SAPIENZA
UNIVERSITÀ DI ROMA

**Corso di laurea in Ingegneria dell'Informazione**
**Indirizzo Informatica**

Reti e sistemi operativi

# Memoria virtuale

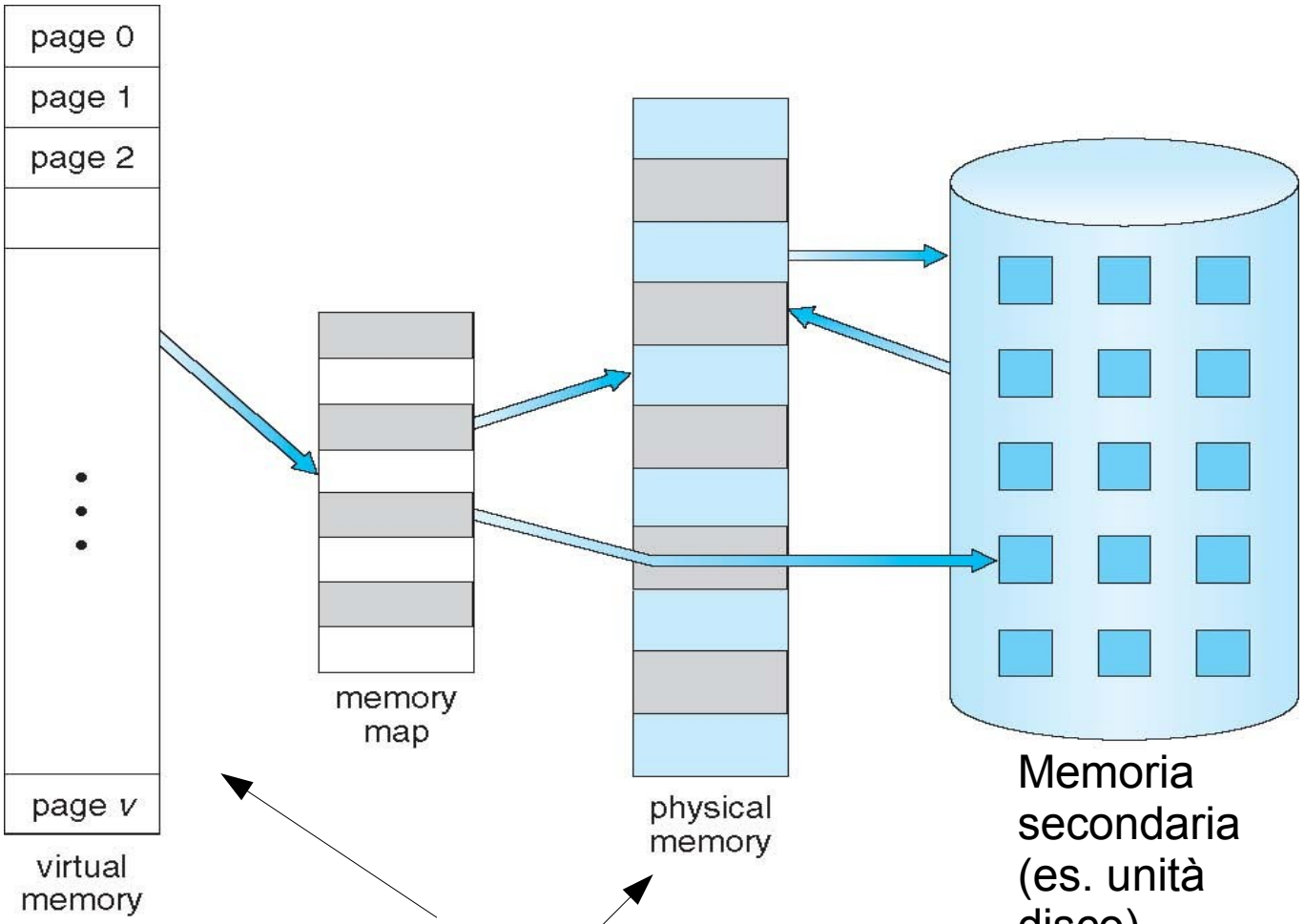
# Motivazioni (1/2)

- Gestione della memoria vista in precedenza: richiede in generale che tutto il processo sia presente in memoria centrale.
- E' stato però introdotto il concetto si **swapping**, che sta alla base del concetto di memoria virtuale
- **Memoria virtuale**: separazione della memoria logica dell'utente dalla memoria fisica.
  - Solo **parti del programma** devono trovarsi in memoria per l'esecuzione.
  - La quantità di memoria percepita dai processi (spazio logico) può essere **più grande** della memoria fisica → ovvero, viene utilizzata parte della memoria di massa (es. HD) come memoria aggiuntiva attraverso procedure di swapping

# Motivazioni (2/2)

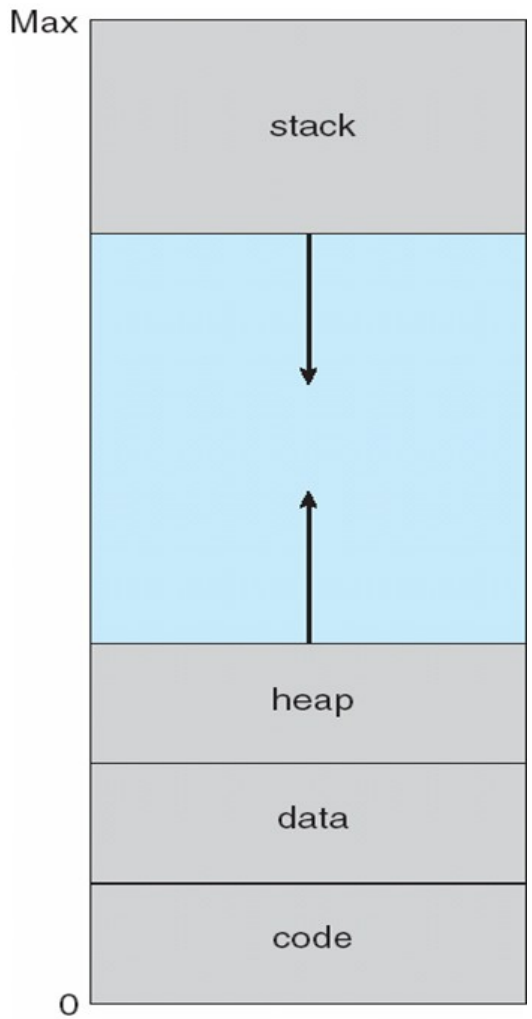
- La possibilità di portare in memoria centrale solo parti del programma permette di minimizzare la quantità di memoria utilizzata, evitando di caricare parti di programma che magari non verranno mai utilizzate
- Come già introdotto in precedenza, attraverso la memoria virtuale porzioni di spazio fisico possono essere condivise da più processi.
- La memoria virtuale può essere implementata per mezzo di:
  - Paginazione su richiesta
  - Segmentazione su richiesta

# RAM + memoria secondaria



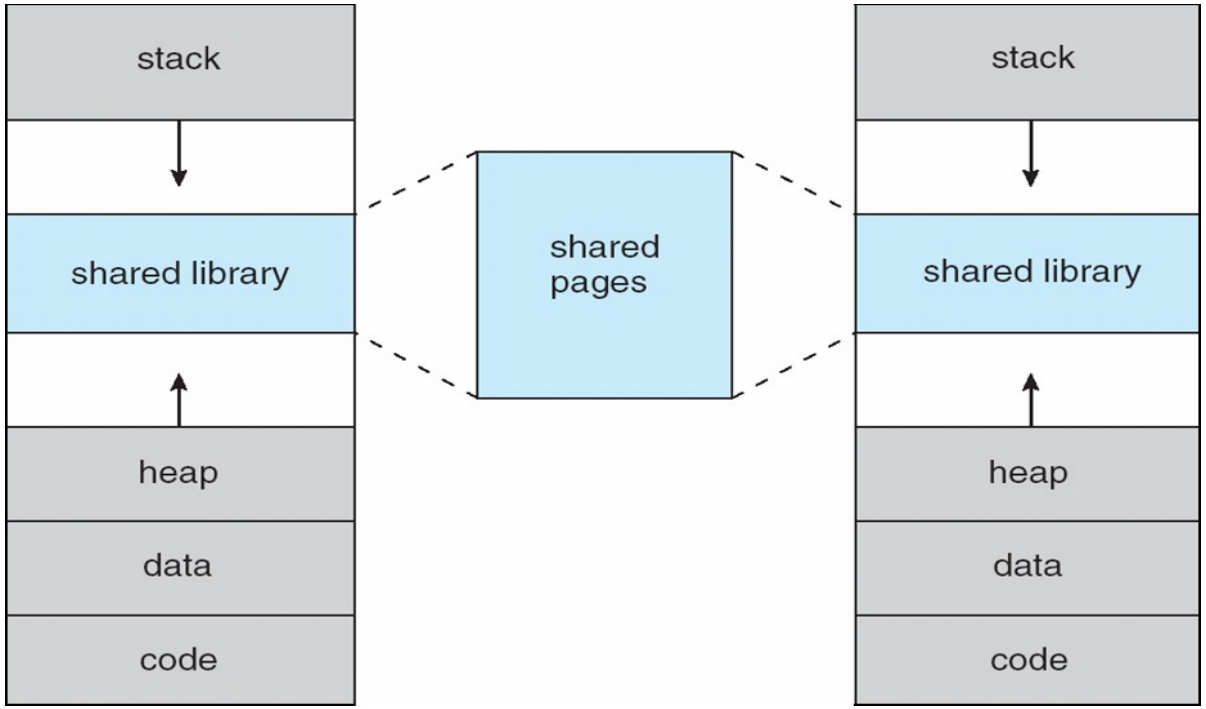
# Spazio virtuale degli indirizzi

- Idea della memoria virtuale: **memoria sparsa**, ovvero un rappresentata da uno spazio di indirizzi molto grande (possibilmente il massimo indirizzabile) con grandi spazi vuoti (hole)
- Tali hole sono solo virtuali: ad esempio, la separazione tra heap e stack può essere massima, evitando qualsiasi problema di riallocamento e spostamento di memoria
- Al solito, la traduzione da indirizzi virtuali a fisici è trasparente ai programmi



