Chapter 9: Virtual Memory

Chapter 9: Virtual Memory

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

Objectives

- To describe the benefits of a virtual memory system
  - Goal of memory-management strategies: keep many processes in main memory to allow multi-programming; see Chap-8
    - Problem: Entire processes must be in memory before they can execute
  - Virtual Memory technique: running process need not be in memory entirely
    - Programs can be larger than physical memory
    - Abstraction of main memory; need not concern with storage limitations
    - Allows easy sharing of files and memory
    - Provide efficient mechanism for process creation
- 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

Background

Background

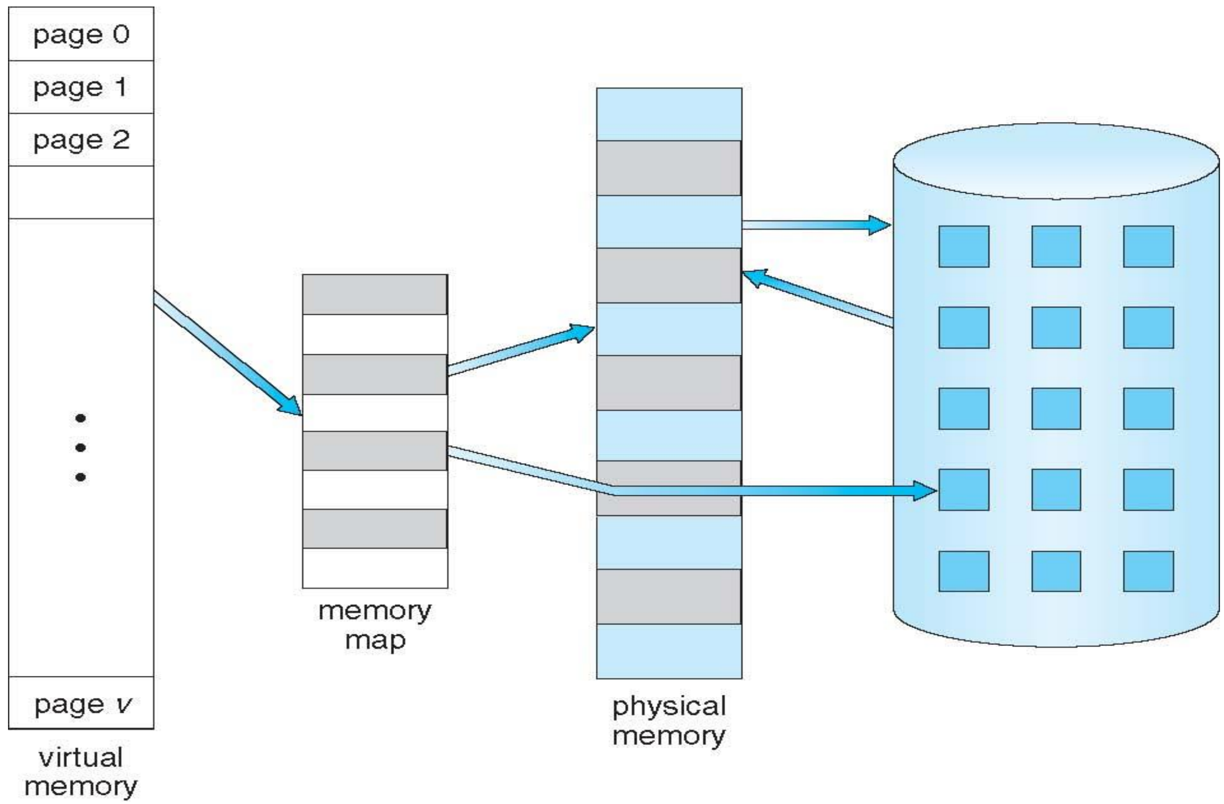
- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures; are all seldom used
- Entire program code not needed (in main memory) at same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running
    - Thus, more programs run at the same time
    - Increased CPU utilization and throughput with no increase in response time or turnaround time; more multi-programming

Background (Cont.)

- **Virtual address space** – logical view of how process is stored in memory
  - Meanwhile, physical memory organized in page frames; not contiguous (see Chap-8)
- Virtual memory can be implemented via:
  - Demand paging
  - Demand segmentation

Virtual Memory That is Larger Than Physical Memory

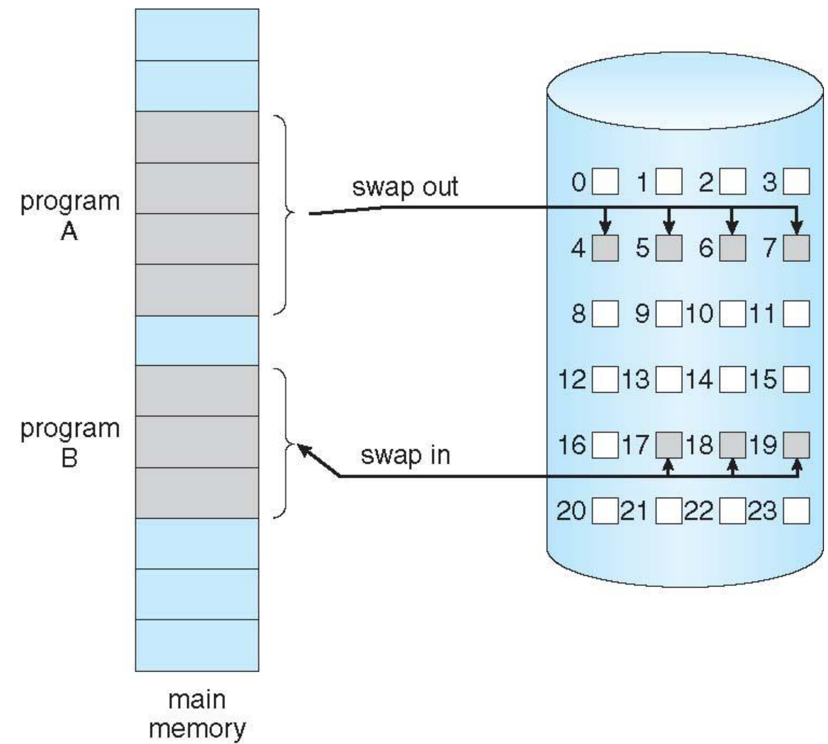
- So far, we assumed all pages of a process shall be present in Physical memory at the time of execution but in reality only a few pages will be present in the physical memory.



Demand Paging

Demand Paging

- An entire process could be brought 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 swap in a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



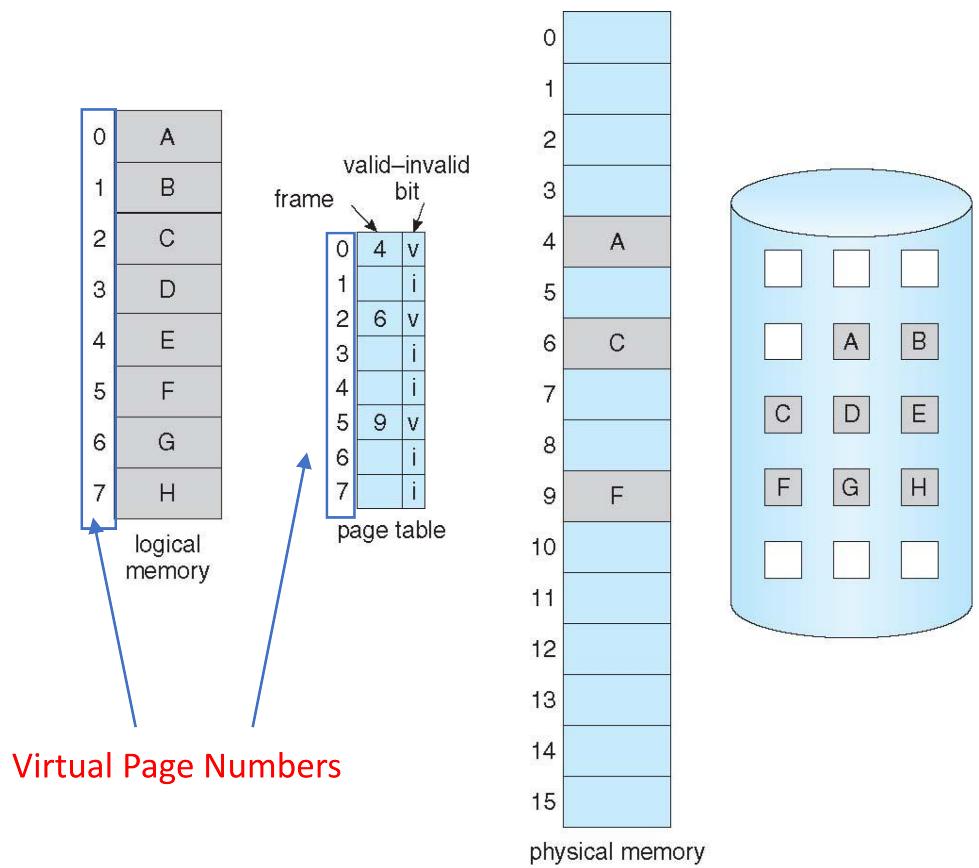
Demand Paging: Basic Concepts

- When swapping in a process, the pager ***guesses*** which pages will be used before swapping out again
- The pager brings in only those needed pages into memory
  - Thus, decreases swap time and amount of needed physical memory
- *How to determine that set of pages?*
- Need new MMU functionality to implement **demand paging**
  - To distinguish between in-memory pages and on-disk pages
    - Uses the valid—invalid scheme of Slide-40 Chap-8
- If pages needed are already **memory resident**
  - Execution proceeds normally
- 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:

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



Page Fault

- What if the process refers to (i.e., tries to access) a page not in-memory ?
- If there is a reference to a page, first reference to that page will trap to operating system:

