OPERATING SYSTEMS

Memory Management

Likitha P

Department of Computer Science

OPERATING SYSTEMS

Virtual Memory

Likitha P

Department of Computer Science

Slides Credits for all PPTs of this course

- The slides/diagrams in this course are an **adaptation**, **combination**, and **enhancement** of material from the following resources and persons:

1. Prof. Venkatesh Prasad, Department of CSE, PES University.
2. Prof. Chandravva Hebbi, Department of CSE, PES University.
3. Slides of Operating System Concepts, Abraham Silberschatz, Peter Baer Galvin, Greg Gagne - 9th edition 2013 and some slides from 10th edition 2018.
4. Some conceptual text and diagram from Operating Systems - Internals and Design Principles, William Stallings, 9th edition 2018.
5. Some presentation transcripts from A. Frank – P. Weisberg.
6. Some conceptual text from Operating Systems: Three Easy Pieces, Remzi Arpaci-Dusseau, Andrea Arpaci Dusseau.

Background

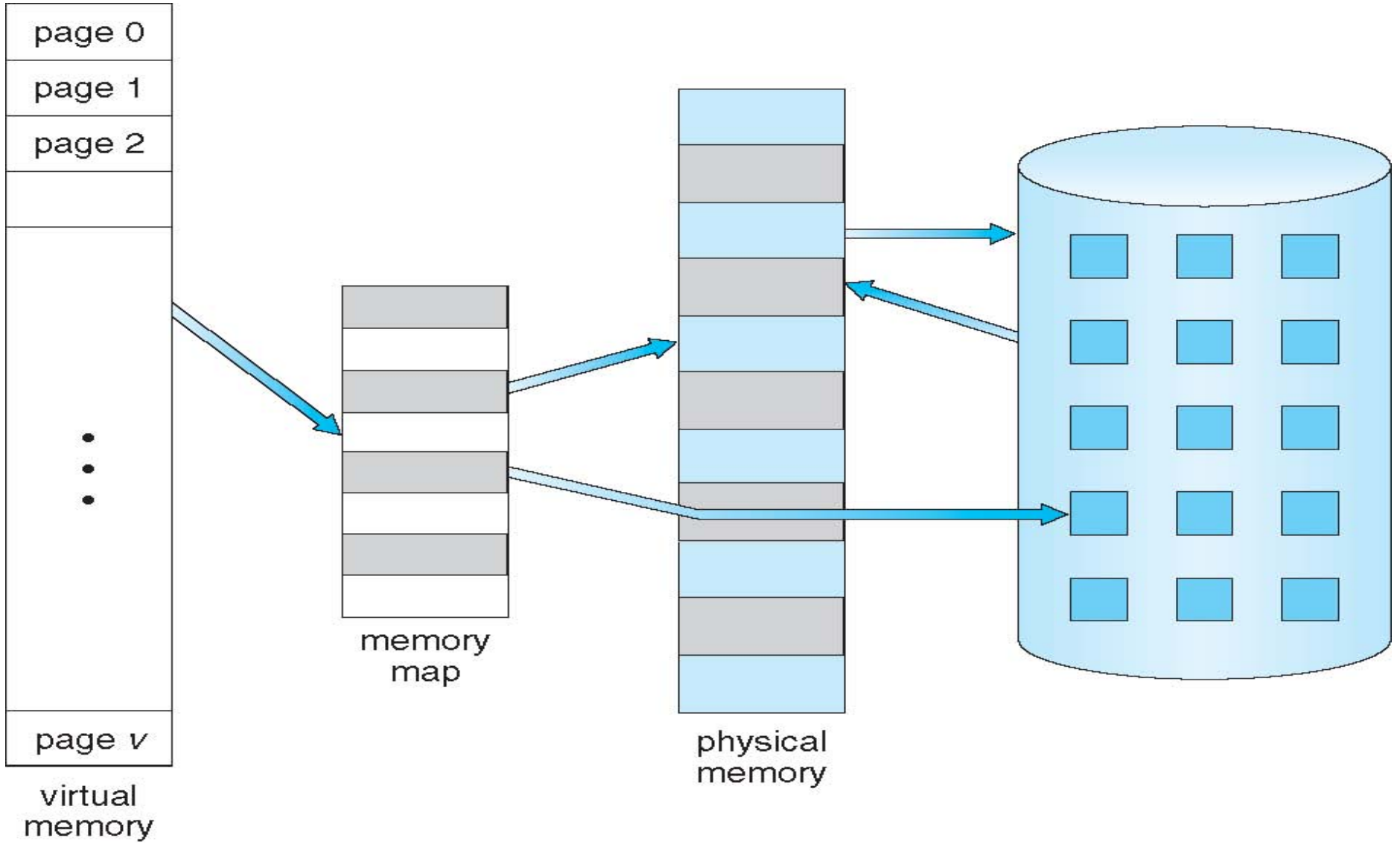
- So far the approach was to place entire logical address space in physical memory.
- However entire program is rarely used. Consider Error handling code, unusual routines, large data structures, etc.
- Thus, entire program code is not needed in the main memory at a time.
- Consider the ability to execute a partially-loaded program
  - Program size is no longer constrained by limits of physical memory.
  - Each program takes lesser 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.

Background (Cont.)

- **Virtual memory** – separation of user logical memory from physical memory. This allows extremely large virtual memory be provided to programmers when only a smaller physical memory is available.
  - 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.
  - More programs can run concurrently.
  - Less I/O needed to load or swap processes.

OPERATING SYSTEMS

Virtual Memory That is Larger Than Physical Memory

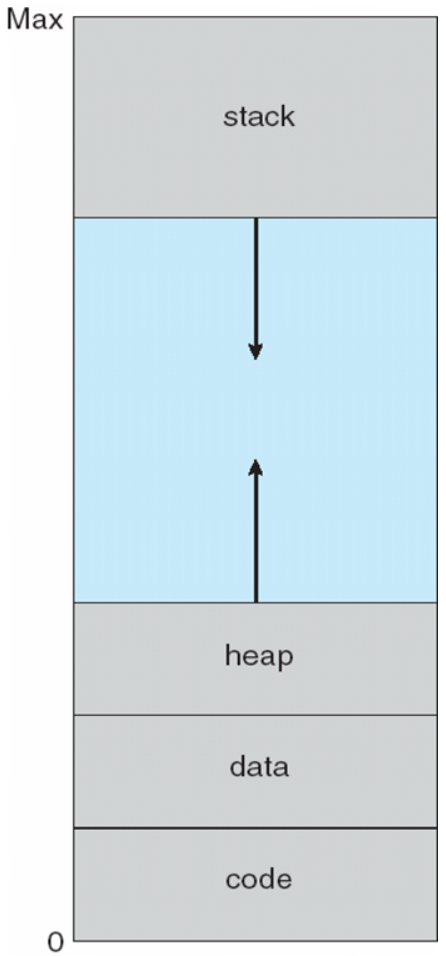


Background (Cont.)

- **Virtual address space** – logical view of how process is stored in memory.
  - Usually start at address 0, contiguous addresses until end of space.
  - Meanwhile, physical memory organized in page frames.
  - MMU must map logical to physical.
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

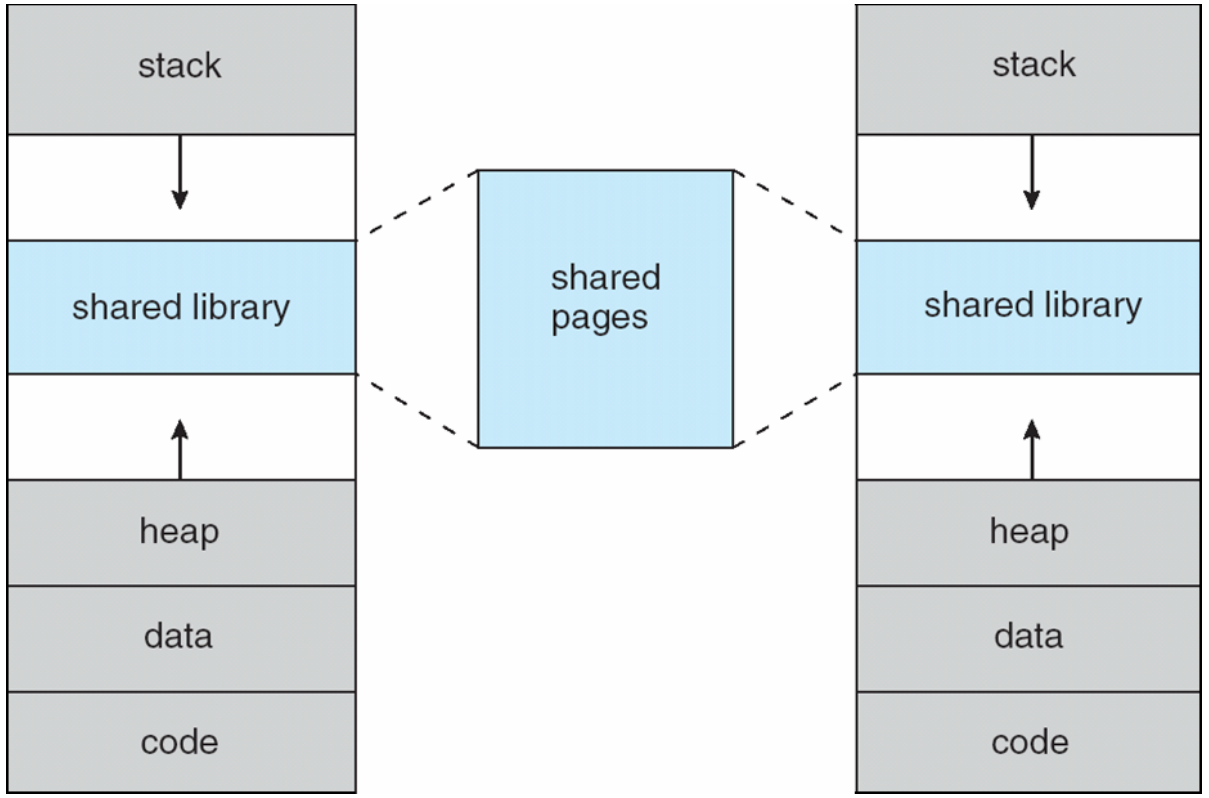
OPERATING SYSTEMS
Virtual-address Space

- Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Heap is used for dynamic memory allocation and stack for successive function calls.
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space.
- Shared memory by mapping pages read-write into virtual address space.
- Pages can be shared during fork(), speeding process creation.



