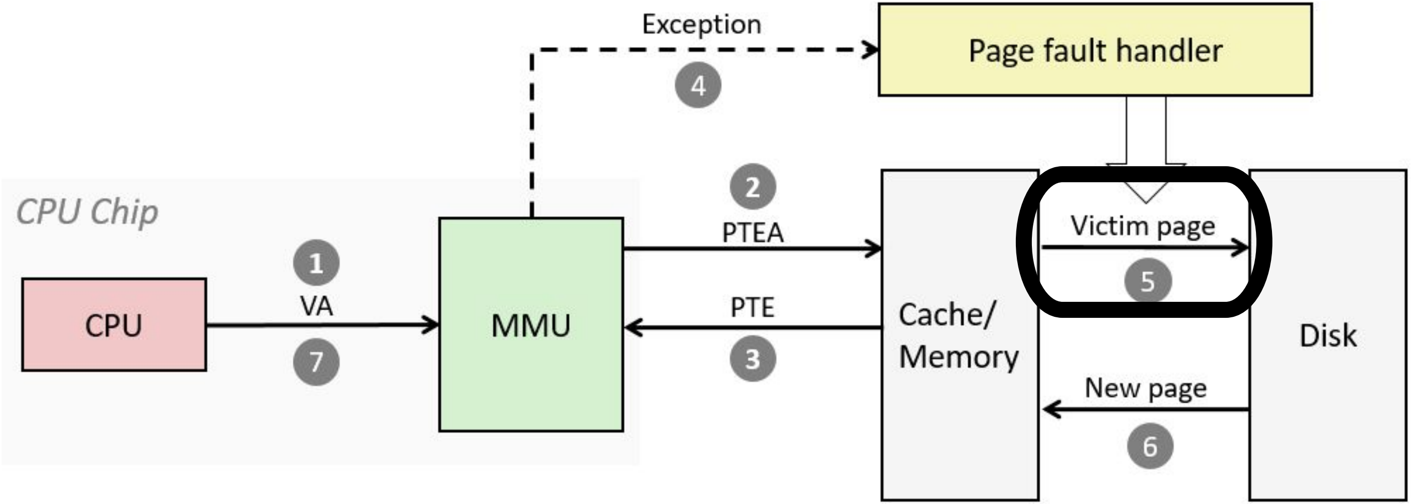
12. Page Replacements

CS 4352 Operating Systems

Adopted from slides of Drs. Silberschatz, Huang, & Bryant

Review

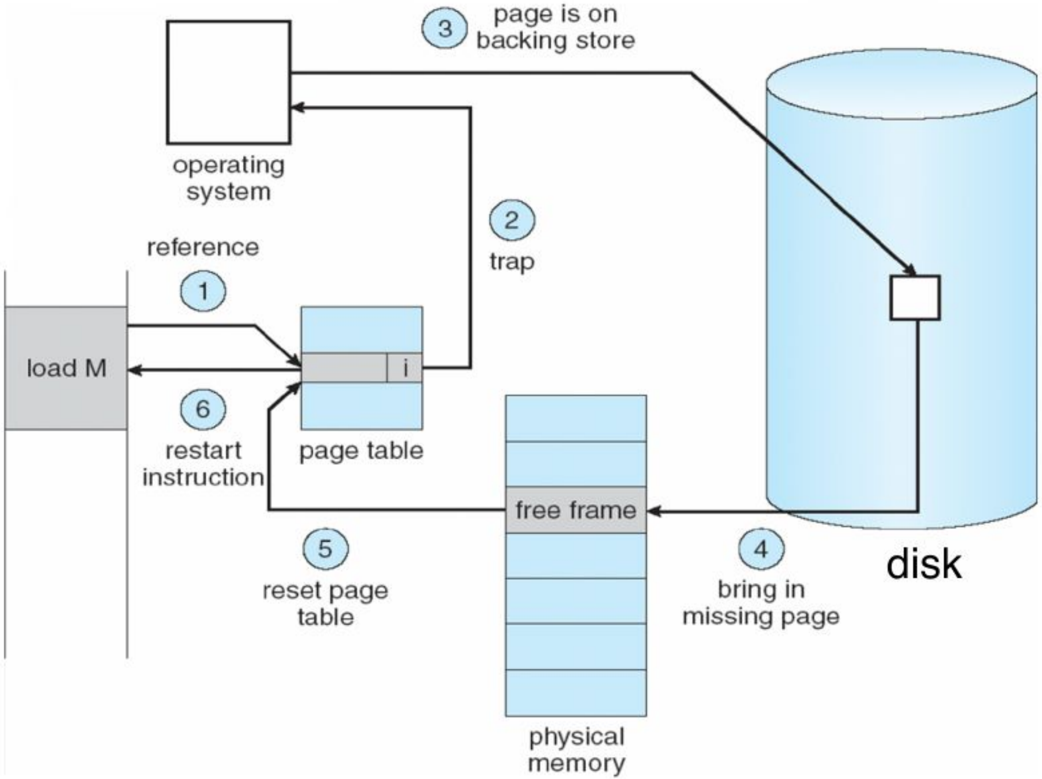


Exception
Page fault handler
4
2
CPU Chip
Victim page
PTEA
1
5
VA
PTE
Cache/
Memory
CPU
MMU
Disk
7
3
New page
6

Paging Review

- Paging from the OS perspective:
  - Pages are evicted to disk when memory is full
  - Pages loaded from disk when referenced again
  - References to evicted pages cause a page fault
  - OS allocates a page frame, reads page from disk
  - When I/O completes, the OS fills in PTE, marks it valid, and restarts faulting process
- Dirty vs. clean pages
  - Actually, only dirty pages (modified) need to be written to disk
