Chapter 3

- A ***process*** can be thought of as a program in execution.
- A ***process*** is a program in execution.
- A ***process*** is the unit of work in a modern time-sharing system.
- A process will need certain resources-such as ***CPU time, memory, files,*** and ***I/O devices*** -to accomplish its task. These resources are allocated to the process either when it is ***created*** or while it is ***executing***.
- A program becomes a ***process*** when an executable file is loaded into ***memory***.
- Each process may be in one of the following states: new, ***running, waiting, ready***, and terminated.
- Only one process can be ***running*** on any processor at any instant. Many processes may be ***ready*** and ***waiting***.
- Each process is represented in the operating system by a ***process control block (PCB).***
- The objective of multiprogramming is to have some process ***running*** at all times, to ***maximize*** CPU utilization.
- The objective of ***time sharing*** is to switch the ***CPU*** among processes so frequently that users can interact with each program while it is ***running***.
- The processes that are residing in main memory and are ready and waiting to ***execute*** are kept on a list called the ***ready*** queue.
- ***CPU scheduler*** selects from among the processes that are ***ready*** to execute and allocates the CPU to one of them.
- The ***long-term*** scheduler controls the degree of multiprogramming (the number of processes in memory).
- The primary distinction between the job scheduler and CPU scheduler lies in ***frequency of execution.***
- The process is swapped out, and is later swapped in, by the ***medium-term*** scheduler.
- The context is represented in the ***PCB*** of the process**.**
- When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process. This task is known as a ***context switch*****.**
- The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ***ready*** queue.
- The list of processes waiting for a particular I/O device is called a ***device*** queue.
- Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique ***process identifier*** (or ***pid***), which is typically an ***integer*** number.
- Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information. There are two fundamental models of interprocess communication: (1) ***shared memory*** and (2) ***message passing***.
- Shared memory is ***faster*** than message passing.
  Because message-passing systems are typically implemented using ***system calls*** and

thus require the more timeconsuming task of kernel intervention.
In shared-memory systems, system calls are required only to establish ***shared-memory regions.*** Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

- Under ***direct communication***, each process that wants to communicate must explicitly name the recipient or sender of the communication.
- With ***indirect*** communication, the messages are sent to and received from ***mailboxes***, or ports.
- Communication in client-server systems may use (1) ***sockets***, (2) remote procedure calls (***RPCs***), or (3) Java's remote method invocation (***RMI***).

Chapter 4

- A ***thread*** is a basic unit of CPU utilization; it comprises a thread ID, a ***program counter,*** a ***register set,*** and a stack. It shares with other threads ***belonging to*** the same process its ***code section***, ***data section,*** and other operating-system ***resources***, such as ***open files*** and signals.
- Threads share the ***memory*** and the ***resources*** of the process to which they belong.
- it is much more ***time consuming*** to create and manage processes than ***threads***.
- Support for threads may be provided either at the ***user*** level, for user threads, or by the ***kernel***, for kernel threads.
- ***User*** threads are supported ***above*** the kernel and are managed ***without*** kernel support, whereas ***kernel*** threads are supported and managed directly by the ***operating system***.
- The many-to-one model maps many user-level threads to one ***kernel*** thread.
- The ***one-to-one*** model maps each user thread to a kernel thread.
- The ***one-to-one*** model provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.
- The ***one-to-one*** model allows multiple threads to run in parallel on multiprocessors.
- The ***one-to-one*** model, creating a user thread requires creating the corresponding kernel thread.
- The ***many-to-many*** model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The ***many-to-one*** model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.
- A ***thread*** is a flow of control within a process.
- A multithreaded process contains several different flows of ***control*** within the same address space.
- The benefits of multithreading include increased responsiveness to the user, resource sharing within the process, economy, and the ability to take advantage of ***multiprocessor*** architectures.
- ***User-level*** threads are threads that are visible to the programmer and are unknown to

the kernel.

