VIT®
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SWE3001-Operating Systems

Prepared By
Dr. L. Mary Shamala
Assistant Professor
SCOPE/VIT

Module 6: Virtual Memory

- Background
- Demand Paging
- Page Replacement
- Allocation of Frames
- Thrashing
- Memory-Mapped Files
- Allocating Kernel Memory

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

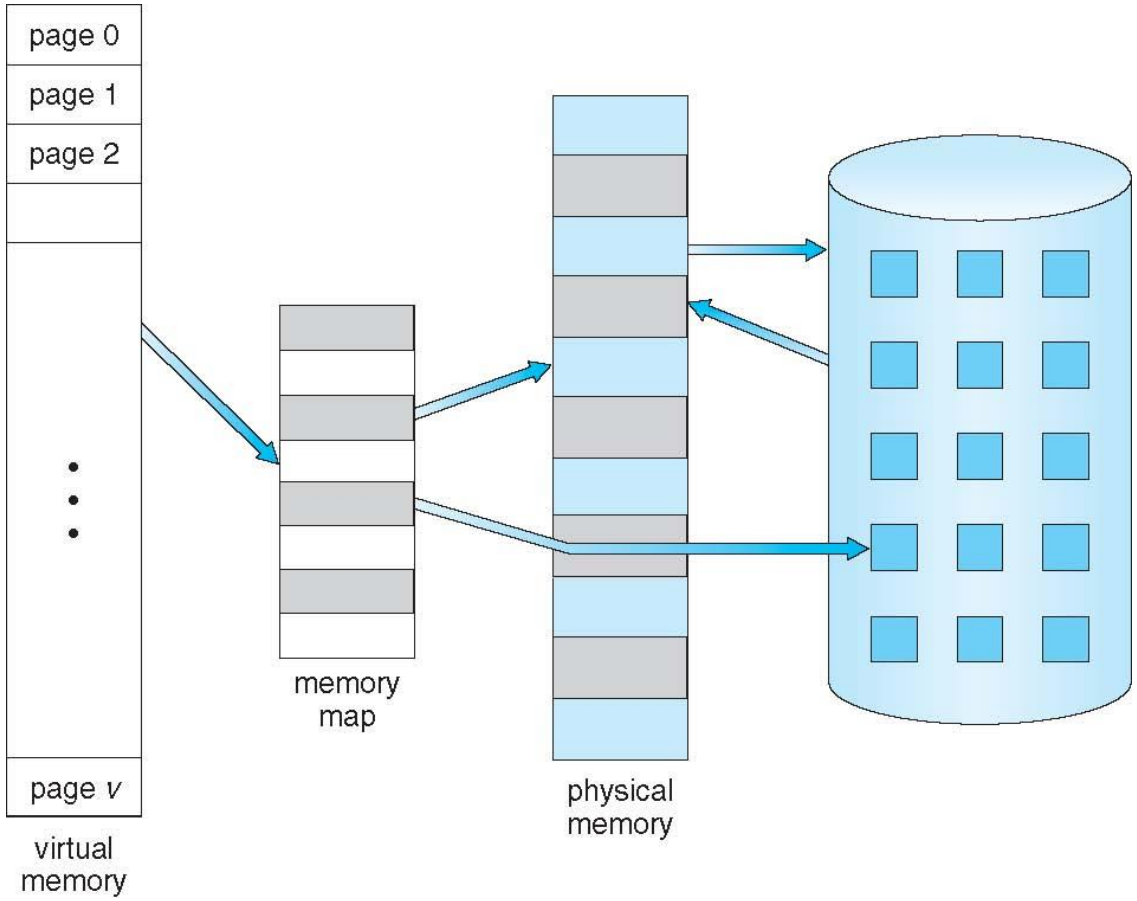
Background

- Code needs to be in memory to execute, but entire program rarely used
  - Error code, unusual routines, large data structures
- Entire program code not needed at 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

Background (Cont.)

- **Virtual memory** is a technique that allows the execution of processes that are not completely in memory.
- Advantages
  - Programs can be larger than physical memory
  - Abstracts main memory into an extremely large, uniform array of storage;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 That is Larger Than Physical Memory

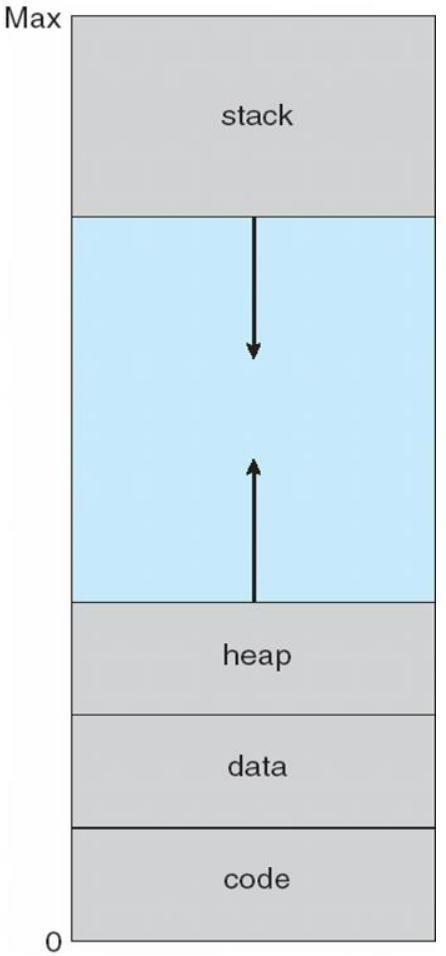


Background (Cont.)

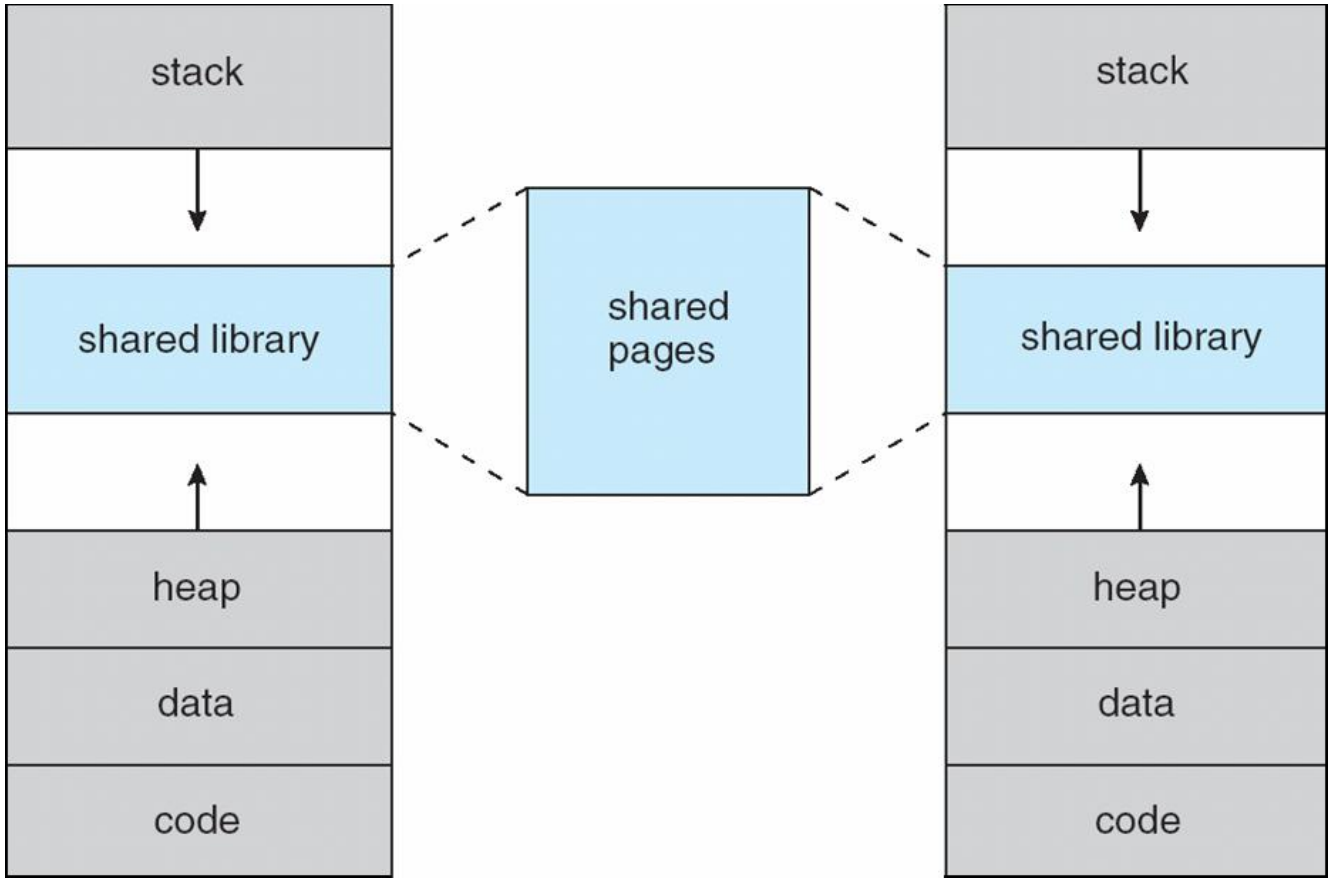
- **Virtual address space** – logical view of how a process is stored in memory
  - Usually start at address 0, contiguous addresses until end of space
  - Meanwhile, physical memory organized in page frames
  - MMU must map logical to physical
- Virtual memory can be implemented via:
  - Demand paging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

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()`, speeding process creation



Shared Library Using Virtual Memory



Demand Paging

- How an executable program might be loaded from disk into memory?