  - Clean pages do not – but you need to know where on disk to read them from again

Use disk to simulate larger virtual memory

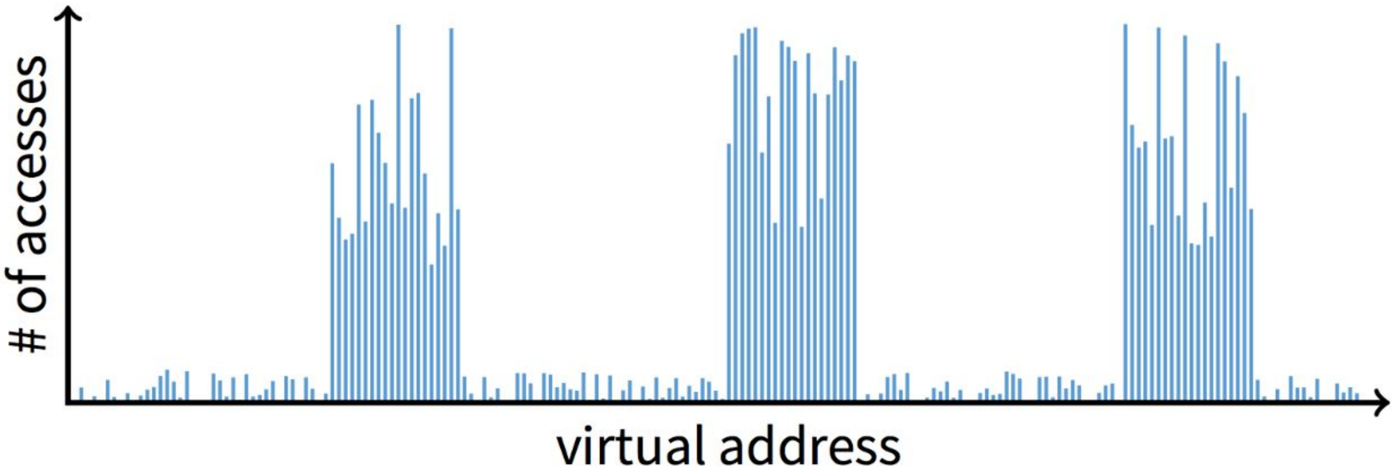


Principle of Locality

- We have mentioned this previously
    - Temporal locality
        - If a memory location is accessed once, it is likely that this memory location will be accessed again in the near future
    - Spatial locality
        - If a memory location is accessed once, it is likely that some nearby memory locations will be accessed in the near future
- Although the cost of paging is high, if it is infrequent enough it is acceptable
    - Processes usually exhibit both kinds of locality during their execution, making paging practical