- The operating-system kernel supports and manages ***kernel-level*** threads.
- Three different types of models relate user and kernel threads: The ***many-to-one*** model maps many user threads to a single kernel thread. The ***one-to-one*** model maps each user thread to a corresponding kernel thread. The ***many-to-many*** model multiplexes many user threads to a smaller or equal number of kernel threads.
- ***Thread libraries*** provide the application programmer with an API for creating and managing threads. Three primary thread libraries are in common use: ***POSIX Pthreads***, ***Win32*** threads for Windows systems, and ***Java*** threads.
- A ***signal*** is used in UNIX systems to notify a process that a particular event has occurred.
- A signal may be received either synchronously or asynchronously, depending on the source of and the ***reason*** for the event being signaled.
- ***Synchronous*** signals are delivered to the same process that performed the operation that ***caused*** the signal.
- When a signal is generated by an event ***external*** to a ***running*** process. That process receives the signal asynchronously.
- ***Synchronous*** signals need to be delivered to the thread ***causing*** the signal and not to other threads in the process.
- To the user-thread library, the ***LWP*** appears to be a *virtual processor* on which the application can schedule a user thread to run. Each LWP is attached to a kernel thread, and it is ***kernel threads*** that the operating system schedules to run on physical processors.
- In general, user-level threads are ***faster*** to create and manage than are kernel threads, as no intervention from the kernel is required.

Chapter 5

- In a single-processor system, only ***one*** process can run at a time; any others must wait until the CPU is free and can be rescheduled.
- The objective of multiprogramming is to have some process ***running*** at all times, to maximize ***CPU*** utilization.
- Process execution consists of a cycle of ***CPU*** execution and ***I/O*** wait. Processes alternate between these two states. Process execution ***begins*** with a ***CPU burst.*** That is ***followed*** by an ***I/O burst***, which is followed by another ***CPU burst,*** then another ***I/O burst,*** and so on. Eventually, the ***final CPU burst*** ends with a system request to terminate execution.
- An ***I/O-bound*** program typically has many ***short*** CPU bursts. A ***CPU-bound*** program might have a few ***long*** CPU bursts.
- Whenever 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 ***short-term*** scheduler (or ***CPU*** scheduler).
- The ***scheduler*** selects a process from the processes in memory that are ***ready*** to execute and allocates the CPU to that process.

- All the processes in the ready queue are lined up waiting for a chance to ***run*** on the CPU. The records in the queues are generally ***process control blocks (PCBs)*** of the processes.
- Under ***nonpreemptive*** scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to the ***waiting*** state.
- The ***dispatcher*** is the module that gives control of the ***CPU*** to the process selected by the short-term scheduler.
- The time it takes for the ***dispatcher*** to stop one process and start another running is known as the dispatch ***latency***.
- The interval from the time of submission of a process to the time of completion is the ***turnaround time***.
- ***Waiting time*** is the sum of the periods spent waiting in the ready queue.
- The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends ***waiting*** in the ***ready*** queue.
- ***Response time*** is the time from the submission of a request until the first response is produced.
- ***Response time*** is the time it takes to start responding, not the time it takes to output the response.
- The PCPS scheduling algorithm is ***nonpreemptive***.
- The SJF algorithm can be either ***preemptive*** or ***nonpreemptive***.
- Priority scheduling can be either ***preemptive*** or ***nonpreemptive***.
- The RR scheduling algorithm is thus ***preemptive***.
- Only ***nonpreemptive*** algorithms can be used for job scheduling.
- ***Preemptive*** and ***nonpreemptive*** algorithms can be used for CPU scheduling.

Chapter 6

- A situation like this, where 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, is called a ***race condition.***
- A ***critical section*** is a segment of ***code***, in which the process may be changing common variables, updating a table, writing a file, and so on.
- The ***critical-section problem*** is to design a ***protocol*** that the processes can use to cooperate.
- when one process is executing in its critical section, no other process is to be ***allowed*** to execute in its critical section.
- A solution to the critical-section problem must satisfy the following three requirements: ***mutual exclusion, progress,*** and ***bounded waiting***.
- A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: ***wait( )*** and ***signal( ).***
- All the modifications to the integer value of the semaphore in the wait() and signal() operations must be executed ***indivisibly***.

