Memory management, part 2: outline

- Page replacement algorithms
- Modeling PR algorithms
  - Working-set model and algorithms
- Virtual memory implementation issues

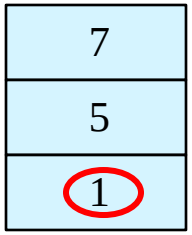
Page Replacement Algorithms

- Page fault forces choice
  - which page must be removed to make room for incoming page?
- Modified page must first be saved
  - unmodified just overwritten
- Better not to choose an often used page
  - will probably need to be brought back in soon

Optimal page replacement algorithm

- Remove the page that will be referenced latest
- Unrealistic: assumes we know future sequence of page references

Example



3 page frames

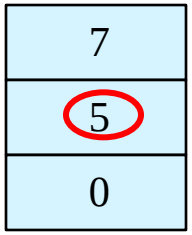
Assume that, starting from this configuration, the sequence of (virtual) page references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First page to remove is last to be used…

Optimal page replacement algorithm

- Remove the page that will be referenced latest
- Unrealistic: assumes we know future sequence of page references

Example



3 page frames

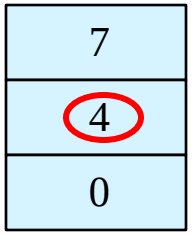
Assume that, starting from this configuration, the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First page to remove is last to be used…

Optimal page replacement algorithm

- Remove the page that will be referenced latest
- Unrealistic: assumes we know future sequence of page references

Example



3 page frames

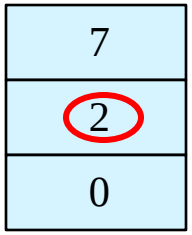
Assume that, starting from this configuration, the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First page to remove is last to be used…

Optimal page replacement algorithm

- Remove the page that will be referenced latest
- Unrealistic: assumes we know future sequence of page references

Example



3 page frames

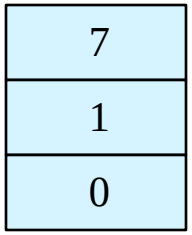
Assume that, starting from this configuration, the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First page to remove is last to be used…

Optimal page replacement algorithm

- Remove the page that will be referenced latest
- Unrealistic: assumes we know future sequence of page references

Example



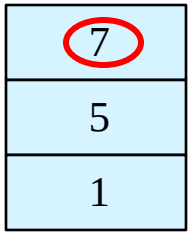
3 page frames

Assume that, starting from this configuration, the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

Altogether 4 page replacements. What if we used FIFO?

Optimal vs. FIFO

Example



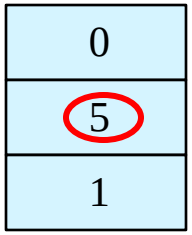
3 page frames

Assume that, starting from this configuration (**7,5,1**), the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First in first out…

Optimal vs. FIFO

Example



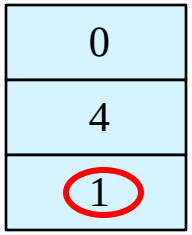
3 page frames

Assume that, starting from this configuration (**7,5,1**), the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First in first out…

Optimal vs. FIFO

Example



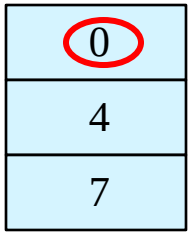
3 page frames

Assume that, starting from this configuration (**7,5,1**), the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First in first out…

Optimal vs. FIFO

Example



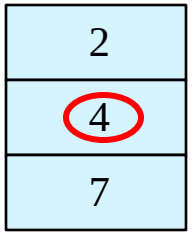
3 page frames

Assume that, starting from this configuration (**7,5,1**), the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First in first out…

Optimal vs. FIFO

Example



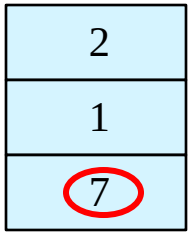
3 page frames

Assume that, starting from this configuration (**7,5,1**), the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First in first out…

Optimal vs. FIFO

Example



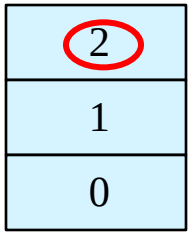
3 page frames

Assume that, starting from this configuration (**7,5,1**), the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First in first out…

Optimal vs. FIFO

Example



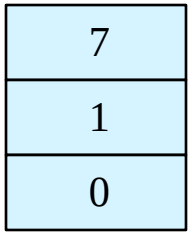
3 page frames

Assume that, starting from this configuration (**7,5,1**), the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

First in first out…

Optimal vs. FIFO

Example



3 page frames

Assume that, starting from this configuration (**7,5,1**), the sequence of references is: **0, 5, 4, 7, 0, 2, 1, 0, 7**

FIFO does 7 replacements, 3 more than optimal.

Page replacement: **NRU** - Not Recently Used

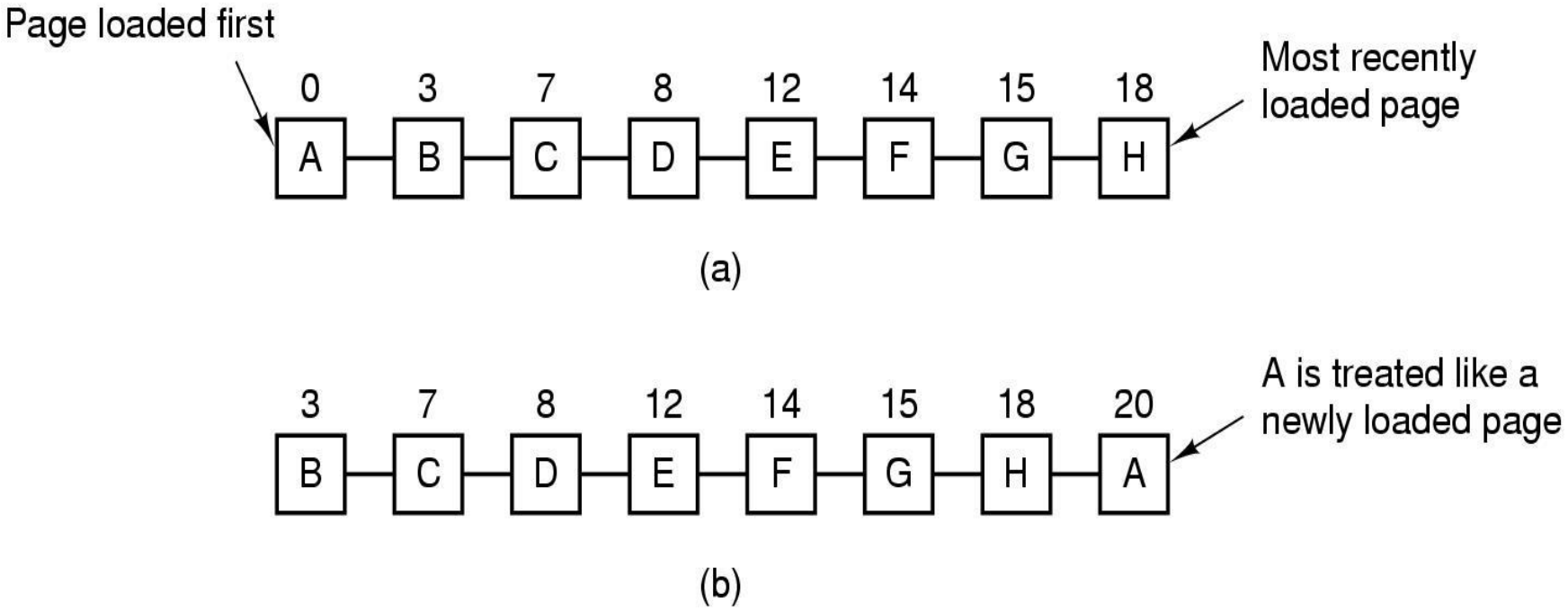
- There are 4 classes of pages, according to the *referenced* and *modified* bits
- *Select a page at random* from the *least-needed class*
- Easy scheme to implement
- Prefers a *frequently referenced* (unmodified) page on an *"old modified"* page
- How can a page belong to class B?

