Virtual Memory

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

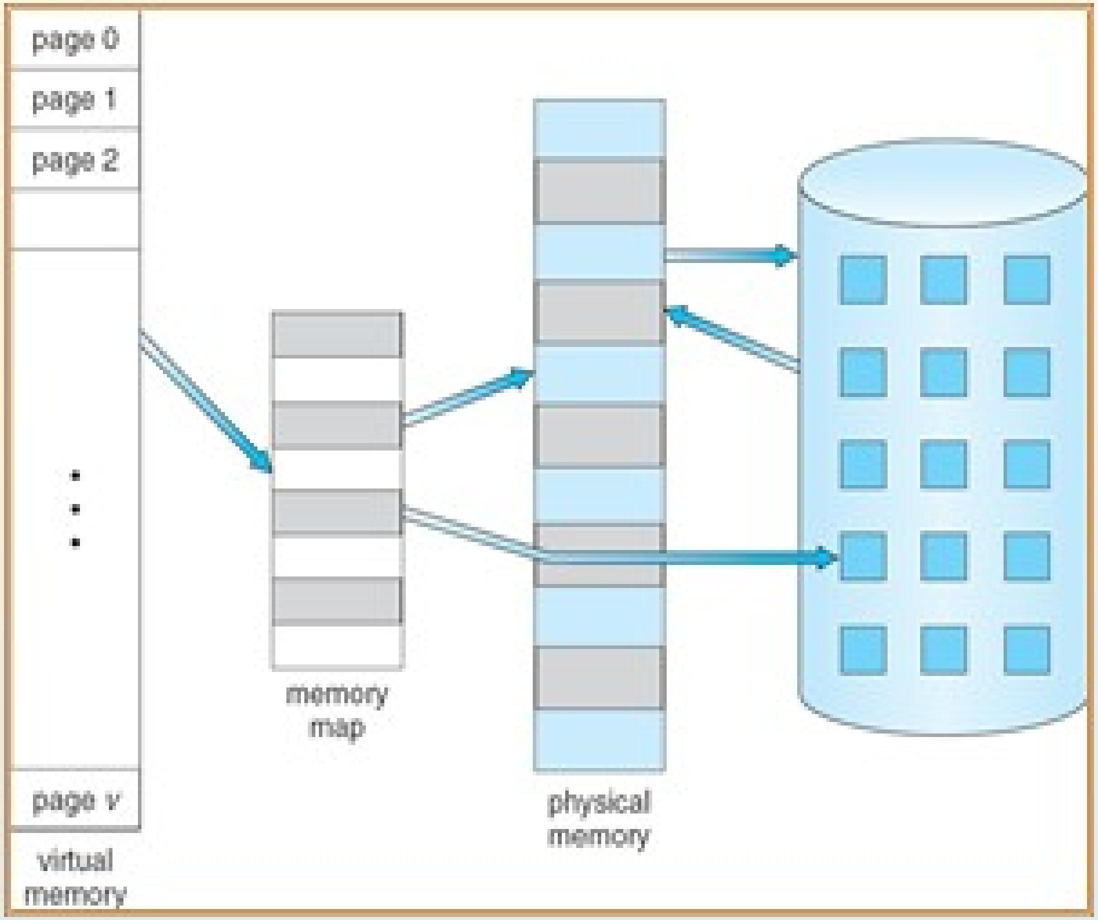
What is Virtual Memory?

- Technique that allows the execution of processes that are not completely in memory.
- Abstracts main memory into an extremely large, uniform array of storage.
- Allows processes to share files easily and to implement shared memory.

Background

- For a program to get executed the entire logical address space should be placed in physical memory
- But it need not be required or also practically possible. Few examples are
  - Error codes seldom occur
  - Size of array is not fully utilized
  - Options and features which are rarely used

Virtual Memory That is Larger Than Physical Memory



Benefits

- Program length is not restricted to real memory size. That is, virtual address size can be larger than physical address size.
- Can run more programs because those space originally allocated for the un-loaded parts can be used by other programs.
- Save load/swap I/O time because we do not have to load/swap a complete program.

