- Introduction to Virtual Memory
- Demand Paging
- Process creation: Copy- on write

Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples

Objective

- 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
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed

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

Virtual Memory

- Feature of OS
- Enables to run larger process with smaller available memory

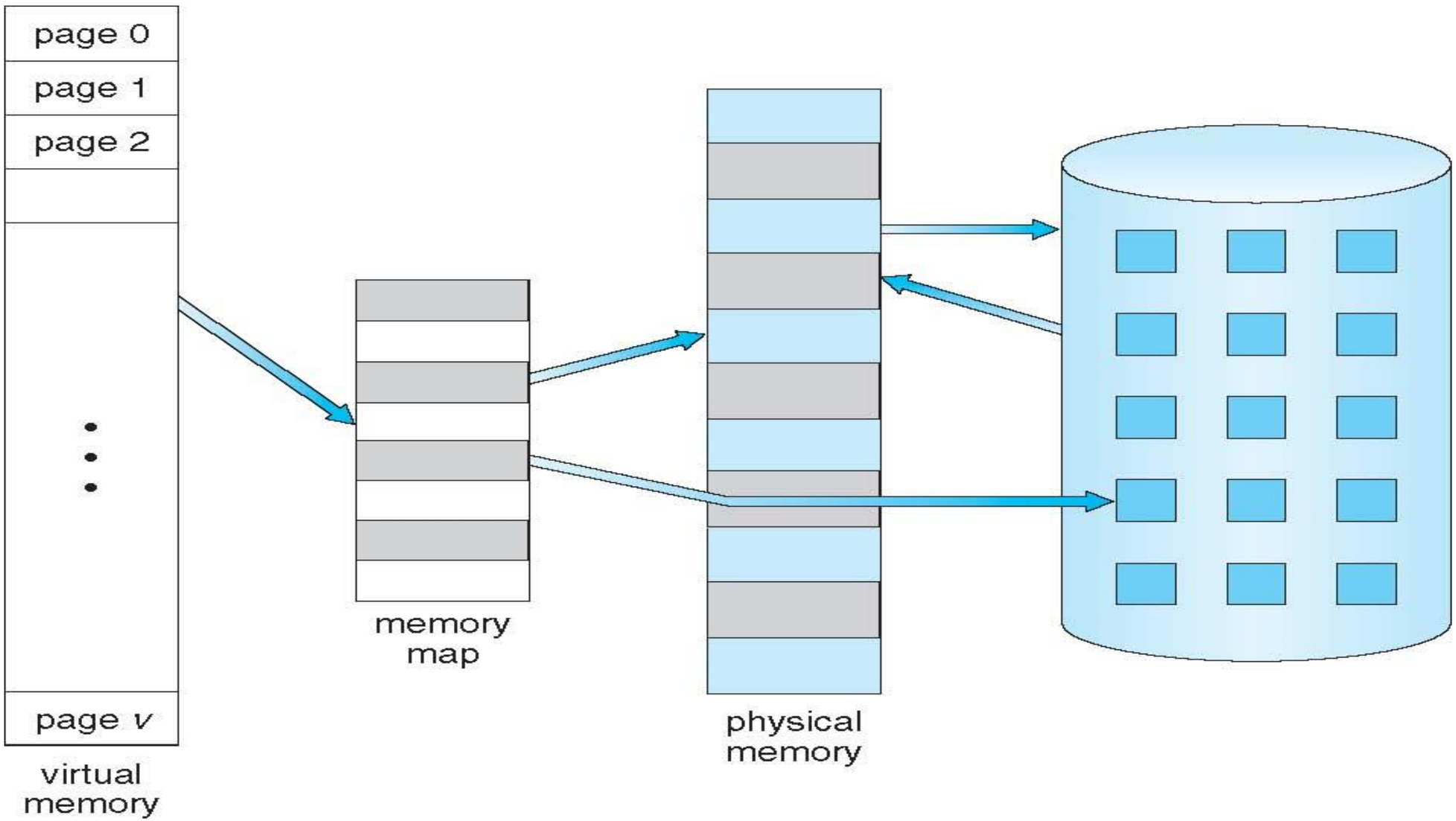
CHITKARA UNIVERSITY

- **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

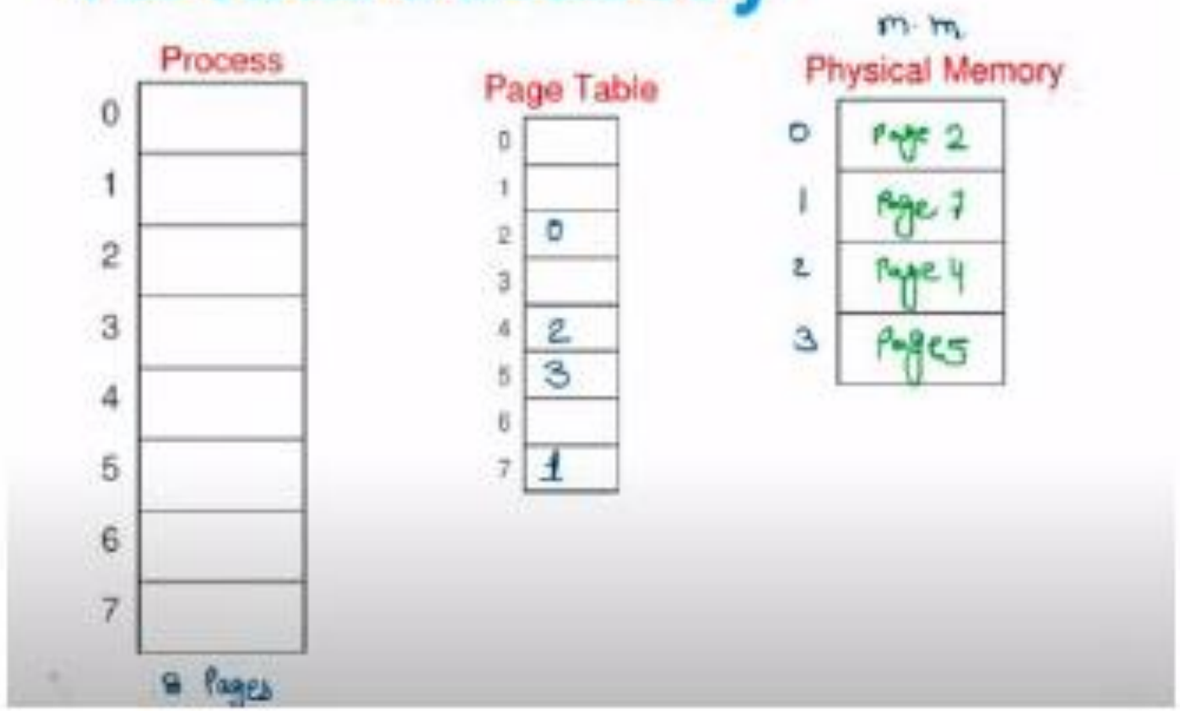
Cont..

