



MEMORY MANAGEMENT

Memory management strategies

- Background
- Swapping
- Contiguous Memory Allocation
- Segmentation
- Paging
- Structure of Page Table

Virtual Memory Management

- Background
- Demand paging
- Copy on write
- Page replacement algorithms
- Allocation of frames
- Thrashing.

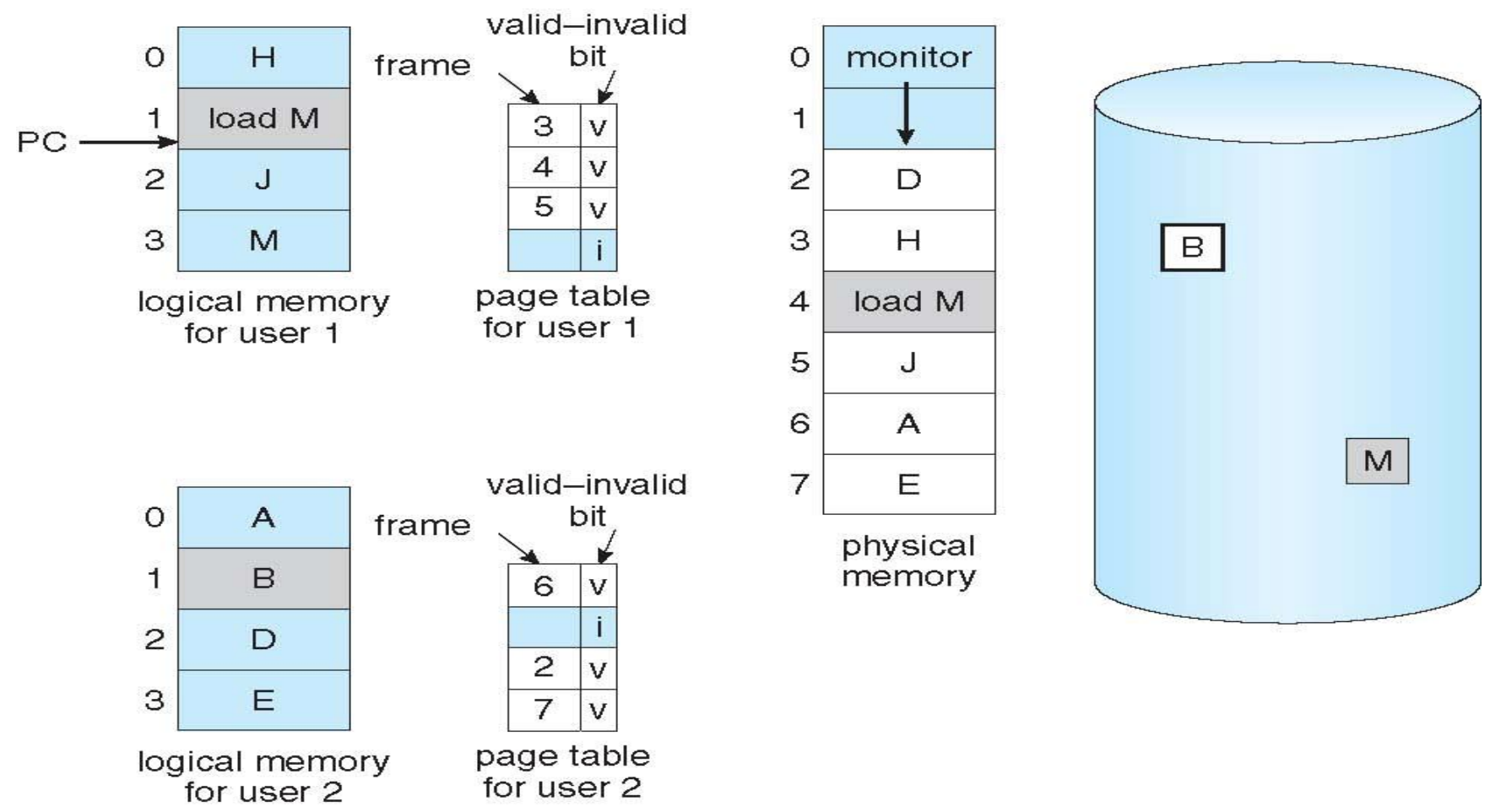


Page Replacement

- 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