80/20 Rule

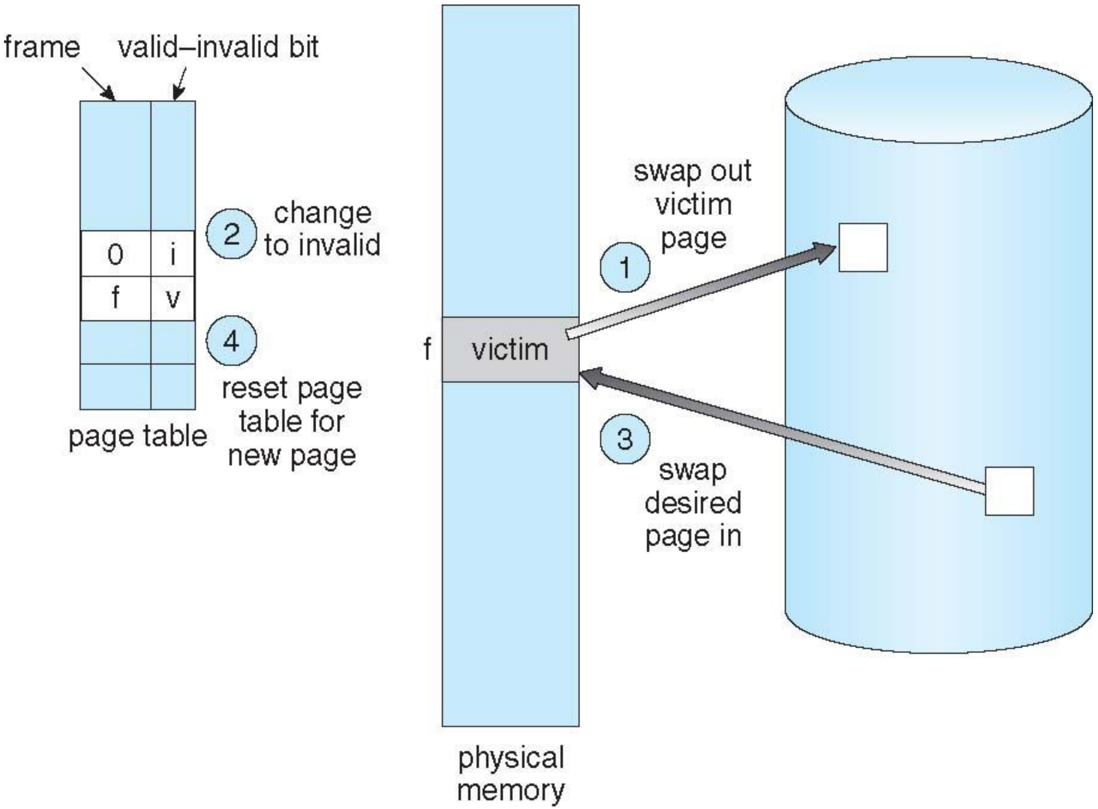
- 20% of memory gets 80% of memory accesses
  - Keep the hot 20% in memory
  - Keep the cold 80% on disk



