CMPT 300

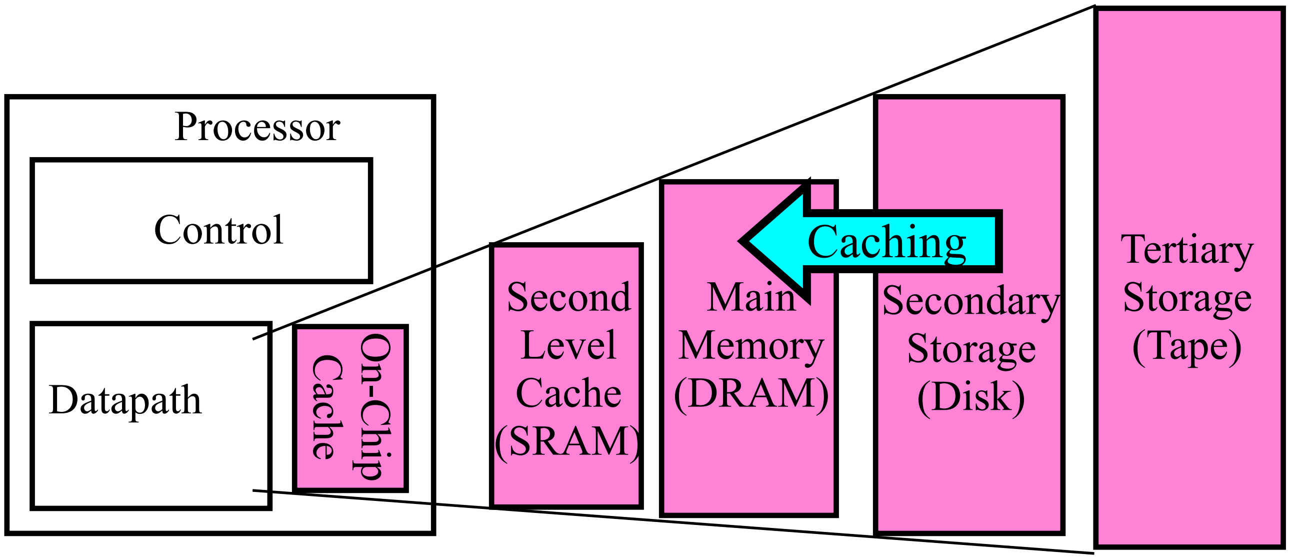
Introduction to Operating Systems

Page Replacement Algorithms

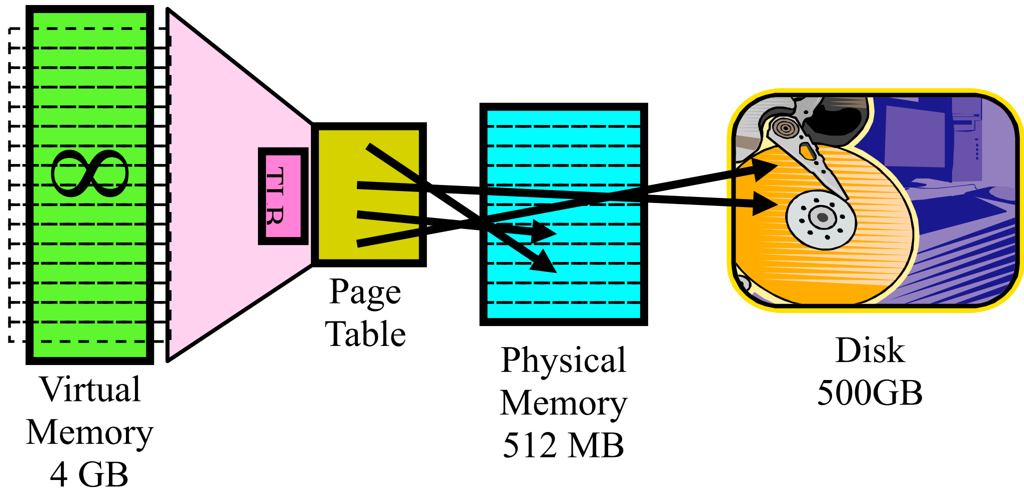
Demand Paging

- Modern programs require a lot of physical memory

- But they don't use all their memory all of the time
  - 90-10 rule: 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: **Swap** overflowed memory to disk



Illusion of Infinite Memory



- Disk is larger than physical memory ⇒
  - In-use virtual memory > physical memory
  - U (VM_proc) larger than physical memory
- Principle: Transparent Level of Indirection (page table)
  - Supports flexible placement of physical data (disk or mem)
  - Variable location of data transparent to user program

Demand Paging is Caching

- Since Demand Paging is Caching, must ask:
  - What is block size? **Page**
  - Search for a page in the cache when look for it?
    - Check TLB, Check PageTable
  - Replacement policy. No space in mem? (i.e. LRU)
    - This requires more explanation… (kinda LRU)
  - What happens on a miss
    - Refill from disk
  - What happens on a write? (write-through, write back)
    - Write-back. Need dirty bit!

Demand Paging Example

- Demand Paging like caching, can compute avg. access time! ("Effective Access Time")
  - EAT = Hit Rate x Hit Time + Miss Rate x Miss Time

- Example:
  - Memory = 200 ns; Average page-fault = 8 ms
  - p = Miss Prob., 1-p = Probably of hit
  - EAT = (1 – p) x 200ns + p x 8,000,000ns
    = 200ns + p x 7,999,800ns

- 1 miss out of 1,000 causes a page fault, then
  - EAT = 8.2 μs: 40x slowdown
- What if want slowdown by less than 10%?
  - 200ns x 1.1 < EAT ⇒ $p < 2.5 \times 10^{-6}$; 1 fault in 400000!

What Factors Lead to Misses?

- Compulsory Misses: First-access to new pages
  - How might we remove these misses?
    - Prefetching: loading them into memory before needed
- Capacity Misses: Not enough memory.
  - How to increase size?
    - Increase amount of DRAM (how cheap?) (not quick fix!)
    - Adjust budget for each process
- Policy Misses: Pages in memory, kicked out prematurely
  - Better replacement policy??

Replacement policy

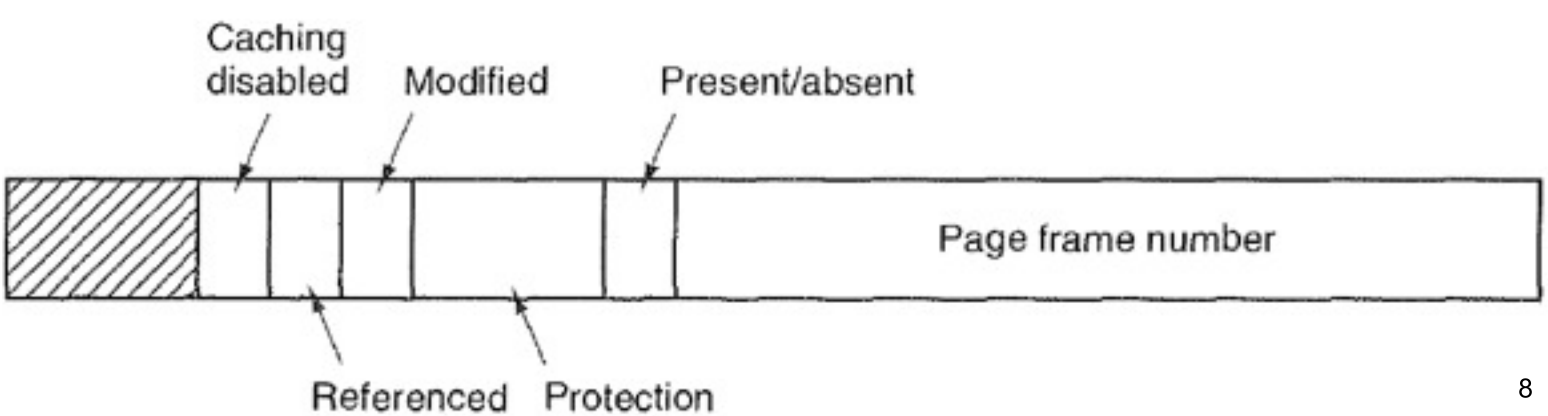
- Why do we care about Replacement?
  - The cost of being wrong is high: must go to disk
  - Must keep important pages in memory, not toss them out

- The simplest algorithm:
  - Random
  - Typical solution for TLB. Simple hardware
  - Unpredictable – makes it hard to make real-time guarantees

Recall: What is in a Page Table Entry (PTE)?