	Referenced=false	Referenced=true
Modified=false	A	C
Modified=true	B	D

2'nd chance FIFO replacement algorithm

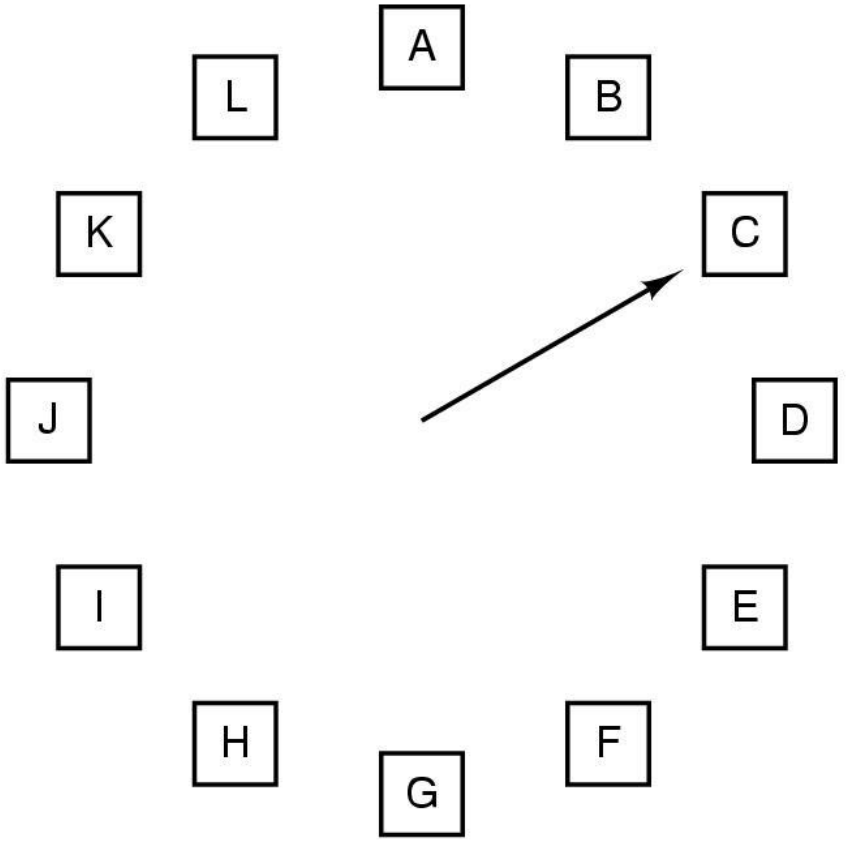
- May be implemented by using a queue
- FIFO: “oldest” page may be most referenced page
- An improvement: “second chance” FIFO
  - Inspect pages from oldest to newest
  - If page's referenced bit is on, give it a second chance:
    - Clear bit
    - Move to end of queue
  - Else
    - Remove page
- “Second chance” FIFO can be implemented more efficiently as a circular queue: the “clock algorithm”

Second Chance Page Replacement Algorithm



- Operation of second chance FIFO
  - pages sorted in FIFO order
  - Page list if fault occurs at time 20, *A* has *R* bit set (numbers above pages are times of insertion to list)
  - When A moves forward its R bit is cleared, timestamp updated

The Clock Page Replacement Algorithm



When a page fault occurs, the page the hand is pointing to is inspected. The action taken depends on the R bit:

R = 0: Evict the page

R = 1: Clear R and advance hand

LRU - Least Recently Used

- Most recently used pages have high probability of being referenced again
- Replace page used least recently
- Not easy to implement – total order must be maintained
- Possible hardware solutions
  - Use a large HW-manipulated counter, store counter value in page entry on each reference, select page with smallest value.
  - Use an nXn bit array (see next slide)
    - When page-frame k is referenced, set all bits of row k to 1 and all bits of column k to 0.
    - The row with lowest binary value is least recently used

LRU with bit tables

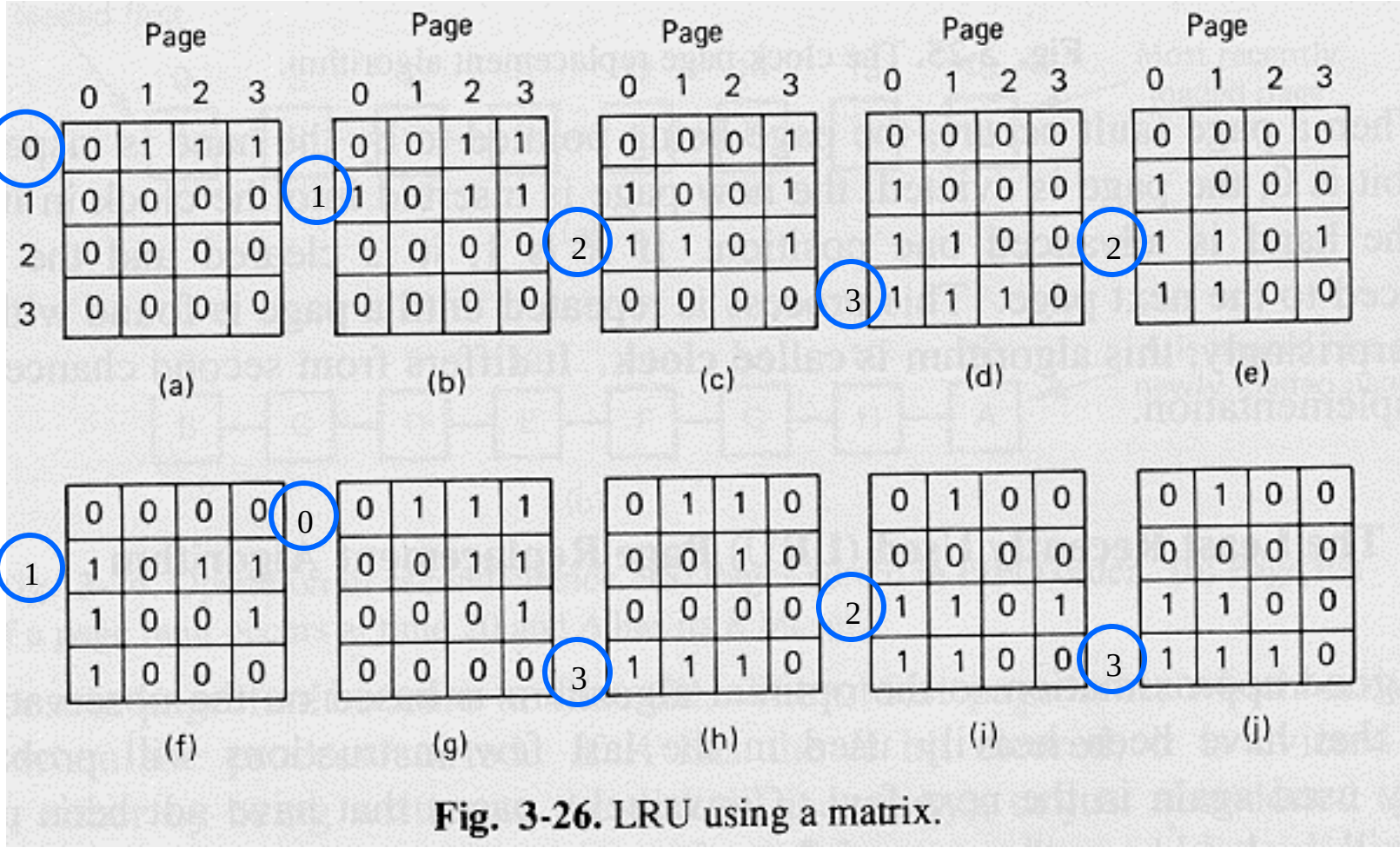


Fig. 3-26. LRU using a matrix.

