USAL
جامعة العلوم والاداب اللبنانية
UNIVERSITY OF SCIENCES & ARTS IN LEBANON

CSCI320 – Operating Systems

Ahmad Fadlallah

USAL
جامعة العلوم والاداب اللبنانية
UNIVERSITY OF SCIENCES & ARTS IN LEBANON

Memory Management

Virtual Memory

References

- These slides are based on the official slides of the following textbooks
  - Operating Systems: Internals and Design Principles (9th Edition), William Stallings, Pearson
  - Operating System Concepts with Java (10th Edition), Abraham Silberschatz, Peter B. Galvin and Greg Gagne, Wiley

Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model

Outline

- Background
- Demand Paging
- Copy-On-Write
- Page Replacement
- Allocation of Frames
- Thrashing

Background

Demand Paging
Copy-On-Write
Page Replacement
Allocation of Frames
Thrashing

Background

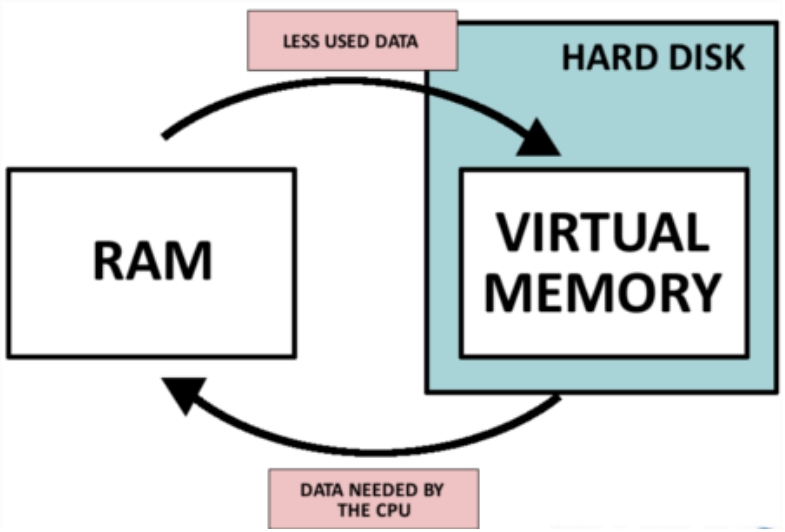
- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- **Entire program code not needed at same time**

Background (Cont.)

- Ability to execute partially-loaded program in memory
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running ➔ more programs run at the same time
    - **Increase** CPU utilization and throughput
    - **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.)

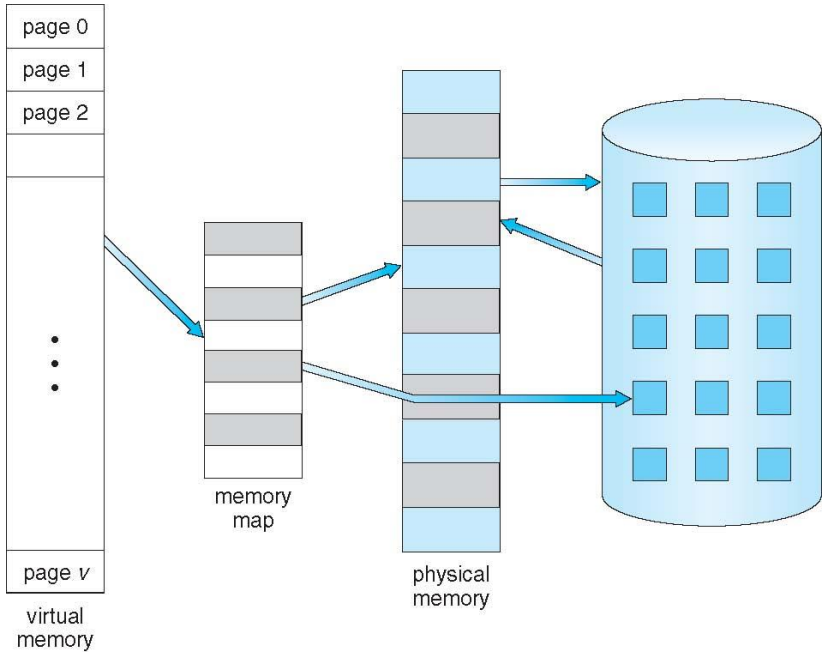
- **Virtual memory**: Separation of user logical memory from physical memory
- Only **part of the program** needs to be in memory for execution

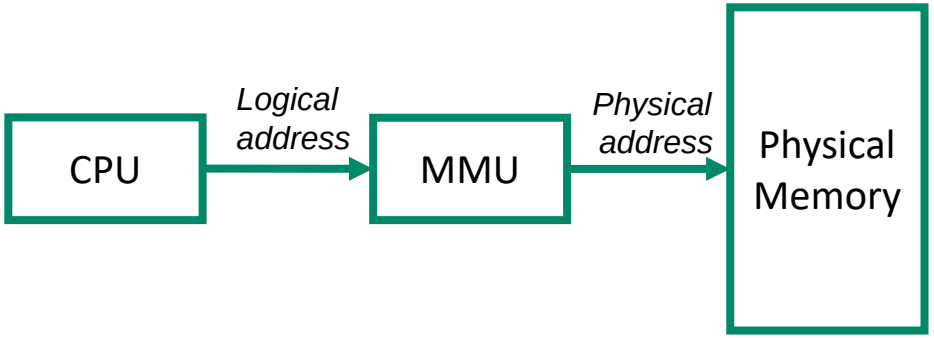


- Logical address space >> physical address space
- Allows address spaces to be **shared** by several processes
- More programs running concurrently
- Less I/O needed to load or swap processes

Background (Cont.)

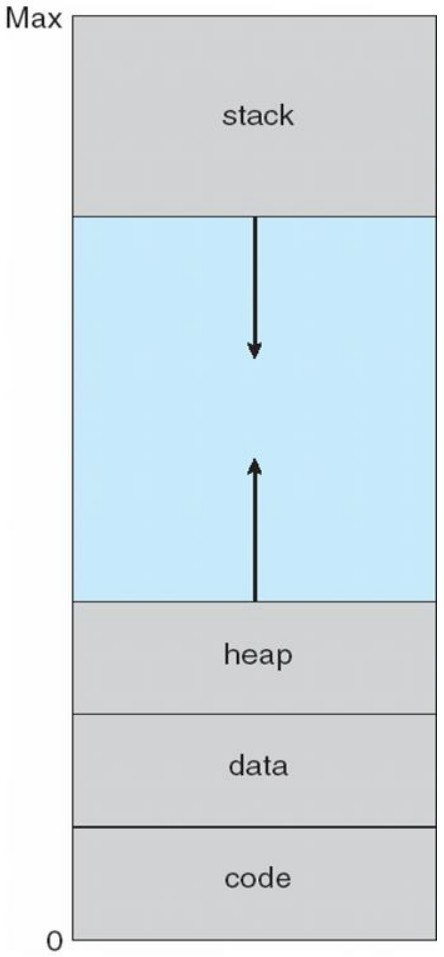
- **Virtual address space** of a process: logical view of how process is stored in memory
  - Usually starts at address 0, contiguous addresses **until end of space**
  - Physical memory is organized in **page frames**
    - May **not be contiguous**
  - MMU maps logical to physical





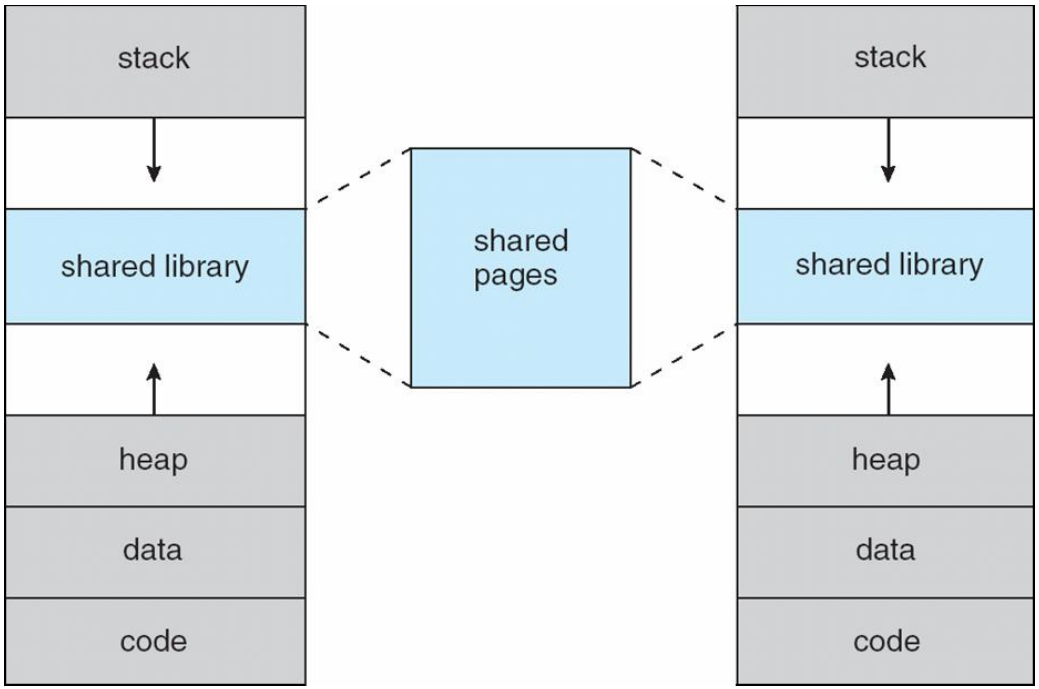
Virtual-address Space

- 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 a **hole**
- No physical memory needed until heap or stack grows to a given new page



Shared Library Using Virtual Memory

- **System libraries** shared via mapping into virtual address space
- **Shared memory** by mapping pages read-write into virtual address space



Background (Cont.)

- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Background

Demand Paging

Copy-On-Write

Page Replacement

Allocation of Frames

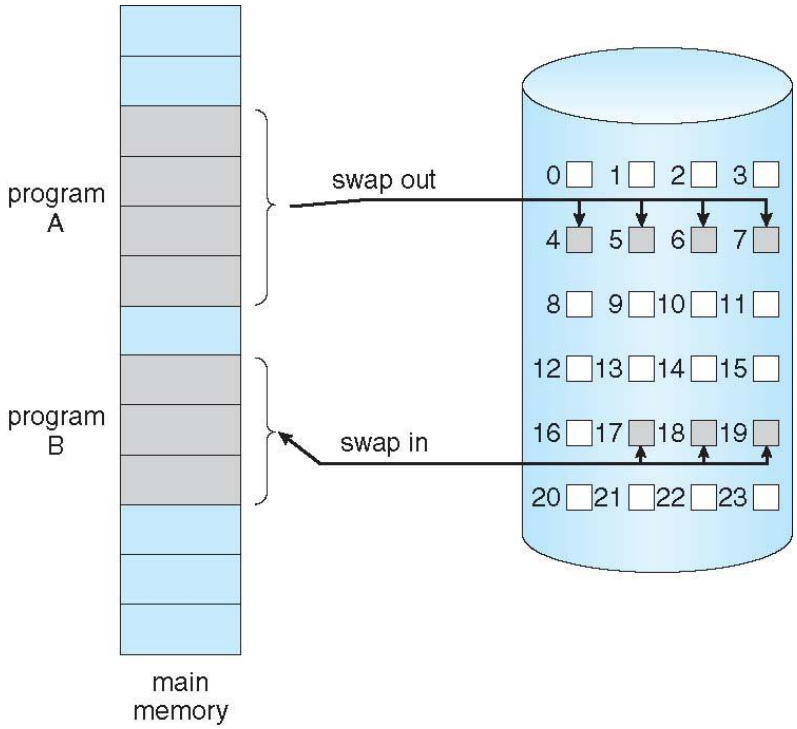
Thrashing

Demand Paging

- Bring a page into memory only when it is needed during execution
- Pages never accessed are never loaded into physical memory
- Less I/O needed, no unnecessary I/O
- Less memory needed
- Faster response
- More users

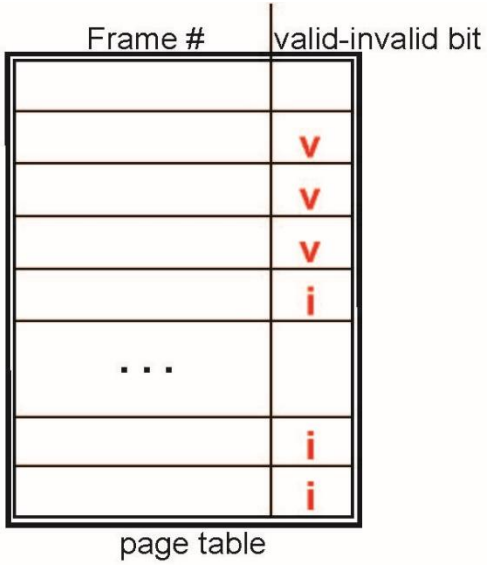
Demand Paging (cont'd)

- Similar to paging system with **swapping**
- 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 will be needed
  - Swapper that deals with pages is a **pager**

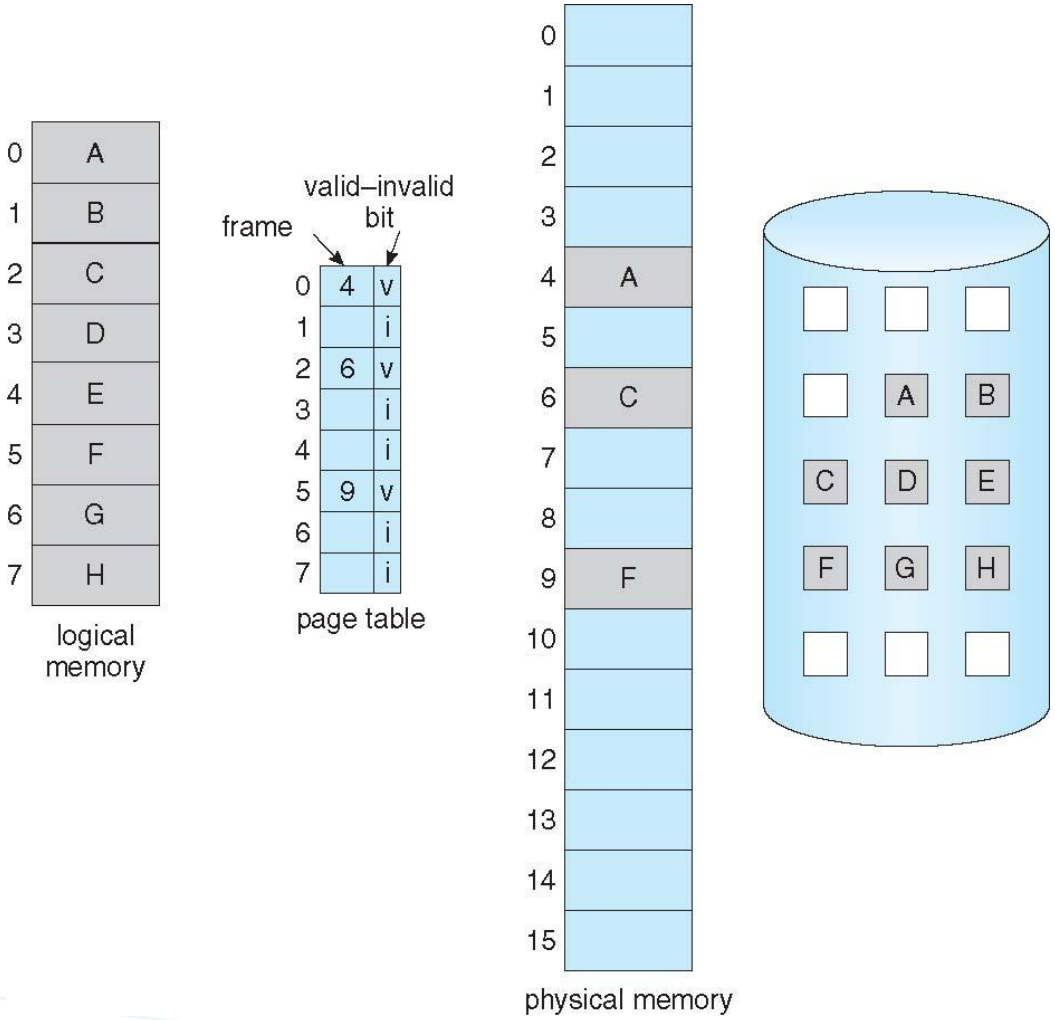


Valid-Invalid Bit (Reminder)

- With each page table entry, a **valid–invalid bit** is associated
  - v ⇒ in-memory / memory resident
  - `i` ⇒ not-in-memory (or not in the logical address space)
- Initially valid–invalid bit is set to `i` on all entries
- During MMU address translation, if valid–invalid bit in page table entry is i ⇒ page fault

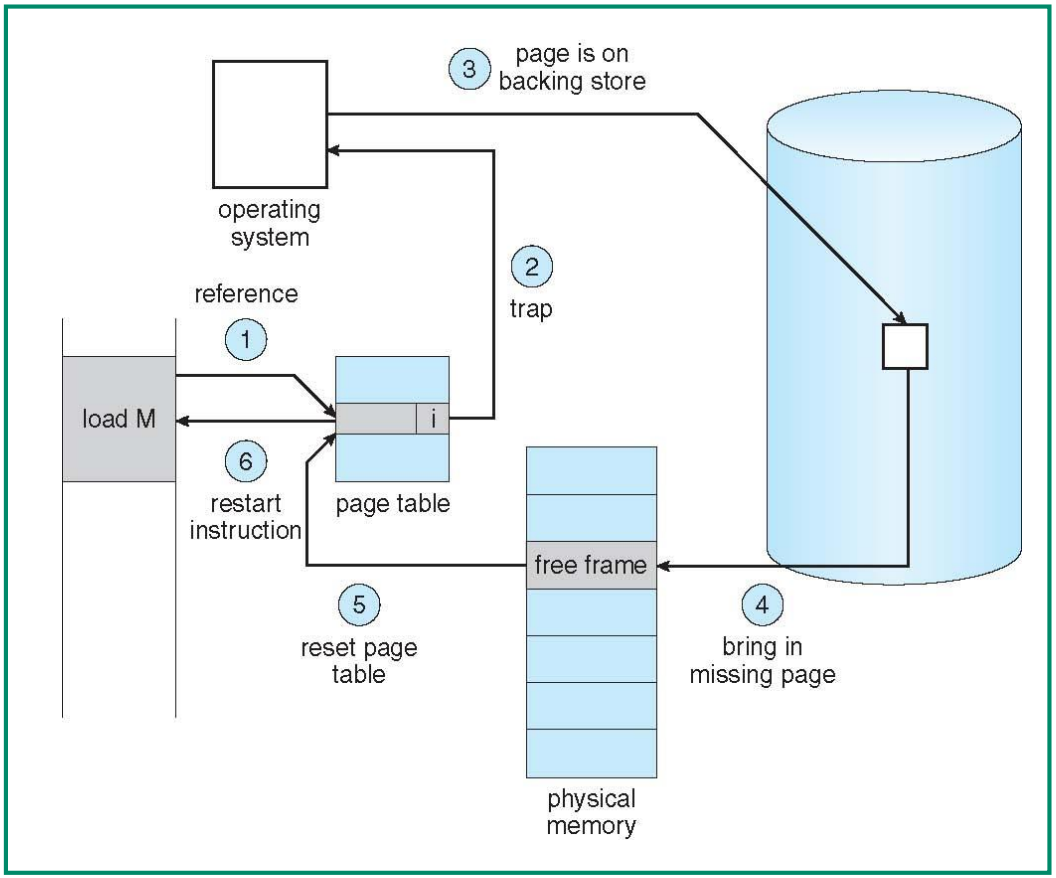


Page Table When Some Pages Are Not in Main Memory



Page Fault

- If there is a reference to a page, **first reference** to that page will trap to operating system: **page fault**



