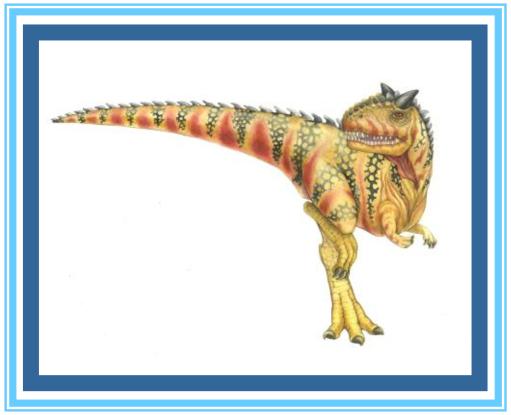
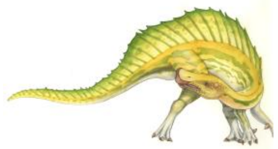
Chapter 8: Virtual Memory



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
  - Less I/O needed to load or swap programs into memory -> each user program runs faster

Virtual memory – separation of user logical memory from physical memory



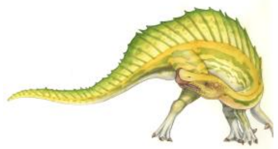
Background (Cont.)

Virtual address space – logical view of how process is stored in memory

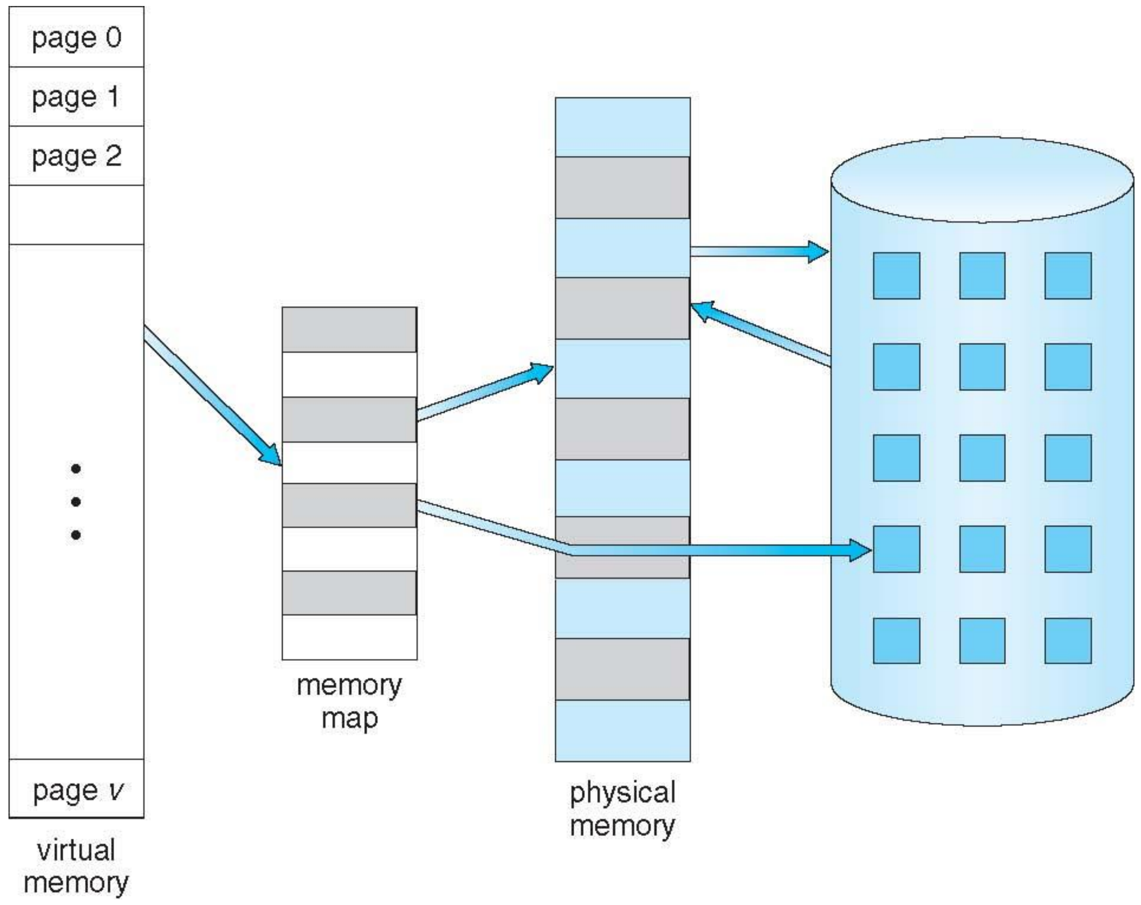
- Usually start at address 0, contiguous addresses until end of space
- Meanwhile, physical memory organized in page frames
- MMU must map logical to physical

