Memoria virtuale

Ivan Scagnetto

Università di Udine — Facoltà di Scienze MM.FF.NN.

A.A. 2010-2011

Memoria Virtuale

- Memoria virtuale: separazione della memoria logica vista dall'utente/programmatore dalla memoria fisica.
- Solo parte del programma e dei dati devono stare in memoria affinché il processo possa essere eseguito (resident set).

Memoria Virtuale: motivazioni

Molti vantaggi sia per gli utenti che per il sistema:

- Lo spazio logico può essere molto più grande di quello fisico.
- Meno consumo di memoria $\Rightarrow$ più processi in esecuzione $\Rightarrow$ maggiore multiprogrammazione.
- Meno I/O per caricare in memoria i programmi.

Porta alla necessità di caricare e salvare parti di memoria dei processi da/per il disco al momento dell'esecuzione (runtime).
La memoria virtuale può essere implementata come paginazione su richiesta oppure segmentazione su richiesta

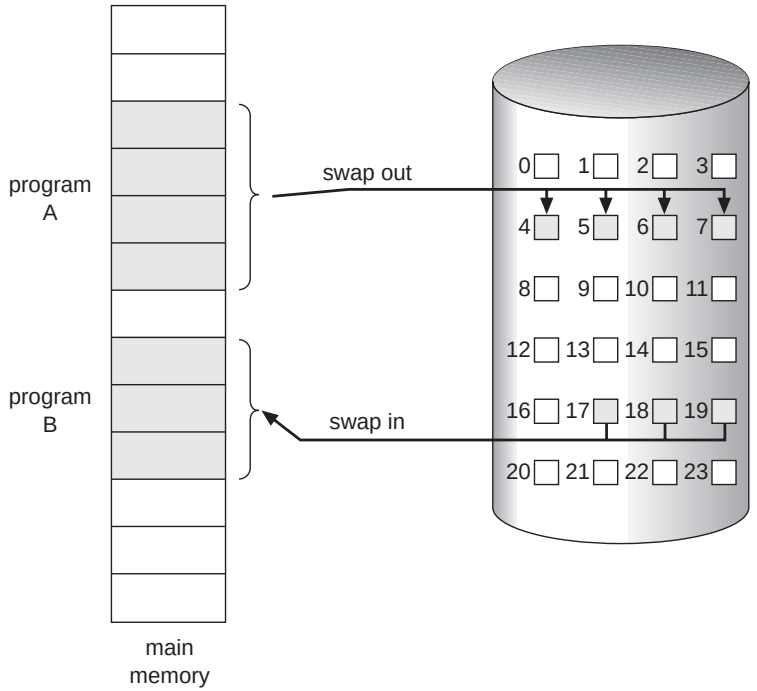
Paginazione su richiesta

Schema a paginazione, ma in cui si carica una pagina in memoria solo quando è necessario:

- meno I/O,
- meno memoria occupata,
- maggiore velocità,
- più utenti/processi.

Una pagina è richiesta quando vi si fa riferimento:

- viene segnalato dalla MMU,
- se l'accesso non è valido $\Rightarrow$ abortisci il processo,
- se la pagina non è in memoria $\Rightarrow$ caricala dal disco.



Swapping vs. Paging

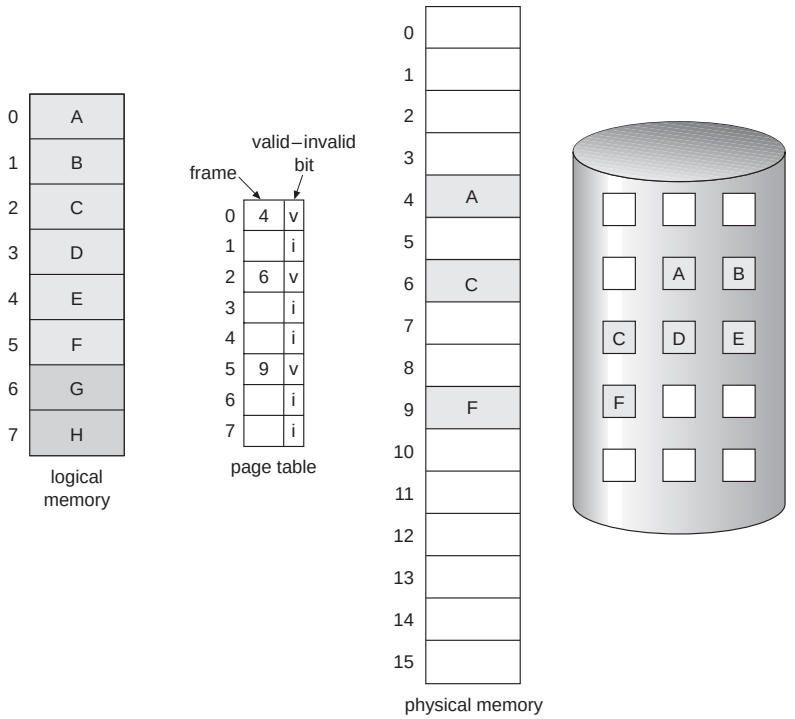
Spesso si confonde swapping con paging:

- Swapping: scambio di interi processi da/per il backing store. Swapper: processo che implementa una politica di swapping (scheduling di medio termine).
- Paging: scambio di gruppi di pagine (sottoinsiemi di processi) da/per il backing store.
- Pager: processo che implementa una politica di gestione delle pagine dei processi (caricamento/scaricamento).

Sono concetti molto diversi, e non esclusivi!
Purtroppo, in alcuni S.O. il pager viene chiamato “swapper” (es.: Linux: `kswapd`).

Valid–Invalid Bit

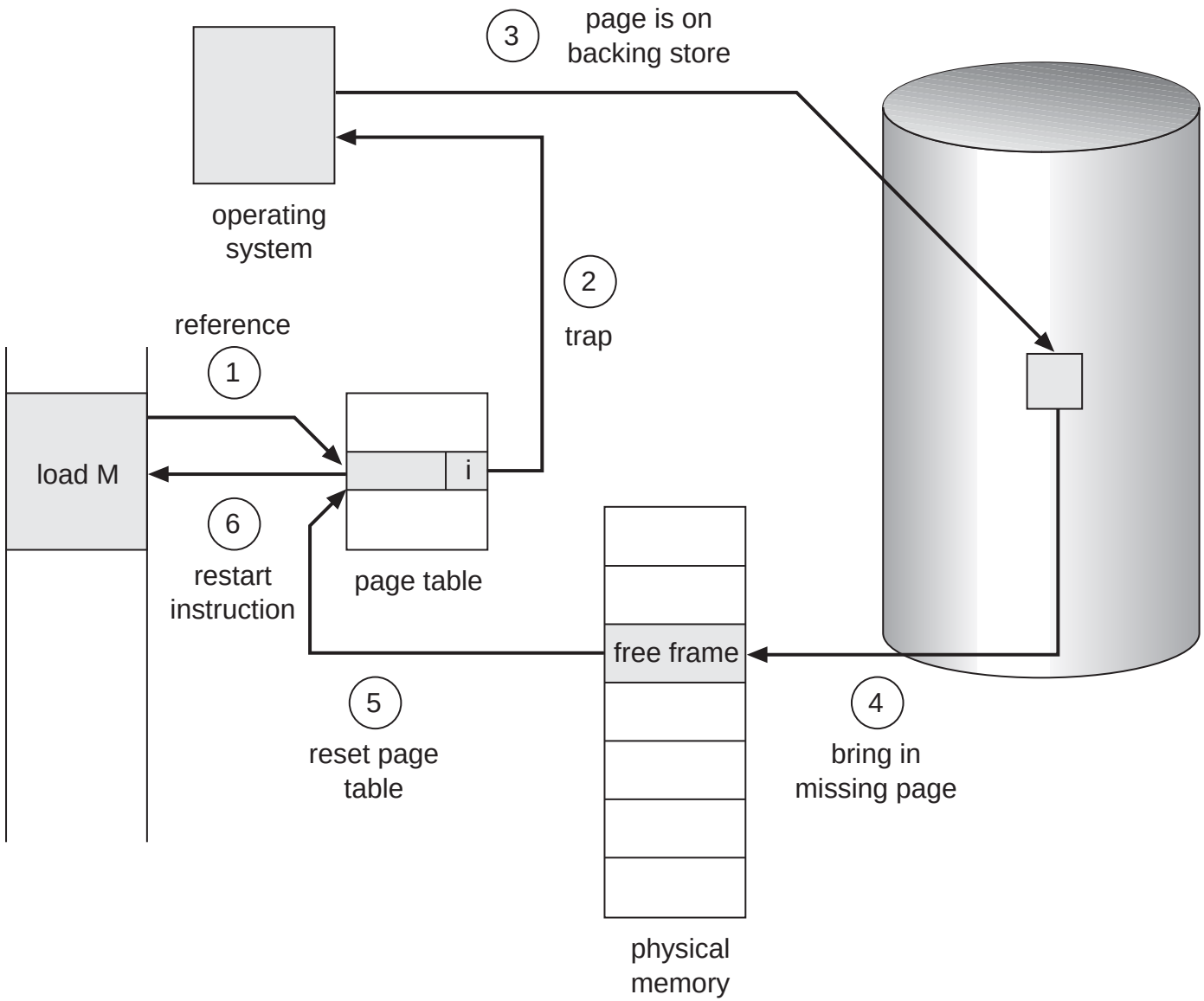
- Ad ogni entry nella page table, si associa un bit di validità. (1 $\Rightarrow$ in-memory, 0 $\Rightarrow$ not-in-memory).
- Inizialmente, il bit di validità è settato a 0 per tutte le pagine.
- La prima volta che si fa riferimento ad una pagina (non presente in memoria), la MMU invia un interrupt alla CPU: page fault.



