

# Memoria Virtuale

# Sommario

---

- Virtualizzazione della memoria
- Algoritmi di sostituzione delle pagine
- Allocazione dei frame
- Trashing
- Segmentazione su richiesta
- Conclusioni

# Virtualizzazione della memoria

---

- Le strategie viste fino ad ora hanno avuto come scopo quello di mantenere in memoria più programmi contemporaneamente per aumentare il livello di multiprocessing.
- Tutte le tecniche viste, si basano comunque sull'ipotesi che **tutto il programma sia in memoria centrale**.
- Si parla di **memoria virtuale** quando non tutto il programma è nella memoria RAM della macchina.
- Ovviamente, per poter eseguire una certa istruzione del programma, ad un dato istante deve essere in memoria fisica sia la porzione di codice che contiene l'istruzione da eseguire ed anche i dati che l'istruzione ha bisogno di utilizzare.

# In sostanza ...

---

- Nella memoria virtuale viene virtualizzato uno spazio di memoria centrale maggiore di quello fisicamente presente o disponibile.
- Per farlo:
  - Si sfutta la separazione tra indirizzi logici e fisici (preesistente)
  - si utilizza uno spazio in memoria secondaria (su dischi veloci) su cui vengono mantenute parte dei processi che non sono indispensabili immediatamente all'esecuzione.

# Vantaggi

---

- Alcuni vantaggi:
  - La dimensione del programma non è vincolata alla dimensione della memoria fisica
  - Poiché ogni programma può utilizzare meno memoria (in ogni istante di calcolo), il numero di programmi che possono essere caricati contemporaneamente aumenta.
  - Diminuisce lo swapping di interi programmi (dovuto ad esempio all'I/O) per cui il tempo di esecuzione migliora.

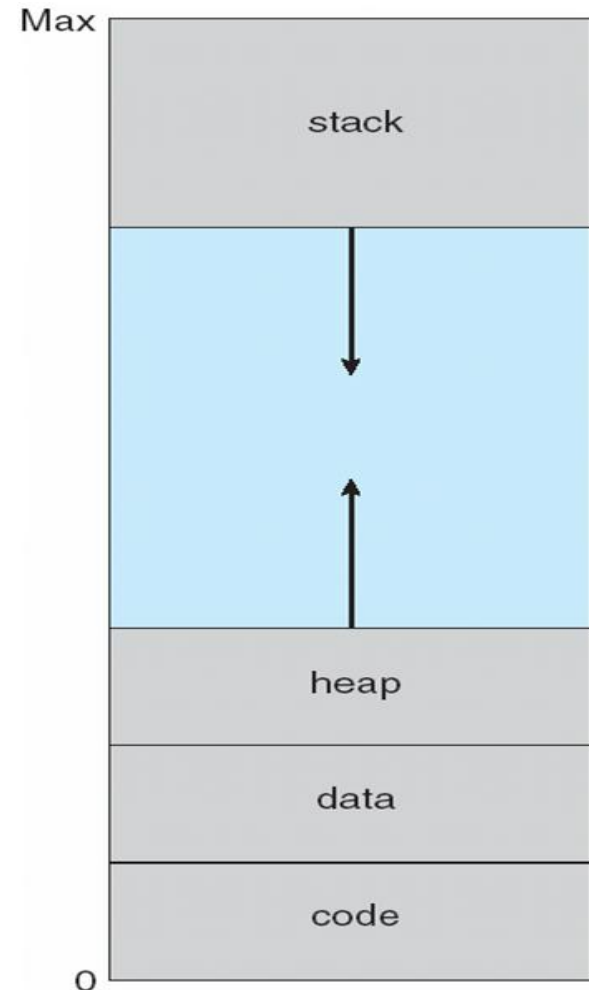
# Vantaggi

---

- Altri vantaggi:
  - Codici complessi possono girare su macchine che hanno dotazioni di memoria povere.
  - Il programmatore non deve più occuparsi della gestione della memoria poiché può utilizzarne quanta vuole.
  - Parti del codice che gestiscono condizioni insolite potrebbero non essere mai caricate.
  - La condivisione di spazi degli indirizzi è semplificata

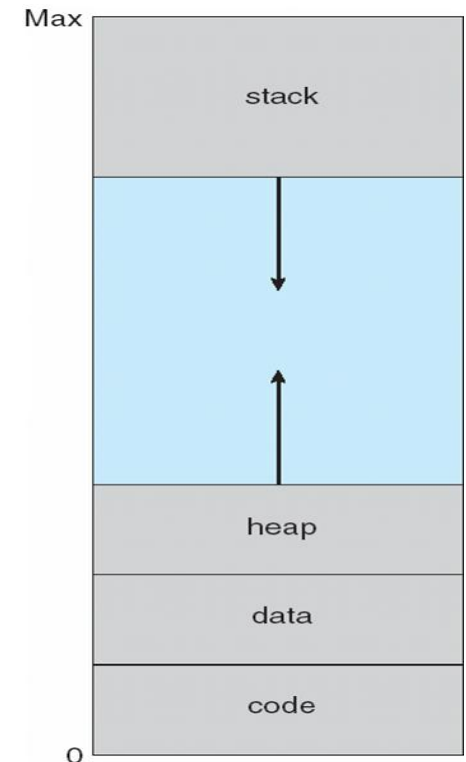
# Indirizzi virtuali

- La memoria virtuale si basa sulla separazione dello spazio degli indirizzi logici da quello degli indirizzi fisici
- Con **spazio degli indirizzi** virtuale si intende la collocazione dei processi dal punto di vista logico (o virtuale).
- Lo spazio degli indirizzi del processo inizia dall'indirizzo 0 e si estende contigualmente a contenere
  - Codice
  - Dati
  - Heap
  - Stack



# Indirizzi virtuali

- Alla memoria del processo è lasciato spazio per crescere:
  - Allo heap che contiene la memoria allocata dinamicamente è lasciato spazio per crescere verso l'alto
  - Allo stack usato per le chiamate a funzione è lasciato spazio per crescere verso il basso
- Nel disegno esiste uno spazio vuoto, detto buco che consente a stack e heap di crescere
- Lo spazio degli indirizzi virtuali con questo tipo di vuoto è detto **sparso**



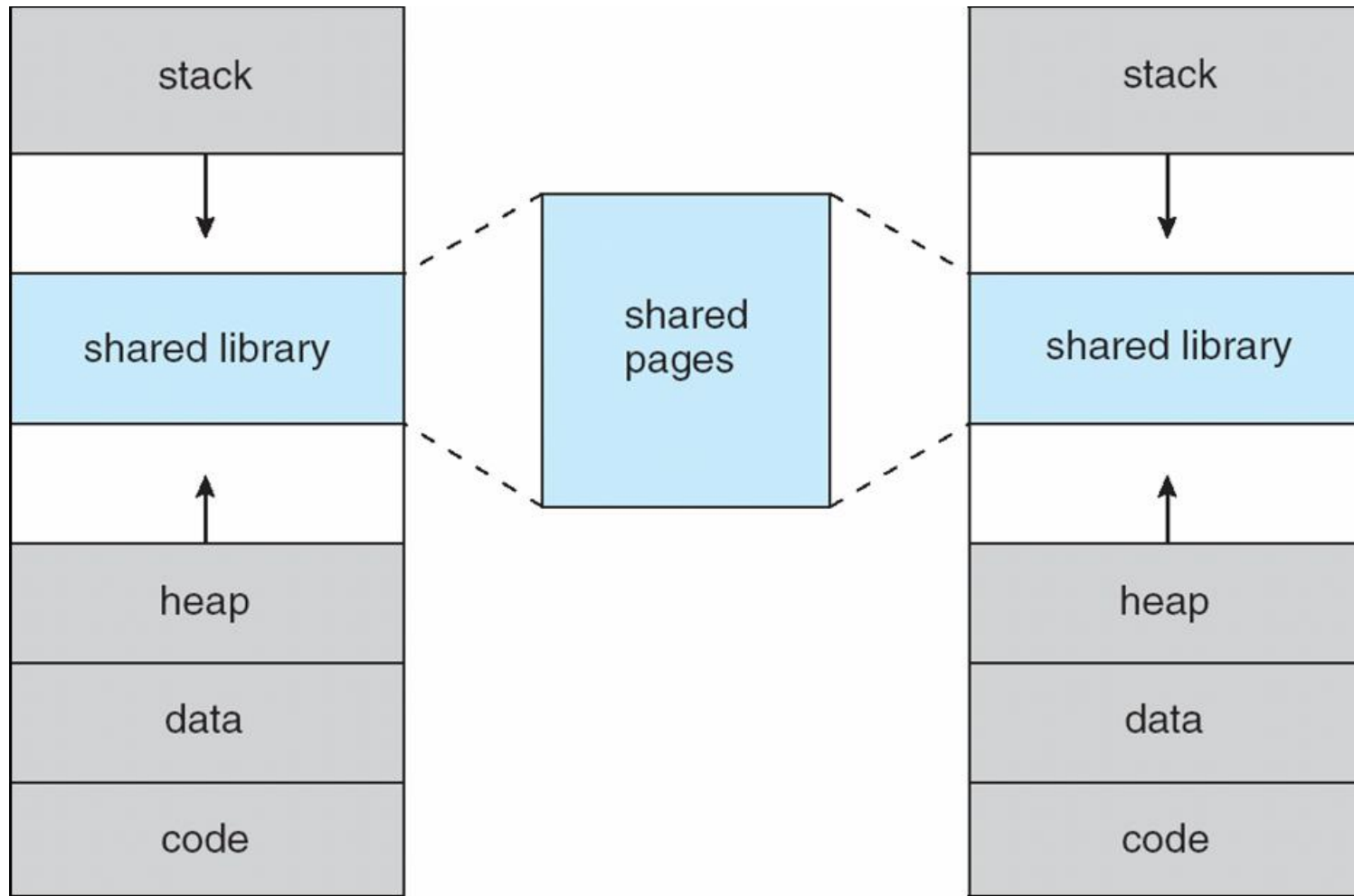


# Indirizzi virtuali

---

- Lo spazio vuoto, oltre alla crescita di heap e stack, consente anche di gestire le librerie caricate dinamicamente e la memoria condivisa
- Viene utilizzato un procedimento detto **mappatura** che consente:
  - al processo di vedere librerie caricate dinamicamente e memoria condivise come parte del proprio spazio degli indirizzi
  - Al sistema di allocare la memoria condivisa una volta sola

# Indirizzi virtuali



# Implementazione

---

- La memoria virtuale è tipicamente implementata mediante una tecnica di **paginazione su richiesta (demand paging)**.
- Generalmente i sistemi offrono anche uno schema di segmentazione paginata, per cui l'utente vede i segmenti e il SO vede e gestisce le pagine.
- Raramente sono utilizzati meccanismi di memoria virtuale basati solo su segmentazione su richiesta.

# Implementazione

---

- La paginazione su richiesta combina le tecniche di paginazione con quelle di swapping:
  - I programmi risiedono in memoria secondaria (generalmente un disco veloce).
  - Per eseguire vengono caricati in memoria centrale, per pagine.
  - Ogni pagina è caricata solo quando è richiesta dall'esecuzione del codice.

