

# Operating Systems

Isfahan University of Technology  
Electrical and Computer Engineering Department

Zeinab Zali

Virtual Memory



# 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
- 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





# 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 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 running concurrently
  - Less I/O needed to load or swap processes



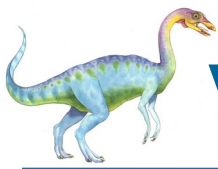


# Background (Cont.)

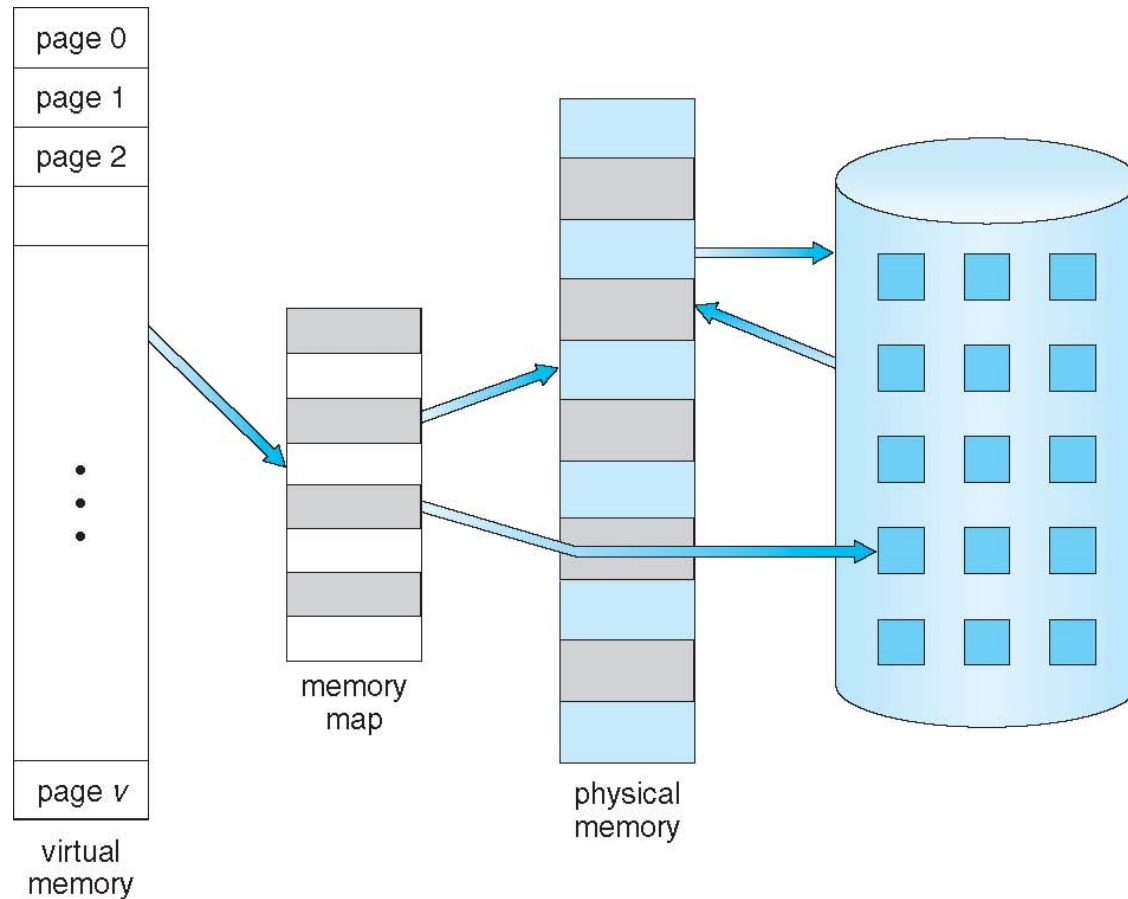
---

- **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**





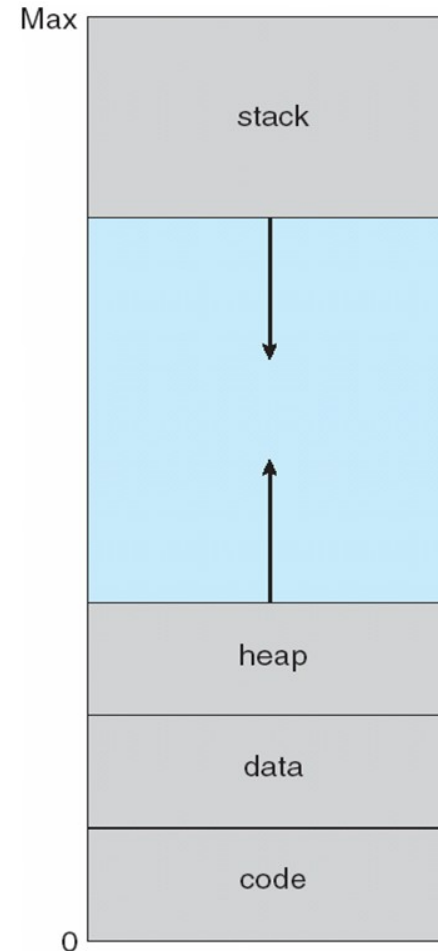
# Virtual Memory That is Larger Than Physical Memory