Page Replacement

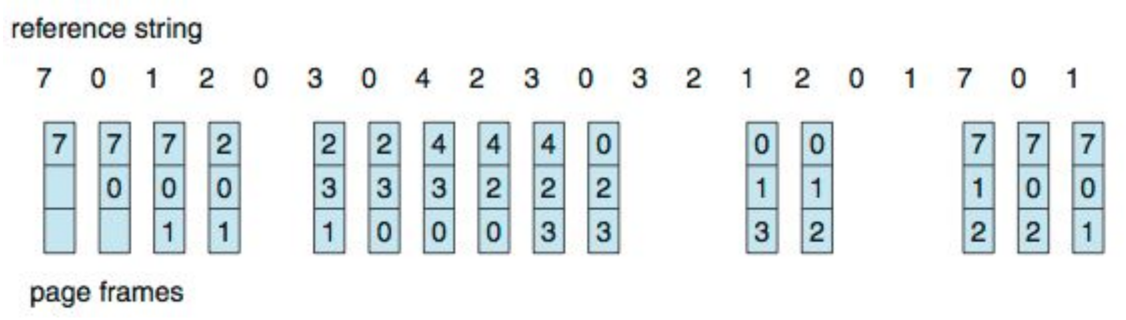
- When a page fault occurs, the OS loads the needed page from disk into a page frame of physical memory
- At some point, the process used all of the page frames it is allowed to use
    - This is likely (much) less than all of available memory
- When this happens, the OS must replace a page for each page faulted in
    - It must evict a page to free up a page frame
- The page replacement policies determines how this is done
    - Find some page in memory, but not really in use, page it out
    - E.g., use modify (dirty) bit to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement enables large virtual memory to be provided on a smaller physical memory

Page Replacement



First-In First-Out (FIFO) Algorithm

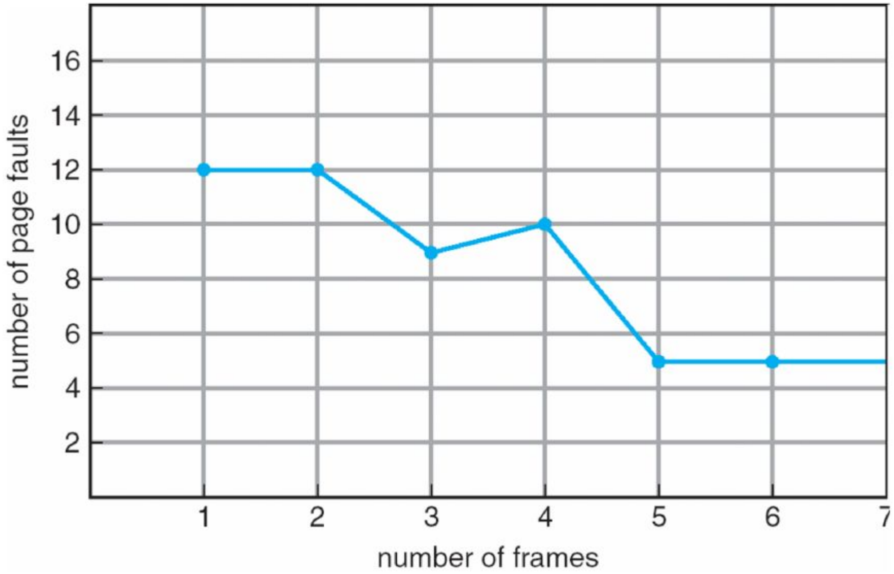
- Just use a FIFO queue and the victim is the first
- Reference string: 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1
- 3 frames (3 pages can be in memory at a time per process)



- 15 page faults

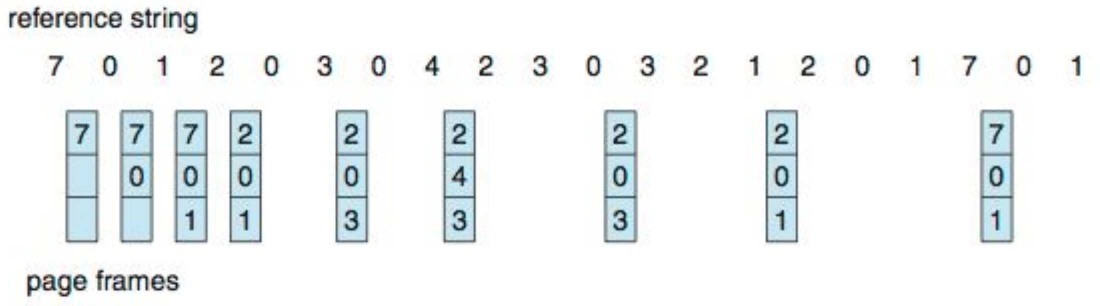
FIFO has Belady’s Anomaly

- Adding more frames can cause more page faults!
  - Example—reference string 1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5
    - 3 page frames: 9 page faults
    - 4 page frames: 10 page faults



