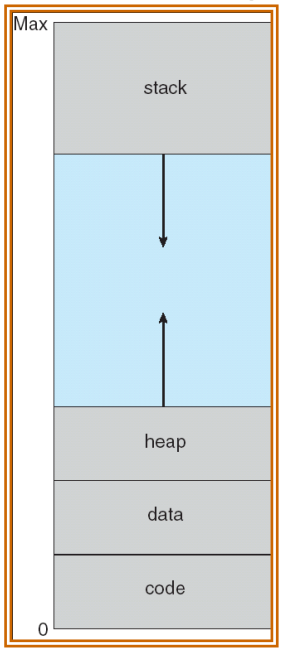
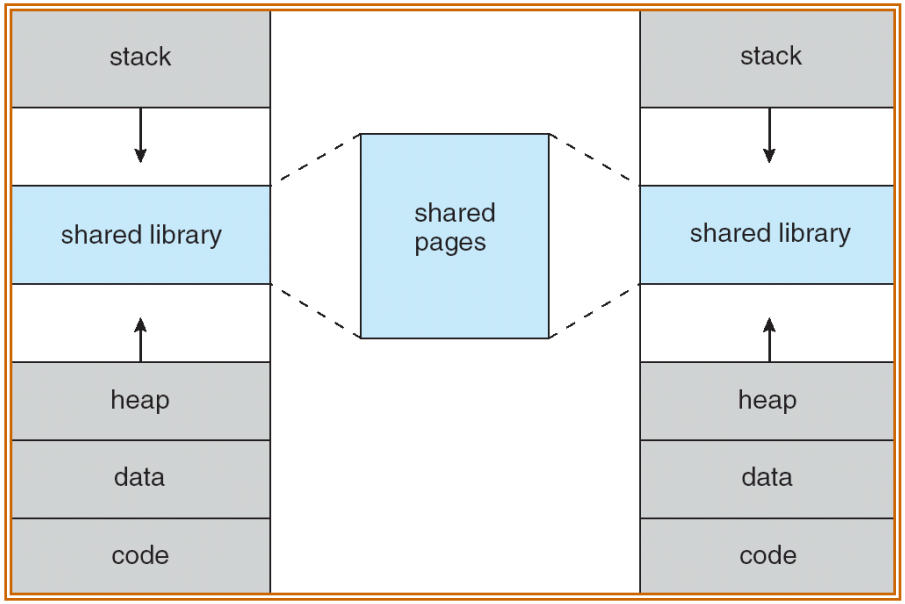
Virtual Memory

Background

- Virtual memory allows the execution of processes that are not completely in main memory
  - Logical address space can be much larger than physical memory – no need to worry about available physical memory
  - More programs could run at the same time
  - Programs would run faster (less need for swapping)
- Certain parts of a program are rarely needed, even the parts that are often needed are not all needed at the same time
- Virtual memory allows files and memory to be shared by two or more processes
  - System libraries, shared memory communication, fork() – process creation