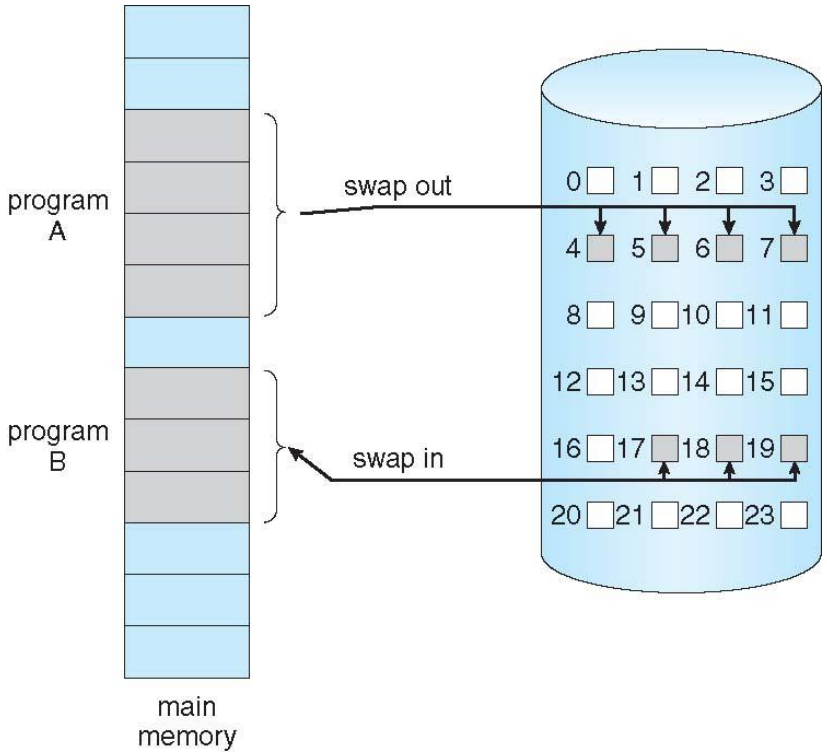
a) Load the entire program in physical memory at program execution time

b) Demand Paging: load pages only as they are needed.

- Pages that are never accessed are thus never loaded into physical memory.
- Benefits
  - Less I/O needed, no unnecessary I/O
  - Less memory needed
  - Faster response
  - More users

Demand Paging…

- A demand-paging system is similar to a paging system with swapping
- Page is needed ⇒ reference to it
  - invalid reference ⇒ abort
  - not-in-memory ⇒ bring to memory
- **Lazy swapper** – never swaps a page into memory unless that page will be needed
  - Swapper that deals with pages is a **pager**

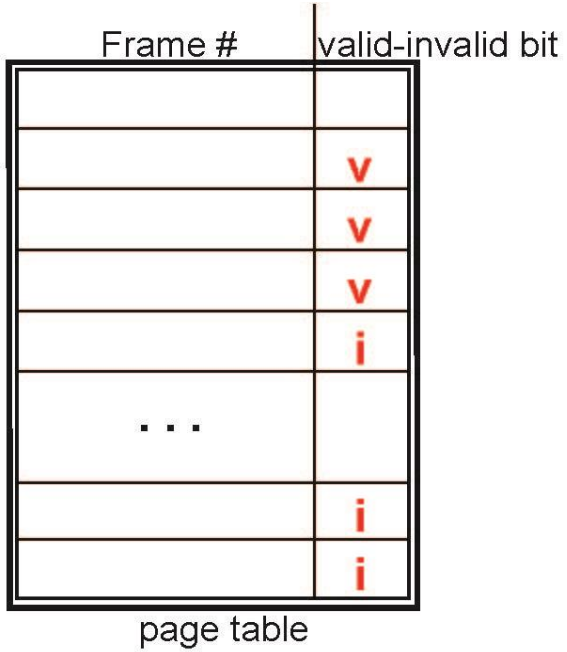


Basic Concepts

- With swapping, pager guesses which pages will be used before swapping out again
- Instead, pager brings in only those 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

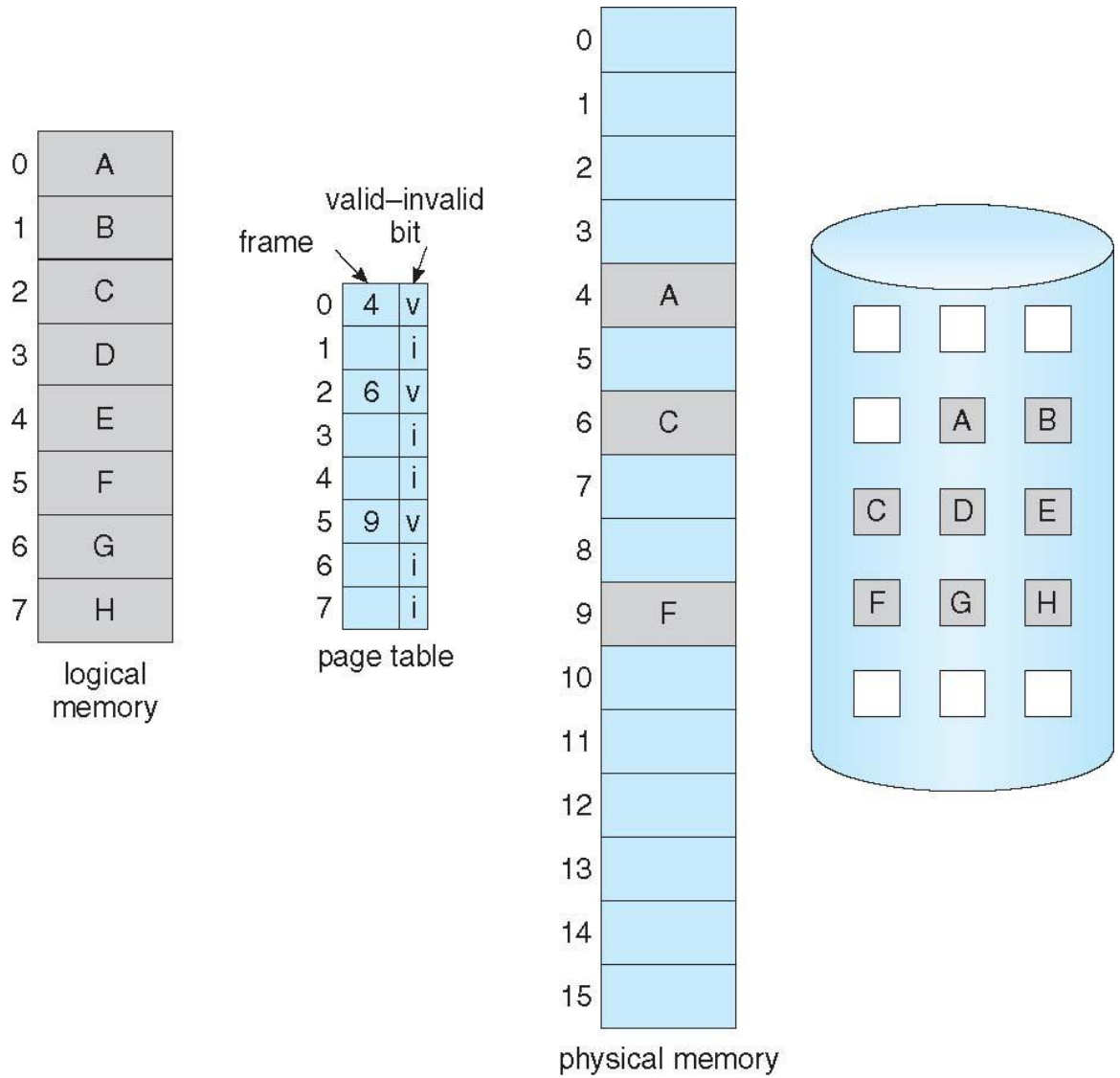
Valid-Invalid Bit

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



- During MMU address translation, if valid–invalid bit in page table entry is **i** ⇒ page fault