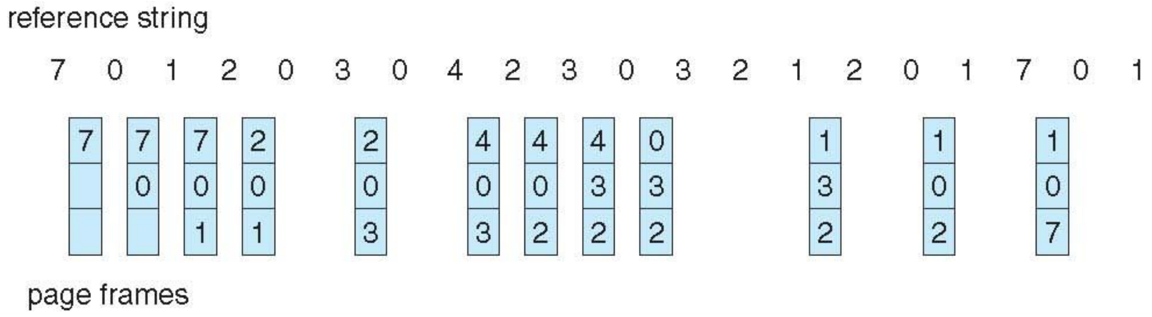
Optimal Algorithm

- 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
- But how to implement?

Possible Implementations

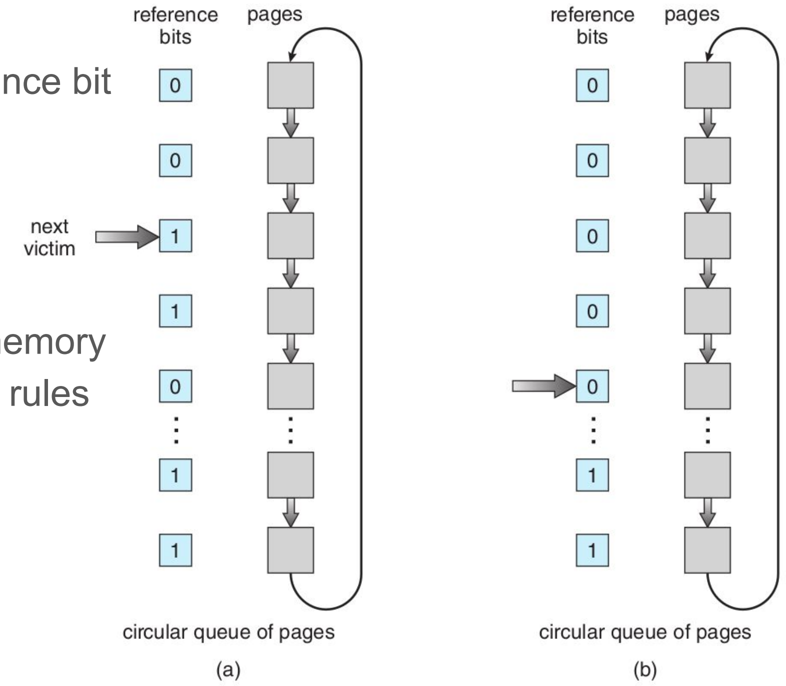
- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
  - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a doubly linked form
  - Page referenced:
    - move it to the top
    - requires 6 pointers to be changed
  - No search for replacement
  - But each update more expensive

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

LRU Approximation Algorithms (Cont.)