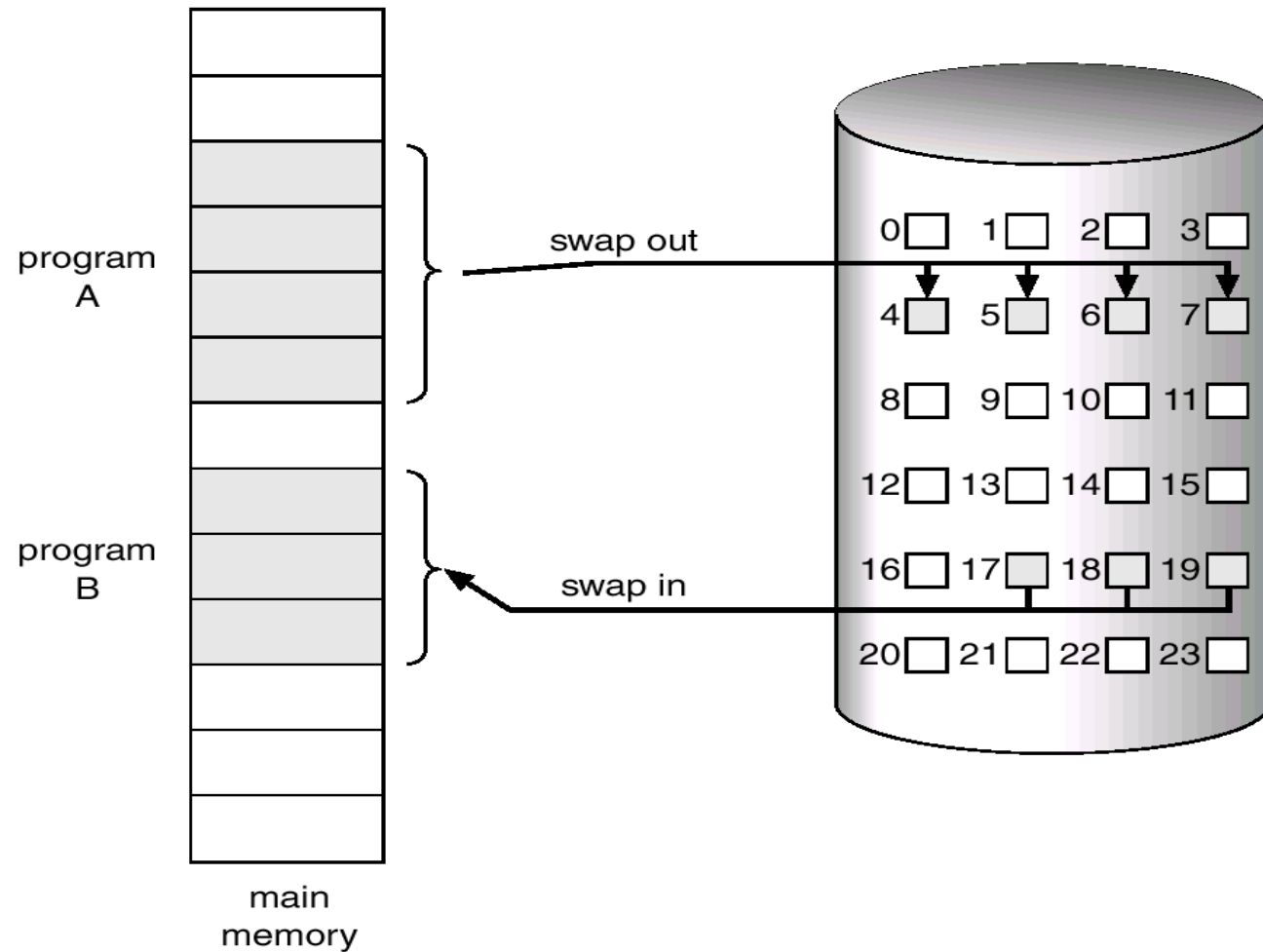
# Implementazione

---

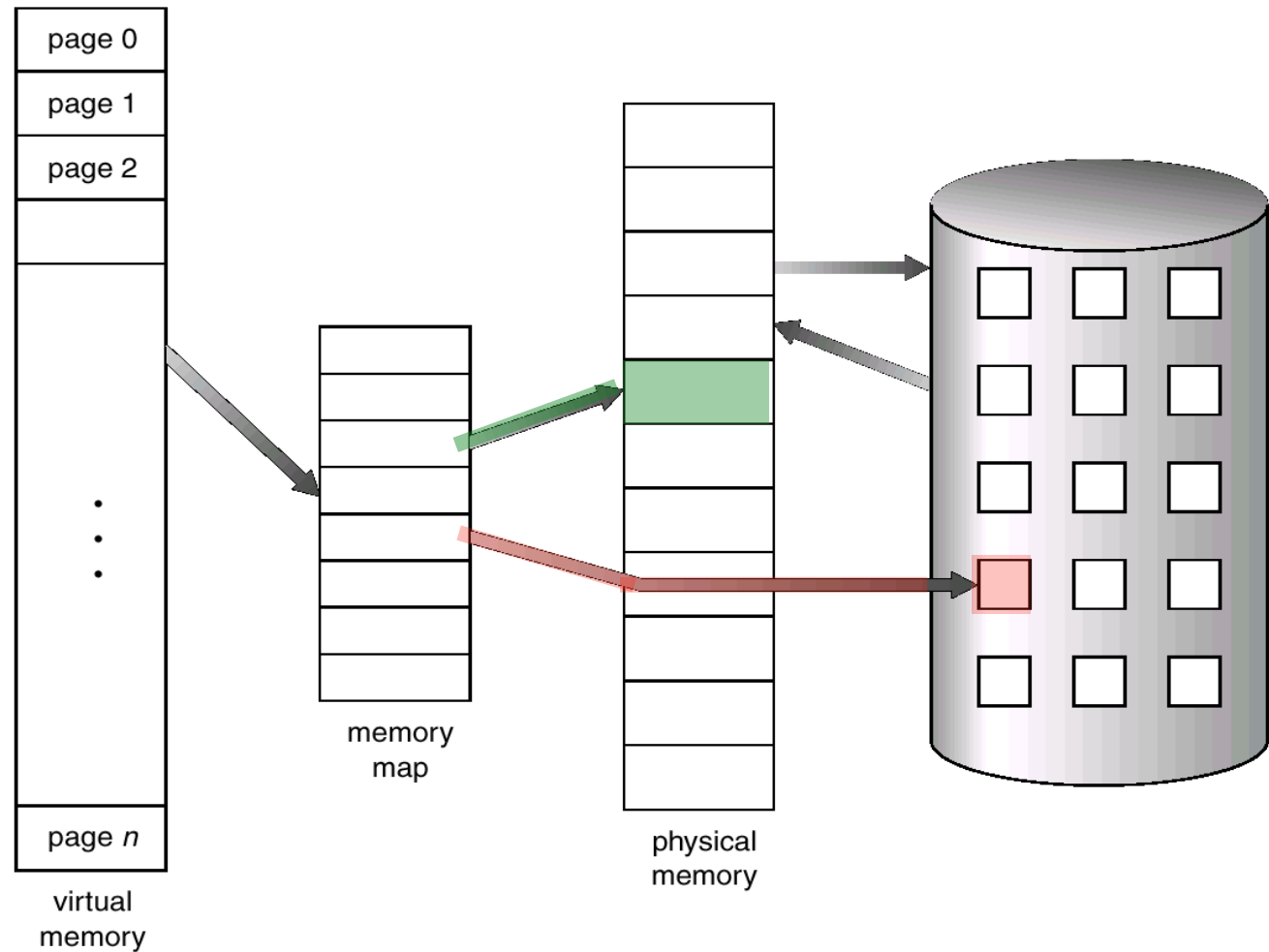
- La componente del SO che si occupa della paginazione a richiesta si chiama **pager** e deve:
  - **Swap-in**: Caricare le pagine richieste che non risiedono già in memoria centrale.
  - **Swap-out**: Decidere quale pagina presente in memoria centrale scaricare quando arriva una richiesta di swap-in e la memoria è tutta occupata.
- Nei sistemi a memoria virtuale lo swap è effettuato pagina per pagina, non processo per processo.

# Origine del termine Swap.

## Swapping del processo



# Swapping della pagina



# Bit di validità

0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H

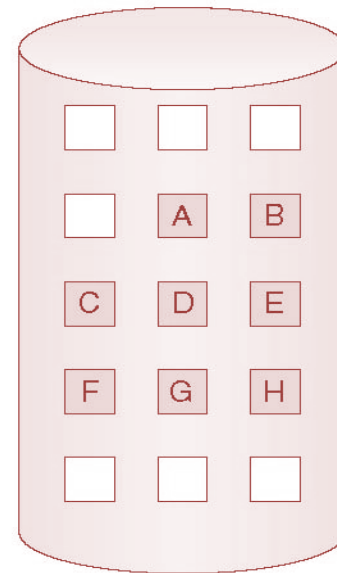
logical  
memory

valid-invalid bit	
frame	bit
0	4 v
1	i
2	6 v
3	i
4	i
5	9 v
6	i
7	i

page table

0	
1	
2	
3	
4	A
5	
6	C
7	
8	
9	F
10	
11	
12	
13	
14	
15	

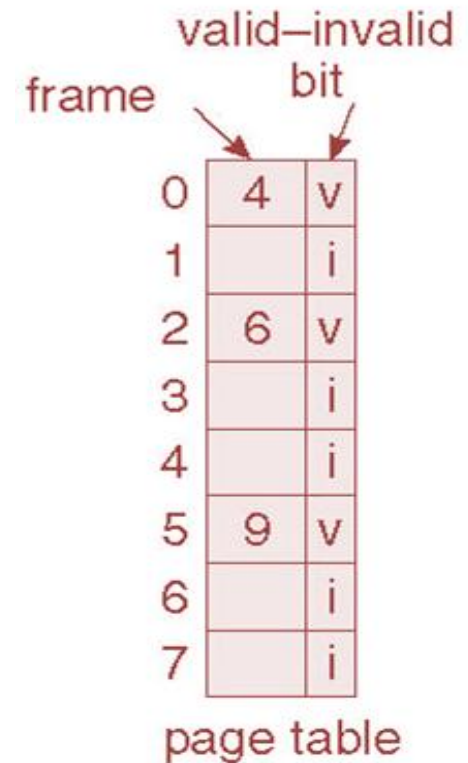
physical memory





# Page Fault

- Quando il processo **richiede un indirizzo**:
  - Se tenta di accedere a un indirizzo con **bit di validità 1**, procede come nella paginazione normale, la pagina c'è e l'indirizzo logico viene convertito in indirizzo fisico.
  - Se tenta di accedere a un indirizzo con **bit di validità 0**, viene generato un trap denominato **page fault** che può essere dovuto ad un errore nella gestione degli indirizzi oppure alla necessità di caricare in memoria la pagina mancante.



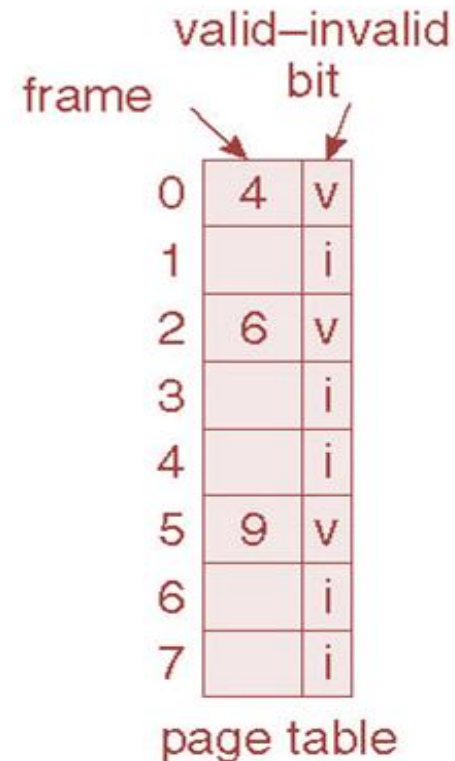
The diagram shows a page table with 8 rows, indexed 0 to 7. Each row has two columns: 'frame' and 'valid-invalid bit'. Arrows point from the column headers to their respective columns. The 'frame' column contains the values 4, 6, and 9 for rows 0, 2, and 5 respectively, and is empty for rows 1, 3, 4, 6, and 7. The 'valid-invalid bit' column contains 'v' for rows 0, 2, and 5, and 'i' for rows 1, 3, 4, 6, and 7. The entire table is labeled 'page table' at the bottom.

	frame	valid-invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

page table

# Page Fault

- Riassumendo:
  - Bit di validità 1 (v), pagina in memoria
  - Bit di validità 0 (i), pagina NON in memoria
  - Inizialmente il valore del bit di validità è 0 (i) per tutte le pagine: nessuna è caricata
- Durante la traduzione degli indirizzi effettuata dalla MMU se il bit di validità è 0 (i) viene generato un page fault, che come tutti i trap interrompe l'esecuzione del processo e produce l'esecuzione del gestore delle interruzioni.



