

Memoria Virtuale

Memoria Virtuale

- Definizione
 - è la tecnica che permette l'esecuzione di processi che non sono completamente caricati in memoria principale
- Considerazioni
 - permette di eseguire in concorrenza processi che nel loro complesso (o anche singolarmente) hanno necessità di memoria maggiore di quella disponibile
 - cioè permette di separare la memoria logica vista dall'utente (programmatore) dalla memoria fisica

Memoria Virtuale

- Requisiti di un'architettura di Von Neumann (ciclo di fetch ecc.)
 - Le istruzioni da eseguire e i dati su cui operano devono essere in memoria principale
- tuttavia
 - non è necessario che l'intero spazio di indirizzamento logico di un processo sia in memoria principale
 - i processi non utilizzano tutto il loro spazio di indirizzamento *contemporaneamente*
 - Ad esempio
 - * routine di gestione di errore
 - * strutture dati allocate con dimensioni massime ma utilizzate solo parzialmente

Memoria Virtuale

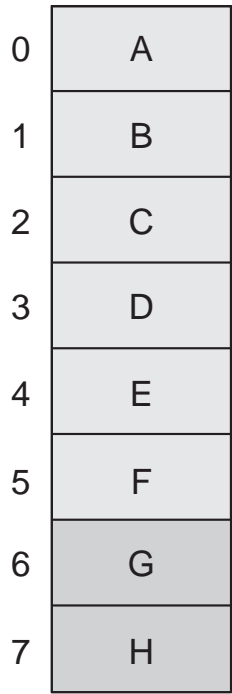
- Implementazione

- ogni processo ha accesso ad uno *spazio di indirizzamento virtuale* che può essere più grande di quello fisico
- gli indirizzi virtuali
 - * possono essere mappati su indirizzi fisici della memoria principale
 - * oppure, possono essere mappati su memoria secondaria (disco)
- in caso di accesso ad indirizzi virtuali mappati in memoria secondaria
 - * i dati associati vengono trasferiti in memoria principale
 - * se la memoria principale è piena, si spostano in memoria secondaria i dati in memoria principale che sono considerati meno utili

Memoria Virtuale - Implementazione

- Paginazione su richiesta (*demand paging*)
 - si utilizza la tecnica della paginazione, ammettendo però che alcune pagine possano essere in memoria secondaria
- Nella tabella delle pagine di ogni singolo processo
 - si utilizza un bit (bit di validità) che indica se la pagina è presente o meno in memoria principale
- Quando un processo tenta di accedere ad una pagina non in memoria
 - il processore genera una *trap* (*page fault*)
 - una componente del S.O. (*pager*) si occupa di caricare la pagina mancante e di aggiornare di conseguenza la tabella delle pagine

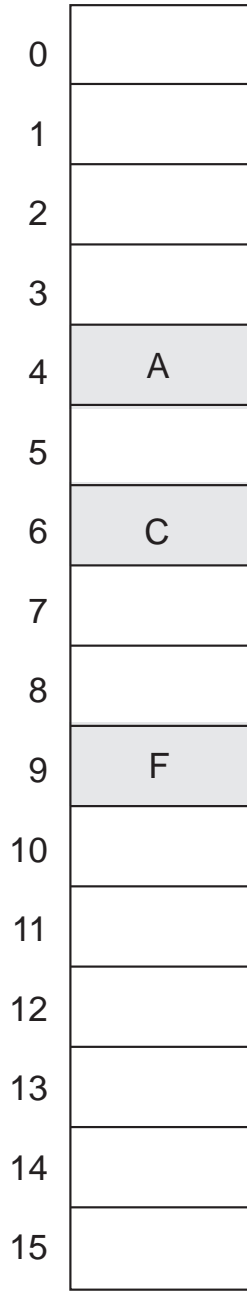
Paginazione su richiesta - Esempio



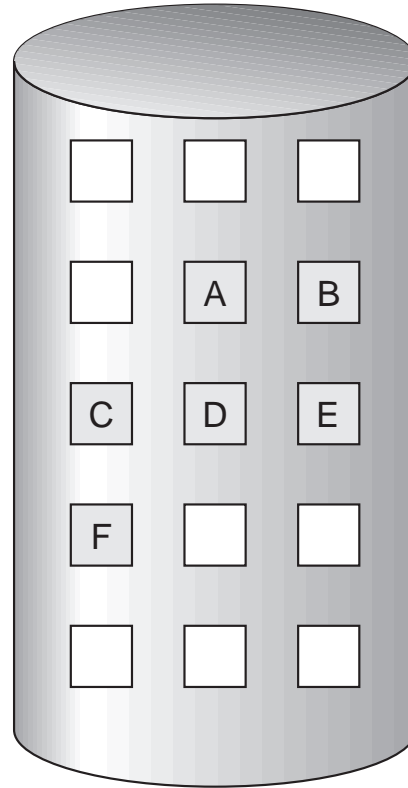
logical memory

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

page table



physical memory



Swapping vs. Paging

Spesso si confonde *swapping* con *paging*

- Swap:
 - con questo termine si intende l'azione di copiare l'intera area di memoria usata da un processo
 - * dalla memoria secondaria alla memoria principale (*swap-in*)
 - * dalla memoria principale alla memoria secondaria (*swap-out*)
 - Utilizzata fino all'introduzione del demand paging
- Swapper: processo che implementa una politica di swapping (scheduling di medio termine)

- Demand Paging
 - scambio (tra memoria principale e secondaria) di gruppi di pagine appartenenti ai processi
 - può essere vista come una tecnica di swap di tipo *lazy*, cioè viene caricato solo ciò che serve
- Pager: processo che implementa una politica di gestione delle pagine dei processi
- Anche se è una terminologia obsoleta, in alcuni S.O. il pager viene chiamato swapper (es.: Linux: `kswapd`)
- La *swap area* indica comunque l'area utilizzata dal pager in memoria secondaria (per ospitare le pagine)