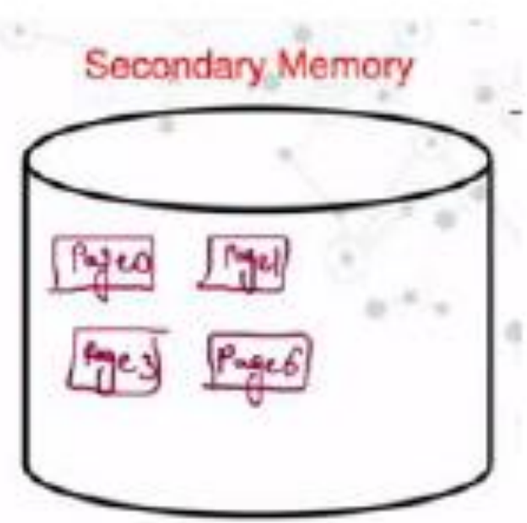
- **Virtual address space** – logical view of how process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging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

Virtual Memory That is Larger Than Physical Memory



Virtual Memory



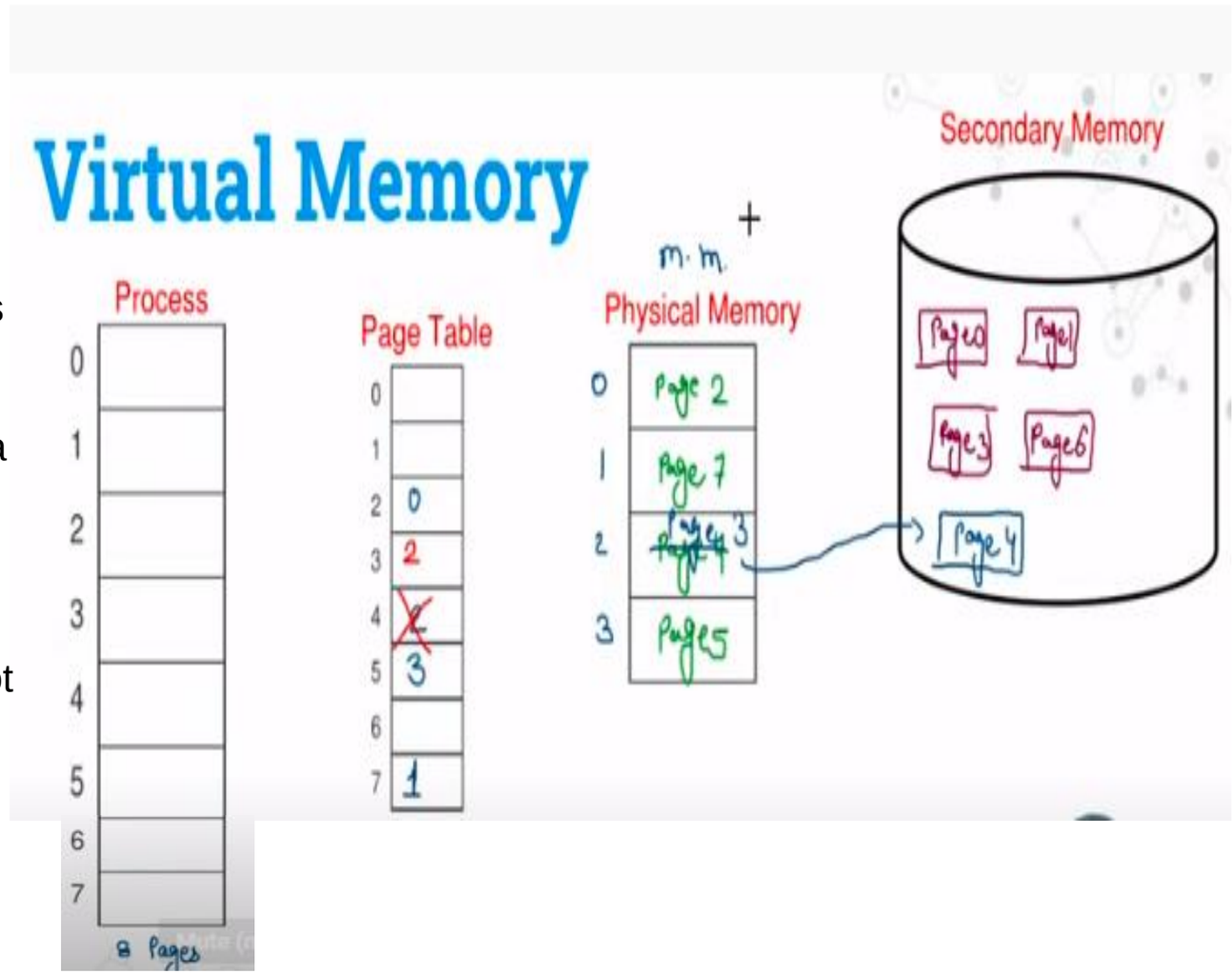


Demand Paging

- Bring pages in memory when CPU demands

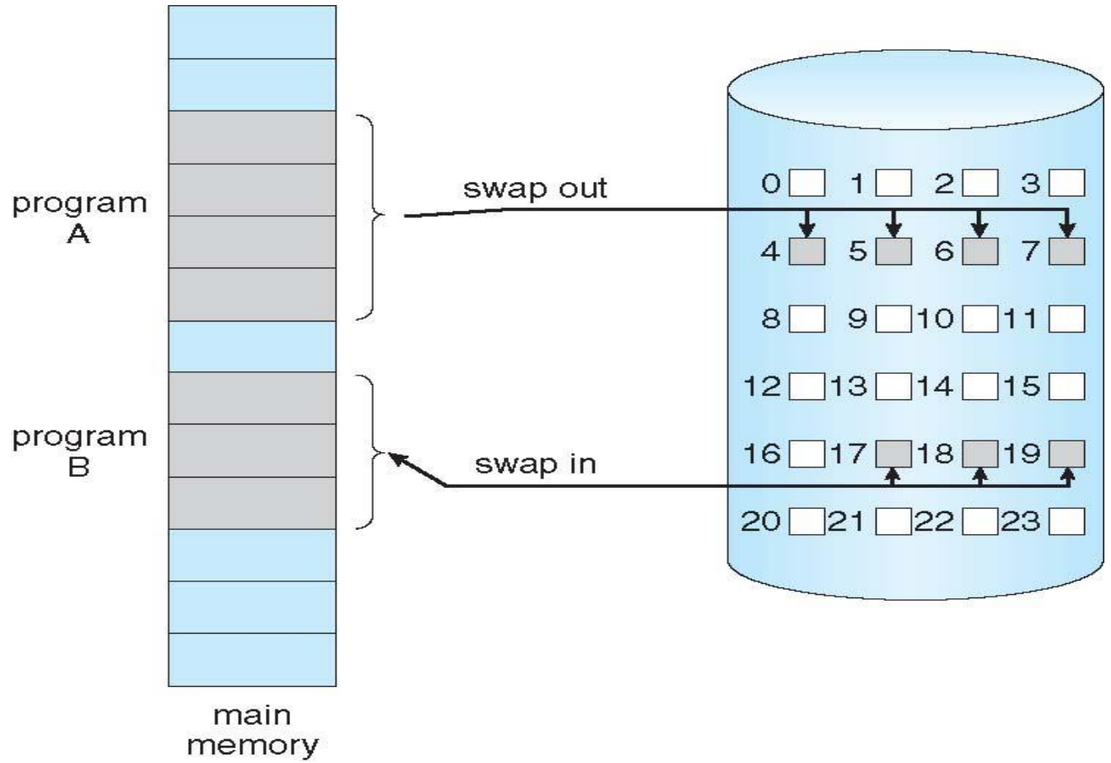
Page Fault:

- If CPU demands a page which is not in main memory, it is called a Page fault.
- In case of page fault **OS brings faulted page** from the secondary memory to main memory (M.M.) by replacing a page (if needed) and updates the page table.
- CPU generates logical address (to access instruction /data) through this logical address it gets to know which page it exists .
- Further it checks from page table , if the page exists or not in M.M. , if not then an interrupt is generated which is serviced by OS. By bringing it from secondary memory , by rendering **page fault service**.