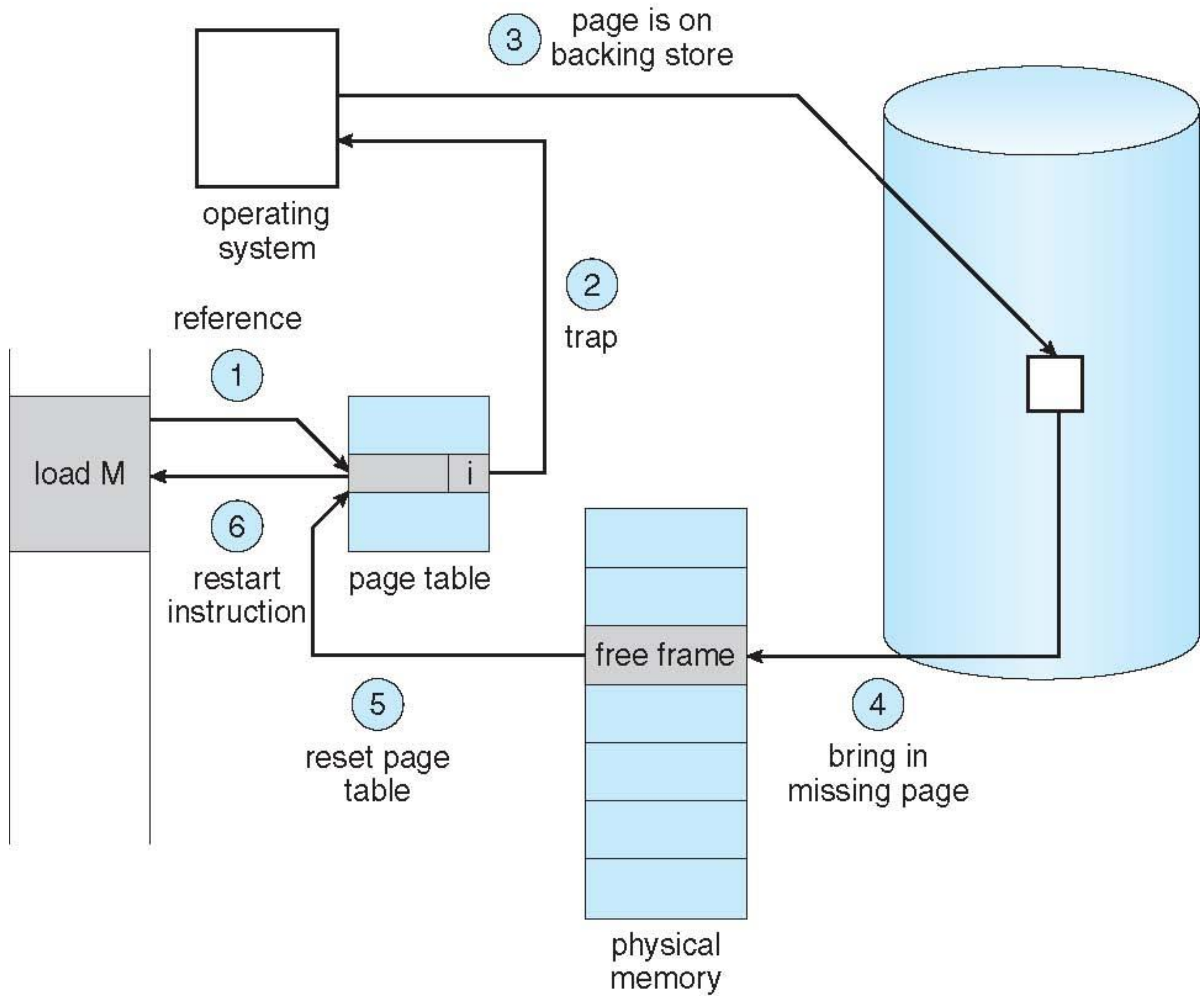
    page fault

- Procedure for handling a page fault

1. Operating system looks at the reference address to decide:
    - Illegle address $\Rightarrow$ abort
        - address is not in logical address space of process
    - Just not in memory => bring to memory
        - logical address is valid but page is simply 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**

- Restart the instruction that caused the page fault and and resume process execution

Steps in Handling a Page Fault



Performance of Demand Paging

- What is the ***Effective Access Time*** 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

- Not all steps mentioned in previous slide are necessary in every case; e.g., Step-6
- Three major components of the page-fault service time
  - Service the interrupt; between 1-100 microseconds
  - Read the page – lots of time; at least 8ms
  - Restart the process – again just a small amount of time (b/w 1-100 microseconds)
- Page Fault Rate $0 \le p \le 1.0$
  - if p = 0; then there is no page faults
  - if p = 1, then every memory reference causes a page-fault
- **Effective Access Time (EAT)**

$$\text{EAT} = (1 - p) \times \text{memory access} + p\ (\text{page\_fault\_time})$$

where:

page_fault_time = page fault overhead + swap page out+ swap page in+ restart overhead

Demand Paging Example

- Memory access time = 200 nanoseconds; between 10 to 200ns in most computers
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (1 – p) x 200 + p x 8,000,000

  = 200 + p x 7,999,800; thus EAT is directly proportional to p
- If one access out of 1,000 (p=0.001) causes a page fault, then

  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40 (when p=0). Because of demand paging
- If want performance degradation < 10 percent, i.e., EAT = 220
  - 220 > 200 + 7,999,800 x p

    20 > 7,999,800 x p
  - Thus we must have p < .0000025

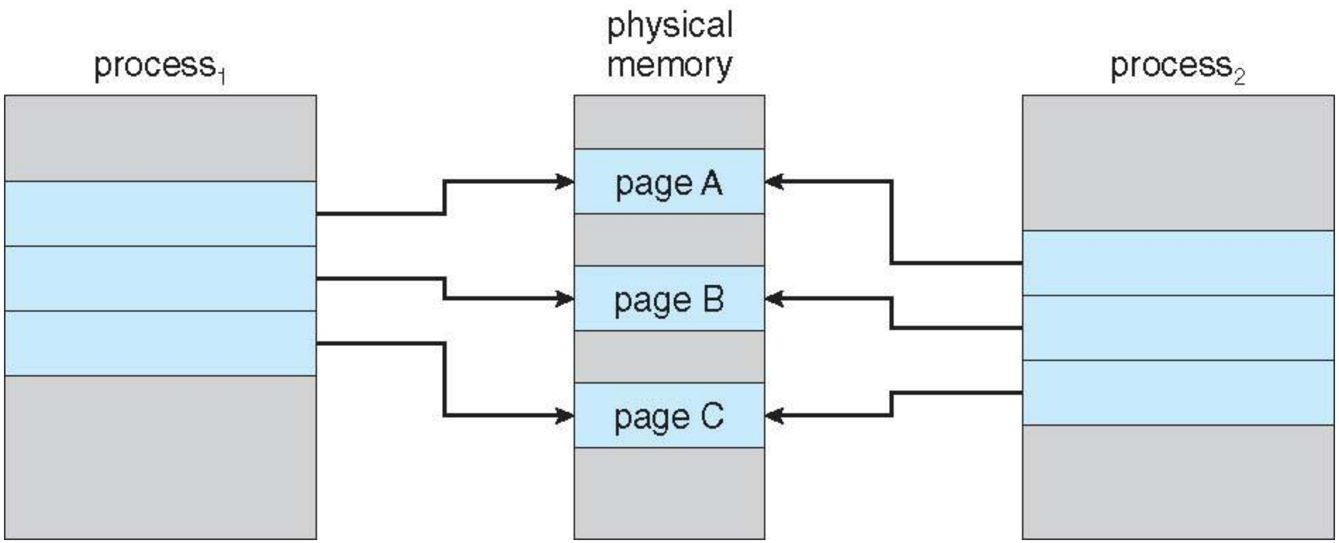
  That is, to keep slowdown due to demand paging
  - < one page fault in every 399,999 memory accesses

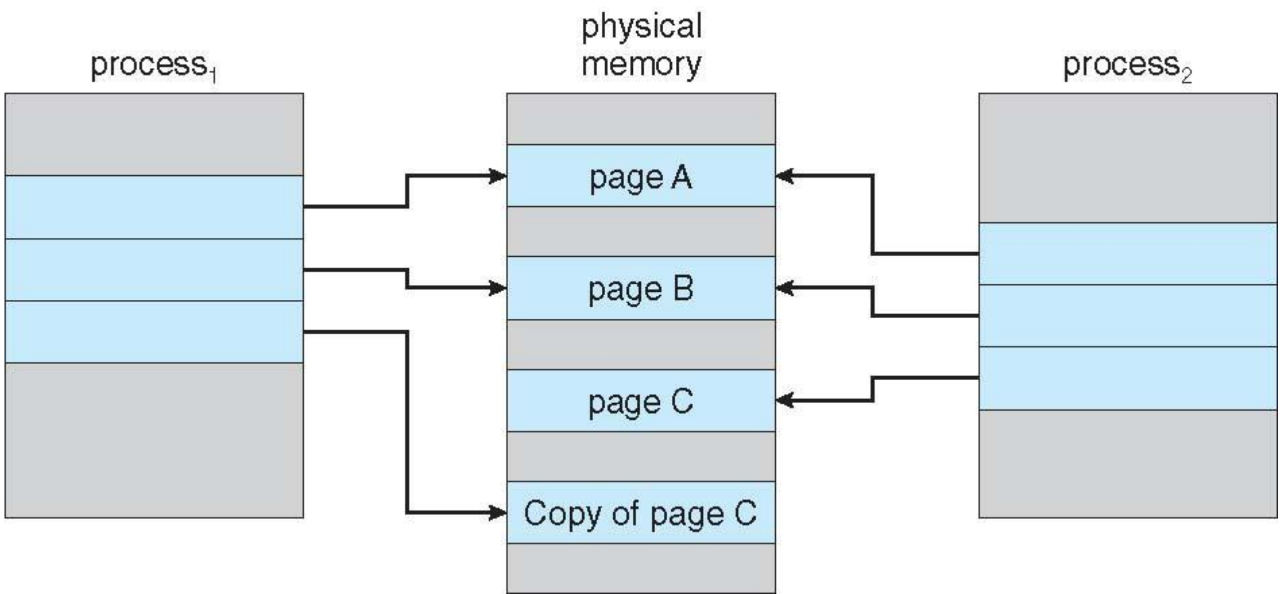
Copy-on-Write

Copy-on-Write

- **Fork()** system call may initially bypass the need for demand paging using page sharing
  - For() produces two very similar processes -- > same code, data , stack.
- **Copy-on-Write** → duplicate the page only if it needs to be modified
- **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
  - **Zero-fill-on-demand** pages have been zeroed-out before being allocated, thus erasing the previous contents
  - 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

Before and After Process 1 Modifies Page C





What Happens if There is no Free Frame?

