ECE 350
Real-time Operating Systems

Lecture 9: Demand Paging

Prof. Seyed Majid Zahedi

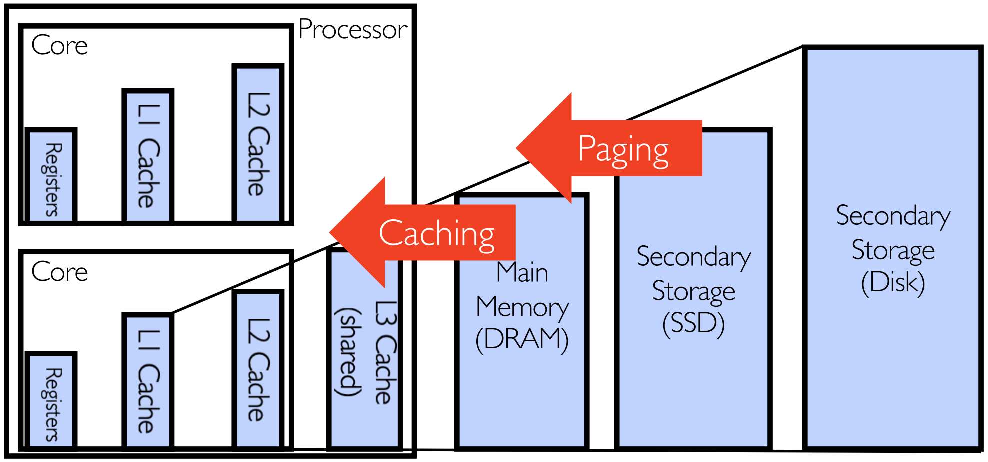
https://ece.uwaterloo.ca/~smzahedi

Outline

- Demand paging
- Replacement policies
  - FIFO, MIN, LRU
- Clock algorithm
- Nth-chance algorithm

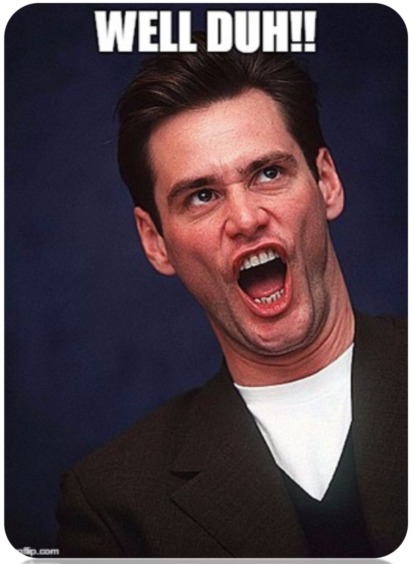
Demand Paging

- Modern programs require a lot of physical memory
  - Memory per system is growing faster than 25%-30% per year
- But they don't use all their memory most of the time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's code to be in memory
- Solution: demand paging (also known as paging)
  - Use main memory as cache for disk



Demand Paging is Just Caching …

- What is block size?
  - One page
- What is organization of cache structure?
  - Fully associative
- How do we find pages in cache?
  - First check TLB, then page-table traversal
- What is page replacement policy?
  - This requires more explanation… (coming next!)
- What happens on misses?
  - Go to lower level (i.e., disk) to resolve miss
- What happens on writes?
  - Write-back



Recall: x86 64-bit PTE

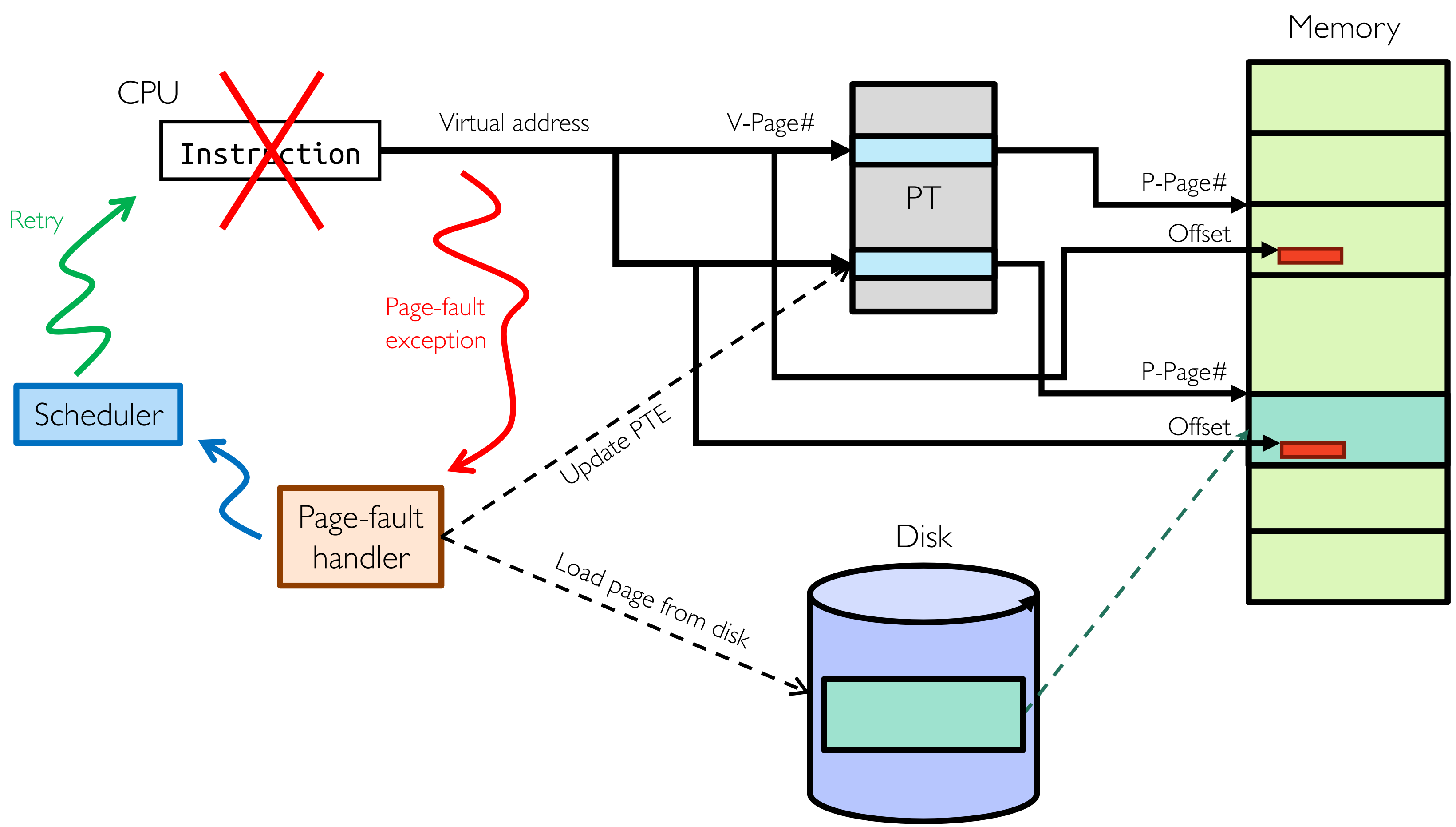
NX	SW	Reserved	P-Page Number	U	P	CW	GL	L	D	A	CD	WT	O	W	V
63	62-52	51-40	39-12	11	10	9	8	7	6	5	4	3	2	1	0