Virtual memory can be implemented via:

- Demand paging
- Demand segmentation

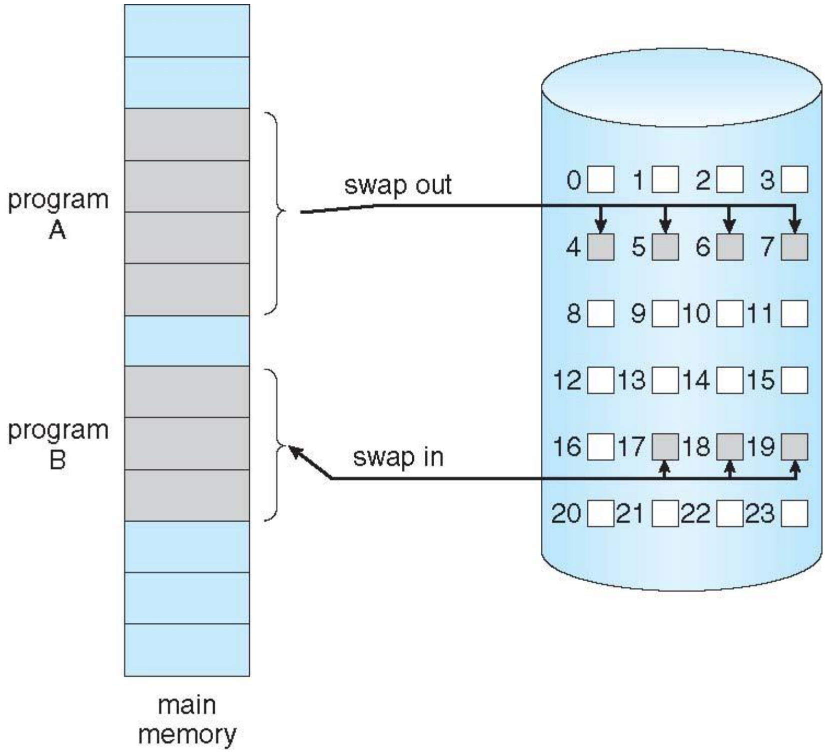


Virtual Memory That is Larger Than Physical Memory



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
  - More users
- Page is needed ⇒ reference to it
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - **pager** is concerned with the individual pages of a process

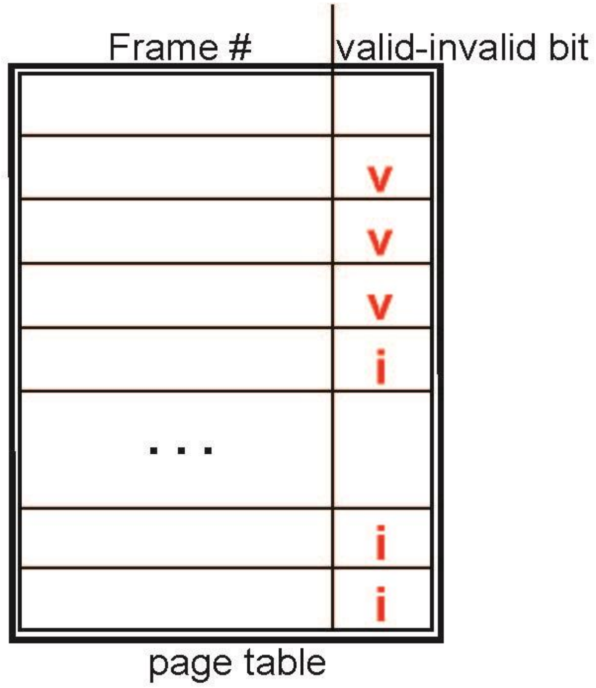


Basic Concepts

- If pages needed are already **memory resident**
  - Execute the pages
- Otherwise, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those pages into memory
- 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

