Chapter 1: Introduction **OS Definition**: Since the OS looks different there is no universal definition, but we all know what it is meant to provide us: A more common definition, and the one we usually follow, is that the operating system is the one program running at all times on the computer-usually called the kernel. The operating system includes the always-running kernel, middleware frameworks that ease application development and provide features, and system programs that aid in managing the system while its running. **OS Goals:** The fundamental goal of computer systems is to execute programs and to make solving user problems easier. An operating system acts as an intermediary between the user of the computer and the computer hardware. Idea of **Abstraction**: The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner. It is a piece of special 'software' that provides services to support the applications to run, it manages the computer hardware. It is special because it acts as an intermediary translator. The bridge is important because the users can be malicious and dumb. We need to have a mediated control that will provide a 'user friendly' interface to the hardware, because people can damage the hardware. **Resource Allocation** From the computer view, the OS is the resource allocator and acts as the manager of the resources. It will decide within conflicting requests for resources, OS decides how to allocate them to specific programs. Different types of resources included CPU, memory, and I/O devices. **Control Program** As a control program, it manages the execution of the user programs to prevent errors and improper use of the computer. **Computer System Structure**: A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and a user. Where the hardware- consists of the central processing unit, the memory, I/O devices, the application programs- consists of word processors, spreadsheets, compilers, and web browsers, the operating system- controls hardware and coordinates its use among application programs.

A modern computer is comprised of one or more CPUs and a number of device controllers connected through a common bus that provides access between components and shared memory.

Interrupts: This is how a controller informs the device driver that it has finished its operation **Interrupt Vector**: The basic function of an interrupt is that it transfers control to the appropriate interrupt service routine, where it will invoke a generic routine to examine interrupt information and then call the specific handler. But like imagine a gillion interrupts happening very frequently. Using a table of pointers is so much faster, you don't need the 'generic routine'. These locations hold the addresses of the interrupt service routines for the various devices. This array, or interrupt vector, of addresses is then indexed by a unique number, given with the interrupt request, to provide the address of the interrupt service routine for the interrupting device. **Polling Vs. Vectored Interrupt System: Polling:** Reads the status register over and over and over again until the busy bit is clear. This becomes inefficient when it is attempted repeatedly yet rarely finds a device ready for service. "Are you ready? Are you ready? Are you ready?" **Vectored Interrupt System**: Arranges for hardware controller to notify the CPU when the device becomes ready for service. **Multiprocessor Architectures**: (Symmetric vs. Asymmetric) Primary advantage of a multiprocessor architecture is that by increasing the number of processors, more work done so you have increased throughput. But remember that the speed-up for having 7 processors is not 7, it is less than that because of the amount of overhead to switch between processors when doing a task. **Symmetric:** Each CPU processor performs all task, including operating-system functions and user processes. Each CPU has its own set of registers, but a physical memory is shared by a bus. Benefit: Many processes can run at once, 7 processes for 7 CPUs. Disadvantage: The CPU don't share certain data structures, so memory can't be shared dynamically, one CPU can be overloaded while one sits idle. **Asymmetric**: All scheduling decisions, I/O processing and other system activities handled by a single master core. Benefit: One core accesses the system data structures, reducing the need for data sharing. Disadvantage: When master core becomes a bottleneck where overall system performance may be reduced.

Dual-mode: (User Mode vs. Kernel Mode) Dual-mode operation allows OS to protect itself and other system components. A properly designed operating system must ensure that a malicious program can NOT cause other programs or the OS itself to execute incorrectly. A bit, called the mode bit, is added to the hardware of the computer to indicate the current mode: kernel (0) or user (1). With the mode bit, we can distinguish between a task that is executed on behalf of the operating system and one that is executed on behalf of the user. Mode bit provided by hardware. It provides the ability to distinguish when system is running user code or kernel code. Some instructions designated as privileged, only executable in kernel mode. System call changes mode to kernel, return from call resets it to user. Note: At system boot time, the hardware starts in kernel mode. The OS loads and then starts the user applications in user mode. When the OS gains control of the computer, it is in kernel mode.

