Il linker o il loader associano agli indirizzi rilocabili indirizzi assoluti

# Fasi della trasformazione



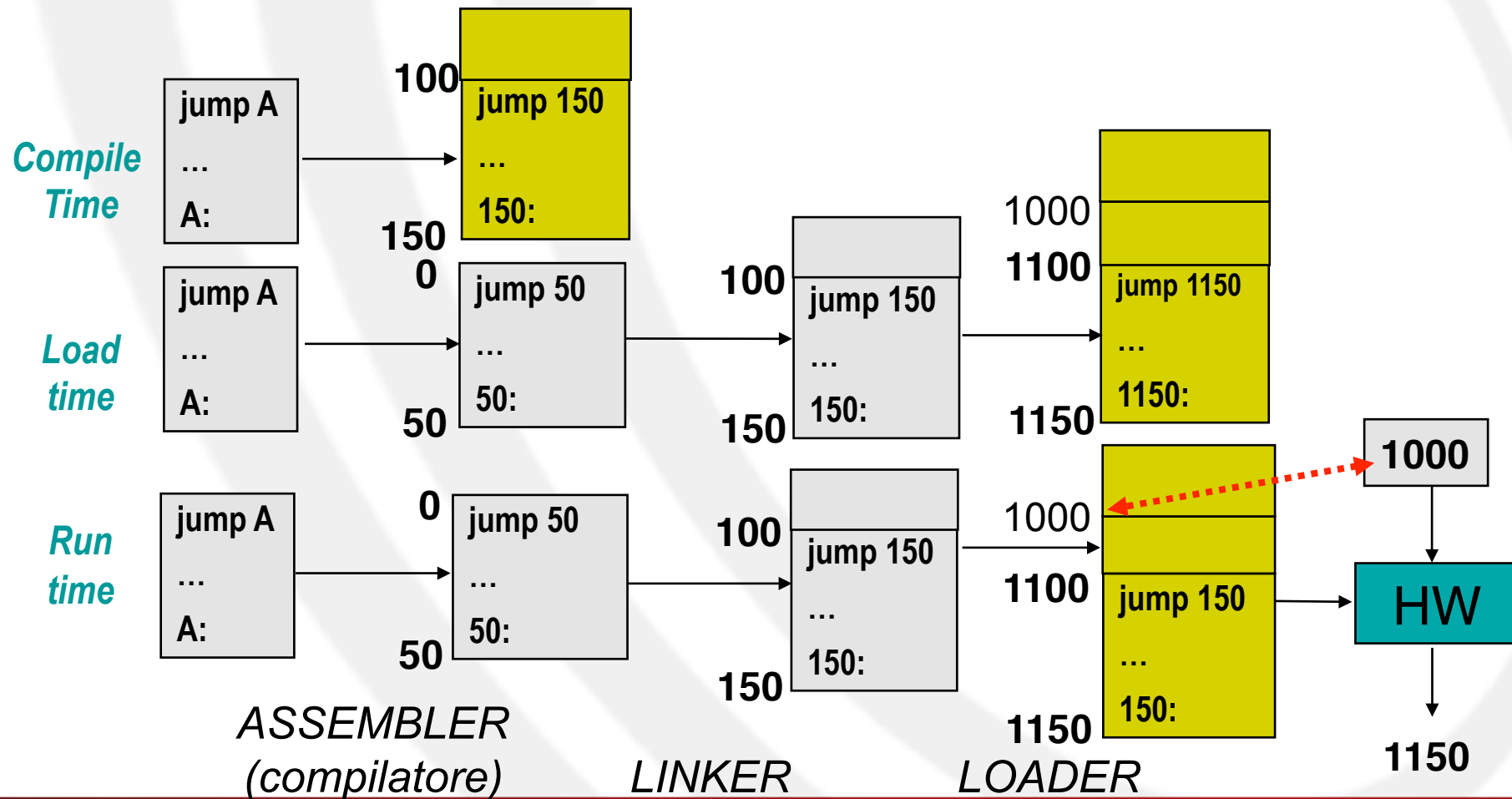
- Gli indirizzi hanno diverse rappresentazioni nelle varie fasi di costruzione di un programma
- Il collegamento tra indirizzi simbolici e indirizzi fisici viene detto *binding*

# Binding & indirizzi

- Il binding di dati e istruzioni a indirizzi di memoria può avvenire in tre momenti distinti

- statico**
  - **Al tempo di compilazione (compile time)**
    - Se è noto a priori in quale parte della memoria risiederà il processo, è possibile generare codice assoluto
    - Se la locazione di partenza cambia, necessario (ri)compilare
  - **Al tempo di caricamento (load time)**
    - Necessario generare codice rilocabile (=riposizionabile)
      - Indirizzi relativi (es. 128 byte dall'inizio del programma)
      - Se cambia l'indirizzo di riferimento, devo (ri)caricare
- dinamico**
  - **Al tempo di esecuzione (run time)**
    - Binding posticipato se il processo può essere spostato durante l'esecuzione in posizioni diverse della memoria
    - Richiesto supporto hardware (per efficienza)

# Binding - esempio





# Collegamento (linking)

- Statico
  - Tradizionale: tutti i riferimenti sono definiti prima dell'esecuzione
  - L'immagine del processo contiene una copia delle librerie usate
- Dinamico
  - Link posticipato al tempo di esecuzione
  - Il codice del programma non contiene il codice delle librerie ma solo un riferimento (stub) per poterle recuperare (es. Windows DLL)

# Caricamento (loading)

- Statico
  - Tradizionale: tutto il codice è caricato in memoria al tempo dell'esecuzione
- Dinamico
  - Caricamento posticipato dei moduli in corrispondenza del primo utilizzo
  - Codice non utilizzato, non caricato
  - Utile per codice con molti casi “speciali”

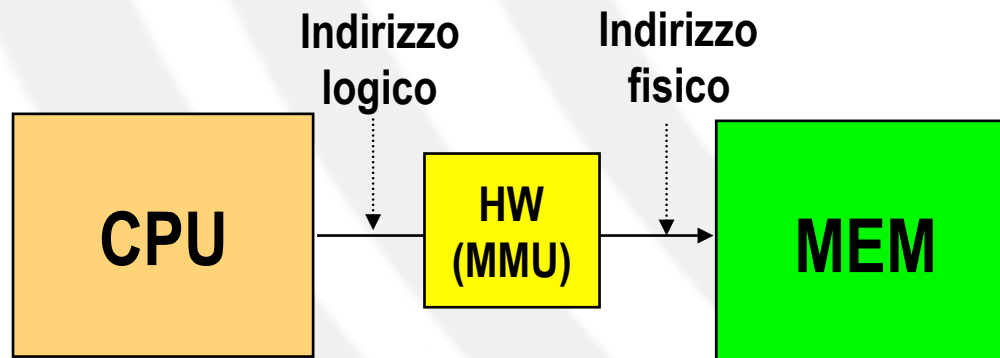
# Spazi di indirizzamento

- Lo spazio di indirizzamento logico è legato a uno spazio di indirizzamento fisico
  - indirizzo logico:
    - generato dalla CPU
    - detto anche indirizzo virtuale
  - indirizzo fisico:
    - visto dalla memoria
- Nel binding a compile o load-time
  - indirizzo fisico e logico coincidono
- Nel binding a run-time
  - indirizzo fisico e logico sono generalmente diversi

# Binding statico vs. dinamico



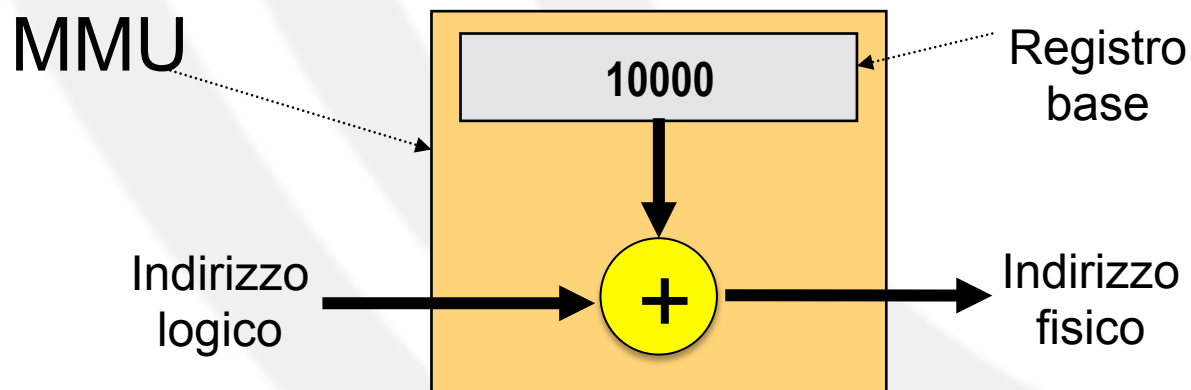
STATICO:  
Compile time  
& load time



DINAMICO:  
Run time  
(tempo di esecuzione)

# Memory Management Unit (MMU)

- Dispositivo hardware che mappa indirizzi virtuali (logici) in indirizzi fisici
- Schema base:
  - Il valore del registro di rilocazione è aggiunto ad ogni indirizzo generato da un processo, e inviato alla memoria



# Considerazioni

- In un sistema *multiprogrammato* non è possibile conoscere in anticipo dove un processo può essere posizionato in memoria
  - Binding a tempo di compilazione non possibile
- L'esigenza di avere lo *swap* impedisce di poter utilizzare indirizzi rilocati in modo statico
  - Binding a tempo di caricamento non possibile
- Conseguenze:
  - Rilocalizzazione dinamica:
    - Usata per sistemi “complessi”
    - Gestione della memoria nel S.O.
  - Rilocalizzazione statica:
    - Possibile in sistemi per applicazioni specifiche
    - Limitata gestione della memoria nel S.O.