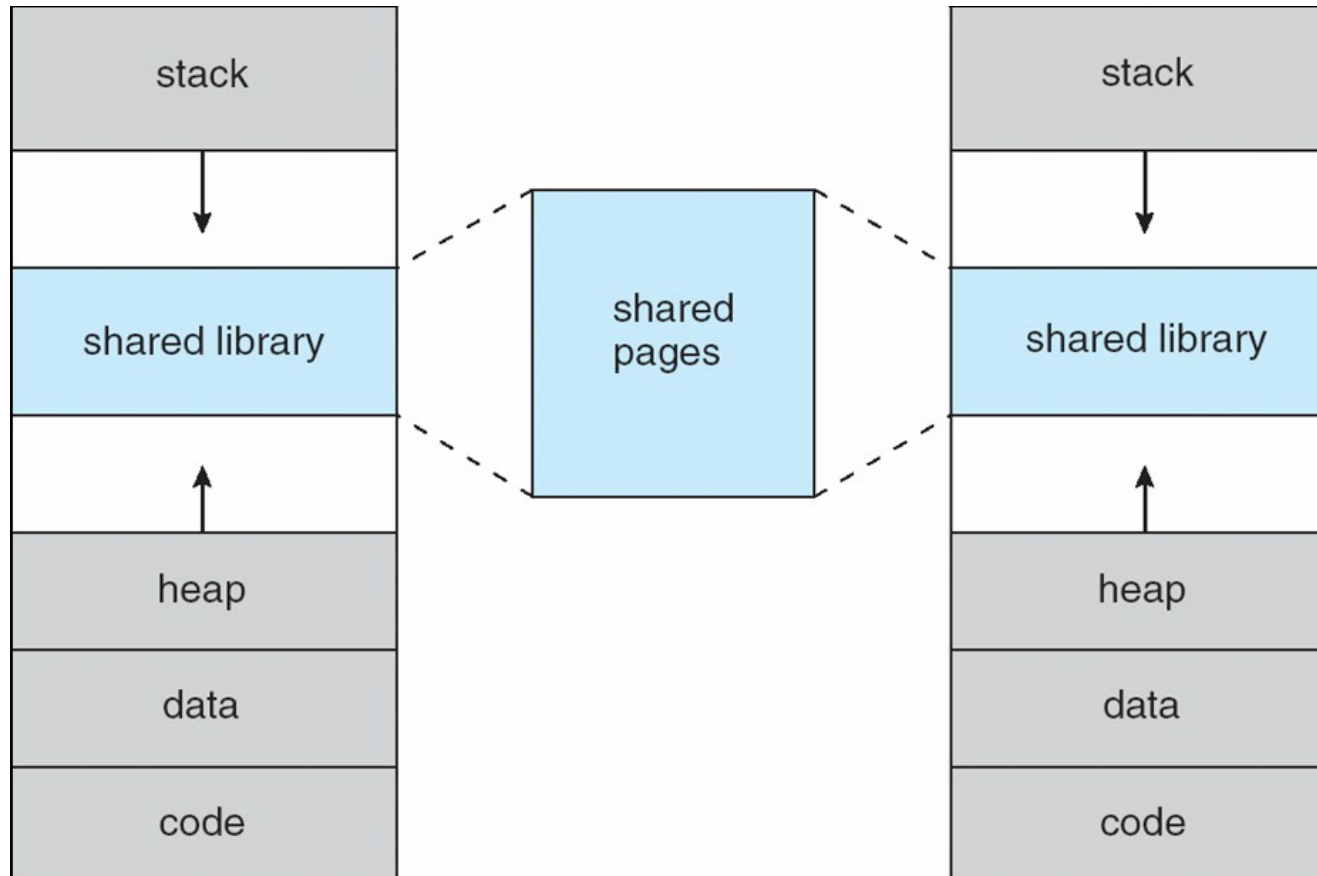
# Virtual-address Space

- 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
- 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





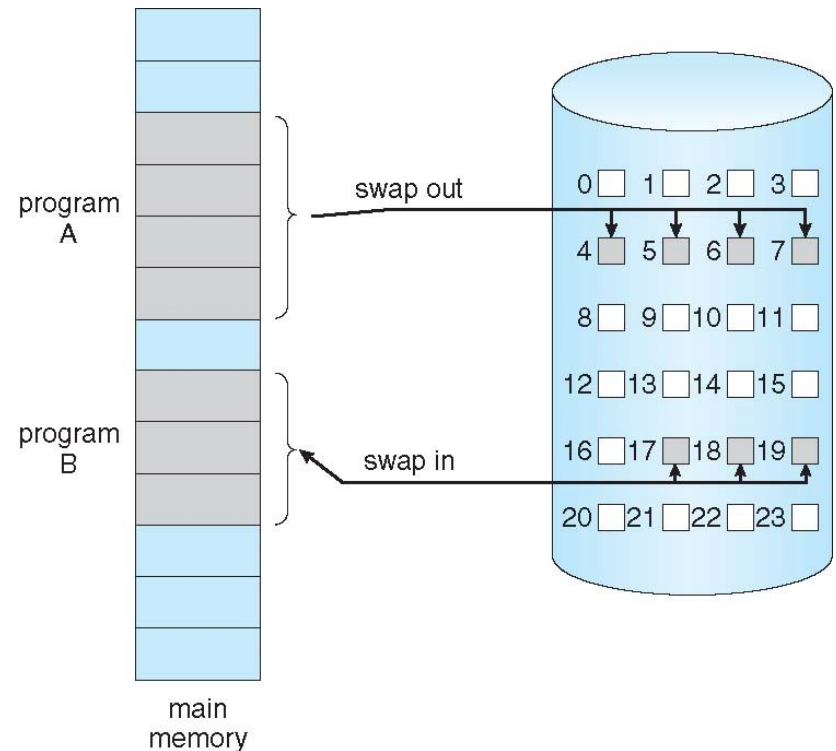
# Shared Library Using Virtual Memory





# Demand Paging

- Could 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
- Similar to paging system with swapping
- Page is needed  $\Rightarrow$  reference to it
  - invalid reference  $\Rightarrow$  abort
  - not-in-memory  $\Rightarrow$  bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**







# Basic Concepts

---

- With swapping, pager guesses which pages will be used before swapping out again
- pager brings in only those pages into memory
- 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 and 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**  $\Rightarrow$  in-memory – **memory resident**, **i**  $\Rightarrow$  not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:

| Frame # | valid-invalid bit |
|---------|-------------------|
|         |                   |
|         | <b>v</b>          |
|         | <b>v</b>          |
|         | <b>v</b>          |
|         | <b>i</b>          |
| ...     |                   |
|         | <b>i</b>          |
|         | <b>i</b>          |

page table

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





# Page Table When Some Pages Are Not in Main Memory

|   |   |
|---|---|
| 0 | A |
| 1 | B |
| 2 | C |
| 3 | D |
| 4 | E |
| 5 | F |
| 6 | G |
| 7 | H |

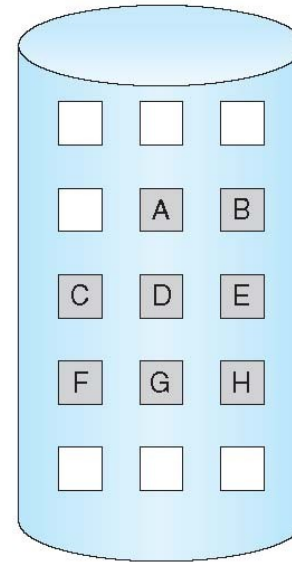
logical  
memory

| valid-invalid<br>bit |   |   |
|----------------------|---|---|
| frame                |   |   |
| 0                    | 4 | v |
| 1                    |   | i |
| 2                    | 6 | v |
| 3                    |   | i |
| 4                    |   | i |
| 5                    | 9 | v |
| 6                    |   | i |
| 7                    |   | i |

page table

|    |
|----|
| 0  |
| 1  |
| 2  |
| 3  |
| 4  |
| 5  |
| 6  |
| 7  |
| 8  |
| 9  |
| 10 |
| 11 |
| 12 |
| 13 |
| 14 |
| 15 |

