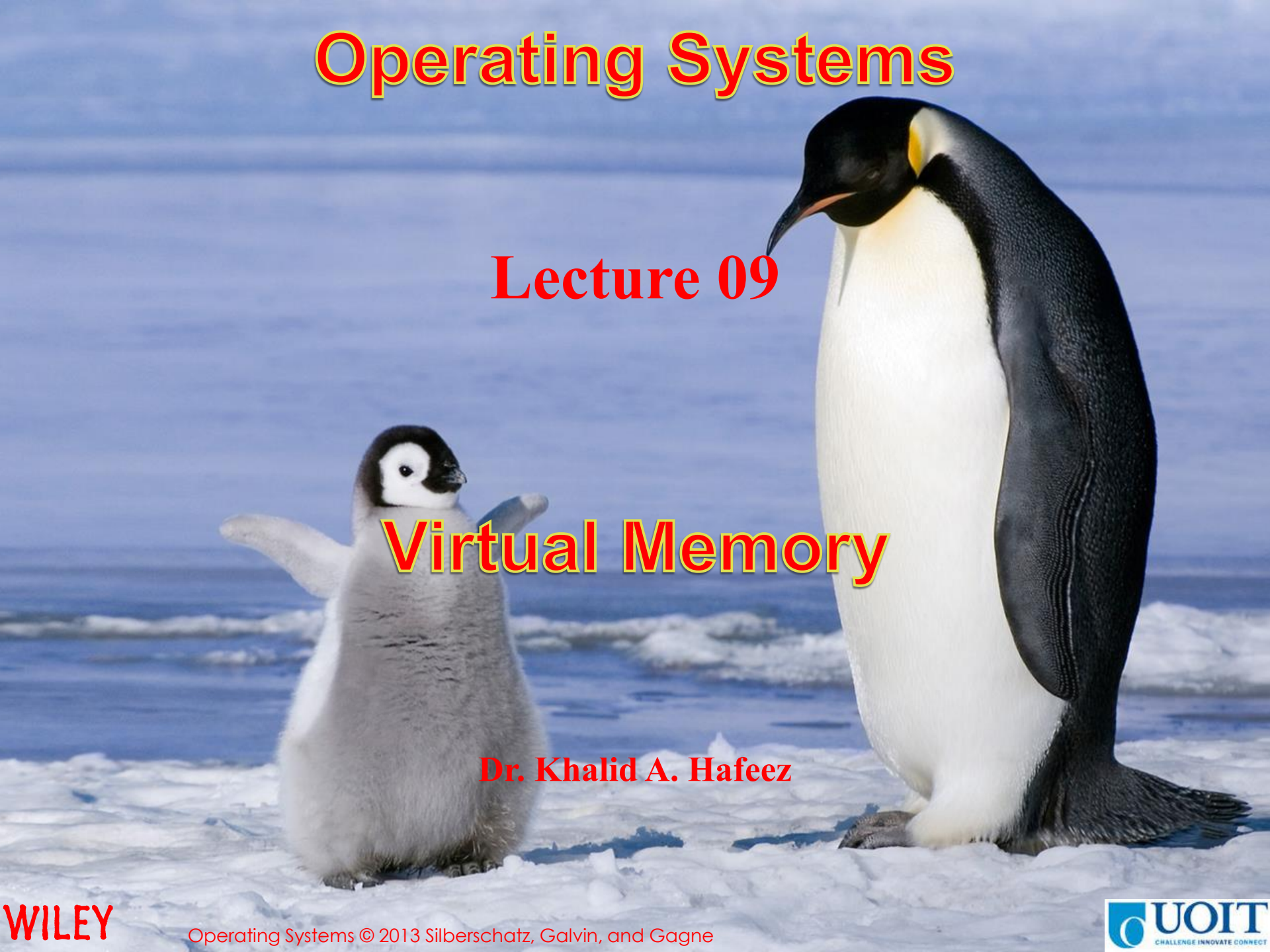
# Operating Systems

## Lecture 09

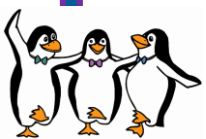
## Virtual Memory

Dr. Khalid A. Hafeez



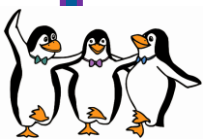
# Virtual Memory

- Background
- Demand Paging
- Copy-on-Write
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory
- Other Considerations
- Operating-System Examples



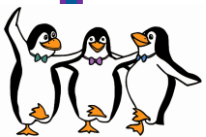
# Objectives

- To describe the benefits of a virtual memory system
- To explain the concepts of demand paging, page-replacement algorithms, and allocation of page frames
- To discuss the principle of the working-set model
- To examine the relationship between shared memory and memory-mapped files
- To explore how kernel memory is managed



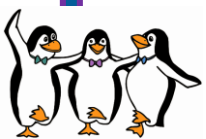
# Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures like arrays
- 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, so more programs can 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



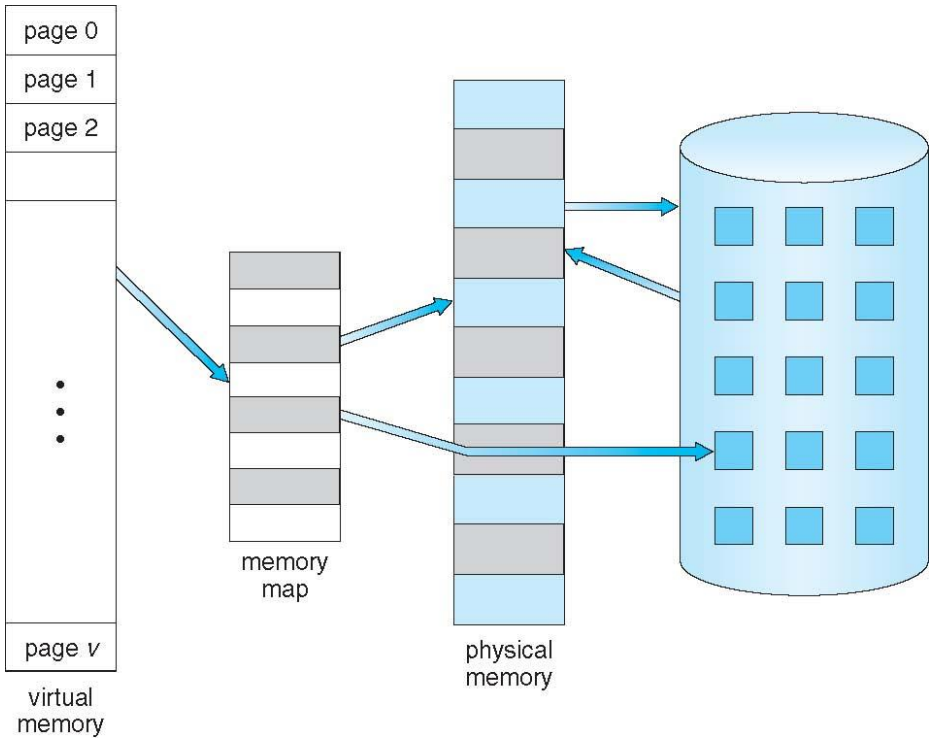
# Background

- The goal of all memory-management strategies is to keep many processes in memory simultaneously to allow multiprogramming.
  - But, they need the entire process to be in memory before it can be executed.
- **Virtual memory**
  - It is a technique that allows the execution of processes that are not completely in memory.
  - It **separates** logical memory as viewed by the user 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 by
    - More programs running concurrently
    - Less I/O needed to load or swap user programs into memory, so each user program would run faster.



# Background

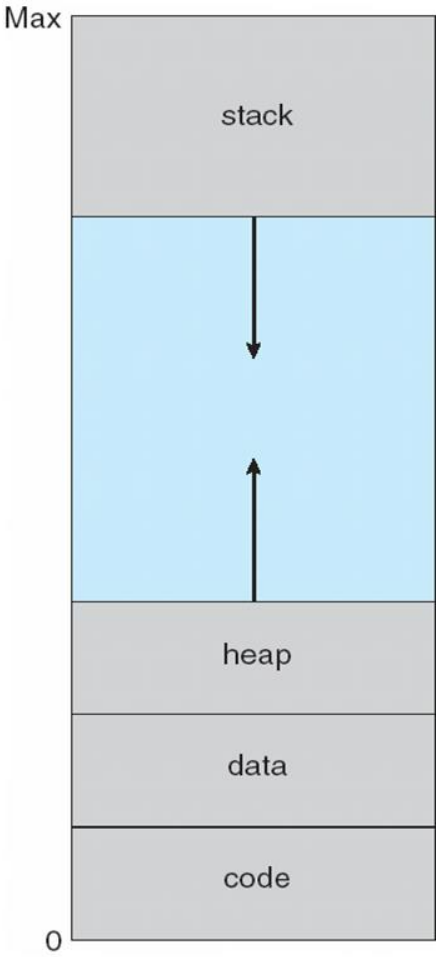
- **Virtual address space**: logical view of how a process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical addresses
- **Virtual memory can be implemented via:**
  - Demand paging
  - Demand segmentation

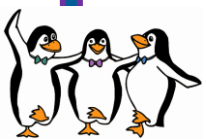



Virtual Memory That is Larger Than Physical Memory

# Background

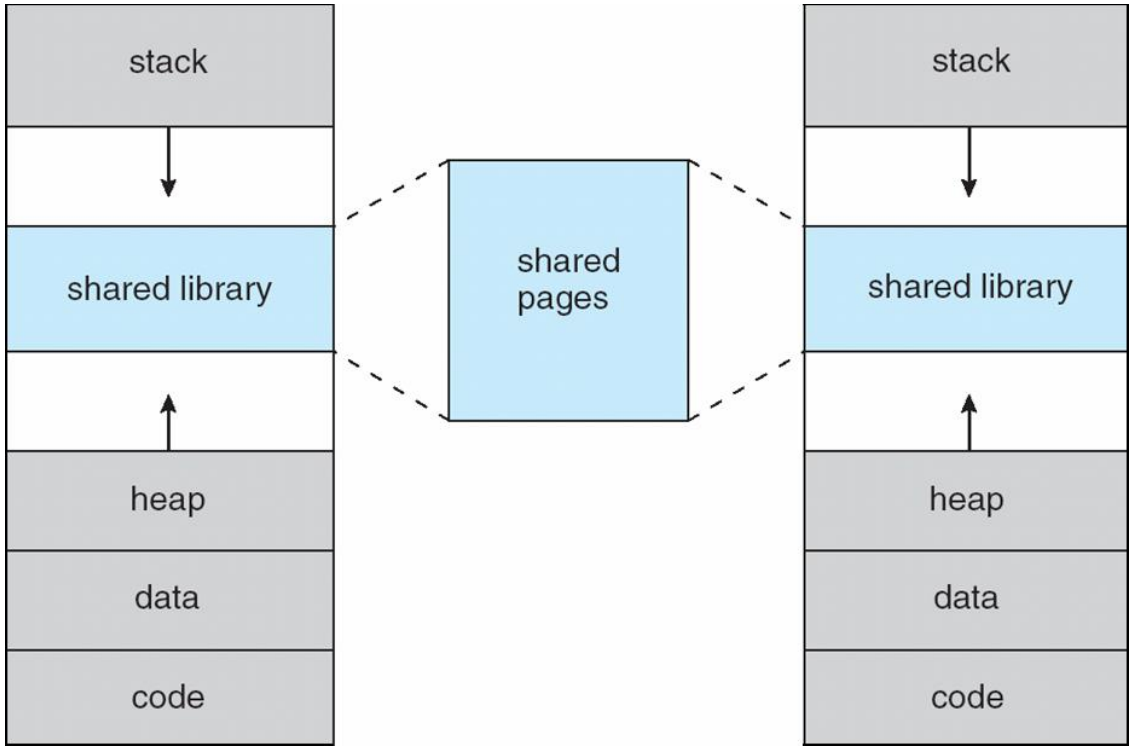
- Virtual-address Space
  - Usually design logical address space for stack to start at Max logical address and grow “down” while heap grows “up”
    - Maximizes address space use
    - Unused address space between the two is hole
      - No physical memory needed until heap or stack grows to a given 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(), thus speeding up process creation



