CSE150
Operating Systems
Lecture 17

Page Allocation and Replacement (cont.)

Review: Demand Paging Mechanisms

- PTE helps us implement demand paging
  - Valid ⇒ Page in memory, PTE points at physical page
  - Not Valid ⇒ Page not in memory; use info in PTE to find it on disk when necessary
- Suppose user references page with invalid PTE?
  - Memory Management Unit (MMU) traps to OS
    - » Resulting trap is a “Page Fault”

Cache

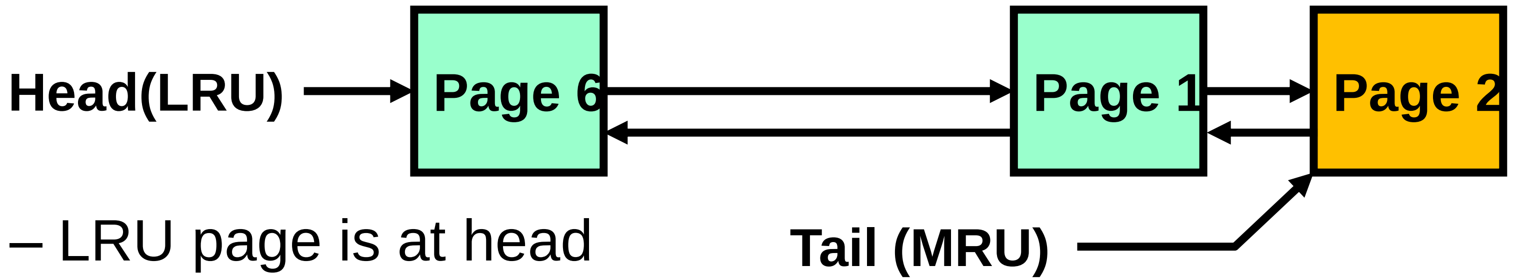
  - What does OS do on a Page Fault?:
    - » Choose an old page to replace
    - » If old page modified (“D=1”), write contents back to disk
    - » Change its PTE to be invalid
    - » Load new page into memory from disk
    - » Update page table entry
    - » Continue thread from original faulting location
  - While pulling pages off disk for one process, OS runs another process from ready queue

Review: Page Replacement Policies

- Why do we care about Replacement Policy?
  - Replacement is an issue with any cache
  - Particularly important with pages
    - » The cost of being wrong is high: must go to disk
    - » Must keep important pages in memory, not toss them out
- FIFO (First In, First Out)
  - Throw out oldest page. Be fair – let every page live in memory for same amount of time.
  - Bad, because throws out heavily used pages instead of infrequently used pages
- MIN (Minimum):
  - Replace page that won't be used for the longest time
  - Great, but can't really know future…
  - Makes good comparison case, however
- RANDOM:
  - Pick random page for every replacement
  - Typical solution for TLB's. Simple hardware
  - Unpredictable – makes it hard to make real-time guarantees

Review: Replacement Policies (Con't)

- LRU (Least Recently Used):
  - Replace page that hasn't been used for the longest time
  - Programs have locality, so if something not used for a while, unlikely to be used in the near future.
  - Seems like LRU should be a good approximation to MIN.
- Different if we access a page that is already loaded:



  - LRU page is at head
  - When a page is used again, **remove from list**, add it to tail.
  - Eject head if list longer than capacity
- Problems with this scheme for paging? Too Expensive
  - Updates are happening on page **use**, not just swapping
  - List structure requires extra pointers compared to FIFO, more updates

Today

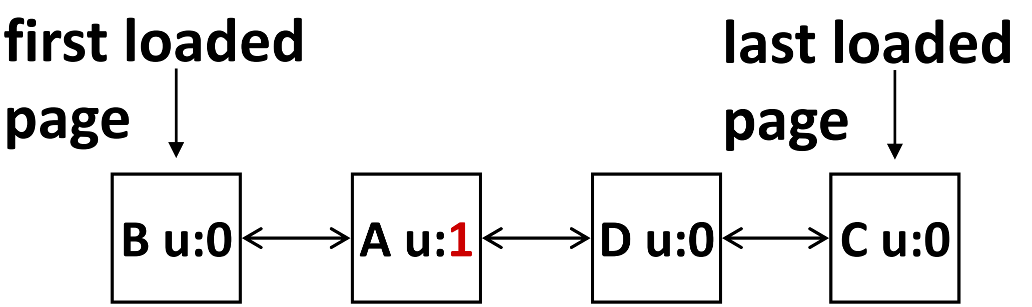
- Page Replacement Policies
  - Second Chance
  - Clock Algorithm
  - N^{th} chance clock algorithm
- Page Allocation Policies
- Working Set/Thrashing

