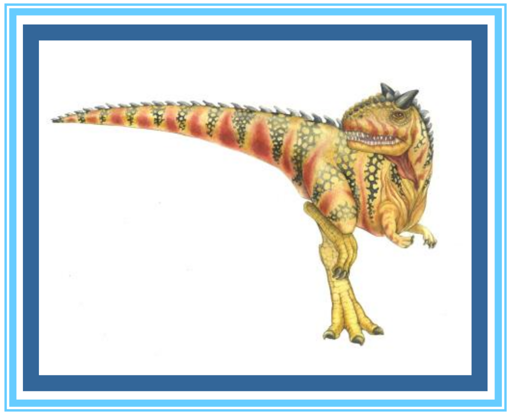
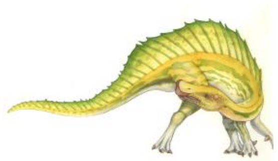
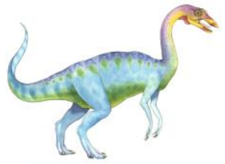
Chapter 10: Virtual Memory



Background

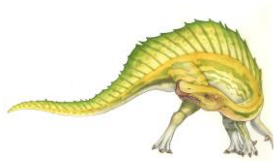
- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at the same time
- Consider ability to execute partially-loaded program
  - Program no longer constrained by limits of physical memory
  - Each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





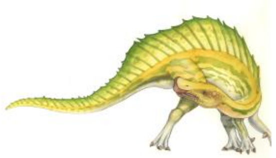
Virtual memory

- **Virtual memory** – separation of user logical memory from physical memory
  - Only part of the program needs to be in memory for execution
  - Logical address space can therefore be much larger than physical address space
  - Allows address spaces to be shared by several processes
  - Allows for more efficient process creation
  - More programs running concurrently
  - Less I/O needed to load or swap processes

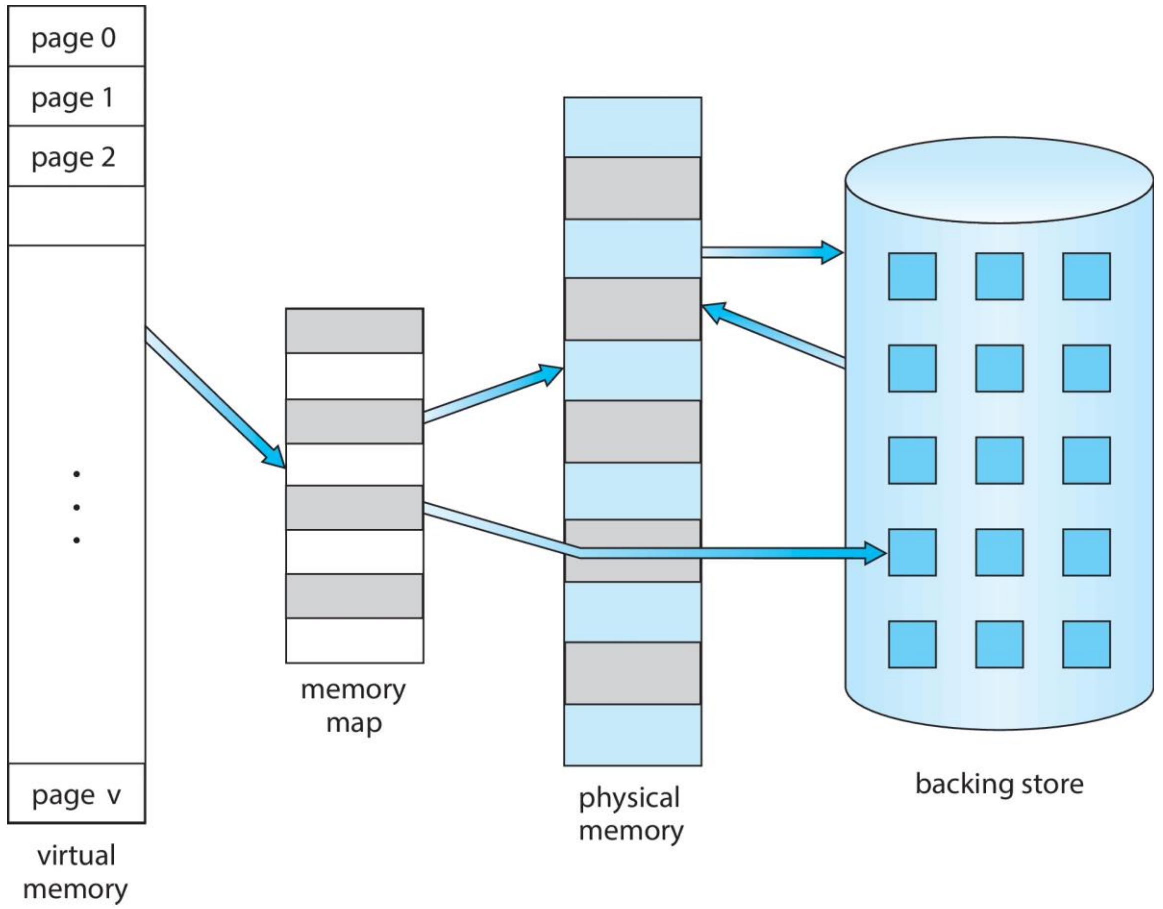


Virtual Memory (Cont.)

- **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

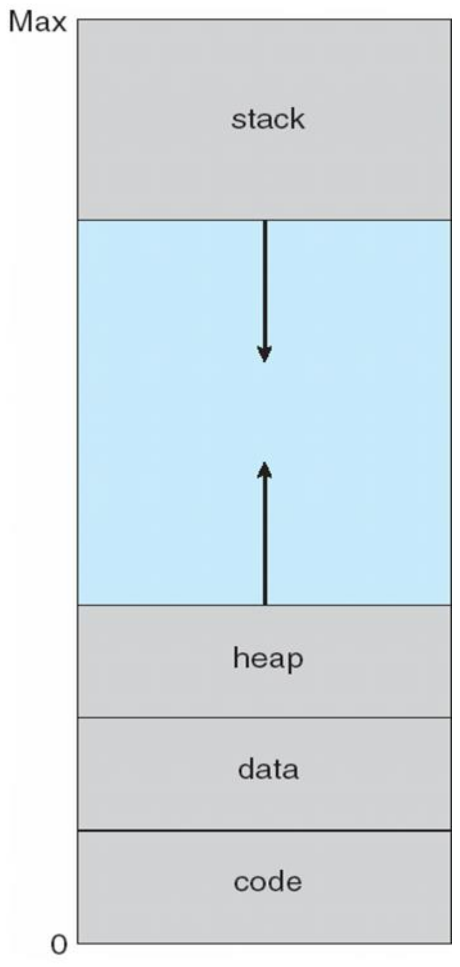


Virtual Memory That is Larger Than Physical Memory

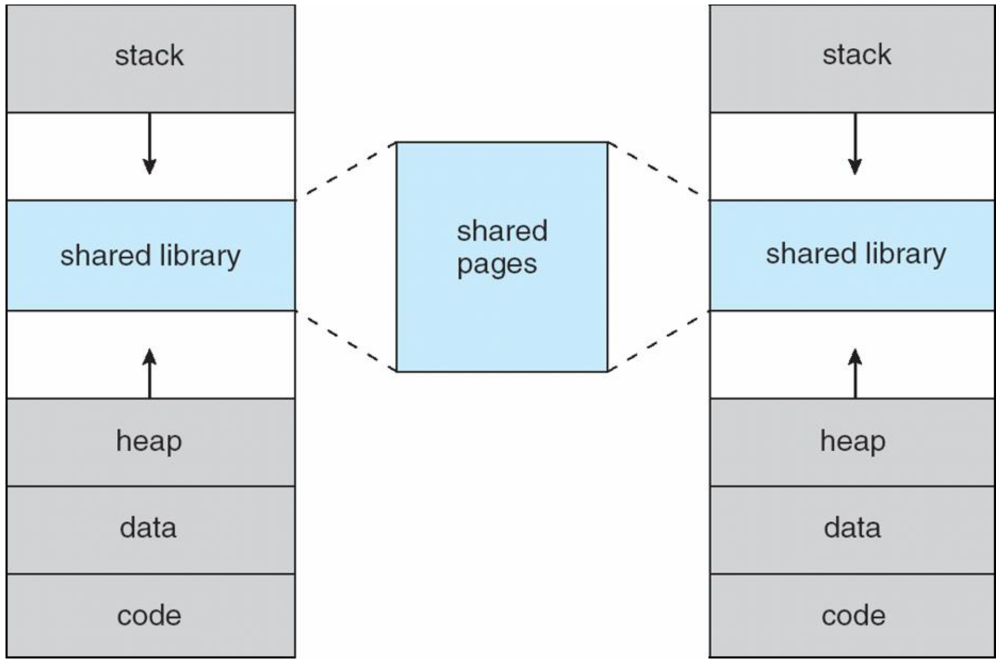


Virtual-address Space

- Usually design logical address space for the stack to start at Max logical address and grow "down" while heap grows "up"
  - Maximizes address space use
  - Unused address space between the two is hole
    - No physical memory needed until heap or stack grows to a given new page
- Enables **sparse** address spaces with holes left for growth, dynamically linked libraries, etc.
- System libraries shared via mapping into virtual address space
- Shared memory by mapping pages read-write into virtual address space
- Pages can be shared during **`fork()`**, speeding process creation