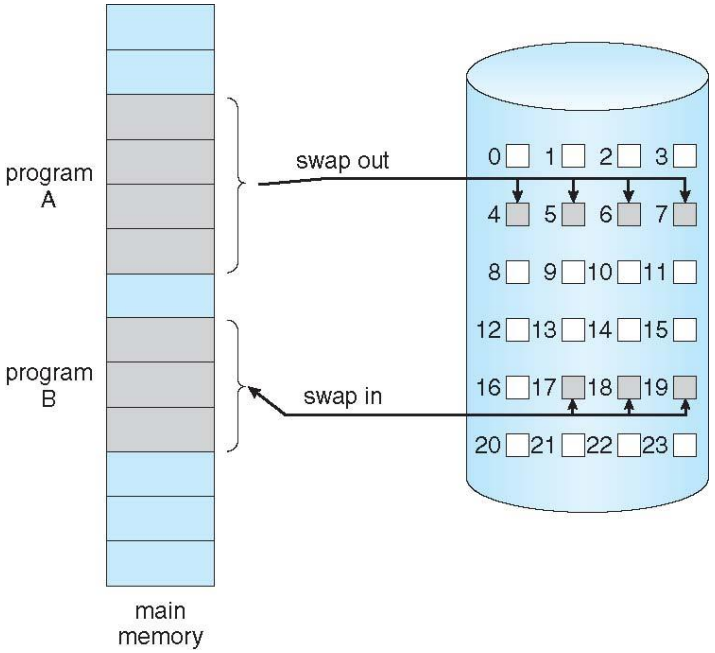
# Background

- Virtual-address Space
  - Shared Library Using Virtual Memory



# Demand Paging

- Demand Paging: Bring a page into memory **only** when it is needed
  - Less I/O needed
  - Less memory needed
  - Faster response
  - More users

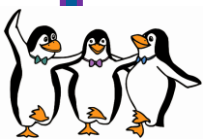



- Demand Paging is similar to a paging system with swapping
- When page is needed ⇒ reference to it:
  - If invalid reference (out of address space) ⇒ abort the process
  - If not-in-memory ⇒ bring page to memory
- We use a **Lazy swapper** which never swaps a page into memory unless that page will be needed
  - Swapper that deals with pages we need to call it a **pager**

# Demand Paging

## Basic Concepts

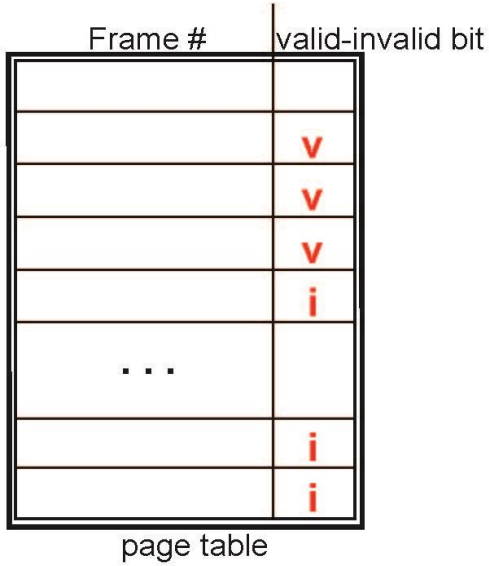
- When a process is to be swapped in, the pager **guesses** which pages will be used before it will be swapped out again
- Instead of swapping in a whole process, pager brings in only those needed pages into memory
- How to determine that set of pages?
  - Need new MMU functionality to implement demand 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



# Demand Paging

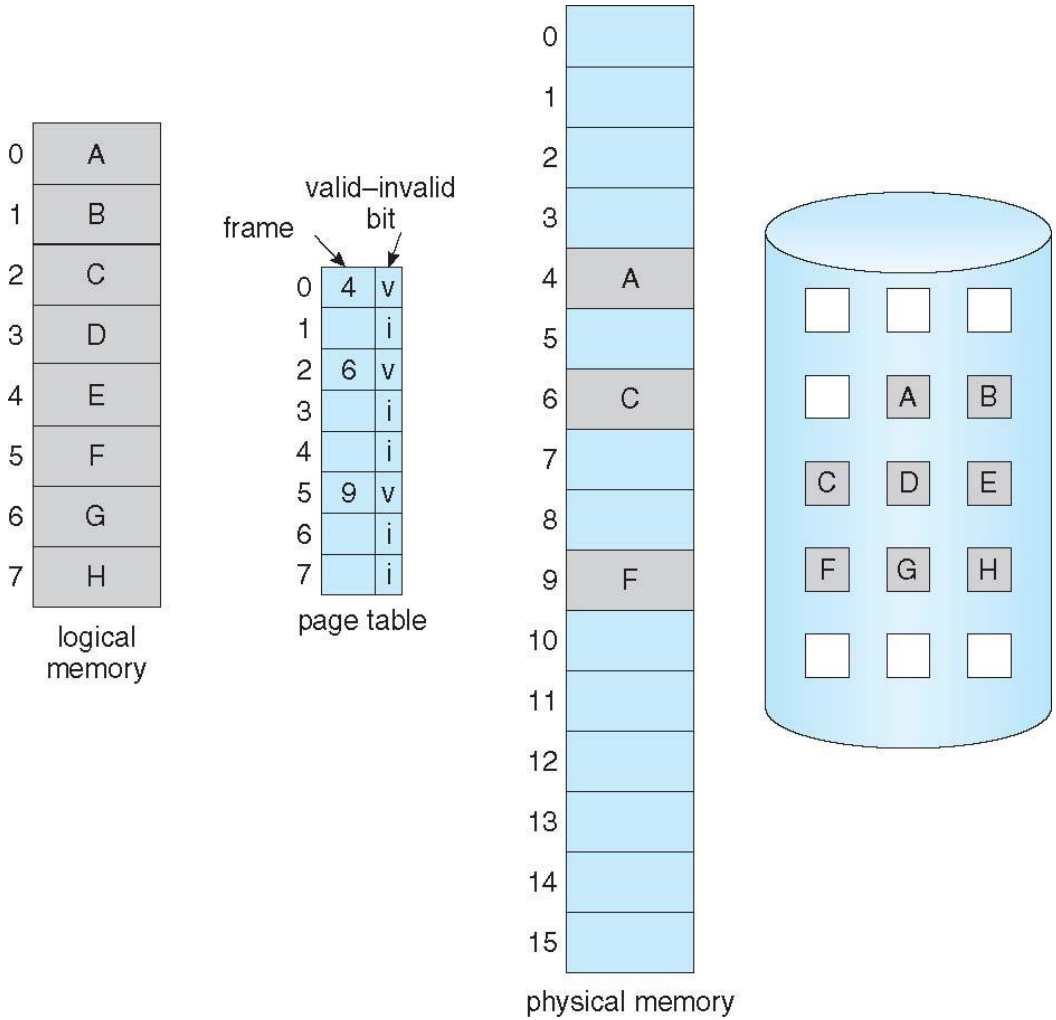
## Valid-Invalid Bit

- With each page table entry a valid–invalid bit is associated
  - ***v*** (1) ⇒ in-memory (**memory resident**), execution proceeds normally
  - ***i*** (0) ⇒ not-in-memory (page fault), causing a trap to the OS
- Initially valid–invalid bit is set to ***i*** (0) 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



# Demand Paging

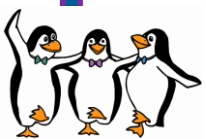
- Page table when some pages are not in main memory



