Chapters 5 (Process Synchronization) Study Guide

- Race Condition
  - several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place
  - To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter . To make such a guarantee, we require that the processes be synchronized in some way.
- The critical section problem
  - process may be changing common variables/updating table/writing file, etc.
  - When one process is in the critical section, no other may be in the critical section
  - process must ask permission to enter critical section in entry section, may follow critical section with exit section, then remainder section. Set the flag when process is in critical section, set it to 1. When it is done, it is set back to 0.
- Solutions to critical section problem:
- Mutual Exclusion
  - if process P_i is executing in its critical section, then no other processes can be executing in their critical sections
- Progress
  - If no process is executing in its critical section and there exist some process that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefintely.
- Bounded waiting - bound must exist on the # of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Software solution (Peterson Solution)
    - two process solution
    - cannot be interrupted flag ready to run on critical section
  - Hardware Solution (Lock, spinlock, Mutex Lock, Semaphores (Counting and binary semaphores)

- •Locking - that is, protecting critical regions through the use of locks
- •Spinlock (Mutex Lock [Mutual Lock]) - uses the idea of busy waiting (wasting CPU Time) to solve the critical section problem
- •Semaphores - a signaling mechanism which provides access to resources, the signal goes out when a resource becomes available.
  - •Semaphore has an unlimited number of resources in comparison to mutex, as mutex locks for a single resource
  - •It is an integer variable that is used for signaling resources w/ two atomic operations
  - •wait() to see if a resource is available by checking the value of semaphore to indicate whether a resource is available or not, if its <= 0 then no resources are available.
  - •signal() is to indicate that a resource is available for a process.
  - •Counting Semaphore - integer value ranges over an unrestricted domain
  - •Binary semaphore - integer value is either 0 or 1, very similar to mutex locks

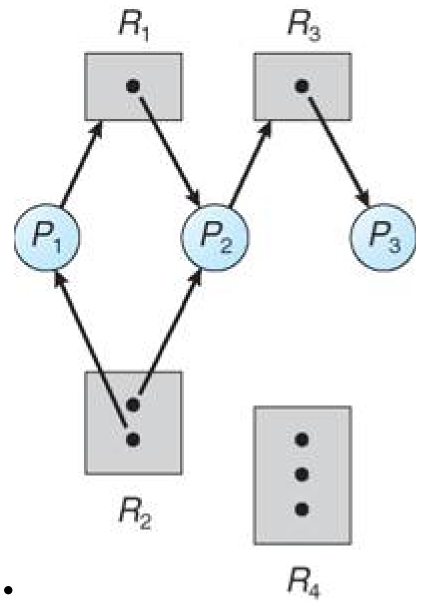
Classical problems:

- •Bounded waiting
  - •more than one semaphore is used in this case, there are three semaphores actually used.
  - •Each buffer hold one item, and the three seamphores are mutex,full,and empty.
  - •Mutex acts as a one/zero
  - •Semaphores are integer values that keep track of the numbers in full/empty, mutex and monitors the bound
- •Readers-writers problem (First & Second)
  - •data set shared among an umber of concurrent processes
    - •Readers can only read
    - •Writers can read/write
  - •Problem - allow multiple readers to read at the same time
  - •Only one single writer can access shared data at the same time
  - •Shared Data
    - •data set

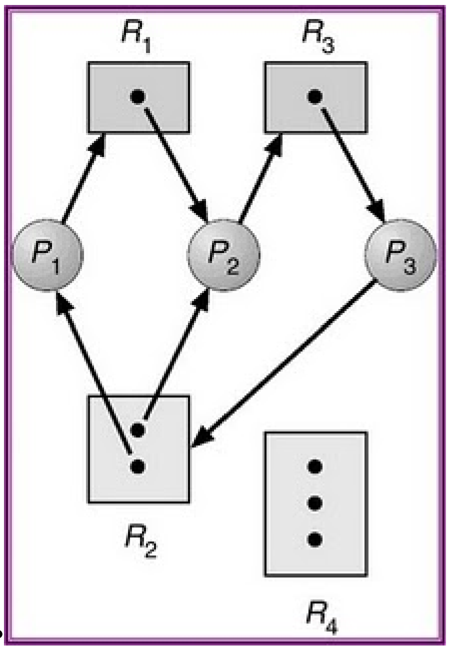
        - •rw_mutex - initialized to 1
        - •mutex - initalized to 1
        - •read_count - initialized to 0
    - •Variations:
        - •first - no reader keeps waiting unless writer has permission to use shared object
        - •second - once writer is ready it performs write ASAP
        - •Both could have starvation and lead to even more variations
        - •solved by system kernels providing reader-writer locks
- •Dining philosophers problem
    - •Distribute 5 chopsticks and allow philosophers to eat not adjacent with one another.
    - •Deadlock handling
        - •Allow at most 4 philosophers to be sitting simulataneously at the table
        - •Allow a philiosopher to pick up forks only if both are available (picking must be done in a critical section)
        - •Asymmetric solution - an odd number philosopher picks up the left chopstick and the the right chopstick, even numbered philsopher picks up the first right chopstick and then the left chopstick
- •Monitors
    - •use wait and signal in that order. Semaphores shouldn't skip one of these operations, else there will be a deadlock
    - •high level abstraction used as a mechanism for process synchronization
    - •ADT (internal variables only) accessible by code within the procedure
    - •1 process may be active with the monitor at a time
    - •not powerful enough to monitor some synchronization schemes