physical memory





# Page Fault

---

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

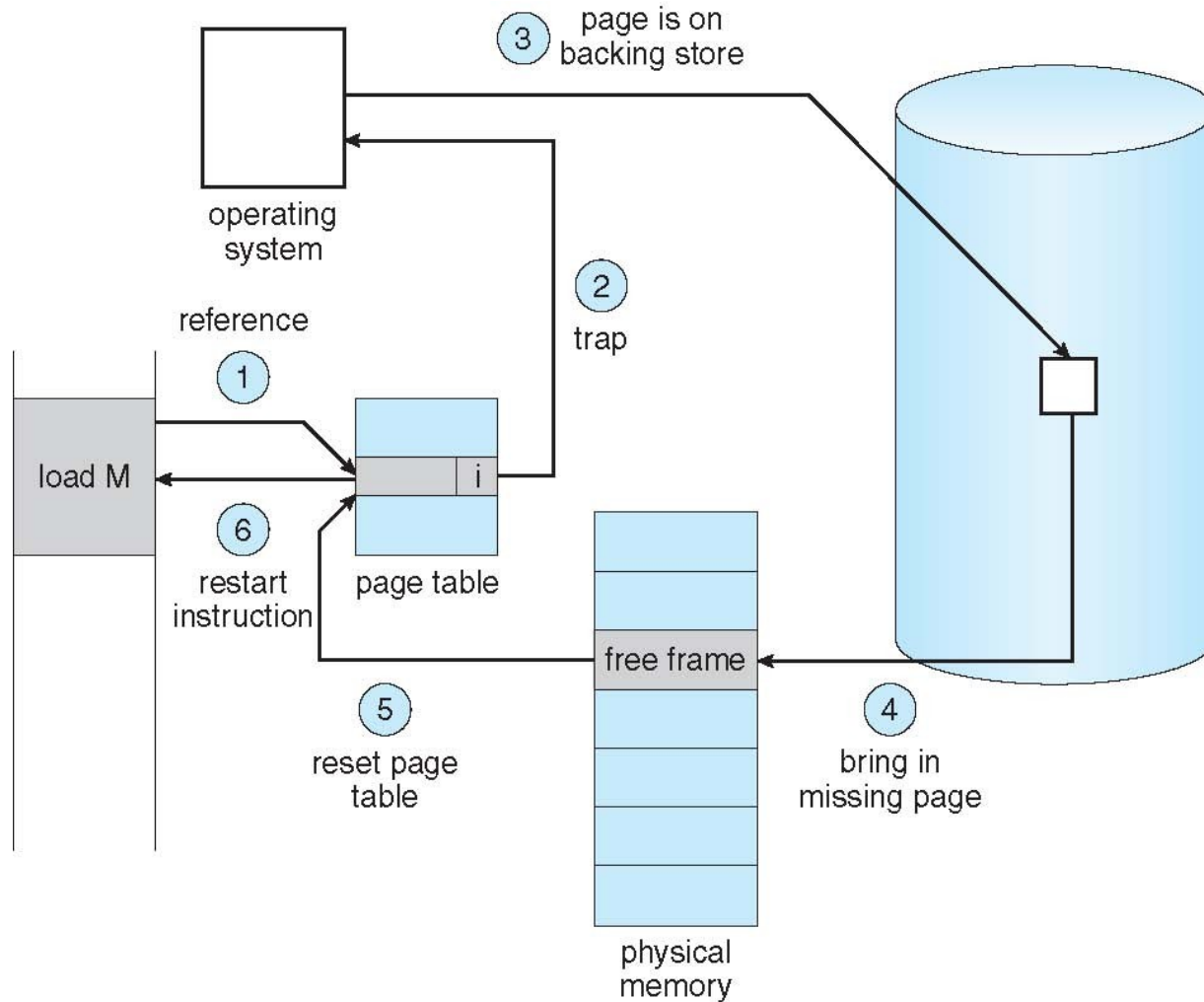
## page fault

1. Operating system looks at another table to decide:
  - Invalid reference  $\Rightarrow$  abort
  - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory  
Set validation bit = **v**
5. Restart the instruction that caused the page fault





# Steps in Handling a Page Fault





# 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





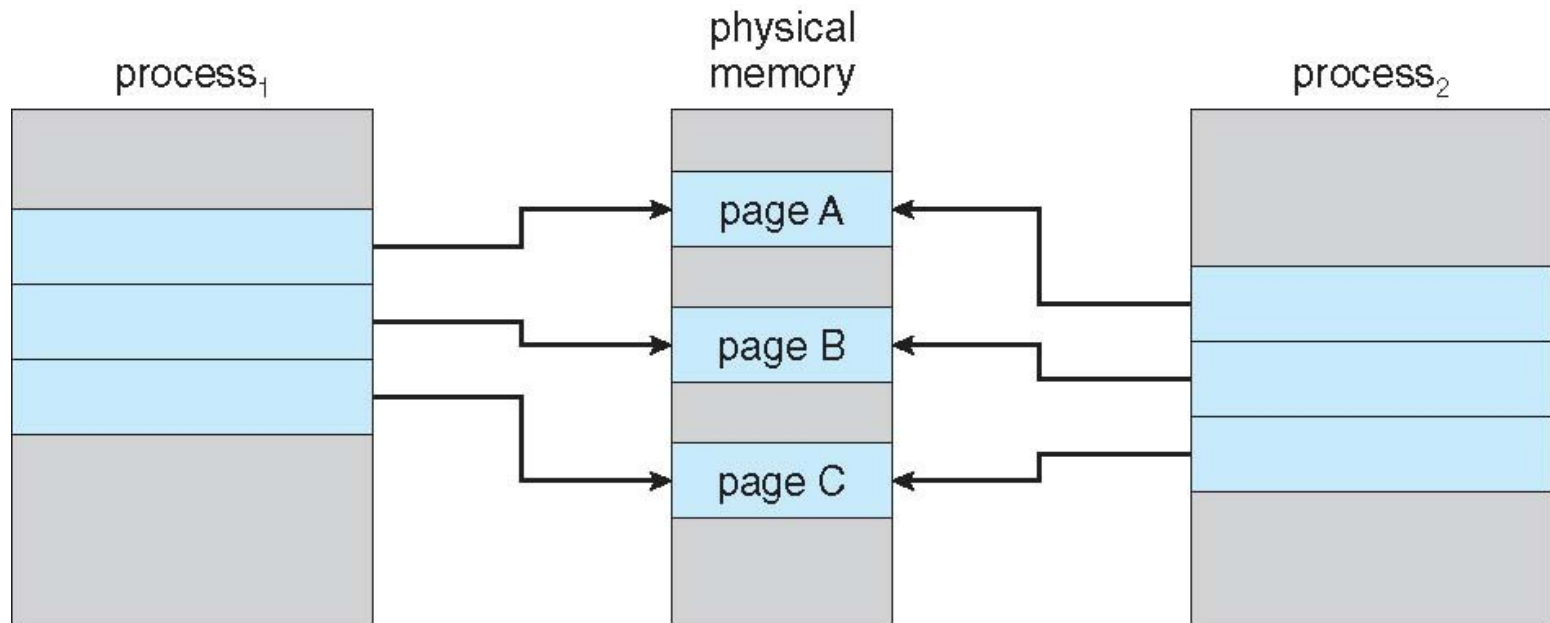
# Copy-on-Write

- **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
- `vfork()` variation on `fork()` system call has parent suspend and child using address space of parent
  - Designed to have child call `exec()`
  - Very efficient
  - Ex. for programming a shell