- Many pages need to be loaded but not enough free frames available for them
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- Solution: Page replacement-when paging in pages of a process but no free frames
  - terminate the process? **No**
  - swap out some process? **Yes**, but not always a good option
  - Find currently un-used frame to free it; Page it out and page in process page
    - Replacing the un-used memory page with the new page
  - Performance – want an ***algorithm*** which will result in minimum number of page faults
- Same page may be brought into memory several times

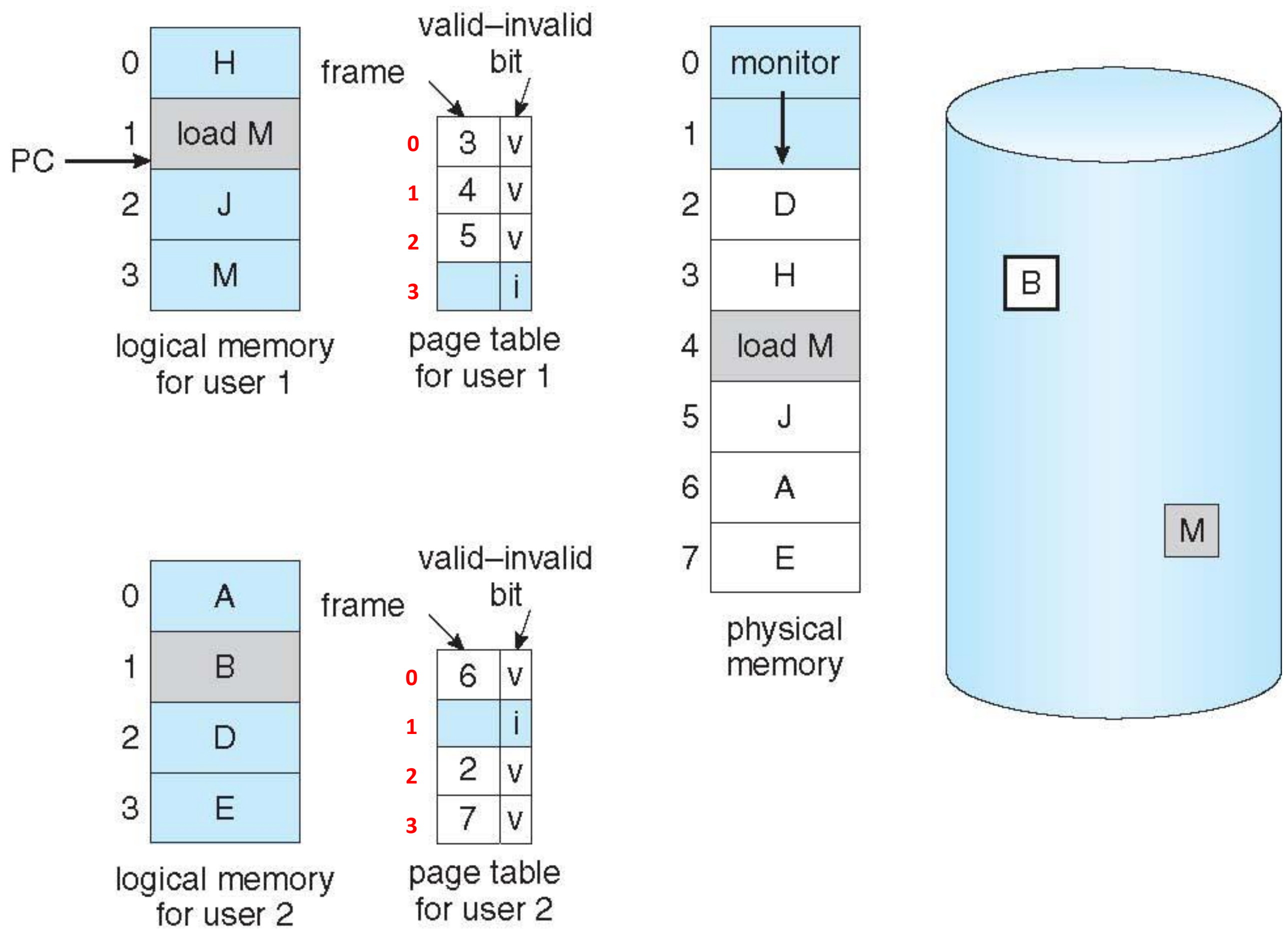
Chapter 9: Virtual Memory

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

Page Replacement

Need for Page Replacement

- Need for page replacement
  - Prevent **over-allocation** of memory by modifying page-fault service routine to include page replacement
  - Page fault occurs while executing ‘load M’ in page#1 in process 1.
  - Below, page#1 needs to load M, which has not been loaded yet due to the unavailability of the frame in the physical memory



Basic Page Replacement Algorithm

- The page-fault service routine is modified to include page replacement

1. Find the location of the desired page on disk

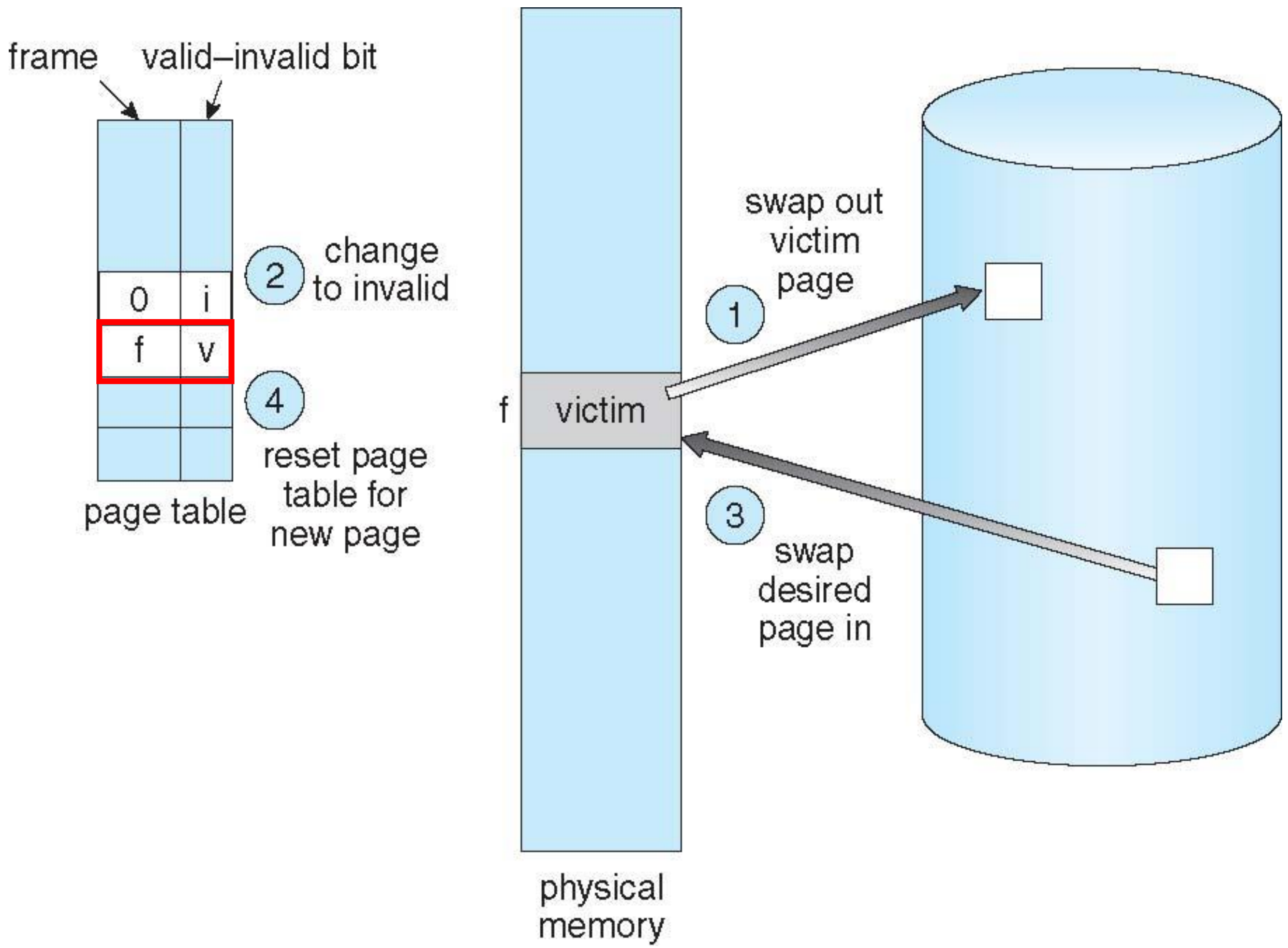
2. Find a free frame:
   - If there is a free frame, use it
   - If there is no free frame, use a ***page replacement algorithm*** to select a **victim frame**
     - Write victim frame to disk **[if dirty]**; change the page and the frame tables accordingly

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
  - Only if no frames are free; one ***page in*** required and one ***page out*** required

Page Replacement



Using modify-bit (or dirty bit) to reduce overhead

- What if the **victim** page has not been modified since the last access?
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
  - Each page or frame is associated with a ***modify bit***
  - Set by the hardware whenever a page is modified
- Page replacement completes separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
  - A user process of 20 pages can be executed in 10 frames simply by using demand-paging and using a page-replacement algorithm to find a free frame whenever necessary