# Demand Paging

## Page Fault

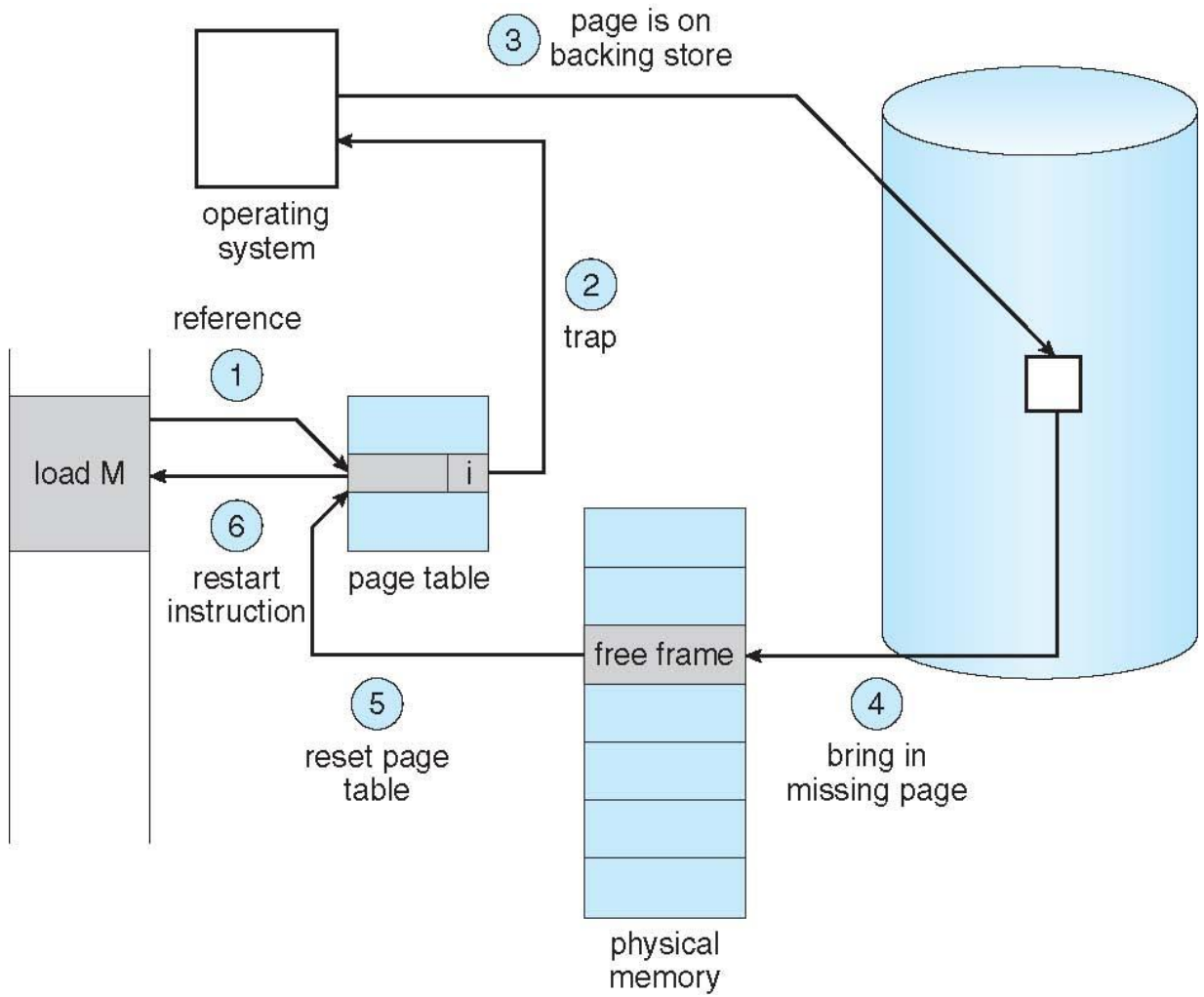
- If there is a reference to a page, first reference to that page will trap to operating system: **page fault**
- **Procedure for handling this page fault:**
  1. The OS looks first at an internal table (usually kept with the process control block (PCB)) to decide if it was:
     - Invalid reference ⇒ abort
     - Just not in memory ⇒ page it in
  2. Find a free frame
  3. Swap page into the new frame via scheduled disk operation
  4. Set the internal tables to indicate the page is now in memory
     Set validation bit **v =1**
  5. Restart the instruction that caused the page fault



# Demand Paging

## Page Fault

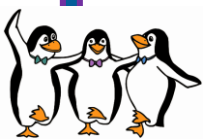
- Steps in Handling a Page Fault



# Demand Paging

## Aspects of Demand Paging

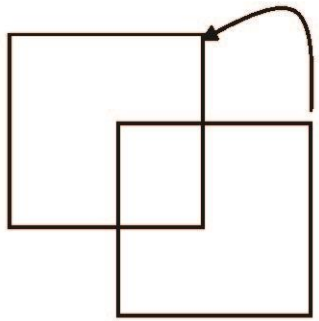
- Extreme case: we can start a process with *none* of its pages in memory
  - OS sets instruction pointer to the first instruction of the process, and found it non-memory-resident → page fault
  - This will repeat for every other pages on their first access
  - This is called **Pure demand paging**
- **Hardware support needed for demand paging**
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
  - The ability to restart any instruction after a page fault.

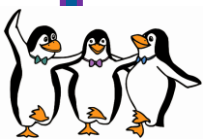


# Demand Paging

## Instruction Restart

- Consider an instruction that could access several different locations such as a block move that may overlap
  - The source block may have been modified, in which case we cannot simply restart the instruction.
  - **Two solutions:**
    - The microcode computes and attempts to access both ends of both blocks. If there will be a page fault, this will happen before anything is modified.
      Uses temporary registers to hold the values of overwritten locations.
  - This is the only architectural problem resulting from allowing demand paging



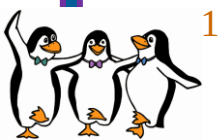


# Demand Paging

## Performance of Demand Paging

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

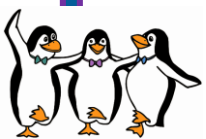


# Demand Paging

## Performance of Demand Paging

- Demand paging has significant effect on the computer's performance.
- **Three major components of the page-fault service time:**
  - Service the page fault interrupt: with careful coding this time can be reduced to just several hundred instructions needed (1-100μs)
  - Read in the page: lots of time (8ms)
  - Restart the process: again just a small amount of time (1-100μs)

- Assume that the Page Fault Rate $0 \le p \le 1$, we need to make sure $p \approx 0$
  - if $p = 0$ no page faults
  - if $p = 1$, every reference is a fault
- Thus, Effective Access Time (EAT) = $(1 - p)$ x memory access time +
  $p$ x (page fault overhead +
  swap page out +
  swap page in +
  restart the process)



# Demand Paging

## 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*

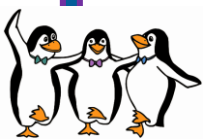
  *= 200 + p x 7,999,800*
- If one access out of *1,000* causes a page fault, then

  *EAT = 8.2 microseconds.*

  This is a slowdown by a factor of (*8.2 μs/200ns*) = ***40***
- If we want the performance degradation to be **< 10%**
  - *(200 + 7,999,800 x p)/(200ns) < 200x110%*

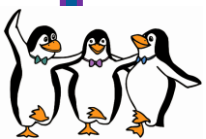
    *220 > 200 + 7,999,800 x p*

    *20 > 7,999,800 x p*
  - *p < .0000025*
  - *p < one page fault in every 400,000 memory accesses*



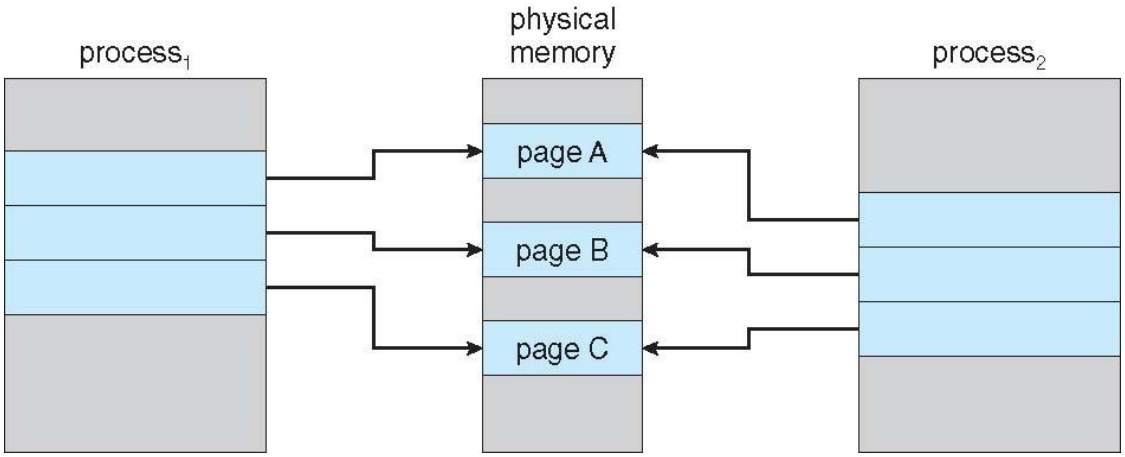
# Copy-on-Write

- Virtual memory allows other benefits during process creation:
  - **Copy-on-Write** (COW) is a technique that allows both parent and child processes to initially ***share*** the same pages in memory
    - If either process modifies a shared page, only then that page is 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 that always have *free* frames for *fast* demand page execution
    - Why zero-out a page before allocating it?
  - A variation of *fork()* system call, called *vfork()* is used to suspend the parent while the child is using the address space of the parent
    - Designed for the case when the child has to call *exec()* immediately so it does not write anything in the parent's address space
    - It is very efficient: it does not use Copy-on-Write (no extra memory use).