Reference string is: 0,1,2,3,2,1,0,3,2,3

Why is this algorithm correct?

Claim 1

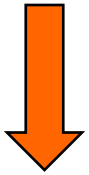
The diagonal is always composed of 0's.

Claim 2

Right after a page i is referenced, matrix row i has the maximum binary value (all other rows have at least one more 0 in addition to that in the i'th column)

Claim 3

For all distinct i, j, k, a reference to page k does not change the order between matrix lines i, j.



A new referenced page gets line with maximum value and does not change previous order, so a simple induction proof works.

NFU - Not Frequently Used
Approximating LRU in software

- In order to record frequently used pages add a *counter* field to each page frame – but don't update on each memory reference, update every **clock tick.**
- At each clock tick, add the R bit to the counters (and zero the bit)
- Select page with *lowest counter* for replacement
- problem: *remembers everything…*
- remedy (an *"aging"* algorithm):
  - *shift-right* the counter before adding the reference bit
  - add the reference bit *at the left* (Most Significant Bit)

NFU - the “aging” simulation version

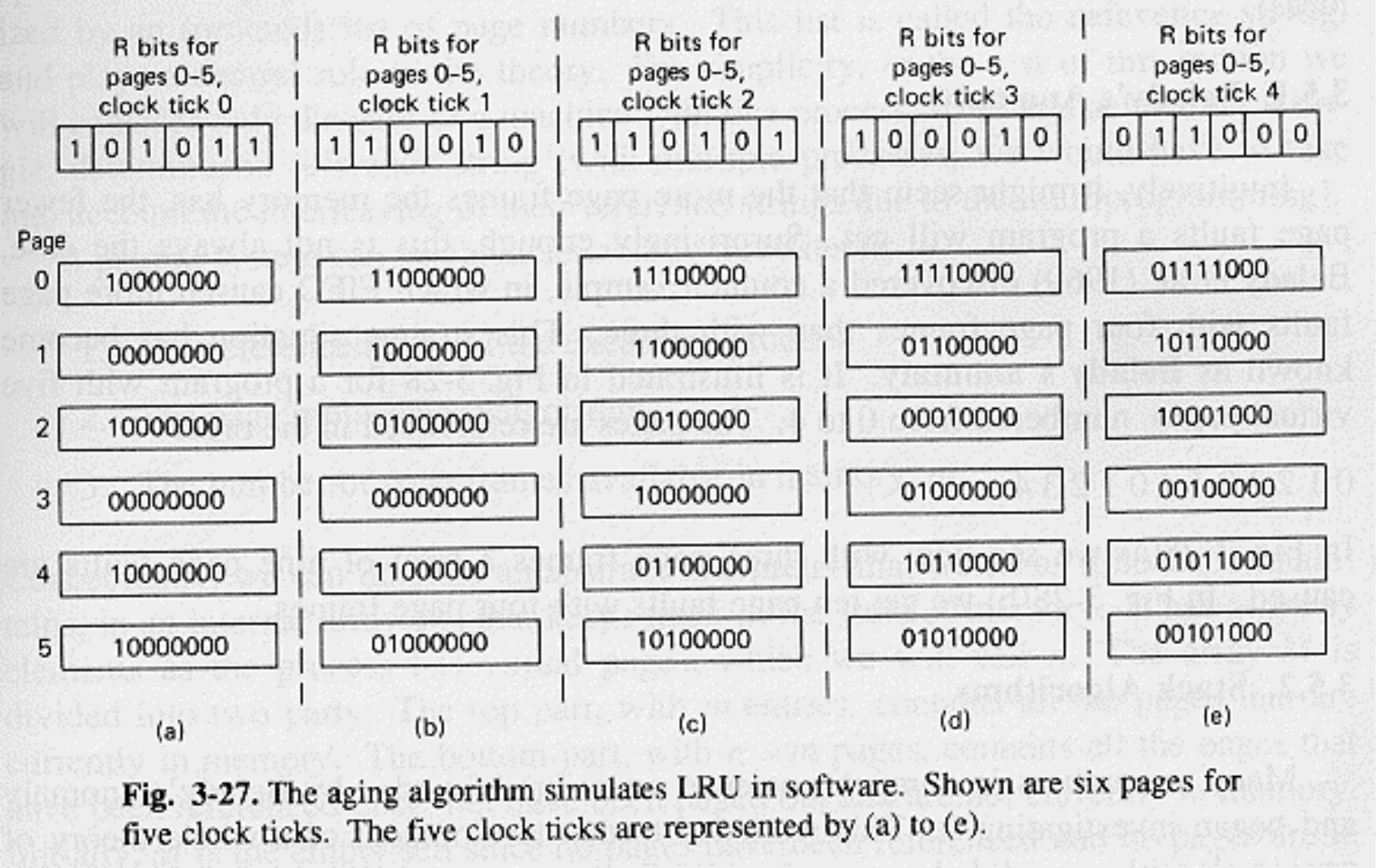


Fig. 3-27. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

Differences between LRU and NFU

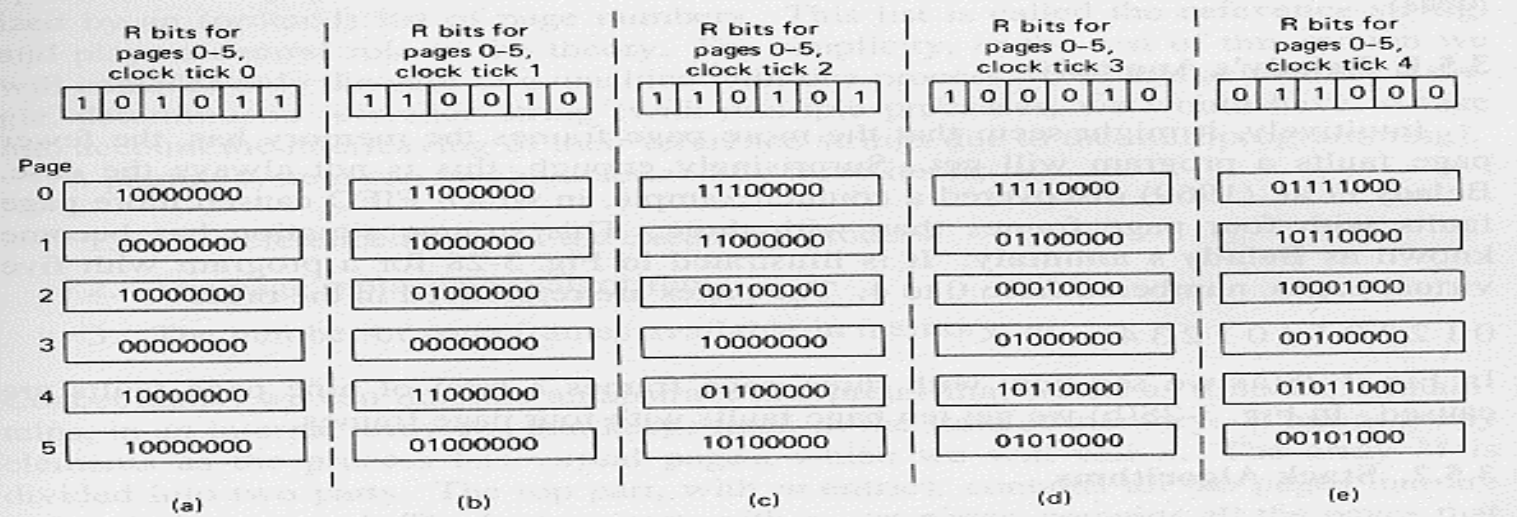


Fig. 3-27. The aging algorithm simulates LRU in software. Shown are six pages for five clock ticks. The five clock ticks are represented by (a) to (e).

- If two pages have the same number of zeroes before the first '1', whom should we select? (E.g., processes 3, 5 after (e))
- If two pages have 0 counters, whom should we select? (counter has too few bits…)
- Therefore NFU is only an approximation of LRU.