- V: Valid
- W: Read/write
- O: Owner (user/kernel)
- WT: Write-through (more on this soon)
- CD: Cache-disabled (page cannot be cached)
- A: Accessed: page has been accessed recently
- D: Dirty bit (page has been modified recently)
- L: Large page
- G: Global
- CP: Copy-on-write
- P: Prototype PTE
- U: Reserved
- SW: Software (working set index)
- NX: No-execute

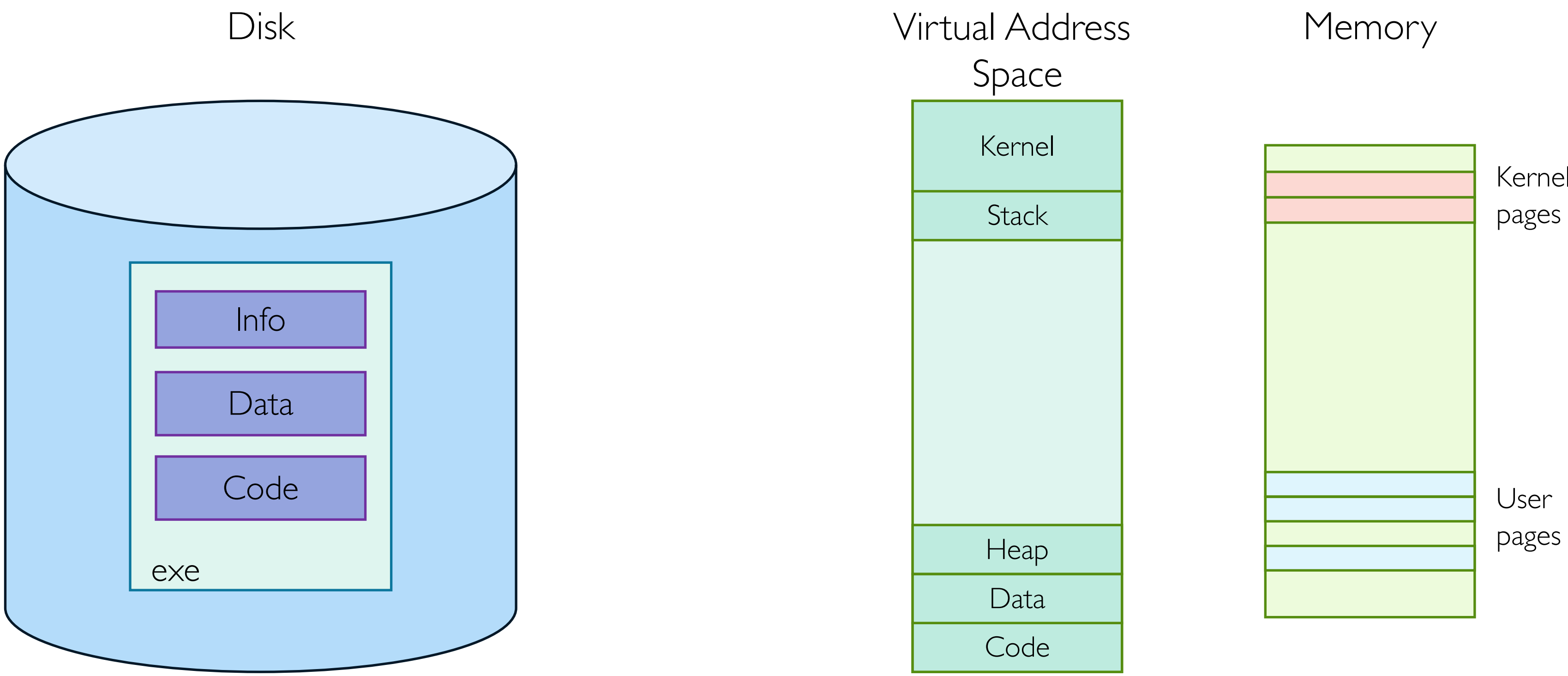
Demand Paging Overview

- PTE helps us implement demand paging
  - Valid $\Rightarrow$ page in memory, PTE points to physical page
  - Invalid $\Rightarrow$ page not in memory; use info in PTE to find it on disk when necessary
- What happens on references to page with invalid PTE?
  - Page-fault exception $\Rightarrow$ trap to OS
- What does OS do on page fault?
  - Allocate physical page to referenced virtual page
  - Load new page into memory from disk and make PTE valid
- What if there are no free physical pages?
  - Evict one and write back its content to disk if it has been modified (i.e., dirty bit is set)
  - Invalidate PTEs and TLB entries pointing to evicted physical page
- While pulling pages off disk for one process, run another one from ready queue
  - Suspended process sits on disk's waiting queue

Paging Big Picture!