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 (diagram on right)
- 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**

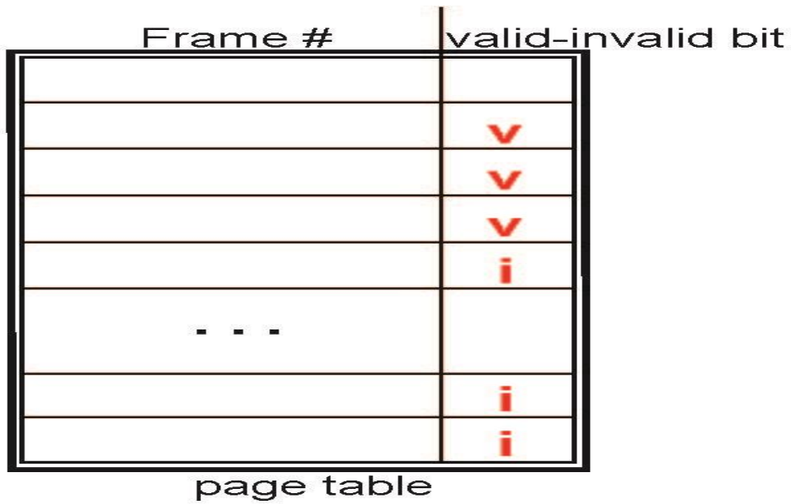


Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, 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