Memory management, part 2: outline

- Page replacement algorithms
- Modeling PR algorithms
  - Working-set model and algorithms
- Virtual memory implementation issues

Belady's anomaly

Belady's anomaly
The same algorithm may cause MORE page faults with MORE page frames!

Example: FIFO with reference string 012301401234

	0	1	2	3	0	1	4	0	1	2	3	4
Youngest frame	0	1	2	3	0	1	4	4	4	2	3	3
		0	1	2	3	0	1	1	1	4	2	2
Oldest frame			0	1	2	3	0	0	0	1	4	4
	P	P	P	P	P	P	P			P	P	

9 page faults

	0	1	2	3	0	1	4	0	1	2	3	4
Youngest frame	0	1	2	3	3	3	4	0	1	2	3	4
		0	1	2	2	2	3	4	0	1	2	3
			0	1	1	1	2	3	4	0	1	2
Oldest frame				0	0	0	1	2	3	4	0	1
	P	P	P	P			P	P	P	P	P	P

10 page faults!

Modeling page replacement algorithms

- Reference string – sequence of page accesses made by process
- number of virtual pages ***n***
- number of physical page frames ***m*** (we assume a single process)
- a page replacement algorithm can be represented by an array M of ***n*** rows

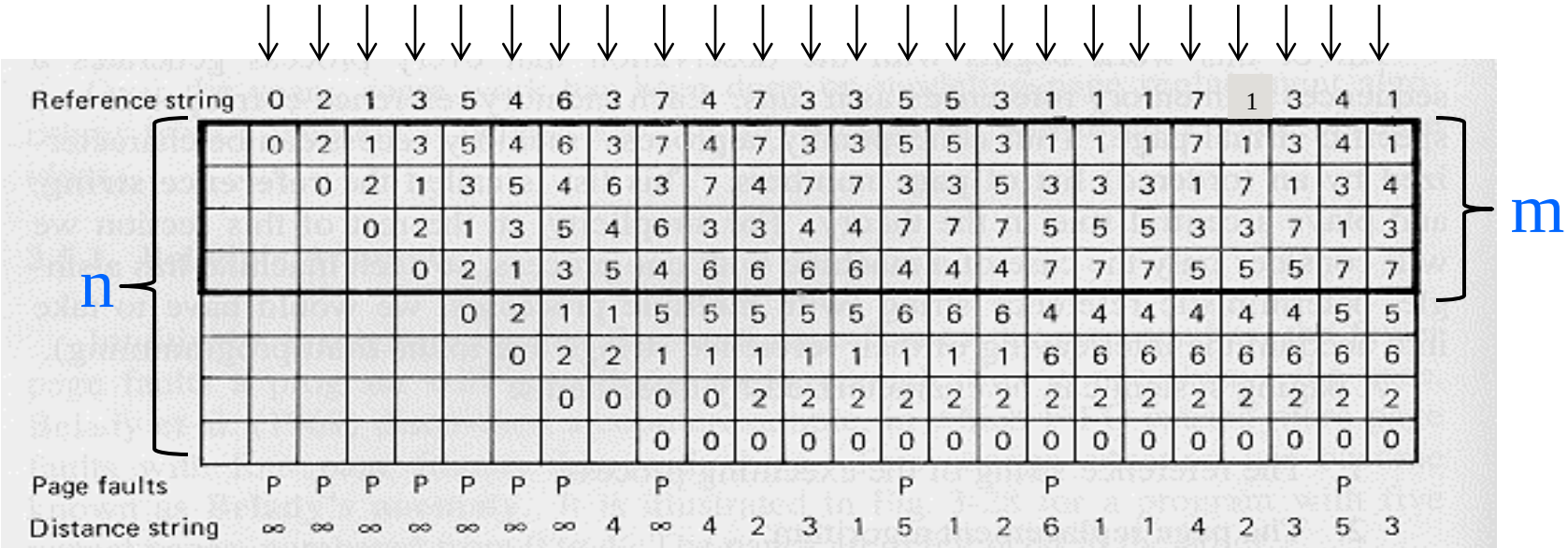


Fig. 3-29. The state of the memory array, *M*, after each item in the reference string is processed. The distance string will be discussed in the next section.

Stack Algorithms

M(m, r): for a fixed process P, the set of virtual pages in memory after the r'th reference of process P, with memory size m.

Definition: stack algorithms

A page replacement algorithm is a stack algorithm if, for every **P**, **m** and reference string **r**, it holds that $M(m, r) \subseteq M(m+1, r)$

- *Stack algorithms do not suffer from Belady's anomaly*
- Example: LRU, optimal replacement
- FIFO ***is not*** a stack algorithm
- Useful definition

  Distance string: distance of referenced page from top of stack

Computing page faults number

- C_i is the number of times that ***i*** is in the ***distance*** string
- The number of page faults when we have ***m*** page frames is