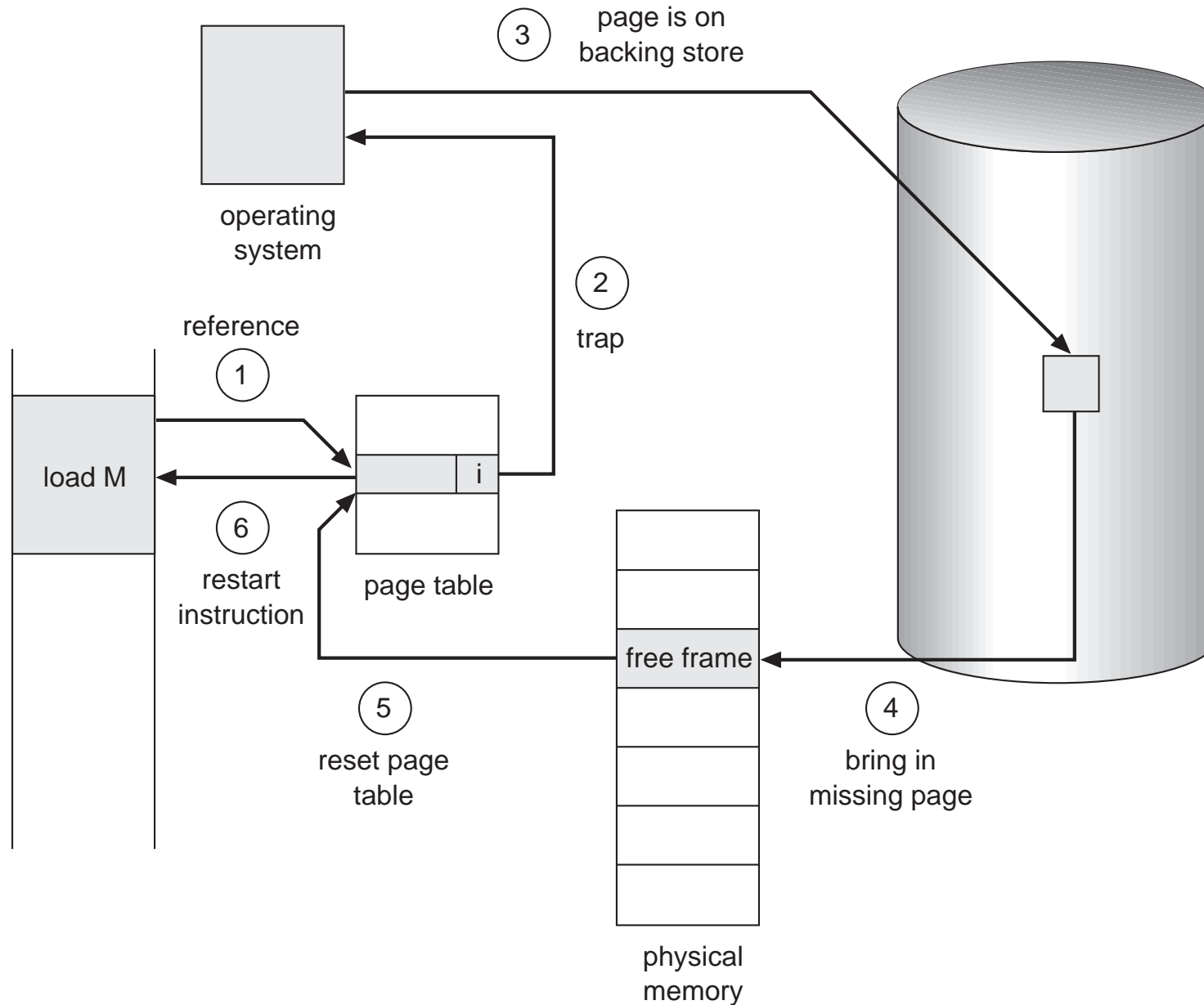
Gestione del Page Fault

Vediamo più in dettaglio come viene gestito un page fault

- Supponiamo che la memoria principale sia completamente allocata
- Il processo in CPU esegue l'istruzione `load M` dove `M` è un indirizzo virtuale
- Il S.O. controlla che `M` sia un'indirizzo all'interno dello spazio di indirizzamento del processo in questione; in caso contrario genera un'eccezione (segmentation fault) e abortisce il programma
- Se l'accesso è valido e la pagina logica richiesta è in memoria principale:
 - si accede alla locazione di memoria nella corrispondente pagina fisica e si completa l'istruzione

- Se la pagina logica richiesta non è in memoria principale (bit di validità=0):
 - si invoca (tramite una trap) la routine di gestione dei page fault che deve:
 - * Trovare qualche pagina in memoria, che non sia usata, e scaricarla nella swap area su disco (*swap-out* della pagina)
 - * Caricare la pagina richiesta nel frame così liberato (*swap-in* della pagina)
 - * Aggiornare le tabelle delle pagine (settando il bit di validità a 1)
 - L'istruzione che ha causato il page fault deve essere rieseguita in modo consistente

Gestione dei Page Fault - Esempio



Performance della paginazione su richiesta

- Sia p la frequenza di page fault $0 \leq p \leq 1$:
 - se $p = 0$ non si hanno page fault
 - se $p = 1$ ogni riferimento in memoria porta ad un page fault
- Inoltre siano
 - T_m = tempo per un accesso alla memoria principale
 - T_{pf} = overhead per la gestione del page fault
 - T_{sin} = tempo richiesto per swap-in
 - T_{sout} = tempo richiesto per swap-out
 - T_r = overhead richiesto per restart di un'istruzione
- Allora il *tempo effettivo di accesso* in memoria (EAT) corrisponde a

$$EAT = (1 - p) \cdot T_m + p \cdot (T_{pf} + T_{sin} + T_{sout} + T_r)$$

Valutazione della paginazione su richiesta

- Supponiamo che $T_m = 60$ nsec (tempo accesso alla mem. principale) e che $T_{sin} = T_{sout} = 5msec = 5 * 10^6$ nsec
- Inoltre assumiamo che il 50% delle volte che una pagina deve essere rimpiazzata debba essere scaricata su disco
cioè con frequenza $p/2$ dobbiamo effettuare sia swap-in che swap-out

- Supponiamo che l'overhead di page fault e restart siano trascurabili allora

$$\begin{aligned} EAT &= 60 \cdot (1 - p) + (T_{sin} + T_{sout}) \cdot p/2 + (T_{sin}) \cdot p/2 \\ &= 60 \cdot (1 - p) + 5 \cdot 10^6 \cdot 1.5 \cdot p = 60 + (7.5 \cdot 10^6 - 60)p(nsec) \end{aligned}$$

- Si ha un degrado del 10% rispetto a T_m (cioè EAT 110 percento di T_m)
quando $66 = 60 + (7.5 \cdot 10^6 - 60) \cdot p$ cioè

$$p = 6 / (7.5 \cdot 10^6 - 60) = 1 / 1250000$$

- cioè se si verifica un page fault ogni milione (circa) di riferimenti in memoria

Considerazioni sul Demand Paging

- Problema di performance: serve un algoritmo di rimpiazzamento che porti al minor numero di page fault possibile
- L'accesso all'area di swap deve essere il più veloce possibile e quindi è meglio tenerla separata dal file system (possibilmente anche su un device dedicato) ed accedervi direttamente (senza passare per il file system).

Creazione dei processi: Copy-on-Write

La memoria virtuale con demand paging porta dei benefici anche alla fase di creazione dei processi, ad esempio con tecniche quali Copy-on-Write:

- La tecnica del 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.

- Questa tecnica permette una creazione più veloce dei processi
- Per evitare problemi di consistenza di dati le pagine libere devono essere allocate da un set di pagine azzerate

Memory-Mapped I/O

La memoria virtuale con demand paging può essere utilizzata anche per rendere più efficiente la gestione dell'I/O, ad esempio con tecniche quali Memory-Mapped file I/O:

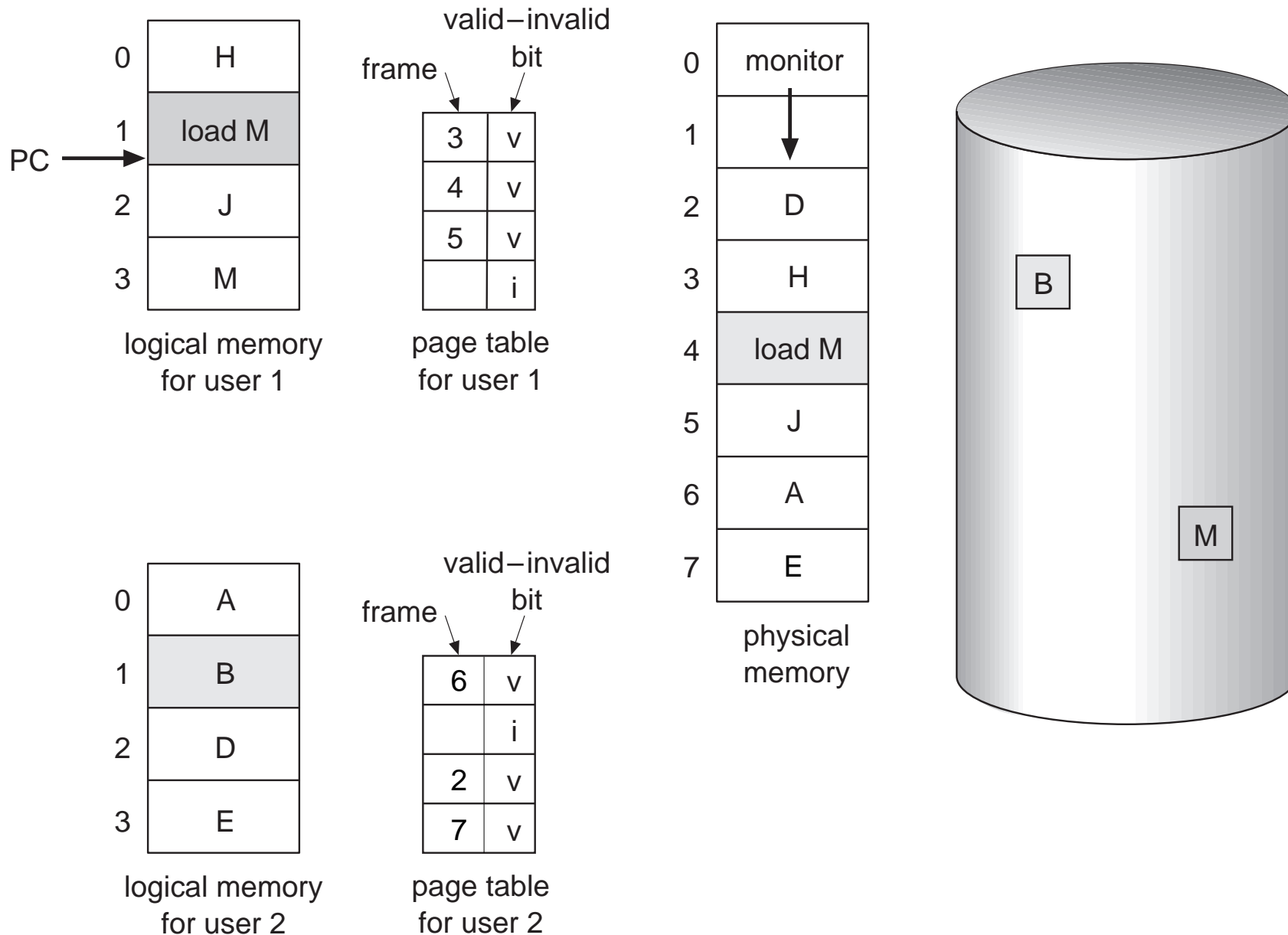
- La tecnica del Memory-mapped file I/O permette di gestire l'I/O di un file (dati in memoria secondaria) tramite accessi alla memoria principale:
 - Ogni blocco di un file viene *mappato* su una pagina di memoria virtuale
 - Un file può essere così letto come se fosse in memoria, con demand paging.
 - Dopo che un blocco è stato letto una volta, rimane caricato in memoria senza doverlo rileggere.
- Condividendo gli stessi frame in cui viene caricato, più processi possono condividere lo stesso file.

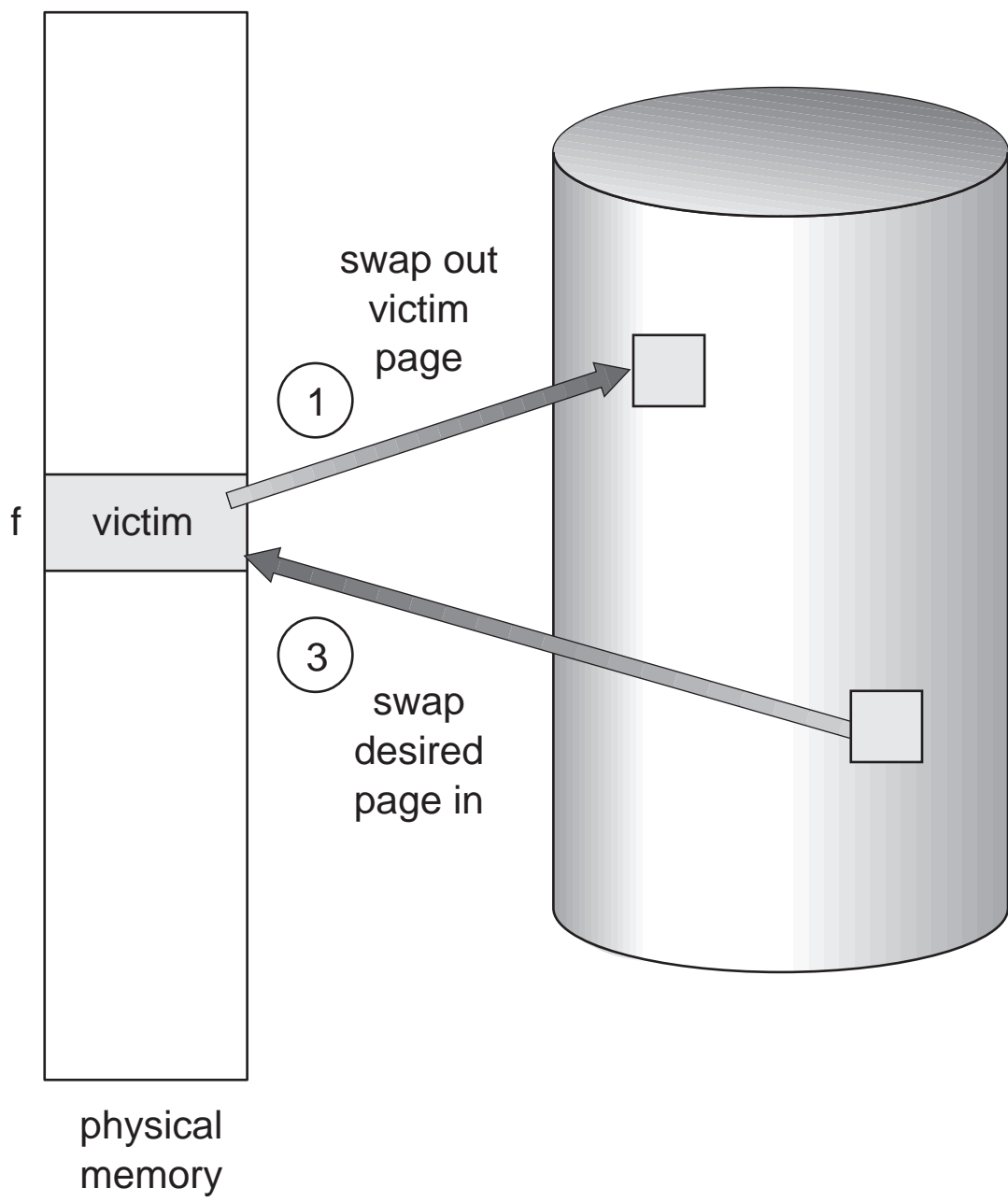
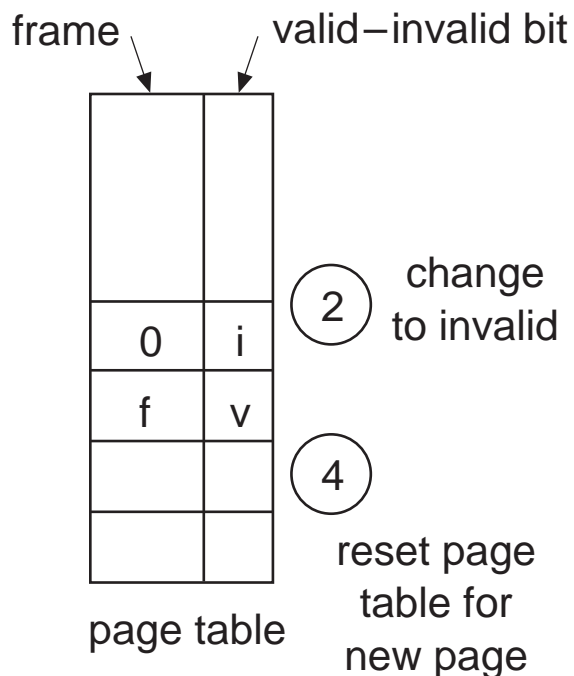
Strategie ed algoritmi per il rimpiazzamento delle pagine

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 esistono 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 (cont.)