- *Page frame number.* Physical memory address of this page
- *Present/absent bit,* (*valid bit) :* 1 : the page is in memory. 0: the page is **NOT** currently in memory. Page fault to get page from disk.

- *Protection bits* accesses permitted on the page. Read, Write, and EXE.
- *Modified (M) bit,* also called *dirty bit,* Page is written. Writeback on swap out
- *Referenced (R) bit,* is set whenever a page is Read/Write.
  - M and R bits are very useful to page replacement algorithms
- *Caching disabled bit,* Device registers. Memory Mapped I/O



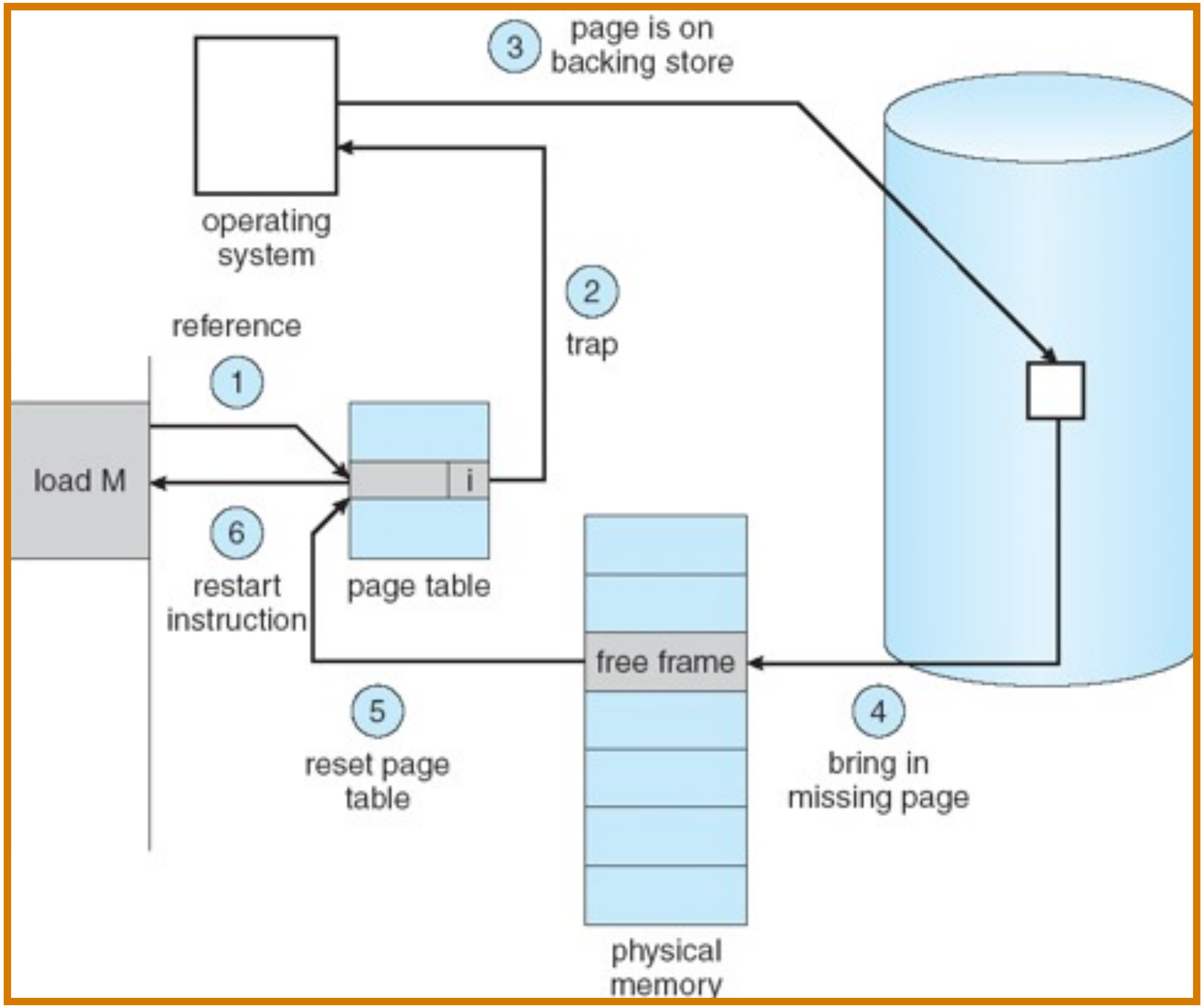
R & M bits

- Referenced (R) bit : Page recently used.
  - Set to 1 when page is Read/Write
  - OS defines a *clock period*. Every clock period, the R bit for each page is reset to 0.
    - R=0 → page is old (not used for some time)
    - R=1 → page is new (recently used)
- Modified (M) bit indicates if the page has been modified (written to)
  - The flag is reset when the page is saved to disk
  - When a page is removed from physical memory
    - M=1 → it will be saved to disk
    - M=0 → it will be abandoned and not saved to disk

Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Present ⇒ Page in memory, PTE points at physical page
  - Absent ⇒ Use PTE to find it on disk
- Absent PTE ⇒ Page fault
  - Choose Old page; Writeback, if necessary.
  - Change its PTE and any cached TLB to be invalid
  - Load new page into memory from disk
  - Update PTE, invalidate TLB for new entry; Restart thread
- TLB for new page will be loaded when thread continues!
- While pulling pages off disk, OS runs another process

Steps in Handling a Page Fault



Page Replacement Algorithms

- Optimal (OPT)
- Not recently used (NRU)
- First-In, First-Out (FIFO)
- Second chance (SC)
- Least recently used (LRU)
- Not frequently used (NFU)
- Aging algorithm
- Clock algorithm
- Working set
- WSClock

OPT page replacement

- Replace page that won't be used for the longest time
- Optimal, but infeasible in practice, since can't really know future…
- Good Baseline (can't do better)

Not recently used (NRU)

- Use the referenced and modified bits in the page table entries. 4 possibilities:
  1. NOT referenced, not modified
  2. NOT referenced, modified (not recently touched)
  3. Referenced, NOT modified
  4. Referenced, modified
- When a page fault occurs, prioritize in order 1--4

- If there is more than one page in the lowest group, randomly choose one page from the group.