Page Table When Some Pages Are Not in Main Memory

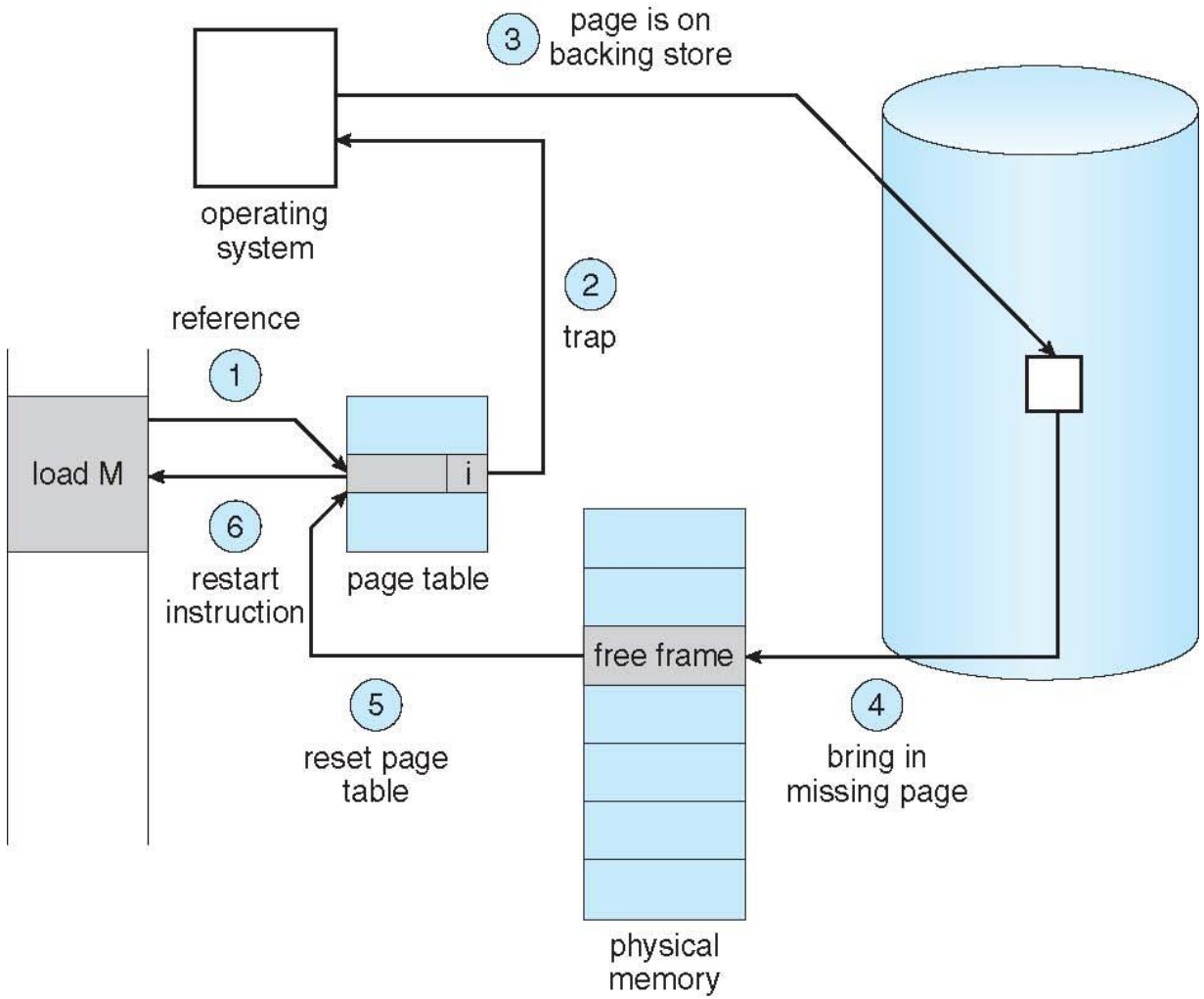


Page Fault

- Access to a page marked invalid causes a **page fault**
- If there is a reference to a page, first reference to that page will trap to operating system:

1. Operating system looks at an internal table of the process to decide valid or invalid memory access
2. Invalid reference $\Rightarrow$ abort; If valid and not in memory, now page it in
3. Find free frame
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



Aspects of Demand Paging

- Extreme case – start process with *no* pages in memory
  - OS sets instruction pointer to first instruction of process, non-memory-resident -> page fault
  - And for every other process pages on first access
  - **Pure demand paging-** never bring a page into memory until it is required
- Actually, a given instruction could access multiple pages -> multiple page faults
  - Pain decreased because of **locality of reference**
- Hardware support needed for demand paging
  - Page table with valid / invalid bit
  - Secondary memory (swap device with **swap space**)
- A crucial requirement for demand paging is the ability to restart any instruction after a page fault

Performance of Demand Paging

- Three major activities
  - Service the interrupt – careful coding means just several hundred instructions needed
  - Read the page – lots of time
  - Restart the process – again just a small amount of time
- Let p be the probability of a page fault or Page Fault Rate ($0 \leq p \leq 1$)
  - if p = 0 no page faults
  - if p = 1, every reference is a fault

Effective Access Time = (1 – p) × ma + p × page fault time.

where ma→memory access

Page Fault time→ page fault overhead+ swap page out+ swap page in

Performance of Demand Paging…

- A page fault causes the following sequence to occur:

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:
    1. Wait in a queue for this device until the read request is serviced
    2. Wait for the device seek and/or latency time
    3. 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

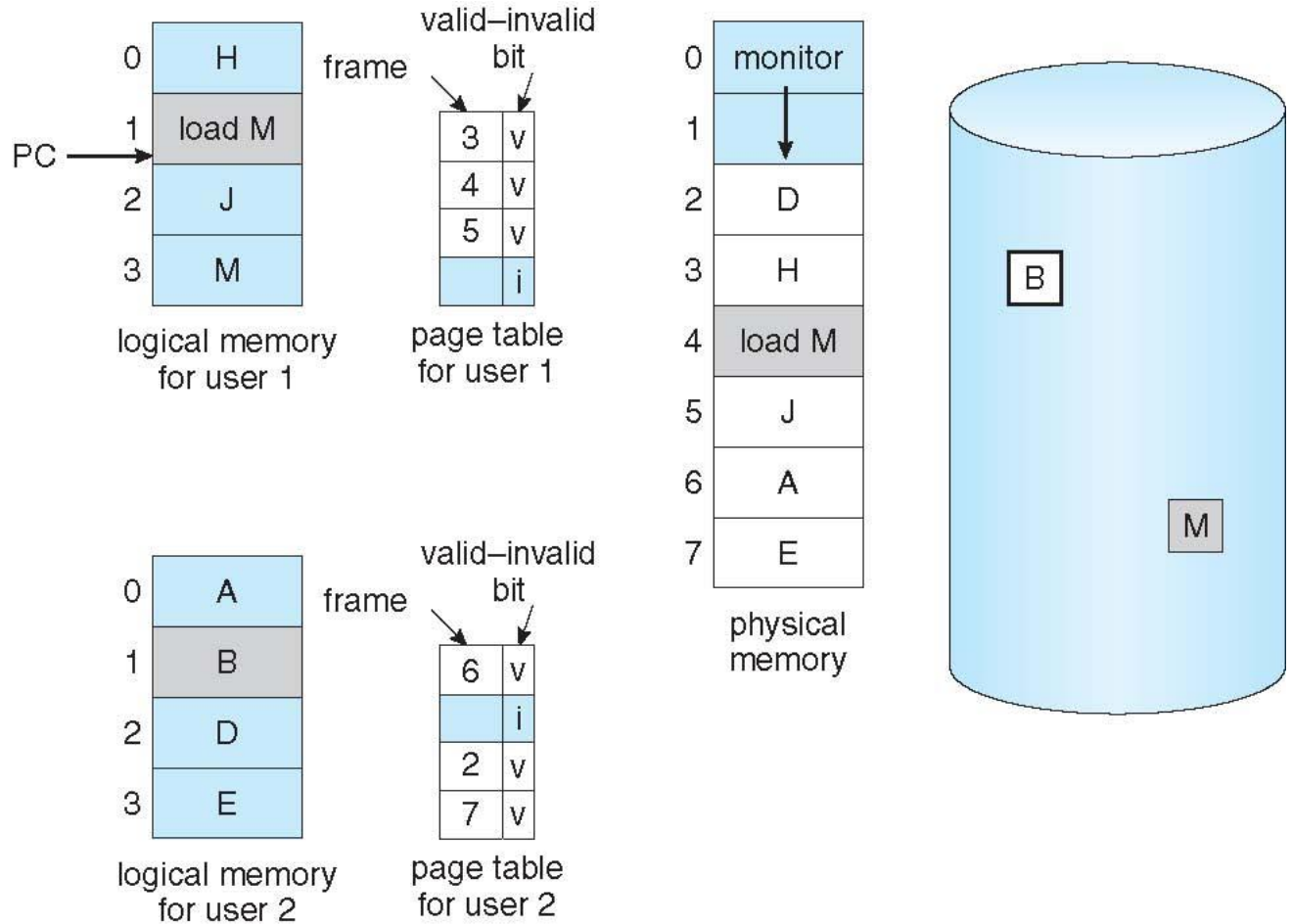
Demand Paging Example

- 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
- Hence, the effective access time is directly proportional to the page-fault rate.
- 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

What Happens if There is no Free Frame?

- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- How much to allocate to each?
- 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