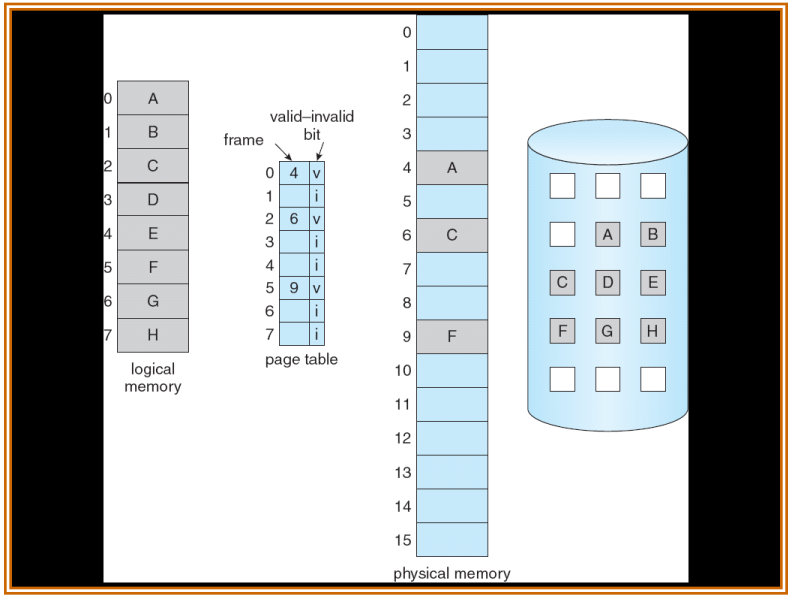


Sparse address spaces

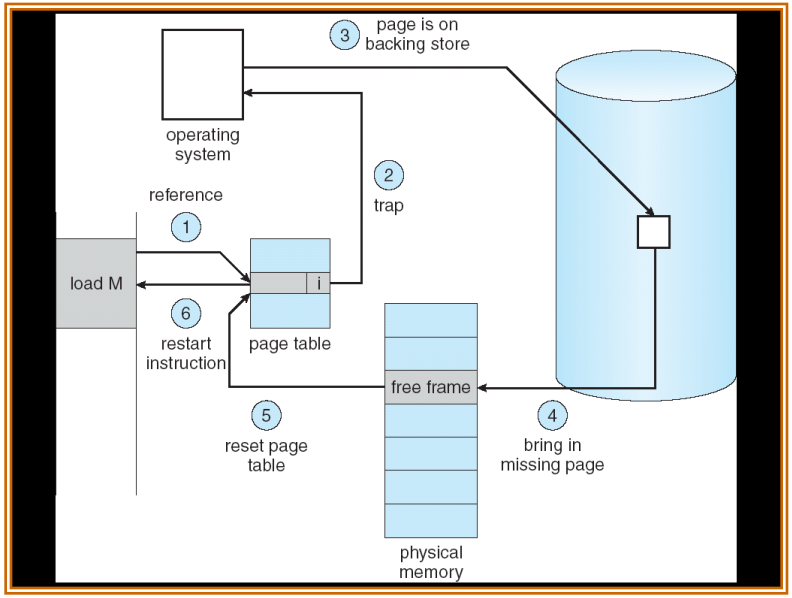


Demand Paging (1)

- Load page into main memory only when they are needed
  - Similar to a paging system with swapping
- Lazy swapper (pager in fact!)
  - Guess which pages to load
  - Cause a page-fault trap when accessing page that is not memory resident



Demand Paging (2)



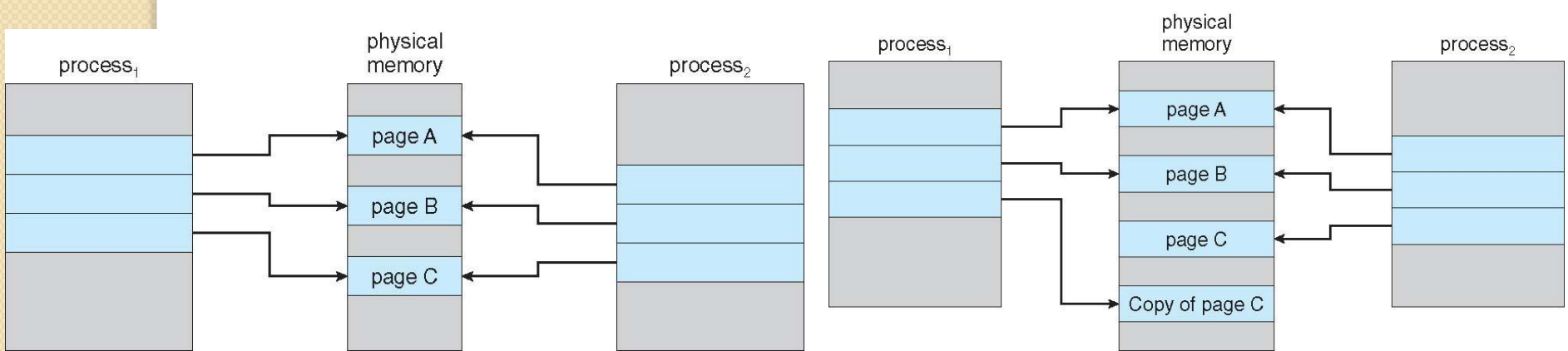
- Page fault handling
- Pure demand paging
- Locality of reference
  - Multiple page faults caused by a single instruction are unlikely
- Hardware requirements
  - Page table with invalid bits
  - Secondary memory – swap space
  - Ability to restart a process at exactly the same place
    - This can be tricky! – it is not easy to introduce demand paging in a system

Demand Paging (3)

- Demand paging seriously affects the performance of a computer
- Page Fault Rate $0 \leq p \leq 1.0$
  - If p = 0 no page faults
  - If p = 1, every reference is a fault
  - Effective Access Time (EAT) = (1 – p) x memory access + p (page fault overhead + swap page out + swap page in + restart overhead)
  - Memory access time = 200 nanoseconds
  - 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
  - If one access out of 1,000 causes a page fault, then EAT = 8.2 microseconds
  - This is a slowdown by a factor of 40!!
  - If we want a slowdown < 10%, then we shouldn't have more than 1 in 399,990 memory access cause a page fault!!
- Use of the swap space can improve things
  - For binary files (do not really change) the file system itself can be a backing store

