CSE 330 – Final Exam Format

[1. Processes and Threads 4](#_Toc7385379)

[PCB 4](#_Toc7385380)

[Process state transition diagram 4](#_Toc7385381)

[Process Image 4](#_Toc7385382)

[Context Switching 4](#_Toc7385383)

[KLTs 4](#_Toc7385384)

[ULTs 4](#_Toc7385385)

[Hybrid 4](#_Toc7385386)

[2. Process Synchronization 4](#_Toc7385387)

[CPU Burst 4](#_Toc7385388)

[CPU Scheduler (short term scheduler / dispatcher) 4](#_Toc7385389)

[o Dispatcher: 5](#_Toc7385390)

[Preemptive vs Non-preemptive Scheduling 5](#_Toc7385391)

[o Non-preemptive Scheduling: 5](#_Toc7385392)

[o Preemptive Scheduling: 5](#_Toc7385393)

[Dispatch Latency 5](#_Toc7385394)

[Scheduling Criteria – User vs. System: 5](#_Toc7385395)

[ CPU Utilization: 5](#_Toc7385396)

[ Throughput: 5](#_Toc7385397)

[ Turnaround Time: 5](#_Toc7385398)

[ Waiting Time: 6](#_Toc7385399)

[ Response Time: 6](#_Toc7385400)

[Scheduling Algorithms 11](#_Toc7385401)

[o Gantt chart 11](#_Toc7385402)

[o FCFS – First Come First Serve 11](#_Toc7385403)

[o SJF – Shortest Job First 11](#_Toc7385404)

[o SRTF – Shortest Remaining Time First 11](#_Toc7385405)

[o RR, given time quantum 12](#_Toc7385406)

[o Priority 12](#_Toc7385407)

[o Multi-level Queue 12](#_Toc7385408)

[o Multi-level Feedback Queue 12](#_Toc7385409)

[Thread scheduling: process vs system-contention scope 6](#_Toc7385410)

[o Contention Scope 6](#_Toc7385411)

[o Pthread Scheduling 6](#_Toc7385412)

[Multi-processor scheduling: asymmetric vs symmetric multi-processor scheduling 6](#_Toc7385413)

[o Asymmetric Scheduling 6](#_Toc7385414)

[o Symmetric Scheduling 6](#_Toc7385415)

[Multi-core processors: memory stalls, load balancing (push vs. pull strategies), processor affinity 7](#_Toc7385416)

[o Multi-core Processor Architectures 7](#_Toc7385417)

[o Memory Stalls 7](#_Toc7385418)

[o Load Balancing 7](#_Toc7385419)

[o Processor Affinity 7](#_Toc7385420)

[Heterogeneous Multi-processing 7](#_Toc7385421)

[Real-time CPU scheduling: soft vs. hard real-time systems 7](#_Toc7385422)

[Soft real-time systems 7](#_Toc7385423)

[Hard real-time systems 7](#_Toc7385424)

[Earliest-deadline-first (EDG) scheduling 7](#_Toc7385425)

[Chapter 7 – Synchronization 8](#_Toc7385426)

[Bounded-buffer problem (Producer-Consumer Problem) 8](#_Toc7385427)

[Readers-writers problem 8](#_Toc7385428)

[Dining philosophers problem 9](#_Toc7385429)

[Synchronization problems such as cat-dog problem on midterm 2 10](#_Toc7385430)

[3. Deadlock 10](#_Toc7385431)

[Starvation vs. Deadlock 10](#_Toc7385432)

[Conditions 10](#_Toc7385433)

[Prevention 11](#_Toc7385434)

[Avoidance 11](#_Toc7385435)

[Detection 11](#_Toc7385436)

[4. CPU Scheduling 11](#_Toc7385437)

[Something like the question on midterm 2 11](#_Toc7385438)

[5. Virtual Memory 13](#_Toc7385439)

[**Virtual Memory**: separation of logical memory, perceived by users, from physical memory 13](#_Toc7385440)

[**Swapping**: moving information from memory to a backing store temporarily 14](#_Toc7385441)

[**Memory Allocation:** 14](#_Toc7385442)

[**Paging**: memory-management scheme in which physical address space of processes may be non-contiguous 15](#_Toc7385443)

[**Segmentation**: memory-management scheme in which logical address space is a collection of segments, each with a name and length 21](#_Toc7385444)

[Paged Segmentation 21](#_Toc7385445)

[6. Page Replacement 22](#_Toc7385446)

[Basic Page Replacement 22](#_Toc7385447)

[Page replacement algorithms 22](#_Toc7385448)

[FIFO (First-in-First-out) Page Replacement: 22](#_Toc7385449)

[LRU (Least Recently Used) Page Replacement: 23](#_Toc7385450)

[ARC (Adaptive Replacement Cache) Page Replacement: 23](#_Toc7385451)

[Paged segmentation 24](#_Toc7385452)

[7. Mass-storage Structures, I/O Systems 24](#_Toc7385453)

[Access times: HDD, RAID, SSD 24](#_Toc7385454)

[HDDs 24](#_Toc7385455)

[RAID 24](#_Toc7385456)

[SSD/Flash 25](#_Toc7385457)

[Disk Scheduling 25](#_Toc7385458)

[Polling 25](#_Toc7385459)

[Interrupt-driven 25](#_Toc7385460)

[DMA 25](#_Toc7385461)

[Polling / Interrupt-driven / DMA 26](#_Toc7385462)

[8. File System Interface & Implementation 26](#_Toc7385463)

[FCB (File Control Block) 26](#_Toc7385464)

[Allocating logical space on a physical disk 26](#_Toc7385465)

[Keeping track of free blocks 27](#_Toc7385466)

[9. Principle of Locality 28](#_Toc7385467)

[10. OMP and Pthreads 30](#_Toc7385468)

[Write a multithreaded program using OMP & Pthreads 30](#_Toc7385469)

# 1. Processes and Threads

## PCB

## Process state transition diagram

## Process Image

## Context Switching

### KLTs

### ULTs

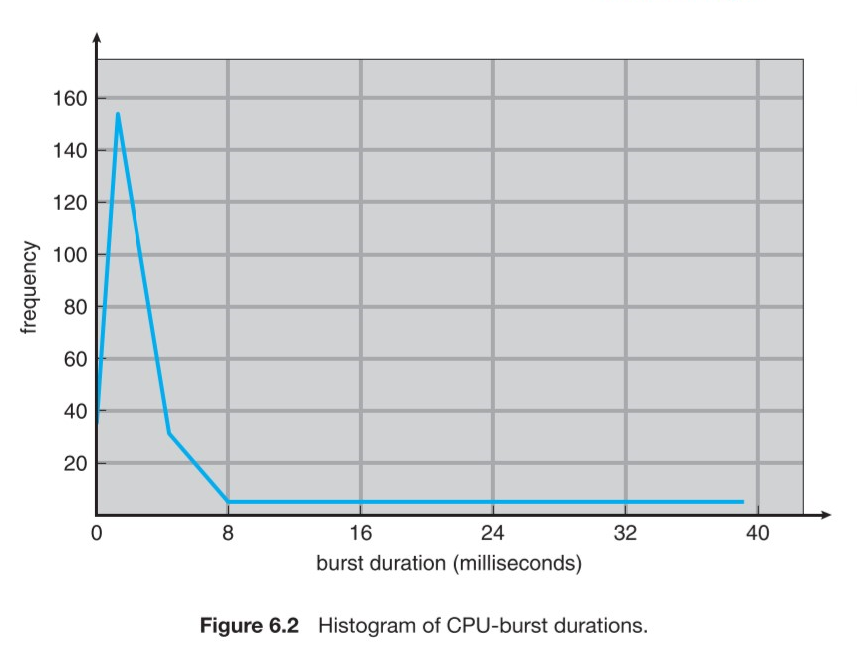
### Hybrid

# 2. Process Synchronization

## CPU Burst

The execution of a process by the CPU

Most processes have a CPU burst where they perform computations, followed by an I/O burst, where they access I/O devices.