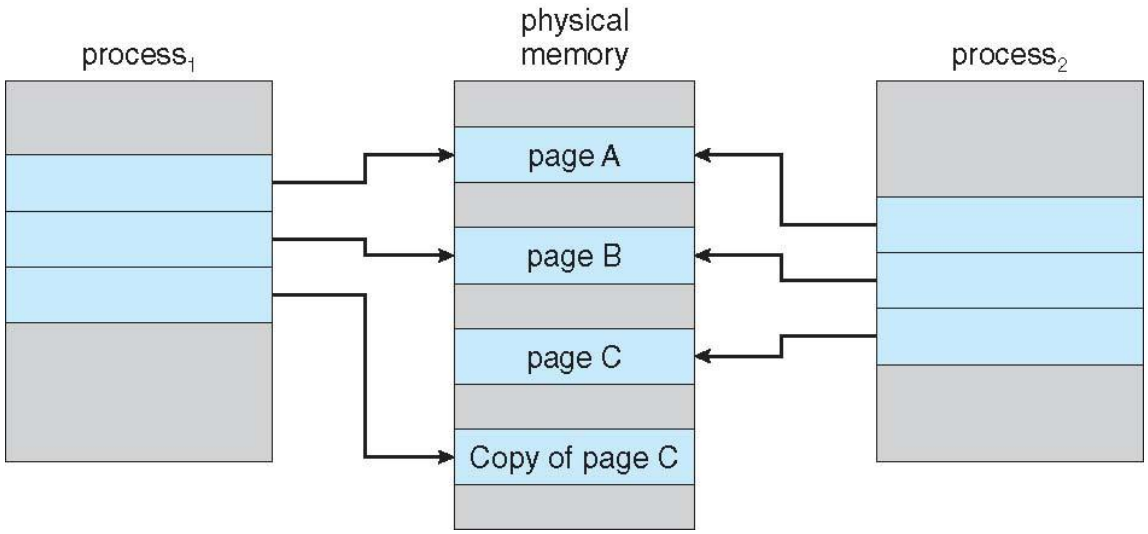


# Copy-on-Write

- **Before Process 1 Modifies Page C**

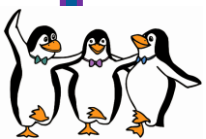



- **After Process 1 Modifies Page C**



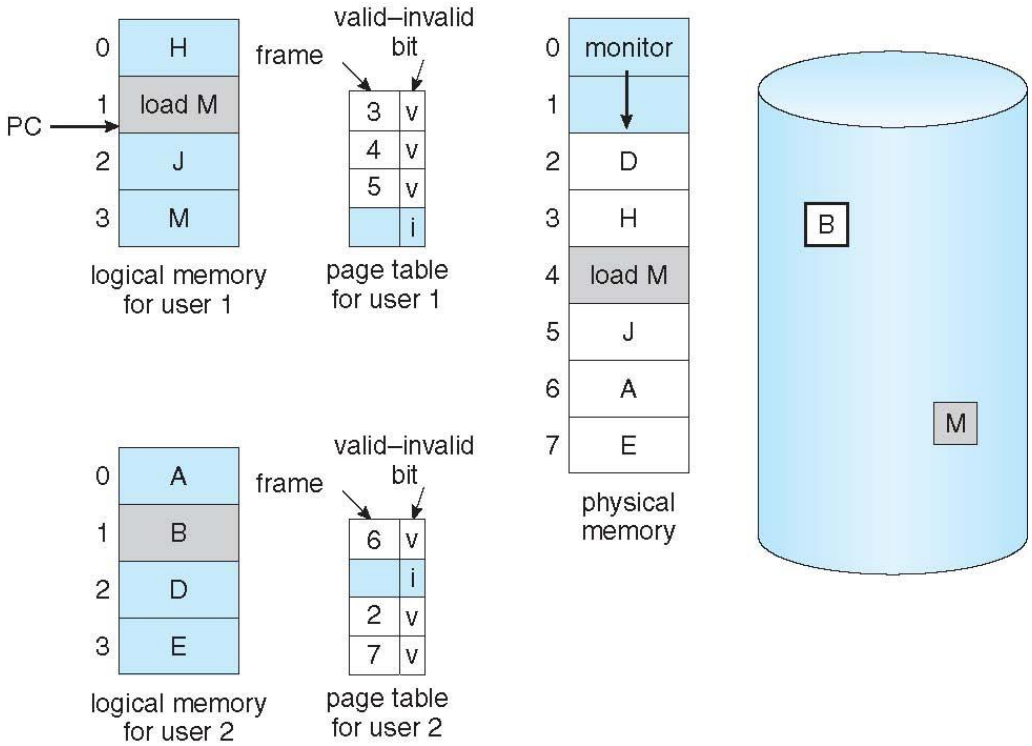
# Page Replacement

- What happens if there is **no free** frame?
  - Page frames can be used up by processes, kernel, I/O buffers, etc
  - How much to allocate to each?
  - While a user process is executing, a page fault occurs.
  - The operating system determines where the desired page is residing on the disk but then finds that there are no free frames on the free-frame list; all memory is in use
  - Page replacement: find a 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

- What happens if there is **no free** frame?
  - 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

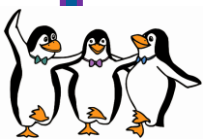


# Page Replacement

## Basic Page Replacement

1. Find the location of the desired page on disk
2. Find a free frame:
   a) If there is a free frame, use it
   b) If there is no free frame, use a page replacement algorithm to select a **victim frame**
   c) 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

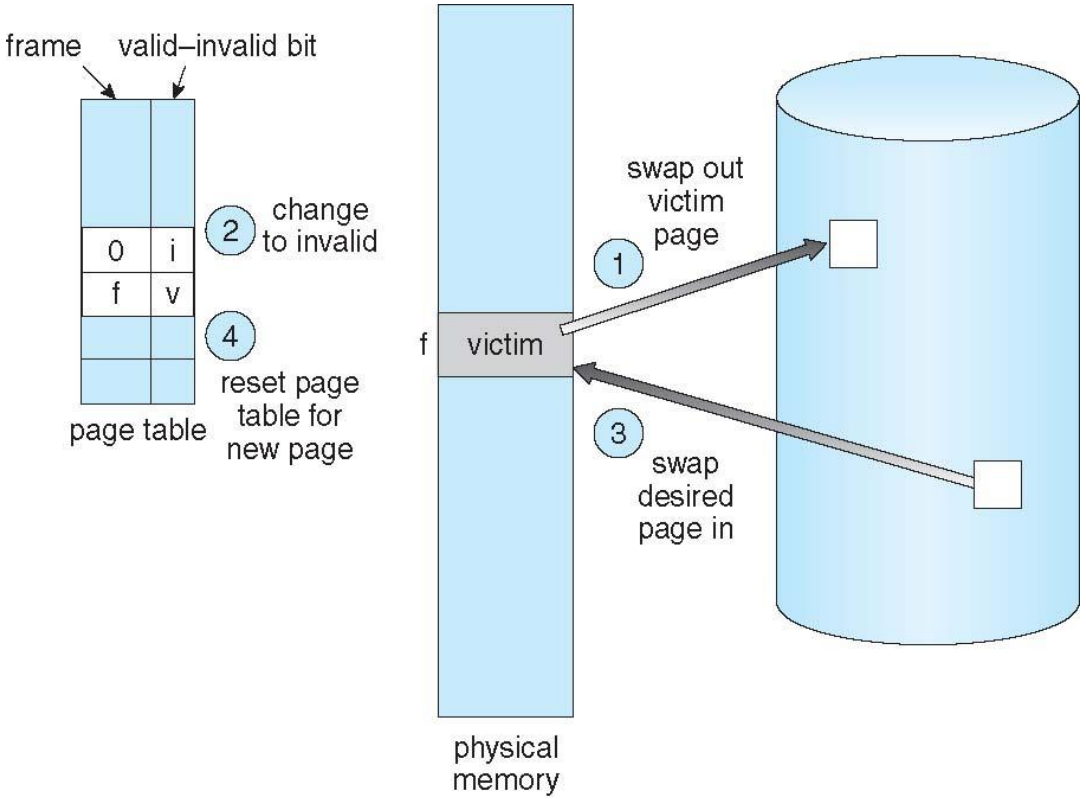
- Note now potentially we need 2 page transfers for a page fault → increasing EAT



# Page Replacement

## Basic Page Replacement

- Process accessed to page “f”, but it was not in memory (invalid page)
- No free-frame exists → MMU subsystem searches for a victim to swap out
  - page “o” is chosen as victim and is swapped out (table is updated)
  - page “f” is swapped in to memory → page table is updated: “f” is valid
- Process is restarted to reference page “f”



# Page Replacement

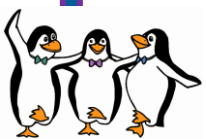
## Page and Frame Replacement Algorithms

- **We need two algorithms to implement demand paging:**
  1. **Frame-allocation algorithm** determines
     - How many frames to give each process and which frames to replace
  2. **Page-replacement algorithm**
     - We want lowest page-fault rate on both first access and re-access

- The algorithm can be evaluated 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 memory
- In all our examples, the **reference string** of referenced page numbers is

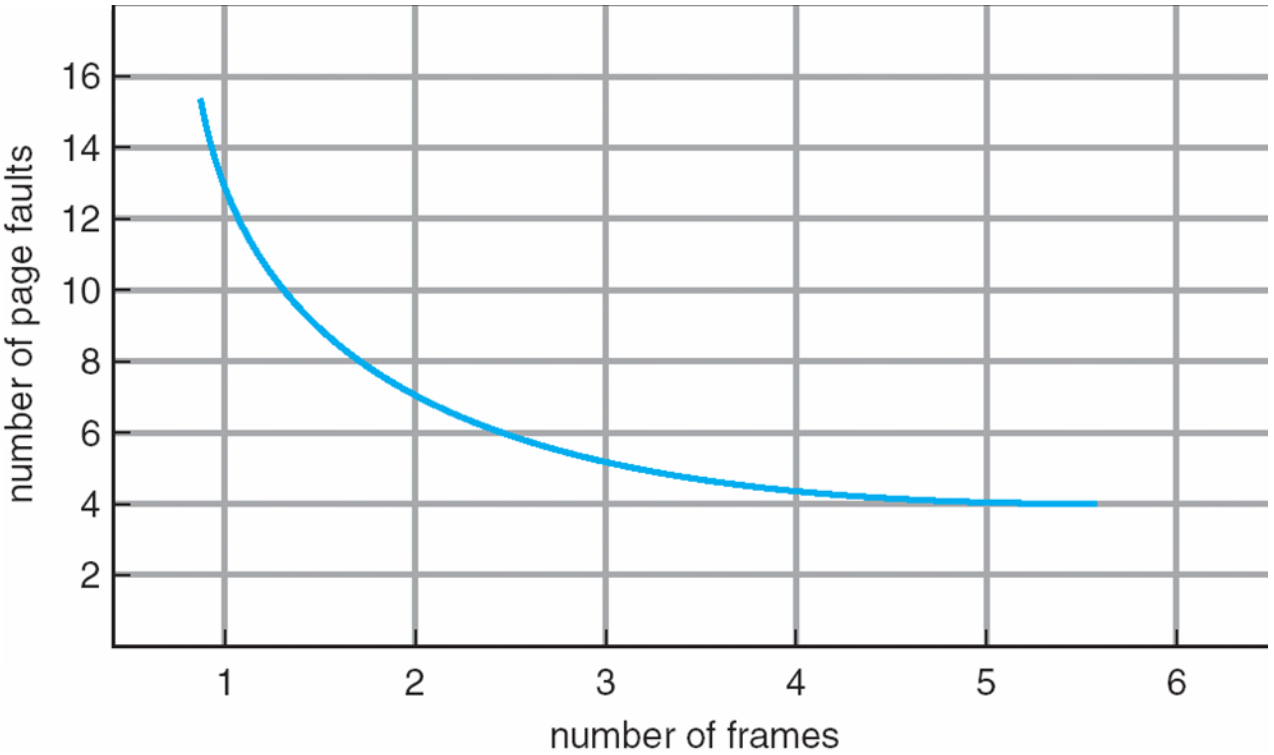
  **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**

  for a memory with **three** frames.



# Page Replacement

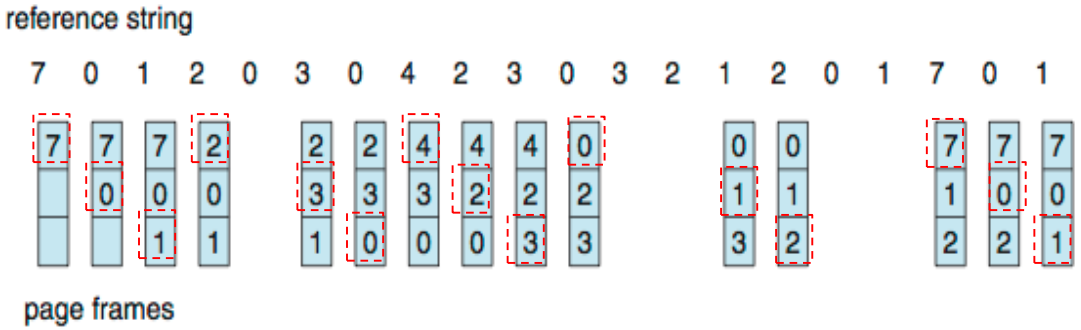
- Page and Frame Replacement Algorithms
  - Graph of Page Faults Versus The Number of Frames
    - As the number of frames increases, the number of page faults drops to some minimal level.