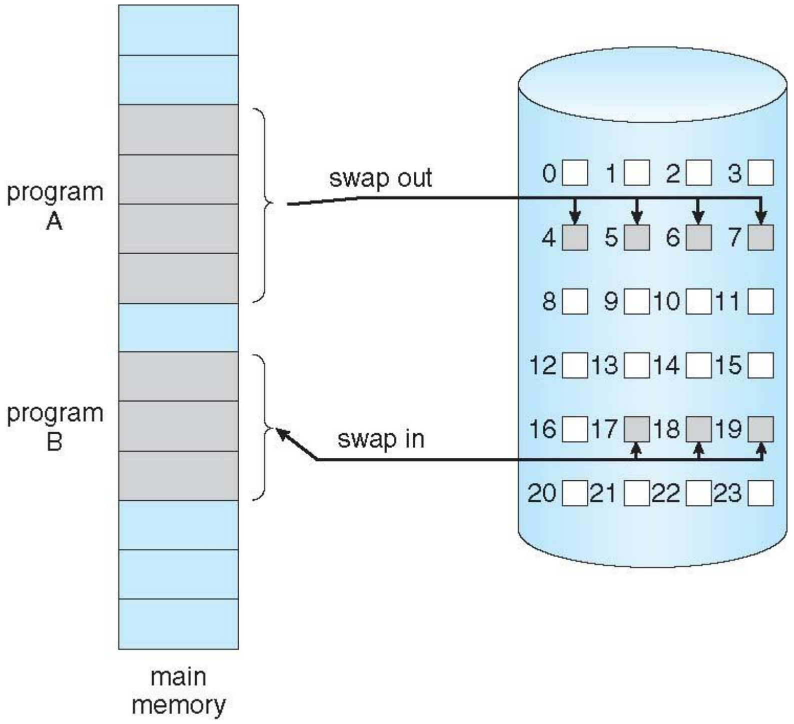


Shared Library Using Virtual Memory



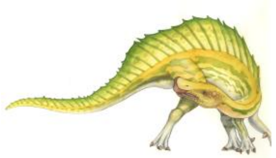
Demand Paging

- Instead of bringing the entire program into memory at load time, 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 reference
  - Invalid reference ⇒ abort
  - Not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless page will be needed
  - Swapper that deals with pages is a **pager**



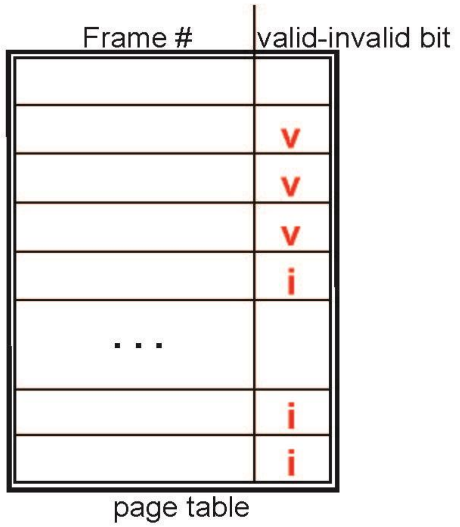
Basic Concepts

- With swapping, the pager guesses which pages will be used before swapping them out again
  - How to determine that set of pages?
- Need new MMU functionality to implement demand paging
- If pages needed are already **memory resident**
  - No difference from non-demand paging
- 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
- Use page table with valid-invalid bit (see chapter 9)



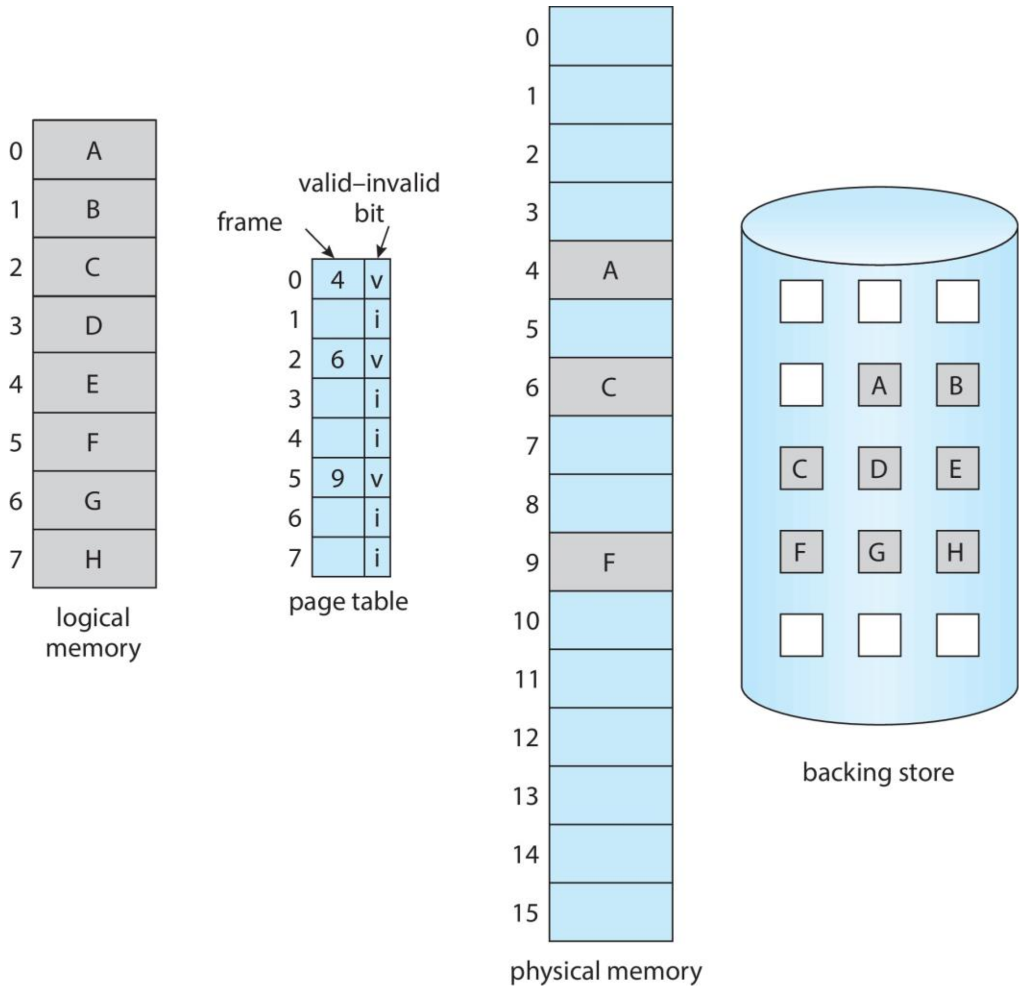
Page table with Valid-Invalid Bit

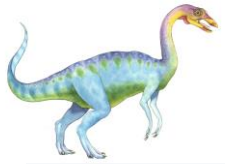
- With each page table entry a valid–invalid bit is associated ($\mathbf{v} \Rightarrow$ in-memory, $\mathbf{i} \Rightarrow$ not-in-memory)
- Initially valid–invalid bit is set to **i** on all entries
- Example of a page table snapshot:



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

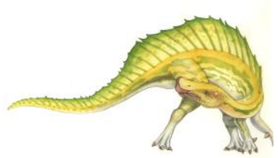
Page Table When Some Pages Are Not in Main Memory



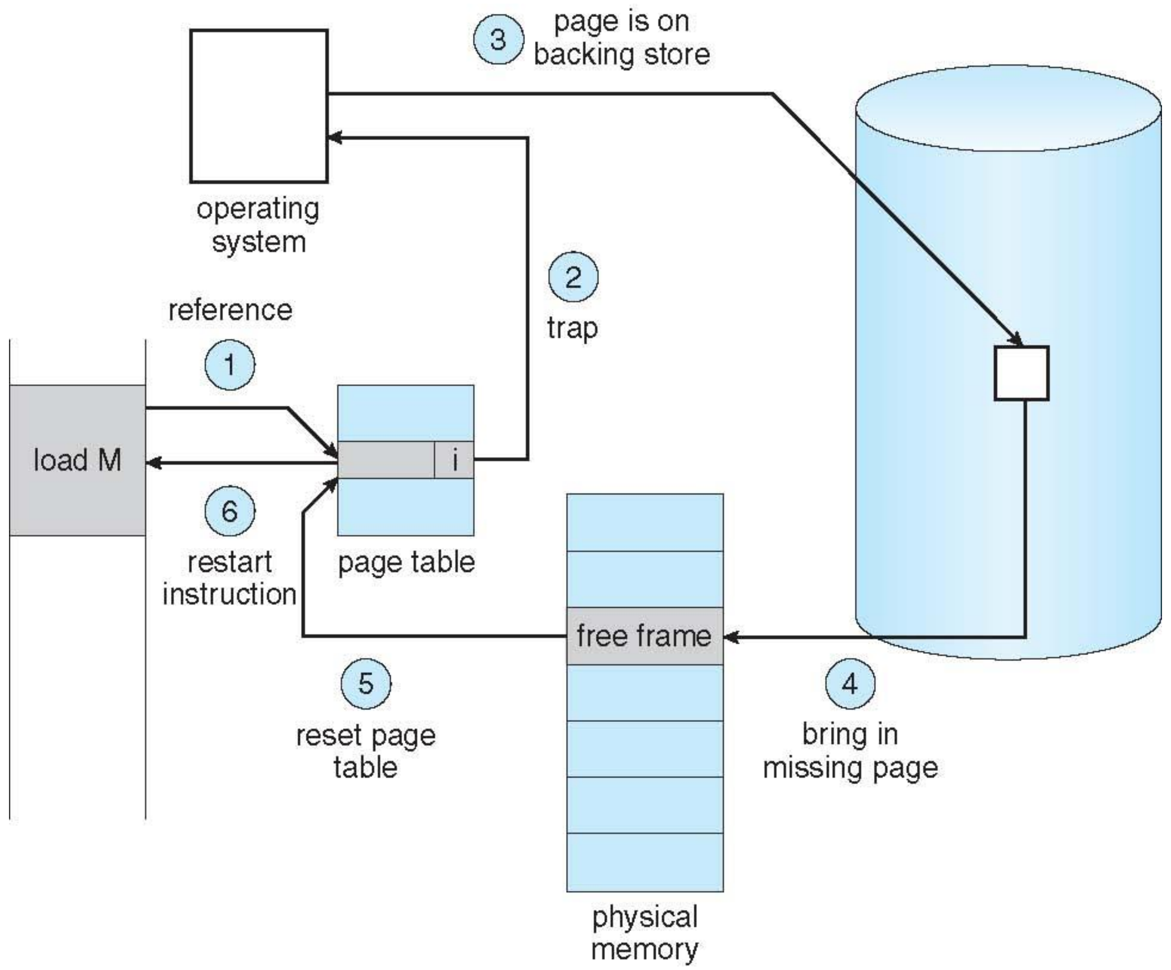


Steps in Handling Page Fault

1. The first reference to a page will trap to operating system
   - Page fault
2. Operating system looks at another table to decide:
   - Invalid reference $\Rightarrow$ abort
   - Just not in memory (go to step 3)
3. Find free frame (what if there is none?)
4. Swap page into frame via scheduled disk operation
5. Reset tables to indicate page now in memory
   Set validation bit = **v**
6. Restart the instruction that caused the page fault

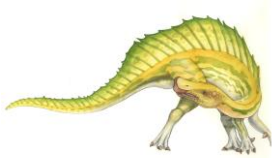


Steps in Handling a Page Fault (Cont.)