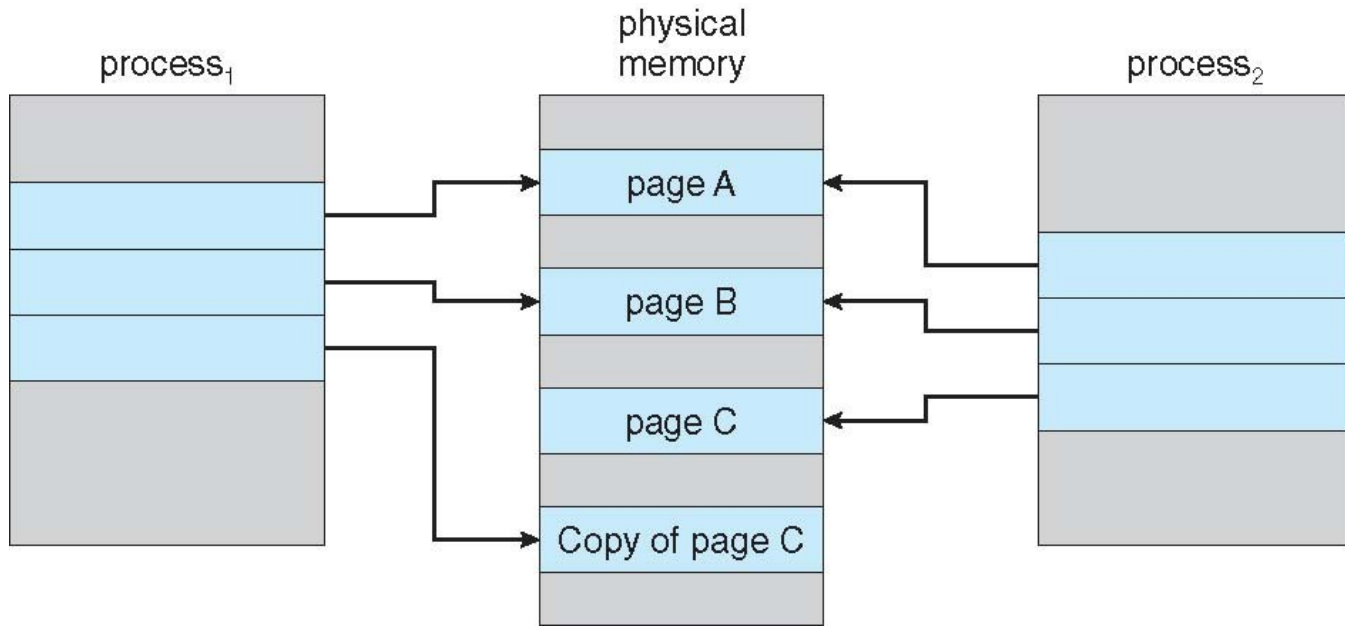
# Before Process 1 Modifies Page C







# After Process 1 Modifies Page C





# What Happens if There is no Free Frame?

---

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Page replacement – find some page in memory, but not really in use, 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



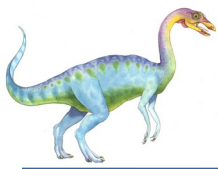


# Page Replacement

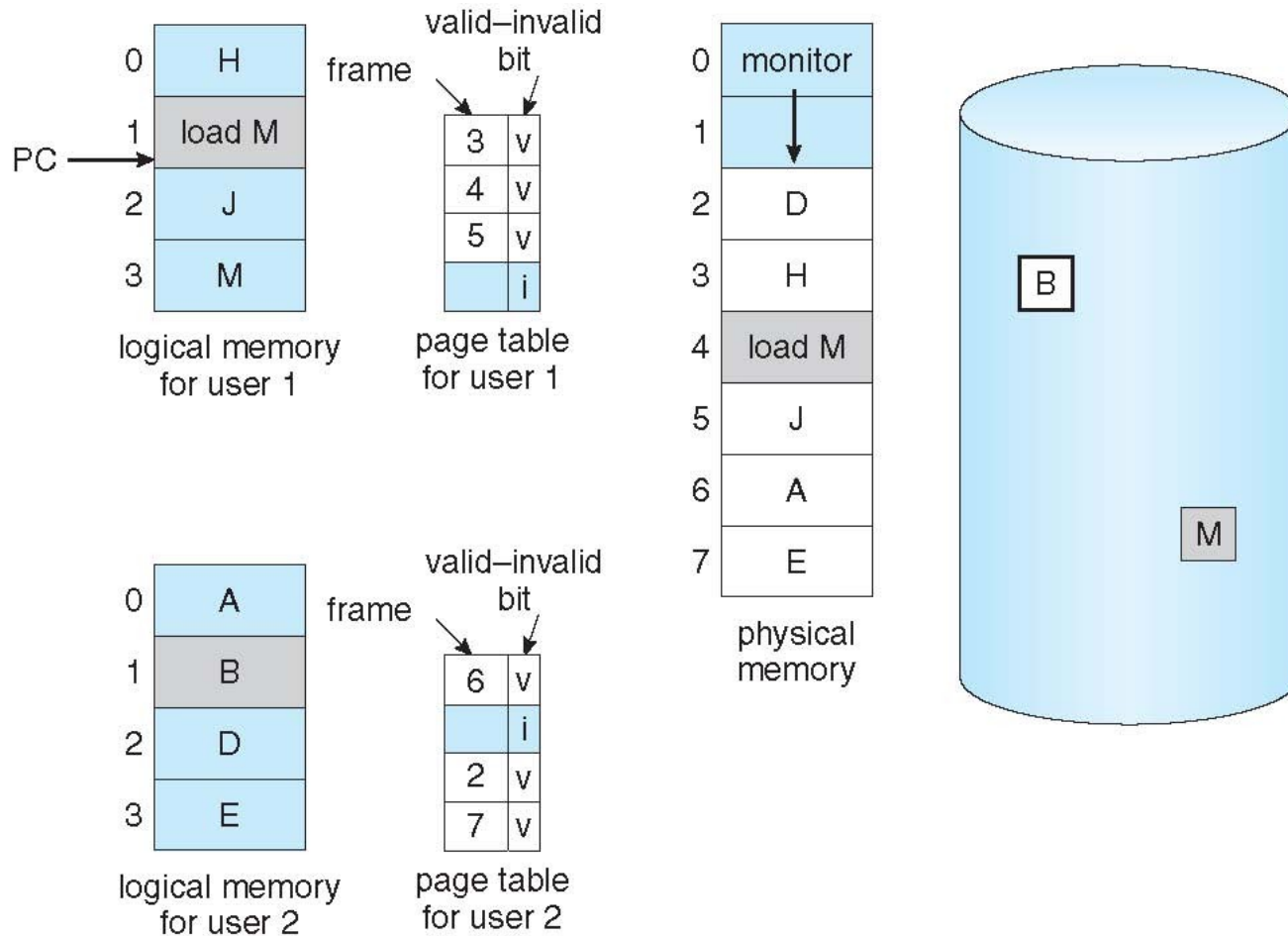
---

- Prevent **over-allocation** of memory 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 – large virtual memory can be provided on a smaller physical memory