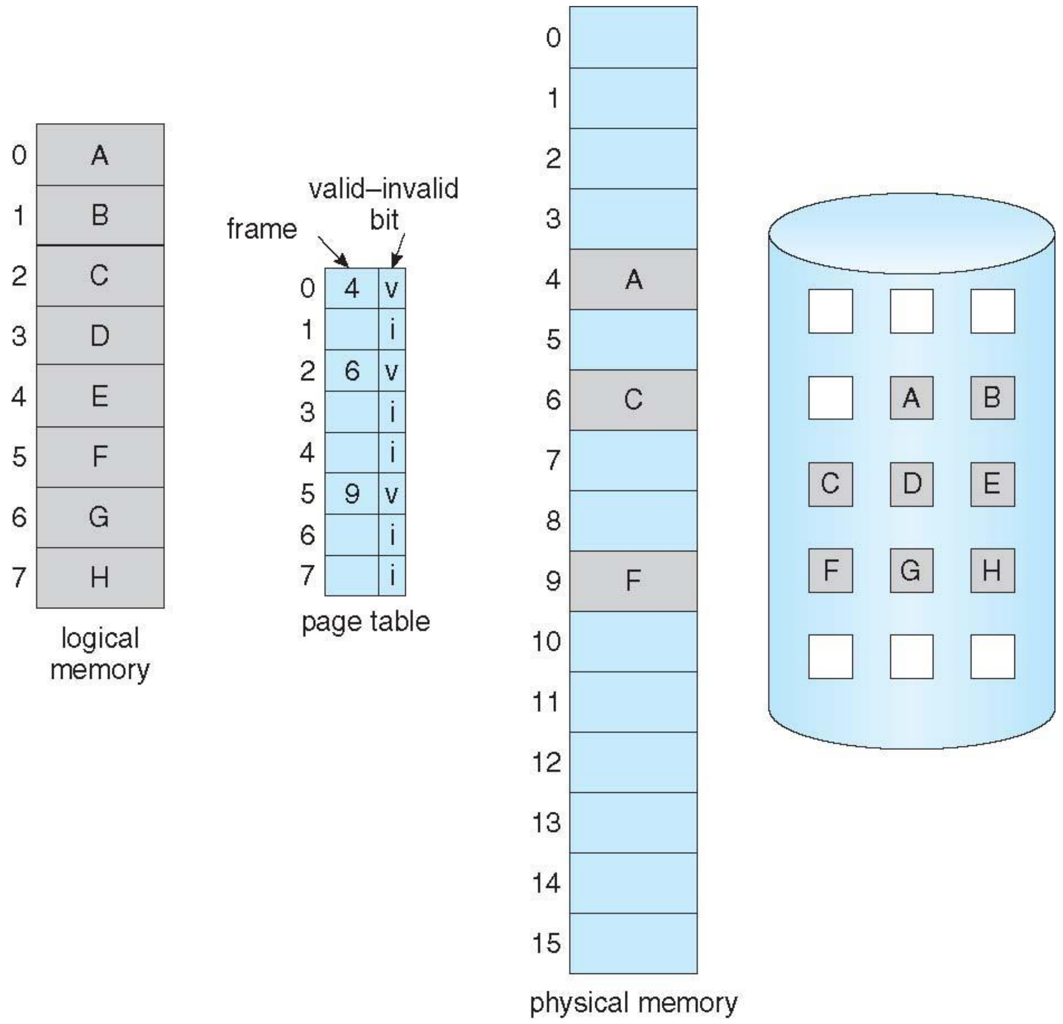
Valid-Invalid Bit

- 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

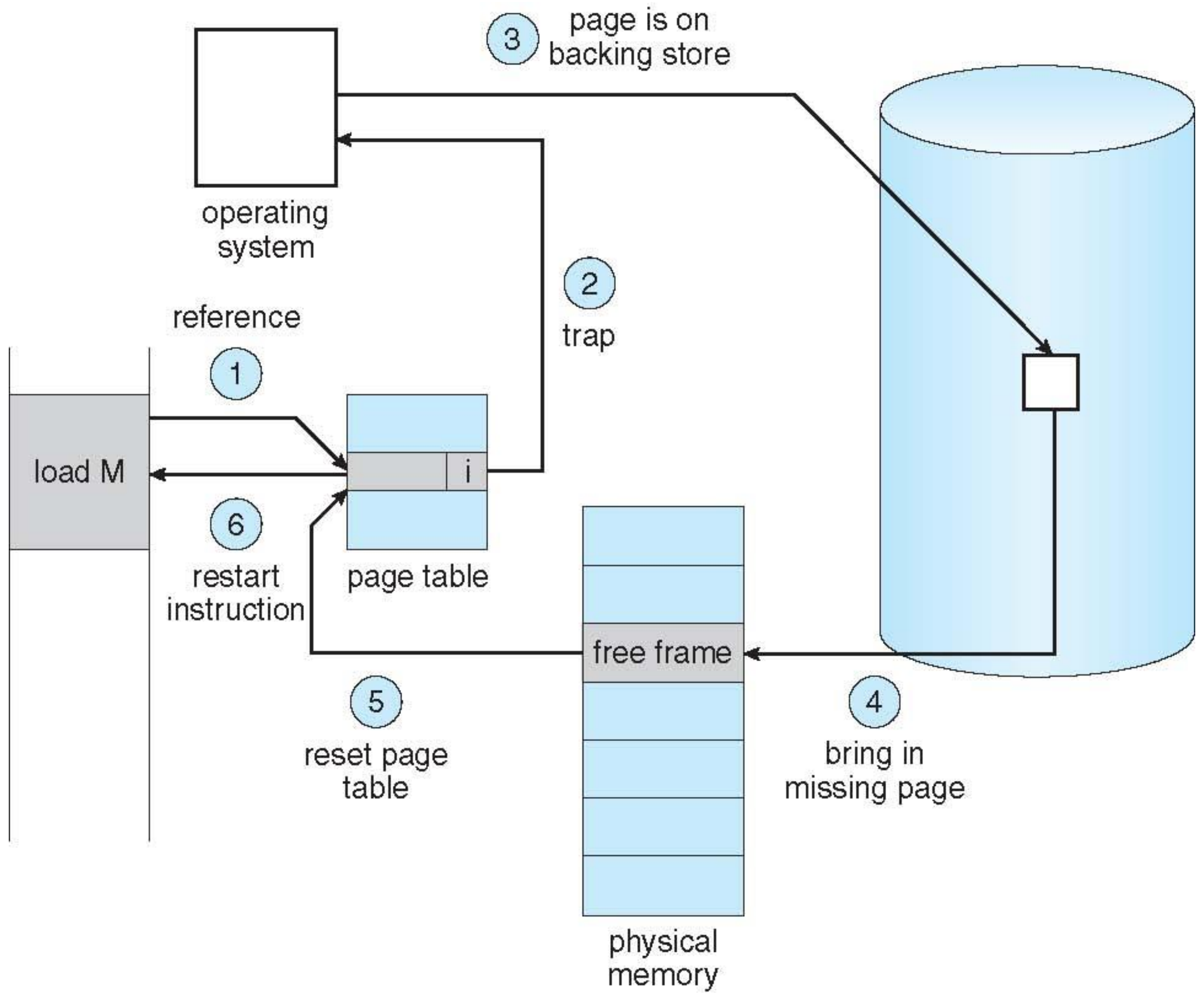


Page Fault

- If Access to a page marked invalid causes a page fault
- Operating system looks at another table to decide:
  - Invalid reference $\Rightarrow$ abort
  - Just not in memory

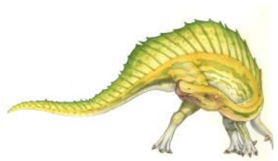
1. Find free frame
2. Swap page into frame via scheduled disk operation
3. Reset tables to indicate page now in memory
   Set validation bit = **v**
4. Restart the instruction that caused the page fault

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 (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 $0 \le p \le 1$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Effective Access Time (EAT)

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

Demand Paging Example

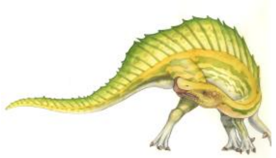
- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

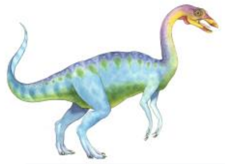
  = 200 – p x 200 + p x 8,000,000

  = 200 + p x 7,999,800
- If one access out of 1,000 causes a page fault, then

  EAT = 8200 nanoseconds.

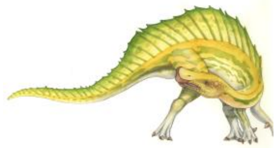
  This is a slowdown by a factor of 40!!
- If want performance degradation < 10 percent
  - 220 > 200 + 7,999,800 x p
    20 > 7,999,800 x p
  - p < .0000025
  - < one page fault in every 400,000 memory accesses





What Happens if There is no Free Frame?

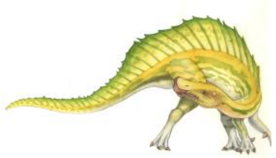
- we assumed that each page faults at most once, when it is first referenced
- To increase our degree of multiprogramming, we are over-allocating memory
- Page fault $\rightarrow$ There are no free frames on the free-frame list
  - Algorithm – terminate? swap out? replace the page?
- Page replacement – find some page in memory, but not really in use, page it out
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times