OS - Chapter 7 (Deadlock) - Study Guide

- What is a deadlock?
  - occurs when processes try to get access to resources and there is some type of conflict or lack of resources
- Deadlock Characterization
  - Mutual Exclusion - process at a time can use a resource
  - Hold & wait - process holding at least one resource is waiting to acquire additional resources held by other processes
  - No preemption - resource released only voluntarily by process holding it, after that process has completed its tasks
  - Circular wait (cycle)- exists a set of waiting processes such that they are all connected to one another by the original process holder.
    - If this exists, it is a major warning that there is a deadlock in a system
- Resource Allocation Graph
  - No deadlock

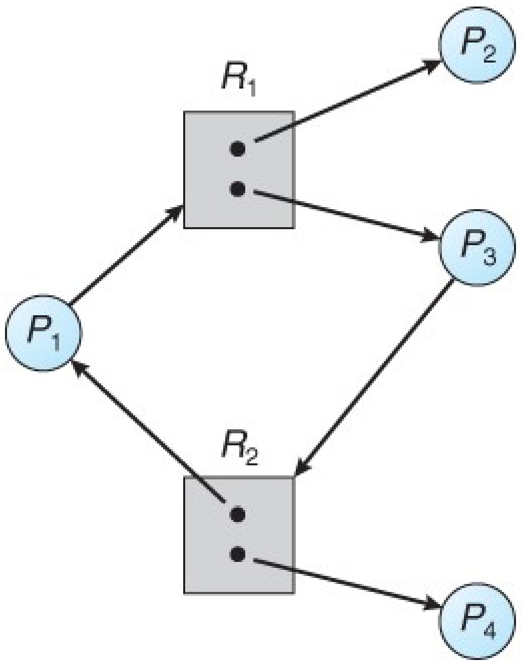


    -
  - With Deadlock



•

•Cycle w/o Deadlock



•

•Basic facts about cycles in a graph

- •If graph contains no cycles ⇒ no deadlock
- •If graph contains a cycle ⇒
  - •if only one instance per resource type, then deadlock
  - •if several instances per resource type, possibility of deadlock

•Methods for handling deadlocks

- •Based off the characterizations of deadlocks, deadlock will occur if all of them occur simultaneously.

- Deadlock prevention
  - Prevent at least one of the conditions from ever becoming true
  - Allow the system to enter deadlock state, detect it, and then recover
- Deadlock avoidance
  - Have a priori information on how each process will be utilizing the resources to decide whether or not the system is the safe state for each resource allocation.

•Safe and Unsafe states
- Safe – Not in danger of deadlock
- Unsafe - In danger of deadlock as the conditions of deadlock are met.