Routine di gestione del Page Fault

- Il S.O. controlla, guardando in un'altra tabella, se è stato un accesso non valido (fuori dallo spazio indirizzi virtuali assegnati al processo) $\Rightarrow$ abort del processo (segmentation fault).
- Se l'accesso è valido, ma la pagina non è in memoria:
  - trovare qualche pagina in memoria, ma in realtà non usata, e scaricarla su disco (swap out),
  - caricare la pagina richiesta nel frame così liberato (swap in),
  - aggiornare le tabelle delle pagine.
- L'istruzione che ha causato il page fault deve essere rieseguita in modo consistente.
  $\Rightarrow$ vincoli sull'architettura della macchina. Es: la MVC dell'IBM 360.

Page Fault: gestione



Performance del paging on-demand

- p = Page fault rate; $0 \leq p \leq 1$
  - $p = 0 \Rightarrow$ nessun page fault
  - $p = 1 \Rightarrow$ ogni riferimento in memoria porta ad un page fault
- Tempo effettivo di accesso (EAT)

$$
\begin{aligned}
EAT \quad = \quad & (1 - p) \times \text{accesso alla memoria} \\
& + p(\text{overhead di page fault} \\
& \quad [+\text{swap page out}] \\
& \quad + \text{swap page in} \\
& \quad + \text{overhead di restart})
\end{aligned}
$$

Esempio di Demand Paging

- Tempo di accesso alla memoria (comprensivo del tempo di traduzione): 60 nsec.
- Assumiamo che il 50% delle volte che una pagina deve essere rimpiazzata, essa sia stata modificata e quindi debba essere scaricata su disco.
- Swap Page Time = 5 msec = $5e6$ nsec (disco molto veloce!).
- $EAT = 60(1 - p) + 5e6 * 1.5 * p = 60 + (7.5e6 - 60)p$ in nsec.
- Si ha un degrado del 10% quando $p = 6/(7.5e6 - 60) = 1/1250000$.

Considerazioni sul Demand Paging

- Problema di performance: si vuole un algoritmo di rimpiazzamento che porti al minor numero di page fault possibile.
- L'area di swap deve essere il più veloce possibile $\Rightarrow$ meglio tenerla separata dal file system (possibilmente anche su un device dedicato) ed accedervi direttamente (senza passare per il file system). Blocchi fisici = frame in memoria.
- La memoria virtuale con demand paging porta benefici anche al momento della creazione dei processi.

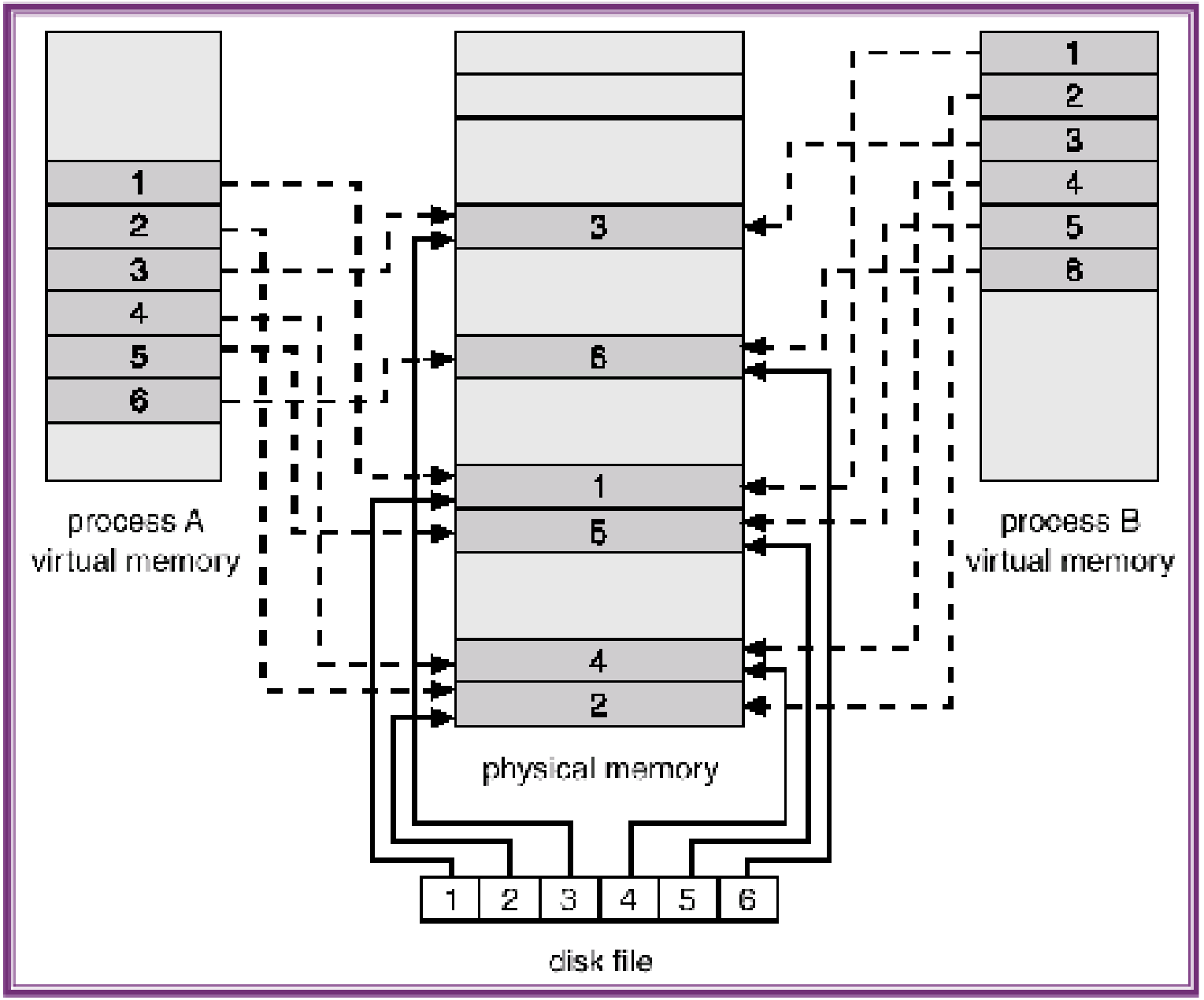
Creazione dei processi: Copy on Write (COW)

- Il Copy-on-Write permette al padre e al figlio di condividere inizialmente le stesse pagine in memoria.
- Una pagina viene copiata se e quando viene acceduta in scrittura.
- COW permette una creazione più veloce dei processi.
- Le pagine libere devono essere allocate da un set di pagine azzerate (per evitare che un processo possa accedere a informazioni “sensibili” memorizzate in precedenza nelle pagine in questione da un altro processo):
  - i sistemi operativi solitamente mantengono un pool di pagine libere da cui sceglierne una in caso di necessità;
  - es.: Windows esegue periodicamente un processo azzeratore che ripulisce le pagine disponibili presenti nel pool di pagine libere per l’allocazione (in modo da cancellare ogni traccia di eventuali contenuti preesistenti).

Creazione dei processi: Memory-Mapped I/O

- Memory-mapped file I/O permette di gestire l'I/O di file come accessi in memoria: ogni blocco di un file viene mappato su una pagina di memoria virtuale.
- Un file (es. DLL, .so) può essere così letto come se fosse in memoria, con demand paging. Dopo che un blocco è stato letto una volta, rimane caricato in memoria senza doverlo rileggere.
- La gestione dell'I/O è molto semplificata.
- Più processi possono condividere lo stesso file, condividendo gli stessi frame in cui viene caricato.