Implementing LRU & Second Chance

- Perfect:
  - Timestamp page on each reference
  - Keep list of pages ordered by time of reference
  - Too expensive to implement in reality
- Second Chance Algorithm:
  - Approximate LRU (approx. to approx. to MIN)
    - » Replace an old page, not the oldest page
  - FIFO with "use" bit
- Details
  - A "use" bit per physical page
    - » Set when page accessed
    - » If not set, not referenced since last time use bit was cleared
  - On page fault check page at head of queue
    - » If use bit=1 → clear bit, and move page to tail (give the page second chance!)
    - » If use bit=0 → replace page
  - Moving pages to tail still complex

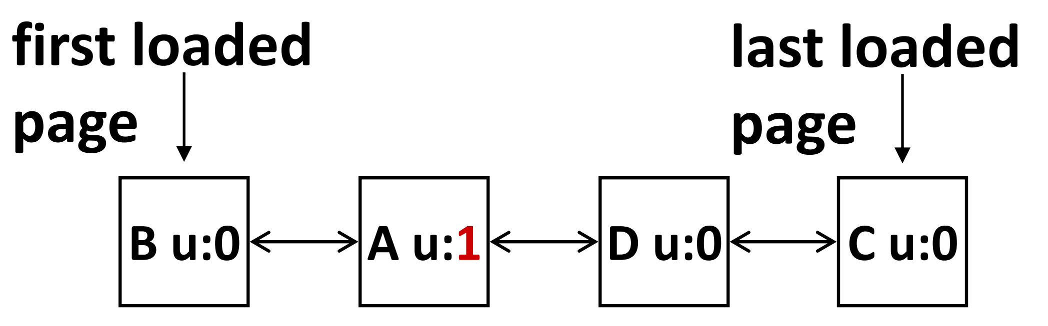
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives



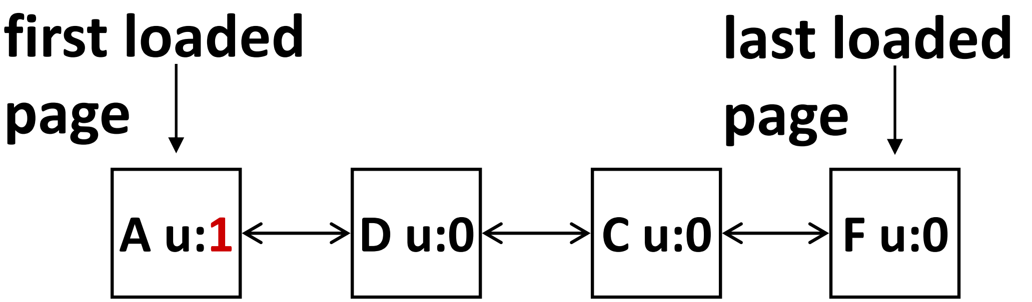
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives



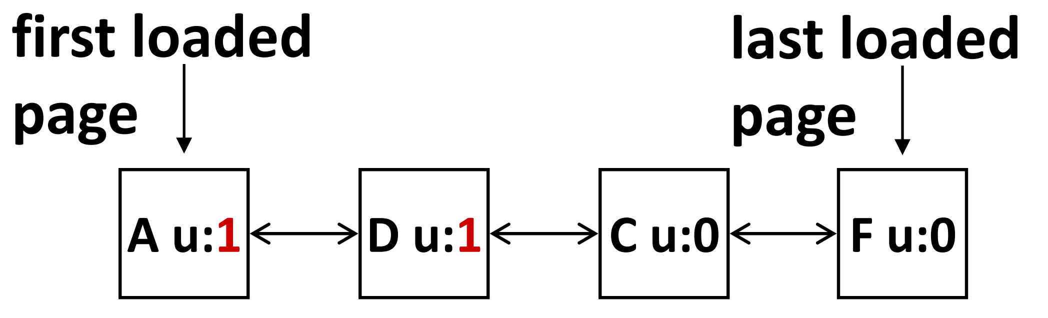
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives



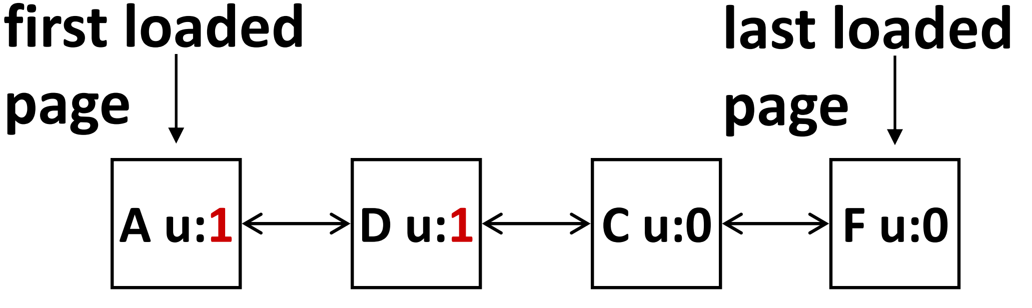
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D



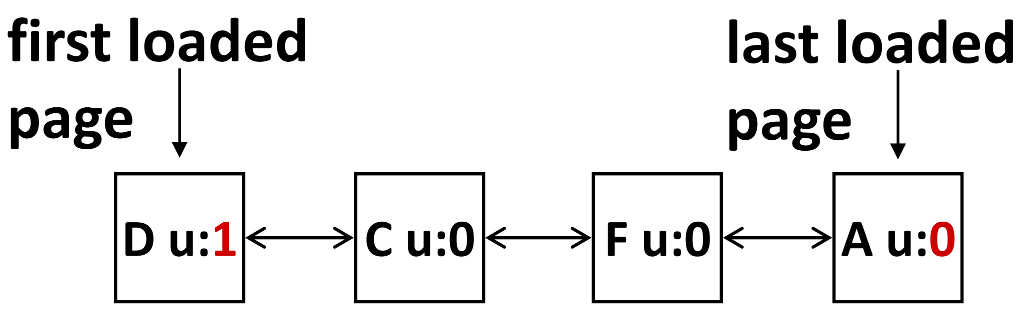
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



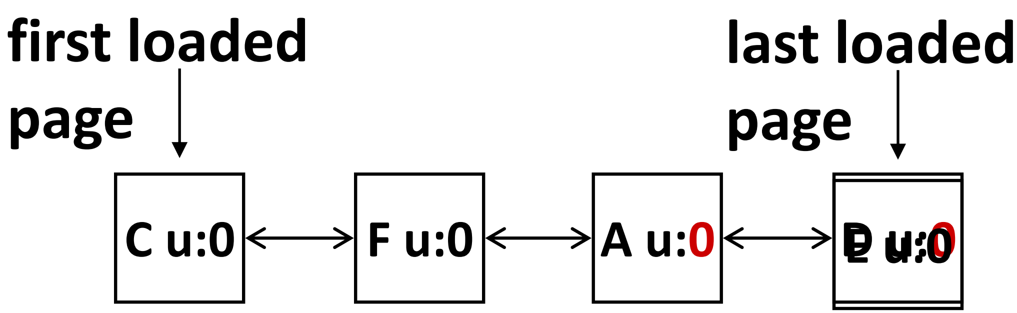
Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



Second Chance Illustration

- Max page frames = 4
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives

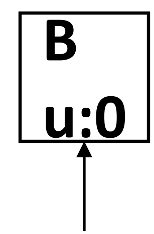


Clock Algorithm

- Clock Algorithm: more efficient implementation of second chance algorithm
  - Arrange physical pages in circle with single clock hand
- Details:
  - On page fault:
    - » Check use bit: 1→used recently; clear and leave it alone
      0→selected candidate for replacement
    - » Advance clock hand (not real time)
  - Will always find a page or loop forever?
    - » Even if all use bits set, will eventually loop around