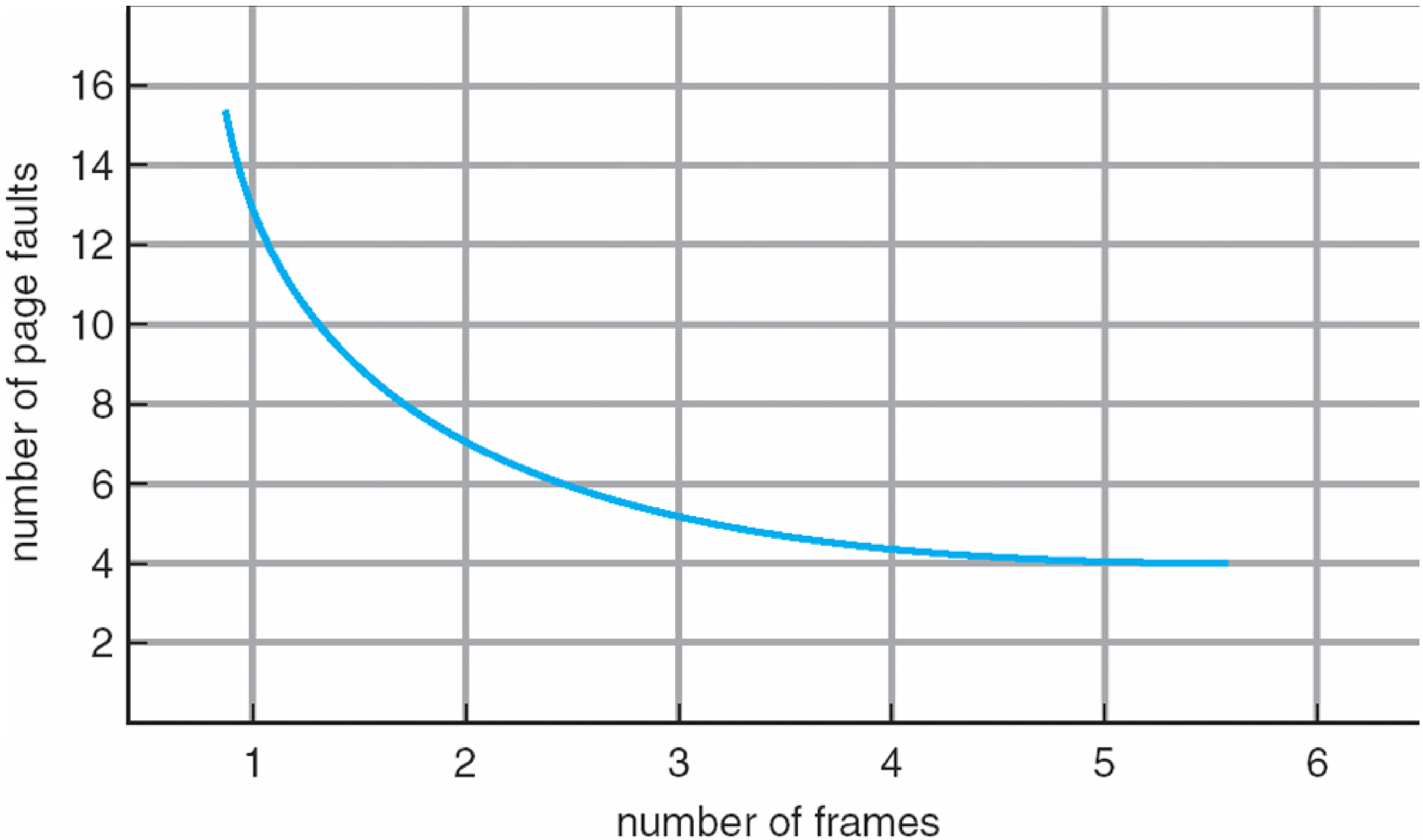
Page and Frame Replacement Algorithms

- Two major demand-paging problems : frame allocation and page replacement
- **Frame-allocation algorithm** determines
  - How many frames to give each process?
  - Which frames to replace?; when page replacement is required.
- **Page-replacement algorithm**
  - We want lowest page-fault rate on both first access and re-access
- How to evaluate Page Replacement Algorithm: 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 ***p***, 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

for a memory with three frames.

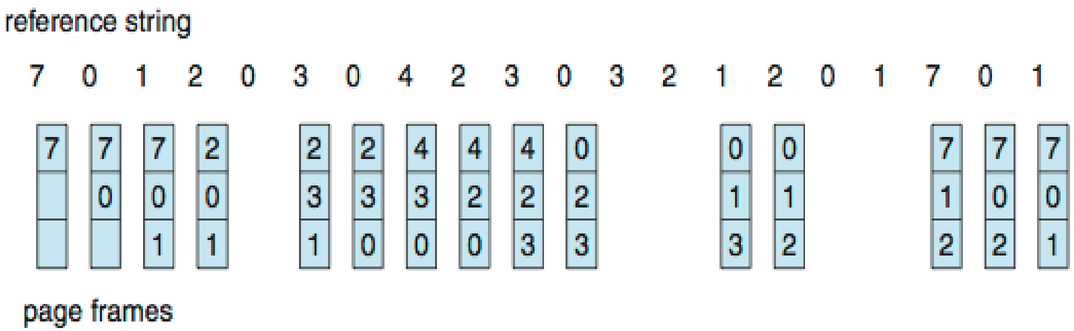
Graph of Page Faults Versus The Number of Frames



1. 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**
- And memory = 3 frames (3 pages can be in memory at a time per process)
- Each page brought into memory is also inserted into a first-in first-out queue
  - Page to be replaced is the oldest page; the one at the head of the queue

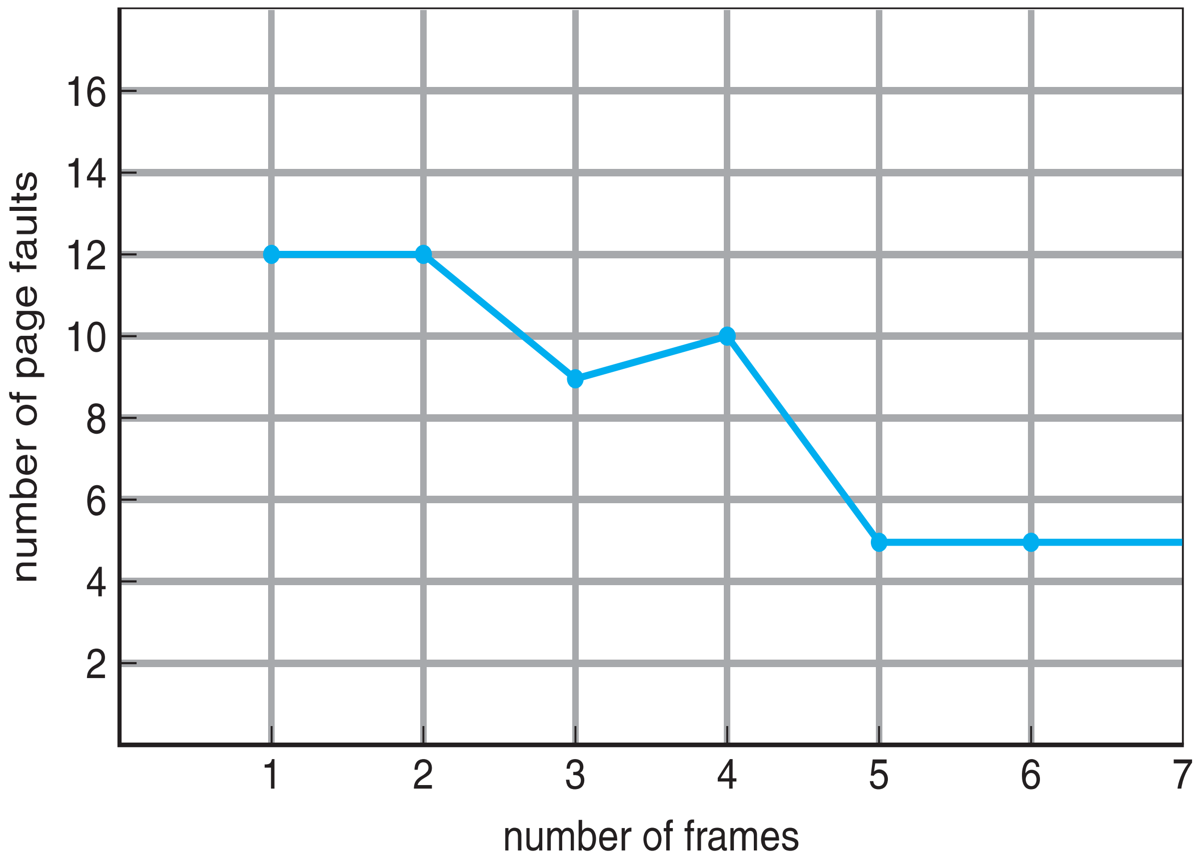
Our example yields 15 page faults



- 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

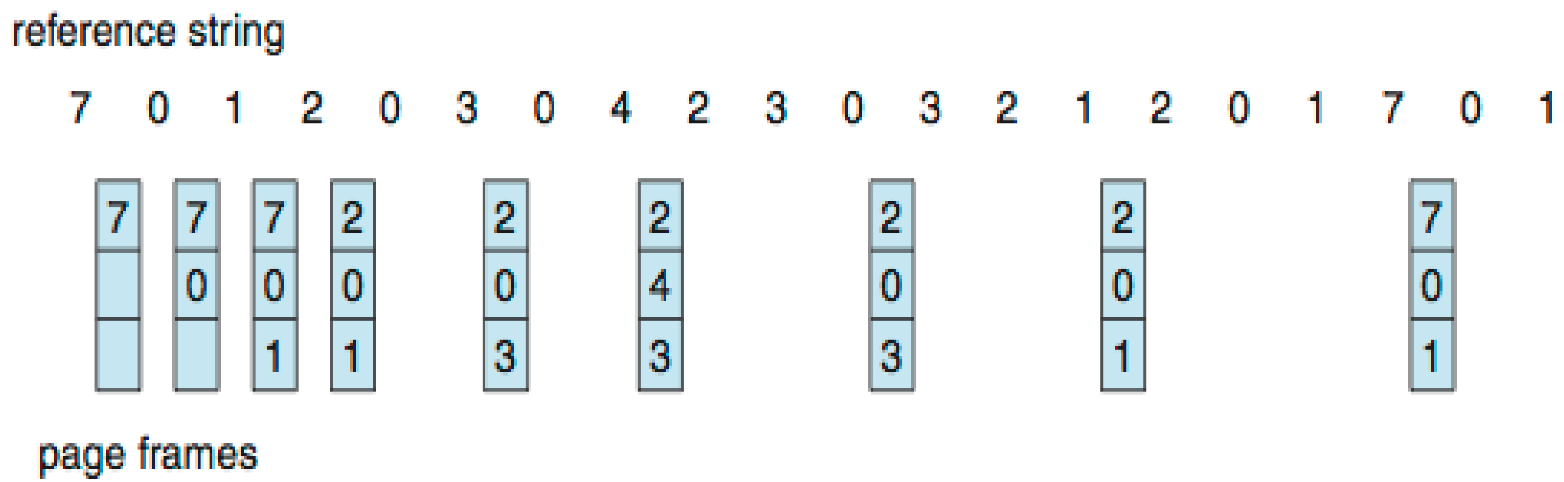
FIFO Illustrating Belady's Anomaly

- Notice that the number of faults are ten for four frames, is greater than the number of faults (nine) for three frames.
  - consider 1,2,3,4,1,2,5,1,2,3,4,5
  - This most unexpected result is known as ***Belady's anomaly***.