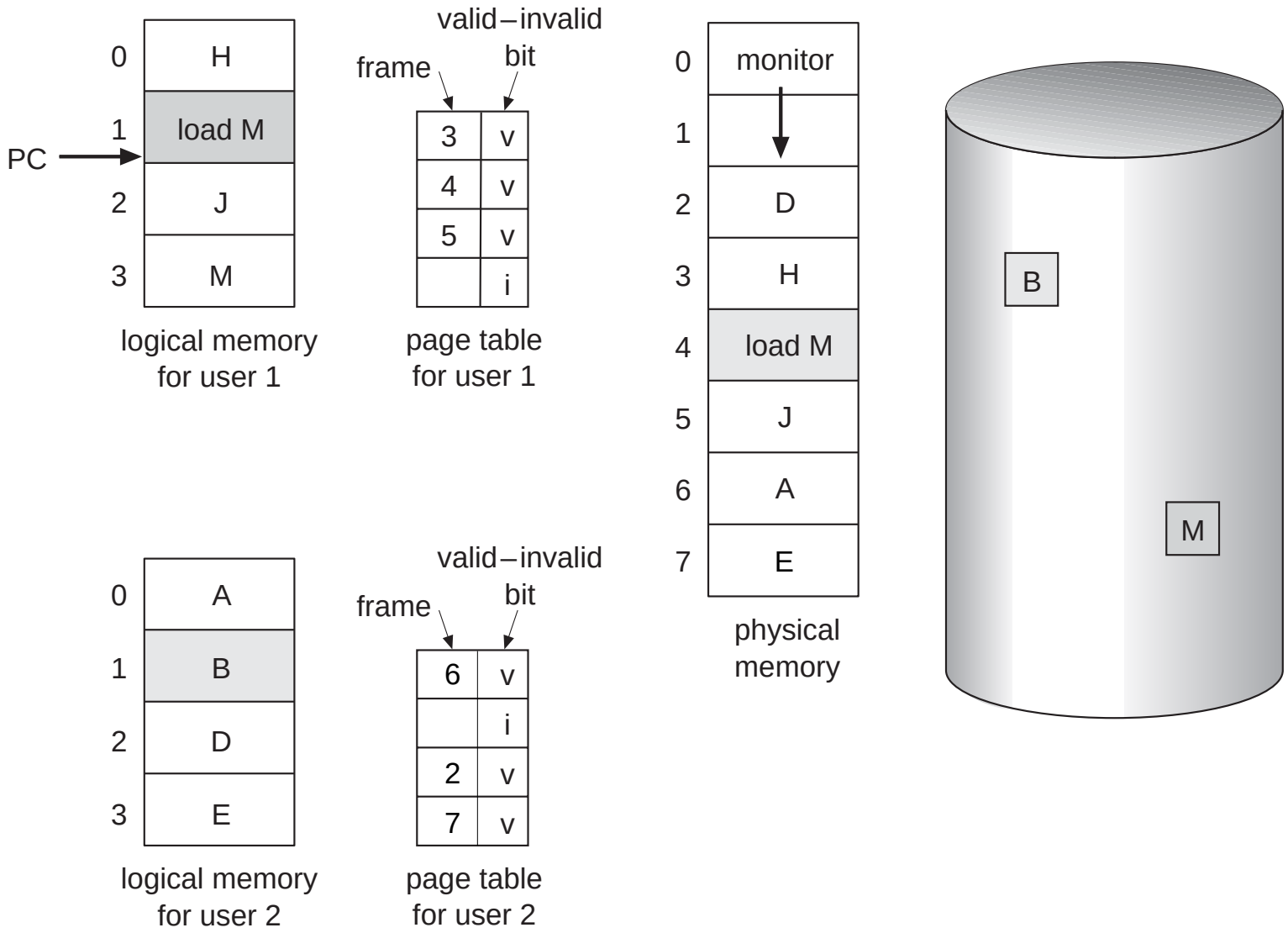
Creazione dei processi: Memory-Mapped I/O



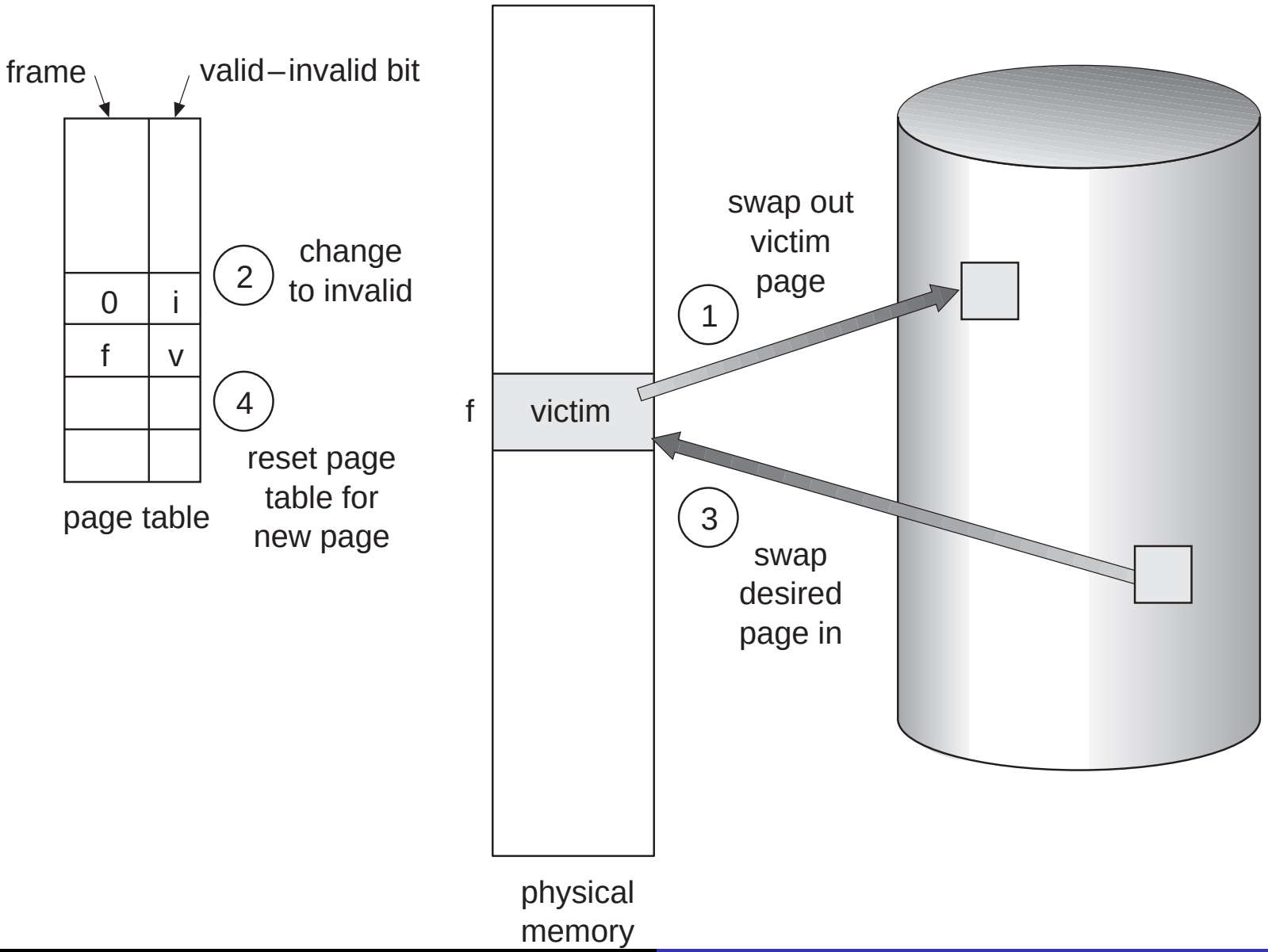
Sostituzione delle pagine

- Aumentando il grado di multiprogrammazione, la memoria viene sovrallocata: la somma degli spazi logici dei processi in esecuzione è superiore alla dimensione della memoria fisica.
- Ad un page fault, può succedere che non esistano frame liberi.
- Si modifica la routine di gestione del page fault, aggiungendo la sostituzione delle pagine che libera un frame occupato (vittima).
- Bit di modifica (dirty bit): segnala quali pagine sono state modificate, e quindi devono essere salvate su disco. Riduce l'overhead.
- Il rimpiazzamento di pagina completa la separazione tra memoria logica e memoria fisica: una memoria logica di grandi dimensioni può essere implementata con una piccola memoria fisica.

Sostituzione delle pagine



Sostituzione delle pagine

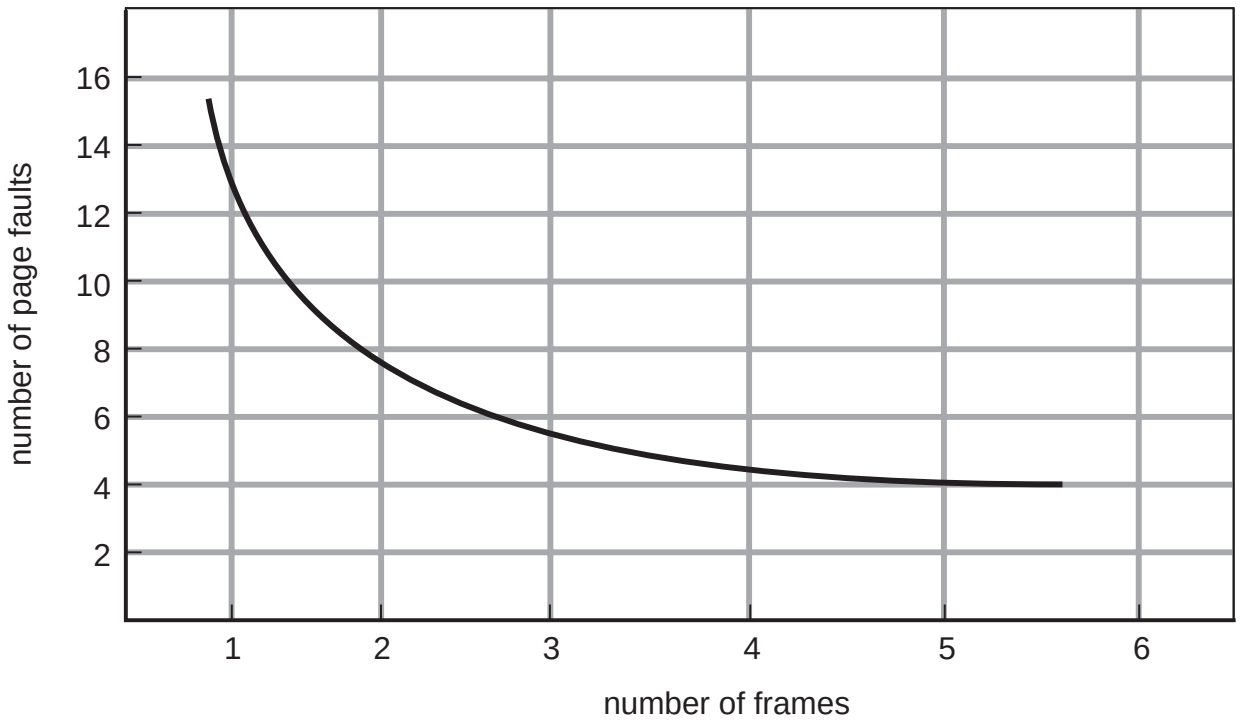


Algoritmi di rimpiazzamento delle pagine

- È un problema molto comune, non solo nella gestione della memoria (es: cache di CPU, di disco, di web server. . .).
- Si mira a minimizzare il page-fault rate.
- Un modo per valutare questi algoritmi: provarli su una sequenza prefissata di accessi alla memoria, e contare il numero di page fault.
- In tutti i nostri esempi, la sequenza sarà relativa a 5 pagine accedute in questo ordine `1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5`.