Chapter 2: OS Structures System Calls: System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf. System calls provide an interface to the service made available by an operating system. Application Programming Interface (API): A simple program can make heavy use of the operating system. Frequently, systems execute thousands of system calls per second. If you look at the example, just copying a source file to a destination file requires a lot.

The API specifies a set of functions that are available to an application programmer, including the parameters that are passed to each function and the return values the programmer can expect. A programmer can access an API via a library of code provided by the operating system. How does an API and system calls interact? Behind the scenes, the functions that make up an API typically invoke the actual system calls on behalf of the application programmer. What are the benefits of an API? 1. Program portability, program can compile on any other system that all supports the same API. 2. System calls are more detailed and difficult to work with than the API available to an application programmer. Run-Time Environment (RTE): The RTE provides a system-call interface that serves as the link to system calls made available by the operating system. The system-call interface intercepts function calls in the API and invokes the necessary system calls within the operating system. System-call interface maintains a table indexed according to these numbers. The system call interface invokes the intended system call in OS kernel and returns status of the system call and any return values. The caller need know nothing about how the system call is implemented.

Just needs to obey API and understand what OS will do as a result call. Most details of OS interface hidden from programmer by API. ABSTRACTION! Parameters: In a system call, sometimes more information is required than simply the identity of the desired system call, like it needs to specify the file or device to use as a source. Three general methods are used to pass parameters: 1. Pass parameters in registers, but what if there are more parameters then registers available? 2. Parameters are stored in a block, or table in memory and the address of this block is passed as a parameter in a register. 3. Parameters can be placed, or pushed, onto a stack by the program and popped off the stack by the operating system. Approaches 2 & 3 are preferred because they do not limit the number or length of paraments being passed. OS Layered Approach (Pros and Cons): There are two approaches to an operating system structure. Tightly Coupled: Changes to one part of the system can have wide-ranging effects on other parts. Loosely Coupled: A system is divided into separate, smaller components that have specific and limited functionality. All these components together comprise the kernel. Changes to one part doesn't affect other components. Layered Approach: In which the operating system is broken into a number of layers or levels. The bottom layer is hardware and the highest layer is the user interface. Advantage: Simplicity of construction and debugging, the layers are selected so that each uses functions and services of only lower-level layers. This simplifies debugging and system verification. You can debug layer by layer, easy to see where the problem is. Each layer also provides abstraction that follows it, you don't need to know how these operations are implemented, just what they do. Disadvantage: How do you appropriately define the functions of each layer? This is hard to do. Performance of system sinks, because the user program has to go through multiple layers to obtain some service. More layer is equal to more traverse which takes longer. Microkernels: This method structures the operating system by removing all nonessential components from the kernel and implementing them as user-level programs that reside in separate address spaces. So, a smaller kernel, that provides minimal process and memory management and communication facility. Microkernel's Communication:

Provides communication between the client program and the various services that are also running in the user space. Communication is through message passing, see diagram below. Advantages: Makes extending the OS easier, all new services are added to a user space and changes to kernel are small, because well the kernel is small. Smaller kernel a has higher portability to work another hardware design. More secure too because most services are running as user processes, not kernel, so if one thing fails the remainder is untouched. Disadvantages: Overhead is garbage, so performance suffers. When two user-level services communicate, messages must be copied between services, so separate address spaces. And the OS has to switch from one process to the next to exchange messages. Switching costs time. Modules: Current and best approach for structure is Loadable Kernel Modules (LKM). A kernel has a set of core components and can link in additional services via modules, either at boot time or run time. In plain English, this means… Major idea of this design is for the kernel to provide core services, while other services are implemented dynamically as the kernel is running. Linking 'dynamically' is better than just adding/changing directly the kernel because you don't have to recompile the kernel every time a change is made. Why Module is better than a Microkernel or the Layered Approach? This much better than a layered system because it is like a layered system but more flexible, any module can call any other module, not just the one 'beneath' it. Like a microkernel because there is a primary module that has only core functions but better because it does not need to invoke message passing to communicate.