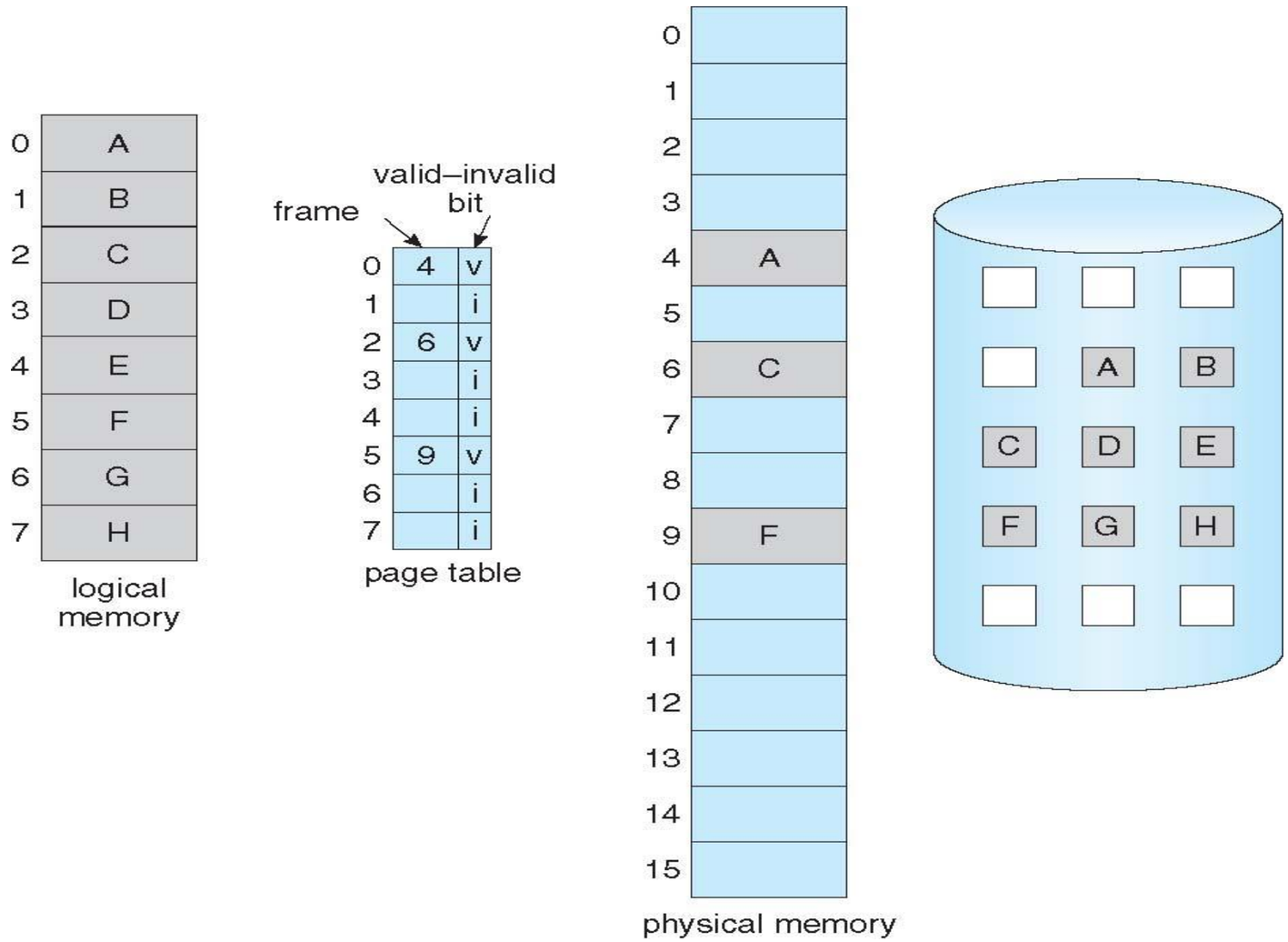
Valid-Invalid Bit

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



- 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



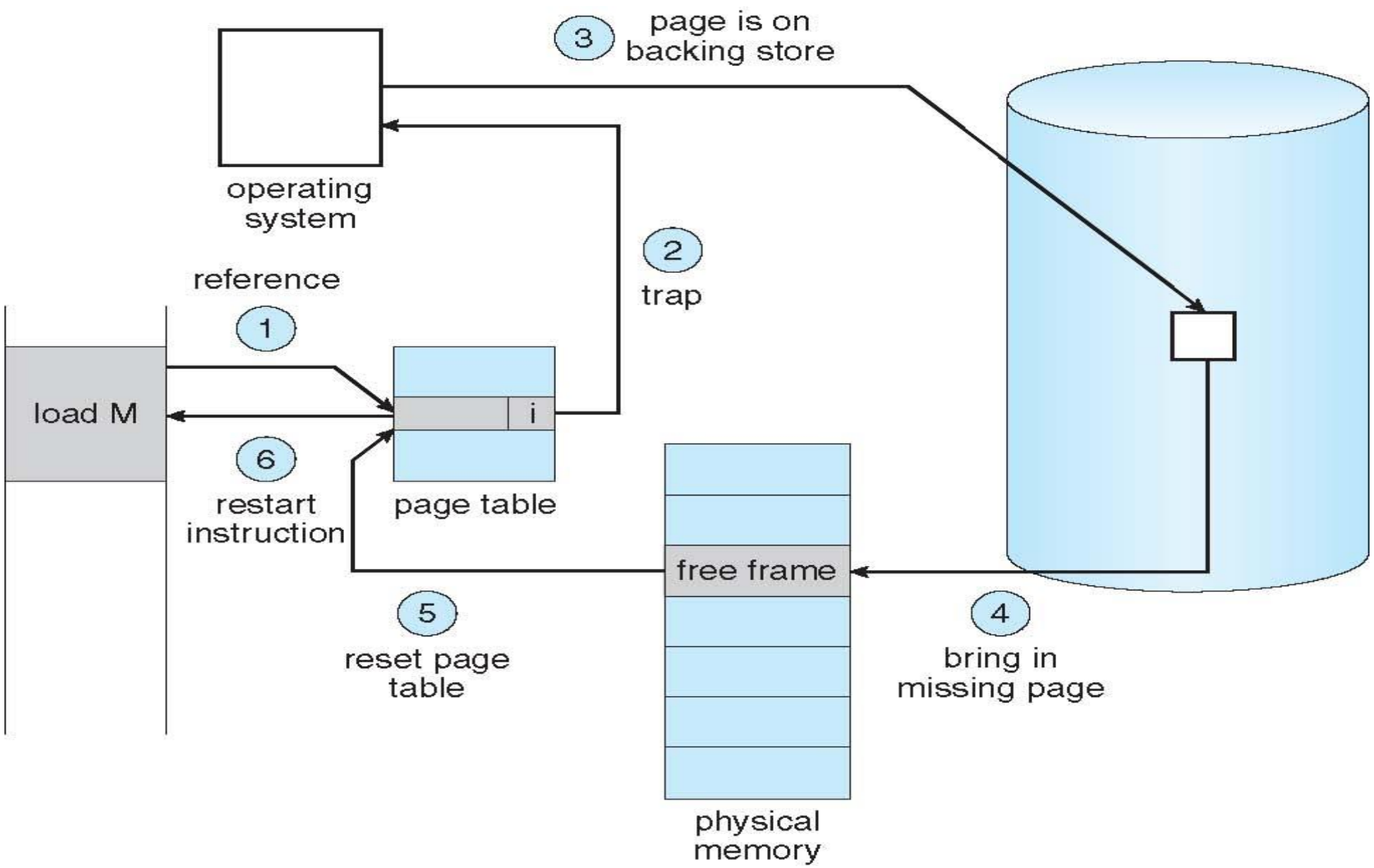
Page Fault

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

  page fault

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ 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

Performance of Demand Paging

- Stages in Demand Paging (worse 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:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. Begin the transfer of the page to a free frame
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.)

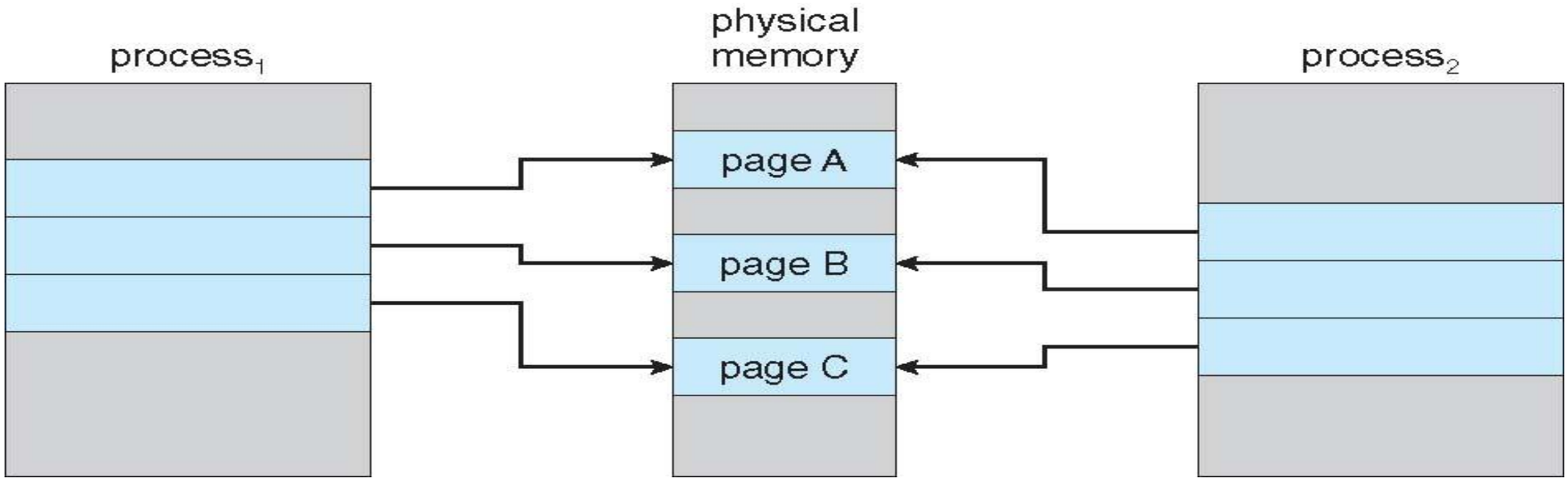
- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \leq p \leq 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference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

$$EAT = (1 - p) \times \text{memory access} + p\ (\text{page fault overhead} + \text{swap page out} + \text{swap page in})$$

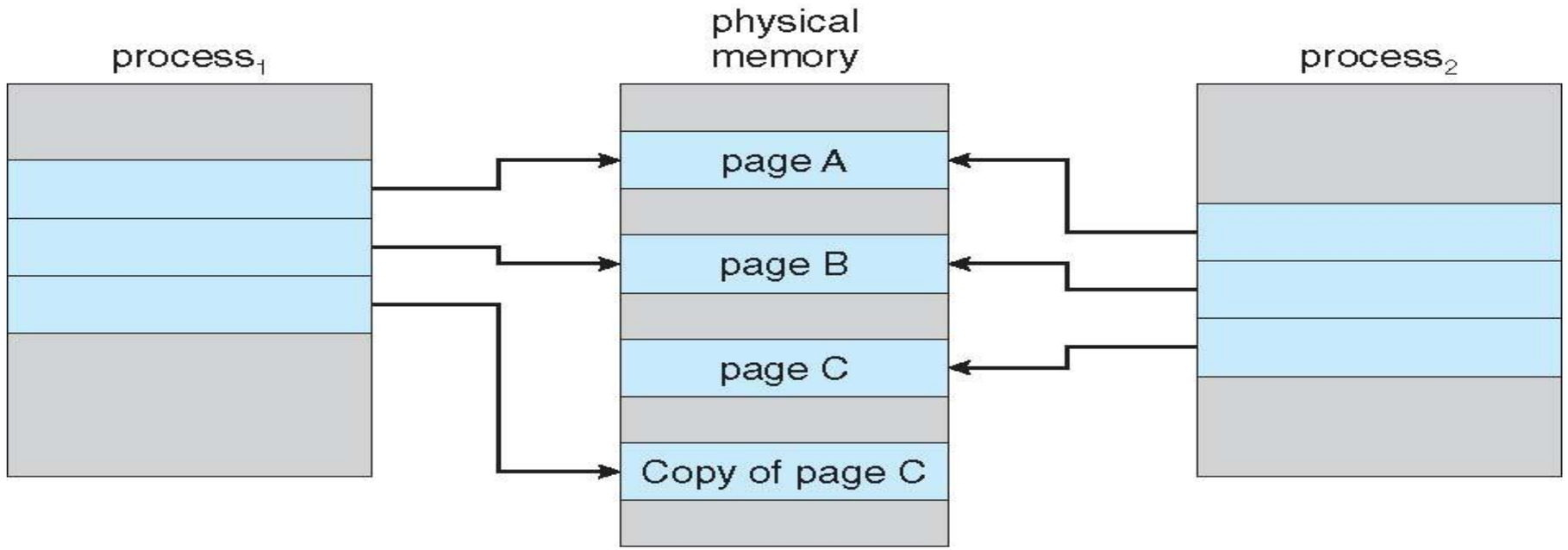
Copy-on-Write

- **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** 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?
- 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