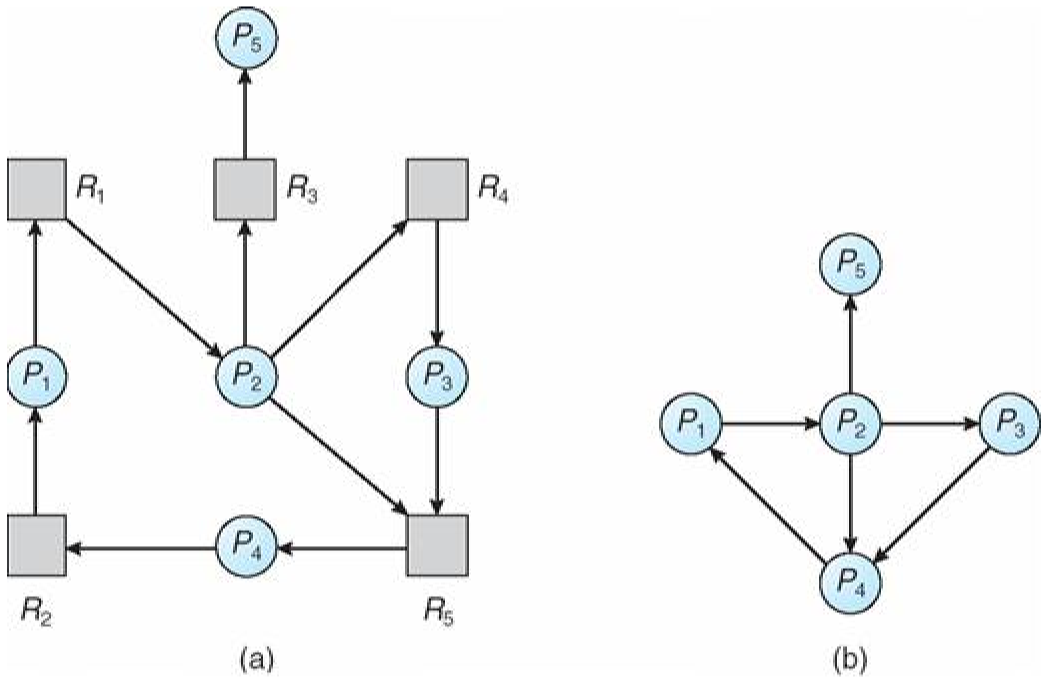
•Avoidance algorithms - trying to avoid deadlock in general by using algorithms to determine what to do next, single-instance resources vs multiple instance resources, each one uses a different algorithm from the beginning
- Single instance of a resource type – use resource allocation graph
  - One of the main requirements of this scheme to ensure deadlock never occurs is that the processes declare in advance, the maximum number of resource types that they may need.
  - When one becomes an assignment edge for a resource allocation graph on slide 25, it causes a cycle and an unsafe state. But two claim edges makes the graph in safe state
  - multiple instances of a resource type – use the Banker’s algorithm
    - Each process must state a priori claim maximum use
    - This algorithm refuses to do the allocation even if a process requests an open resource becuase allocating it, could cause it from safe state to unsafe state, or unsafe state to deadlock
    - Timer used to deallocate resources after a certain amount of time
    - # of resources, m = # of resource types
    - Available
    - Max
    - Allocation

        •Need
        •Look here at slides for more information, slide 28

•Claim edge in a resource allocation graph scheme

•Deadlock detection algorithms:

- Single instance resource type – wait-for-graph



-

- Figure a is the resource allocation graph and figure b is the corresponding wait-for-graph

- Multiple instance resource type – Banker’s algorithm

•Detection-Algorithm usage

    •When and how often deadlock occurs?
    •If detection algorithm is invoked arbitrarily, there may be many cycles in the resource graph and so we would not be able to tell which of the many deadlocked processes “caused” the deadlock.

•Recovery from deadlock:

- Process termination
- Abort all dead processes
- abort one process at a time until deadlock cycle is eliminated
- Resource preemption
- selecting victim to minimize cost
- Cost refers to the order we abort in bullet points

- rollback - return to some safe state, restart process for that state
- Starvation - same process may always be picked as victim, include # of rollback in cost factor

Ch. 8 (Main Memory) Study Guide

- •What is the main purpose of memory management and the role of base and limit registers for this process?
  - •used to accomplish goal of ensuring correct operation in the correct address
- •Hardware address protection
  - •Making sure that the virtual address that the CPU has created, is out of the range of the base of another address but still within the limit of that address as well.
- •Different stages of address binding of instructions and data to memory
  - •there is some support that allows the programs to not run from the beginning of address 000.
  - •compiled code can bind to relocatable addresses
    - •relocatable means that it can be binded to an existing and moving module
    - •addresses are not exactly the same when code is compiled, it will vary
- •Absolute code
  - •The code is at a specific location where it is not changed
- •Relocatable code
  - •relocatable means that it can be binded to an existing and moving module
- •Logical (or virtual) address vs. physical address
  - •Logical address - generated by CPU, also referred to as a virtual address
  - •Physical address - address seen by the memory unit
- •Memory Management Unit (MMU) & relocation register
  - •Hardware device that maps virtual to physical addresses
  - •Base Register - relocation register
  - •Relocation register is the same as the base register
- •Dynamic linking
  - •Static linking - system libraries and program code combined by the loader in to the binary program image
  - •Dynamic linking - linking postponed until execution time
  - •Stub - small piece of code to locate appropriate memory resident library routine