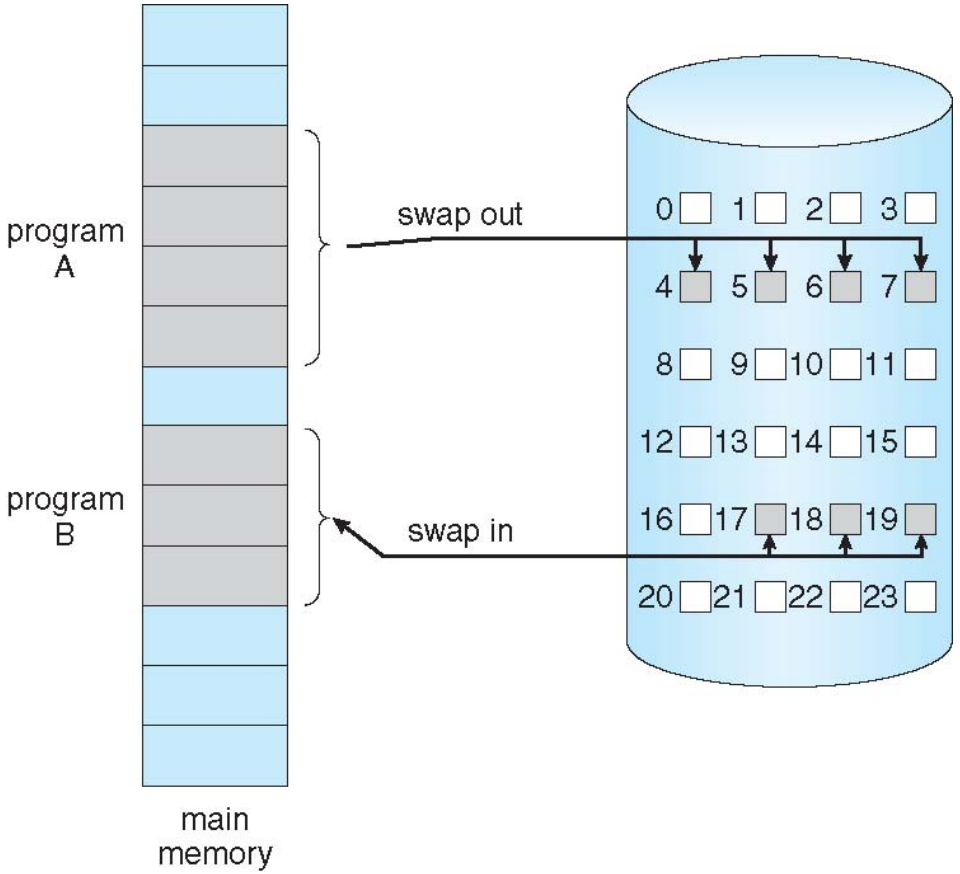
OPERATING SYSTEMS
Shared Library Using Virtual Memory

- In addition to allowing a separation of logical and physical memory, virtual memory allows sharing of files and memory.
- Benefits of sharing:
  - System libraries are shared via mapping into virtual address space. Typically mapped in read-only format.
  - Processes can share memory by mapping pages.
  - Pages can be shared during fork(), speeding process creation.



Virtual Memory Implementation : Demand Paging

- Similar to a paging system with swapping.
- One can bring entire process into memory at load time.
- Or bring a page into memory only when it is needed
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page is needed.
  - Swapper that deals with pages is a **pager.**

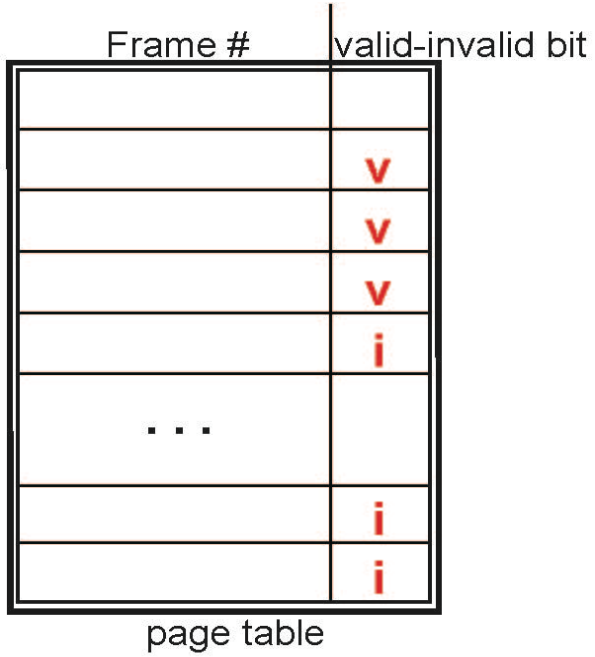


Demand Paging – Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again.
- Instead of entire process, pager brings into memory only those pages which will be used before swap out.
- How to determine that set of pages?
  - Need new MMU functionality to implement demand paging.
- If pages needed are already **memory resident**
  - No difference from non demand-paging
- If page needed is not memory resident
  - Need to detect and load the page into memory from storage
    - Without changing program behavior
    - Without programmer needing to change code

Valid-Invalid Bit

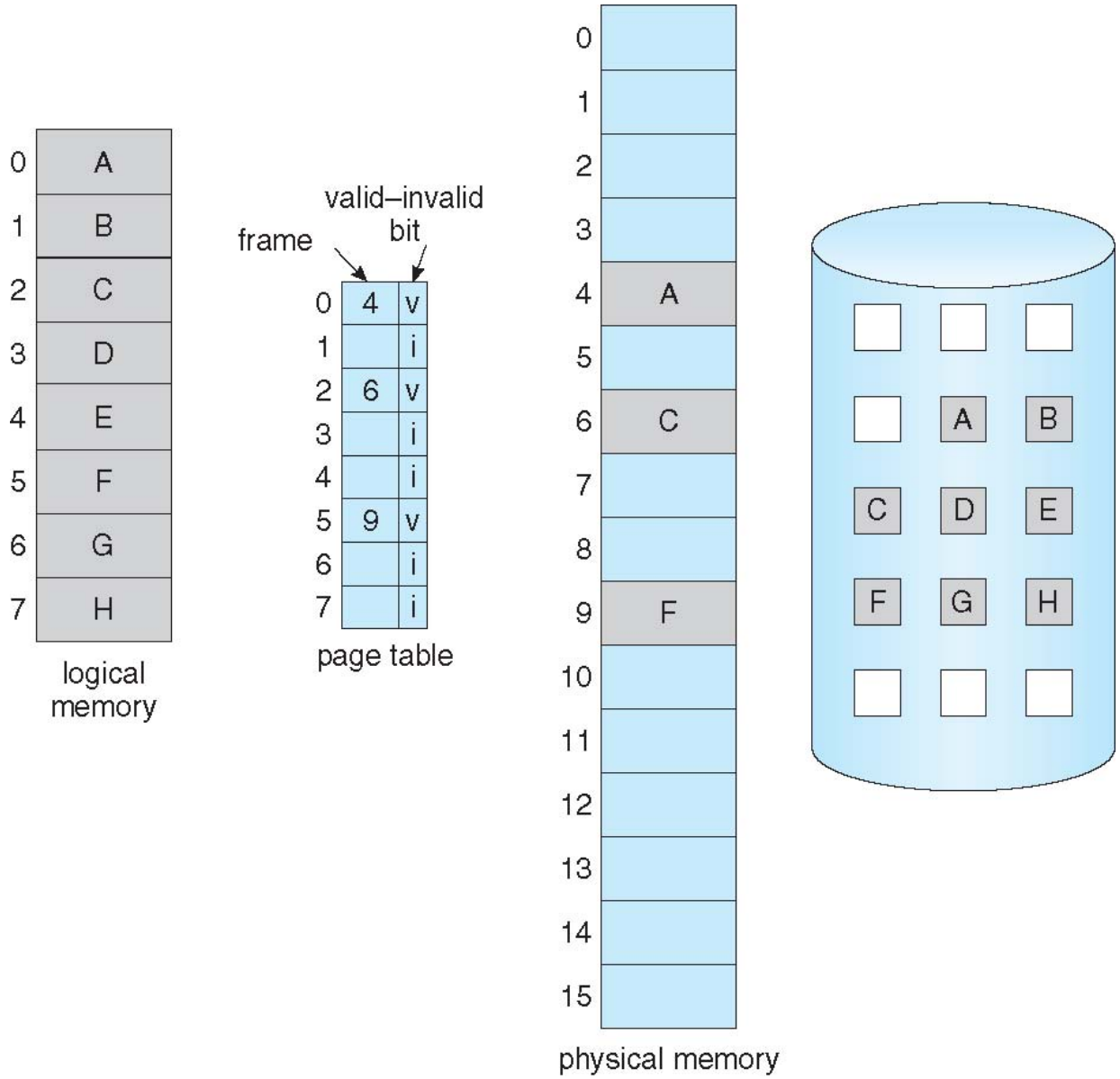
- With each page table entry a valid–invalid bit is associated:
  (**v** ⇒ in memory – **memory resident**, **i** ⇒ not in memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table:



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

OPERATING SYSTEMS

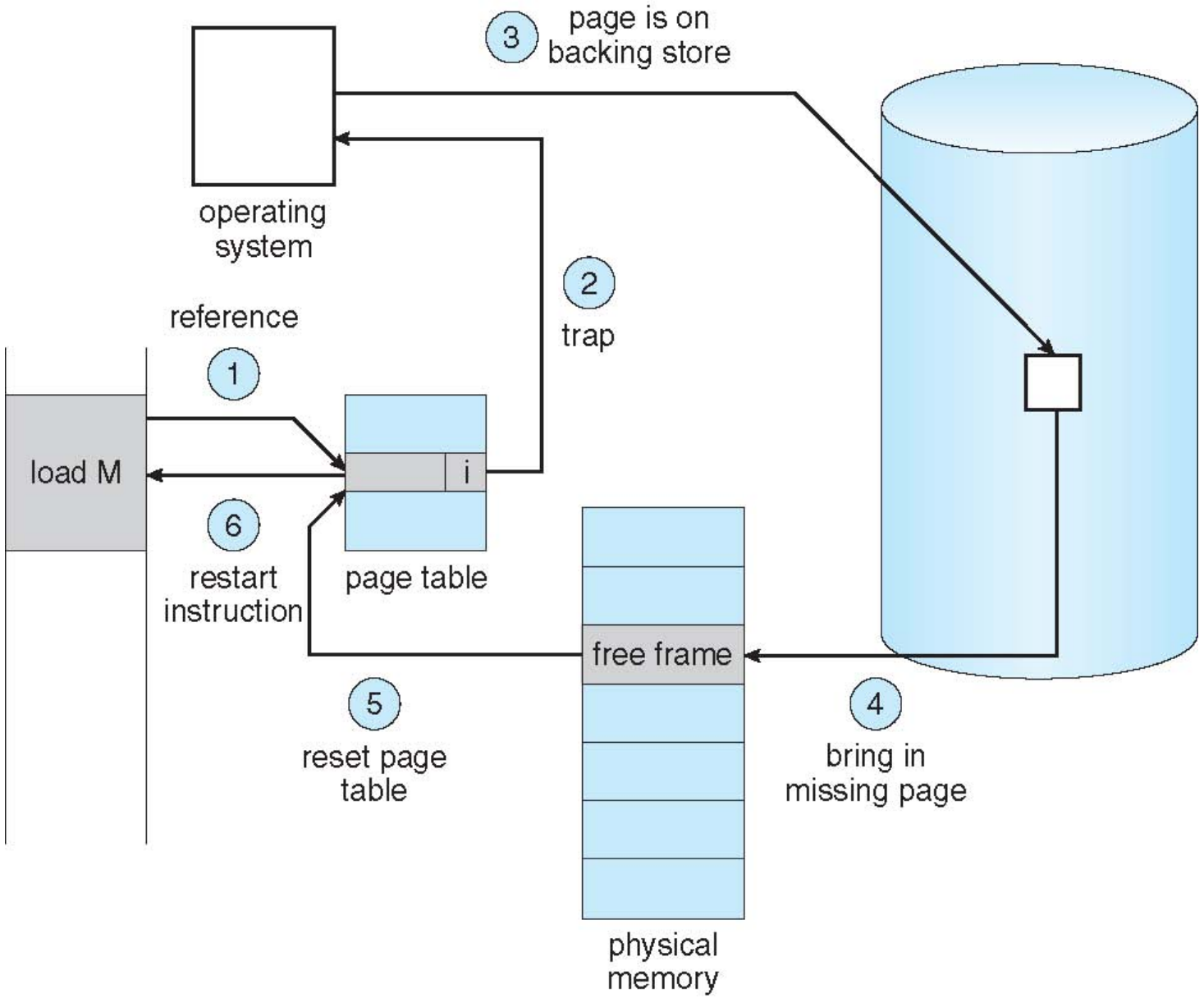
Page Table When Some Pages Are Not in Main Memory



Steps in Handling Page Fault

1. If there is a reference to a page, first reference to that page will trap to operating system
   - **Page fault**
2. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
3. Find free frame.
4. Swap page into frame via scheduled disk operation.
5. Reset tables to indicate page now in memory and set validation bit = **v.**
6. Restart the instruction that caused the page fault.

OPERATING SYSTEMS
Steps in Handling Page Fault (Cont.)



Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, not memory-resident -> page fault.
  - **Pure demand paging** – Never bring a page into memory until it is required.
- Theoretically, a given instruction could access multiple pages -> multiple page faults.
  - However this behavior is unlikely because of **locality of reference.** Process migrates from one locality (i.e., a set of pages that are actively used together) to another.
- Hardware support needed for demand paging:
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart

Instruction Restart

- Consider an instruction that could access several different locations (Eg: move some bytes from source location to destination):
  - Page fault might occur after the move is partially done.
  - Source block may have been modified so we cannot simply restart the instruction.
  - In one solution, the microcode computes and attempts to access both ends of both blocks.
    - Page fault can occur before anything is modified
  - The other solution uses temporary registers to hold the values of overwritten locations.
    - If Page fault occurs, all the old values are written back into memory before the trap occurs

Performance of Demand Paging

- Stages in Demand Paging (worst case):

1. Trap to the operating system.
2. Save the user registers and process state.
3. Determine that the interrupt was a page fault.
4. Check that the page reference was legal and determine the location of the page on the disk.
5. Issue a read from the disk to a free frame:
    a) Wait in a queue for this device until the read request is serviced.
    b) Wait for the device seek and/or latency time.
    c) Begin the transfer of the page to a free frame.

Performance of Demand Paging (Cont.)

6. While waiting, allocate the CPU to some other user
7. Receive an interrupt from the disk I/O subsystem (I/O completed)
8. Save the registers and process state for the other user
9. Determine that the interrupt was from the disk
10. Correct the page table and other tables to show page is now in memory
11. Wait for the CPU to be allocated to this process again
12. Restore the user registers, process state, and new page table, and then resume the interrupted instruction

Performance of Demand Paging (Cont.)

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

$$EAT = (1 - p) \times \text{memory access} + p \times \text{page-fault service time}$$

Demand Paging Example

- Let Memory access time = 200 ns and Average page-fault service time = 8 ms.
- EAT = (1 – p) x 200ns + p (8 ms)

  = (1 – p) x 200 + (p x 8,000,000) ns = 200 + (p x 7,999,800) ns
- If one access out of 1,000 causes a page fault i.e. p = 0.001, then

  EAT = 8200 ns

  This is a slowdown by a factor of 40 (i.e. 8200/200).
- If we want performance degradation < 10 percent,
  - 220 > 200 + 7,999,800 x p
    or p < .0000025
  - < one page fault in every 400,000 memory accesses
  - EAT is directly proportional to page-fault rate

Demand Paging Optimizations

- Swap space I/O is faster than file system I/O even if on the same device
  - Swap space is allocated in larger blocks, less management needed than file system.
  - Copy entire process image to swap space at process load time.
  - Then page in and out of swap space.
  - Used in older BSD Unix.

Demand Paging Optimizations (Cont.)