Operating system looks at another table to decide:

- Invalid reference ⇒ abort
- Just not in memory

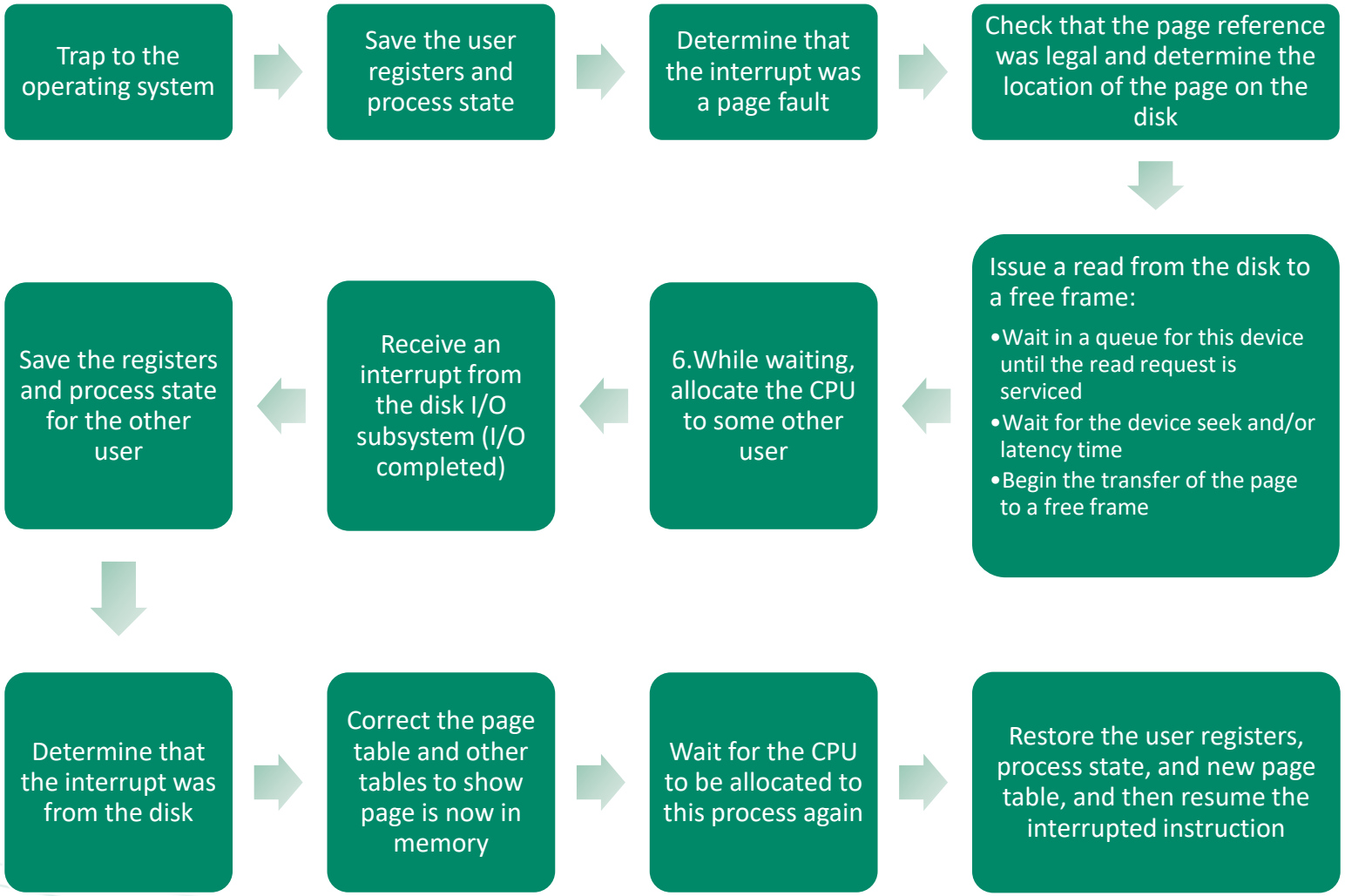
Swap page into frame via scheduled disk operation

Restart the instruction that caused the page fault

Find free frame

Reset tables to indicate page now in memory => validation bit = **v**

Performance of Demand Paging (Worst Case)



Performance of Demand Paging (Cont.)

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

EAT = (1 – p) * memory access
+ p * (page fault overhead + swap page out + swap page in)

Demand Paging Example

EAT = (1 – p) * memory access
+ p * (page fault overhead + swap page out + swap page in)

- Memory access time = **200 ns**
- Average page-fault service time = 8 ms = **8,000,000 ns.**
- EAT = (1 – p) x 200 ns + p (8 ms)
  = (1 – p) x 200 + p x 8,000,000 ns
  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then
  EAT = 8.2 microseconds.
  This is a slowdown by a factor of 40!!
- For a performance degradation < 10%
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses

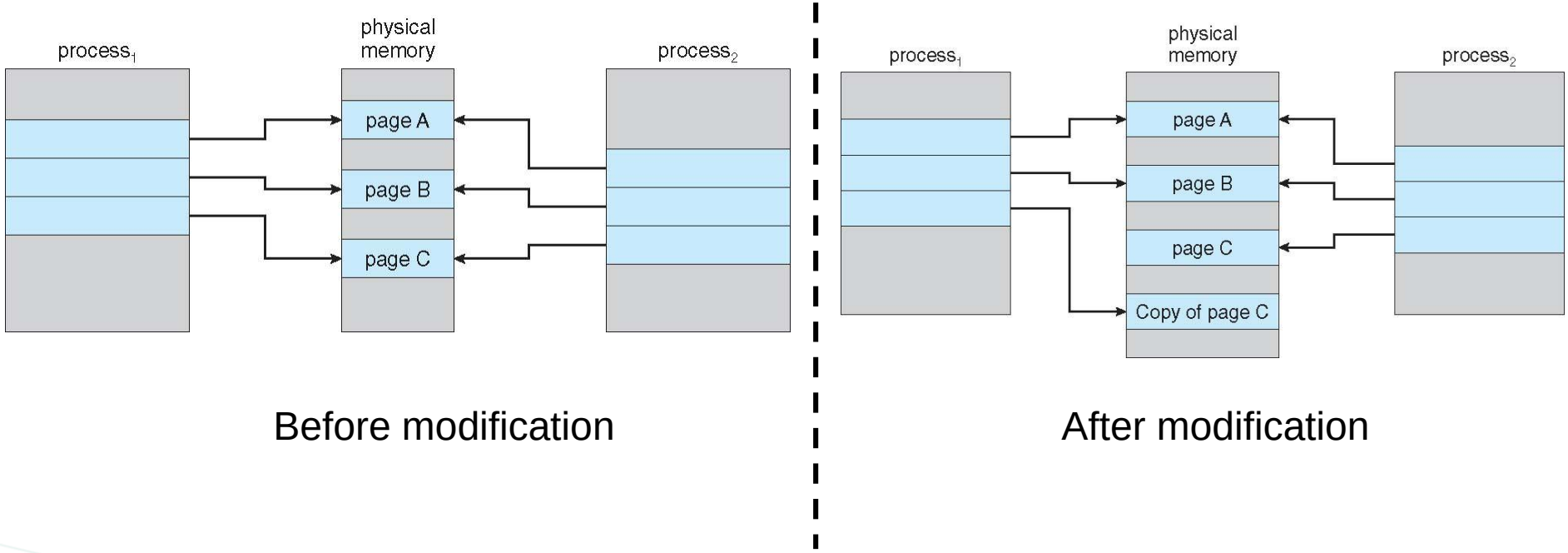
Background
Demand Paging

Copy-On-Write

Page Replacement
Allocation of Frames
Thrashing

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*

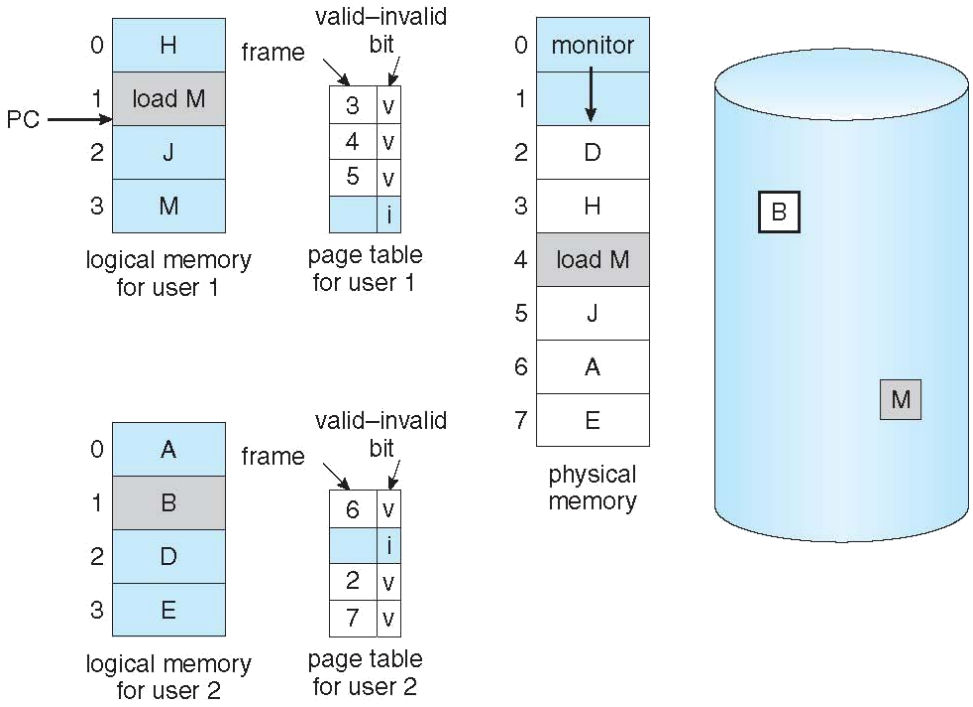


Copy-on-Write (cont'd)

- Copy-On-Write 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 pages
  - Pool should always have free frames for fast demand page execution
    - Don't want to have to free a frame as well as other processing on page fault
  - Why zero-out a page before allocating it?