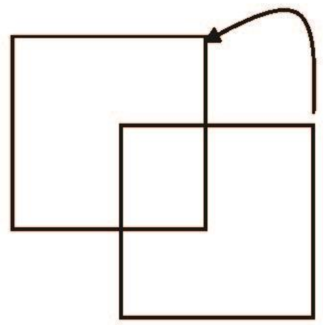
Aspects of Demand Paging

- **Pure demand paging:** start process with *no* pages in memory
  - OS sets program counter to pointer to the first instruction of the process, non-memory-resident $\Rightarrow$ page fault
  - And for every other process pages on first access
- Actually, a given instruction could access multiple pages $\Rightarrow$ multiple page faults
  - Consider fetch and decode of instruction which adds 2 numbers from memory and stores result back to memory
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - Instruction restart



Instruction Restart

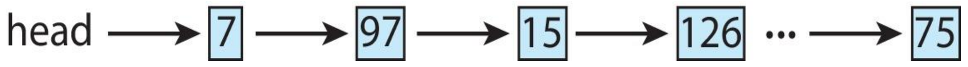
- Consider an instruction that could access several different locations
  - Block move



  - Auto increment/decrement location
  - Restart the whole operation?
    - What if source and destination overlap?

Free-Frame List

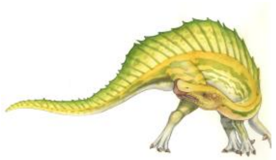
- When a page fault occurs, the operating system must bring the desired page from secondary storage into main memory.
- Most operating systems maintain a **free-frame list** -- a pool of free frames for satisfying such requests.



- Operating system typically allocate free frames using a technique known as **zero-fill-on-demand** -- the content of the frames is zeroed-out before being allocated.
- When a system starts up, all available memory is placed on the free-frame list.

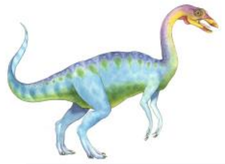
Stages in Demand Paging – Worse Case

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. Find a free frame. If there is none available, evict a page.
6. Issue a read from the disk to a free frame:
   a) Wait in a queue for this device until the read request is serviced
   b) Wait for the device seek and/or latency time
   c) Begin the transfer of the page to a free frame



Stages in Demand Paging (Cont.)

7. While waiting, allocate the CPU to some other user
8. Receive an interrupt from the disk I/O subsystem (I/O completed)
9. Save the registers and process state for the other user
10. Determine that the interrupt was from the disk
11. Correct the page table and other tables to show page is now in memory
12. Wait for the CPU to be allocated to this process again
13. Restore the user registers, process state, and new page table, and then resume the interrupted instruction



Performance of Demand Paging

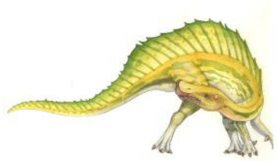
- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Input the page from disk – lots of time
  - Restart the process – again just a small amount of time
- Page Fault Rate $0 \le p \le 1$
  - If $p = 0$ no page faults
  - If $p = 1$, every reference is a fault
- Effective Access Time (EAT)

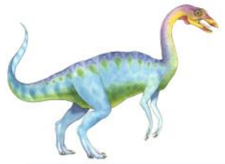
  EAT = $(1 – p)$ x memory access

  + p (page fault overhead

  + swap page out

  + swap page in)





Demand Paging Example

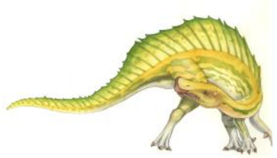
- Memory access time = 200 nanoseconds
- Average page-fault service time = 8 milliseconds
- EAT = (1 – p) x 200 + p (8 milliseconds)

  = (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

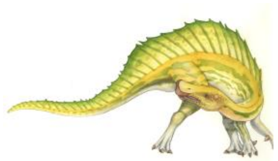
  EAT = 8.2 microseconds.

  This is a slowdown by a factor of 40!!
- If 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



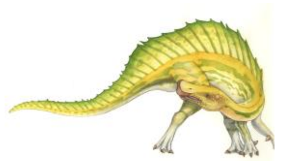
Demand Paging Optimizations

- Swap space I/O faster than file system I/O even if on the same device
  - Swap allocated in larger chunks; less management needed than file system
- Copy entire process image to swap space at process load time
  - Then page in and out of swap space
  - Used in older BSD Unix
- Demand page in from program binary on disk, but discard rather than paging out when freeing frame
  - Used in Solaris and current BSD
  - Still need to write to swap space
    - Pages not associated with a file (like stack and heap) – **anonymous memory**
    - Pages modified in memory but not yet written back to the file system
- Mobile systems
  - Typically, don't support swapping
  - Instead, demand page from file system and reclaim read-only pages (such as code)

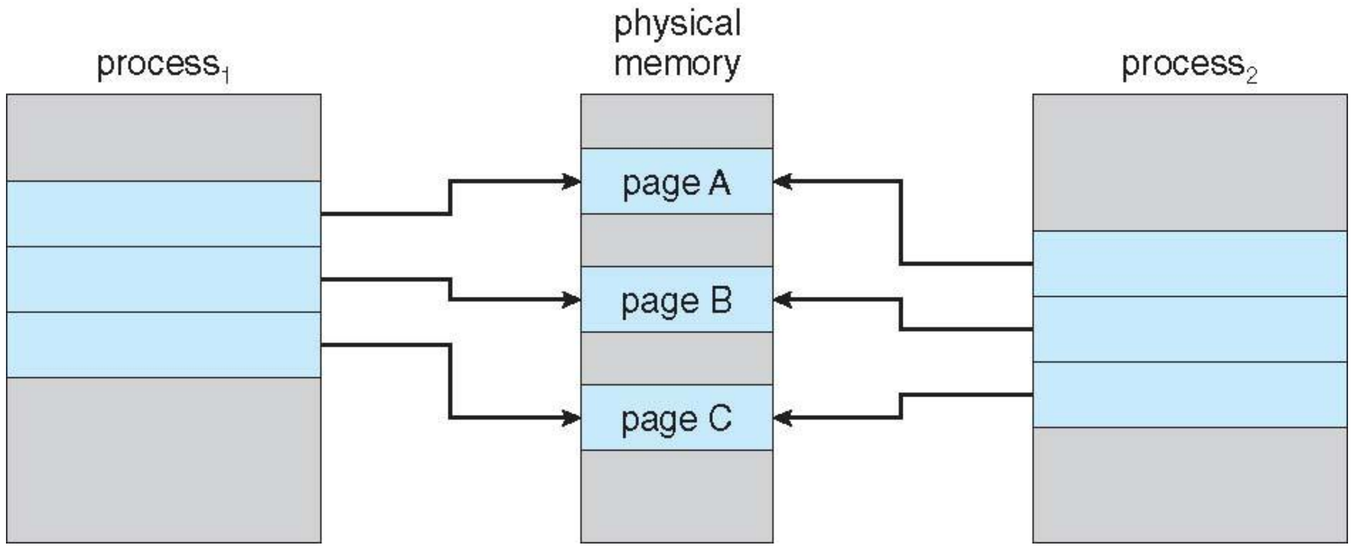


Copy-on-Write

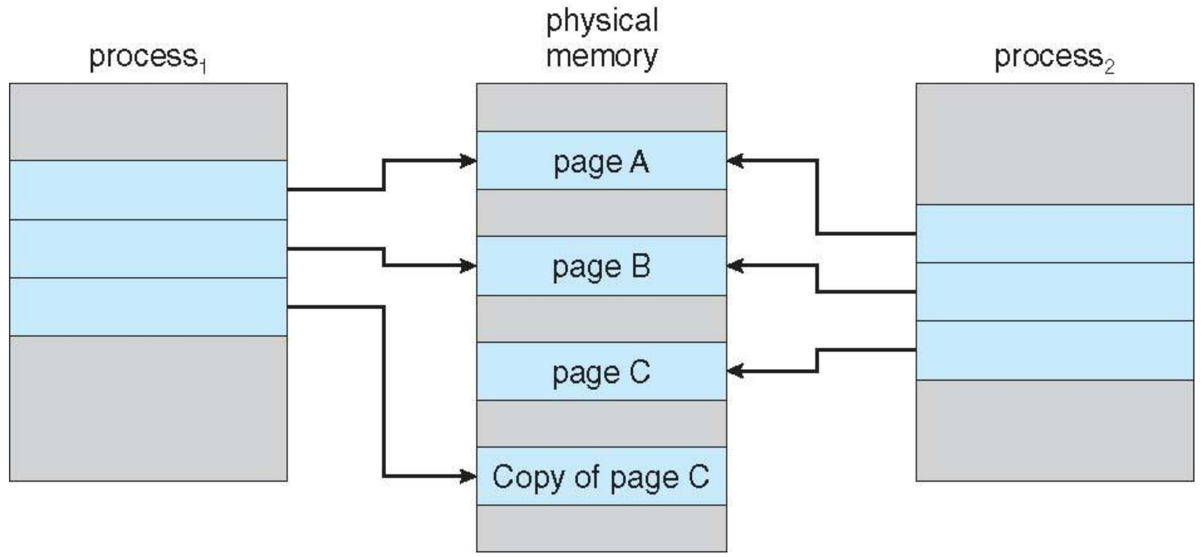
- **Copy-on-Write** (COW) allows both parent and child processes to initially ***share*** the same pages in memory
  - If either process modifies a shared page, only then is the page copied
- COW allows more efficient process creation as only modified pages are copied
- In general, free pages are allocated from a **pool** of **zero-fill-on-demand** pages
  - 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
  - Why zero-out a page before allocating it?
- `vfork()` variation on `fork()` system call has parent suspend and child using copy-on-write address space of parent
  - Designed to have child call `exec()`
  - Very efficient

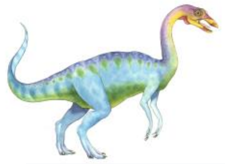


Before Process 1 Modifies Page C



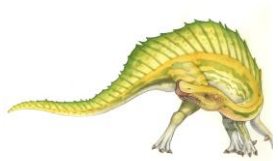
After Process 1 Modifies Page C





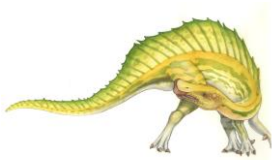
What Happens if There is no Free Frame?

- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance – want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times

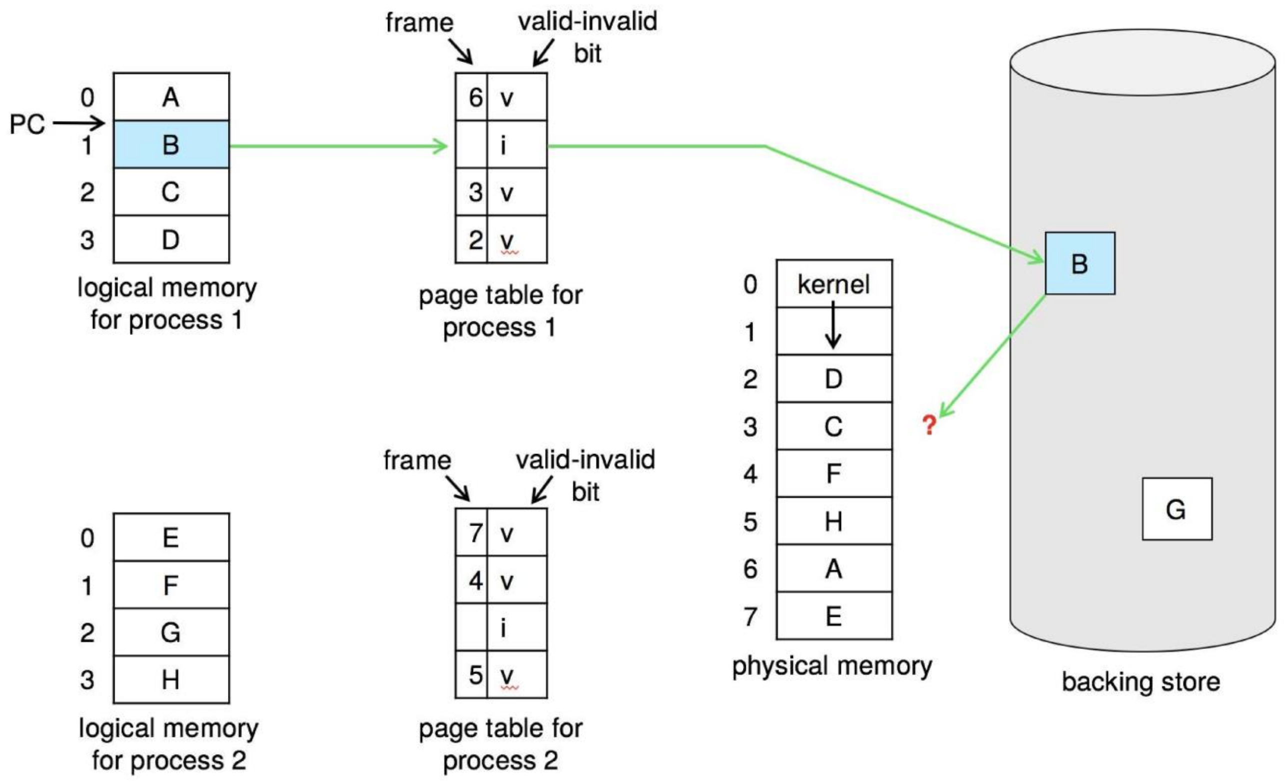


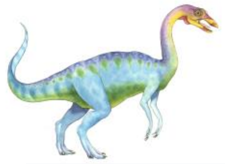
Page Replacement

- 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

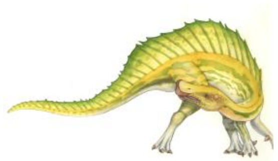




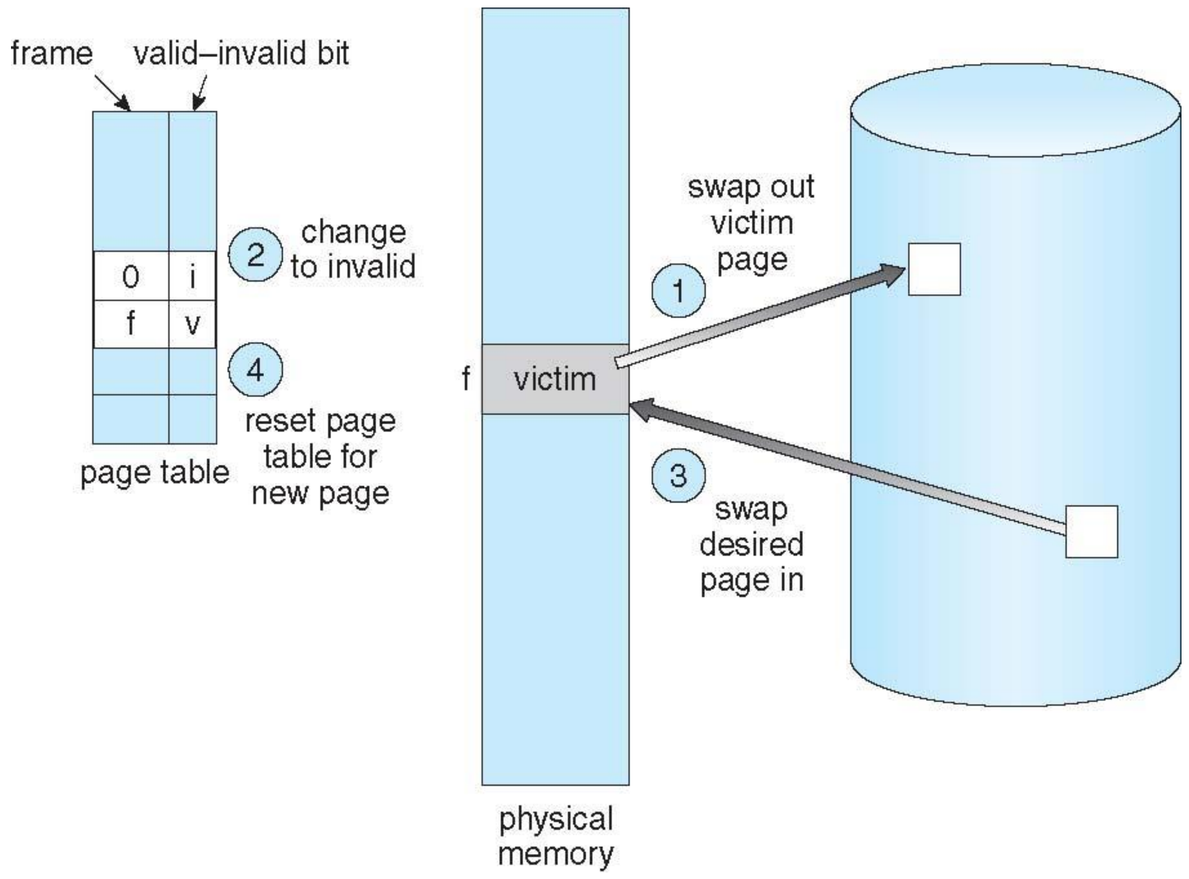
Basic Page Replacement

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

Note now potentially 2 page transfers for page fault – increasing EAT



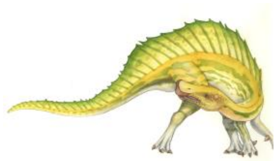
Page Replacement



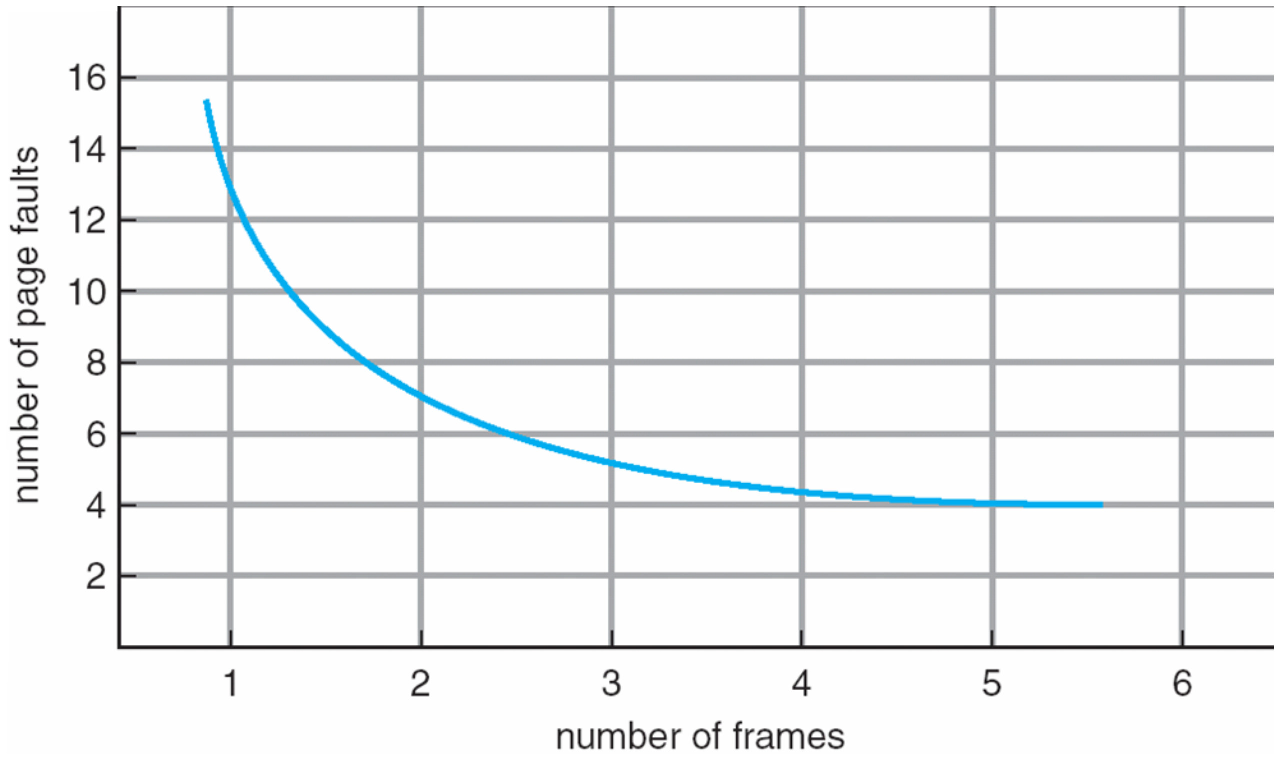
Page and Frame Replacement Algorithms

- **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

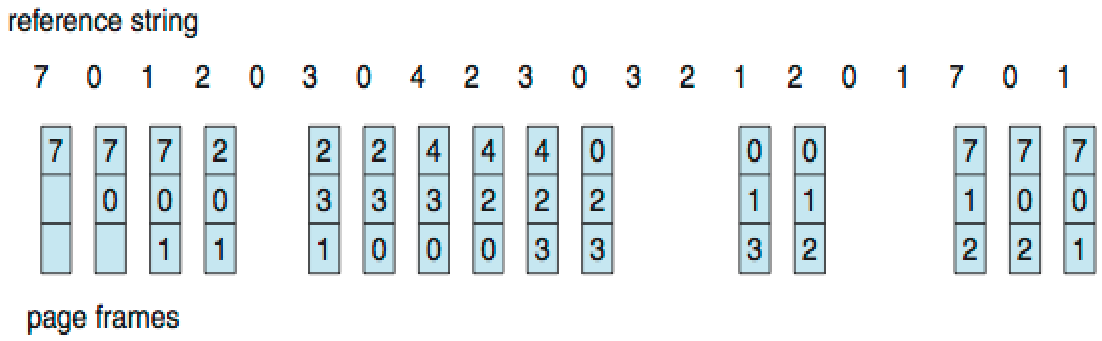


Graph of Page Faults Versus the Number of Frames



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)