Chapter 3: Processes Process Definition: Process: A process is a program in execution. A process will need certain resources – such as CPU time, memory, files and I/O devices to accomplish its' tasks. These resources are typically allocated to the process while it is executing. System: A system is a collection of processes; system code and user code being executed. These processes may execute concurrently. A system therefore consists of a collection of processes, some executing user code, others executing operating system code. Thread: Multiple processes are called threads. What is the difference between a process and a program? A process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. A program is static. Process Layout in Memory: Process is a program in execution. The status of the current activity of a process is represented by the value of the program counter and the contents of the processor's registers. The memory layout of a process is typically divided into multiple sections. 1. Text Section: The executable code. Fixed. 2. Data Section: Global variables. Fixed. 3. Heap Section: Memory that is dynamically allocated during program run time. 4. Stack Section: Temporary data storage when invoking functions (such as function parameters, return addresses, and local variables).

OS is responsible for making sure data and heap do not overlap as they dynamically change. States & State Transitions: As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

1. New: The process is being created. 2. Running: Instructions are being executed. 3. Waiting: The process is waiting for some event to occur (such as an I/O completion or reception of a signal). 4. Ready: The process is waiting to be assigned to a processor. 5. Terminated: The process has finished execution.

Process Control Block: Each process is represented in the operating system by a process control block (PCB) – also called a task control block. Process State: The state may be new, ready, running, waiting, halted, and so on. Program Counter: The counter indicates the address of the next instruction to be executed for these processes. CPU Registers: The registers vary in number and type, depending on the computer architecture. This state information must be saved when an interrupt occurs. CPU-Scheduling Information: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters. Memory-management Information: May include value of base and limit registers, page tables, or segment tables, depend on memory system used by the OS. Accounting Information: Amount of CPU and real time used, time limits, process numbers, … I/O Status Information: List of I/O devices allocated to the process. PCB serves as the repository for the data needed to start, or restarts, a process. Types of Process: Balancing the objectives of multiprogramming and time sharing also requires taking the general behavior of a process into account. In general, most processes can be described as either I/O bound, or CPU bound. CPU Bound: A process that spends more of its time doing computations. I/O Bound: A process that spends more of its time doing I/O than it spends doing computations. Context Switching: Interrupts or System Call cause the operating system to change a CPU core from its current task and to run a kernel routine.

When an interrupt occurs, the system needs to save the current context of the process running on the CPU core so that it can restore that context when its processing is done. The context is presented in the PCB of the process. We perform a state save of the current state of the CPU core, either kernel or user mode. And then state restore to resume operations. Switching the CPU core performs a state save of the current process and state restore of a different process. This task is known as a context switch, it is pure overhead because the system does not do any useful work while switching. Context Switching Speeds: Switching speed depends on the complexity of the OS and the PCB, the more complex the longer it takes to switch. Switching speed depends on the number of registers available in the hardware. Of course, if there are more active processes than there are register sets, the system resorts to copying register data to and from memory. Also, the more complex the operating system, the greater the amount of work that must be done during a context switch. Long-term and Short-Term Scheduling: The operating system is responsible for managing the scheduling activities. The main memory cannot always accommodate all processes at runtime for multi-programmed OS. The operating system will need to decide on which process to execute next (short-term) and which processes will be brought to the main memory (long-term). Schedulers: Short-term Scheduler (CPU Scheduler) selects which process should be executed next and allocates CPU. This must be fast because it is invoked frequently. Long-term Scheduler (Job Scheduler) selects which processes should be brought into the ready queue. It might be slow because it is not invoked frequently. This LTS controls the degree of multiprogramming, which is basically when there are more processes than cores, and excess processes will have to wait until a core is free and can be rescheduled where the number of processes in memory is the degree of multiprogramming. Process Creation & Termination: fork (): System call that creates a new process. exec (): System call used after fork () to replace the process' memory space with a new program. wait (): Returns status data from child to parent using wait (). The parent process may wait for termination of a child process by using the wait () system call. The call returns status information and the pid of the terminated process.

