Virtual Memory

Outline

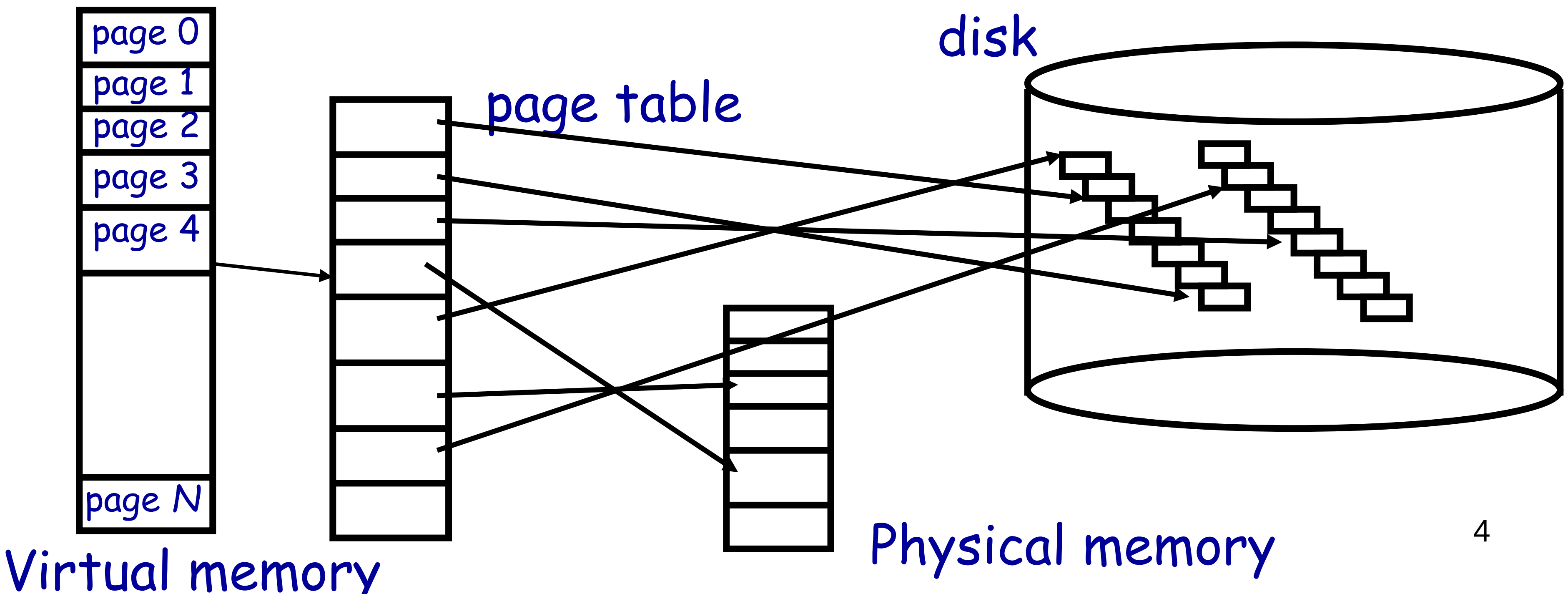
- Background
- Virtual memory
- How does it work?
  - Page faults
  - Resuming after page faults
- When to fetch?
- What to replace?
  - Page replacement algorithms
    - FIFO, OPT, LRU (Clock)
  - Page Buffering
  - Allocating Pages to processes

Background

- Modern programs require a lot of physical memory
- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at same time
  - 90-10 rule: programs spend 90% of their time in 10% of their code
  - Wasteful to require all of user's program code to be in memory
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running -> more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

What is virtual memory?

- Each process has illusion of large address space
  - 2^{32} for 32-bit addressing
- However, physical memory is much smaller
- How do we give this illusion to multiple processes?
  - Virtual Memory: some addresses reside in disk (more precisely, swap space on the disk) and fetched on demand



Virtual Memory

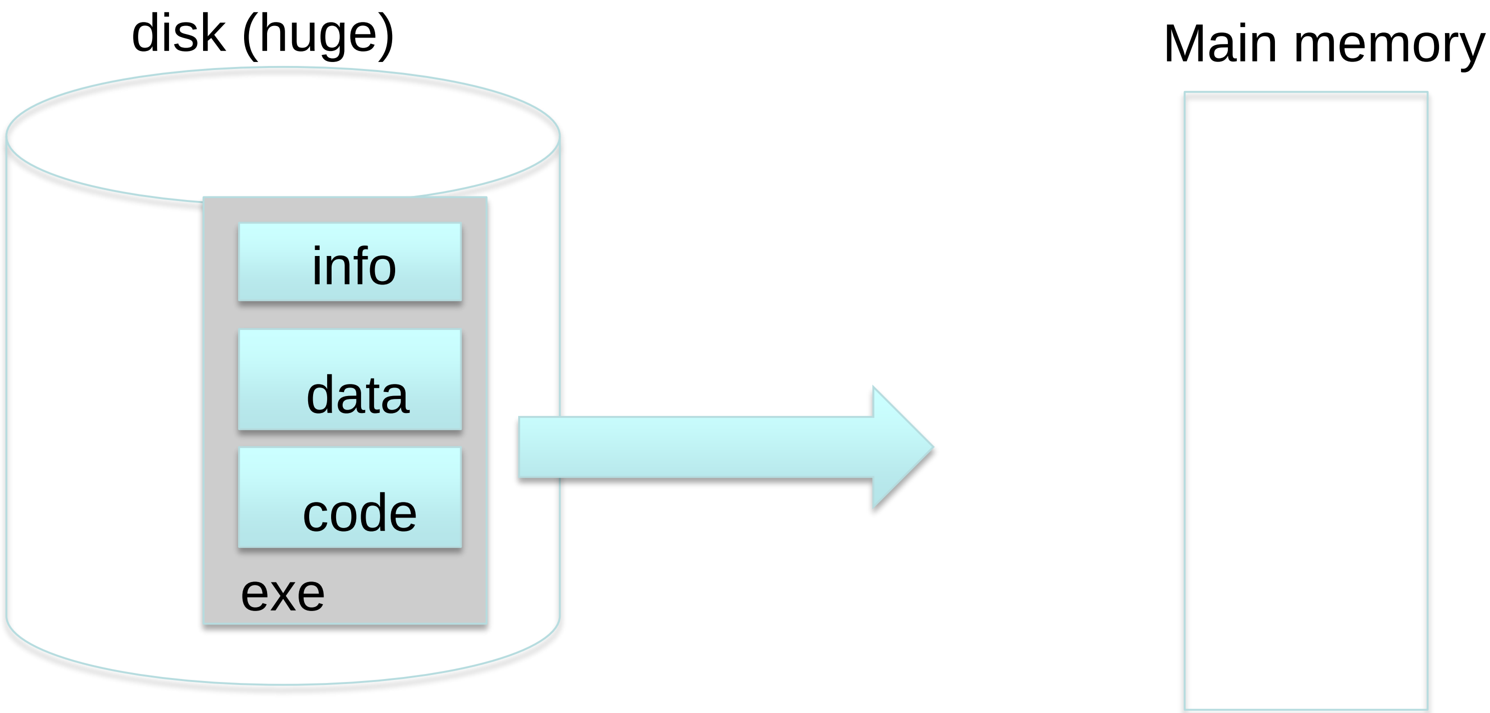
- Separates users logical memory from physical memory.
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation & increases degree of multiprogramming
  - Improves response to the processes (low I/O than in swapping)
  - Better than Dynamic loading where only main program is loaded initially & other routines are loaded on demand with the help of relocatable linking loader

Virtual Memory

- Without Virtual Memory: Load entire process in memory (swapping in), run it, and exit
  - Is slow (for big processes)
  - Wasteful (might not require everything)
- Solutions: partial residency
  - Paging: only bring in pages, not all pages of process
  - Demand paging: bring only pages that are required as and when needed
  - Also Demand Segmentation
- Where to fetch page from?
  - Have a contiguous space in disk: swap space/file