## CPU Scheduler (short term scheduler / dispatcher)

* When CPU becomes idle, the OS must select a process in the ready queue to execute
  + This is performed by the **short-term scheduler**
* Scheduling decisions may take place under four circumstances:
  1. Process switches from running state to waiting state
  2. Process switches from running to ready state
  3. Process switches from waiting to ready state
  4. Process terminates

### Dispatcher:

* + The component in CPU scheduling that gives control of the CPU to the process selected by the short-term scheduler. Has the following functions:
    - Switching context
    - Switching to user mode
    - Jumping to the proper location in the user program to restart it
* The dispatcher MUST be designed to be as fast as possible because it is invoked during every process switch.

## Preemptive vs Non-preemptive Scheduling

### Non-preemptive Scheduling:

* + scheduling decisions only take place under 1 and 4 above (in CPU Scheduler).
  + Once time has been allocated to a process, it has the CPU time to use, regardless of what it chooses to do with it.

### Preemptive Scheduling:

* + the opposite of non-preemptive scheduling; scheduling decisions can take place under all 4 conditions above.

## Dispatch Latency

The amount of time it takes for the dispatcher to stop one process and start another.

## Scheduling Criteria – User vs. System:

### CPU Utilization:

* + attempt to keep the CPU as busy as possible

### Throughput:

* + Number of processes completed per time unit

### Turnaround Time:

* + how long it takes to execute a process; the sum of periods waiting to get into memory, executing on the CPU, waiting in the ready queue, and I/O.

### Waiting Time:

* + the CPU scheduling algorithm does not affect the amount of time that a process spends waiting in the ready queue; is the sum of the periods spent waiting in the ready queue.

### Response Time:

* + the time from the submission of a request until the first response is produced. It is the time It takes to start responding, not the time it takes to output a response.

## Thread scheduling: process vs system-contention scope

### Contention Scope

* + One distinction between user and kernel threads is in scheduling
  + **Process Contention Scope:** user-level threads; threads fight for CPU time for the same process
  + **System Contention Scope:** kernel-level threads; the kernel can decide how to share CPU time between the threads

### Pthread Scheduling

* + Pthreads allows process contention scope as well as system contention scope
  + PTHREAD\_SCOPE\_PROCESS schedules a thread using PCS
  + PTHREAD\_SCOPE\_SYSTEM schedules a thread using SCS
  + If a system allows the many-many model, PTHREAD\_SCOPE\_PROCESS will perform the many to many model
  + pthread\_attr\_getscope(pthread\_attr\_t\* attr, int scope) <- Gets the current scope
  + pthread\_attr\_setscope(pthread\_attr\_t\* attr, int\* scope) <- Sets the scope of the thread

## Multi-processor scheduling: asymmetric vs symmetric multi-processor scheduling

### Asymmetric Scheduling

* + One master CPU/core (the server) should handle the following:
    - Scheduling decisions
    - I/O processing
    - Other system activities
  + Simple, because only one CPU accesses the system’s data structures, reducing the need for data sharing and synchronization

### Symmetric Scheduling

* + Each processor / core is self-scheduling
  + Processes may be in a common ready queue / each processor may have its own private queue of ready processes
  + Scheduler must be programmed carefully to prevent data sharing and corruption issues

## Multi-core processors: memory stalls, load balancing (push vs. pull strategies), processor affinity

### Multi-core Processor Architectures

* + One core / Multiple CPU
  + Multiple cores / Single CPU
  + Multiple threads / Multiple cores

### Memory Stalls

* + Waiting for data from RAM that is not in cache memory
  + Caused by a cache miss or other reasons

### Load Balancing

* + Attempting to distribute the workload evenly between all processors in an SMP system

#### Push Strategy

* + - A specific task periodically checks the load on each processor and rebalances if it there is an imbalance
    - Processes are “pushed” from an overloaded to an idle / less busy CPU

#### Pull Strategy

* + - Idle processor “pulls” a waiting task from a busy processor
  + Often, the two strategies are used in conjunction

### Processor Affinity

* + A process has an affinity for the processor on which it is currently running -> the process can only run on said processor.

## Heterogeneous Multi-processing

Example: ARM processors often have the big.LITTLE architecture, which consists of a set of powerful processors for general runtime use, plus some processors which use less power for other tasks.

## Real-time CPU scheduling: soft vs. hard real-time systems

### Soft real-time systems

* Provide no guarantee as to when a critical real-time process will be scheduled

### Hard real-time systems

* Have stricter requirements; a task MUST be serviced before its deadline

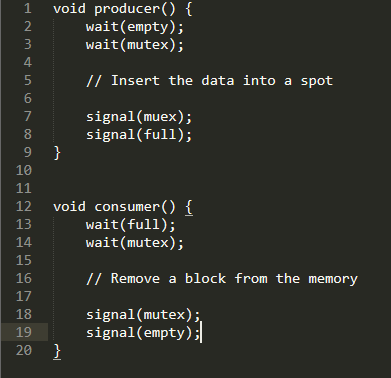
### Earliest-deadline-first (EDG) scheduling

* Whichever process has the earliest deadline should be serviced first

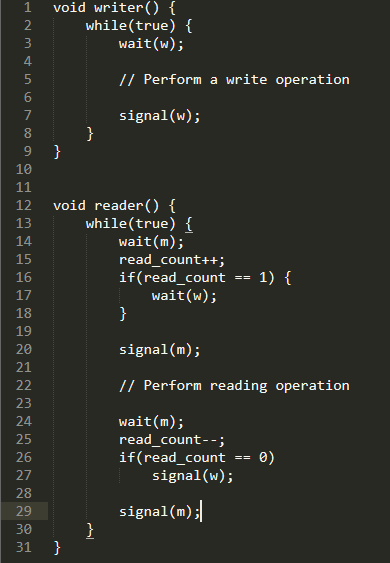
## 

## Chapter 7 – Synchronization

### Bounded-buffer problem (Producer-Consumer Problem)

* There is a buffer of slots, each of which can store one unit of data
* Two processes are running – the **producer** and the **consumer**
* A producer tries to insert data into an empty slot of the buffer
* A consumer tries to remove data from a filled slot in the buffer
* This creates a race condition when the producer and consumer are executing concurrently
* Solution to the problem:
  + 
  + Producer
    1. Waits until there is at least one empty slot
    2. Decrements the empty semaphore because one block will be inserted
    3. Acquires mutual exclusion to the buffer
    4. Inserts the block
    5. Releases from ME and increments full
  + Consumer
    1. Waits until one full slot is in the buffer
    2. Decrements the full semaphore because a slot is emptied
    3. Consumer acquires ME for the buffer
    4. Consumer removes a block from the buffer
    5. Consumer releases ME for the buffer
    6. Empty semaphore is incremented by one because an empty slot has been created

### Readers-writers problem

* One process can write at a time
* Multiple processes can read at a time
* Mutex m and semaphore w are used; an integer variable reader\_count is used to track the number of readers accessing the resource
* read\_count is initialized to 0; m and w are initialized to 1
* Solution:
  + 
* The write worker waits until w is available
* is acquired whenever read\_count is updated by a process
* When a reader wants to access the resource, it increments read\_count, then accesses the resources, then decrements the read\_count value
* The semaphore is used by the first reader which enters the critical section and the last reader which exits the critical section
  + Allows the writer to unblock
  + When the last reader exits, it allows the writer to continue

### Dining philosophers problem

* There are 5 philosophers sitting around a circular dining table – each has five chopsticks and a bowl of rice in the middle
* At any instant, a philosopher is either thinking or eating
* When a philosopher wants to eat, he uses two chopsticks, one from his left, and one from his right
* When a philosopher wants to think, he keeps down both chopsticks in their original place
* A philosopher can think for an indefinite amount of time
* A philosopher can only eat for so long
* Solution:
  + 5 semaphores are needed, for each chopstick
  + Code for each philosopher: 
* Issues we Encounter
  + If all five philosophers are simultaneously hungry, and each of them pick up one chopstick, then a deadlock situation occurs; they will be waiting for another chopstick forever
  + Solutions:
    1. A philosopher must be allowed to pick up the chopsticks if only both left and right are available. (ME?)
    2. Allow only four philosophers to sit at a table so there is always a chopstick left on the table

### Synchronization problems such as cat-dog problem on midterm 2

# 3. Deadlock

When a waiting process is never able to change state because the requested resources are not available at a time; often caused because resources requested are held by other waiting processes

## Starvation vs. Deadlock

* **Starvation:** when a low priority process does not get access to resources it needs because there is a high priority process accessing the resources
  + The entire system of processes hasn’t come to halt in this case
* **Deadlock:** when all the processes do not get access to resources because every process is waiting for some other processes
  + Cycle in the RAG