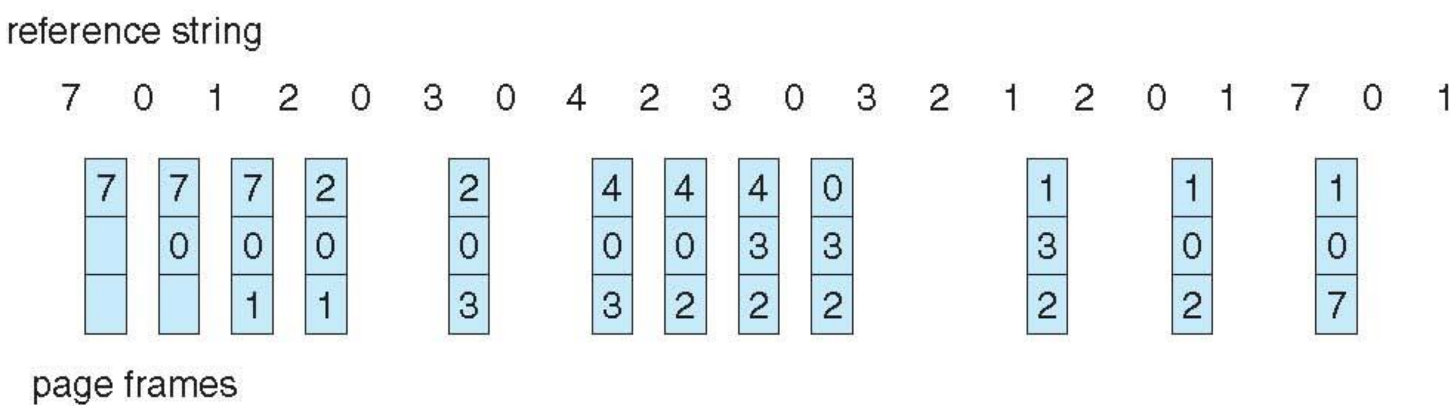
2. Optimal Page Replacement Algorithm

- Replace page that will not be used for longest period of time
  - The number of page faults nine (9) is optimal for the example
- Unfortunately, OPR is not feasible to implement
  - Because: we can't know the future; i.e., what is the next page?
    - We have assumed that we know the reference string. No, we don't
- OPR is used only for comparing with new algorithms; how close to the optimal?



3. Least Recently Used (LRU) Algorithm

- Use past knowledge (rather than future) as an approximation of the near 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 (15 page faults) but worse than OPT
- Generally good algorithm and frequently used
- Algorithm is feasible but not easy to implement.
    - LRU algorithm may require substantial hardware support

3. LRU Algorithm (Cont.)

LRU Implementation Method 1: Counter implementation

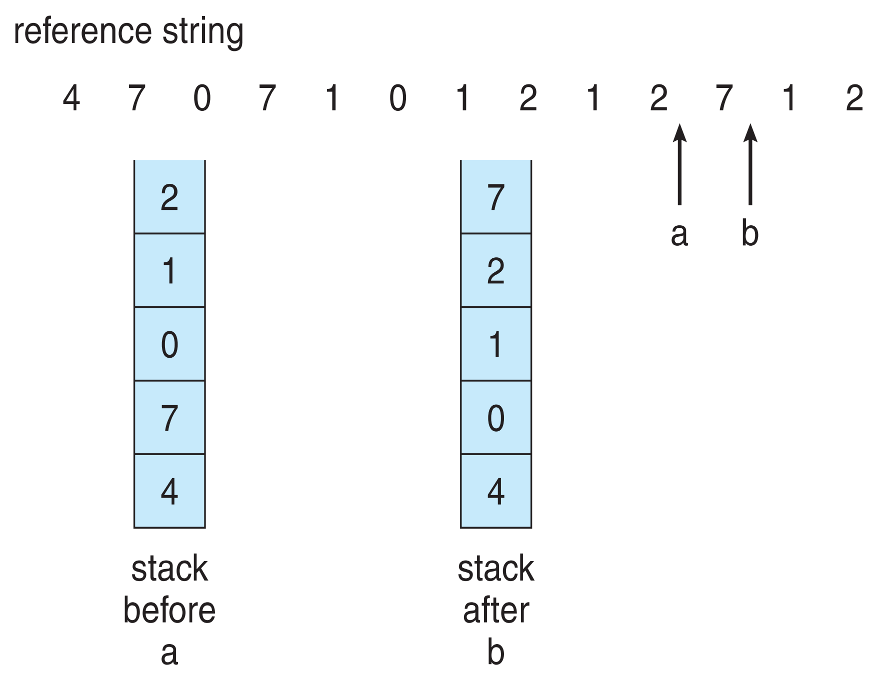
- Every page-table entry has a counter (time-of-use field which it acquires from clock);
- The counter is incremented at every memory access.
- When a page needs to be changed, look at the counters to find smallest value
  - Search through table needed; to find the LRU Page

3. LRU Algorithm (Cont.)

LRU Implementation Method 2: Stack implementation

- Keep a stack of page numbers
- When a page is referenced:
  - move it to the top; most recently used page is always at the top of stack
- But each update more expensive
  - entries must be removed from the middle of the stack.
  - E.g., after 7 is removed from the middle of the stack when it is referenced and put it on the top.

- LRU and OPT are cases of **stack algorithms** that don’t have Belady’s Anomaly



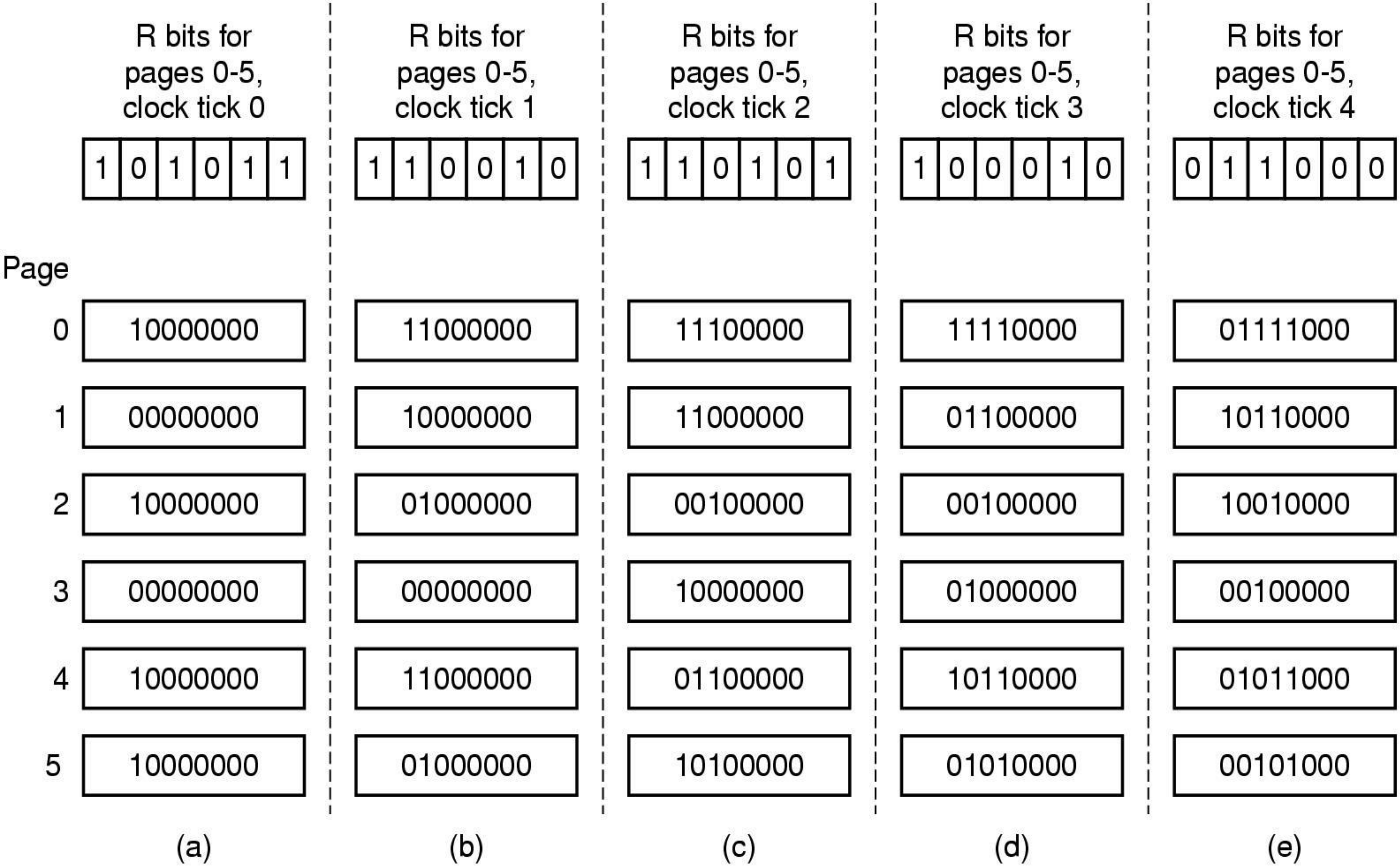
LRU Approximation Page Replacement Algorithms

- **True LRU** needs special hardware and still slow
- **Solution: use 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 can determine which pages have been used and which have not been used by examining the reference bits.
    - We do not know the ***order***, however.