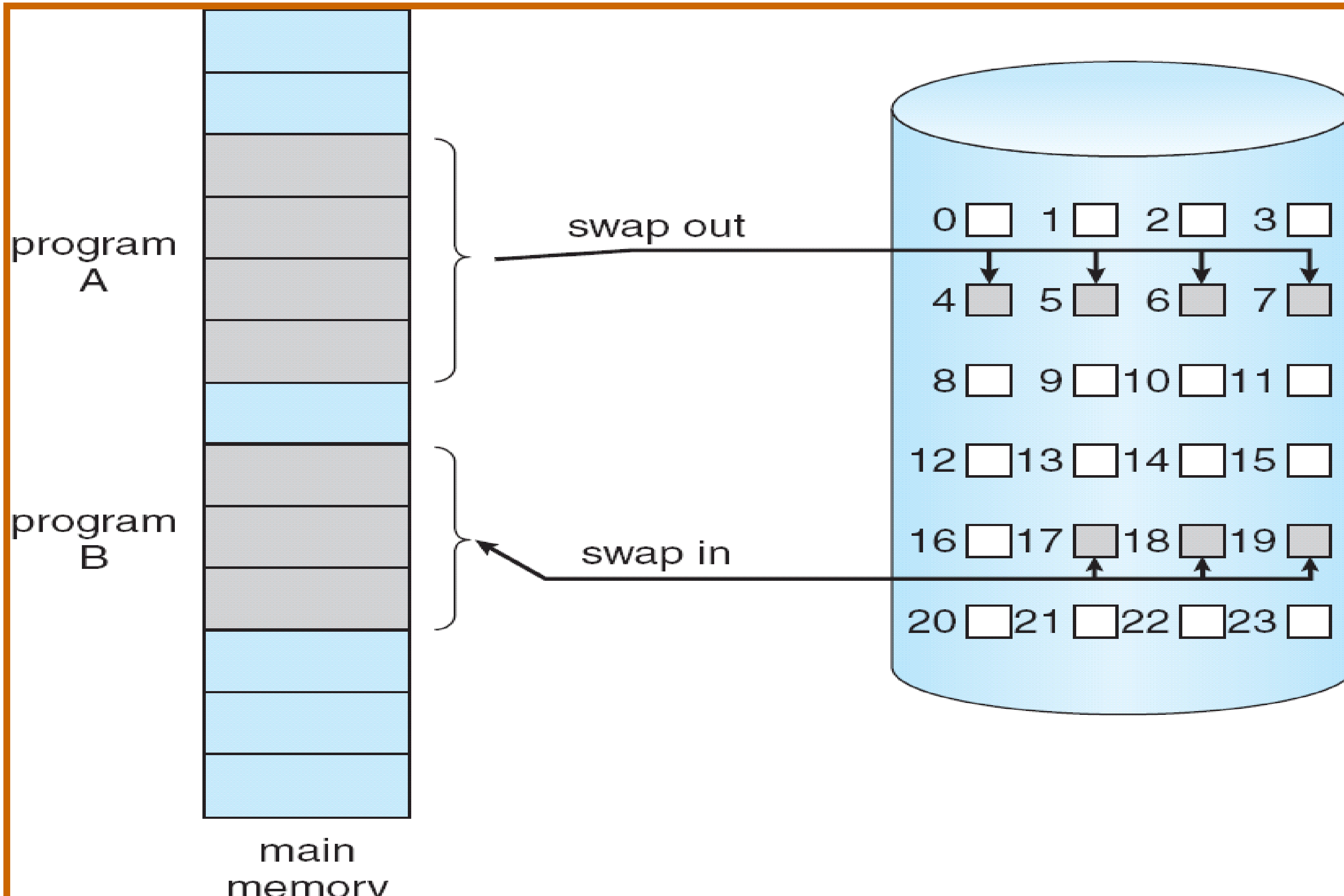
Virtual memory is implemented using DEMAND PAGING

Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - 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 will be needed
  - Swapper that deals with pages is a **pager**

Transfer of a Paged Memory to Contiguous Disk Space

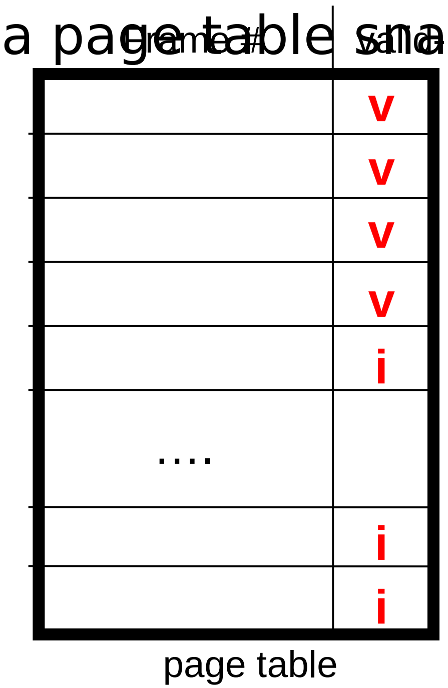


Basic concepts

- When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again
- The pager brings only those pages into memory
- Valid-invalid bit scheme determines which pages are there in the memory and which are there in the disk.