Chapter 4: Threads Thread Definition: Example: Take a thread word-processor program. A single-thread word-processor program allows the process to perform only one task at a time. A user cannot type characters and run a spell checker at the same time. But modern operations systems have 'stepped-they're-game-up' and allow a process to perform multiple tasks at once so multiple threads in parallel. A multi-threaded word-processor can assign one thread to manage user input while another thread runs spell checker. Thread is a set of instructions or a task that is excited by a process. The more threads, the more tasks a process can accomplish. Multiple threads within a process will share the address space, open files, and other resources. They are: 1. Responsiveness: Multi-threading an application that is "interactive" allows a program to continue to run even if part of it is blocked or performing a lengthy operation, the user feels an increased responsiveness. 2. Resource-sharing: Processes share their resources through things that have to be arranged by the programmer, threads share memory and resources of a process by default. 3. Economy: Instead of creating new processes, which is costly for memory and resource, it saves memory and resource to just create a new thread, the overhead might be garbage with switching between threads, but it has faster context switching then with processes. 4. Scalability: Threads can run in parallel on different processing cores. A single-threaded process can run on only one processor despite how many are available. Concurrency and Parallelism: A concurrent system supports more than one task by allowing all the tasks to make progress. In contrast, a parallel system can perform more than one task simultaneously. Thus, it is possible to have concurrency without parallelism. Multi-threading Models: A relationship between user-level threads and kernel-level threads must exist. Many-to-one: Maps many user-level threads to one kernel thread. Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems. One-to-one: Maps each user thread to a kernel thread. This allows multiple threads to run in parallel on multi-processors. And there is also concurrency because it allows another thread to run when one thread makes a blocking system call. Only drawback, there are a large number of kernel threads that can burden a system. Many-to-many: Multiplexes many user-level threads to smaller of equal number of kernel threads. The many-to-many model suffers from neither of these shortcomings: developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. The number of kernel threads does not have to equal the number of user threads like the one-to-one model. Instead, the number can be equal or less.

User-level & Kernel-level Threads: User-level Thread: User threads are supported above the kernel and are managed without kernel support. This is a thread that the operating system does not know about. The OS only knows about the process, so it can schedule the process but not the threads within the process. There is no context switch, this is faster to switch then a kernel-level thread! Kernel-level Threads: Kernel threads are supported and managed directly by the operating system. This is a thread that the operating system knows about. It is a lightweight process because the memory management information doesn't need to change, this is a fast context switch, but still a context switch. Well, what does support mean? This means the management and shedding of the threads. Thread Library: Thread management is done by the thread library in user space. It provides a programmer with an API for creating and managing threads. There are two ways to implement a thread library: 1. Provide a library entirely in user space with no kernel support. The code and data structures exist in user space and invokes local function calls to access the API. 2. Implement a kernel-level library supported directly by the OS. The code and data structures exist in the kernel space and that means invoking a function in the API requires a system call to the kernel. Implicit Threading: With the growth of multi-core processing, the growth of threads is increasing and so is the impact of their difficulties and challenges. One way to address these difficulties and better support the design of concurrent and parallel applications is to transfer the creation and management of threading from application developers to compilers and run-time libraries. This strategy, termed implicit threading, is an increasingly popular trend. The solution with dealing with all these problems. Let the compilers and run-time libraries deal with thread creation. The advantage of this approach is that developers only need to identify parallel tasks, and the libraries determine the specific details of thread creation and management. Thread Pools: The general idea behind a thread pool is to create a number of threads at start-up and place them into a pool, where they sit and wait for work. When a server receives a request, rather than creating a thread, it instead submits the request to the thread pool and resumes waiting for additional requests. If there is an available thread in the pool, it is awakened, and the request is serviced immediately. If the pool contains no available thread, the task is queued until one becomes free. Once a thread completes its service, it returns to the pool and awaits more work. OpenMP: OpenMP is a set of compiler directives as well as an API for programs written in C, C++, or FORTRAN that provides support for parallel programming in shared memory environments. OpenMP identifies parallel regions as blocks of code that may run in parallel. Application developers insert compiler directives into their code at parallel regions, and these directives instruct the OpenMP run- time library to execute the region in parallel. This creates as many threads as there are processing cores in the system. Grand Central Dispatch: Similar to OpenMP: It is a combination of a run-time library, an API, and language extensions that allow developers to identify sections of code (tasks) to run in parallel. Like OpenMP, GCD manages most of the details of threading. Similar to Thread Pool: GCD schedules tasks for run-time execution by placing them on a dispatch queue. When it removes a task from a queue, it assigns the task to an available thread from a pool of threads that it manages. But different in that it has two types of dispatch queues: serial and concurrent. Serial: Tasks that are placed in the serial queue are removed in FIFO order. Once a task has be removed from the queue, it must complete execution before another task is removed. Good in ensuring the sequential execution of tasks! Concurrent: Several tasks may be removed from the queue at a time, allowing multiple tasks to execute in parallel. Threading Issues: The semantics of the fork () and exec () system calls change in a mulita-threaded program. Fork (): If one thread in a program calls for (), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have two versions of fork (), one that duplicates all heads and another that duplicates only the thread that invokes the fork () system call. Exec (): If a thread invokes the exec () system call, the program specified in the parameter to exec () will replace the entire process- including all threads.