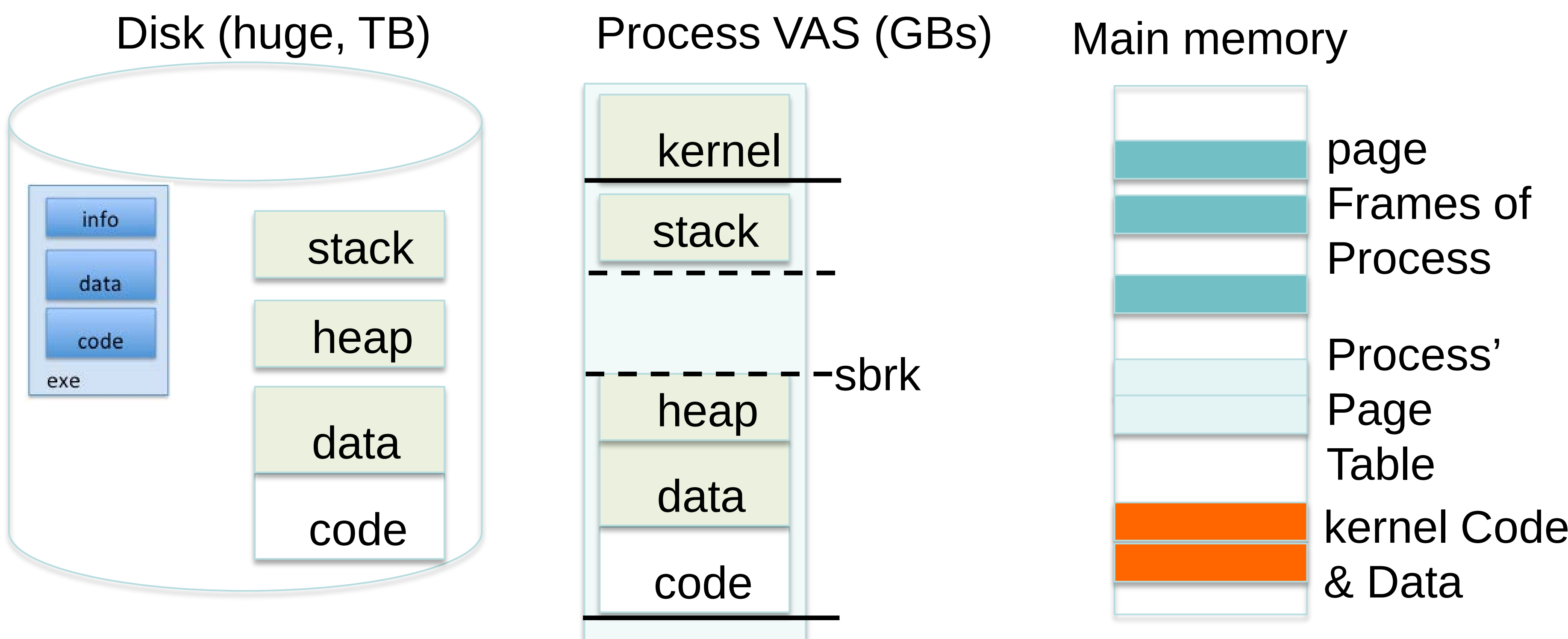
https://opensource.com/article/18/9/swap-space-linux-systems

Loading an executable into memory



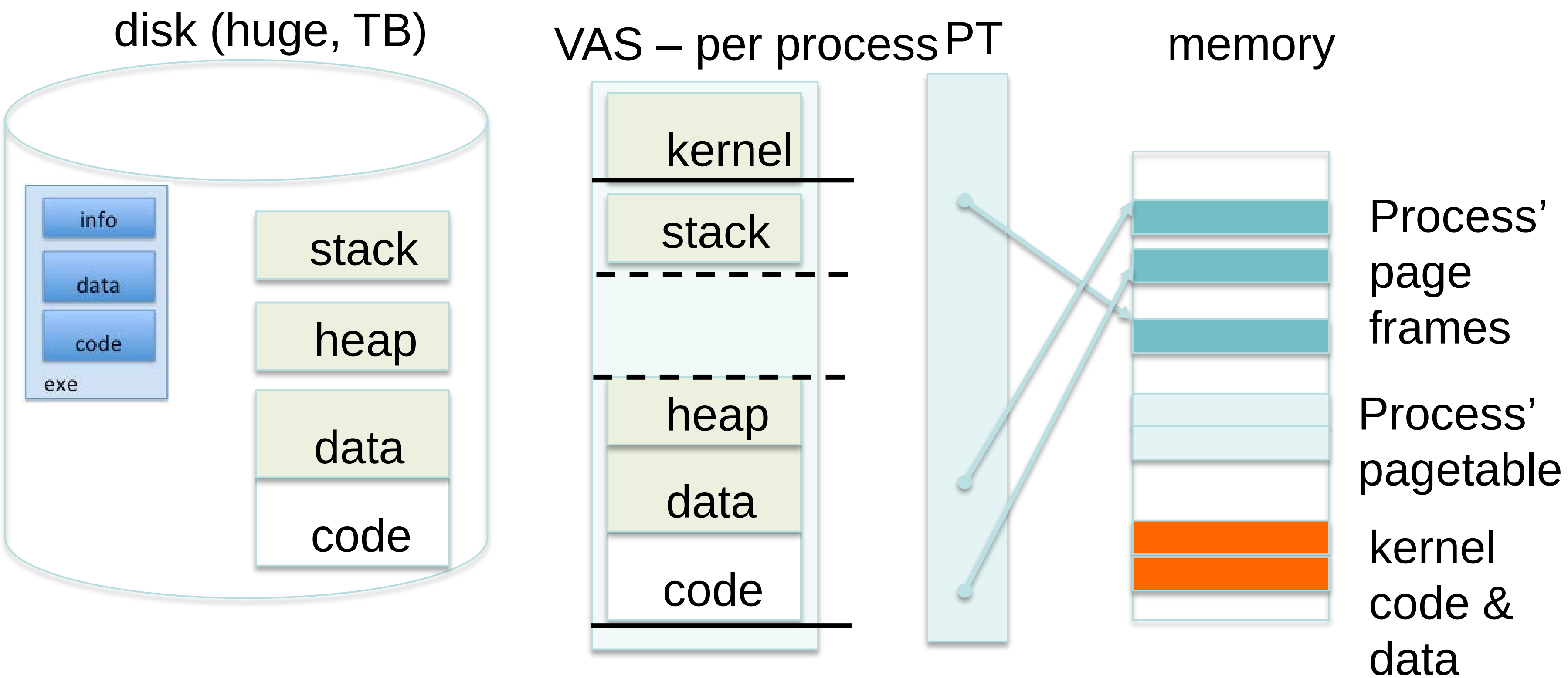
- .exe
  - lives on disk in the file system
  - contains contents of code & data segments, relocation entries and symbols
  - OS loads it into memory, initializes registers (and initial stack pointer)
  - program sets up stack and heap upon initialization:
    [`crt0` (C runtime initialization before Main() is called)]

Create Virtual Address Space of a Process



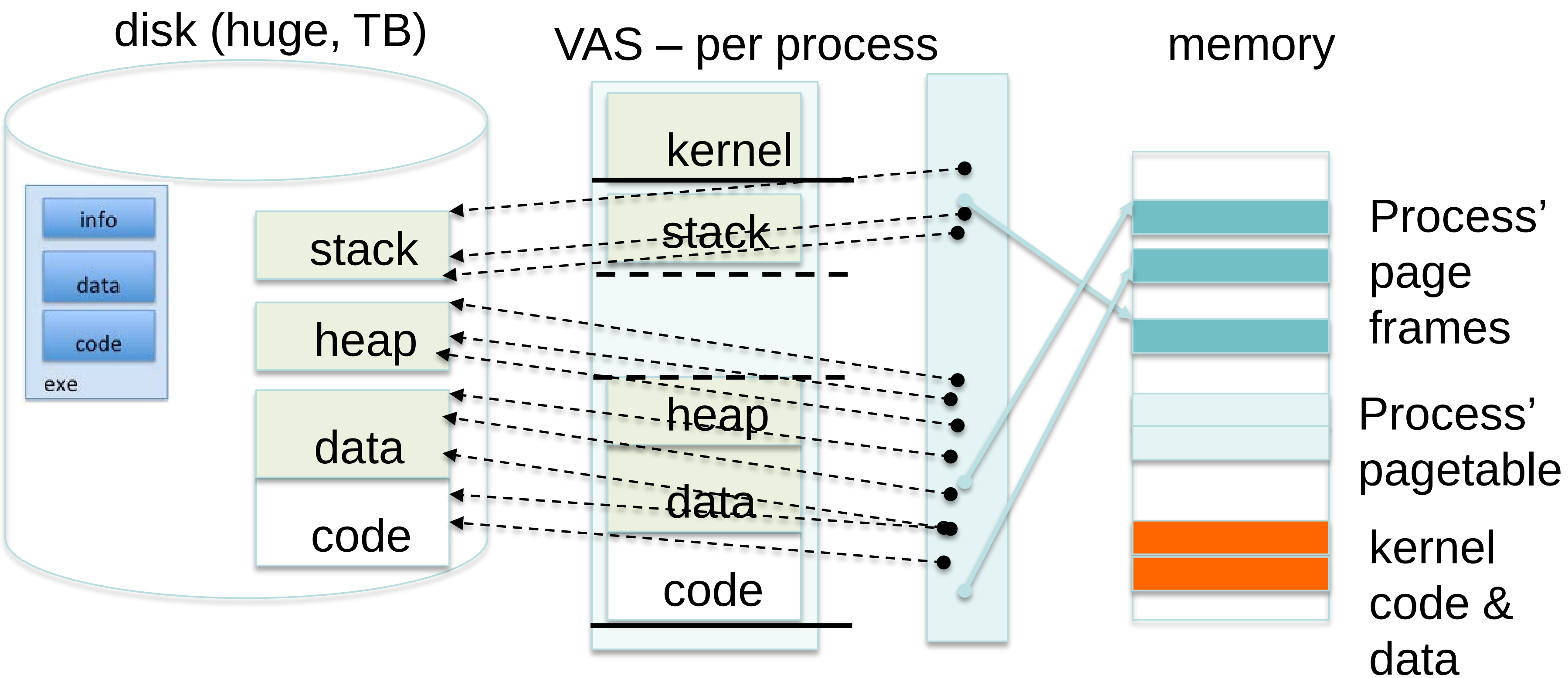
- Kernel mapped into VAS of process, but not accessible to process!
- Utilized pages in the VAS are backed by a page block on disk
  - Called the backing store or swap space/file
  - Swap space I/O faster than file system I/O even if on the same device
    - Swap allocated in larger chunks, less management needed than file system
    - Pages swapped into and out of memory as needed for *every* process