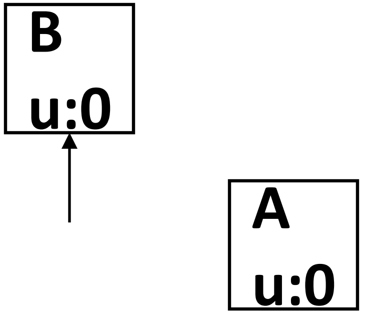
Clock Replacement Illustration

- Max page table size 4
- Invariant: point at oldest page
  - Page B arrives



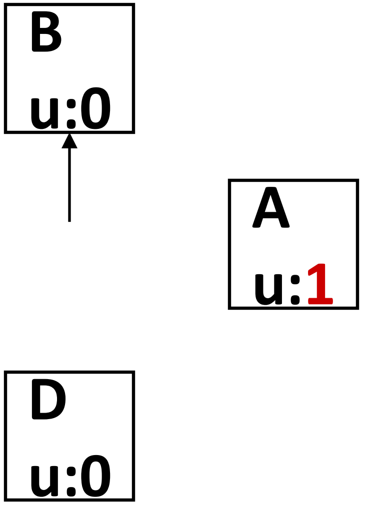
Clock Replacement Illustration

- Max page frames = 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A



Clock Replacement Illustration

- Max page frames = 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives

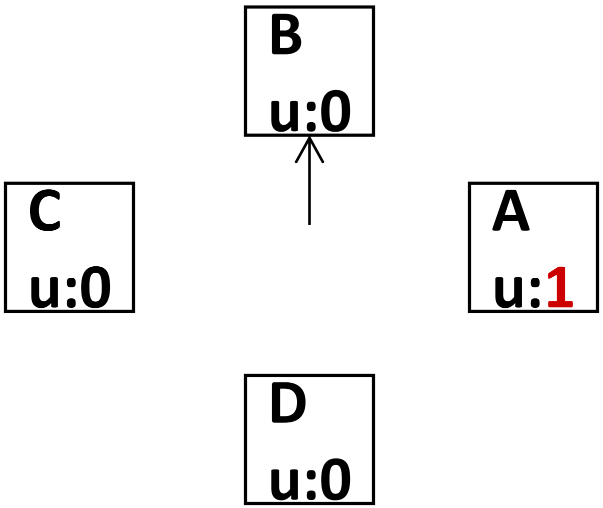


Clock Replacement Illustration

- Max page frames = 4

- Invariant: point at oldest page

  – Page B arrives
  – Page A arrives
  – Access page A
  – Page D arrives
  – Page C arrives

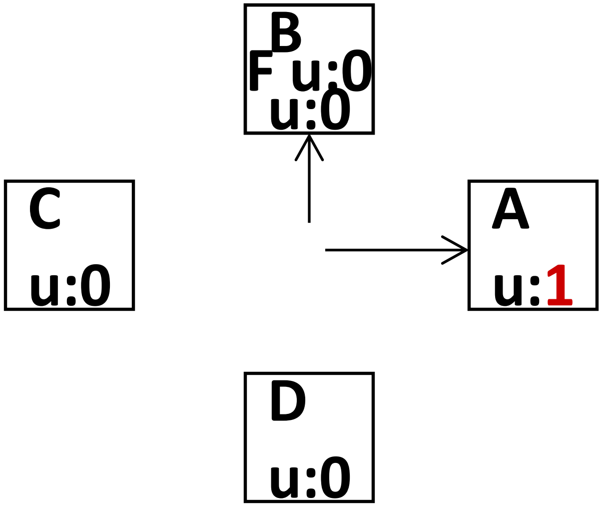


Clock Replacement Illustration

- Max page table frames = 4

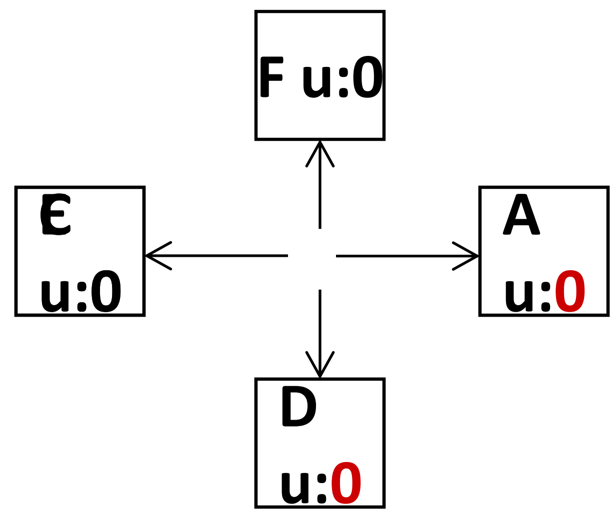
- Invariant: point at oldest page

  – Page B arrives
  – Page A arrives
  – Access page A
  – Page D arrives
  – Page C arrives
  – Page F arrives
  – Access page D