LRU Approximation: Additional- Reference-bits Algorithms

- Ordering Information can be obtained by recording the reference bits at regular interval.
  - Keep an 8-bit byte for each page in a table in memory.
  - At regular intervals, say every 100ms, shift the reference bit of each page into the high-order bit of the byte, shifting the other bits right by 1 bit and discarding the low order bit.
  - Each reference byte keeps the history of the page use (aging) for the last eight time intervals.
  - If we interpret the reference byte as an unsigned integer, the page with the lowest number is the LRU page.
  - If Shift register = 0000 0000 → if page has not been used for eight time periods
  - If Shift register = 1111 1111 → if page is used at least once in each time period
  - A page with history register value 11000100 has been used more recently than 01110111

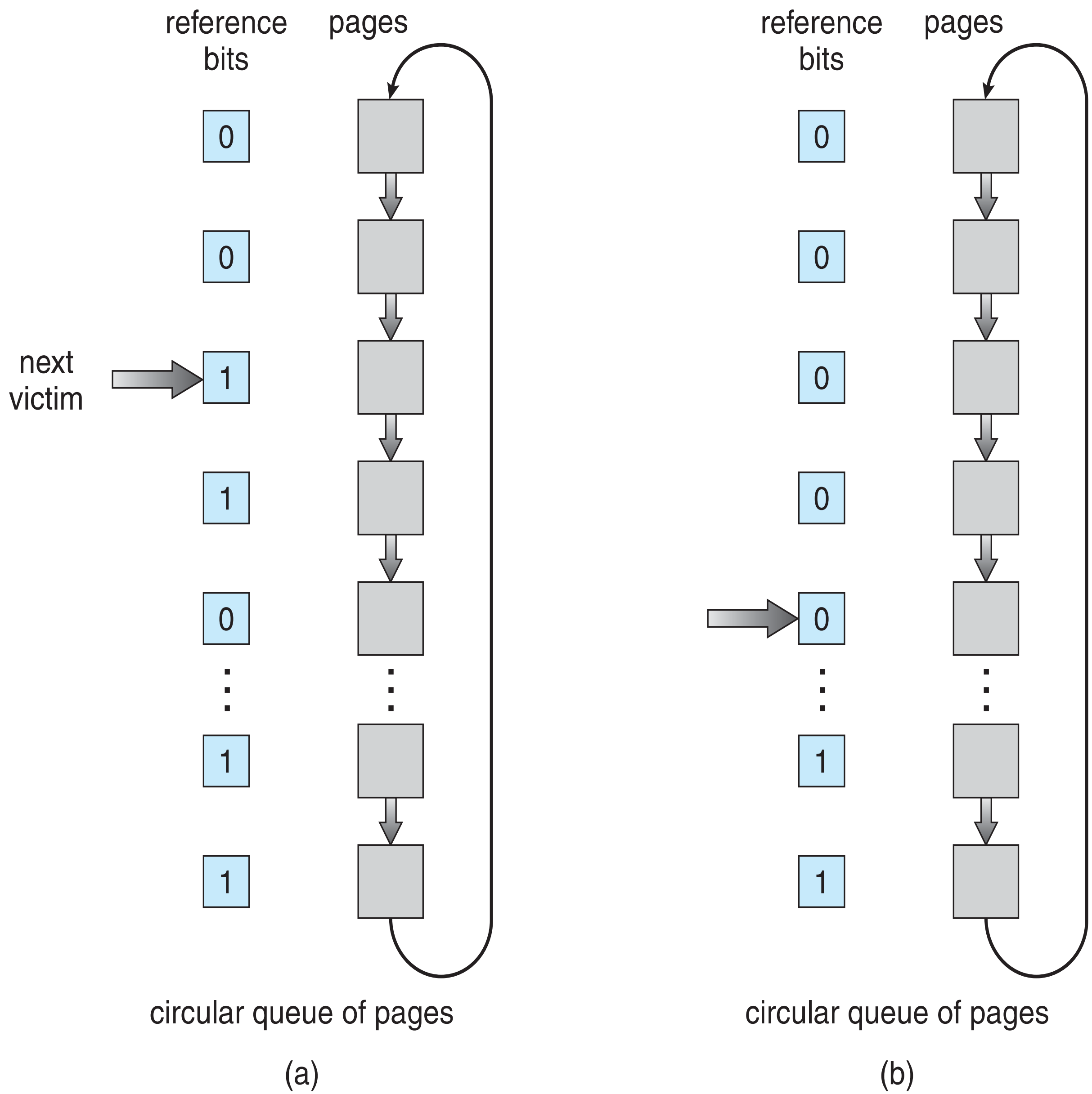
Reference Byte Example



LRU Approximation: Second-chance Algorithm

- The set of frames candidate for replacement is considered as a circular buffer.
- Initially the reference bit for each frame is set to 1.
- If page to be replaced has
    - Reference bit = 0 -> replace it
    - reference bit = 1 then:
        - set reference bit 0, leave page in memory [give it a chance]
        - replace next page, subject to same rules.

Second-Chance (clock) Page-Replacement Algorithm



The Clock Policy: Another Example

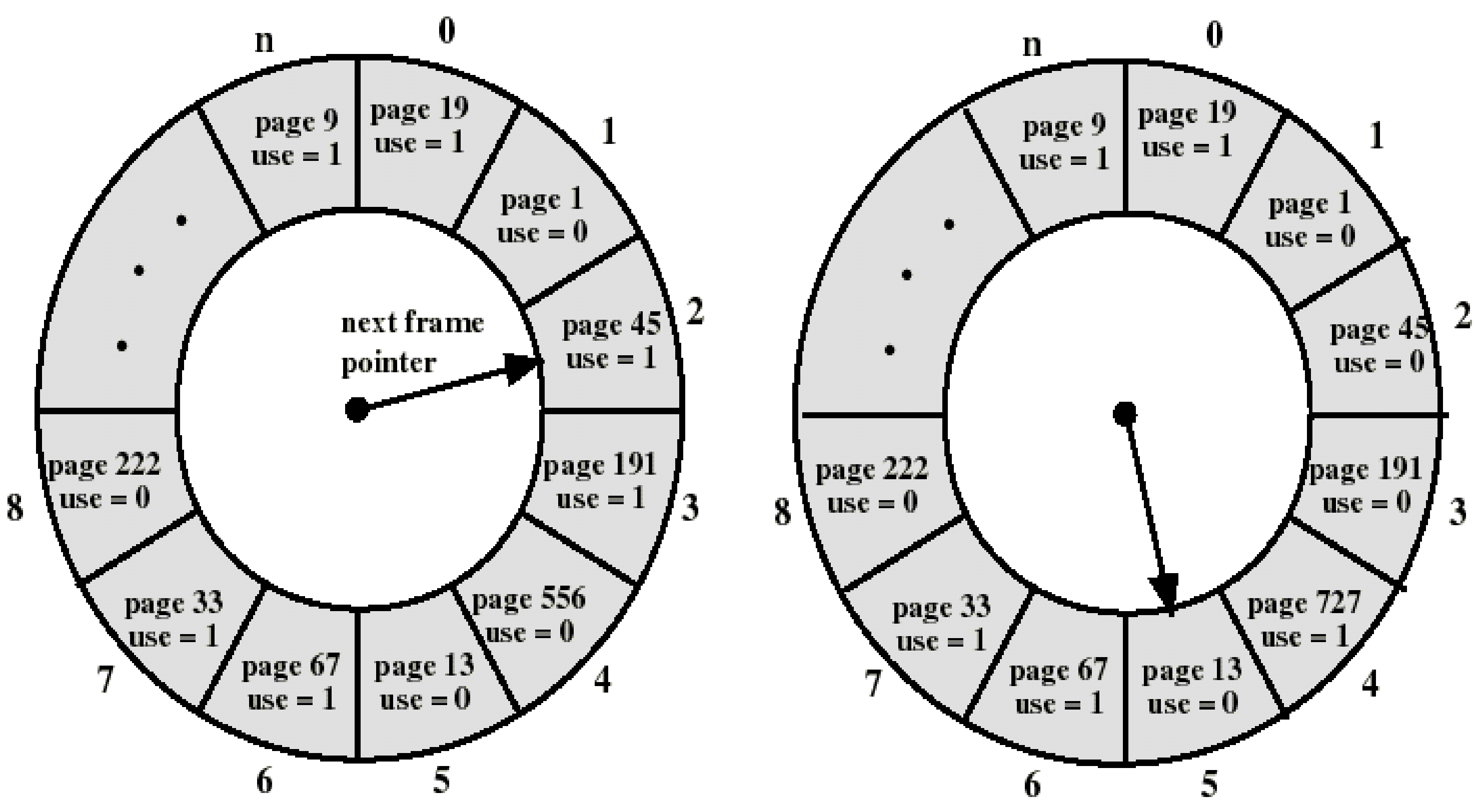
In this example, Use bit = reference bit

Before Page Replacement:

Reference bit for Page#45 and 191 are set to 0.

After Page replacement:

Reference bit for page#556 is 0 so it is paged out (replaced). Now the next candidate for replacement is page 13.