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

Thank You

- **Page Replacement**
- **Allocation of Frame**
- **Thrashing**

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

Basic Page Replacement

CHITKARA UNIVERSITY

1. Find the location of the desired page on disk

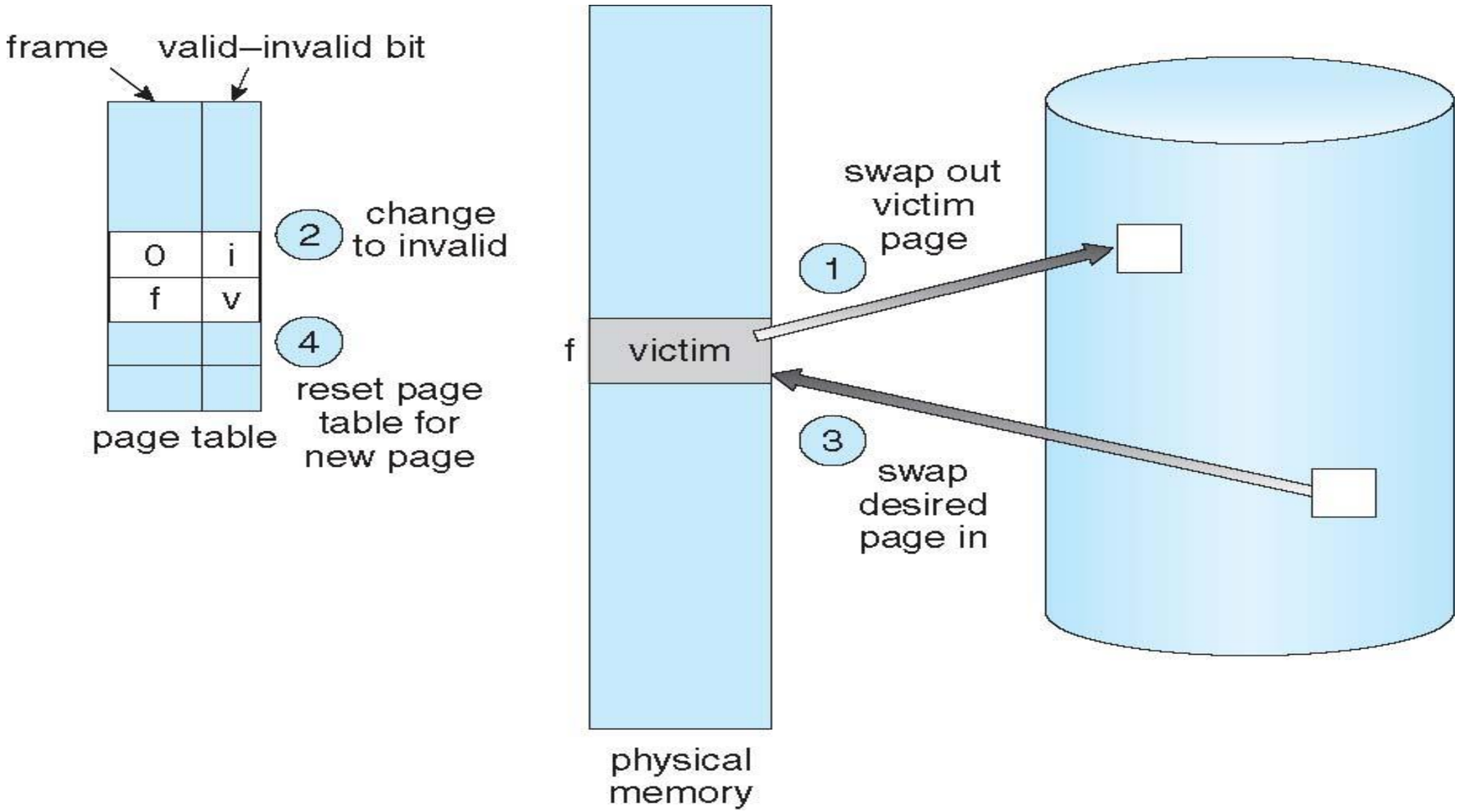
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a page replacement algorithm to select a **victim frame**
     - Write victim frame to disk if dirty

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 – increasing EAT

Page Replacement

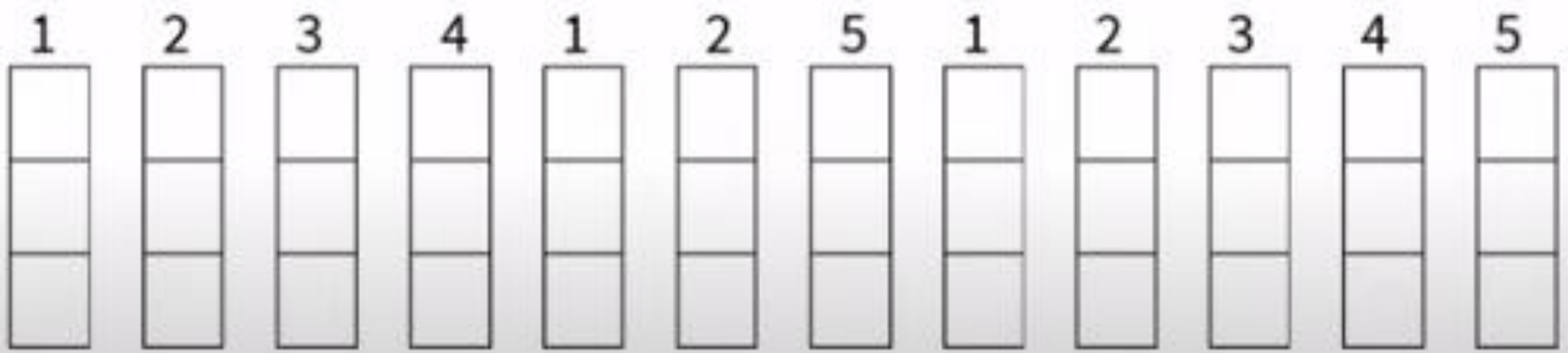


First In First Out (FIFO)