# Need For Page Replacement





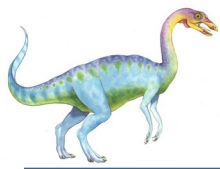
# Basic Page Replacement

---

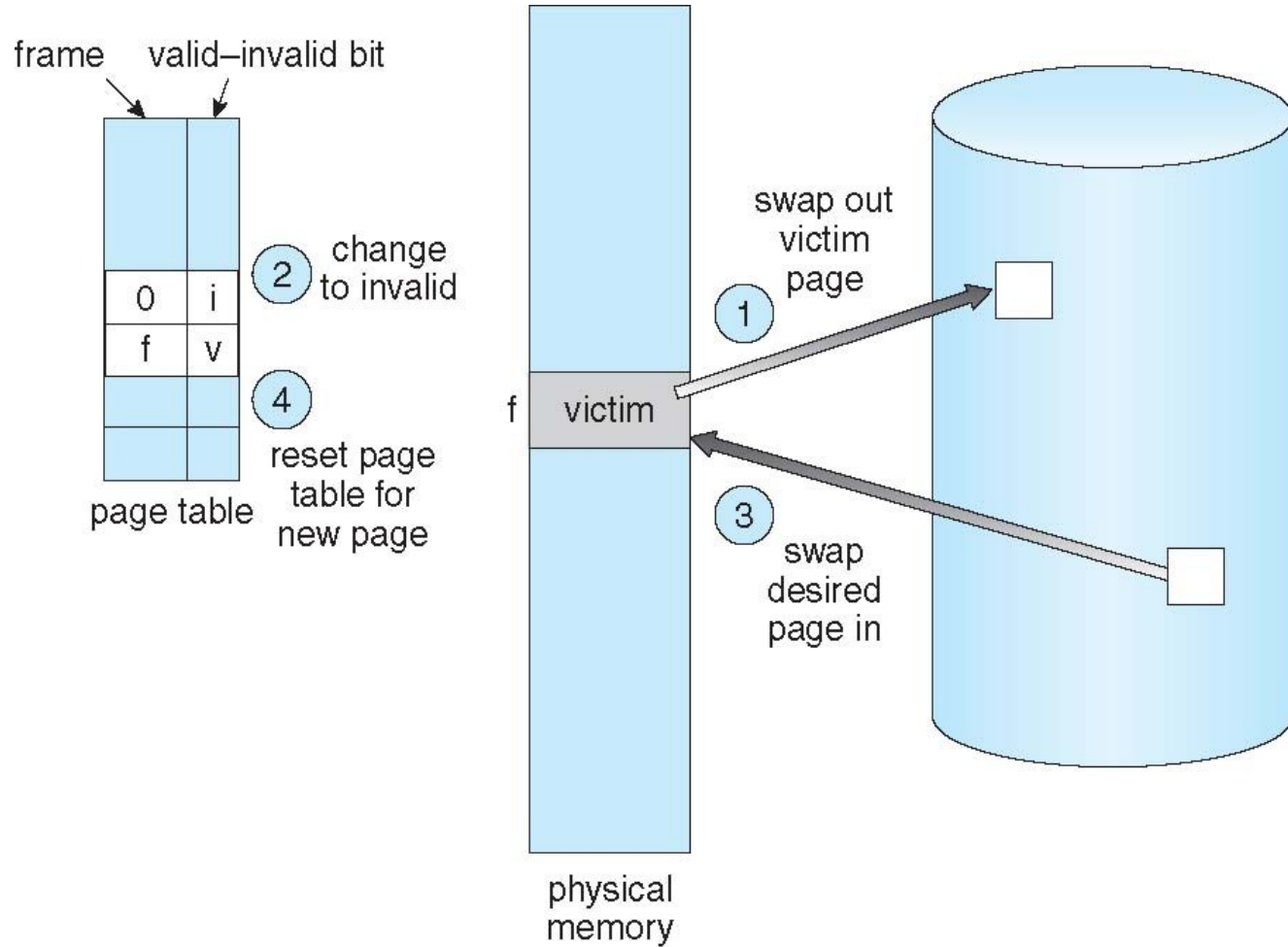
1. Find the location of the desired page on disk
2. Find a free frame:
  - If there is a free frame, use it
  - If there is no free frame, use a page replacement algorithm to select a **victim frame**
    - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame; update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

Note now potentially 2 page transfers for page fault





# Page Replacement





# Page and Frame Replacement Algorithms

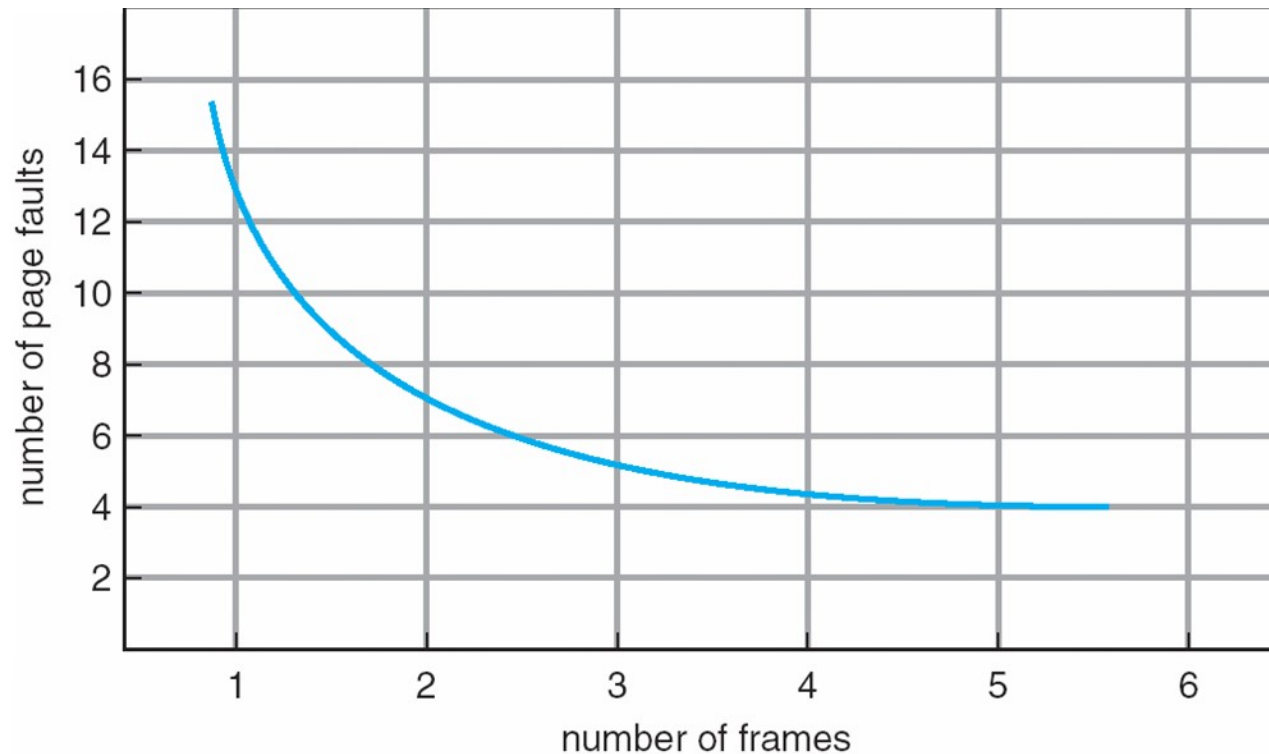
- **Frame-allocation algorithm** determines
  - How many frames to give each process
  - Which frames to replace
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) 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**