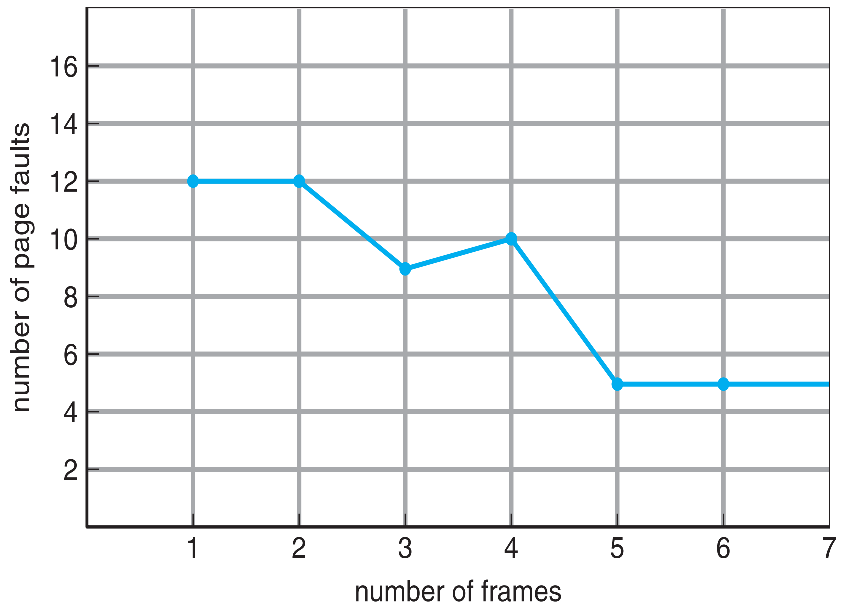


15 page faults

- How to track ages of pages?
  - Just use a FIFO queue

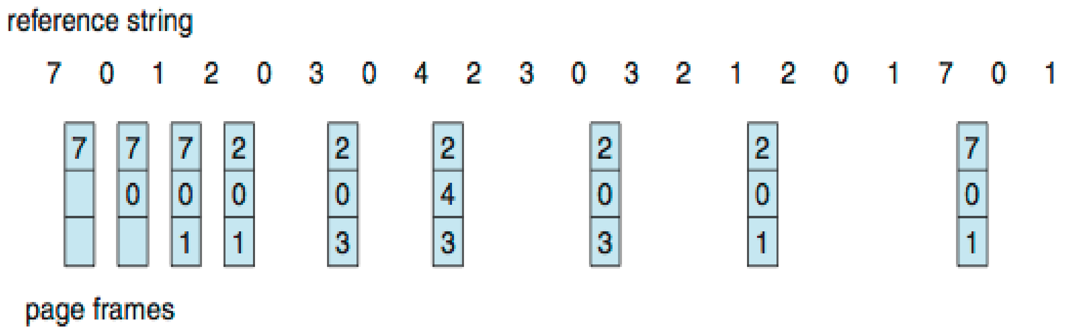
Belady's Anomaly

- Consider the string 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
- Graph illustrating Belady's Anomaly



Optimal Algorithm

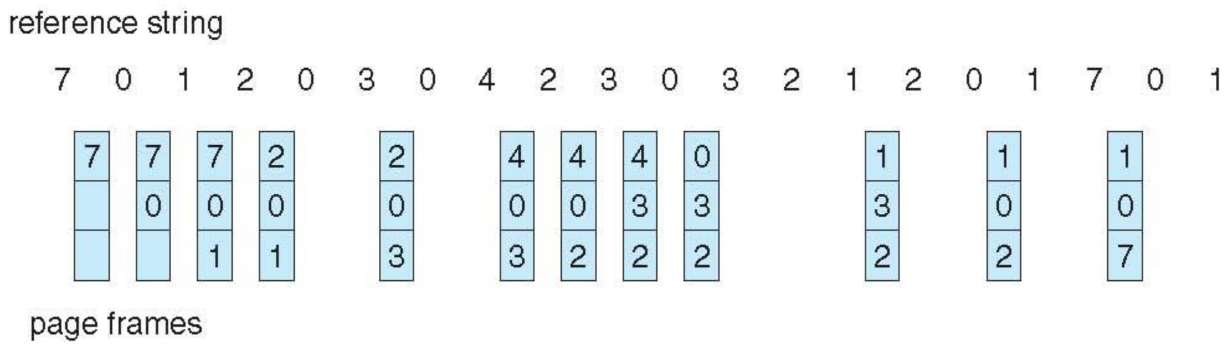
- Replace page that will not be used for longest period of time
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
    - Optimal algorithm



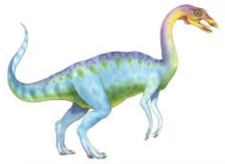
    - 9 page faults
- How do you implement this?
    - Can't read the future
- Used for measuring how well your algorithm performs
- Optimal is an example of **stack algorithms** that don't suffer from Belady's Anomaly

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 is another example of stack algorithms; thus, it does not suffer from Belady's Anomaly

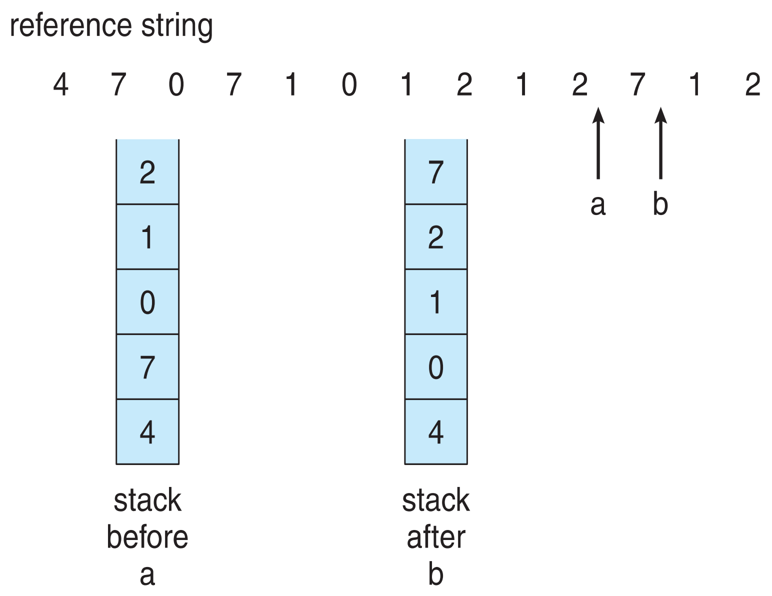


LRU Algorithm Implementation

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

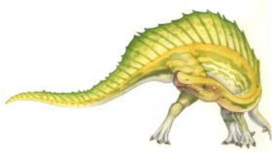
Stack Implementation

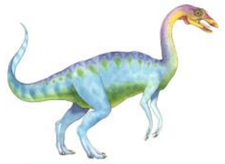
- Use of a stack to record most recent page references



LRU Approximation Algorithms

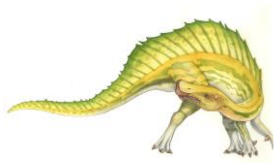
- Needs special hardware
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced, bit set to 1
- When a page needs to be replaced, replace any with reference bit = 0 (if one exists)
  - We do not know the order, however



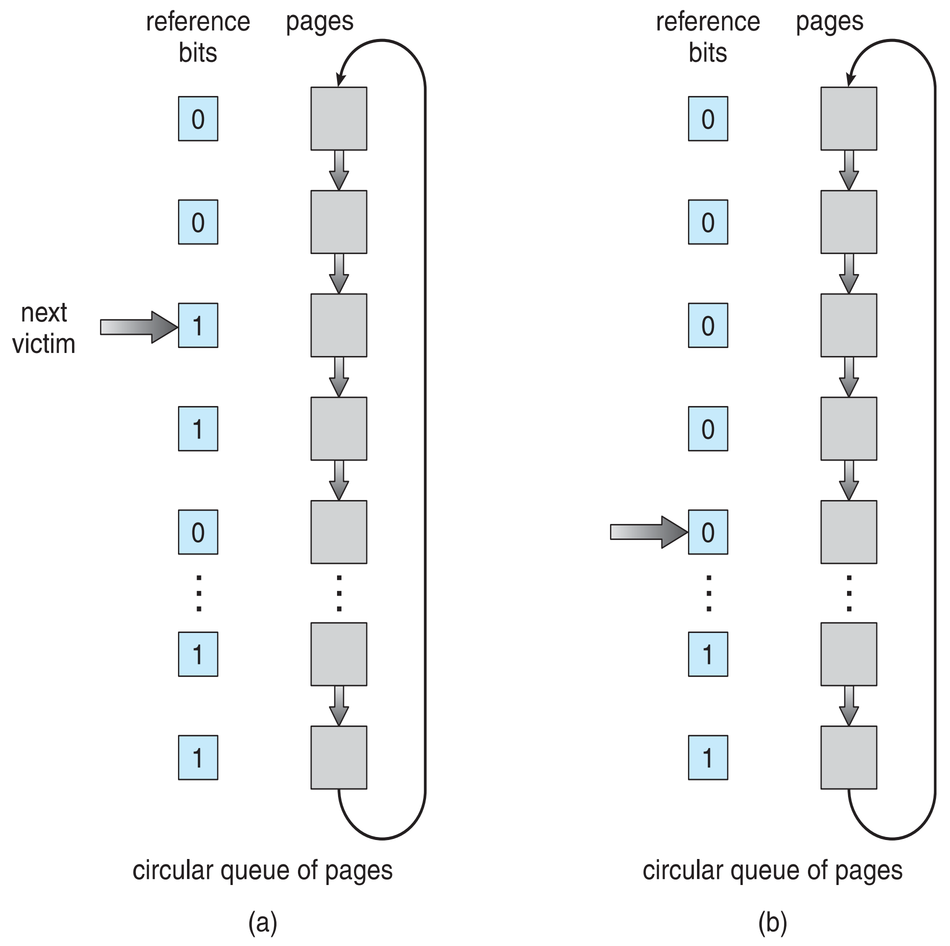


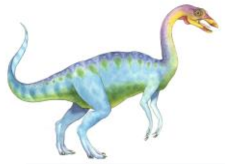
LRU Approximation Algorithms (cont.)