The diagram shows a page table with 8 entries (frames 0 to 7). Each entry contains a frame number and a valid-invalid bit. The valid-invalid bit is labeled 'v' for valid and 'i' for invalid. The frame number is labeled 'frame' and the bit is labeled 'bit'. The page table is labeled 'page table' at the bottom.

	frame	valid-invalid bit
0	4	v
1		i
2	6	v
3		i
4		i
5	9	v
6		i
7		i

# Gestione del Page Fault

---

- La procedura di gestione del **page fault** (avviata come gestore delle interruzioni) è la seguente:
  1. Viene controllata la tabella interna del processo per stabilire se l'indirizzo logico è valido.
  2. Se il riferimento non era valido il processo viene terminato. Altrimenti se il riferimento è valido viene iniziato il trasferimento della pagina.

# Gestione del Page Fault (segue)

---

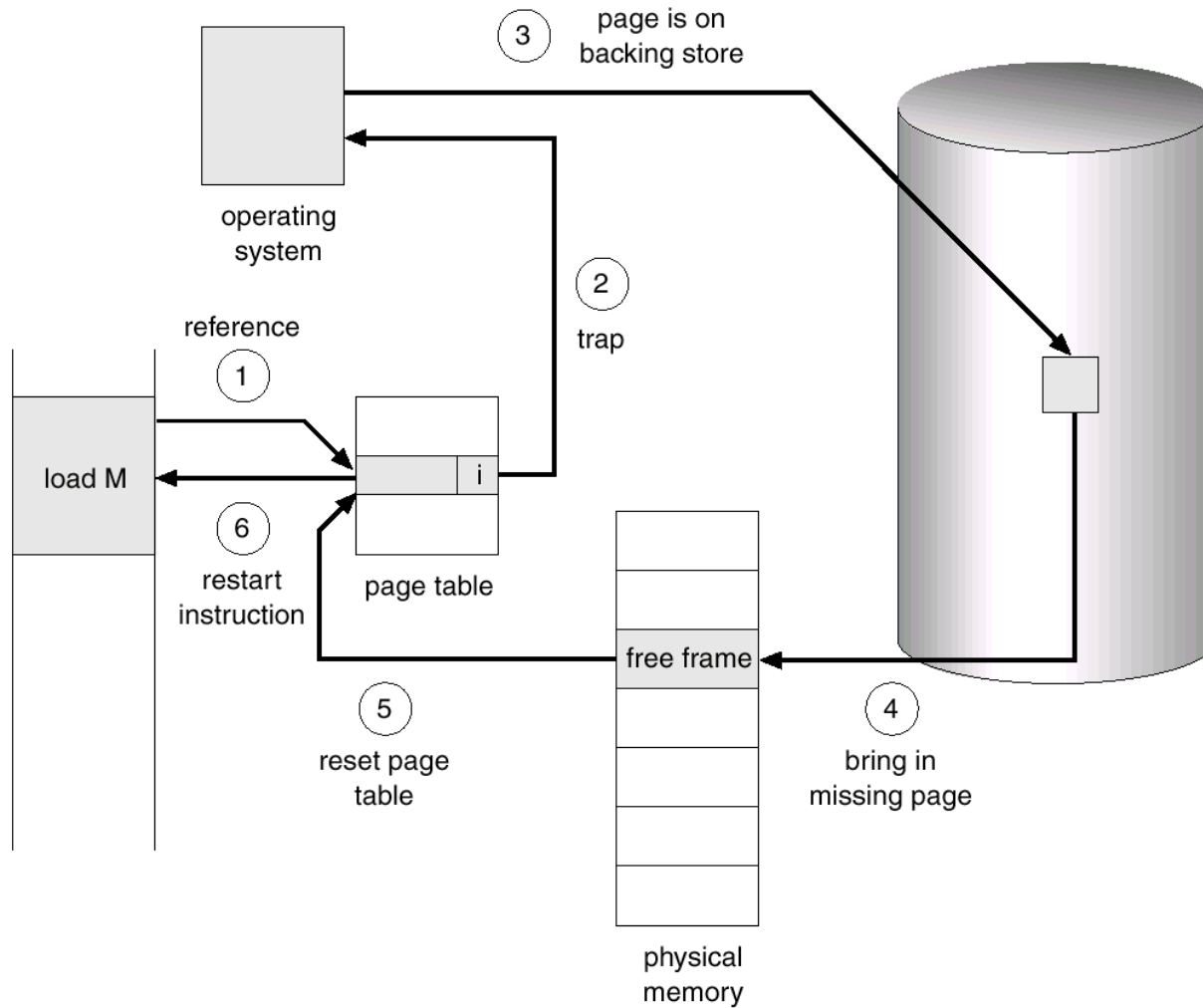
3. Viene individuato un frame libero.
4. Viene caricata la pagina mancante, dal disco al frame scelto.
5. Viene modificata la tabella interna del processo e la tabella generale delle pagine.
6. Viene riavviata l'esecuzione interrotta con il trap di page fault. A questo punto il programma può accedere all'indirizzo che è in memoria centrale.

# On demand puro

---

- È possibile avviare l'esecuzione di un processo senza pagine in memoria:
  - Quando il SO carica nel program counter l'indirizzo della prima istruzione, genera un primo page fault
  - Una volta portata la prima pagina in memoria, le altre vengono caricate quando sono indirizzate perché contengono dati o codice.
- Questo schema di esecuzione è detta **paginazione su richiesta pura**

# Page Fault



# Hardware

---

- L'**hardware** richiesto per la gestione della memoria virtuale tramite demand paging è dunque sostanzialmente lo stesso necessario per la **paginazione** e lo **swapping**:
  - **Tabella delle pagine**, per il reperimento delle pagine, a cui è aggiunto il bit di validità.
  - **Backing store**, realizzato su supporti di memoria secondaria, per la memorizzazione stabile delle pagine
  - **Apposita MMU**.

# Page Fault

---

- Il **page fault** (come tutti i trap e gli interrupt):
  - Interrompe l'esecuzione del processo;
  - Induce il salvataggio dello stato del processo (registri, program counter, ecc.)
  - Avvia la routine di gestione del trap.
- Al termine dell'esecuzione della routine di gestione il processo può essere riavviato esattamente dalla istruzione macchina in cui si era interrotto, con la differenza che la locazione richiesta (o meglio la pagina in cui essa è contenuta) è stata caricata in memoria.



# Accessi

---

- Un programma può accedere a più locazioni di memoria con una sola istruzione, ovvero può provocare più di un page fault (e conseguentemente più di una interruzione) per ogni istruzione.
- Questo fenomeno è in realtà molto limitato grazie al fatto che i programmi tendono ad avere **località di riferimento**, ovvero ad allocare in posizioni vicine, riferimenti “vicini”.

# Prestazioni

---

- La paginazione su richiesta può avere un effetto notevole sulle **prestazioni** di un SO:
  - Se la pagina è in memoria il tempo di accesso corrisponde al tempo di accesso alla memoria.
  - Se la pagina non è in memoria e si verifica il page fault, il tempo di accesso include il tempo di caricamento della pagina in memoria e il tempo di accesso alla memoria.

# Prestazioni

---

- Supponiamo che **p** sia la probabilità di avere un page fault. Ovviamente  $0 \leq p \leq 1$ .
- Indichiamo con **ma** il tempo di accesso in memoria e con **pf** il tempo di accesso con page fault.
- Allora il tempo di accesso effettivo medio sarà:  
$$(1-p) \times \mathbf{ma} + p \times \mathbf{pf}$$
- Il tempo di accesso effettivo dipende dal tempo di gestione del page fault e dalla frequenza dei page fault.

# Eventi scatenati dal page fault (1/3)

---

- La sequenza di azioni che vengono effettuate in seguito al page fault è la seguente:
  1. Viene generato un trap
  2. Vengono salvati i registri e lo stato
  3. Viene determinata la natura del fault
  4. Viene controllata la validità del riferimento

.....

# Eventi scatenati dal page fault (2/3)

---

5. Viene caricata la pagina sul frame libero:
  - Attesa che la richiesta di accesso al disco sia servita (scheduling del disco)
  - Attesa del posizionamento e/o latenza
  - Inizio trasferimento della pagina sul frame.
6. Durante l'attesa la CPU può essere rischedulata ad un altro processo
7. Interrupt per l'I/O del disco completato
8. Salvataggio dei registri dell'altro processo in esecuzione

# Eventi scatenati dal page fault (3/3)

---

9. Determinazione della natura dell'interrupt
10. Correzione della tabella delle pagine
11. Attesa che la CPU venga di nuovo allocata a questo processo
12. Recupero dello stato del processo e dei suoi registri e ripresa dell'esecuzione interrotta.

# Morale

---

- In un sistema con memoria virtuale basata su paginazione su richiesta è importante **tenere basso il numero dei page fault** altrimenti il tempo effettivo di accesso aumenta a dismisura e si rallenta notevolmente l'esecuzione dei processi.
- E' cruciale **scegliere bene il meccanismo di sostituzione** delle pagine!

# Prestazioni

---

- la paginazione su richiesta ha anche **effetti positivi**:
  - Se un processo utilizza solo una parte delle pagine, la paginazione su richiesta consente di eliminare i tempi di caricamento delle pagine che non verranno utilizzate.
  - Questo significa poter aumentare il grado di multiprogrammazione e quindi anche l'utilizzo e il throughput della CPU.



# Copiatura su scrittura

---

- La chiamata alla fork genera un processo figlio che usa una copia dello spazio degli indirizzi del processo genitore, generato duplicando le pagine del processo genitore.
- Solitamente però lo spazio degli indirizzi viene sostituito con una exec e dunque la generazione di un processo figlio produce a stretto giro un page fault e rende inutile la copiatura fatta in precedenza.

# Copiatura su scrittura

---

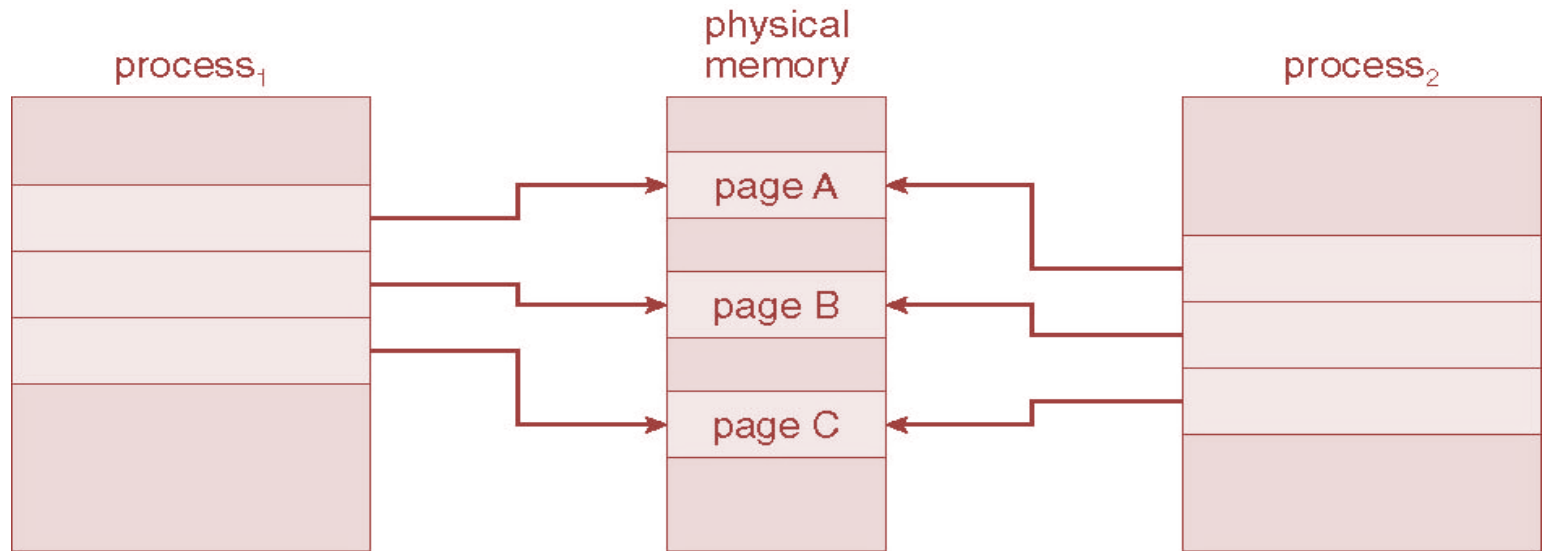
- Per evitare questo fenomeno si utilizza una tecnica detta **copiatura su scrittura** (copy on write) basata sulla iniziale condivisione delle pagine da parte del processo padre e del processo figlio.
- Le pagine condivise vengono marcate come pagine da copiare (duplicare) in caso di scrittura, ad indicare che se/quando uno dei due processi tenta una scrittura, allora i due spazi degli indirizzi si devono separare ovvero il SO deve fare una copia della pagina per ciascuno dei due processi e modificare quella opportuna.