# Memoria condivisa

- Librerie condivise possono essere mappate con indirizzi logici (virtuali) read-only
- In maniera analoga, memoria condivisa può essere mappata con indirizzi logici (virtuali) read-only

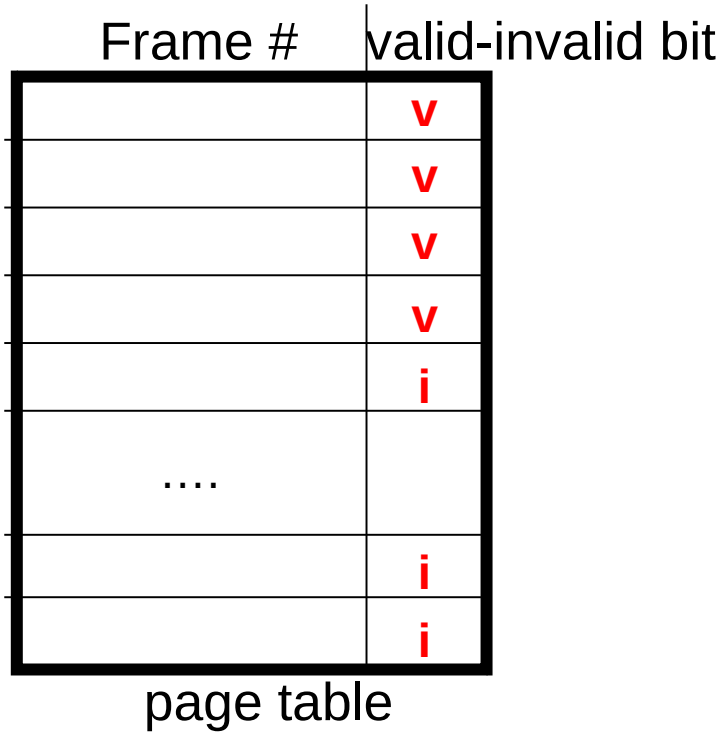


# Paginazione su richiesta

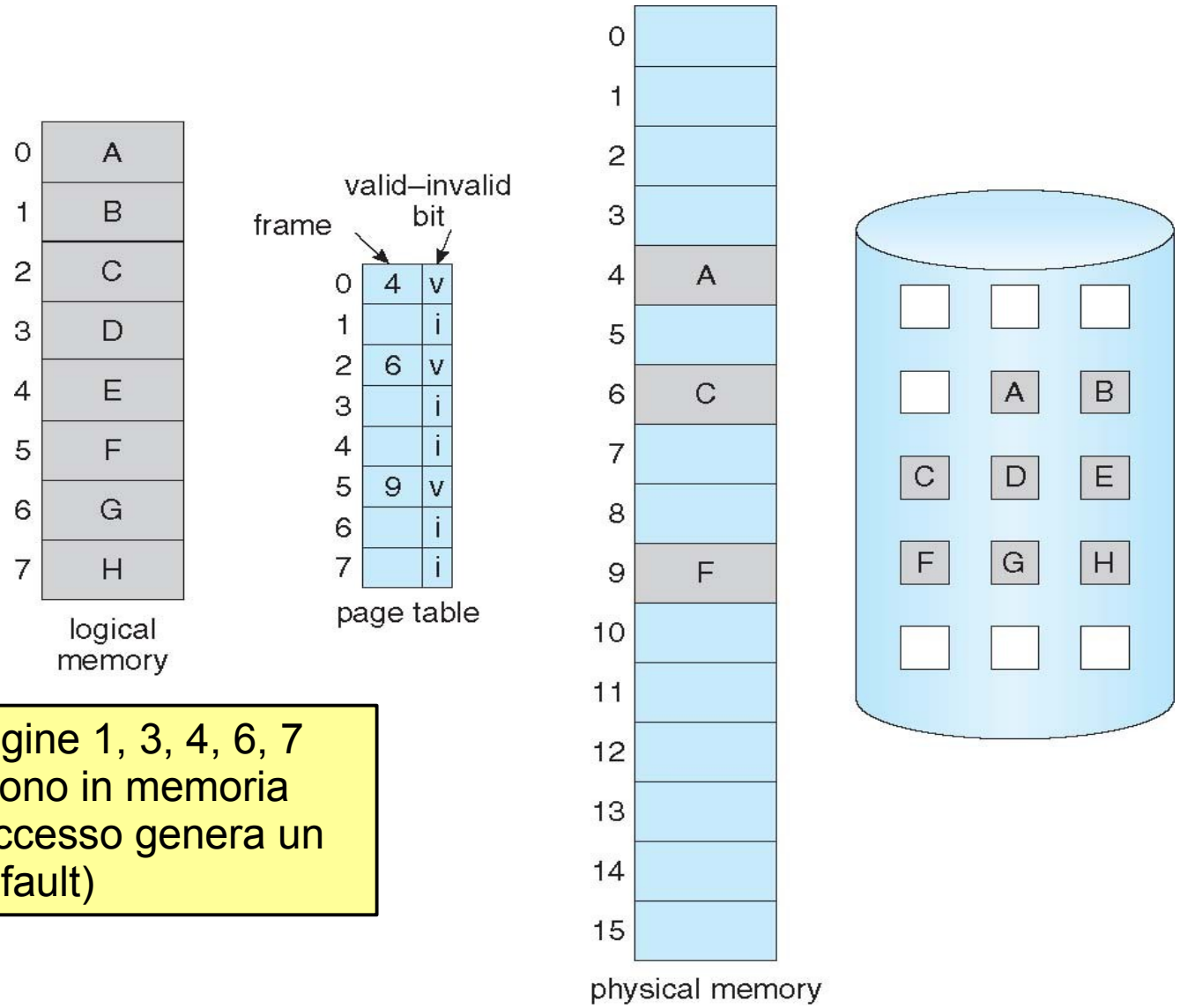
- Le pagine sono caricate in memoria solamente quando ne viene esplicitamente richiesto un accesso
- **Lazy swapper (swapper pigro)**: porta una pagina in memoria solo se è necessaria
  - Occorrono meno operazioni di I/O
  - Viene impiegata meno memoria
  - Si ha una risposta più veloce
  - Si possono gestire più utenti
- Richiesta (accesso) ad una pagina
  - Riferimento non valido → abort
  - Pagina non in memoria → trasferimento in memoria

# Bit di validità (1/2)

- Un bit di validità viene associato a ciascun elemento della tabella delle pagine (1 → in memoria, 0 → non in memoria, **page fault**).
- Inizialmente il bit di validità viene posto a 0 per tutte le pagine.



# Bit di validità (2/2)

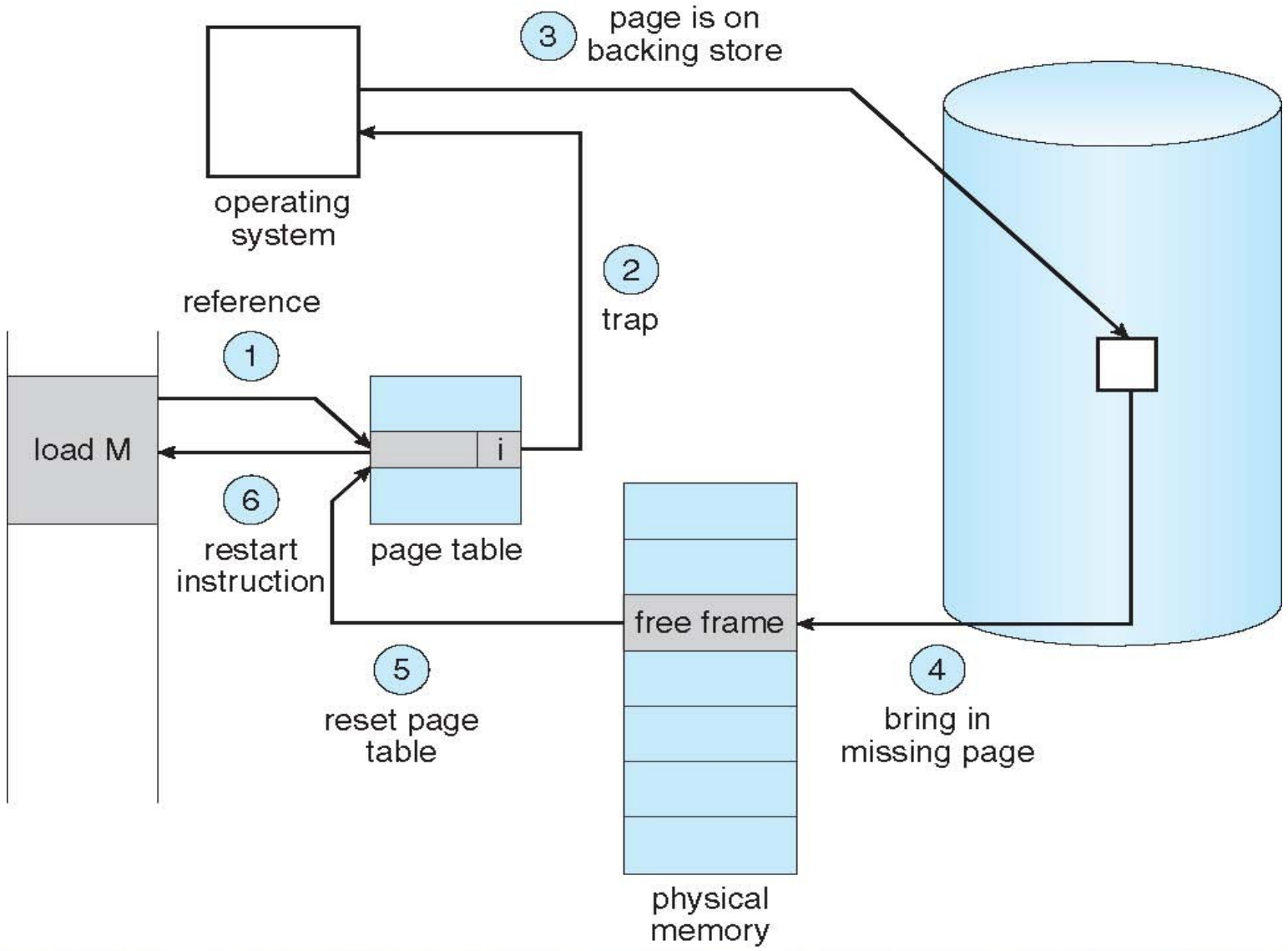



Le pagine 1, 3, 4, 6, 7 non sono in memoria (un accesso genera un page fault)