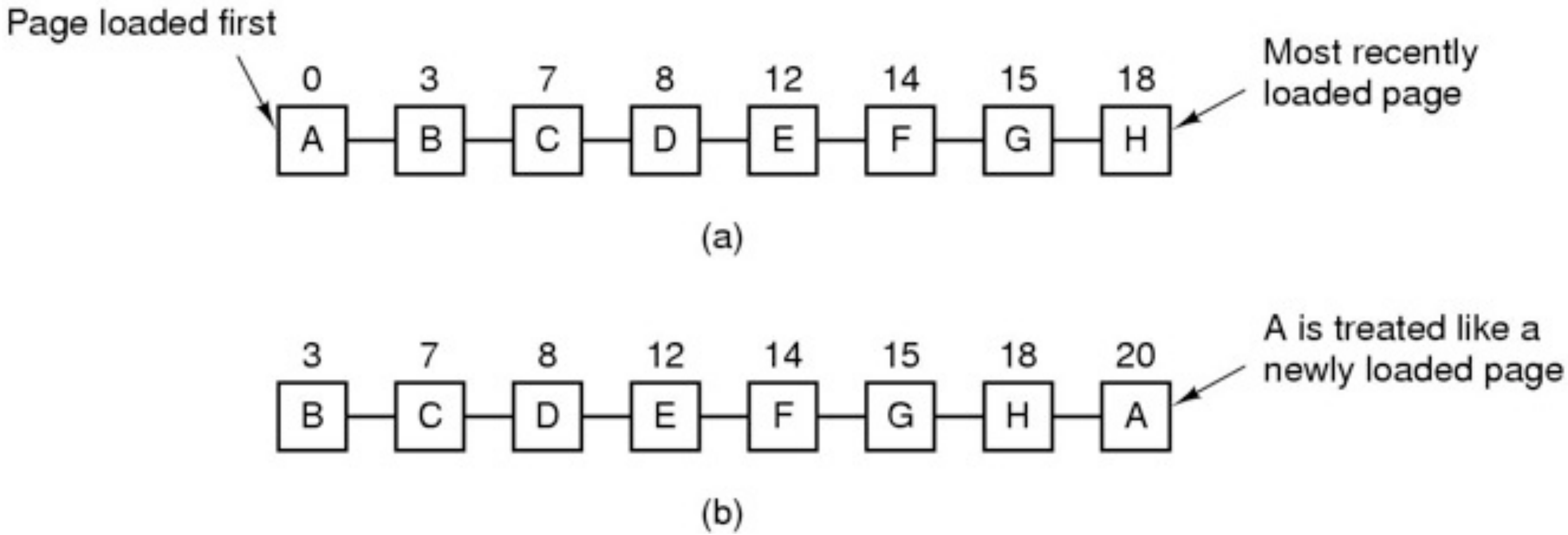
- Rationale: replace the page that has not been referenced or modified.

FIFO

- Throw out oldest page.
  - Be fair – let every page live in memory for same amount of time.
- Bad, because it may throw out heavily used pages instead of infrequently used
- **Second-chance algorithm** gives recently-used pages a second chance

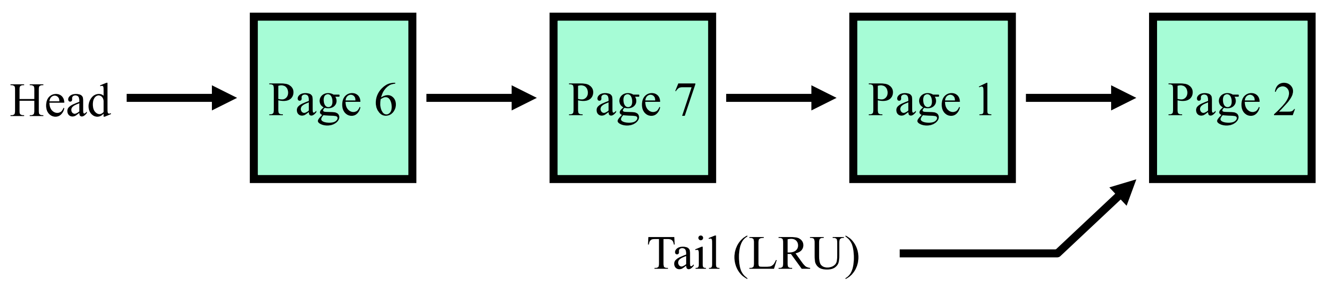
Second-Chance Algorithm

- Give recently-used pages a 2nd chance
  - If the OLDEST page has R=0, then choose it for replacement; if R=1, then move to end of LRU stack.



Least Recently Used (LRU)

- Replace page that hasn't been used for the longest time
- Programs have locality, if page not used for a while, unlikely to be used in the near future.
- Seems like LRU should be a good approximation to OPT.



- Expensive
  - List must be updated at every memory reference!

- In practice, people approximate LRU

Example: FIFO

- Consider a cache size of 3 page frames,
- Accesses : A B C A B D A D B C B
- FIFO Page replacement:
  - FIFO: 7 faults.
  - When referencing D, replacing A is bad choice, since need A again right away

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A					D				C	
2		B					A				
3			C						B		

Example: OPT

- Suppose we have the same reference stream:
  - A B C A B D A D B C B
- Consider OPT Page replacement:
  - 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future (C).
- What will LRU do?
  - Same decisions as OPT here, but won't always be true!

Ref: / Page:	A	B	C	A	B	D	A	D	B	C	B
1	A									C	
2		B									
3			C			D					

When will LRU perform badly?

- Consider Stream: A B C D A B C D A B C D
- LRU (same as FIFO here):
  - Every reference is a page fault!

Ref: / Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A			D			C			B		
2		B			A			D			C	
3			C			B			A			D

OPT Does much better

- But it's not implementable

Ref: Page:	A	B	C	D	A	B	C	D	A	B	C	D
1	A									B		
2		B					C					
3			C	D								

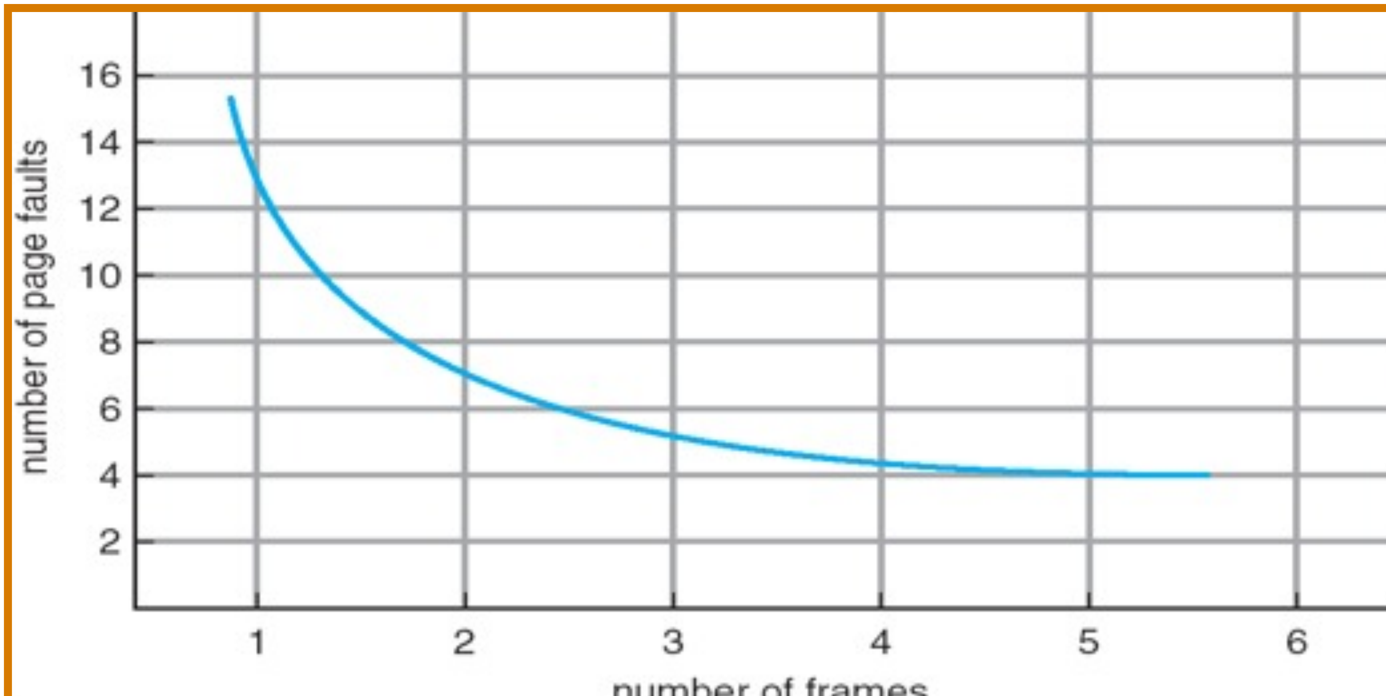
Exercise

- Consider a cache size of 3 page frames, and following reference stream of virtual pages:
  - 7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1
  - Run FIFO, OPT and LRU on this example.
- Answer:
  - FIFO: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/fifopagereplacement.htm
  - OPT: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/optimalpagereplacement.htm
  - LRU: http://cs.uttyler.edu/Faculty/Rainwater/COSC3355/Animations/lrupagereplacement.htm

Graph of Page Faults Versus The Number of Page Frames

- One desirable property: More memory the miss rate goes down
  - Does this always happen? Seems like it should, right?