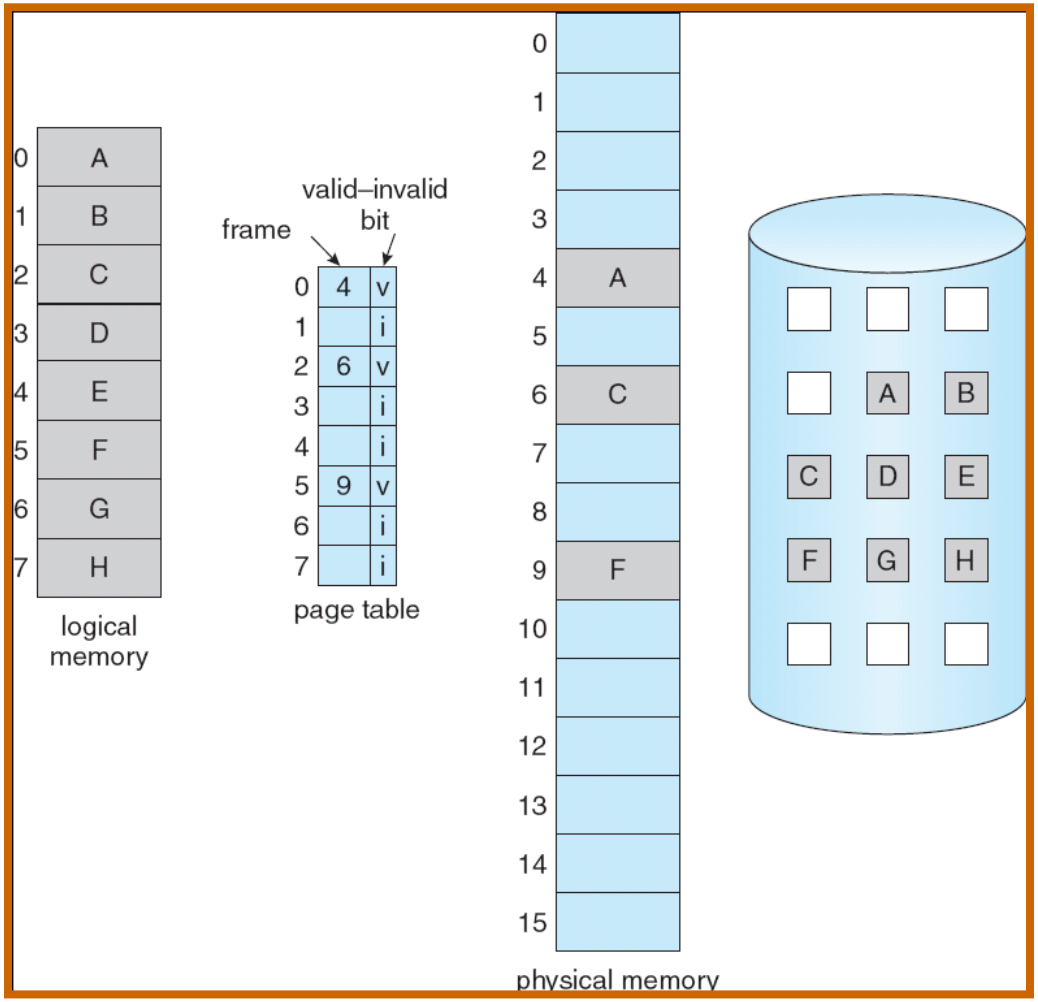
Valid-Invalid Bit

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



- During address translation, if valid–invalid bit in page table entry

Page Table When Some Pages Are Not in Main Memory



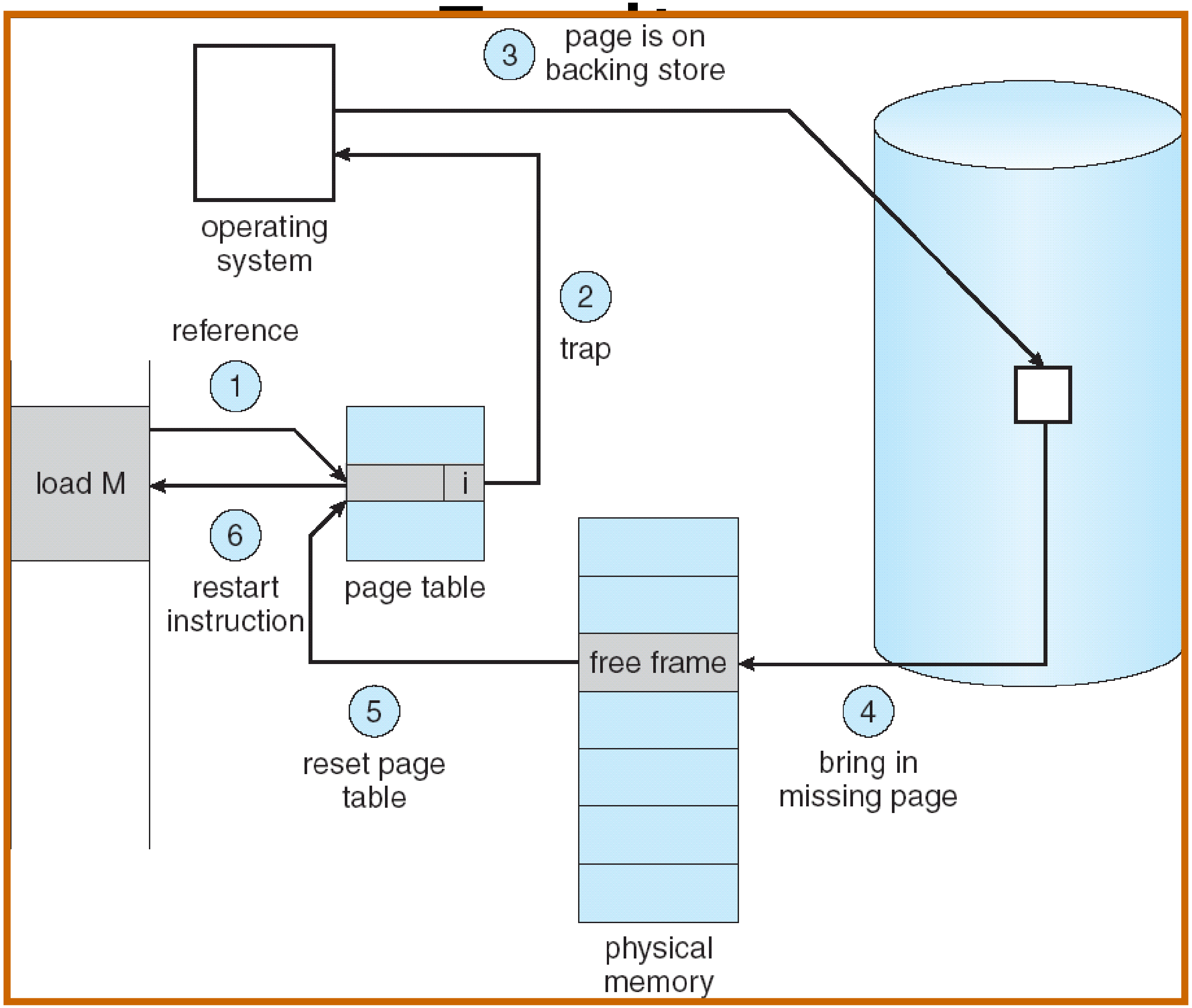
Page Fault

- 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



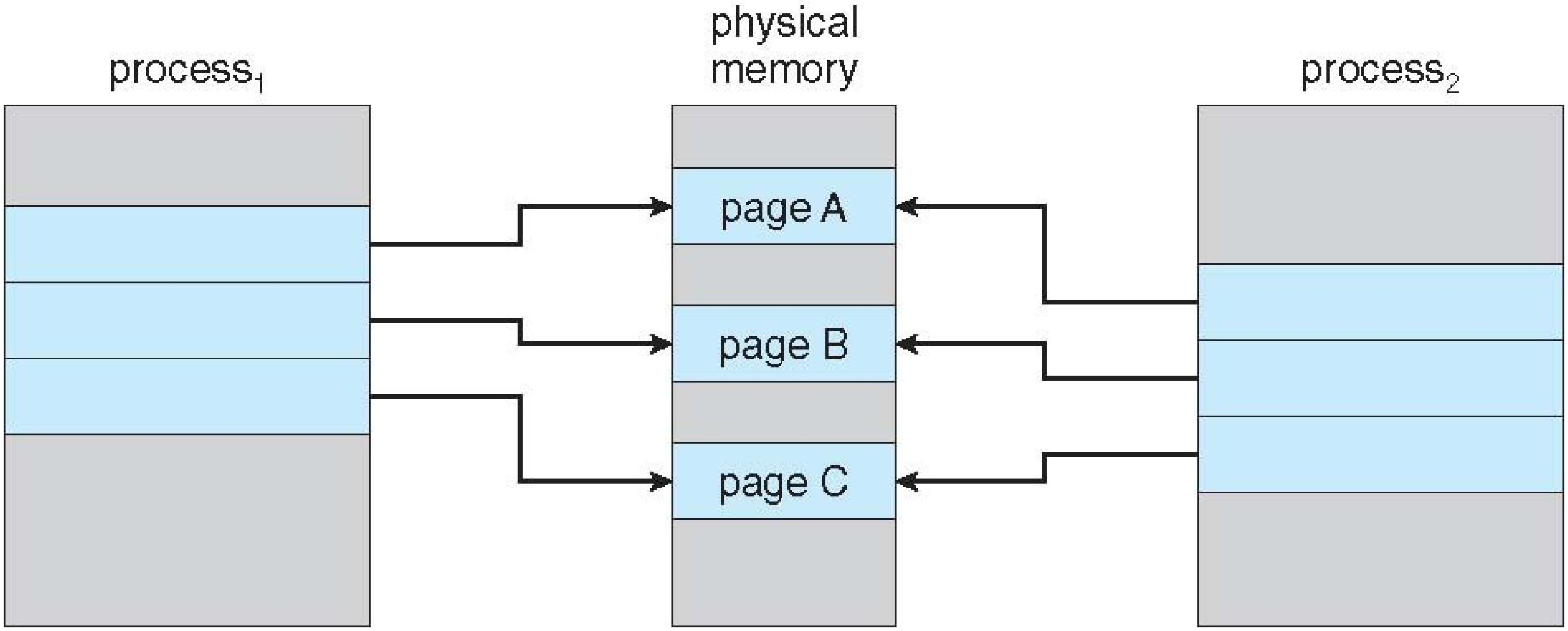
Aspects of Demand Paging