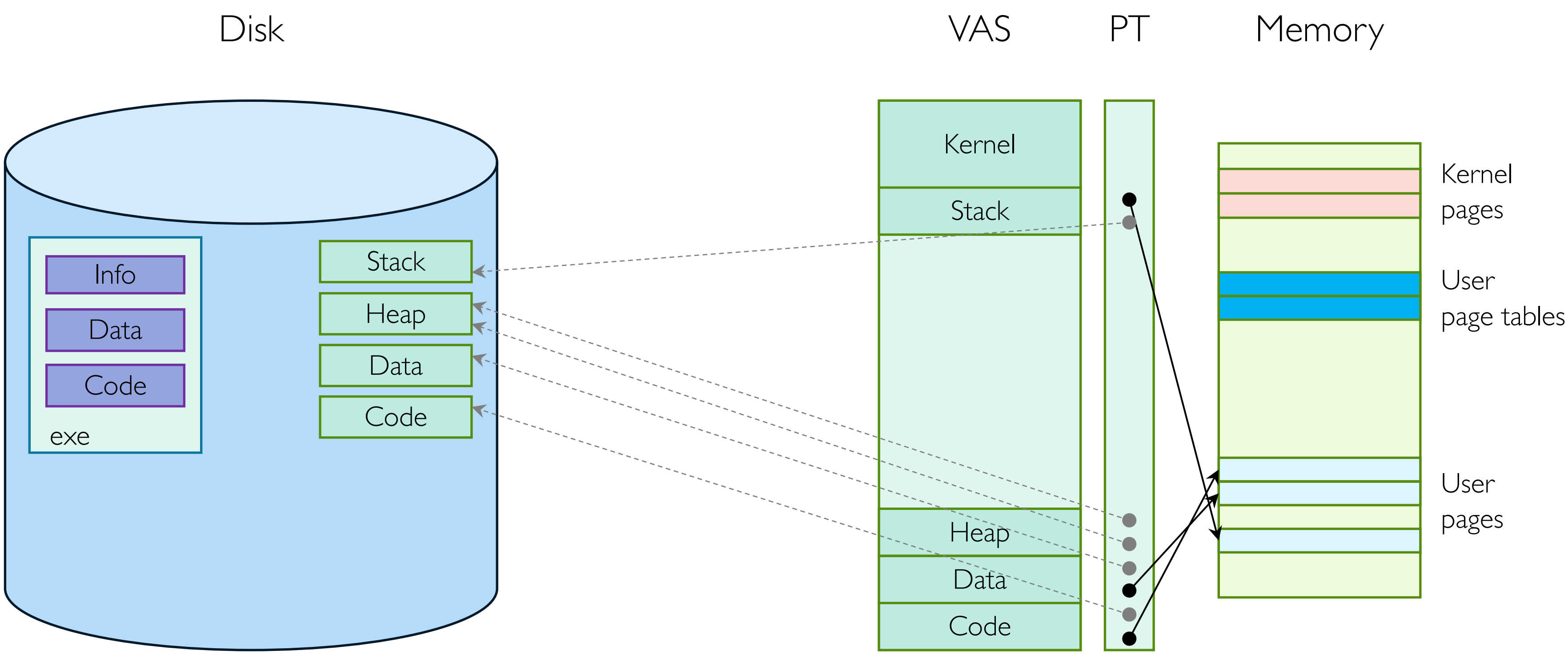


Example: Loading Executable



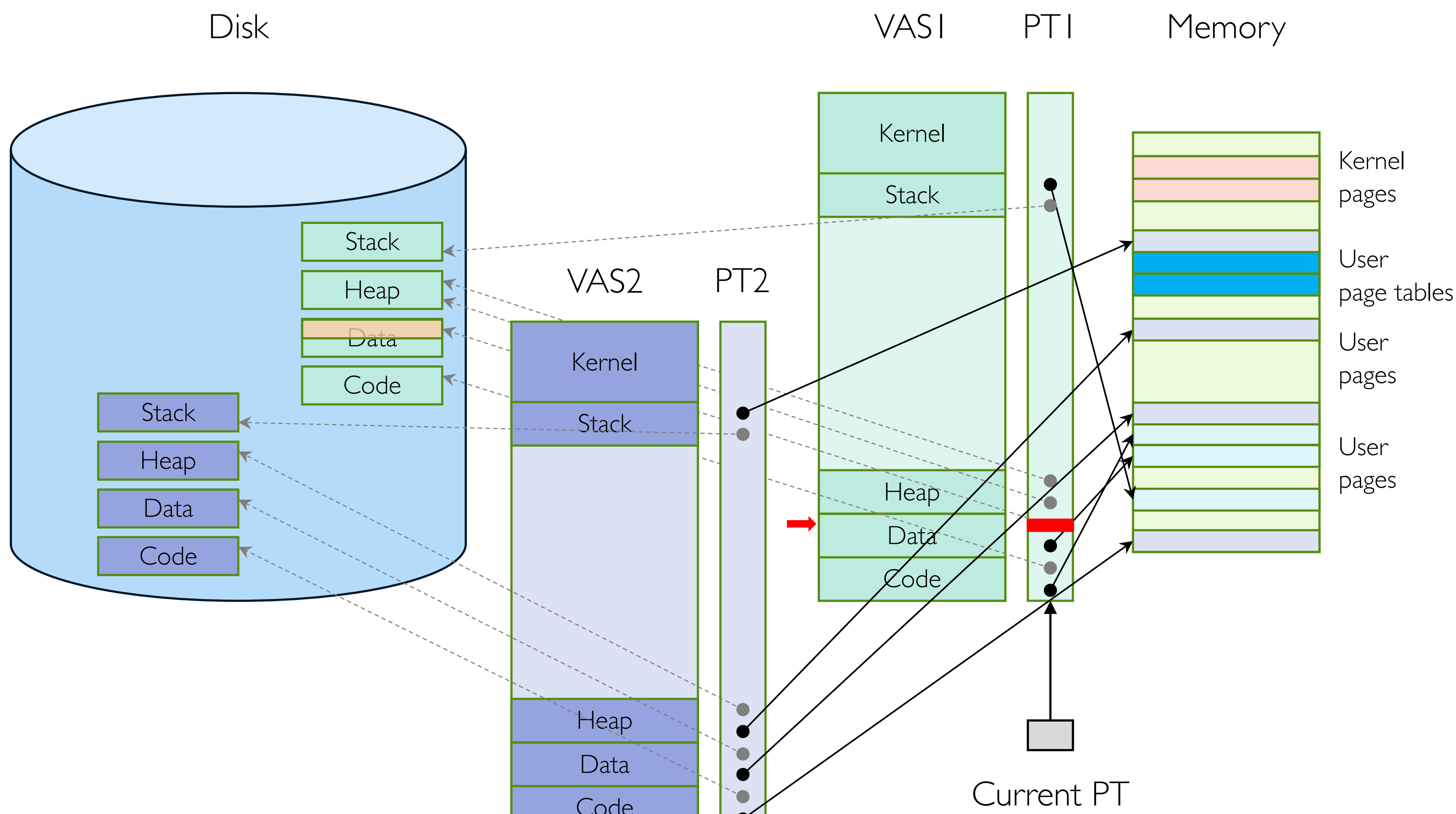
- Each executable file lives on disk in file system
  - Contains contents of code & data segments, relocation entries and symbols
- OS loads executable file into memory, initializes registers (and initial stack pointer)
- Program sets up stack and heap upon initialization (e.g., `crt0()` in C)