- No: Belady's anomaly
  - Certain algorithms (FIFO) don't have this obvious property!



BeLady's anomaly

- Does adding memory reduce number of page faults?
  - Yes for LRU and OPT; not necessarily for FIFO! (Called Belady's anomaly)
- After adding memory:
  - With FIFO, contents can be completely different
  - In contrast, with LRU or OPT, contents of memory with X frames subset of contents with X+1 Pages.

Ref: / Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

9 page faults

Ref: / Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A						E				D	
2		B						A				E
3			C						B			
4				D						C		

10 page faults

Implementing LRU

- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality

- Replace an old page, not the oldest.
- Hardware techniques
  - 64-bit counter; n x n matrix
- Software techniques
  - Not recently used (NRU)
  - Aging Algorithm
  - Clock Algorithm

LRU in hardware

- Implementation #1:
  - 64 bit counter, C, incremented on every access
  - Each page also has a 64 bit counter
  - When page is referenced, C copied to counter.
  - Page with lowest counter is oldest.

LRU in hardware

- Implementation #2:
  - Given n page frames, let M be a n x n matrix of bits initially all 0.
  - Reference to page frame k occurs.
  - Set all bits in row k of M to 1.
  - Set all bits in column k of M to 0.
  - Row with lowest binary value is least recently used.

LRU in hardware: implementation #2 example

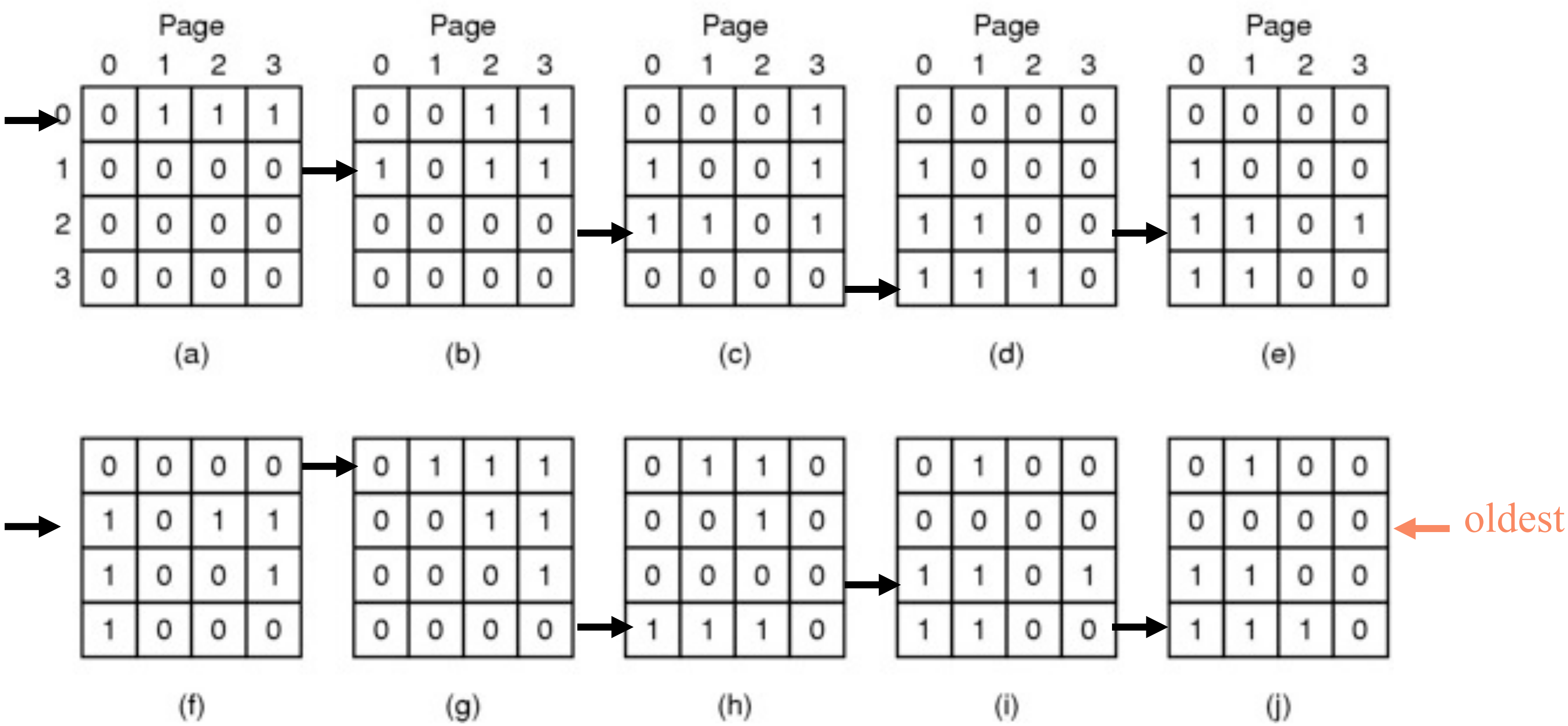


Figure 3-17. LRU using a matrix when pages are referenced in the order 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