Need For Page Replacement

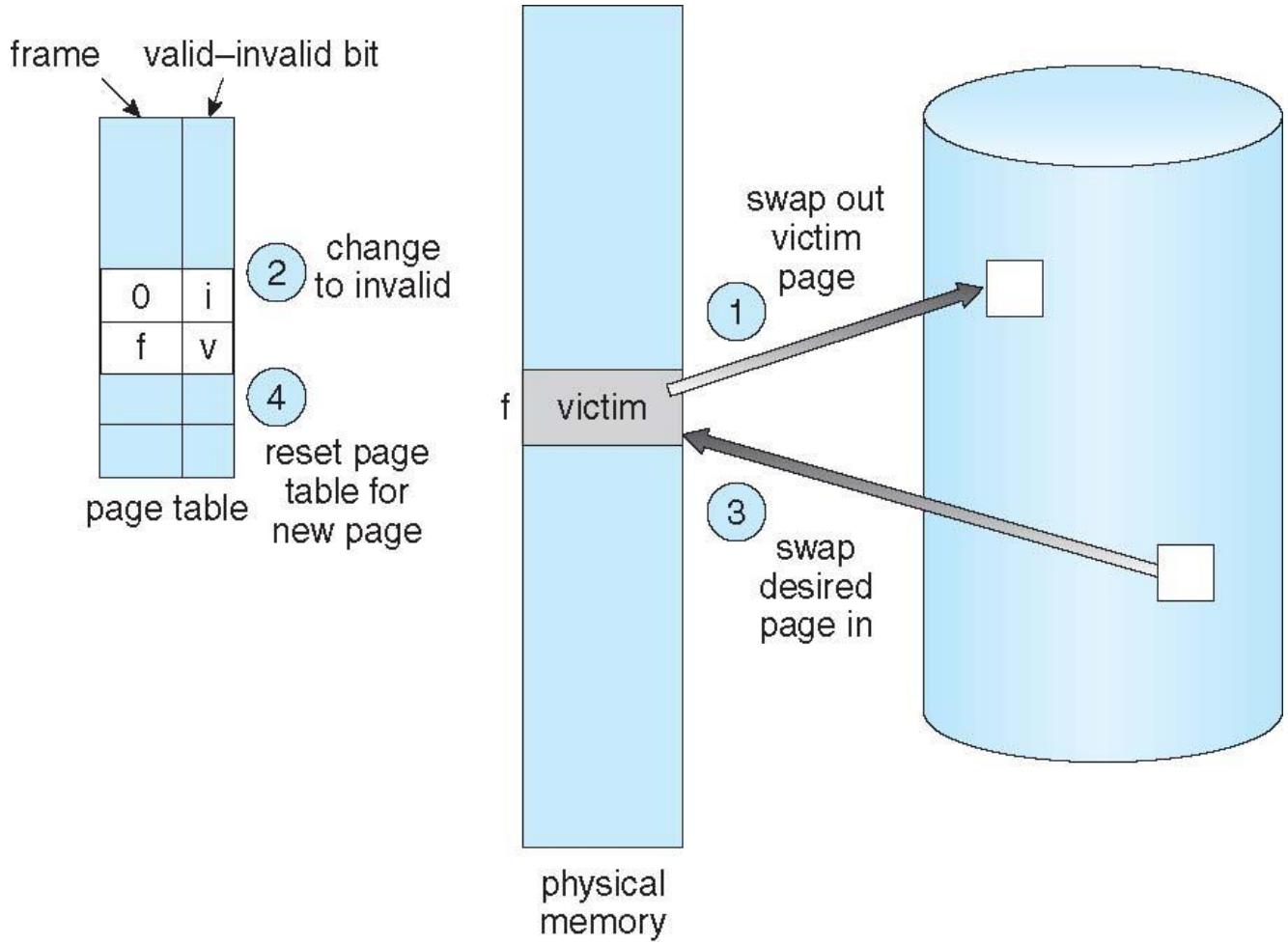


Basic Page Replacement

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



Basic Page Replacement…

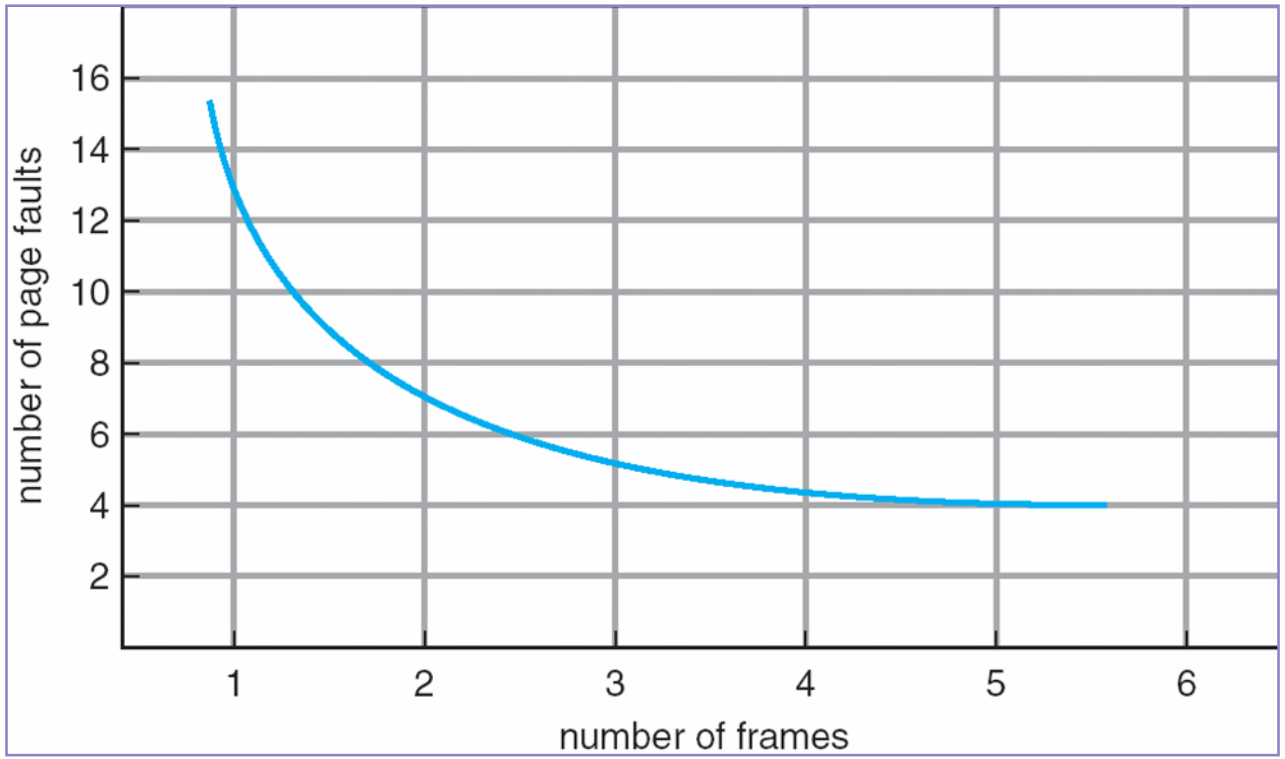
- Use **modify** (**dirty**) **bit** to reduce overhead of page transfers – only modified pages are written to disk
- Page replacement is basic to demand paging.
- It completes the separation between logical memory and physical memory – large virtual memory can be provided on a smaller physical memory
- Two major problems to implement demand paging:
  - frame-allocation algorithm
  - page-replacement algorithm

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

Graph of Page Faults Versus The Number of Frames

For example, if we trace a particular process, we might record the following address sequence:
0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105
At 100 bytes per page, this sequence is reduced to the following reference string:
1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1

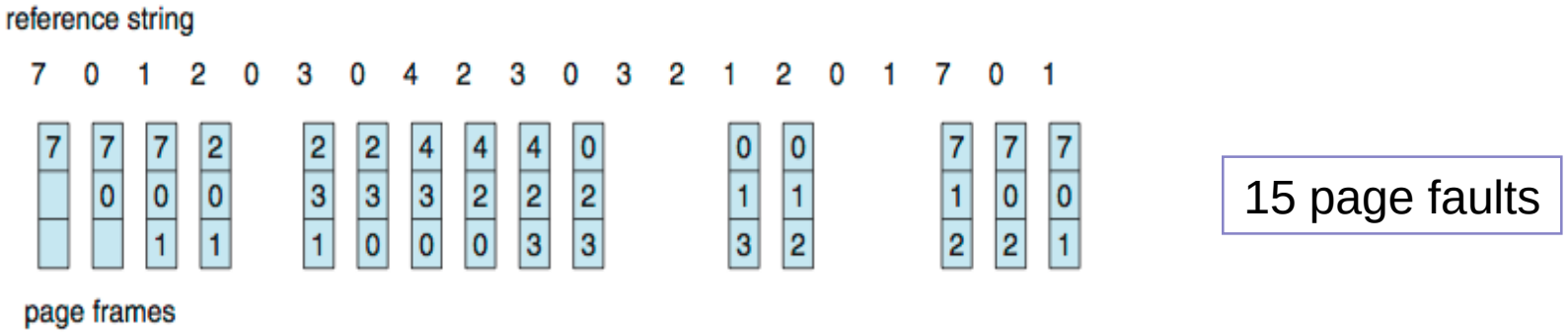


Page Replacement Algorithms

- First-In-First-Out (FIFO) Algorithm
- Optimal Algorithm
- Least Recently Used (LRU) Algorithm
- LRU Approximation Algorithms
  - Additional Reference bits
  - Second-chance algorithm
  - Enhanced Second-Chance Algorithm
- Counting-based Page Replacement
- Page Buffering Algorithms