# SCHEMI DI GESTIONE DELLA MEMORIA

# Schemi di gestione della memoria

- Allocazione contigua
- Paginazione
- Segmentazione
- Segmentazione paginata

Prevedono che il programma sia interamente caricato in memoria

- NOTA
  - Soluzioni realistiche utilizzano memoria virtuale



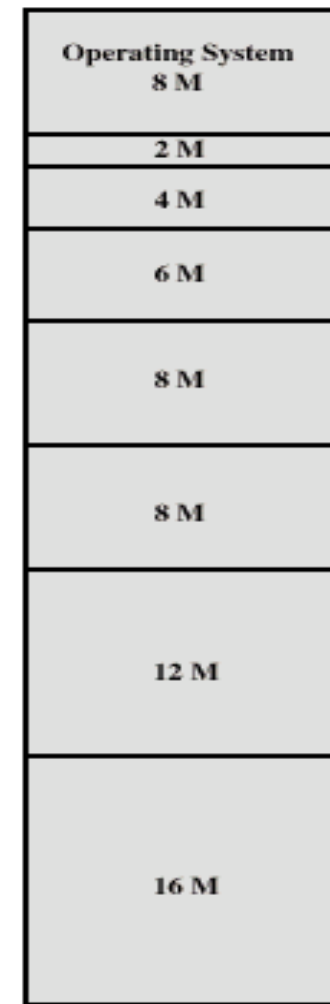
# ALLOCAZIONE CONTIGUA

# Allocazione contigua

- Processi allocati in memoria in posizioni contigue all'interno di una partizione
- La memoria è suddivisa in partizioni:
  - Partizioni fisse
  - Partizioni variabili
- Esempio:
  - Processo con dimensione dell'immagine di 10K
  - Occupa 10K consecutivi

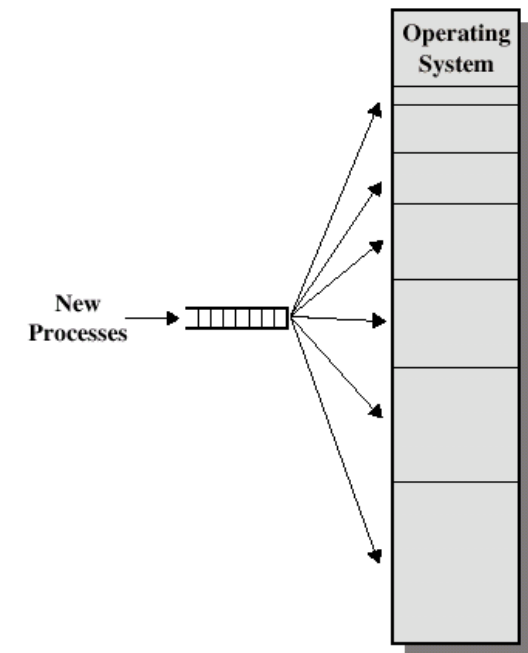
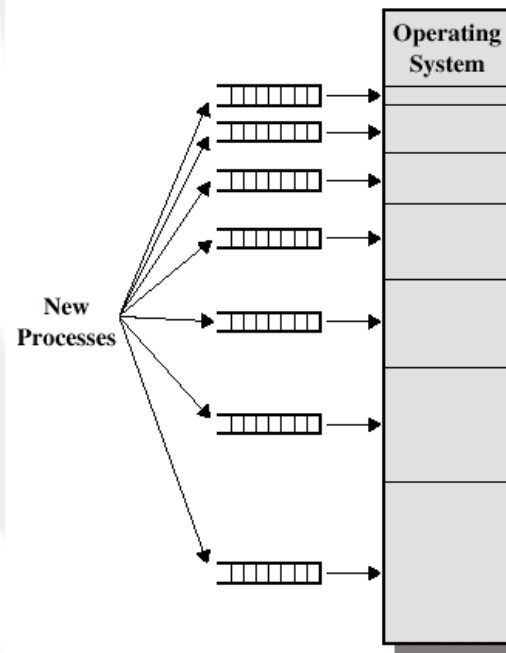
# Tecnica delle partizioni fisse

- La memoria è:
  - insieme di partizioni di dimensioni predefinite (tipicamente diverse)
- Problematiche:
  - Assegnazione di memoria ai job
  - Supporto per la rilocalizzazione dinamica



# Assegnazione della memoria

- Effettuata dallo scheduling a lungo termine
- Due opzioni:
  - Una coda per partizione
  - Coda singola

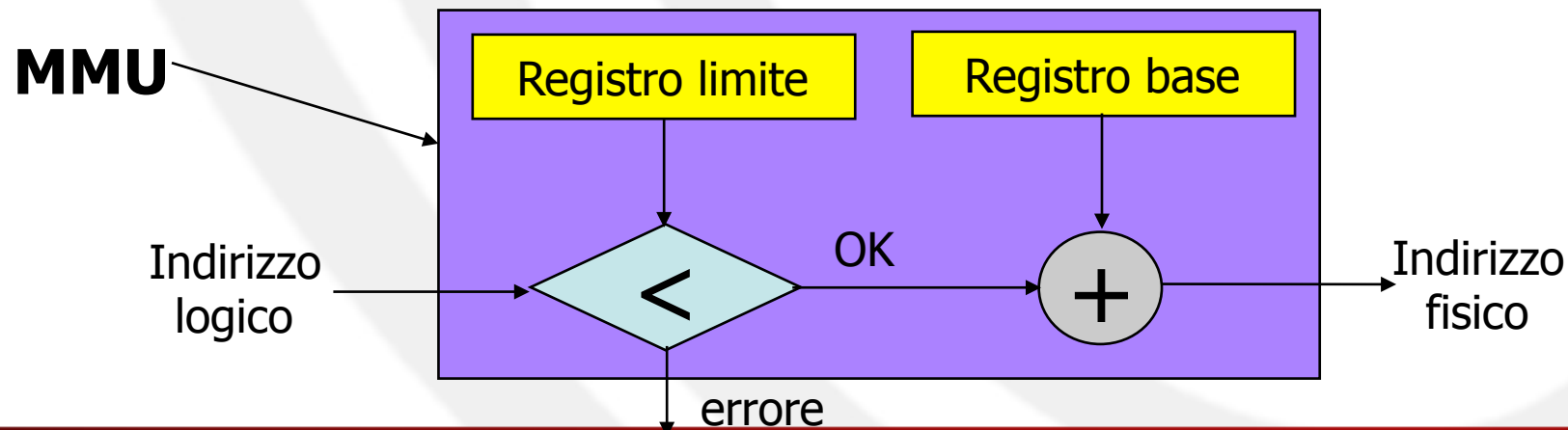


# Assegnazione della memoria

- Una coda per partizione:
  - Il processo viene assegnato alla partizione più piccola in grado di contenerlo
  - Poco flessibile
    - Possono esserci partizioni vuote e job nelle altre code
- Coda unica gestita con politica:
  - FCFS
    - Facile, ma vi è un basso utilizzo della memoria
  - Scansione della coda
    - Best-fit-only
      - Scelta del job con dimensioni più simili alla partizione
    - Best-available-fit
      - Scelta del primo job che può stare nella partizione

# Supporto per la rilocalazione

- MMU consiste di *registri di rilocalazione* per proteggere lo spazio dei vari processi (attivamente e passivamente)
  - Contengono:
    - Valore dell'indirizzo più basso (registro base o di rilocalazione)
    - Limite superiore dello spazio logico (registro limite)
  - Ogni indirizzo logico deve risultare  $<$  del limite



# Considerazioni

- Vantaggi
  - Relativa semplicità
- Svantaggi
  - Grado di multiprogrammazione limitato dal n° di partizioni
  - Frammentazione → spreco di memoria
    - Frammentazione interna
      - Interna alla partizione
      - Se la dimensione della partizione è più grande della dimensione del job
    - Frammentazione esterna
      - Se vi sono partizioni non utilizzate che non soddisfano le esigenze dei processi in attesa

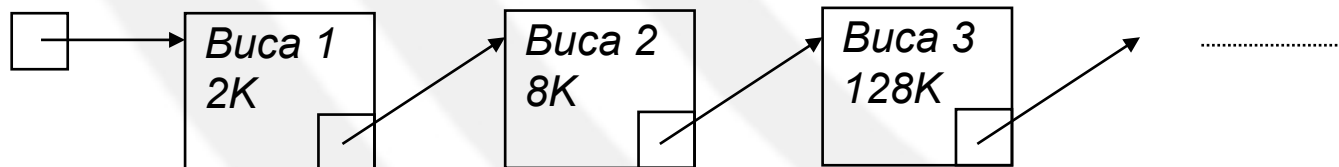
# Tecnica delle partizioni variabili

- Spazio utente diviso in partizioni di dimensioni variabili
  - di dimensioni identiche alla dimensione dei processi
- Motivazione:
  - Eliminare la frammentazione interna
- Problematiche:
  - Assegnazione di memoria ai job
  - Supporto per la rilocalizzazione dinamica