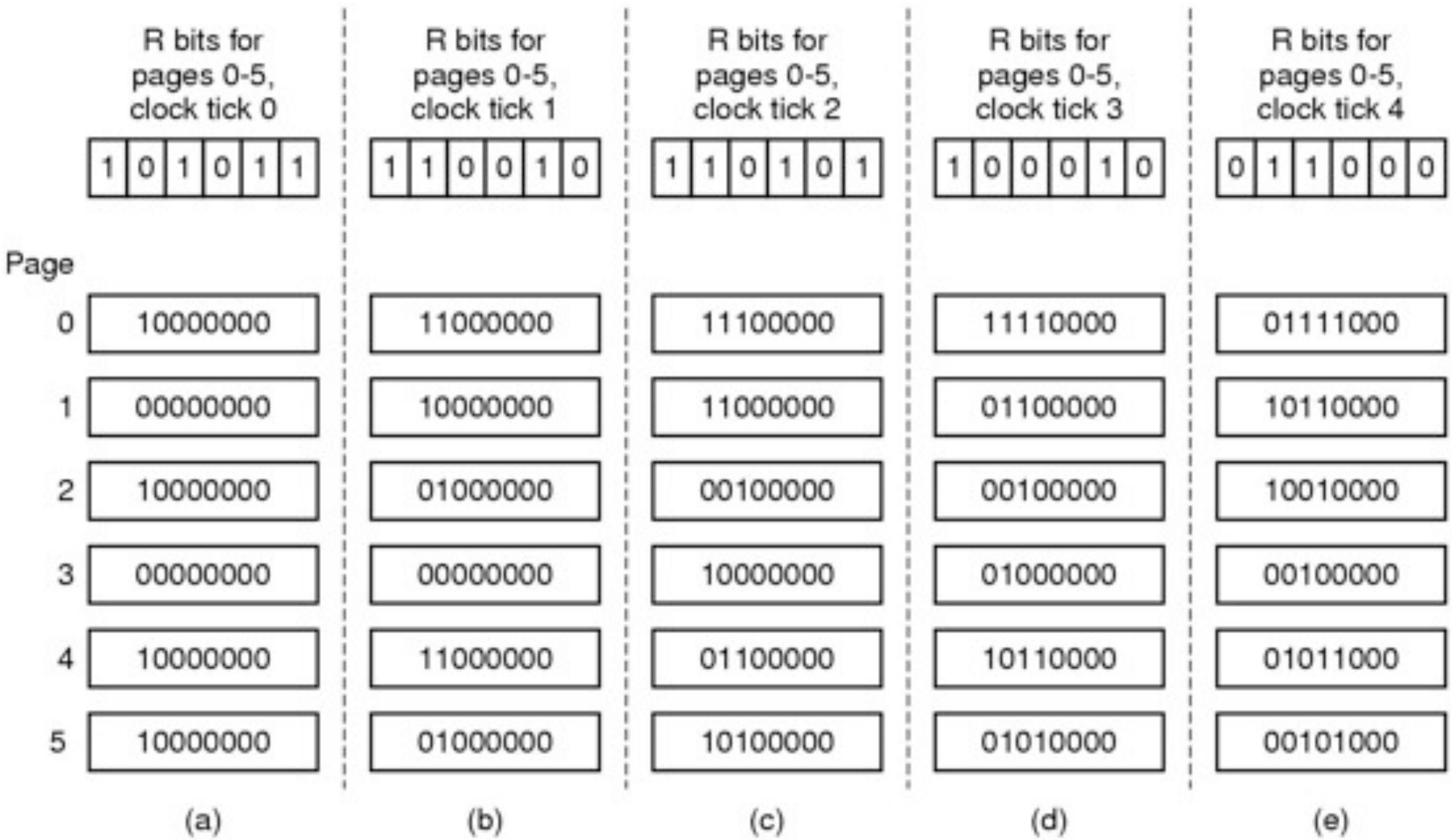
Not frequently used (NFU)

- A software counter associated with each page,
- At end of each clock period, the operating system scans all the pages in memory.
- For each page, the *R bit* (0 or 1), is added to the counter (arithmetic addition), which roughly keeps track of how often each page has been referenced. When a page fault occurs, the page with the smallest counter is chosen for replacement.
- Problem: It never forgets!
  - So pages that were frequently referenced (during initialization for example) but are no longer needed appear to be FU.

Aging algorithm

- Idea: Gradually forget the past
  - A k-bit software counter is associated with each page, the counter is initialized to 0
  - Shift all counters to right 1 bit before R bit is added in.
  - Then R bit is added to MSb (Most Significant (leftmost) bit)
  - Page with lowest counter value is chosen for removal.

Aging algorithm example



- Shown are six pages for five clock periods. The five clock periods are represented by (a) to (e).

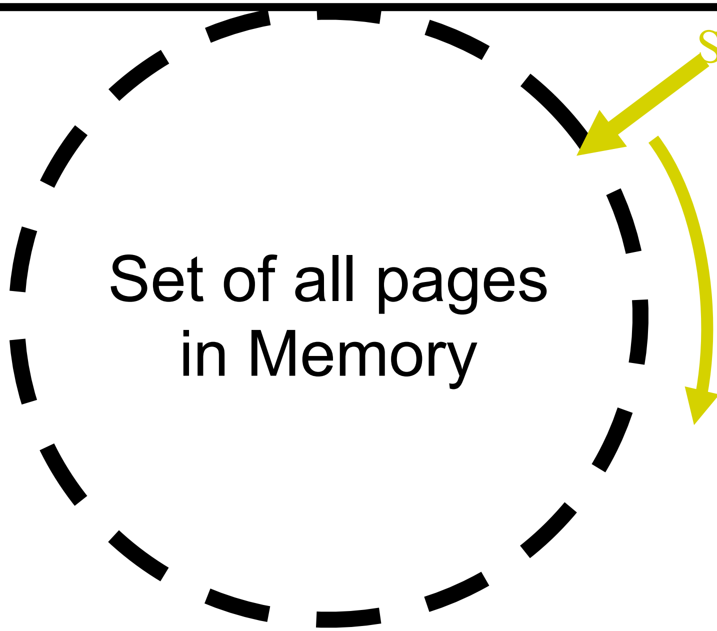
Aging vs. LRU

- Aging has a finite history of memory
  - Consider aging with an 8-bit counter with value 0. It cannot distinguish between a page referenced 9 clock periods ago, and another referenced 1000 block periods ago.
  - If the counter has infinitely many bits, then it implements LRU exactly.
- 8 bits generally enough
  - If clock period is 20ms, a history of 160ms is perhaps adequate

Clock Algorithm

- A variant of second-chance algorithm
- Recall "R" (reference) bit in PTE:
  - Clear it at page-fault events
- Arrange physical page frames in a circle with single clock hand. On each page fault:
  - Advance clock hand (not real-time)
  - Check R bit:
    - R=1→used recently; clear and leave alone
    - R=0→selected candidate for replacement
- Will always find a page or loop forever?
  - Even if all R bits set, will eventually loop around ⇒ FIFO

Clock Algorithm



Single Clock Hand:

Advances only on page fault!
Check for pages not used recently
Mark pages as not used recently

- What if hand moving slowly?
  - Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Lots of page faults and/or lots of reference bits set
- Animation: http://gaia.ecs.csus.edu/~zhangd/oscal/ClockFiles/Clock.htm (usrname/passwd: CSC139/csus.os.prin)
- Uncheck "use modified bit" button. Note that it uses "U" instead of "R" for the reference bit.

N^{th} Chance version of Clock Algorithm

- N^{th} chance algorithm: Give page N chances
  - OS keeps counter per page: # sweeps
  - On page fault, OS checks R bit:
    - R=1⇒clear R bit and also set counter to N
    - R=0⇒decrement count; if count=0, replace page
  - Means that clock hand has to sweep by N times

- How do we pick N?
  - **Large N**? Better approx to LRU
    - N ~ 1K, good approximation
  - Small N? More efficient
    - Otherwise might have to look a long way to find free page

- What about dirty pages?
  - Overhead to replace a dirty page, give more chance?
    - Clean pages, use N=1; Dirty pages, use N=2

Allocation of Page Frames

- Vend physical frame among different procs?
  - Does every process get the same fraction?
  - Should we completely swap some processes?

- Each process needs *min. guarantee*
  - Want to make sure that all processes that are loaded into memory can make forward progress

- Example: IBM 370: 6 pages to handle SS MOVE instruction:
  - instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*

Possible Replacement Scopes:

- Possible Replacement Scopes:
- Global replacement – process selects replacement frame from entire set; one process can take a frame from another
  - Achieve effective utilization.

