Sistemi Operativi

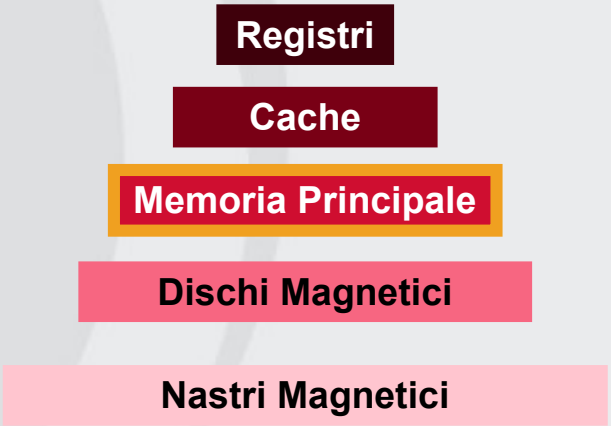
Principi di gestione della memoria
Swapping
Memoria virtuale

Necessità di gestire la memoria

- La memoria è una risorsa importante e deve essere gestita attentamente
- Un programmatore vorrebbe:
  - una memoria infinita
  - veloce
  - non volatile
  - poco costosa
- Questi desideri possono essere solo parzialmente soddisfatti (sono contraddittori)

Gerarchia di memoria

- Si cerca di raggiungere un compromesso sfruttando la **gerarchia di memoria**



Gerarchia di memoria

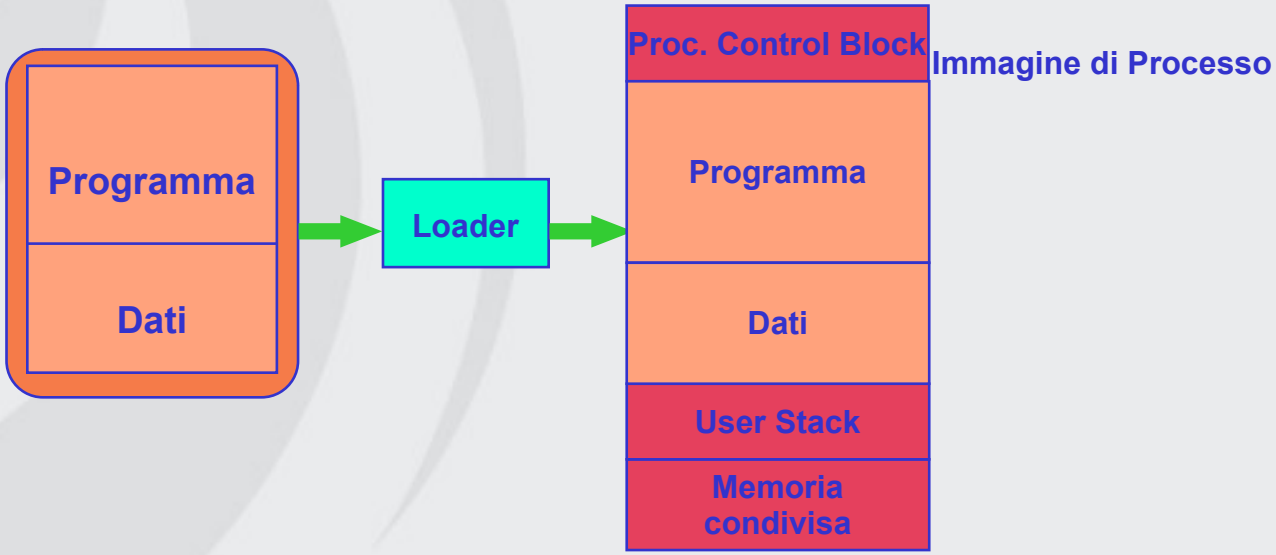
Dispositivo di memoria	Velocità di accesso	Capacità	Costo	Volatilità
Registri	1 ns	1 KB	molto alto	alta
Cache	qualche ns	> 1 MB	alto	alta
Memoria principale	10 ns	> 1 GB	<10$/GB	alta
Dischi magnetici	10 ms	1 TB	<100$/TB	bassa
Nastri magnetici	decine di s	< 1 TB	basso	bassa

Gestore della memoria

- La parte del SO che gestisce la memoria è il **Gestore di Memoria** i cui compiti sono:
  - tenere traccia di quali parti di memoria sono in uso e quali non lo sono
  - allocare la memoria ai processi che la necessitano e deallocarla
  - gestire lo swapping tra la memoria principale e il disco quando la memoria principale non è sufficientemente grande per mantenere tutti i processi
  - in definitiva cercare di sfruttare al meglio la gerarchia di memoria

Concetti base

- Il problema fondamentale da risolvere è
  - il passaggio da **programma eseguibile** (su memoria di massa)
  - a **processo in esecuzione** (in memoria di lavoro)



Address binding

- L'associazione tra istruzioni del programma e indirizzi di memoria può essere stabilita in momenti diversi:
  - Al momento della compilazione - Indirizzamento assoluto, se ad un dato momento l'indirizzo del programma deve cambiare occorre ricompilare il programma.
    Uno schema di questo tipo è utilizzato nei programmi .com del DOS
  - Al momento del caricamento - Il codice generato dal compilatore viene detto **rilocabile**, è il loader che fa le traduzioni opportune
  - Durante l'esecuzione
    - Il programma può essere spostato durante l'esecuzione stessa
- Non sempre è necessario caricare in memoria l'intero programma, con il **Dynamic Loading** una porzione di codice (una funzione C per esempio) è caricato solo se viene effettivamente eseguita

Librerie dinamiche

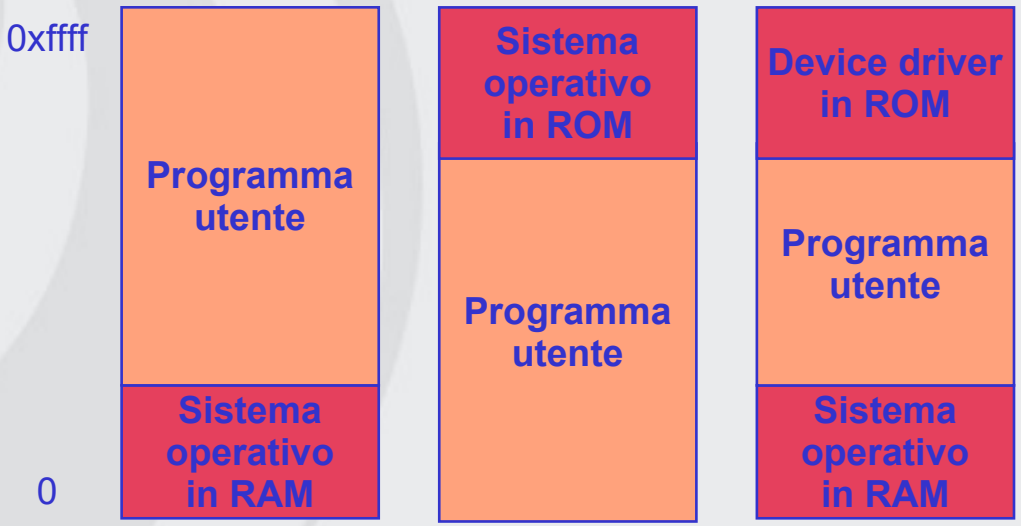
- Alcuni sistemi operativi permettono l'uso delle librerie dinamiche (**shared library, file .dll in Windows, .so in Unix**): in questo caso le librerie sono *linkate* solo al momento dell'esecuzione del programma
- Questa tecnica permette numerosi vantaggi:
  - i programmi sono molto più piccoli
  - l'aggiornamento delle librerie è più semplice (basta sostituirle non vi è bisogno di ricompilare i programmi)
  - vi può essere la possibilità che se una procedura è utilizzata da più di un processo se ne possa conservare una sola copia effettiva in memoria

Monoprogrammazione

- Lo schema più semplice di gestione della memoria è quello di avere un solo processo alla volta in memoria e consentire al processo di utilizzarla tutta
- Questo schema non è utilizzato (nemmeno sui computer meno costosi) in quanto implica che ogni processo contenga i driver di periferica per ogni dispositivo di I/O che utilizza

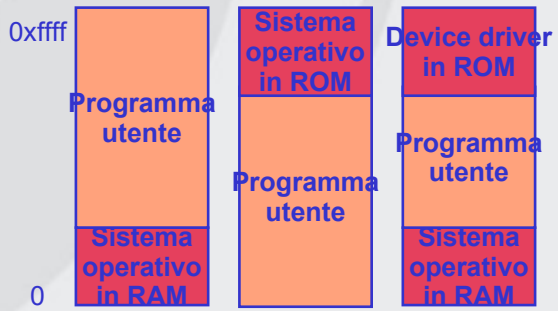
Monoprogrammazione

- Sui microcomputer viene quindi utilizzata la seguente strategia:



Monoprogrammazione

- La memoria viene divisa tra il SO e un singolo processo utente
- Il SO può essere:
  - nella RAM (Random Access Memory) in memoria bassa
  - nella ROM (Read Only Memory) in memoria alta
  - i driver di periferica nella ROM e il resto del SO nella RAM
- Quando il sistema è organizzato in questo modo solo un processo alla volta può essere in esecuzione



Multiprogrammazione

- Vantaggi della multiprogrammazione
  - rende più semplice programmare un'applicazione dividendola in due o più processi
  - fornisce un servizio interattivo a più utenti contemporaneamente
  - evita spreco di tempo di CPU
    dato che la maggior parte dei processi passa gran parte del tempo aspettando che vengano completate azioni di I/O del disco, in un sistema monoprogrammato durante questi intervalli di tempo la CPU non lavora

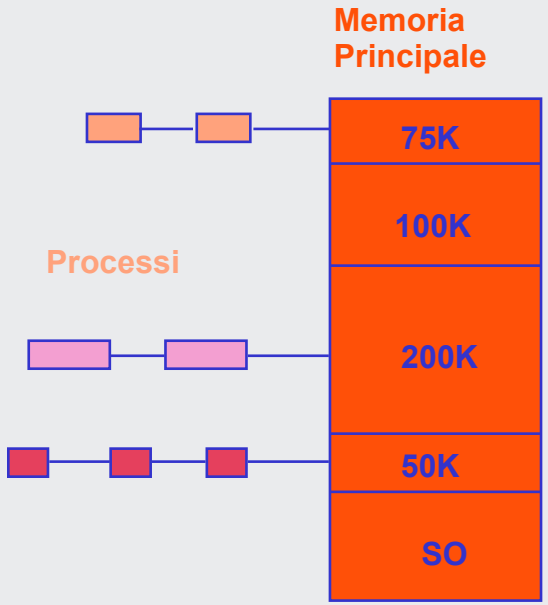
Multiprogrammazione

- **Ipotesi base**:
  - Supponendo che il processo medio sia in esecuzione per il 20% del tempo che risiede in memoria, con 5 processi in memoria, la CPU dovrebbe essere sempre occupata (assumendo ottimisticamente che i 5 processi non siano mai in attesa di I/O contemporaneamente)
- Come organizzare la memoria per poter gestire la multiprogrammazione?
- Soluzione più semplice:

  Multiprogrammazione con partizioni fisse
  - la memoria viene divisa in n partizioni (di grandezza eventualmente diversa)

Code separate

- Soluzione a partizioni fisse e code separate



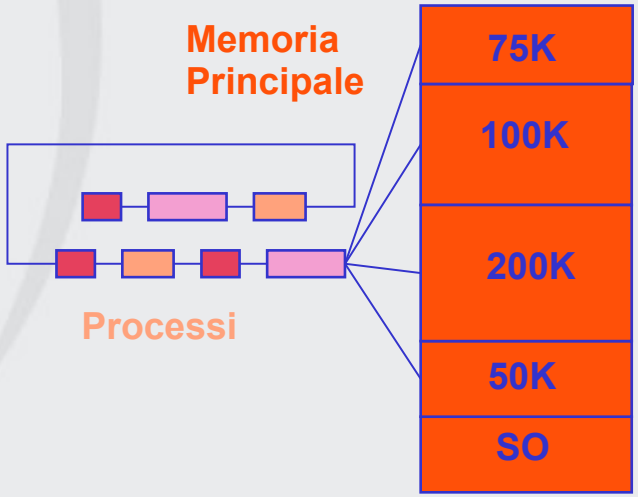
- quando arriva un job viene messo nella coda di input della più piccola partizione in grado di contenerlo

Code separate: problemi

- le partizioni sono fisse e quindi solo parzialmente occupate quindi lo spazio non utilizzato dal processo è perso
- svantaggio delle code di input separate:
  - quando la coda di input per una partizione piccola è piena mentre quella per una partizione grande è vuota, si ha un grande spreco di risorse

Coda unica

- **Soluzione a partizioni fisse e coda di input singola**
- quando si libera una partizione, il job più prossimo all'uscita della coda e con dimensione inferiore alla partizione viene caricato e quindi eseguito

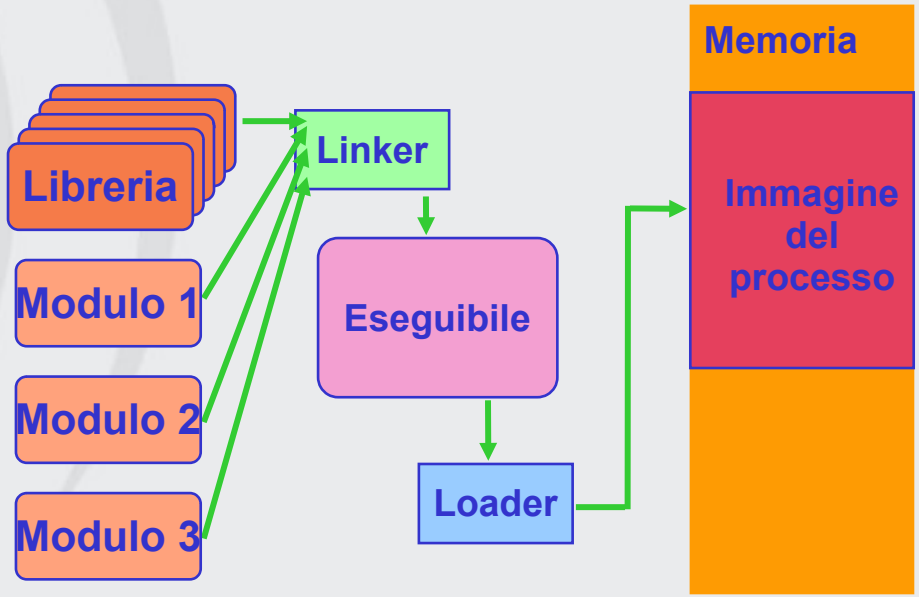


Coda unica: problemi

- Spreco di partizioni grandi per job piccoli
  Soluzione: cercare in tutta la coda il più grande job che può essere contenuto dalla partizione
  - Problema di questa soluzione:
    vi è discriminazione per i job piccoli perché sprecano spazio (assumendo i job piccoli come interattivi, essi dovrebbero avere il miglior servizio)
    - I Soluzione: avere almeno una partizione piccola per consentire l'esecuzione dei job piccoli
    - II Soluzione: una regola che consenta di stabilire che un job in attesa di esecuzione non venga ignorato più di k volte. Ogni volta che viene ignorato acquisisce un punto; quando ha k punti non può essere ulteriormente ignorato.
  - L'algoritmo di scheduling risulta complesso

Rilocazione

- Quando un programma viene *linkato*, il linker deve conoscere l'indirizzo di memoria corrispondente all'inizio del programma



Rilocazione: problemi

- Gli indirizzi relativi vanno trasformati in indirizzi assoluti
- Una possibile soluzione: è la rilocazione durante il caricamento
  occorre definire una lista o bit map che specifichi quali elementi vanno rilocati e quali no
- Rimane il problema della **protezione**: i programmi, definendo indirizzi assoluti, possono costruire un'istruzione che legge o scrive qualsiasi parola di memoria
  - in sistemi multiutente non è accettabile

Rilocazione e protezione

- Soluzione adottata dall'IBM per proteggere il 360
  - divisione della memoria in blocchi di 2 Kbyte e assegnamento a ogni blocco di un codice di protezione di 4 bit
  - assegnamento al PSW (Program Status Word) di una chiave di 4 bit
  - il processo in esecuzione ha la possibilità di accesso alla memoria solo se il codice di protezione corrisponde alla chiave nel PSW
  - solo il SO può cambiare i codici di protezione e la chiave
  - i processi utente non interferiscono tra loro e con il SO stesso

Rilocazione e protezione

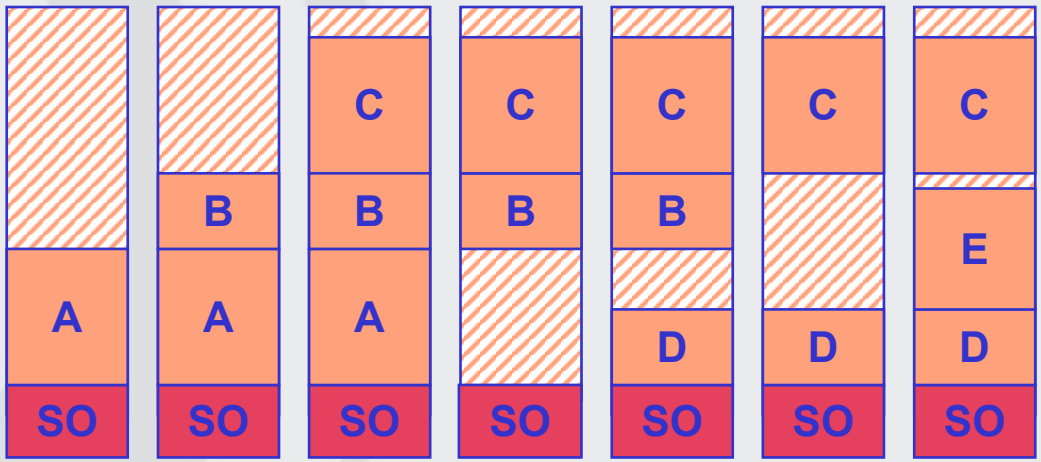
Soluzione con **registri base e limite**

- Quando un processo viene selezionato dallo scheduler per essere eseguito:
  - nel registro **base** viene caricato l'indirizzo di inizio della sua partizione
  - nel registro **limite** viene caricata la lunghezza della partizione
- Ad ogni accesso alla memoria:
  1. a ogni indirizzo di memoria generato viene automaticamente sommato il contenuto del registro base prima di essere effettivamente inviato alla memoria
  2. poi viene eseguito un controllo degli indirizzi anche sul registro limite affinché non vi siano tentativi di accesso alla memoria fuori dalla partizione corrente
- I registri base e limite non sono modificabili dall'utente

Partizioni variabili

- Nei sistemi multiutente normalmente la memoria è insufficiente per tutti i processi degli utenti attivi
- È necessario trasferire l'immagine dei processi in eccesso su disco, lo spostamento dei processi da memoria a disco e viceversa viene detto **swapping**
- Il problema con le partizioni fisse è lo spreco di memoria per programmi più piccoli delle partizioni che li contengono soluzione: **partizioni variabili**