## Conditions

* **Mutual Exclusion**: at least one resource must be held in a non-shareable mode – only one process can use it at one time; other processes must wait for that resource
* **Hold and Wait**: a process must be holding at least one resource and waiting to acquire additional resources
* **No preemption**: resources cannot be preempted – a resource can only be released voluntarily by the process holding it after that process completed its task
* **Circular Wait**: a cycle of processes waiting for resources before they can continue

## Prevention

* A set of methods to ensure at least one of the necessary conditions for deadlock CANNOT hold
  + Banker’s algorithm

## Avoidance

* The operating system must be given extra information in advance, concerning which resources a process will request and whether it should wait

## Detection

* The operating system can determine a deadlock if all processes are waiting on a resource to be readied up, but the processes holding on the resources are waiting too

# 4. CPU Scheduling

## Scheduling Algorithms

### Gantt chart

* + A bar-type chart which illustrates time usage
  + X-axis is time
  + Y-axis is task

### FCFS – First Come First Serve

* + The process that requests the CPU first is allocated the CPU first
  + Easily managed by a FIFO queue
  + The average waiting time under the FCFS policy tends to be quite long
  + **Convoy Effect:** all other processes wait for one long-running process to get off the CPU.

### SJF – Shortest Job First

* + Determines the next burst length for each process
  + Run processes which have the shortest burst length first
  + If there is a tie in the burst length of two processes, FCFS is used
  + Next CPU burst is often predicted as an exponential average of the measured lengths of previous CPU bursts

### SRTF – Shortest Remaining Time First

* + Determines which process requires the least compute time, and runs that first
  + If there is a tie, use FCFS

### RR, given time quantum

* + Designed for time-sharing systems
  + Like FSFS, but preemption is added to enable the system to switch between processes
  + A unit of time – **time quantum** – or **time slice** is defined – usually 10-100 ms in length
  + Ready queue is treated as a circular queue
  + New processes are added to the tail of the queue

### Priority

* + SJF is a type of priority-scheduling algorithm
  + A priority is associated with each process, and the CPU is allocated to the process with the highest priority.
  + Equal-priority processes are scheduled FCFS
  + Some systems use higher numbers to consider a higher priority; others use a lower number to consider a higher priority
  + Problems:
    - **Indefinite Blocking / Starvation:** a process is ready to run but is waiting for the CPU can be considered blocked
    - A heavy stream of higher-priority processes can prevent the consideration of other processes

### Multi-level Queue

* + Used when processes can be classified into different groups
  + Often, a division is made between **foreground** (interactive) processes, and **background** (batch) processes
    - Foreground / background applications have different response-time requirements – they have different scheduling needs
  + The ready queue is partitioned into multiple separate queues
  + Processes are “permanently” assigned to a queue, usually because of a process property such as memory size, process priority or process type
  + Each queue has its own scheduling algorithm
  + Scheduling over the queues often uses the FCFS algorithm

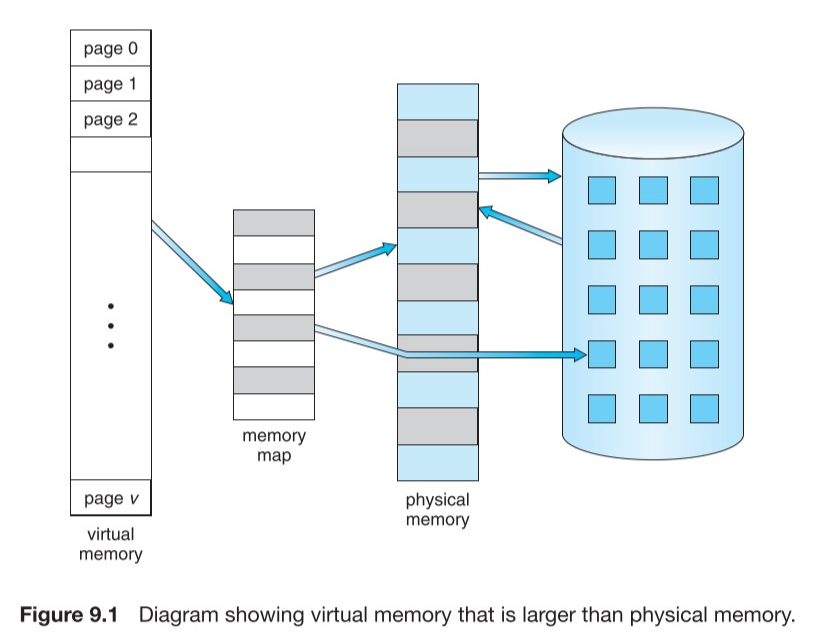
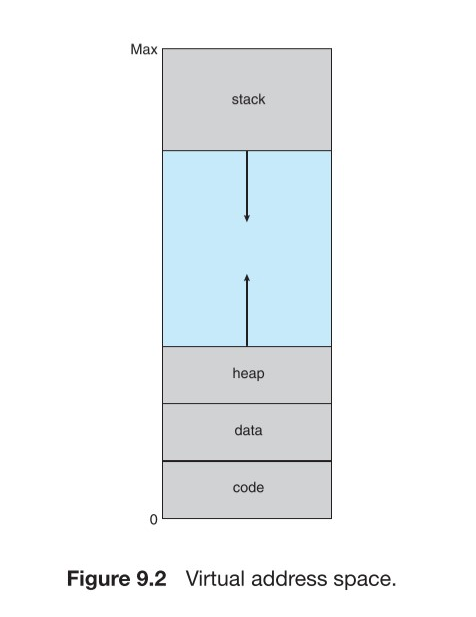
### Multi-level Feedback Queue

* + In multi-level queues, processes are permanently assigned to a queue when they enter the system
  + Now, we can move processes around, between the different queues
  + It a process uses too much CPU, it can be moved to a lower priority state
  + If a process is waiting too long, it may be moved up in its priority

# 5. Virtual Memory

* **Stall**: when the CPU must pause execution because it does not have the data necessary to complete its operation
* **Cache**: intermediate memory between the main memory and the CPU, which is significantly easier and faster to access
* We must ensure that each process has its own memory space – this ensures that the process cannot access memory that does not belong to it
  + Often implemented using a **base register**, which holds the first legal memory address and **limit register**, which holds the size of this address range
* **Logical Memory Addresses:** memory addresses seen by the processor
* **Physical Memory Addresses**: the actual memory addresses on physical hardware
* **Memory Management Unit (MMU)**: a hardware unit inside the CPU, which translates logical memory addresses to physical memory addresses
* **Backing Store**: Also known as **secondary memory**; holds pages that are not present on main memory
  + The secondary memory is often known as a **swap device** with space allocated as **swap space**. We can ”**swap**” pages on and off between this space and primary memory as necessary

## **Virtual Memory**: separation of logical memory, perceived by users, from physical memory

* Virtual memory is larger than physical memory: 
* Stack grows downwards (addresses subtract to grow the stack); heap [tends to] grow up from the code:
  + 
* **Page Fault**: thrown when a process accesses a page marked invalid; causes:
  + Memory might be in the backing store
  + Handling:
    1. Check internal table in PCB to determine whether the access was valid or not
    2. If the reference was invalid, terminate the process – illegal access!
    3. Find a free frame
    4. Request a backing store read of the memory location into memory
    5. Modify the page table to reflect a valid page
    6. Continue the process, retrying the failed instruction
  + Note that we MUST continue restart the instruction that caused the fault, after we restore its page (assume we did not terminate the process)
    - The process must be in the exact spot and state as before the fault

## **Swapping**: moving information from memory to a backing store temporarily

Used to increase the degree of multiprogramming in the system

Allows programs to run, which require more memory than available in the system

The system must maintain a **ready queue** consisting of whose memory images are on the backing store, or are in memory and ready to run