# Page fault (1/2)

- Se non ci sono stati accesso ad una pagina, il primo accesso causa un trap (**interrupt**) al SO → page fault, il SO consulta una tabella per decidere se si tratta di riferimento non valido (abort del processo) o una pagina non in memoria.
- Nel secondo caso:
  - Seleziona un frame vuoto
  - Sposta la pagina nel frame
  - Aggiorna le tabelle (bit di validità = 1)
  - Viene riavviata l'istruzione che era stata interrotta

# Page fault (2/2)



# Aspetti legati alla paginazione su richiesta

- Caso estremo (**paginazione su richiesta pura**): un processo viene avviato senza caricare in memoria nessuna pagina
  - La prima istruzione genera un page fault, e così via
- Un'istruzione può accedere a riferimenti che si trovano in più pagine, magari non presenti in memoria. Problemi:
  - Molti page fault → Problema mitigato dalla **località dei riferimenti**
  - Un page fault potrebbe interrompere la sequenza di operazioni: è necessario ripetere l'intera sequenza una volta risolti i page fault → Idea: Accedere a tutti i blocchi coinvolti prima di iniziare le operazioni

# Paginazione su richiesta: prestazioni

- Se non vi sono frame liberi in memoria, è necessario provvedere alla sostituzione di almeno un frame: si richiede ciò di trovare un frame in memoria che non viene utilizzato attualmente, che verrà spostato a disco (swap-out).
- **Algoritmo di selezione**: è richiesto un metodo che produca il minimo numero di page fault.
  - La stessa pagina infatti può essere riportata in memoria più volte.

# Page Fault Rate

- Page Fault Rate: $0 \leq p \leq 1.0$
  - Se p = 0 non si hanno page fault;
  - Se p = 1, ciascun riferimento è un fault.
- Tempo medio di accesso (Effective Access Time, EAT):
  - **EAT = (1 - p) * t[accesso alla memoria] + p * t[page fault]**
  - Ove t[page fault] = t[swap out di pagina] + t[swap in di pagina] + overhead di restart
- Durante lo swap-in e lo swap-out, il processo viene spostato in una waiting queue associata al disco

# Esempio di paginazione su richiesta

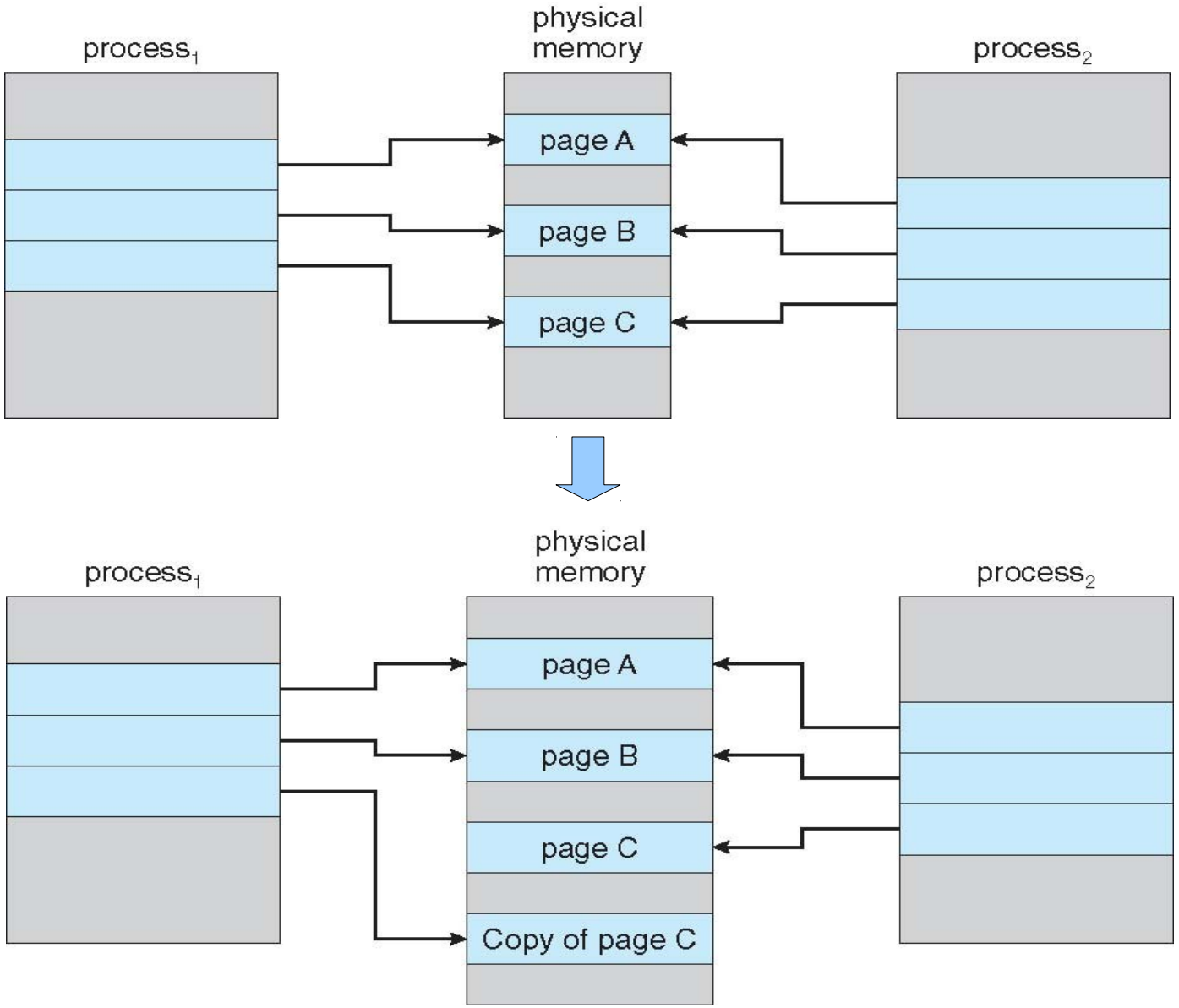
- Tempo di accesso alla memoria = 1 µsec
- Tempo di swap per pagina = 10 msec = 10000 µsec
- Il 50% delle volte che una pagina viene rimpiazzata ha subito delle modifiche e deve essere sottoposta a swap out.

  **EAT = (1 – p) * 1 + p * ((1+0.5)*10000) ≈ 1 + 15000*p (in µsec)**
- Se un accesso su 1000 causa un page fault, EAT = 16 µsec:
  - **Con la paginazione su richiesta l'accesso in memoria viene rallentato di un fattore 16**. Se si desidera un rallentamento inferiore al 10%:
    1.1 > 1 + 15000p → p < 0.0000067
    cioè può essere permesso meno di un page fault ogni 150000 accessi in memoria.

# Copy-on-Write (1/2)

- Spesso dopo aver creato un nuovo processo (es. con fork) viene caricato un nuovo programma (con exec()) → la copia iniziale è così inutile
- **Copy-on-write**: tecnica che permette di creare nuovi processi in minor tempo, allo stesso tempo minimizzando il numero di nuove pagine che devono essere allocate.
- La copia su scrittura permette sia al processo genitore, sia al figlio, di condividere inizialmente pagine di memoria.
- **Se uno dei processi cerca di modificare una pagina, solo allora la pagina è copiata.**
- Solo le pagine che possono essere modificate vengono copiate, le read-only vengono invece condivise.

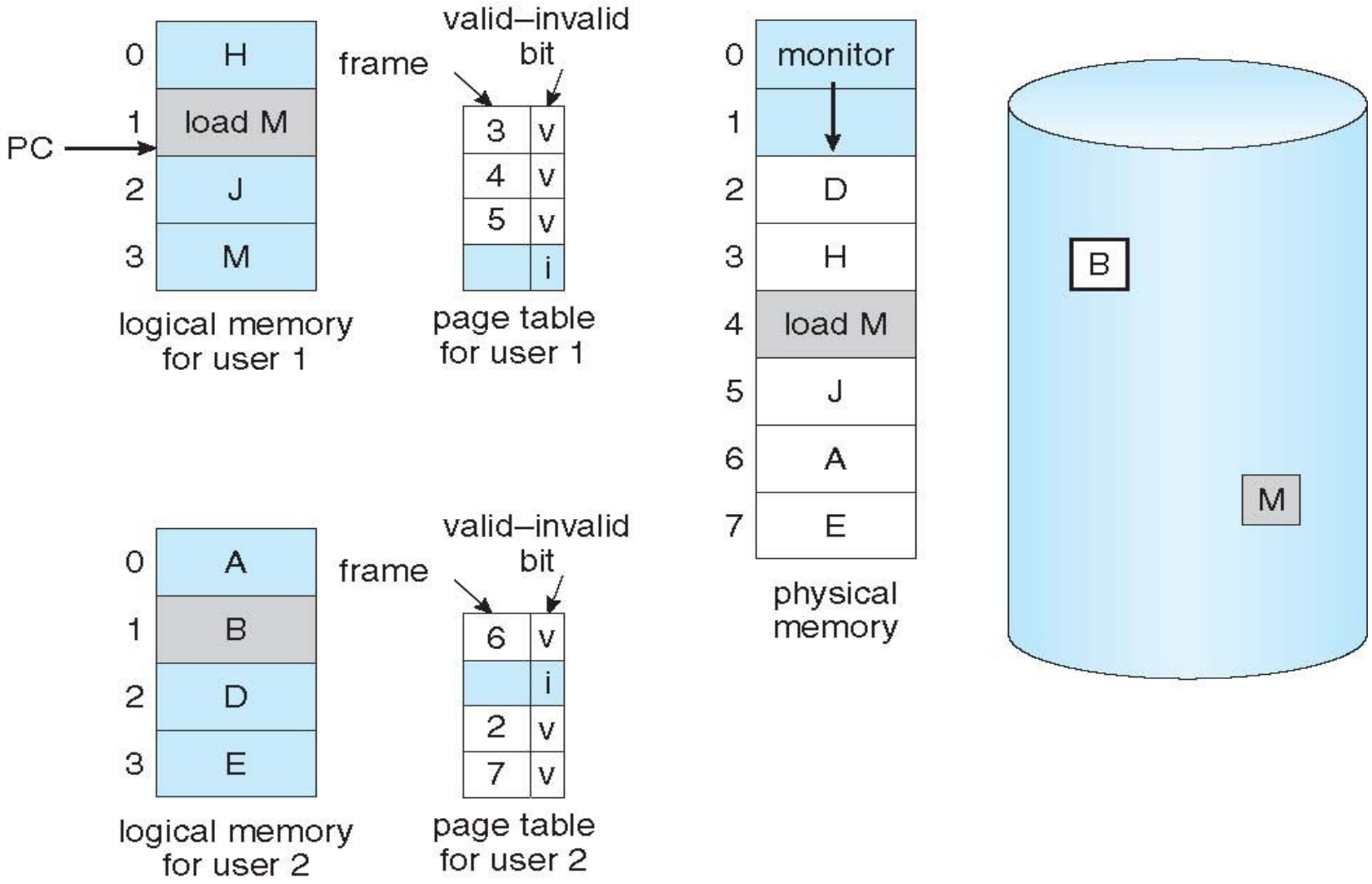
# Copy-on-Write (2/2)