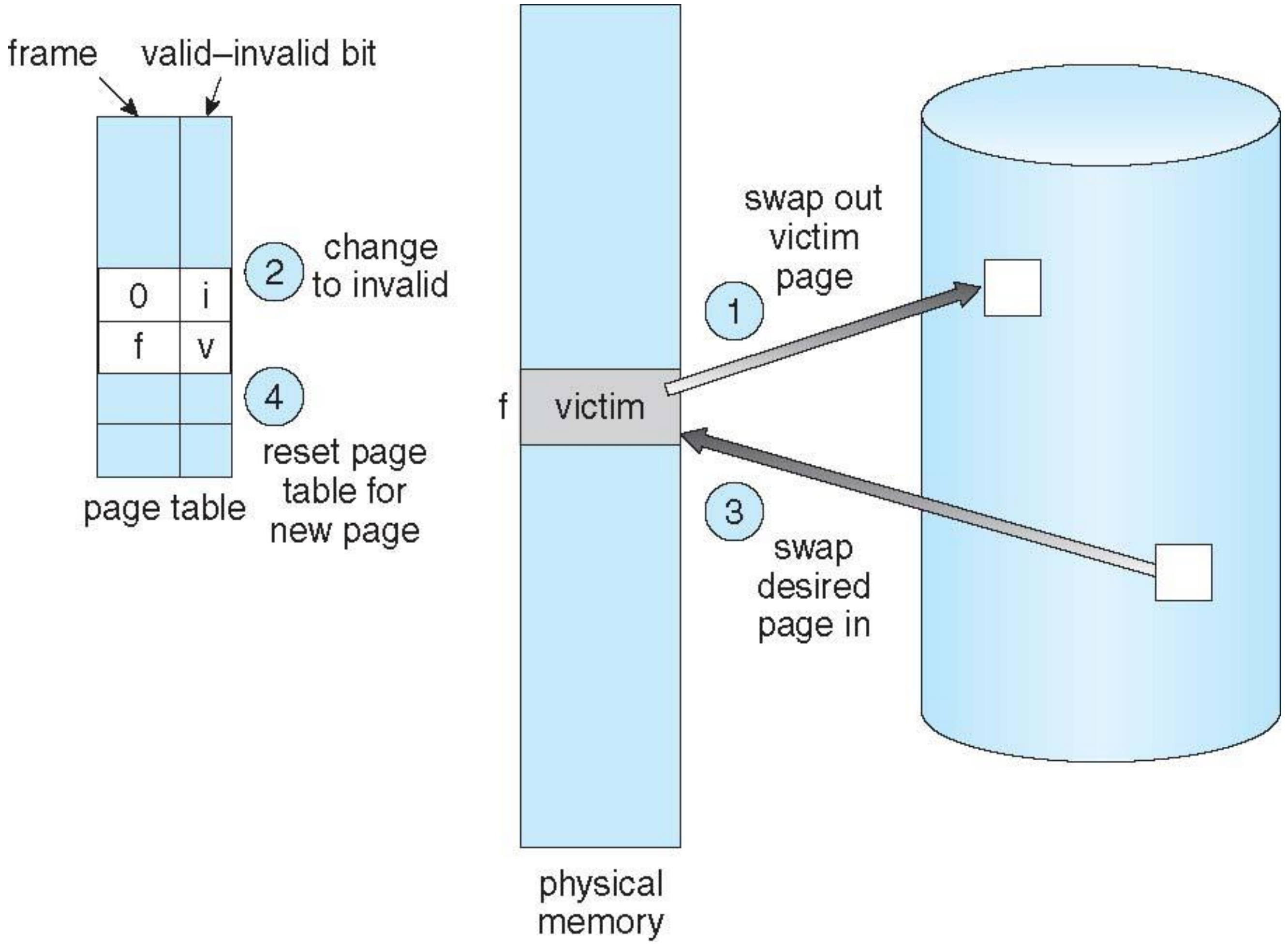
Basic Page Replacement

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

Max 2 page transfers for page fault – increasing EAT



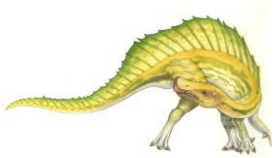
Page Replacement



Page Replacement Algorithms

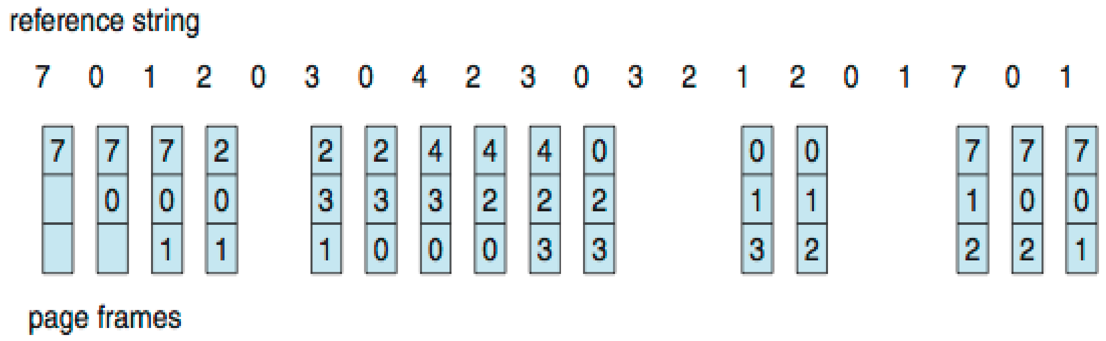
- **Page-replacement algorithm**
  - Want lowest page-fault rate on both first access and re-access
- Evaluate algorithm by running it on a particular string of memory references (reference string) and computing the number of page faults on that string
  - String is just page numbers, not full addresses
  - 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