Need For Page Replacement





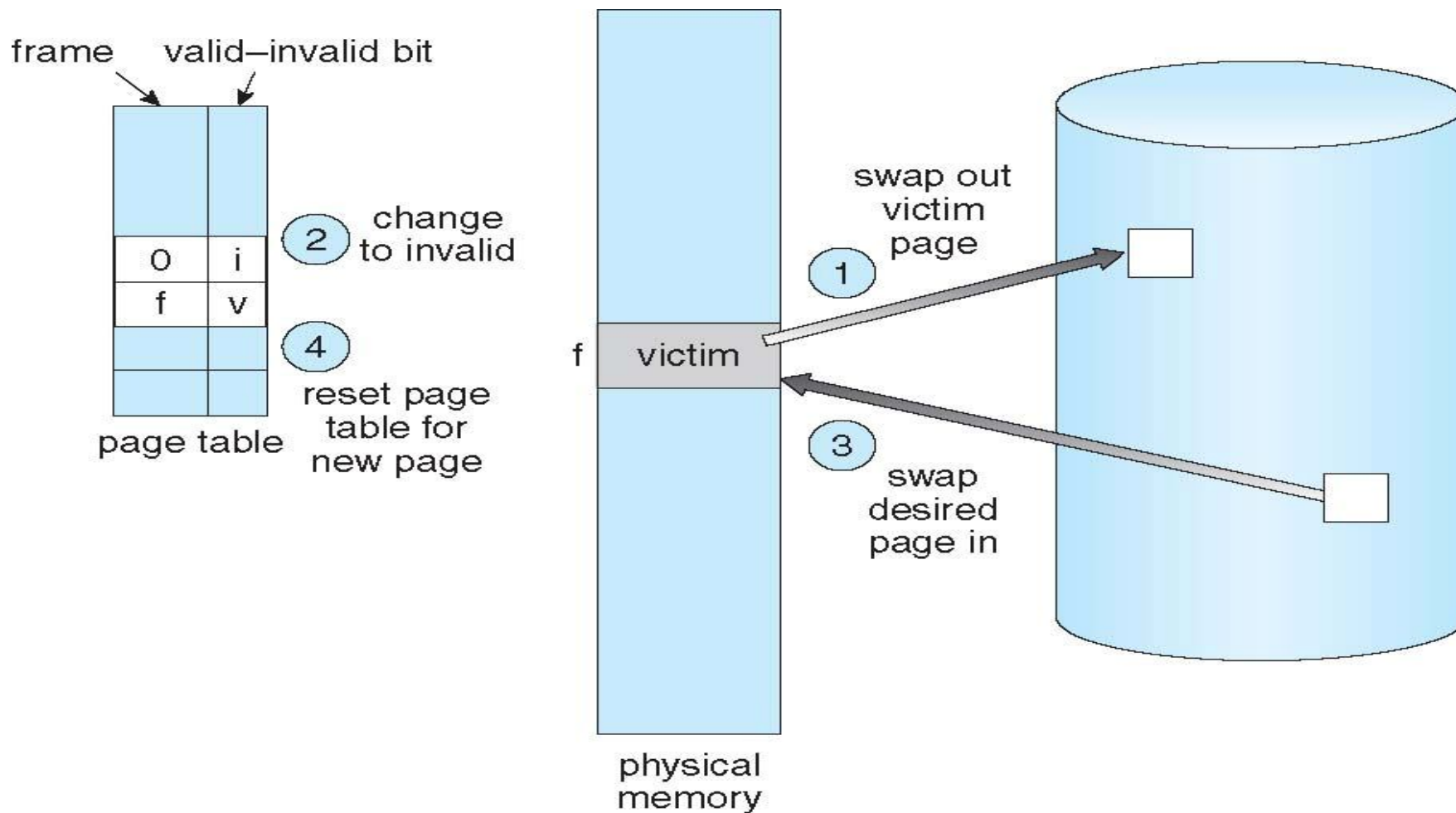
Basic Page Replacement

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 – increasing EAT



Page Replacement





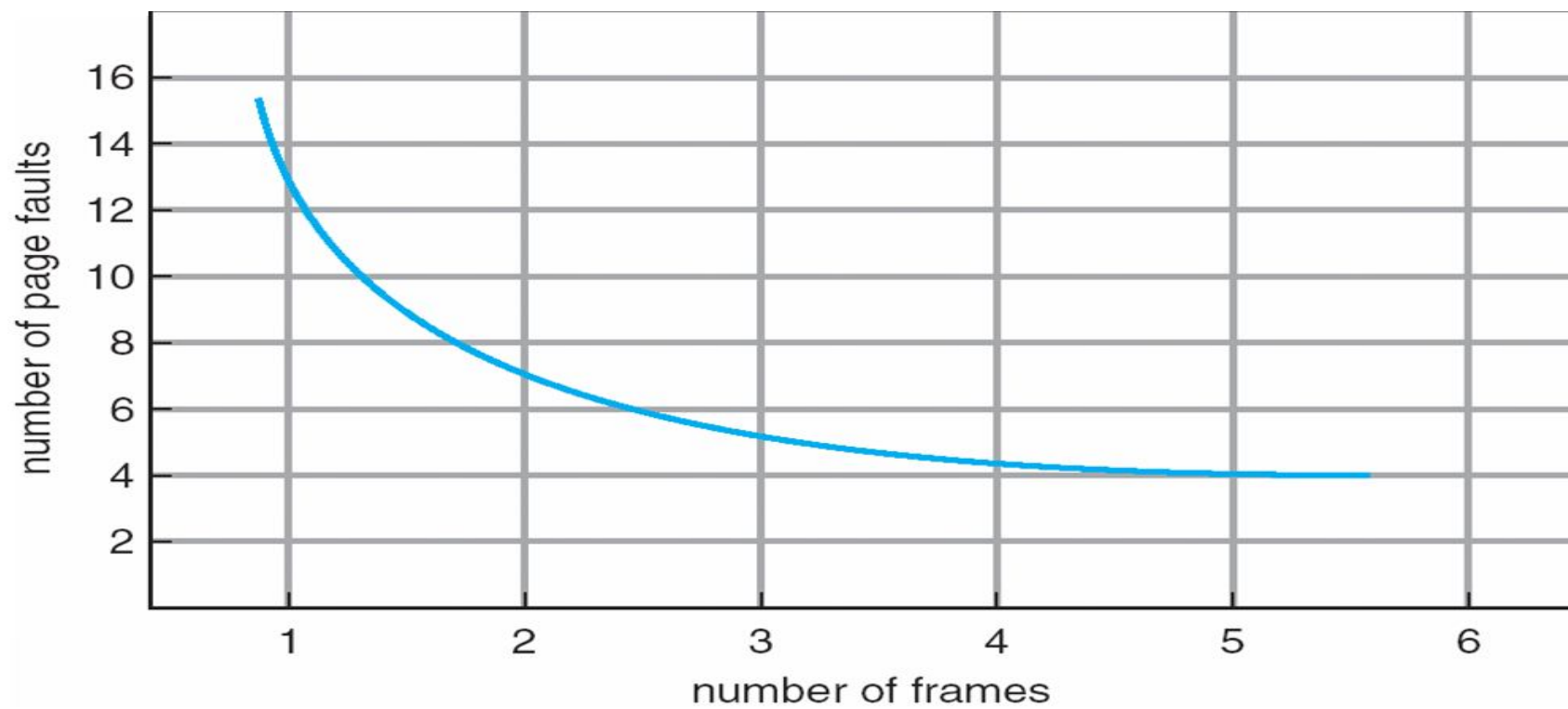
Page and Frame Replacement Algorithms

- **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



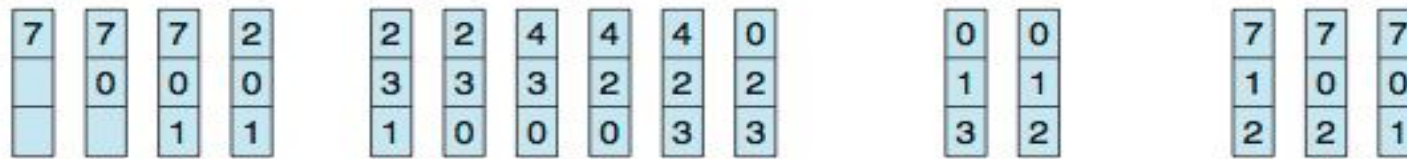


First-In-First-Out (FIFO) Algorithm