    - stub replaces itself with the address of the routine and carries out the routine
- Contiguous allocation
    - main memory must be capable of supporting OS and user processes
    - Two partitions:
        - Resident Operating System, held in low memory w/ interrupt vector
        - user processes then held in high memory
        - Each process contained in single contiguous section of memory
- Multiple-Partition allocation
    - improve upon contiguous allocation
    - Don't want a small process to be allocated to a large address space of a process, else it is just wasteful
    - Hole - block of available memory; holes of various size are scattered throughout memory
- Dynamic storage-allocation (First-fit, Best-fit & Worst-fit)
    - first fit- allocate first hole that is big enough
    - best-fit- allocate the smallest hole that is big enough; must search entire list unless ordered by size
    - worse case: allocate largest hole; must search entire list
        - produce largest leftover hole
- Fragmentation (External & Internal)
    - External Fragmentation - total memory space exists to satisfy request but it's not contiguous
        - allow process to be loaded into slightly larger hole, whatever the leftovers is cannot be touched
    - Internal Fragmentation - allocated memory may be slightly larger than requested memory; size difference is memory internal to a partition, but not being used.
        - Resources scattered everywhere, nobody can touch it, it is internal to the process
        - Cannot be touched/modified, only be touched/shared by other processes
- Compaction
    - Takes all the free memory together and put it into one block
    - only possible if relocation is dynamic and done at execution time
- Segmentation

- when writing a program, there are several segments to it (main program, procedure, function, method, etc...)
- memory management scheme that supports user view of memory

- Paging (briefly describe it and give example)
  - physical address space of a process can be noncontiguous; process is allocated to physical memory whenever latter is available
    - avoids external fragmentation
    - avoids problem of varying sized memory chunks
- Implementation of Page Table
  - Page-table base register (PTBR) points to page table
  - Page-table length register (PTLR) inidicates size of page table
  - scheme - every data/instruction access requries two memory accesses
    - solved by associative memory
- Transaction Look-aside buffers (TLBs)
  - special fast-lookup hardware
- Memory protection (valid-invalid) bit
  - valid bit - protection bit used to whether or not the process is read/write, or read & write
    - valid - in logical adress space
    - invalid - not in logical address space
- Shared pages
  - Shared code - one copy of read-only shared among processes
    - similar to multiple-threads or sharing the same process space
  - Private Code and data
    - each process keeps code of code and data
    - have separate page tables
- Structure of the page table (Hierarchical, hashed and inverted)
  - Hierarchial Pages
    - break up logical address space into multiple page tables
    - simple technique is a two level page table for 32 bit
    - page that page table
  - Hashed Page Tables
    - Virtual page number is hashed into a page table
    - Each element contains virtual page #, value of mapped page frame, pointer to next element
    - similar to linkedlist

- Inverted Page Tables
    - looks for additional information about the process that the page belongs to
    - because there is no page table that is keeps track of all the pages, there will be an increase in time needed to search the table when a page refrence occurs
    - To fix this, a hash table can limit the search to one, and can use TLB

Ch. 9 (Virtual Memory)

Basic Concept: Code needs to be in memory to execute, but the entire program not needed at the same time and some parts rarely used (i.e., error code, unusual routines, large data structures).

Virtual Memory:

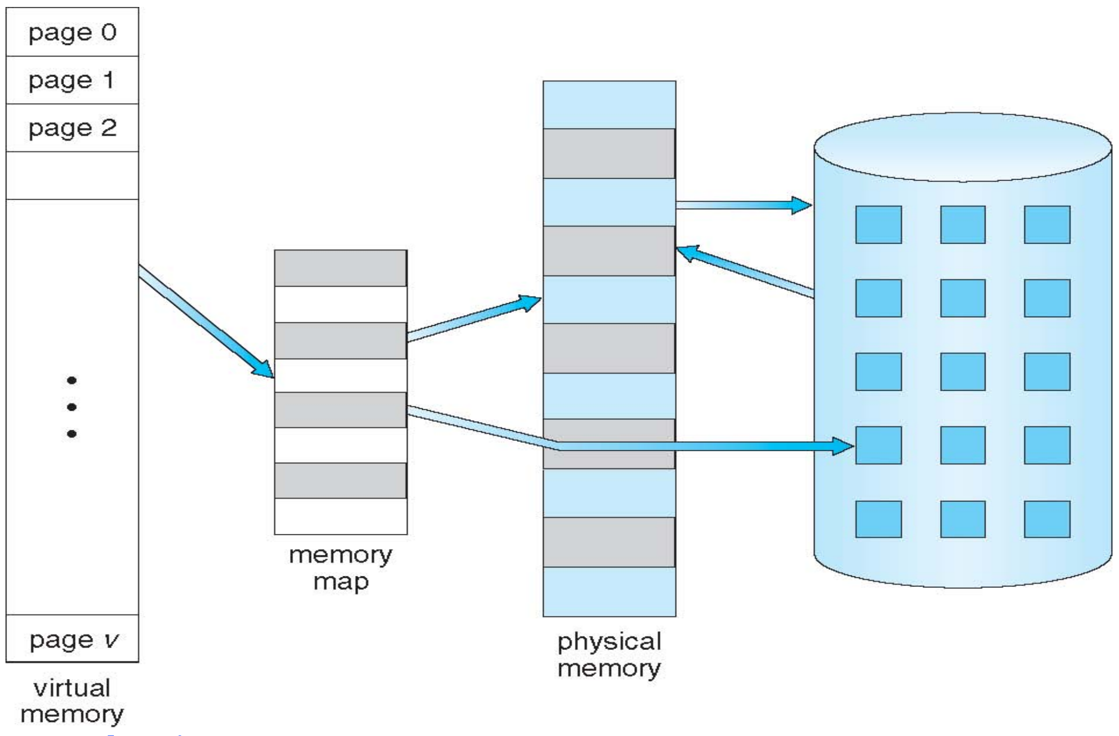
- Allows execution of partially-loaded program in memory.
- Programs no longer constrained by limits of physical memory
- Since each program takes less memory while running -> more programs run concurrently, resulting in increased CPU utilization and throughput with no increase in response time or turnaround time.
- Allows for more efficient process creation
- Less I/O needed to load or swap programs into memory -> each user program runs faster

Virtual address space – logical view of how process is stored in memory

- Usually start at address 0, contiguous addresses until end of space
- Physical memory organized in page frames
- MMU must map logical to physical

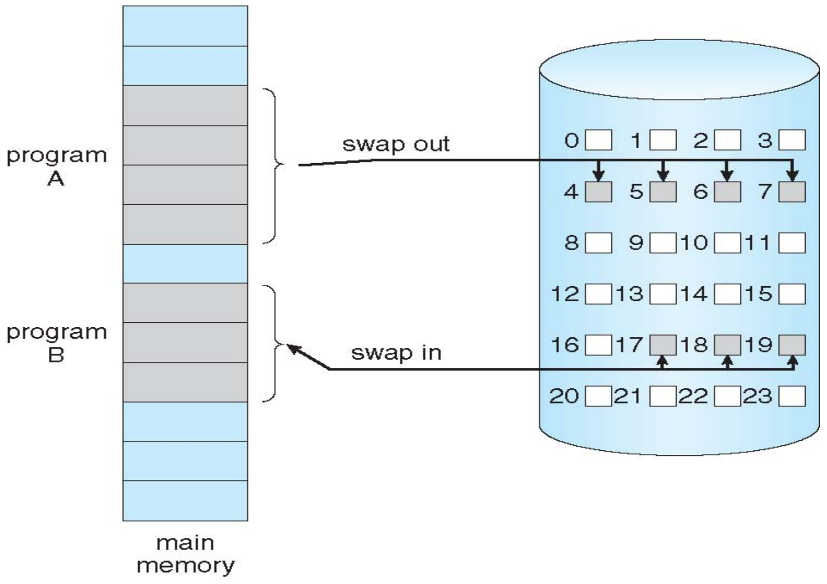
Virtual memory can be implemented via**:**
- **Demand paging**
- **Demand segmentation**

Example of virtual memory that is larger than physical memory



Demand Paging

Bring a page into memory only when it is needed



Page is needed ⇒ reference to it

- **invalid reference ⇒ abort** **(that's when page does not exist in process' address space)**
- **not-in-memory ⇒ bring to memory** **(that means process does exist in process' address space, but it's on disk)**

Lazy swapper – never swaps a page into memory unless page will be needed
Swapper that deals with pages is a **pager**

With swapping, pager guesses which pages will be used before swapping out again
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**