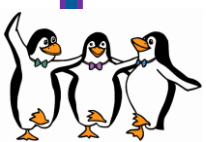
# Page Replacement

## First-In-First-Out (FIFO) Algorithm

- We can create a FIFO queue to hold all page numbers in memory.
  - Replace the page at the head of the queue.
  - Insert new page at the tail of the queue.
- **Example**: 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)
    - There are 15 page faults

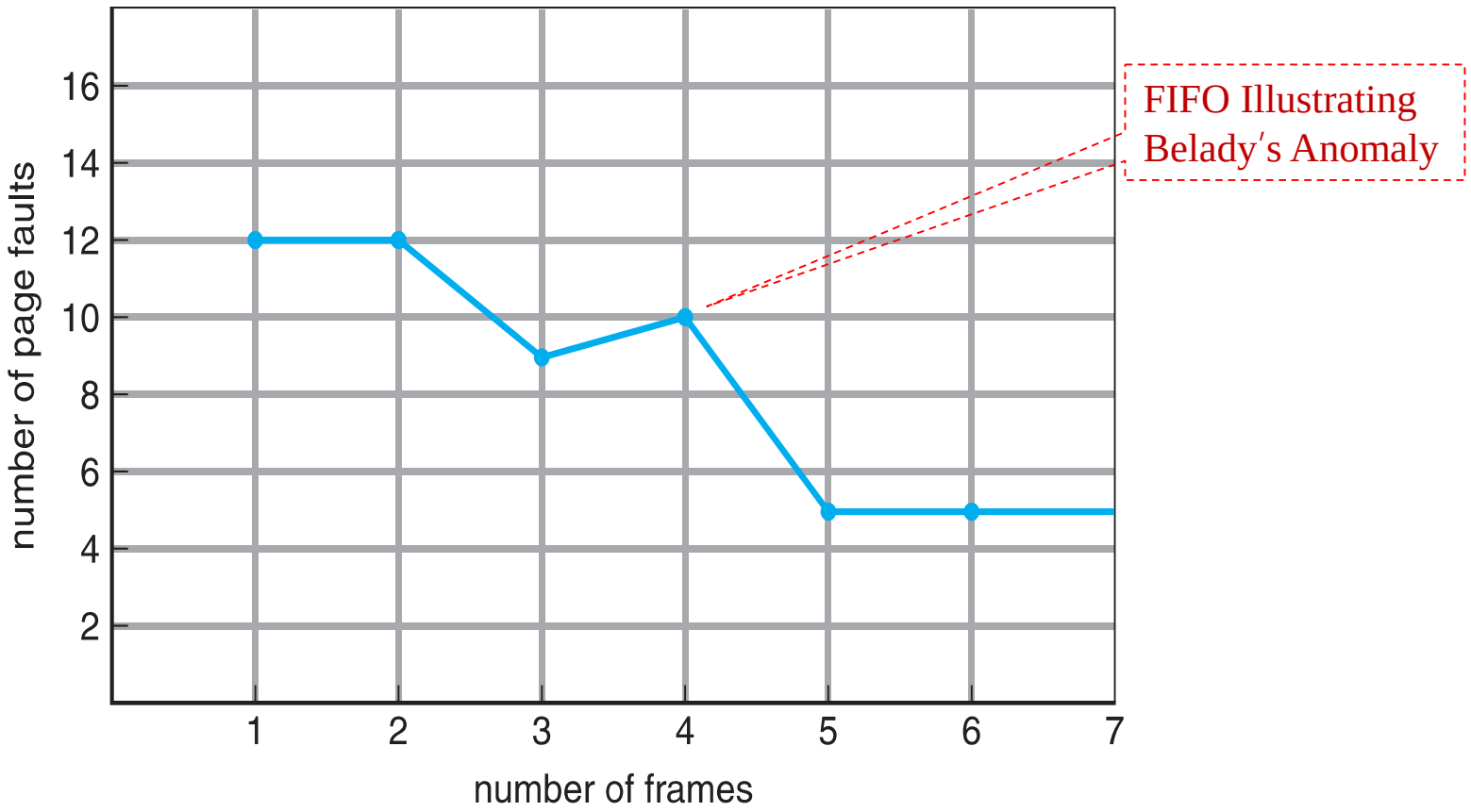



- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5 with 4 frames
  - Adding more frames may cause more page faults!
    - **Belady's Anomaly:** for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.



# Page Replacement

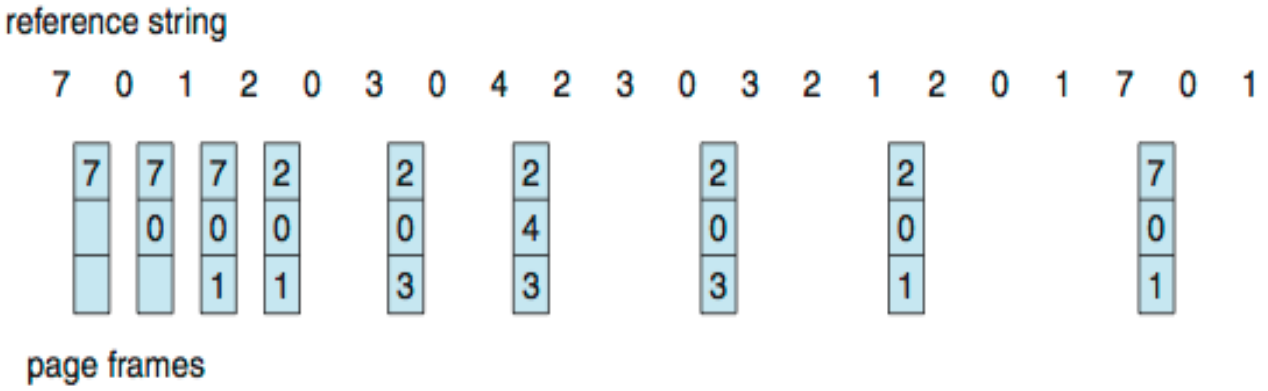
- First-In-First-Out (FIFO) Algorithm
  - **Belady's Anomaly:** for some page-replacement algorithms, the page-fault rate may increase as the number of allocated frames increases.
  - Consider 1,2,3,4,1,2,5,1,2,3,4,5

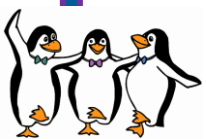


# Page Replacement

## Optimal Page Replacement Algorithm

- Replace the page that will not be used for longest period of time
  - It does not suffer from Belady's Anomaly
  - For the previous example: 9 faults only
    - Since the first 3 page-faults are common to all algorithms, OPR is twice better than FIFO
- How do you know this?
  - Can't read the future
- Used only for comparisons: measuring how well the new algorithm performs

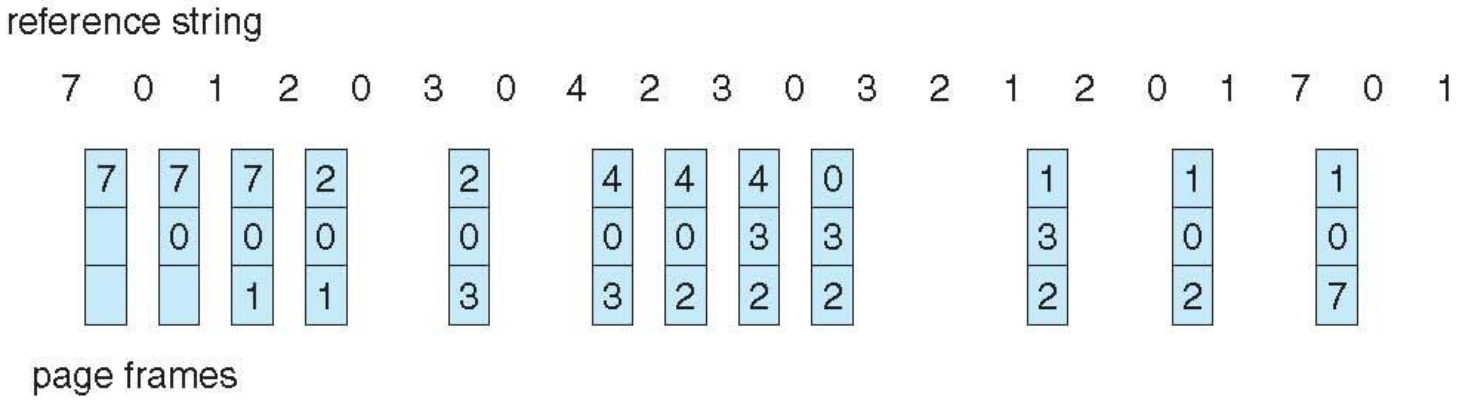


# Page Replacement

## Least Recently Used (LRU) Algorithm

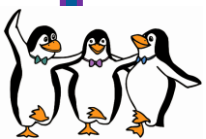
- Use the past knowledge rather than future
- Replace page that has not been used for the longest period of time
- Associate the time of the last use with each page




- With LRU: we have 12 faults → better than FIFO but worse than OPT
- Generally it is a good algorithm and is frequently used
- But how to implement the LRU?

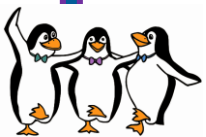
# Page Replacement

- Least Recently Used (LRU) Algorithm
  - Counter implementation
    - Every page entry has a **counter**; every time the 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 the smallest value
    - Disadvantages:
      - It needs to search through the table to find the LRU page
      - It needs to write to memory (to the time-of-use field in the page table) for each memory access



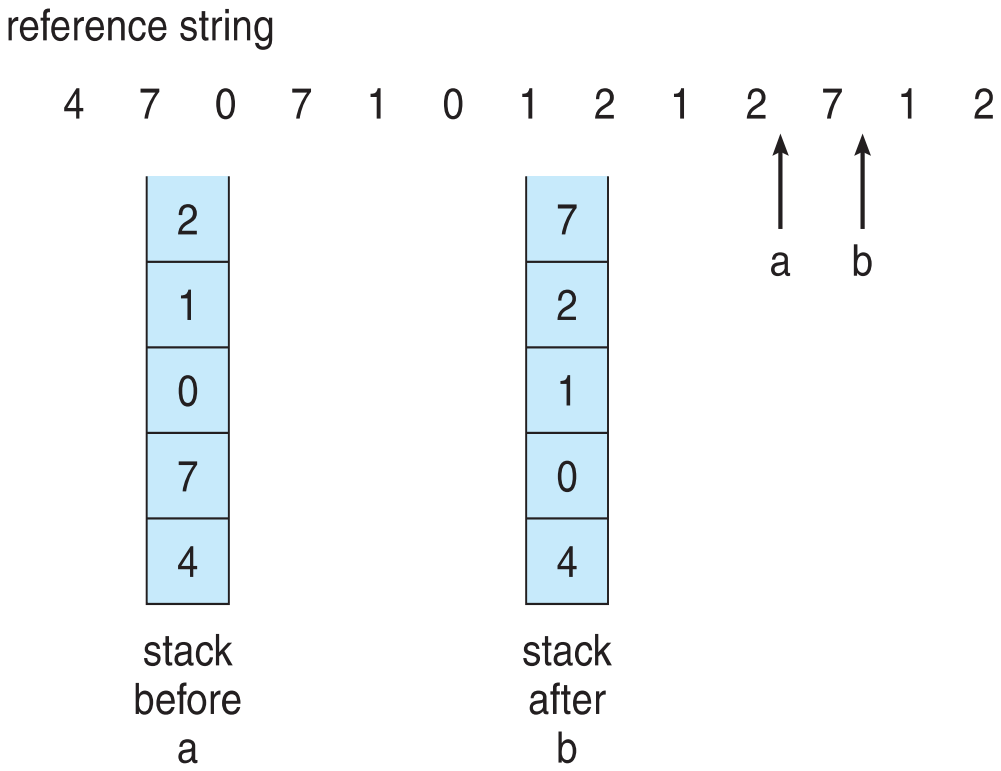
# Page Replacement

- Least Recently Used (LRU) Algorithm
  - Stack implementation
    - Keep a stack of page numbers in a **double link list** form:
    - If page is referenced:
      - Move it to the top (head of the list)
        - » It may require 6 pointers to be changed
    - But each update is more expensive, but there is no search for a replacement
      - The tail pointer points to the bottom of the stack, which is the LRU page.
  - LRU and OPT are cases of **stack algorithms** that don′t have Belady′s Anomaly



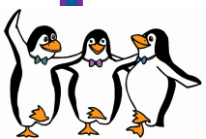
# Page Replacement

- Least Recently Used (LRU) Algorithm
  - Use of a stack to record most recent page references



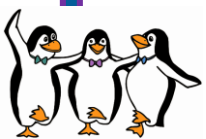
# Page Replacement

- LRU-Approximation Page Replacement Algorithms
  - LRU needs special hardware and still slow
    - Not many computer systems provide hardware support for true LRU page replacement (they use FIFO)
    - Many systems provide some help, in the form of a *reference bit*.