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.
- In generale (ma non sempre) maggiore memoria implica minor tasso di page fault
- 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 di 5 pagine in questo ordine

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

Algoritmo First-In-First-Out (FIFO)

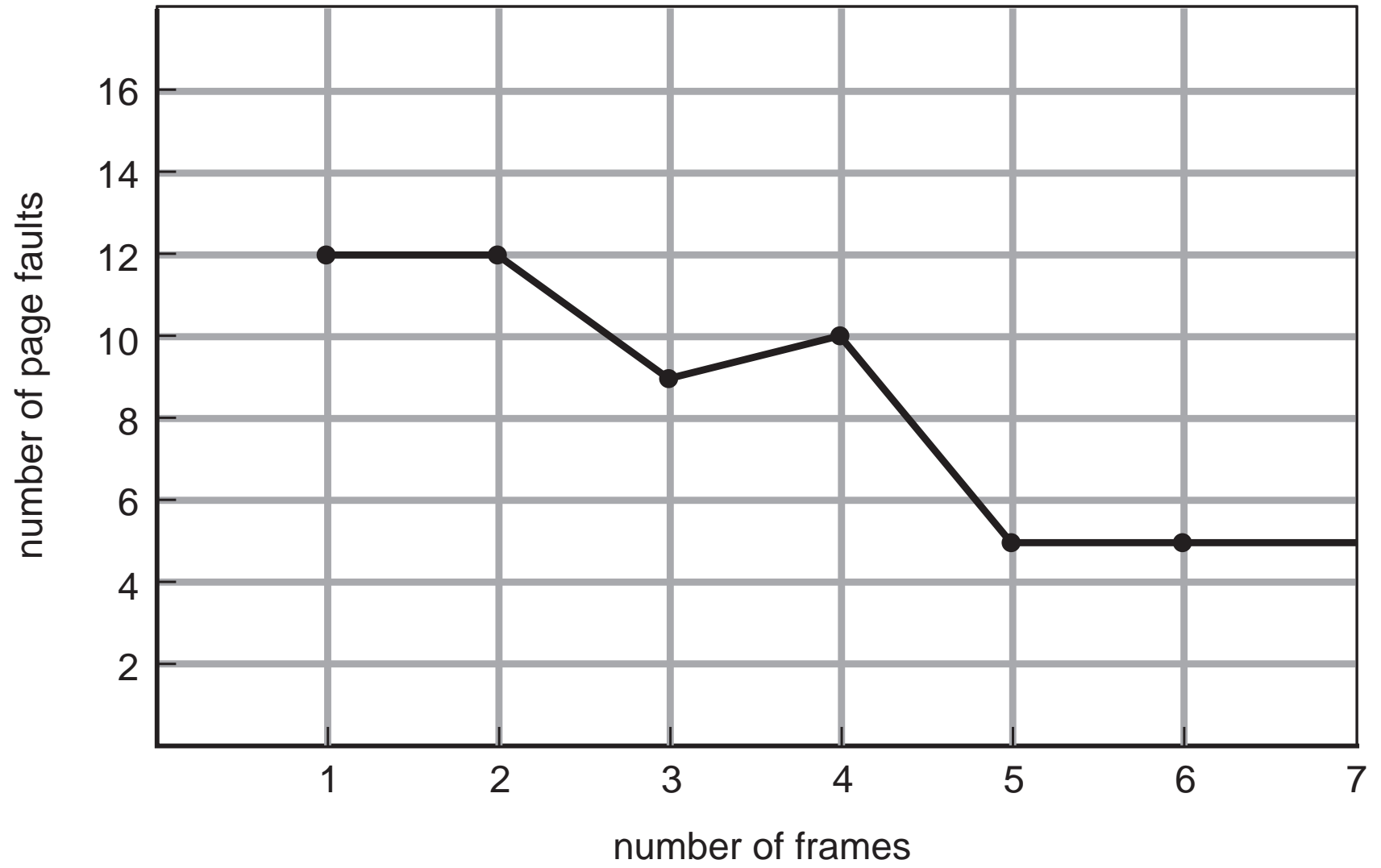
- Si rimpiazza la pagina che da più tempo è in memoria

- Data la sequenza:

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

- 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

↗ – page fault!



Algoritmo ottimale (OPT o MIN)

- Si rimpiazza la pagina che non verrà riusata per il periodo più lungo
- Data la sequenza

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

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 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 frames 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 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 riferisce 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 (Cont.)

Implementazione a stack

- si tiene uno stack di numeri di pagina in un lista double-linked (puntatori next, previous, head, tail)
- Quando si riferisce 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 riferisce 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

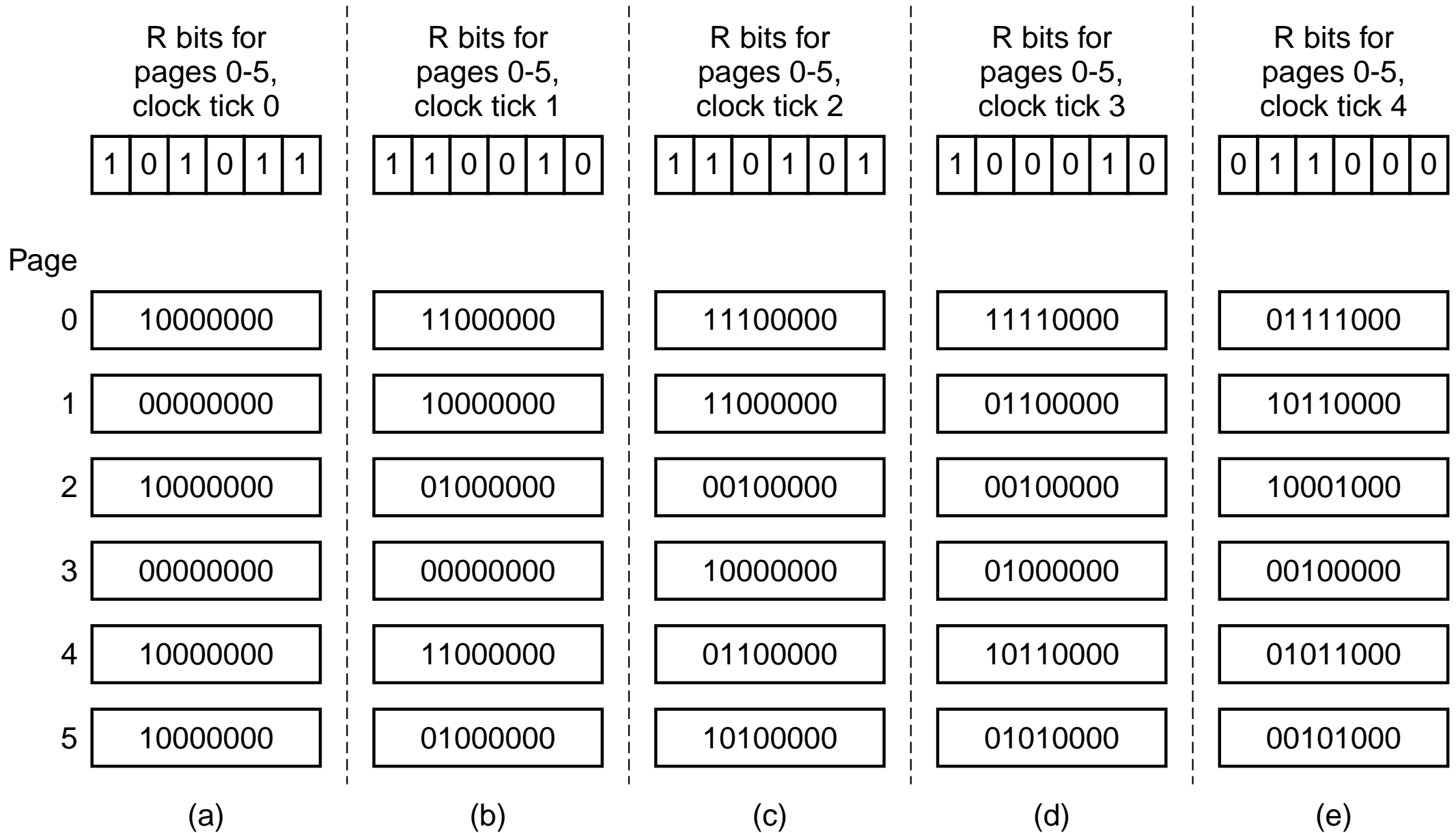
Aggiungere bit supplementari di riferimento, con peso diverso.