- 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

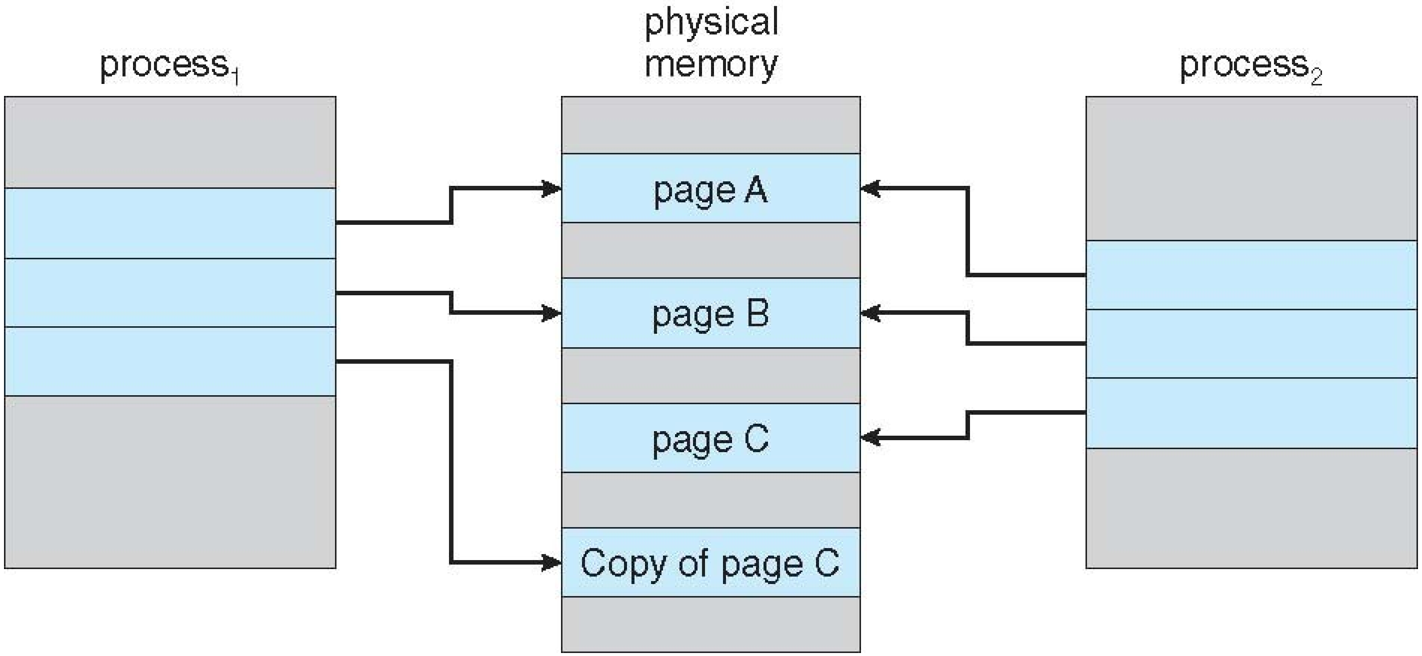
Copy-on-Write

- **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
- When is a page going to be duplicated using copy-on-write?
  - Depends on the location from where a free page is allocated
- OS uses Zero-fill-on-demand technique to allocate these pages.
- UNIX uses vfork() instead of fork() command