- 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

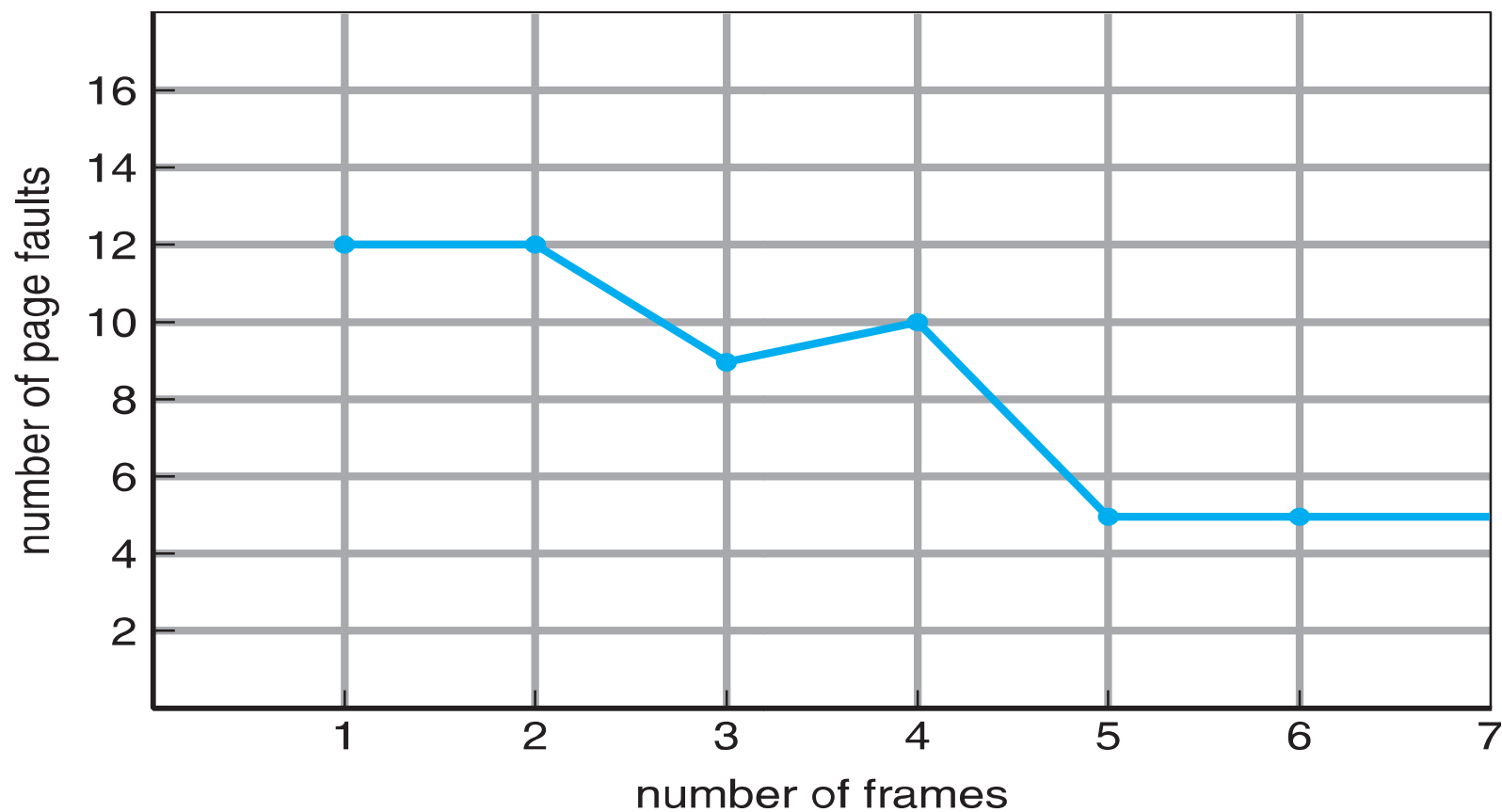


page frames

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



page frames

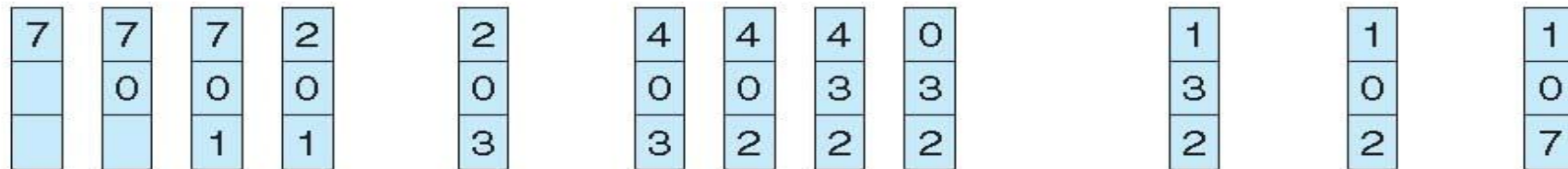


Least Recently Used (LRU) Algorithm

- 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



page frames

- **12 faults** – better than FIFO but worse than OPT
- Generally good algorithm and frequently used



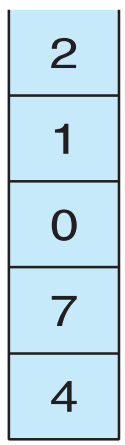
- **Counter implementation**
 - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
 - When a page needs to be changed, look at the counters to find smallest value
 - Search through table needed
- **Stack implementation**
 - Keep a stack of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



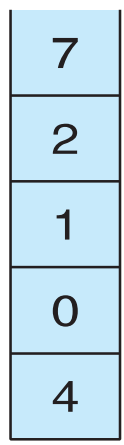
Use Of A Stack to Record Most Recent Page References