- Ad ogni pagina si associa un array di bit, inizialmente =0
- Ad intervalli regolari, un interrupt del timer fa partire una routine che sposta di un bit a destra (right-shift) gli array di tutte le pagine immettendo nel bit più significativo di ogni array il corrispondente bit di riferimento, che poi viene posto a 0
- Si rimpiazza la pagina che ha il numero binario più piccolo 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.



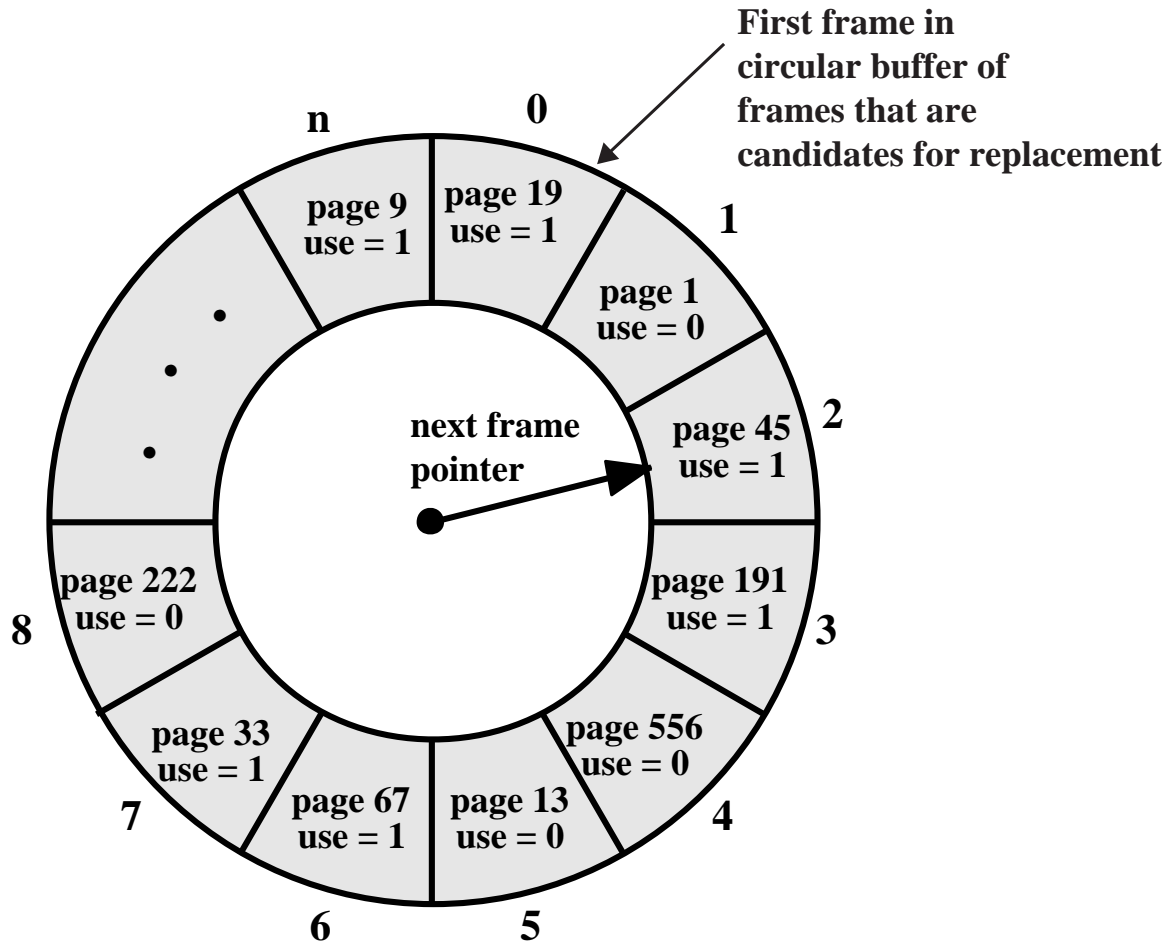
Approssimazioni di LRU: CLOCK (o “Second chance”)

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

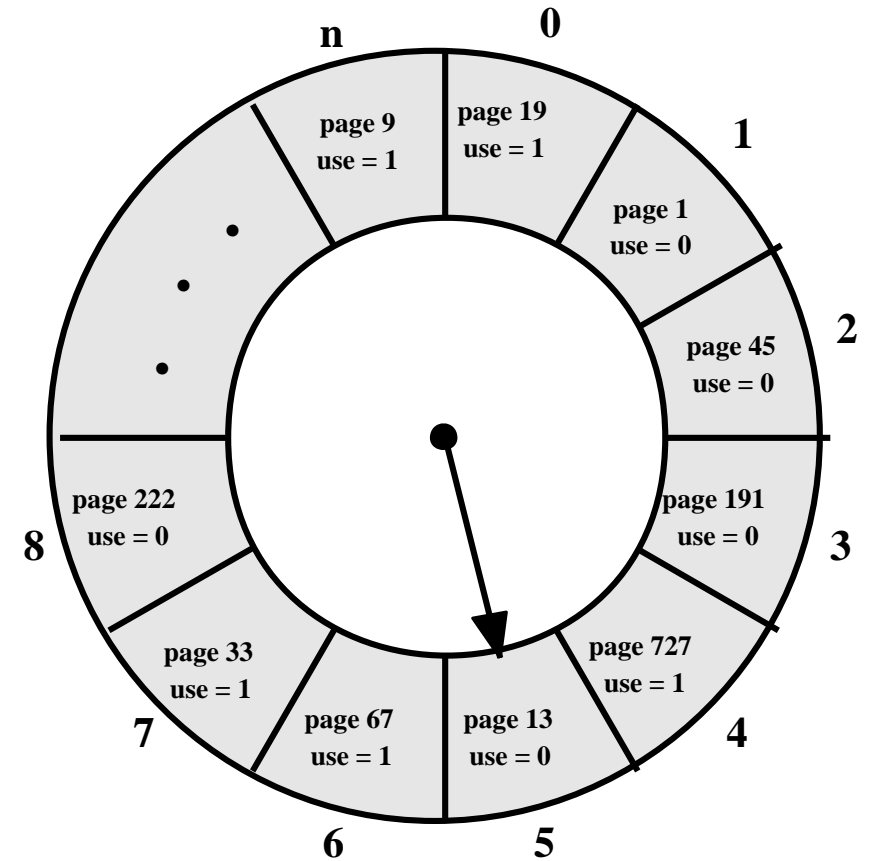
- Utilizza il reference bit.
- Si segue un ordine “ad orologio”
- Se la pagina candidato ha il reference bit = 0, rimpiazzala
- se ha il bit = 1, allora
 - imposta il reference bit 0.
 - lascia la pagina in memoria
 - passa alla prossima pagina, seguendo le stesse regole