# Copiatura su scrittura

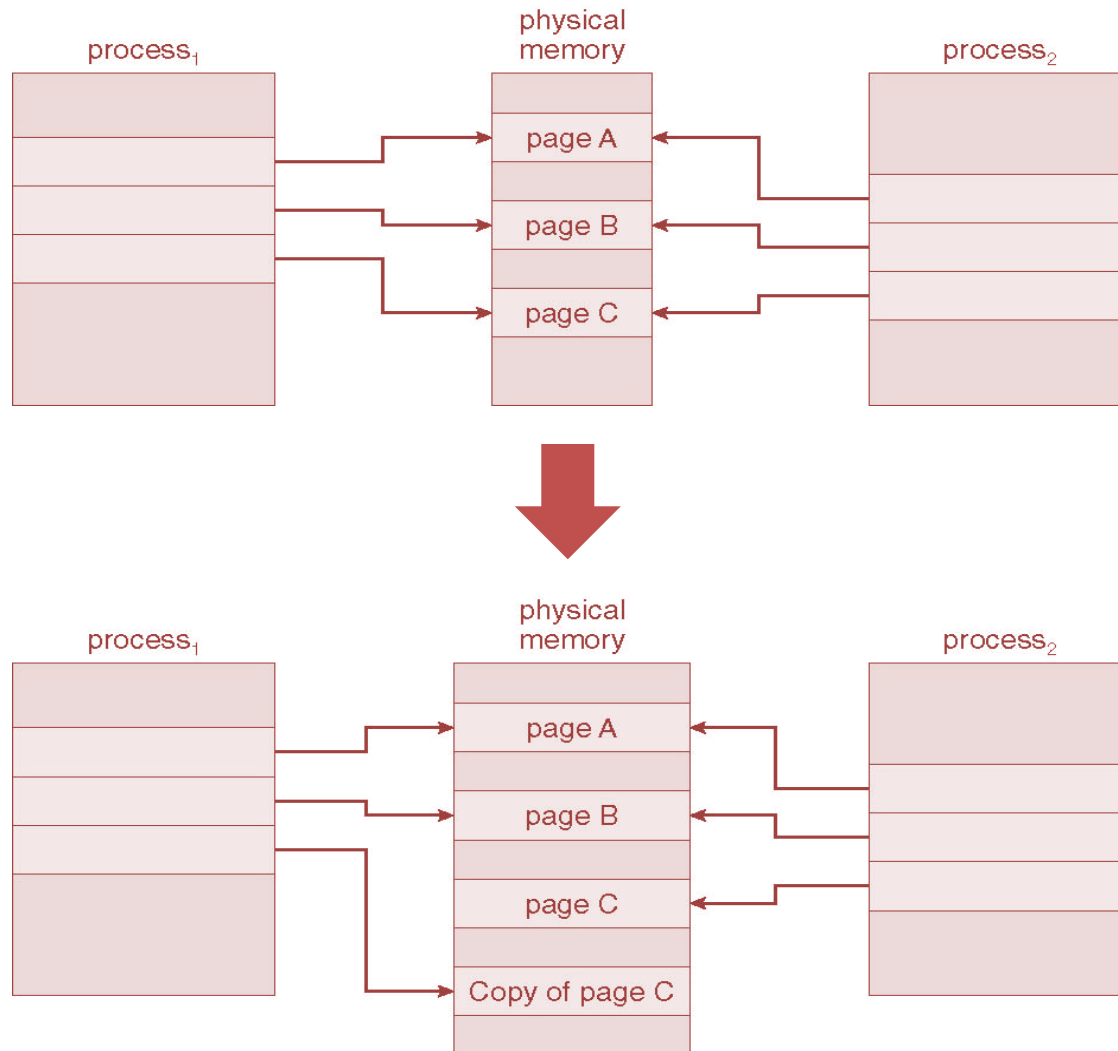
---

- Per esempio se il processo figlio tenta di modificare una pagina dello stack, il SO considera la pagina (marchiata come copy on write) da copiare nello spazio degli indirizzi del processo figlio.
- Il processo figlio modifica la sua copia della pagina e non quella del processo padre
- È importante capire dove recuperare la pagina libera necessaria per il copy on write; di solito **si usano pagine prelevate da un pool** che il SO tiene libere per questo tipo di esigenze

# Copiatura su scrittura



# Copiatura su scrittura



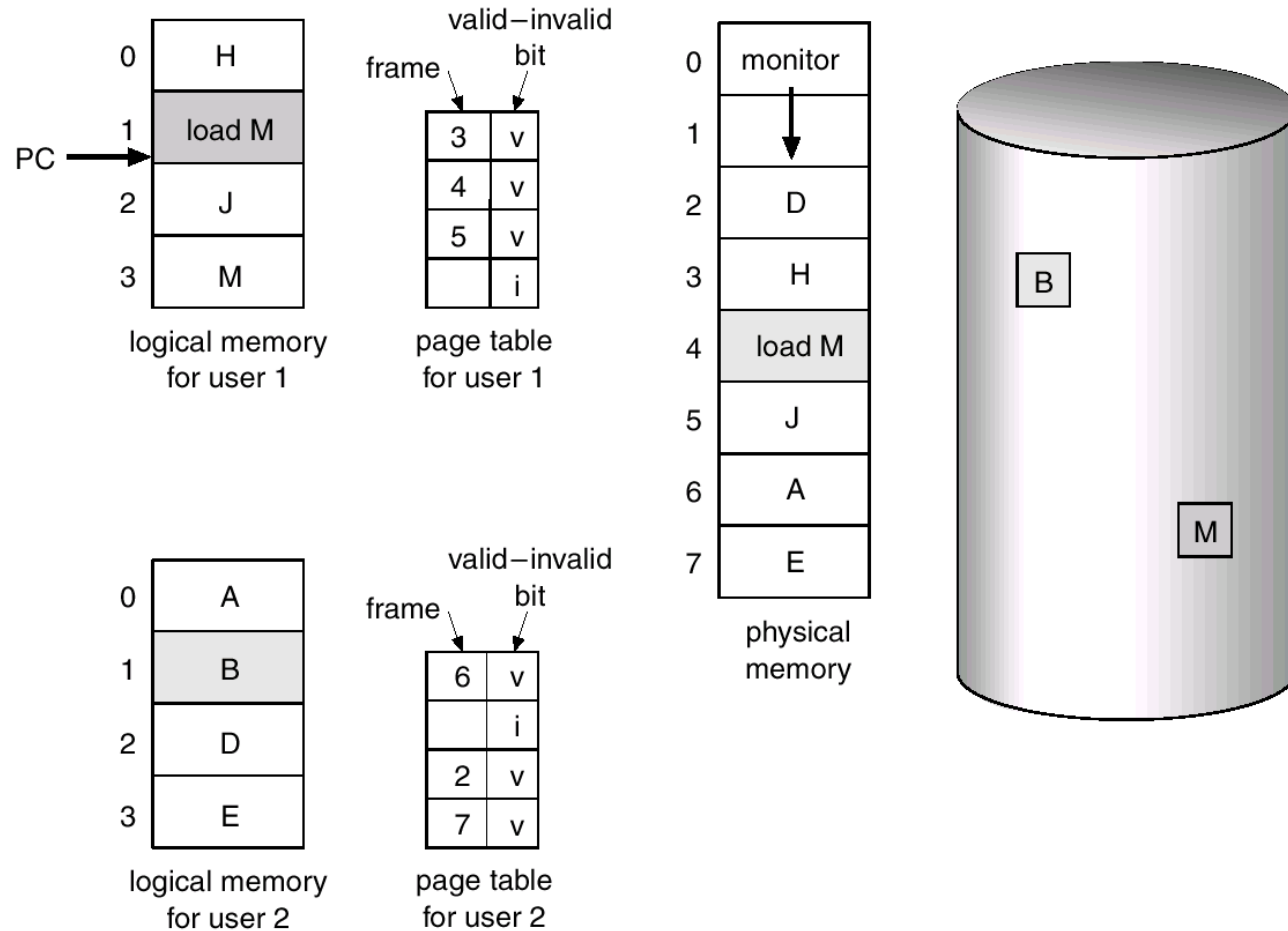
# Sostituzione delle pagine

---

- Se nessun frame è libero è possibile liberarne uno occupato scaricando il suo contenuto dalla memoria centrale alla backing store.
- La **sostituzione delle pagine** (una è stata scaricata – **swap out** - e una è stata caricata – **swap in** -) deve poi essere registrata nelle tabella delle pagine.

# Rimpiazzamento delle Pagine

## Page Replacement

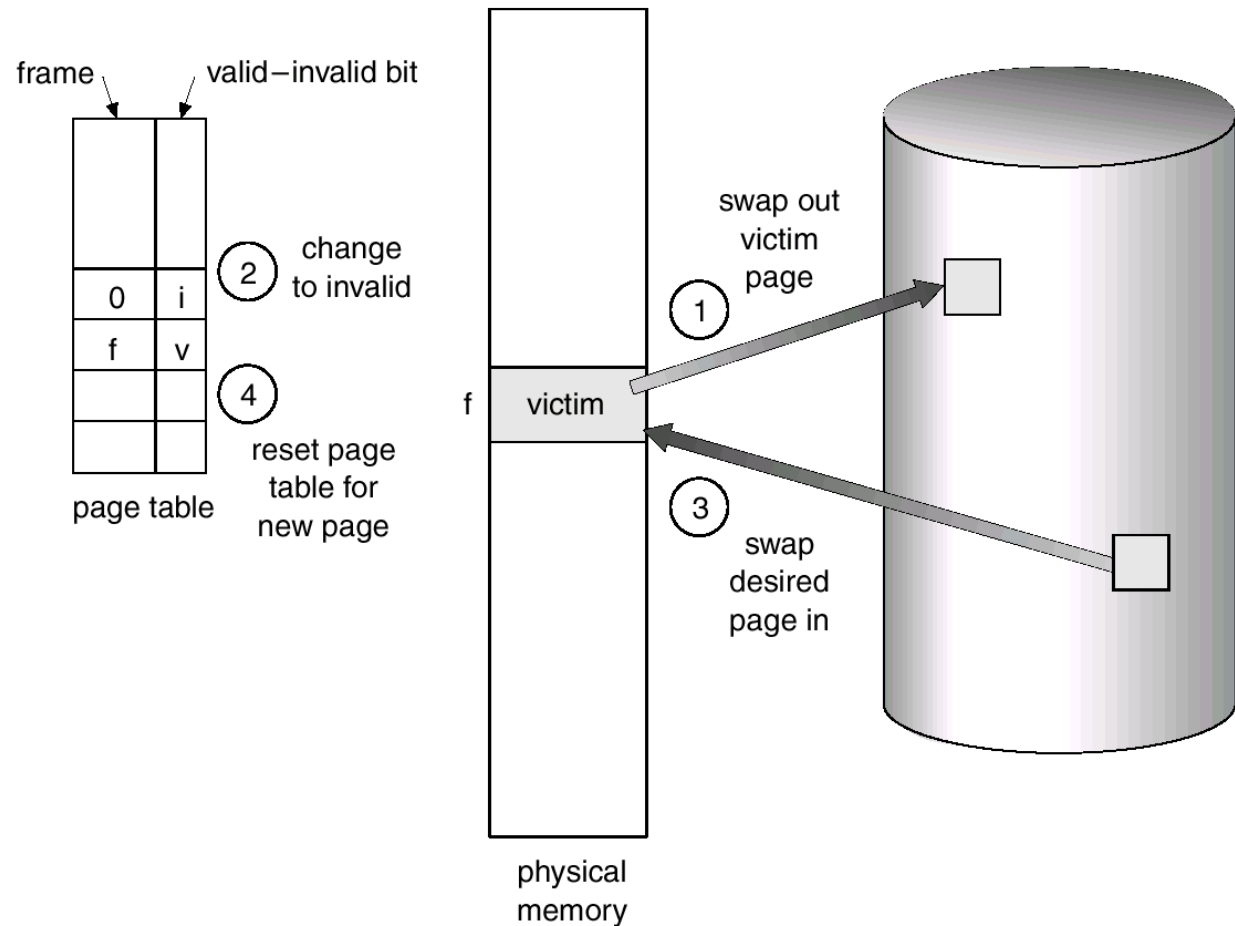


# Sostituzione delle pagine

- La sostituzione delle pagine avviene quindi nel modo seguente:
  1. Individuazione della locazione richiesta
  2. Individuazione del frame libero
    - a. Se esiste un frame libero viene utilizzato.
    - b. Altrimenti viene selezionata una **vittima** utilizzando un apposito algoritmo
    - c. La vittima viene scaricata (cioè scritta su disco) e vengono opportunamente aggiornate le tabelle
  3. La pagina richiesta viene carica e vengono opportunamente aggiornate le tabelle
  4. Viene riavviato il processo utente.