Create Virtual Address Space of a Process



- Process' Page Table maps entire VAS
  - Resident pages to the page frames in main memory they occupy
  - The portion of Page Table that the HW needs to access must be resident in main memory

Provide Backing Store for VAS



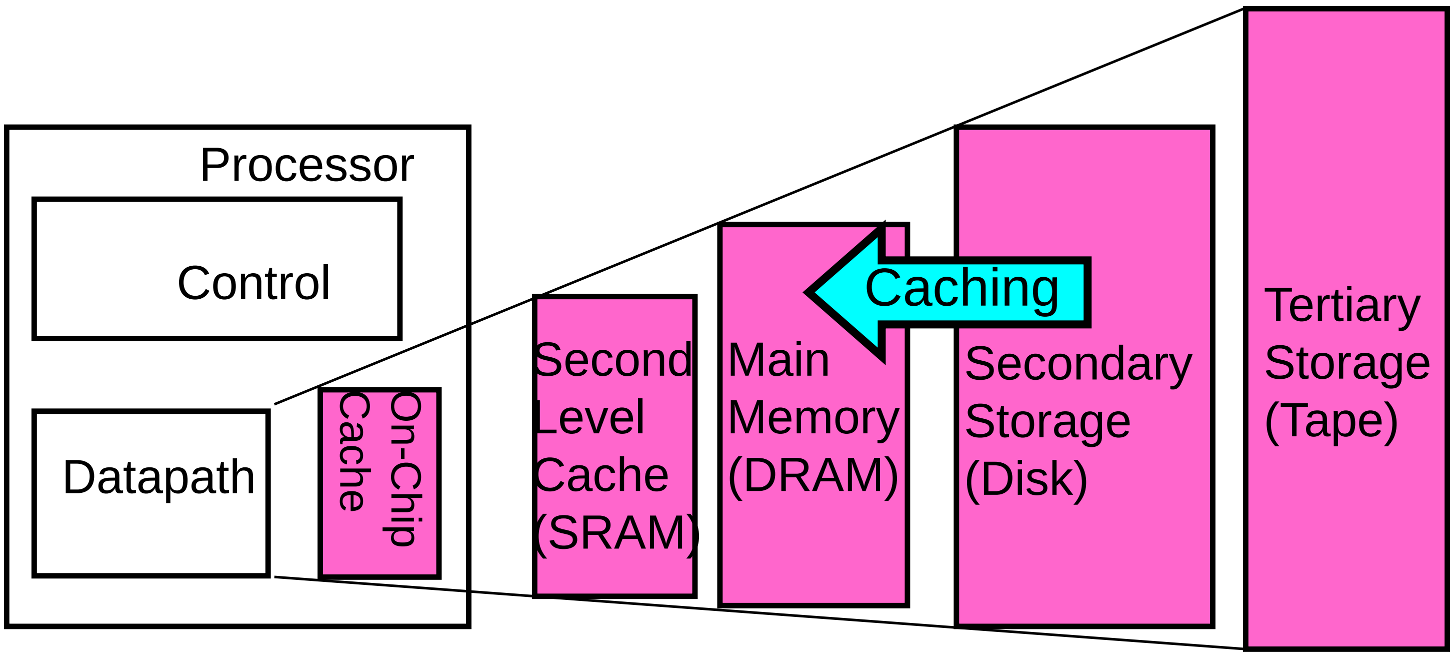
- Process' Page Table maps entire VAS
- Resident pages mapped to page frames
- For all other pages, OS must record where to find them on disk's swap space

What Data Structure Maps Non-Resident Pages to Disk?

- `FindBlock(PID, page#)` $\longrightarrow$ `disk_block`
  - Some OSs utilize spare space in PTE for paged blocks
  - Like the PT, but purely software
- Where to store it?
  - In memory – can be compact representation if swap storage is contiguous on disk
  - Could use hash table (like Inverted PT)
- Usually want backing store for resident pages too
- May map code segment directly to on-disk image
  - Saves a copy of code to swap file
- May share code segment with multiple instances of the program

Demand Paging

- **Lazy swapper** – never swaps a page into memory unless page will be needed
- It uses main memory as cache for disk