- 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



Other Replacement Algorithms

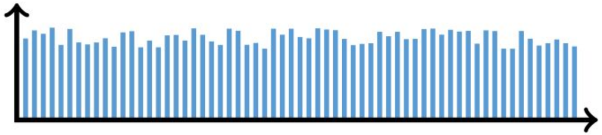
- Random eviction
  - Simple to implement
  - Not bad actually
- LFU (least frequently used) eviction
  - Instead of just a reference bit, count # times each page accessed
  - Least frequently accessed may not be very useful
  - Decay usage counts over time
- MFU (most frequently used) algorithm
  - Because page with the smallest count was probably just brought in and has yet to be used
- Neither LFU nor MFU used very commonly

Fixed v.s. Variable Space

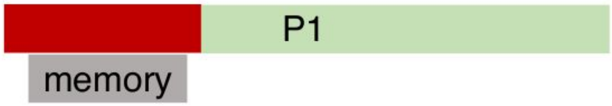
- How to determine how much memory to give to each process?
- Fixed space algorithms
  - Each process is given a limit of pages it can use
  - When it reaches the limit, it replaces from its own pages
  - Local replacement
    - Some processes may do well while others suffer
- Variable space algorithms
  - Process' set of pages grows and shrinks dynamically
  - Global replacement
    - One process can ruin it for the rest

Thrashing

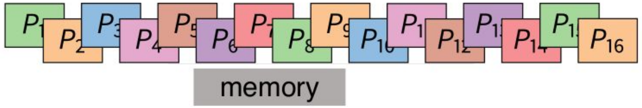
- When OS spent most of the time in paging data back and forth from disk
  - Little time spent doing useful work (making progress)
  - In this situation, the system is overcommitted
    - No idea which pages should be in memory to reduce faults
- Reasons for Thrashing
  - Access pattern has no temporal locality



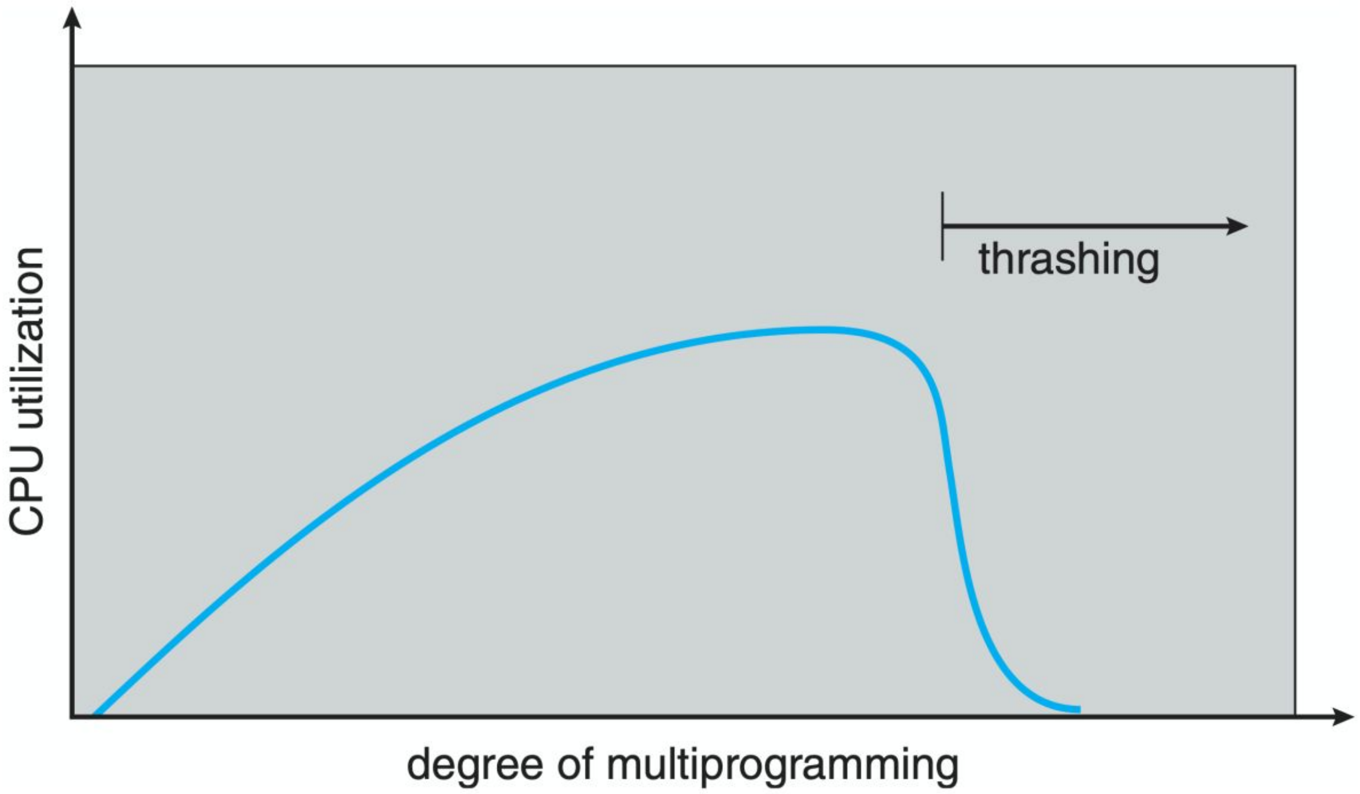
  - Hot memory does not fit in physical memory