# Page Replacement



# Dirty bit

---

- Lo scaricamento può essere evitato se la pagina da scaricare non è stata modificata.
- Per verificare questa condizione occorre aggiungere alla pagina un **bit di modifica (dirty bit)** che viene gestito via hardware:
  - Quando la pagina è caricata viene messo a 0
  - Quando la pagina è scritta viene messo a 1
  - Quando la pagina è scaricata viene fatta la scrittura su disco solo se il bit di modifica è a 1.

# Paginazione a richiesta

---

- Il meccanismo di sostituzione illustrato sino ad ora necessita della definizione di due componenti fondamentali:
  - L'algoritmo per l'**allocazione dei frame** che decide quanti frame devono essere allocati a ciascun processo.
  - L'algoritmo per la **sostituzione delle pagine** che decide quale pagina debba essere scaricata se non ci sono frame liberi.

# Sostituzione delle pagine

---

- Il numero di page fault è influenzato fortemente dai ricaricamenti ovvero dalla scelta di fare swap out di una pagina che servirà entro breve e dovrà pertanto rifare swap in.
- Confrontiamo i diversi algoritmi di sostituzione delle pagine per capire quanti fault generano.
- Consideriamo generalmente un algoritmo tanto più buono quanti meno page fault genera (perché miriamo a minimizzare il tempo effettivo di accesso).

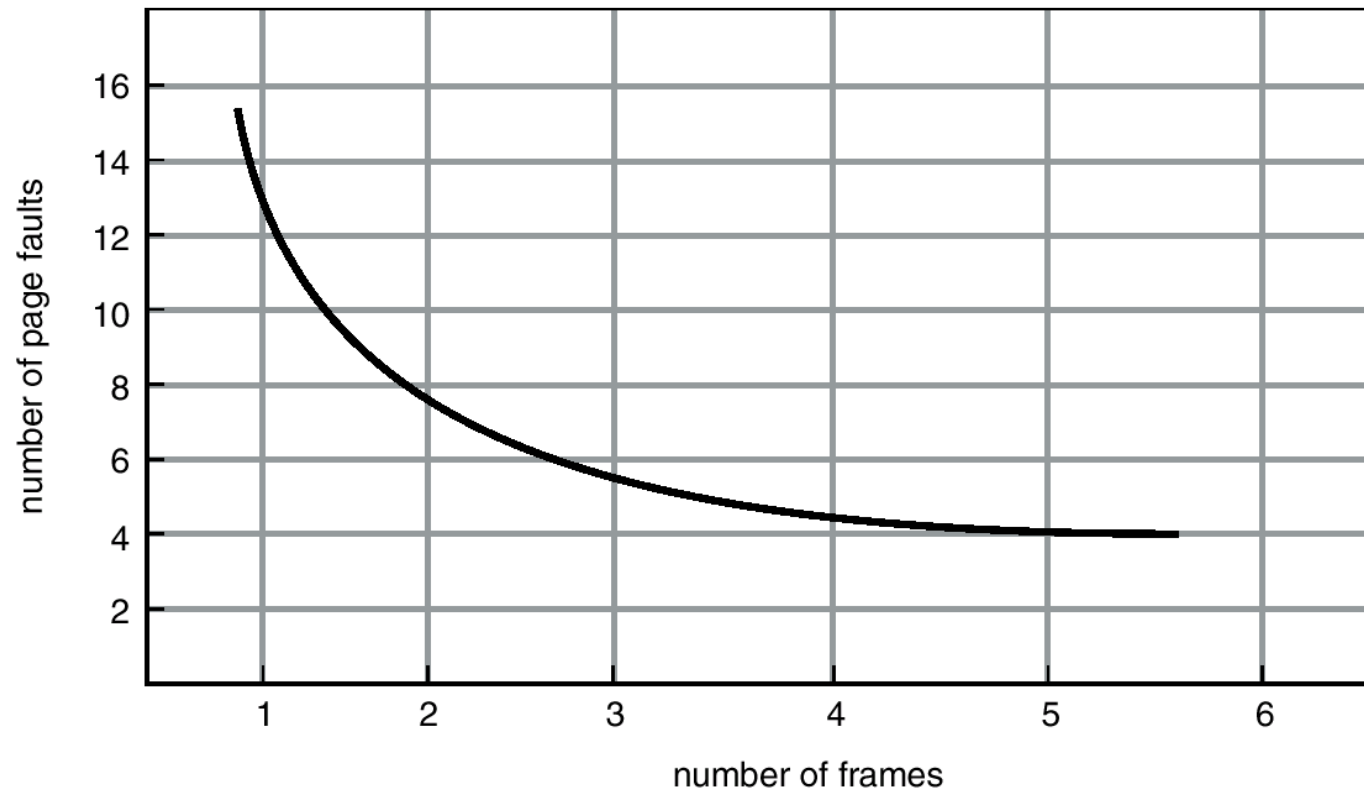
# Sostituzione delle pagine

- Utilizzeremo una stessa sequenza tipo di accessi alla memoria (stringa dei riferimenti) per comparare le prestazioni dei diversi algoritmi su una memoria con 3 frame:

**7,0,1,2,0,3,0,4,2,3,0,3,2,1,2,0,1,7,0,1**

- **Nota bene:** In generale il numero dei page fault tenderebbe a diminuire aumentando il numero dei frame.

# Page fault vs frame number



# FIFO

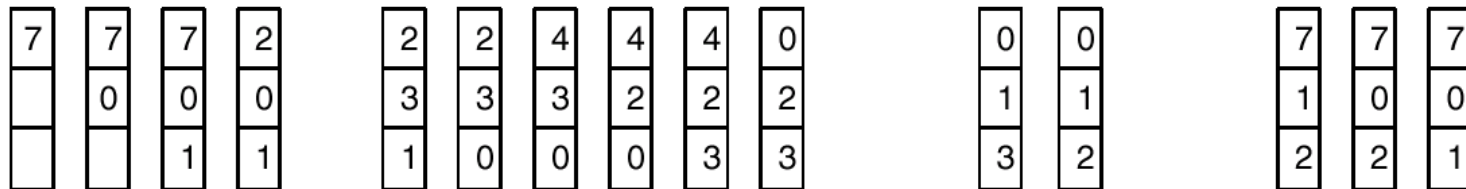
---

- Un primo approccio alla scelta della pagina da scaricare è quello di usare un algoritmo di tipo **First In First Out (FIFO)**.
- L'algoritmo associa ad ogni pagina l'istante in cui è stata caricata in memoria e, se c'è necessità di scaricare una pagina, sceglie la pagina più vecchia, ovvero quella che ha l'istante di caricamento più basso.

# FIFO

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

Page Fault

.....

NON Page Fault



# FIFO

---

- L'algoritmo di tipo **FIFO** è facile da implementare ma fornisce prestazioni piuttosto basse:
  - La pagina sostituita potrebbe essere un modulo di inizializzazione caricato dal processo all'inizio e da tempo inutilizzato, e allora ci va bene.
  - Potrebbe però anche essere una variabile globale allocata dal processo all'inizio e ancora molto in uso, e allora la scelta è sfortunata.

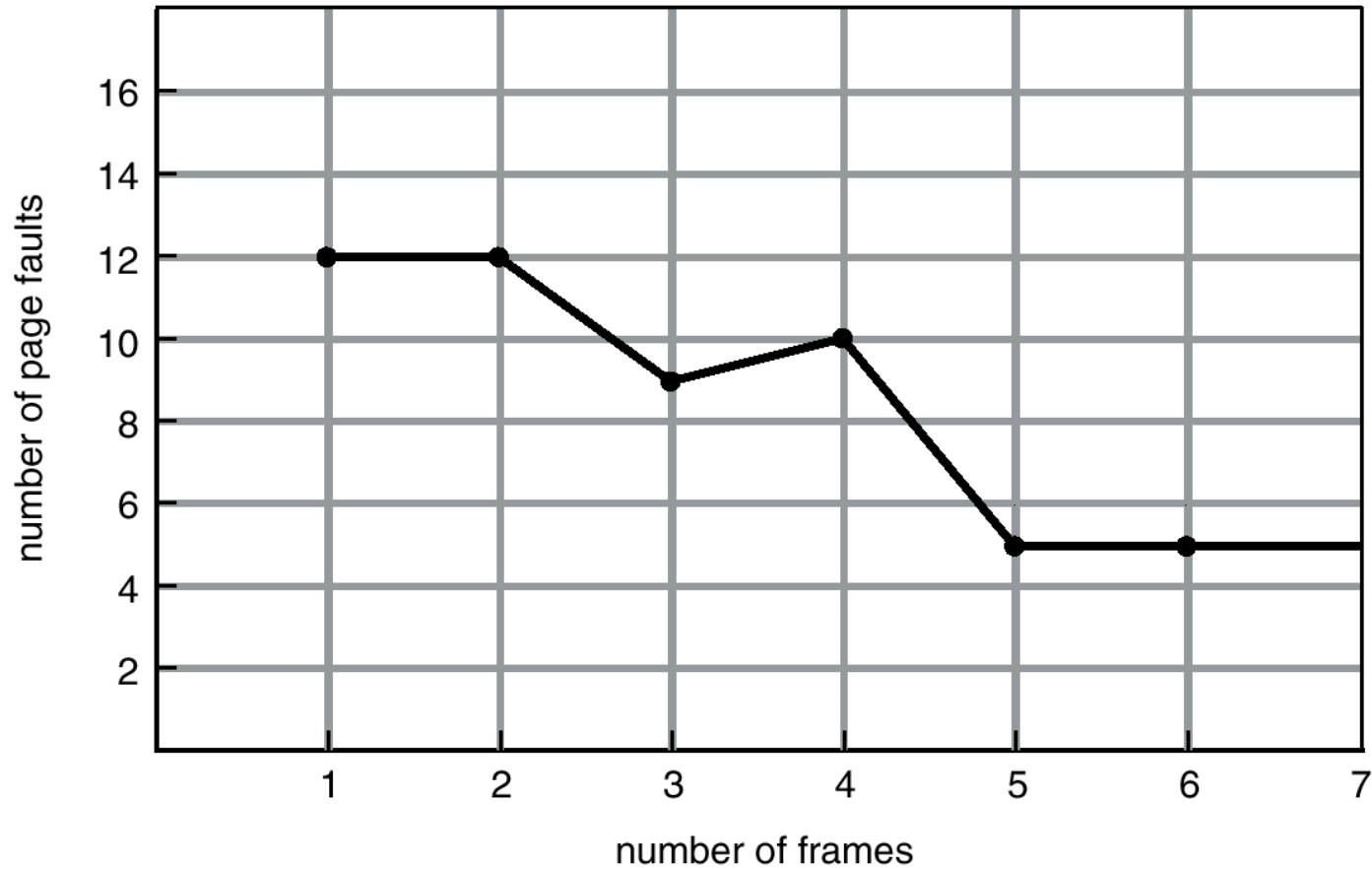
# FIFO

---

- Se la scelta della pagina da sostituire è errata aumenta la frequenza del page fault.
- Con la nostra sequenza abbiamo avuto 15 page fault.
- **Anomalia di Belady:** l'algoritmo FIFO può avere prestazioni peggiori aumentando il numero di frame.
- Provare a casa, ad esempio, la stringa di riferimenti seguente (su 3 e 4 frame):