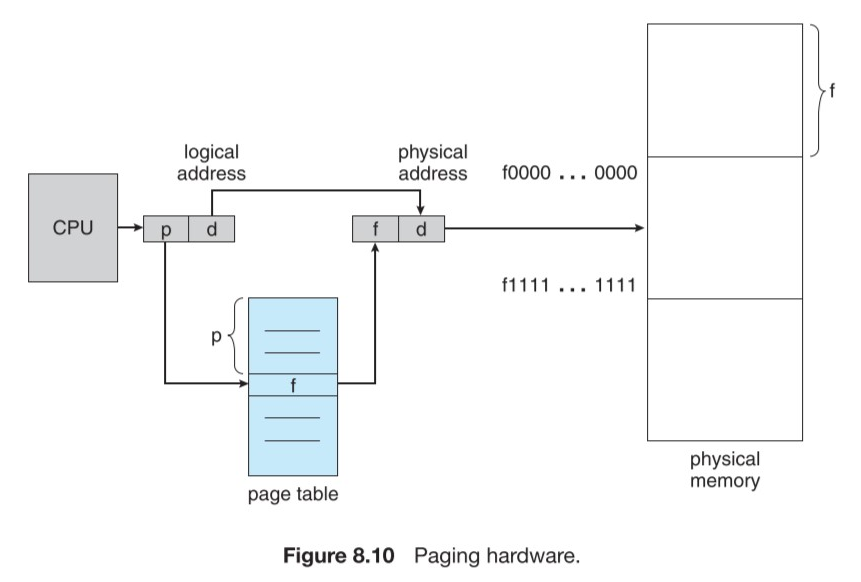
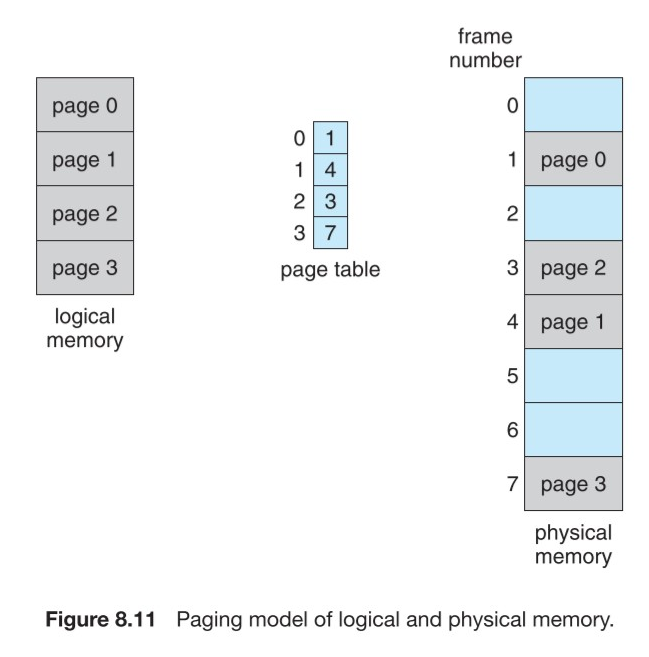
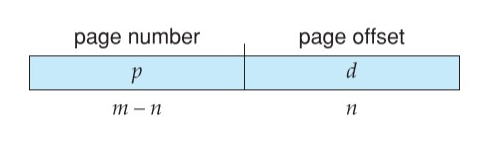
A process must be completely idle if it is to be swapped. If there is I/O pending, it cannot be swapped because the memory needs to be available to be written to

## **Memory Allocation:**

There are many methods through which we can allocate memory for different processes

* Divide memory into several fixed-sized **partitions**
  + Degree of Multiprogramming is limited by the number of partitions
* **Variable-partition scheme**:
  + OS keeps a table of which parts of memory are available / occupied
  + Initially, the entire memory is available to user processes, and is considered as one large **hole**
  + Eventually, memory will contain a set of holes of different sizes
  + When a process arrives, the system finds a hole that is large enough using the following possible methods – the **dynamic storage-allocation problem**:
    1. **First-fit**: allocate the first hole that is big enough
    2. **Best-fit**: find the smallest hole that is big enough
    3. **Worst-fit**: allocate the largest hole that is available
    4. Studies show that first- and best-fit algorithms work the best, but they both tend to be as good as each other; first-fit tends to be a bit faster.

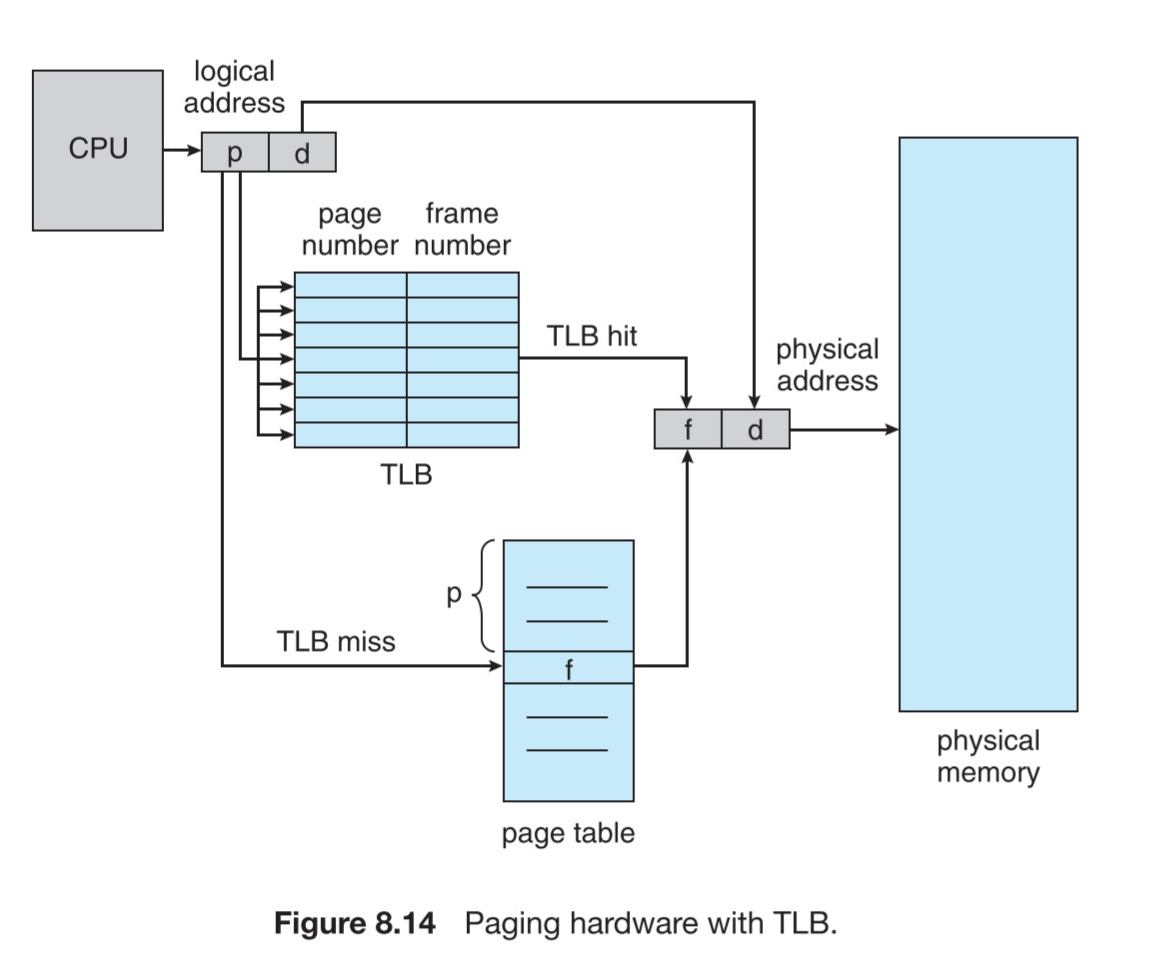
## **Paging**: memory-management scheme in which physical address space of processes may be non-contiguous