Clock Replacement Illustration

- Max page frames = 4
- Invariant: point at oldest page
  - Page B arrives
  - Page A arrives
  - Access page A
  - Page D arrives
  - Page C arrives
  - Page F arrives
  - Access page D
  - Page E arrives



Clock Algorithm: Discussion

- What if hand moving slowly?
  - Good sign or bad sign?
    - » 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

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 use bit:
    - » 1⇒clear use and also clear counter (used in last sweep)
    - » 0⇒increment counter; if count=N, replace page
  - Means that clock hand has to sweep by N times without page being used before page is replaced
- How do we pick N?
  - Why pick large N? Better approx to LRU
    - » If N ~ 1K, really good approximation
  - Why pick small N? More efficient
    - » Otherwise might have to look a long way to find free page
- What about dirty pages?
  - Takes extra overhead to replace a dirty page, so give dirty pages an extra chance before replacing
  - Common approach:
    - » Clean pages, use N=1
    - » Dirty pages, use N=2 (and write back to disk when N=1)

Allocation of Page Frames (Memory Pages)

- How do we allocate memory among different processes?
  - Does every process get the same fraction of memory? Different fractions?
  - Should we completely swap some processes out of memory?
- Each process needs *minimum* number of pages
  - Want to make sure that all processes that are loaded into memory can make forward progress
- Possible Replacement Scopes:
  - Global replacement – process selects replacement frame from set of all frames; one process can take a frame from another
  - Local replacement – each process selects from only its own set of allocated frames

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:

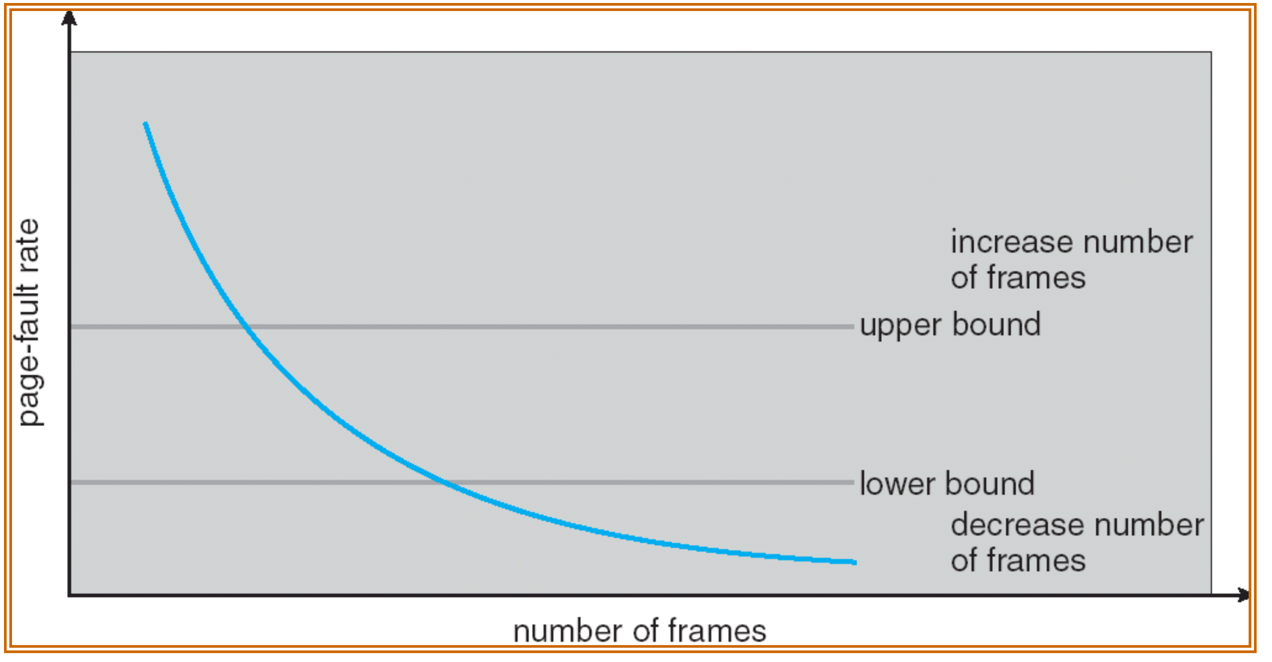
    s_i = size of process p_i and $S = \Sigma s_i$

    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: If process p_i generates a page fault, select for replacement a frame from a process with lower priority number
- Perhaps we should use an adaptive scheme instead???
  - What if some application just needs more memory?