# Assegnazione della memoria

- Vista della memoria: insieme di buche (*hole*)
  - Buca = blocco di memoria disponibile
- Il S.O. mantiene informazioni su:
  - partizioni allocate
  - buche
- Quando arriva un processo, gli viene allocata memoria usando la buca che lo può contenere



# Assegnazione memoria - esempio

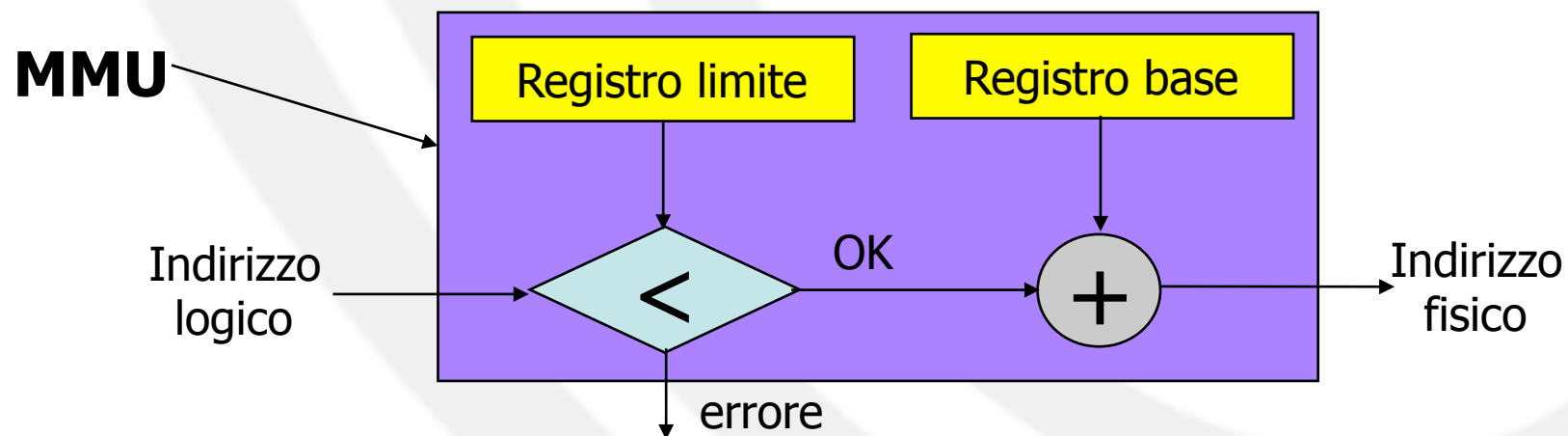
OS	OS	OS	OS	OS
Processo 5	Processo 5	Processo 5	Processo 5	Processo 5
Processo 8		Processo 9	Processo 9	
Processo 2	Processo 2		Processo 10	Processo 10
		Processo 2	Processo 2	Processo 2

# Assegnazione della memoria

- Come soddisfare la richiesta di  $n$  celle di memoria data una lista di buche libere?
- Strategie:
  - *First-fit*: alloca la prima buca grande a sufficienza
  - *Best-fit*: alloca la più piccola buca grande a sufficienza
    - richiede la scansione della lista
    - minimo spreco
  - *Worst-fit*: alloca la buca più grande
    - richiede la scansione della lista
    - lascia la buca di dimensioni più grandi
- *First fit* è tipicamente la migliore

# Supporto per la rilocalizzazione

- Come per partizioni fisse
  - Registri di rilocalizzazione per proteggere lo spazio dei vari processi (attivamente e passivamente)



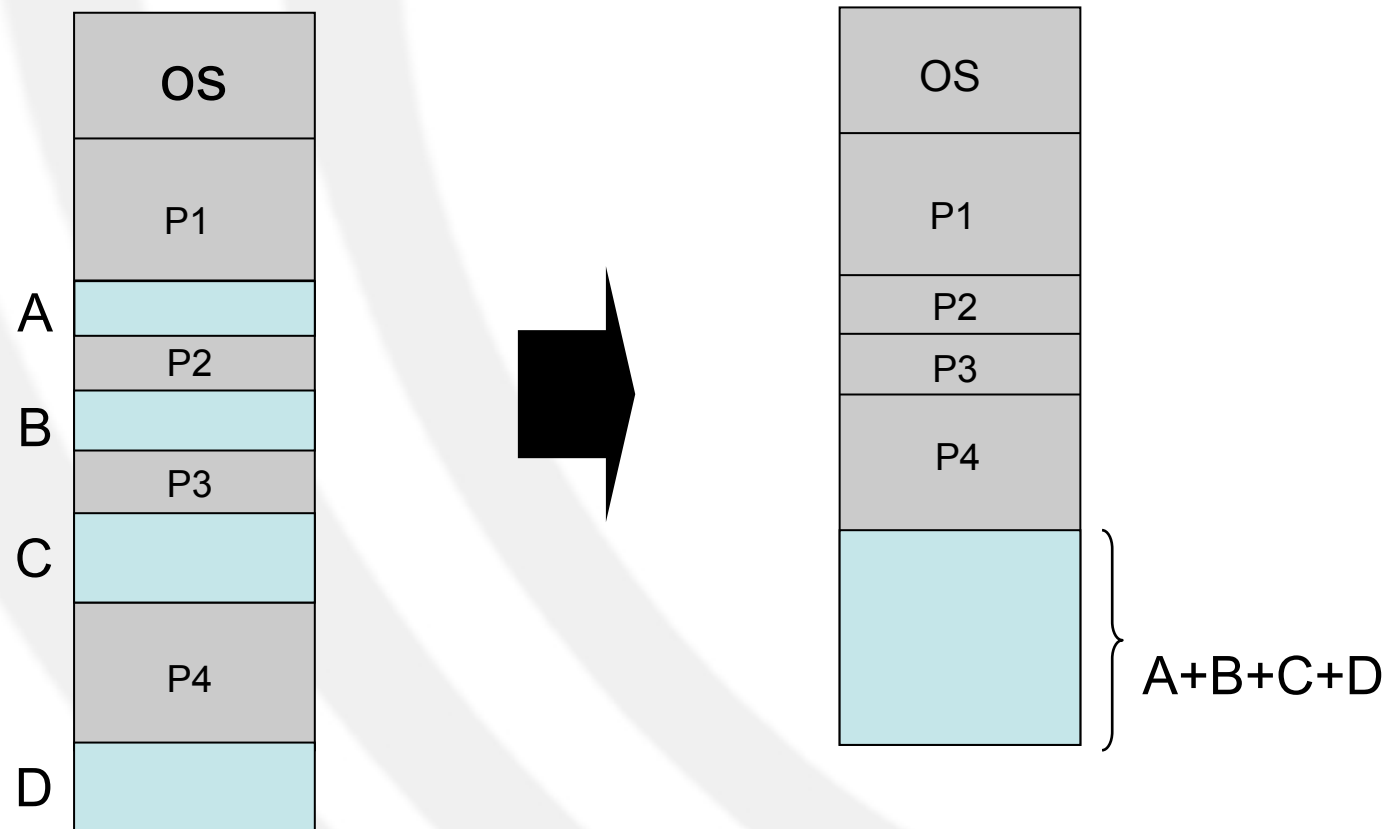
# Considerazioni

- Vantaggi
  - No frammentazione interna per costruzione
- Svantaggi
  - Frammentazione esterna: esiste lo spazio disponibile in memoria, ma non è contiguo
    - Con first fit, dati  $N$  blocchi allocati,  $0.5*N$  blocchi vanno persi

# Riduzione della frammentazione

- Soluzione intuitiva: compattazione
  - Contenuto della memoria spostato in modo da rendere contigui i blocchi di un processo
  - Possibile solo se la rilocazione è dinamica
    - Richiede modifica del registro base
  - Costosa: quanta memoria spostato?