Partizioni variabili



Partizioni variabili

- Differenze tra partizioni fisse e partizioni variabili:
  - Con le partizioni variabili il numero, la locazione e la dimensione delle partizioni varia dinamicamente mentre con le partizioni fisse questi parametri sono stabiliti a priori
  - La flessibilità delle partizioni variabili migliora l'utilizzo della memoria; con le partizioni fisse ci sono problemi con partizioni troppo piccole o troppo grandi
  - Le partizioni variabili complicano, rispetto alle partizioni fisse, la gestione delle operazioni di allocazione e deallocazione della memoria

Partizioni variabili

- Quando si formano troppi buchi in memoria è possibile combinare tutti gli spazi liberi in memoria in un unico grande spazio muovendo tutti i processi in memoria verso il basso
- Questa tecnica viene chiamata **compattazione della memoria** generalmente non viene utilizzata perché richiede molto tempo di CPU

Allocazione di memoria

- Quanta memoria dovrebbe essere allocata per un processo quando viene creato o viene portato in memoria tramite swapping?
  - **processi a dimensione fissa** l'allocazione della memoria al processo è semplice viene assegnata al processo esattamente la memoria che necessita

Allocazione di memoria

- Se si sa che i processi tendono a crescere conviene lasciare spazio a disposizione del processo

Allocazione di memoria

Quando un processo cerca di crescere:

- se il processo è adiacente ad uno spazio libero, questo può essere allocato
- se il processo è adiacente a un altro processo:
  1. può essere spostato in uno spazio di memoria libero sufficientemente grande da contenerlo
  2. uno o più processi dovranno essere trasferiti su disco per creare uno spazio libero abbastanza grande da contenerlo
- se il processo non può crescere in memoria e l'area di swapping su disco è piena, il processo deve aspettare o essere "ucciso"

Gestione della memoria

- Uso di bitmap
- Uso di liste
- Altri approcci

Gestione con bitmap

- Con questa tecnica la memoria è suddivisa in unità di allocazione: a ognuna corrisponde un bit nella bitmap

11111000
11111111
10011111
11110001

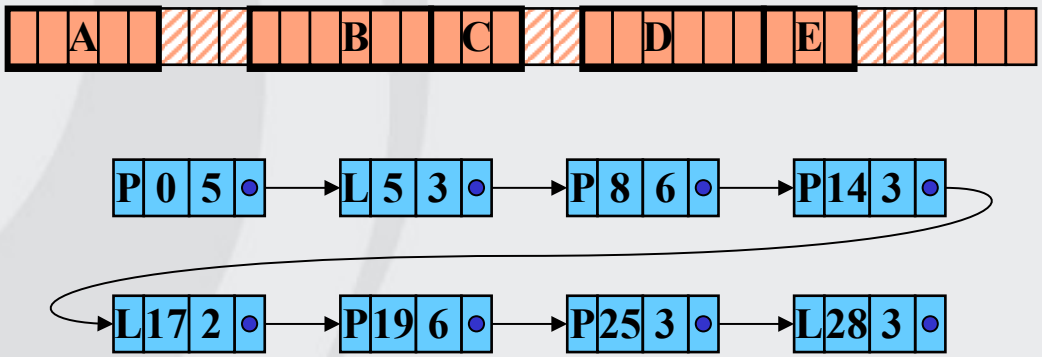
- La scelta della dimensione dell'unità di memoria è importante perché si riflette sulle dimensioni della bitmap e sull'ottimizzazione dell'occupazione di spazio

Gestione con bitmap

- **Vantaggio**:
  - metodo semplice per tenere traccia delle parole di memoria in una quantità fissata di memoria
- **Svantaggio**:
  - per poter eseguire un processo di k unità, il gestore di memoria deve ricercare k zeri consecutivi nella bitmap
  - questa è una operazione lenta per cui le bitmap sono poco utilizzate

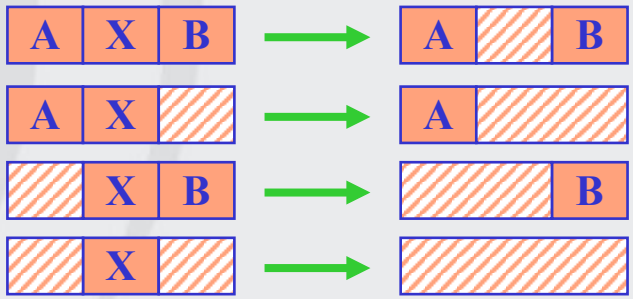
Gestione con liste

- Tecnica con la quale si tiene traccia dei segmenti di memoria allocati e liberi
- Per **segmento** si intende una zona di memoria assegnata ad un processo oppure una zona libera tra due assegnate



Gestione con liste

- I processi e le zone libere sono tenuti in una lista ordinata per indirizzo
  in questo modo le operazioni di aggiornamento risultano semplificate
- La liberazione di una zona di memoria si risolve in quattro casi possibili:

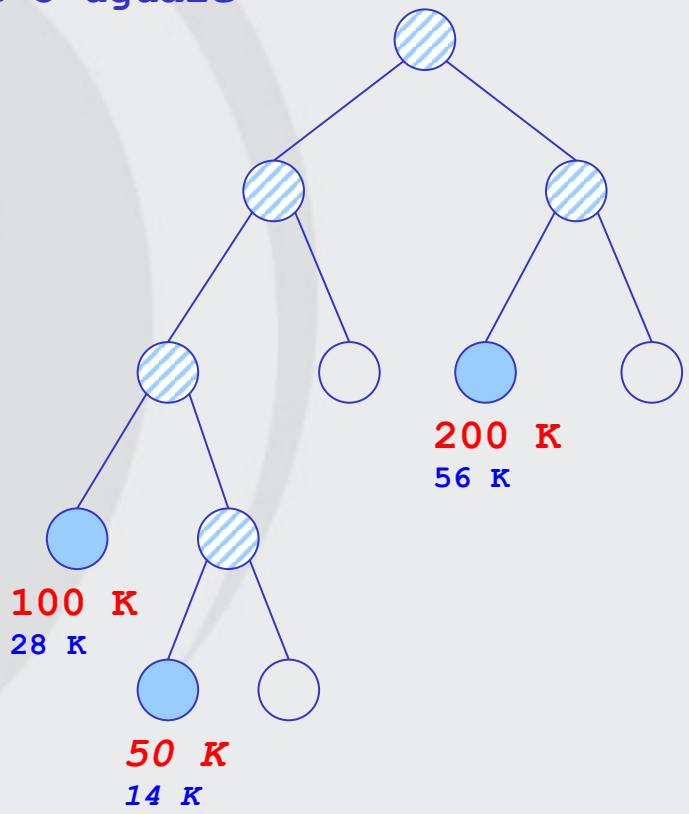


Gestione con liste

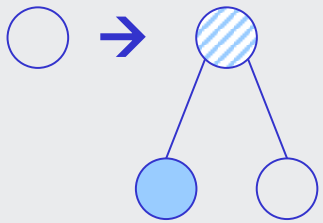
- Mediamente vi è un numero di segmenti liberi pari a metà del del numero di processi
  - Non vi è simmetria tra segmenti liberi e occupati in quanto non vi possono essere segmenti liberi consecutivi
  - Per semplificare le operazioni può essere conveniente usare una lista a doppia concatenazione

Buddy system

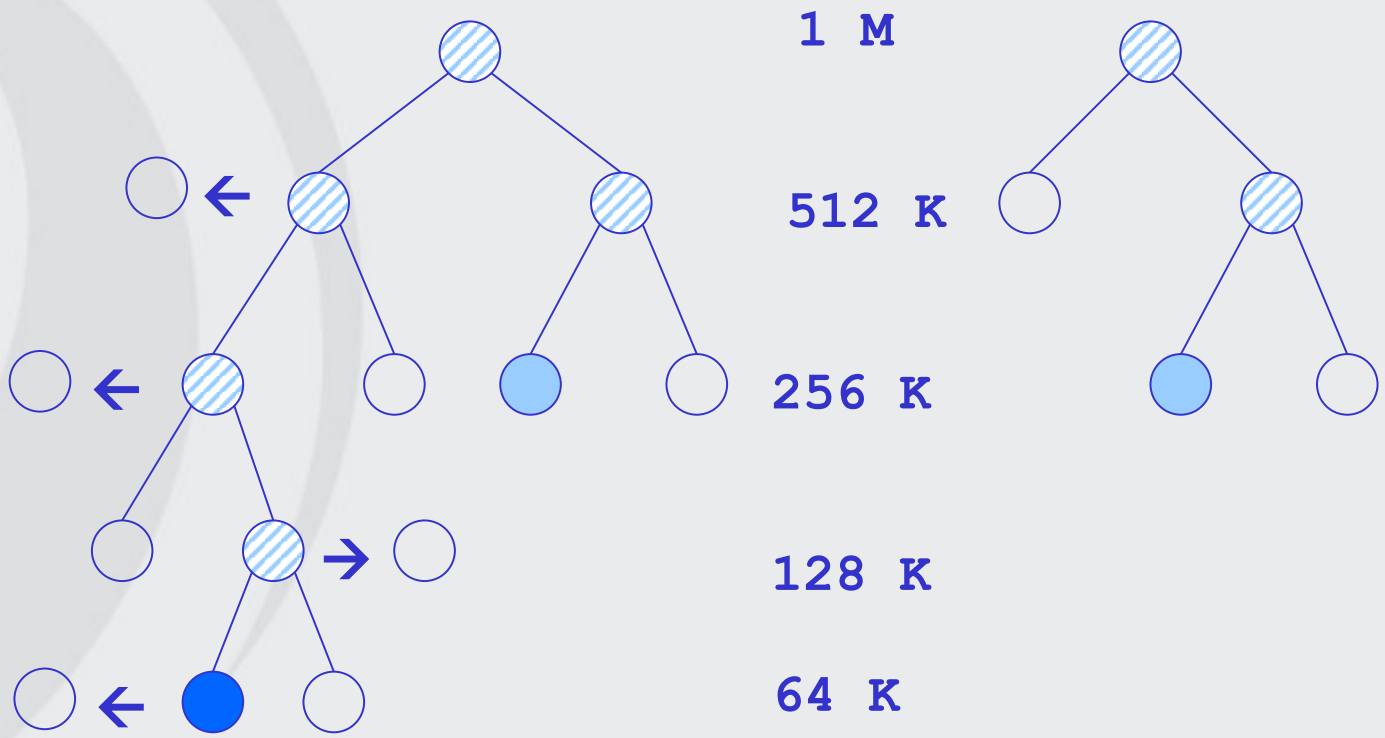
Ogni blocco di memoria viene arrotondato alla prima potenza di 2 superiore o uguale



Se entra un nuovo processo di 90 K cerco un blocco di 128 K



Buddy system: rilascio



Se un processo esce

Tende a sprecare memoria a causa dei blocchi di dimensione fissa

Algoritmi di allocazione

- First fit (primo spazio utile)
- Next fit (spazio utile successivo)
- Best fit (miglior spazio utile)
- Worst fit (peggior spazio utile)

First fit

- Il gestore di memoria scandisce la lista dei segmenti finché trova la prima zona libera abbastanza grande

- la zona viene divisa in due parti, una per il processo e una per la memoria non utilizzata
- È un algoritmo veloce perché limita al massimo le operazioni di ricerca
- **Next fit**: come il precedente, ma ogni ricerca inizia dal punto lasciato alla ricerca precedente
  - simulazioni mostrano prestazioni leggermente peggiori di first fit

Best fit

- Il gestore di memoria scandisce tutta la lista dei segmenti e sceglie la più piccola zona libera sufficientemente grande da contenere il processo
  più lento di first fit, spreca più memoria di first fit e next fit perché lascia zone di memoria troppo piccole per essere utilizzate
- **Worst fit**
  - per risolvere il problema precedente sceglie la più grande zona libera
  - non presenta comunque buone prestazioni

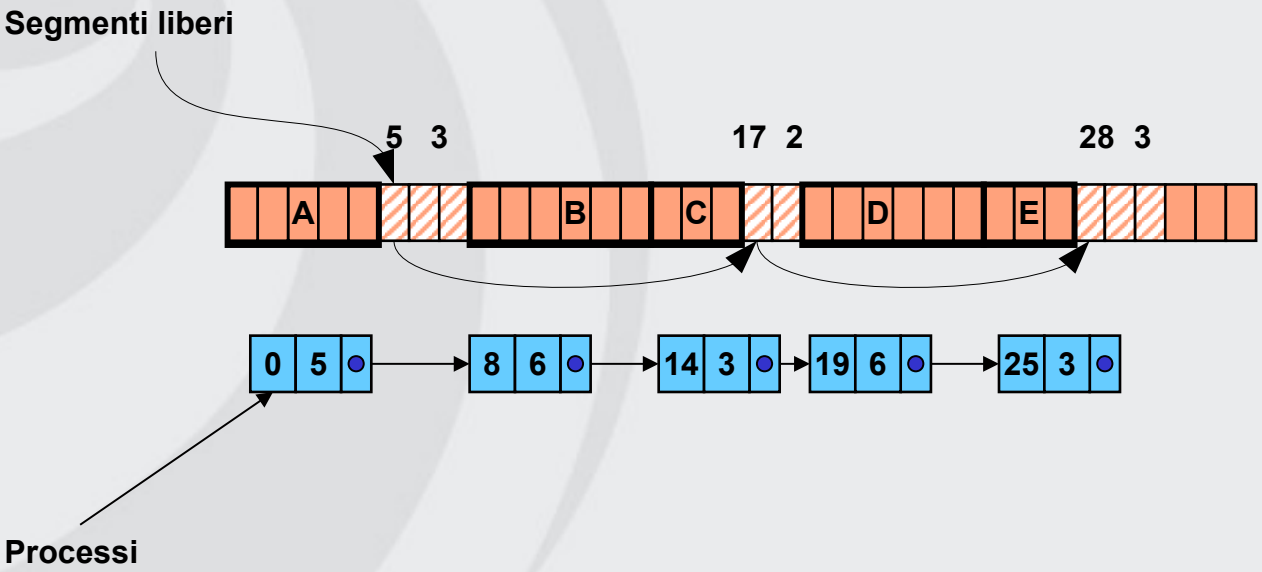
Commenti

- Questi 4 algoritmi possono essere velocizzati tenendo separate le liste per i processi e gli spazi liberi
  - La gestione di due liste separate comporta rallentamento e maggior complessità quando vengono deallocati spazi di memoria
- Ordinando la lista degli spazi vuoti in ordine crescente di dimensione, si ottimizza la ricerca per il Best Fit

Commenti

- Un quinto algoritmo è il **Quick Fit** che mantiene liste separate per le zone di memoria con dimensioni maggiormente richieste
  - È molto veloce nella ricerca di uno spazio libero di dimensioni fissate
  - Come tutti gli altri è costoso in termini di tempo per l'aggiornamento delle liste successivo alla deallocazione di memoria (tempo di ricerca per verificare possibili ***merge*** tra spazi liberi adiacenti)

Gestione con liste separate



Gestione della memoria

Memoria virtuale

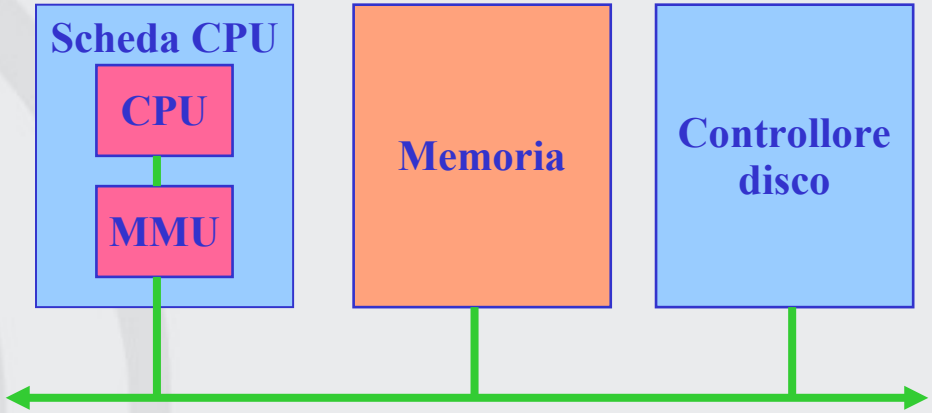
Memoria virtuale

- Principio base:
  - la dimensione totale di programma, dati e stack può eccedere la quantità di memoria fisica disponibile (per il processo)
  - Il Sistema Operativo mantiene in memoria principale solo le parti del programma in uso e mantiene il resto su disco
- Si citano due approcci diversi:
  - Paginazione
  - Segmentazione

Paginazione

- In qualsiasi computer, ogni programma può produrre un insieme di indirizzi di memoria
- Questi indirizzi generati dal programma sono detti **Indirizzi Virtuali** e formano lo **Spazio di Indirizzamento Virtuale**
- Nei computer senza memoria virtuale, l'indirizzo virtuale viene messo direttamente sul bus di memoria, quindi la parola di memoria fisica con lo stesso indirizzo viene letta o scritta
- Quando viene utilizzata la memoria virtuale gli indirizzi virtuali non vengono messi direttamente sul bus di memoria ma vengono mandati alla **Memory Management Unit (MMU)** un chip che mappa gli indirizzi virtuali sugli indirizzi della memoria fisica

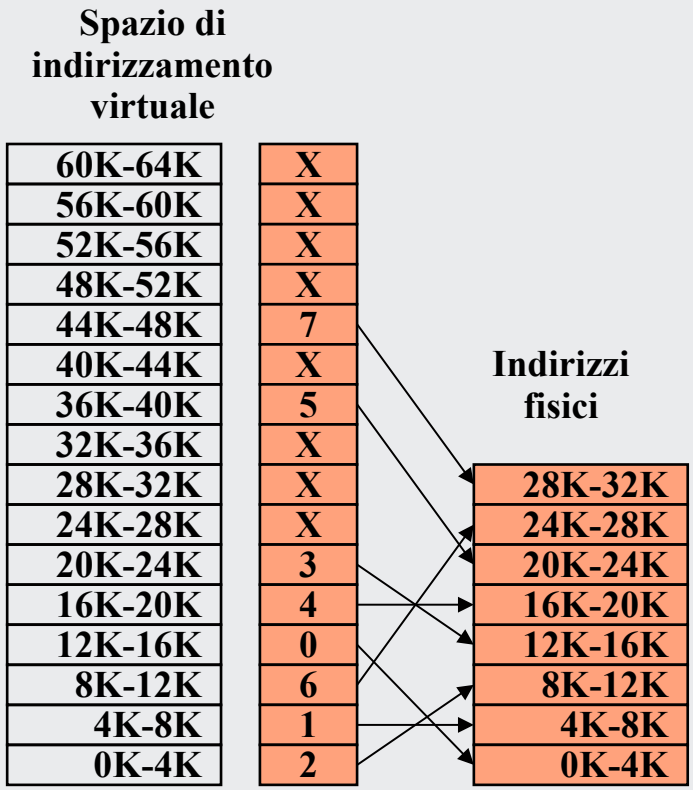
Paginazione



- Lo spazio di indirizzamento virtuale è diviso in **Pagine** mentre le unità nella memoria fisica corrispondenti alle pagine sono detti **Frame di pagina**
- Pagine e frame hanno sempre la stessa dimensione, normalmente da 512 byte a 16M, i trasferimenti fra la memoria ed il disco avvengono sempre per unità di una pagina

Esempio di paginazione

- La macchina ha 64K di memoria virtuale e 32K di memoria fisica, le pagine sono di 4K