# Page Replacement

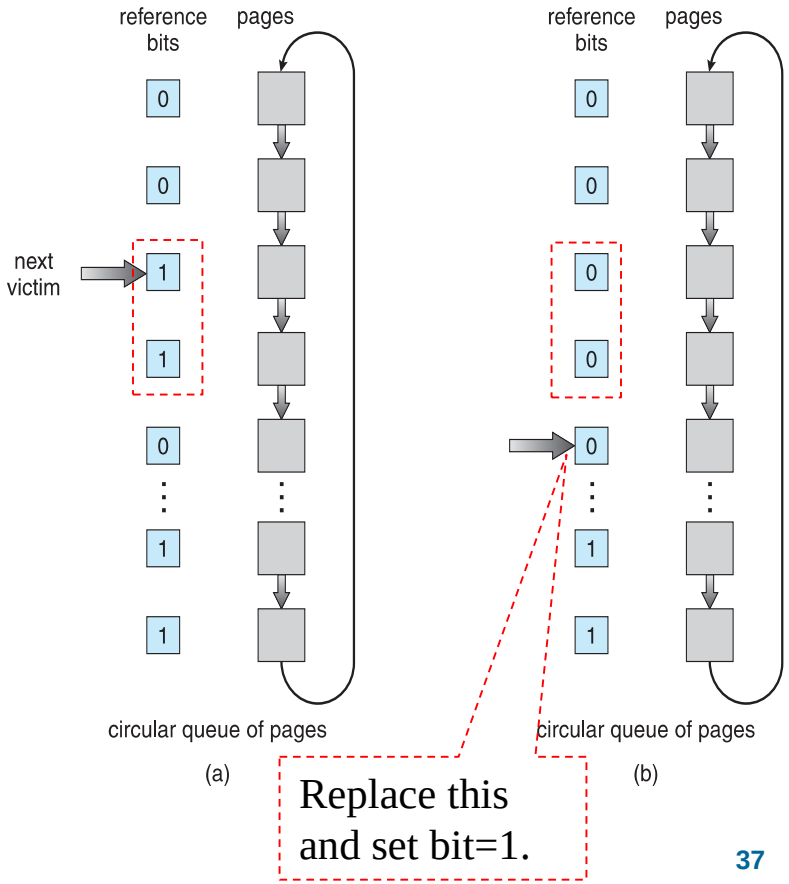
- LRU-Approximation Page Replacement Algorithms
  - **Additional-Reference-Bits Algorithm**
    - With each page in the page table, associate a reference bit, initially = 0
    - When a page is referenced, the bit set to 1
    - Replace any page with reference bit = 0 (if one exists)
      - We do not know the order, (it is approximation)



# Page Replacement

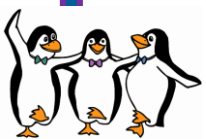
- LRU-Approximation Page Replacement Algorithms
  - **Second-chance Algorithm**
    - Generally it is a FIFO, plus hardware-provided a reference bit
    - It is called **Clock** algorithm (the pointer points to the page to be replaced next)replace this and set bit=1

- If a page to be replaced has:
  - Reference bit = 0, then replace it
  - Reference bit = 1 then:
    - » Set reference bit 0, leave the page in memory
    - » Replace the next page, subject to the same rules
      - » The new page is inserted in the circular queue in that position.



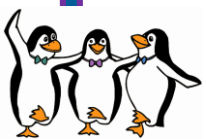
# Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes?
  - Keep 3 free frames reserved on the free-frame list at all times.
    - Thus, when a page fault occurs, there is a free frame available to page into. While the page swap is taking place, a replacement can be selected to be written to the disk as the process continues to execute.
  - Each process needs a ***minimum*** number of frames
    - Example: IBM 370 – 6 pages to handle MVC MOVE instruction:
      - instruction is 6 bytes, might span over 2 pages
      - 2 pages to handle *from* (assume indirect addressing)
      - 2 pages to handle *to* (assume indirect addressing)
    - Thus, we must place a **limit** on the levels of indirection
  - Thus, the minimum number of frames per process is defined by the architecture,
  - The maximum number is defined by the amount of available physical memory.



# Allocation of Frames

- How do we allocate the fixed amount of free memory among the various processes?
  - Two major allocation schemes
    - fixed allocation
    - priority allocation
  - Many variations



# Allocation of Frames

## Fixed Allocation

- **Equal allocation**: For example, if there are 103 frames (after allocating frames for the OS) and 5 processes, give each process 20 frames
  - Keep some (3) as free frame buffer pool

- **Proportional allocation:** Allocate according to the size of a process
  - Dynamic as degree of multiprogramming, process sizes change

  - $s_i$ = size of process $p_i$
  - $S = \sum s_i$
  - $m$ = total number of frames
  - $a_i$ = allocation for $p_i = \frac{s_i}{S} \times m$

*example*:

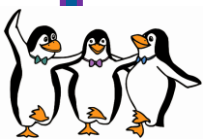
$m = 64$

$s_1 = 10$

$s_2 = 127$

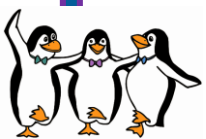
$a_1 = \frac{10}{137} \times 62 \approx 4$

$a_2 = \frac{127}{137} \times 62 \approx 57$



# Allocation of Frames

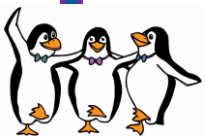
- Fixed Allocation
  - **Priority Allocation**
    - Use a proportional allocation scheme using priorities rather than the size of the process or on a combination of size and priority.
    - If process $P_i$ generates a page fault,
      - select for replacement one of its frames
      - select for replacement a frame from a process with lower priority number



# Allocation of Frames

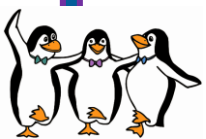
## Global vs. Local Allocation

- Another important factor in the way frames are allocated to the various processes is page replacement
  - **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 it is more common
  - **Local replacement**: each process selects from only its own set of allocated frames
    - More consistent per-process performance
    - But possibly underutilized memory



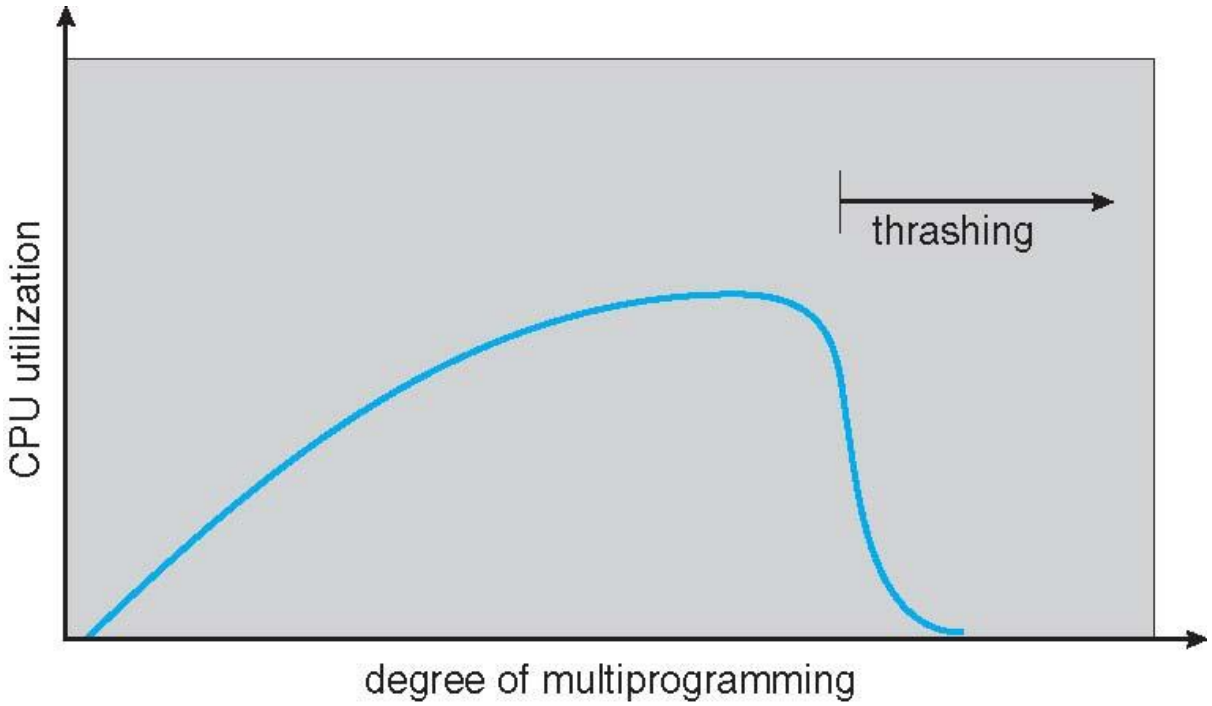
# Thrashing

- If the number of frames allocated to a low-priority process falls below the minimum number required by the computer architecture, we must suspend that process's execution and free all of its frames
- If a process does not have "enough" pages, the page-fault rate is very high:
  - It will page fault to get a new page
  - The new page replaces an existing frame, which will be needed back very quickly
  - This leads to low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
      - The OS will add a new process to the system (worsen the situation)
  - This high paging activity is called **thrashing**:
    - **Thrashing**: a process is busy swapping pages in and out and spending more time paging than executing



# Thrashing

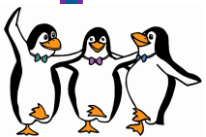
- As the degree of multiprogramming increases, CPU utilization also increases slowly, until a maximum is reached.
- If the degree of multiprogramming is increased even further, thrashing sets in, and CPU utilization drops sharply.
  - At this point, to increase CPU utilization and stop thrashing, we must actually **decrease** the degree of multiprogramming.