Chapter 5: CPU Scheduling CPU Scheduling Criteria: CPU-Scheduling Decisions 1. When a process switches from a running state to the waiting state. [Invoking a wait ().] 2. When a process switches from the running state to the ready state. [When an interrupt occurs.] 3. When a process switches from the waiting state to the ready state. [Completion of I/O.] 4. When a process terminates. Non-preemptive vs. Preemptive Scheduling: Non-preemptive: When scheduling takes place under circumstances of 1 & 4, once a CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state. Preemptive: A running process may be forced to release the CPU even though it is neither completed nor blocked. Take the following case study for example… Two processes that share data. While one process is updating the data, it is preempted so that the second process can run. The second process then tries to read the data which are in an inconsistent state. CPU Scheduler and Dispatcher: Dispatcher: Module that gives control of the CPU's core to the process selected by the CPU scheduler. How is this done? It must switch context from one process to another, then switch to user mode, and then it must jump to the proper location in the user program to resume that program. The time it takes for the dispatcher to stop one process and start another running is known as dispatch latency. Scheduler: When the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The selection process is carried out by the CPU scheduler which selects a process from the processes in memory that are ready to execute and allocated the CPU to that process. CPU Scheduling Criteria: How a CPU-scheduler chooses a process is due to its algorithm, and different algorithms have different properties. CPU Utilization: We want to keep the CPU as busy as possible. Throughput: If the CPU is busy executing processes, then work is being done. Turnaround Time: How long it takes to execute that process, the interval of time from submission to execution is turnaround time. Waiting Time: It affects only the amount of time that a process spends waiting in the ready queue. Response Time: Time interval from submission until the first response is produced, it is the time it takes to start responding, not the time it takes to output the response.

Meeting-the-deadline: Real-time systems. Scheduling Algorithms: FCFS: First Come, First-Served Scheduling: With this scheme, the process that requests the CPU first is allocated the CPU first. This is implemented using a FIFO queue, but the downside is that the average waiting time is often long. But what if a short process is behind a long process? This may lead to poor overlap of I/O and CPU since CPU-bound processes will force I/O bound processes to wait for the CPU, leaving the I/O devices idle, hence the convey effect. SJFS: Shortest-Job, First Scheduling: This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available it is assigned to the process that has the smallest next CPU burst. If there is a tie, FCFS is used to break the tie. But how do we know what the length of the next CPU request is? This algorithm can either be preemptive or nonprimitive: Preemptive: Take the idea that a new process process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process, so the SRTF Shortest-remaining-time-first, will preempt the current executing process.