Nota: se tutti i bit=1, degenera in un FIFO

Buona approssimazione di LRU; usato (con varianti) in molti sistemi



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

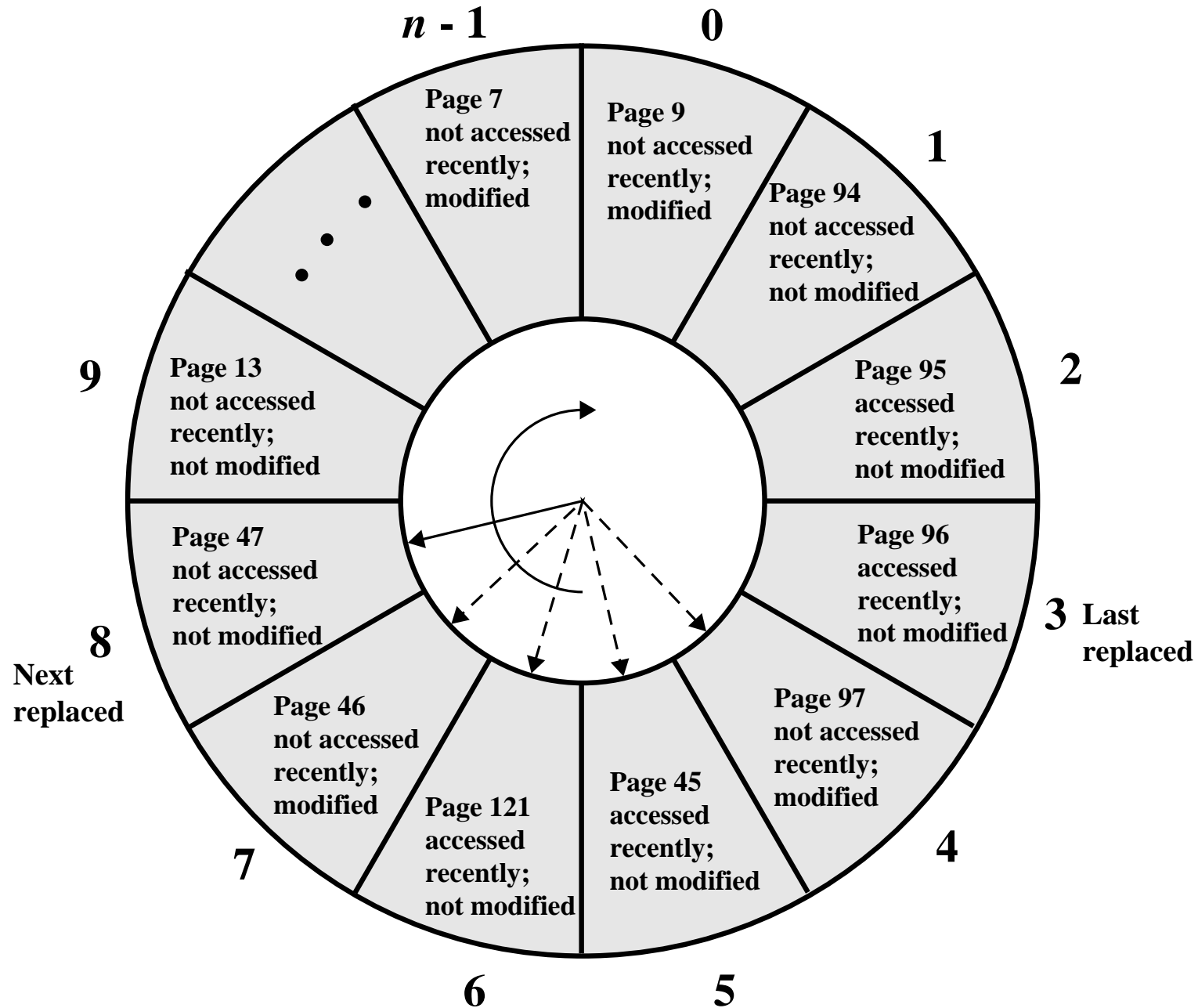


(b) State of buffer just after the next page replacement

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 a 9.x)

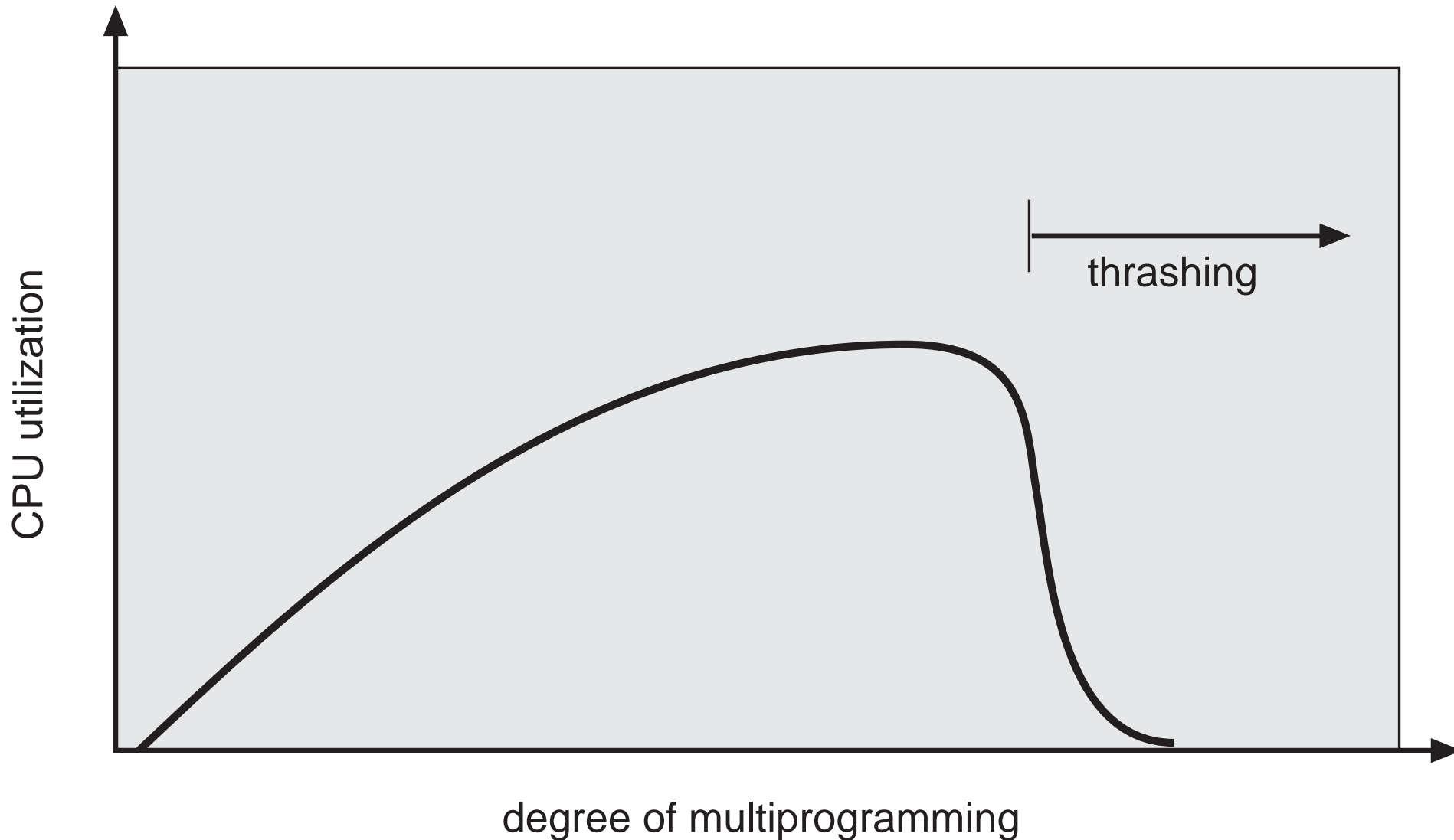
First frame in
circular buffer
for this process