Esempio di paginazione

istruzione:

```
move reg, 0x0000
```

viene interpretata come:

```
move reg, 0x2000
```

istruzione:

```
move reg, 0x2000
```

viene interpretata come:

```
move reg, 0x6000
```

istruzione:

```
move reg, 0x2010
```

viene interpretata come:

```
move reg, 0x6010
```

Page fault

- Alcune pagine virtuali non possono avere corrispondenza in memoria fisica
- Negli attuali circuiti HW un **bit presente/assente** è utilizzato per tenere traccia se la pagina mappata è sulla memoria fisica oppure no
- Ad esempio l'istruzione:

```
move reg, 0x8020
```

non risulta mappata in una pagina in memoria

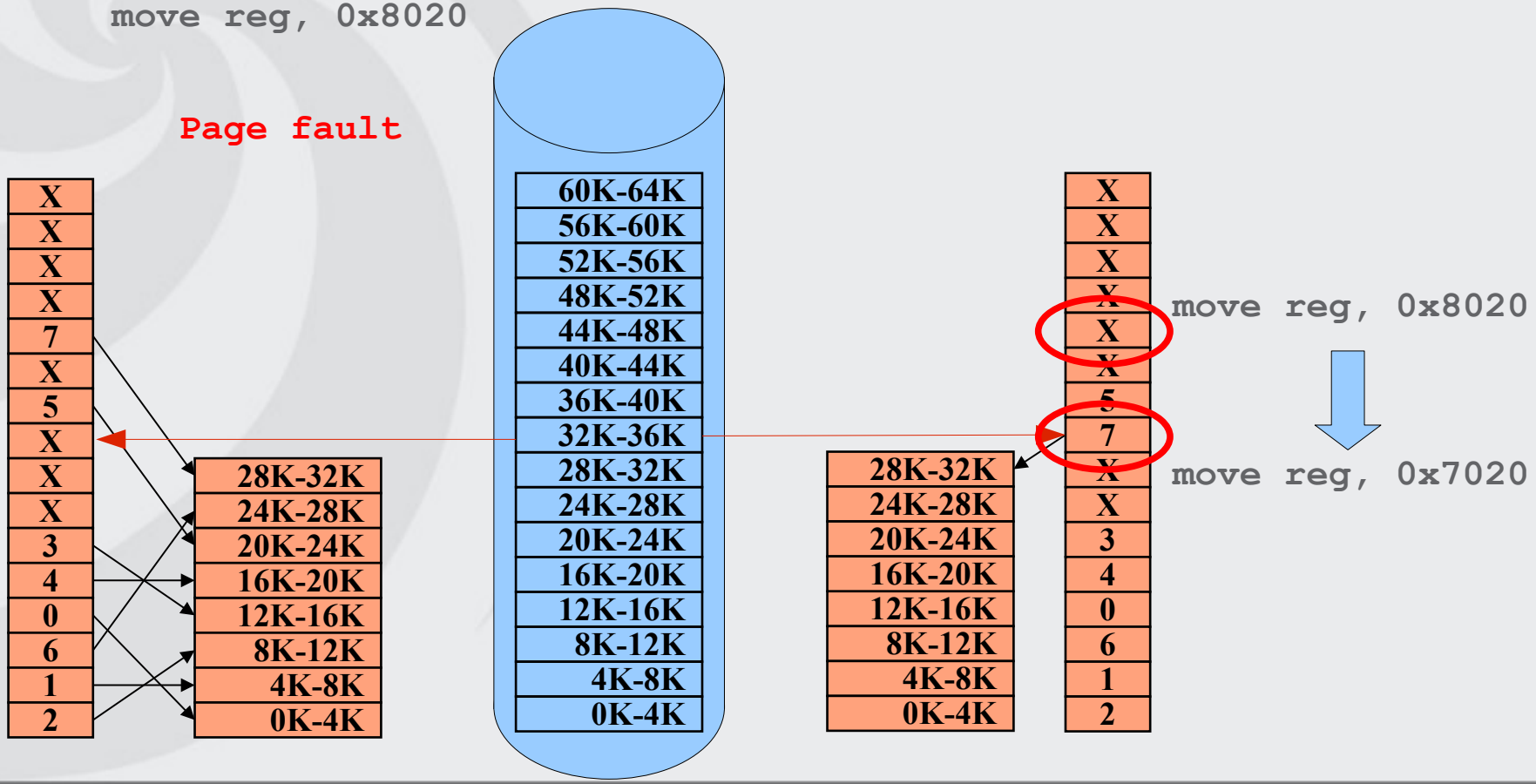
Page fault

- La MMU causa un'eccezione della CPU al SO detta:

  page fault

  - il SO sceglie un frame di pagina poco utilizzato
  - salva il suo contenuto su disco (se necessario)
  - recupera la pagina referenziata e la alloca nel frame appena liberato (fetching della pagina)
  - aggiorna la mappa
  - riparte con l'istruzione bloccata

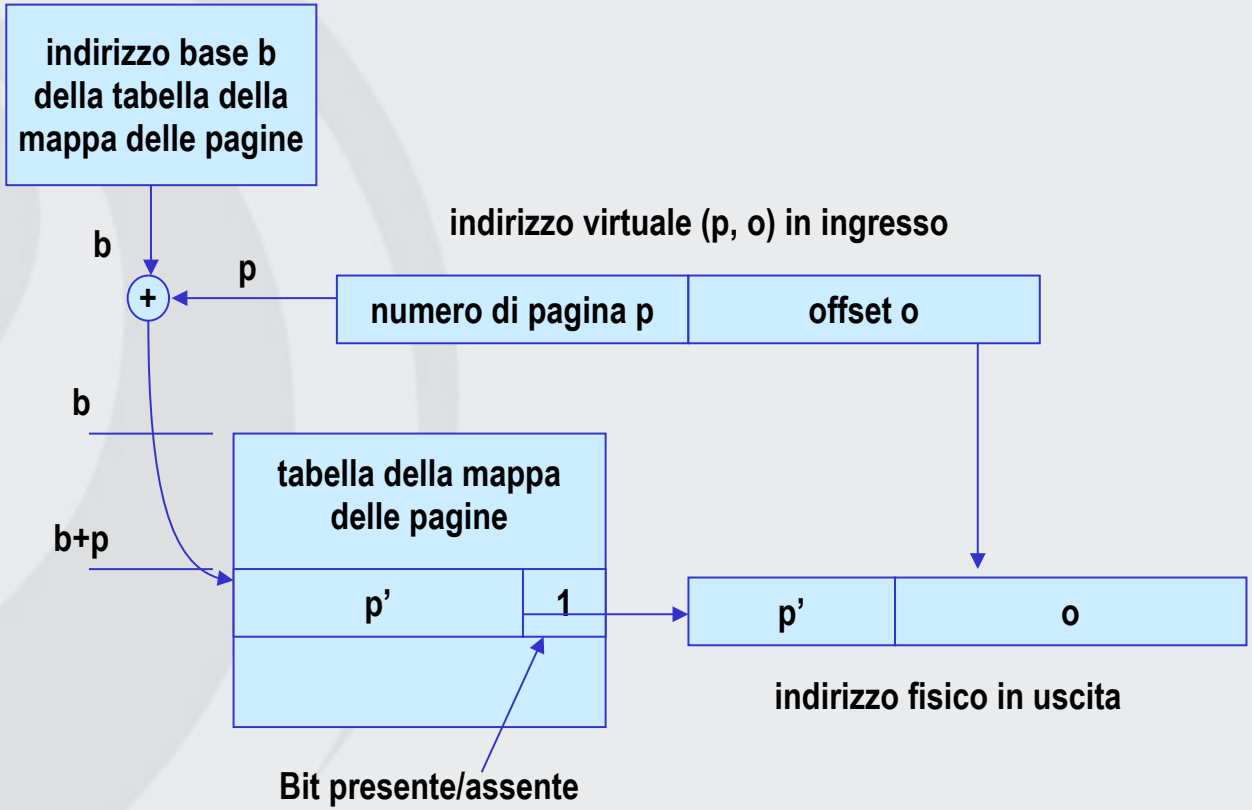
Esempio di paginazione



Funzionamento MMU

- L'indirizzo virtuale di 16 bit (**NB: è un esempio**) viene diviso in:
  - un numero di pagina di 4 bit
  - un offset di 12 bit
- Il numero di pagina viene utilizzato come indice nella tabella delle pagine così da ottenere l'indirizzo fisico
- Se il bit presente/assente è 0 viene causata una eccezione
- Se il bit presente/assente è 1 il numero del frame trovato nella tabella delle pagine viene copiato nei 3 bit di ordine alto del registro di output con i 12 bit dell'offset (copiati senza modifiche dall'indirizzo virtuale)
- Il contenuto del registro di output viene messo sul bus di memoria come indirizzo di memoria fisica

Funzionamento MMU



La tabella delle pagine

- Scopo della tabella delle pagine è mappare le pagine virtuali sui frame
- Il modello descritto è semplice, vi sono però dei problemi
  - La tabella delle pagine può essere molto grande
  - Il mapping (da virtuale a fisico) deve essere veloce

La tabella delle pagine

- Il modello più semplice consiste di una singola tabella delle pagine costituita da un array di registri hardware con un elemento per ogni pagina virtuale indicizzato dal numero di pagina virtuale. Quando viene iniziato un processo, il SO carica la sua tabella delle pagine nei registri
  - **vantaggi**: il mapping è immediato, nessun riferimento in memoria
  - **svantaggi**: il mapping è potenzialmente costoso (se la tabella delle pagine è grande) e il caricamento della tabella delle pagine nei registri ad ogni context switch può alterare le prestazioni (si noti che deve esistere una tabella diversa per ogni processo)

La tabella delle pagine

- All'estremo opposto la soluzione è mantenere la tabella delle pagine interamente in memoria. L'hardware necessario è un singolo registro che punta all'inizio della tabella delle pagine
  - **vantaggi**: consente di modificare la mappa di memoria ad un context switch ricaricando un solo registro
  - **svantaggi**: durante l'esecuzione di un'istruzione richiede uno o più riferimenti in memoria per leggere gli elementi della tabella delle pagine

Tabelle a più livelli

- Il segreto del metodo a tabelle delle pagine multilivello è evitare di tenere tutte le tabelle delle pagine in memoria per tutto il tempo

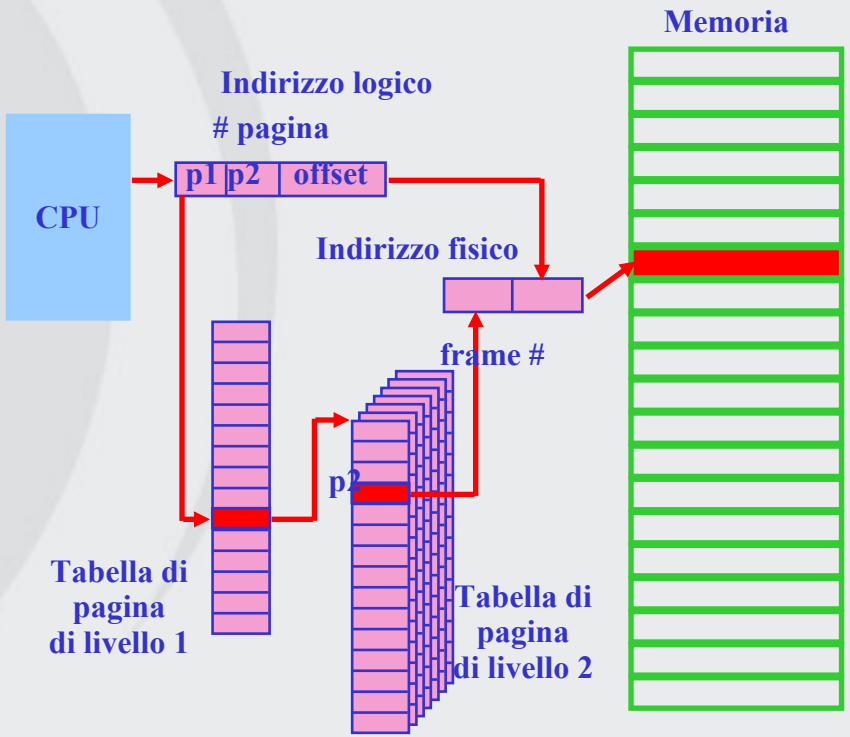


Tabelle a più livelli

- Esempio a 32 bit

Ogni tabella indirizza una sezione di memoria di 4M (2^{22})

Sono sufficienti 3 tabelle di secondo livello

Tabelle a più livelli

- Struttura di ogni elemento della tabella:
  - Il campo più importante è il numero del frame di pagina
  - Il bit presente/assente consente di verificare se la pagina virtuale corrispondente all'elemento è in memoria oppure no
  - I bit di protezione contengono le informazioni su quali tipi di accesso sono consentiti (lettura, scrittura ed eventualmente esecuzione di una pagina)
  - I bit di pagina modificata e referenziata tengono traccia dell'utilizzo della pagina
  - Il bit di caching disabilitato è importante per le pagine che mappano su registri di periferiche invece che in memoria

Memoria associativa

- Le tabelle delle pagine vengono tenute in memoria per le loro grandi dimensioni, ciò può penalizzare fortemente le prestazioni (una singola istruzione può fare riferimento a più indirizzi)
- I programmi tendono a fare la maggior parte dei riferimenti a un piccolo numero di pagine, quindi solo una piccola parte degli elementi nella tabella delle pagine vengono letti frequentemente
- Da cui una possibile soluzione: la **memoria associativa** o **TLB** (Translation Lookaside Buffer): un dispositivo hardware per mappare gli indirizzi virtuali su indirizzi fisici senza utilizzare la tabella delle pagine

Memoria associativa

```
#define N 0x1000
int v1[N], v2[N];
int i, a, b, …;

for(i=0; i<N; i++)
{
     a = v1[i] ;
     …
     v2[i] = b ;
}
```

Valido	Pagina virtuale	Modificato	Protezione	Frame di pagina	
1	140	1	rw-	31	variabili locali
1	20	0	r-x	38	codice
1	130	1	rw-	29	vettori
1	129	0	rw-	62	
1	19	0	r-x	50	codice
1	21	0	r-x	45	
1	860	1	rw-	14	stack
1	861	1	rw-	75	

Stampa indirizzi di memoria

osboxes@osboxes: ~

```
#include <stdio.h>

int v[100];

void f(int a[])
{
    int i;
    printf("i:\t%p\n", &i);
    printf("a:\t%p\n", &a);
    printf("a[0]:\t%p\n", a);
    printf("f:\t%p\n", f);
    printf("printf:\t%p\n", printf);
}

int main(int argc, char* argv[])
{
    f(v);
    *((int *) f) = 1; operazione non lecita: il codice non è modificabile
    return 0;
}
```

Esecuzione

sistema a 32 bit		a 64 bit	
i:	0x61cc3c	i:	0x7ffd8f4da424
a:	0x61cc50	a:	0x7ffd8f4da418
a[0]:	0x406100	a[0]:	0x6010c0
f:	0x4011a0	f:	0x4006b6
printf:	0x401248	printf:	0x400490
Segmentation fault		Segmentation fault	

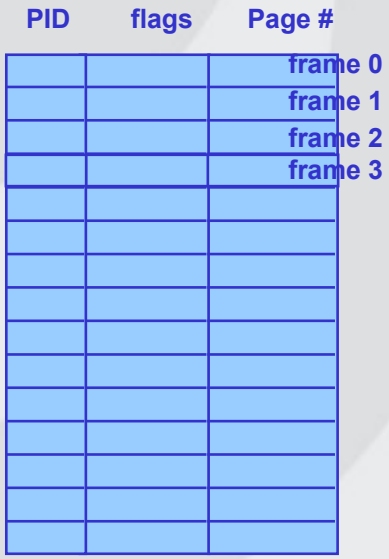
Codice: indirizzi bassi

Stack: indirizzi alti

Memoria associativa

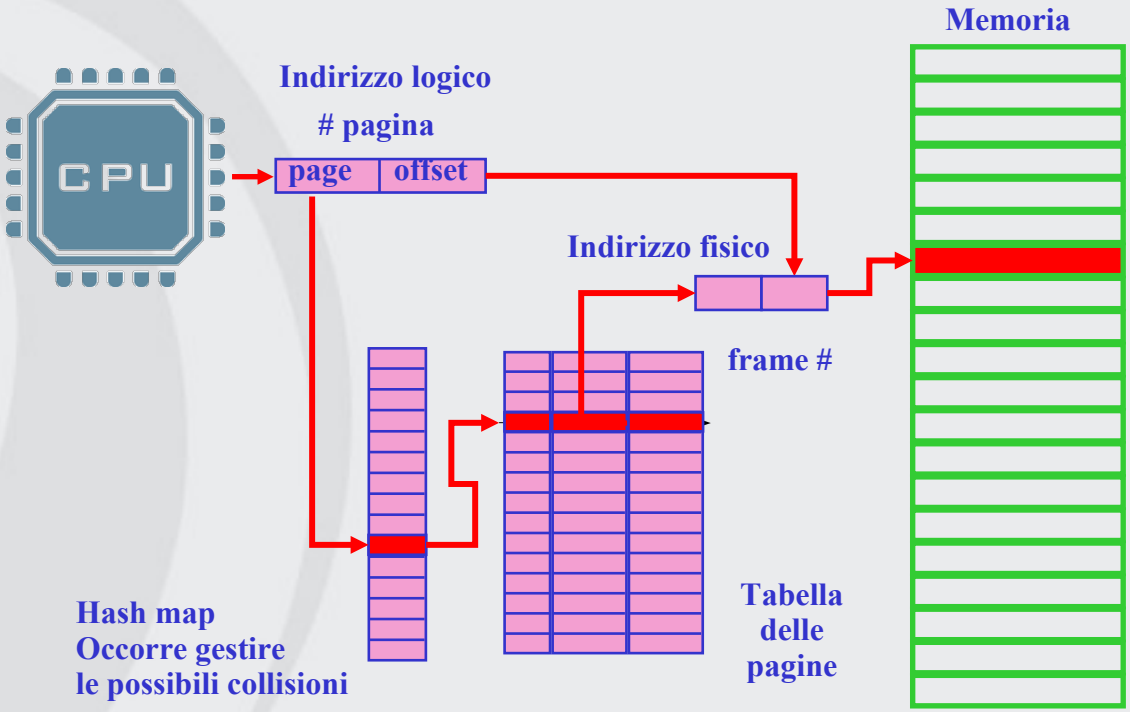
- Ogni elemento nella memoria associativa contiene informazioni su una pagina, in particolare:
  - il numero di pagina virtuale
  - un bit di modifica della pagina
  - il codice di protezione (permessi di lettura/scrittura/esecuzione della pagina)
  - il numero del frame fisico in cui la pagina è situata
  - un bit che indica se l'elemento è valido o meno
- Questi campi hanno una corrispondenza uno a uno con i campi nella tabella delle pagine

Tabelle delle pagine invertite



- Le tabelle descritte richiedono un elemento per ogni pagina virtuale:
  - con uno spazio di indizzamento di 2^{32} e pagine di 4K sono necessari 2^{20} elementi (almeno 4M di dati, per ogni processo), con 64 bit di indirizzamento la tabella raggiunge la dimensione di milioni di Giga
- La soluzione è la **tabella delle pagine inversa**. In questo caso la tabella contiene un elemento solo per ogni pagina effettivamente in memoria, ogni elemento allora contiene la coppia (processo, pagina virtuale)
  - Il problema è la traduzione degli indirizzi virtuali, occorre infatti consultare l'intera tabella, non un singolo elemento:
    la soluzione è data dall'uso della memoria associativa, ed eventualmente sfruttare metodi di codifica hash nel caso in cui la pagina cercata non sia nella memoria associativa