$$F_m = \sum_{k=m+1}^{n} C_k + C_\infty$$

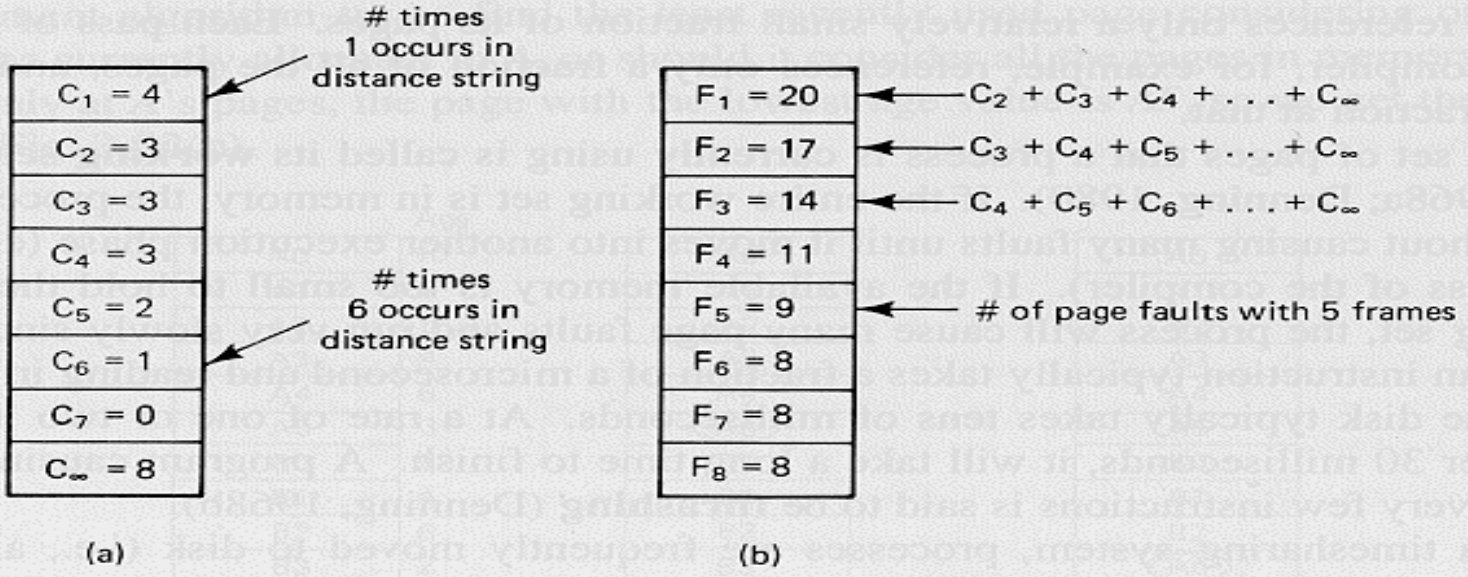
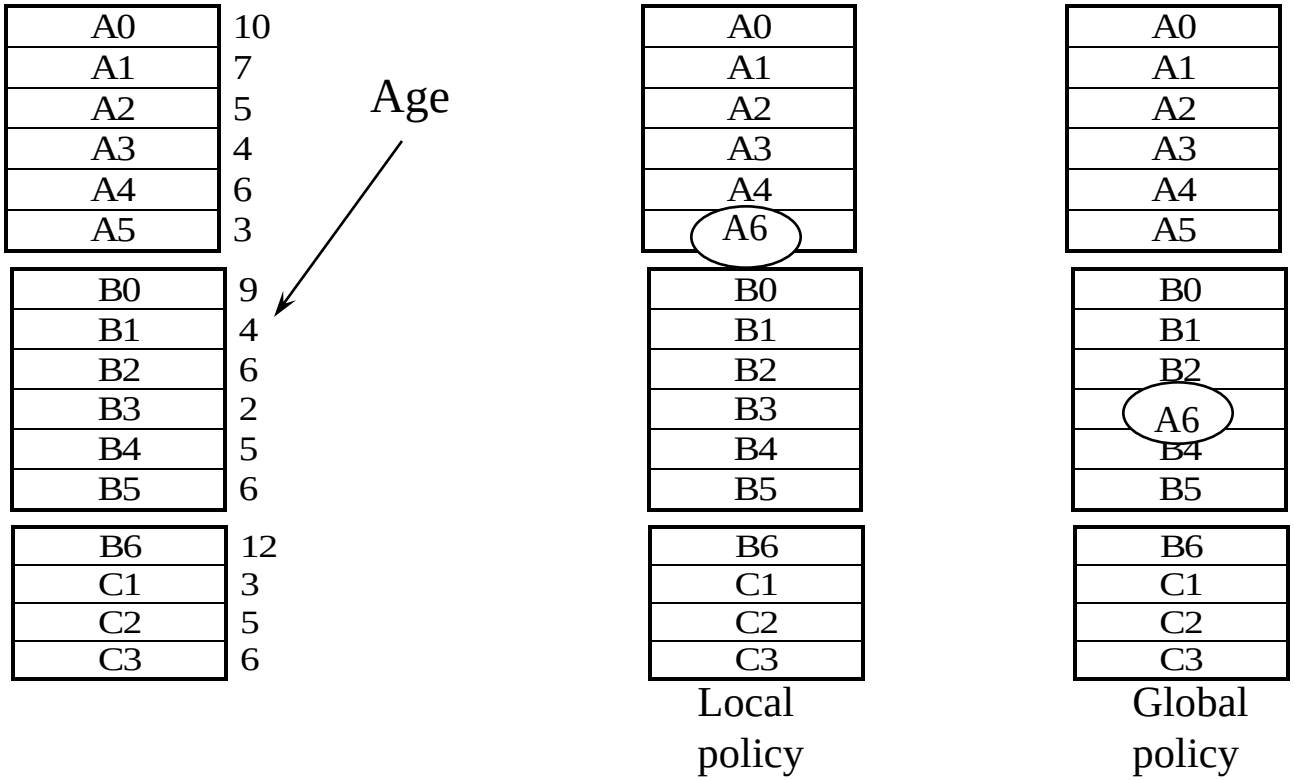


Fig. 3-31. Computation of the page fault rate from the distance string. (a) The *C* vector. (b) *F* vector.

Back to page replacement algorithms: Taking multiprogramming into account

- Local vs. global algorithms
- "Fair share" is not the best policy (*static* !!)
- Allocate according to process size – so, so…
- Must be a minimum for running a process...

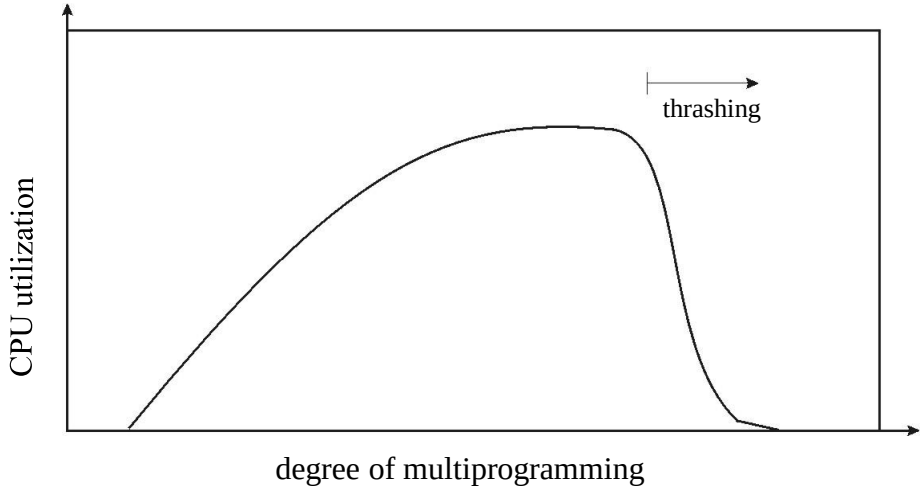


Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - Low CPU utilization
  - `Chain reaction’ may cause OS to think it needs to increase the degree of multiprogramming
  - More processes added to the system

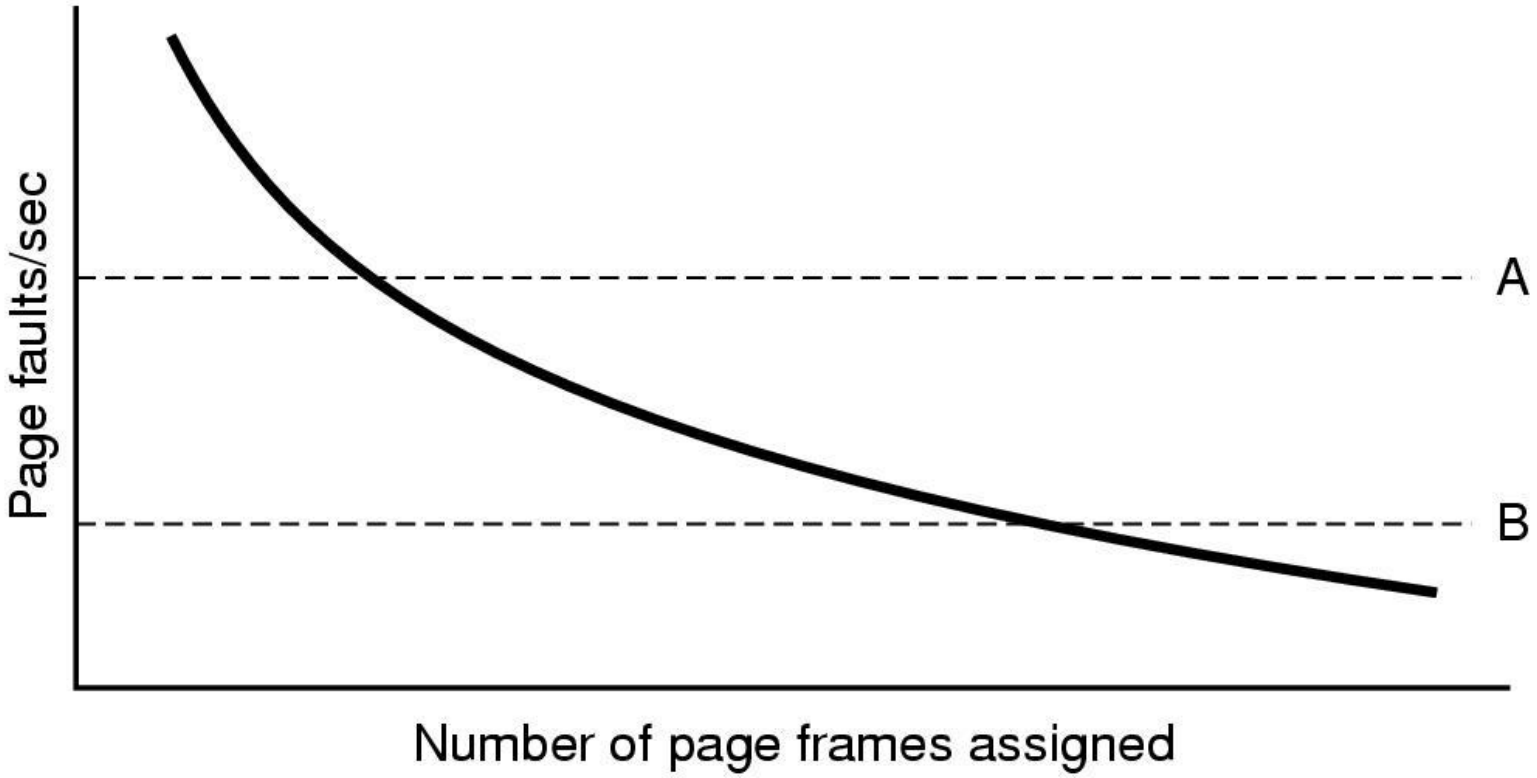
- Thrashing ≡ a process busy in swapping pages in and out