Thrashing

- Se un processo non ha “abbastanza” pagine, il page-fault rate è molto alto. Questo porta a
 - 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

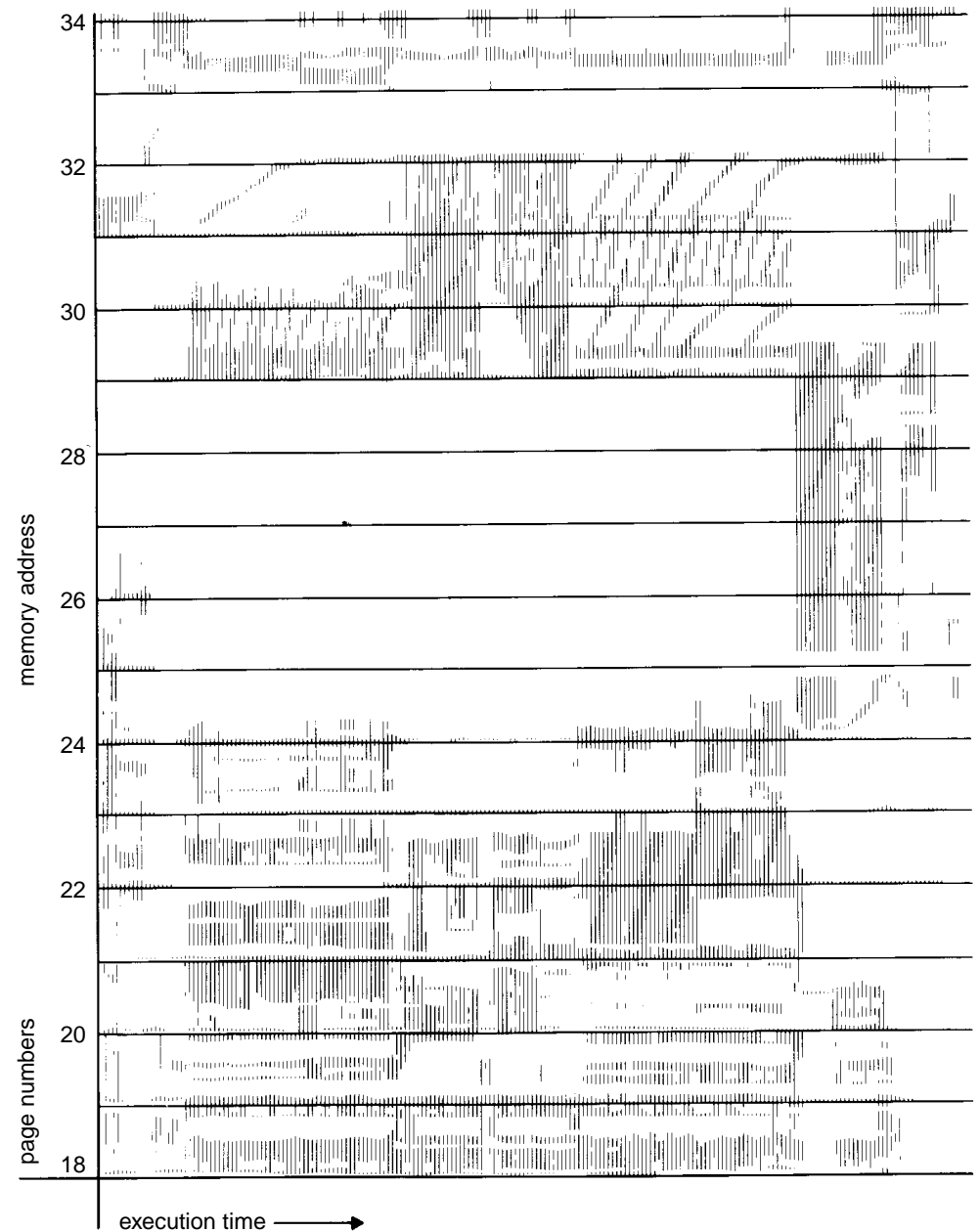
- Il thrashing del sistema avviene quando la memoria fisica è inferiore somma delle località dei processi in esecuzione. Può essere causato da un processo che si espande e in presenza di rimpiazzamento globale.



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 istruzioni

- 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
- $D = \sum WSS_i \equiv$ 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: $\Delta = 10000$
 - 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* (TVC) del processo (numero 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 marca quella più vecchia (con minore tempo di ultimo riferimento). Alla peggio, questa viene cancellata.

2204 Current virtual time

Information about one page {

Time of last use →

Page referenced during this tick →

Page not referenced during this tick →

⋮	
2084	1
2003	1
1980	1
1213	0
2014	1
2020	1
2032	1
1620	0

R (Referenced) bit

Scan all pages examining R bit:

- if (R == 1)
set time of last use to current virtual time
- if (R == 0 and age > τ)
remove this page
- if (R == 0 and age ≤ τ)
remember the smallest time