**1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.**

# Anomalia di Belady



# Algoritmo ottimale (non implementabile)

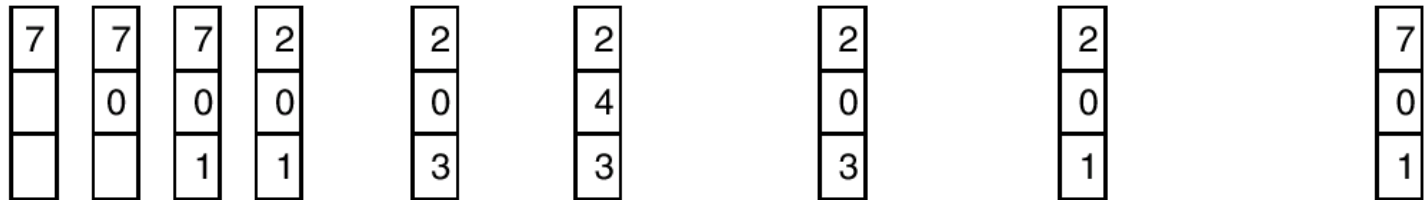
---

- Potendo conoscere la sequenza completa dei riferimenti richiesti, si può scrivere un **algoritmo ottimale** (o **algoritmo minimo**) che renda minimo il numero di page fault che avvengono complessivamente.
- L'algoritmo funziona scaricando la pagina che non verrà utilizzata per il periodo di tempo più lungo (ma si parla del futuro).

# Algoritmo ottimale

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1



page frames

# Algoritmo ottimale

---

- L'**algoritmo ottimale** non soffre dell'anomalia di Belady.
- Rispetto all'algoritmo FIFO abbiamo ottenuto (sulla stringa di riferimenti di prova, con 3 frame) un miglioramento notevole: 9 page fault al posto di 15.
- E' però **impossibile da implementare** perché richiede conoscenza sui riferimenti richiesti nel futuro (che sono imprevedibili).
- Viene utilizzato come **lower bound**: meglio di così non si può fare.

# Least Recently Used (LRU) (usata meno recentemente)

---

- Usiamo come approssimazione di un futuro vicino (quale pagina dilazionerà la generazione del page fault) un passato recente (quale pagina è inutilizzata da più tempo).
- L'algoritmo **Least Recently Used (LRU)** scarica la pagina che non viene acceduta da più tempo, supponendo che se non la si usa da un po', probabilmente non la si userà per un altro po'.

# LRU

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

7	7	7	2					2		4	4	4	0					1		1		1		
	0	0	0					0		0	0	3	3					3		0		0		
		1	1					3		3	2	2	2					2		2		7		

page frames



# LRU

---

- **LRU** rappresenta l'algoritmo ottimale tra quelli con analisi dei dati a ritroso.
- Rispetto all'algoritmo ottimale, ci sono casi in cui “sbaglia”, quando una pagina richiesta molto tempo fa viene scaricata ma sta in realtà per essere richiesta di nuovo.
- Non è soggetto all'anomalia di Belady.

# LRU

- LRU rappresenta una buona politica di sostituzione (molto utilizzata) ma è costoso da implementare.
- Il problema di determinare l'ordine in cui rimuovere le pagine può essere risolto sostanzialmente in due modi:
  - Con **contatori**: la tabella delle pagine non viene ordinata.
  - Con uno **stack**: la tabella delle pagine viene ordinata.
  - Entrambi i modi necessitano, ad ogni accesso ad una pagina, di modificare qualcosa nella tabella delle pagine.
  - Richiederebbero hardware apposito, troppo costoso.

# Contatori

---

- A ogni elemento della tabella delle pagine viene associato un campo **contatore** (o **clock**) che memorizza l'ora dell'ultimo utilizzo.
- Il contatore viene aggiornato ad ogni accesso alla pagina.
- Viene sostituita la pagina con il contatore più basso (ovvero quella non acceduta da più tempo).

# Contatori

---

- Questo meccanismo richiede:
  - Un accesso in modifica alla tabella delle pagine per ogni accesso alla pagina (per aggiornare il contatore)
  - Una ricerca (sequenziale) sulla tabella delle pagine per cercare la pagina con contatore minimo
- Si può pensare di migliorare le prestazioni mantenendo una struttura ordinata.

# Stack

---

- Si può pensare allora di utilizzare uno **stack** al posto della tabella:
  - Ogni volta che una pagina viene acceduta viene portata in cima allo stack.
  - Quando deve essere scaricata una pagina viene presa quella in fondo allo stack
- L'implementazione migliore è quella basata su una lista a doppio concatenamento.

# Stack

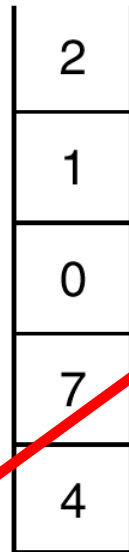
---

- In questo caso:
  - Ogni accesso alla memoria costa l'aggiornamento di  $n$  puntatori (l'elemento da portare al top potrebbe essere in mezzo o in fondo allo stack)
  - Abbiamo evitato il costo lineare dello scaricamento poiché sappiamo quale elemento scaricare

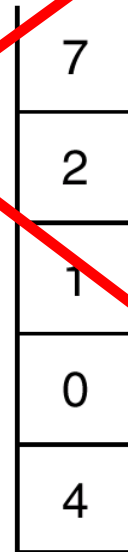
# Stack

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2



stack before a



stack after b

↑  
a

↑  
b

# LRU

---

- Il fatto che l'anomalia di **Belady** non si verifichi con LRU è facilmente dimostrabile usando l'implementazione a stack:
  - Se il numero di frame aumenta di 1 lo stack aumenta di 1 e quindi conterrà tutte le pagine dello stack di lunghezza  $n$  più una.
  - Le prestazioni non possono peggiorare aggiungendo un frame.



# LRU

---

- In entrambi ogni accesso alla memoria provoca una **modifica alla tabella delle pagine**.
- Supportare questa operazione via software (generando un interrupt per ogni accesso alla memoria gestito dal SO) è particolarmente oneroso e può aumentare il tempo di accesso alle locazioni di un fattore pari a 10 volte.

# LRU

---

- Sono poche le architetture che possono permettersi una implementazione hardware di LRU.
- Più tipicamente viene messo a disposizione uno dei seguenti supporti:
  - **Bit di riferimento:** 0 se la pagina non è più stata acceduta, 1 altrimenti.
  - **Conteggio degli accessi:** contatore del numero degli accessi effettuati dal caricamento della pagina.

# Bit di riferimento

---

- Il **bit di riferimento** di per se non è sufficiente a scegliere una pagina da scaricare.
- Tipicamente viene azzerato al momento della generazione della tabella delle pagine.
- Per utilizzarlo è necessario ri-azzerarlo periodicamente in modo che sia rappresentativo di uno stato recente della macchina.
- Esistono diversi algoritmi che operano in questo modo mirando ad approssimarsi ad LRU.

# Bit Supplementari

- Algoritmo con **bit supplementari di riferimento**:
  - Ogni N millisecondi (supponiamo 100) viene salvato il bit di riferimento di tutte le pagine.
  - Per il salvataggio si usa un byte per ogni pagina e si scrive il bit più significativo shiftando gli altri verso le posizioni meno significative.
  - Dopo il salvataggio, il bit di riferimento viene azzerato. Se entro i successivi N millisecondi la pagina viene acceduta, il bit tornerà ad 1 e nel prossimo salvataggio verrà salvato 1, altrimenti verrà salvato 0.
  - La pagina usata meno di recente è quella che ha il byte con valore più basso

# Seconda Chance

---

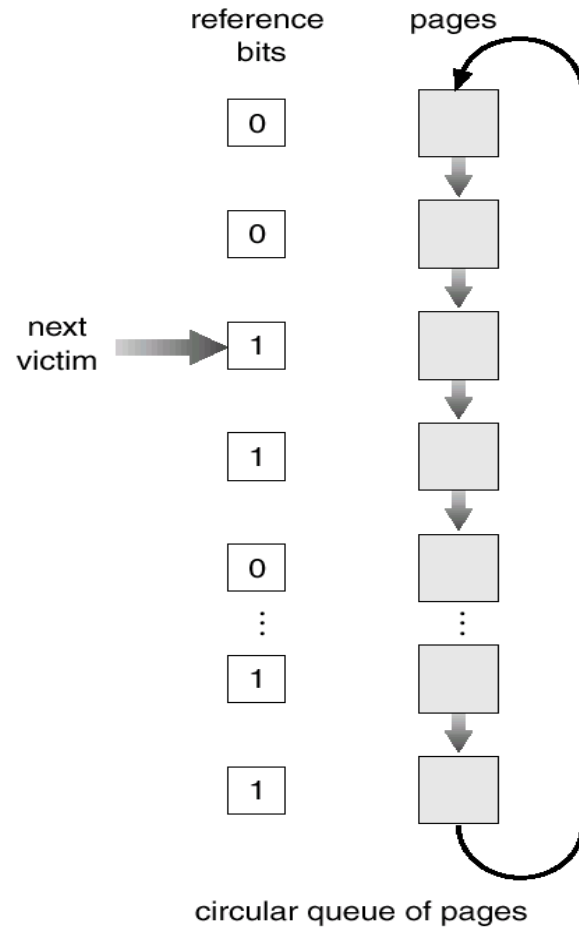
- Algoritmo **Seconda Chance**: si comporta in modo simile all'algoritmo visto in precedenza ma utilizza un solo bit per ciascuna pagina. Il bit viene messo ad 1 quando si accede alla pagina
- Le pagine sono tenute in una coda circolare e viene gestito un puntatore al prossimo elemento da sostituire che si comporta così:
  - Se trova un bit 0 sostituisce.
  - Se trova un bit 1 lo azzera.

# Seconda Chance

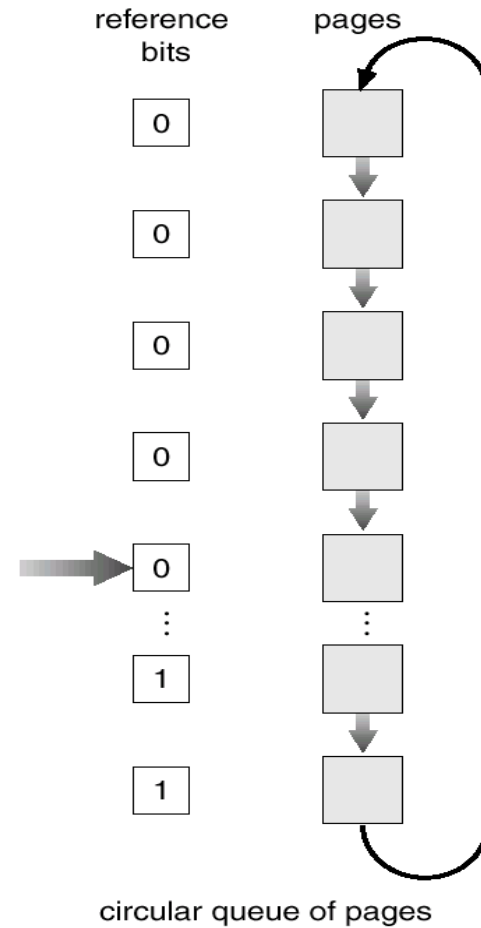
---

- In questo modo:
  - La pagina con bit 1 ha una **seconda chance** perché può rimettere il bit a 1 (con un accesso) prima che il puntatore torni in quel punto
  - Se la pagina con bit 1 non viene riacceduta prima che il puntatore abbia fatto il giro della coda, allora verrà scaricata.
- Se tutti i bit sono a 1 la sostituzione degenera in una sostituzione FIFO

# Seconda Chance



(a)



(b)

# Seconda Chance migliorato

---

- L'algoritmo **Seconda Chance** può essere **migliorato** usando come base la coppia:
  - **Bit di riferimento**: la pagina è stata/ non è stata acceduta di recente.
  - **Bit di modifica (dirty bit)**: la pagina è stata/non è stata modificata dal suo caricamento (e di conseguenza deve/non deve essere salvata).
- Obiettivo: privilegiare lo scaricamento di pagine non modificate che si fa senza salvataggio dei valori su disco.