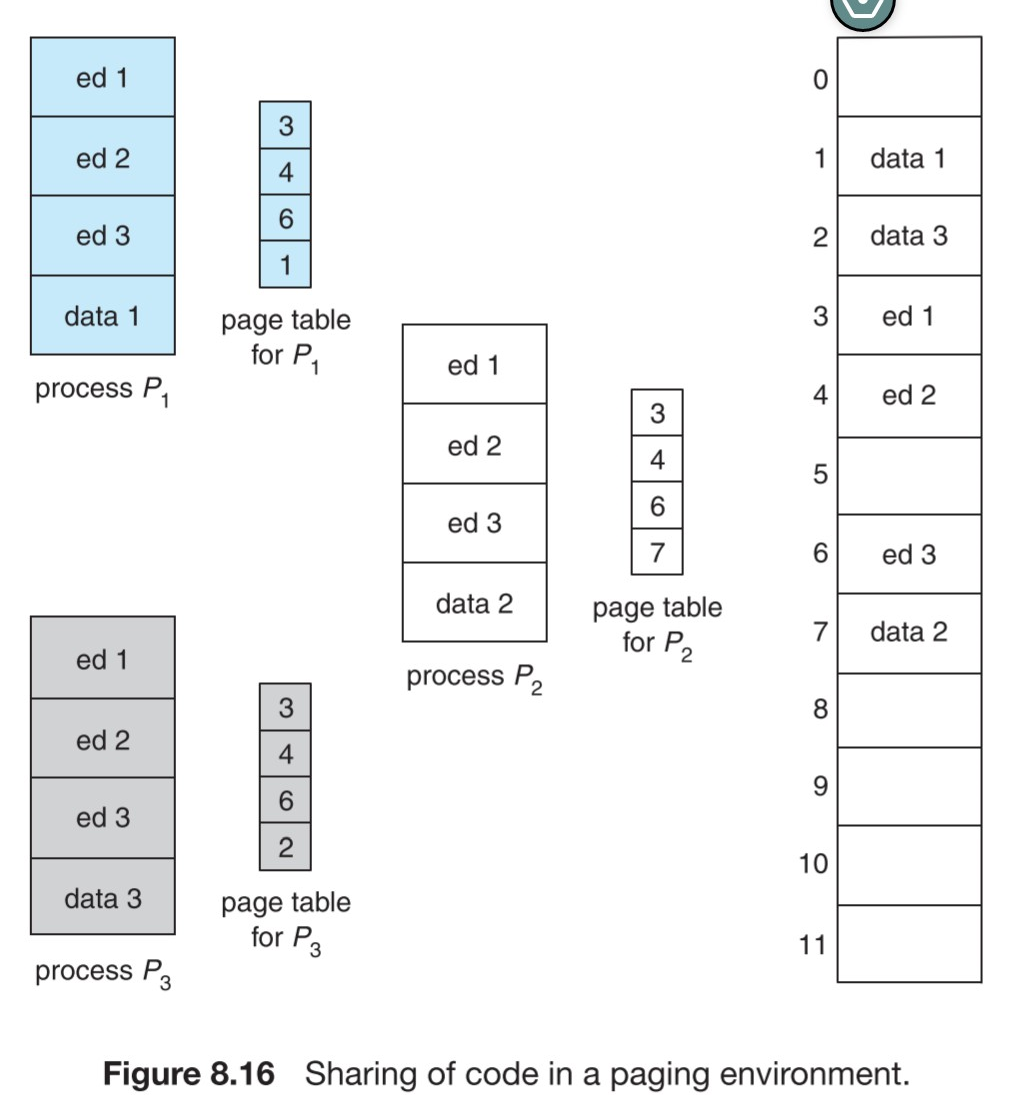
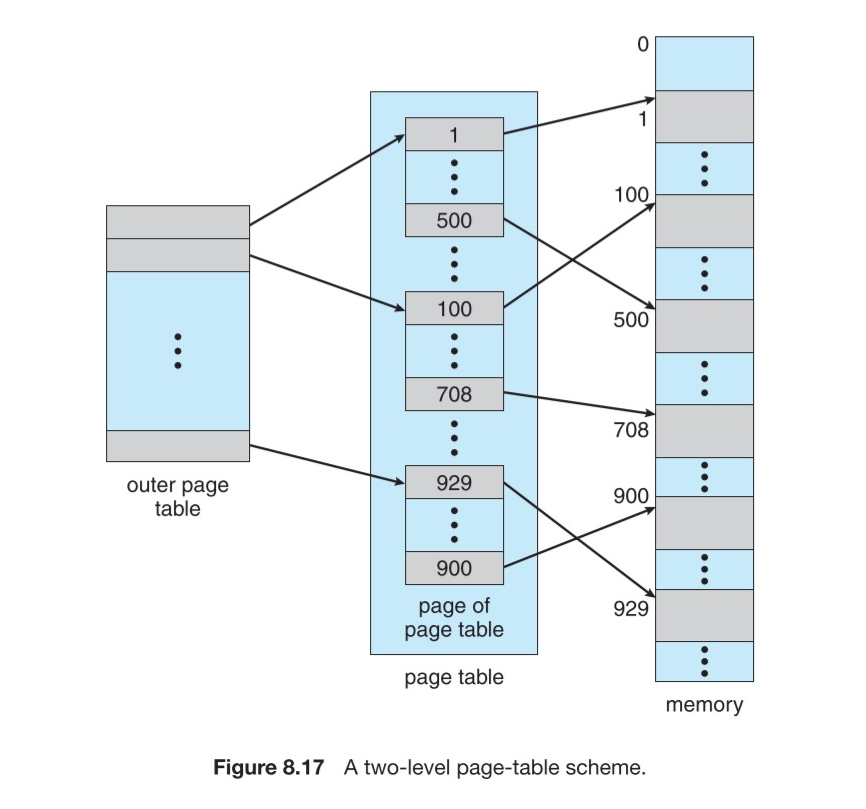
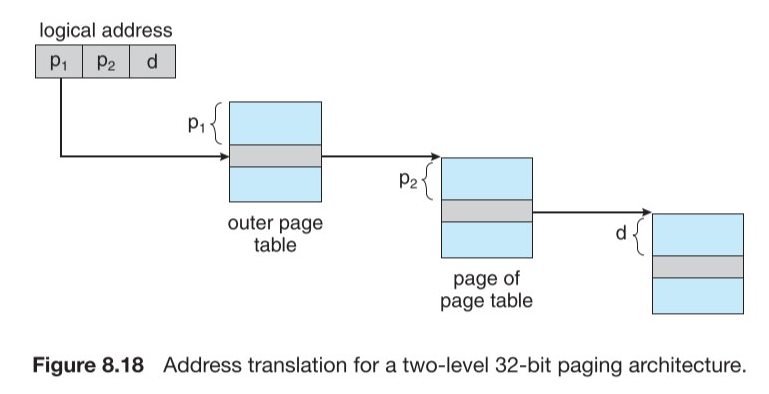
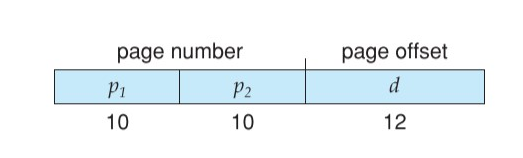
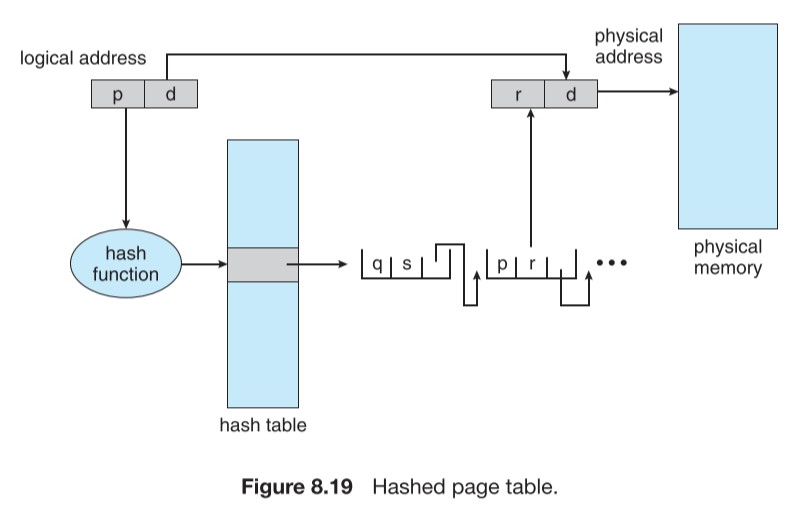
* Permits physical address space to be non-contiguous
* Does not require compaction, unlike segmentation
* Solves the problem of fitting memory chunks of variable sizes into the backing store
  + Backing store also has fragmentation issues, but access is significantly slower, so compaction is impractical
* **Basic Method of Paging**
  + Physical memory is broken into fixed-sized blocks – **frames**
  + Logical memory is broken into fixed blocks of the same size as the frames, called **pages**
  + When a process is to be executed, its pages are loaded into any available frames
  + The backing store is broken into fixed-size blocks of the same size as the frames – **blocks**
  + Hardware support is generated by dividing the address into two parts: the page number and the page offset :
  + The page number is used as an index for the **page table**
  + The **page table** contains the base address of each page in physical memory
  + How basic paging is implemented by hardware typically:
  + This setup makes it possible such that logical memory may look contiguous, but, is scattered throughout physical memory: 
  + **Paging in Hardware**: certain peripherals must exist on the memory bus on the CPU to enable paging support
    - The page size is defined by hardware; the size is a power of , usually between per page
    - A power of two is used so that:
      * upper bits of the address designate the page number
      * lower bits of the address designate the page offset
      * What an address looks like:
  + Issues of basic paging:
    - A process may require on a system with pages. This results in a requirement for . This must be rounded up to , or allocated, meaning a wastage, or **internal fragmentation** of .
      * A worst case would result in requiring an entire new page
    - **Important Calculations:**
      * where is the modulo operator
      * Page size must be
    - If process size isn’t dependent on page size, we can expect internal fragmentation to average at . This suggests we want to minimize page size
    - We cannot have infinitely small pages because we need to hold the page relationship in a table, which requires memory
  + Page tables
    - Different operating systems handle paging differently
    - Some OS allocate a page table for each process; the pointer to the page table is stored with other register values in the PCB
    - Page tables may be implemented with registers, or certain special memory addresses
      * Registers are practical if there are few pages in the table
      * Memory is practical if there is, for example, a couple million entries
        + **[ the following will definitely not be on the exam]**
        + Registers require 6 transistors / bit on high speed silicon logic, so they are incredibly expensive. You can’t have many registers of large sizes
        + Limits: ~1-4 KB of registers in a CPU usually
        + DRAM is usually not limited due to the price / space and special silicon processes to sacrifice speed for space.
        + Many architectures use SRAM for this.
      * A **page-table base register (PTBR)** points to the page table; to change, requires only one pointer to change
        + Reduces context switch time (changing 1 register basically takes one clock cycle 😊)
        + Memory access is slowed down by a factor of because we have to access memory twice – once for the table, to get the address, then to actually access the memory
        + We can speed up PTBR using a hardware lookup cache, the **translation look-aside buffer (TLB)**