- Operating systems often distinguish between counting and binary semaphores. The value of a ***counting*** semaphore can range over an unrestricted domain. The value of a ***binary*** semaphore can range only between 0 and 1.
- Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the ***number*** of resources available.
- Each process that wishes to use a resource performs a ***wait()*** operation on the semaphore. When a process releases a resource, it performs a ***signal()*** operation.
- When the count for the semaphore goes to ***0***, all resources are being used.
- A process that is blocked, waiting on a semaphore S, should be restarted when some ***other*** process executes a ***signal()*** operation.
- The ***block()*** operation suspends the process that invokes it. The ***wakeup(P)*** operation resumes the execution of a blocked process P.
- If the semaphore value is negative, its magnitude is the number of processes ***waiting on*** that semaphore.
- We must guarantee that no two processes can execute wait() and signal() operations on the ***same*** semaphore at the same time.
- A set of processes is in a ***deadlock*** state when every process in the set is waiting for an event that can be caused only by another process in the set.
- ***Starvation*** or indefinite ***blocking*** is a situation in which processes wait indefinitely within the semaphore.
- The ***monitor*** type is one fundamental high-level synchronization construct.
- The ***monitor*** type contains the declaration of variables whose values define the state of an instance of that type, along with the bodies of procedures or functions that operate on those variables.
- A procedure defined within a monitor can ***access only*** those variables declared locally within the monitor and its formal parameters. Similarly, the local variables of a monitor can ***be accessed by only*** the local procedures.

Chapter 7

- A set of processes is in a ***deadlock*** state when every process in the set is waiting for an event that can be caused only by another process in the set.
- A ***deadlock*** situation can arise if the following four conditions hold simultaneously in a system: ***mutual exclusion, hold and wait, no preemption,*** and ***circular wait.***
- If the resource-allocation graph contains no cycles, then no process in the system is ***deadlocked***.
- In a given resource-allocation graph,
    1. if each resource type has ***exactly one*** instance, then a ***cycle*** implies that a ***deadlock*** has occurred.
    2. if the cycle involves only a set of resource types, each of which has only a ***single*** instance, then a deadlock has occurred.
    3. Each process involved in the cycle is ***deadlocked***.
    4. In this case, a cycle in the graph is both a ***necessary*** and a ***sufficient***

condition for the existence of deadlock.

- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred. In this case, a cycle in the graph is a ***necessary*** but not a ***sufficient*** condition for the existence of deadlock.
- We can use a protocol to ***prevent*** or ***avoid*** deadlocks. ensuring that the system will ***never enter*** a deadlock state.
- To ensure that deadlocks never occur, the system can use either a ***deadlock-prevention*** or a ***deadlock-avoidance*** scheme.
- ***Deadlock prevention*** provides a set of methods for ensuring that at least one of the ***necessary*** conditions cannot hold.
- For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can ***prevent*** the occurrence of a deadlock.
- The mutual-exclusion condition must hold for ***nonsharable*** resources.
- ***Sharable*** resources (such as read-only files) do not require mutually exclusive access and thus cannot be involved in a deadlock.
- To ensure that the ***hold-and-wait*** condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- The ***deadlock-avoidance*** algorithm dynamically examines the resource-allocation state to ensure that a ***circular-wait*** condition can never exist.
- A state is ***safe*** if the system can allocate resources to each process (up to its maximum) in some order and still avoid a deadlock.
- A system is in a ***safe*** state only if there exists a ***safe sequence.***
- If no such sequence exists, then the system state is said to be ***unsafc***.
- A ***deadlock*** state occurs when two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes.
- A deadlock can occur only if four necessary conditions hold simultaneously in the system: ***mutual exclusion, hold and wait, no preemption***, and ***circular wait.***
- To ***prevent*** deadlocks, we can ensure that at least one of the necessary conditions never holds.