+

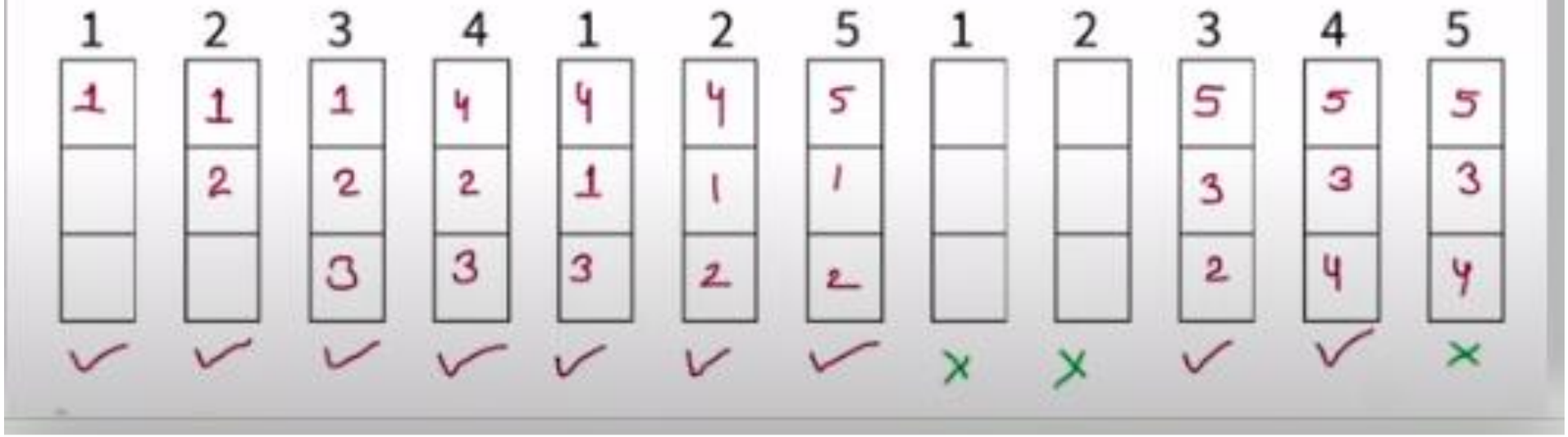
Assume:

- Number of frames = 3 (All empty initially)
- Page reference sequence: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



Assume:

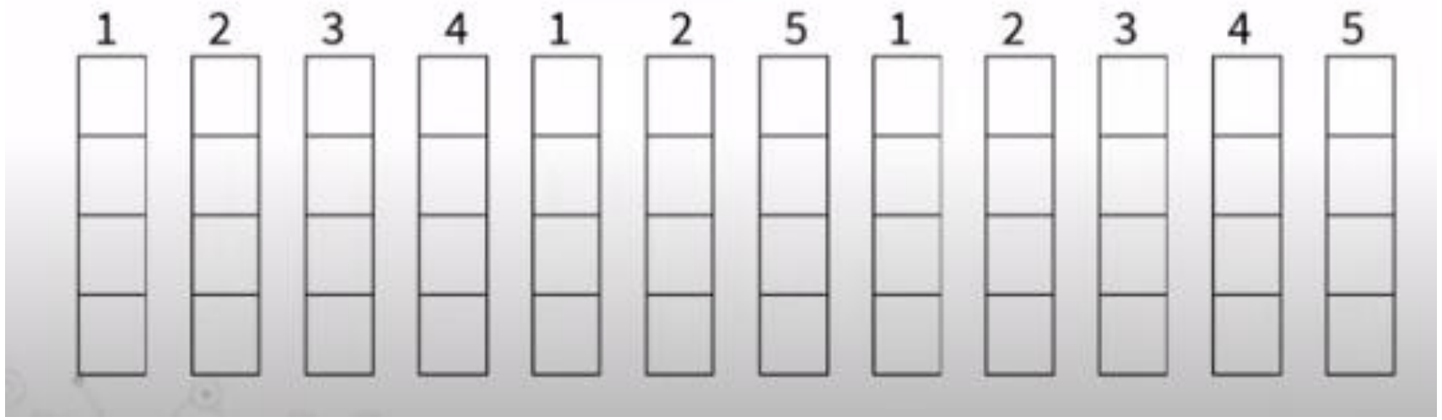
- Number of frames = 3 (All empty initially)
- Page reference sequence: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



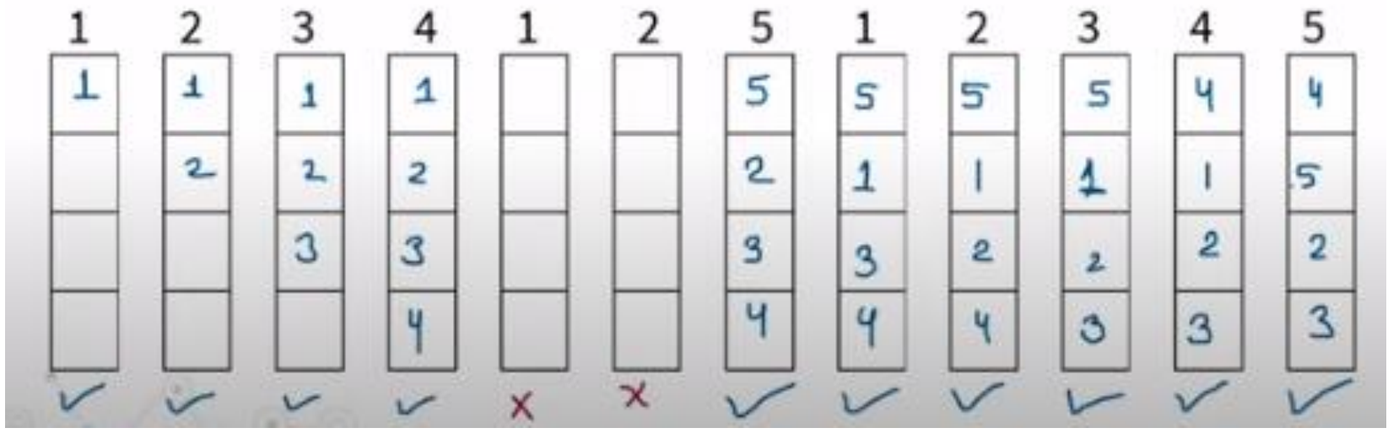
No. of page faults =9
Page fault rate =9/12

Assume:

- Number of frames = 4 (All empty initially)
- Page reference sequence: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



- Number of frames = 4 (All empty initially)
- Page reference sequence: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



2 page hit
Page fault =10

Belady's Anomaly = In few page reference sequence , increasing number of frames , increases number of page faults.
Only FIFO policy suffers from this.

Advantages

1. Simple and easy to implement.
2. Low overhead.

Disadvantages:

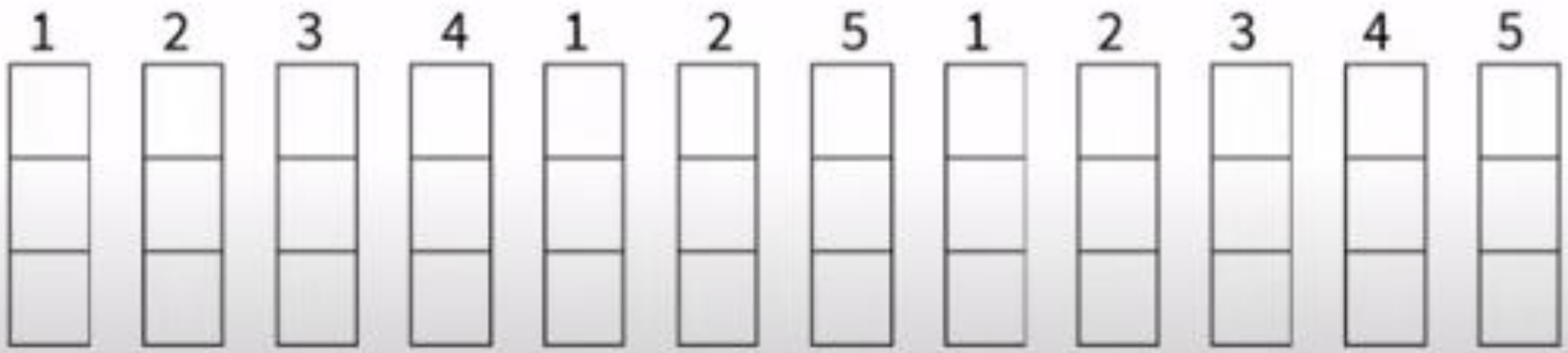
1. Poor performance.
2. Doesn't consider the frequency of use or last used time, , **Simply replaces the** oldest page.
3. Suffers from Belady's Anomaly