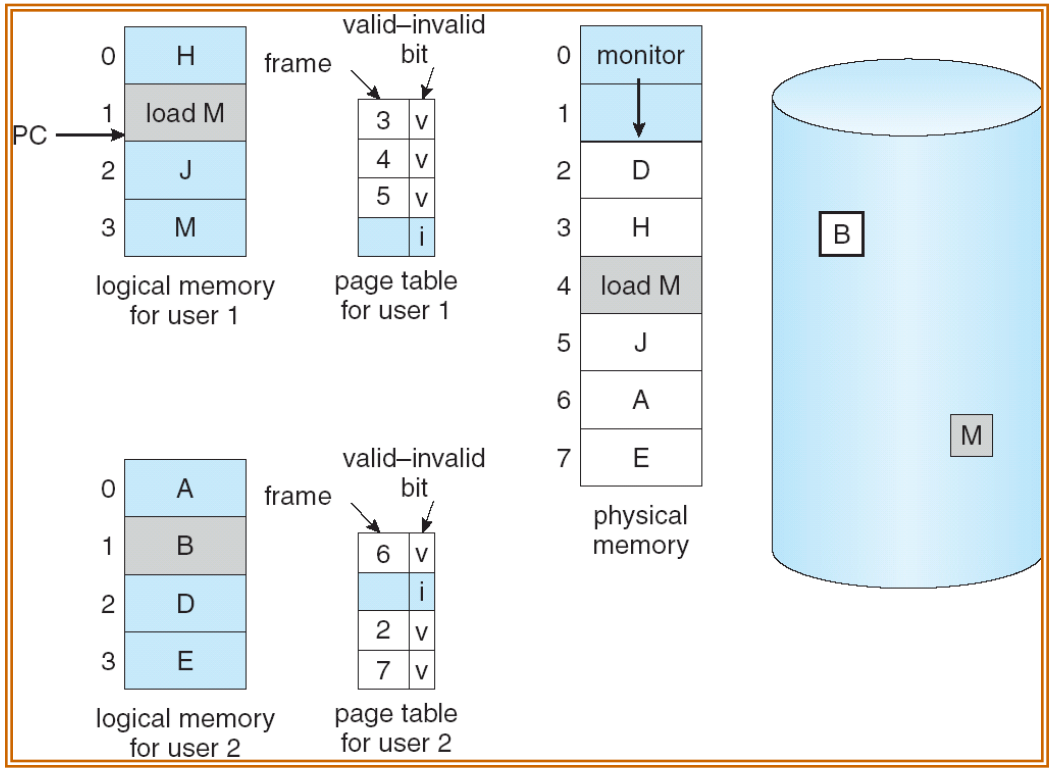
Copy-on-Write

- Memory management during fork()
  - Pool of free pages (stack or heap allocation)
  - Zero-fill-on-demand
- vfork() – virtual memory fork
  - Suspend parent process
  - Changes to memory by child process are visible to parent process



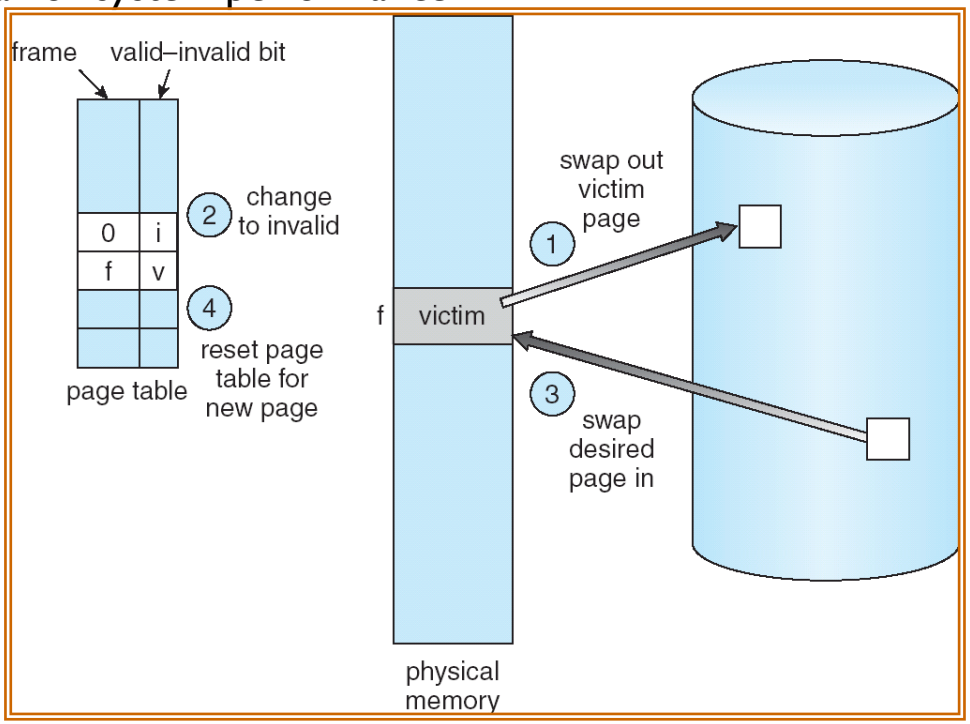
Page Replacement (1)