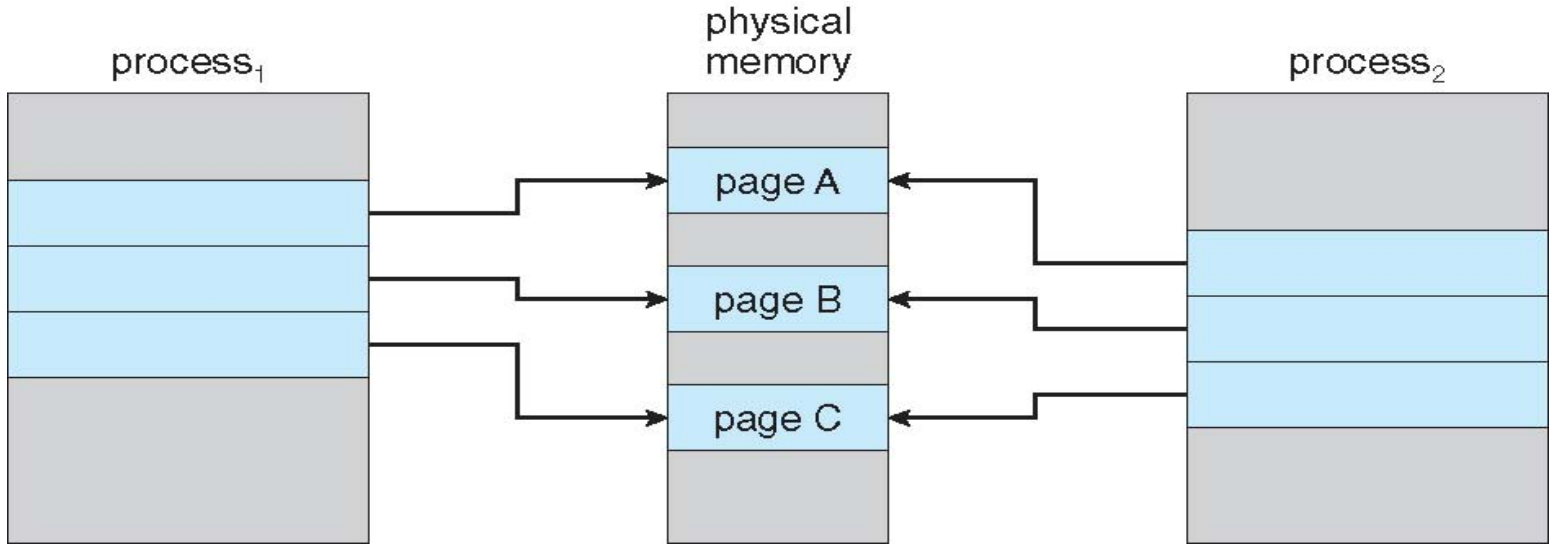
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD Unix.
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically don't support swapping.
  - Instead, demand page from file system and reclaim read-only pages (such as code) from applications.

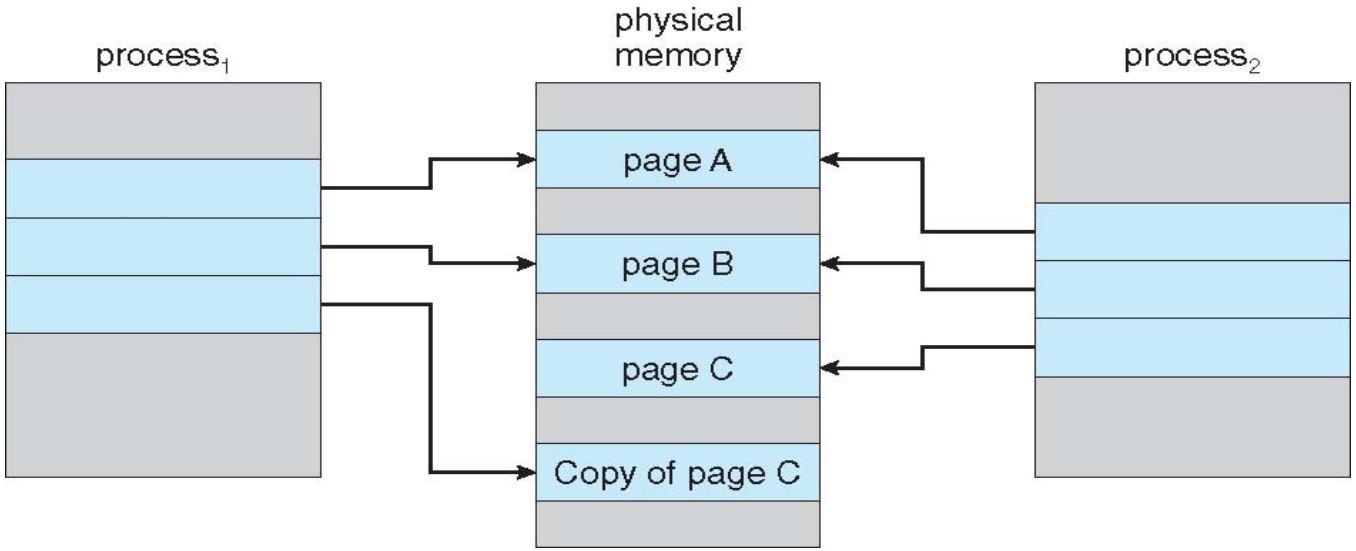
Copy-on-Write

- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied.
- COW allows more efficient process creation as only modified pages are copied.
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** frames.
- vfork() variation on fork() system call has parent suspend and child using copy-on-write address space of parent.
  - Designed to have child call exec().
  - Very efficient.

OPERATING SYSTEMS

Before and After Process 1 Modifies Page C



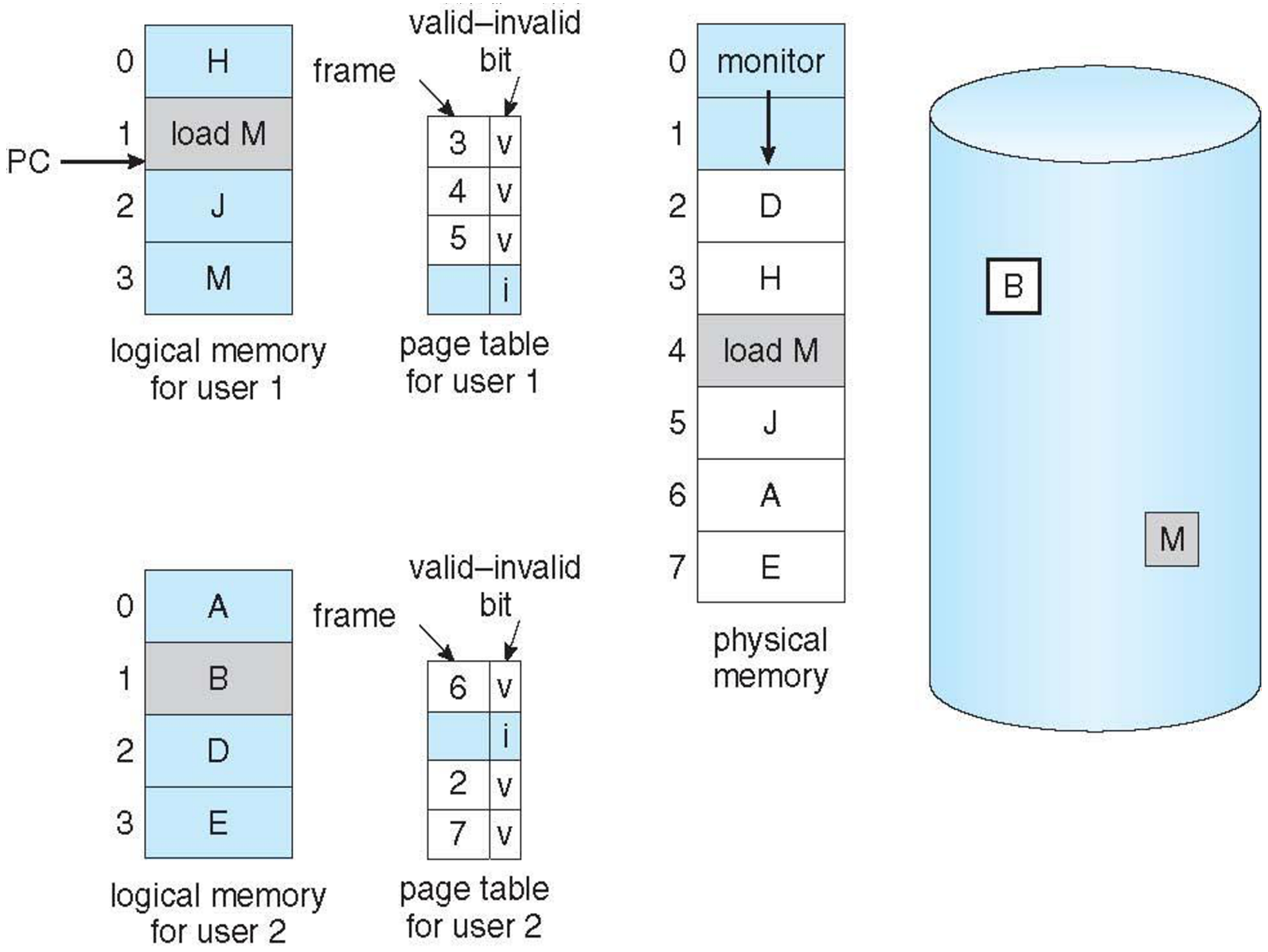


What Happens if There is no Free Frame?