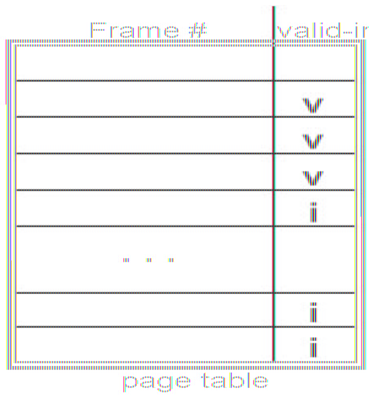
- No demand paging needed

If page needed and not memory resident

- Need to detect and load the page into memory from storage

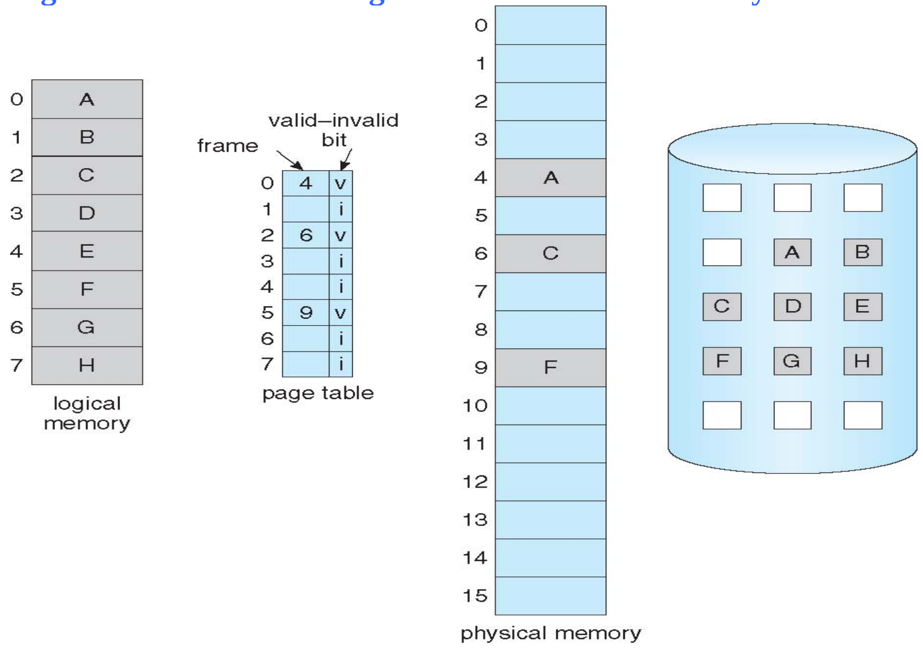
Valid-Invalid Bit

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



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

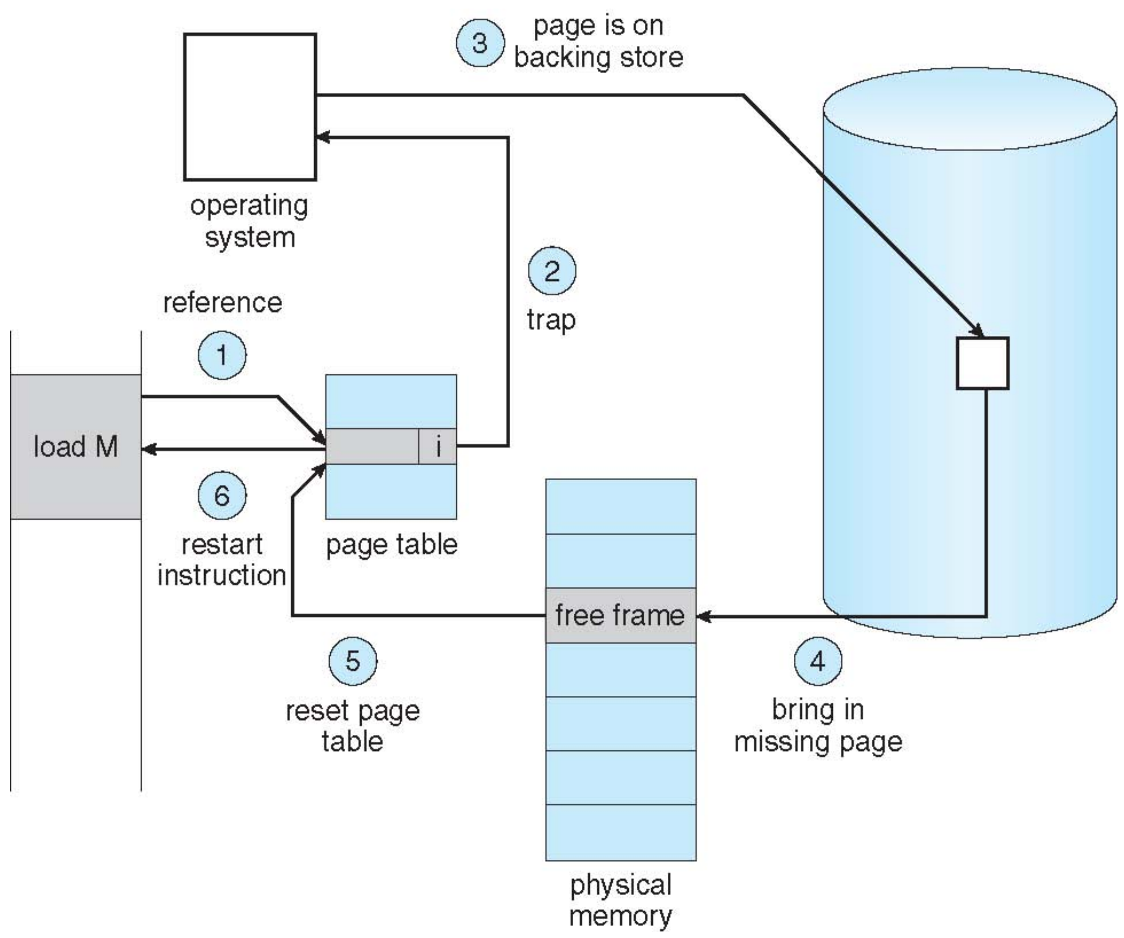


Page Fault

If there is a reference to a page, first reference to that page will
trap to operating system: **page fault**