First-In-First-Out (FIFO) Algorithm

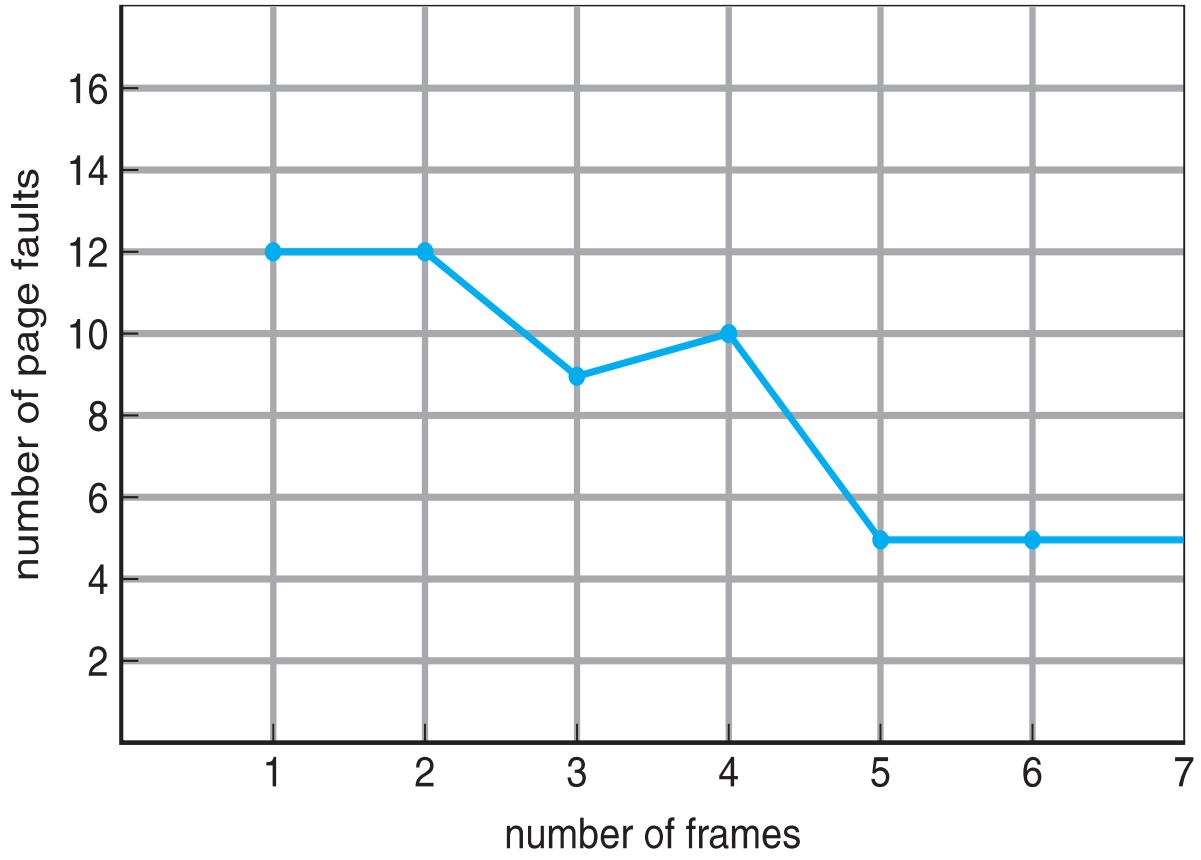
- Reference string: **7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1**
- 3 frames (3 pages can be in memory at a time per process)



15 page faults

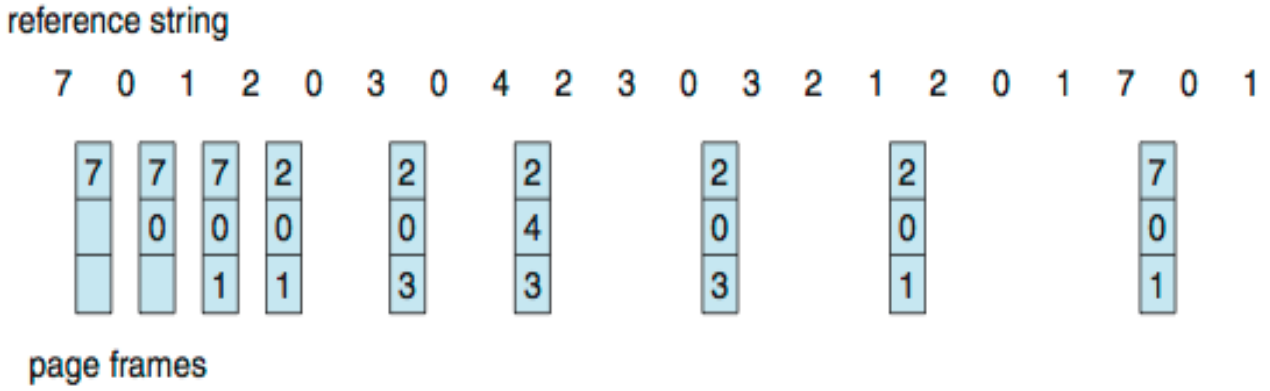
- Can vary by reference string: consider 1,2,3,4,1,2,5,1,2,3,4,5
  - Adding more frames can cause more page faults!
    - **Belady's Anomaly**
- How to track ages of pages?
  - Just use a FIFO queue

FIFO Illustrating Belady's Anomaly



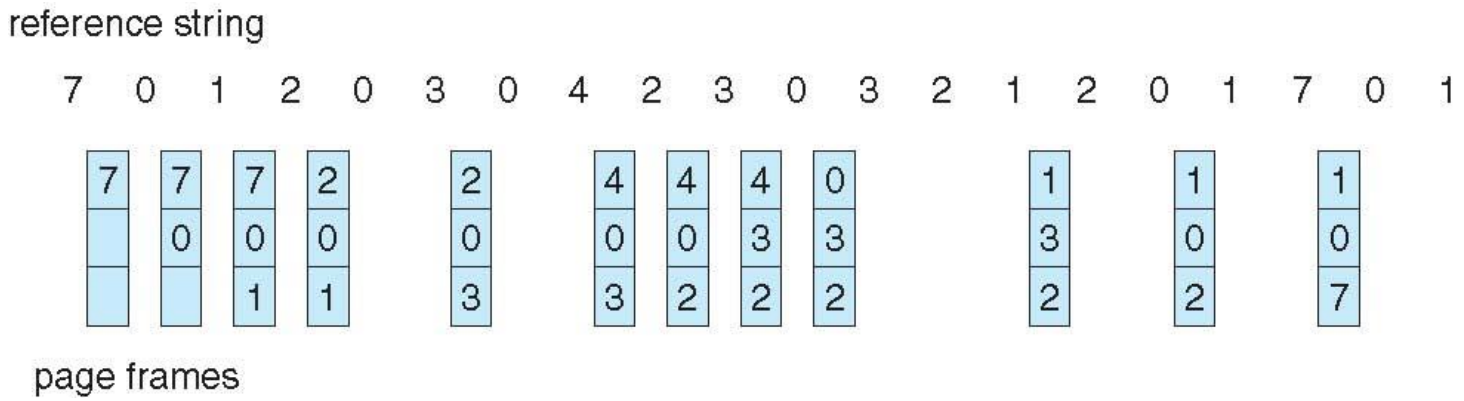
Optimal Algorithm

- Replace page that will not be used for longest period of time
  - 9 is optimal for the example
- How do you know this?
  - Can't read the future
- Used for measuring how well your algorithm performs



Least Recently Used (LRU) Algorithm

- Use past knowledge rather than future
- Replace page that has not been used in the most amount of time
- Associate time of last use with each page

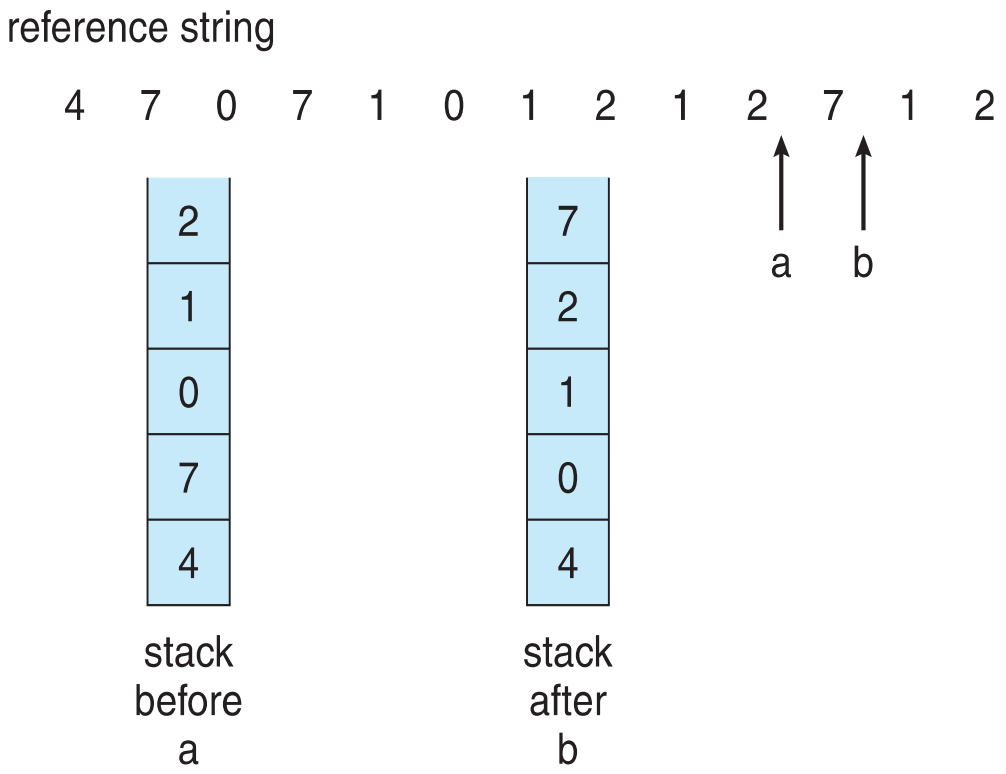


- 12 faults – better than FIFO but worse than OPT
- Generally good algorithm and frequently used
- But how to implement?

LRU Algorithm (Cont.)