- Over-allocate memory by increasing the degree of multiprogramming
- Memory allocation for I/O buffers
  - Fixed percentage or let them compete



Page Replacement (2)

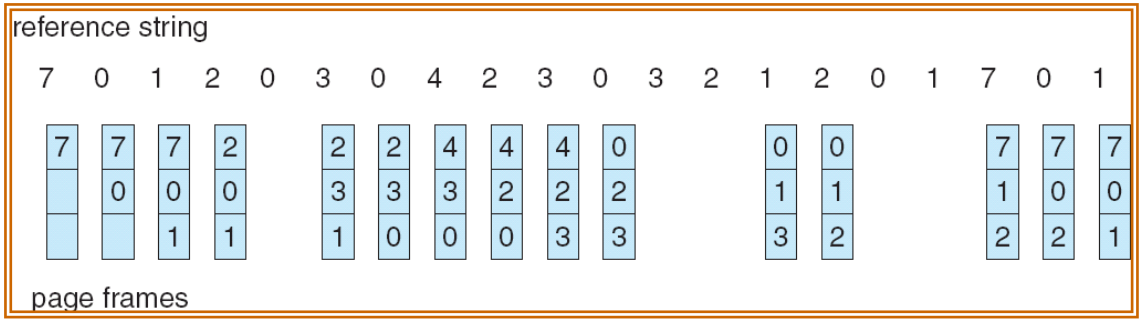
- Frame allocation algorithm
- Page replacement algorithm – minimise page fault rate
  - Evaluate using a reference string
  - Only consider page no. & ignore multiple consecutive references to the same page
  - Increasing the number of frames reduces the page fault rate (exponentially)!
- Both crucial for system performance



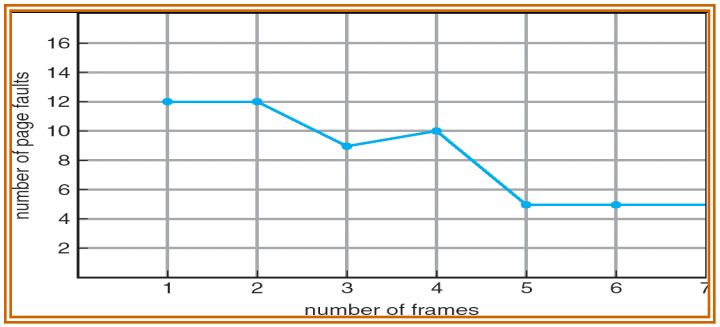
modify or dirty bit

Page Replacement (3)

- FIFO page replacement
  - FIFO queue to hold pages in memory
  - + easy to understand and program
  - - performance may be poor, depends on how pages are used