reference string

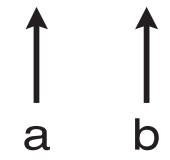
4 7 0 7 1 0 1 2 1 2 7 1 2



stack
before
a



stack
after
b



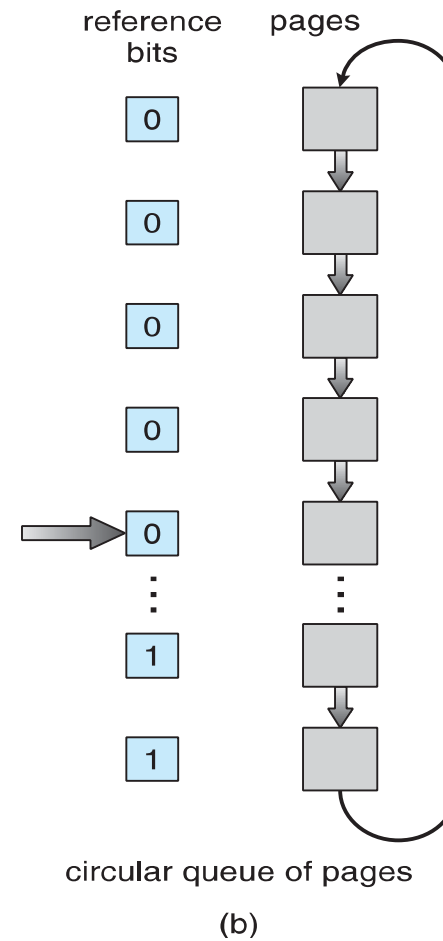
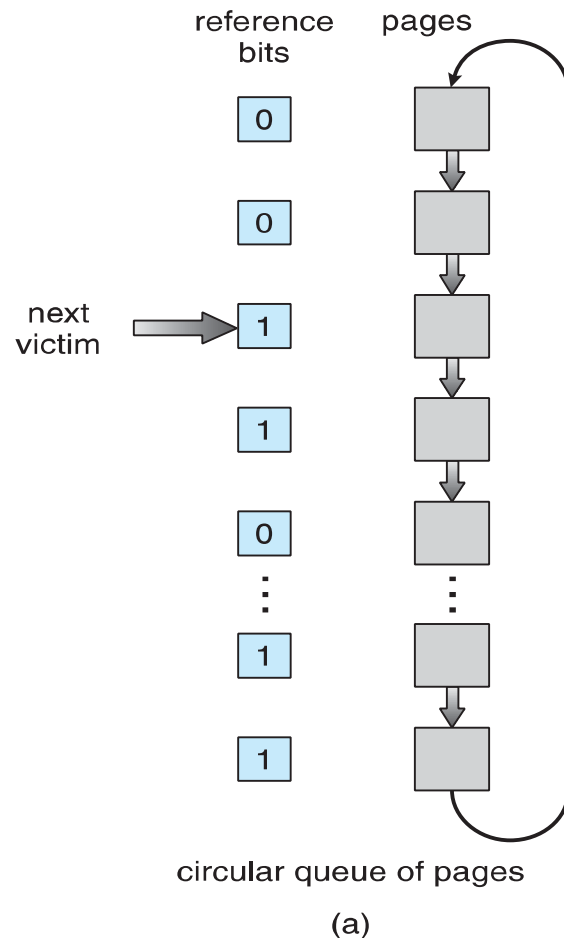


LRU Approximation Algorithms

- 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





Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)
 1. (0, 0) **neither recently used not modified** – best page to replace
 2. (0, 1) **not recently used but modified** – not quite as good, must write out before replacement
 3. (1, 0) **recently used but clean** – probably will be used again soon
 4. (1, 1) **recently used and modified** – probably will be used again soon and need to write out before replacement



Counting Algorithms

- Keep a counter of the number of references that have been made to each page
 - Not common
- **Least Frequently Used (LFU) Algorithm**: replaces page with smallest count
- **Most Frequently Used (MFU) Algorithm**: based on the argument that the page with the smallest count was probably just brought in and has yet to be used

Page-Buffering Algorithms

- Keep a pool of free frames, always
 - Then frame available when needed, not found at fault time
 - Read page into free frame and select victim to evict and add to free pool
 - When convenient, evict victim
- Possibly, keep list of modified pages
 - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
 - If referenced again before reused, no need to load contents again from disk
 - Generally useful to reduce penalty if wrong victim frame selected



Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e. databases
- Memory intensive applications can cause double buffering
 - OS keeps copy of page in memory as I/O buffer
 - Application keeps page in memory for its own work
- Operating system can given direct access to the disk, getting out of the way of the applications
 - **Raw disk** mode
- Bypasses buffering, locking, etc



Allocation of Frames

- Each process needs **minimum** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
 - instruction is 6 bytes, might span 2 pages
 - 2 pages to handle *from*
 - 2 pages to handle *to*
- **Maximum** of course is total 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 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
 - 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
 - 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



Non-Uniform Memory Access

- So far all memory accessed equally
- Many systems are **NUMA** – speed of access to memory varies
 - Consider system boards containing CPUs and memory, interconnected over a system bus
- Optimal performance comes from allocating memory “close 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 by Solaris by creating **lgroups**
 - Structure to track CPU / Memory low latency groups
 - Used my schedule and pager
 - When possible schedule all threads of a process and allocate all memory for that process within the lgroup