Extremely fast

Usually implemented directly in the pipeline

**ASIDs (Address-space identifiers)** typically used to determine if the running process owns the page – failure results in **TLB miss**



* + - **Protection**
      * Certain protection bits are associated with each frame – usually kept in the page table
        + Usually determine read/write/execute permissions
        + Valid/invalid bits can state whether a process can access a page, or if a trap is to be thrown
    - **Shared Pages**
      * If code is **reentrant (the code doesn’t modify itself)**, it can be shared safely
      * 
* **Structure of the Page Table**
  + Most modern systems support large logical address spaces ( to bytes, sometimes even more!
  + A basic page table would become excessively large – to the point of difficulty in storage
  + In a 32-bit system with 4 KB , if each entry required 4 B, each process would require for its page table:
  + We want to divide the page table into smaller pieces; several ways exist
    - **Two-level Paging Algorithm**: the page table itself is also paged:
      * Logical address of the page table is divided into an inner and outer page number:
      * 
      * 
      * What the logical address breaks down into now:
      * Many architectures support two-level tables in hardware
    - Two-level paging is inappropriate for 64-bit systems. Even with an inner and outer table, the page tables would be unacceptably large
    - **Hashed Page Tables**: the hash is a virtual page number
      * The page table is a hash-map, where the key is the virtual page number, and the mapped value is the physical address
      * 
    - **Inverted Page Tables**: one page table for the entire system
      * Contains an entry for each frame in physical memory
      * Using information about the process, the logical address is calculated

## **Segmentation**: memory-management scheme in which logical address space is a collection of segments, each with a name and length

* The addresses specify both the segment name and the offset within the segment
* Segments are considered by a **segment number** instead of a segment name for simplicity
* Logical address, thus, are defined by a two-tuple:
* When a program is compiled, different segments are generated for:
  + Code
  + Global variables
  + Heap
  + Stack
  + Libraries
* Libraries linked in during compilation are assigned separate segments
  + The loader takes all segments and assigns them segment numbers
* **Segmentation Hardware**
  + The programmer refers to objects in a program using a 2-dimensional address
  + Memory is still one-dimensional
  + A **segment table** is used to map this two-dimensional space into memory
  + Each entry has a **segment base** and a **segment limit**

## Paged Segmentation

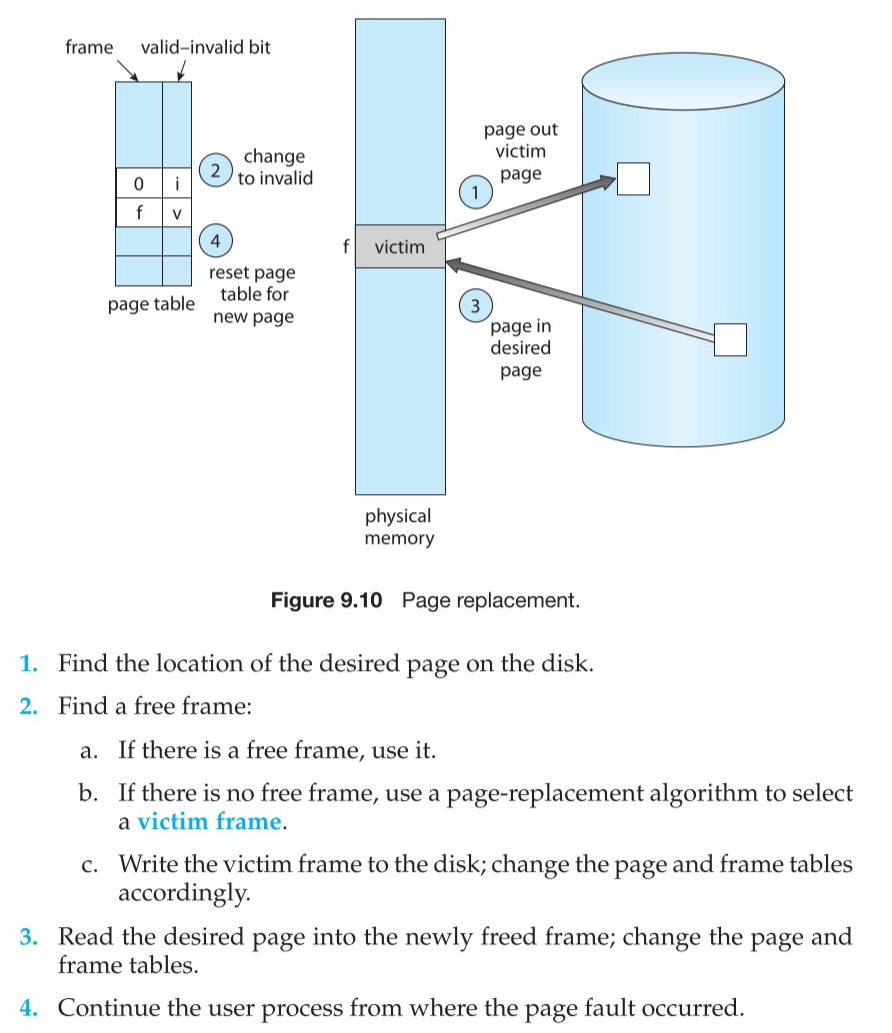
# 6. Page Replacement

Page replacement is important for improving the degree of multiprogramming. Perhaps, we have 10 processes, each with 10 pages. This means that we have allocated 100 pages total. If the system can only hold 50 pages, we must swap out 50 pages. Page replacement algorithms determine how to swap in or out pages for processes.

Due to the principle of locality, we may only be using 5 pages for each process, meaning 50 pages are actively being used. If a process decides to go rogue and access more pages, how do we handle it?