Example: Provide Backing Store

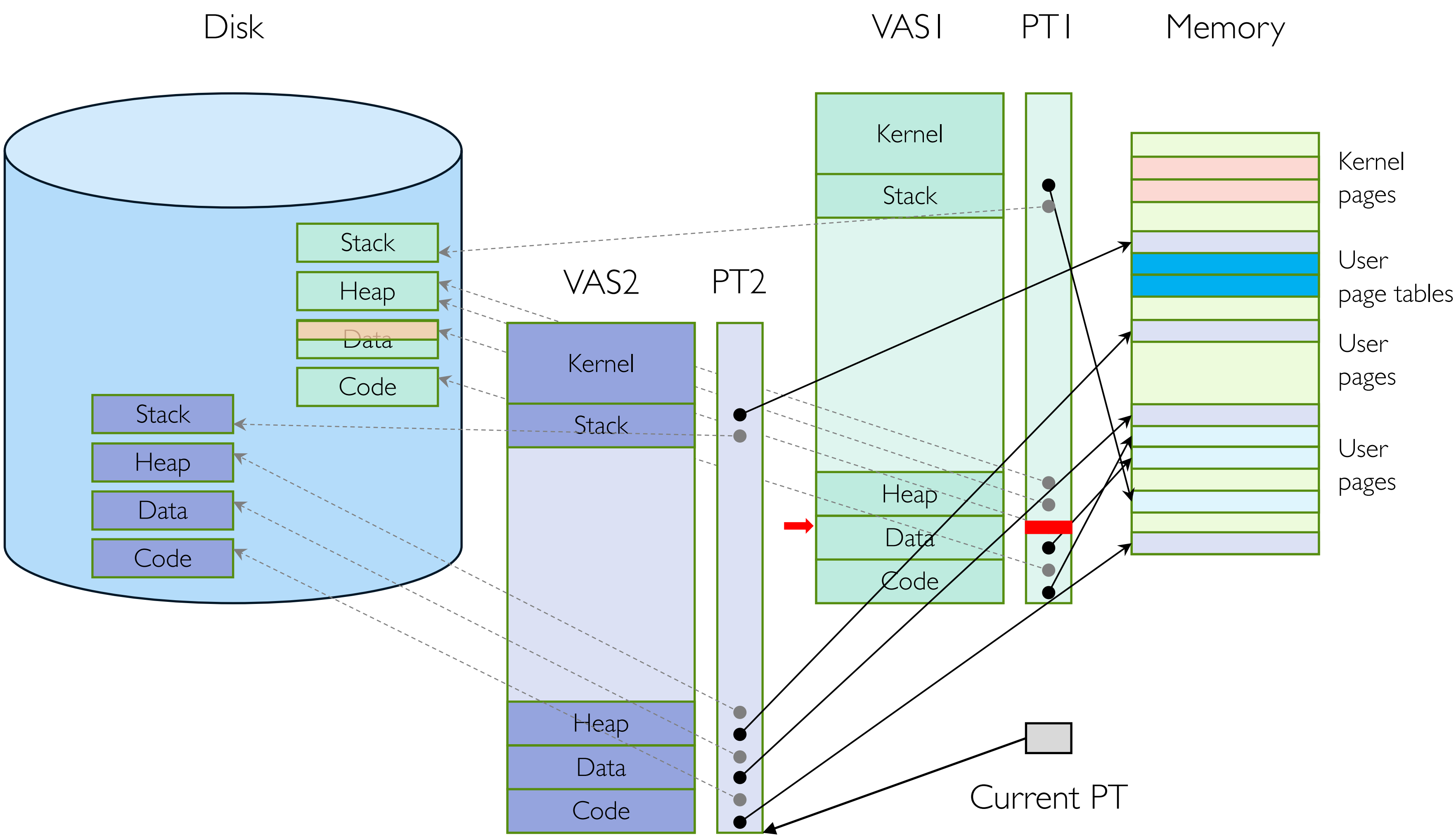


- All used virtual pages are backed by page blocks on disk (called backing store or swap file)
- User page tables map entire virtual address space
    - OS must record where to find non-resident virtual pages on disk
    - Some OSs utilize spare space in PTE for paged blocks
    - Portion of page tables that HW needs to access must be also resident in memory