  - Each process fits individually, but too many for system



Thrashing & Multiprogramming

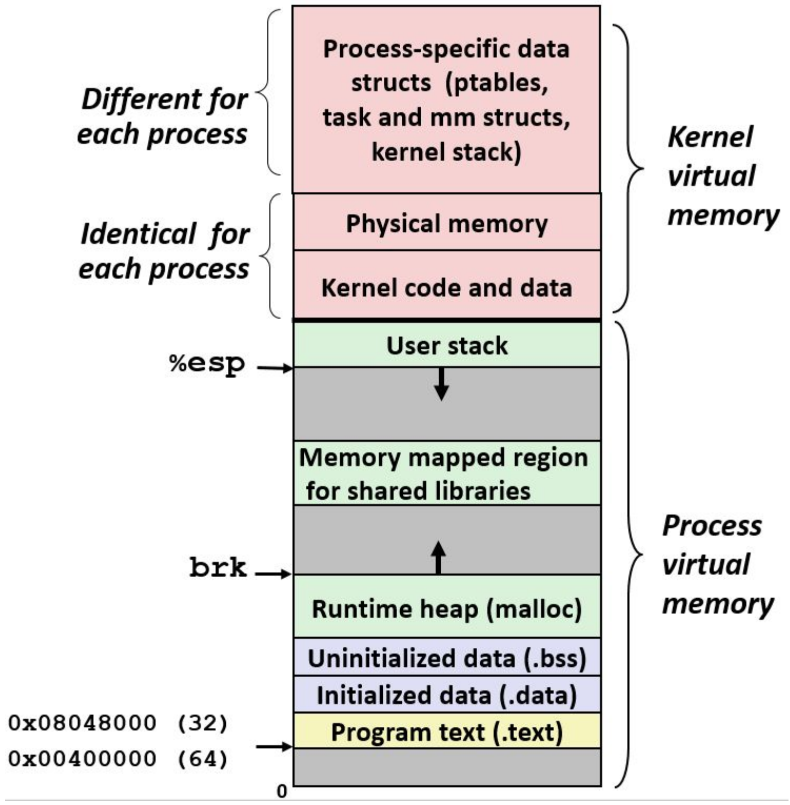


Dealing with Thrashing

- Only run processes if memory requirements can be satisfied
- Write out all pages of a process
- OOM
- Buy more memory...