Tabelle delle pagine invertite



Se la pagina non è in memoria, occorre utilizzare una tabella tradizionale, talvolta tenuta su disco

Gestione della memoria

Algoritmi di rimpiazzamento

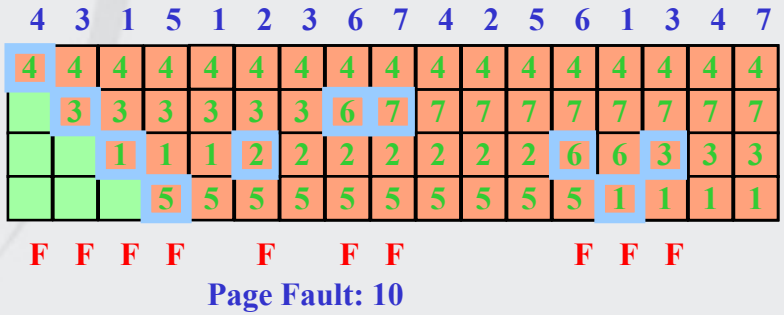
Algoritmi di rimpiazzamento

Per ogni page fault il SO deve:

- scegliere una pagina da rimuovere dalla memoria per creare spazio per la nuova pagina
  - se la pagina da rimuovere è stata modificata, la copia su disco deve essere aggiornata
  - se non c'è stata alcuna modifica, la nuova pagina sovrascrive quella da rimuovere
- Scegliendo la pagina da rimpiazzare con un algoritmo per sostituire pagine poco utilizzate (e non a caso) può portare a un miglioramento delle prestazioni

Algoritmo ottimo

- Ad ogni pagina viene assegnata un'etichetta corrispondente al numero di istruzioni che dovranno essere eseguite prima che la pagina venga referenziata
- L'algoritmo ottimo consiste nel rimuovere la pagina con l'etichetta maggiore



Algoritmo ottimo

- **Non è realizzabile** in quanto non è possibile conoscere a priori (quindi al momento del page fault) quali pagine verranno referenziate
  - Sarebbe necessario prevedere il futuro

Algoritmo ottimo

- Potrebbe essere utile eseguire il programma con un simulatore e tenere traccia dei riferimenti alle pagine
  - L'algoritmo ottimo può essere utilizzato per una seconda esecuzione utilizzando le informazioni sui riferimenti alle pagine della prima esecuzione
  - È utile come confronto per la valutazione degli algoritmi di rimpiazzamento pagine

Algoritmo FIFO

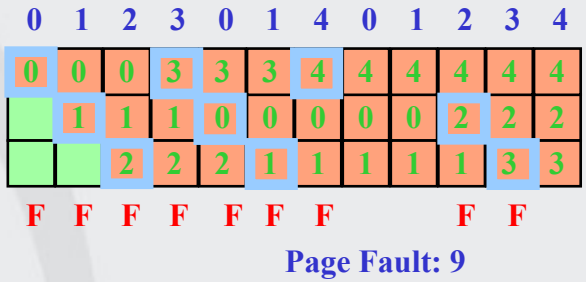
- Il SO mantiene una lista concatenata di tutte le pagine in memoria con:
  - la pagina più vecchia in testa
  - la pagina più recente in coda
- Quando avviene un page fault viene rimossa la pagina in testa alla lista e la nuova pagina viene aggiunta in coda alla lista
- L'algoritmo FIFO in questa forma viene utilizzato raramente infatti non è detto che la pagina più vecchia sia effettivamente la meno referenziata

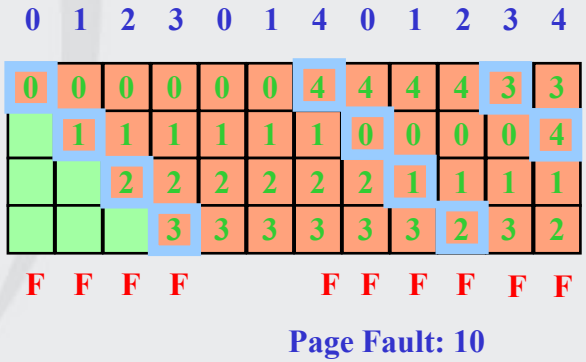
Algoritmo FIFO

Page Fault: 12

Page Fault: 10

Anomalia di Belady





Algoritmo NRU

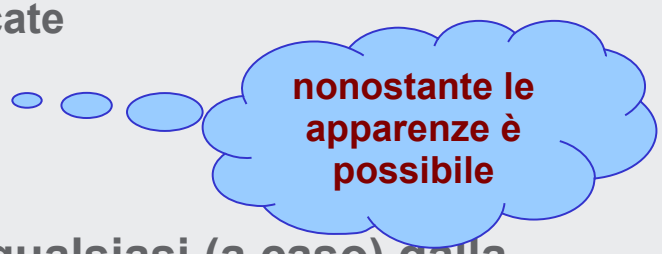
- La maggior parte dei computer con memoria virtuale prevede 2 bit per la raccolta di informazioni sull'utilizzo delle pagine:
  - il bit R: indica che la pagina è stata referenziata (letta o scritta)
  - il bit M: indica che la pagina è stata modificata (scritta)

Algoritmo NRU (Not Recently Used)

- Inizialmente i bit R e M vengono impostati a 0 dal SO
- Periodicamente (ad esempio ad ogni interrupt del clock), il bit R viene azzerato per distinguere le pagine non referenziate recentemente dalle altre

Algoritmo NRU

- Quando avviene un page fault, il SO controlla tutte le pagine e le divide in quattro classi in base al valore corrente dei bit R e M
  - Classe 0: non referenziate, non modificate
  - Classe 1: non referenziate, modificate
  - Classe 2: referenziate, non modificate
  - Classe 3: referenziate, modificate



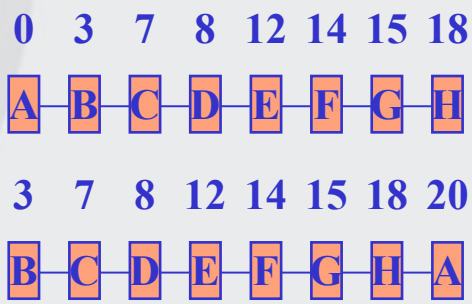
- L'algoritmo NRU rimuove una pagina qualsiasi (a caso) dalla classe di numero inferiore che sia non vuota
- Caratteristiche di NRU
  - è facile da implementare
  - ha prestazioni che, anche se non ottimali, sono spesso adeguate

Algoritmo Second Chance

- Con una semplice modifica dell'algoritmo FIFO è possibile evitare il problema della rimozione di pagine molto utilizzate
- Viene controllato il bit R della pagina più vecchia:
  - se vale 0, la pagina è sia vecchia che non utilizzata e viene rimpiazzata immediatamente
  - se vale 1, il bit viene azzerato e la pagina viene messa in coda alla lista come se fosse appena stata caricata in memoria

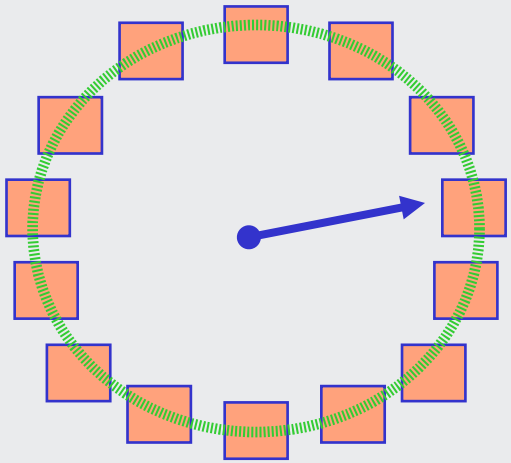
Algoritmo Second Chance

- L'algoritmo si basa sulla ricerca di una pagina che non sia stata referenziata nel precedente intervallo di clock
  Se tutte le pagine sono state referenziate degenera nell'algoritmo FIFO puro



Algoritmo clock

- L'algoritmo second chance è inefficiente perché sposta costantemente le pagine sulla lista
- Un approccio migliore consiste nel tenere tutte le pagine su una lista circolare nella forma di un orologio con una lancetta che punta alla pagina più vecchia



Algoritmo clock

- Quando avviene un page fault:
  - se il bit R della pagina vale 0, la pagina puntata dalla lancetta viene rimossa, la nuova pagina viene inserita nell'orologio al suo posto e la lancetta viene spostata in avanti di una posizione
  - se il bit R della pagina vale 1, viene azzerato e la lancetta viene spostata in avanti di una posizione
- Questo processo viene ripetuto finché non viene trovata una pagina con R = 0
- Differisce dall'algoritmo second chance solo per l'implementazione

Algoritmo LRU

- È una buona approssimazione all'algoritmo ottimo e si basa sulla seguente osservazione:
- le pagine molto utilizzate nelle ultime istruzioni saranno probabilmente molto utilizzate nelle successive istruzioni così come le pagine poco utilizzate recentemente continueranno a non essere utilizzate per molto tempo

Algoritmo LRU (Least Recently Used)

- Quando avviene un page fault viene rimpiazzata la pagina non utilizzata da più tempo

Algoritmo LRU

- È un algoritmo costoso poiché deve essere mantenuta una lista concatenata di tutte le pagine in memoria (ordinata in base al tempo trascorso dall'ultimo utilizzo), questa lista deve essere aggiornata ad ogni riferimento in memoria
- Eseguire, ad ogni istruzione, operazioni di ricerca e manipolazione in una lista concatenata è molto costoso in termini di tempo
- Ci sono soluzioni per l'implementazione di LRU con hardware speciale

Algoritmo LRU

LRU: implementazione H/W

- Sequenza di pagine: 0 1 2 3 2 1 0 3 2 3

```
0 0111 0011 0001 0000 0000
1 0000 1011 1001 1000 1000
2 0000 0000 1101 1100 1101
3 0000 0000 0000 1110 1100

0 0000 0111 0110 0100 0100
1 1011 0011 0010 0000 0000
2 1001 0001 0000 1101 1100
3 1000 0000 1110 1100 1110
```

  - Ad ogni accesso viene messa ad **uno la riga** corrispondente alla pagina
  - poi viene messa a **zero la colonna** corrispondente

Algoritmo NFU

- Una approssimazione dell'algoritmo LRU è

Algoritmo NFU (Not Frequently Used)

- Ad ogni pagina viene associato un contatore inizialmente posto a 0
- Ad ogni clock viene sommato al contatore il bit R
- Al momento di un page fault viene rimpiazzata la pagina con contatore minimo

Algoritmo NFU

- Il problema è che NFU non ***dimentica*** nulla:
  - se una pagina è stata molto utilizzata non verrà più rimossa anche se non più utile
- È stata proposta una versione modificata con **invecchiamento (aging)**:
  - ad ogni clock il contatore scorre a destra introducendo R come bit più significativo

Bit R	101011	110010	110101	100010	011000
0	10000000	11000000	11100000	11110000	01111000
1	00000000	10000000	11000000	01100000	10110000
2	10000000	01000000	00100000	00010000	10001000
3	00000000	00000000	10000000	01000000	00100000
4	10000000	11000000	01100000	10110000	01011000
5	10000000	01000000	10100000	01010000	00101000
	t=1	t=2	t=3	t=4	t=5

Modello a Working Set

- Nel modello più semplice di paginazione le pagine vengono caricate in memoria solo al momento del page fault, si parla di paginazione su richiesta (on demand)
- La maggior parte dei processi esibiscono la località di riferimenti cioè durante qualsiasi fase dell'esecuzione il processo fa riferimenti ad un piccolo insieme di pagine
- L'insieme di pagine che un processo sta correntemente utilizzando viene detto working set
- Più è alto il numero pagine in memoria più è probabile che il working set sia completamente in memoria (e quindi non avverranno page fault)
- Un programma che causa page fault per ogni piccolo insieme di istruzioni viene detto in thrashing

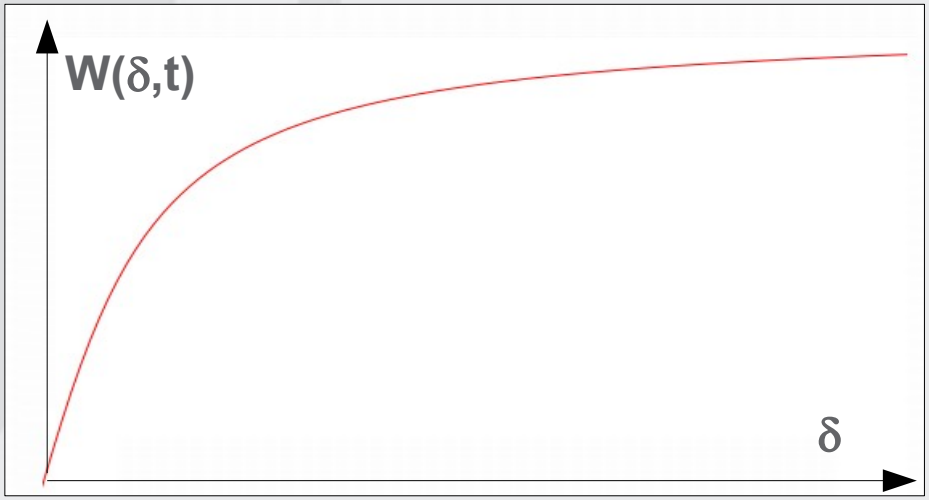
Working Set

- Il WS con parametro δ al tempo t, $W(\delta,t)$ è l'insieme delle pagine a cui ci si è riferiti nelle ultime δ unità di tempo
- Proprietà:

$$W(\delta,t) \subseteq W(\delta+1,t)$$

$$1 \leq |W(\delta,t)| \leq \min(\delta,N)$$

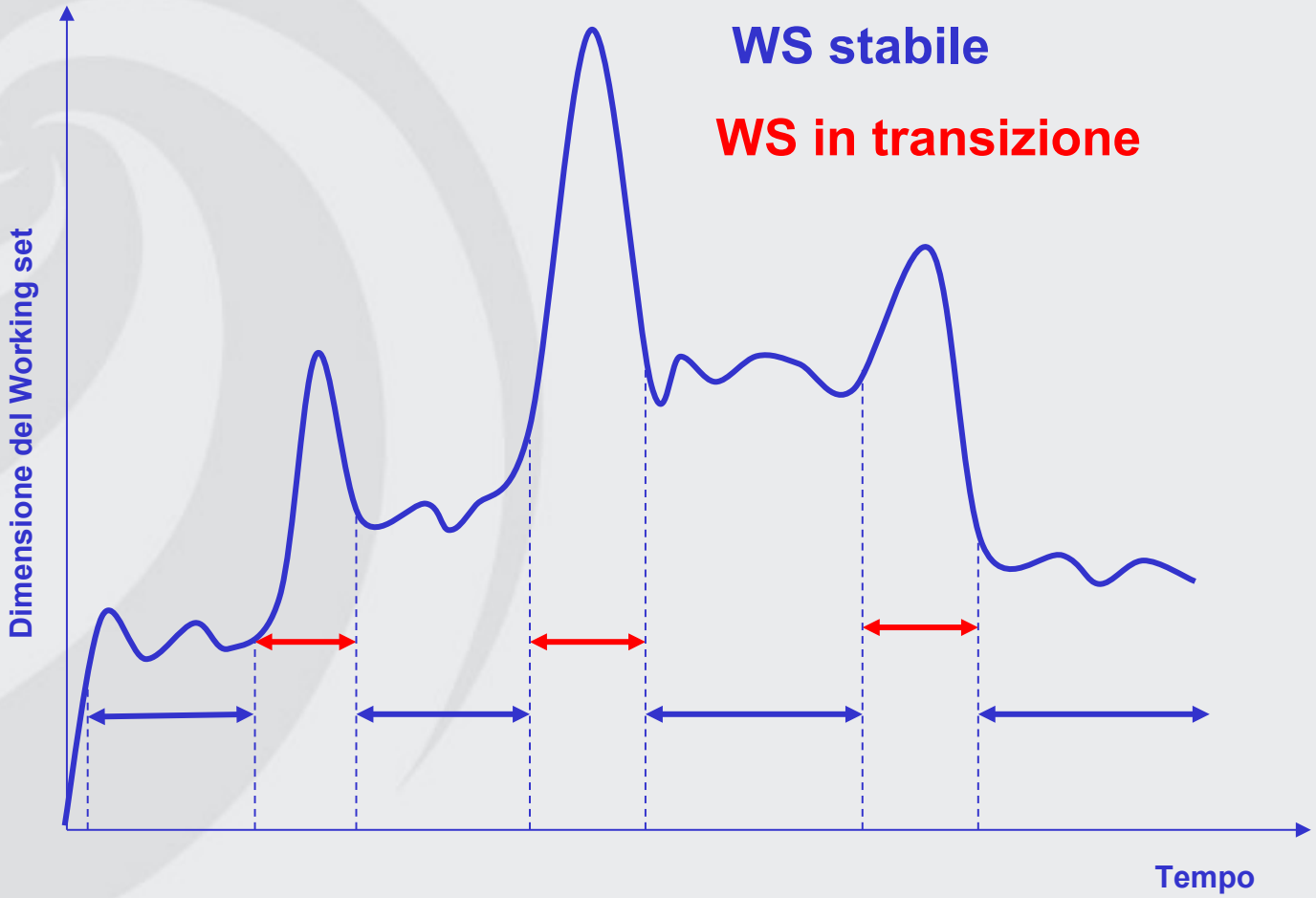
N numero di pagine necessarie al processo



Working set

Sequenza di riferimenti		Dimensione della finestra (2÷5)			
24		24	24	24	24
15		24 15	24 15	24 15	24 15
18		15 18	24 15 18	24 15 18	24 15 18
23		18 23	15 18 23	24 15 18 23	24 15 18 23
24		23 24	18 23 24	-	-
17		24 17	23 24 17	18 23 24 17	15 18 23 24 17
18		17 18	24 17 18	-	18 23 24 17
24		18 24	-	24 17 18	-
18		-	18 24	-	24 17 18
17		18 17	24 18 17	-	-
17		17	18 17	-	-
15		17 15	17 15	18 17 15	24 18 17 15
24		15 24	17 15 24	17 15 24	-
17		24 17	-	-	17 15 24
24		-	24 17	-	-
18		24 18	17 24 18	24 17 18	15 24 17 18

Working set


WS stabile
WS in transizione
Dimensione del Working set
Tempo

Prestazioni e page fault

Qual è la perdita di prestazioni per un tasso di page fault di 1 su 100000?

- access time senza page fault: 100 ns
- tempo di accesso a memoria
  0.5 * 20 ms + 0.5 * 40 ms = 30 ms
  con un rimpiazzamento di pagine del 50%

Effective Access Time (p = probabilità di page fault)

EAT =

p * accesso con page fault +
(1-p) accesso senza page fault

EAT =

0.00001 * 30 ms + (1-0.00001) * 100 ns =
300 ns + 99.999 ns = 400 ns

rappresenta una perdita di prestazioni del 300%

Prestazioni e page fault

Quale è la percentuale di fault per un 10% di perdita di prestazioni (EAT=110 ns)?