Before Process 1 Modifies Page C



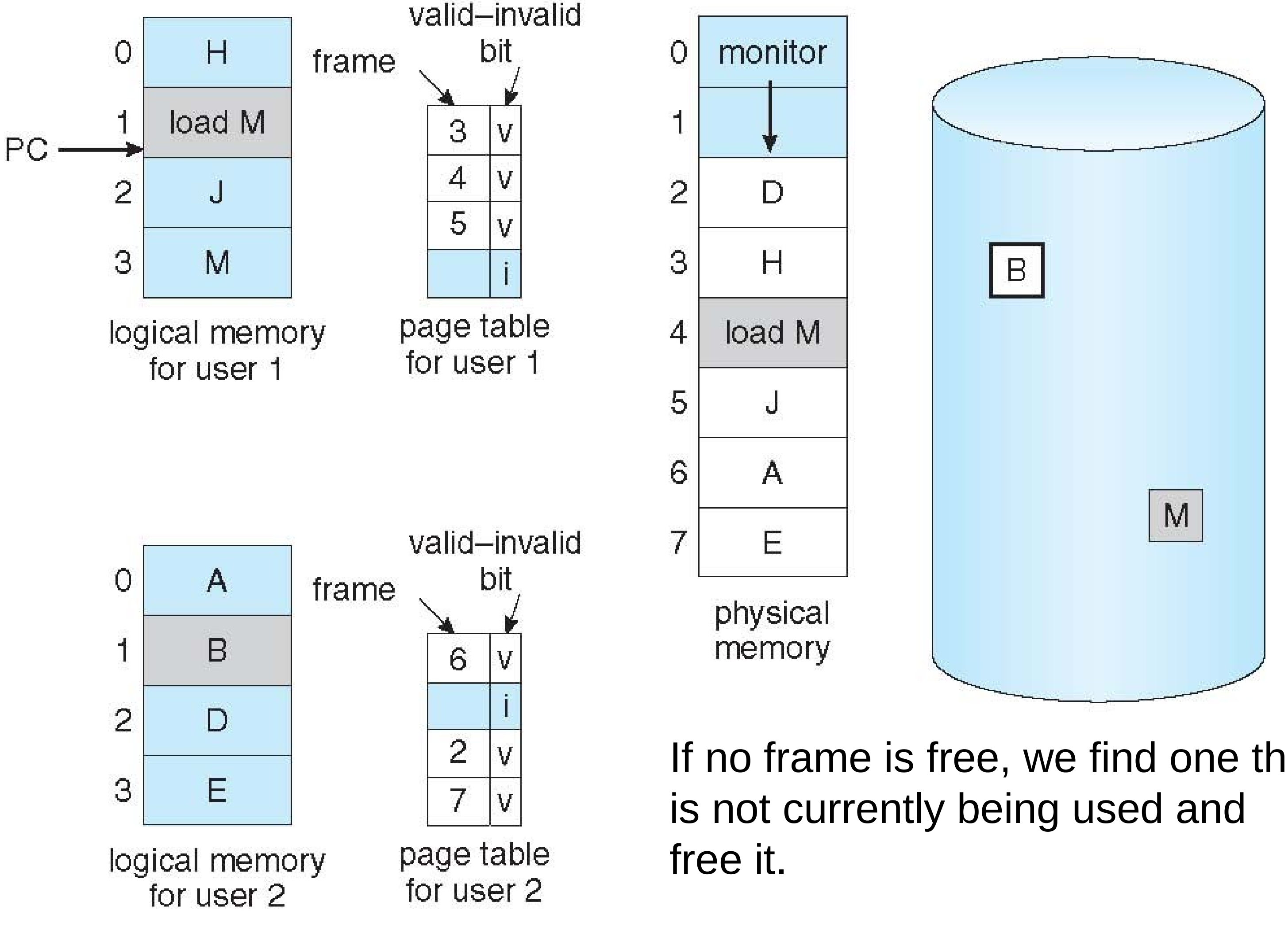
After Process 1 Modifies Page C



Page Replacement

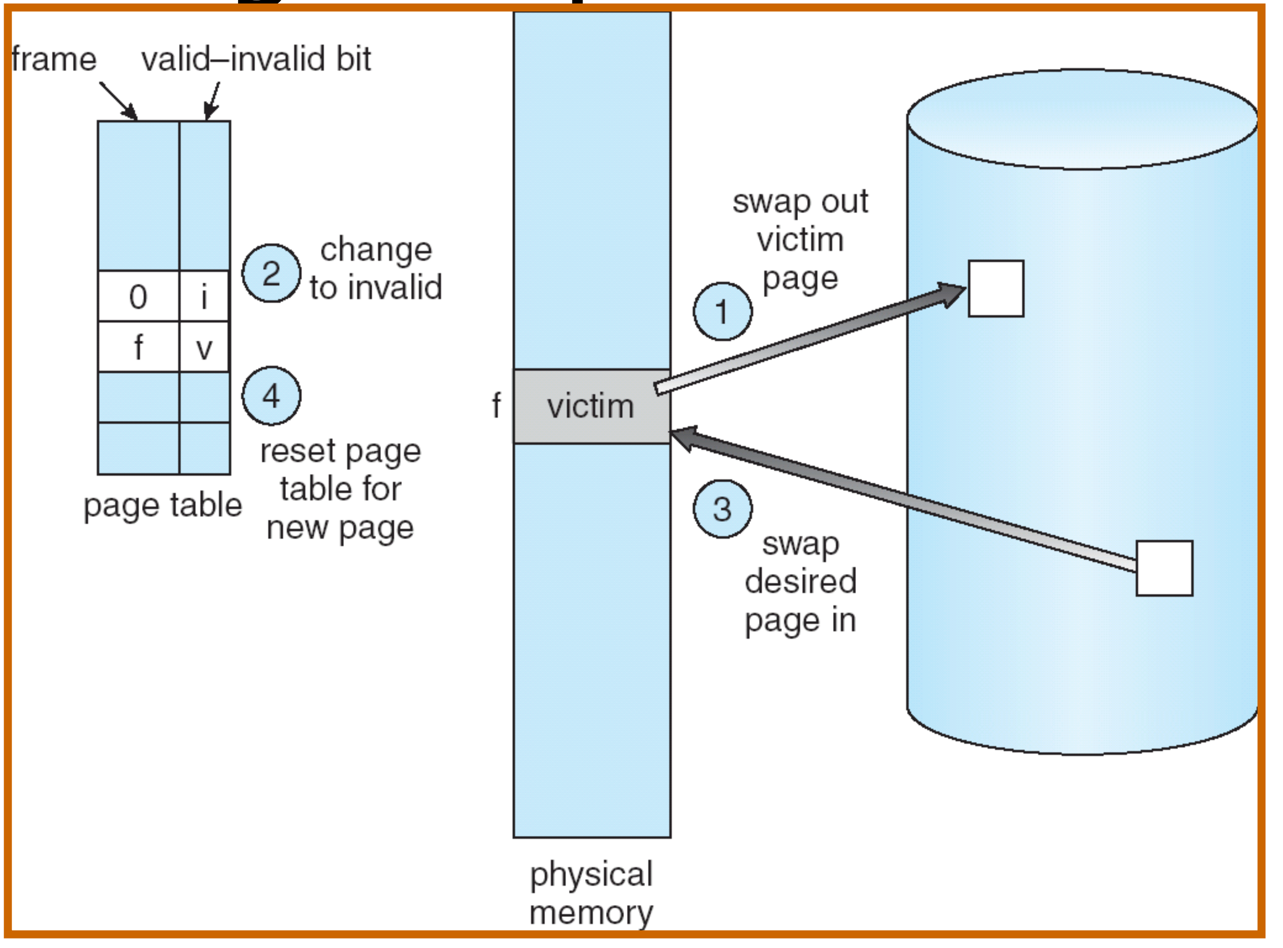
- Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement completes separation between logical memory and physical memory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

Need For Page Replacement



If no frame is free, we find one that is not currently being used and free it.

Page Replacement



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

Page replacement algorithms

- FIFO
- Optimal