1. Operating system looks at another table to decide:
   - Invalid reference ⇒ abort
   - Just not in memory
2. Find free frame
3. Swap page into frame via scheduled disk operation
4. Reset tables to indicate page now in memory & set validation bit = **v**
5. Restart the instruction that caused the page fault

Steps in Handling a Page Fault

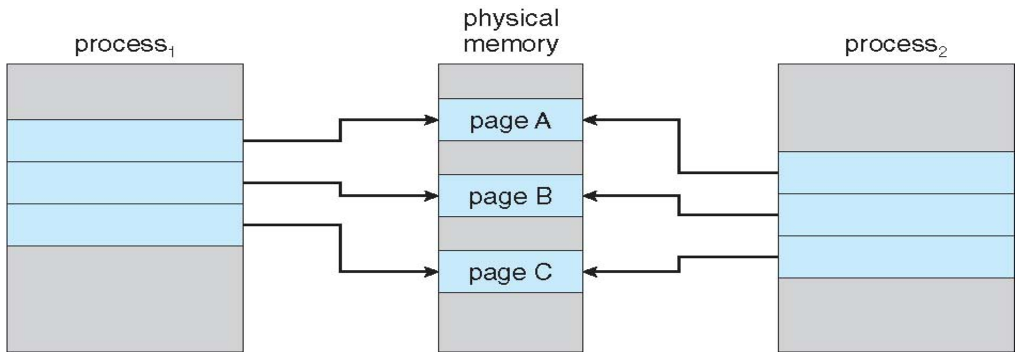


<u>Extreme case:</u> – start process with *no* pages in memory - **Pure demand paging**