Nonpremptive: Where the algorithm will allow the currently running process to finish its CPU burst. Round-Robin Scheduling: Similar to the FCFS scheduling, but preemption is added to enable the system to switch between processes. You might still be lost and asking yourself how does it work exactly? A time quantum is defined, which is just a unit of time. Imagine the ready queue as a circular queue, where the CPU scheduler goes around the ready queue allocating the CPU to each process for a time interval of up to 1-time quantum. The CPU scheduler picks the first process from the ready queue sets a time interrupt after 1-time quantum and dispatches the process.

Priority Scheduling: A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal priority processes are scheduled in FCFS order. Starvation: Low priority processes may never execute. Aging: As time progresses increase the priority of the process. Multi-level Queue Scheduling: The scheduler then selects the process with the highest priority to run and depending on how the queues are managed with an O(n) search may be necessary to determine the highest-priority process. It is easier to have different queues with different priorities, like one for higher-priority and one for lower-priority. A multi-level queue algorithm can be used to partition processes into several queues based on the process types. Foreground: Interactive processes which have higher priority than background processes. Background: Batch processes. Well what the heck is Multi-level Feedback Queue Scheduling? Its kind of sounds the same. When a process enters either a foreground or background queue, it is inflexible, it can't move between queues. The feedback version of this type of queueing allows a process to move between queues. But if the nature of a process doesn't change then what is the point? The idea is to separate processes according to the characteristics of their CPU bursts. If one process is taking too much CPU time, it gets moved to the lower-priority queue. This is a form of aging that prevents starvation!

What is the purpose of a process control block (PCB)? When is it updated and when is it read by the operating system?
Each process is represented in the operating system by a process control block. It serves as a repository for the data needed to start or restart a process. It is read by the operating system during a context switch where the system needs to save the current context of the process running on a CPU which is represented in the PCB of the process. When the process made transitions from one state to another, the operating system must update the process state.
Given two processes in the READY state, one that is CPU-bound and one that is I/O bound, which process should be given a higher priority for running next (all other things being equal)? Justify your answer.
The I/O bound process should be given a higher priority. Since it will run for a short time then issue an I/O request, the CPU-bound process will be able to run at the same time that the I/O device is performing the I/O request – maximizing system resources. There us a convoy effect as all the short process wait for the big process to get off the CPU, this results in lower CPU and device utilization.
What effect does the size of the time quantum have on the performance of a round robin algorithm?
You don't want the time quantum to be too big where it will just mimic a First-Come First-Served algorithm. But you don't want it too small where many context switches occur slowing the execution og the process.
When a process executes a fork() system call, a duplicate process (i.e the child process) is created. How does the code in the processes (since it is identical in both the parent and child processes) known which process is the parent and which is the child?
It knows depending on the return value of the fork(), where the child returns 0 and the parent returns the process ID.
When multiple processes need to cooperate, there is a choice between shared memory and message passing. Compare and contrast these two techniques. Make sure to clarify the role of the operating system in each.
Shared memory: OS allocates a region of memory that is shared by more than one process (must be on the same machine to do this!). Usually done with page tables; processes can then read/write the shared locations at memory speeds without OS intervention.
Message passing: processes communicate using send and receive these are system calls that invoke OS services; the OS is involved in every interaction to copy messages to/from address spaces.
Describe the difference between "CPU-bound" processes and "I/O-bound" processes.
I/O-bound will run for a shorter time then the CPU-bound.
What needs to be save and restored on a context switch between two threads in the same process? What if two are in different processes? Be brief and explicit.
The process context switch needs to save and restore all of the process state, including program counter, registers, memory mapping, accounting, and other resource information.
The thread context switch needs to save the program counter and registers. The memory mapping, accounting, and other resources information stays the same.
Describe the difference between fork() and exec():
Exec() system call follows the fork(), exec() overwrites a process and fork() creates a new process.
Why would two processes want to use shared memory for communication instead of using message passing?
Shared memory is faster communication than message passing.
Describe the difference between "preemptive" scheduling and "non-preemptive" scheduling.
Non-preemptive: Once a CPU has been allocated to a process, the process keeps the CPU until it releases it either by terminating or by switching to the waiting state.
Preemptive: A running process may be forced to release the CPU even though it is neither completed nor blocked.