- LRU

First In First Out(FIFO)

- Associates with each page the time when that page was brought into memory
- When a page must be replaced, the oldest page is replaced
- FIFO queue is maintained to hold all pages in memory
- The one at the head of Q is replaced and the page brought into memory is inserted at the tail of Q

Page Replacement Algorithms

Page request sequence

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

FIFO (Assume frame size 3 and No frames preloaded)

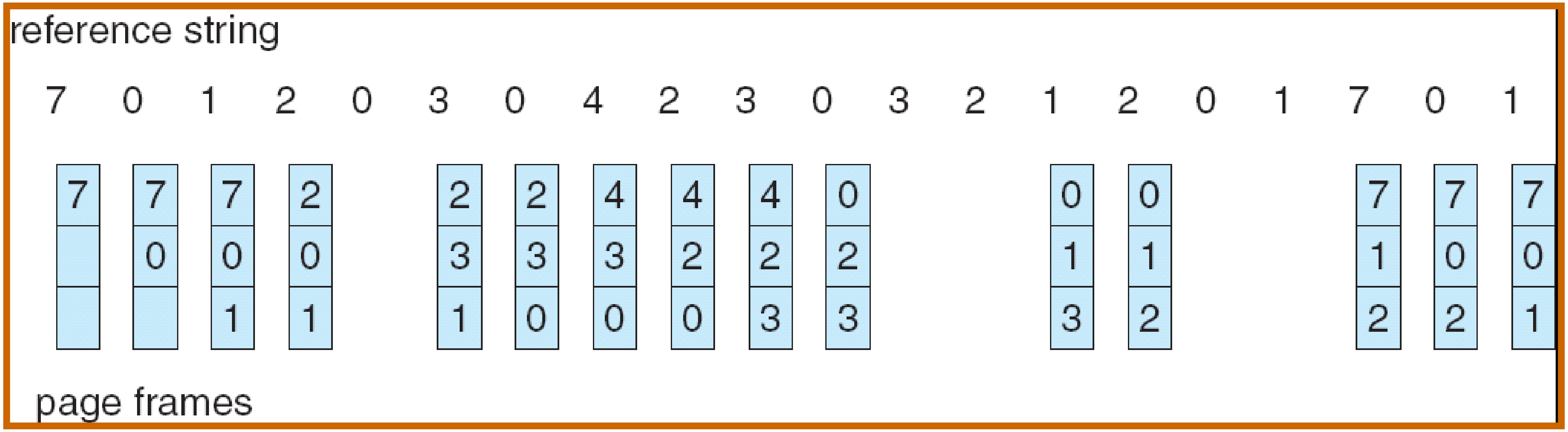
Pages required: 3 1 3 4 2 4 1 2 3 1 2 4 2 3 1 3