# Compattazione - esempio



# Tecnica del *buddy system*

- Compromesso tra partizioni fisse e variabili
  - La memoria è vista come una serie di liste di blocchi di dimensione  $2^k$ , con  $L < k < U$ 
    - $2^L$  = più piccolo blocco allocato (es. 4K)
    - $2^U$  = più grande blocco allocato (es. tutta la memoria)
  - La memoria è disponibile sotto forma di blocchi di dimensione  $2^k$
  - All'inizio tutta la memoria è disponibile
    - La lista di blocchi di dimensione  $2^U$  contiene un solo blocco che rappresenta tutta la memoria
    - Le altre liste sono vuote



# Tecnica del *buddy system*

- Quando arriva una richiesta di dimensione  $s$ , si cerca un blocco libero con dimensione “adatta” purché sia pari a una potenza del 2
  - Se  $2^{U-1} < s \leq 2^U$  l'intero blocco di dimensione  $2^U$  viene allocato
  - Altrimenti il blocco  $2^U$  è diviso in due blocchi di dimensione  $2^{U-1}$
  - Se  $2^{U-2} < s \leq 2^{U-1}$  l'intero blocco di dimensione  $2^{U-1}$  viene allocato
  - Altrimenti il blocco  $2^{U-1}$  è diviso in due blocchi di dimensione  $2^{U-2}$
  - ... .. e così via fino ad arrivare (al limite) al blocco di dimensione  $2^L$

## Tecnica del *buddy system*

- Quando un processo rilascia la memoria, il suo blocco torna a far parte della lista dei blocchi di dimensione corrispondente
  - Se si formano 2 blocchi adiacenti di dimensione  $2^K$ , è possibile compattarli ottenendo un unico blocco libero di dimensione  $2^{K+1}$
- Vantaggio
  - La compattazione richiede solo di scorrere la lista dei blocchi di dimensione  $2^K$ , quindi è veloce
- Svantaggio
  - Frammentazione interna dovuta solo ai blocchi di dimensione  $2^L$

# Buddy system - esempio

1 Mbyte block	1 M				
Request 100 K	A = 128 K	128 K	256 K	512 K	
Request 240 K	A = 128 K	128 K	B = 256 K	512 K	
Request 64 K	A = 128 K	C = 64 K	64 K	B = 256 K	512 K
Request 256 K	A = 128 K	C = 64 K	64 K	B = 256 K	D = 256 K
Release B	A = 128 K	C = 64 K	64 K	256 K	D = 256 K
Release A	128 K	C = 64 K	64 K	256 K	D = 256 K
Request 75 K	E = 128 K	C = 64 K	64 K	256 K	D = 256 K
Release C	E = 128 K	128 K	256 K	D = 256 K	256 K
Release E	512 K			D = 256 K	256 K
Release D	1 M				



# PAGINAZIONE

# Paginazione

- Tecnica per eliminare la frammentazione esterna
- Idea:
  - Permettere che lo spazio di indirizzamento fisico di un processo sia non-contiguo
  - Si alloca memoria fisica dove essa è disponibile
- Memoria fisica:
  - divisa in blocchi di dimensione fissa detti *frame*
    - Tipico 512 byte ... 8K byte
- Memoria logica:
  - divisa in blocchi della stessa dimensione detti *pagine*

# Paginazione

- Per eseguire un programma avente dimensione  $n$  pagine, bisogna trovare  $n$  frame liberi
- Si utilizza una tabella delle pagine (*page table*) per mantenere traccia di quale frame corrisponde a quale pagina
  - Una tabella delle pagine per ogni processo
  - viene usata per tradurre un indirizzo logico in un indirizzo fisico

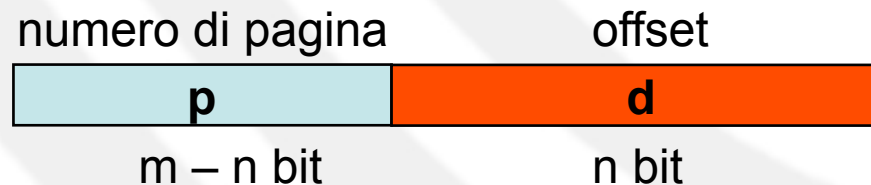
## Paginazione - esempio

- Dimensione della pagina = 1KB
- Dimensione del programma = 2.3KB
- Necessarie 3 pagine
  - dell'ultima pagina si userà solo 0.3KB
- E' ancora possibile della frammentazione interna, ma solo nell'ultima pagina

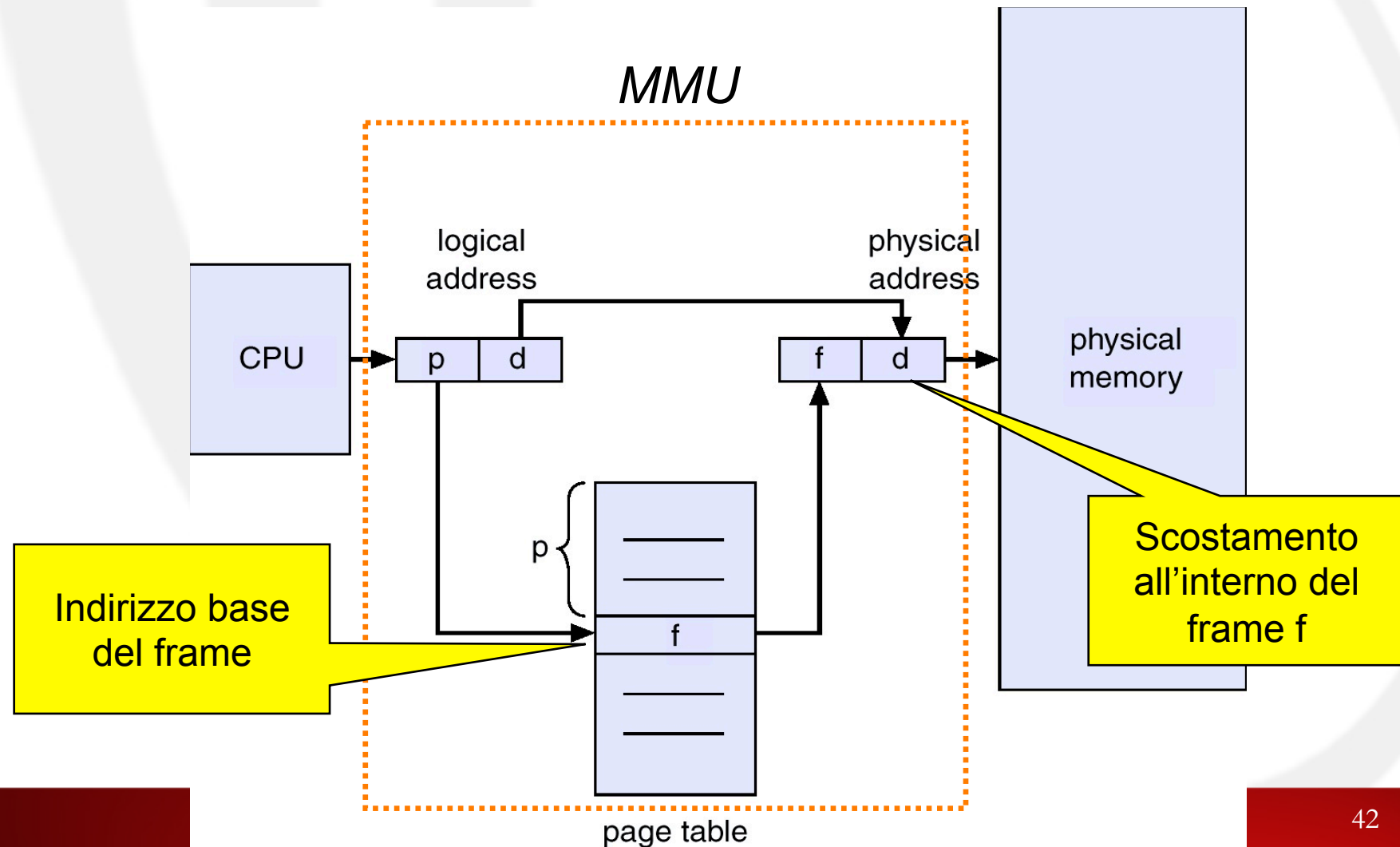


# Traduzione degli indirizzi

- L'indirizzo generato dalla CPU viene diviso in due parti:
  - numero di pagina ( $p$ )
    - Usato come indice nella tabella delle pagine che contiene l'indirizzo di base di ogni frame
  - offset ( $d$ )
    - Combinato con l'indirizzo base definisce l'indirizzo fisico che viene inviato alla memoria
- Se la dimensione della memoria è  $2^m$  e quella di una pagina è  $2^n$  (parole/byte), l'indirizzo è suddiviso così:

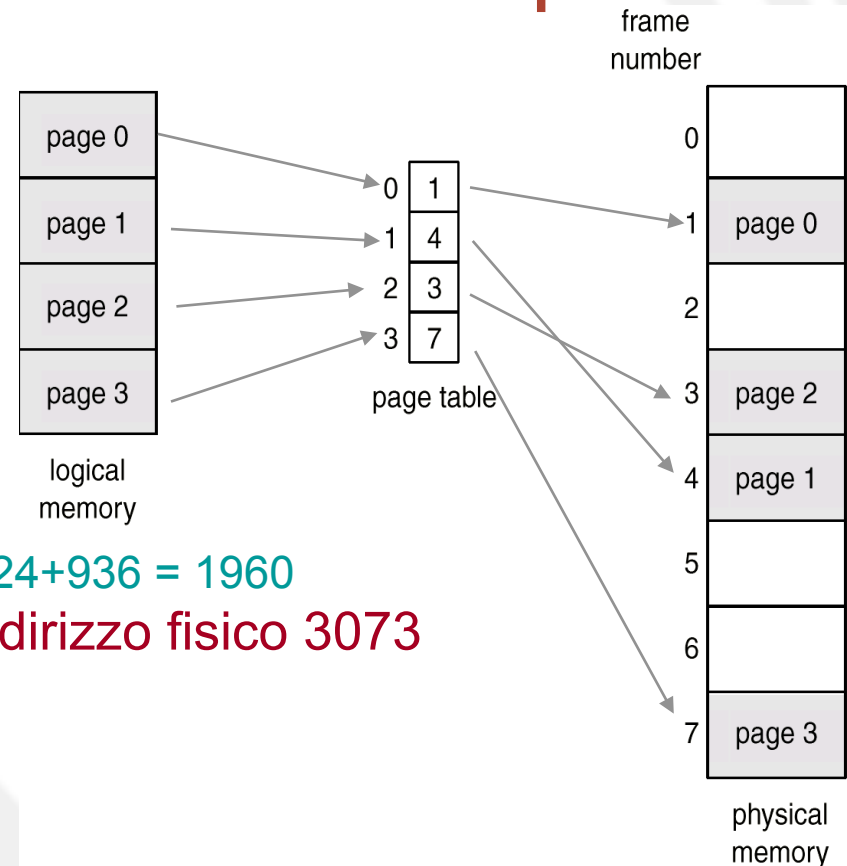


# Traduzione degli indirizzi - architettura



# Traduzione degli indirizzi – esempio

- Memoria da 8KB
- Pagine di 1KB
  - L'indirizzo logico 936 diventa l'indirizzo fisico 1960
    - pagina 0 (0...1K)
      - (inizio pagina = 0)
      - offset = 936
    - pagina 0 → frame 1 (1024...2047)
    - indirizzo fisico = 1024 + offset = 1024+936 = 1960
  - L'indirizzo logico 2049 diventa l'indirizzo fisico 3073
    - pagina 2 (2048...3071)
      - inizio pagina = 2048
      - offset = 1
    - pagina 2 → frame 3 (3072...4095)
    - indirizzo fisico = 3072 + 1 = 3073



# Implementazione della tabella delle pagine

- L'efficienza è fondamentale
- Soluzioni:
  - Implementazione tramite registri
  - Implementazione in memoria
    - Tabella delle pagine multilivello
    - Tabella delle pagine invertita

# Implementazione tramite registri

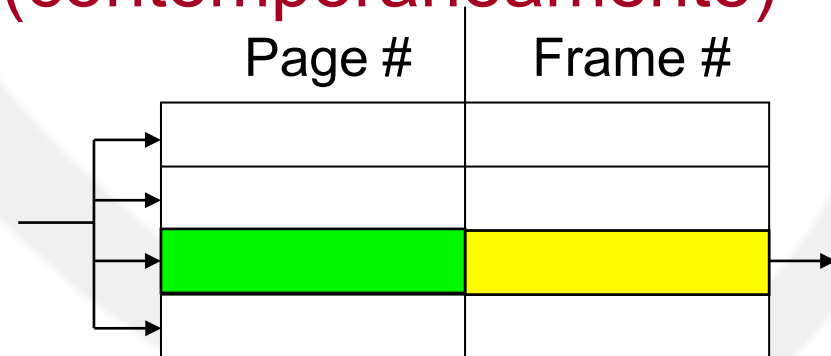
- Le entry (righe) della tabella delle pagine sono mantenute nei registri
  - Soluzione efficiente ma...
  - ... fattibile se il numero di entry è limitato (perché ci sono pochi registri)
  - ... allunga i tempi di contex switch perché richiede salvataggio dei registri

# Implementazione in memoria

- La tabella risiede in memoria
- Vengono utilizzati due registri:
  - Page-table base register (PTBR)
    - punta alla tabella delle pagine
  - Page-table length register (PTLR) [opzionale]
    - contiene la dimensione della tabella delle pagine
- Il context switch è più breve perché richiede solo modifica del PTBR (PTLR)
- Problema:
  - ogni accesso a dati/istruzioni richiede due accessi in memoria (tabella delle pagine + dato/istruzione)

# Implementazione in memoria

- Il problema del doppio accesso può essere risolto tramite una cache veloce detta *translation look-aside buffers* (TLB)
- Funzionamento:
  - Confronto dell'elemento fornito con il campo chiave di tutte le entry (contemporaneamente)
  - Tabella delle pagine
    - Chiave = n° di pagina
    - Valore = n° di frame



# Implementazione in memoria

- Il TLB è molto costoso
  - quindi nel TLB viene memorizzato solo un piccolo sottoinsieme delle entry della tabella delle pagine
  - ad ogni context switch il TLB viene ripulito per evitare mapping di indirizzi errati
- Durante un accesso alla memoria:
  - se la pagina cercata è nel TLB, il TLB restituisce il numero di frame con un singolo accesso
    - tempo richiesto < 10% tempo richiesto in assenza di TLB
  - altrimenti è necessario accedere alla tabella delle pagine in memoria
  - hit ratio  $\alpha$  = % delle volte in cui una pagina si trova nel TLB
- Necessario definire il concetto di *tempo di accesso effettivo*



# Tempo di accesso effettivo

- Tempo di accesso effettivo (EAT)
  - $EAT = (T_{MEM} + T_{TLB}) \alpha + (2 T_{MEM} + T_{TLB}) (1 - \alpha)$
  - dove:
    - $T_{TLB}$  = tempo di accesso a TLB
    - $T_{MEM}$  = tempo di accesso a memoria
- Esempio:
  - $T_{MEM} = 90ns$
  - $T_{TLB} = 10ns$
  - $\alpha = 90\%$
  - $EAT = 100 * 0.9 + 190 * 0.1 = 109 \sim 1.2 T_{MEM}$

# Protezione

- Realizzata associando bit di protezione ad ogni frame
- Esempio:

- Bit di validità (*valid-invalid bit*) per ogni entry della tabella delle pagine

Utile per  
memoria  
virtuale

- “valid”: la pagina associata è nello spazio di indirizzamento logico del processo
    - “invalid”: la pagina associata NON è nello spazio di indirizzamento logico del processo

- Bit di accesso

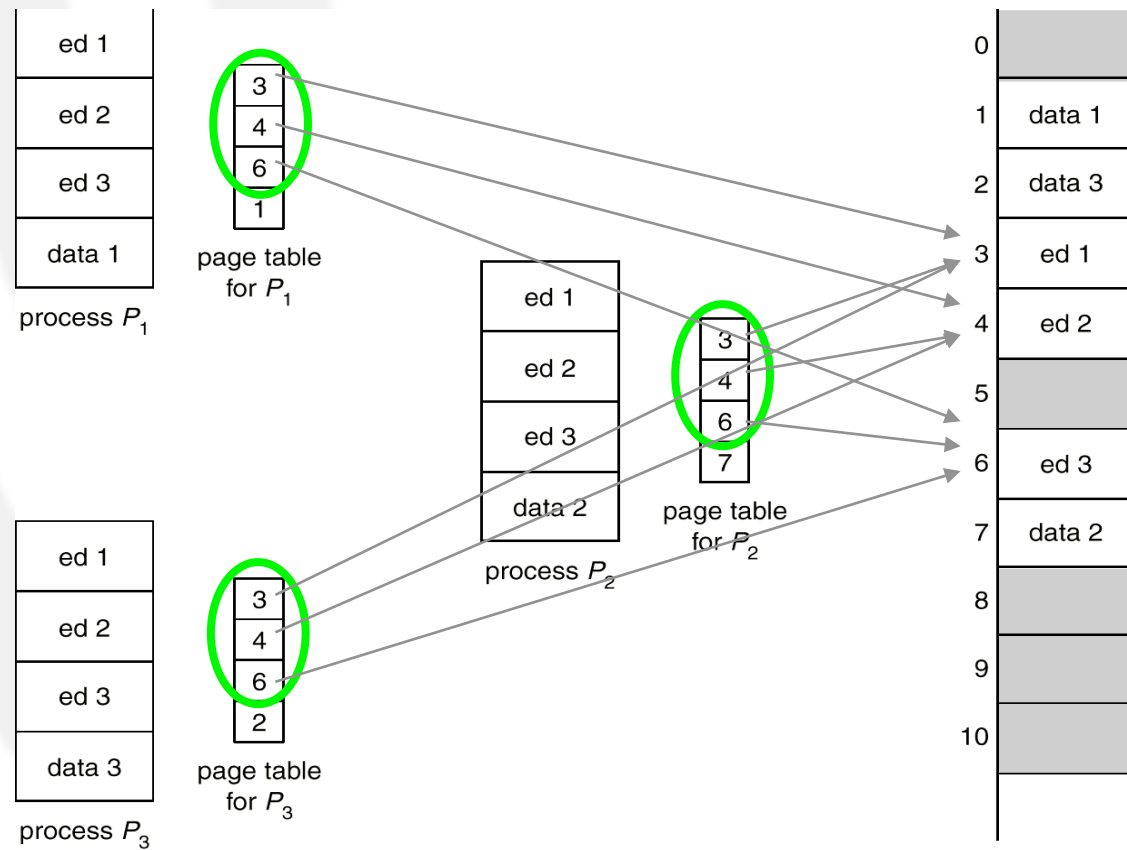
- Per marcare una pagina modificabile o meno (read-only)
    - Per marcare una pagina eseguibile o meno
    - ...