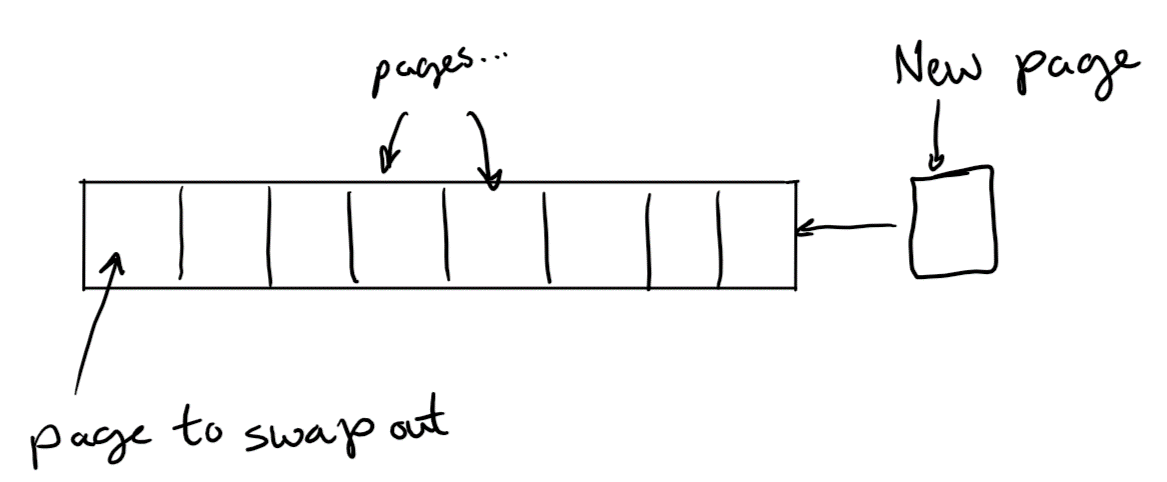
We could terminate the process, but this is terrible, because the user should not be aware that their processes are running on a paged system. We must instead swap pages.

## Basic Page Replacement

* If no frame is free, we find one that isn’t being used, and free it by swapping it
* Swap in the needed frame to continue execution
* 

## Page replacement algorithms

### FIFO (First-in-First-out) Page Replacement:

* Basically a queue data structure
* The first element to be swapped in, is swapped out to make space for the new page
* 
* Very simple to implement

### LRU (Least Recently Used) Page Replacement:

* Pages are kept in a list of a fixed length
* When a page is accessed, it is moved to the front of the list
* When a page is swapped in, the page at the back of the list (the last page) is swapped out. The new page is swapped in and moved to the front of the list
* Requires hardware to keep track of which pages are accessed the most
  + Can be approximated using hardware to mark if a page was used recently, however

### ARC (Adaptive Replacement Cache) Page Replacement:

* An algorithm to balance priority between an LRU cache and an LFU cache
* Scan-resistant (scans would wreck FIFO and LRU caches)
* Extremely complex to implement
* Algorithm:
  + is the cache containing pages
  + The cache contains pages in total
  + is an LRU cache
  + is an LFU cache
  + is a list of pages seen exactly once recently
  + is a list of pages seen at least twice recently
  + is a ghost list of entries recently evicted from the cache; still tracked
  + is a ghost list of entries recently evicted from the cache; still tracked
  + size of cache (in pages)
  + – cache hit
    - Move to the MRU position in
  + – cache miss, but recently evicted from
    - Move to the MRU position in (cache it)
  + – cache miss, but recently evicted from
    - Move to the MRU position in (cache it)
  + – complete cache miss
    - Move to the front of
  + During cases 2 and 3, “invest” in the lists that are performing the best:
    - Case 2: increase target size of
    - Case 3: decrease target size of

## Paged segmentation

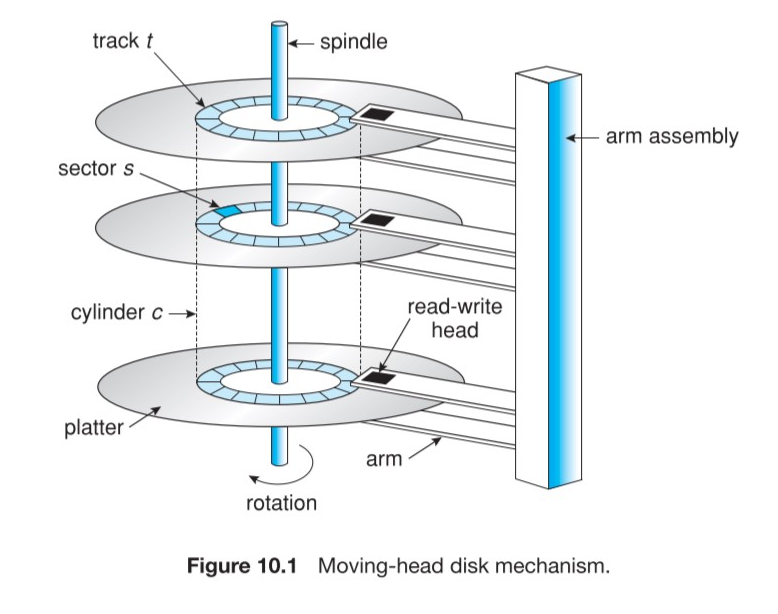
* Paged segmentation allows code to be altered, compiled independently, and linked to allow independent loading
* Segments are logical units that can be shared and protected

# 7. Mass-storage Structures, I/O Systems

## Access times: HDD, RAID, SSD

* **Transfer rate**: bytes/second at which data is transferred to or from the storage
* **Positioning Time**: time required to move the arm to the right spot
* **Random Access Time**:
  + **Seek time**: time required to move the arm to the right position
  + **Rotational Latency**: time required for the disk to rotate to the right angle

### HDDs

* Provide the bulk of secondary storage for many computers
* Multiple spinning platters, each coated with magnetically-reorganizable nanoparticles
* Read-write head that can reorganize the magnetic nanoparticles on the platters, and read their state
* A moving arm, which moves the heads across the disk
* The physical storage space is divided between:
* 

### RAID

* Data striping spreads load across multiple disks, improving performance
* Redundancy improves reliability

### SSD/Flash

* Faster
* Lower capacity
* More expensive
* More reliable
* No moving parts – lower seek times
* Can’t overwrite; must read a block, update it, and rewrite the block
* NVM controller algorithm:
  + Pages are a mix of valid & invalid
  + GC to free invalid page space
    - Blocks are marked as deleted, not overwritten, to reduce strain
  + Wear-leveling spreads the write load across the entire flash

## Disk Scheduling

* FCFS algorithm:
  + Process I/O requests in the order they are received
* SSTF algorithm:
  + Select the request which would require the lowest seek time first (similar to SJF scheduling)
* SCAN algorithm:
  + “Scan” across the disk, processing requests in an order from the lowest disk address to the highest

## Polling

* Repeatedly check on an interval until an I/O device is available
* CPU is required THROUHGOUT the entire transfer

## Interrupt-driven

* A hardware interrupt asynchronously tells the operating system that an I/O event occurred; perhaps:
  + Device is ready for data
  + Device completed a transfer
  + Transfer failed
* CPU is required during each of the interrupts

## DMA

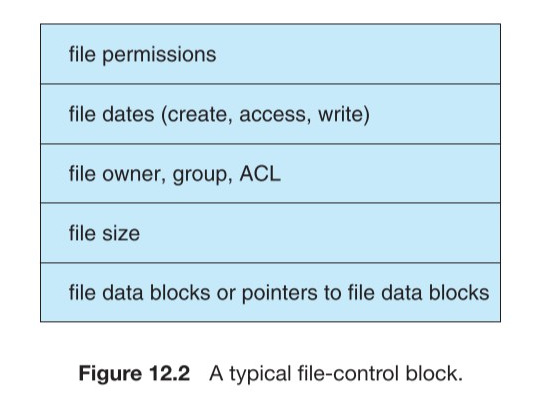
* A FSM in hardware, usually on the CPU, is requested to perform an I/O action
* The FSM is configured via hardware registers
* The FSM requires NO CPU time except during initialization
* The FSM can be set up to throw an interrupt upon completion of its task
* DMA works by a FSM locking the memory bus, fetching memory, unlocking the memory bus, and performing the transfer task