Frames										
	3	3	3	1	4	2	3	1	4	2
		1	1	4	2	3	1	4	2	3
			4	2	3	1	4	2	3	1

Number of page faults=10

Number of page replacement = 7

FIFO Page Replacement



Page faults:15
Page replacements:12

Adv and Disadv of FIFO

Adv

Easy to understand and program

Disadv

- Performance not always good
- The older pages may be initialization files which would be required throughout
- Increases the page fault rate and slows process execution.

What is belady's anomaly

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

Compute using 4 frames

Compare the page faults by using frame size 3

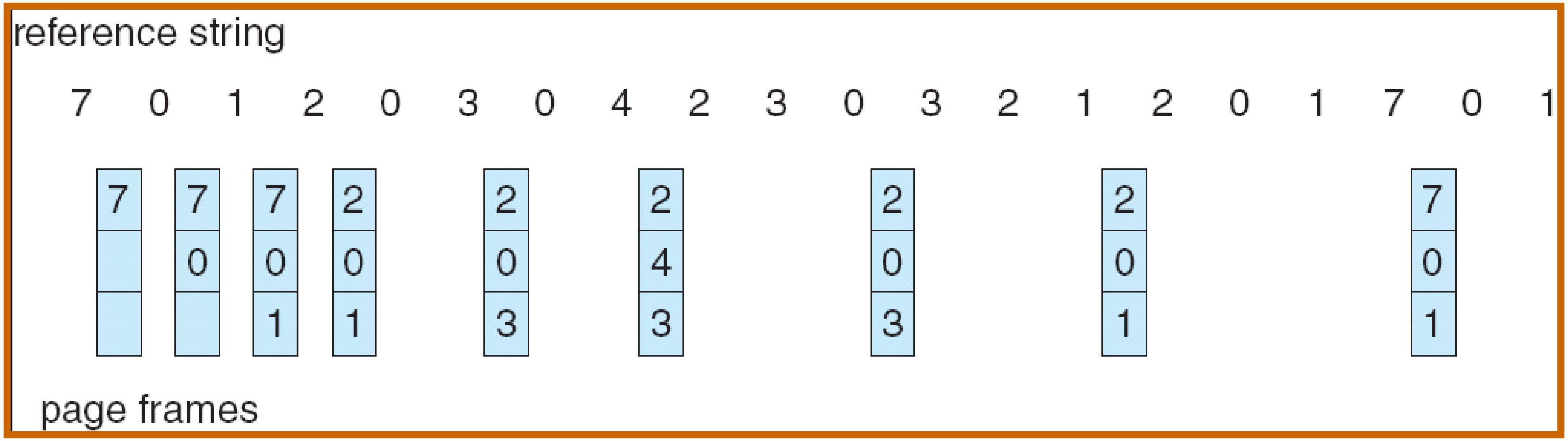
Difference is because of belady's anomaly

Optimal Algorithm

- Result of discovery of Belady’s anomaly was optimal page replacement algorithm
- Has the lowest page-fault rate of all algorithms
- Algorithm does not exist. Why?

Difficult to implement becos it requires future knowledge of reference string

Optimal Page Replacement



Number of page faults:- 9
Number of replacements:-6

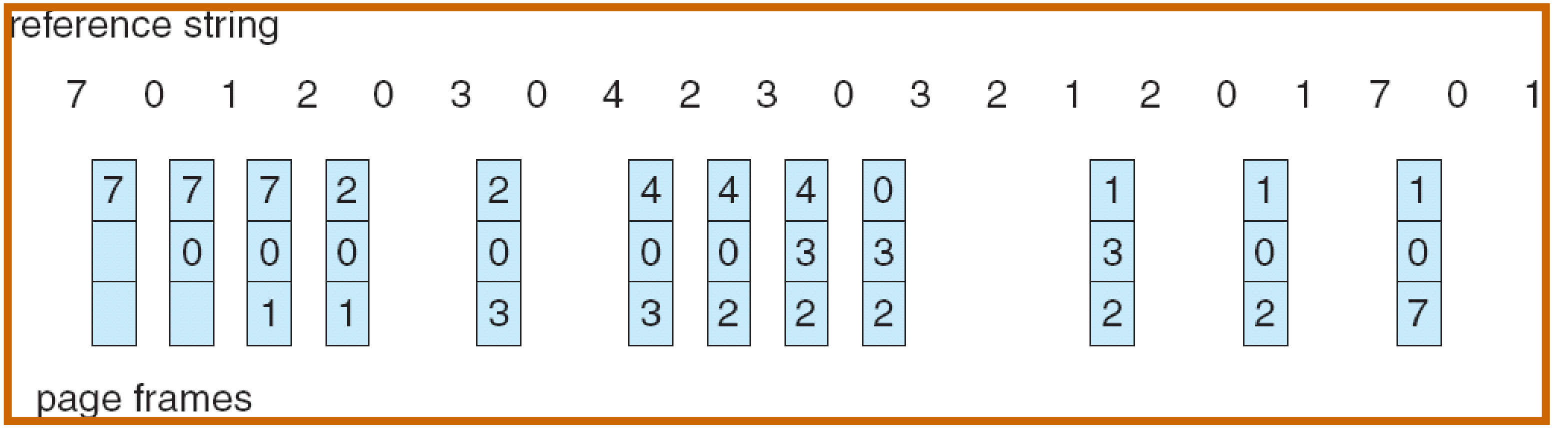
Adv and Disadv of Optimal Page replacement algorithm