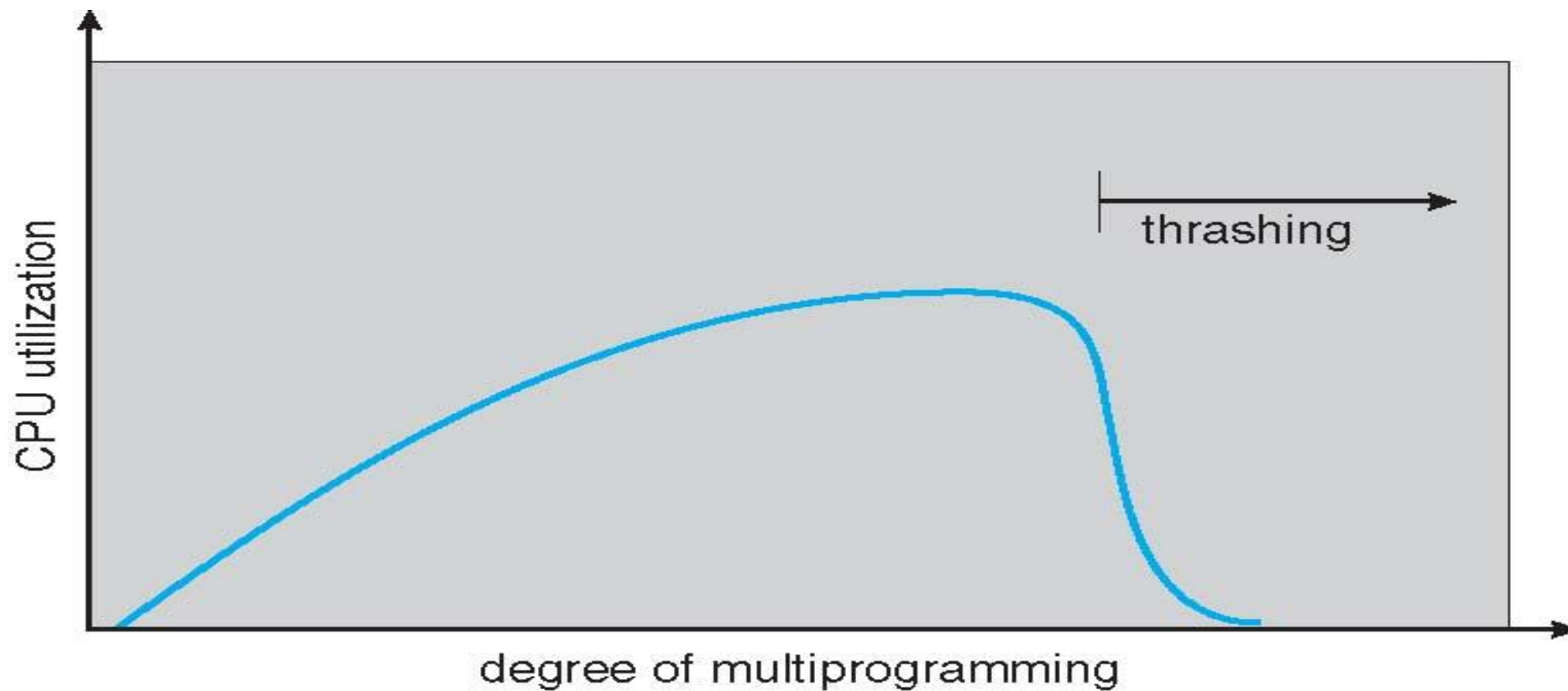


Thrashing

- 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

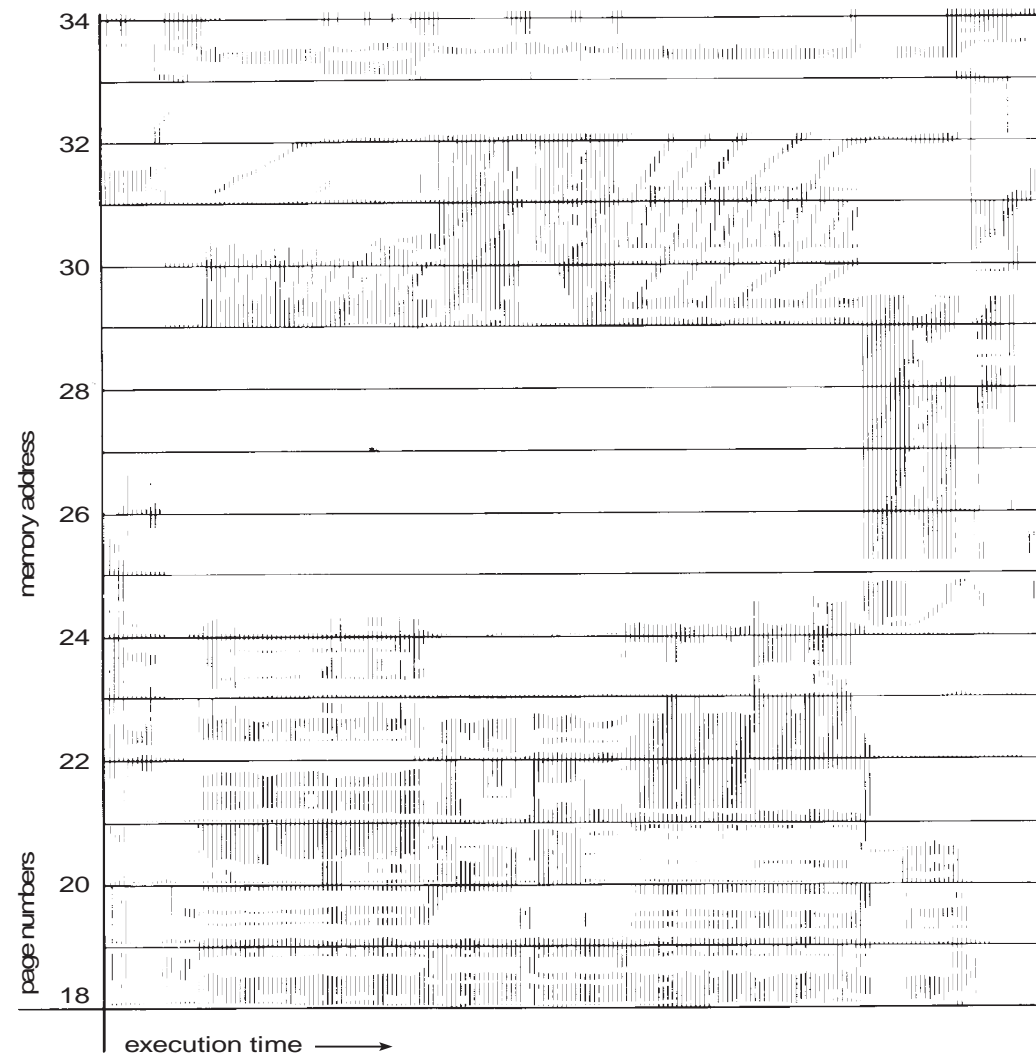
- Why does demand paging work?