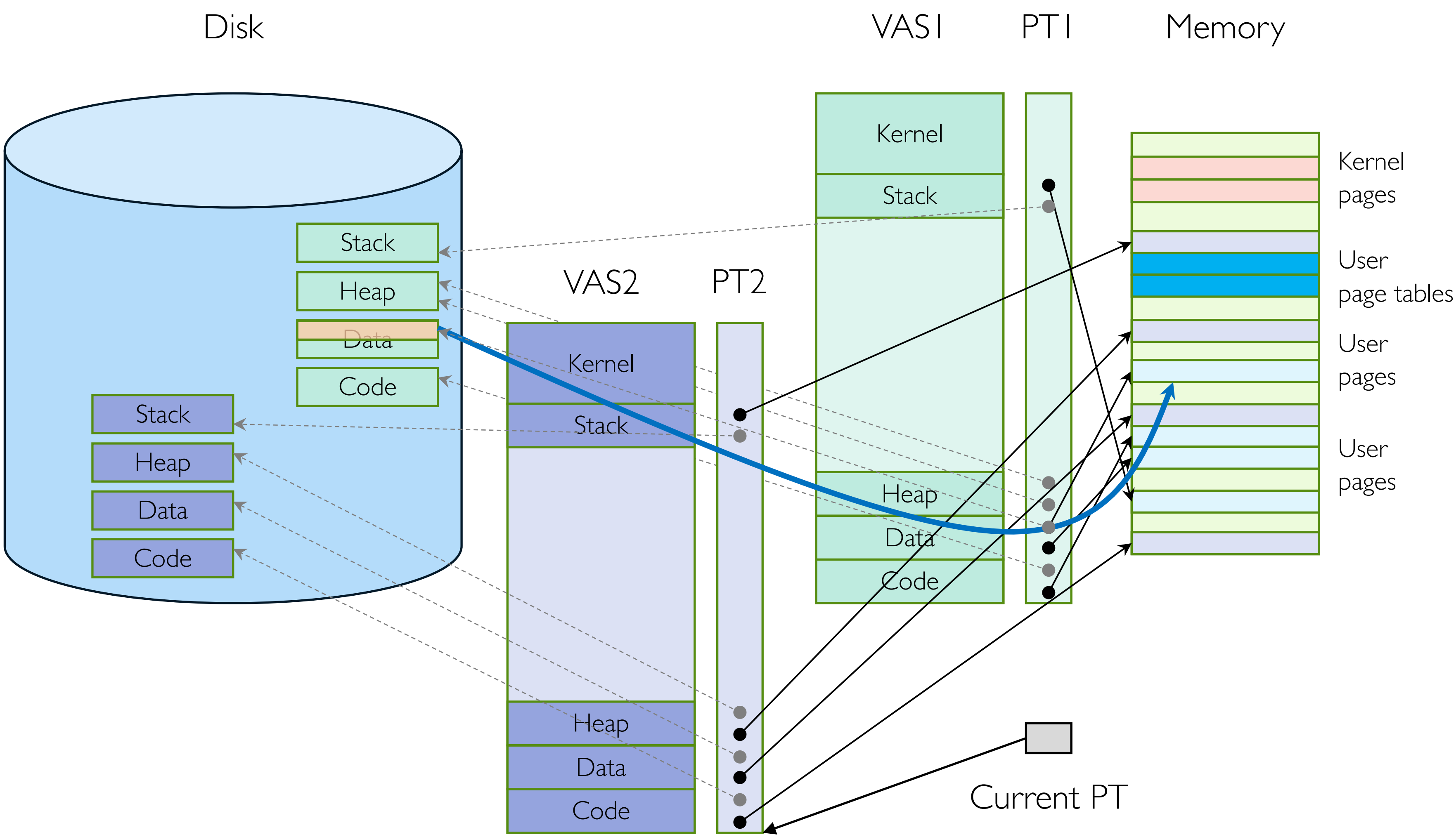
Example: On Page Fault …



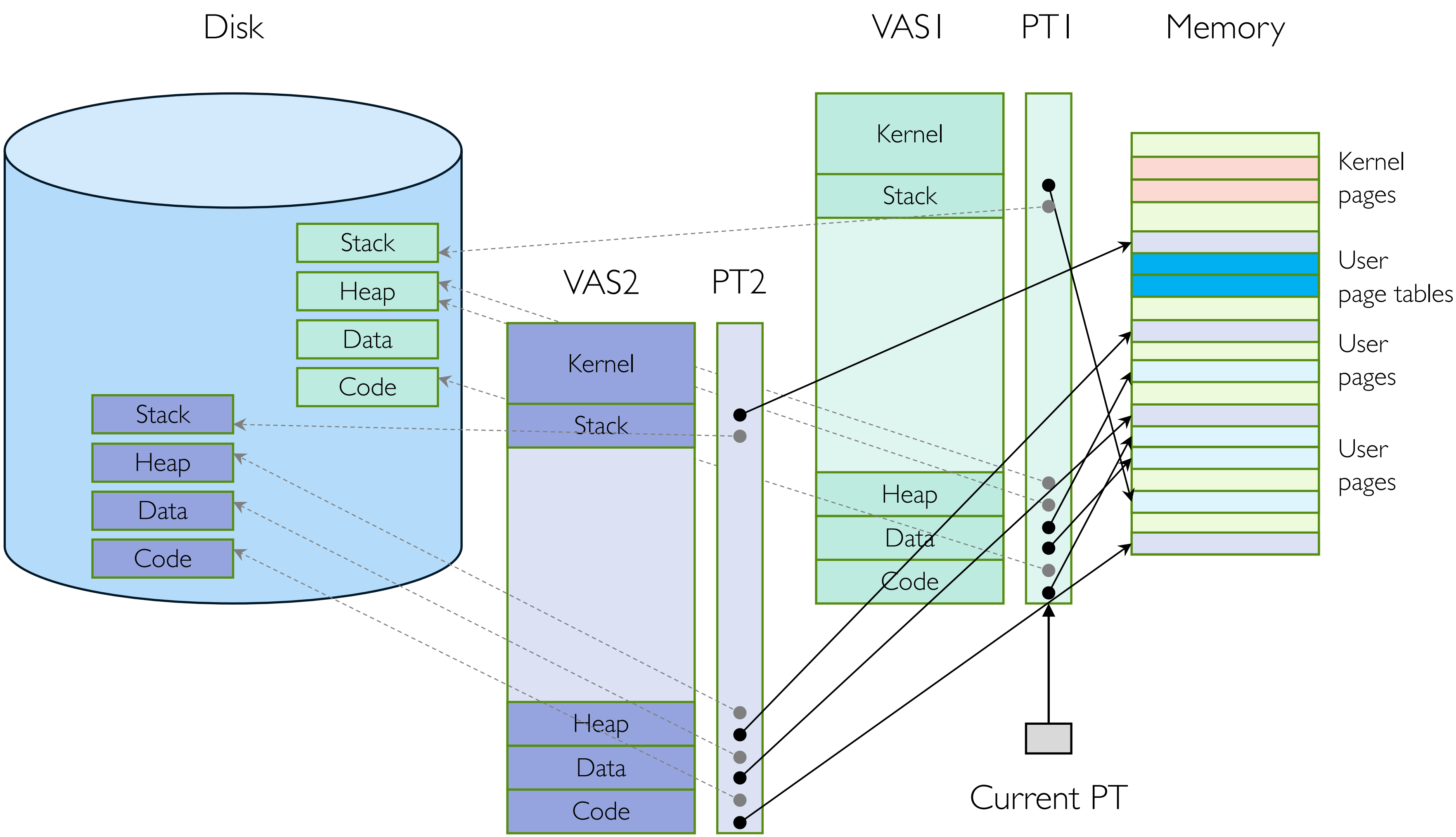
Example: On Page Fault … Schedule Other Process