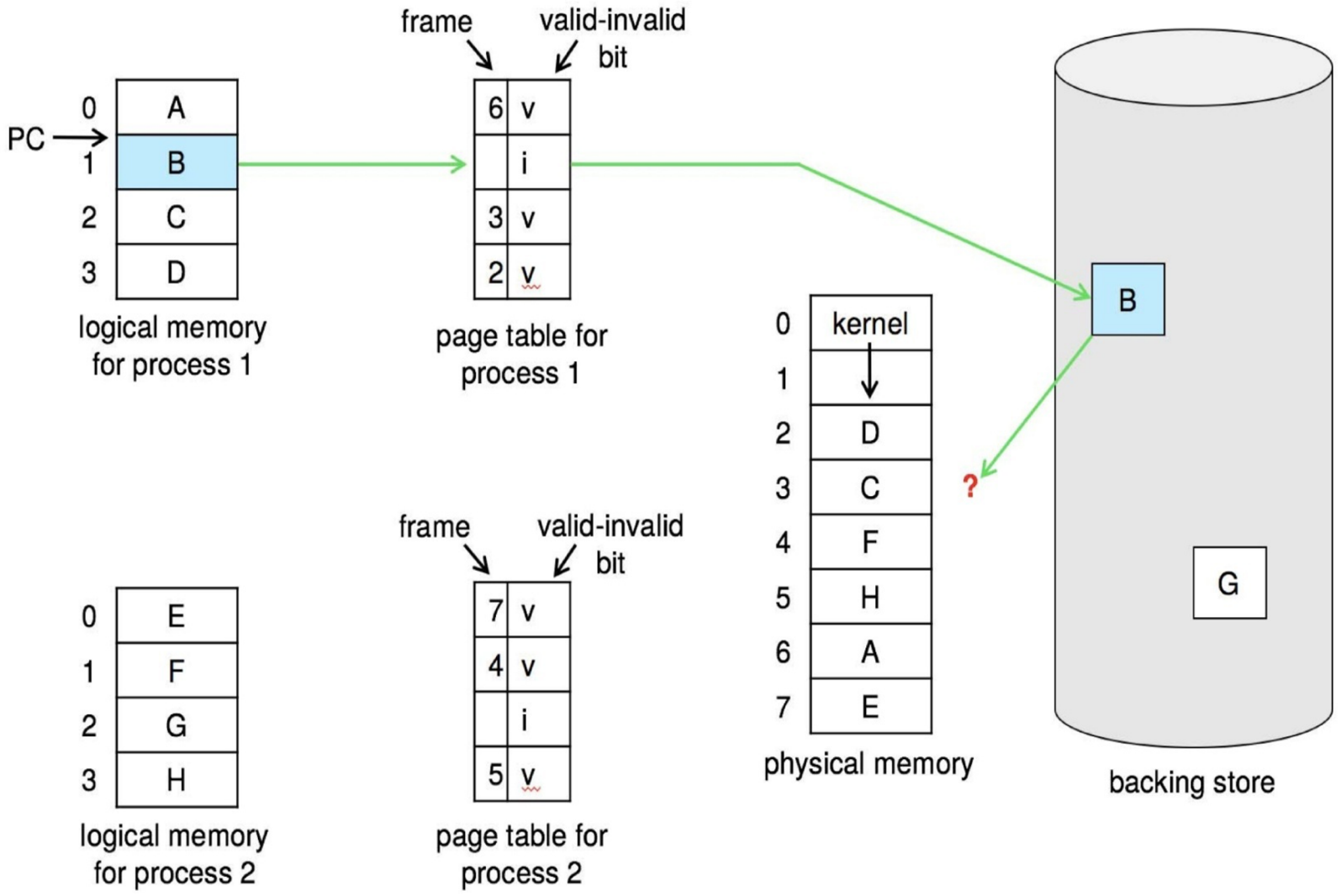
- Frames are used by process pages and are also required by the kernel, I/O buffers, etc.
- How many frames to allocate to each?
- Page replacement – find some page in memory that is not in use and page it out.
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults.
- Same page may be brought into memory several times.

OPERATING SYSTEMS

Need For Page Replacement – Example 1



Need For Page Replacement – Example 2

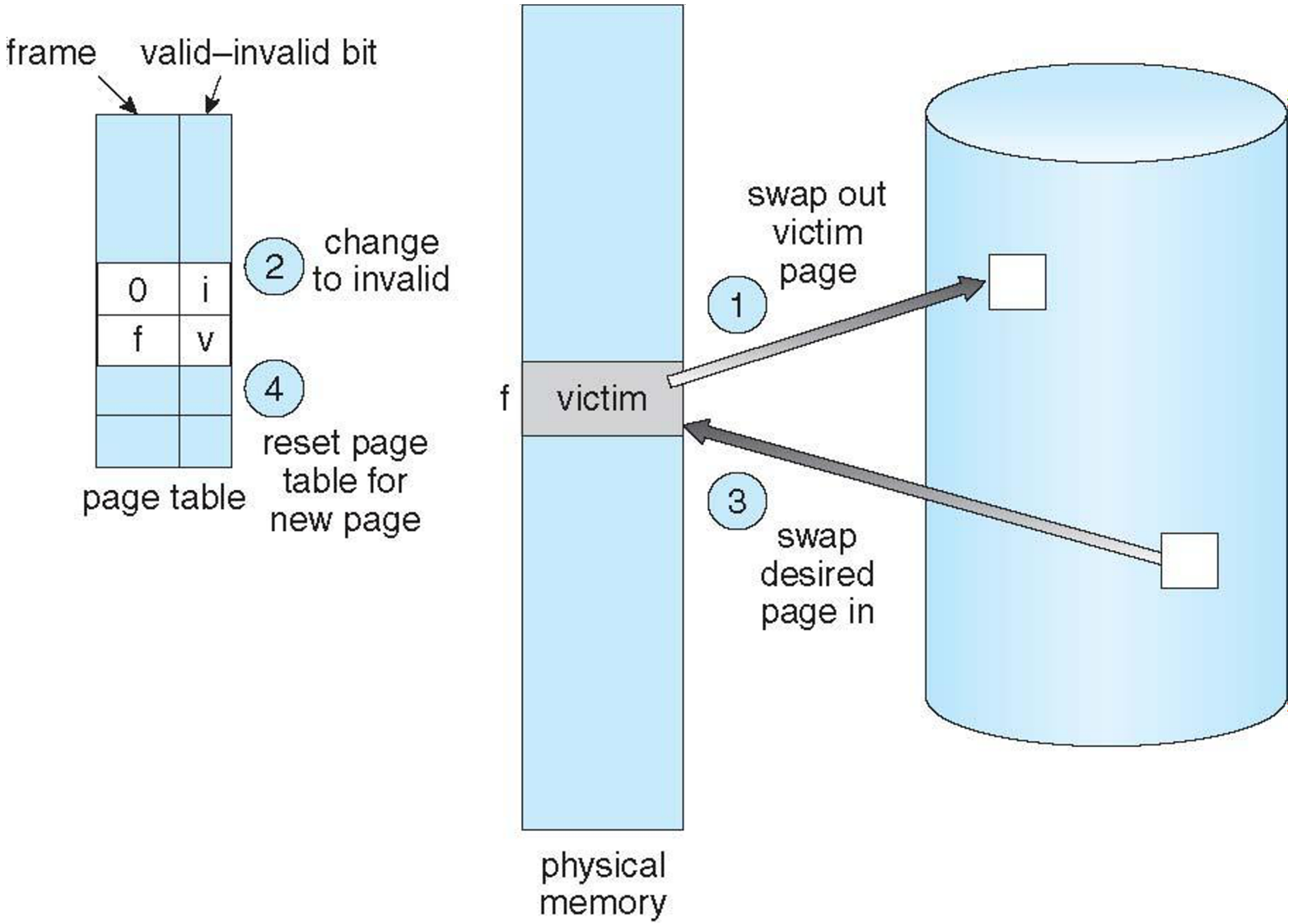


Basic Page Replacement

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 it is dirty(modified). This reduces overhead of page transfers.
3. Bring the desired page into the (newly) free frame; update the page table.
4. Continue the process by restarting the instruction that caused the trap.

Note: There are potentially 2 page transfers for page fault, thereby increasing EAT.

Basic Page Replacement (Cont.)



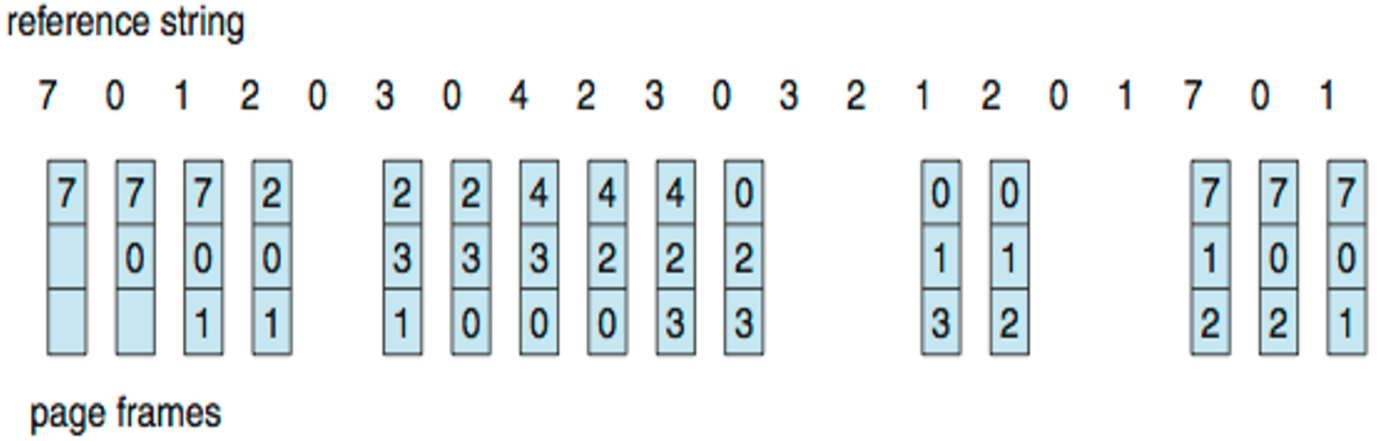
Page Replacement Algorithms

- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access.
  - Evaluate algorithm by running it on a particular string of memory references and computing the number of page faults on that string.
    - > String is just page numbers, not full addresses.
    - > Repeated access to the same page does not cause a page fault.
    - > Results depend on number of frames available.
- In all our examples, the **reference string** of referenced page numbers is **7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.**
- FIFO, LRU, Priority, and Optimal Page-replacement algorithms are discussed.