- Counter implementation
  - Every page entry has a counter; every time page is referenced through this entry, copy the clock into the counter
  - When a page needs to be changed, look at the counters to find smallest value
    - Search through table needed
- Stack implementation
  - Keep a stack of page numbers in a 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
- LRU and OPT are cases of **stack algorithms** that don't have Belady's Anomaly

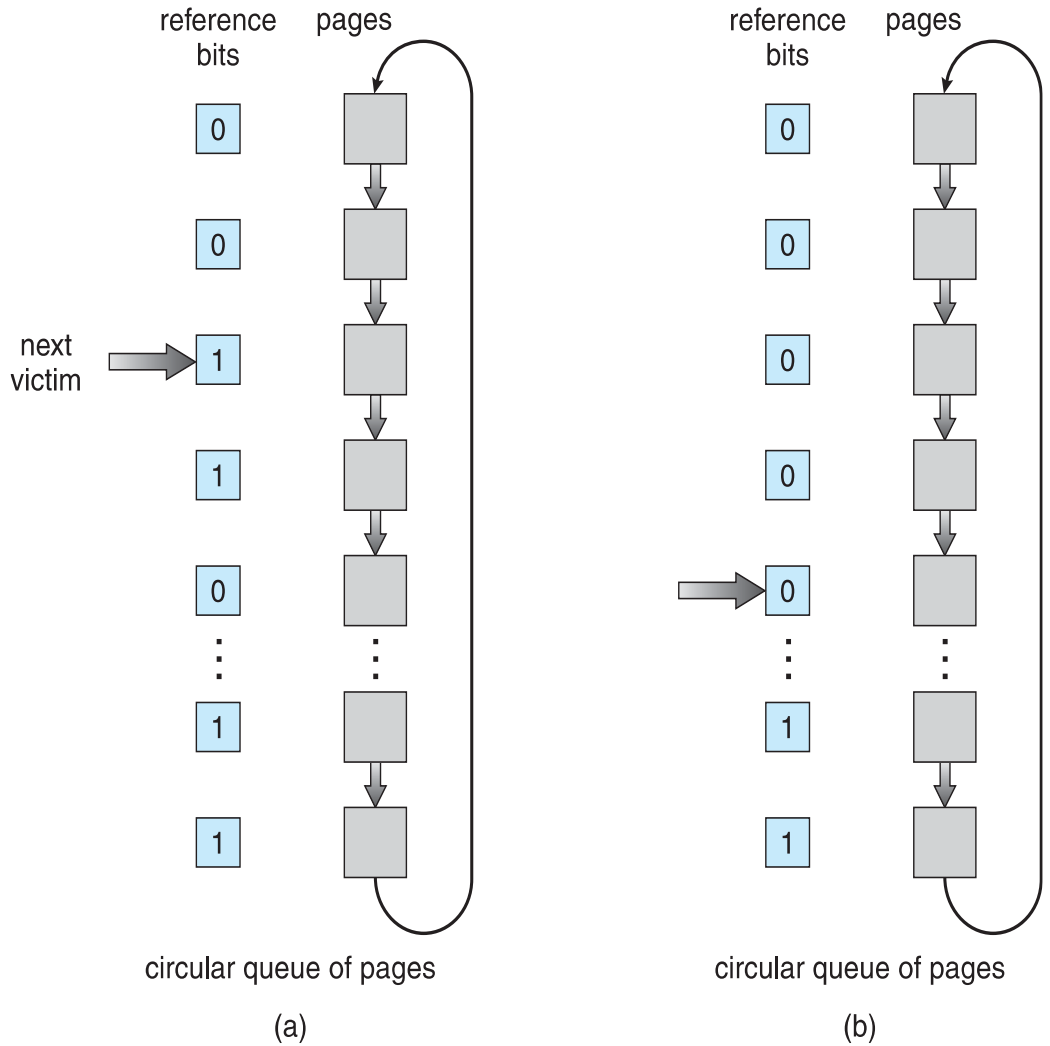
Use Of A Stack to Record Most Recent Page References



LRU Approximation Page Replacement

- LRU needs special hardware and still slow
- **Reference bit**
  - With each page associate a bit, initially = 0
  - When page is referenced bit set to 1
  - Replace any with reference bit = 0 (if one exists)
    - We do not know the order, however
- **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 (clock) Page-Replacement Algorithm



Enhanced Second-Chance Algorithm

- Improve algorithm by using reference bit and modify bit (if available) in concert
- Take ordered pair (reference, modify)

1. (0, 0) neither recently used not modified – best page to replace
2. (0, 1) not recently used but modified – not quite as good, must write out before replacement
3. (1, 0) recently used but clean – probably will be used again soon
4. (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-Based Page Replacement

- Keep a counter of the number of references that have been made to each page
- **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, always
  - Then frame available when needed, not found at fault time
  - 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 given 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*** is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

Fixed Allocation

- **Equal allocation** –If there are m frames and n processes, **m/n frames**(equal share) are allocated to each process
  - For example, if there are 93 frames (after allocating frames for the OS) and 5 processes, give each process 18 frames
  - Keep some as the free frame buffer pool

- **Proportional allocation** – Allocate available memory according to the size of process
  - Dynamic as degree of multiprogramming, process sizes change
  - Let the virtual memory size for process p_i be $v(p_i)$.
  - Let there be n processes and m frames.
  - Then the total virtual memory size is $V=\Sigma\, v(p_i)$.
  - Allocate a_i= **(v(p)/V)*m frames** to process p_i.

Priority Allocation

- With either equal or proportional allocation, a high-priority process is treated the same as a low-priority process.
- Use a proportional allocation scheme using priorities rather than size

- 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

Example

- Consider a system having 64 frames and 4 processes with the following virtual memory sizes:

  v(1)=16, v(2)128; v(3)=64 and v(4)=48
- Here, n=64; p=4
- For equal allocation, n/p frames are allocated to each process=> **64/4=16 frames to each process**
- For proportional allocation, V=16+128+64+48=256; It allocates

  (16/256)*64= **4 frames to process 1**

  (128/256)*64= **32 frames to process 2**

  (64/256)*64= **16 frames to process 3**

  (48/256)*64= **12 frames to process 4**

Global vs. Local Allocation

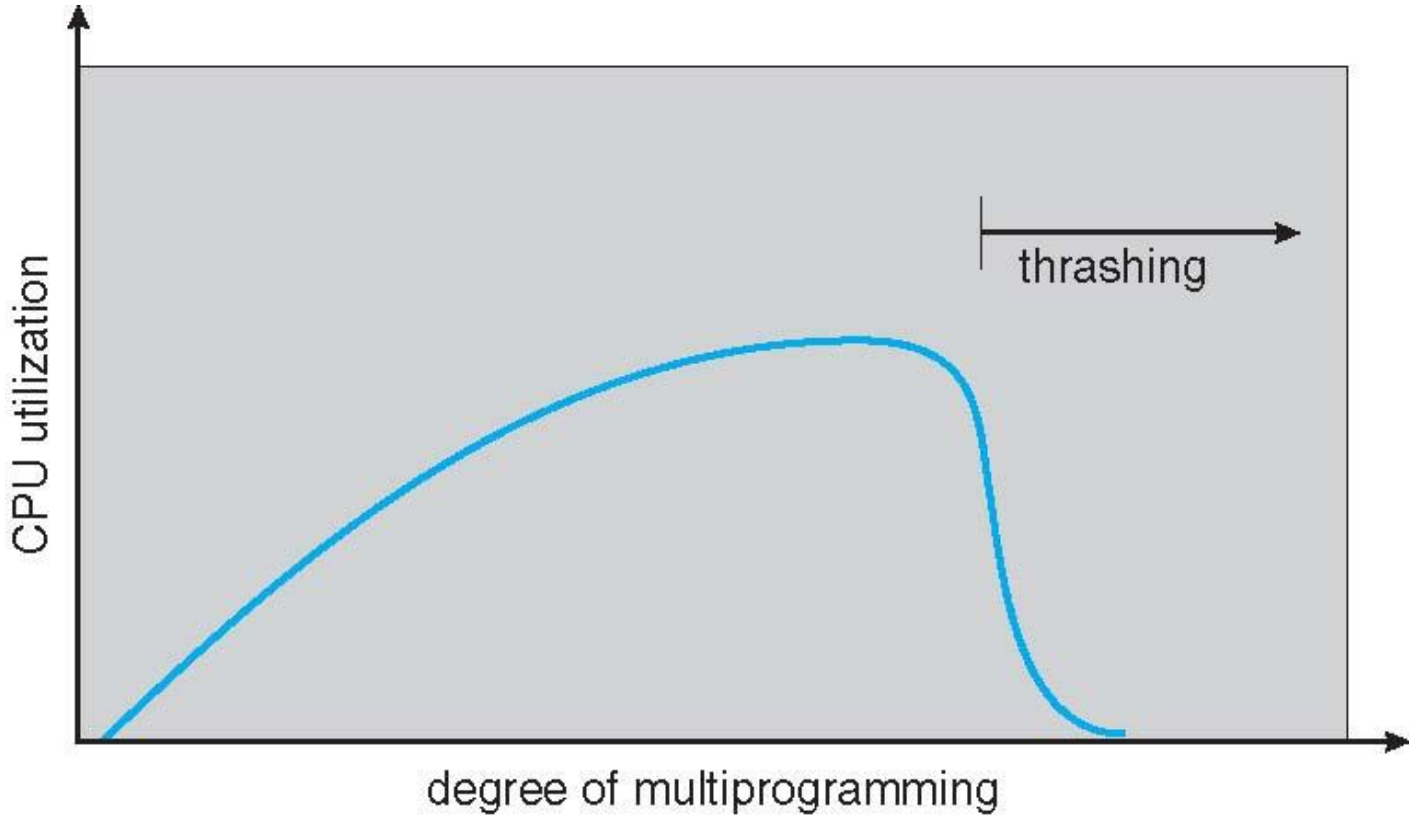
- Classify page-replacement algorithms into two broad categories:
- **Global replacement** – process selects a replacement frame from the set of all frames; one process can take a frame from another
  - But then process execution time can vary greatly
  - But greater throughput so more common

- **Local replacement** – each process selects from only its own set of allocated frames
  - More consistent per-process performance
  - But possibly underutilized memory

Thrashing

- **Thrashing** ≡ a process is busy swapping pages in and out
- **High paging activity is called thrashing**.
- A process is thrashing if it is spending more time paging than executing.
- If a process does not have "enough" pages, the page-fault rate is very high
  - Page fault to get page
  - Replace existing frame
  - But quickly need replaced frame back
  - This leads to:
    - Low CPU utilization
    - Operating system thinking that it needs to increase the degree of multiprogramming
    - Another process added to the system
- Thrashing causes considerable degradation in system performance

Thrashing (Cont.)