What Happens if There is no Free Frame?

- Frames can be used by *process pages* as well as by the *kernel*, *I/O buffers*, etc.
- **How much Frames to allocate to each?**
  - Fixed percentage
  - Open competition
- **What if there is no free frame?**
  - Terminate user process
  - Swap out a process
  - Page replacement



Background
Demand Paging
Copy-On-Write

Page Replacement

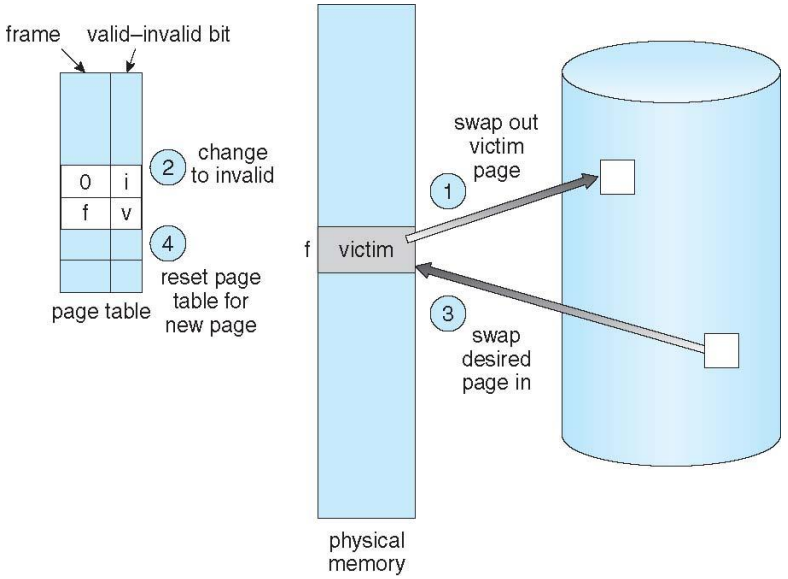
Allocation of Frames
Thrashing

Page Replacement

- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Page replacement completes separation between logical memory and physical memory => large virtual memory can be provided on a smaller physical memory

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 dirty**
3. Bring the desired page into the (newly) free frame; update the page and frame tables
4. Continue the process by restarting the instruction that caused the trap



Use **modify** (**dirty**) **bit** to reduce overhead of page transfers: only modified pages are written to disk

Potentially 2 page transfers for page fault – increasing EAT

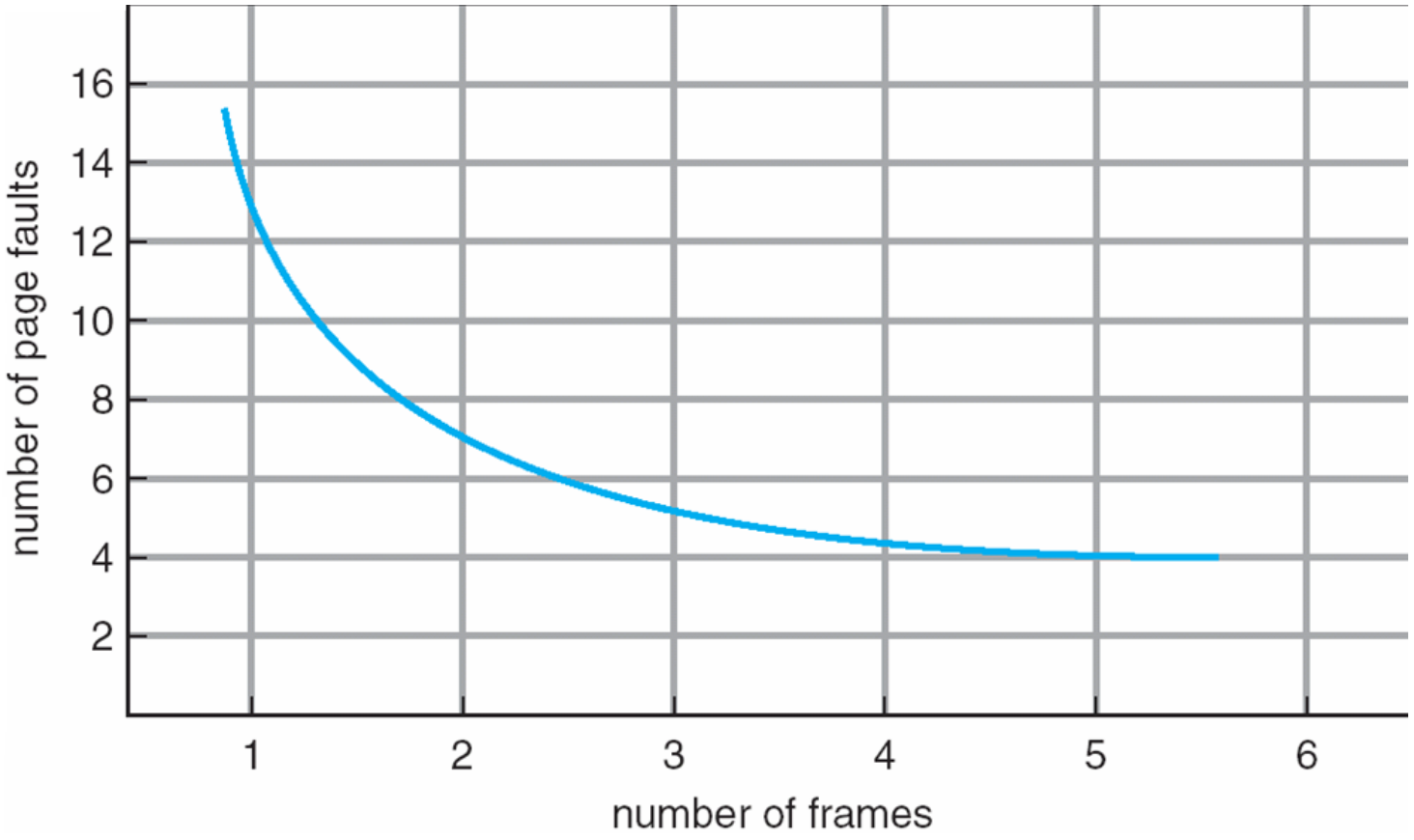
Page and Frame Replacement Algorithms

- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want **lowest page-fault rate** on both first access and re-access

Page and Frame Replacement Algorithms (cont'd)

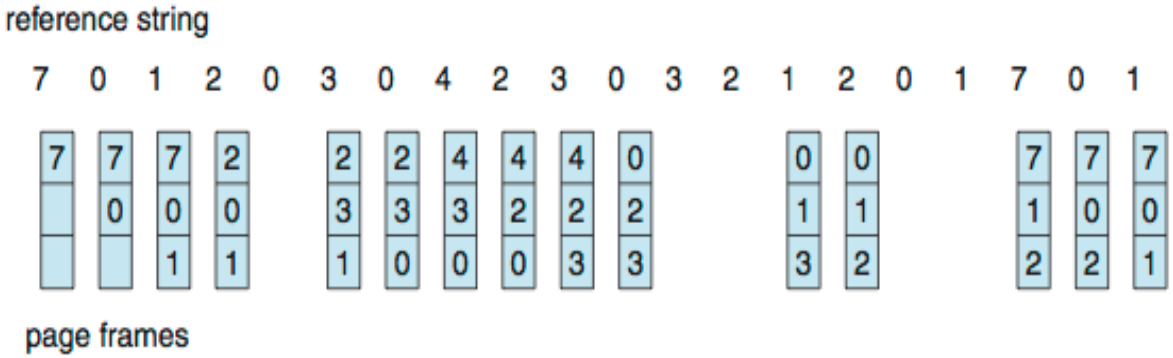
- Evaluate algorithm by:
  1. Running it on a particular string of memory references (**reference string**) and
  2. Computing the number of page faults on that string
- Reference string is just page numbers, not full addresses
- Repeated access to the same page does not cause a page fault
- Results depend on number of frames available
- In all our examples, the **reference string** of referenced page numbers is