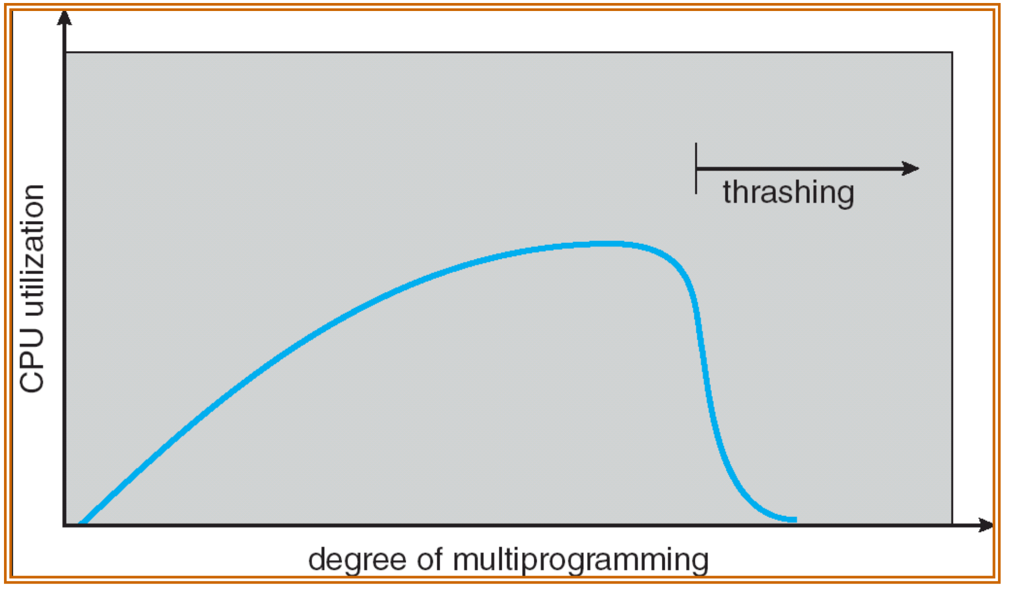
Page-Fault Frequency Allocation

- 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?