# Graph of Page Faults Versus The Number of Frames







# First-In-First-Out (FIFO) Algorithm

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

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

|   |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 7 | 7 | 2 | 2 | 2 | 4 | 4 | 4 | 0 | 0 | 0 | 7 | 7 | 7 |
|   | 0 | 0 | 0 | 3 | 3 | 3 | 2 | 2 | 2 | 1 | 1 | 1 | 0 | 0 |
|   |   | 1 | 1 | 1 | 0 | 0 | 0 | 3 | 3 | 3 | 2 | 2 | 2 | 1 |

page frames

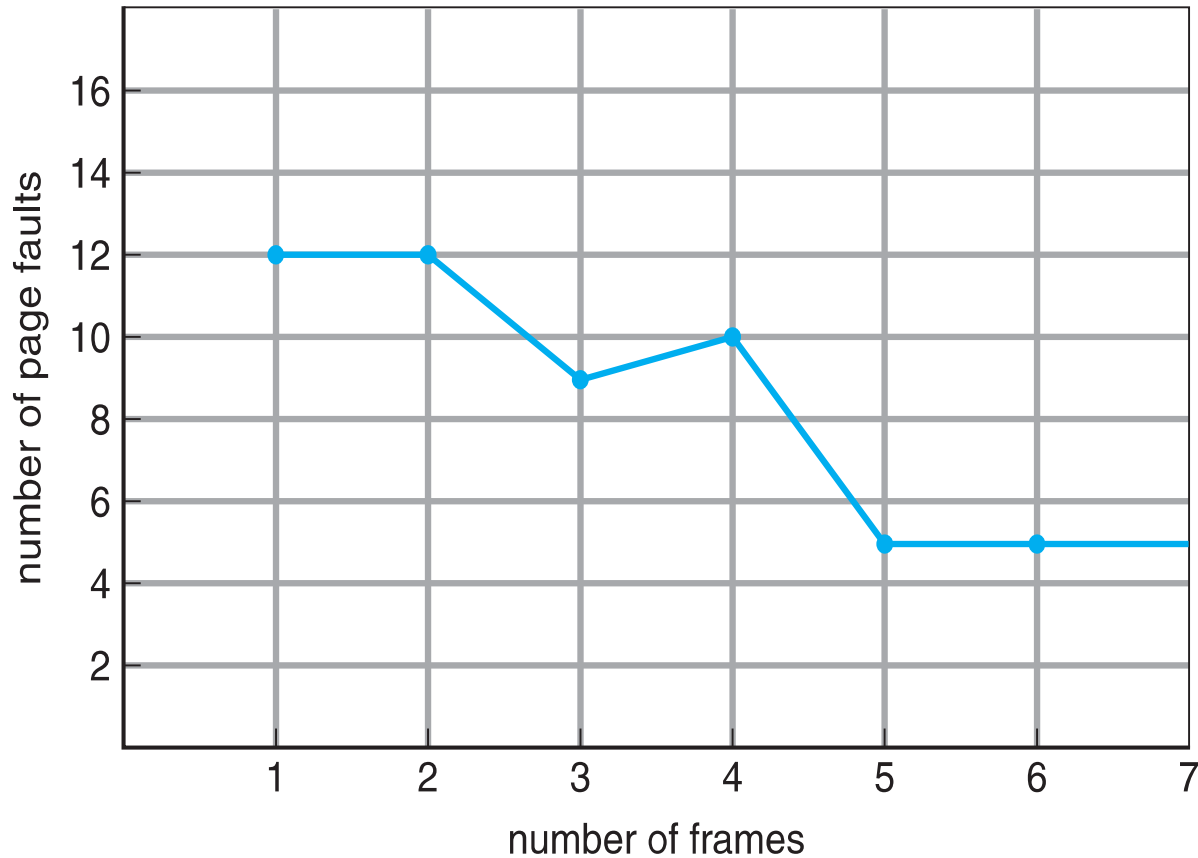
15 page faults

- 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





# Optimal Algorithm

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

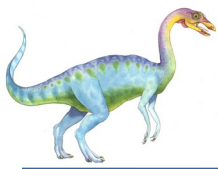
reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

|   |   |   |   |  |   |  |   |  |   |  |   |  |  |  |  |  |   |  |  |
|---|---|---|---|--|---|--|---|--|---|--|---|--|--|--|--|--|---|--|--|
| 7 | 7 | 7 | 2 |  | 2 |  | 2 |  | 2 |  | 2 |  |  |  |  |  | 7 |  |  |
|   | 0 | 0 | 0 |  | 0 |  | 4 |  | 0 |  | 0 |  |  |  |  |  | 0 |  |  |
|   |   | 1 | 1 |  | 3 |  | 3 |  | 3 |  | 1 |  |  |  |  |  | 1 |  |  |

page frames





# 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

reference string

7 0 1 2 0 3 0 4 2 3 0 3 2 1 2 0 1 7 0 1

|   |   |   |   |  |   |  |   |   |   |   |  |  |   |  |   |  |   |  |  |
|---|---|---|---|--|---|--|---|---|---|---|--|--|---|--|---|--|---|--|--|
| 7 | 7 | 7 | 2 |  | 2 |  | 4 | 4 | 4 | 0 |  |  | 1 |  | 1 |  | 1 |  |  |
|   | 0 | 0 | 0 |  | 0 |  | 0 | 0 | 3 | 3 |  |  | 3 |  | 0 |  | 0 |  |  |
|   |   | 1 | 1 |  | 3 |  | 3 | 2 | 2 | 2 |  |  | 2 |  | 2 |  | 7 |  |  |

page frames

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





# LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - ▶ Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a double link form:
  - Page referenced:
    - ▶ move it to the top
    - ▶ requires 6 pointers to be changed (for 6 virtual page)
  - But each update more expensive
  - No search for replacement
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly





# Use Of A Stack to Record Most Recent Page References

reference string

4 7 0 7 1 0 1 2 1 2 7 1 2

|   |
|---|
| 2 |
| 1 |
| 0 |
| 7 |
| 4 |

stack  
before  
a

|   |
|---|
| 7 |
| 2 |
| 1 |
| 0 |
| 4 |

stack  
after  
b

↑  
a

↑  
b





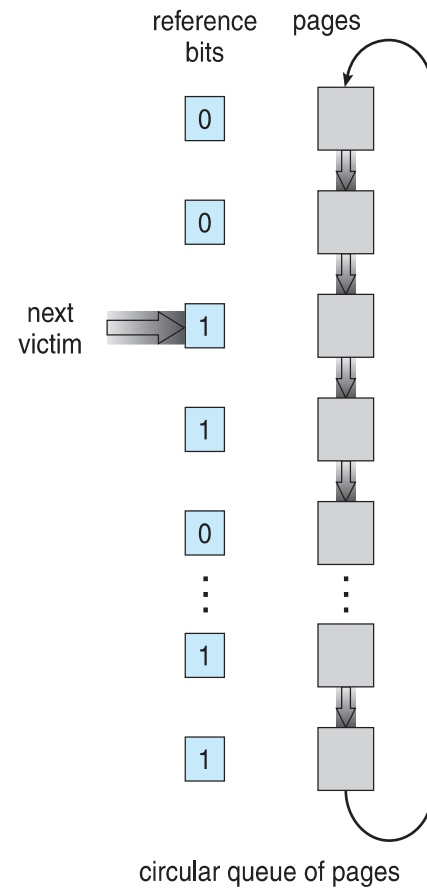
# LRU Approximation Algorithms

- LRU needs special hardware and still slow
- **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 one exists)
    - ▶ We do not know the order, however
- **Second-chance algorithm**
  - Generally FIFO, plus hardware-provided reference bit
  - **Clock** replacement
  - If page to be replaced has
    - ▶ Reference bit = 0 -> replace it
    - ▶ reference bit = 1 then:
      - set reference bit 0, leave page in memory
      - replace next page, subject to same rules

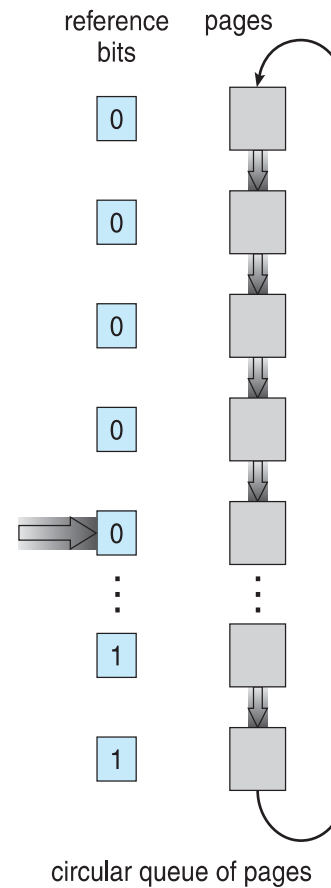




# Second-Chance (clock) Page-Replacement Algorithm



(a)



(b)







# Allocation of Frames

---

- Each process needs ***minimum*** number of frames
  - The minimum number of frames is defined by the computer architecture
  - we must have enough frames to hold all the different pages that any single instruction can reference.
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations





# 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 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
- If process  $P_i$  generates a page fault,
  - select for replacement one of its frames (**local replacement**)
  - select for replacement a frame from a process with lower priority number (**global replacement**)





# 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
  - But then process execution time can vary greatly
  - But greater throughput so more common
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory





# Reclaiming Pages

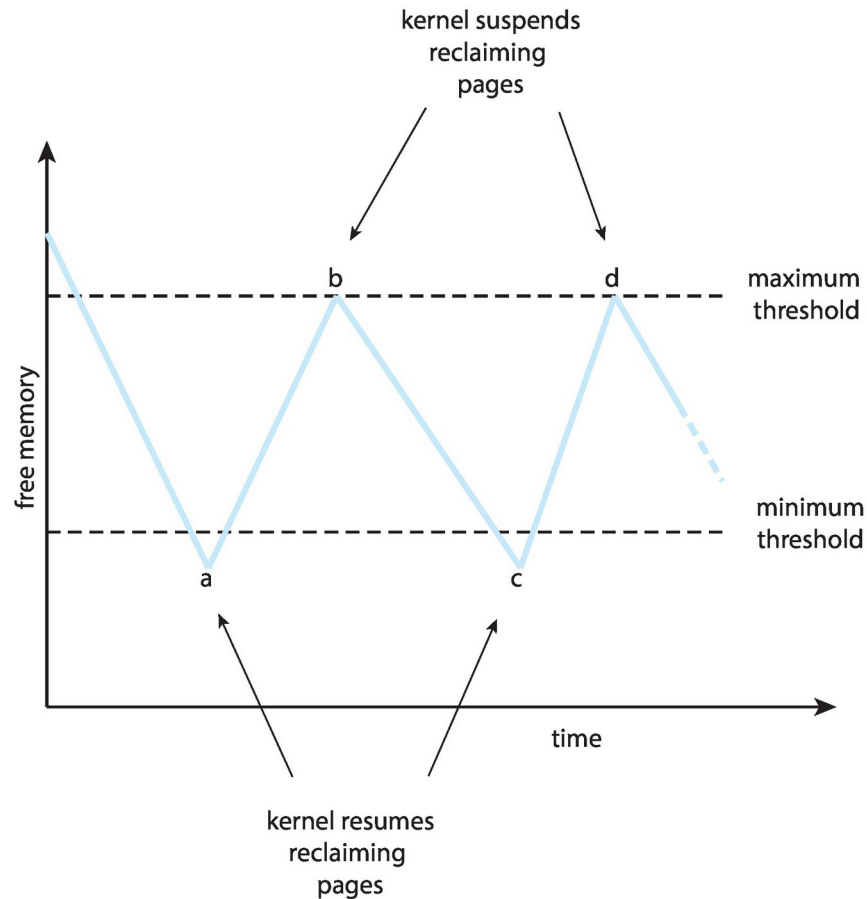
---

- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the **free-frame list**, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.