Graph of Page Faults Versus The Number of Frames



First-In-First-Out (FIFO) Algorithm

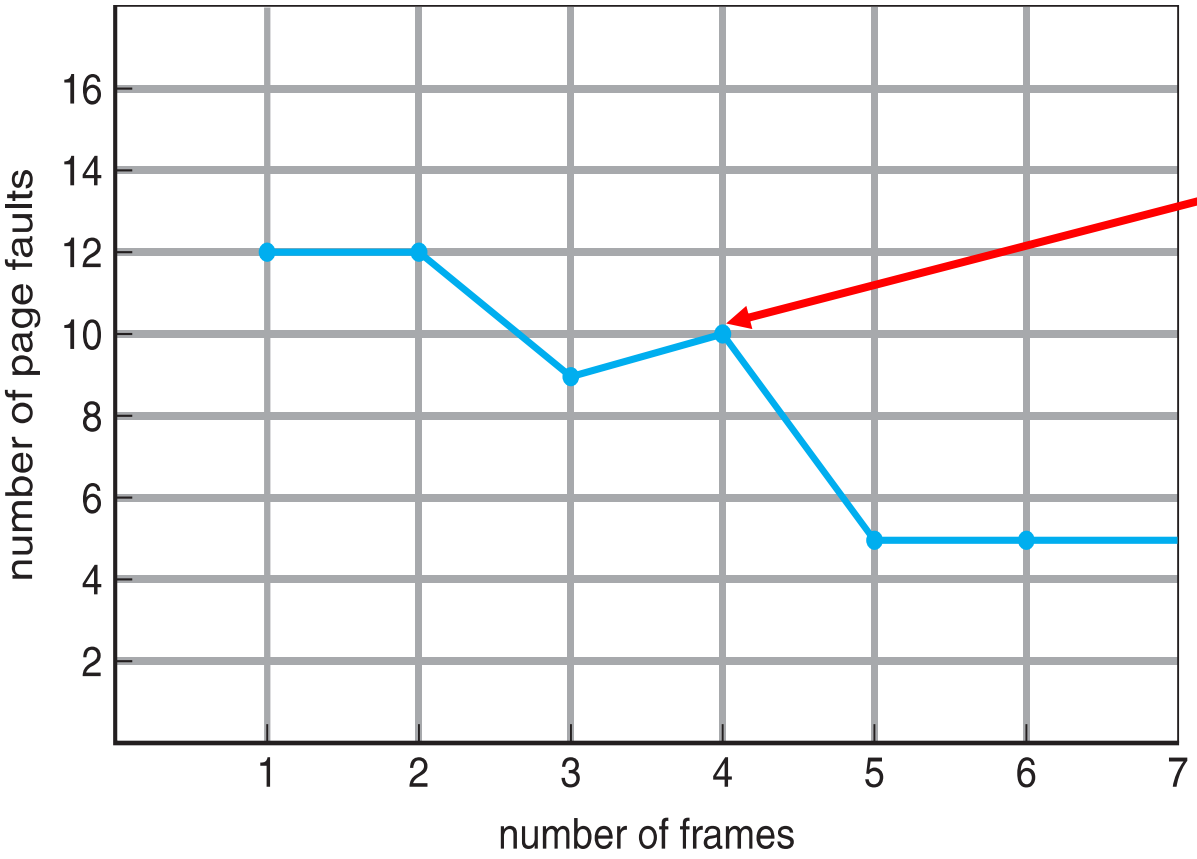
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

- Can vary by reference string: consider **1,2,3,4,1,2,5,1,2,3,4,5**
- Adding more frames can cause more page faults!
  - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly



Belady's Anomaly

FIFO Illustrating Belady's Anomaly - Example

pages = 3
page faults = 9

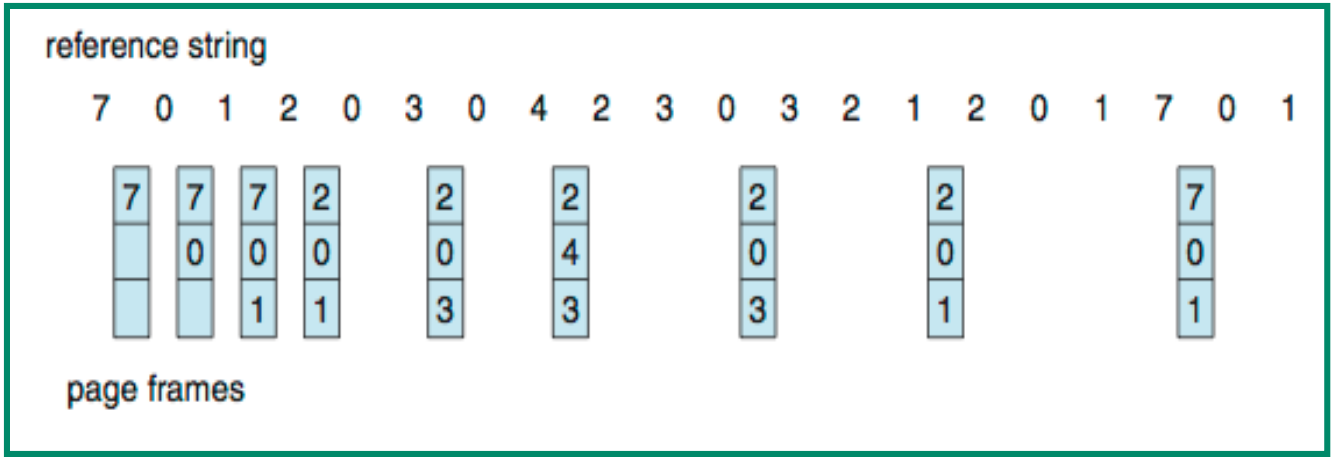
Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest Page	3	2	1	0	3	2	4	4	4	1	0	0
		3	2	1	0	3	2	2	2	4	1	1
Oldest Page			3	2	1	0	3	3	3	2	4	4

pages = 4
page faults = 10

Page Requests	3	2	1	0	3	2	4	3	2	1	0	4
Newest Page	3	2	1	0	0	0	4	3	2	1	0	4
		3	2	1	1	1	0	4	3	2	1	0
			3	2	2	2	1	0	4	3	2	1
Oldest Page				3	3	3	2	1	0	4	3	2

Optimal Algorithm

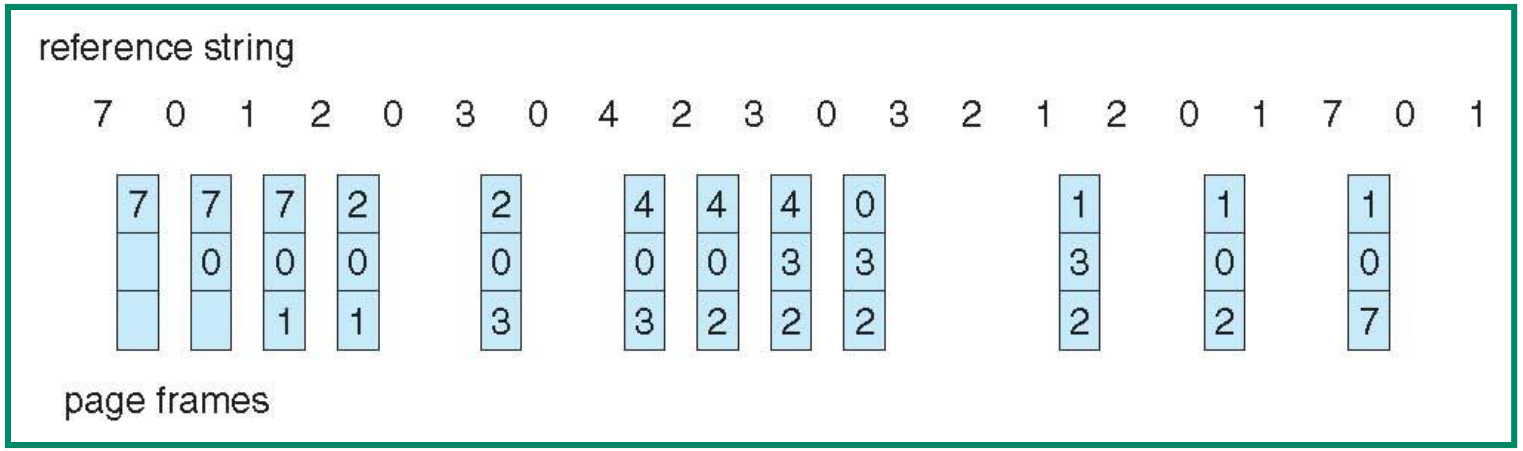
- Replace page that will not be used for **longest period of time**
- How do you know this?
  - Can't read the future
- **Used for measuring how well your algorithm performs**



9 page faults is optimal for the example

Least Recently Used (LRU) Algorithm

- Use **past knowledge** rather than future
- Replace page that has not been used in the most amount of time
- Associate **time of last use** with each page



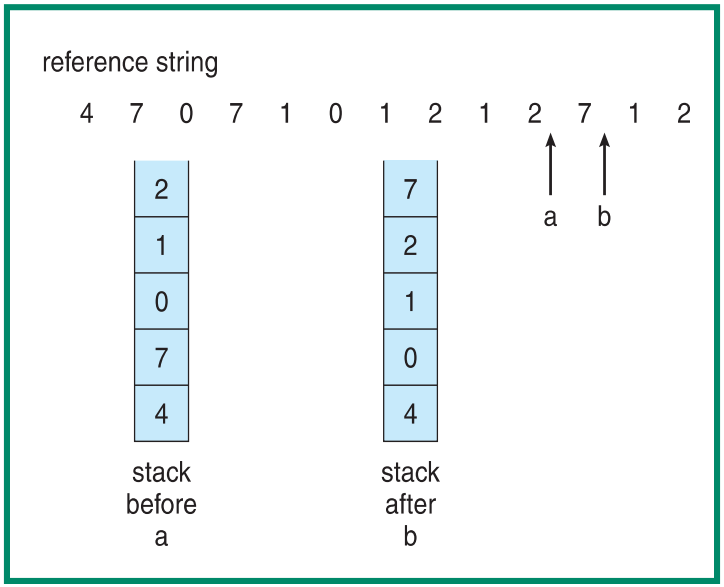
- 12 faults – better than FIFO but worse than OPT
- **Generally good algorithm and frequently used**
- But how to implement?

LRU Algorithm (Cont.)