# Pagine condivise

- Codice condiviso
  - Vi è un'unica copia fisica, ma più copie logiche (una per processo)
  - Il codice *read-only* (rientrante) può essere condiviso tra processi
    - es.: text editor, compilatori, window manager
- I dati in generale saranno diversi da processo a processo (più copie fisiche e logiche)

Non cambia mai durante l'esecuzione

# Codice condiviso - esempio



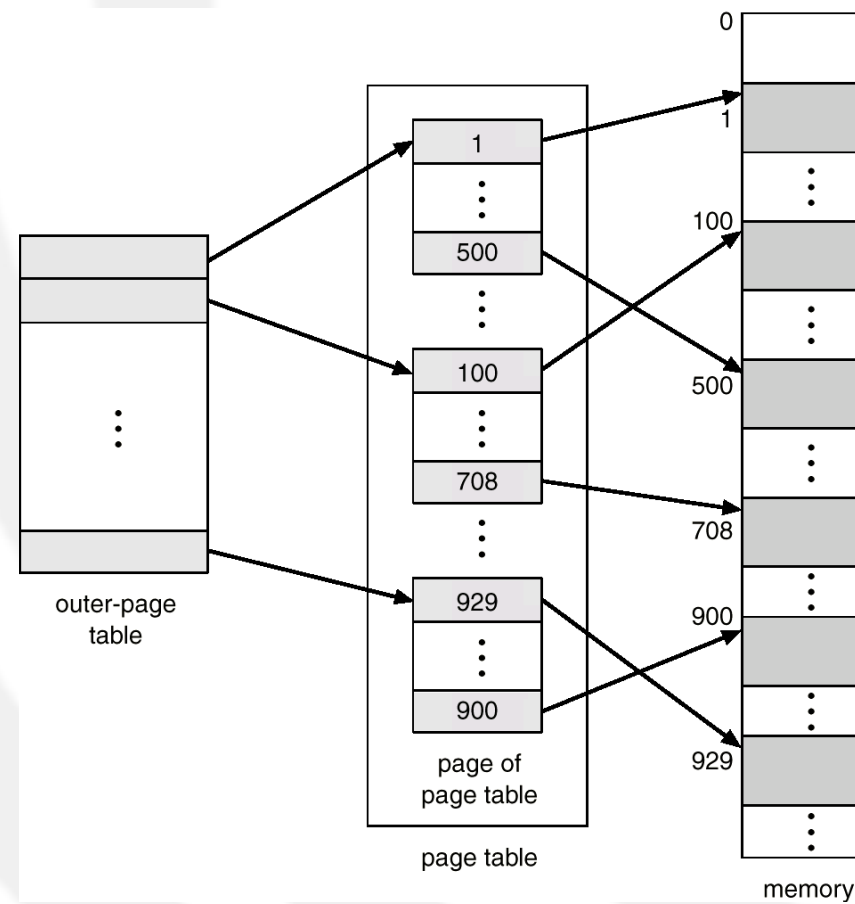
# Spazio di indirizzamento

- Nelle architetture moderne con 32 (o 64) bit di indirizzo, lo spazio di indirizzamento virtuale è  $2^{32}$  (o  $2^{64}$ ), ovvero molto maggiore dello spazio fisico!
  - Es.: se  $|frame| = 4K = 2^{12} \Rightarrow 2^{32}/2^{12} = 2^{20} \cong 1.000.000$  entry nella tabella delle pagine (per processo!)
- Sono necessari meccanismi per gestire il problema della dimensione della tabella delle pagine:
  - Paginazione della tabella delle pagine
    - Tabella delle pagine multilivello
  - Tabella delle pagine invertita

## Paginazione multilivello

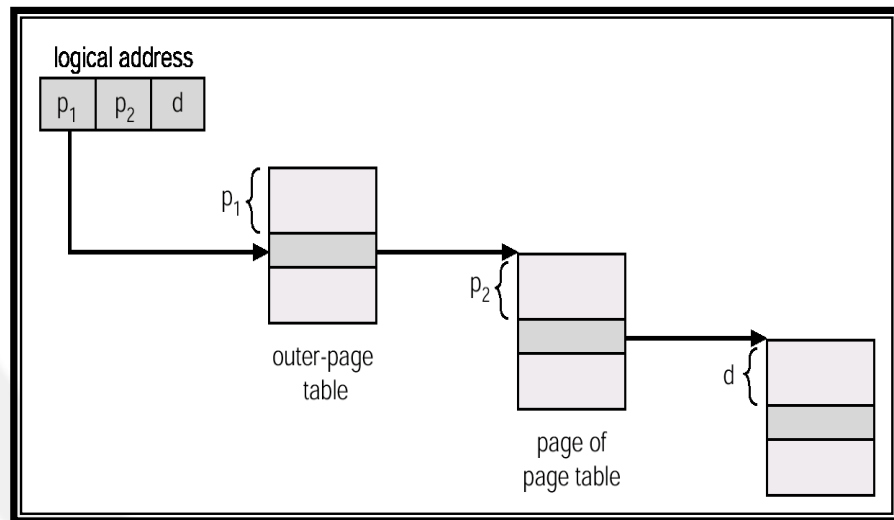
- Equivalente a “paginare” la tabella delle pagine
- In pratica, solo alcune parti della tabella delle pagine sono memorizzate esplicitamente in memoria, le altre sono su disco
- Possibili versioni a 2,3,4 livelli

# Paginazione a due livelli



# Esempio

- Indirizzo logico a 32-bit
- Dimensione pagina = 4K  
=  $2^{12}$
- 12 bit per offset (d)
- 20 bit per numero di pagina
  - p1: indice della tabella delle pagine esterna (10 bit)
  - p2: offset all'interno della pagina della page table interna (10 bit)





## Esempio (continua)

p1	p2	offset
0000000001	0000000011	000000000100

- p1 = 1
  - recuperare la tabella di 2° livello corrispondente all'indirizzo indicato dalla prima riga
- p2 = 3
  - recuperare l'indirizzo /3 indicato nella terza riga della tabella di 2° livello
- offset = 4
  - da sommare al valore /3 fornito dalla tabella di 2° livello

# Paginazione multilivello e prestazioni

- Ogni livello è memorizzato come una tabella separata in memoria, la conversione dell'indirizzo logico in quello fisico può richiedere 4 accessi a memoria (per paginazione a 3 livelli)
- Il TLB mantiene comunque le prestazioni ragionevoli
- Esempio:
  - $T_{MEM} = 90ns$
  - $T_{TLB} = 10ns$
  - $\alpha = 90\%$
  - $EAT = (10+90)*0,9 + (10+360)*0,1 = 127ns \sim 1.4 T_{MEM}$

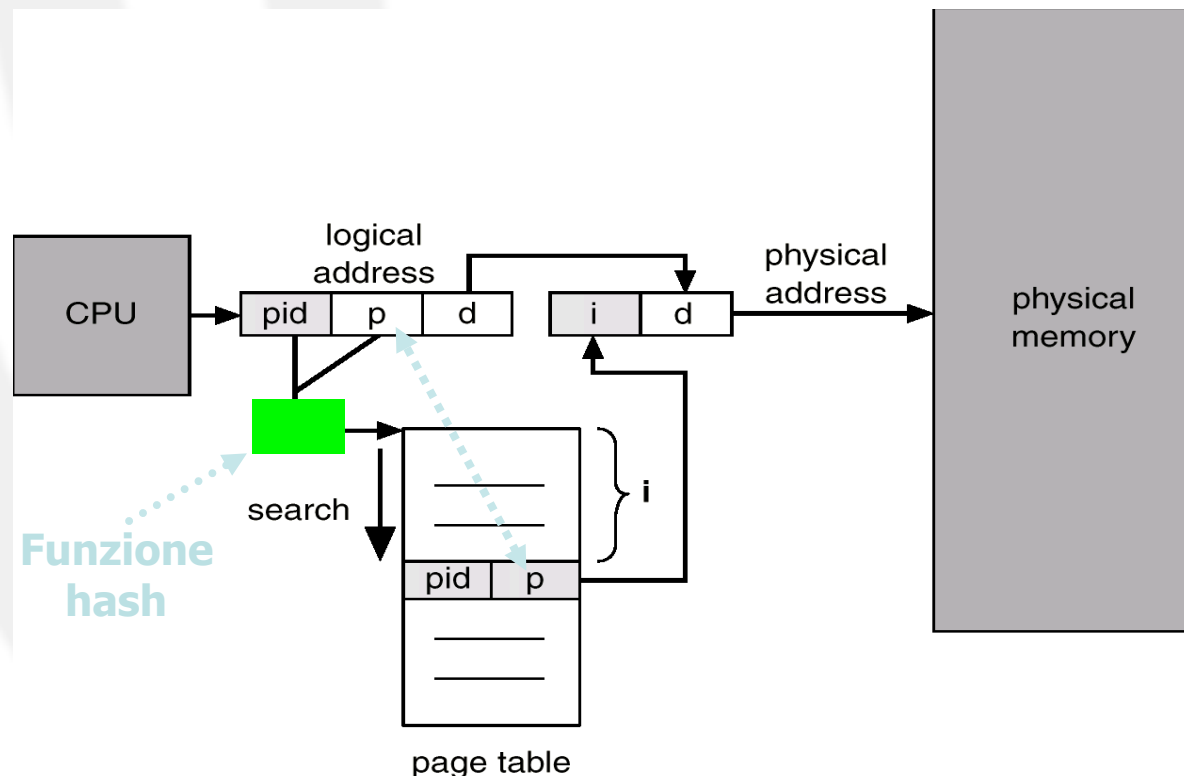
# Tabella delle pagine invertita

- Tabella unica nel sistema (non per processo!) avente una entry per ogni frame (pagina fisica), contenente:
  - indirizzo virtuale della pagina che occupa quel frame
  - “Informazioni” sul processo che usa quella pagina
- Problema:
  - Più di un indirizzo virtuale può corrispondere ad un dato indirizzo fisico (frame)
- Conseguenza:
  - E' necessario cercare il valore desiderato
  - Vi è un aumento del tempo necessario per cercare un riferimento ad una pagina