# Sostituzione delle pagine (1/4)

- La sovrallocazione della memoria si verifica quando è richiesta più memoria di quella effettivamente disponibile → la sostituzione delle pagine può risolvere questo problema
- Si impiega un bit di modifica (**dirty**) per ridurre il sovraccarico dei trasferimenti di pagine: solo le pagine modificate vengono riscritte sul disco.
- La sostituzione delle pagine completa la separazione fra memoria logica e memoria fisica: una grande memoria virtuale può essere fornita ad un sistema con poca memoria fisica.

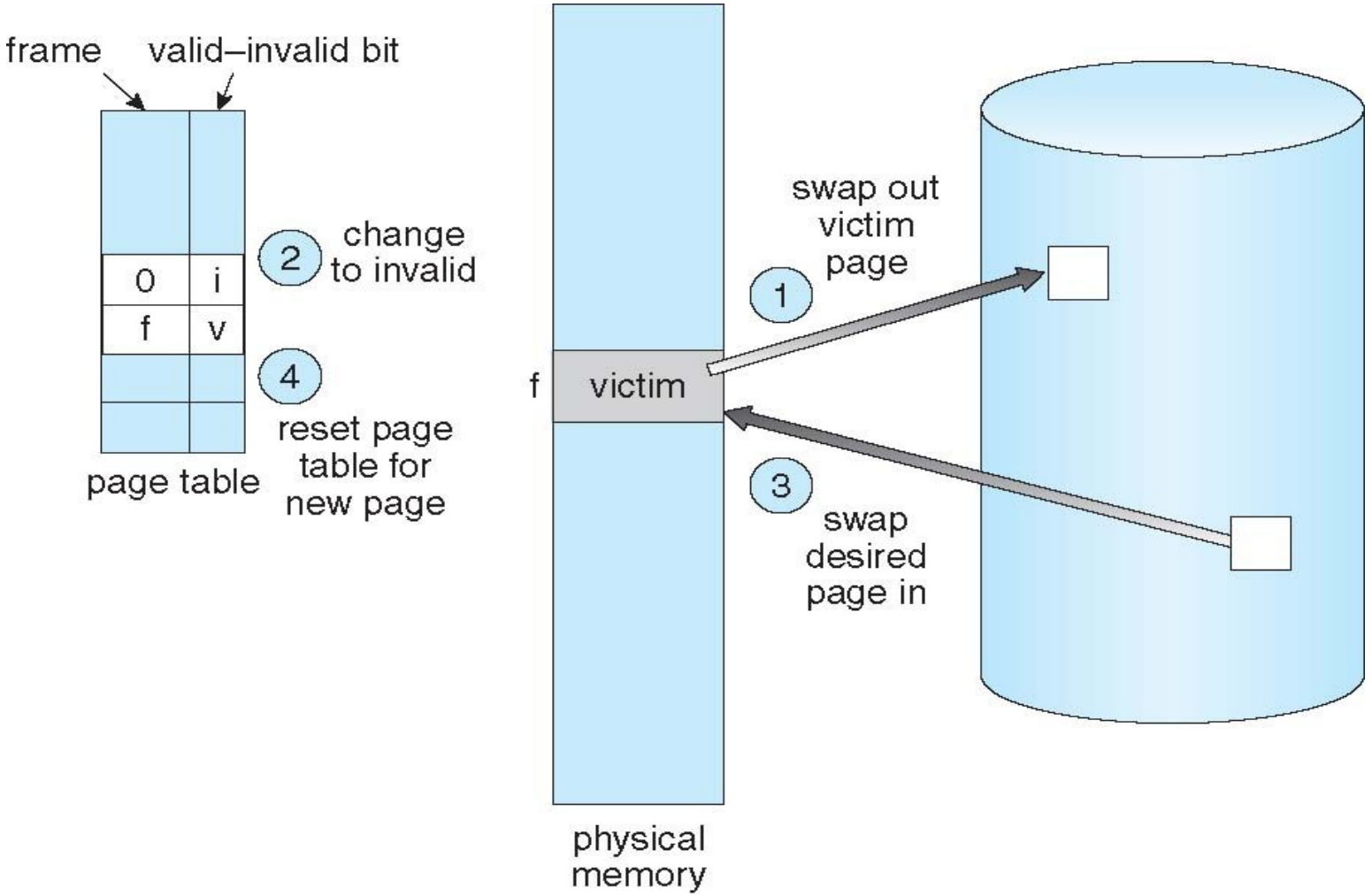
# Sostituzione delle pagine (2/4)



# Sostituzione delle pagine (3/4)

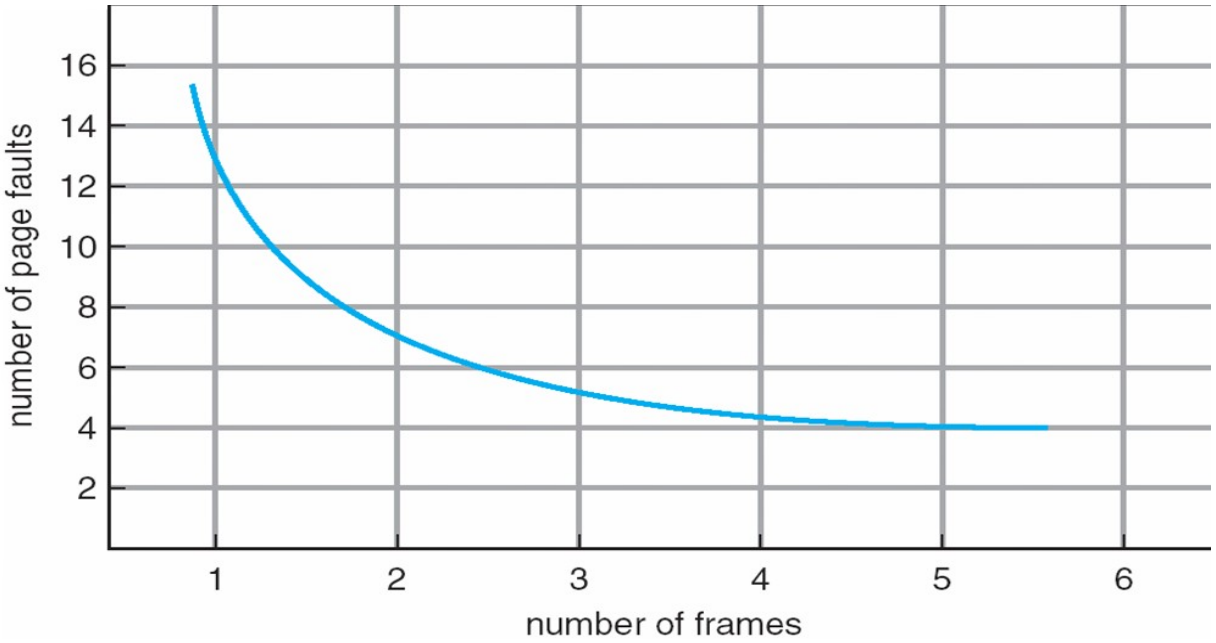
- Individuazione della pagina richiesta su disco.
- Individuazione di un frame libero:
  - Se esiste un frame libero, viene utilizzato.
  - Altrimenti viene utilizzato un algoritmo di sostituzione per selezionare un **frame "vittima"**.
  - La pagina vittima viene scritta sul disco; le tabelle delle pagine e dei frame vengono modificate conformemente.
- Lettura della pagina richiesta nel frame appena liberato; modifica delle tabelle delle pagine e dei frame.
- Riavvio del processo utente.