(a) State of buffer just prior to a page replacement

(b) State of buffer just after the next page replacement

Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available):
- 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

Enhanced Second-Chance Algorithm

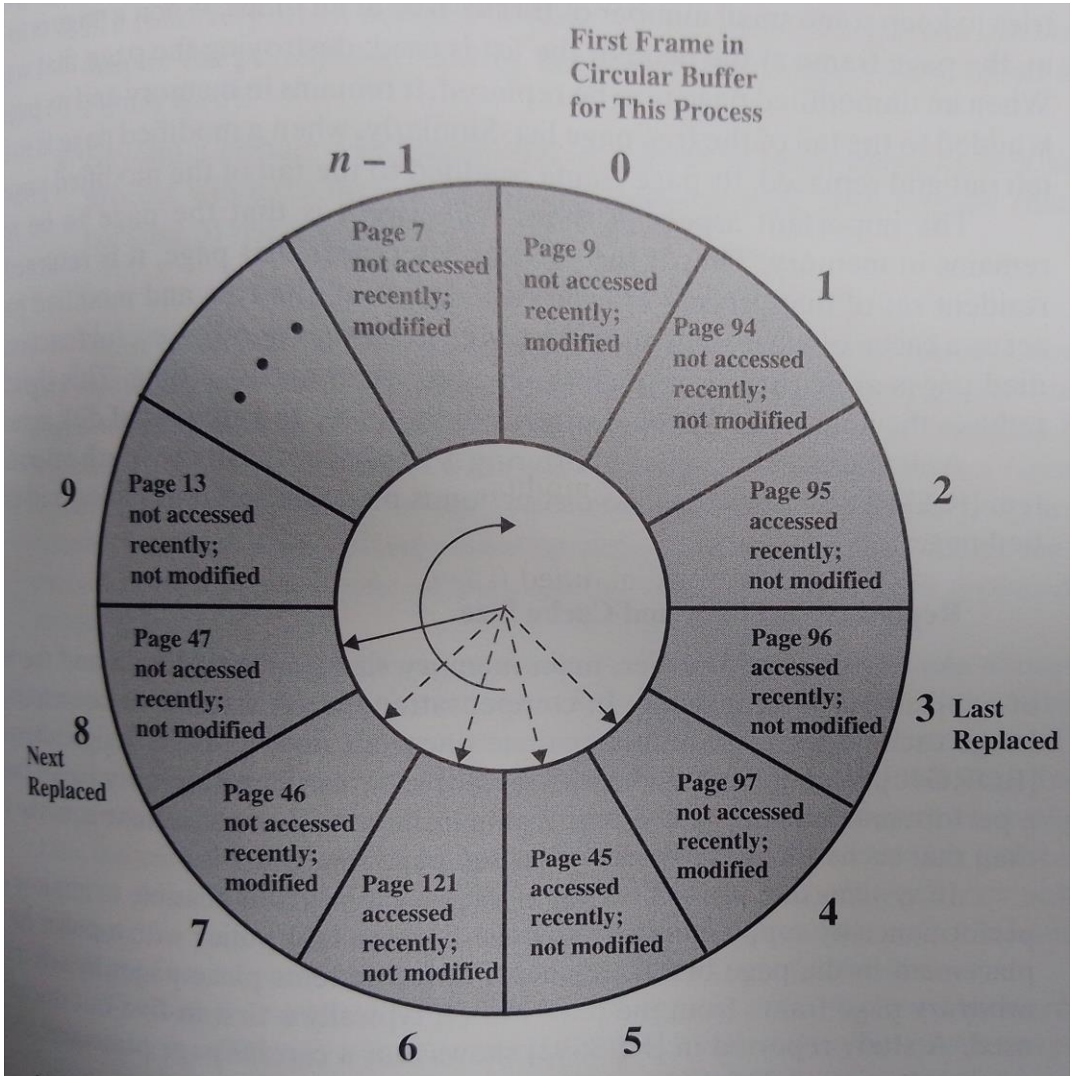


Figure 8.16 The Clock Page-Replacement Algorithm [GOLD89]

Counting based Replacement 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
  - An actively used page should have a large count value
    - But... Pages may be heavily used initially and never used again

- **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

- Counting-based algorithms are very expensive to implement, and they do not approximate OPT replacement well

Allocation of Frames

Allocation of Frames

- Each process needs to be allocated a ***minimum*** number of frames
  - Number of page-faults increases as the # of allocated frames decreases
  - The minimum number of frames is defined by the computer architecture

- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Frame 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
  - **Problem:** Large and small size processes are allocated equal number of frames
- **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 $p_i = \frac{s_i}{S} \times m$

$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} \times 62 \approx 4$$
$$a_2 = \frac{127}{137} \times 62 \approx 57$$

Frame Allocation: Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- Ratio of frames depends on the priorities of processes, or, on a combination of both their priorities and their sizes.
- We may want to allocate more frames to a high-priority process, in order to speed up its execution, to the detriment of low-priority processes.
- In this case, the replacement algorithm is modified to consider process's priorities.
- 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 (frames allocated to other process may have not been used)

Thrashing

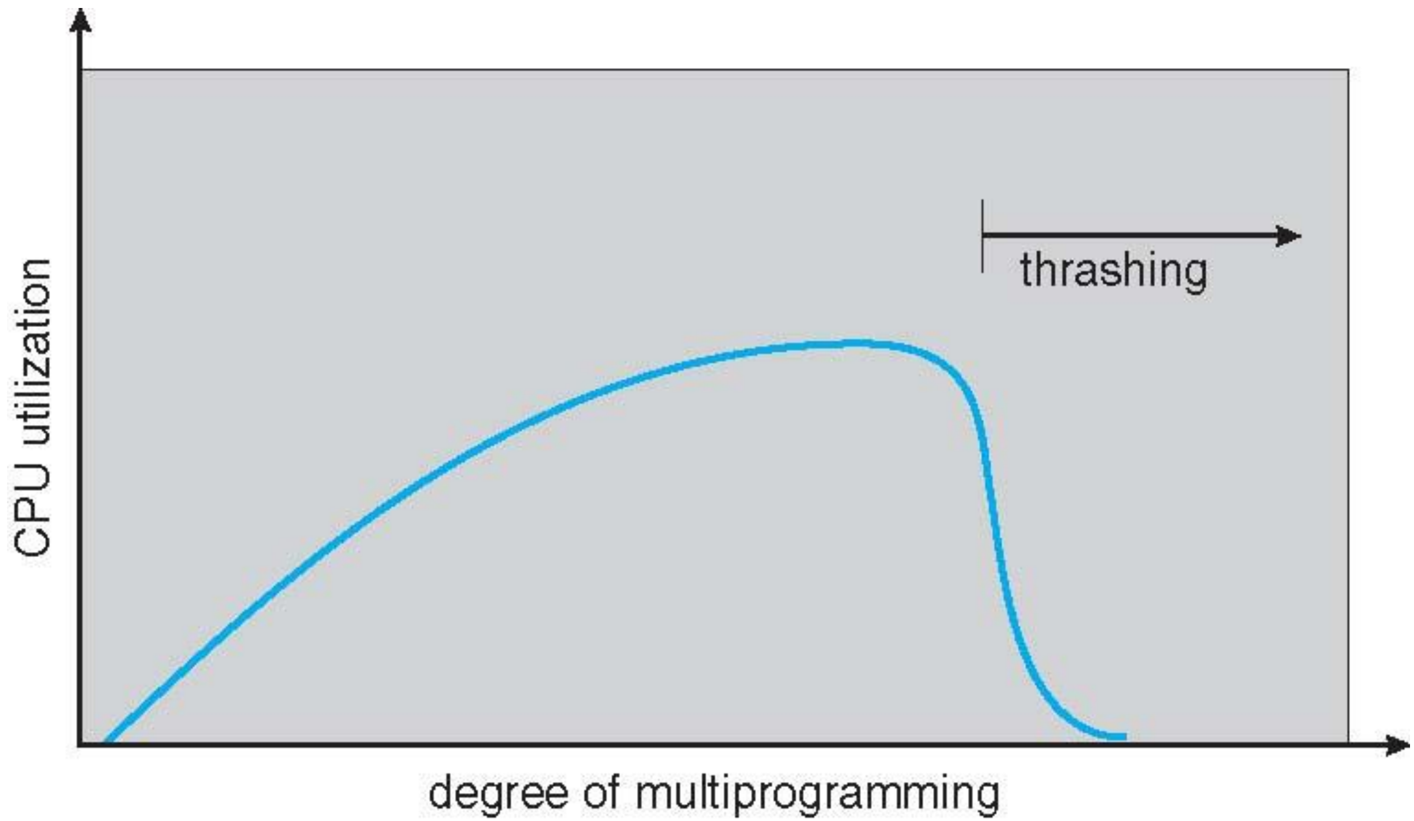
Thrashing

- 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** ≡ a process is busy swapping pages in and out.
  In detail, when the OS brings one piece in, it must throw another out. If it throws out a piece just before it is about to be used, the it will just have to go to get that piece again almost immediately. Too much of this leads to a condition know as **Thrashing**.

Thrashing (Cont.)

Thrashing results in significant performance problems.