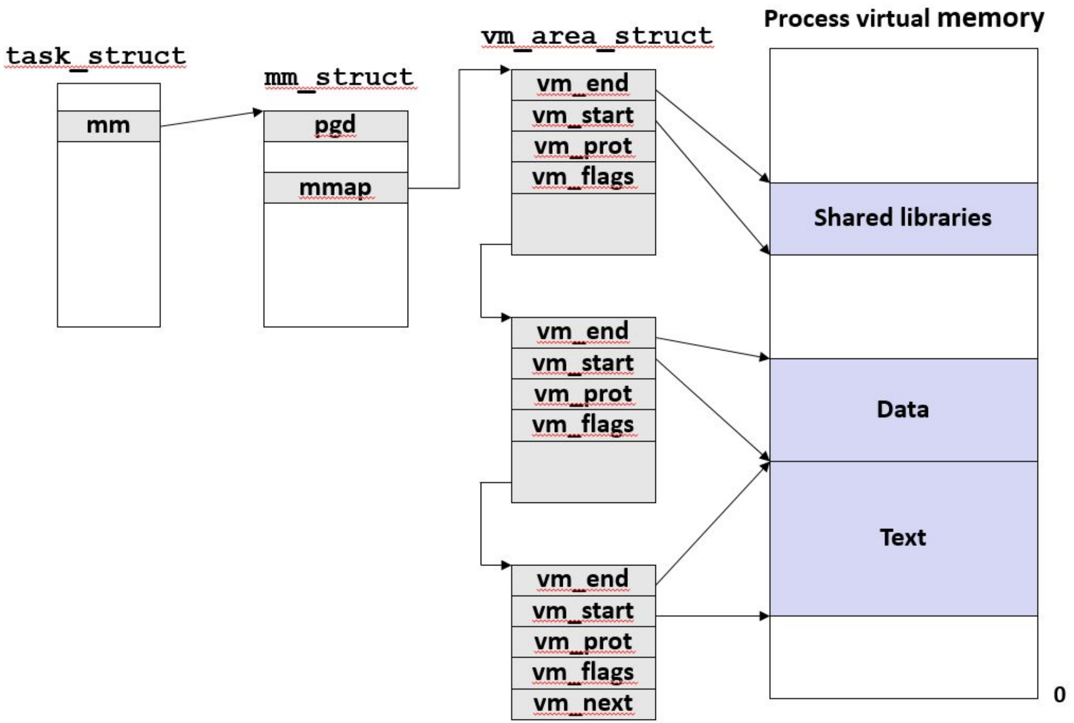
Linux VM revisited

- We're familiar with the address space diagram on the right
- But how does the Linux kernel view this?



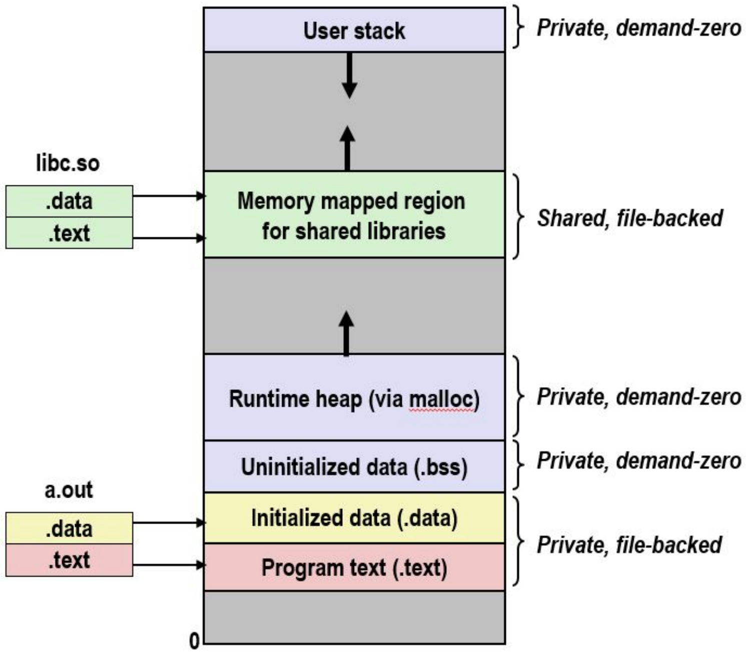
Linux Representation of Memory

- The Linux kernel organizes VM as a collection of “areas”
  - vm_start and vm_end: point to the beginning and end of the area
  - vm_prot: describes read/write permissions
  - vm_flags: tells whether pages in this area are private
  - vm_next: points to next area vm_area_struct in list



exec() in More Detail

- Now we can see how exec works in detail!
- “Blowing away” the address space of the current process means freeing the vm_area_structs and page tables for old areas
- New vm_area_structs and page tables are created for new areas
  - Programs and initialized data are backed by object files
  - .bss and stack backed by anonymous files
- Set the program counter (PC) to the entry point in the .text section



Homework

Read Chapters 11 & 12

Next Lecture

We start looking at storage management