# Sostituzione delle pagine (4/4)



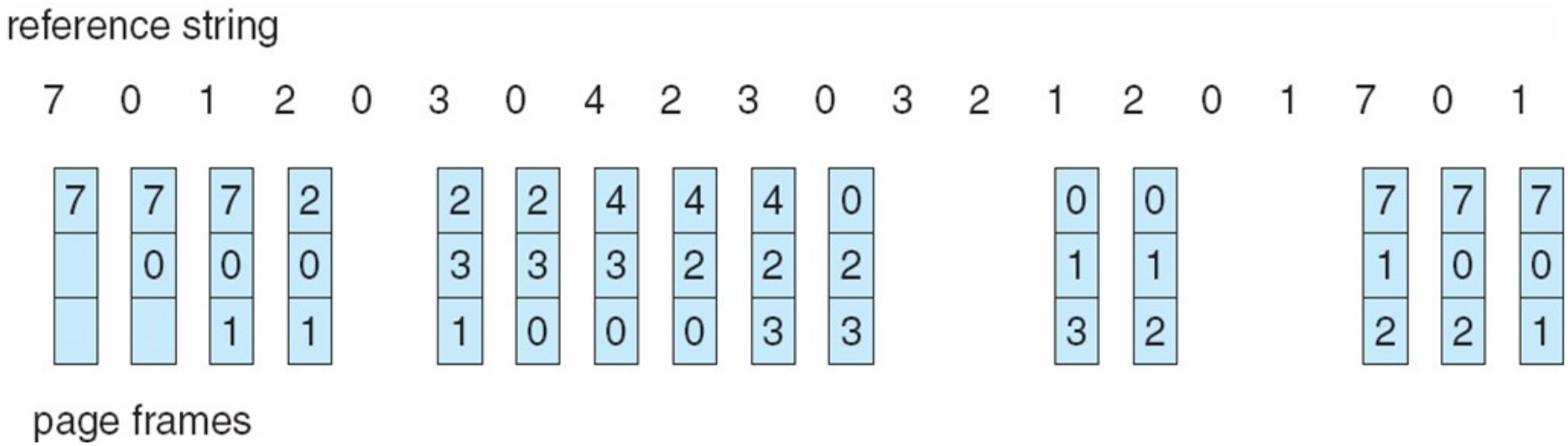
# Algoritmi per la sostituzione delle pagine

- **L'obiettivo è minimizzare la frequenza di page fault**.
- Si valutano gli algoritmi eseguendoli su una particolare stringa di riferimenti a memoria (reference string, ovvero una sequenza di pagine a cui si vuole accedere ) e contando il numero di page fault su tale stringa.



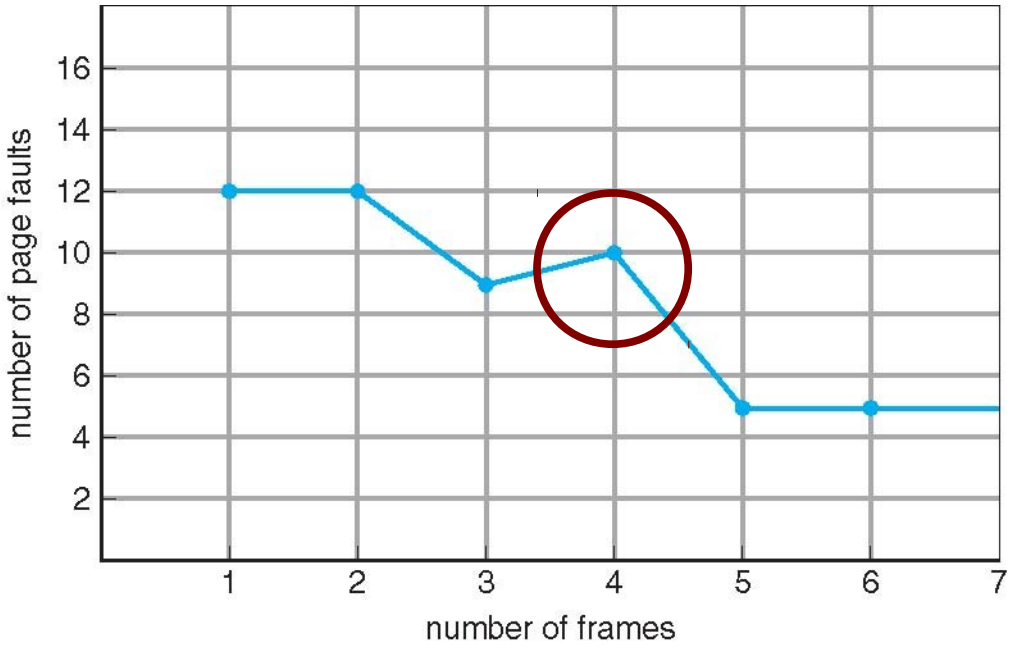
# Algoritmo First-In-First-Out (1/2)

- Idea: Sostituire la pagina più vecchia portata in memoria, usando semplicemente una coda FIFO per ordinare le pagine



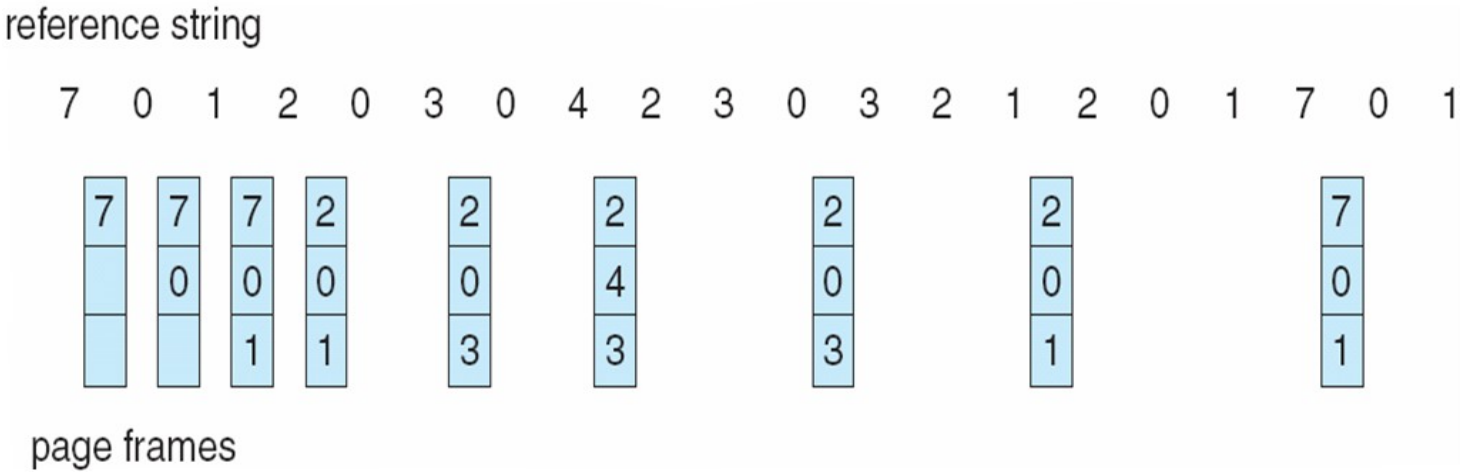
# Algoritmo First-In-First-Out (2/2)

- **Problema algoritmo FIFO**: anomalia di Belady: aumentando il numero di frame potrebbe portare ad un aumento dei page fault



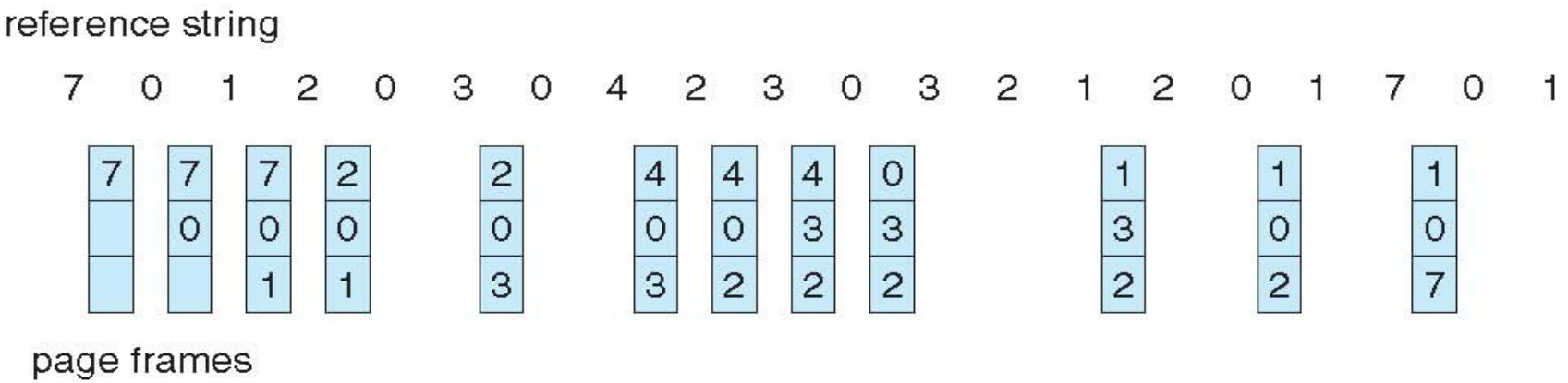
# Algoritmo ottimo

- Idea: Sostituire la pagina che non **verrà** usata per il periodo di tempo più lungo.
- E' un algoritmo ottimo, ma **non è possibile conoscere a priori l'ordine con cui verranno richieste le pagine**. → viene usato come riferimento per gli altri algoritmi



# Algoritmo Least–Recently–Used (LRU)

- Idea: Sostituire la pagina che non **viene** usata per il periodo di tempo più lungo.