First-In-First-Out (FIFO) Algorithm

- Reference string: **7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1.**
- Consider the system has 3 frames:

  The number of page faults here are 15



- How can we reduce the number of page faults?

  Increase the number of available frames.

- What factors affect number of page faults?

  Reference string length, number of frames available, size of cache, etc.

OPERATING SYSTEMS

FIFO Illustrating Belady's Anomaly

- Can increasing the number of frames per process reduce the number of page faults?
- Consider the reference string: **1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5.**
- System with 3 frames

The number of page faults are 9.

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

- System with 4 frames

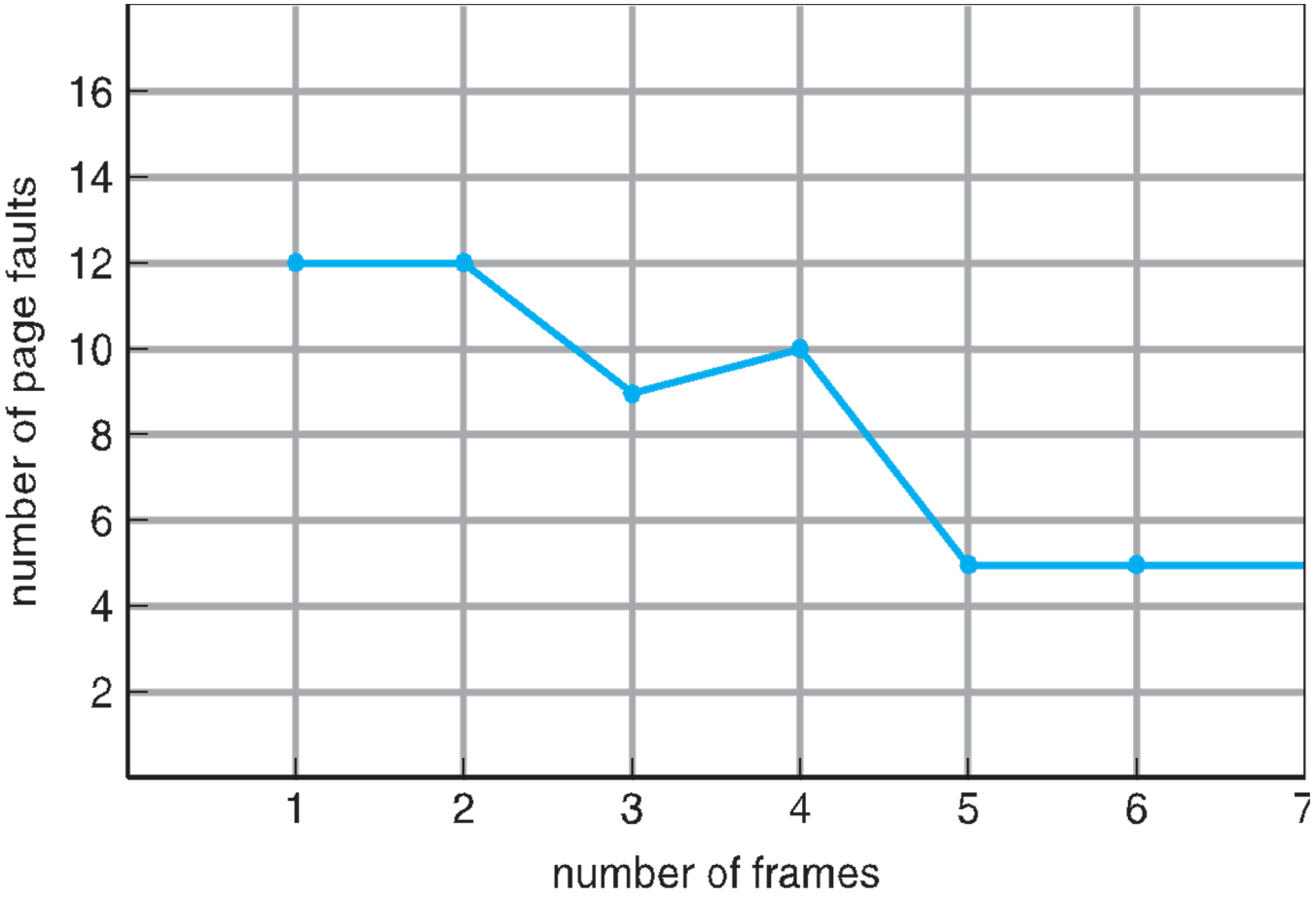
The number of page faults are 10.

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

- Thus, increasing the number of frames does not always reduce the number of page faults. This is called Belady's Anamoly.

OPERATING SYSTEMS

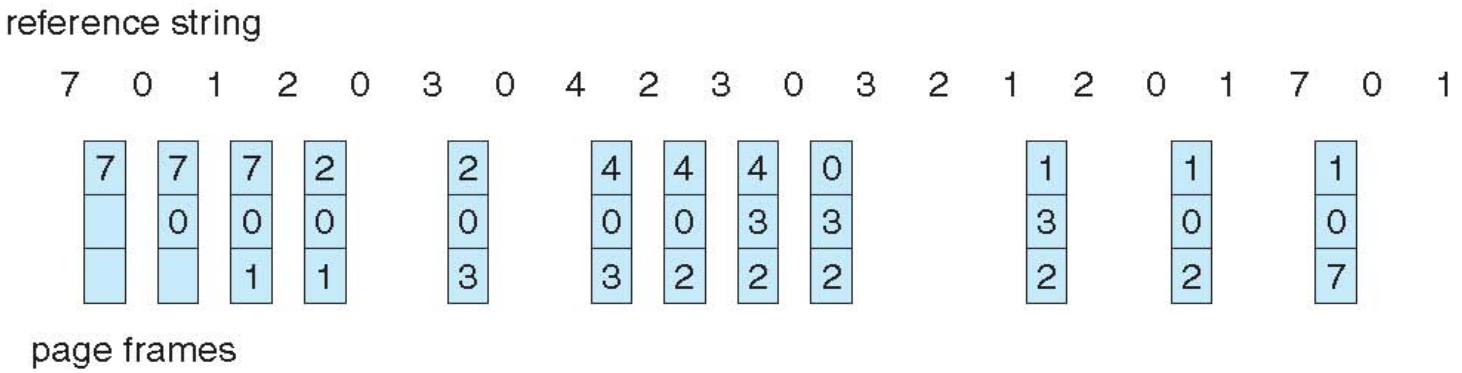
Graph Illustrating Belady's Anomaly



For the previous reference string, the number of page faults for 3 frames is 9 but for 4 frames is 10. This is Belady's Anomaly.

Least Recently Used (LRU) Algorithm

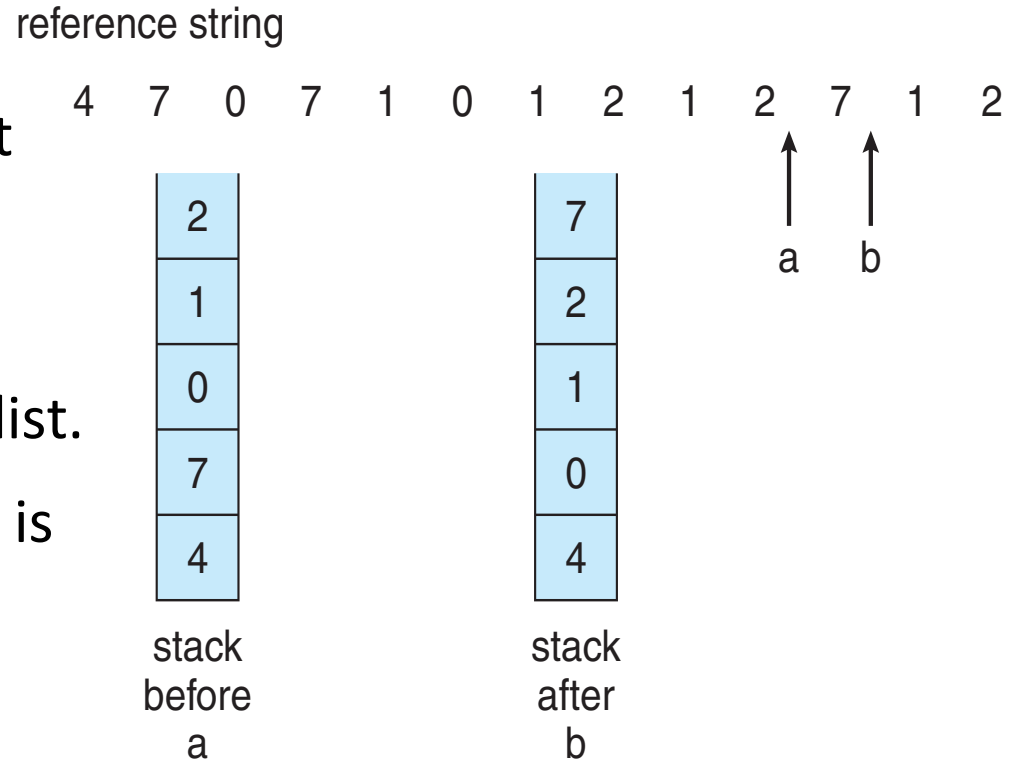
- Use past knowledge and replace page that has not been used in the most amount of time.
- Associate time of last use with each page.