- **Counter implementation**
  - Add a **logical clock** (counter) to the CPU
  - The clock is incremented for every memory reference
  - Associate with each page-table entry a **time-of-use field**
  - Each time page is referenced through this entry, copy the clock into the time-of-use field
  - When a page needs to be changed, **look at the counters to find smallest value**
  - Search through table is needed and write to memory

LRU Algorithm (Cont.)

- **Stack implementation**
  - Keep a stack of page numbers
  - When a page is referenced, move it to the top
  - MRU page is **always on the top**
  - LRU page is **always at the bottom**
  - Better to use *doubly linked list* (Head and Tail pointer)
    - Requires 6 pointers to be changed (at worst)



LRU Algorithm (Cont.)

- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly
- A **stack algorithm** is an algorithm for which it can be shown that the set of pages in memory for ***n* frames** is always a ***subset*** of the set of pages that would be in memory with ***n*+1 frames**
  - In LRU: **n most recent frames** are subset of **n+1 recent frames**
- Both implementation of LRU algorithms require hardware assistance
  - **Software implementation** will drastically slow down the process

LRU Approximation Algorithms/ Reference Bit

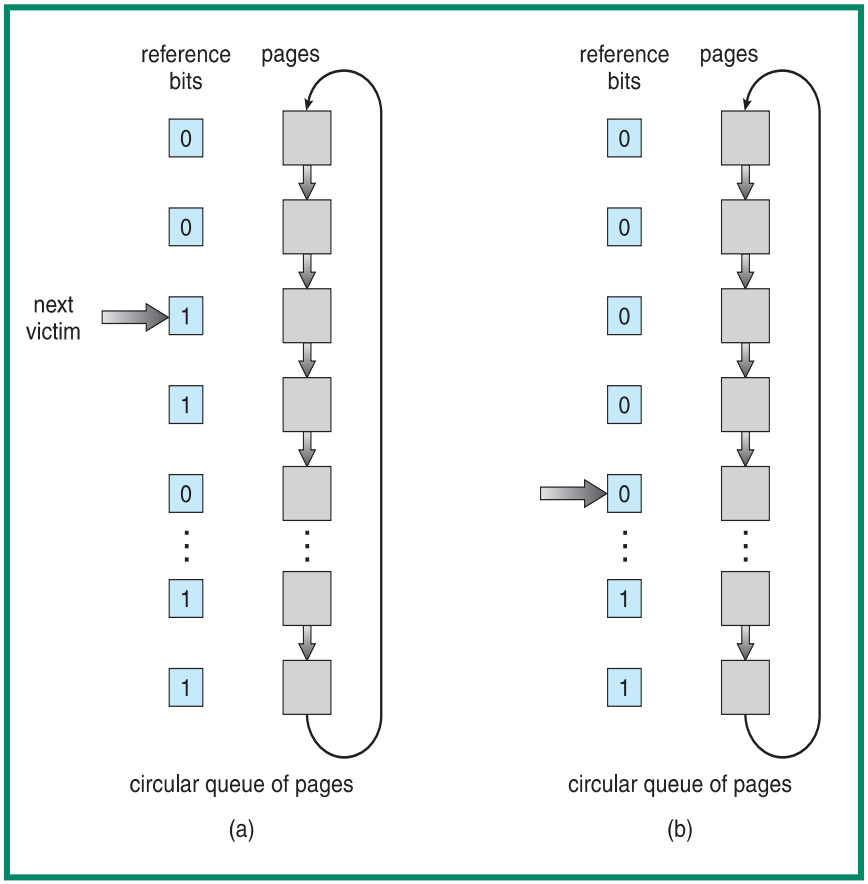
- **With each page associate a bit**, initially = 0
- When **page is referenced**, bit set to 1
- **Replace any with reference bit = 0** (if exists)
  - BUT We do not know the order of use
- Possibility for additional reference bits (8 bits)
  - When the page is referenced, **add a 1 in the high order bit** and **shift the other bits right** => discard the least significant bit
  - When the page is NOT referenced, **add a 0 in the high order bit** and **shift the other bits right** => discard the least significant bit

Referenced	Previous value	Updated value
Yes	10111001	11011100
No	01011001	00101100

  - Treat the byte as unsigned int => **Lowest number is the LRU**

LRU Approximation Algorithms/ Second Chance Algorithm

- FIFO + hardware-provided reference bit
- Referred to as **Clock Algorithm**
- If page to be replaced has
  - Reference bit = 0 ➔ Replace it
  - Reference bit = 1 ➔ Give it a second chance
    - Set reference bit to 0 and leave page in memory
    - Replace next page, subject to same rules



LRU Approximation Algorithms/ Enhanced Second-chance algorithm

- Takes into consideration the I/O required
- Uses two bits: **reference bit** and **modify bit**
- Four possible classes

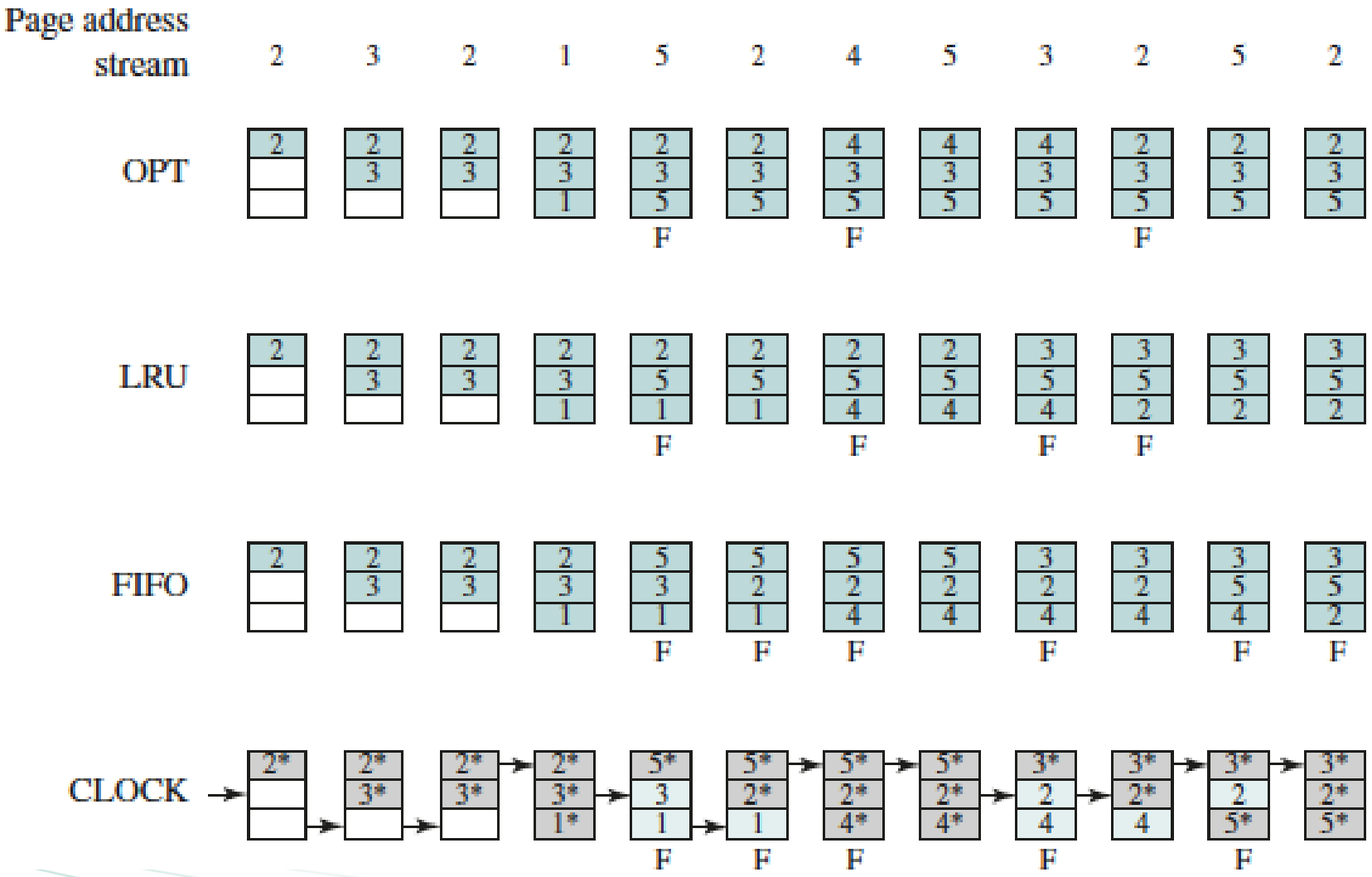
(0, 0)	Neither recently used Nor modified	Best page to replace
(0,1)	Not recently used but modified	Not quite as good, because the page will need to be written out before replacement
(1,0)	Recently used but clean	Probably will be used again soon
(1,1)	Recently used and modified	Probably will be used again soon, and the page will be need to be written out to disk before it can be replaced

- Clock replacement but replace the first page encountered in the lowest non-empty class
- May require to scan the circular queue several times before finding the page to replace

Counting-Based algorithms

- Keep a counter of the number of references that have been made to each page
- **Least Frequently Used (LFU)**
  - Replace the **page with smallest count**
  - **Problem**: page heavily used but never used again
  - **Solution**: shifting the counts right by 1 bit at regular intervals
- **Most Frequently Used (MFU)**
  - Replace the page with **greatest counter!**
  - **Argument**: page of the smallest counter was probably just brought in and has yet to be used

Page Replacement Algorithms - Example



Exercise

Consider the following page reference string:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6.

How many page faults would occur for the following replacement algorithms, assuming one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.