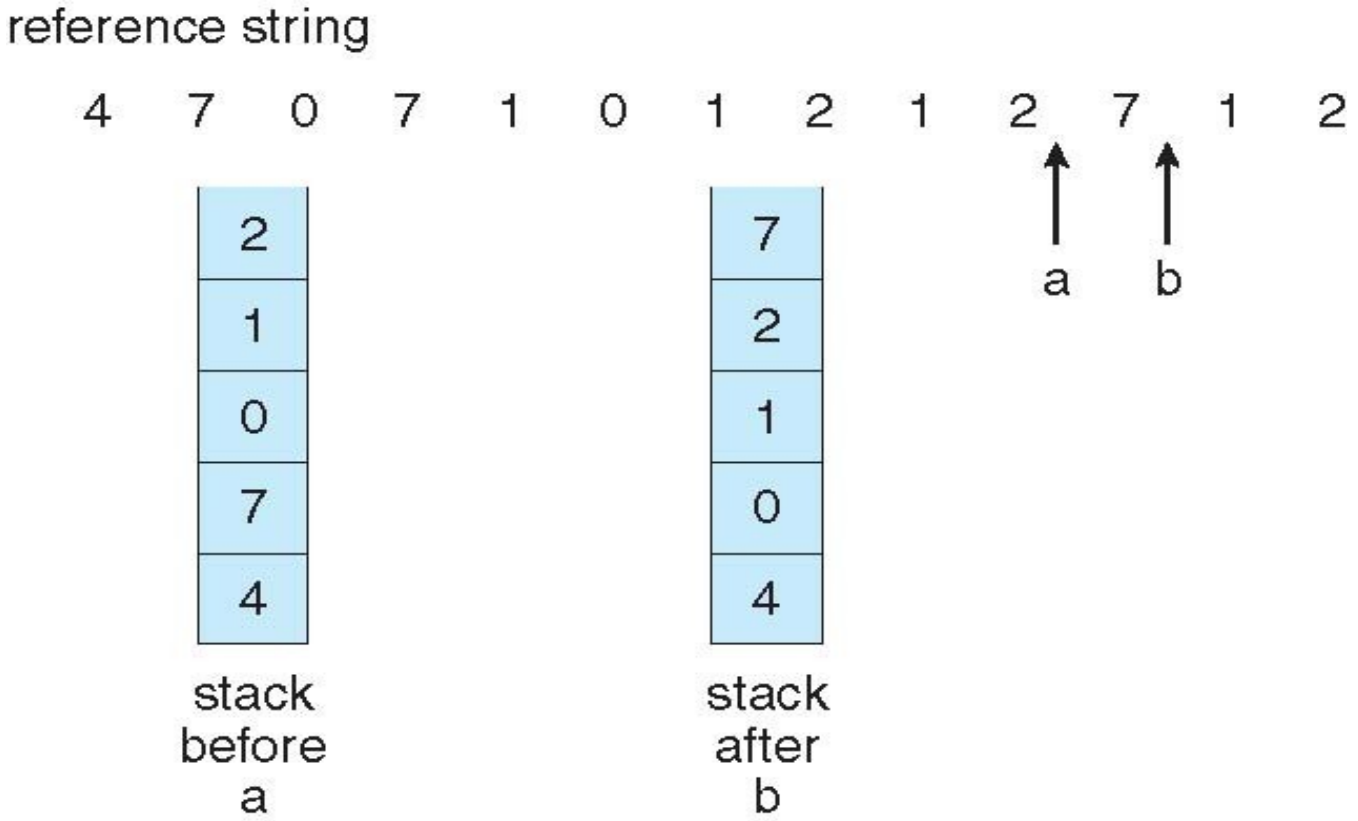

Algoritmo ottimo e LRU non soffrono dell'anomalia di Belady

# Implementazione LRU (1/2)

- **Implementazione con contatore**: ciascuna pagina ha un contatore:
  - Ogni volta che si fa riferimento alla pagina, si copia l'orologio nel contatore.
  - Per rimuovere una pagina, si analizzano i contatori → **necessaria una ricerca in tutta la tabella delle pagine**
- **Implementazione con stack**: si tiene uno stack di numeri di pagina in forma doubly linked list, la pagina che viene richiesta viene spostata in testa allo stack, in coda vi sono le pagine non usate da più tempo.
- In entrambi i casi è richiesto un hardware dedicato, dal momento che sono richiesti vari accessi alla memoria (aggiornamento clock o stack) **per ogni accesso richiesto**.

# Implementazione LRU (2/2)

- Esempio di implementazione con stack



# Algoritmi LRU approssimati (1/3)

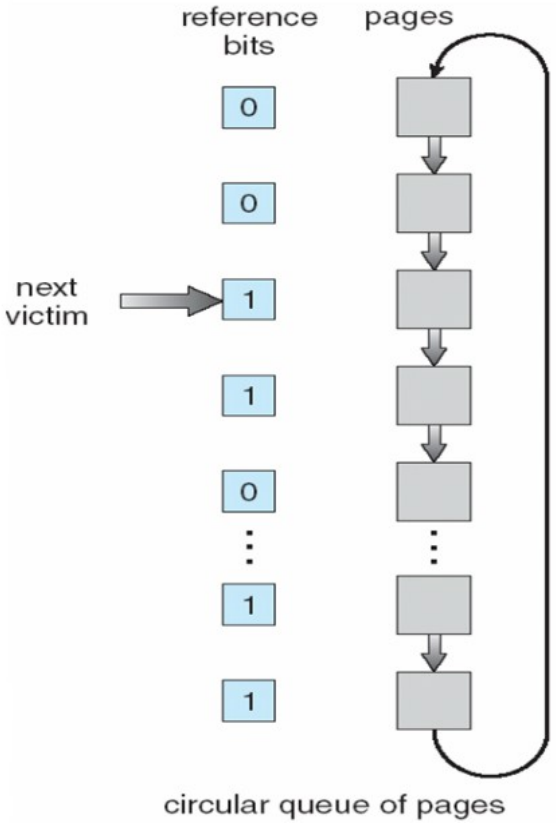
- Non sempre è disponibile un supporto hardware a LRU
  - E' spesso necessario usare soluzioni software più semplici ma meno efficaci come FIFO
  - Si utilizzando delle soluzioni hardware più semplici
- **Bit di riferimento**
  - A ciascuna pagina si associa un bit, inizialmente = 0.
  - Quando si fa riferimento alla pagina si pone il bit a 1.
  - Si rimpiazza la pagina il cui bit=0 (se esiste).
  - **Problema**: Non si conosce l'ordine di accesso alle pagine.

# Algoritmi LRU approssimati (2/3)

- **Bit di riferimento supplementari**
  - A ciascuna pagina si associa un insieme di bit (es 1 byte) inizialmente = 0.
  - Ciclicamente (grazie all'intervento di un timer) il S.O. sposta il bit di riferimento nel bit più significativo, spostando gli altri di 1 e scartando il meno significativo.
  - Questi registri a scorrimento contengono l'ordine d'uso delle pagine.
  - Numeri maggiori, corrispondono pagine usate più di recente → in caso di più pagine con stesso valore, si usa strategia FIFO per scegliere la prossima vittima

# Algoritmi LRU approssimati (3/3)

- **Seconda Chance (clock)**
- DI base FIFO, con l'aggiunta del bit di riferimento.
- Se la prossima pagina da rimpiazzare, secondo la politica FIFO, ha bit di riferimento settato, viene data una “seconda possibilità”, ovvero non viene scelta ma si resetta il bit di riferimento.
- **Seconda Chance migliorato**: si usa anche il bit di modifica (dirty): la migliore pagina da scegliere è quella né recentemente usata, né modificata.



# Algoritmi con conteggio

- Viene utilizzato un contatore che tiene traccia del numero di accessi che sono stati fatti a ciascuna pagina.
- **Algoritmo LFU (Least Frequently Used)**: si rimpiazza la pagina col valore più basso del contatore.
  - Se la pagina è stata usata molto nella fase iniziale può non venire mai rimpiazzata.
- **Algoritmo MFU (Most Frequently Used)**: si assume che la pagina col valore più basso del contatore è stata spostata recentemente in memoria e deve ancora essere impiegata.
- Entrambi poco diffusi anche per una non facile implementazione hardware a fronte di prestazioni modeste rispetto all'algoritmo ottimo

# Bufferizzazione delle pagine

- **Idea**: Si posticipano le operazioni di swap out ad istanti oppotuni
- Il sistema operativo mantiene un set di frame liberi (*free frames pool*), subito disponibili in caso di page fault
- La pagina selezionata per essere sostituita non viene subito salvata a disco (swap out) → **la scrittura viene posticipata ad istanti meno critici, quando ad esempio l'unità disco è libera**.
- **Variazione 1**: Una simile idea è utilizzata mantenendo una lista delle pagine modificate (dirty bit = 1): se il disco non è impegnato, a turno si salva a disco una pagina modificata, resettando il dirty bit → si aumenta la probabilità di selezionare vittime prive di modifiche.
- **Variazione 2:** Si mantiene una lista delle pagine ancora presenti nel free frames pool → in caso di accesso ad una di esse, la richiesta può essere servita subito

# Allocazione dei frame

- Come vengono distribuiti i frame tra i vari processi in esecuzione?
- Un numero **minimo** deve essere comunque garantito, per permettere l'esecuzione ininterrotta (ovvero, senza page fault) di istruzioni che richiedono l'accesso a più di una pagina (es. istruzioni di spostamento della memoria con indirizzamento indiretto)

Indirizzamento indiretto: Il campo indirizzo dell'istruzione contiene l'indirizzo di una cella che contiene l'indirizzo dell'area di memoria a cui si vuole accedere.

- Numero massimo → limitato da quantità di memoria
- Si hanno due schemi principali di assegnazione:
  - Assegnazione fissa.
  - Assegnazione con priorità.

# Allocazione fissa

- Assegnazione uniforme: stessa memoria per ogni processo.
  - Es.: con 100 frame a disposizione e 5 processi, si assegnano 20 pagine a ciascun processo.
- Assegnazione proporzionale: i frame sono allocati sulla base della dimensione del processo.

$s_i = \text{size of process } p_i$

$S = \sum s_i$

$m = \text{total number of frames}$

$a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 64 \approx 5$

$a_2 = \frac{127}{137} \times 64 \approx 59$

Ovviamente in entrambi i casi la quota allocata per processo diminuisce se aumenta il grado di multi programmazione, ovvero il numero di processi in esecuzione

# Allocazione basata su priorità

- Allocazione proporzionale basata sulle priorità dei processi piuttosto che sulla loro dimensione. Si seleziona per la sostituzione:
  - Un frame che appartiene al processo, oppure:
  - Un frame utilizzato da un processo con priorità più basso.