Locality model

- Process migrates from one locality to another
 - Localities may overlap
- Why does thrashing occur?
 Σ size of locality > total memory size
 - Limit effects by using local or priority page replacement



Locality In A Memory-Reference Pattern



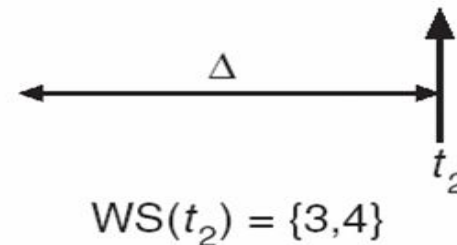
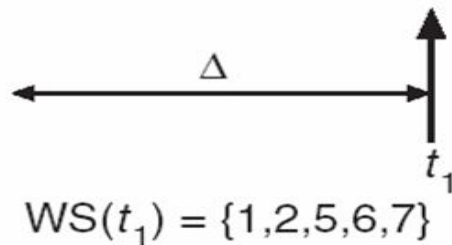


Working-Set Model

- $\Delta \equiv$ working-set window \equiv a fixed number of page references
Example: 10,000 instructions
- 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
- $D = \sum WSS_i \equiv$ total demand frames
 - Approximation of locality
- if $D > m \Rightarrow$ Thrashing
- Policy if $D > m$, then suspend or swap out one of the processes

page reference table

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





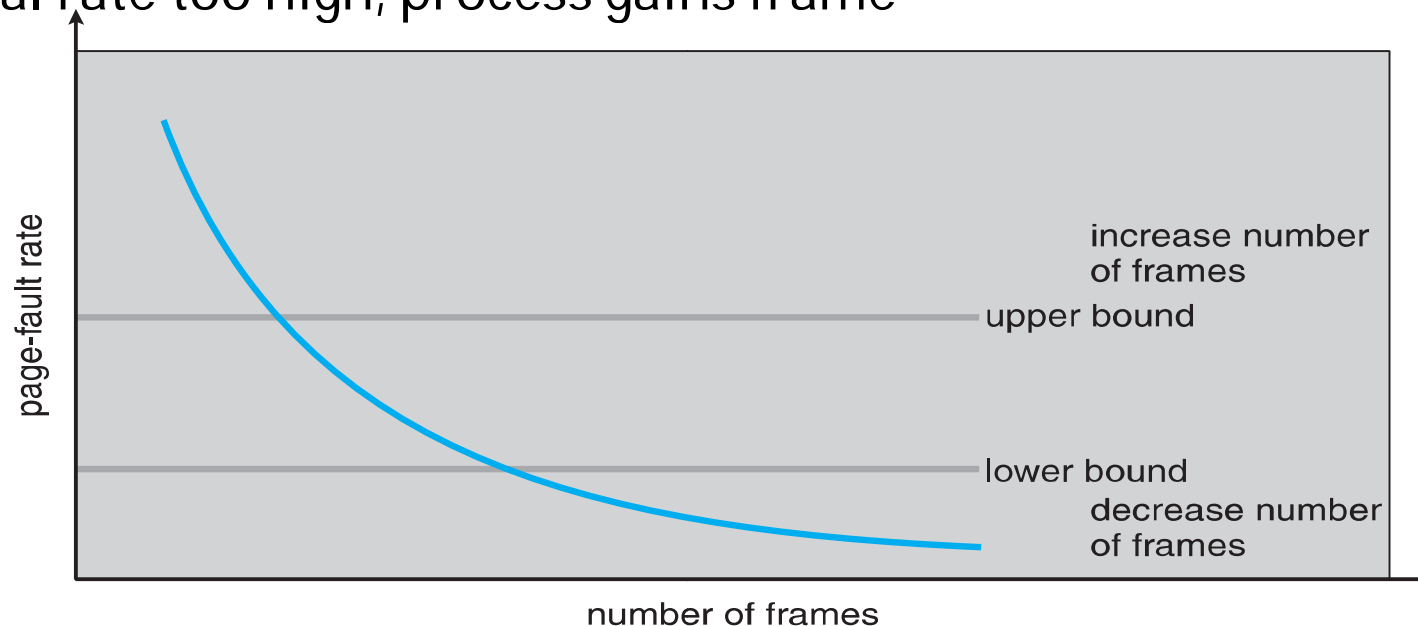
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: $\Delta = 10,000$
 - Timer interrupts after every 5000 time units
 - Keep in memory 2 bits for each page
 - Whenever a timer interrupts copy and sets 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?
- Improvement = 10 bits and interrupt every 1000 time units