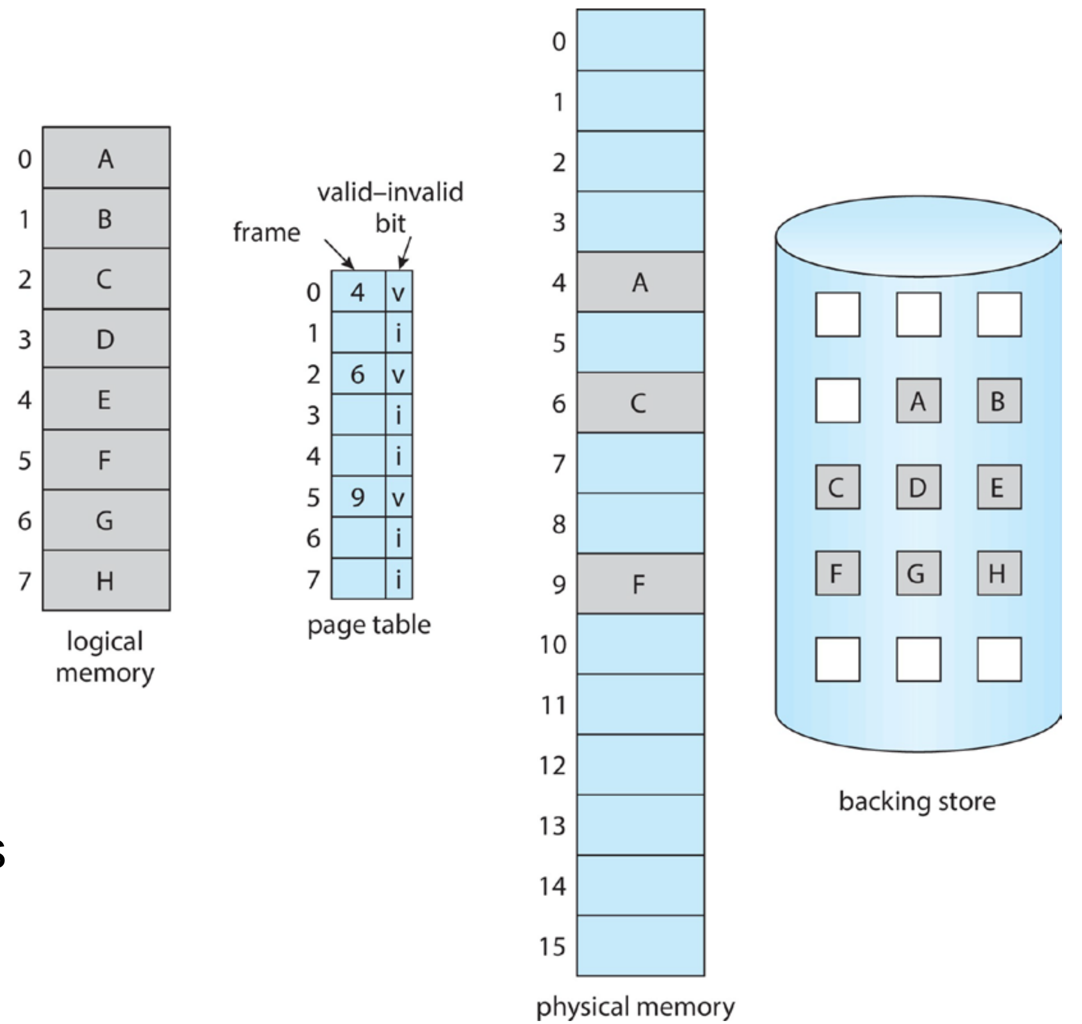


Demand Paging is Caching

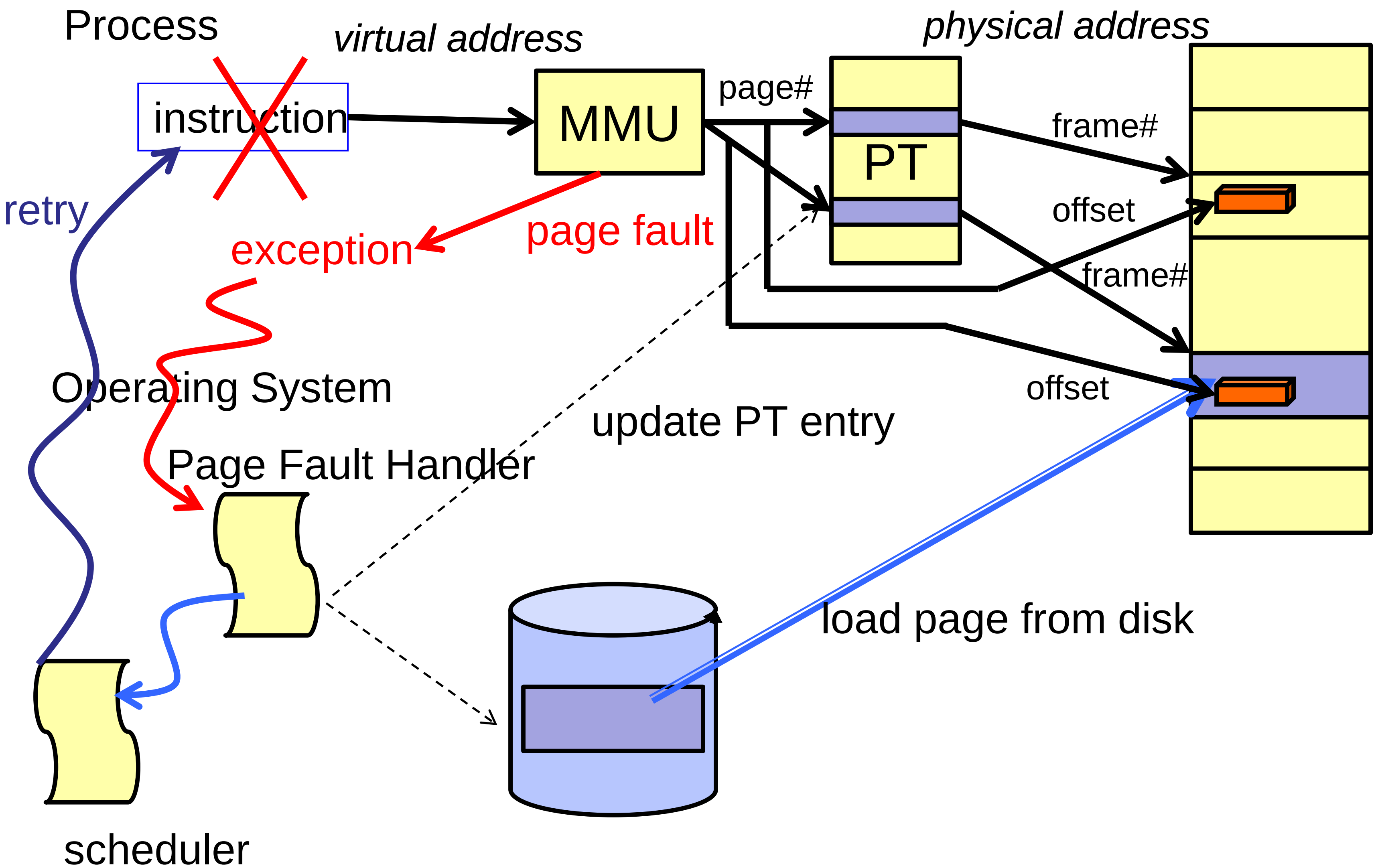
- Since Demand Paging is Caching, must ask:
  - What is block size?
    - 1 page
  - What is organization of this cache (i.e. direct-mapped, set-associative, fully-associative)?
    - Fully associative: arbitrary virtual $\rightarrow$ physical mapping
  - How do we find a page in the cache when look for it?
    - First check TLB, then page-table traversal
  - What is page replacement policy? (i.e. LRU, Random…)
    - This requires more explanation… (kinda LRU)
  - What happens on a miss?
    - Go to lower level to fill miss (i.e. disk)
  - What happens on a write? (write-through, write back)
    - Definitely write-back. Need dirty bit in Page Table entry!

How does Demand Paging work?

- Modify Page Tables with another bit ("valid") in PTE
  - If page in memory, *valid = 1*, else *valid = 0*
  - *Initially valid=0 for all entries in page table*
  - If page is in memory, translation works as before
  - If page is not in memory, MMU traps to OS and resulting trap is called as **page fault**



Steps in Handling a Page Fault



What happens on Page Faults?

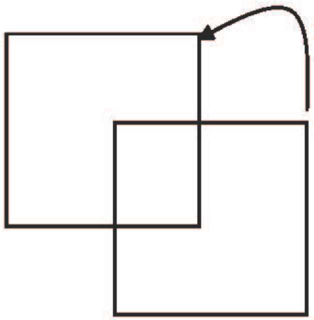
- On a page fault:
  - OS finds a free frame, or evicts one from memory (which one?)
    - Want knowledge of the future?
  - Issues disk request to fetch data for page (what to fetch?)
    - Just the requested page, or more?
  - Block current process (moved to I/O waiting queue), context switch to new process (how?)
    - Process might be in the middle of executing an instruction
  - When disk request completes, set valid bit in page table of blocked process to 1, and move process from I/O waiting queue to CPU Ready Queue

Other Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging**
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Resuming after a page fault

- Should be able to restart the instruction
- For RISC processors this is simple:
  - Instructions are idempotent until references are done
- More complicated for CISC:
  - Block Move: e.g., MVC instruction in IBM 360/370
  - E.g. move 256 bytes from one location to another
  - Possible solutions:
    - Ensure pages are in memory before the instruction executes
    - Use temporary registers to store intermediate (partial) state/results



When to fetch?

- Demand Paging (reactive):
  - Fetch a page when it faults
  - So, incurs page fetching delay
- Prepaging (hybrid):
  - Get the page on fault + some of its neighbors, or
  - Get all pages in use last time process was swapped out

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page from Disk – a lot of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1.0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

$$\begin{aligned} EAT = (1 - p) &\times \text{memory access time} \\ &+ p \times (\text{page fault overhead} + \\ &\quad \text{swap page out} + \text{swap page in} \\ &\quad + \text{restart overhead}) \end{aligned}$$

Demand Paging Example

- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault

  EAT = ?

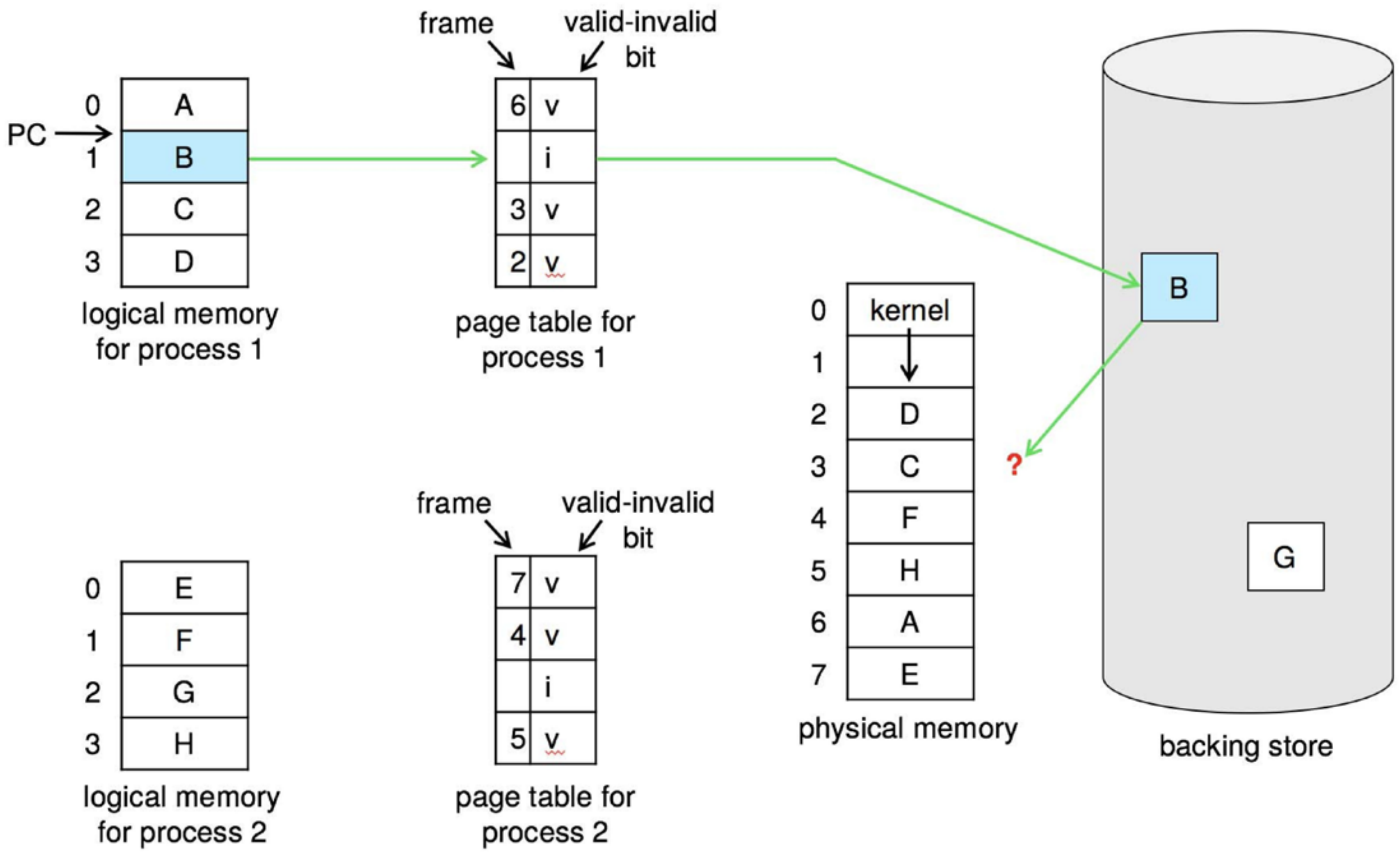
  EAT = 8.2 microseconds

  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

What Factors Lead to Page faults?