- **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



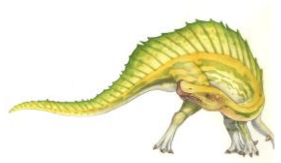
Second-chance Algorithm





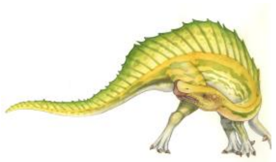
Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify):
  - (0, 0) neither recently used not modified – best page to replace
  - (0, 1) not recently used but modified – not quite as good, must write out before replacement
  - (1, 0) recently used but clean – probably will be used again soon
  - (1, 1) recently used and modified – probably will be used again soon and need to write out before replacement
- When page replacement called for, use the clock scheme but use the four classes replace page in lowest non-empty class
  - Might need to search circular queue several times



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

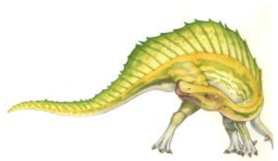
- Keep a pool of free frames, which is never empty
  - Thus, frame is always available when needed
  - Read page into free frame and select victim to evict and add to free pool
  - When convenient, evict victim
- Possibly, keep list of modified pages
  - When backing store otherwise idle, write pages there and set to non-dirty
- Possibly, keep free frame contents intact and note what is in them
  - If referenced again before reused, no need to load contents again from disk
  - Generally useful to reduce penalty if wrong victim frame selected

Applications and Page Replacement

- All of these algorithms have OS guessing about future page access
- Some applications have better knowledge – i.e., databases
- Memory intensive applications can cause double buffering
  - OS keeps copy of page in memory as I/O buffer
  - Application keeps page in memory for its own work
- Operating system can provide direct access to the disk, getting out of the way of the applications
  - **Raw disk** mode
- Bypasses buffering, locking, etc.

Allocation of Frames

- Each process needs ***minimum*** number of frames
- Example: IBM 370 – 6 pages to handle SS MOVE instruction:
  - Instruction is 6 bytes, might span 2 pages
  - 2 pages to handle *from*
  - 2 pages to handle *to*
- ***Maximum*** of course is total frames in the system
- Two major allocation schemes
  - Fixed allocation
  - Priority allocation
- Many variations



Fixed Allocation

- Equal allocation – all processes gets the same number of frames.
  - 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
  - Dynamic as degree of multiprogramming, process sizes change
  - Example

$$s_i = \text{size of process } p_i$$

$$S = \sum s_i$$

$$m = \text{total number of frames}$$

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

$$m = 64$$

$$s_1 = 10$$

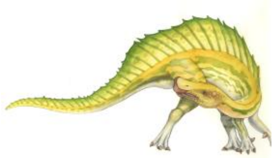
$$s_2 = 127$$

$$a_1 = \frac{10}{137} ´ 62 » 4$$

$$a_2 = \frac{127}{137} ´ 62 » 57$$

Global vs. Local Allocation

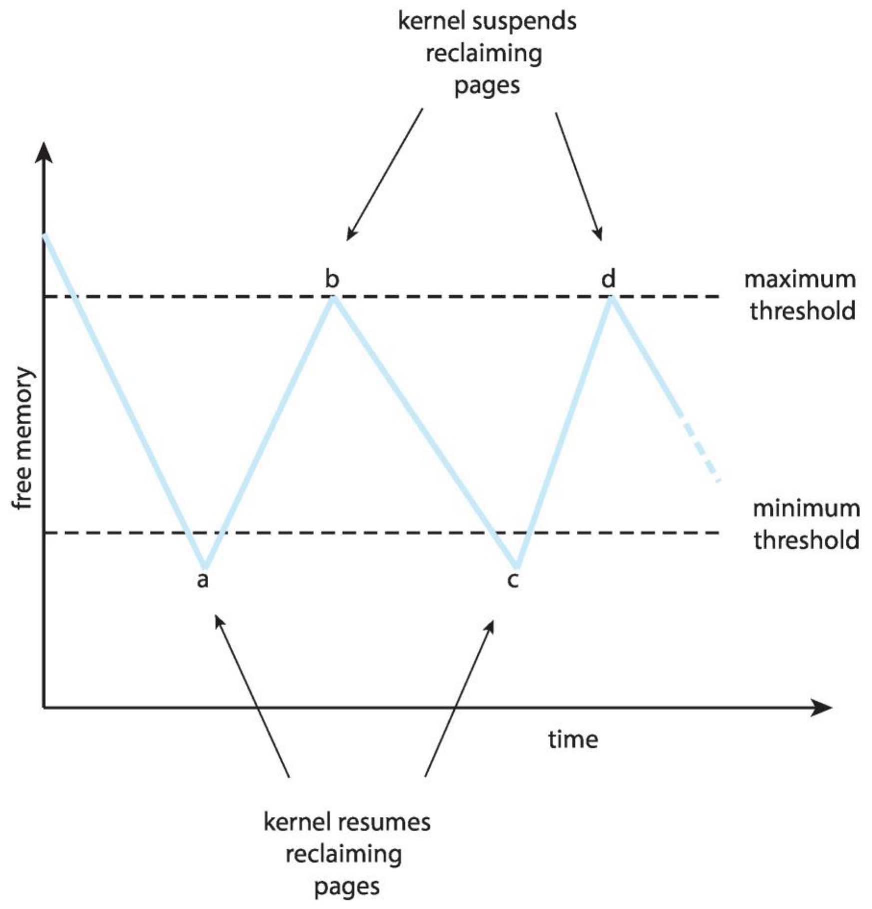
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - Process execution time can vary greatly
  - Greater throughput so more commonly used
- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory
  - What if a process does not have enough frames?



Reclaiming Pages

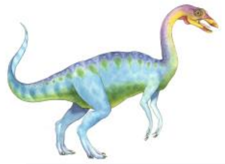
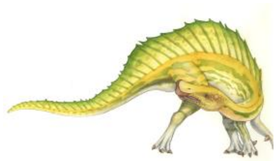
- A strategy to implement global page-replacement policy
- All memory requests are satisfied from the free-frame list, rather than waiting for the list to drop to zero before we begin selecting pages for replacement,
- Page replacement is triggered when the list falls below a certain threshold.
- This strategy attempts to ensure there is always sufficient free memory to satisfy new requests.

Reclaiming Pages Example



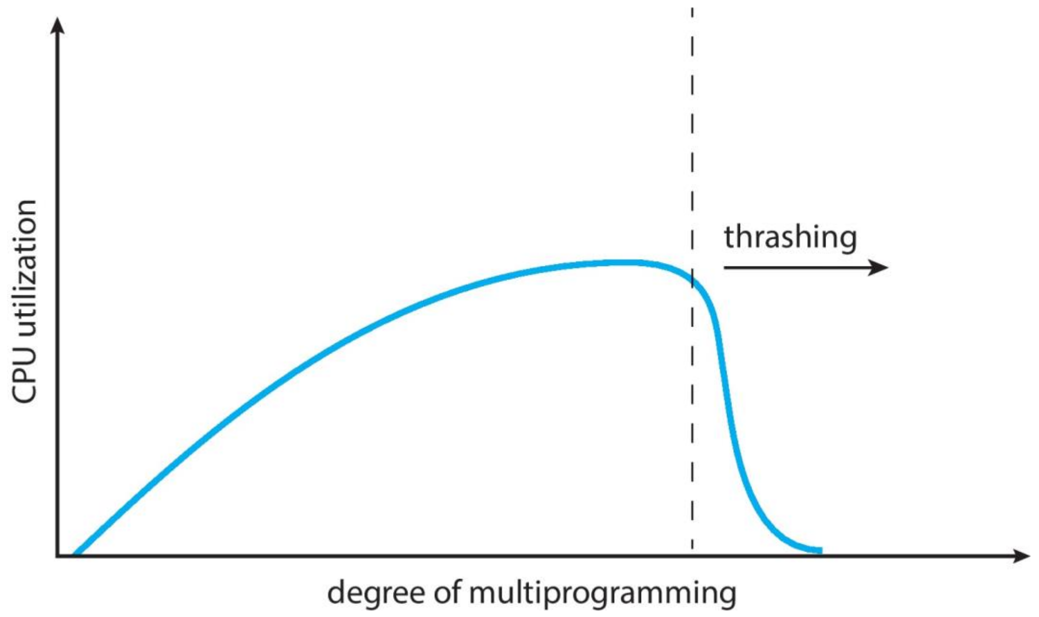
Thrashing

- 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 the 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
  - Which results in even higher page fault rate

Thrashing (Cont.)

- **Thrashing**. A process is busy swapping pages in and out



Demand Paging and Thrashing

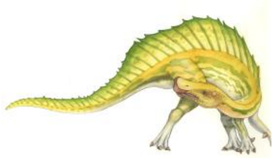
- Why does demand paging work?