Example: On Page Fault … Update PTE



Example: Resume from Faulting Instruction



Demand Paging Cost Model

- Effective access time (EAT) = Hit time + Miss ratio × Miss time
- Example:
  - Memory access time = 200ns, avg page-fault service time = 8ms, and miss ratio = p
  - EAT = 200ns + p × 8ms = 200ns + p × 8,000,000ns
- If one out of 1,000 accesses causes page fault, then EAT = 8.2μs
  - 40x slowdown!
- What if we want slowdown of less than 10%?
  - $200\text{ns} \times 1.1 > \text{EAT} \Rightarrow p < 2.5 \times 10^{-6}$
  - This is approximately single page fault in every 400,000 accesses!

What Factors Lead to Misses?

- Compulsory misses
  - Pages that have never been paged into memory before
  - How might we remove these misses?
    - Prefetching: loading them into memory before needed
    - Need to predict future somehow!
- Capacity misses
  - Not enough memory; must somehow increase available memory size
  - Can we do this?
    - One option is increasing amount of DRAM (not quick fix!)
    - Another option is adjusting percentage of memory allocated to process if multiple processes are in memory
- Conflict misses
  - Technically, conflict misses don't exist in virtual memory, since it is "fully-associative" cache
- Policy misses
  - Caused when pages were in memory, but kicked out prematurely because of replacement policy
  - How to fix this?
    - Better replacement policy

Page Replacement Policies

- Random
  - Pick random page for every replacement
  - + Simple hardware (typical solution for TLB's)
  - – Very unpredictable (makes it hard to provide any real-time guarantees)
- First-in-first-out (FIFO)
  - Throw out oldest page
  - + Fair (let every page live in memory for same amount of time)
  - – Not optimal (could throw out heavily used pages instead of infrequently used)
- Minimum (MIN)
  - Replace page that won't be used for the longest time in future
  - + Optimal (perfect benchmark)
  - – Impractical (how can we really know future?)
- Least-recently-used (LRU):
  - Replace page that hasn't been used for the longest time (if it hasn't been used for a while, it's unlikely to be used in near future)
  - + Seems like LRU should be good approximation to MIN
  - – High implement overhead (need to track all references to all pages)

Example: FIFO

- Suppose we have 3 p-pages , 4 v-pages, and following reference stream:

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

- FIFO: 7 faults
- When referencing D, replacing A is bad choice, since we'll need A again right away

Example: MIN

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

- MIN: 5 faults
  - Where will D be brought in? Look for page not referenced farthest in future
- What will LRU do?
  - Same decisions as MIN here but won't always be true!

When Will LRU Perform Badly?

Page \ Ref	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

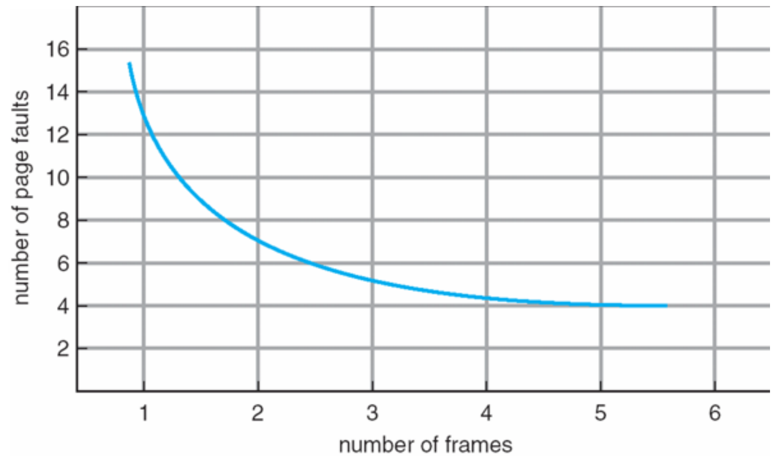
- Every reference leads to page fault!

When will LRU Perform Badly? (cont.)

- MIN Does much better

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

Memory Size and Page Fault Rate



- One desirable property: When you add memory the miss rate drops
  - Does this always happen?
  - Seems like it should, right?
- No: Bélády's anomaly
  - Certain replacement policies don't have this obvious property!