- For the given reference string, this algorithm produces 12 faults.
- Generally a good algorithm and is frequently used.

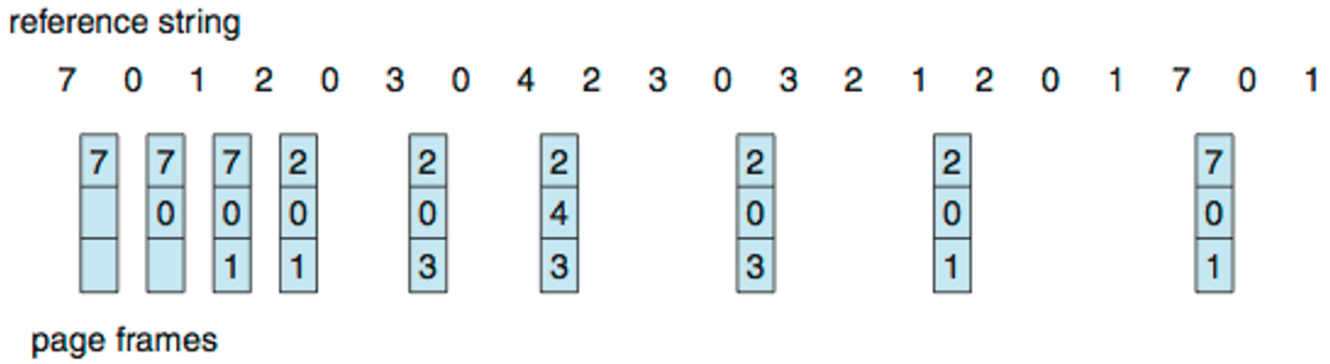
LRU Algorithm (Cont.)

- Counter implementation:
  - Every page entry has a counter; Each time the page is referenced through this entry, copy the clock into the counter.
  - When a page needs to be replaced, look at the counters of all pages to find oldest page (smallest counter value).
- Stack implementation:
  - Keep a stack of page numbers in a doubly linked list.
  - No searching for victim, but each update of stack is expensive.
- LRU and Optimal are cases of **stack algorithms** that don't have Belady's Anomaly.

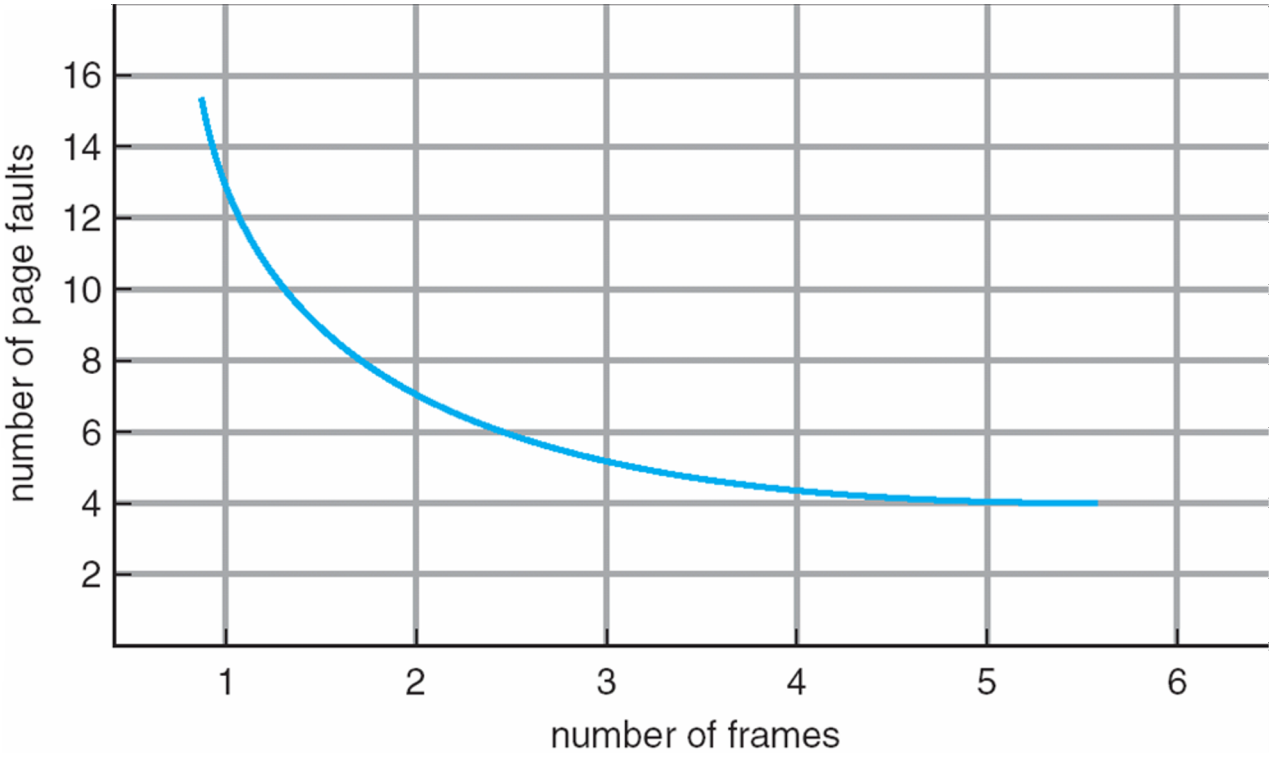


OPERATING SYSTEMS
Optimal Algorithm

- Replace a page that will not be accessed for longest period of time.
- How do you know this?
  - Can’t read the future. The reference string is never known in advance.
- Used for measuring how well an algorithm performs. It is a benchmark algorithm that has the lowest page fault rate and never suffers from Belady’s Anomaly.



Graph of Page Faults Versus The Number of Frames



Allocation of Frames

Frame-allocation algorithm should determine how many frames are to be allocated to each process.

- Each process needs a minimum number of frames:

  Eg: In IBM 370, 6 pages are required to handle a MOVE instruction:

  - instruction is 6 bytes, might span 2 pages.
  - 2 pages to handle *from.*
  - 2 pages to handle *to.*

- The maximum number of frames that can be allocated is total number of frames in the system.
- Two major allocation schemes:
  - Fixed allocation.
  - Priority allocation.

Fixed Allocation

- Equal allocation – For example, if there are 100 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames.
  - Keep some as free frame buffer pool.
- Proportional allocation – Allocate frames based on the size of process.
  - Dynamic as degree of multiprogramming and process sizes change.

$s_i = \text{size of process } p_i$

$S = \sum s_i$

$m = \text{total number of frames}$

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

Consider an example,

m = 62

s_1 = 10

s_2 = 127

a_1 = 10/137 X 62 = 4

a_2 = 127/137 X62 = 57

Priority Allocation

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

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

Global vs. Local Allocation

- **Global replacement** – process selects a replacement frame from the set of all frames; one process can use a frame of another process.
  - Here, process execution time can vary greatly.
  - But it provides greater throughput thus is used more commonly.

- **Local replacement** – each process selects victim frame from only its own set of allocated frames.
  - Provides more consistent per-process performance.
  - Could possibly lead to underutilized memory.