110 ns

> p * 30 ms + (1-p) * 100 ns
> p * 30,000,000 ns + 100ns

10 ns > p * 30,000,000 ns

p < 10/30,000,000 (1 su 3 milioni)

Località

- Per località si indica che il prossimo accesso di memoria sarà nelle vicinanze di quello corrente
  - località spaziale: il prossimo indirizzo differirà di poco
  - località temporale: la stessa cella (o pagina) sarà molto probabilmente riutilizzata nel futuro prossimo
- La località è la ragione che giustifica un basso numero di page fault (altrimenti non spiegabile)

Località

```
int mat[1024][1024];

max=mat[0][0];
for(i=0; i<1024; i++)
  for(j=0; j<1024; j++)
    if(max<mat[i][j])
      max = mat[i][j];
```

Località temporale
continuo ad usare
le variabili i, j

Località spaziale
dopo mat[10][101]
uso mat[10][102]

Località

```
int mat[1024][1024];

max=mat[0][0];
for(i=0; i<1024; i++)
  for(j=0; j<1024; j++)
    if(max<mat[j][i])
      max = mat[j][i];
```

- Questa seconda soluzione potrebbe causare un numero notevole di page fault (**perché?**)

Strategia di fetch

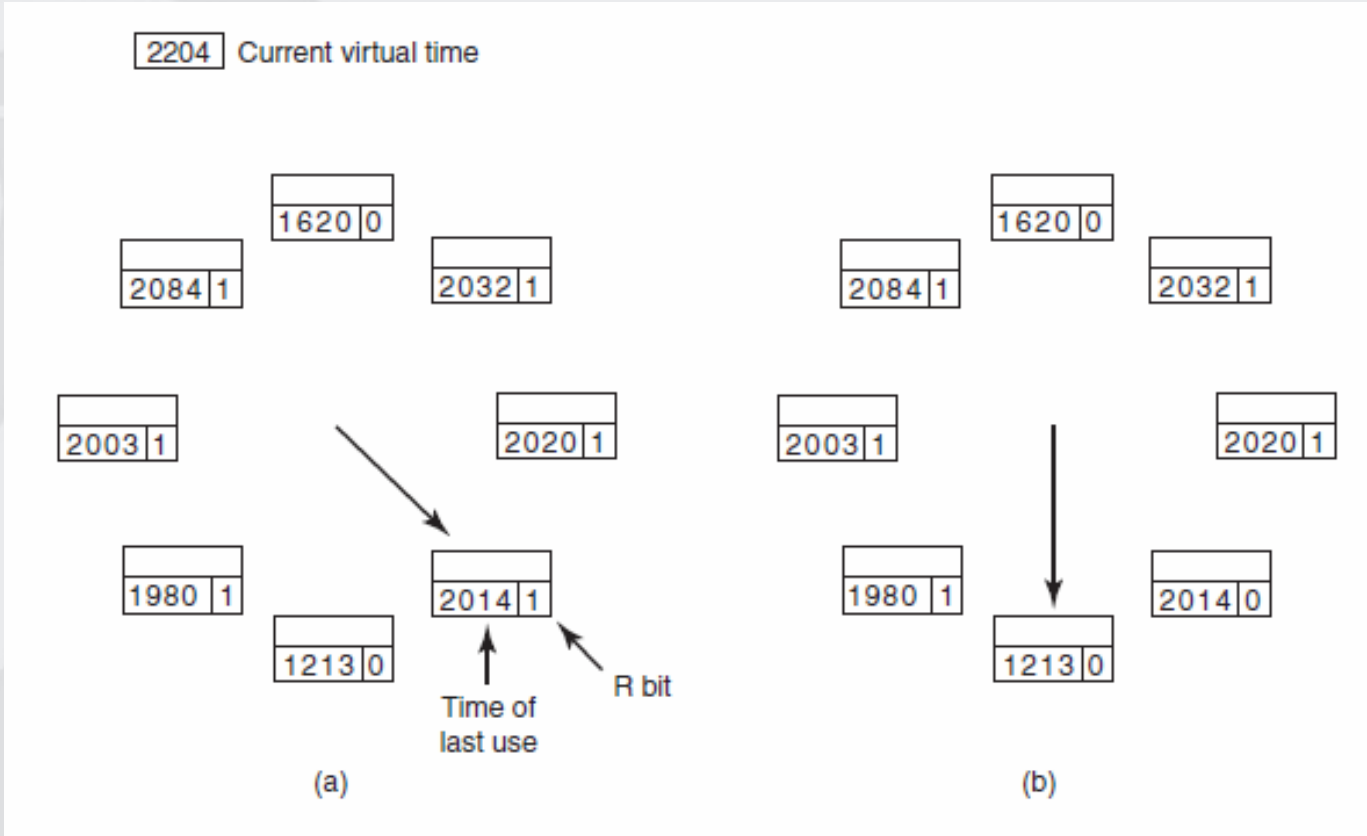
Stabilisce quando una pagina deve essere trasferita in memoria

- La paginazione su richiesta (demand paging) porta una pagina in memoria solo quando si ha un riferimento a quella
  - Si hanno molti page fault quando un processo parte
- Prepaging porta in memoria più pagine del necessario
  - È più efficiente se le pagine su disco sono contigue

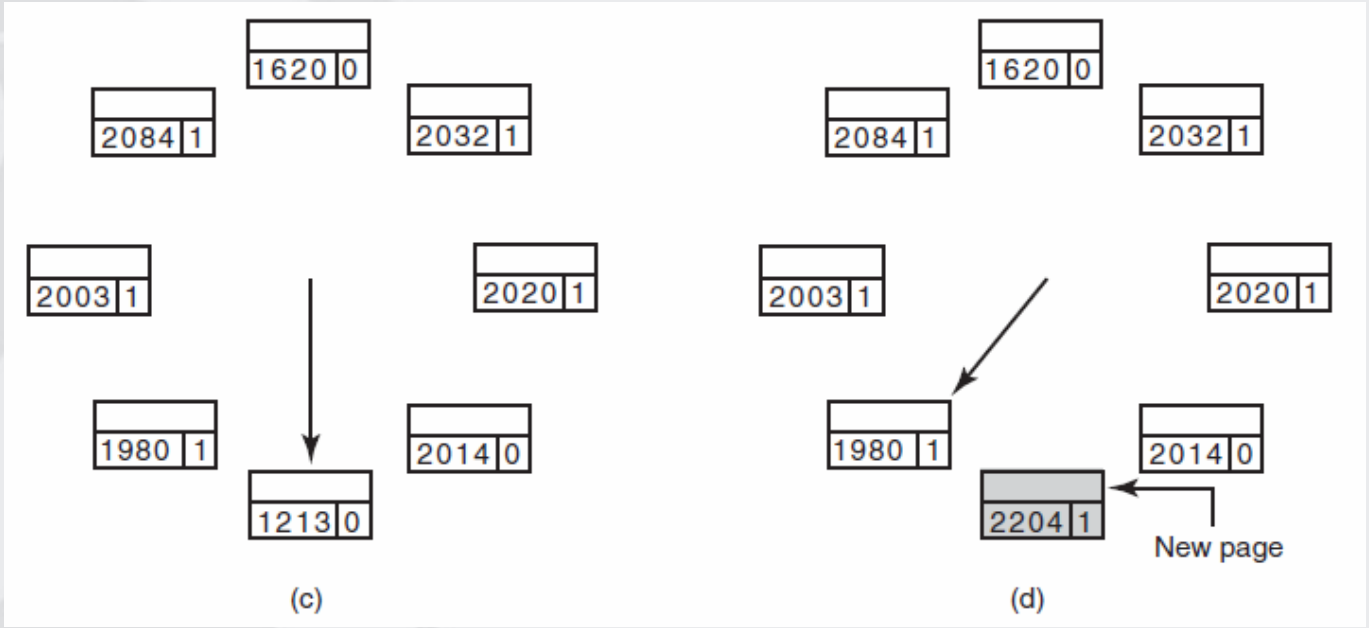
Prepaging

- Per risolvere il problema del thrashing molti sistemi a paginazione utilizzano il prepaging prima di eseguire un processo, vengono caricate in memoria le pagine del working set relative al processo
- Per implementare il modello Working Set è necessario che il Sistema Operativo tenga traccia di quali pagine sono nel working set
  Un modo per controllare questa informazione è quello di utilizzare l'algoritmo di invecchiamento
- Le informazioni sul working set possono essere utilizzate per migliorare le prestazioni dell'algoritmo clock (questo nuovo algoritmo viene detto wsclock): la pagina viene sostituita solo se non appartiene al working set

WSClock - R=1


2204 Current virtual time
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 1
1213 0
Time of last use
R bit
(a)
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 0
1213 0
(b)

WSClock - R=0


1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 0
1213 0
(c)
1620 0
2084 1
2032 1
2003 1
2020 1
1980 1
2014 0
2204 1
New page
(d)

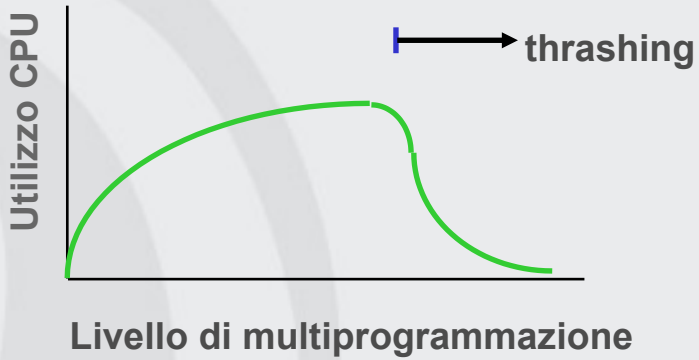
Gestione della memoria

Criteri di progetto della paginazione

Algoritmi Locali e Globali

- Come dovrebbe essere allocata la memoria tra i processi eseguibili e concorrenti?
- Gli algoritmi di rimpiazzamento di pagine che usano strategie **locali** assegnano a ogni processo una quantità fissa di memoria
- Gli algoritmi **globali** allocano dinamicamente i frame tra i processi eseguibili
- In generale gli algoritmi globali danno risultati migliori, in particolare quando la dimensione del working set varia durante la ***vita*** di un processo, tuttavia il sistema deve continuamente decidere quanti frame assegnare a ogni processo

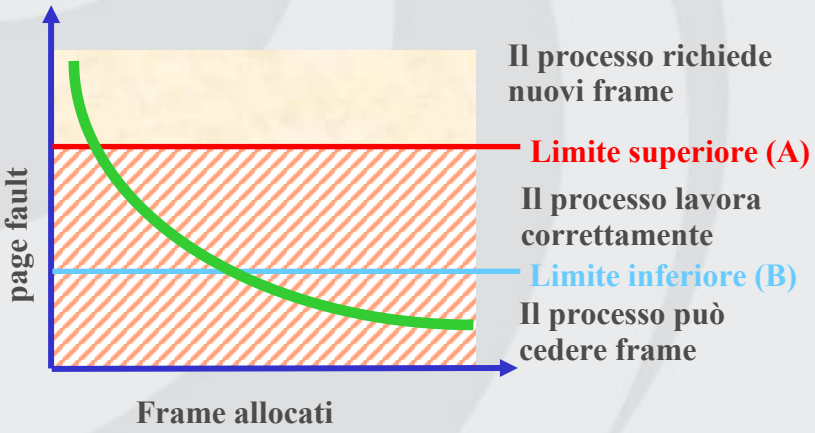
Thrashing



Spesso il thrashing è causato da un numero eccessivo di processi in memoria

Page Fault Frequency

- Utilizzando la gran parte degli algoritmi la frequenza di fault decresce all'aumentare del numero di pagine assegnate



L'algoritmo Page Fault Frequency cerca di mantenere la frequenza degli errori in un intervallo ragionevole:

- Se il numero di processi è troppo grande per mantenerli tutti sotto **A** qualche processo viene rimosso
- Se un processo è sotto **B** allora ha troppa memoria e parte delle sue pagine possono essere utilizzate da altri processi

Paging daemon

- È un processo in background che viene risvegliato periodicamente per ispezionare lo stato della memoria
- Se il numero di frame liberi in memoria non è sufficiente si occupa di selezionare, tramite l'algoritmo di rimpiazzamento pagine prescelto, le pagine da eliminare (se queste sono state modificate, vengono riscritte su disco)
- Vantaggio: migliori prestazioni rispetto alla ricerca di un frame libero nel momento in cui serve

Dimensione delle pagine

- La dimensione delle pagine è un parametro del sistema, le motivazioni nella scelta rispondono a esigenze contrastanti:
- A favore di pagine ***piccole***:
  - Un blocco di memoria non riempirà esattamente un numero intero di pagine, mediamente quindi metà dell'ultima pagina viene *sprecata*
- A favore di pagine ***grandi***:
  - con pagine piccole è necessaria una tabella delle pagine grande
  - il trasferimento di una pagina piccola da disco richiede quasi lo stesso tempo di una pagina grande

Esempio

- **s** dimensione media processo (128K)
- **p** dimensione pagine
- **e** dimensione di un elemento nella tabella delle pagine (8 byte)

- **s/p** numero di pagine in memoria e quindi se/p dimensione tabella
- **p/2** memoria sprecata

il costo allora è dato da `costo = se/p + p/2`

derivando rispetto a p

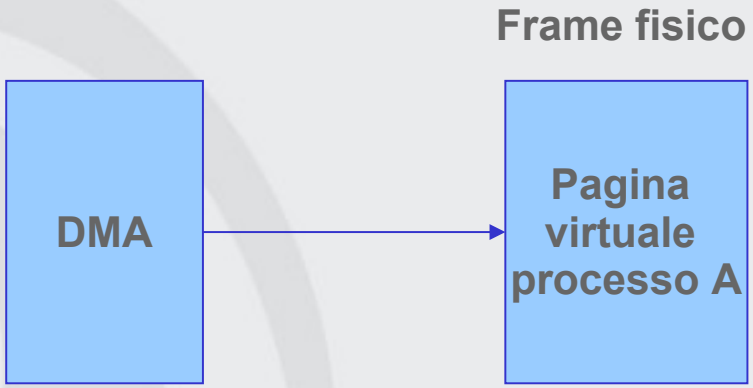
$-se/p^2 + 1/2 = 0$ da cui `p = sqrt(2se)` (1448 byte in pratica 1 o 2 K)

La maggior parte dei SO usa dimensioni di pagina da 512 byte a 64K

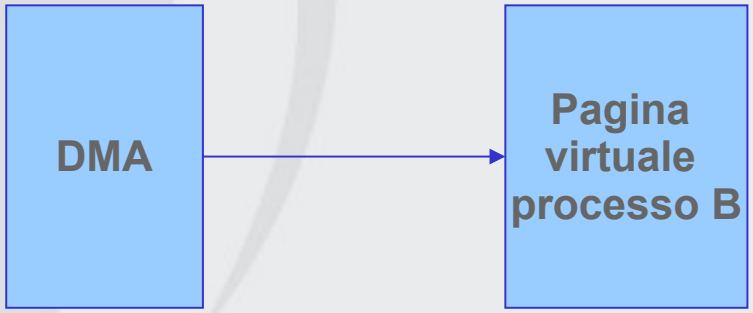
Interfaccia della memoria virtuale

- In alcuni sistemi avanzati i programmatori hanno un certo controllo sulla mappa di memoria
- Un motivo per consentire ai programmatori il controllo sulla mappa di memoria è di consentire a due o più programmi di condividere la stessa memoria: se è possibile dare un nome ad una zona di memoria, un processo può darlo ad un altro processo in modo che quest'ultimo possa inserire la pagina nella sua tabella
- La condivisione delle pagine può implementare un sistema a scambio di messaggi ad elevate prestazioni

Blocco di pagine in memoria

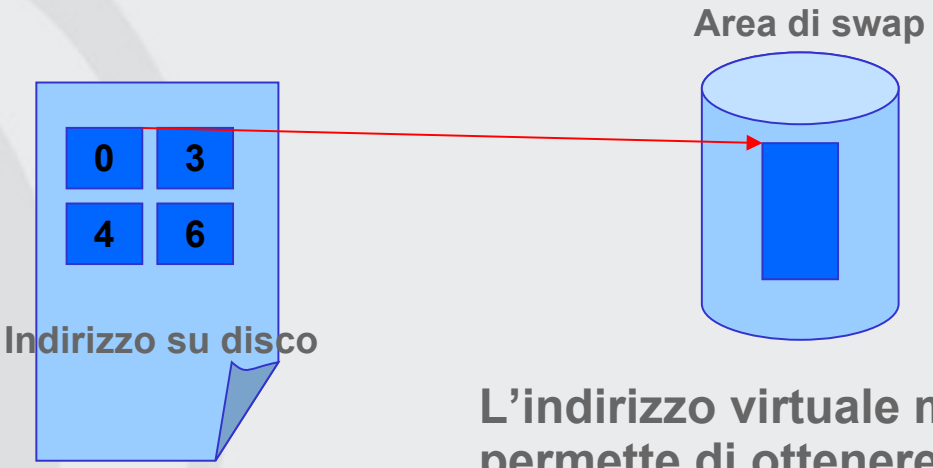


L'algoritmo globale toglie il frame ad A e lo assegna a B



Il DMA va a scrivere i dati nello spazio di B, devo bloccare la pagina in memoria (pinning)

Memoria secondaria

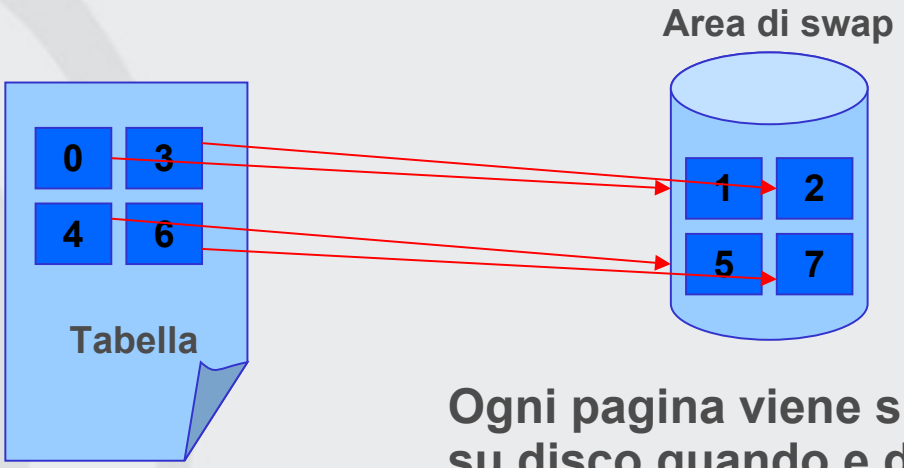


L'indirizzo virtuale mi permette di ottenere l'indirizzo su disco

Due approcci:

- Tutto il processo viene caricato nell'area di swap
- Tutto il processo viene caricato in memoria