Thrashing Diagram



- Increasing multiprogramming level initially increases CPU utilization
- Beyond some point, thrashing sets in and multiprogramming level must be **decreased**.

Process thrashing as function of page frames



The impact of page-faults

- For a page-fault rate ***p***, memory access time of 100 nanosecs and page-fault service time of 25 millisecs, the effective access time is *(1-p) x 100 + p x 25,000,000*
- For ***p*=**0.001 the effective access time is still larger than 100 nanosecs by a factor of 250
- For a goal of only 10% degradation in access time we need ***p = 0.0 000 004***
- Difficult to know how much frames to allocate to processes – differ in size; structure; priority;…

Memory management, part 2: outline

- Page replacement algorithms
- Modeling PR algorithms
  - Working-set model and algorithms
- Virtual memory implementation issues

Working-Set Model: assumptions

- Locality of reference: during any phase of execution, a process is accessing a small number of pages – the process' *working set*.
- Process migrates from one locality to the other; localities may overlap
- If assigned physical memory is smaller than working set we get thrashing
- The working set of a process is the set of pages used by the Δ most recent memory references (for some predetermined Δ)
- *WS(t)* denotes the size of the working set at time t (for some fixed Δ)
- Optional: use *pre-paging* to bring a process' WS

Working set model

Page reference table

. . . 2 6 1 5 7 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 . . .

$WS(t_1) = \{1,2,5,6,7\}$ $WS(t_2) = \{3,4\}$

Figure 9.16 Working-set model

Working-Set Model

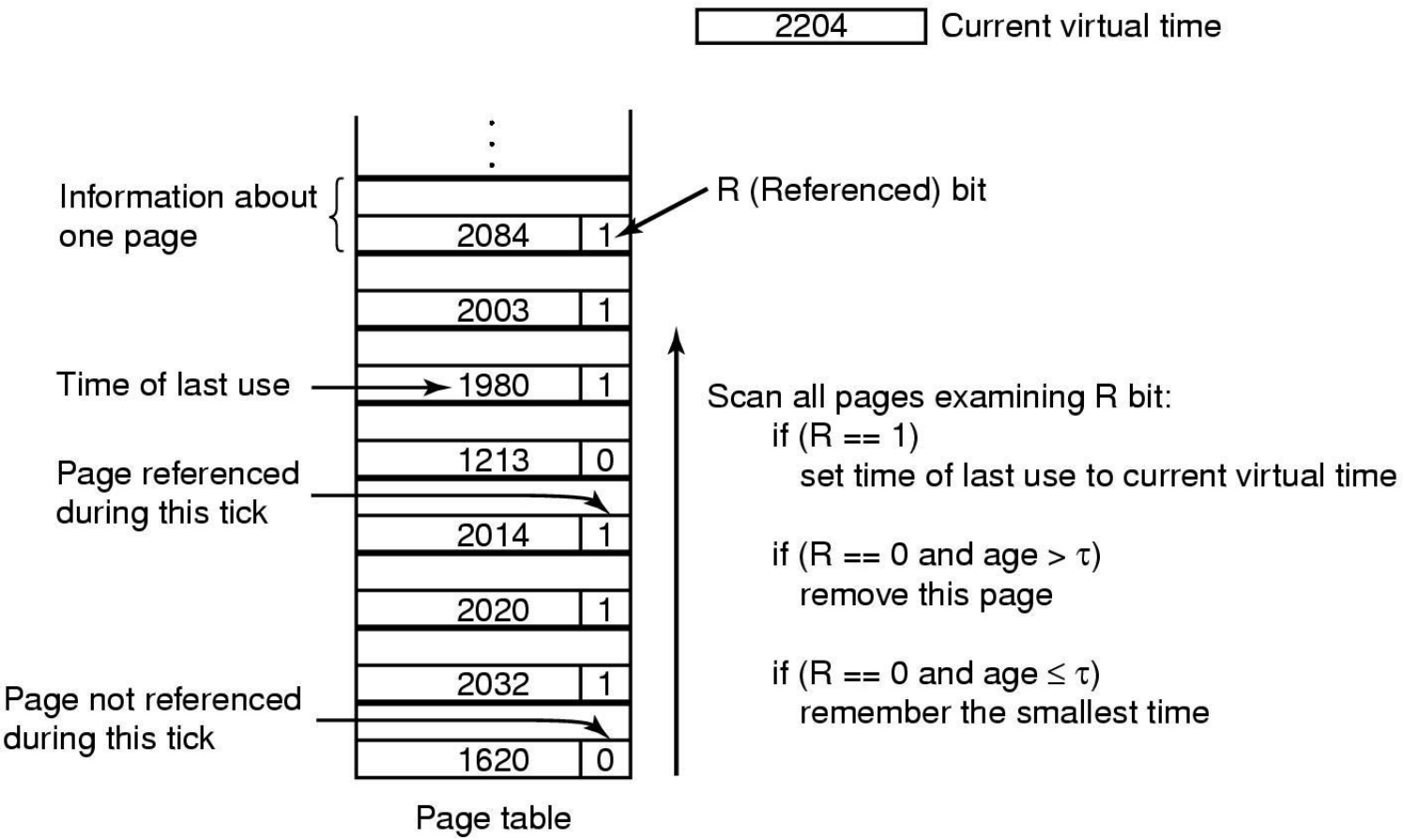
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  - If Δ too small - will not encompass entire locality.
  - If Δ too large - will encompass several localities.
  - If $\Delta = \infty \Rightarrow$ will encompass entire program.
- $D_t = \sum WS(t) \equiv$ total size of all working sets at time t
  - If $D_t > m \Rightarrow$ thrashing
  - Policy: if $D_t > m$, then suspend one of the processes.

How can we estimate WS(t) without doing an update at every memory reference?