# Allocazione globale e locale

- **Sostituzione globale**: si seleziona il frame da sostituire dall'insieme di tutti i frame; si può selezionare un frame utilizzato da un processo diverso. Metodo generalmente usato nei sistemi reali.
  - Vantaggi: maggior utilizzo delle risorse
  - Limitazioni: Minor controllo e quindi determinismo temporale, ovvero un processo non può controllare la propria frequenza di page fault.
- **Sostituzione locale**: ciascun processo può utilizzare un set di frame in precedenza allocati in maniera esclusiva al processo stesso:
  - Vantaggi: determinismo temporale
  - Limitazioni: sotto-utilizzo delle pagine

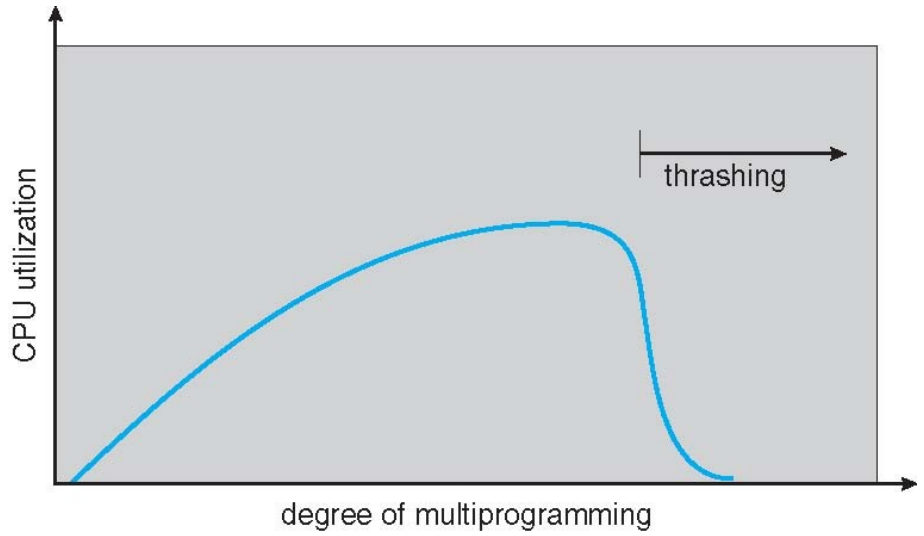
# Thrashing (1/2)

- Se un processo non ha abbastanza pagine, la frequenza di page fault è molto alta:
  - Page fault → nuova pagina B viene richiesta, un frame precedentemente occupato dalla pagina A viene sovrascritto.
  - Il processo richiede di nuovo A → **ancora page fault**.
- → **Thrashing**: i processi sono occupati soprattutto in operazioni di swap-out e swap-in.
- Conseguenze: sotto utilizzo della CPU → il S.O. aumenta il grado di multiprogrammazione (es. mette in esecuzione un nuovo processo) per aumentare l'utilizzo delle risorse, in realtà ottiene l'effetto contrario
  - **Il fenomeno del thrashing aumenta**.

# Thrashing (2/2)

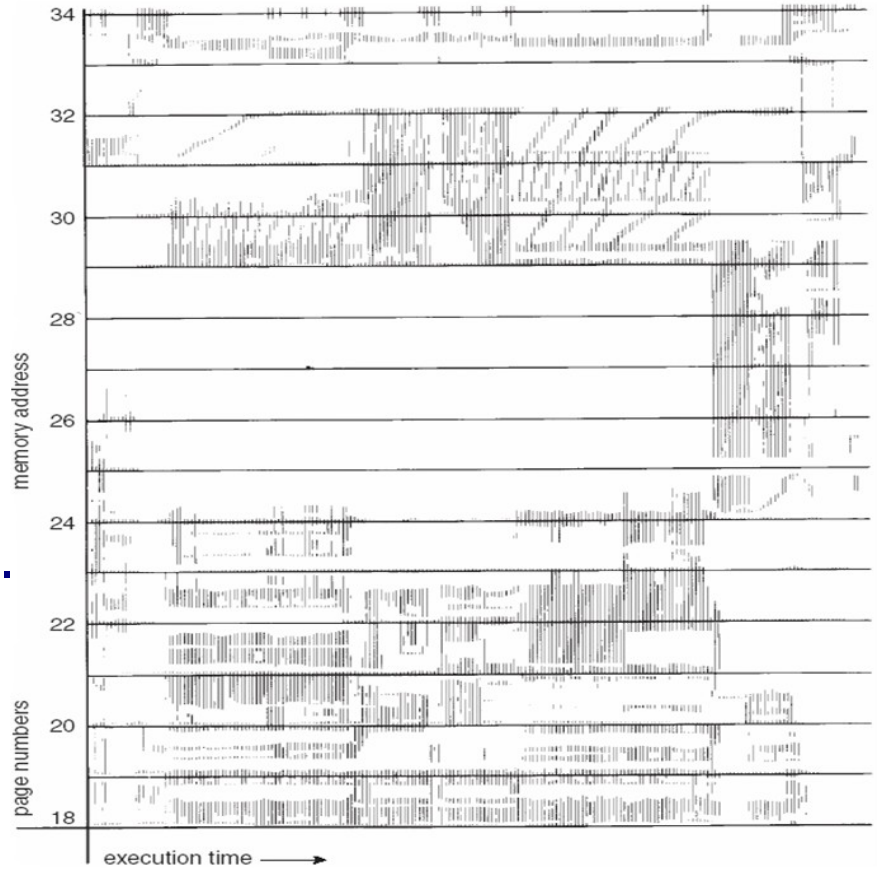
- Perché la paginazione funziona?
- Modello di **località**:
  - Località → insieme di pagine usate attivamente in maniera (quasi) contemporanea.
  - Il processo passa da una località ad un'altra.
  - Le località possono essere sovrapposte.

- Perché avviene il thrashing? → **Dimensione località > dimensione della memoria disponibile per il processo.**



# Località in una sequenza di riferimenti a memoria

- Le località hanno una caratteristica **spazio–temporale**.
- Quando viene chiamata una funzione, si definisce una nuova località: vengono fatti riferimenti alle sue istruzioni, alle sue variabili locali ed a un sottoinsieme delle variabili globali.
- Quando la funzione termina, il processo lascia la località corrispondente.
- Le località sono definite dalla struttura del programma e dalle strutture dati relative

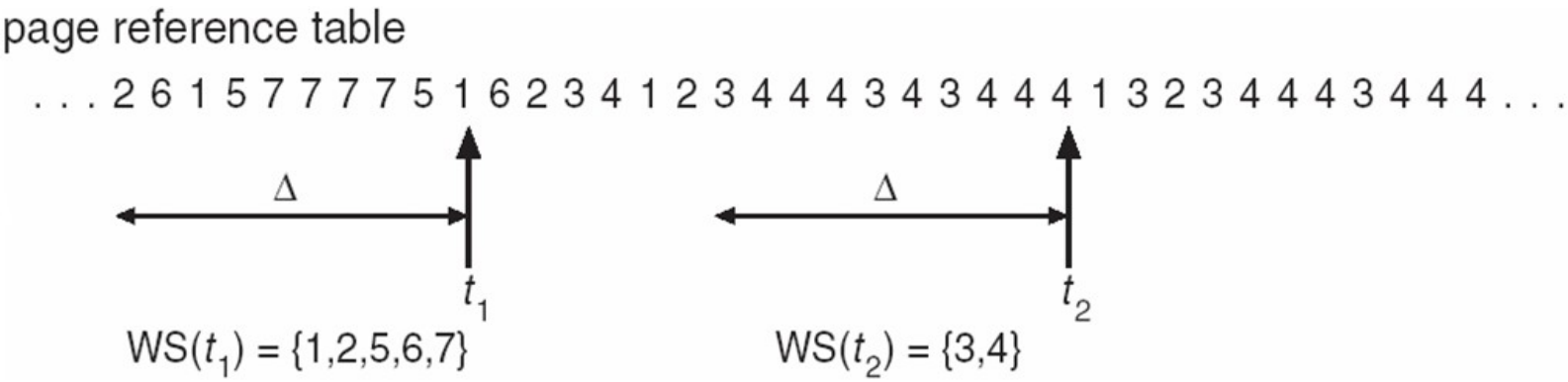



Idea di base per evitare il thrashing: fornire un numero tale di frame ad un un processo sufficiente per allocare le sue località

# Modello Working–Set (1/3)

- Working–set: approssimazione delle località
- $\Delta \rightarrow$ finestra di working–set = un numero fisso di riferimenti (accessi) a pagina, esempio: 10.000 istruzioni **consecutive**.
- **WSS** (working–set del processo P) = numero di pagine i ireferenziate nel più recente $\Delta$ (varia col tempo):
  - Se $\Delta$ è troppo piccolo non comprende tutta la località.
  - Se $\Delta$ è troppo grande comprenderà più località.
  - Se $\Delta = \infty \rightarrow$ comprende l'intero programma.
- $D = \Sigma$ **WSS**$_i \equiv$ numero totale di blocchi richiesti dai processi in un dato istante.

# Modello Working–Set (2/3)



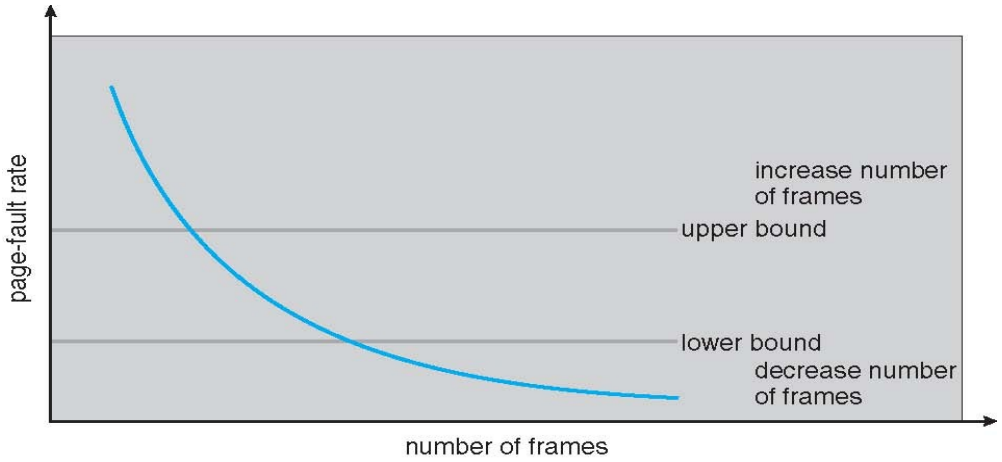

- Se Σ **WSS**$_i$ > m (numero totale dei frame liberi) → **Thrashing**
  - **Soluzione**: il S.O. sospende un processo o libera alcuni frame (swap-out).