# Seconda Chance migliorato

---

- Il sistema può quindi trovarsi nelle seguenti situazioni:
  - **(0,0)** né utilizzato, né modificato, migliore pagina da sostituire.
  - **(0,1)** non utilizzato di recente ma modificato. Potrebbe non servire più ma deve comunque essere riscritto.
  - **(1,0)** non modificato ma usato di recente. Potrebbe essere riusato ma non va riscritto.
  - **(1,1)** usato di recente e modificato, peggiore pagina da sostituire.

# Buffering delle pagine

---

- Un significativo aumento delle prestazioni si può avere **liberando in modo preventivo** le pagine.
- Si mantiene un insieme di frame liberi che possono essere ceduti al processo senza operazioni di swap out.
- Lo swap out comincia dopo lo swap in per mantenere inalterato il numero dei frame liberi.

# Buffering delle pagine

---

- Un'estensione di questa idea è quella di **sottoporre le pagine a salvataggi preventivi** quando il dispositivo di paginazione è inattivo.
- Questo significa azzerare il bit di modifica e operare più velocemente lo swap out.

# Allocazione dei frame

---

- Tipicamente saranno presenti **più processi** e a ciascuno dovrà essere assegnata una certa quantità di frame in memoria.
- Il sistema ha alcuni vincoli dovuti al fatto che esiste un **numero minimo di frame** per processo:
  - Deve essere in memoria l'istruzione e i dati per la sua esecuzione.
  - Meno frame ha a disposizione il processo e più page fault genererà.

# Allocazione dei frame

---

- Il modo più semplice per allocare gli **m** frame del sistema tra gli **n** processi è quello di darne una parte uguale ( **$m/n$** ) a ciascuno.
- I processi possono avere dimensioni diverse per cui questa scelta è poco flessibile.
- Si può allora ricorrere alla **allocazione proporzionale** in cui la memoria viene allocata al processo in base alla sua dimensione

# Allocazione dei frame

---

- L'allocazione proporzionale deve comunque tenere conto del numero minimo di frame per processo.
- Il **numero dei processi potrebbe aumentare** e in questo caso andrebbe sottratta una parte dei frame a ciascun processo in esubero (quelli che hanno il numero minimo di frame non possono essere coinvolti)

# Sostituzione

---

- In presenza di più processi la sostituzione può essere:
  - Sostituzione **globale**: la richiesta è soddisfatta prelevando un frame da un processo qualunque.
  - Sostituzione **locale**: la richiesta è soddisfatta prelevando un frame dal processo che ha generato il trap.
- Ognuno dei due approcci presenta vantaggi e svantaggi.

# Sostituzione

---

- Sostituzione **globale**:
  - Consente di gestire la priorità.
  - Non consente al processo di controllare la propria frequenza di page fault.
- Sostituzione **locale**:
  - Non si adatta alle richieste in modo flessibile.
  - Non consente la gestione della priorità.
  - Consente al processo di controllare la propria frequenza di page fault.



# Thrashing

---

- Se si gestisce con sostituzione globale la priorità può accadere che un processo con priorità alta porti un processo con priorità bassa a un **numero di frame inferiore al minimo**.
- Occorre quindi richiamare lo scheduling a medio termine e:
  - Rimuovere il processo dalla ready queue.
  - Liberare le pagine restanti

# Thrashing

---

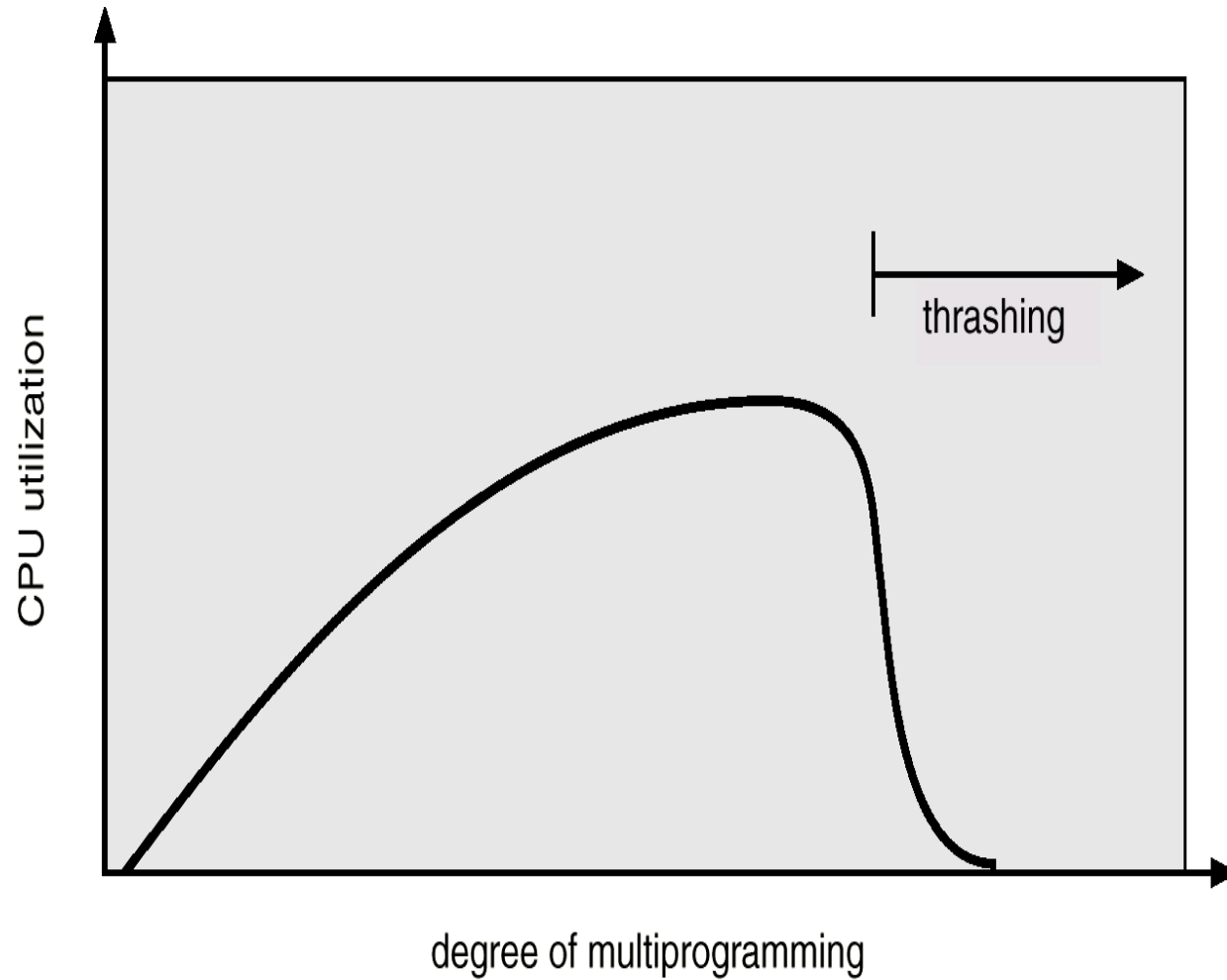
- Se non si rimuove il processo, il rischio è che generi continuamente page fault:
  - Ogni fault soddisfatto può in realtà rimuovere un'altra pagina indispensabile generando altri page fault.
  - Il processo è in **thrashing** cioè sta perdendo più tempo in attività correlate alla paginazione che in attività di calcolo.

# Thrashing

---

- Un **circolo vizioso** può essere innescato nel modo seguente:
  - Il SO rileva un basso uso della CPU e aumenta il grado di multiprogrammazione
  - Carica nuovi processi che allocano nuovi frame a discapito dei processi già in memoria che riducono il numero di frame sotto la soglia minima
  - I processi fanno continui swap e cala l'uso della CPU

# Thrashing



# Prevenzione del Thrashing

---

- Una soluzione è quella di **evitare** che un primo processo in thrash possa mandarne in thrash un altro, sospendendo preventivamente il primo processo, ma non è una soluzione efficace perché il primo processo resta nella coda di scheduling.
- Occorre evitare di ridurre il numero delle pagine del processo sotto una certa **soglia**.
- Se il processo ha bisogno di più frames rispetto alla propria soglia, ma non ci sono frames liberi, allora può essere sospeso.

# Località

---

- Per sapere quale è la soglia giusta (non sempre basta il numero minimo di frame!) ci si può basare sul **modello di località** che stabilisce quali porzioni del processo debbano essere caricate contemporaneamente in ciascuna fase di vita del processo.
- ES: una subroutine che quando entra in funzione opera su un insieme di variabili globali.

# Working set

---

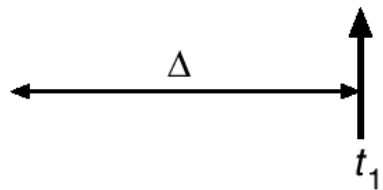
- Per studiare la soglia di frame necessari, con il modello di località, solitamente si usano i **working set** ovvero l'insieme delle pagine che contengono gli ultimi  $\Delta$  (numero fissato) riferimenti.
- $\Delta$  è detto **finestra del working set**.
  - Se una pagina è in uso attivo si trova nel working set.
  - Se non è più utilizzata da  $\Delta$  unità di tempo, esce dal working set.

# Working set

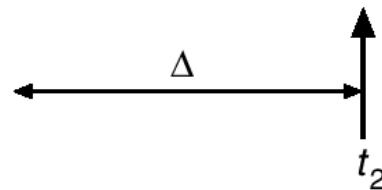
$$\Delta = 10$$

page reference table

... 2 6 1 5 7 7 7 5 1 6 2 3 4 1 2 3 4 4 4 3 4 3 4 4 4 1 3 2 3 4 4 4 3 4 4 4 ...



$$WS(t_1) = \{1, 2, 5, 6, 7\}$$



$$WS(t_2) = \{3, 4\}$$



# Working set

---

- L'approssimazione del working set al modello di località è la seguente:
  - Se  $\Delta$  è troppo piccolo non include l'intera località
  - Se  $\Delta$  è troppo grande può sovrapporre più località
  - Se  $\Delta$  è infinito include tutto il processo.

# Dimensione

- Calcolando la **dimensione del working set** di un processo  $i$  (**WSS<sub>i</sub>**) si può determinare la richiesta totale di frame come:

$$D = \sum WSS_i$$

- Se la richiesta totale è maggiore del numero totale dei frame ( **$D > m$** ) si verifica thrashing poiché alcuni processi non dispongono di frame sufficienti a contenere il working set.