Bélády's Anomaly

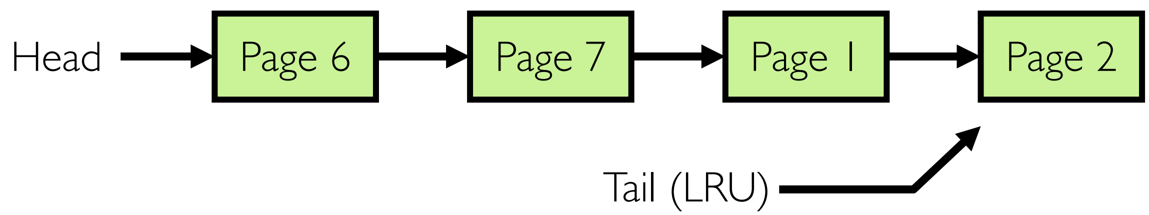
Ref: Page:	A	B	C	D	A	B	E	A	B	C	D	E
1	A			D			E					
2		B			A					C		
3			C			B					D	

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		

- After adding memory:
  - With FIFO, contents can be completely different
  - With LRU or MIN, contents of memory with X pages are a subset of contents with X+1 Page

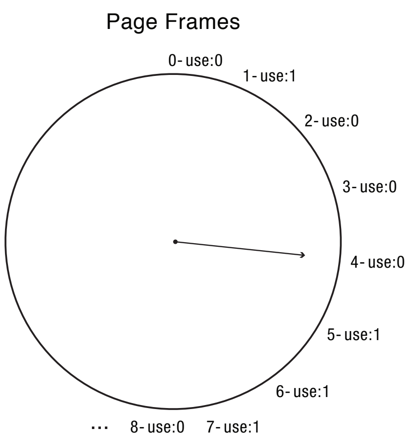
LRU Implementation

- How to implement LRU? Use a list!



  - On each use, remove page from list and place at head, LRU page is at tail
- Problems with this scheme for paging?
  - Need to know when each page is used to change its position in list
  - Add extra overhead to each memory access

Clock Algorithm: LRU Approximation



- Arrange physical pages in circle with single clock hand
- Page-table walk sets accessed bit of PTE on TLB miss
  - No change on further accesses resolved in TLB!
    (recall: TLB entries usually don't have accessed bit)
- On page fault, advance clock hand and then check access bit
  - If `1`, clear it, invalidate TLB entry, advance clock hand, and repeat
  - If `0`, pick candidate for replacement and terminate
- Clock algorithm finds an old page, not the oldest page
- Will this algorithm always find replacement page, or does it loop forever?
  - If all use accessed bits are set, clock hand will eventually loop around ⇒ FIFO

Clock Algorithm: Discussion

- What if hand is moving slowly? Is it a good sign or a bad sign?
  - A good sign! Not many page faults and/or find page quickly
- What if hand is moving quickly?
  - Not a good sign! Lots of page faults and/or lots of reference bits set
- One way to view clock algorithm
  - Crude partitioning of pages into two groups: young and old
  - Why not partition into more than 2 groups?

N^{th}-chance Algorithm: Modified Clock Algorithm

- Give each page N chances
  - OS keeps counter per page to track number of times it qualifies for replacement
  - On page fault, advance clock hand and check access bit
    - `1` → clear it, invalidate TLB entry, clear counter, advance clock hand, and repeat
    - `0` → increment counter; if counter is N, pick as replacement candidate

- How do we pick N?
  - Large N: better approximation to LRU, more overhead to find replacement candidate
  - Small N: more efficient, less accurate
- What about dirty pages?
  - It takes extra overhead to replace dirty page, let dirty pages survive one extra sweep
  - If counter is N and dirty bit is set, decrement counter and write back to disk

Clock Algorithms: Discussion

- Can run synchronously with page-fault handler
  - When page-fault handler, run clock algorithm to find next page to evict

- Can run asynchronously with page-fault handler
  - Maintain pool of candidate pages
  - On page fault, evict one page from pool
  - Run clock algorithm when size of pool decreases beyond fixed threshold
  - Write dirty pages back to disk when they are added to pool
  - Remove page from pool if it is accessed before eviction

Allocation of Physical Pages

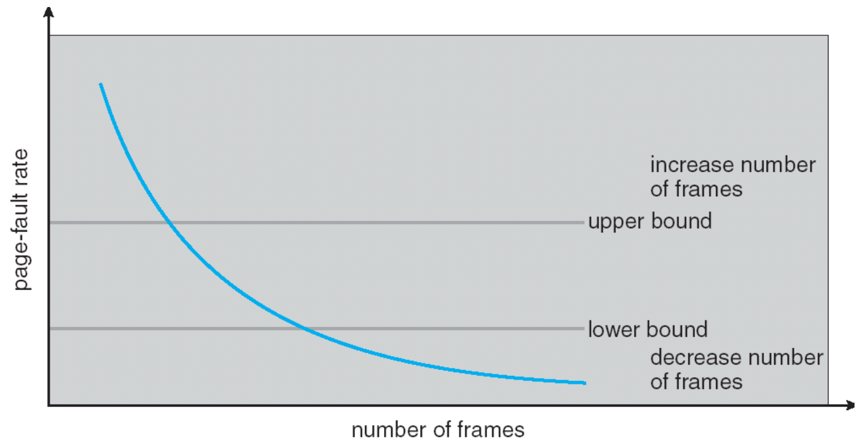
- How do we allocate memory among different processes?
  - Does every process get same fraction of memory?
  - Should we completely swap some processes out of memory?
- Each process needs minimum number of pages
  - All processes loaded into memory should make progress
- Possible replacement scopes
  - Global replacement – to make space for one process's page, replacement is selected from all processes' pages
  - Local replacement – to make space for one process's page, replacement is selected from process' set of allocated pages

Fixed-priority Allocation

- Equal allocation (fixed scheme)
  - Every process gets same amount of memory
  - Example: 100 physical pages, 5 processes → Each. process gets 20 pages
- Proportional allocation (fixed scheme)
  - Allocate according to size of process
  - Computation proceeds as follows:
    - s_i = size of process p_i and S = sum of s_i's for all p_i's
    - m = total number of physical pages
    - a_i = allocation for pi = $(s_i \times m) / S$
- Priority allocation
  - Proportional scheme using priorities rather than size
  - Possible behavior: If process p_i generates page fault, select for replacement page from process with lower priority number
- Perhaps we should use an *adaptive* scheme instead?
  - What if some application just needs more memory?