- **Compulsory page faults:**
  - 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! ***More later***
- **Capacity page faults:**
  - Not enough memory. Must somehow increase available memory size.
  - Can we do this?
    - One option: Increase amount of DRAM (not quick fix!)
    - Another option: If multiple processes in memory: adjust percentage of memory allocated to each one!
- **Conflict Misses:**
  - Technically, conflict misses don't exist in virtual memory, since it is a "fully-associative" cache
- **Policy Misses:**
  - Caused when pages were in memory, but kicked out prematurely because of the replacement policy
  - How to fix? Need a better replacement policy!

Need for Page Replacement



All memory is in use; then how to address page fault?

What to replace?

- What happens if there is no free frame in memory?
  – find some page in memory, but not really in use, swap it out
- Page Replacement
  – When process has used up all frames it is allowed to use
  – OS must select a page to eject from memory to allow new page
  – The page to eject is selected using the Page Replacement Algorithm

- Goal: Select page that minimizes future page faults

Page Replacement

- Prevent over-allocation of memory to processes by modifying page-fault service routine to include page replacement
- Use **modify (dirty) bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory
  - A large virtual memory can be provided on a smaller physical memory

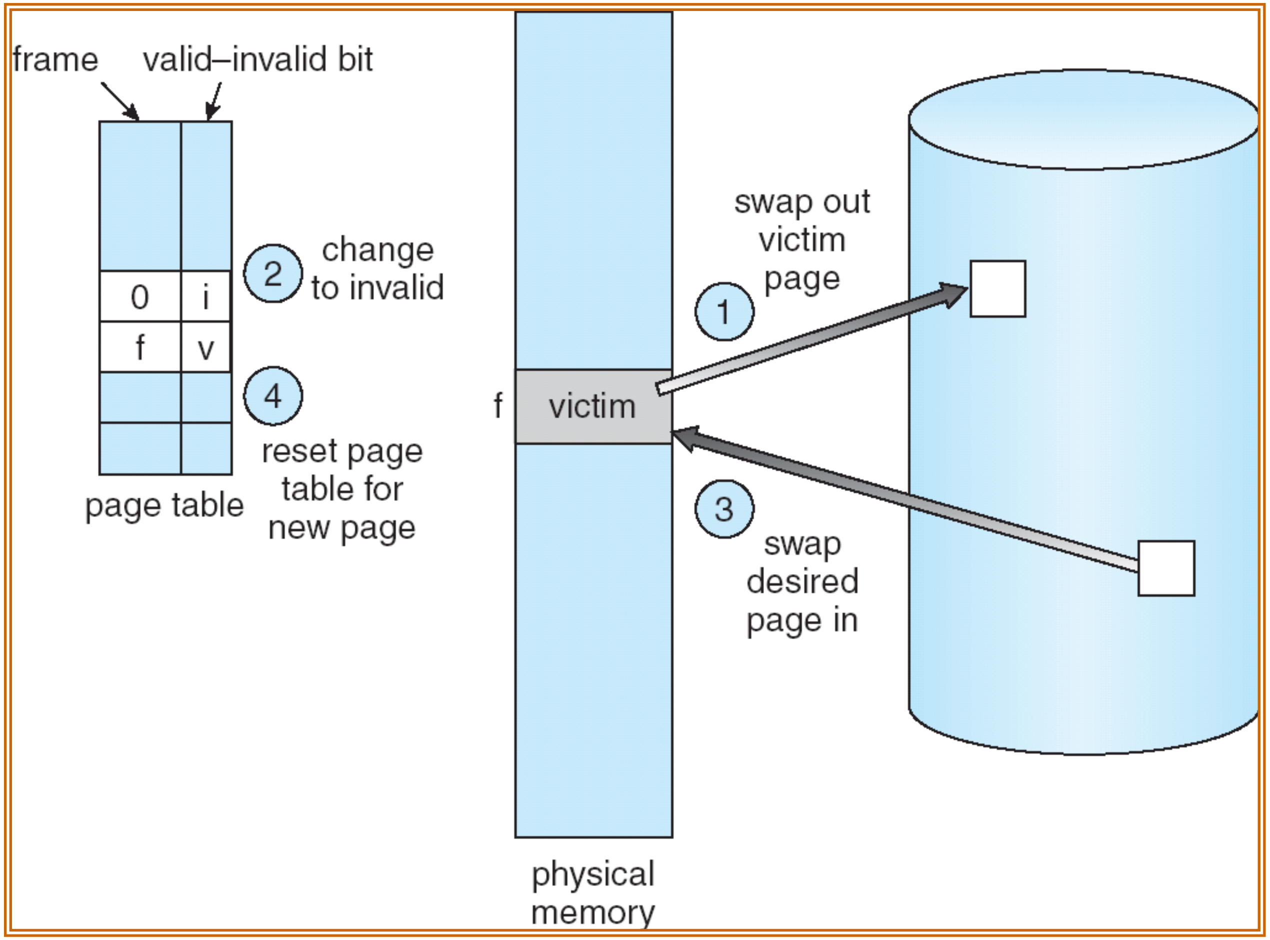
Steps in Page Replacement

1. Find the location of the desired page on disk.
2. Find a free frame:
   - If there is a free frame, use it.
   - If there is no free frame, use a page replacement algorithm to select a *victim* frame.
     - Write victim frame to disk if dirty
3. Read the desired page into the (newly) free frame.
   - Update the page and frame tables.
4. Continue the process by restarting the instruction that caused page fault

→ Potentially 2 page transfers for single page fault

- Increases EAT

Page Replacement



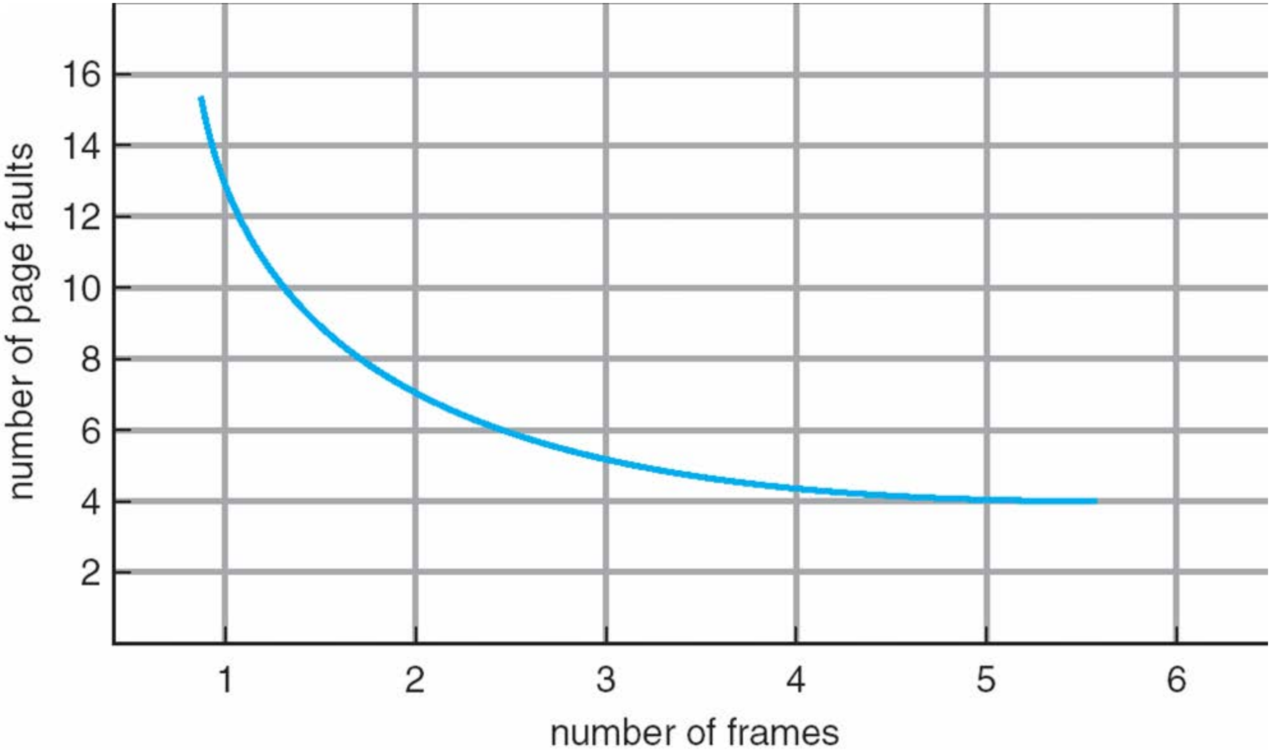
Assumption: Victim page and new page belong to the same process