Non-Uniform Memory Access

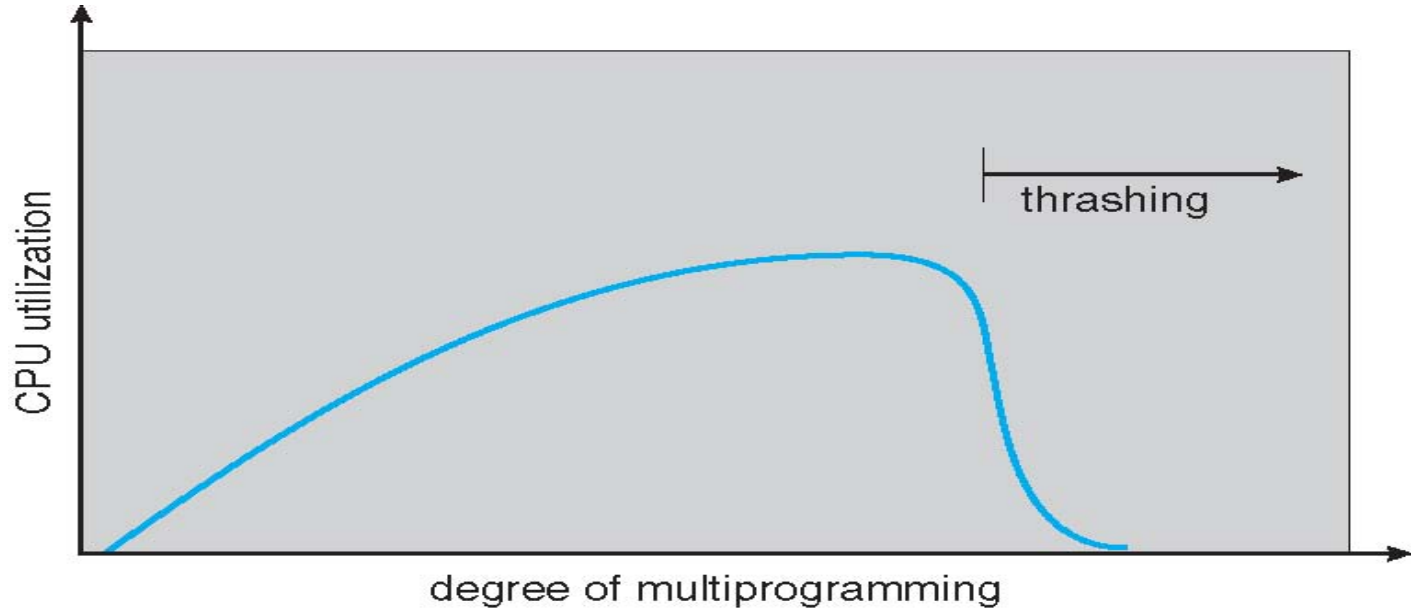
- So far, we have assumed all memory accesses require the same amount of time.
- Many systems have **NUMA** i.e. speed of memory access varies.
  - Consider system boards containing CPUs and memory, interconnected over a system bus.
- Optimal performance comes from allocating memory "closer to" the CPU on which the thread is scheduled and modifying the scheduler to schedule the thread on the same system board when possible.
  - Solved on Solaris by creating **lgroups**
    - Structure to track CPU / Memory low latency groups
    - When possible schedule all threads of a process and allocate all memory for that process within the lgroup (otherwise it picks nearby lgroups).

Thrashing

- If a process does not have the sufficient number of required pages in physical memory, then page-fault rate is very high.
  - Page fault to get page, then replace existing frame.
  - But if replaced frame is needed again then the whole cycle repeats.
  - This leads to:
    - Low CPU utilization.
    - Operating system thinking it needs to increase the degree of multiprogramming.
- **Thrashing** ≡ a process is busy swapping pages in and out and thus the execution does not progress.

Cause of Thrashing

- Thrashing results in severe performance problems.
- As the degree of multiprogramming increases, CPU utilization also increases.



Cause of Thrashing

- Why does thrashing occur?
  Σ size of locality > total memory size
- We can Limit the effects of thrashing by using a **local replacement algorithm** (or priority replacement algorithm)
  - If process starts thrashing, it cannot use frames from other processes.
- How does demand paging work? **Locality model**
  - Process migrates from one locality to another.
  - Localities may overlap.
  - Limit thrashing by using local or priority page replacement

Working Set

- The working-set model is based on the assumption of locality.
- The parameter, Δ , defines the working-set window.
- It examines the Δ most recent page references.
- The set of pages in the Δ most recent page references is the working set.
- If a page is in active use, it will be in the working set.
- If it is no longer being used, it will be dropped from the working set.
- Thus, the working set is an approximation of the program's locality
- Approximate the **W-S** model with interval timer + a reference bit

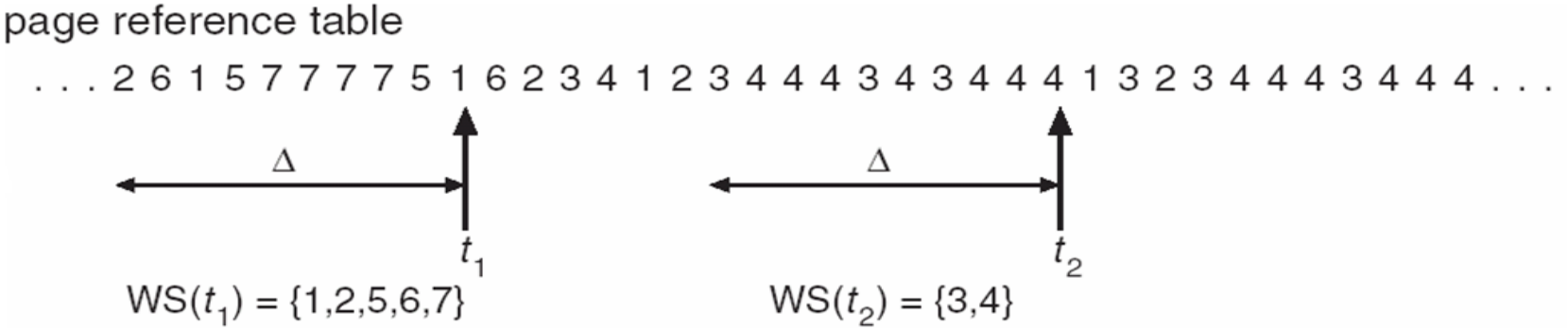
Keeping Track of the Working Set

- Example: Δ = 10,000 references
  - Timer interrupts every 5000 time units.
  - Keep in memory 2 bits for each page.
  - Whenever a timer interrupts, copy and set the values of all reference bits to 0.
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set.
- Why is this not completely accurate?
  - We cannot tell where, within an interval of 5000, a reference occurred.
- Improvement => 10 bits and interrupt every 1000 time units but overhead to service more frequent interrupts.

Working-Set Model

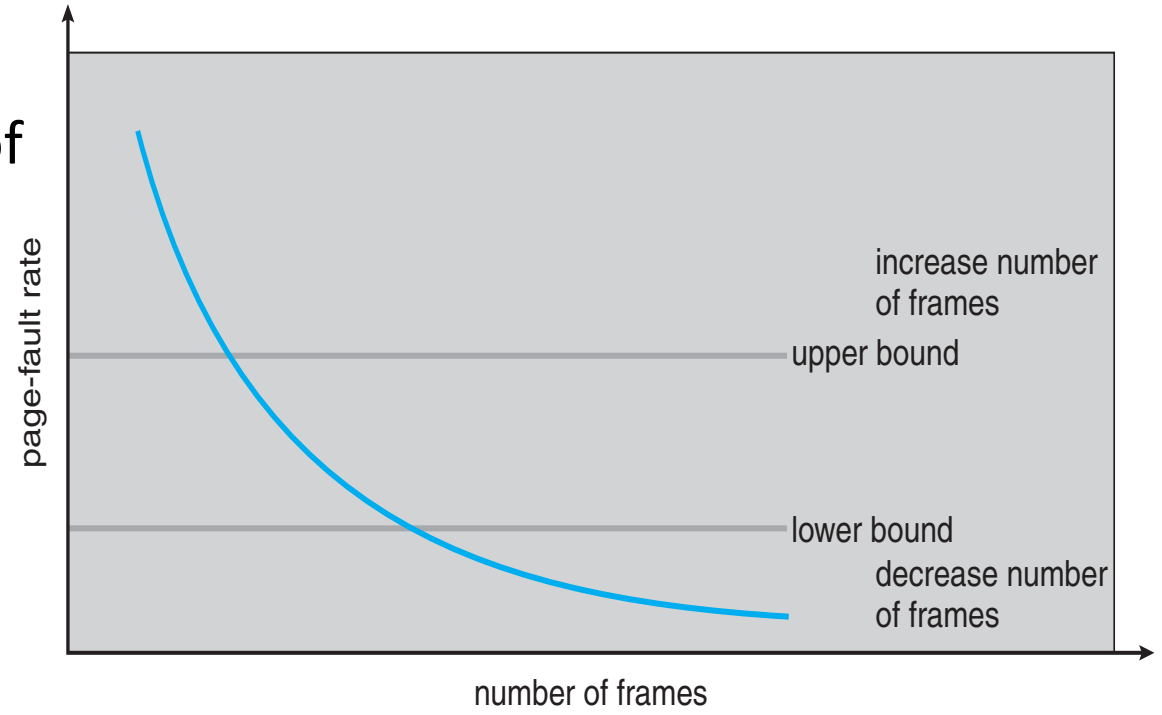
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (parameter that varies in time)
  - if Δ too small will not encompass entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program

- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- if $D > m$ *(available frames)* $\Rightarrow$ Thrashing will occur. Suspend or swap out one of the processes.



Page-Fault Frequency

- More direct approach than W-S model
  - Control the page-fault rate.
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy:
  - If actual rate too low, process loses frame.
  - If actual rate too high, process gains frame.
  - Processes can be suspended if the number of free frames are zero.



THANK YOU

Likitha P
Department of Computer Science Engineering

likithap@pes.edu