First-In-First-Out (FIFO) Algorithm

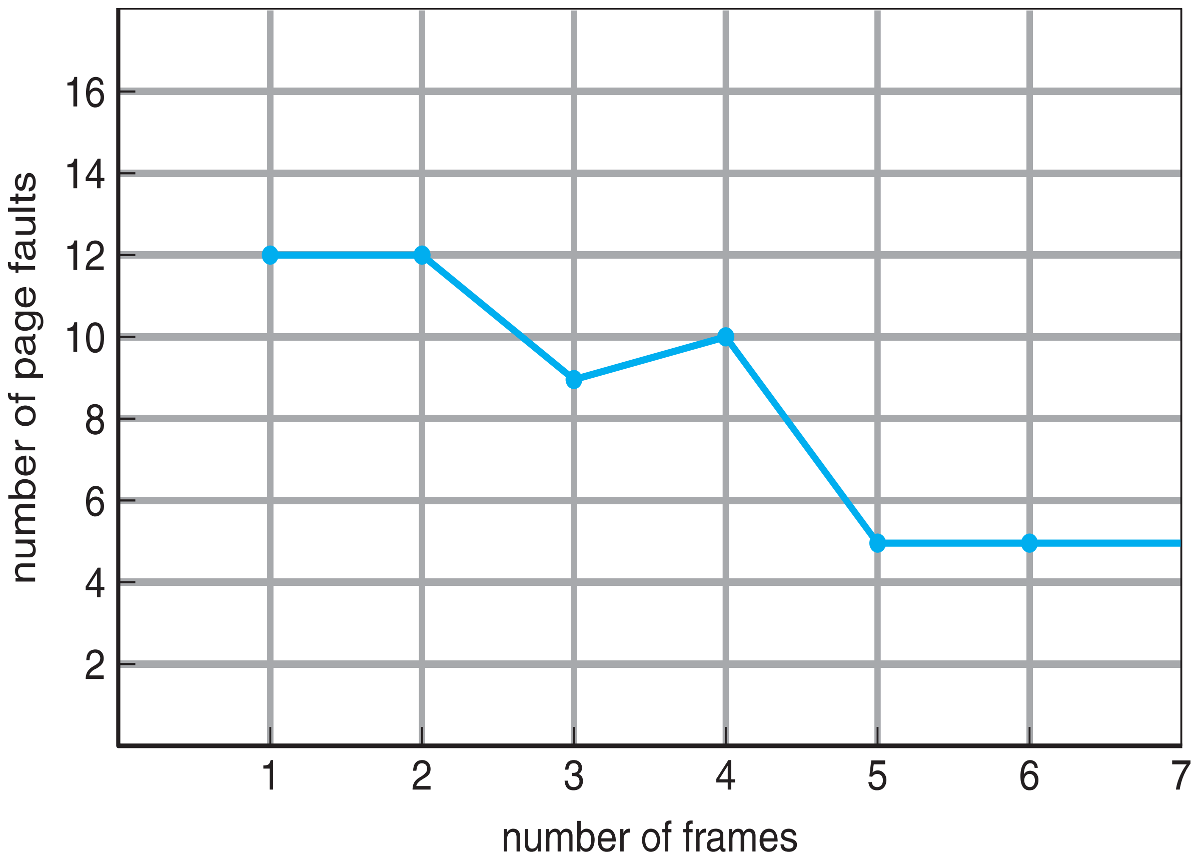
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