Optimal Policy

Replace the page which will rarely(never) going to be referred in future

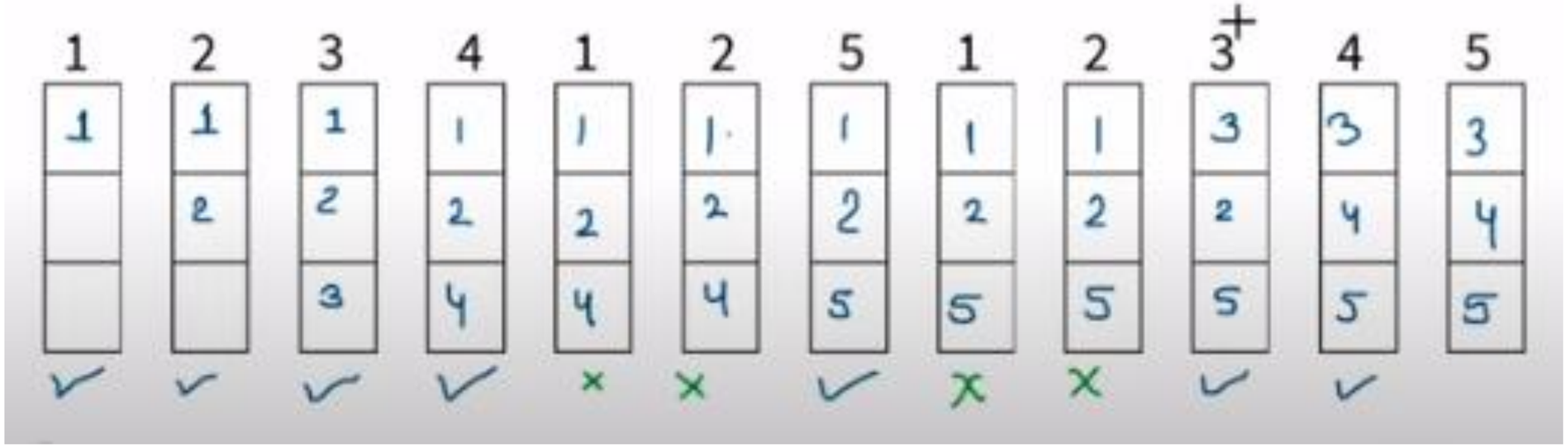
Assume:

- Number of frames = 3 (All empty initially)
- Page reference sequence: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



Optimal Policy

- Number of frames = 3 (All empty initially)
- Page reference sequence: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



No. of page faults = 7
Page fault rate =7/12

It Provides minimum number of page faults.

Optimal Policy

Advantages

1. Easy to Implement
2. Simple data structures are used
3. Highly efficient

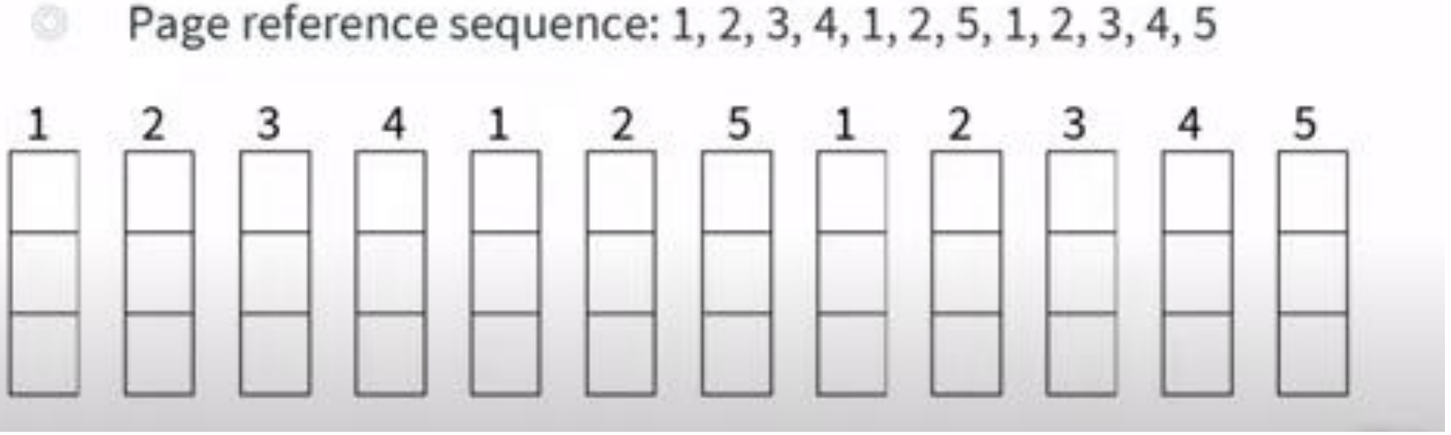
Disadvantages:

1. Requires future knowledge of the program
2. Time-consuming

Least Recently Used (LRU)

Assume:

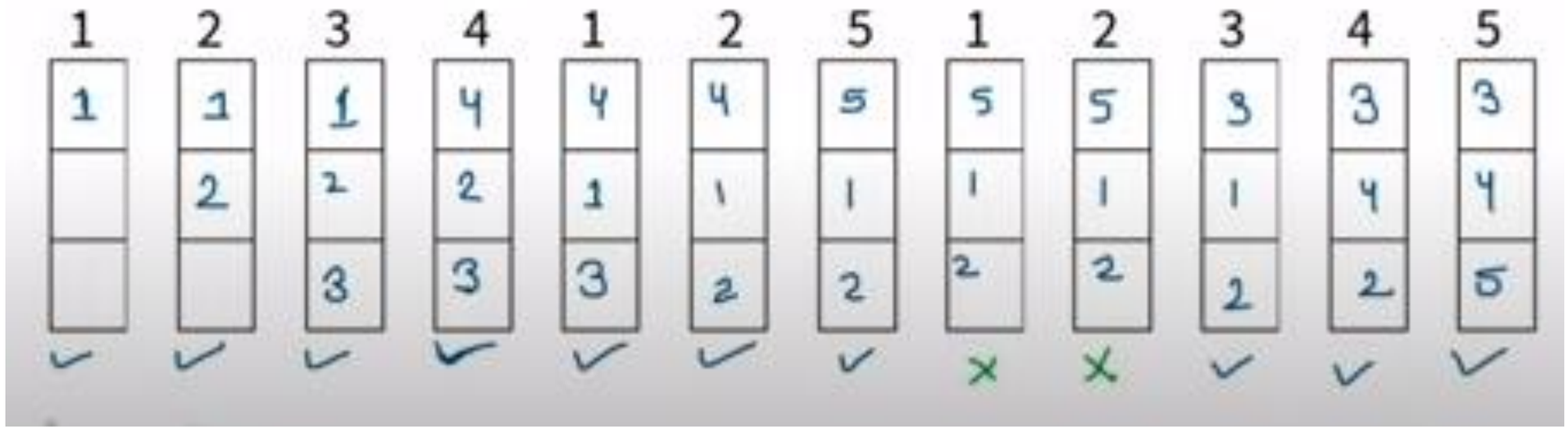
- Number of frames = 3 (All empty initially)
- Page reference sequence: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



Replace that page which CPU has not used since long time.