# Tabella delle pagine invertita

- La ricerca non può essere sequenziale (efficienza!)
  - Soluzione
    - Usiamo l'equivalente di una tabella hash
    - Riduzione del tempo di ricerca da  $O(n)$  a  $O(1)$
- In pratica, è necessario un meccanismo per gestire le collisioni quando diversi indirizzi virtuali corrispondono allo stesso frame

# Tabella delle pagine invertita: architettura concettuale

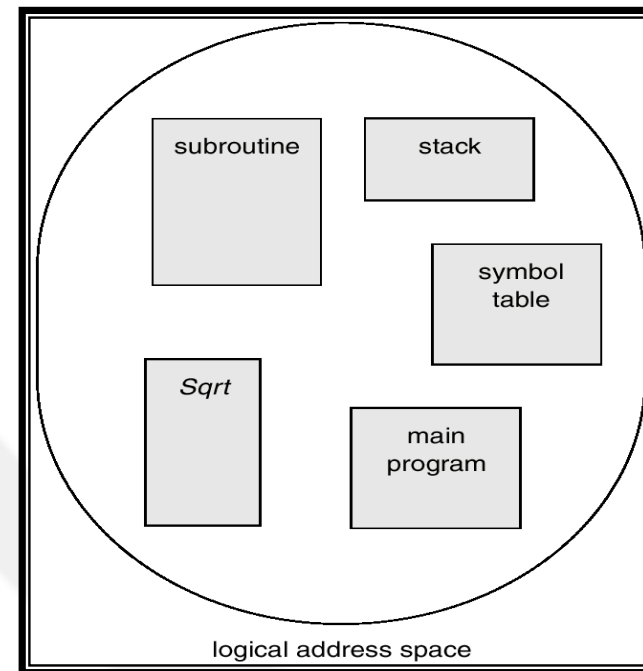


**Tabella unica nel sistema ordinata per frame**

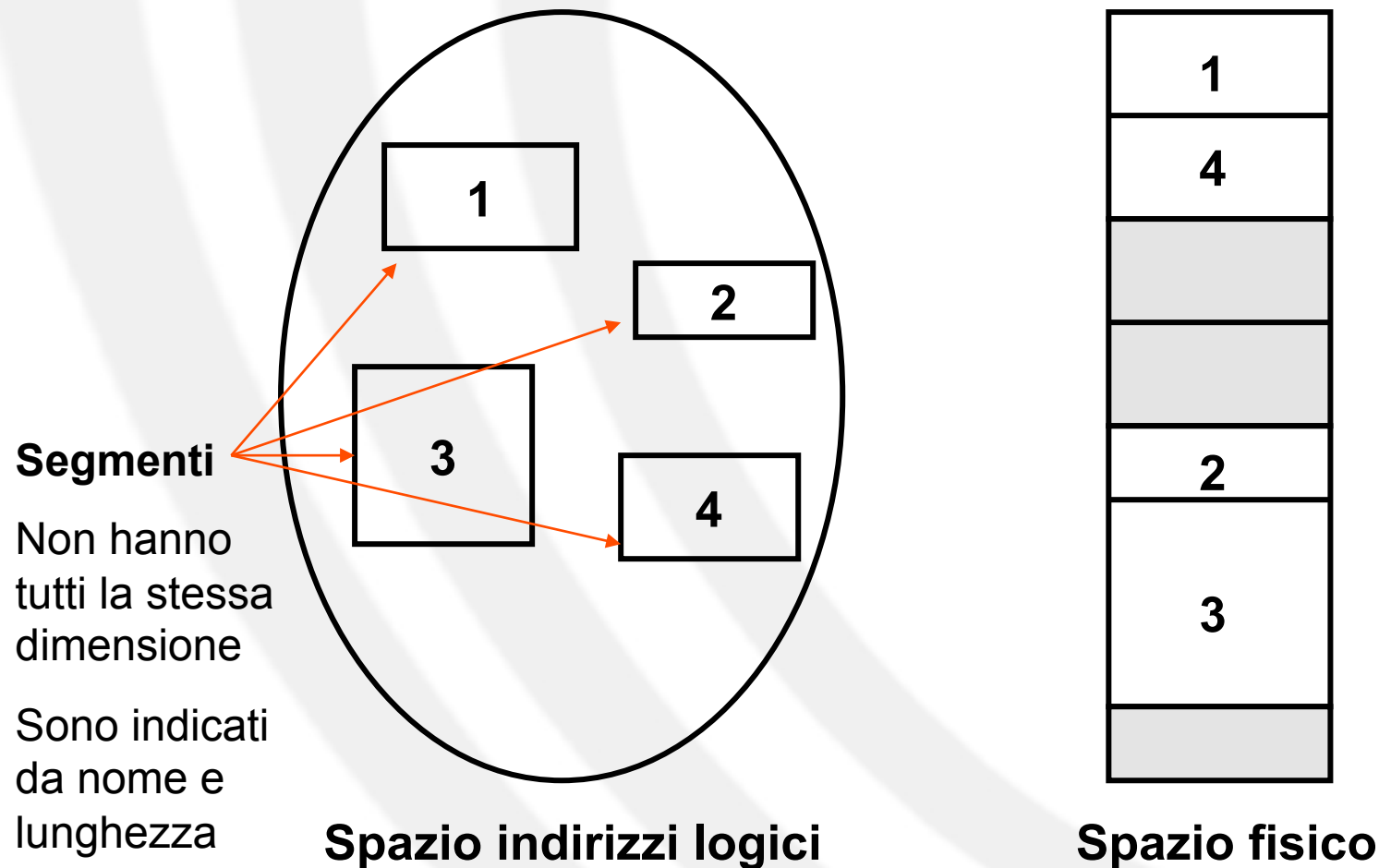
# SEGMENTAZIONE

# Segmentazione - motivazioni

- Schema di gestione della memoria che supporta la vista che l'utente ha della memoria
- Programma:
  - collezione di segmenti
- Segmento:
  - unità logica quale:
    - main
    - procedure
    - funzioni
    - variabili locali e globali
    - stack
    - symbol table
    - vettori