Page Fault vs. Numero di Frame

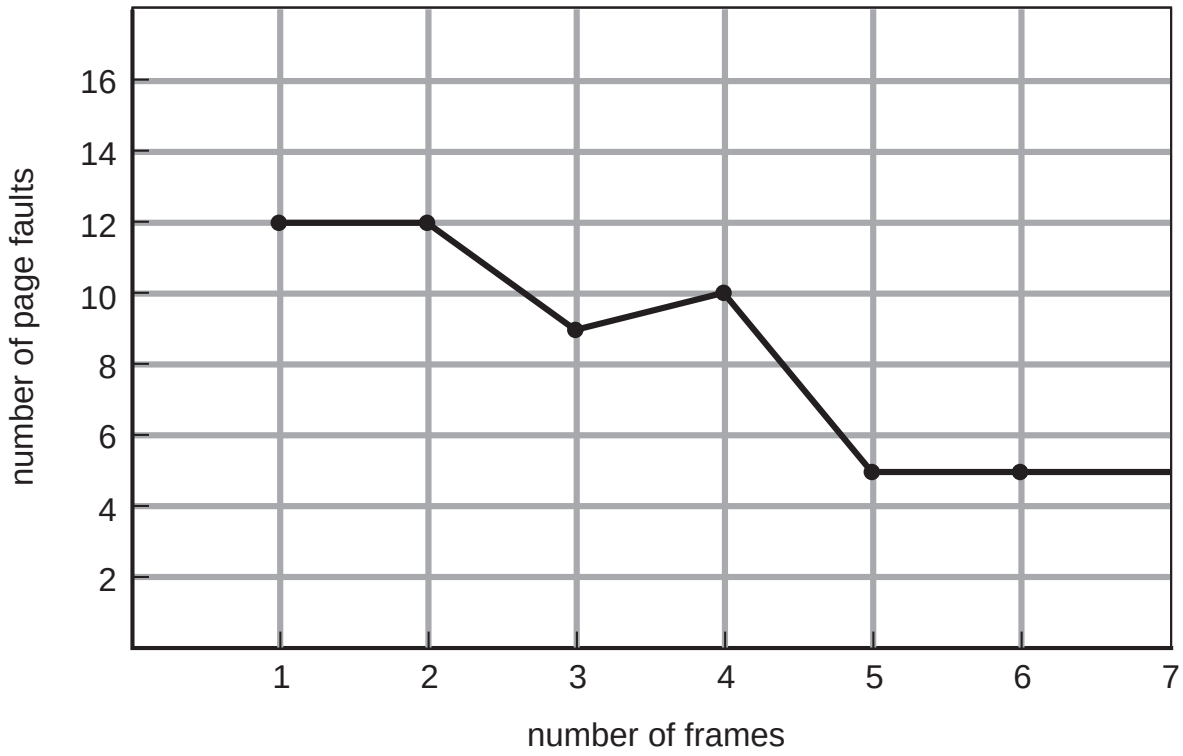
- In generale ci si aspetta che il numero di page fault cali all'aumentare del numero di frame disponibili sul sistema:



- Aggiungere nuovi moduli di memoria al sistema aumenta il numero di frame disponibili.

Algoritmo First-In-First-Out (FIFO)

- Si rimpiazza la pagina che da più tempo è in memoria.



- Con 3 frame (3 pagine per volta possono essere in memoria): 9 page fault.
- Con 4 frame: 10 page fault.
- Il rimpiazzamento FIFO soffre dell'anomalia di Belady: + memoria fisica $\not\Rightarrow$ – page fault!

Algoritmo ottimale (OPT o MIN)

- Si rimpiazza la pagina che non verrà riusata per il periodo più lungo.
- Con 4 frame: 6 page fault.
- Tra tutti gli algoritmi, è quello che porta al minore numero di page fault e non soffre dell'anomalia di Belady.
- Ma come si può prevedere quando verrà riusata una pagina?
- Algoritmo usato (come riferimento) in confronti con altri algoritmi.

Algoritmo Least Recently Used (LRU)

- Approssimazione di OPT: studiare il passato per prevedere il futuro.
- Si rimpiazza la pagina che da più tempo non viene usata.
- Con 4 frame: 8 page fault.
- È la soluzione ottima con ricerca all'indietro nel tempo: LRU su una stringa di riferimenti r è OPT sulla stringa $reverse(r)$.
- Quindi la frequenza di page fault per la LRU è la stessa di OPT su stringhe invertite.
- Non soffre dell'anomalia di Belady (è un algoritmo di stack).
- Generalmente è una buona soluzione.
- Problema: LRU necessita di notevole assistenza hardware.

Matrice di memoria

Dato un algoritmo di rimpiazzamento, e una reference string, si definisce la **matrice di memoria**: $M(m, r)$ è l'insieme delle pagine caricate all'istante r avendo m frame a disposizione.
Esempio di matrice di memoria per LRU:

Reference string	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
	0	2	1	3	5	4	6	3	7	4	7	3	3	5	5	3	1	1	1	7	1	3	4	1
		0	2	1	3	5	4	6	3	7	4	7	7	3	3	5	3	3	3	1	7	1	3	4
			0	2	1	3	5	4	6	3	3	4	4	7	7	7	5	5	5	3	3	7	1	3
				0	2	1	3	5	4	6	6	6	6	4	4	4	7	7	7	5	5	5	7	7
					0	2	1	1	5	5	5	5	5	6	6	6	4	4	4	4	4	4	5	5
						0	2	2	1	1	1	1	1	1	1	1	6	6	6	6	6	6	6	6
							0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
									0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Page faults	P	P	P	P	P	P	P		P					P			P						P	
Distance string	∞	∞	∞	∞	∞	∞	∞	4	∞	4	2	3	1	5	1	2	6	1	1	4	2	3	5	3

Algoritmi di Stack

Un algoritmo di rimpiazzamento si dice di stack se per ogni reference string r, per ogni memoria m:

$$M(m, r) \subseteq M(m + 1, r)$$

Ad esempio, OPT e LRU sono algoritmi di stack. FIFO non è di stack.

Fatto: Gli algoritmi di stack non soffrono dell'anomalia di Belady.

Implementazioni di LRU

Implementazione a contatori

- La MMU ha un contatore (32-64 bit) che viene automaticamente incrementato dopo ogni accesso in memoria.
- Ogni entry nella page table ha un registro (reference time).
- Ogni volta che si fa riferimento ad una pagina, si copia il contatore nel registro della entry corrispondente.
- Quando si deve liberare un frame, si cerca la pagina con il registro più basso.

Molto dispendioso, se la ricerca viene parallelizzata in hardware.

Implementazioni di LRU

Implementazione a stack:

- si tiene uno stack di numeri di pagina in un lista double-linked;
- quando si fa riferimento ad una pagina, la si sposta sul top dello stack (Richiede la modifica di 6 puntatori);
- quando si deve liberare un frame, la pagina da swappare è quella in fondo allo stack: non serve fare una ricerca.

Implementabile in software (microcodice). Costoso in termini di tempo.

Approssimazioni di LRU: reference bit e NFU

Bit di riferimento (reference bit)

- Associare ad ogni pagina un bit R, inizialmente =0.
- Quando si fa riferimento alla pagina, R viene settato a 1.
- Si rimpiazza la pagina che ha $R = 0$ (se esiste).
- Non si può conoscere l'ordine: impreciso.

Variante: **Not Frequently Used** (NFU)

- Ad ogni pagina si associa un contatore.
- Ad intervalli regolari (tick, tip. 10-20ms), per ogni entry si somma il reference bit al contatore.
- Problema: pagine usate molto tempo fa contano come quelle recenti.

Approssimazioni di LRU: aging

Aggiungere bit supplementari di riferimento, con peso diverso.