Page Replacement Algorithms

- Want lowest page-fault rate
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string.
- How to get reference strings?
  – Extract Page numbers from logical addresses generated by CPU
  – Remove successive references to same page from the string as they do not cause page faults
    - Assumption: No context-switch!
- Page faults also depend on number of frames available
- In all our following examples, the reference string is

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

Page Faults Vs Number of Frames Allocated per Process



Note: No. of page faults saturates but does not reach Zero

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames (3 pages can be in memory at a time per process)

	1,	2,	3,	4,	1,	2,	5,	1,	2,	3,	4,	5
Frame 1	1	1	1	4	4	4	5	5	5	5	5	5
Frame 2		2	2	2	1	1	1	1	1	3	3	3
Frame 3			3	3	3	2	2	2	2	2	4	4

First-In-First-Out (FIFO) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
- 3 frames → 9 page faults

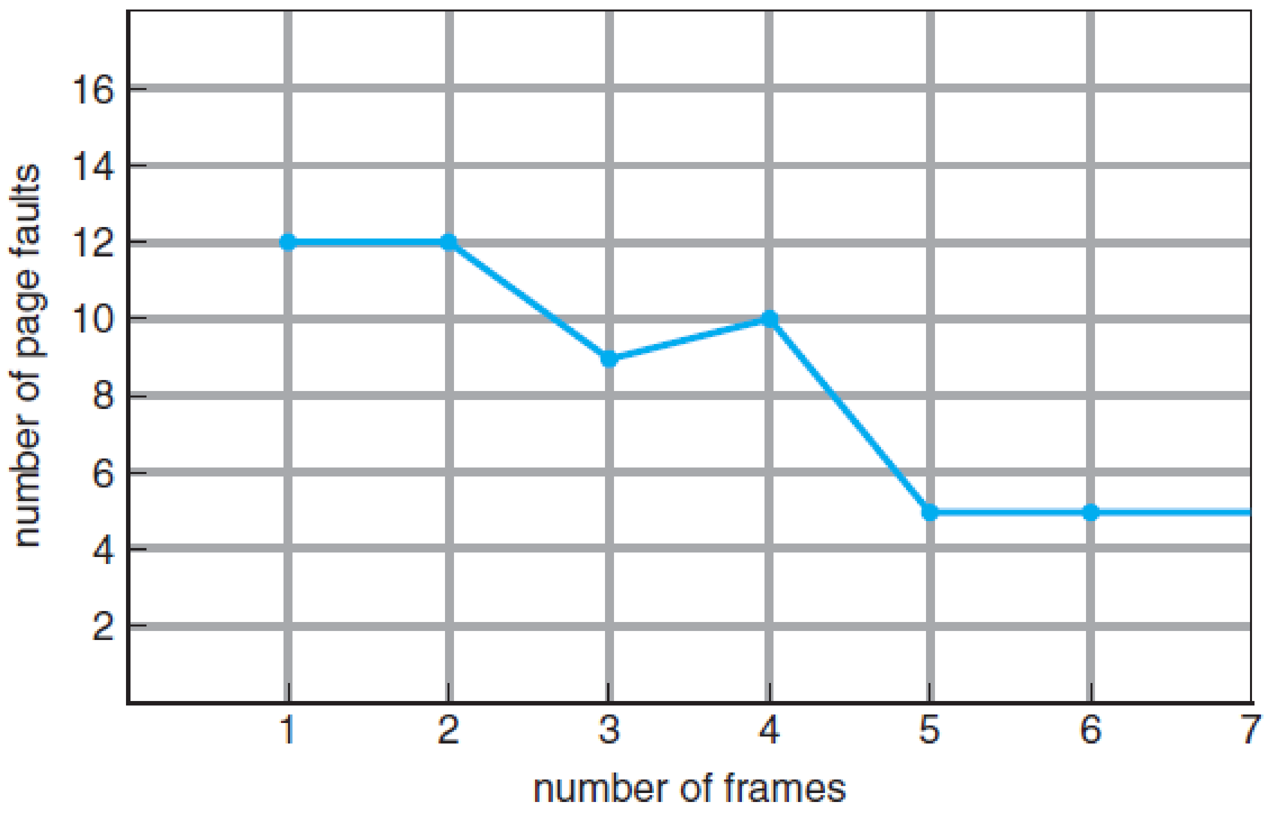
- 4 frames → How many page faults?

1	1	5	4	
2	2	1	5	10 page faults
3	3	2		
4	4	3		

FIFO Replacement – Belady's Anomaly

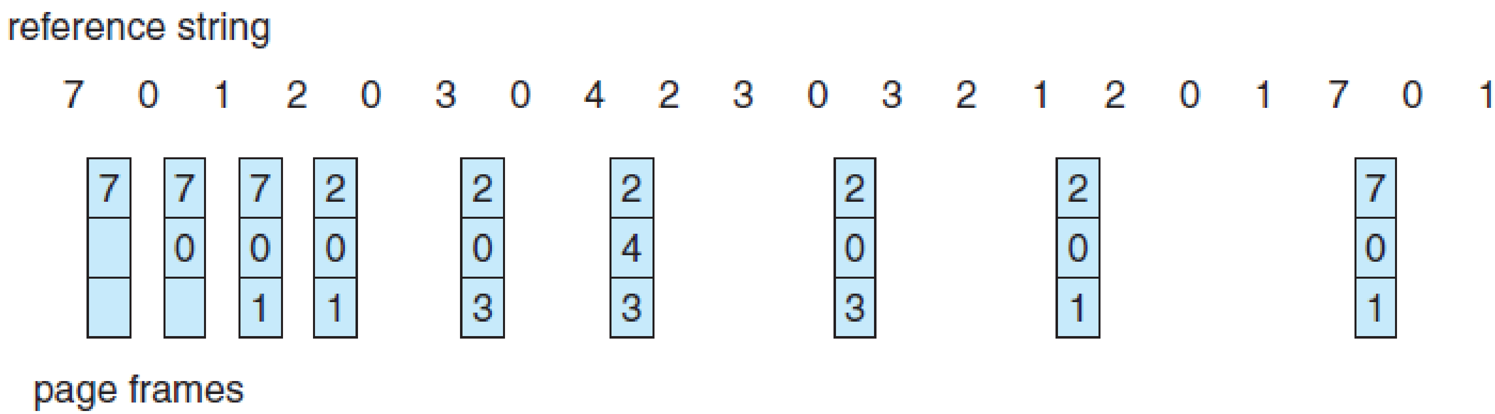
– more frames ⇒ more page faults!

FIFO Illustrating Belady’s Anamoly



Optimal Algorithm (OPT)

- Replace page that will not be used for longest period of time.



- How do you know this?
  - Requires future knowledge of page references
  - So, we can’t implement it directly
- Used for measuring how well your algorithm performs.

Optimal Algorithm (OPT)

- 4 frames example

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

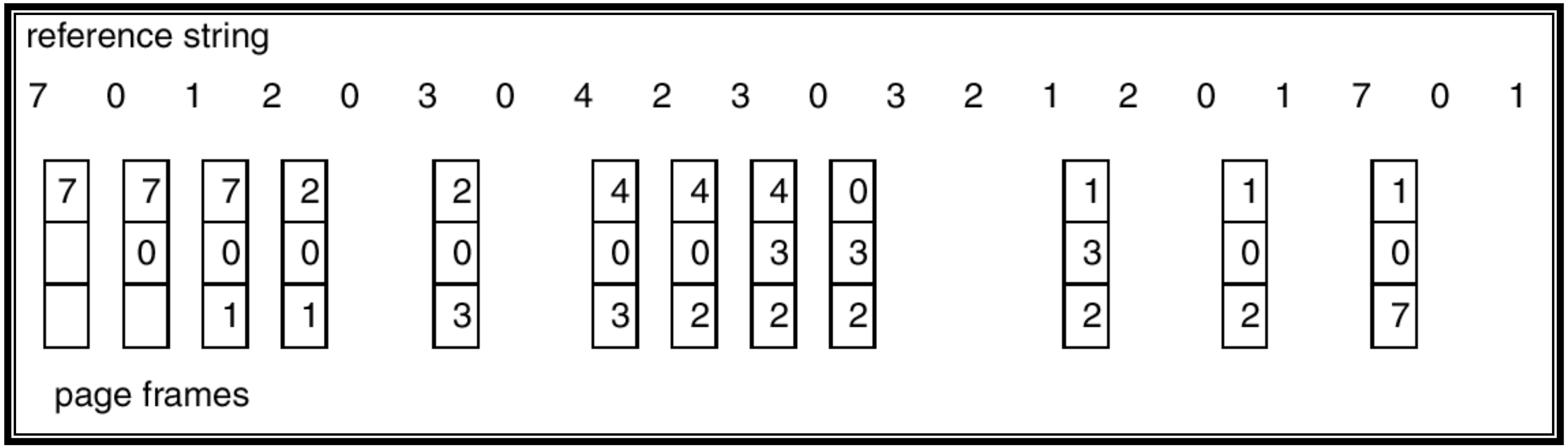
1	4
2	
3	
4	5

6 page faults

Least Recently Used (LRU) Algorithm

- FIFO uses the time when page was brought into memory
- OPT uses the time when page is to be used in future
- Recent past as an approximation of the near future
  - Replace page that has not been used for the longest period of time (LRU)
- LRU: OPT looking backward in time
- Page fault rate for OPT on S (reference string) is same as that for OPT on S^R
- Similarly, page fault rate for LRU on S (reference string) is same as that for LRU on S^R