# Thrashing

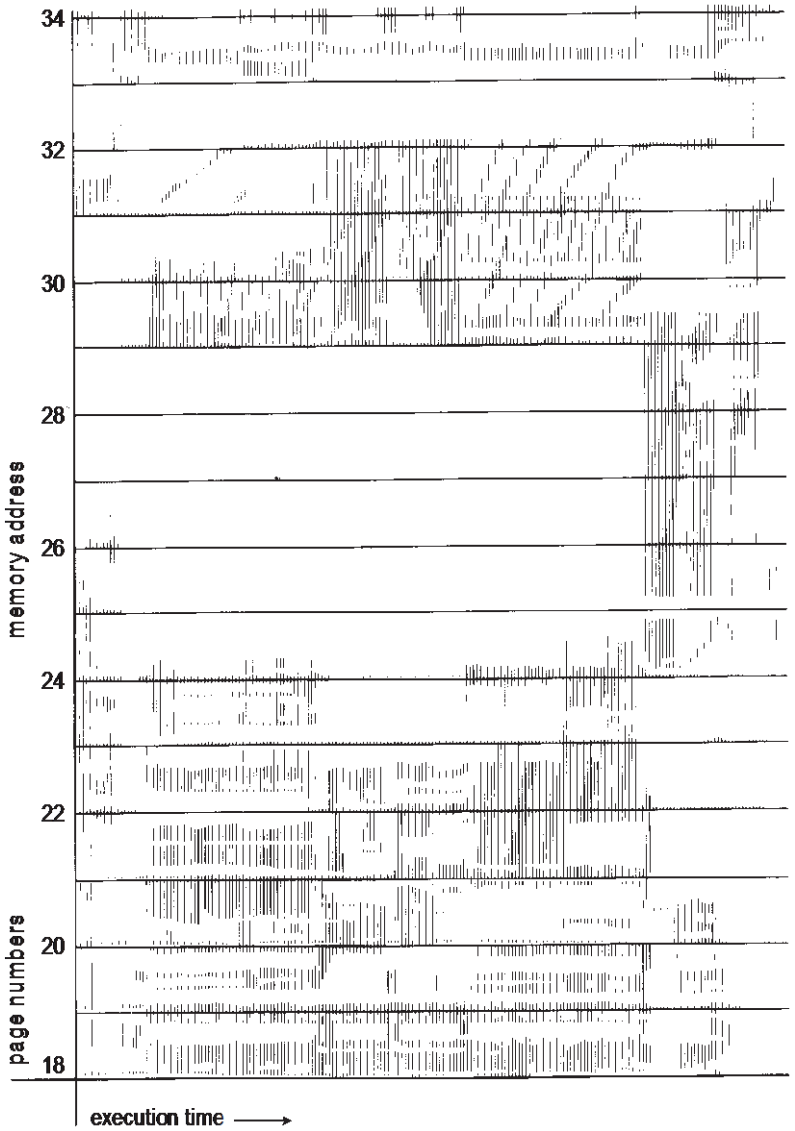
## Demand Paging and Thrashing

- To prevent thrashing, we must provide a process with as many frames as it needs.
- But how do we know how many frames it "needs"?
  - The **working-set strategy** that determines how many frames a process is actually using.
    - This approach defines the locality model of process execution.
      - A **locality** is a set of pages that are actively used together
    - Process migrates from one locality to another
    - Localities may overlap
- Why does thrashing occur?
  - Because Σ size of locality > total memory size
    - Limit effects by using local or priority page replacement



# Thrashing

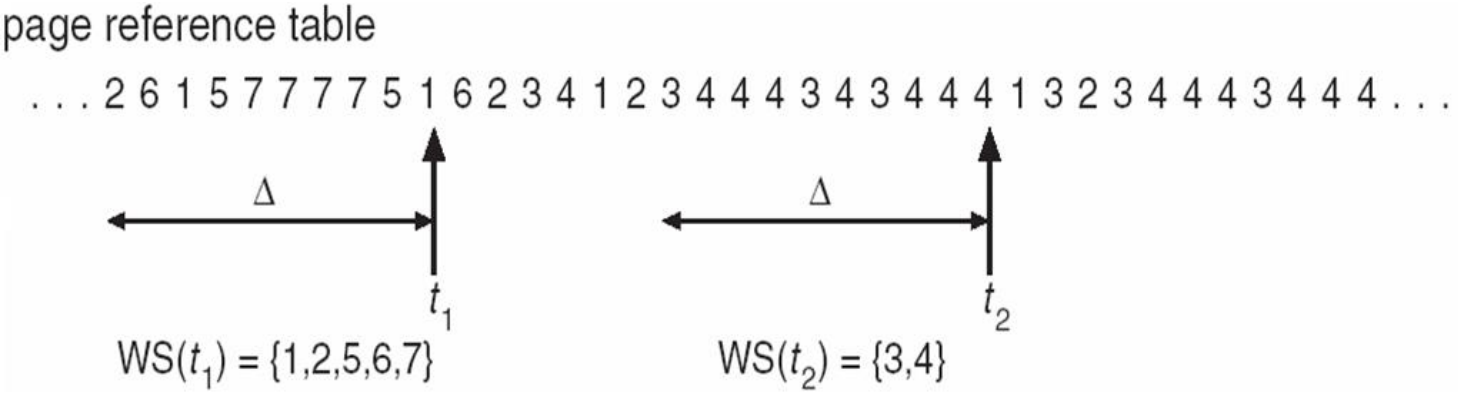
- Locality In A Memory-Reference Pattern



# Thrashing

## Working-Set Model

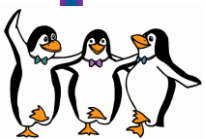
- The working-set model is based on the assumption of locality.
  - This model uses a parameter, $\Delta$, to define the working-set window.
  - $\Delta \equiv$ working-set window $\equiv$ fixed number of most recent page references
    Example: 10,000 references
  - Example: if $\Delta = 10$ memory references



# Thrashing

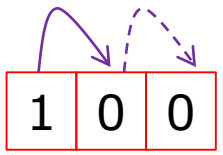
## Working-Set Model

- Define $WSS_i$ (working-set size of Process $P_i$) = total number of pages referenced in the most recent $\Delta$ (varies in time)
  - if $\Delta$ too small will not encompass entire locality
  - if $\Delta$ too large will encompass several localities
  - if $\Delta = \infty \Rightarrow$ will encompass entire program
- Let $D = \Sigma\ WSS_i \equiv$ total demand frames from all processes (it is approximation of locality)
  - if $D > m \Rightarrow$ Thrashing
  - Policy if $D >$ m, then suspend or swap out one of the processes

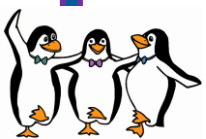


# Thrashing

- Working-Set Model
  - Keeping Track of the Working Set
    - We can approximate the working-set model with a fixed-interval timer interrupt and a reference bit
    - Example: assume that $\Delta$ = 10,000 references
      - Assume that the timer interrupts after every 5000 references
      - Keep in memory **2** bits for each page
      - Whenever a timer interrupts, we first copy and then sets the values of all reference bits to 0 for each page



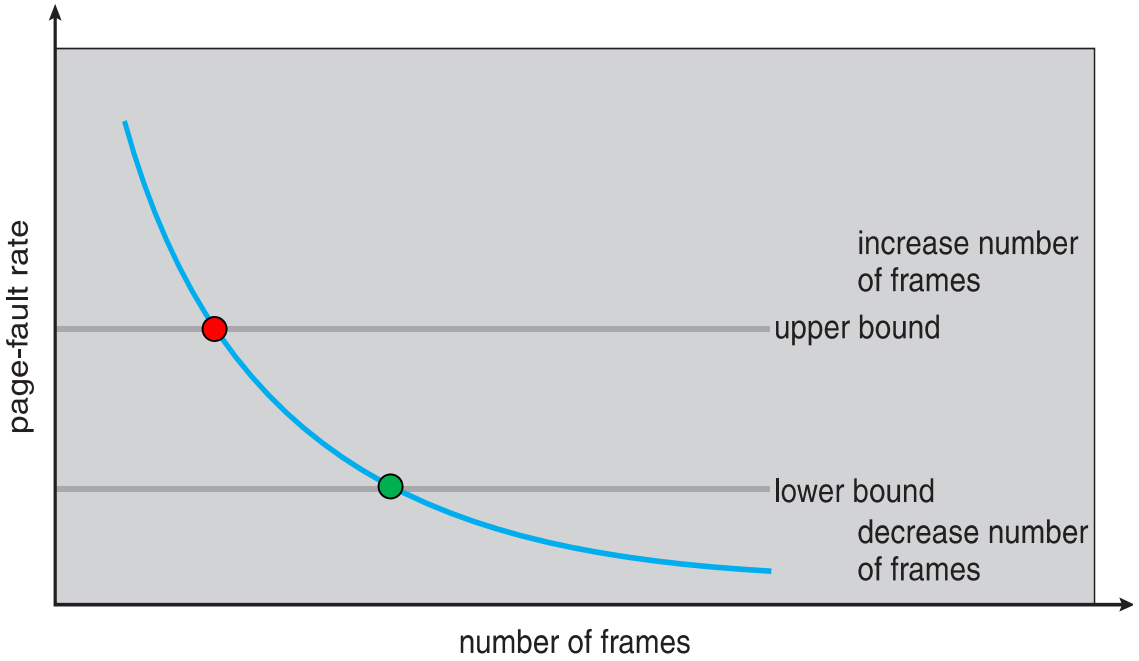

      - If one of these three bits in memory = 1 $\Rightarrow$ page should be in the working set of that process
      - If all three bits for a page are 0, then that page should be removed from the working set
        - » It is not been used within the last 10,000 to 15,000 references.