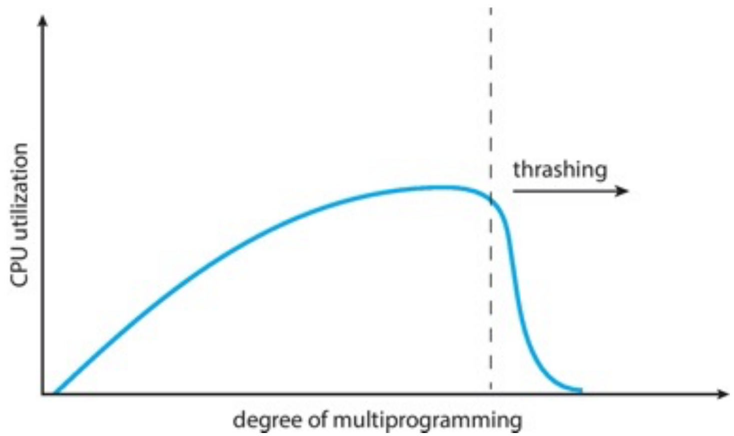
Page-fault Rate: Capacity Misses

- Can we reduce capacity misses by dynamically changing # of pages per application?



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

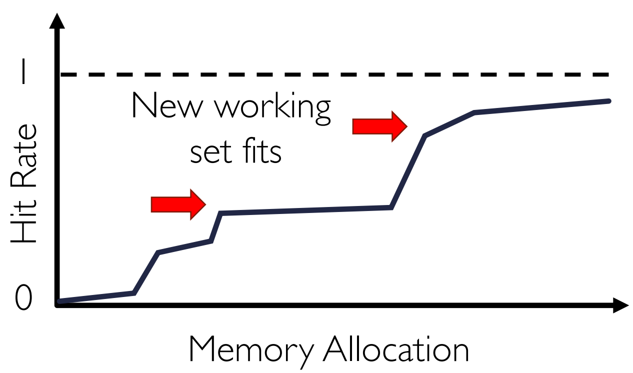
Thrashing



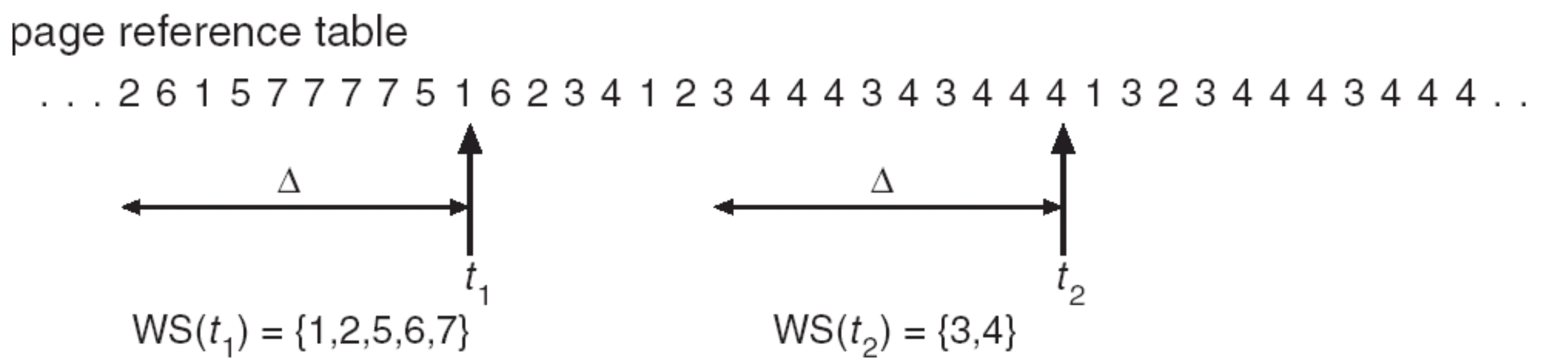
- If process does not have "enough" pages, page-fault rate is very high which leads to
  - Low CPU utilization
  - OS spends most of its time swapping pages to disk
- Thrashing ≡ process is busy swapping pages in and out disk
- Questions:
  - How do we detect thrashing?
  - What is best response to thrashing?

Locality In Memory References

- Working set: set of pages referenced in sampling window
- Not enough memory for working set causes thrashing
- At any sampling window, hit rate is impacted by number of working sets that fit into memory



Working-set Model



- $\Delta \equiv$ sampling window $\equiv$ fixed number of page references
  - Example: 10,000 instructions
- WS_i (working set of p_i) = total set of pages referenced in 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!

Page-fault Rate: Compulsory Misses

- Recall that compulsory misses are misses that occur first time that 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 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 track working set of applications
  - When swapping process back in, swap in working set

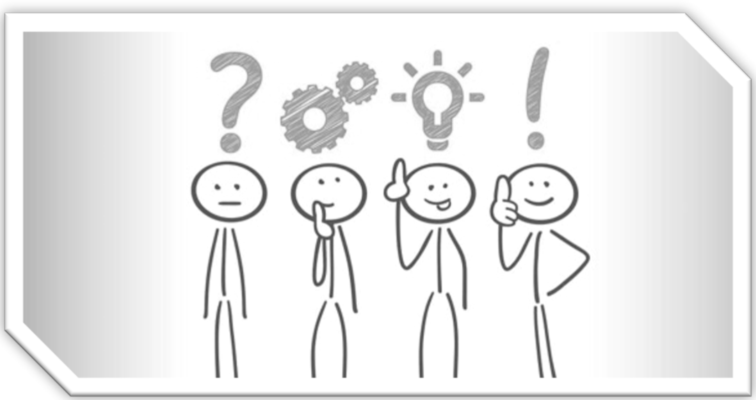
Core-map: Reverse Page Mapping

- Physical page frames often shared by many different address spaces/page tables
  - All children forked from given process
  - Shared memory pages between processes
- Whatever reverse mapping mechanism that is in place must be very fast
  - Must hunt down all page tables pointing at given page frame when freeing a page
  - Must hunt down all PTEs when seeing if pages "active"
- Implementation options:
  - For every page descriptor, keep linked list of page table entries that point to it
    - Management nightmare – expensive
  - Linux 2.6: object-based reverse mapping
    - Link together memory region descriptors instead (much coarser granularity)

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, then can replace
- N^{th}-chance clock algorithm: Another approximate LRU
  - Give pages multiple passes of clock hand before replacing
- Thrashing: 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

Questions?



Acknowledgment

- Slides by courtesy of Anderson, Culler, Stoica, Silberschatz, Joseph, and Canny