# Reclaiming Pages Example





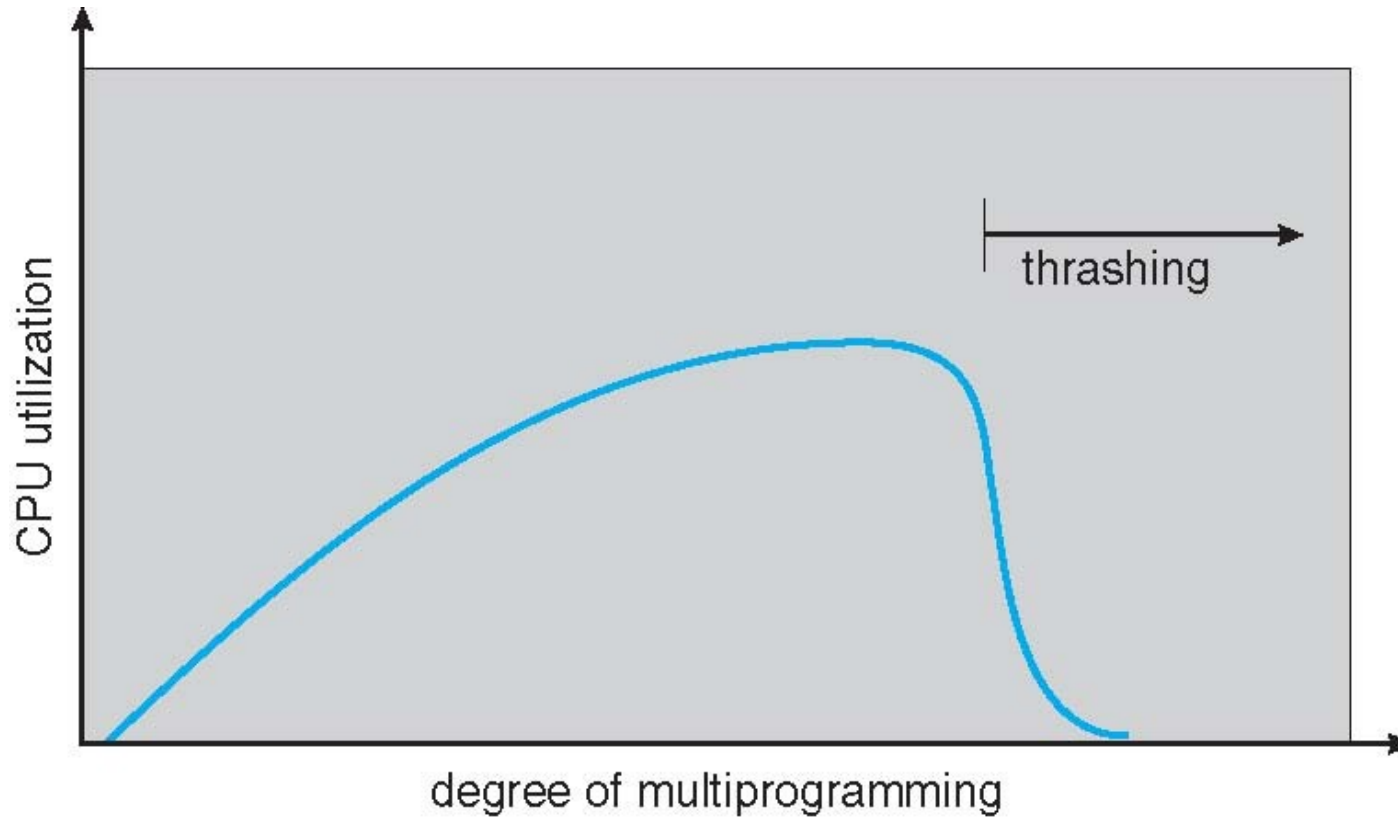
# Thrashing

- If a process does not have “enough” pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - ▶ Low CPU utilization
    - ▶ Operating system thinking that it needs to increase the degree of multiprogramming
    - ▶ Another process added to the system
- **Thrashing**  $\equiv$  a process is busy swapping pages in and out





# Thrashing (Cont.)







# Demand Paging and Thrashing

- Why does demand paging work?

## Locality model

- Process migrates from one locality to another
- Localities may overlap

- Why does thrashing occur?

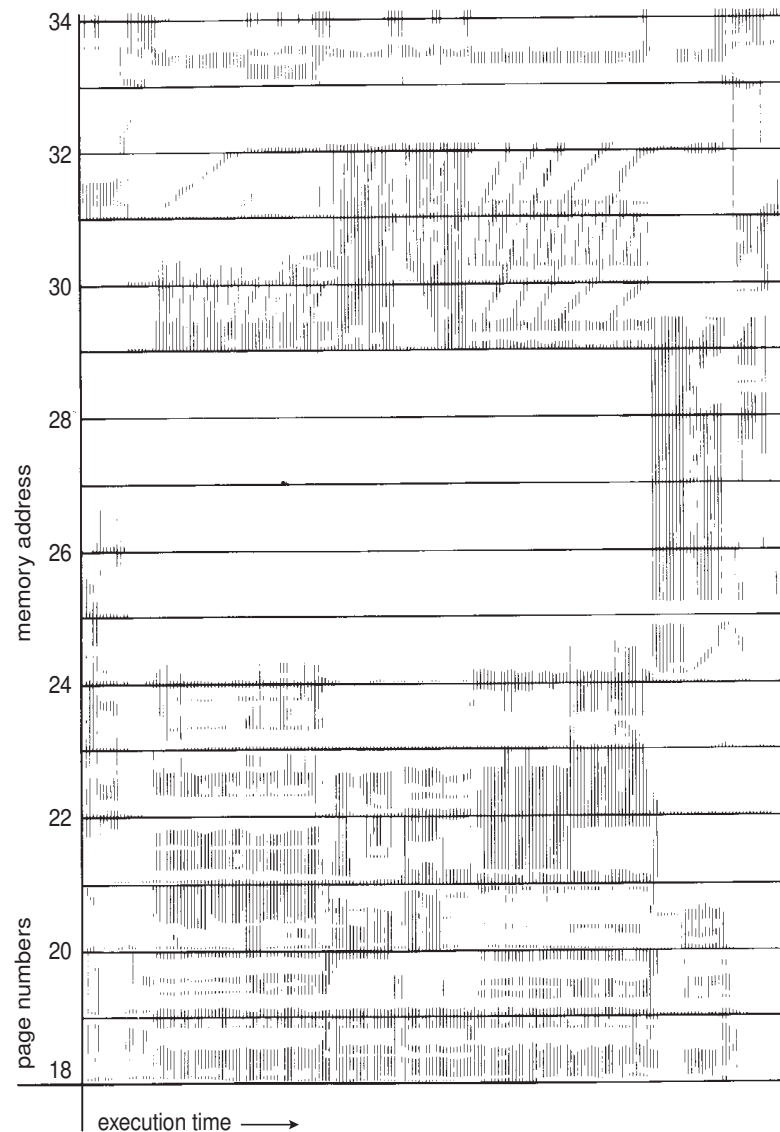
$\Sigma$  size of locality > total memory size

- Limit effects by using local or priority page replacement





# Locality In A Memory-Reference Pattern





# Allocating Kernel Memory

---

- Treated differently from user memory
- Often allocated from a free-memory pool
  - Kernel requests memory for structures of varying sizes
    - ▶ As a result, the kernel must use memory conservatively and attempt to minimize waste due to fragmentation
  - Some kernel memory needs to be contiguous
    - ▶ i.e., for device I/O



# End of Chapter 10

---