LRU Page Replacement



Least Recently Used (LRU) Algorithm

- Reference string: 1, 2, 3, 4, 1, 2, **5**, 1, 2, **3**, **4**, **5**

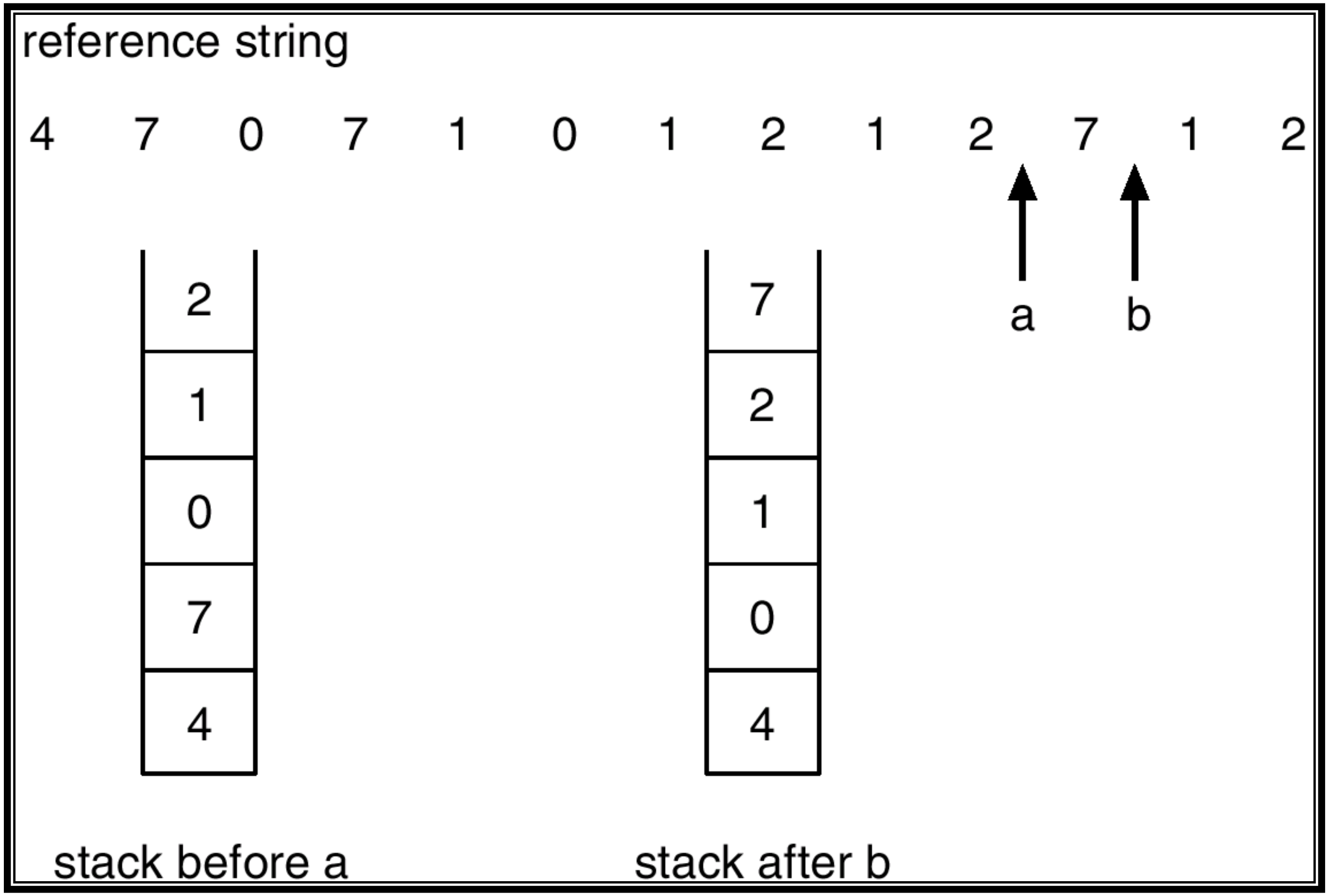
1	1	1	1	**5**
2	2	2	2	2
3	**5**	5	**4**	4
4	4	**3**	3	3

- 8 page faults vs 10 in FIFO
- Initial 4 page faults are ignored during comparison of page replacement algorithms
  - All algorithms including OPT suffer the initial page faults!

LRU Algorithm: Implementation

- Counter implementation
  - Every page table entry (PTE) has a *time-of-use* field
    - Every time a page is referenced, copy the contents of CPU clock register into the time-of-use field of its PTE.
  - Victim page: look at the time-of-field entries to determine page with the smallest time value for replacement.
  - Requires search in page table and updates to fields on each reference; Clocks may overflow
- Stack implementation – keep a stack of page numbers in a doubly linked list data structure:
  - Page referenced:
    - move it to the top of the stack
    - requires 6 pointers to be changed
  - No search for replacement as the bottom one is the LRU page

Use of a Stack to record the Most Recent Page References



LRU Page Replacement

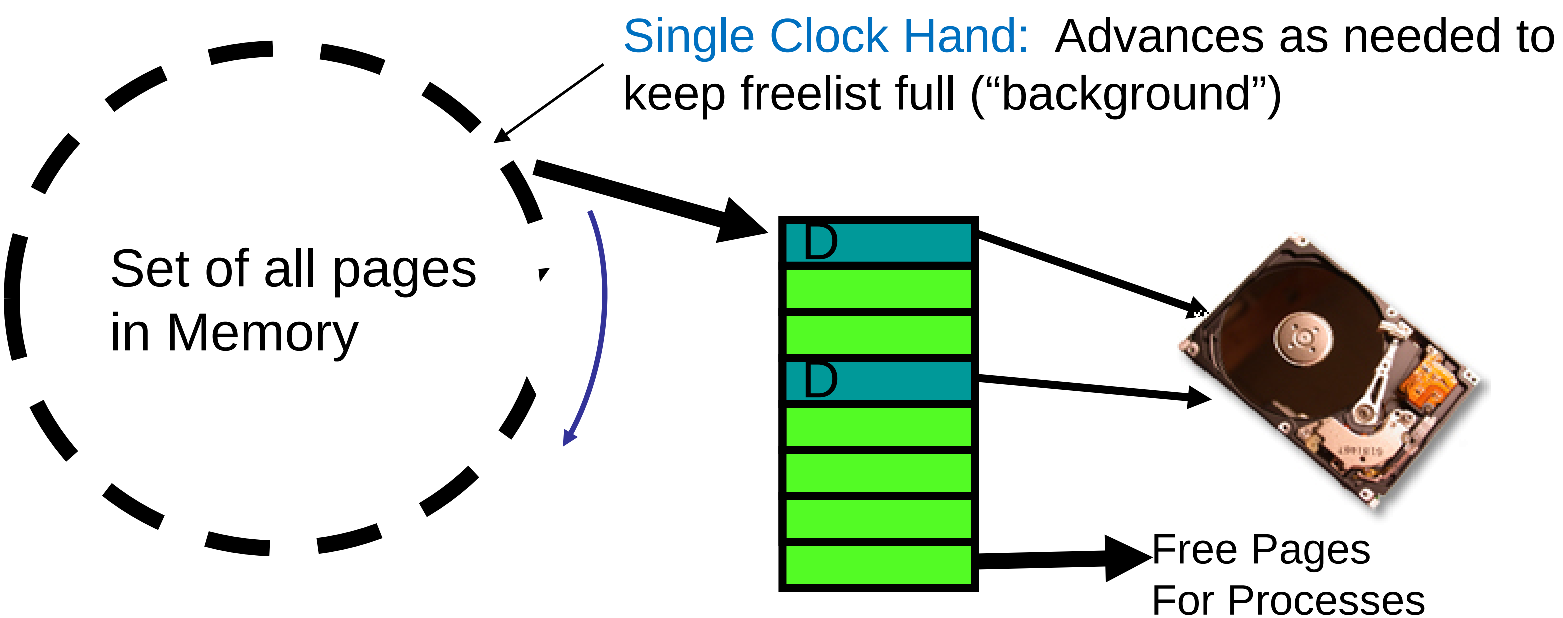
- Like OPT, LRU does not suffer from Belady's anomaly
- Both belong to class of algorithms called Stack algos
- Stack Algo: Set of Pages in memory for n frames is always subset of Set of Pages that would be in memory for n+1 frames
- Both Counter and Stack implementations of LRU incur huge overhead
  - Every memory reference lead to updating of Clock fields or Stack
  - So, we need LRU approximation algos to cut-down the cost!!

