Operating Systems - Virtual Memory

Mridul Sankar Barik

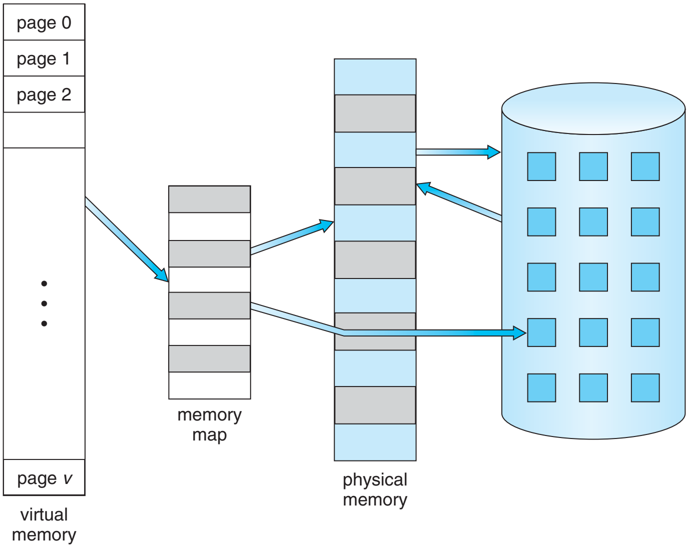
Jadavpur University

2024

Background I

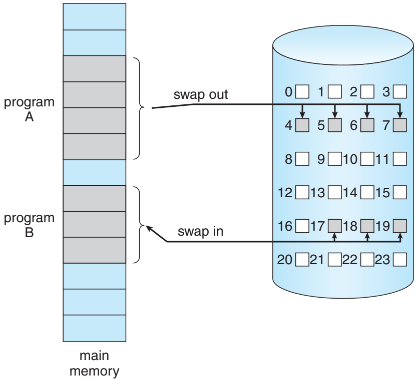
- Code needs to be in memory to execute, but entire program is rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at the same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each program takes less memory while running $\rightarrow$ more programs run at the same time
  - Less I/O needed to load or swap programs into memory $\rightarrow$ each user program runs faster

Virtual Memory That is Larger Than Physical Memory



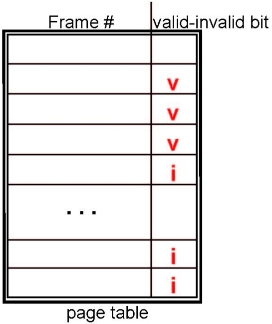
Demand Paging

- Bring a page into memory only when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users
- Similar to paging system with swapping
- Page is needed $\Rightarrow$ reference to it
  - invalid reference $\Rightarrow$ abort
  - not-in-memory $\Rightarrow$ bring to memory