# Thrashing

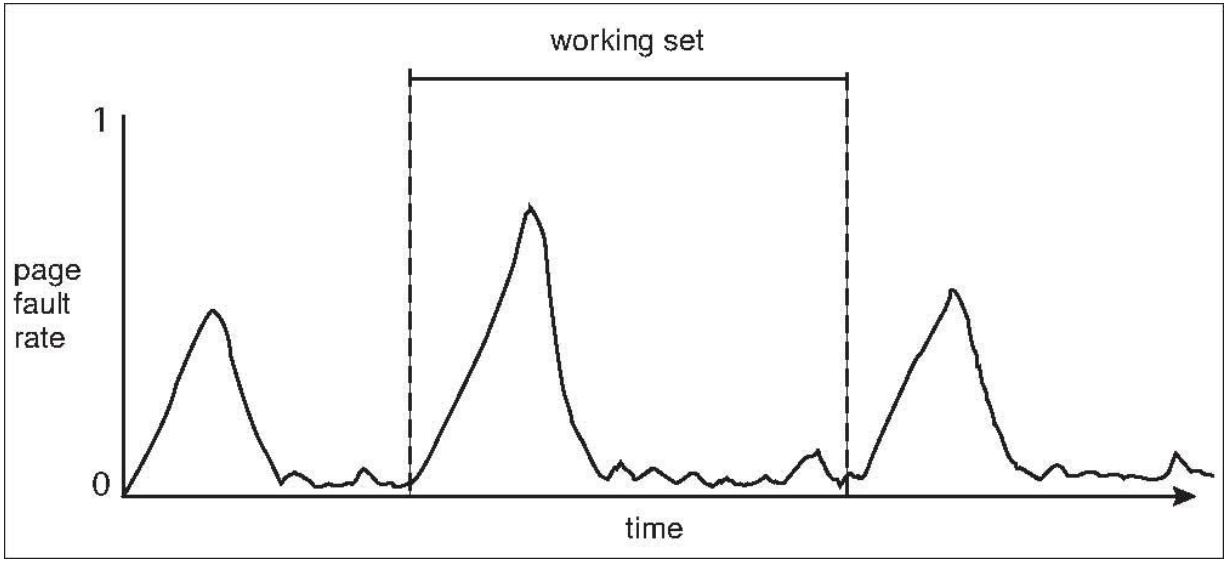
## Page-Fault Frequency (PFF)

- It is a more direct approach than the working-set model
- Establish "acceptable" **page-fault frequency** (**PFF**) rate and use local replacement policy
  - If actual page-fault rate < lower limit, **remove** a frame from the process.
  - If actual page-fault rate > upper limit, **allocate** the process another frame
    - If no free frames are available, we must select a victim process and swap it out to backing store.



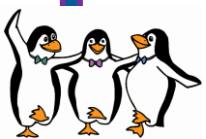
# Thrashing

- 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
    - A peak in the page-fault rate occurs when we begin demand-paging a new locality.
    - However, once the working set of this new locality is in memory, the page-fault rate falls.



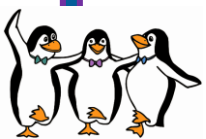
# Memory-Mapped Files

- A file on disk can be used by the system calls *open(), read(), and write().*
- To save time, we can use the virtual memory to treat file I/O as routine memory accesses. It is know as **memory-mapped file I/O**
  - Memory-mapped file I/O **maps** a disk block to page(s) in memory
  - A file is initially read using the demand paging
    - A page-sized portion of the file is read from the file system into a physical page
    - Subsequent reads/writes to/from the file are treated as ordinary memory accesses
  - Simplifies and speeds file access by driving file I/O through memory rather than read() and write() system calls
  - Also allows several processes to map the same file allowing the pages in memory to be shared
  - But when does written data make it to disk?
    - Periodically and / or at file close() time
    - For example, when the pager scans for dirty pages

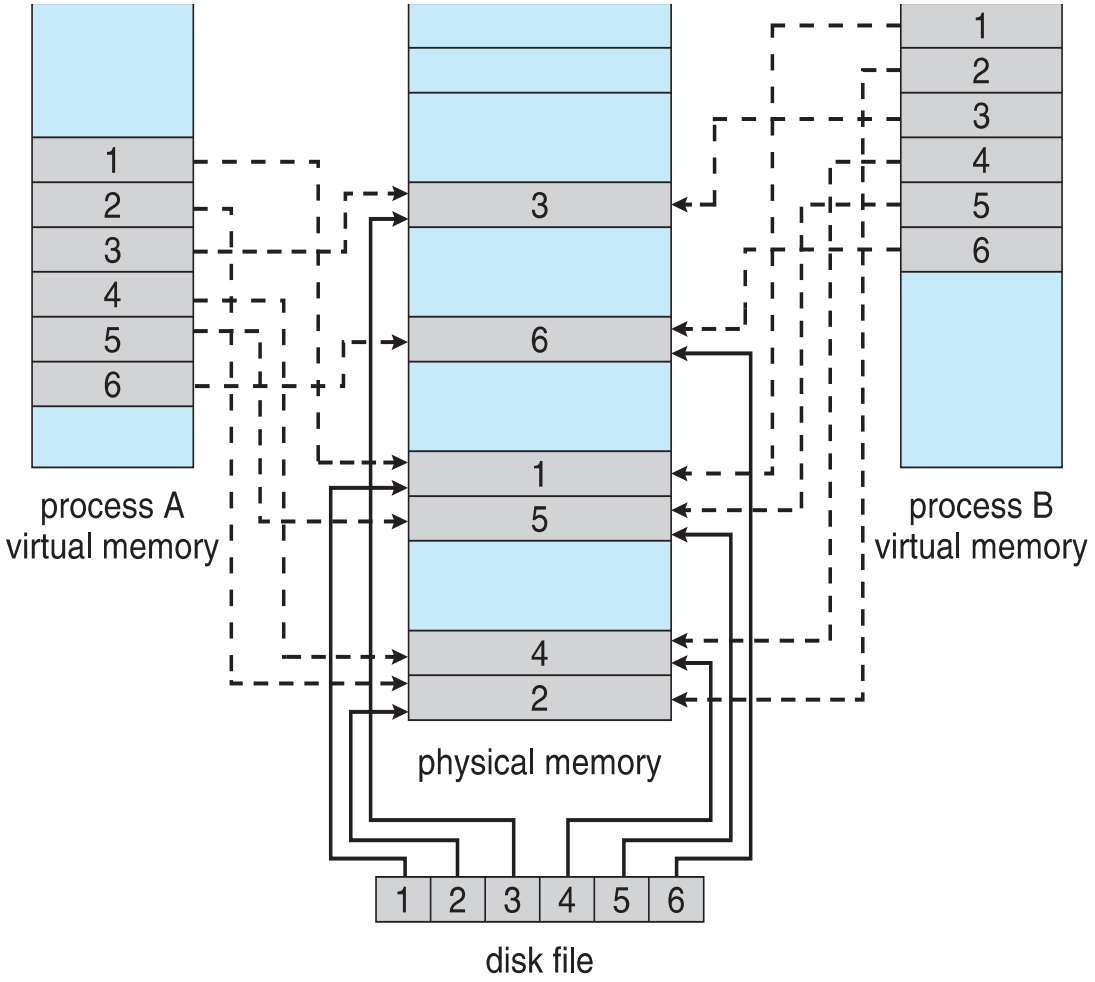


# Memory-Mapped Files

- Memory-Mapped File Technique for all I/O
  - Some OSes uses memory mapped files for standard I/O
  - Process can explicitly request memory mapping a file via mmap() system call (such as Solaris)
    - Now the file is mapped into the process's address space
  - If the user uses the standard I/O (open(), read(), write(), close()),
    - The file is memory mapped but this time into the kernel address space
      - Process still does read() and write()
        - » Copies data to and from kernel space and user space
  - Copy-On-Write can be used for read/write non-shared pages
  - Memory mapped files can be used for shared memory (although again via separate system calls)

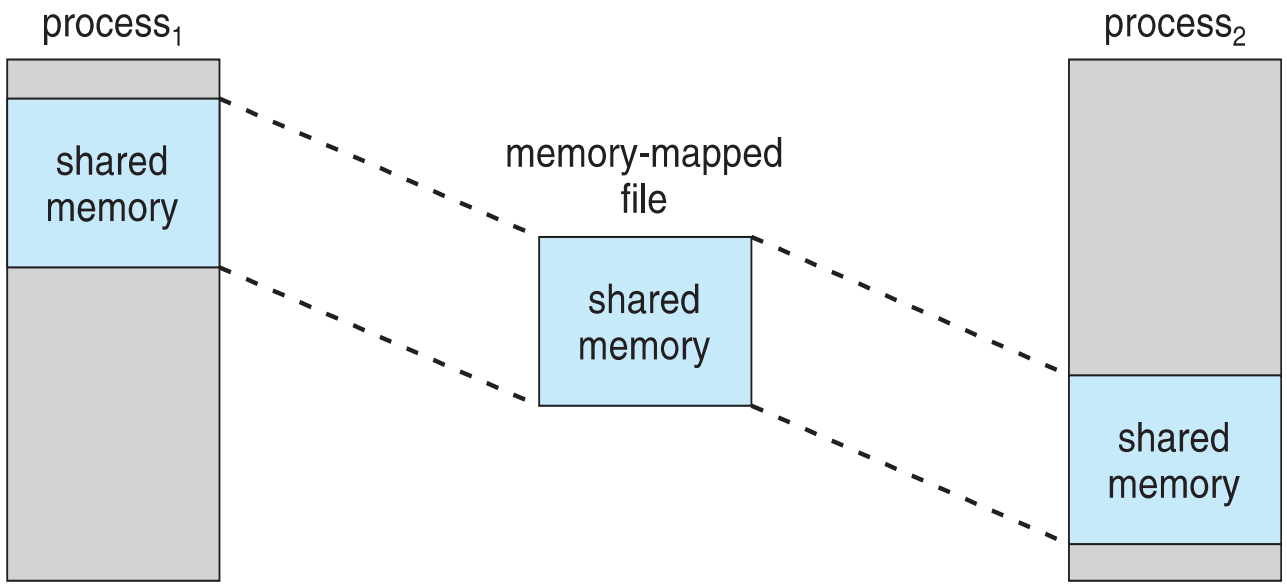


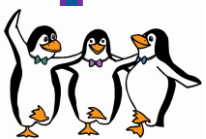
# Memory-Mapped Files



# Memory-Mapped Files

- Shared Memory via Memory-Mapped I/O



# Examples:

- Q1:

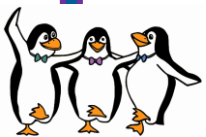
  Consider the following page reference string:

  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 3, 4, 0r 5 frames?

  Remember all frames are initially empty, so your first unique pages will all cost one fault each.

  a) LRU replacement
  b) FIFO replacement
  c) Optimal replacement



# Examples:

- Q2:

  Consider a demand-paged computer system where the degree of multiprogramming is currently fixed at four. The system was recently measure to determine utilization of CPU and the paging disk. The results are one of the following alternatives.

  For each case, what is happening?

  Can the degree of multiprogramming be increased to increase the CPU utilization? Is the paging helping?

  a) CPU utilization 13 percent; disk utilization 97 percent

  b) CPU utilization 87 percent; disk utilization 3 percent

  c) CPU utilization 13 percent; disk utilization 3 percent

  Answer:

  a. Thrashing is occurring.

  b. CPU utilization is sufficiently high to leave things alone, and increase degree of multiprogramming.

  c. Increase the degree of multiprogramming.