- Local replacement – each process selects from only its own set of allocated frames
  - Achieve memory isolation among processes

Fixed/Priority Allocation

- Equal allocation (Fixed Scheme):
  - Every process gets same amount of memory
  - Example: 100 frames, 5 processes$\Rightarrow$process gets 20 frames
- Proportional allocation (Fixed Scheme)
  - Allocate according to the size of process
  - Computation proceeds as follows:

s_i = size of process p_i and $S = \Sigma s_i$

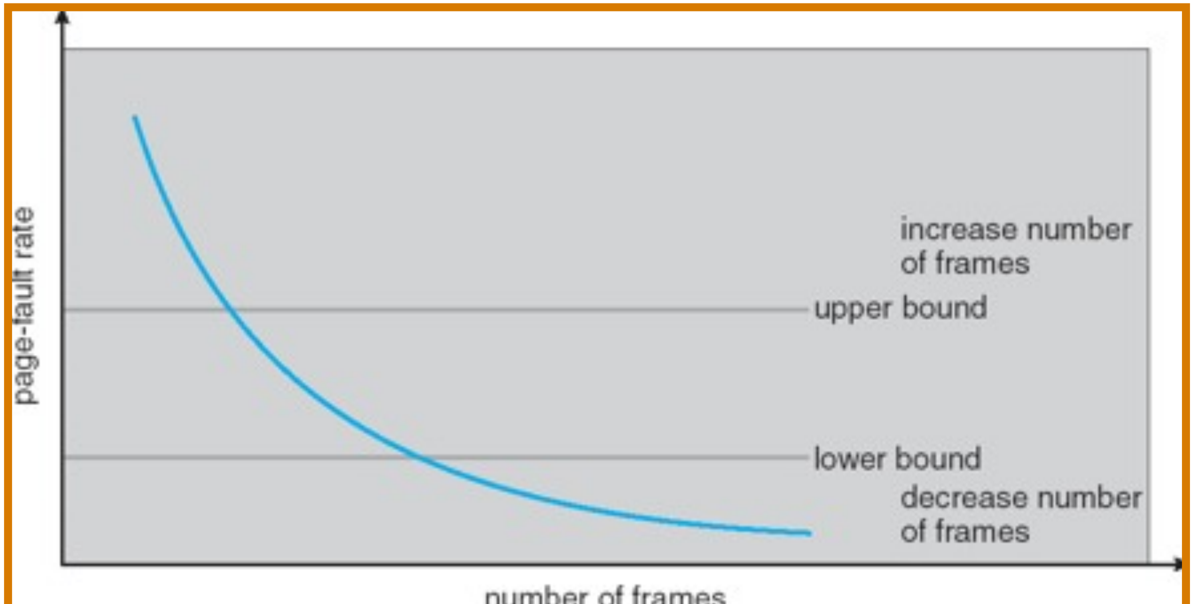
m = total number of frames

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

- Priority Allocation:
  - Proportional scheme using priorities rather than size
    - Same type of computation as previous scheme
  - Possible behavior: Select lower priority process.

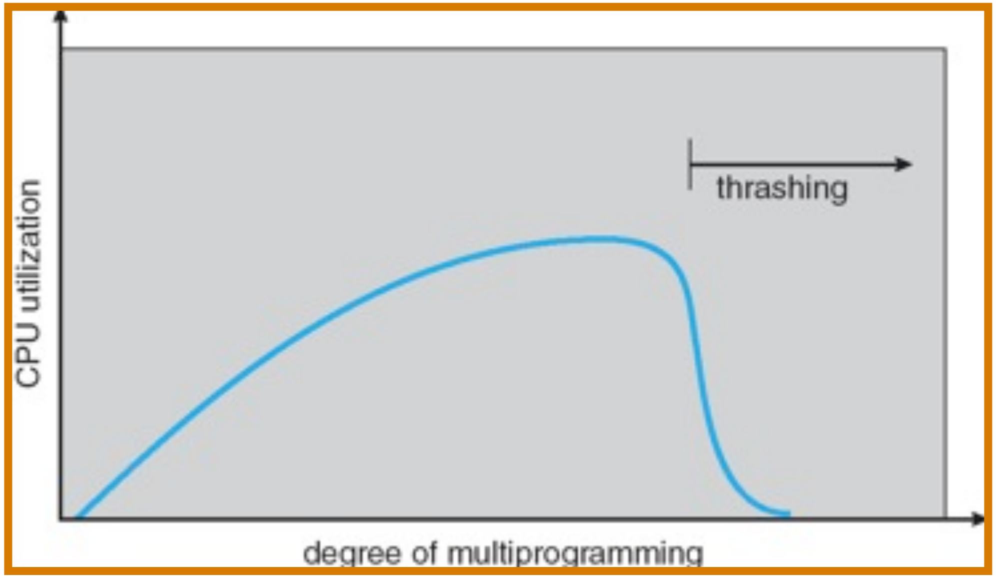
Page-Fault Frequency Allocation

- Can we reduce Capacity misses by dynamically changing the number of pages/application?



- Establish "acceptable" page-fault rate
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Question: What if we just don't have enough memory?