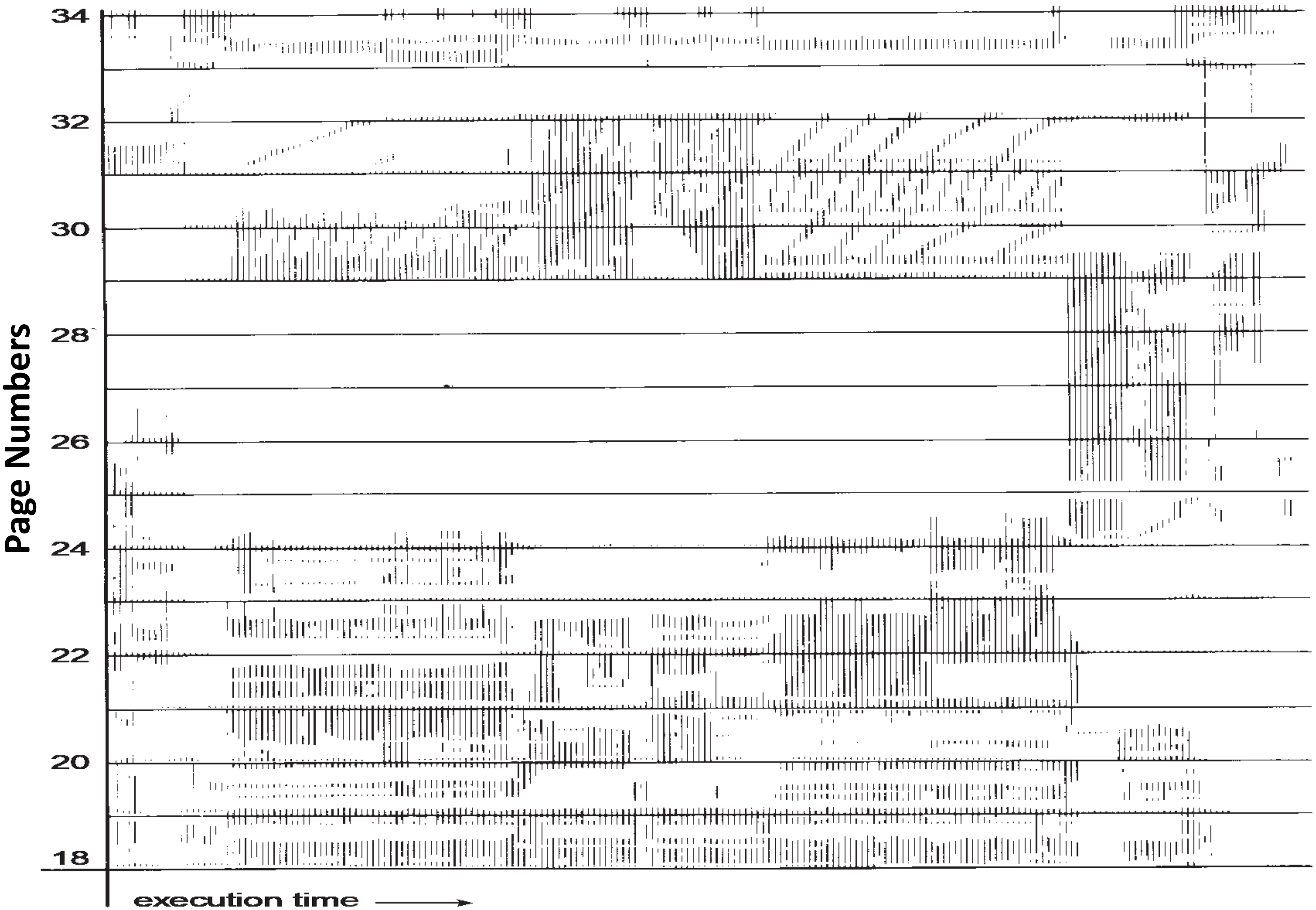
Demand Paging and Thrashing

- Why does demand paging work?
  Locality model
  - Process migrates from one locality to another
  - Localities may overlap
- A ***locality*** is a ***set of pages*** that are actively used together. A program is composed of several different ***localities***, which may overlap (diagram on next slide).

- Why does thrashing occur?
  If the total demand is greater than the total number of available frames ($D > m$), thrashing will occur, because some processes will not have enough frames.

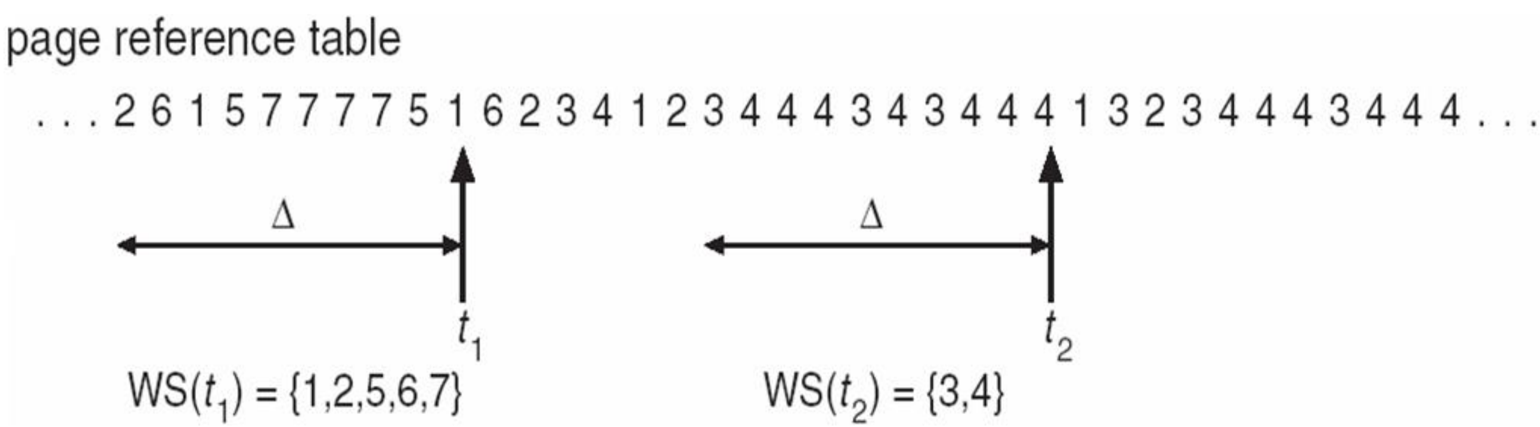
  (Σ size of locality > total memory size)
- We can ***limit*** the effects of thrashing using a local replacement algorithm or priority replacement algorithm. However, the problem is still not entirely solved.
- To prevent thrashing, a process must be allocated as many frames as it needs.
  - One of several techniques is known as *Working Set Strategy*.

Locality In A Memory-Reference Pattern



Working-Set Model

- The set of pages that a process is currently using is called its ***working set***
- When the entire working set is in main memory, few page faults will occur.
- A ***paging system*** which keeps track of the working set and makes sure it is in memory before the process is run is using the ***working set model*** .
- $\Delta \equiv$ working-set window $\equiv$ a fixed number of most recent page references
  Example: 10,000 instructions
- WSS_i (***working set size*** 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
- The working set is an approximation of the program's locality.



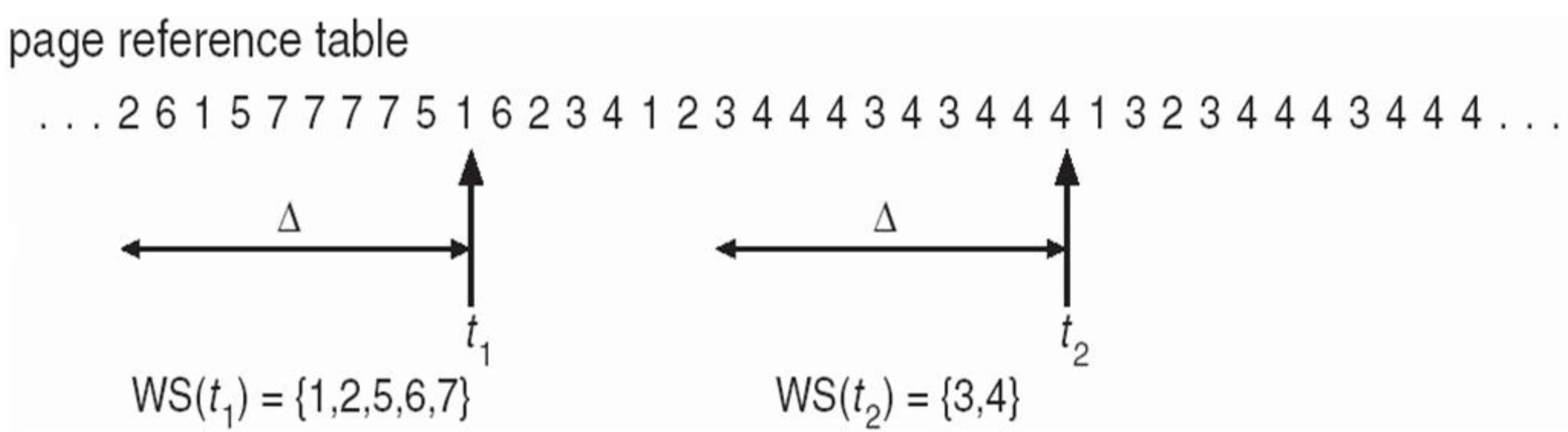
Working-Set Model

- WSS_i :working set size of Process P_i

  $D = \Sigma\ WSS_i \equiv$ total demand frames

- if $D > m \Rightarrow$ Thrashing

- Policy: if D > m, then ***suspend*** or ***swap out*** one of the processes

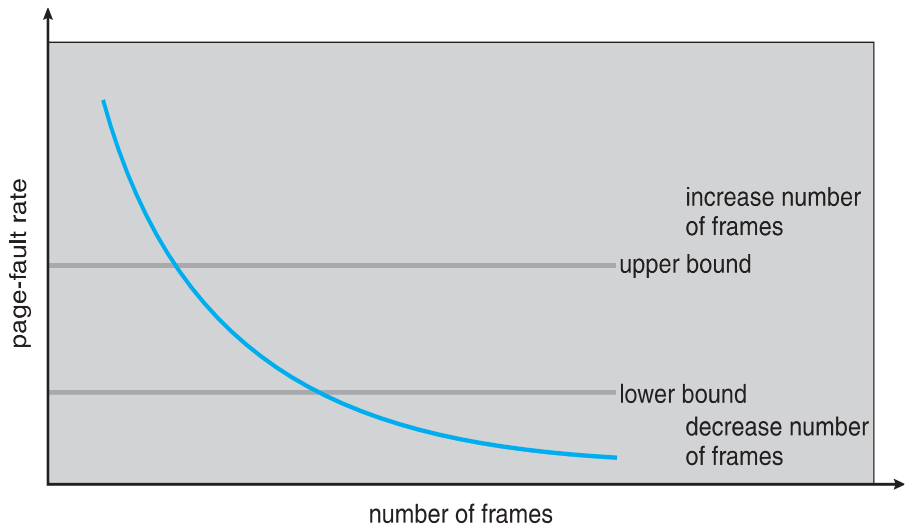


Keeping Track of the Working Set

- The difficulty with the working-set model is keeping track of the working set.
- Working sets tend to increase temporarily as a program moves from one phase of execution to another:
- Solution:
- 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 we copy and clear the reference bit values of each page
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
  - Because we cannot tell where in an interval of 5000, a reference occurred.
- Improvement = 10 bits and interrupt every 1000 time units

Page-Fault Frequency

- More direct approach than WSS
- PFF = page faults / instructions executed
- If PFF rises above threshold, process needs more memory.
  - Not enough memory on the system? → Swap out.
- If PFF sinks below threshold, memory can be taken away.
- 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



End of Chapter 9