  Locality model

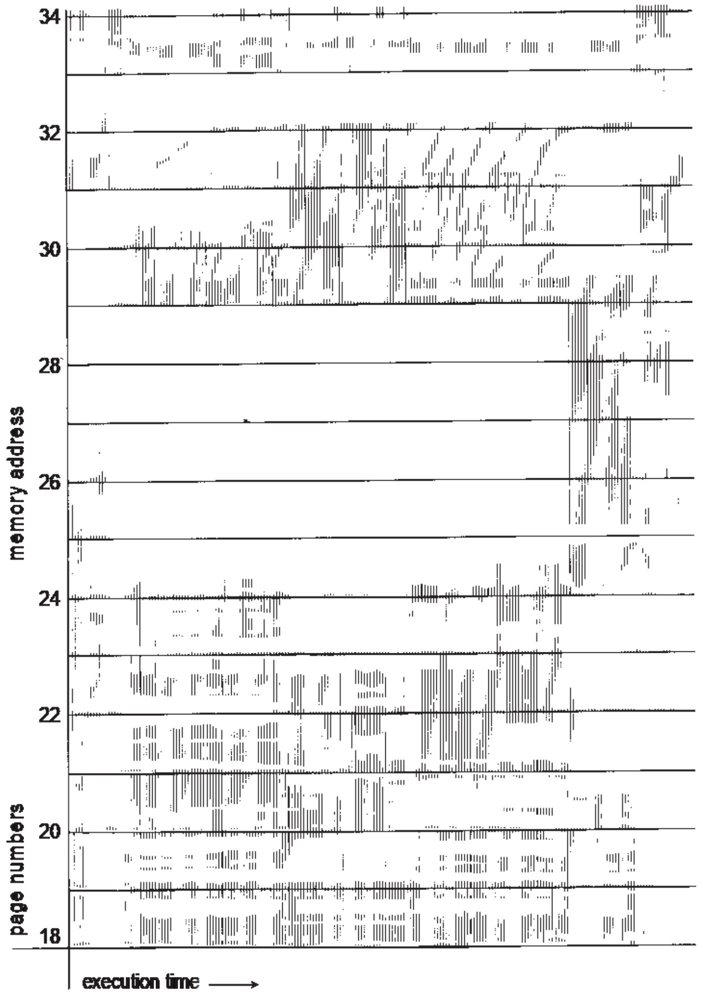
  - Process migrates from one locality to another
  - Localities may overlap
- Why does thrashing occur?

  Σ size of locality > total memory size

- To avoid trashing:
  - Calculate the Σ size of locality
  - Policy: if Σ size of locality > total memory size
    - suspend or swap out one of the processes
- Issue: how to calculate "Σ size of locality"

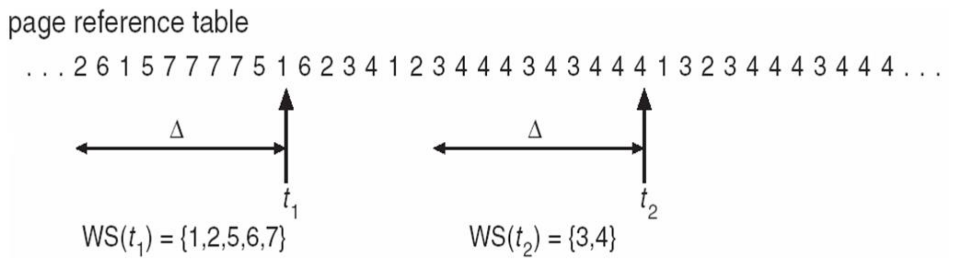


Locality In A Memory-Reference Pattern

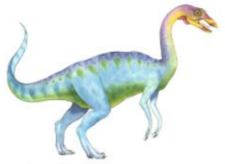


Working-Set Model

- $\Delta \equiv$ working-set window $\equiv$ a fixed number of page references
  - Example: 10,000 instructions
- WSS_i (working set of Process P_i) = total number of pages referenced in the most recent Δ (varies in time)
- Example

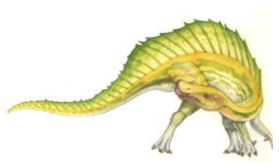


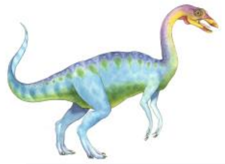
- Observation
  - if Δ too small will not encompass the entire locality
  - if Δ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program



Working-Set Model (Cont.)

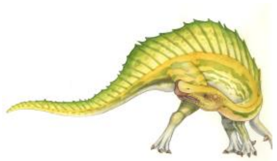
- $D = \Sigma\ WSS_i \equiv$ total demand frames
  - Approximation of locality
- Let m = total number of frames
- If $D > m \Rightarrow$ Thrashing
- Policy if D > m, then suspend or swap out one of the processes





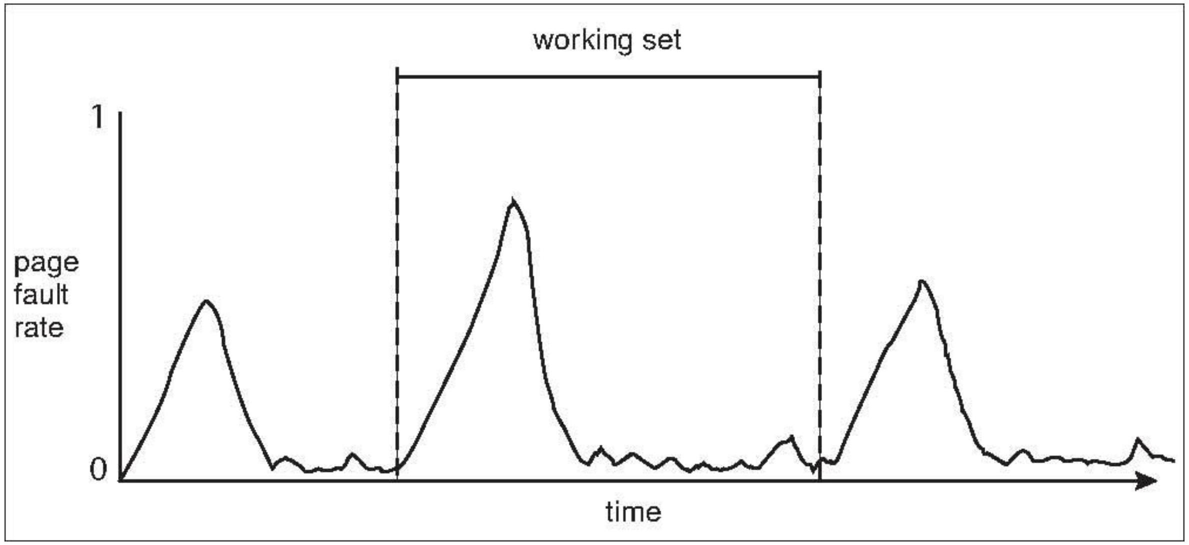
Keeping Track of the Working Set

- Approximate with interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page *i*
    - $B1_i$ and $B2_i$
  - Whenever a timer interrupts copy the reference to one of the B_j and sets the values of all reference bits to 0
  - If either $B1_i$ or $B2_i$ = 1, it implies that Page *i* is in the working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units



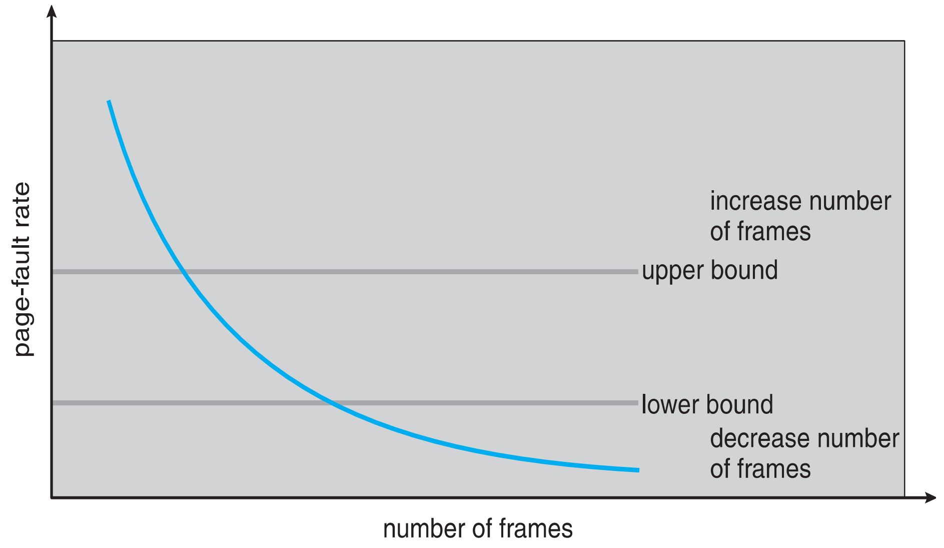
Working Sets and Page Fault Rates

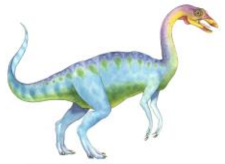
- Direct relationship between working set of a process and its page-fault rate
- Working set changes over time
- Peaks and valleys over time



Page-Fault Frequency Algorithm

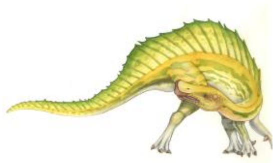
- More direct approach than WSS
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual rate too low, process loses frame
  - If actual rate too high, process gains frame
- Example

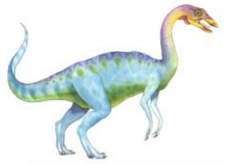




Allocating Kernel Memory

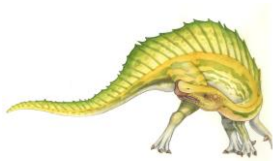
- Treated differently from user memory
- Often allocated from a free-memory pool
    - Kernel requests memory for structures of varying sizes
    - Some kernel memory needs to be contiguous
        - i.e., for device I/O
- Two schemes:
    - Buddy System
    - Slab Allocator





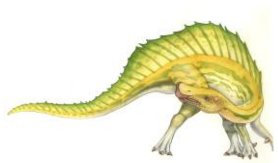
Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory allocated using **power-of-2 allocator**
  - Satisfies requests in units sized as power of 2
  - Request rounded up to next highest power of 2
  - When smaller allocation needed than is available, current chunk split into two buddies of next-lower power of 2
    - Continue until appropriate sized chunk available