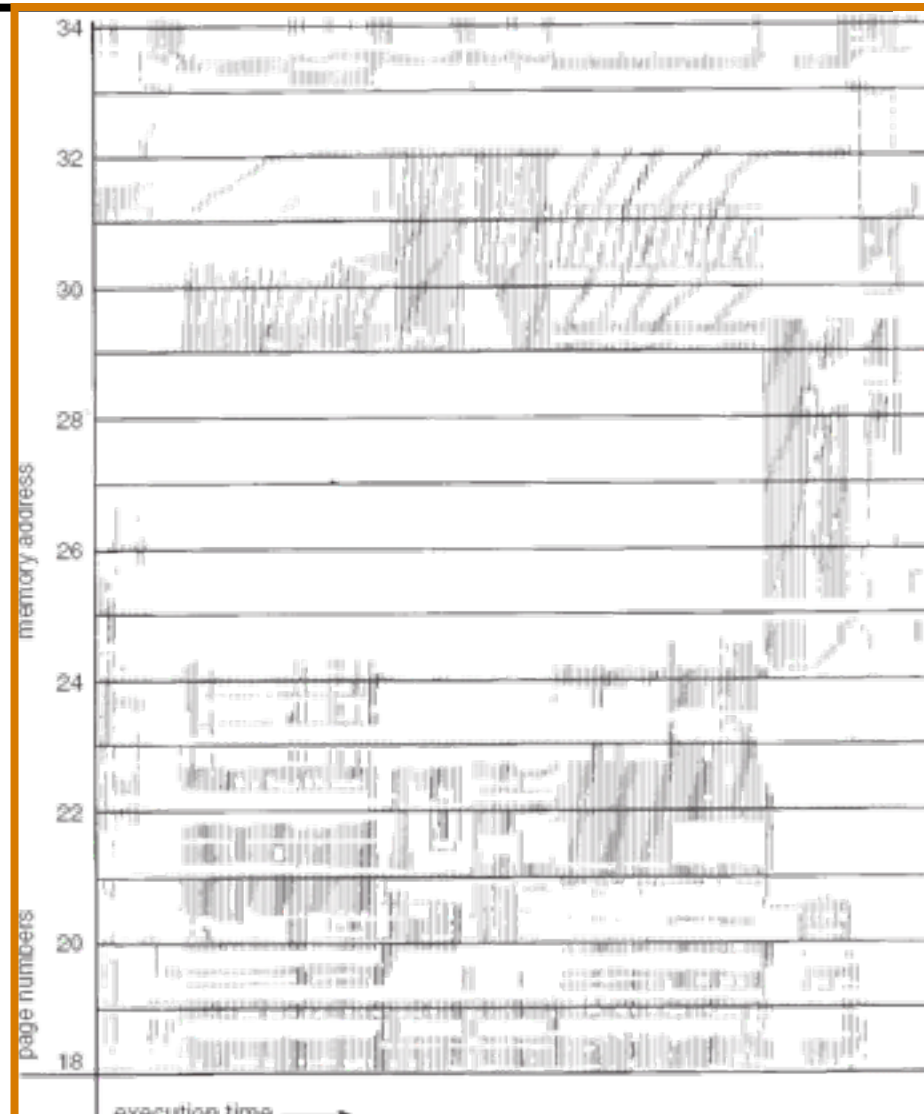
Thrashing



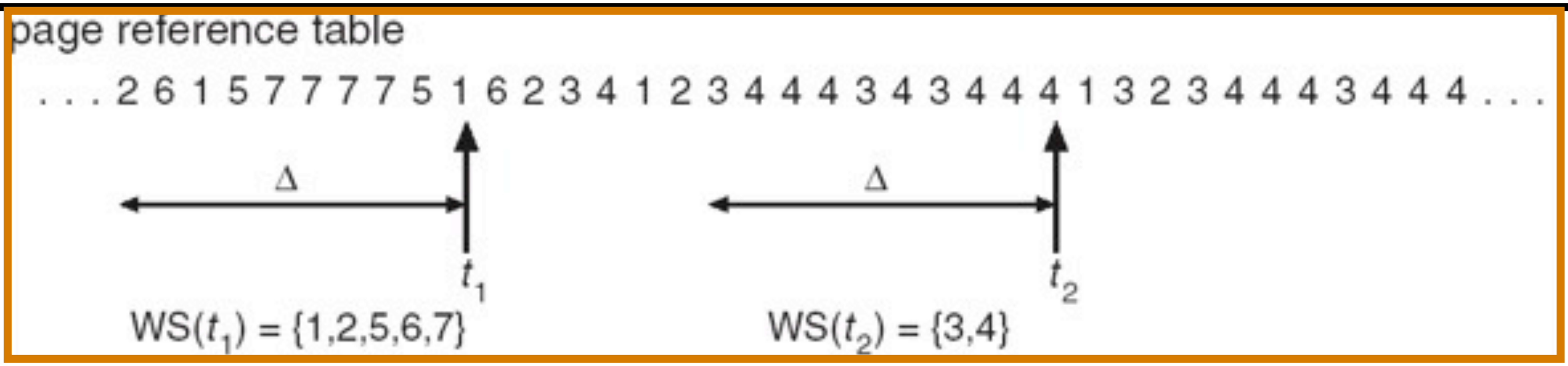
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

Locality In A Memory-Reference Pattern

- Program Memory Access Patterns have temporal and spatial locality
  - Group of Pages accessed along a given time slice called the "Working Set"
  - Working Set defines minimum number of pages needed for process to behave well
- Not enough memory for Working Set⇒Thrashing
  - Better to swap out process?



Working-Set Model



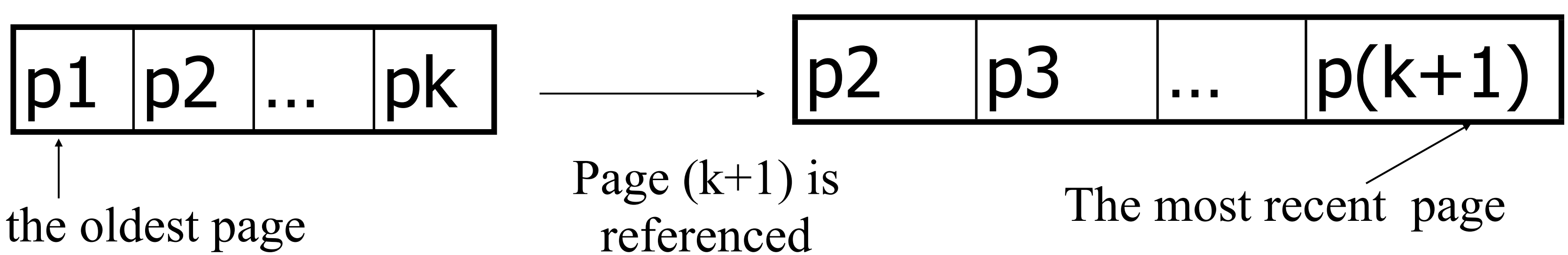
- $\Delta \equiv$ working-set window $\equiv$ fixed number of page references
  - Example: 10 million references
- WS_i (working set of Process P_i) = total set of pages referenced in the most recent Δ (varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- $D = \Sigma |WS_i| \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
  - Policy: if $D >$ m, then suspend one of the processes
  - This can improve overall system behavior by a lot!

What about Compulsory Misses?

- Recall that compulsory misses are misses that occur the first time that a page is seen
  - Pages that are touched for the first time
  - Pages that are touched after process is swapped out/ swapped back in
- Clustering:
  - On a page-fault, bring in multiple pages “around” the faulting page
  - Since efficiency of disk reads increases with sequential reads, makes sense to read several sequential pages
- Working Set Tracking:
  - Use algorithm to try to track working set of application
  - When swapping process back in, swap in working set

Maintaining WS: A Simple Way

- Store page numbers in a shift register of length *k*, and with every memory reference, we do
  - Shift the register left one position, and
  - Insert the most recently referenced page number on the right
  - The set of *k* page numbers in the register is the working set.
- Too expensive to do this for each memory reference.



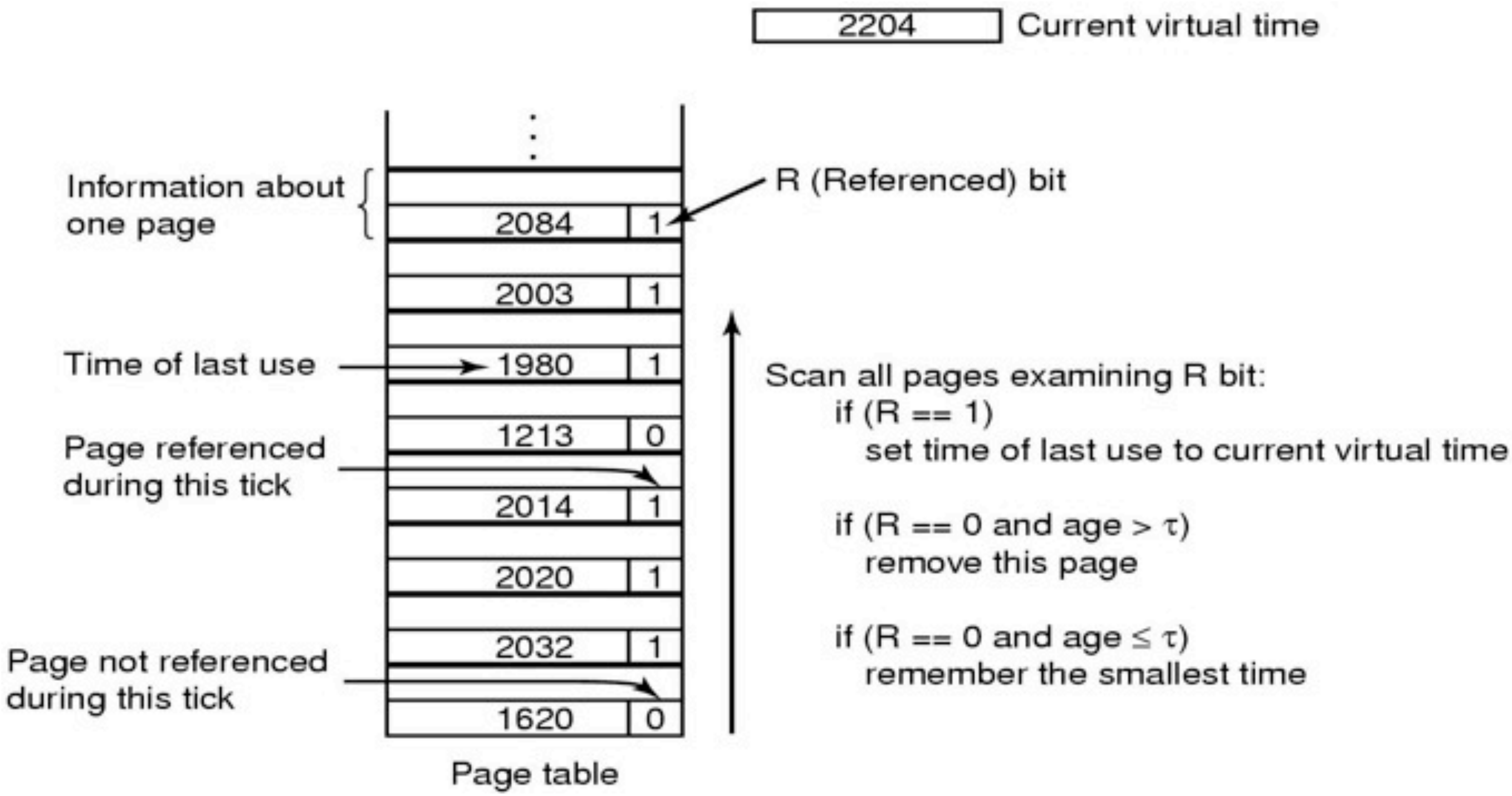
Implementation: Defining a working set

- Since not practical to keep history of past Δ memory references, use *working set window* of τ ms.
  - e.g., instead of defining working set as those pages used during previous 10 million references, define it as pages used during past working set window of 100ms
  - Note: not wall-clock time! If a process starts running at time T, and runs for 40ms at time T+100ms, it's execution time is 40ms. (the other 60ms is used for running other processes)
- We use the term *current virtual time* to denote execution time of a process since its start
  - Working set of a process is set of pages it referenced during the past τ ms of virtual time