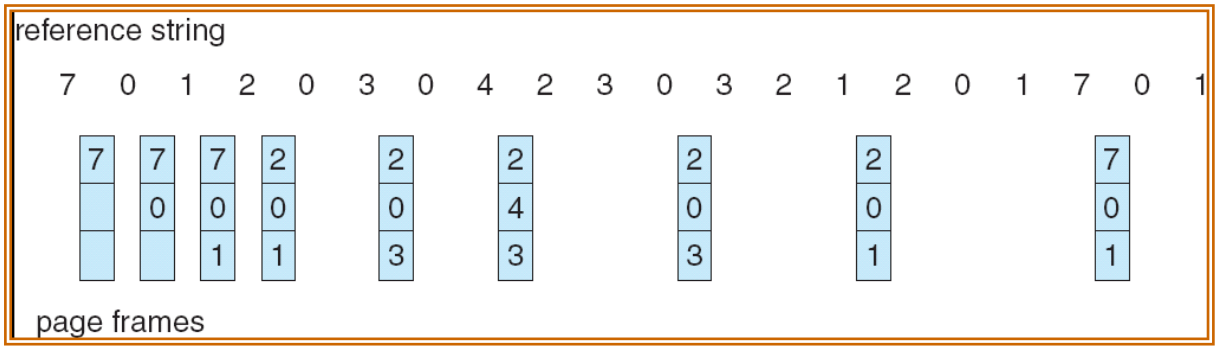


Belady's anomaly



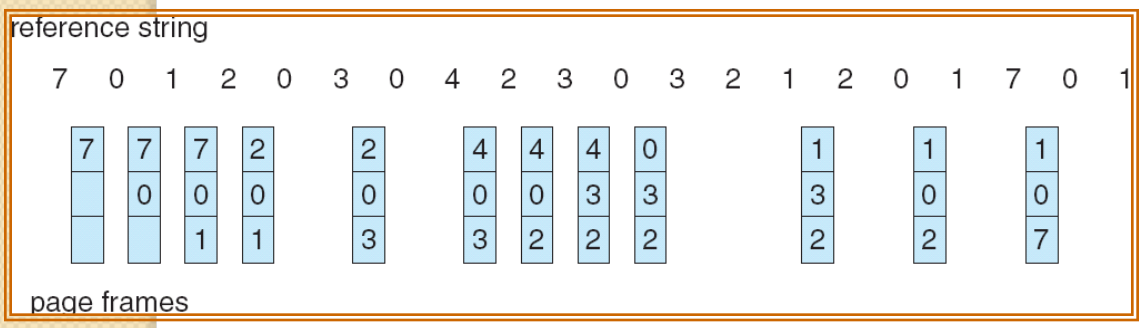
Page Replacement (4)

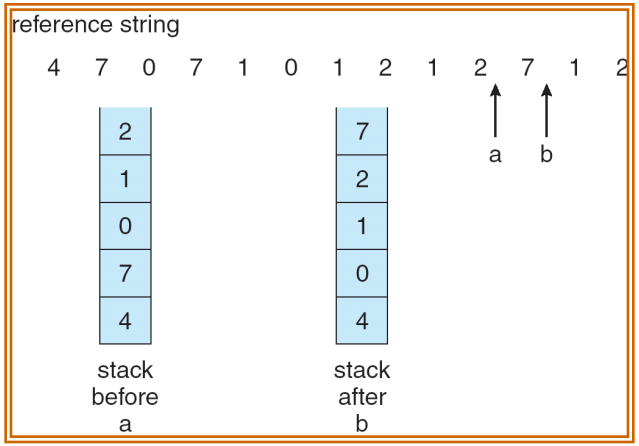
- Optimal page replacement
  - Lowest page-fault rate, do not suffer from Belady's anomaly
  - Replace the page that will not be used for the longest period
- Almost impossible to implement
  - Used to compare other algorithms



Page Replacement (5)

- Least Recently Used (LRU) page replacement
  - Use the recent past as an approximation of the future – considered good and is often used
  - Interesting observation applying LRU or the optimal on the reverse reference string gives the same number of page faults
  - May require substantial hardware assistance
    - Counters – cost: search of page table + memory access for counter update
      - Overflow, context switching
    - Stack – referenced pages put on top of the stack (doubly linked list with head and tail)
  - Stack algorithms do not suffer from Belady's anomaly
    - The set of pages in memory for N frames is always a subset of the set of pages for N+1 frames





Page Replacement (6)

- LRU-approximation page replacement
  - Limited hardware support in the form of a reference bit
    - All clear in the beginning, set at every access
    - Replace pages with clear reference bits
  - Additional reference bits algorithm
    - Keep a byte for each page
    - Update the byte every period by shifting the reference bit from the right
    - The page with the lowest value in the byte is replaced
  - Second-chance algorithm – zero history bits
    - FIFO that considers reference bit
    - Clock algorithm – circular queue (advance pointer and clear bits)
      - FIFO when all bits are set
  - Enhanced second chance algorithm
    - Consider both reference and modify bits as ordered pair
    - Preference to unmodified pages to reduce I/O

Page Replacement (7)