- Gives the best result.
- Reduces page fault
- But difficult ot implement because it requires future knowledge of the reference string.
- Mainly used for comparison studies.

LRU page replacement algorithm

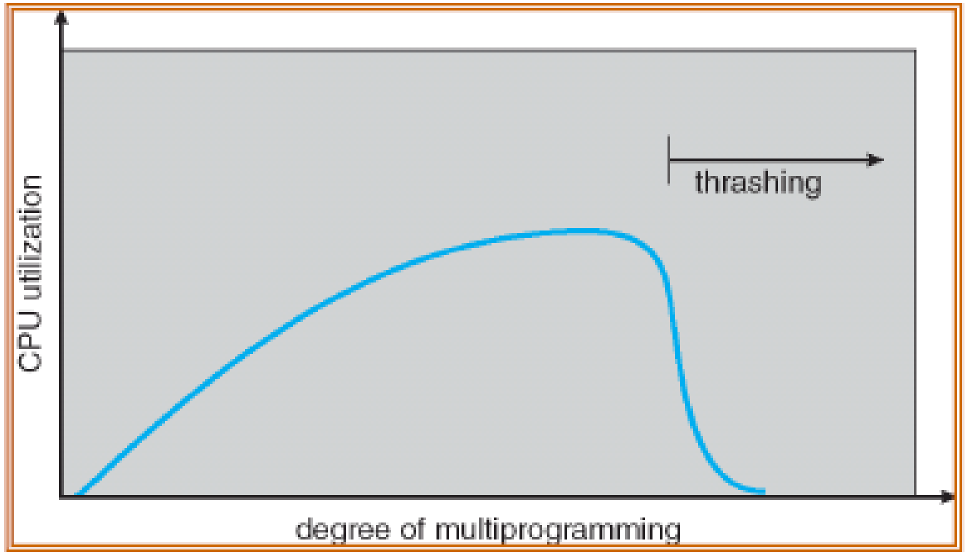
- Use the recent past as an approximation of near future then we replace the page that has not been used for the longest period of time. (Least Recently Used)

LRU Page Replacement



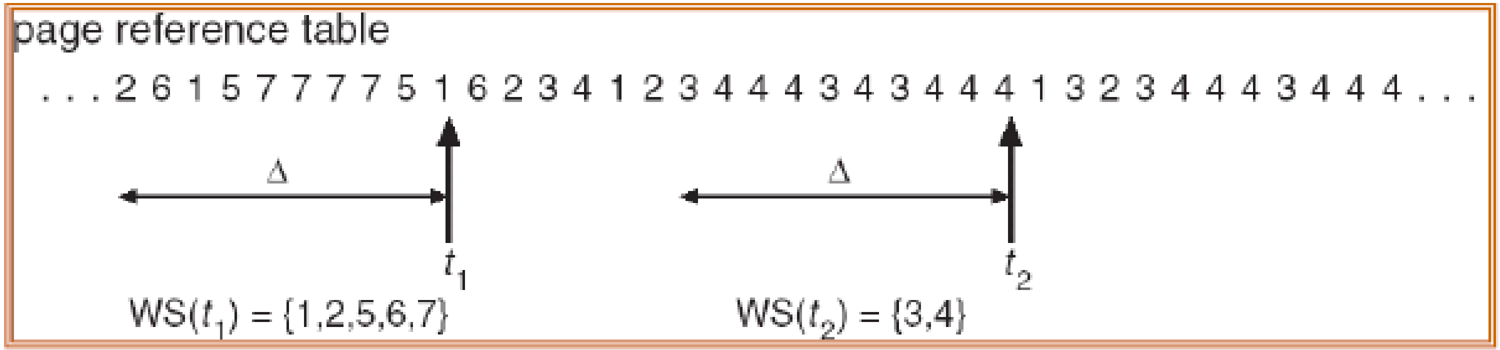
Number of page faults:- 12
Number of page replacements:- 9

Thrashing



- If a process does not have "enough" pages, the page-fault rate is very high. This leads to:
  - low CPU utilization
  - operating system spends most of its time swapping to disk
- Thrashing $\equiv$ a process is busy swapping pages in and out
- Questions:
  - How do we detect Thrashing?
  - What is best response to Thrashing?

Working-Set Model



- $\Delta \equiv$ working-set window $\equiv$ fixed number of page references
  - Example: 10,000 instructions
- WS_i (working set of Process P_i) = total set 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
- $D = \Sigma |WS_i| \equiv$ total demand frames
- if $D > m \Rightarrow$ Thrashing
  - Policy: if D > m, then suspend/swap out processes
  - This can improve overall system behavior by a lot!