Memoria secondaria



Ogni pagina viene spostata su disco quando e dove necessario

Non occorre duplicare codice, file mappati in memoria, …

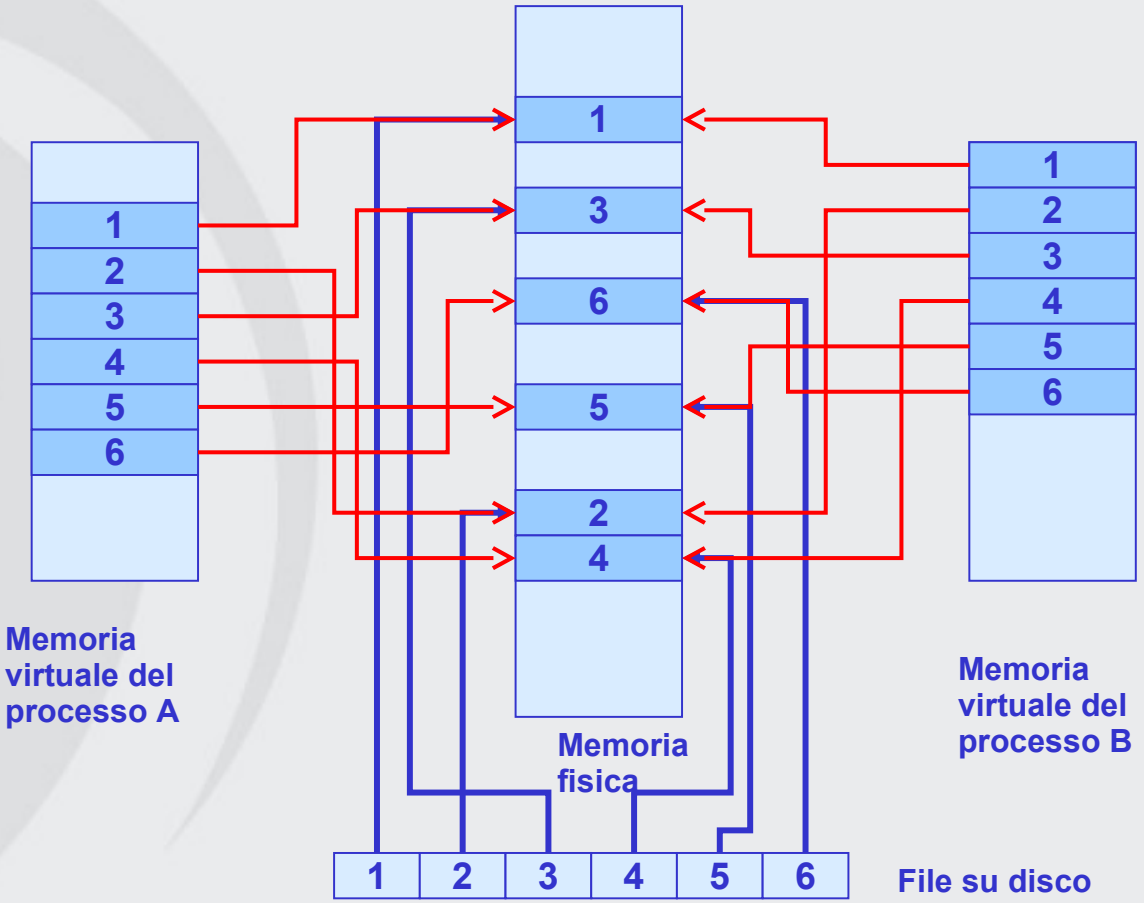
Memory-Mapped File

- Memory-mapped file I/O consente che le operazioni di I/O possano essere trattate come un normale accesso in memoria *mappando* i blocchi su disco su pagine in memoria

- Quando si richiede l'uso di un blocco su disco, questo viene caricato in memoria
  - L'accesso risulta semplificato in quanto non si usano più le chiamate di sistema read() write()

- Più processi possono condividere in memoria gli stessi file

Memory Mapped Files



Memory Mapped Files

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

int main(int argc, char *argv[])
{
   char *pt, pt1[1024];
   int fd, len=sizeof(pt1), offset=0;

   fd = open(__FILE__, O_RDONLY);
   pt = mmap(NULL, len, PROT_READ,MAP_SHARED,fd, offset);

   memcpy(pt1, pt, len);

   return 0;
}
```

Memory Mapped Files

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

   // file aperto in lettura e scrittura
   fd = open(TEST_FILE, O_RDWR);
   if(fd<0) … // problemi con il file

   // memoria condivisa in lettura e scrittura
   pt = mmap(NULL, len, PROT_READ | PROT_WRITE,
      MAP_SHARED,fd, offset);

   if(pt == MAP_FAILED) …  // problemi di mappatura

// modalità (quasi) analoghe con Windows
```

Gestione della memoria

Segmentazione

Segmentazione

- Con l'approccio precedente la memoria virtuale è unidimensionale (gli indirizzi vanno da 0 all'indirizzo massimo): può essere utile mantenere due o più spazi di indirizzamento separati

Esempio

Un compilatore ha molte tabelle che vengono costruite durante la compilazione e che crescono durante la compilazione

- In uno spazio di indirizzamento unidimensionale ci possono essere tabelle con molto spazio libero e altre completamente occupate o addirittura che necessitano altro spazio
- Ci sono alcune possibili soluzioni ma ciò che effettivamente serve è liberare l'utente dal compito di gestire tabelle che si espandono e contraggono

Segmentazione

- La soluzione diretta e generale a problemi di questo tipo è la segmentazione
- Ogni segmento è una sequenza lineare di indirizzi da 0 a un massimo
  - la lunghezza di ogni segmento può variare da 0 a un massimo consentito e può variare durante l'esecuzione
  - segmenti diversi possono avere lunghezze diverse

Segmentazione

- Per specificare un indirizzo in una memoria segmentata (o bidimensionale), il programma deve fornire un indirizzo costituito da due parti:
  1. numero di segmento
  2. indirizzo all'interno del segmento
- un segmento può contenere una procedura, un array, uno stack o un insieme di variabili ma solitamente non contiene un insieme misto di questi
- Un segmento è un'entità logica della quale il programmatore è cosciente e utilizza come una singola entità logica

Vantaggi della segmentazione

- Semplifica la gestione di strutture dati che crescono o diminuiscono
- Se ogni procedura occupa un segmento separato, con indirizzo di inizio all'interno del segmento pari a 0, la segmentazione semplifica il linking di procedure compilate separatamente
- Facilita la condivisione di procedure o dati tra alcuni processi
- Dato che ogni segmento è un'entità logica nota al programmatore (procedura, array o stack), segmenti diversi possono avere diversi tipi di protezione

Confronto fra segmentazione e paginazione

Considerazioni	Paginazione	Segmentazione
Il programma è a conoscenza del metodo?	No	Sì
Quanti spazi di indirizzamento ci sono?	1	Molti
È possibile che lo spazio di indirizzamento superi la dimensione fisica della memoria?	Sì	Sì
È possibile distinguere e proteggere separatamente procedure e dati?	No	Sì
È facile gestire tabelle di dimensioni variabili?	No	Sì
Facilita la condivisione di procedure fra gli utenti?	No	Sì
Perché è stato proposto questo metodo?	Per avere uno spazio di indirizzamento di grandi dimensioni	Per poter distinguere codice e dati in spazi di indirizzamento logicamente separati e per facilitare la condivisione e la protezione

Implementazione della segmentazione pura

- L'implementazione di paginazione e segmentazione ha una differenza fondamentale: le pagine hanno dimensione fissa, i segmenti variabile
- Se segmenti vengono continuamente caricati e sostituiti in memoria, dopo un certo tempo si formeranno in memoria delle zone non utilizzate:
  è il fenomeno del **checkerboarding** (frammentazione esterna)
- Una possibile soluzione è la compattazione

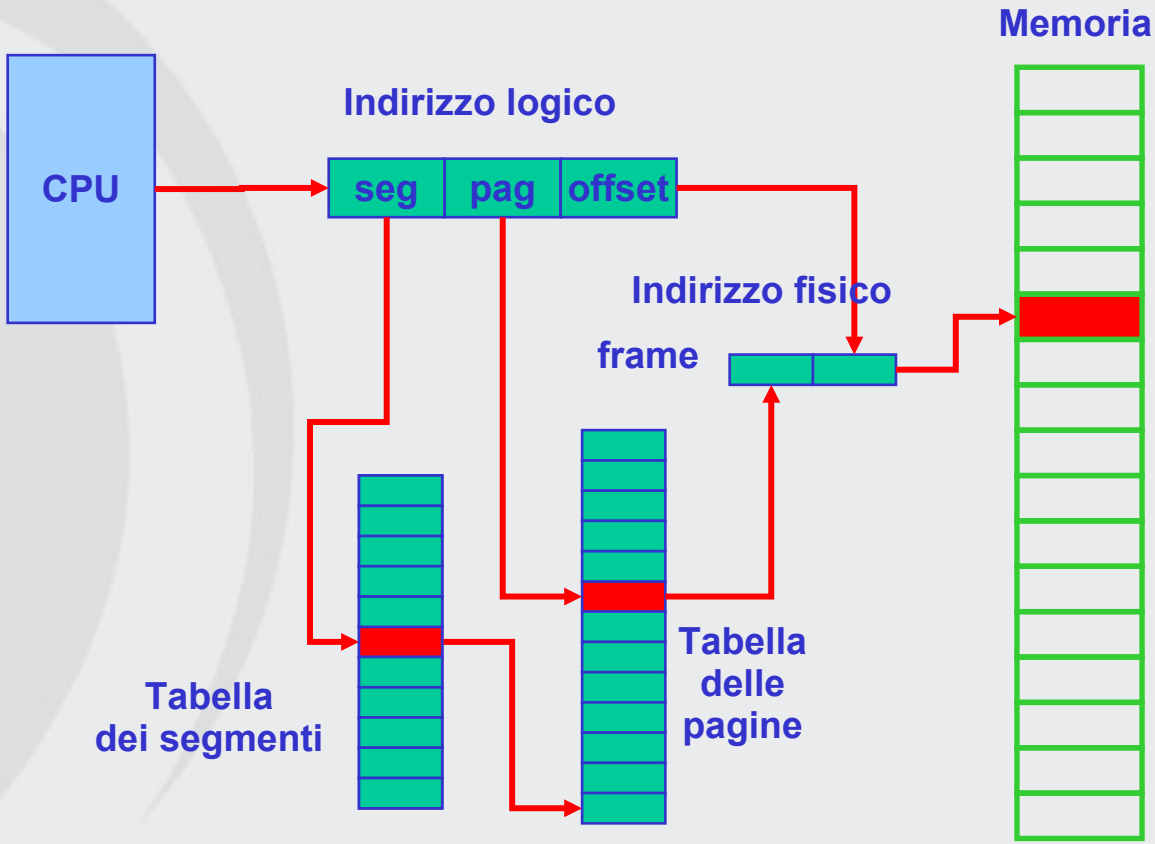
Segmentazione con paginazione

- La tecnica è stata proposta per sfruttare contemporaneamente i vantaggi della paginazione e della segmentazione
- Un indirizzo virtuale è costituito da tre parti (esempio MULTICS)

Indice di segmento	Numero di pagina	Offset interno alla pagina
18 bit	6 bit	10 bit

I processori INTEL a 32 bit implementano un meccanismo simile

Segmentazione con paginazione



CVMLab
Computer Vision
& Multimedia Lab

Esercizi

Università
degli Studi
di Pavia

Memoria a partizioni fisse

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

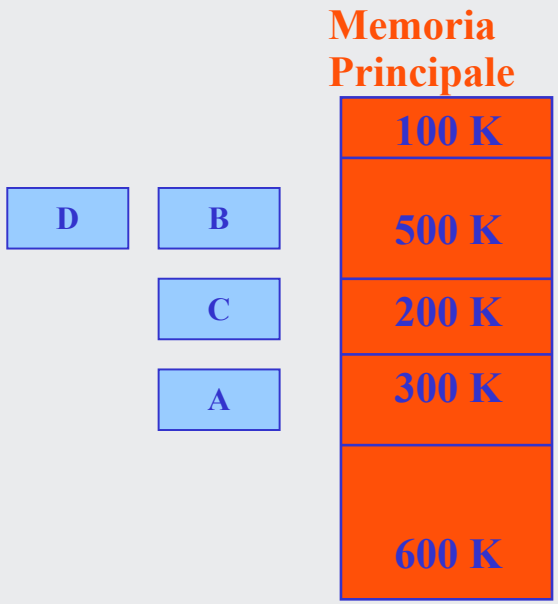
Memoria Principale

Memoria a partizioni fisse

Code separate

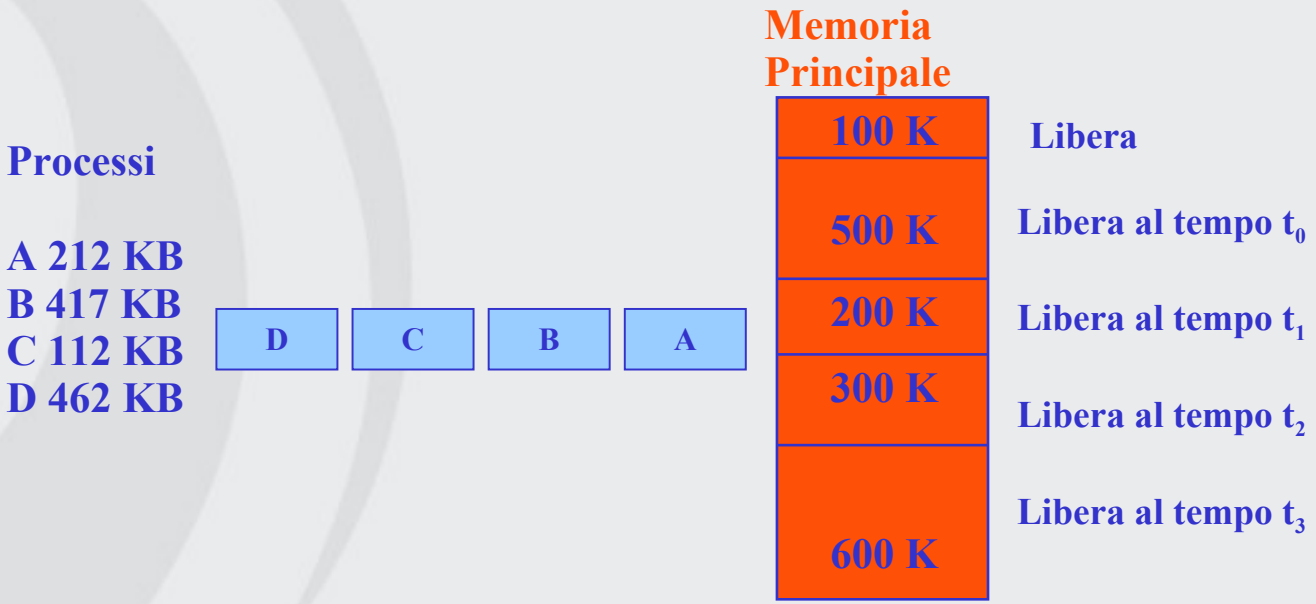
Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB



Memoria a partizioni fisse

Coda unica



$t_0 < t_1 < t_2 < t_3$

Memoria a partizioni fisse

Coda unica – first fit
Quando si libera una partizione scelgo il primo processo in coda

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

Memoria Principale

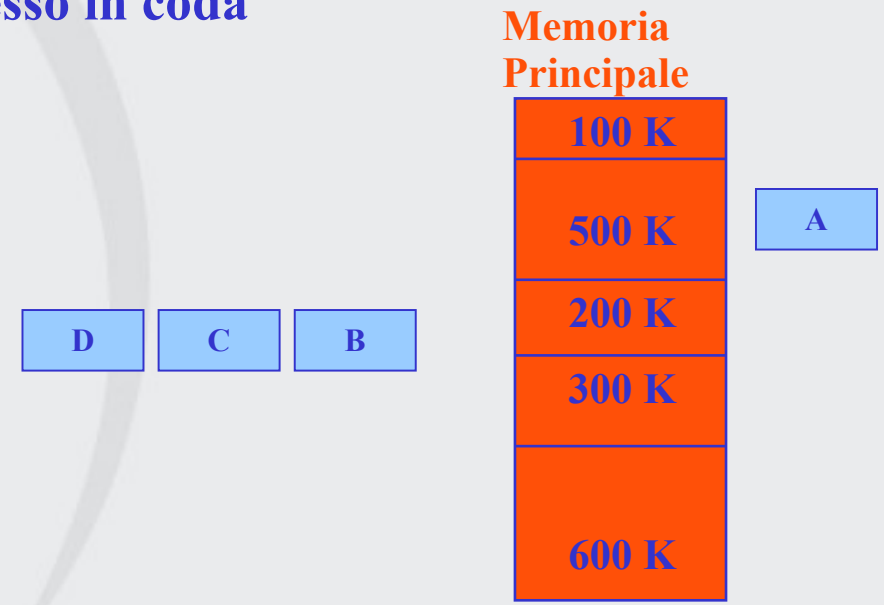
La partizione non è utilizzabile

Memoria a partizioni fisse

Coda unica – first fit
Quando si libera una partizione scelgo il primo processo in coda

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

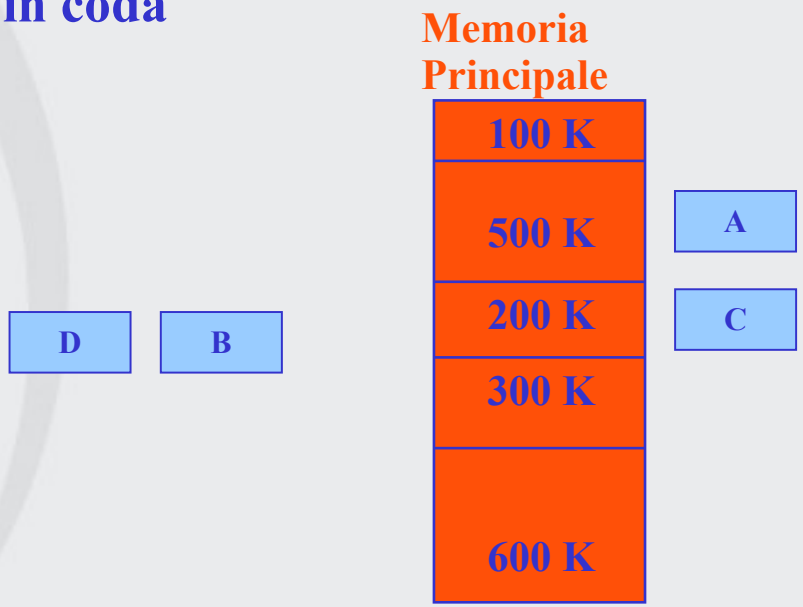


Memoria a partizioni fisse

Coda unica – first fit
Quando si libera una partizione scelgo il primo processo in coda