# Segmentazione - vista logica





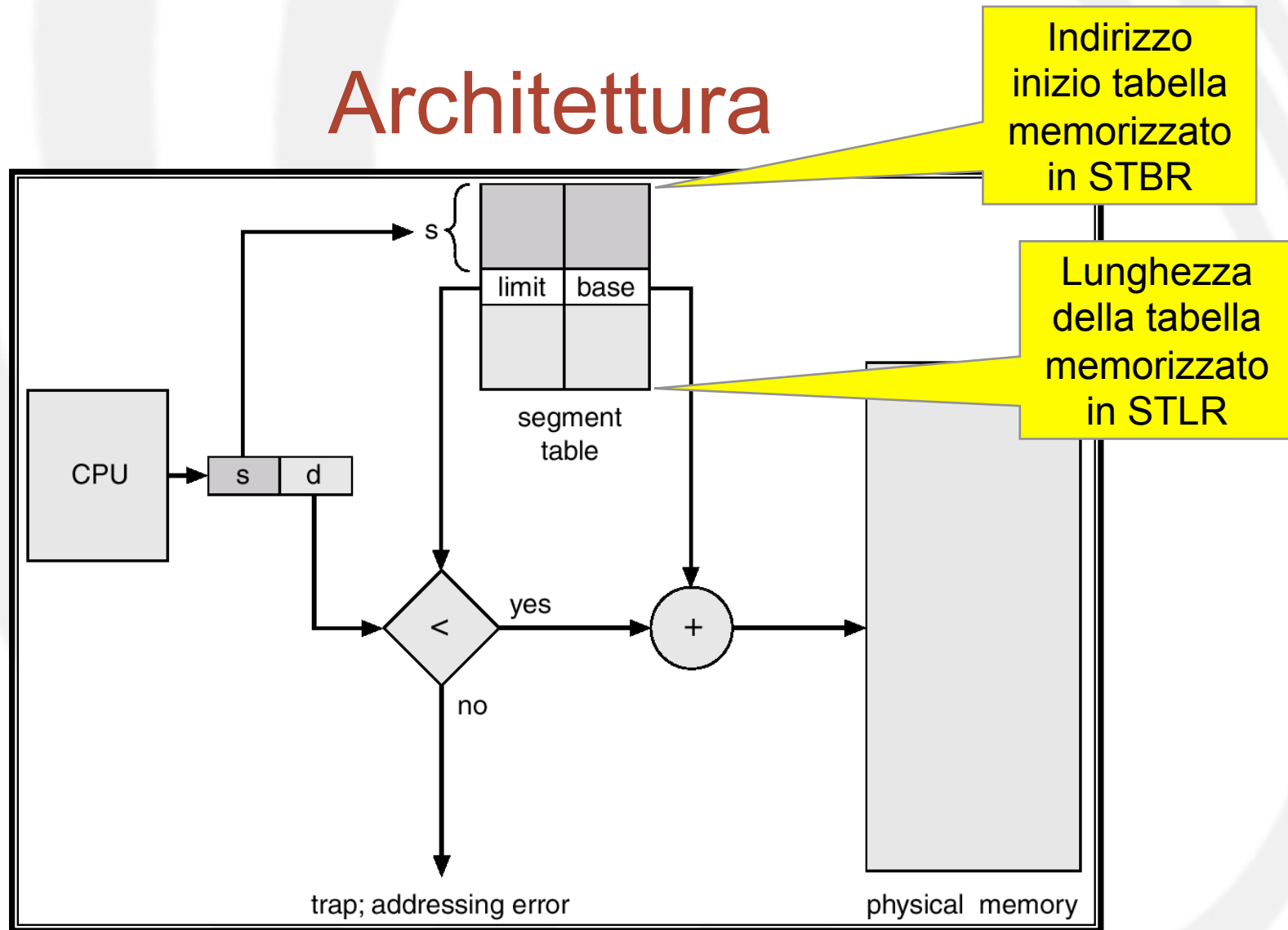
# Segmentazione - architettura

- Indirizzo logico
  - <numero di segmento, offset>
- Tabella dei segmenti (*segment table*)
  - mappa indirizzi logici bidimensionali in indirizzi fisici unidimensionali
  - Ogni entry contiene:
    - base
      - l'indirizzo fisico di partenza del segmento in memoria
    - limite
      - la lunghezza del segmento

# Tabella dei segmenti

- Simile a tabella delle pagine
- In memoria:
  - Segment-table base register (STBR)
    - punta alla locazione della tabella dei segmenti in memoria
  - Segment-table length register (STLR)
    - indica il numero di segmenti usati da un programma
    - un indirizzo logico  $\langle s, d \rangle$  è valido se  $s < \text{STLR}$
    - $\text{STBR} + s =$  indirizzo dell'elemento della tabella dei segmenti da recuperare
- TLB usato per memorizzare le entry maggiormente usate (come per paginazione)

# Architettura



# Traduzione indirizzi - esempio

Tabella dei segmenti

Segmento	Limite	Base
0	600	219
1	14	2300
2	100	90
3	580	1327

- Indirizzi logici

- $\langle 0, 430 \rangle$

- segmento 0, offset 430

- $430 < 600$ ? OK

- indirizzo fisico =  $430 +$   
base di 0 =  $430 + 219 =$   
649

- $\langle 1, 20 \rangle$

- segmento 1, offset 20

- $20 < 14$  NO!

- indirizzo non valido!

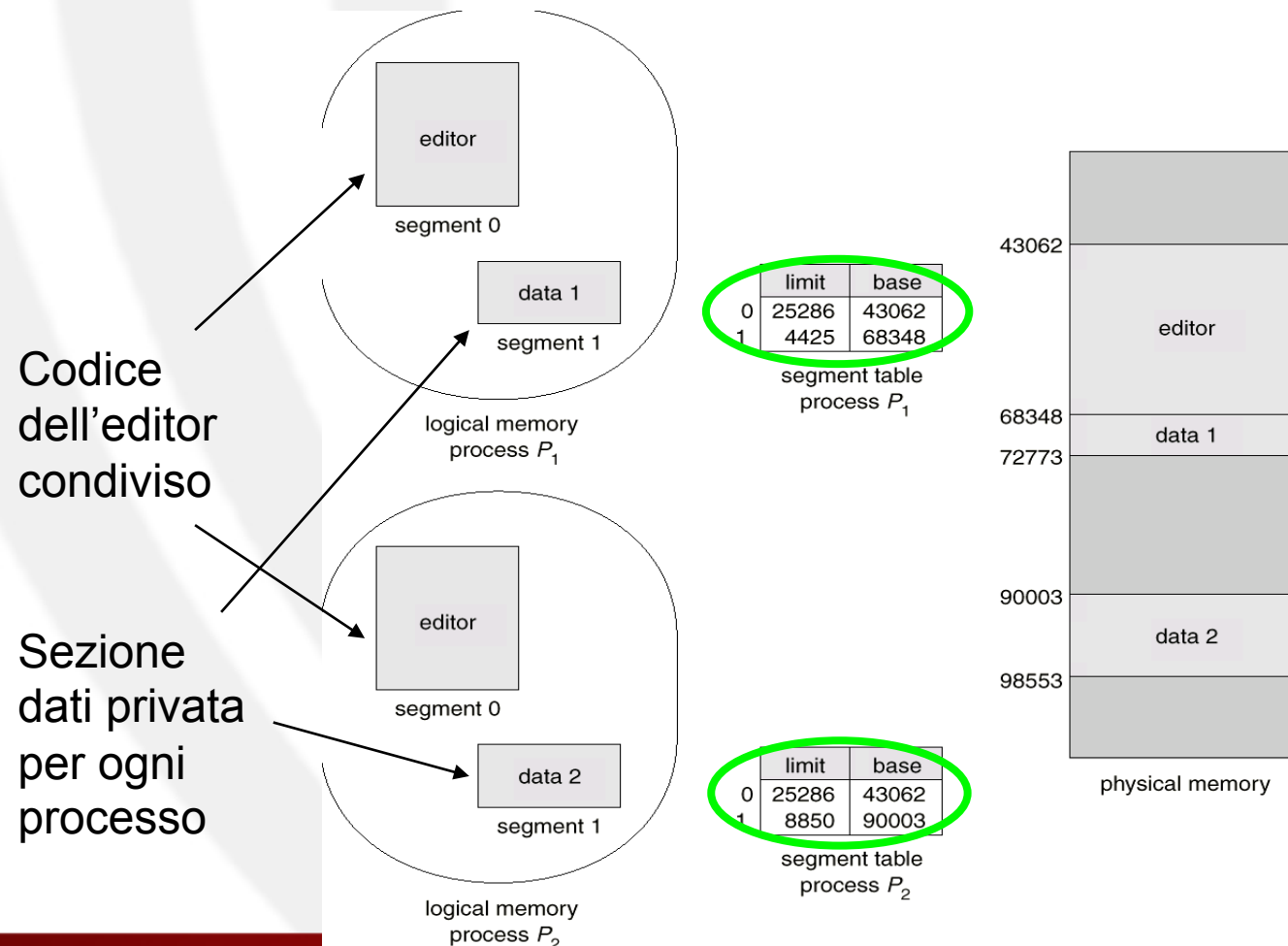
# Protezione

- Segmentazione supporta naturalmente la protezione (e la condivisione) di porzioni di codice
  - Segmento = entità con semantica ben definita (es. dati, istruzioni, ...)
- Protezione
  - ad ogni segmento sono associati
    - bit di modalità (read/write/execute)
    - valid bit (0  $\Rightarrow$  segmento non legale)
      - Es: scrittura su segmento read-only è facilmente riconosciuta e bloccata

# Condivisione

- A livello di segmento
  - Se si vuole condividere qualcosa basta inserirlo in un segmento
- Possibilità di condividere anche parti di un programma (es. funzioni di libreria)

# Esempio di condivisione



# Segmentazione e frammentazione

- S.O. deve allocare spazio in memoria per (tutti) i segmenti di un programma
  - Segmenti hanno lunghezza variabile
  - Allocazione dei segmenti
    - problema di allocazione dinamica risolto con first-fit o best-fit
- Possibilità di frammentazione esterna (specie per segmenti di dimensione significativa)



# Paginazione vs. segmentazione

- Paginazione

- Vantaggi

- Non esiste frammentazione (minima interna)
    - Allocazione dei frame non richiede algoritmi specifici

- Svantaggi

- Separazione tra vista utente e vista fisica della memoria

- Segmentazione

- Vantaggi

- Consistenza tra vista utente e vista fisica della memoria
    - Associazione di protezioni/condivisione ai segmenti

- Svantaggi

- Richiesta allocazione (dinamica) dei segmenti
    - Potenziale frammentazione esterna

# SEGMENTAZIONE PAGINATA

# Segmentazione paginata

- E' possibile combinare i due schemi per migliorarli entrambi
- La soluzione utilizzata consiste nel “paginare” i segmenti
  - Ogni segmento è suddiviso in pagine
  - Ogni segmento possiede la sua tabella delle pagine
  - La tabella dei segmenti non contiene l'indirizzo base di ogni segmento, ma l'indirizzo base delle tabelle delle pagine per ogni segmento
- Elimina il problema dell'allocazione dei segmenti e della frammentazione esterna!

# Architettura (MULTICS)