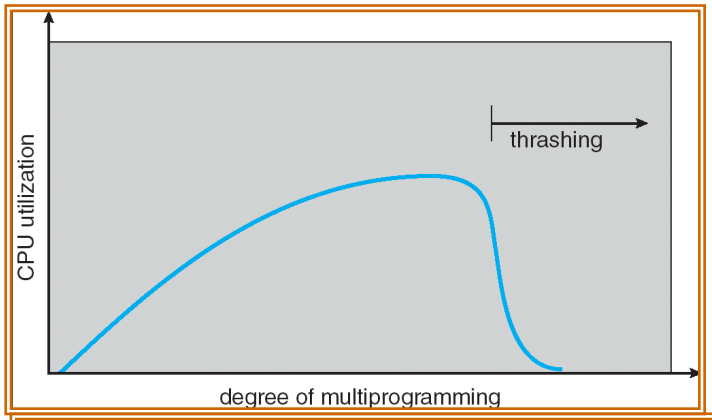
- Counting-based page replacement
  - Least-frequently used (LFU) page replacement algorithm
    - Shift bits regularly to avoid the initialisation problem
  - Most frequently used (MFU)
  - Not good approximations of optimal and not very common
- Page-buffering algorithms
  - Maintain pool of free frames, first read in the read out
  - Utilise the page device when idle to write modified pages
  - Check whether the page is already in the free frame pool
    - Combined with FIFO replacement or Second-chance algorithm
- Applications and page replacement
  - Sometimes its better to let applications management their buffering and page replacement – without the OS
  - Raw disk for raw I/O

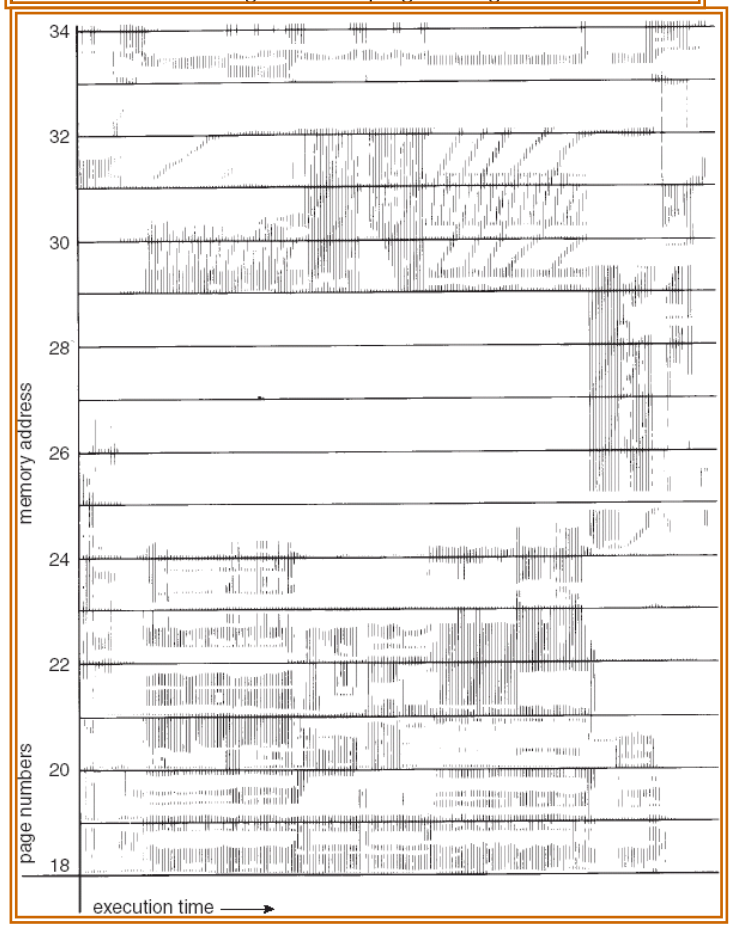
Allocation of Frames

- Allocate "all" free frames to user process
  - Keep some for OS buffering, or for handling page faults or for swapping
- Minimum number of frames
  - Good for performance reasons
  - Allocate at as many frames as an instruction may reference
    - Limit multiple levels of indirection
- Allocation algorithms
  - Equal allocation
  - Proportional allocation – adjust to integer beyond minimum
    - Instead of process size according to priority
  - In either case share is affected by the level of multiprogramming
- Global versus local allocation
  - Global or local page replacement algorithm – general or within process
  - In global allocation processes do not control their page fault rate – performance variation
  - Global allocation leads to greater throughput and is more common