Thrashing …

- Limit the effects of thrashing by using a local replacement algorithm (or priority replacement algorithm
- To prevent thrashing, provide a process with as many frames as it needs.
  - how do we know how many frames it “needs”?
  - The working-set strategy starts by looking at how many frames a process is actually using.
  - This approach defines the **locality model** of process execution.
- **Locality model**
  - Process migrates from one locality to another
  - A locality is a set of pages that are actively used together
  - A program is generally composed of several different localities may overlap
- Why does thrashing occur?
  Σ size of locality > total memory size

Working-Set Model

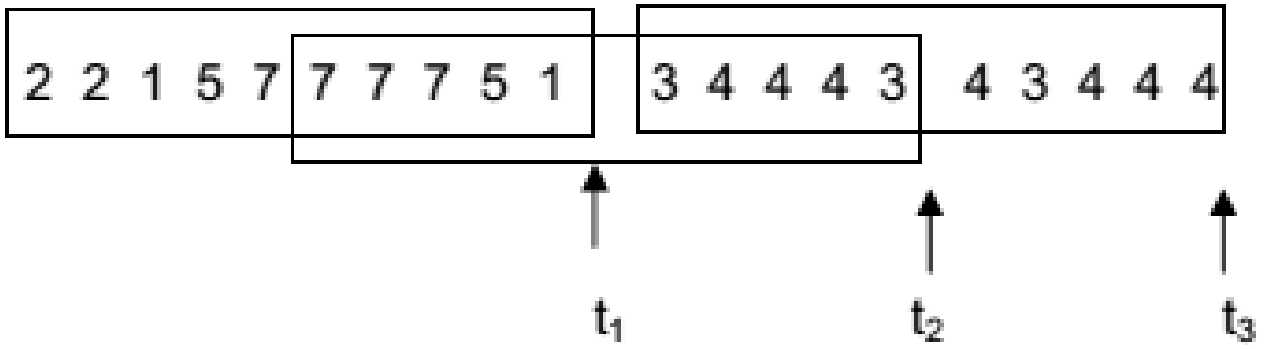
- The working-set model is based on the assumption of locality.
- This model uses a parameter, Δ to define the working-set window.
- The idea is to examine the most recent Δ page references.
- The set of pages in the most recent Δ page references is the working set
- If a page is in active use, it will be in the working set. If it is no longer being used, it will drop from the working set Δ time units after its last reference.
- Thus, the working set is an approximation of the program's locality.
- Working-set strategy prevents thrashing while keeping the degree of multiprogramming as high as possible.
- Thus, it optimizes CPU utilization.

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

Working Set Model

- Assume Δ =10



Working sets of this process at these time instants will be:

WS(t1) = {2,1,5,7}
WS(t2) = {7,5,1,3,4}
WS(t3) = {3,4}

Keeping Track of the Working Set

- The difficulty with the working-set model is keeping track of the working set
  - The working-set window is a moving window.
- Approximate the working-set model with a fixed interval timer + a reference bit
- Example: Δ = 10,000
  - Timer interrupts after every 5000 time units
  - Keep in memory 2 bits for each page
  - Whenever a timer interrupts copy and sets the values of all reference bits to 0
  - If one of the bits in memory = 1 $\Rightarrow$ page in working set
- Why is this not completely accurate?
- Improvement = 10 bits and interrupt every 1000 time units

Memory-Mapped Files

- Consider a sequential read of a file on disk using the standard system calls open() , read() , and write() .
- Each file access requires a system call and disk access.
- Alternatively, we can use the virtual memory techniques to treat file I/O as routine memory accesses.
- This approach, known as **memory mapping a file**, allows a part of the virtual address space to be logically associated with the file.
- This can lead to significant performance increases.

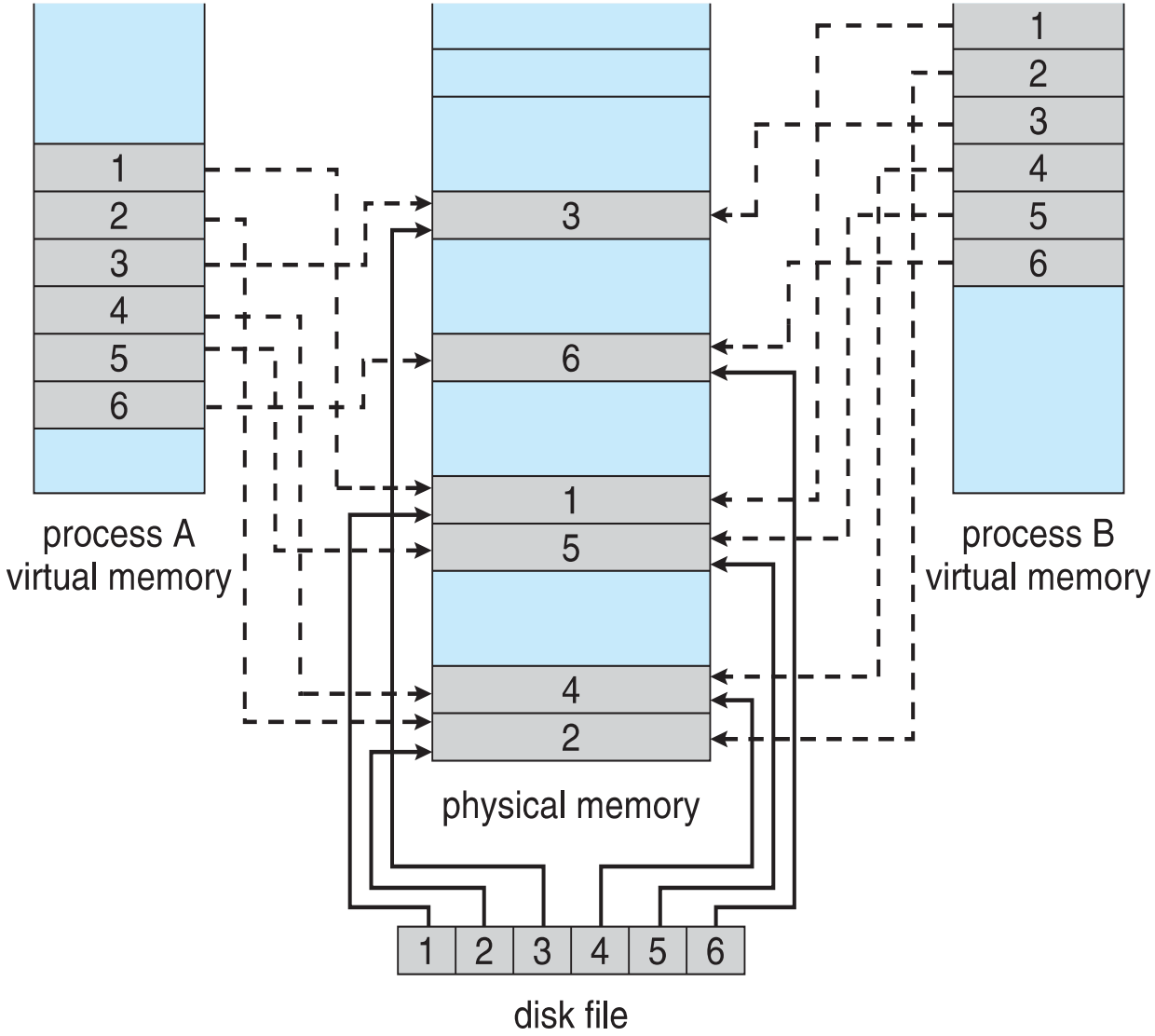
Memory-Mapped Files

- Memory-mapped file I/O allows file I/O to be treated as routine memory access by **mapping** a disk block to a page in memory
- A file is initially read using 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

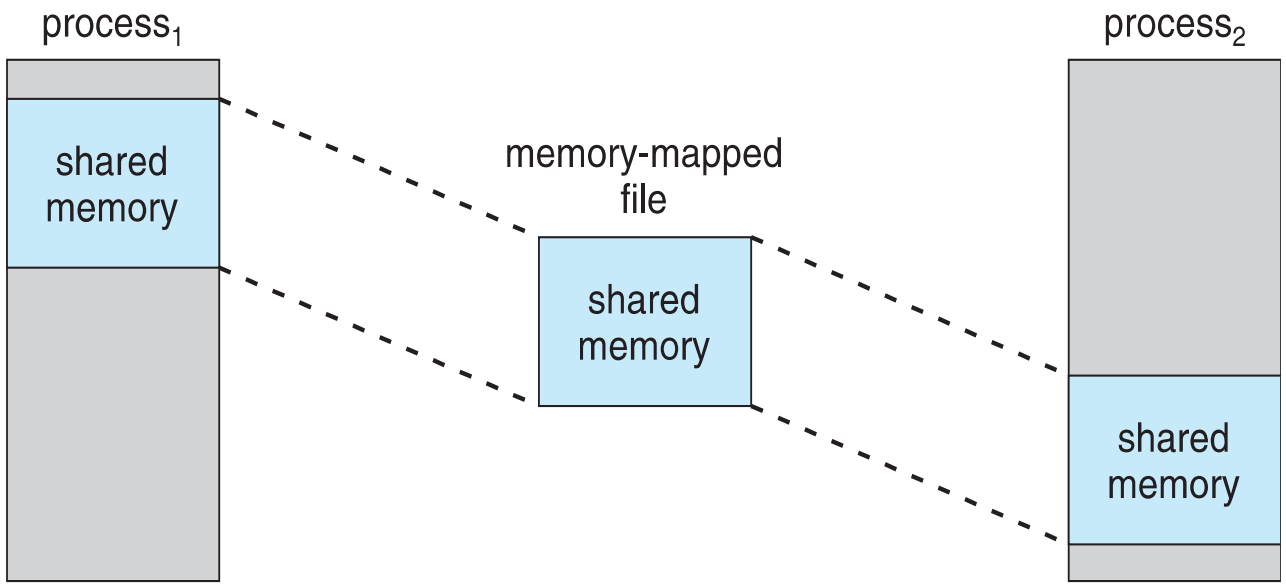
Memory-Mapped File Technique for all I/O