LRU Approximation Algorithms

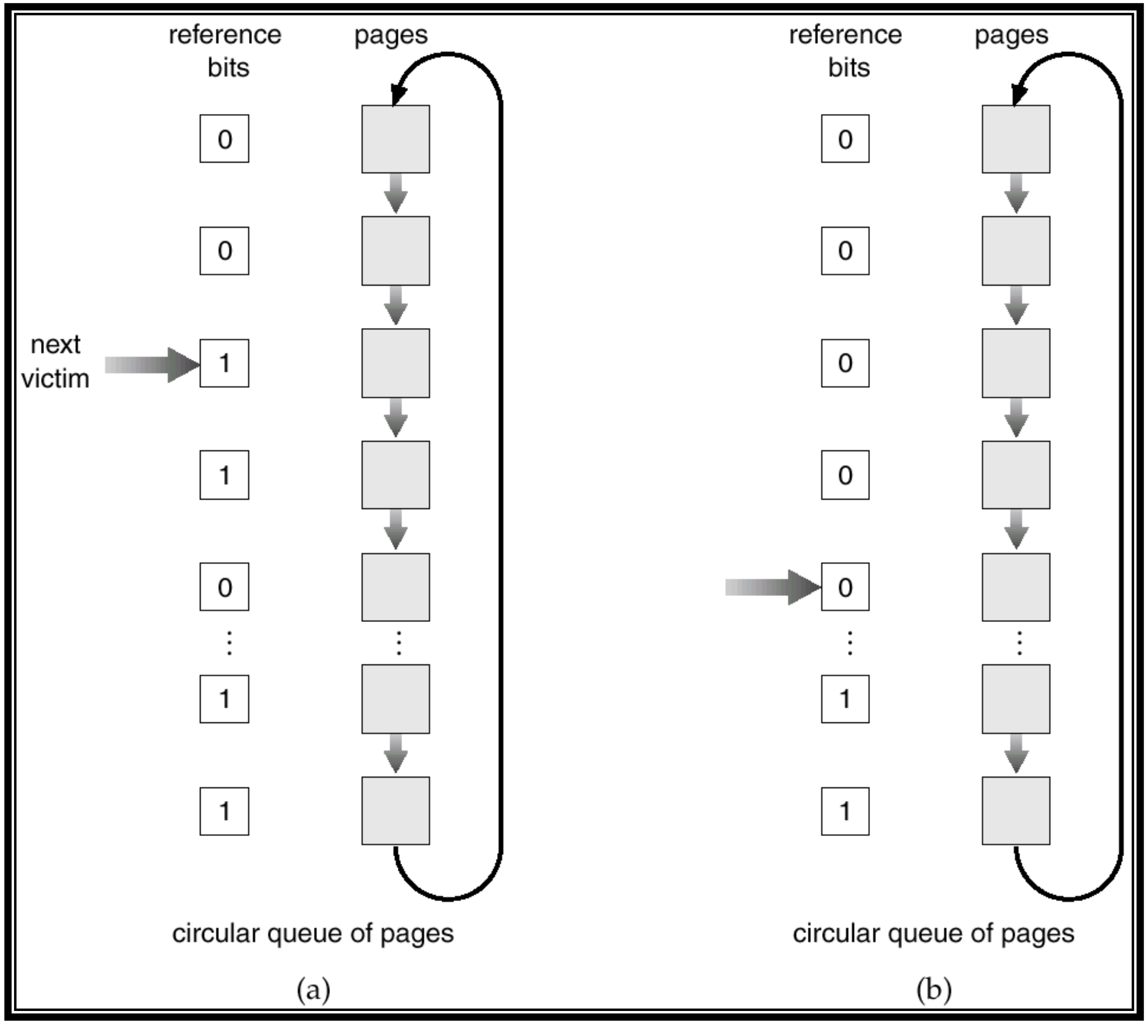
- Reference bit
  - With each page associate a bit in its PTE
    - Initially = 0 for all pages
  - When page is referenced, bit is set to 1 with the hardware support.
  - Replace the one which is 0 (if one exists).
    - We do not know the order, however.
- Extension: (Additional-Reference-Bits Algo)
  - 8-bit shift register per page to keep track of its historical use
  - Periodically, copy reference bit content to the shift register and reset it to 0
  - Replace the one which is having the lowest value in shift register

LRU Approximation Algorithms

- Second chance algorithm
  - A variant of FIFO that uses just the reference bit in PTE.
  - Also called Clock algorithm
  - If page to be replaced (in clock order using FIFO) has reference bit = 1. Then:
    - set reference bit 0.
    - leave page in memory.
    - replace next page (in clock order), subject to same rules.



Second-Chance (clock) Page-Replacement Algorithm



Sensitive to sweeping interval

- Fast: expensive
- Slow: all pages look used

Page Buffering Algorithms

- Keep a pool of free frames, always
  - Then frame available when needed, not found at fault time
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected
    - E.g., FIFO

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available in PTE) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes to replace a page in the lowest non-empty class
  - Might need to search circular queue several times

Counting Algorithms

- Keep a counter of the number of references that have been made to each page.
- LFU: Remove page with lowest count
  - No track of when the page was referenced
  - Use multiple bits. Shift right by 1 at regular intervals for Decaying
- MFU Algorithm: based on the argument that the page with the smallest count was probably just brought in and has yet to be used.
- LFU and MFU do not approximate OPT well and are expensive
  - So not commonly used

Allocation of Frames

- Each process needs **minimum** number of page frames
  - 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**.
- **min** is defined by computer arch
- **max** is defined by DRAM size
- Two major allocation schemes.
  - fixed allocation
  - priority allocation

Fixed Allocation

- Equal allocation – e.g., if 100 frames and 5 processes, give each 20 pages.
- Proportional allocation – Allocate according to the size of process.
  - $s_i = \text{size of process } p_i$
  - $S = \sum s_i$
  - $m = \text{total number of frames}$
  - $a_i = \text{allocation for } p_i = \frac{s_i}{S} \times m$

$$m = 64$$

$$s_i = 10$$

$$s_2 = 127$$

$$a_1 = \frac{10}{137} \times 64 \approx 5$$

$$a_2 = \frac{127}{137} \times 64 \approx 59$$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size.

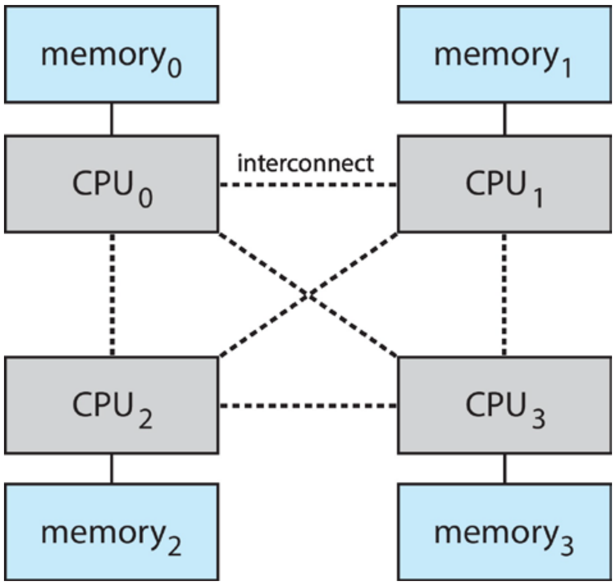
- If process P_i generates a page fault,
  - select for replacement one of its frames.
  - select for replacement a frame from a process with lower priority number.

Global vs. Local Allocation

- **Global** replacement – process selects a replacement frame from the 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.
- Global vs Local:
  - Global: page fault rate is no longer under a process' control
  - Local: More consistent per-process performance, but possibly underutilized memory
  - But Global gives better system throughput, degree of multiprogramming

Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
  - Consider system boards containing CPUs and memory, interconnected over a system bus
- NUMA multiprocessing architecture
  - Optimal performance comes from allocating memory "close to" the CPU on which the process is scheduled
  - What about multi-threaded programs?



Summary

- Demand Paging:
  - Treat main memory as cache of disk/backing Store
  - Cache miss → get page from backing Store
- Transparent Level of Indirection
  - User program is unaware of activities of OS behind scenes
  - Data can be moved without affecting application correctness
- Replacement policies
  - FIFO: Place pages on queue, replace page at end
  - OPT: replace page that will be used farthest in future
  - LRU: Replace page that hasn’t be used for the longest time
- 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