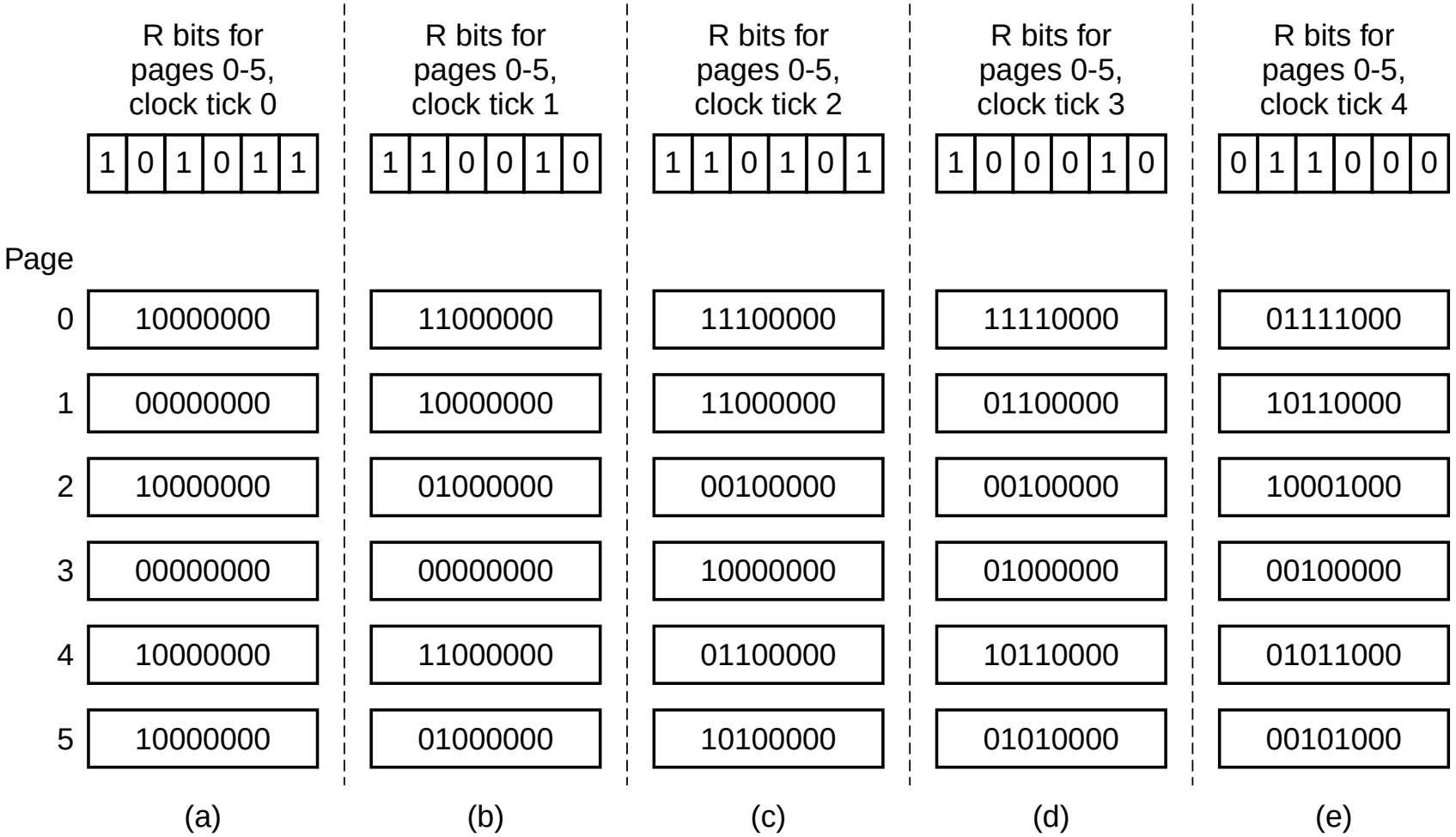
- Ad ogni pagina si associa un array di bit, inizialmente =0.
- Ad intervalli regolari, un interrupt del timer fa partire una routine che shifta gli array di tutte le pagine immettendovi i bit di riferimento, che vengono settati a 0.
- Si rimpiazza la pagina che ha il numero più basso nell'array.

Differenze con LRU:

- Non può distinguere tra pagine accedute nello stesso tick.
- Il numero di bit è finito $\Rightarrow$ la memoria è limitata.

In genere comunque è una buona approssimazione.

Approssimazioni di LRU: aging



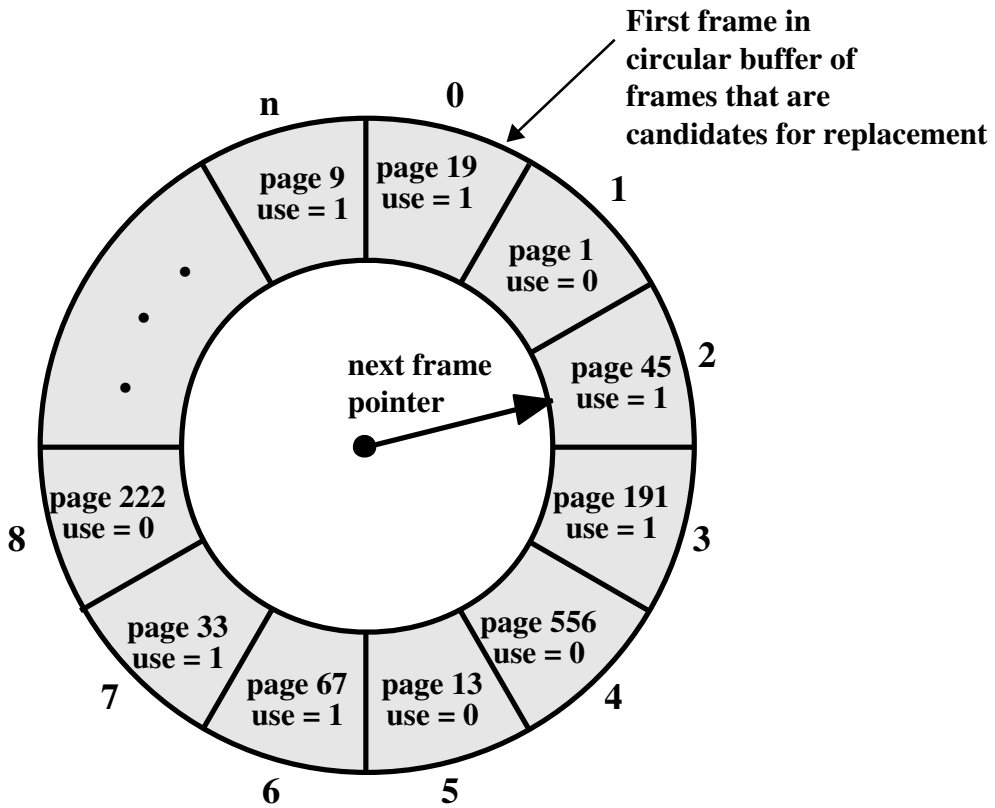
Approssimazioni di LRU: CLOCK (o “Second chance”)

Idea di base: se una pagina è stata usata pesantemente di recente, allora probabilmente verrà usata pesantemente anche prossimamente.

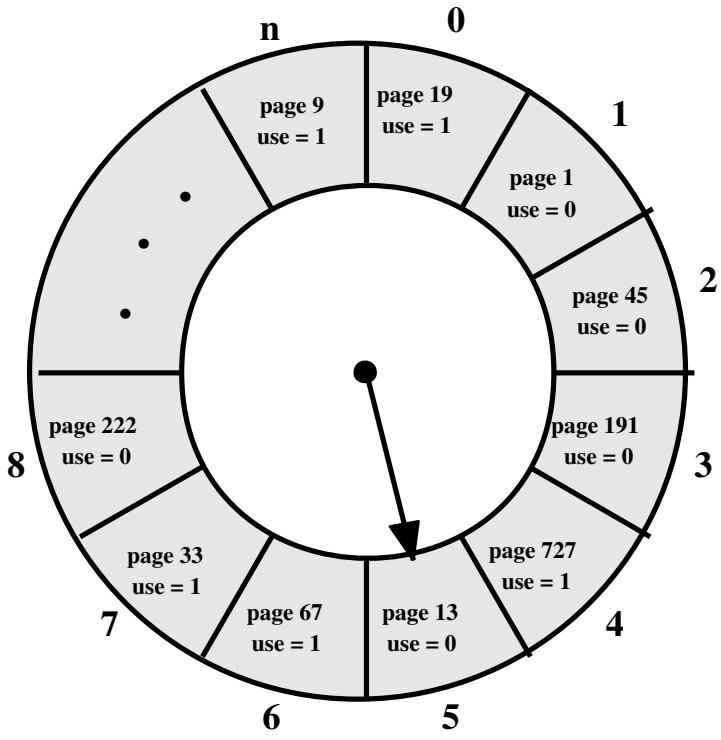
- Utilizza il reference bit.
- Si segue un ordine “ad orologio”.
- Se la pagina candidato ha il reference bit = 0, rimpiazzala.
- se ha il bit = 1, allora:
  - imposta il reference bit a 0,
  - lascia la pagina in memoria,
  - passa alla prossima pagina, seguendo le stesse regole.

Nota: se tutti i bit=1, degenera in un FIFO.
Buona approssimazione di LRU; usato (con varianti) in molti sistemi.

Approssimazioni di LRU: CLOCK (o “Second chance”)