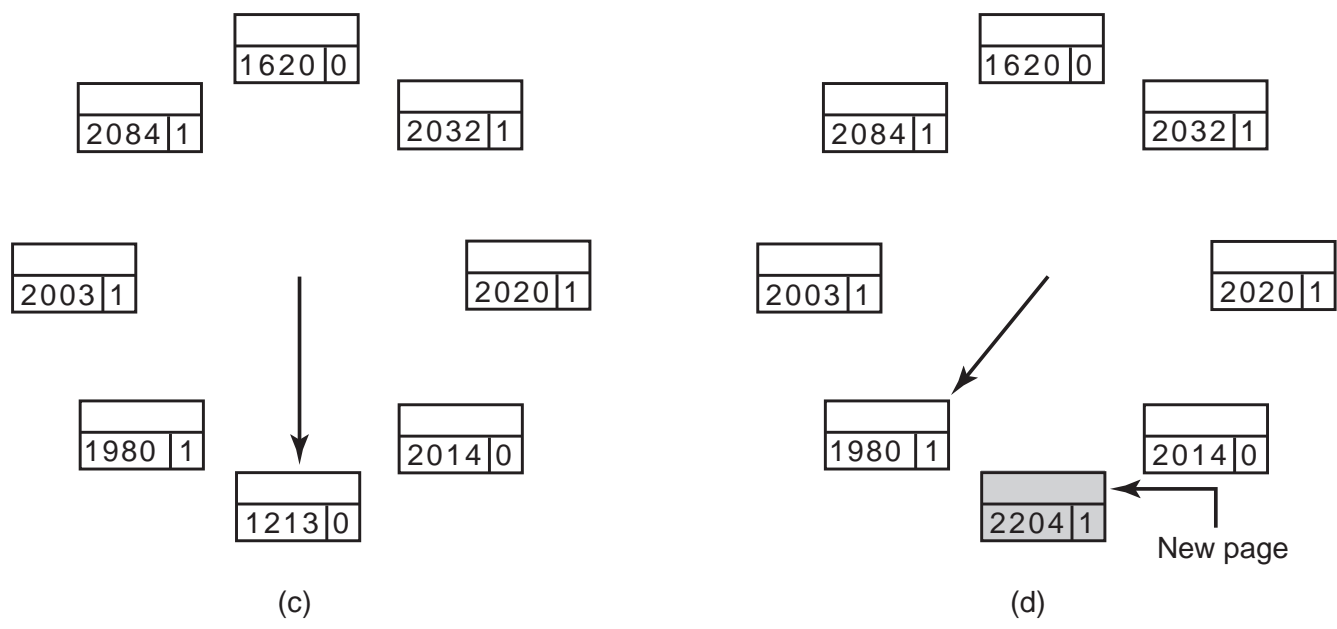
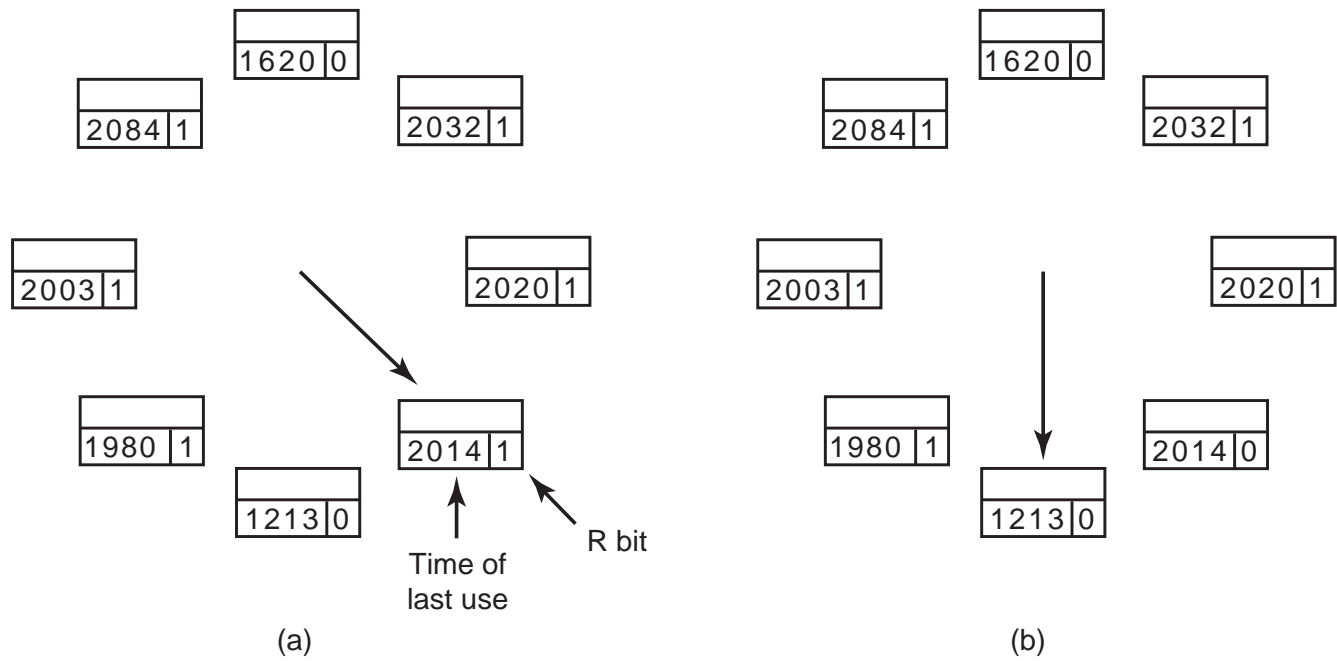
Page table

Algoritmo di rimpiazzamento WSClock

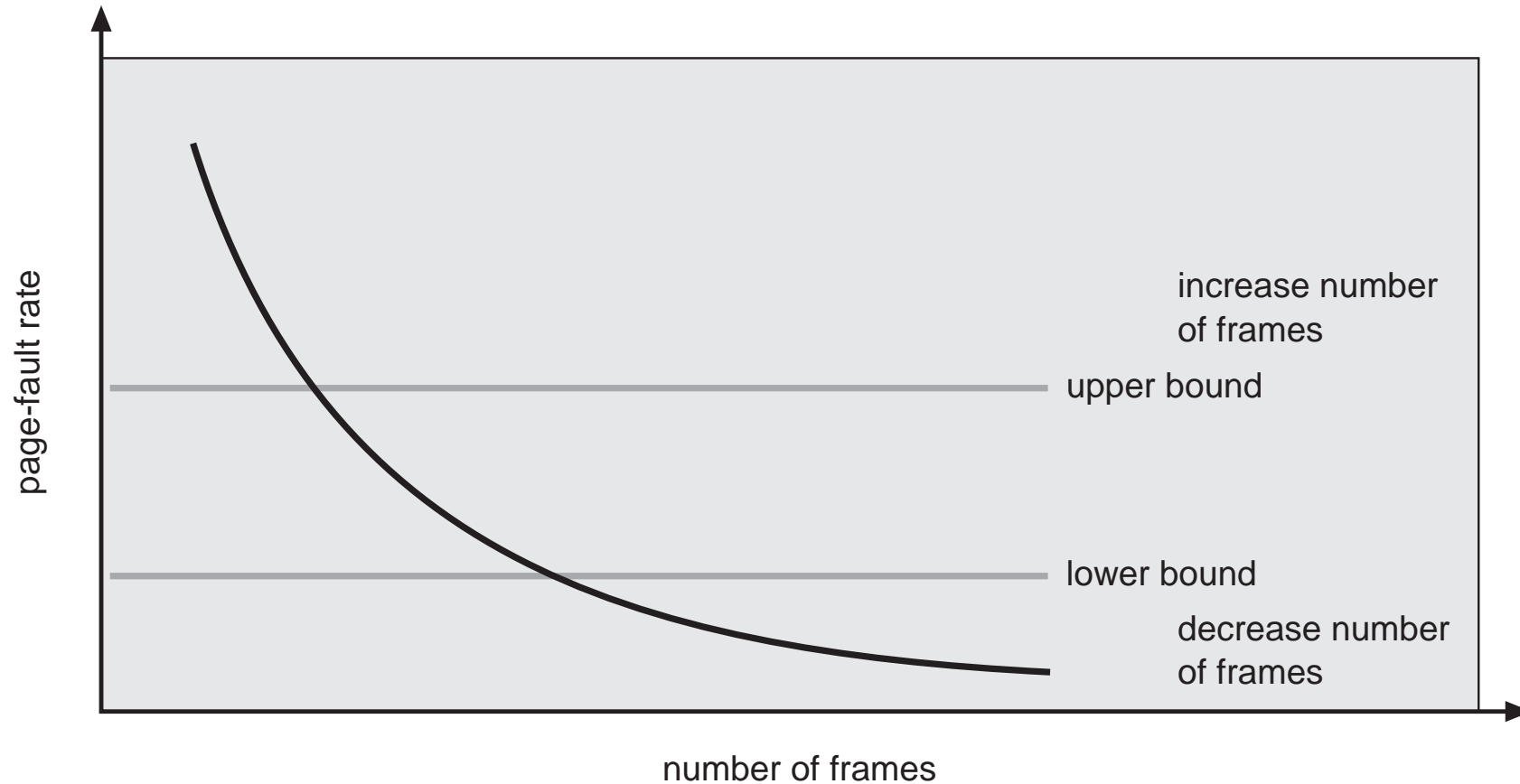
Variante del Clock che tiene conto del Working Set. Invece di contare i riferimenti, si tiene conto di una finestra temporale τ fissata (es. 100ms)

- si mantiene un contatore 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 ad ogni riferimento. La differenza tra questo registro e il contatore si chiama *età* della pagina.
- ad un page fault, si guarda prima la pagina indicata dal puntatore
 - se $R = 1$, si mette $R = 0$ 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.



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 propri 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 vari processi

- Allocazione libera: dare a qualsiasi processo i frame che 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)

- 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: 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 (cont.)

- 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
 - Lockare in memoria i frame contenenti buffer di I/O (*I/O interlock*) ⇒ delicato (un frame lockato potrebbe non essere più rilasciato)