## Polling / Interrupt-driven / DMA

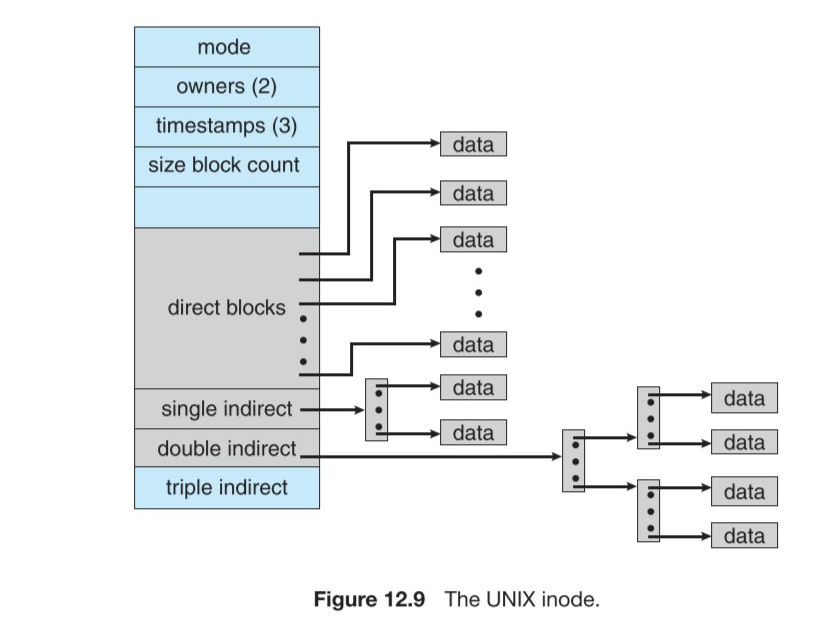
* CPU usage / energy usage / performance:
* DMA is the best option in pretty much every way, but it requires DMA peripherals to exist, which require die space, and are thus, expensive
* Limited DMA channels exist, so if a DMA channel isn’t available at the moment, you may need to fall back to an alternate method of transfer

# 8. File System Interface & Implementation

## FCB (File Control Block)

* Contains information about files and their logical / physical blocks
* 

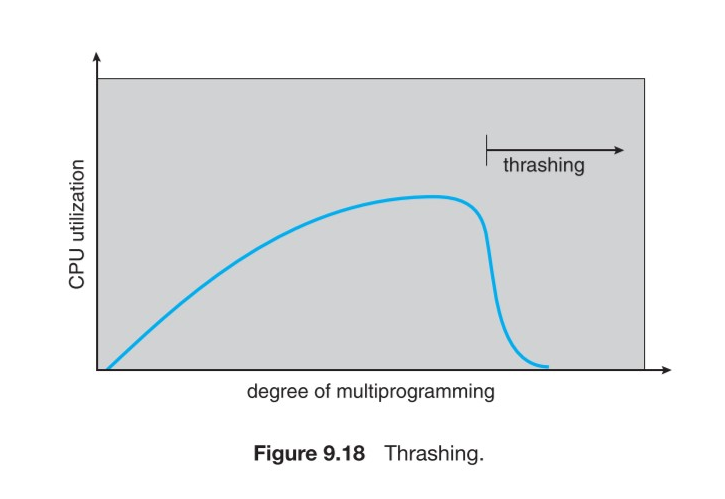
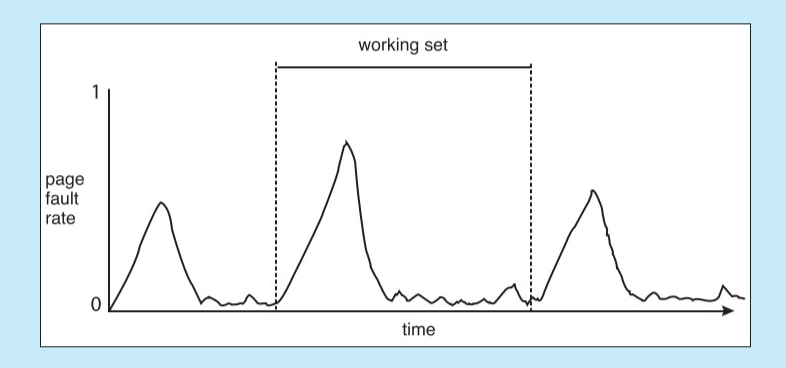
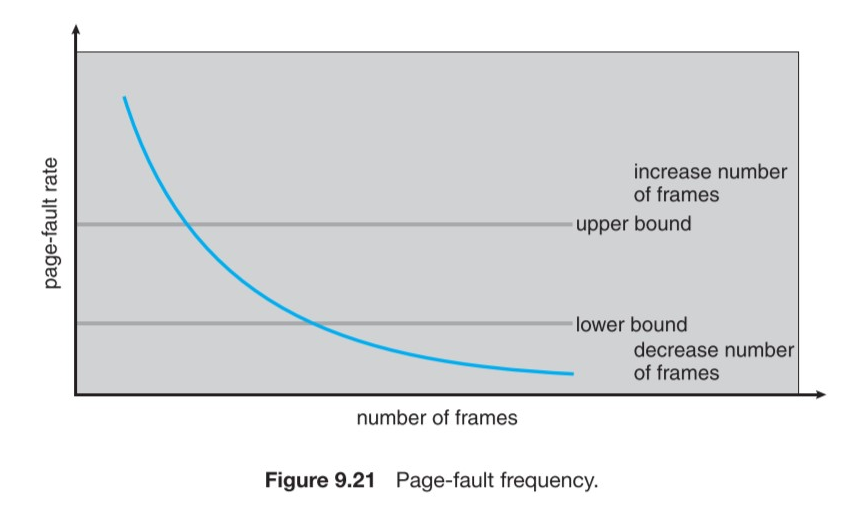
## Allocating logical space on a physical disk

* **Contiguous Allocation**: each file occupies a contiguous block on the disk
  + Defined in terms of the address of the first block / first block number, and the size (usually in allocation units):
  + Access is easy: seek to the correct location, and access as if it were a direct memory address
  + Suffers from external fragmentation
    - As files are removed, they create holes of varying size. If you want to put a large file in the filesystem, you will need to find a hole that can fit the file
    - Can be fixed by memory compaction: defragment the disk by moving files next to each other
* **Linked Allocation**: a file is a linked list of blocks
  + Blocks may be scattered anywhere
  + Random access is slow; you must traverse the linked list to your intended file location
  + Performance can be improved by allocating clusters – adjacent blocks and allowing data to spill linearly between the blocks of the cluster
* **Indexed Allocation**: hybrid between contiguous and linked
  + Each file has an **index block**, which is a list of pointers to blocks scattered throughout the disk
  + Allows efficient direct (random) access, but isn’t as badly affected by fragmentation as contiguous.
  + There are different indexing schemes
    - **Linked**: to allow large files, indexes can be linked one after the other
    - **Multilevel index**:to allow large files, indexes can lead to more index nodes to improve the maximum range
    - **Combined index**:
      * Keep some pointers in the direct block
      * Keep pointers to index blocks to increase the range a bit more
      * The first index pointer contains a single indirect block
      * The second index pointer contains a double indirect block
      * The third index pointer contains a triple indirect block
      * 

## Keeping track of free blocks

* **Bit-vector**:
  + Free-space list is implemented as a bit-vector, where each bit corresponds with a block
  + Simple and efficient
  + Inefficient unless kept in main memory
  + Requires a lot of space, depending on the disk block size
* **Linked List**:
  + Keep a pointer to the first free block
  + Traverse the linked list to find a free block
  + Traversing linked list is slow
* **Grouping**:
  + Stores the addresses of free blocks in the first free block
  + The first of these blocks are actually free
  + The block contains a pointer to another free block with the same layout as above

# 9. Principle of Locality

* Program and data references in a program tend to cluster around each other
* Only a few chunks of memory are actually needed at a time
* It is relatively rare to scan through a large amount of memory
* Working set model:
  + Based on the assumption of locality
  + Uses parameter to define the working set window
  + Idea: examine the most recent page references
    - The set of pages in the most recent page references is the working set
    - The working set is an approximation of the program’s locality
  + If is small, it will not encompass the locality
  + If is large, it may overlap several localities
  + If , the working set is all pages touched in process execution
  + : ☹; 😊; : 😁
  + Remember that we are paying attention to the LAST page accesses to calculate the working set!!!
  + Computing the total working set for each process in the system:
  + If , where is the total number of available frames, then page faults occur!!!
  + Thrashing: when CPU utilization drops because frames cannot efficiently be managed; some processes won’t have enough frames
  + **Page fault rate increases when a process changes locality:** ****
  + Page fault rate goes down if we have more frames available: 

# 10. OMP and Pthreads

## Write a multithreaded program using OMP & Pthreads

Pthread\_t holds a handle to a pthread