Working set algorithm

- Recall: the R bit of a PTE is cleared every clock period. Assume the working set window τ ms spans multiple clock periods.
- *On every page* fault, the page table is scanned to look for a suitable page to evict. The *R bit* of each PTE is examined.
  - If R=1 the page has been accessed this clock period and is part of WS.
    - Its *Time of last use* is updated to the present time.
    - If R=1 for all pages in memory, a random page is evicted
  - If R=0 the age (difference between the present time and *Time of last use*) is determined.
    - If age $> \tau$, then the page is no longer considered to be part of WS. It may be removed and replaced with the new page
    - If age $\leq \tau$, then the page is still in WS. If all pages in physical memory are still in WS, the oldest one is chosen for eviction

Working set algorithm example



WSClock algorithm

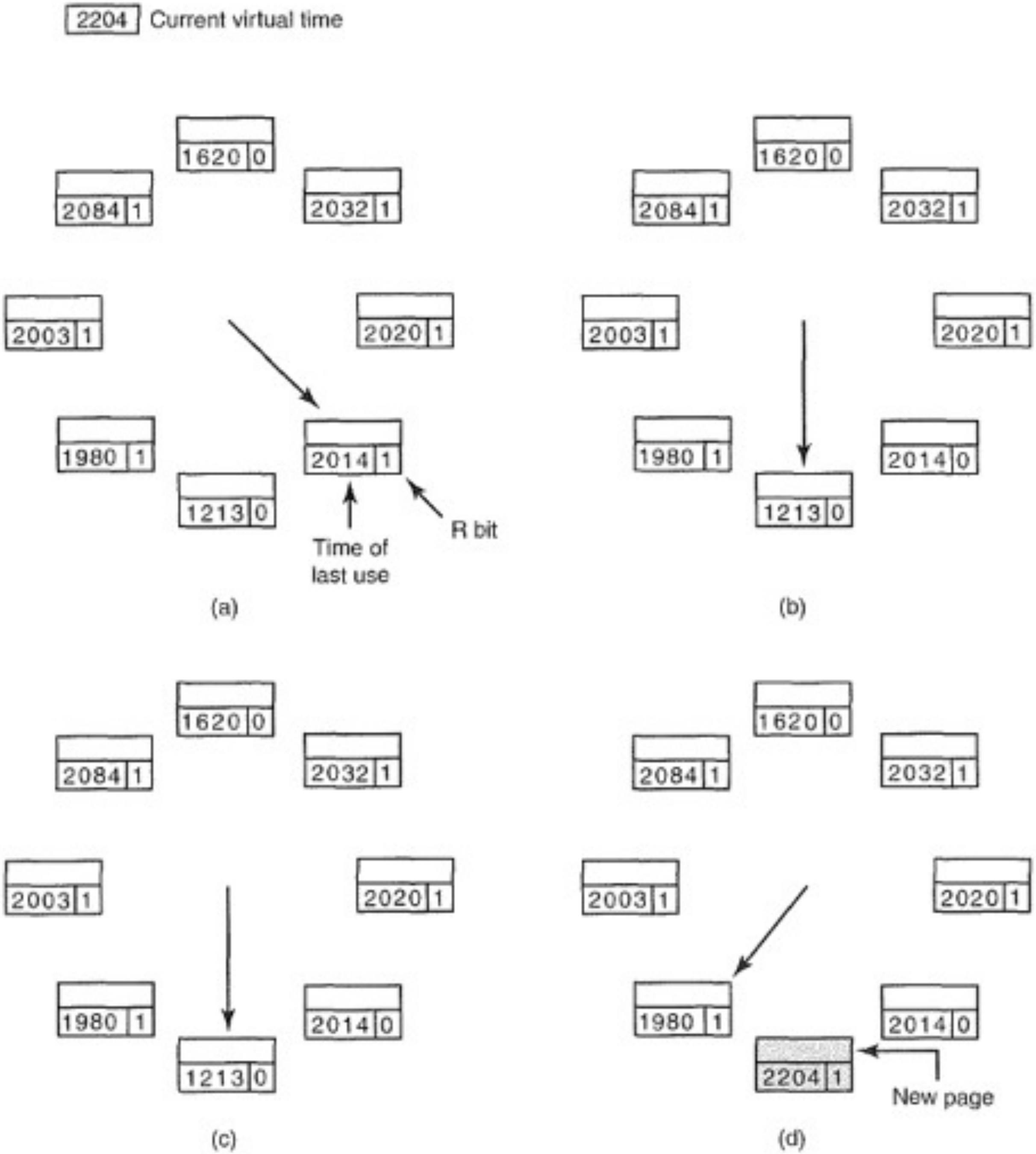
- Scanned page-table at every fault until a victim is located

- WSClock :Clock algorithm + Working set algorithm:
  - Instead of clearing R on timer, clear it at page-fault events

- Arrange physical page frames in a circle with single clock. On each page fault:
  - Advance clock hand (not real time)
  - Check R bit:
    - R=1→used recently; clear and leave alone
    - R=0→additional checking for page age:
      - If age > τ, not in WS; selected candidate for replacement
      - If age ≤ τ, in WS. If all pages in physical memory are still in WS, the oldest one is chosen for eviction

- Worst-case same as WS algorithm, but avg. case better

- (Note: this is a simplified version of WSclock. The algorithm in textbook is more complex and uses the modified bit.)



Operations of the WSClock algorithm.

(a) and (b) give an example of what happens when R = 1. (c) and (d) give an example of R = 0 and $\text{age} > \tau$.

Summary

- Replacement algorithms
  - OPT: Replace page that will be used farthest in future
  - FIFO: Place pages on queue, replace page at end
  - Second-chance: giving recently-used pages a second chance
  - LRU: Replace page used farthest in past
  - Approximations to LRU
    - NFU & Aging:
      - Keep track of recent use history for each page
    - Clock Algorithm:
      - Arrange all pages in circular list
      - Sweep through them, marking as not "in use"
      - If page not "in use" for one pass, than can replace
    - N^{th}-chance clock algorithm
      - Give pages multiple passes of clock hand before replacing
- Working Set:
  - Set of pages touched by a process recently
- Working set algorithm:
  - Tries to keep each working set in memory
- Thrashing: a process is busy swapping pages in and out
  - Process will thrash if working set doesn't fit in memory
  - Need to swap out a process

Summary

Algorithm	Comment
Optimal	Not implementable, good as benchmark
NRU	Very crude
FIFO	Might throw out important pages
Second chance	Big improvement over FIFO
Clock	Realistic
LRU	Excellent, but difficult to implement exactly
NFU	Fairly crude approximation to LRU
Aging	Efficient algorithm approximates LRU well
Working set	Somewhat expensive to implement
WSClock	Good efficient algorithm