- Some OSes uses memory mapped files for standard I/O
- Process can explicitly request memory mapping a file via `mmap()` system call
  - Now file mapped into process address space
- For standard I/O (`open()`, `read()`, `write()`, `close()`), map anyway
  - But map file into kernel address space
  - Process still does read() and write()
    - Copies data to and from kernel space and user space
  - Uses efficient memory management subsystem
    - Avoids needing separate subsystem
- 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



Shared Memory via Memory-Mapped I/O



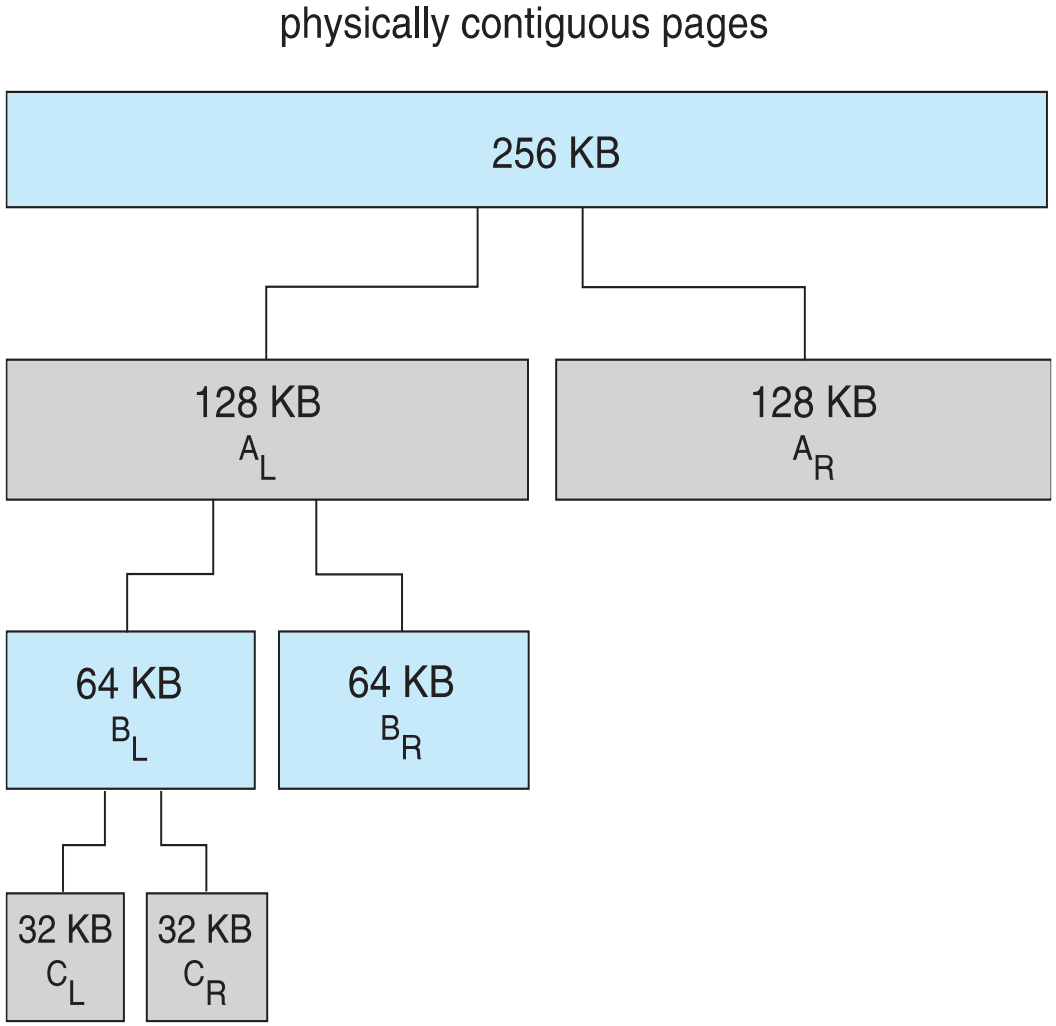
Allocating Kernel Memory

- Treated differently from user memory
- Often allocated from a free-memory pool different from the list used to statisfy user-mode processes
- There are two primary reasons for this:
  - Kernel requests memory for structures of varying sizes
  - Some kernel memory needs to be contiguous
    - I.e. for device I/O
- 2 strategies for managing free memory that is assigned to kernel processes:
  - “buddy system”
  - slab allocation

Buddy System

- Allocates memory from fixed-size segment consisting of physically-contiguous pages
- Memory is 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
- For example, assume 256KB chunk available, kernel requests 21KB
  - Split into 2 buddies 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 21KB request
- Advantage – quickly **coalescing** unused chunks into larger chunk
- Disadvantage - fragmentation

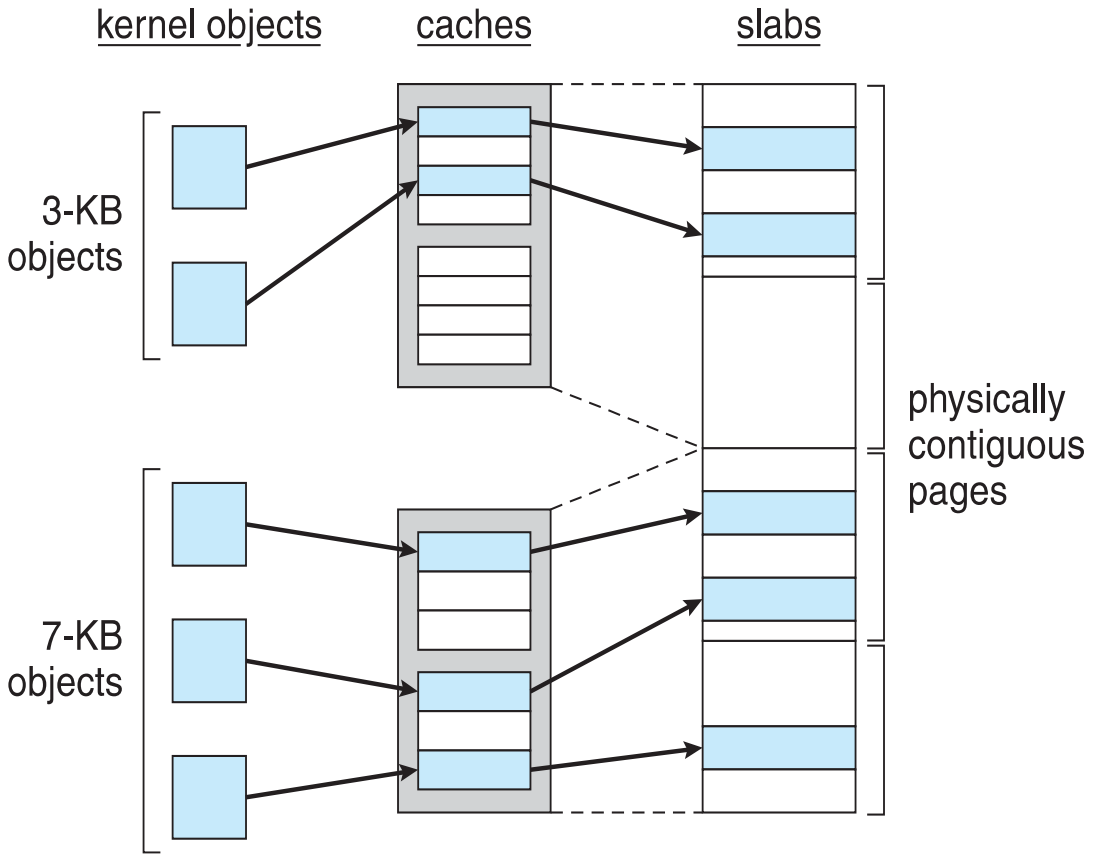
Buddy System Allocator



Slab Allocator

- **Slab** is made up of 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
- The slab-allocation algorithm uses caches to store kernel objects
- When a cache is created, filled with objects marked as **`free`**
- When structures stored, objects marked as **`used`**
- If slab is full of used objects, next object is allocated from empty slab
  - If no empty slabs, new slab allocated
- Benefits include no fragmentation, fast memory request satisfaction

Slab Allocation



Review Questions

- Under what circumstances do page faults occur? Describe the actions taken by the operating system when a page fault occurs.
- 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 one, two, three, four, five, six, and seven frames? Remember that all frames are initially empty, so your first unique pages will cost one fault each.
  - LRU replacement
  - FIFO replacement
  - Optimal replacement

Summary

- Virtual memory is a technique that enables us to map a large logical address space onto a smaller physical memory.
- Virtual memory allows us to run extremely large processes and to raise the degree of multiprogramming, increasing CPU utilization.
- Virtual memory is commonly implemented by demand paging.
- Pure demand paging never brings in a page until that page is referenced.
- If total memory requirements exceed the capacity of physical memory, page-replacement algorithms are used to replace pages from memory to free frames for new pages.
- The frame-allocation policy can be fixed, suggesting local page replacement, or dynamic, suggesting global replacement.