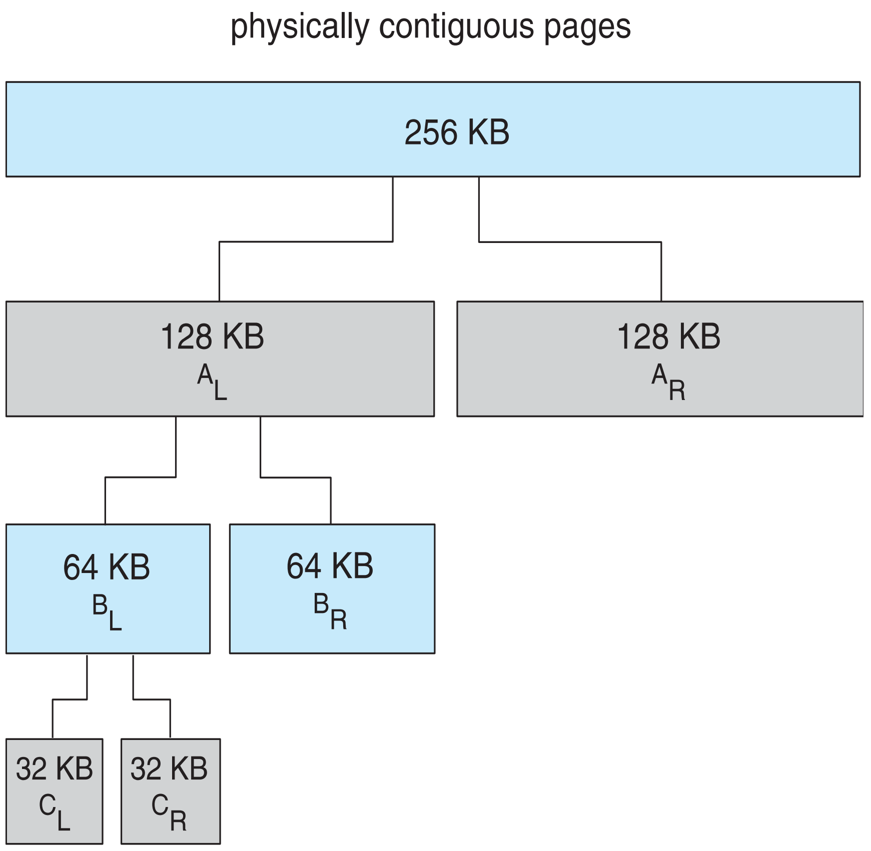


Buddy System Example

- Assume 256KB chunk available, kernel requests 21KB
  - Split into A_L and A_R of 128KB each
    - One further divided into B_L and B_R of 64KB
      - One further into C_L and C_R of 32KB each – one used to satisfy request
- Advantage – quickly **coalesce** unused chunks into larger chunk
- Disadvantage - fragmentation



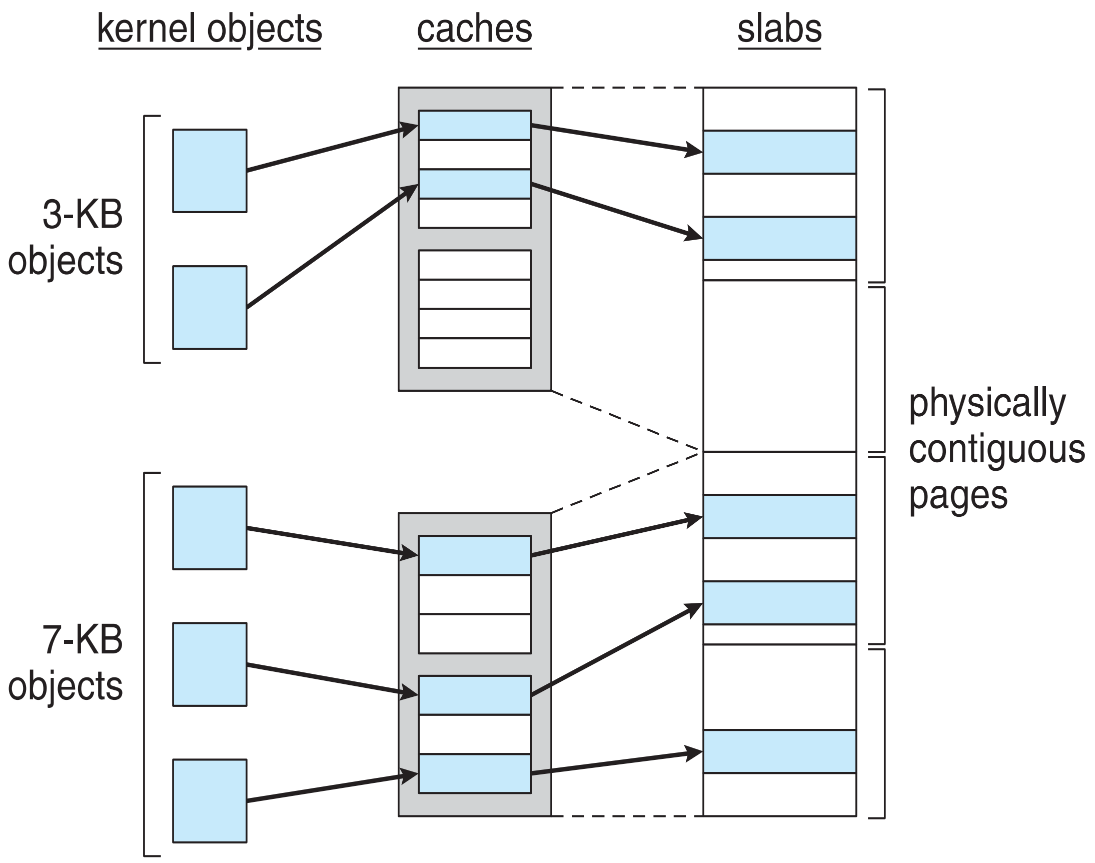
Buddy System Allocator



Slab Allocator

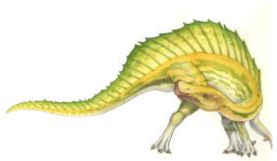
- Alternate strategy
- **Slab** is one or more physically contiguous pages
- **Cache** consists of one or more slabs
- Single cache for each unique kernel data structure
  - Each cache filled with **objects** – instantiations of the data structure
- When cache created, filled with objects marked as **`free`**
- When structures stored, objects marked as **`used`**
- If slab is full of used objects, next object allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



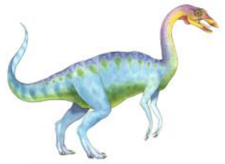
Other Considerations

- Prepaging
- Page size
- TLB reach
- Inverted page table
- Program structure
- I/O interlock and page locking



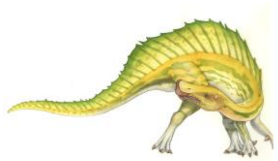
Prepaging

- To reduce the large number of page faults that occurs at process startup
- Prepage all or some of the pages a process will need, before they are referenced
- But if prepaged pages are unused, I/O and memory was wasted
- Assume *s* pages are prepaged and *α* of the pages is used
  - Question: is the cost of ***s * α*** save pages faults is greater or less than the cost of prepaging ***s * (1- α)*** unnecessary pages?
  - If ***α*** is close to 0 $\Rightarrow$ prepaging loses
  - If ***α*** is close to 1 $\Rightarrow$ prepaging wins



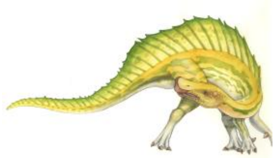
Page Size

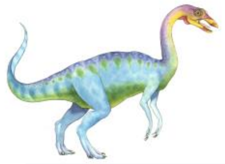
- Sometimes OS designers have a choice
  - Especially if running on custom-built CPU
- Page size selection must take into consideration:
  - Fragmentation
  - Page table size
  - **Resolution**
  - I/O overhead
  - Number of page faults
  - Locality
  - TLB size and effectiveness
- Always power of 2, usually in the range 2^{12} (4,096 bytes) to 2^{22} (4,194,304 bytes)
- On average, growing over time



TLB Reach

- TLB Reach - The amount of memory accessible from the TLB
- TLB Reach = (TLB Size) X (Page Size)
- Ideally, the working set of each process is stored in the TLB
  - Otherwise there is a high degree of page faults
- Increase the Page Size
  - This may lead to an increase in fragmentation as not all applications require a large page size
- Provide Multiple Page Sizes
  - This allows applications that require larger page sizes the opportunity to use them without an increase in fragmentation





Program Structure

- Program structure
  - `int[128,128] data;`
  - Each row is stored in one page
  - Program 1

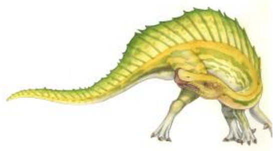
```
for (j = 0; j <128; j++)
     for (i = 0; i < 128; i++)
           data[i,j] = 0;
```

128 x 128 = 16,384 page faults

  - Program 2

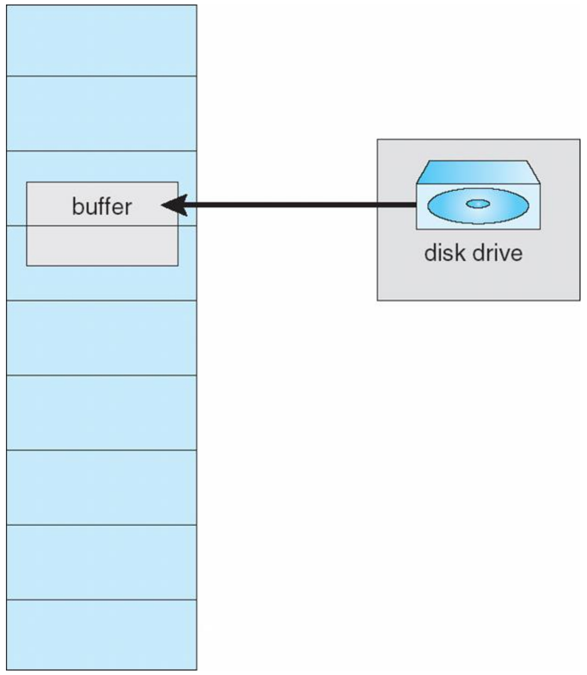
```
for (i = 0; i < 128; i++)
     for (j = 0; j < 128; j++)
           data[i,j] = 0;
```

128 page faults



I/O interlock

- **I/O Interlock** – Pages must sometimes be locked into memory
- Consider I/O - Pages that are used for copying a file from a device must be locked from being selected for eviction by a page replacement algorithm
- **Pinning** of pages to lock into memory



End of Chapter 10