Page-Fault Frequency

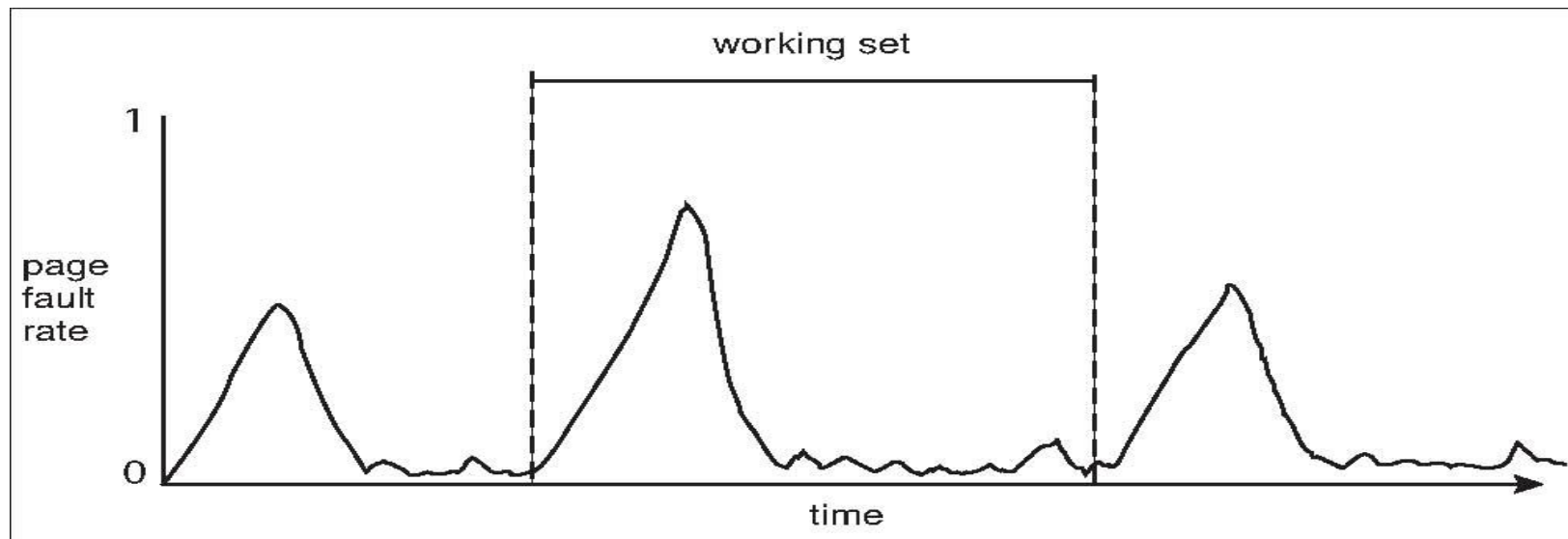
- More direct approach than WSS
- Establish “acceptable” **page-fault frequency (PFF)** rate and use local replacement policy
 - If actual rate too low, process loses frame
 - If actual rate too high, process gains frame





Working Sets and Page Fault Rates

- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time





TEXT BOOK

1. Abraham Silberschatz, Peter B. Galvin, "Operating System Concepts", 10th Edition, John Wiley & Sons, Inc., 2018.
2. Jane W. and S. Liu. "Real-Time Systems". Prentice Hall of India 2018.
3. Andrew S Tanenbaum, Herbert Bos, Modern Operating Pearson , 2015.

REFERENCES

1. William Stallings, "Operating Systems: Internals and Design Principles", 9th Edition, Prentice Hall of India., 2018.
2. D.M.Dhamdhere, "Operating Systems: A Concept based Approach", 3rd Edition, Tata McGraw hill 2016.
3. P.C.Bhatt, "An Introduction to Operating Systems–Concepts and Practice", 4th Edition, Prentice Hall of India., 2013.

THANK YOU