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

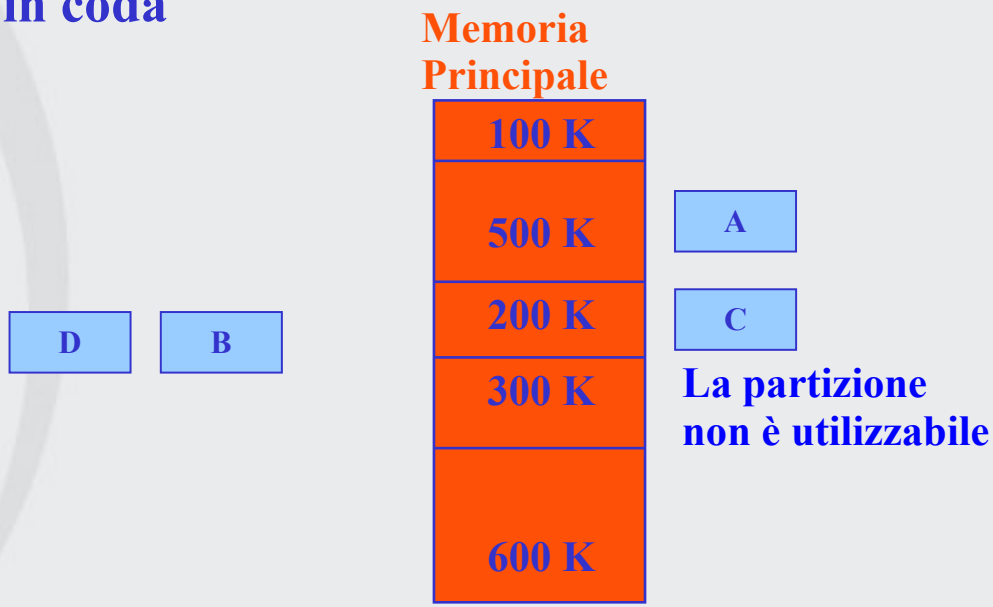


Memoria a partizioni fisse

Coda unica – first fit
Quando si libera una partizione scelgo il primo processo in coda

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

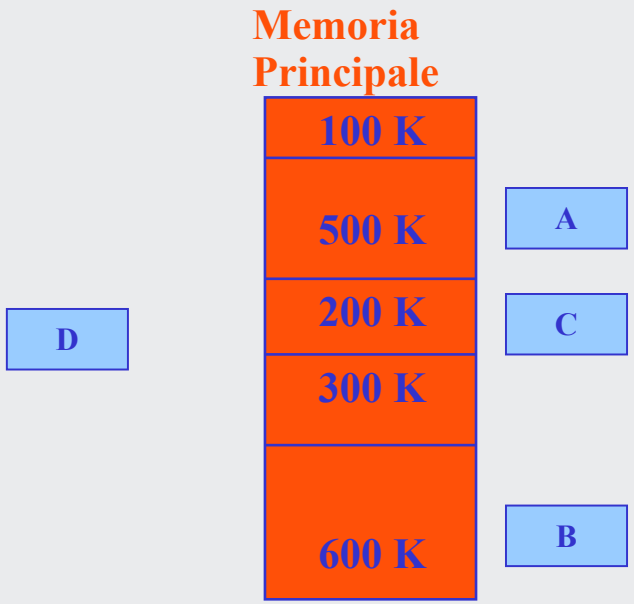


Memoria a partizioni fisse

Coda unica – first fit
Quando si libera una partizione scelgo il primo processo in coda

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

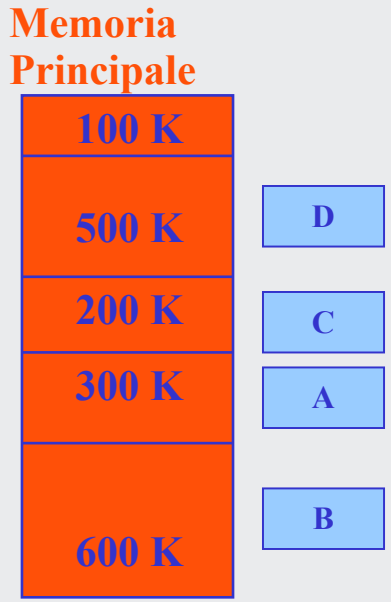


Memoria a partizioni fisse

Coda unica – best fit
Quando si libera una partizione scelgo il processo che meglio si adatta

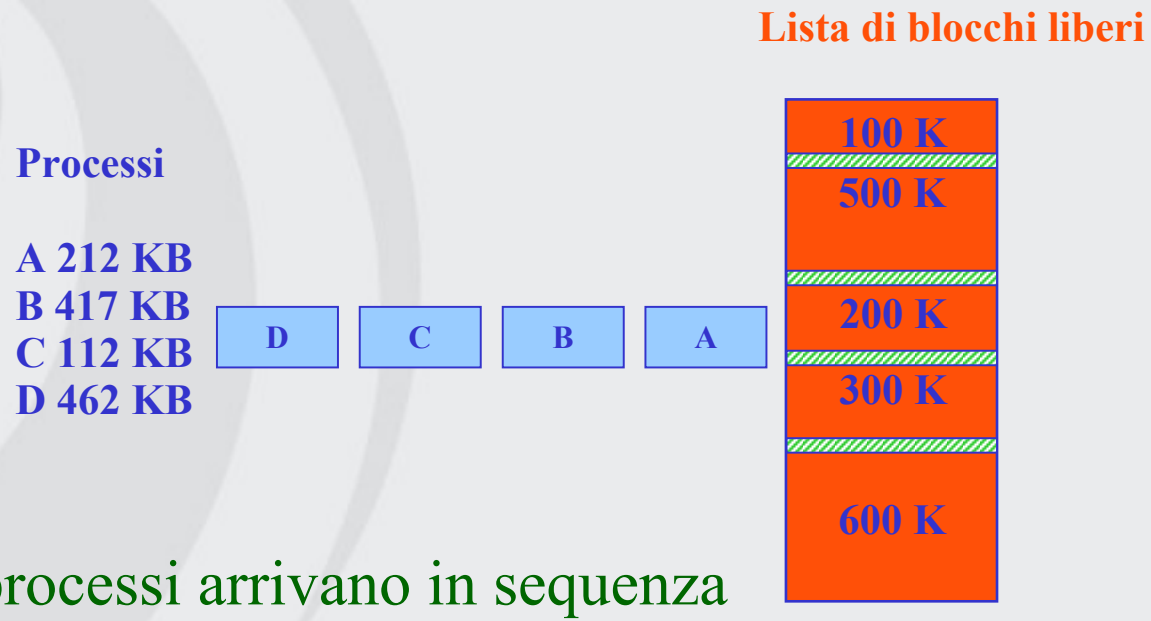
Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB



Memoria a partizioni variabili

first (next best worst) fit



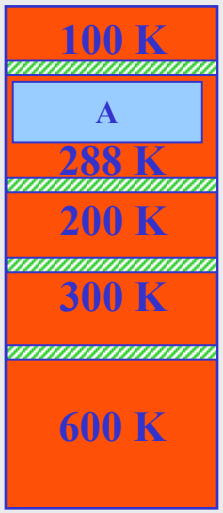
I processi arrivano in sequenza

Memoria a partizioni variabili

first fit

Processi

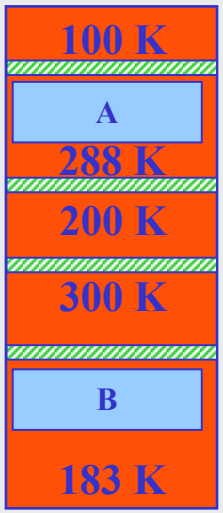
A 212 KB
B 417 KB
C 112 KB
D 462 KB



first fit

Processi

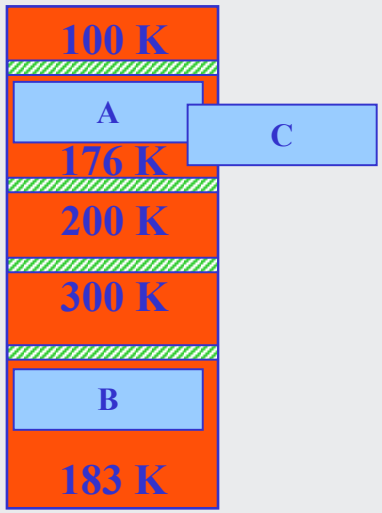
A 212 KB
B 417 KB
C 112 KB
D 462 KB



first fit

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB



Memoria a partizioni variabili

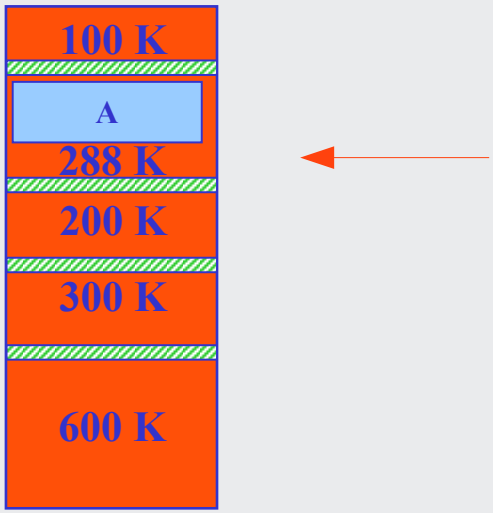
next fit

I processi arrivano in sequenza

next fit

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

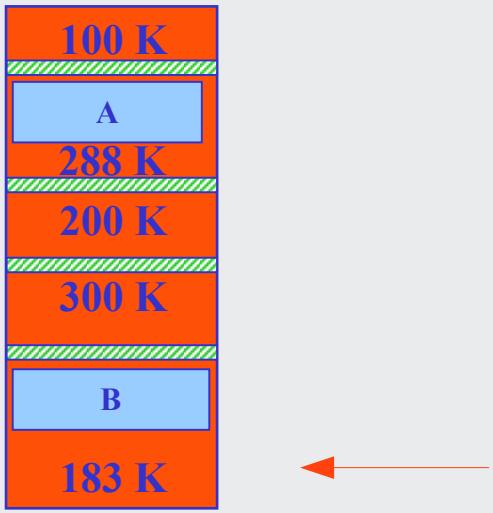


Memoria a partizioni variabili

next fit

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

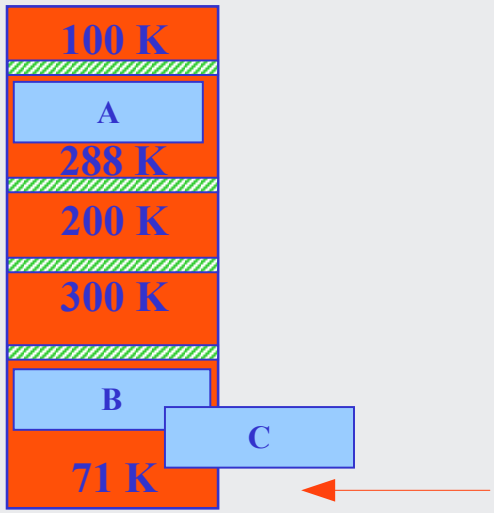


Memoria a partizioni variabili

next fit

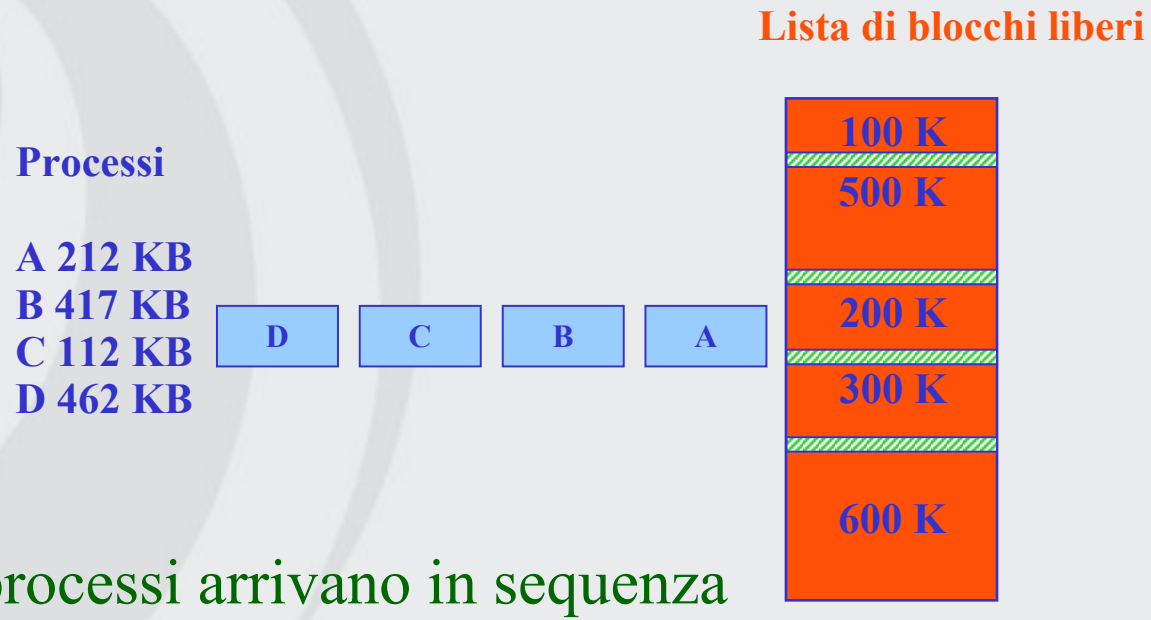
Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB



Memoria a partizioni variabili

best fit

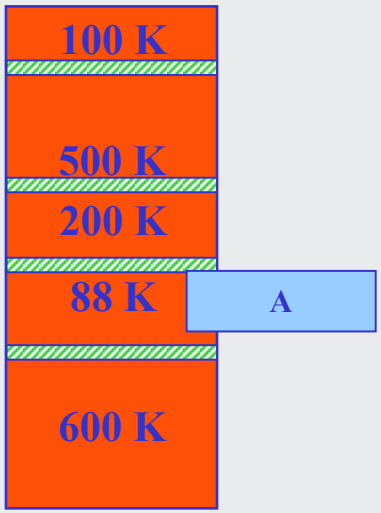


I processi arrivano in sequenza

best fit

Processi

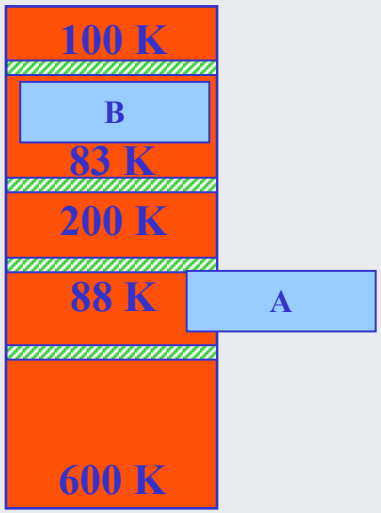
A 212 KB
B 417 KB
C 112 KB
D 462 KB



best fit

Processi

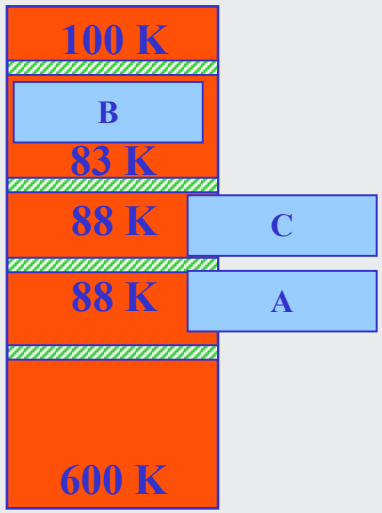
A 212 KB
B 417 KB
C 112 KB
D 462 KB



best fit

Processi

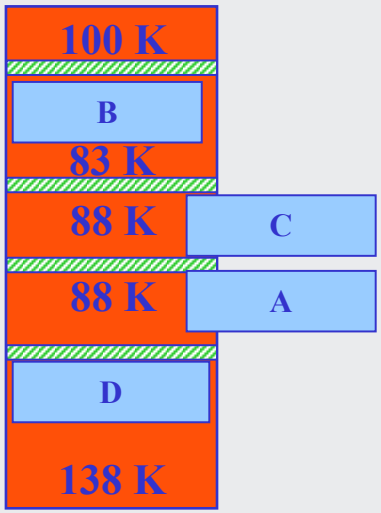
A 212 KB
B 417 KB
C 112 KB
D 462 KB



best fit

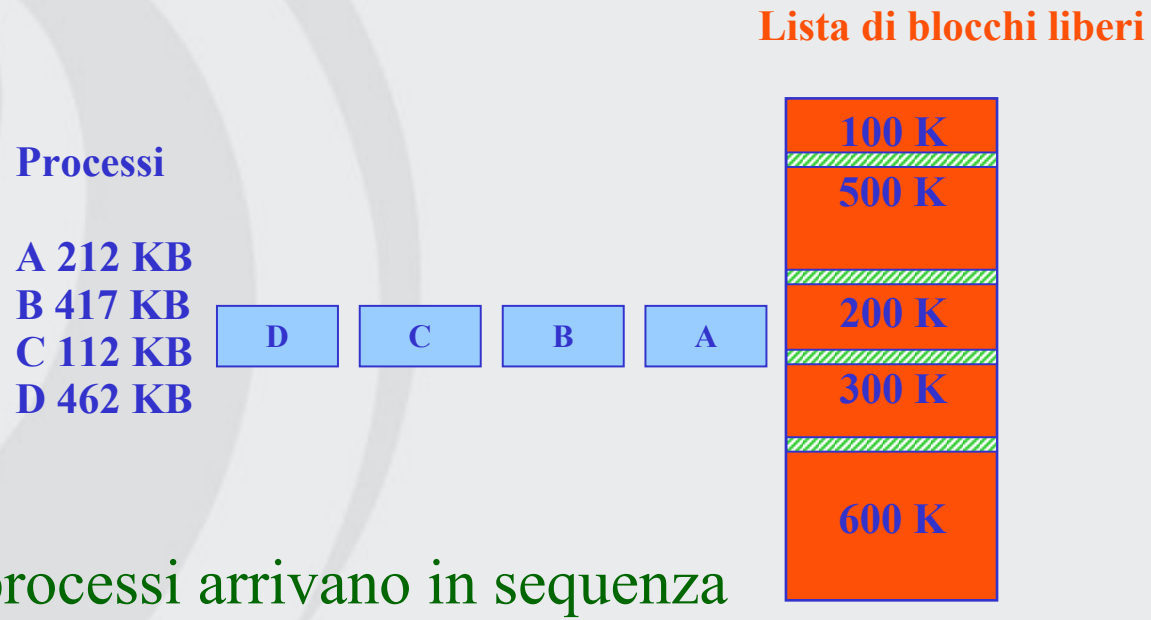
Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB



Memoria a partizioni variabili

worst fit

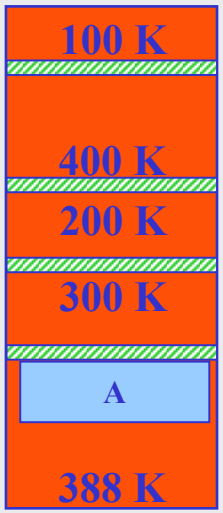


I processi arrivano in sequenza

worst fit

Processi

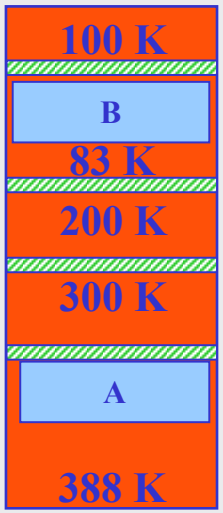
A 212 KB
B 417 KB
C 112 KB
D 462 KB



worst fit

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

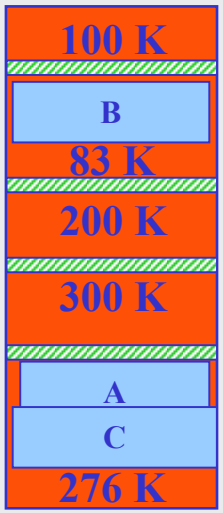


worst fit

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

D



Memoria a partizioni variabili

worst fit

Lista ordinata per dimensione

Processi

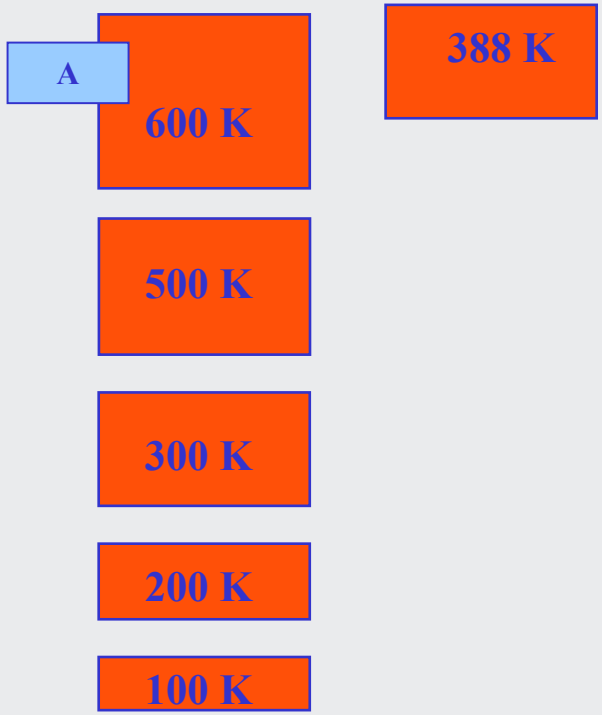
A 212 KB
B 417 KB
C 112 KB
D 462 KB

I processi arrivano in sequenza

Memoria a partizioni variabili

worst fit

Lista ordinata per dimensione



Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

Memoria a partizioni variabili

worst fit

Lista ordinata per dimensione

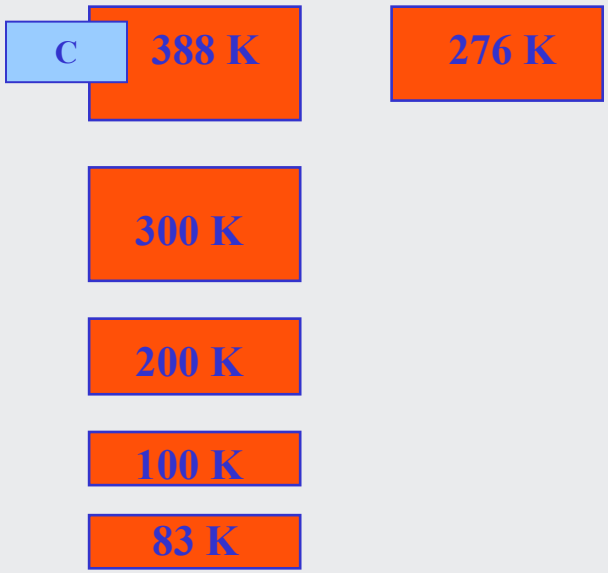
Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

Memoria a partizioni variabili

worst fit

Lista ordinata per dimensione



Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

D

Memoria a partizioni variabili

worst fit **Lista ordinata per dimensione**

Processi

A 212 KB
B 417 KB
C 112 KB
D 462 KB

D

300 K

276 K

200 K

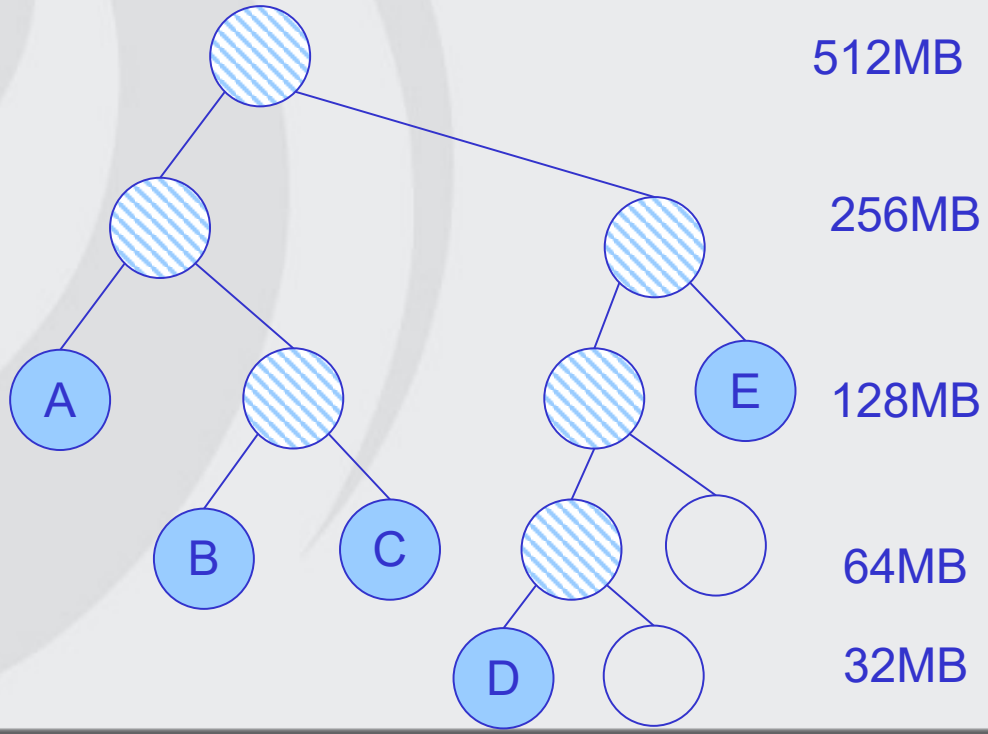
100 K

83 K

Buddy System

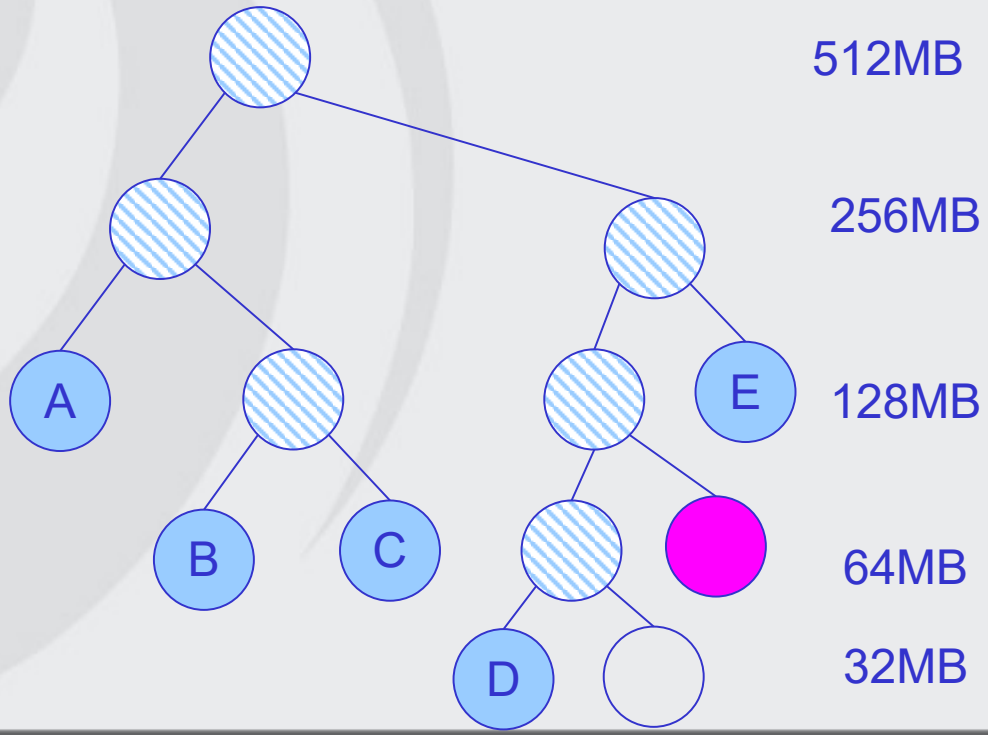
- Mostrare l'evoluzione nel tempo di un sistema che utilizza il metodo di allocazione di memoria Buddy System avendo a disposizione 512MB e dovendo gestire i seguenti Processi:

  (A, 100MB), (B, 50MB), (C, 40MB), (D, 30MB), (E, 70MB)



Buddy System

- Quale è la dimensione massima di un eventuale nuovo processo?



Buddy System

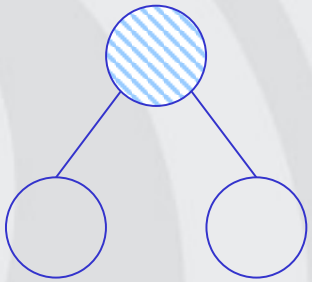
- Mostrare l'evoluzione della memoria con 3 processi di rispettivamente 200KB, 400KB, 50KB (memoria totale a disposizione 1MB)

Buddy System

1MB

Processo A 200 KB

Buddy System

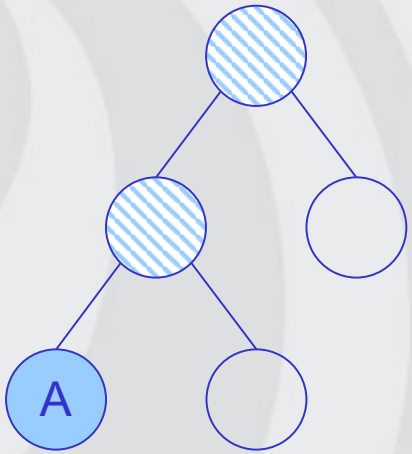


1MB

512KB

Processo A 200 KB

Buddy System



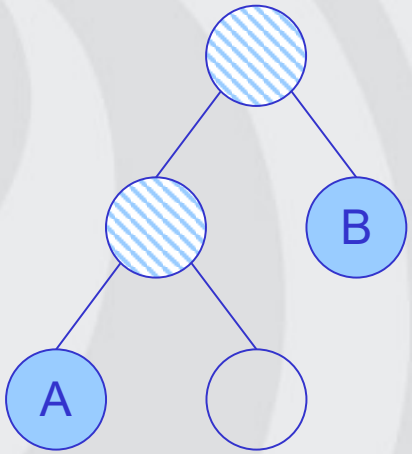
1MB

512KB

256KB

Processo A 200 KB

Buddy System



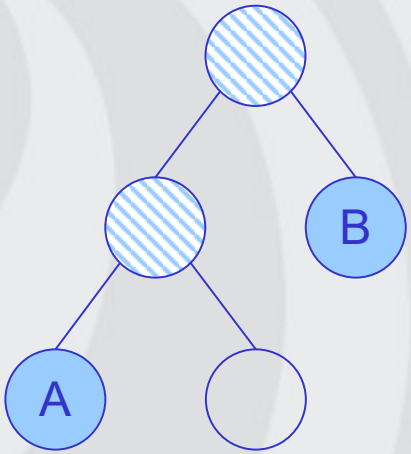
1MB

512KB

256KB

Processo B 400 KB

Buddy System



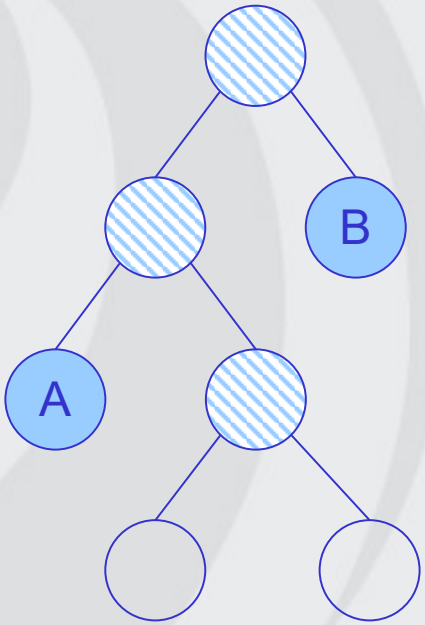
1MB

512KB

256KB

Processo C 50 KB

Buddy System



1MB

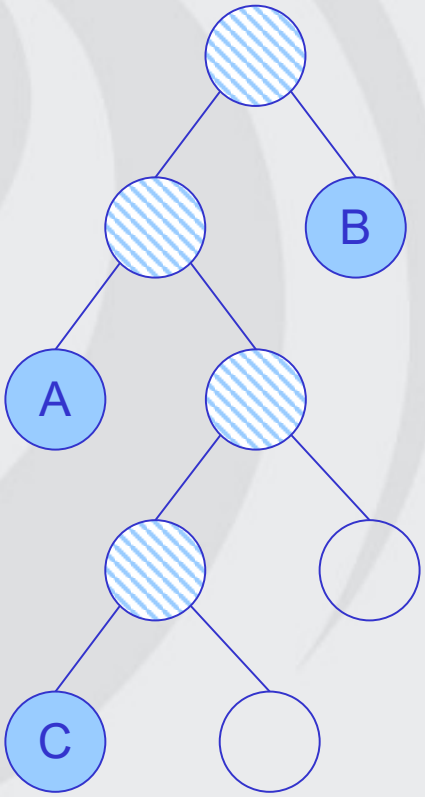
512KB

256KB

128KB

Processo C 50 KB

Buddy System



1MB

512KB

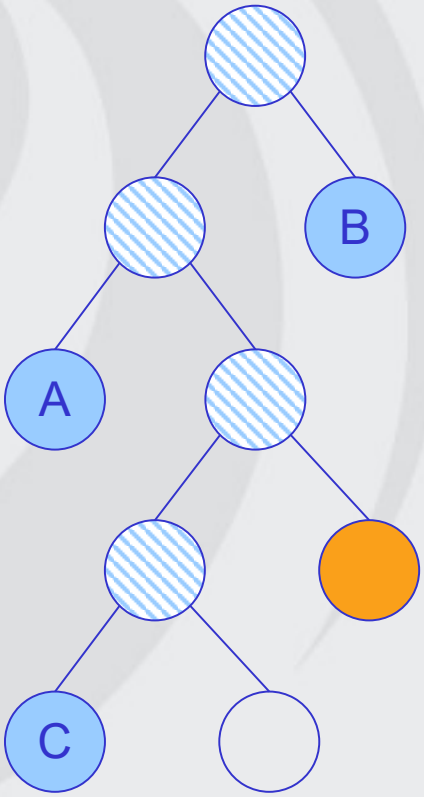
256KB

128KB

64KB

Processo C 50 KB

Buddy System



1MB

512KB

256KB

128KB

64KB

Spazio libero 64+128 KB
Spazio occupato 512+256+64 KB
Spazio inutilizzato 56+112+14 KB

Efficienza 78% (1-182/832)

Dimensione massima di un eventuale nuovo processo 128KB

Rimpiazzamento di pagine

- Descrivere gli algoritmi di rimpiazzamento delle pagine in memoria

  Mostrare come esempio l'effetto con la sequenza di accessi:

  8, 6, 3, 9, 1, 3, 2, 6, 3, 8, 1

  avendo a disposizione 3 pagine.

Rimpiazzamento di pagine

- **Algoritmo ottimo**

  8, 6, 3, 9, 1, 3, 2, 6, 3, 8, 1
  8, 8, 8, 9, 1, 1, 2, 2, 2, 8, 1
  _, 6, 6, 6, 6, 6, 6, 6, 6, 6, 6
  _, _, 3, 3, 3, 3, 3, 3, 3, 3, 3

- **Algoritmo FIFO**

  8, 6, 3, 9, 1, 3, 2, 6, 3, 8, 1
  8, 8, 8, 9, 9, 9, 9, 6, 6, 6, 1
  _, 6, 6, 6, 1, 1, 1, 1, 3, 3, 3
  _, _, 3, 3, 3, 3, 2, 2, 2, 8, 8

Rimpiazzamento di pagine

- Algoritmo LRU

  8, 6, 3, 9, 1, 3, 2, 6, 3, 8, 1
  8, 8, 8, 9, 9, 9, 2, 2, 2, 8, 8
  _, 6, 6, 6, 1, 1, 1, 6, 6, 6, 1
  _, _, 3, 3, 3, 3, 3, 3, 3, 3, 3

- Algoritmo a pila

  8, 6, 3, 9, 1, 3, 2, 6, 3, 8, 1
  _, 8, 6, 3, 9, 1, 3, 2, 6, 3, 8
  _, _, 8, 6, 3, 9, 1, 3, 2, 6, 3

Memoria

Descrivere gli algoritmi di rimpiazzamento delle pagine in memoria

Mostrare l'effetto con la sequenza di accessi:

9, 5, 3, 2, 4, 3, 7, 5, 3, 9, 4, 3, 5, 2, 3

avendo a disposizione 4 pagine.

Algoritmo ottimo

9	5	3	2	4	3	7	5	3	9	4	3	5	2	3
9	9	9	9	9	9	9	9	9	9	4	4	4	4	4
	5	5	5	5	5	5	5	5	5	5	5	5	2	2
		3	3	3	3	3	3	3	3	3	3	3	3	3
			2	4	4	7	7	7	7	7	7	7	7	7

Algoritmo FIFO

9	5	3	2	4	3	7	5	3	9	4	3	5	2	3
9	9	9	9	4	4	4	4	4	9	9	9	9	9	9
	5	5	5	5	5	7	7	7	7	4	4	4	4	4
		3	3	3	3	3	5	5	5	5	5	5	2	2
			2	2	2	2	2	3	3	3	3	3	3	3

Algoritmo LRU

9	5	3	2	4	3	7	5	3	9	4	3	5	2	3
9	9	9	9	4	4	4	4	4	9	9	9	9	2	2
	5	5	5	5	5	7	7	7	7	4	4	4	4	4
		3	3	3	3	3	3	3	5	5	5	5	5	5
			2	2	2	2	5	5	3	3	3	3	3	3

Algoritmo LRU – algoritmo a pila

9	5	3	2	4	3	7	5	3	9	4	3	5	2	3
9	5	3	2	4	3	7	5	3	9	4	3	5	2	3
	9	5	3	2	4	3	7	5	3	9	4	3	5	2
		9	5	3	2	4	3	7	5	3	9	4	3	5
			9	5	5	2	4	4	7	5	5	9	4	4

				9	9	5	2	2	4	7	7	7	9	9
						9	9	9	2	2	2	2	7	7

Tempo medio di accesso

Calcolare la probabilità massima di page fault perché si abbia un tempo medio di accesso ai dati di 12 nanosecondi, con le seguenti ipotesi:

* tempo di accesso in memoria 10 nanosecondi.
* tempo di accesso a disco 30 millisecondi.
* percentuale di pagine modificate 33%.

EAT =

p * accesso con page fault +
(1-p) accesso senza page fault

Tempo medio di accesso

Calcolare la probabilità massima di page fault perché si abbia un tempo medio di accesso ai dati di 12 nanosecondi, con le seguenti ipotesi:

* tempo di accesso in memoria 10 nanosecondi.
* tempo di accesso a disco 30 millisecondi.
* percentuale di pagine modificate 33%.

2/3*30+1/3*(30+30)=40 ms (tempo di accesso a disco pesato)

$P*40\ 10^6 + (1-P)\ 10 < 12 \quad \sim \quad P*40\ 10^6 < 2$

$P = 1 / (20\ 10^6)$

Working set

- Dato il codice seguente (si può ipotizzare che sia parte di una funzione) dare una stima della dimensione del working set durante l'esecuzione del ciclo.

```
#define SIZE 10000000 // 10^7

int *vettore = malloc(SIZE*sizeof(int));
int somma[1000], i;
// ...
for(i=0; i<SIZE; i++)
  somma[i % 1000] += vettore[i];
```