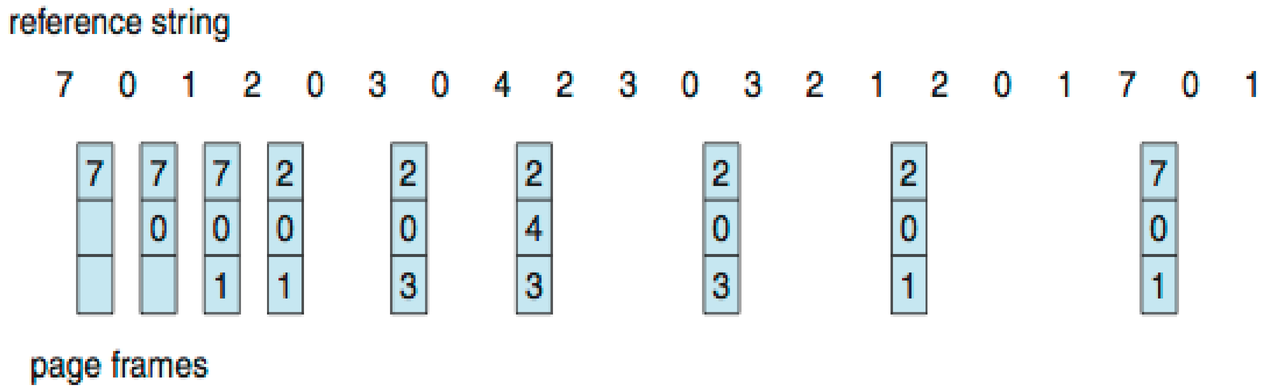
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - For four frames: 10 faults
  - For three frames: 9 faults
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**

FIFO Illustrating Belady’s Anomaly



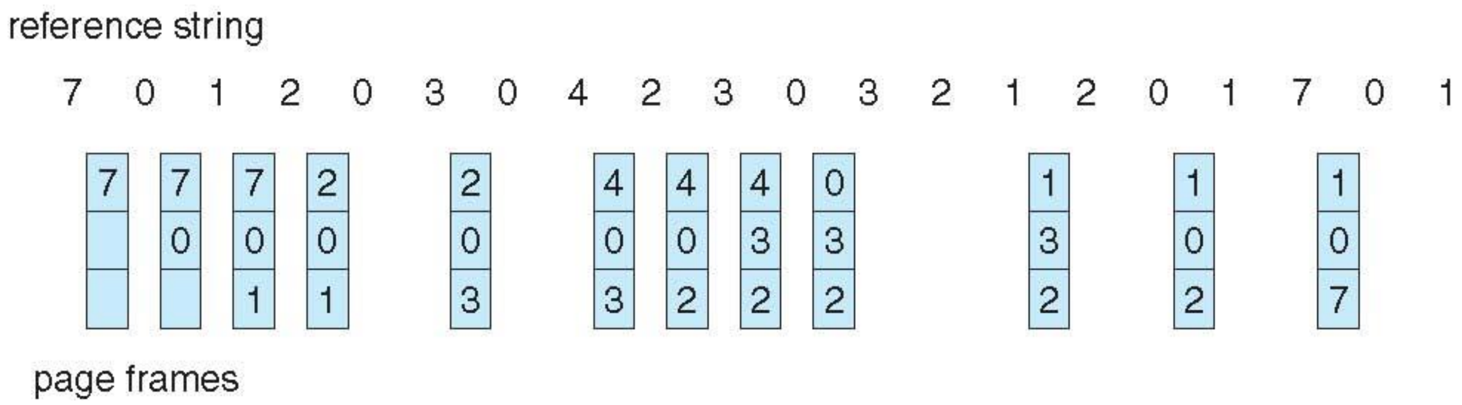
Optimal Algorithm

- 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



Least Recently Used (LRU) Algorithm

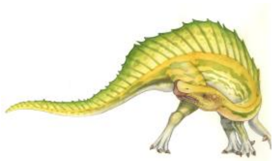
- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page



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

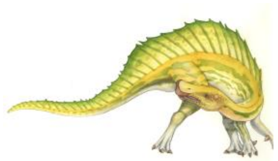
LRU Algorithm (Cont.)

- Clock Counter implementation
  - Every page entry has a clock 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 doubly linked list form:
  - Page referenced:
    - move it to the top
    - Multiple pointers to be changed
  - But each update more expensive
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly



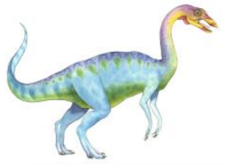
Counting Algorithms

- Keep a counter of the number of references that have been made to each page
  - Not common
- **Lease 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



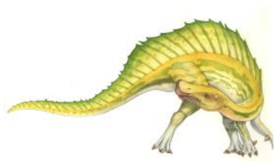
Page-Buffering Algorithms

- 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



Allocation of Frames

- Each process can be given ***minimum*** number of frames
- ***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