(a) State of buffer just prior to a page replacement

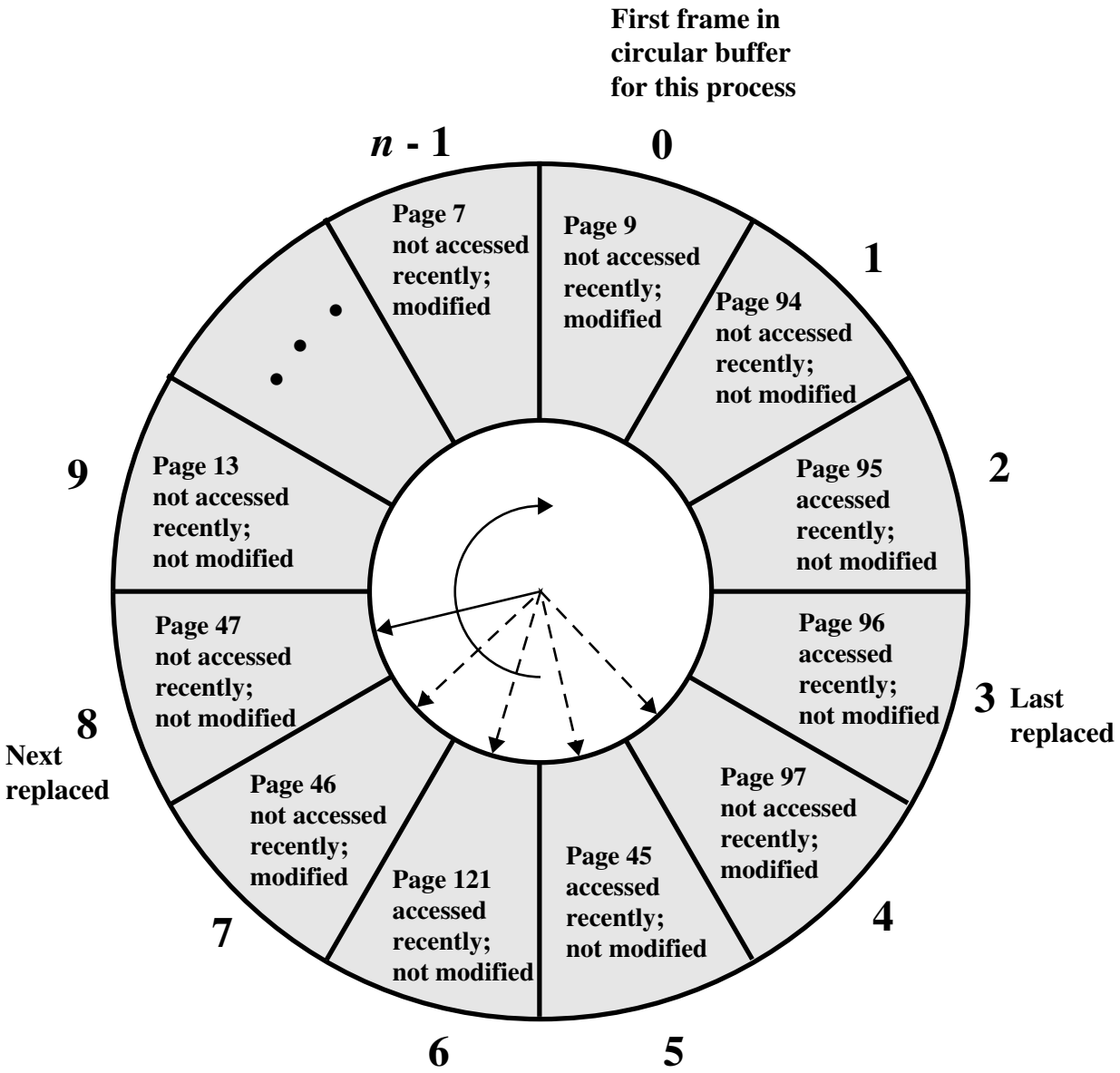


(b) State of buffer just after the next page replacement

Approssimazioni di LRU: CLOCK migliorato

- Usare due bit per pagina: il reference (r) e il dirty (d) bit,
  - non usata recentemente, non modificata ($r = 0, d = 0$): buona,
  - non usata recentemente, ma modificata ($r = 0, d = 1$): meno buona,
  - usata recentemente, non modificata ($r = 1, d = 0$): probabilmente verrà riusata,
  - usata recentemente e modificata ($r = 1, d = 1$): molto usata.
- Si scandisce la coda dei frame più volte:
  1. cerca una pagina con (0,0) senza modificare i bit; fine se trovata,
  2. cerca una pagina con (0,1) azzerando i reference bit; fine se trovata,
  3. vai a 1.
- Usato nel MacOS tradizionale (fino alla versione 9.x)

Approssimazioni di LRU: CLOCK migliorato

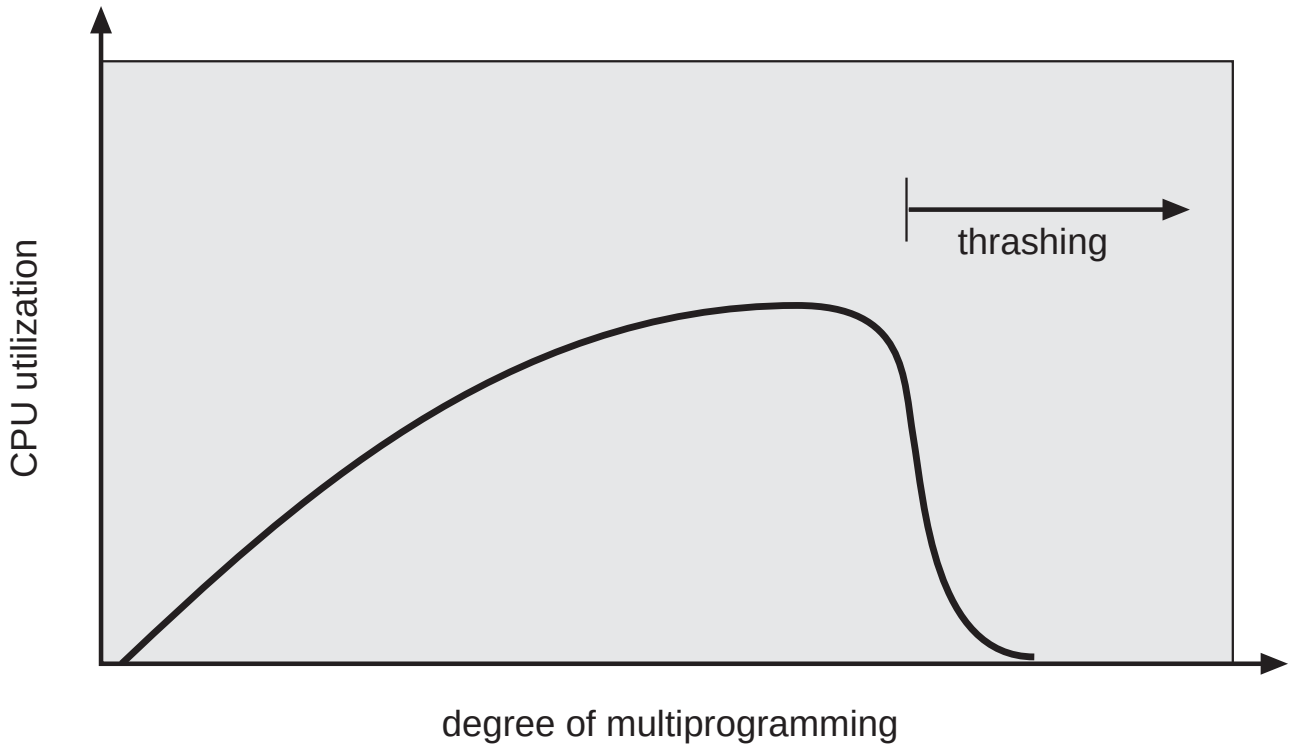


Thrashing

- Se un processo non ha “abbastanza” pagine, il page-fault rate è molto alto. Questo porta a quanto segue:
  - basso utilizzo della CPU (i processi sono impegnati in I/O),
  - il S.O. potrebbe pensare che deve aumentare il grado di multiprogrammazione (errore!),
  - un altro processo viene caricato in memoria.
- Thrashing: uno o più processi spendono la maggior parte del loro tempo a swappare pagine dentro e fuori.
- Il thrashing di un processo avviene quando la memoria assegnatagli è inferiore a quella richiesta dalla sua località.

Thrashing

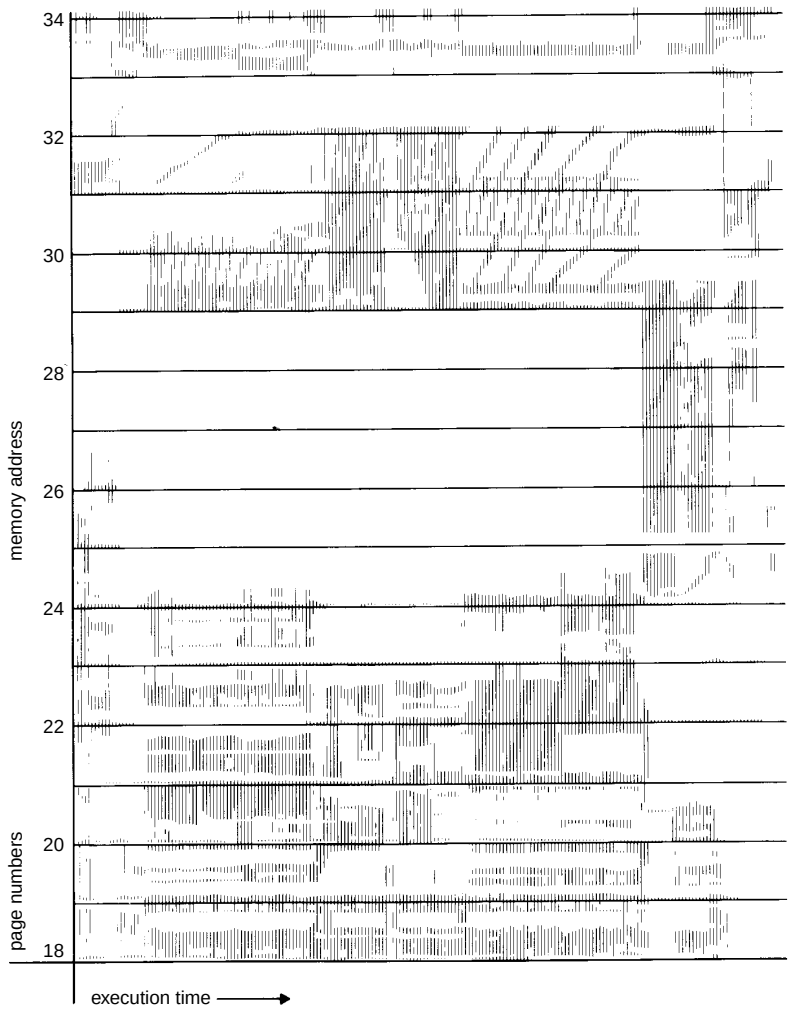
- Il thrashing del sistema avviene quando la memoria fisica è inferiore alla somma delle località dei processi in esecuzione. Può essere causato da un processo che si espande e in presenza di una politica di rimpiazzamento globale.