Least Recently Used (LRU)

- Number of frames = 3 (All empty initially)
- Page reference sequence: 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5



No. of page faults = 10
Page fault rate = 10/12

Least Recently Used (LRU)

Advantages

1. Efficient.
2. Doesn't suffer from Belady's Anomaly

Disadvantages:

1. Complex Implementation
2. Expensive
3. Requires hardware support

Thanks!

Frame Allocation

- How many frames do we allocate per process?
- If it is a single user , single tasking system it is easy , all frames belong to the user process.
- But what happens , if the degree of multiprogramming is more?

Frame Allocation

- What is the minimum number of the frames that a process needs?
- Is page replacement local or global?
- Two major allocation schemes.
  - fixed allocation
  - priority allocation

Fixed allocation:

- Fixed Frame allocation
  - Equal
  - Proportional

Example: Total = 6 frames , 2 processes, P1 = 8KB, P2= 4KB

- Equal : P1 & P2 both will be allocated 3 pages each.
- Proportional
- P1 = 8/(8+4)= 2/3
- P2 = 4/12=1/3
- P1= 2/3*(6)=4 frames
- P2 = 1/3*(6)=2 frames

Priority based page 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.

Local Allocation

1. Local replacement requires that the page being replaced be in a frame belonging to the same process
2. The number of frames belonging to the process will not change
3. This allows processes to control their own page fault rate

Global Allocation

1. The process can replace a page from a set that includes all the frames allocated to user processes
2. High-priority processes can increase their allocation at the expense of lower-priority processes
3. Global allocation makes for more efficient use of frames and their better throughput

Thrashing

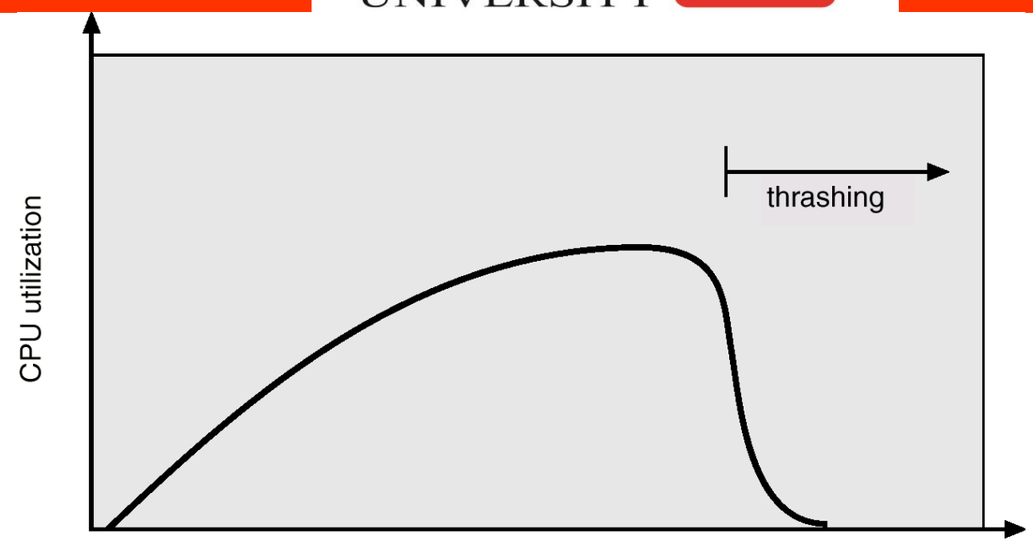
- If a process does not have “enough” pages, the page-fault rate is very high. This leads to:
  - low CPU utilization.
  - operating system thinks that it needs to increase the degree of multiprogramming.
  - another process added to the system.
- Thrashing ≡ a process is busy swapping pages in and out

Thrashing

- High level paging activity is called thrashing
- Degree of multiprogramming : how many processes are in main memory
- CPU spends more time on page fault service as compared to the process execution, which is actual productive task. Hence CPU utilization decreases.

Thrashing Diagram

- Why does paging work?
  Locality model
  - Process migrates from one locality to another.
  - Localities may overlap.
- Why does thrashing occur?
  Σ size of locality > total memory size



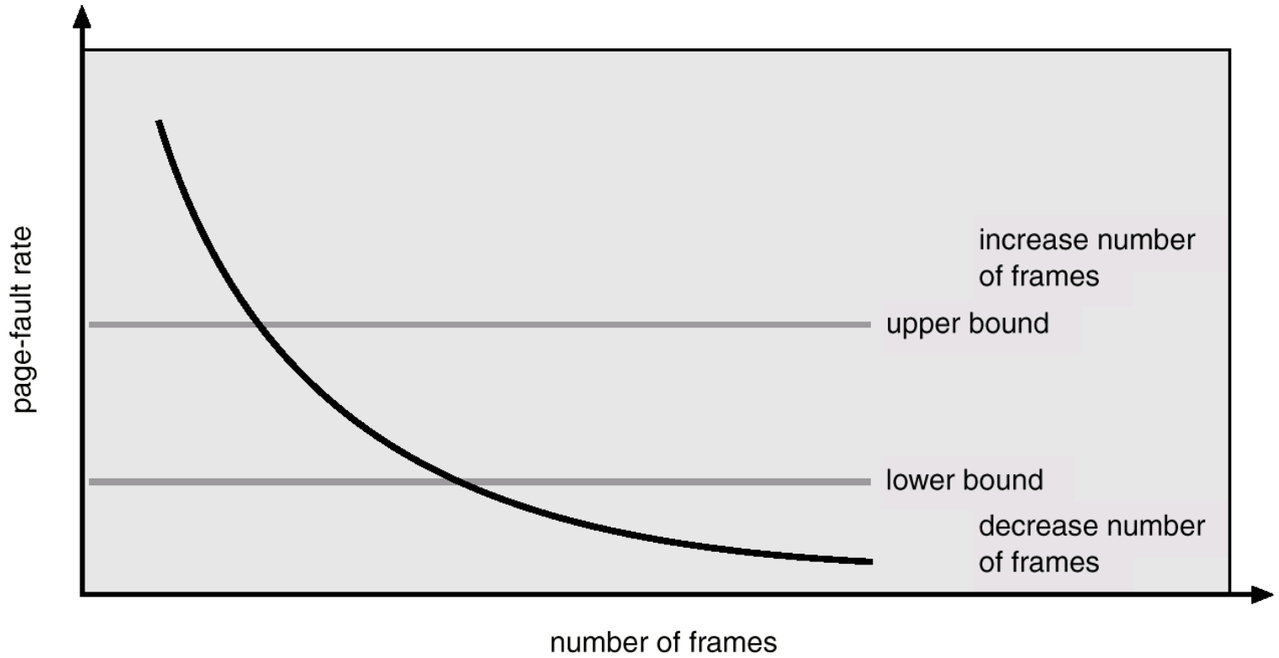
Locality Model

- Locality Model
  - Working Set
  - Page fault frequency

Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  Example: 10,000 instruction
- 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
- Policy if $D >$ m, then suspend one of the processes.

Page-Fault Frequency Scheme



- Establish “acceptable” page-fault rate.
  - If actual rate too low, process loses frame.
  - If actual rate too high, process gains frame.

Thanks !