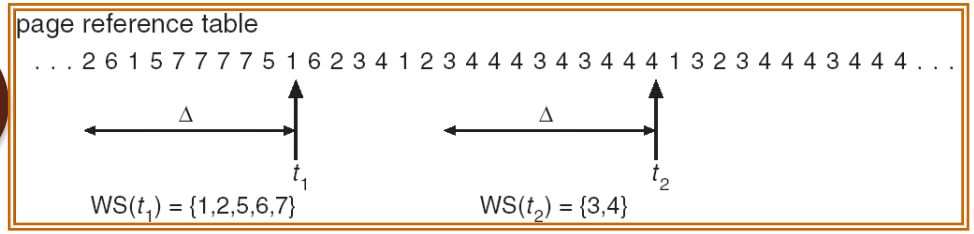
Thrashing (1)

- A process is thrashing if it spends more time paging than executing
  - Frame allocation below the minimum required
- Thrashing is a severe performance problem
- Local replacement algorithm or priority replacement algorithm can limit the effects of thrashing – performance may still suffer
- Locality model of process execution
  - Function calls define a new locality
  - Caching only works because of locality!!
- If fewer frames than the current locality are allocation to a process, then the process will thrash!!!

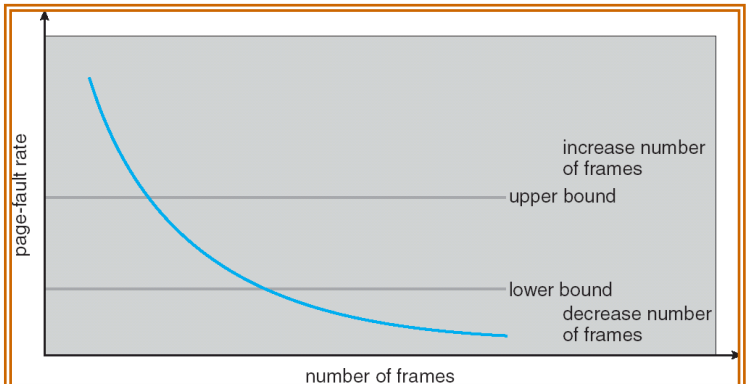




Thrashing (2)

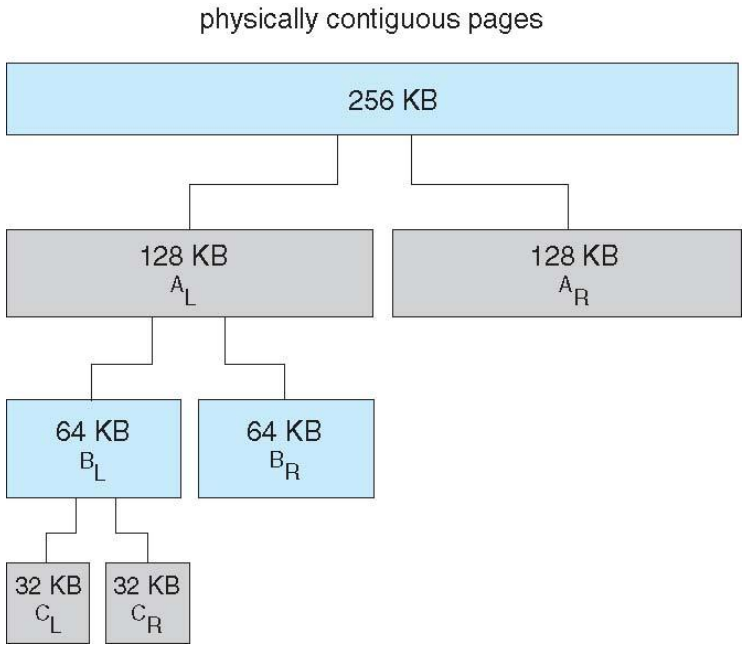


- Working set model
  - $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references, e.g. 10,000 instructions
  - WSSi (working set of Process Pi) = total number of pages referenced in the most recent Δ (varies in time)
    - if Δ too small will not encompass entire locality
    - if Δ too large will encompass several localities
    - if $\Delta = \infty \Rightarrow$ will encompass entire program
  - $D = \Sigma$ WSSi $\equiv$ total demand frames
    - if $D > m \Rightarrow$ Thrashing
  - Policy if D > m, then suspend one of the processes
    - Prevent thrashing while maintaining the level of multiprogramming as high as possible
  - Approximation with fixed-interval timer interrupt and a reference bit
    - More bits and more frequent interrupts – more accurate history
    - Accuracy versus cost!
- Page-fault frequency