# Modello Working–Set (3/3)

- **Problema** working set: calcolo del WSS complesso e dispendioso
- **Soluzione approssimata**: Si approssima con un interrupt del timer e un bit di riferimento
  - Si supponga di settare il timer con interrupt ogni 5000 unità di tempo e di tenere in memoria 2 bit per ogni pagina.
  - Quando si ha un interrupt del timer, **si copiano i valori di tutti i bit di riferimento e si pongono a 0**.
  - Se uno dei bit in memoria è 1 → pagina nel working-set.
- Tale soluzione approssimata è poco accurata perché non indica **dove** è avvenuto l'accesso nell'intervallo di 5000 riferimenti.
- Miglioramento: 10 bit e interruzioni ogni 1000 unità di tempo.

# Frequenza di page fault

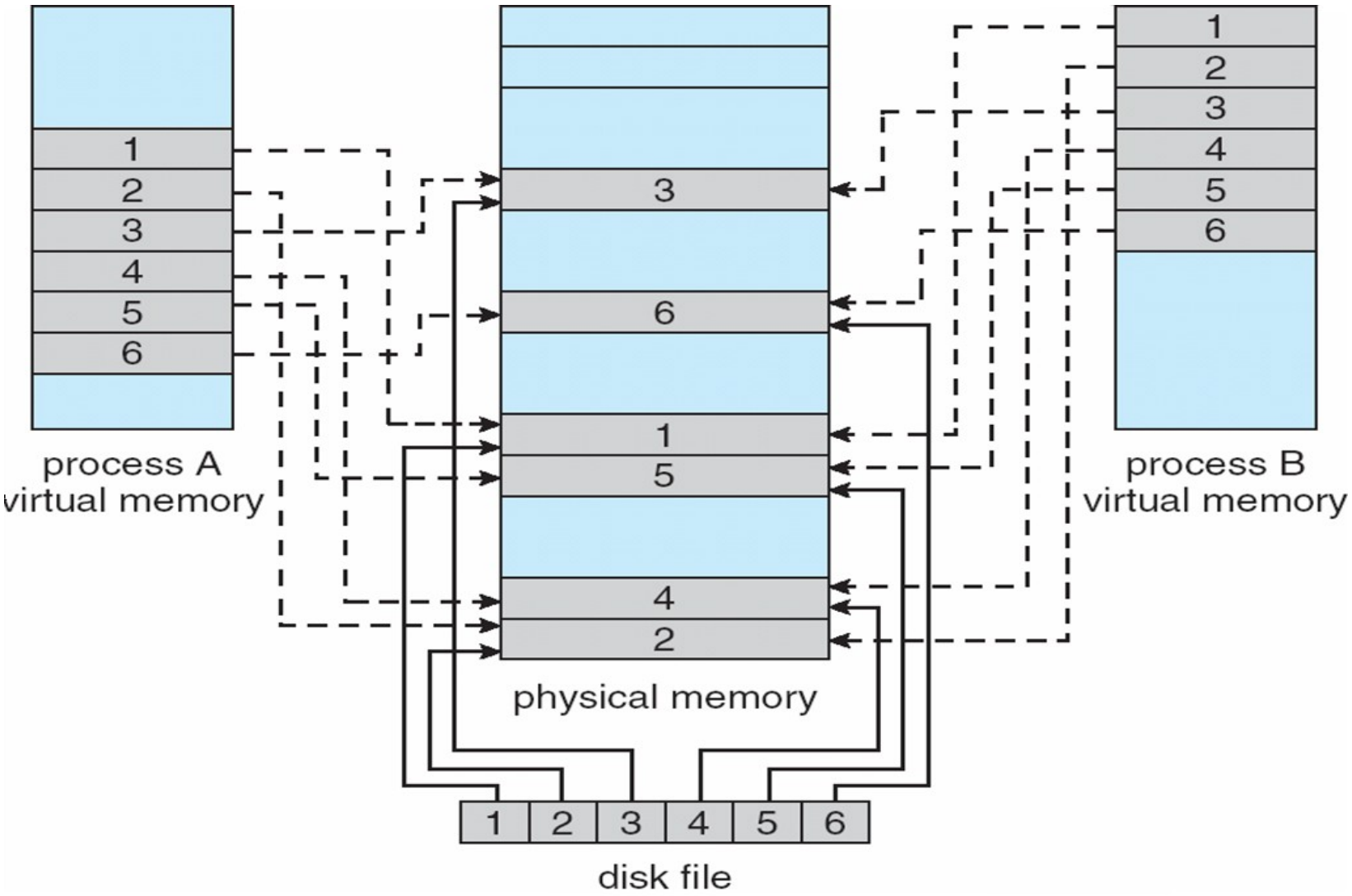
- Soluzione alternativa a working sets, l'idea è di contare i page fault e di alzare o abbassare il numero di frame allocati di conseguenza.
- Si stabilisce una frequenza di page fault “accettabile”.
  - Se la frequenza effettiva è troppo bassa, il processo rilascia dei blocchi.
  - Se la frequenza è troppo elevata, il processo acquisisce nuovi blocchi.



# File memory mapped (1/3)

- L'I/O su file mappati in memoria permette di trattare le operazioni di I/O come ordinari accessi alla memoria, mappando un blocco del disco su una pagina della memoria.
- Un file è letto inizialmente usando la paginazione su richiesta; una parte del file della dimensione di una pagina è prelevata dal disco e memorizzata su una pagina della memoria.
- **Le successive operazioni di lettura e scrittura su file vengono trattate come accessi alla memoria**.
- Tale operazione **velocizza** e **semplifica** l'accesso al file, trattando tali operazioni come I/O attraverso la memoria, piuttosto che chiamate a read() e write().
- Permette anche a più processi di accedere allo stesso file, mediante la condivisione delle pagine della memoria.

# File memory mapped (2/3)



# File memory mapped (3/3)

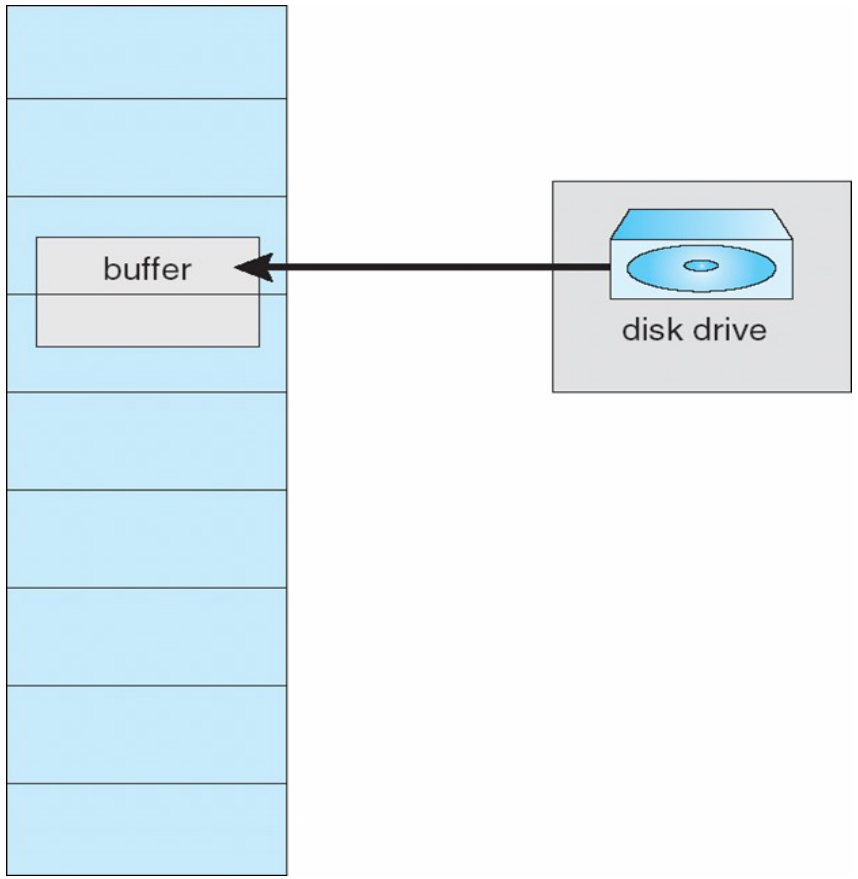
- Quando devono essere scritti i file in memoria?
  - Ovviamente, quando il file viene chiuso
  - Periodicamente il sistema operativo scandisce le pagine mappate a disco, copiando quelle modificate (dirty = 1)
- Alcuni sistemi operativi (es. Linux) effettuano sempre operazioni di lettura e scrittura con system call convenzionali read() e write() in modalità memory mapped.

# Prepaginazione

- Tecnica per velocizzare le operazioni effettuate dopo un context switch.
- **Idea**: Si portano in memoria tutte le pagine richieste in una volta sola. Ad esempio si può memorizzare il working–set al momento della sospensione per I/O per poi riprendere tutte le pagine che gli appartengono.

# I/O Interlock

- In alcuni casi è necessario che alcune pagine **siano sempre mantenute in memoria, e non copiate a disco** (swap-out)
- Le pagine utilizzate per copiare un file da un dispositivo di I/O devono essere bloccate, affinché non possano essere selezionate come vittime da un algoritmo di sostituzione.
- Possibili problemi:
  - **Incoerenza**
  - Performance



# Considerazioni pratiche

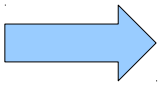
- Il programmatore deve tenere conto delle questioni legate alla località degli accessi, limitando accessi vicini nel tempo ad aree distanti fra di loro.
- Si abbia ad esempio un sistema con pagine da 128 parole, e si consideri l'array bidimensionale:

**int A[128][128];** // Memorizzato per righe

//A[0][0], A[0][1],...

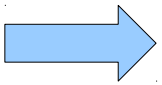
- Si considerino i due frammenti di codice che compiono la stessa operazione:

```
for (j = 0; j < 128; j++)
      for (i = 0; i < 128; i++)
            A[ i ][ j ] = 0;
```



Ogni lettura consecutiva afferisce ad una pagina diversa, al più 128 * 128 = **16384 page fault**

```
for (i = 0; i < 128; i++)
      for (j = 0; j < 128; j++)
            A[ i ][ j ] = 0;
```



Lettura consecutiva all'interno di ogni pagina, al più **128 page fault**