Dynamic Set + Aging

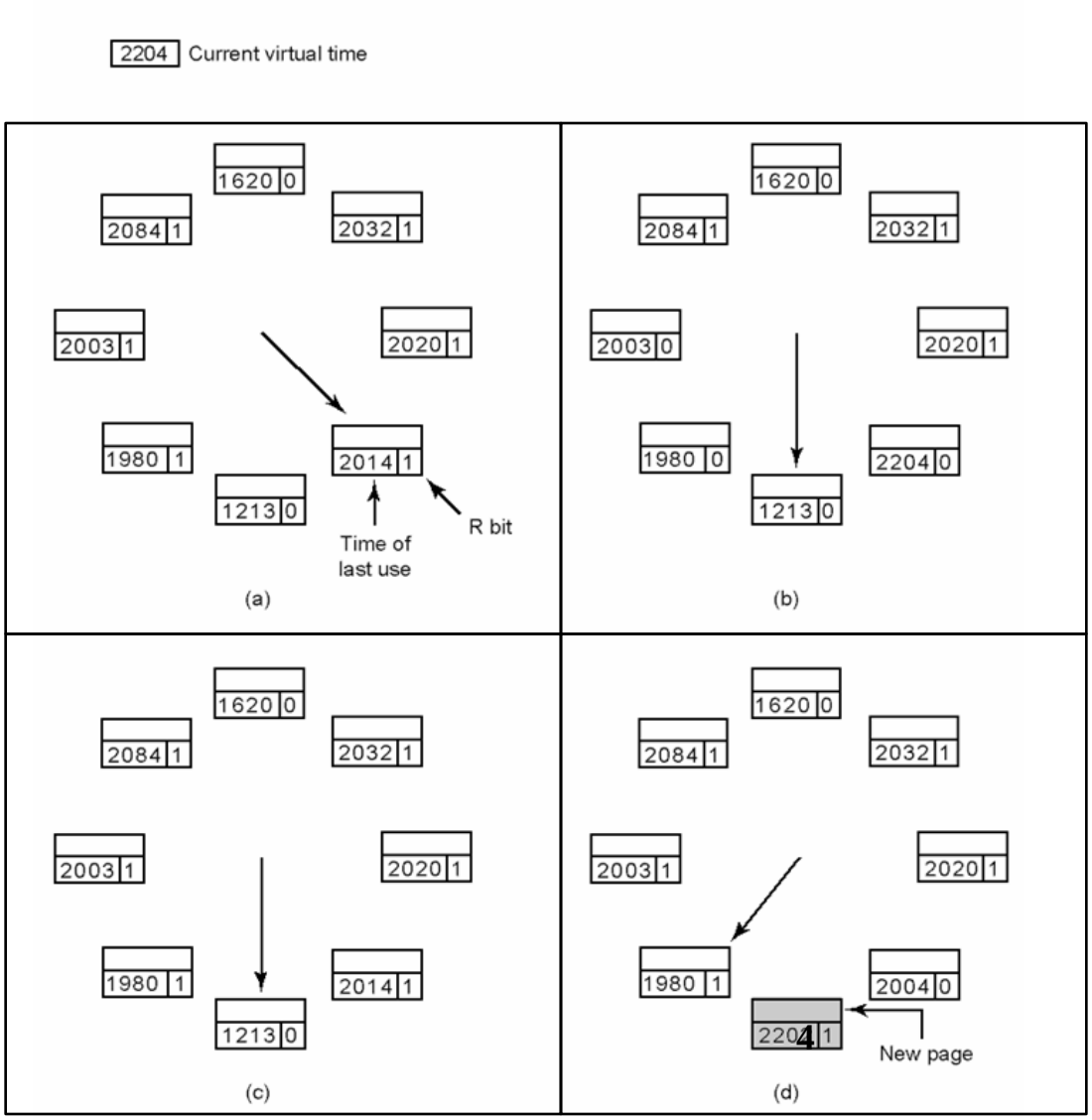
- The working-set window cannot be based on memory references - too expensive
- One way to enlarge the time gap between updates is to use some clock tick triggering
  - Reference bits are updated by the hardware
  - Virtual time maintained for each process (in PCB entry)
  - Every timer tick, update process virtual time and virtual time of referenced paging-table entries; then the R bit is cleared
- Page p's age is the diff between the process' virtual time and the time of p's last reference
- At PF time, the table is scanned and the entry with R=0 and the largest "age" is selected for eviction
- We use process virtual time (rather than global time) since it is more correlated to the process' working set

The Working Set Page Replacement Algorithm (2)



The working set algorithm

The WSClock Page Replacement Algorithm



Dynamic set - Clock Algorithm

- WSClock is in practice a global clock algorithm - for pages held by *all processes* in memory
- Circling the clock, the algorithm uses the reference bit and an additional data structure, *ref(frame),* is set to the current "virtual time" of the process
- WSClock: Use an additional condition that measures "elapsed (process) time" and compares it to τ
- Replace page when two conditions apply
  - reference bit is unset
  - T_p - *ref(frame)* $> \tau$

Dynamic set - WSClock Example

- 3 processes p_0, p_1 and p_2
- Current (virtual) times of the 3 processes are

  $Tp_0 = 50$; $Tp_1 = 70$; $Tp_2 = 90$
- WSClock: replace when $T_p - ref(frame) > \tau$
- The minimal distance ("window size") is **$\tau = 20$**
- The clock hand is currently pointing to page frame 4

page-frames:	0	1	2	3	***4***	5	6	7	8	9	10
ref. bit:	0	0	1	1	1	0	1	0	0	1	0
process ID:	0	1	0	1	2	1	0	0	1	2	2
last ref:	10	30	42	65	81	57	31	37	31	47	55
						13		13	39	>20	

Review of Page Replacement Algorithms

Algorithm	Comment
Optimal	Not implementable, but useful as a benchmark
NRU (Not Recently Used)	Very crude
FIFO (First-In, First-Out)	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU (Least Recently Used)	Excellent, but difficult to implement exactly
NFU (Not Frequently Used)	Fairly crude approximation to LRU
Aging	Efficient algorithm that approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm

Memory management, part 2: outline

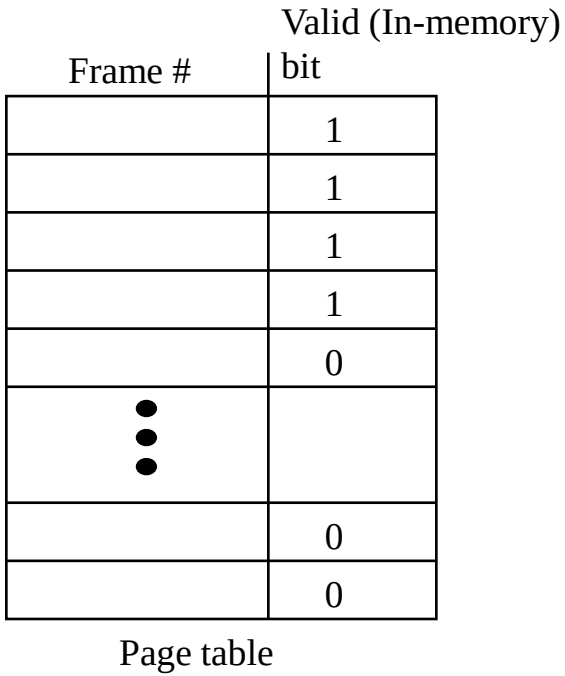
- Page replacement algorithms
- Modeling PR algorithms
  - Working-set model and algorithms
- Virtual memory implementation issues

Operating system involvement in paging

- Upon new process creation
  - Determine initial size of program + data
  - Create a page table: allocate space, initialize
  - Allocate space in swap area
  - Initialize swap area
  - Update info about page-table and swap area in PTE
- Upon process scheduling for execution
  - Reset MMU and flush TLB
  - Select new process' page table as current
  - Optionally bring some of the processes pages to memory
- Upon process exit
  - Release page table, pages and swap area
  - Don't release shared pages if still referenced

Valid (In-memory) Bit

- With each page table entry a bit is associated (1 ⇒ in-memory, 0 ⇒ not-in-memory)
- Initially, valid-invalid is set to 0 on all entries
- During address translation, if in-memory bit in page table entry is 0 ⇒ page fault



Page table

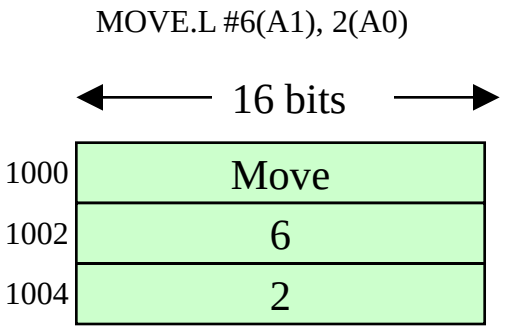
Page Fault

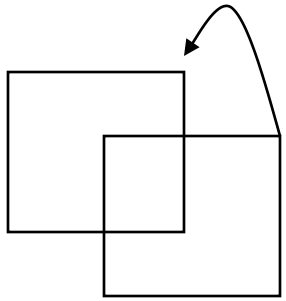
- If there is ever a reference to a page, first reference will trap to OS $\Rightarrow$ page fault
- OS looks at the valid bit to decide:
  - Invalid reference $\Rightarrow$ abort
  - Just not in memory
- Get empty frame (page replacement algorithm)
- Swap page into frame
- Update PTE, in-memory bit = 1
- Restart instruction:

Handling page faults

- The MMU sends a hardware interrupt, PC saved on stack
- Registers are saved, kernel is called
- Kernel discovers the virtual page that caused fault
- Kernel checks *valid* bit and verifies protection. If illegal access – send signal to process. Otherwise…
  - Check for a free page frame. If non available, *apply a page replacement algorithm*
  - If selected page frame is dirty – write to disk (context switch). Mark frame as busy
  - When page frame is clean, read from disk (process still suspended)
  - When page arrives, update page table, mark frame as normal state
  - Upon process re-scheduling, re-execute faulting instruction, reload registers, continue execution

Instruction backup





Block Move

- Above instruction makes 3 memory accesses. How does kernel know which caused the fault and where the instruction starts?
- If instruction does auto-increment/decrement, how do we know if it was already done?

Some machines provide this info in hardware registers,
Otherwise, OS sweats…

Memory access with page faults

P = probability of a page fault

MA = memory access time

PF = time to process page faults

EMA – Effective Memory Access =

(1-p) x MA + P x PF

where

PF = page-fault interrupt service time +

Read-in page time (maybe write-page too?) +

Restart process time

Demand Paging

- Bring a page into memory only when it is needed
  - Less memory needed
  - Faster response
  - More users

- Page is needed ⇒ reference it
  - Invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory

Locking Pages in Memory

- Virtual memory and I/O occasionally interact
- Process A issues read from I/O device into buffer
  - DMA transfer begins, process A blocks
  - process B starts executing
  - Process B causes a page fault
  - Page including buffer copied to by DMA may be chosen to be paged out
- Need to be able to lock page until I/O completes
  - Page exempted from being considered for replacement

Page Sharing

- Multiple processes may execute same code, we would like the text pages to be shared
- Paging out process A pages may cause many page faults for process B that shares them
- Looking up for “evicted” pages in all page tables is impossible
- *Solution*: maintain special data structures for shared pages
- Data pages may be shared also. E.g., when doing *fork()*, the *copy-on-write* mechanism.

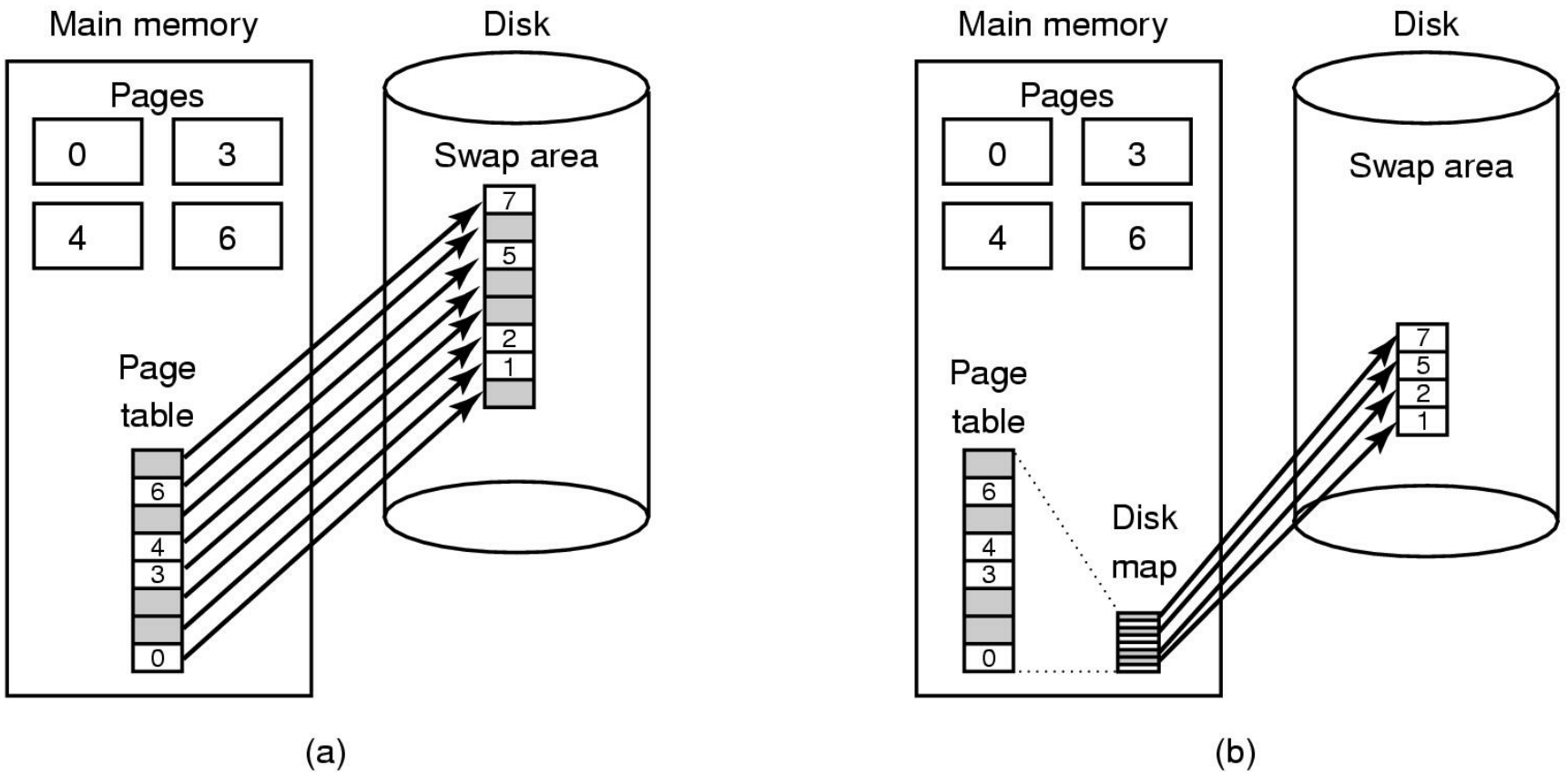
Cleaning Policy

- Pre-paging more efficient than pure demand-paging
- Often implemented by paging daemon
  - periodically inspects state of memory. Cleans pages, possibly removing them
- When too few frames are free
  - selects pages to evict using a replacement algorithm
- It can use same circular list (clock)
  - as regular page replacement algorithm but with a different pointer

Handling the backing store

- Need to store non-resident pages on disk swap area
- Alternative 1 (static) : allocate a fixed chunk of the swap area to process upon creation, keep offset in PTE
  - Problem: memory requirements may change dynamically
- Alternative 2: Reserve separate swap areas for text, data, stack, allow each to extend over multiple chunks
- Alternative 3 (dynamic) : allocate swap space to a page when it is swapped out, de-allocate when back in.
  - *Need to keep swap address for each page*

Backing Store



(a) Paging to static swap area

(b) Backing up pages dynamically

Virtual Memory - Advantages

- Programs may use much smaller physical memory than their maximum requirements (much code or data is unused…)
  - Higher level of multiprogramming
- Programs can use much larger (virtual) memory
  - simplifies programming and enable using powerful software
  - swapping time is smaller
- External fragmentation is eliminated
- More flexible memory protection (but less so than *segmentation*…)

Virtual Memory - Disadvantages

- Special hardware for address translation - some instructions may require 5-6 address translations!
- Difficulties in restarting instructions (chip/microcode complexity)
- Complexity of OS!
- Overhead - a Page-fault is an expensive operation in terms of both CPU and I/O overhead.
- Page-faults bad for real time
- Thrashing problem