1. **A message passing model is:** Easier to implement than a shared memory model for inter-computer communication
2. **A race condition is when:** The correctness of the code depends upon the timing of the execution
3. **A table points to routines that handle interrupts is called _____.** Interrupt vector
4. **A thread control block _____:** Does not include information about the parent process resource allocation
5. **An interrupt is**: A signal that causes the control unit to branch to a specific location
6. **Embedded computers typically run on a _____ operating system.** Real-time
7. **For a single-processor system:** There will never be more than one running process
8. **In a system where round robin is used for CPU scheduling, which of the following is true when a process cannot finish its computation during its current time quantum?** The process's state will be changed from running to ready
9. **In scheduling, the term *aging* involves:** gradually increasing the priority of a process so that a process will eventually execute
10. **Long term scheduling is performed_____:** Typically on submitted jobs
11. **Most often, application programs access system resources using_____:** Application Program Interfaces (APIs)
12. **The major difficulty in designing a layers operating system approach is:** appropriately defining the various layers
13. **The multithreading model is supported by the Linux operating system is___**: One-to-One
14. **The Producer-Consumer problem is related to_____:** The allocation of resources to process states
15. **The text segment of a process address space contains:** The executable code associated with process
16. **What is a trap/exception?**: software generated interrupt caused by an error
17. **What is the main difference between traps and interrupts?** How they are initiated
18. **What is the READY state of a process?** When process is scheduled to run after some execution
19. **When a process is accessing its heap space, it exists in the_____:** Running state
20. **When a process is created using the classical fork() system call, which of the following is NOT inherited by the child process?** Process ID
21. **Which is true about process and threads?** Threads in a process share the same file descriptors
22. **Which of the following is *not* true about message passing?** In direct communication, multiple links may exist between a pair of processes
23. **Which of the following is true about multilevel queue scheduling?** Each queue has its own scheduling algorithm
24. **Which of the following scheduling algorithms will have the longest average response time after many jobs are queued and ran to completion?** FCFS
25. **Which of the following statements is *false* with regards to the Linux CFS scheduler?** There is a singly, system-wide value or **vruntime**
26. **Which of the following would lead you to believe that a given system is an SMP-type system?** Each processor performs all tasks within the operating system

TRUE/ FALSE
A context switch from one process to another can be accomplished without execution OS code in kernel mode **False**
A deadlock-free solution eliminates the possibility of starvation. **False**
A thread can be blocked on multiple condition variables simultaneously **False**
Aging can alleviate the starvation problem of a low priority job. **True**
All process in UNIX first translate to a zombie process upon termination **True**
An advantage of implementing threads in user space is that they don't incur the overhead of having the OS schedule their execution. **True**
An interrupt vector contains the addresses of the handlers for the various interrupts. **True**
Each thread of a process has its own virtual address space **False**
In a monolithic kernel. Most operating system components, e.g., memory management, inter-process communication, and basic synchronization modules, execute outside the kernel. **False**
In round robin scheduling, it is advantages to give each I/O bound process a longer quantum than each CPU-bound process (since this has the effect of fiving the I/O bound process a higher priority) **False**
It is possible to have concurrency without parallelism. **True**
Non-preemptive scheduling algorithms are better for interactive jobs since they tend to favor jobs that require quick responses. **False**
Processes in a microkernel architecture operating system usually communicate using shared memory. **False**
Shortest Remaining Time First is the best preemptive scheduling algorithm that can be implements in an operating system **False**
System calls can be run in either user mode or kernel mode. **False**
The code that changes the system clock runs in user mode. **False**
The main difference between the use of test-and-set and the use of semaphores is that semaphores require the OS to do the busy waiting rather than the user program. **False**
The two primary purposes of an operating system are to manage the resources of the computer and to provide a convenient interface to the hardware for programmers **True**