1. LRU replacement
2. FIFO replacement
3. Optimal replacement
4. Second Chance Algorithm

Background
Demand Paging
Copy-On-Write
Page Replacement

Allocation of Frames

Thrashing

Allocation of Frames

- How do we allocate the **fixed amount of free memory** among the various processes?
- Simple Case
  - Single user system; 128 KB memory, 1KB page size
  - **OS occupying 35 KB => 93 frames for the user process**
  - **Pure demand paging**: all 93 frames are put on free-frame list
  - When a user process started execution, it would generate a sequence of page faults.
    - **The first 93 page faults would all get free frames from the free-frame list**.
  - When the free-frame list was exhausted, a **page-replacement** algorithm would be used to select one of the 93 in-memory pages to be replaced with the 94^{th}, and so on.
  - When the **process terminated**, the 93 frames would once again be placed on the free-frame list.

Allocation of Frames – Minimum Number of Frames

- Each process needs ***minimum*** number of frames
  - Defined by the architecture of the system
    - Minimum number of pages required for an instruction
- ***Maximum*** # of frames = Total frames in the system
- Two major allocation schemes
  - **Fixed allocation**
  - **Priority allocation**
  - Many variations

Fixed Allocation

- **Equal allocation**
  - If there are ***m*** frames (after allocating frames for the OS) and ***n*** processes, give each process ***m/n*** frames
  - Keep some as free frame buffer pool
- **Proportional allocation**
  - Allocate according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change

$s_i =$ size of process p_i

$S = \sum s_i$

$m =$ total number of frames

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

$m = 64$

$s_1 = 10$

$s_2 = 127$

$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$

Priority Allocation

- Use a proportional allocation scheme using priorities rather than size (or both priority and size)
- If process P_i generates a page fault
  - Select for replacement one of its frames

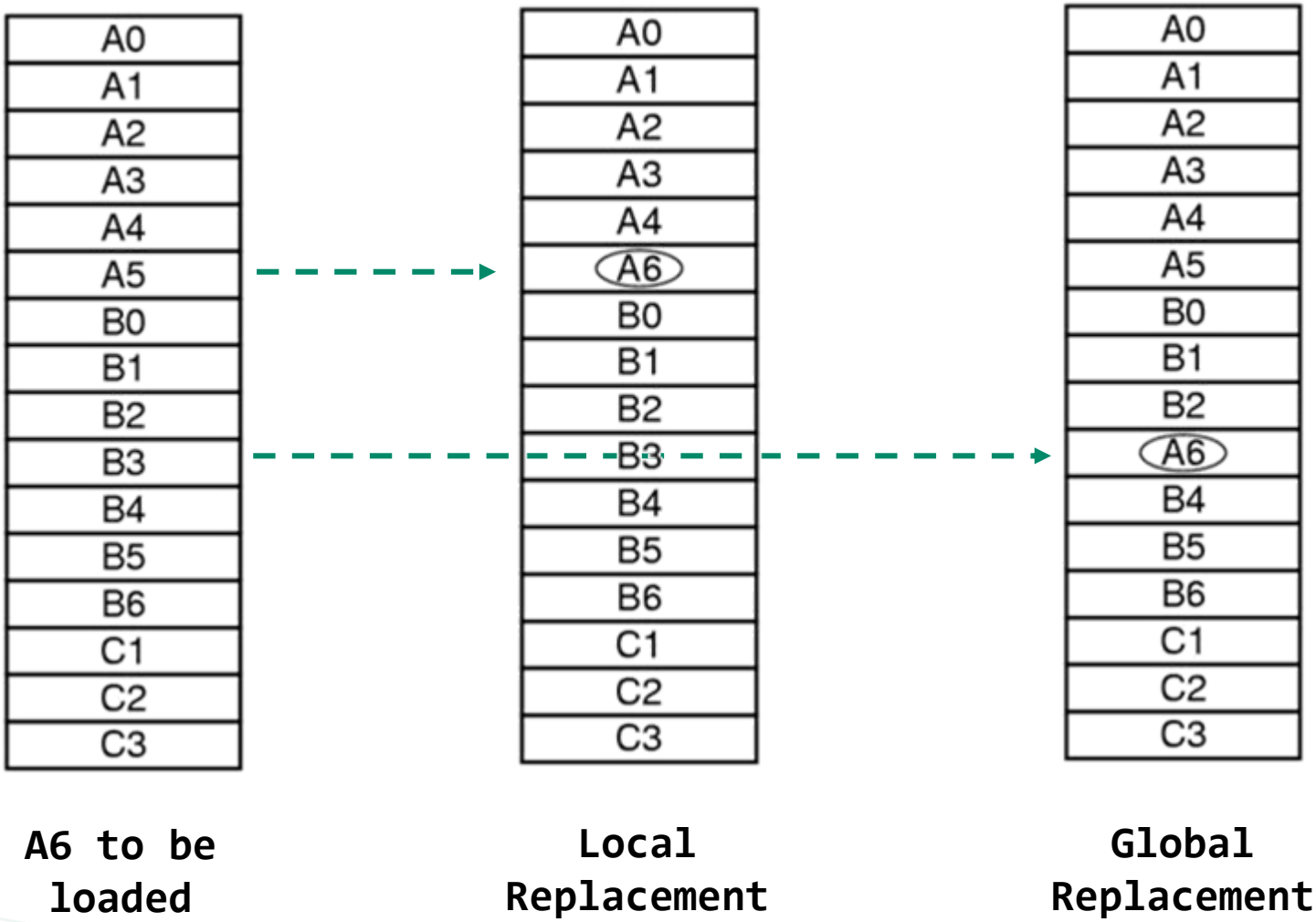
  OR

  - 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
  - \- Process execution time can vary greatly
    - Set of pages in a memory for one process depends on the paging behavior of all processes
  - \+ Greater throughput so more commonly used
- **Local replacement**: each process selects **from only its own set of allocated frames**
  - \+ More consistent per-process performance
  - \- Possibly underutilized memory

Global vs. Local Allocation Example

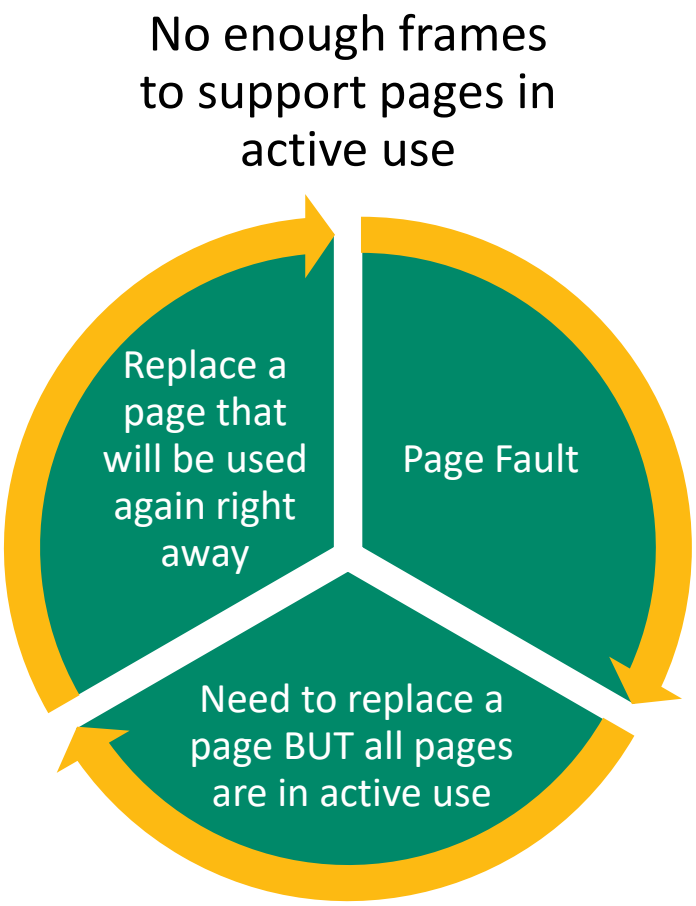


Background
Demand Paging
Copy-On-Write
Page Replacement
Allocation of Frames

Thrashing

Thrashing

- If a process does not have "enough" frames to support pages in active use => Page Fault
  - Need to replace some page
  - BUT <u>all pages are in active use</u> ➔ *Replace a page that will be needed again* ➔ Fault again
  - Results
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system



Thrashing ≡ a process is busy swapping pages in and out
Spending more time on paging than executing

Thrashing - Scenario

1. The operating system **monitors CPU utilization**.
2. If CPU utilization is too low, increase the degree of multiprogramming by introducing a new process to the system.
3. A **global page-replacement** algorithm is used
   - It replaces pages without regard to the process to which they belong.