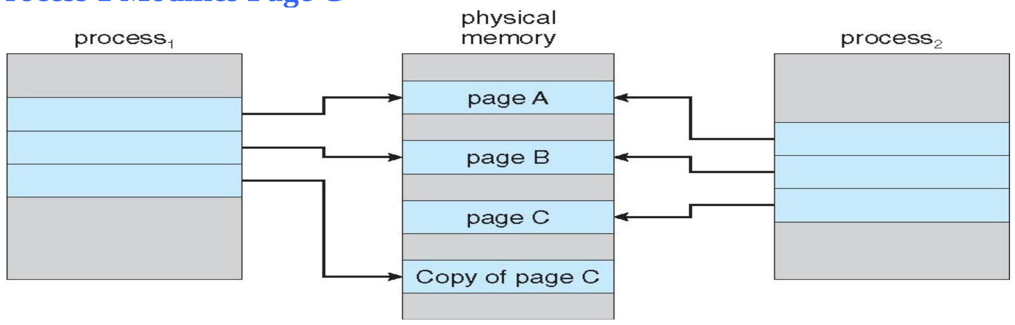
Copy-on-Write

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

Before Process 1 Modifies Page C

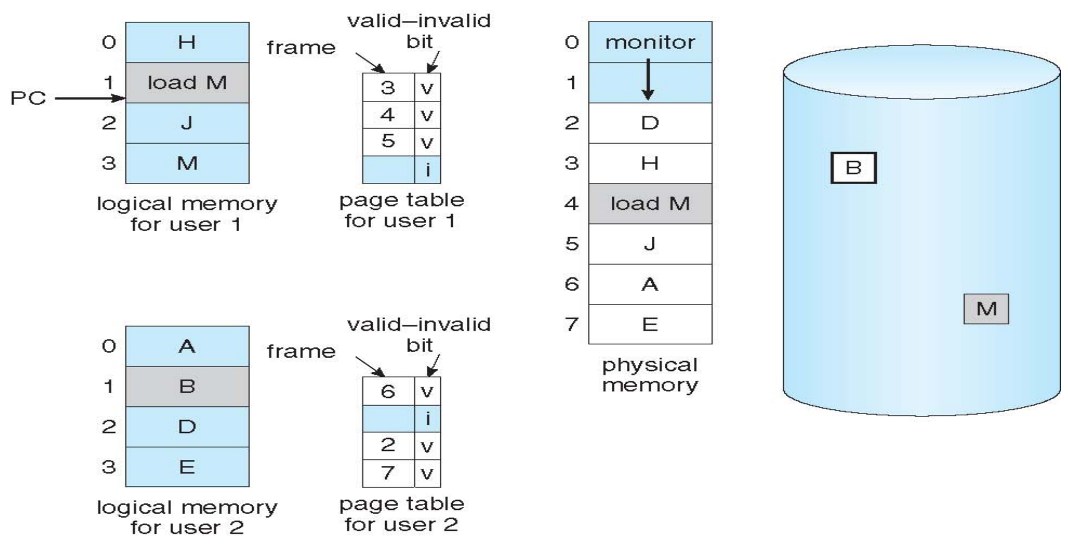


After Process 1 Modifies Page C



What happens if there is no Free Frame?

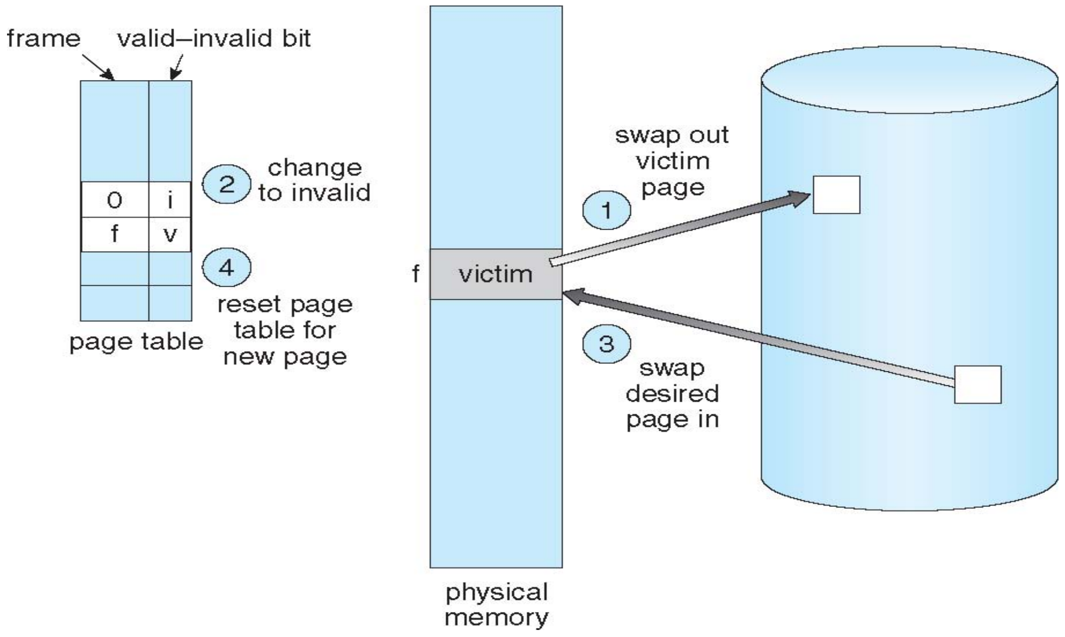
- Used up by process pages
- Also in demand from the kernel, I/O buffers, etc
- Page replacement – find some page in memory, but not really in use, page it out
  - Algorithm – terminate? swap out? replace the page?
  - Performance - want an algorithm which will result in minimum number of page faults
- Same page may be brought into memory several times
- 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



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



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

Allocation of Frames
- Each process needs *minimum* number of frames
- *Maximum* of course is total frames in the system
- Two major allocation schemes
  - fixed allocation
  - priority allocation
- Many variations

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
  - 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
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 ≡ a process is busy swapping pages in and out, resulting in overhead.