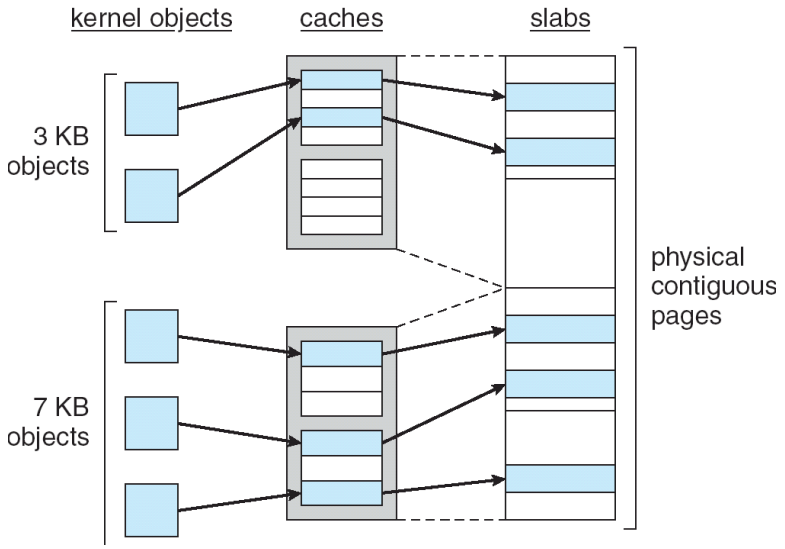
Allocating Kernel Memory (1)

- Kernel allocation from a different free-memory pool
  - Conservative use of memory, not subject to the paging system
  - Physically contiguous allocation for devices that interact directly with memory
- Buddy system – power of 2 allocator (coalescing)
  - Internal fragmentation



Allocating Kernel Memory (2)

- Slab allocation
  - Slab = one or more physically contiguous pages
  - Cache consists of one or more slabs
    - Separate cache for each unique kernel data structure
    - Populated with objects
    - Free versus used objects
    - Slab state: Full, Empty or Partial
  - No fragmentation
  - Quick serving of memory requests



For contemplation (1)

- Consider a demand-paging system with the following time-measured utilizations:

  CPU utilization 20%

  Paging disk 97.7%

  Other I/O devices 5%

  Which (if any) of the following will (probably) improve CPU utilization? Explain your answer.

  - Install a faster CPU.
  - Install a bigger paging disk.
  - Increase the degree of multiprogramming.
  - Decrease the degree of multiprogramming.
  - Install more main memory.
  - Install a faster hard disk or multiple controllers with multiple hard disks.
  - Increase the page size.

For contemplation (2)

- What is the cause of thrashing? How does the system detect thrashing? Once it detects thrashing, what can the system do to eliminate this problem?
- Assume we have a demand-paged memory. The page table is held in registers. It takes 8 milliseconds to service a page fault if an empty page is available or the replaced page is not modified, and 20 milliseconds if the replaced page is modified. Memory access time is 100 nanoseconds. Assume that the page to be replaced is modified 70 percent of the time. What is the maximum acceptable page-fault rate for an effective access time of no more than 200 nanoseconds?
- Assume that you are monitoring the rate at which the pointer in the clock algorithm (which indicates the candidate page for replacement) moves. What can you say about the system if you notice the following behaviour:
  - pointer is moving fast
  - pointer is moving slow

For contemplation (3)

- In a page replacement algorithm what is Belady's anomaly? Consider Least-Recently-Used (LRU) replacement, First-In-First-Out (FIFO) replacement, Optimal replacement, and Second-chance replacement: which of these algorithms suffer from Belady's anomaly and which do not? In each algorithm that suffers from the anomaly demonstrate the problem with an example.
- Consider the following page reference string:

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

  How many page faults would occur for the following replacement algorithms, assuming four or six frames? Note that all frames are initially empty.

  - Least Recently Used (LRU) replacement
  - First In First Out (FIFO) replacement
  - Optimal replacement

  Your answer should clearly demonstrate the various steps of the calculation, not just the final number of page faults.

For contemplation (4)

- What is the copy-on-write feature, and under what circumstances is it beneficial? What hardware support is required to implement this feature?
- The slab allocation algorithm uses a separate cache for each different object type. Assuming there is one cache per object type, explain why this scheme doesn't scale well with multiple CPUs. What can be done to address this scalability issue?
- Assume there is a 1,024KB segment where memory is allocated using the buddy system. Using Figure 9.26 as a guide, draw a tree illustrating how the following memory requests are allocated:
  - Request 240 bytes
  - Request 120 bytes
  - Request 60 bytes
  - Request 130 bytes
- Next modify the tree for the following releases of memory. Perform coalescing whenever possible:
  - Release 240 bytes
  - Release 60 bytes
  - Release 120 bytes