Principio di località

Ma allora, perché la paginazione funziona? Per il principio di località

- Una località è un insieme di pagine che vengono utilizzate attivamente assieme dal processo.
- Il processo, durante l'esecuzione, migra da una località all'altra.
- Le località si possono sovrapporre.



Impedire il thrashing: modello del working-set

- $\Delta \equiv$ working-set window $\equiv$ un numero fisso di riferimenti a pagine.
  Esempio: le pagine a cui hanno fatto riferimento le ultime 10.000 instruzioni.
- WSS_i (working set del processo P_i) = numero totale di pagine riferite nell'ultimo periodo Δ. Varia nel tempo.
  - Se Δ è troppo piccolo, il WS non copre l'intera località.
  - Se Δ è troppo grande, copre più località.
  - Se $\Delta = \infty \Rightarrow$ copre l'intero programma e dati.
- $D = \Sigma\ WSS_i \equiv$ totale frame richiesti.
- Sia $m =$ n. di frame fisici disponibile. Se $D > m \Rightarrow$ thrashing.

Algoritmo di allocazione basato sul working set

- Il sistema monitorizza il ws di ogni processo, allocandogli frame sufficienti per coprire il suo ws.
- Alla creazione di un nuovo processo, questo viene ammesso nella coda ready solo se ci sono frame liberi sufficienti per coprire il suo ws.
- Se $D > m$, allora si sospende uno dei processi per liberare la sua memoria per gli altri (diminuire il grado di multiprogrammazione — scheduling di medio termine).

Si impedisce il thrashing, massimizzando nel contempo l'uso della CPU.

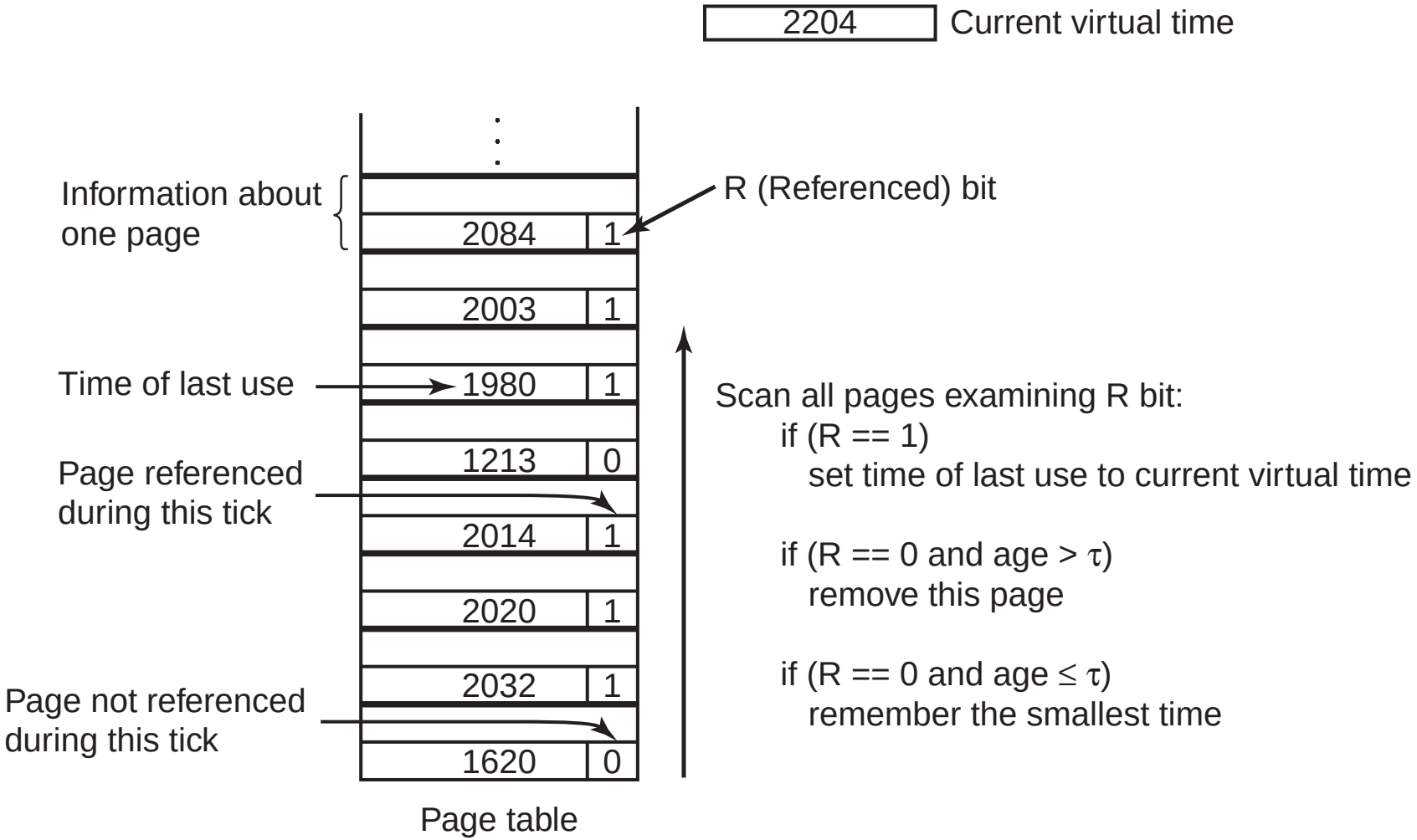
Approssimazione del working set: registri a scorrimento

- Si approssima con un timer e il bit di riferimento.
- Esempio: $\Delta = 10.000$.
  - Si mantengono due bit per ogni pagina (oltre al reference bit).
  - Il timer manda un interrupt ogni 5000 unità di tempo.
  - Quando arriva l'interrupt, si shifta il reference bit di ogni pagina nei due bit in memoria, e lo si cancella.
  - Quando si deve scegliere una vittima: se uno dei tre bit è a 1, allora la pagina è nel working set.
- Implementazione non completamente accurata (scarto di 5000 accessi).
- Miglioramento: 10 bit e interrupt ogni 1000 unità di tempo $\Rightarrow$ più preciso ma anche più costoso da gestire.

Approssimazione del working set: tempo virtuale

- Si mantiene un tempo virtuale corrente del processo (=n, di tick consumati dal processo).
- Si eliminano pagine più vecchie di τ tick.
- Ad ogni pagina, viene associato un registro contenente il tempo di ultimo riferimento.
- Ad un page fault, si controlla la tabella alla ricerca di una vittima:
  - se il reference bit è a 1, si copia il TVC nel registro corrispondente, il reference viene azzerato e la pagina viene saltata;
  - se il reference è a 0 e l'età $> \tau$, la pagina viene rimossa;
  - se il reference è a 0 e l'età $\leq \tau$, si segna quella più vecchia (con minore tempo di ultimo riferimento); alla peggio, questa viene cancellata.

Approssimazione del working set: tempo virtuale



Algoritmo di rimpiazzamento WSClock

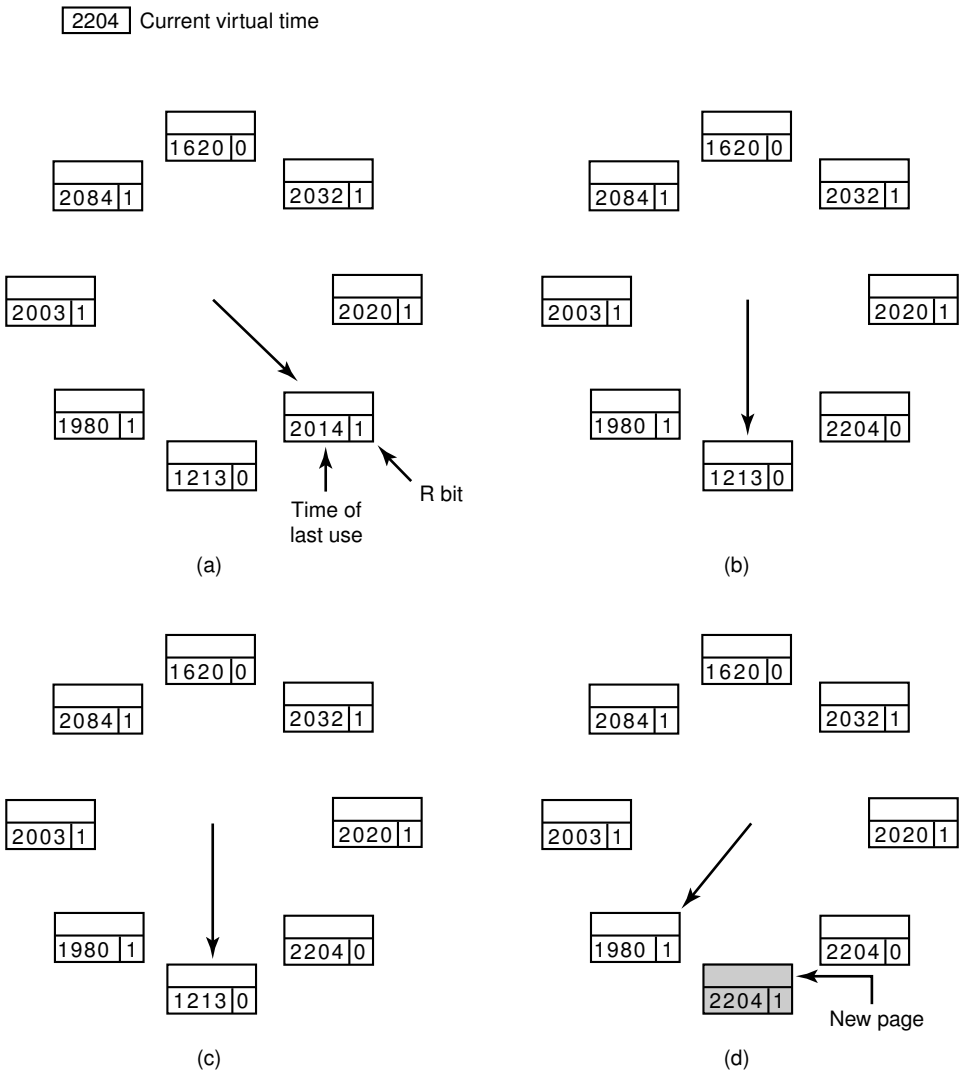
Variante del Clock che tiene conto del Working Set. Invece di contare i riferimenti, si tiene conto di una finestra temporale τ fissata (es. 100ms):