# Frequenza dei page fault

---

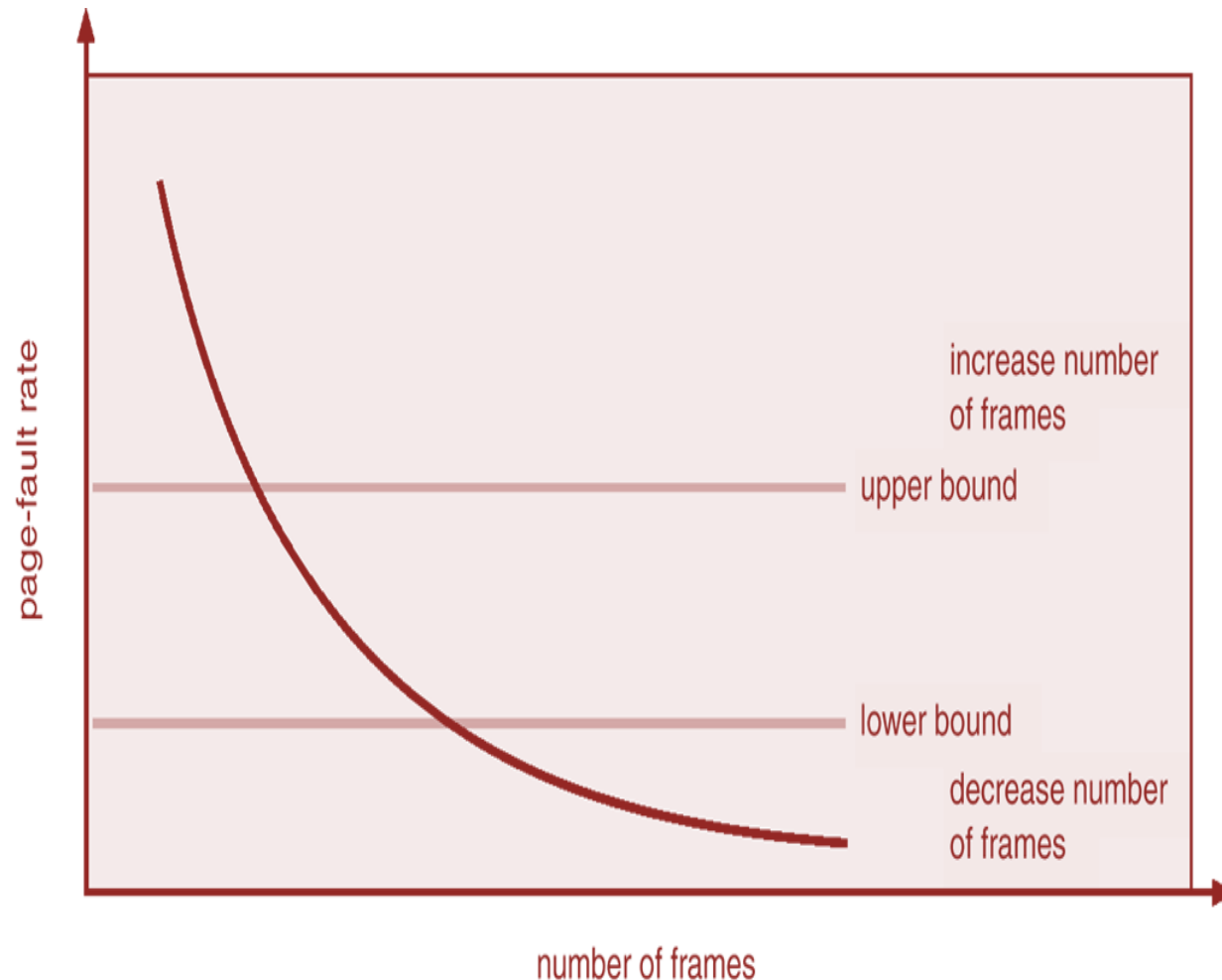
- Il sistema dei working set non è completamente efficace nei confronti della prevenzione dei page fault.
- Nella **prevenzione del thrashing** si utilizza solitamente una strategia più diretta basata sulla **frequenza dei page fault (Page Fault Frequency, PFF)**.

# Frequenza dei page fault

---

- Viene tenuto sotto controllo ogni processo e:
  - quando la frequenza di page fault è eccessiva significa che il processo ha bisogno di più frame. Se non ci sono frame disponibili il processo può essere sospeso.
  - quando la frequenza dei page fault è bassa significa che il processo ha frame in eccesso.

# Frequenza dei page fault



.....

---

.... anche basta così,  
grazie ...

# File mappati in memoria

---

- L'accesso sequenziale a un file è costituito da una sequenza di system call (read e write) che a loro volta generano una sequenza di richieste all'I/O.
- Per rendere più efficiente questa attività si utilizzano tecniche di mappatura dei file in memoria, ottenuta associando un blocco del disco a uno a più pagine residenti in memoria virtuale.
- L'accesso iniziale al file avviene attraverso un page fault ma poi parte del file viene mantenuto in memoria centrale, velocizzando i tempi per l'I/O.
- I blocchi che verranno letti sono prevedibili, dunque il caricamento può essere anticipato

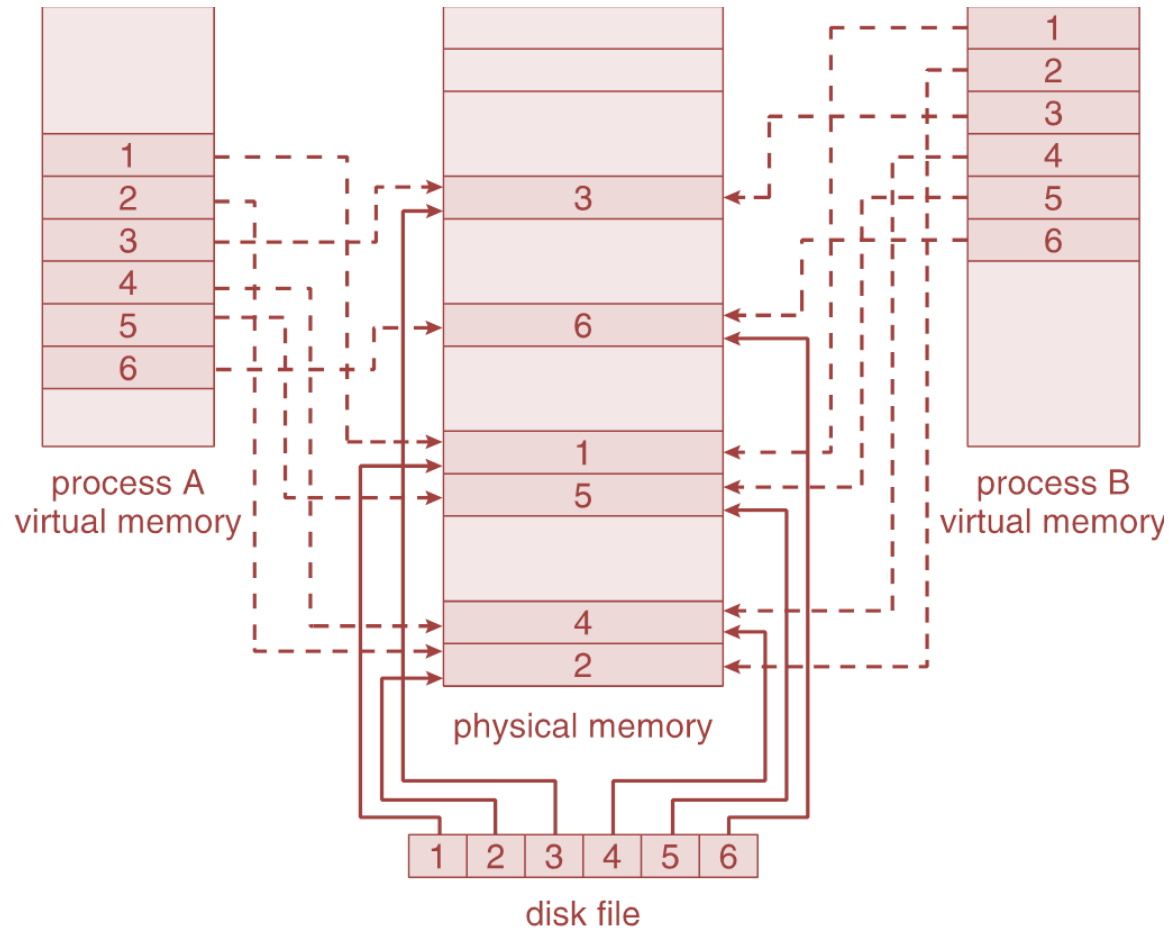
# File mappati in memoria

---

- Le scritture sul file in memoria non corrispondono a immediate scritture sul disco ma dipendono da politiche del SO.
- I file mappati sono condivisibili dai processi, sfruttando la semplicità di condivisione propria della paginazione e della memoria virtuale (e opportuni sistemi di sincronizzazione e mutua esclusione)
- Quando il file viene chiuso, le eventuali modifiche sono tutte trascritte in memoria di massa.
- Alcuni sistemi prevedono una apposita syscall per la mappatura dei file, per esempio Solaris usa la chiamata a sistema **mmap ( )**

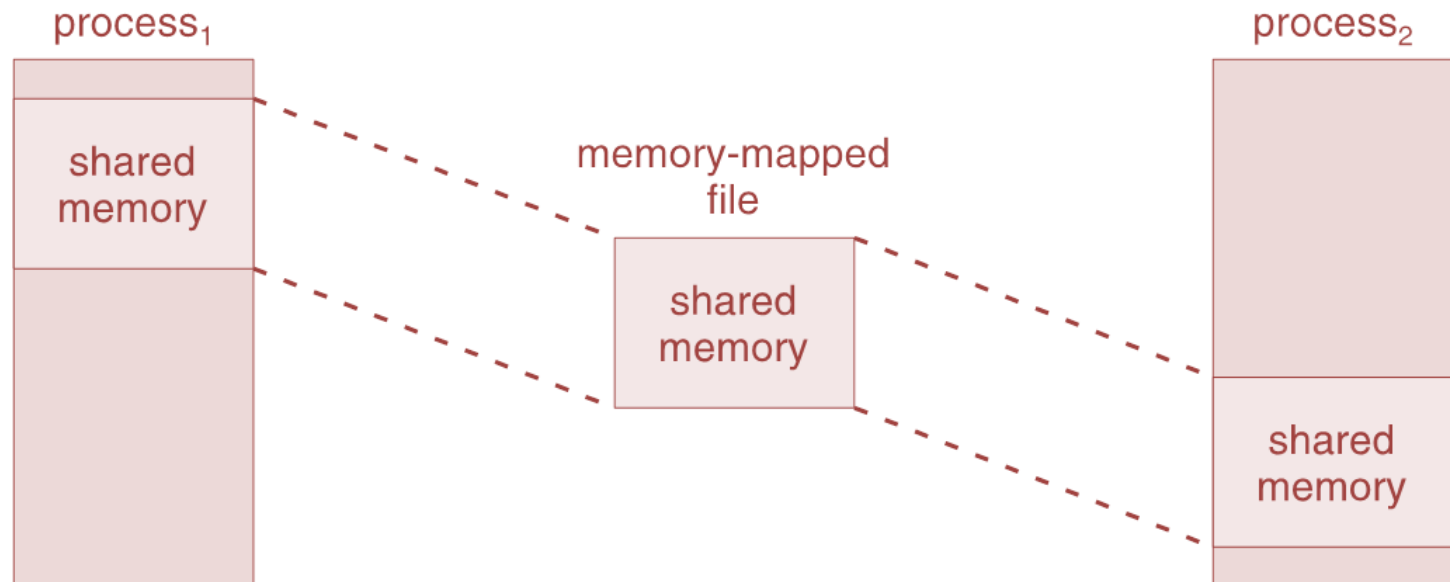


# Condivisione del file mappato



# File mappati in memoria

- In alcuni sistemi della famiglia Windows (ES: Windows XP) la memoria condivisa è realizzata come una forma di mappatura condivisa dei file



# Mappatura dell'I/O

---

- In molte architetture, per rendere più agevole l'accesso all'I/O si utilizza una mappatura dell'I/O in memoria.
- Vengono cioè **mappati in memoria i registri dei controller di I/O** in modo da rendere più efficiente la comunicazione
- Le operazioni di lettura e scrittura su queste aree di memoria corrispondono in realtà a operazioni di trasferimento tra registri dei dispositivi e memoria o viceversa.

# Bibliografia

---

- Silberschatz:
  - capitolo 9