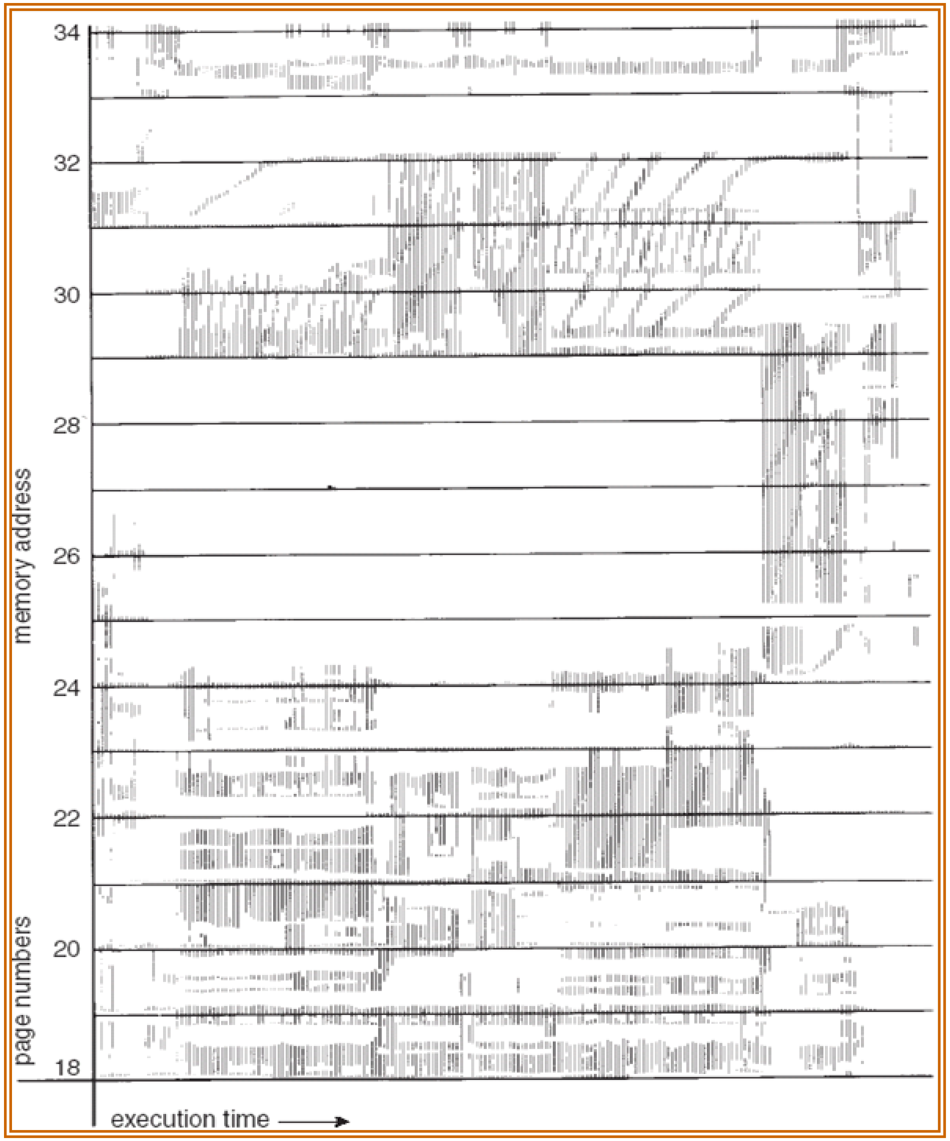
Thrashing



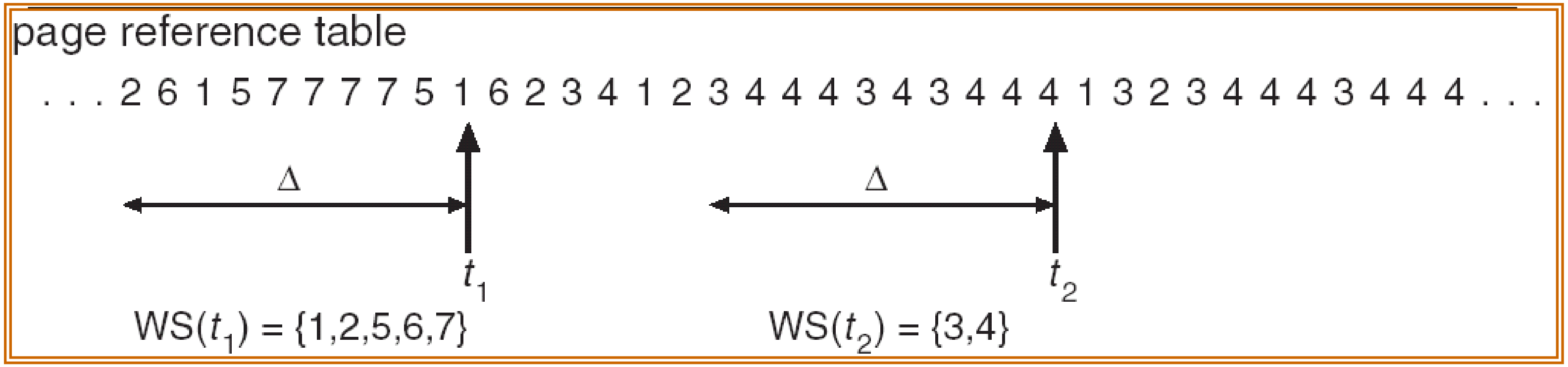
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing ≡ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response 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: 10,000 instructions
- 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/swap out 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

Demand Paging Summary

- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - MIN: Replace page that will be used farthest in future
  - LRU: Replace page used farthest in past
- Clock Algorithm: Approximation to LRU
  - 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: Another approx LRU
  - Give pages multiple passes of clock hand before replacing
- Working Set:
  - Set of pages touched by a process recently
- 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

Next lecture

- Chapter 13 – I/O systems