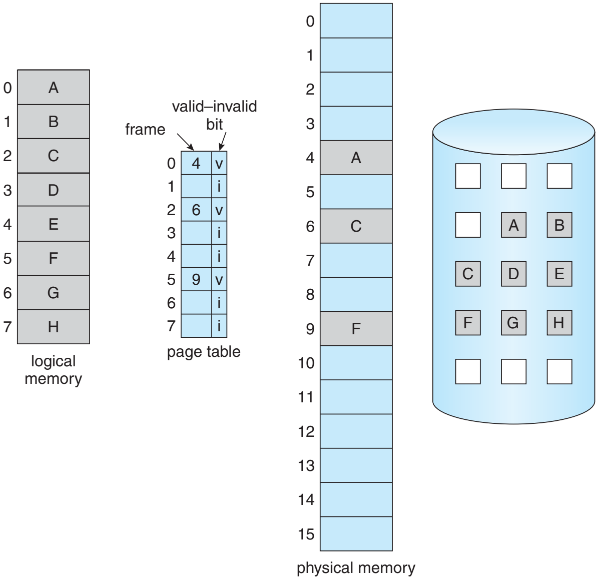
Valid-Invalid Bit

- 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

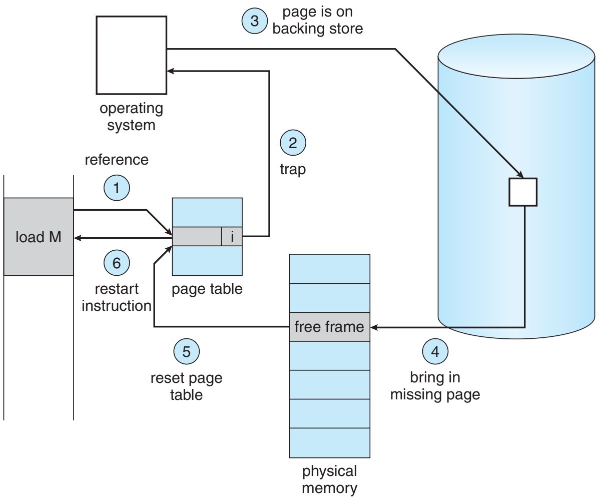
Page Table When Some Pages Are Not in Main Memory



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 $\Rightarrow$ 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
  5. Set validation bit = v
  6. Restart the instruction that caused the page fault

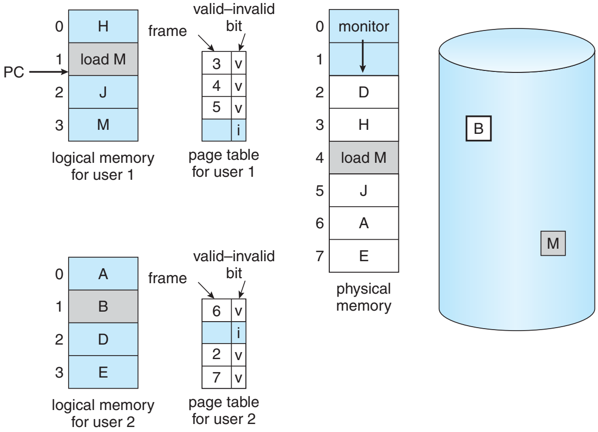
Steps in Handling a Page Fault



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

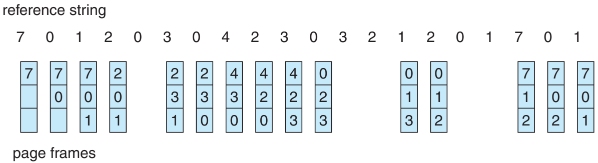
1. Find the location of the desired page on disk
2. Find a free frame:
    - If there is a free frame, use it
    - If there is no free frame, use a page replacement algorithm to select a victim frame
        - Write victim frame to disk if dirty
3. Bring the desired page into the (newly) free frame;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

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

First-In-First-Out (FIFO) Algorithm

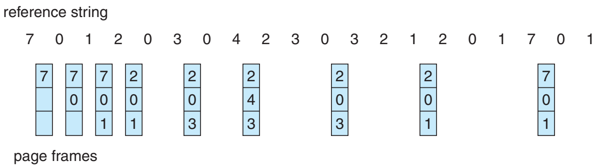
- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



- 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! $\Rightarrow$ Belady's Anomaly
- How to track ages of pages?
  - Just use a FIFO queue

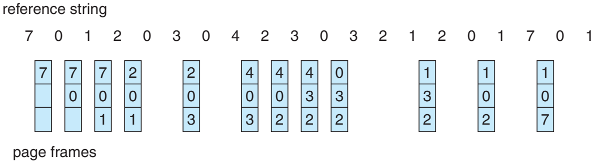
Optimal Algorithm

- 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

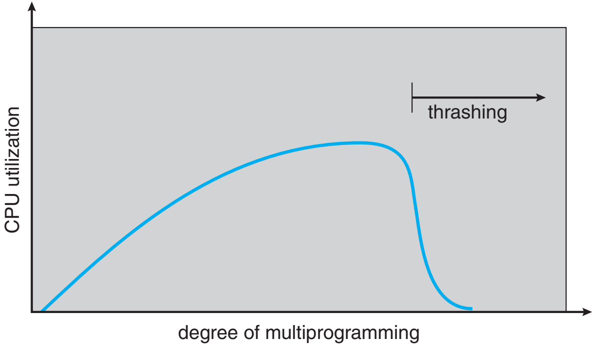


- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used

Thrashing I

- 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 thinks 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 II



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?
  $\sum$ size of locality $>$ total memory size
  - Limit effects by using local or priority page replacement