- si mantiene un contatore T del tempo di CPU impiegato da ogni processo;
- le pagine sono organizzate ad orologio; inizialmente, lista vuota;
- ogni entry contiene i reference e dirty bit R, M, e un registro Time of last use, che viene copiato dal contatore durante l'algoritmo: la differenza tra questo registro e il contatore si chiama età della pagina;

Algoritmo di rimpiazzamento WSClock

- ad un page fault, si guarda prima la pagina indicata dal puntatore:
  - se $R = 1$, si mette $R = 0$, si copia $TLU = T$ e si passa avanti,
  - se $R = 0$ e età $\leq \tau$: è nel working set: si passa avanti,
  - se $R = 0$ e età $> \tau$: se $M = 0$ allora si libera la pagina, altrimenti si schedula un pageout e si passa avanti.
- Cosa succede se si fa un giro completo?
  - Se almeno un pageout è stato schedulato, si continua a girare (aspettando che le pagine schedulate vengano salvate).
  - Altrimenti, significa che tutte le pagine sono nel working set. Soluzione semplice: si rimpiazza una qualsiasi pagina pulita.
  - Se non ci sono neanche pagine pulite, si rimpiazza la pagina corrente.

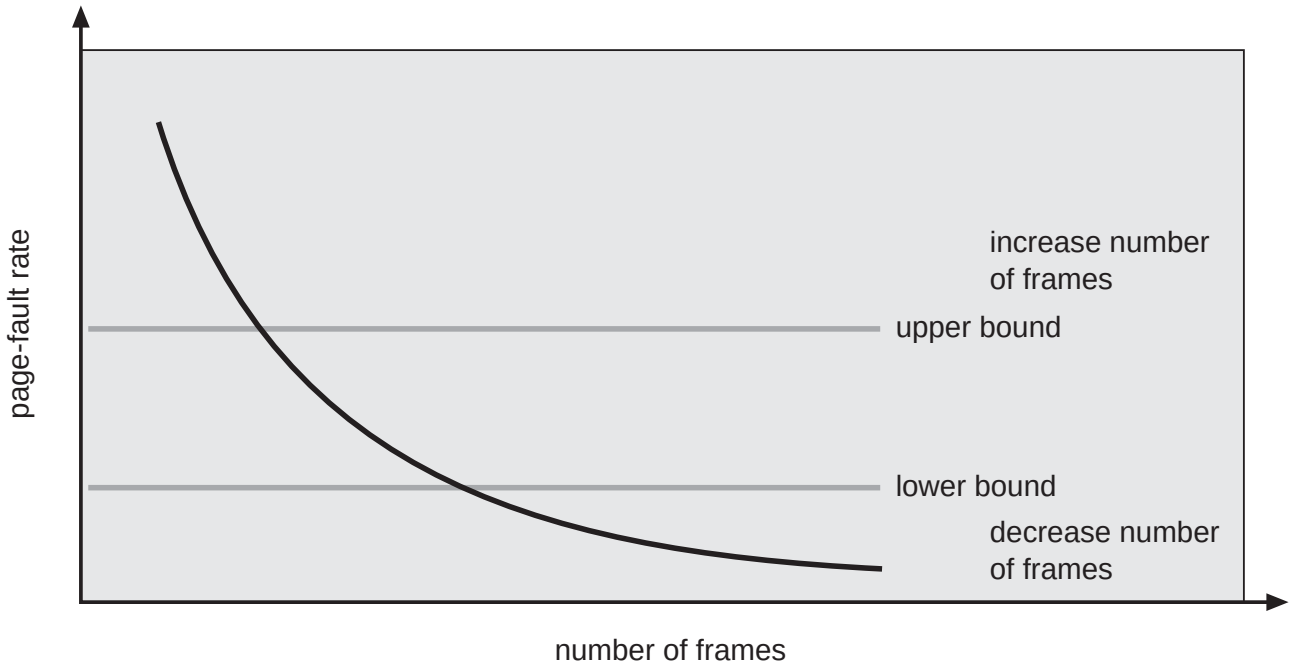
Algoritmo di rimpiazzamento WSClock



(a-b): cosa succede quando $R == 1$.

(c-d): cosa succede quando $R == 0$ e $T - TLU > \tau$.

Impedire il thrashing: frequenza di page-fault



Si stabilisce un page-fault rate “accettabile”

- Se quello attuale è troppo basso, il processo perde un frame
- Se quello attuale è troppo alto, il processo guadagna un frame

Nota: si controlla solo il n. di frame assegnati, non quali pagine sono caricate.

Sostituzione globale vs. locale

- Sostituzione locale: ogni processo può rimpiazzare solo i proprî frame.
  - Mantiene fisso il numero di frame allocati ad un processo (anche se ci sono frame liberi).
  - Il comportamento di un processo non è influenzato da quello degli altri processi.
- Sostituzione globale: un processo sceglie un frame tra tutti i frame del sistema.
  - Un processo può “rubare” un frame ad un altro.
  - Sfrutta meglio la memoria fisica.
  - Il comportamento di un processo dipende da quello degli altri.
- Dipende dall’algoritmo di rimpiazzamento scelto: se è basato su un modello di ws, si usa una sostituzione locale, altrimenti globale.

Algoritmi di allocazione dei frame

- Ogni processo necessita di un numero minimo di pagine imposto dall'architettura (Es.: su IBM 370, possono essere necessarie 6 pagine per poter eseguire l'istruzione `MOV`).

Diversi modi di assegnare i frame ai varî processi.

- Allocazione libera: dare a qualsiasi processo quanti frame desidera.
  Funziona solo se ci sono sufficienti frame liberi.
- Allocazione equa: stesso numero di frame ad ogni processo.
  Porta a sprechi (non tutti i processi hanno le stesse necessità).

Algoritmi di allocazione dei frame

- Allocazione proporzionale: un numero di frame in proporzione a
  - dimensione del processo,
  - sua priorità (Solitamente, ai page fault si prendono frame ai processi a priorità inferiore).

Esempio: due processi da 10 e 127 pagine, su 62 frame:

$$\frac{10}{127+10} * 62 \cong 4 \qquad \frac{127}{127+10} * 62 \cong 57$$

L'allocazione varia al variare del livello di multiprogrammazione: se arriva un terzo processo da 23 frame:

$$\frac{10}{127+10+23} * 62 \cong 3 \qquad \frac{127}{127+10+23} * 62 \cong 49 \qquad \frac{23}{127+10+23} * 62 \cong 8$$

Buffering di pagine

Aggiungere un insieme (free list) di frame liberi agli schemi visti.

- Il sistema cerca di mantenere sempre un po' di frame sulla free list
- quando si libera un frame:
  - se è stato modificato lo si salva su disco,
  - si mette il suo dirty bit a 0,
  - si sposta il frame sulla free list senza cancellarne il contenuto.
- Quando un processo produce un page fault:
  - si vede se la pagina è per caso ancora sulla free list (soft page fault),
  - altrimenti, si prende dalla free list un frame, e vi si carica la pagina richiesta dal disco (hard page fault).

Altre considerazioni

- Prepaging, ovvero, caricare in anticipo le pagine che “probabilmente” verranno usate:
  - applicato al lancio dei programmi e al ripristino di processi sottoposti a swapout di medio termine.
- Selezione della dimensione della pagina: solitamente imposta dall’architettura. Dimensione tipica: 4K-8K. Influenza:
  - frammentazione: meglio piccola,
  - dimensioni della page table: meglio grande,
  - quantità di I/O: meglio piccola,
  - tempo di I/O: meglio grande,
  - località: meglio piccola,
  - n. di page fault: meglio grande.

Altre considerazioni

- La struttura del programma può influenzare il page-fault rate.
  - Array A[1024,1024] **of** integer.
  - Ogni riga è memorizzata in una pagina.
  - Un frame a disposizione.

Programma 1

```
for j := 1 to 1024 do
  for i := 1 to 1024 do
    A[i,j] := 0;
```

1024 × 1024 page faults

Programma 2

```
for i := 1 to 1024 do
  for j := 1 to 1024 do
    A[i,j] := 0;
```

1024 page faults

- Durante I/O, i frame contenenti i buffer non possono essere swappati:
  - I/O solo in memoria di sistema ⇒ costoso,
  - bloccare in memoria i frame contenenti buffer di I/O (I/O interlock) ⇒ delicato (un frame lockato potrebbe non essere più rilasciato).