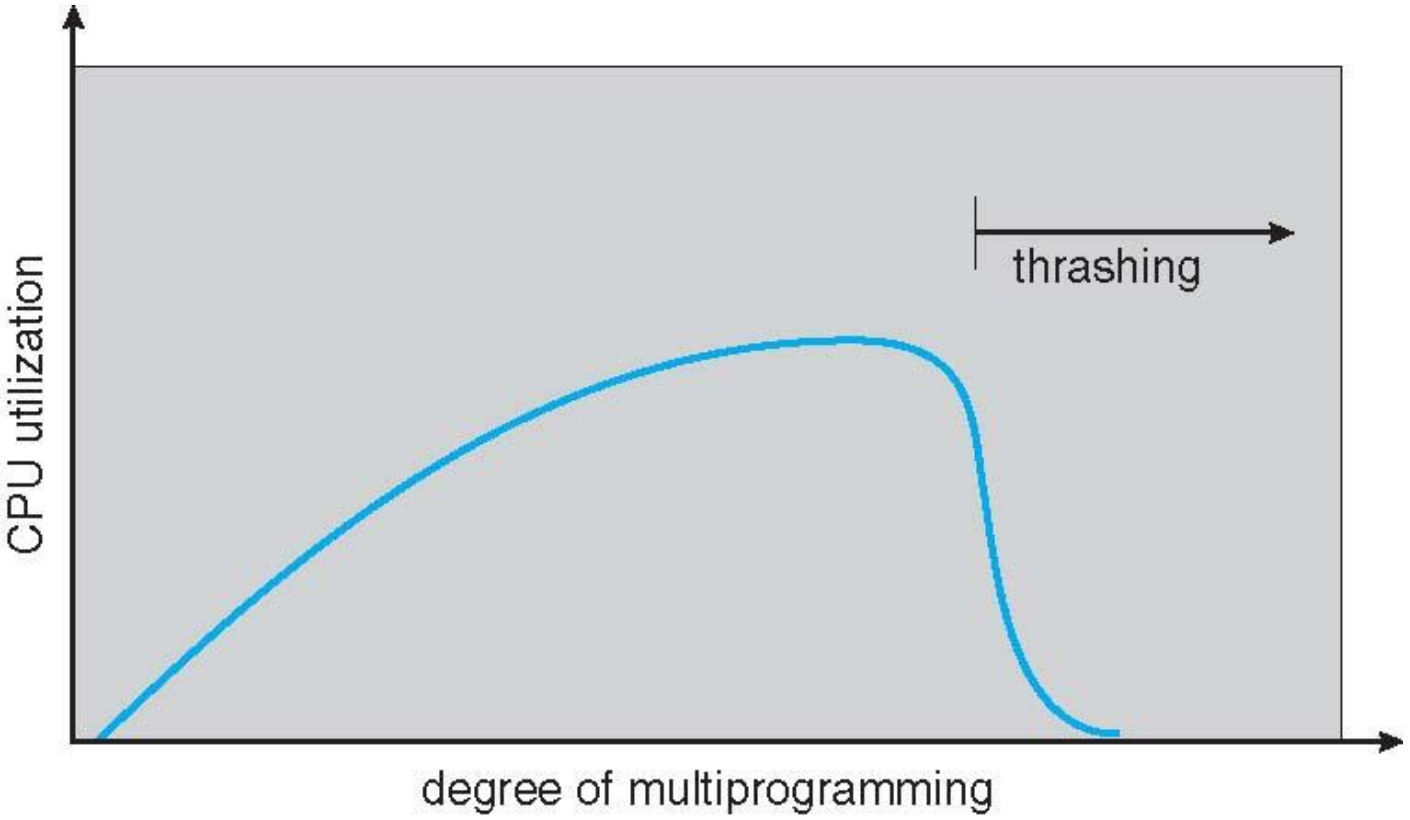
Thrashing – Scenario (cont'd)

4. Suppose that a process **needs more frames**
5. It starts faulting and **taking frames away from other processes** (global page replacement)
6. These processes need those pages=> **they also fault** => **taking frames from other processes**
7. Faulting processes must **use the paging device to swap pages in and out** => They **queue up for the paging device** => the **ready queue** empties.
8. As processes wait for the paging device, CPU utilization decreases

Thrashing – Scenario (cont'd)

9. The CPU scheduler sees the decreasing CPU utilization and ***increases*** the degree of multiprogramming as a result.
10. The new process tries to get started by taking frames from running processes, causing more page faults and a longer queue for the paging device.
11. As a result, CPU utilization drops even further, and the CPU scheduler tries to ***increase*** the degree of multiprogramming even more
12. Thrashing has occurred
    - System throughput drops.
    - Page-fault rate increases tremendously => Effective memory-access time increases.
      - No work is getting done, the processes are spending all their time paging.

Thrashing (Cont.)

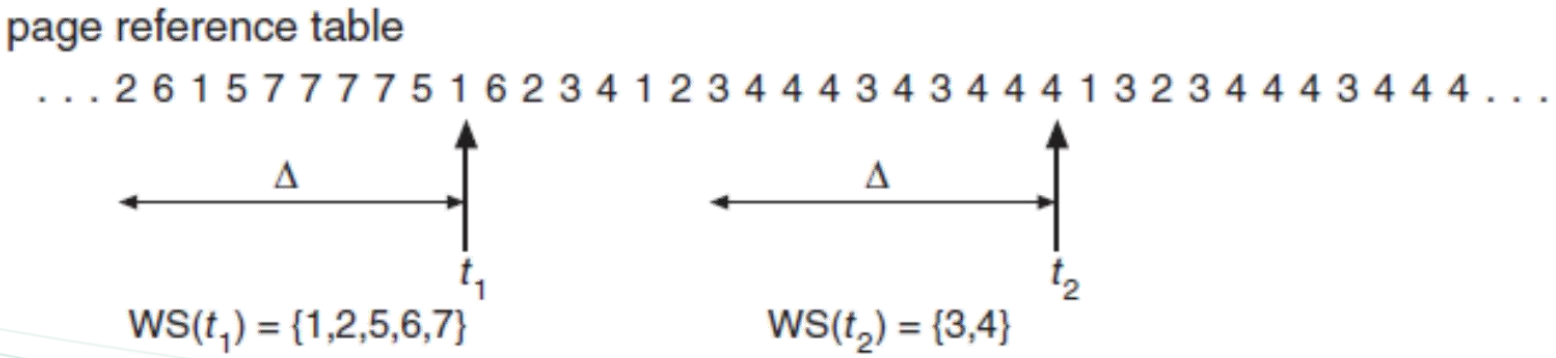


Working-Set Model

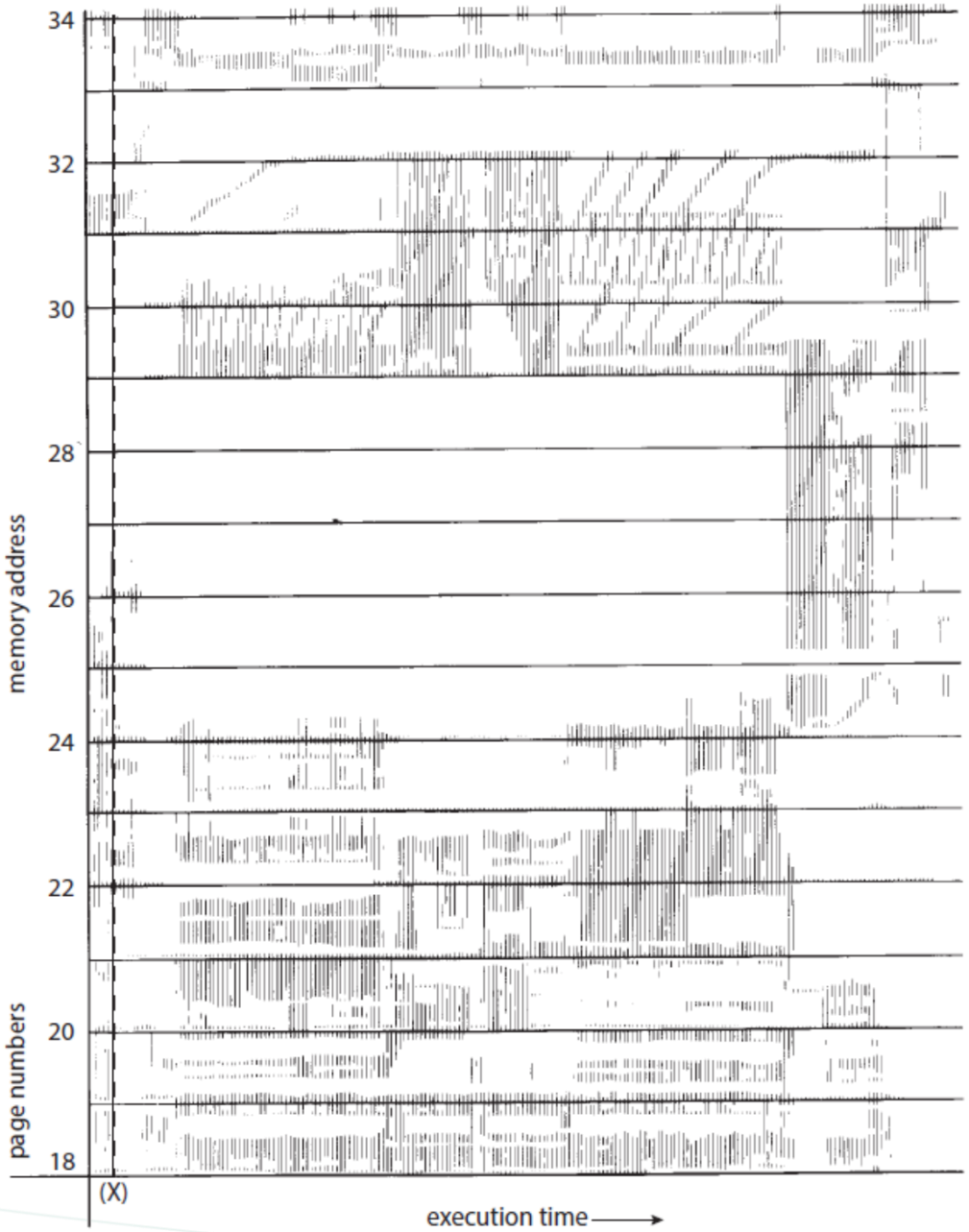
- **How to prevent Thrashing?**
  - Provide a process with as many frames as it needs.
- How do we know how many frames it "needs"?
  - By **looking at how many frames a process is actually using**.
- This approach defines the locality model of process execution.
  - As a process executes, it **moves from locality to locality**.
  - A locality is a **set of pages that are actively used together**
  - A **program** is generally composed of several different localities, which may overlap
- Thrashing occurs when Σ *size of locality > total memory size*

Working-Set Model (cont'd)

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
- WSS_i **(working set of Process** P_i**) =**
  total number 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*
- $D = \Sigma WSS_i \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing (m is the number of available frames)
- **Policy:** if $D > m$, then suspend or swap out one of the processes



Working-Set Model (cont'd)



Page-Fault Frequency

- More direct approach than WSS
- 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