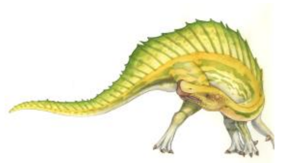
— $s_i =$ size of process p_i

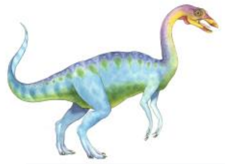
— $S = \sum s_i$

— $m =$ total number of frames

— $a_i =$ allocation for $p_i = \frac{s_i}{S} \times m$

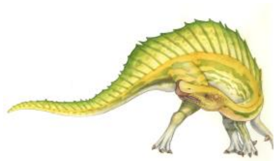
$$m = 64$$
$$s_1 = 10$$
$$s_2 = 127$$
$$a_1 = \frac{10}{137} ´ 62 » 4$$
$$a_2 = \frac{127}{137} ´ 62 » 57$$





Priority Allocation

- Use a proportional allocation scheme using priorities rather than size
- If process $\boldsymbol{P_i}$ 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

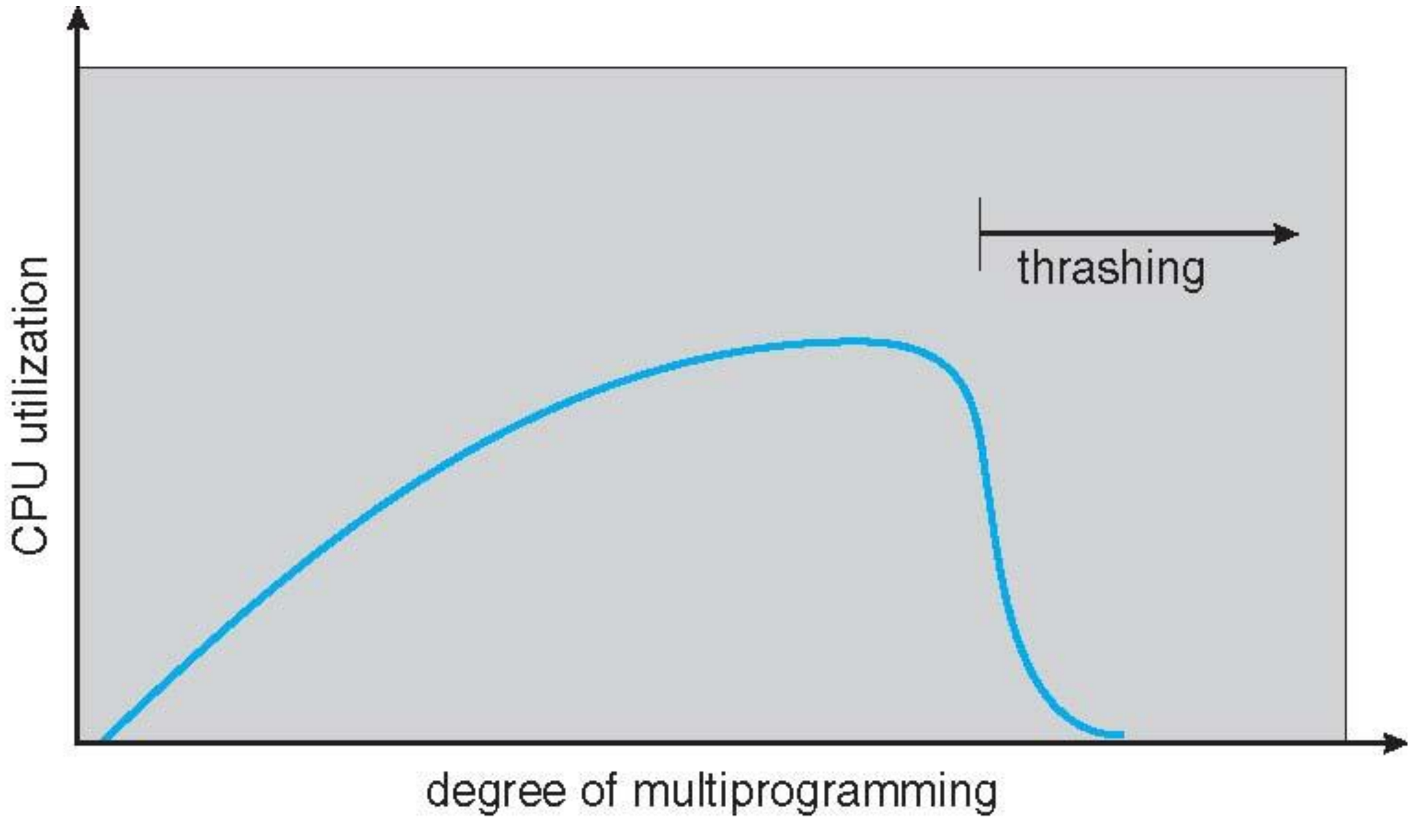
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

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

Thrashing (Cont.)



End of Chapter 8