Operating System

UNIT 1

Introduction

- An *operating system* acts as an intermediary between the user of a computer and the computer hardware.
- The purpose of an operating system is to provide an environment in which a user can execute programs in a *convenient* and *efficient* manner.
- A computer system can be divided roughly into four components: the hardware, the operating system, the application programs, and the users.
- The hardware—the central processing unit (CPU), the memory, and the input/output (I/O) devices—provides the basic computing resources for the system.
- The application programs—such as word processors, spreadsheets, compilers, and Web browsers—define the ways in which these resources are used to solve users' computing problems.
- OS is a resource allocator i.e. it Manages all resources and Decides between conflicting requests for efficient and fair resource use
- OS is a control program i.e. it Controls execution of programs to prevent errors and improper use of the computer
- "The one program running at all times on the computer" is the **kernel**. Everything else is either a system program, or an application program.
- The occurrence of an event is usually signaled by an **interrupt** from either the hardware or the software.
- Hardware may trigger an interrupt at any time by sending a signal to the CPU, usually by way of the system bus.
- Software may trigger an interrupt by executing a special operation called a **system call.**

PROCESS MANAGEMENT

- A process is a program in execution. It is a unit of work within the system.
- Program is a passive entity, process is an active entity. Process needs resources to accomplish its task CPU, memory, I/O, files Initialization data .
- Process termination requires reclaim of any reusable resources
- Single-threaded process has one **program counter** specifying location of next instruction to execute
- Process executes instructions sequentially, one at a time, until completion .
- Multi-threaded process has one program counter per thread.
- Typically system has many processes, some user, some operating system running concurrently on one or more CPUs

- The operating system is responsible for the following activities in connection with process management:
    1. Scheduling processes and threads on the CPUs
    2. Creating and deleting both user and system processes
    3. Suspending and resuming processes
    4. Providing mechanisms for process synchronization
    5. Providing mechanisms for process communication

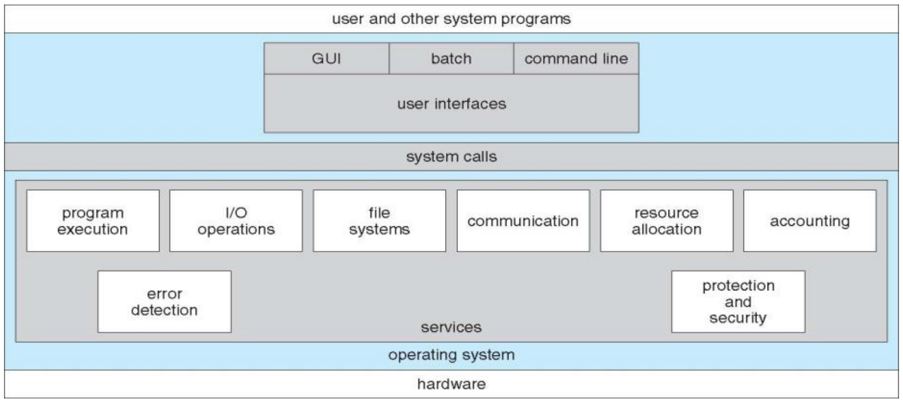
MEMORY MANAGEMENT

- To execute a program all (or part) of the instructions must be in memory .All (or part) of the data that is needed by the program must be in memory.
- Memory management determines what is in memory and Optimizing CPU utilization and computer response to users
- Memory management activities
    1. Keeping track of which parts of memory are currently being used and by whom
    2. Deciding which processes (or parts thereof) and data to move into and out of memory
    3. Allocating and deallocating memory space as needed

OPERATING-SYSTEM SERVICES

- Operating systems provide an environment for execution of programs and services to programs and users.
- One set of operating-system services provides functions that are helpful to the user:
    1. **User interface** - Almost all operating systems have a user interface (**UI**). Varies between **Command-Line (CLI)**, **Graphics User Interface (GUI)**.
    2. **Program execution** - The system must be able to load a program into memory and to run that program, end execution, either normally or abnormally (indicating error)
    3. **I/O operations** - A running program may require I/O, which may involve a file or an I/O device
    4. **File-system manipulation** - The file system is of particular interest. Programs need to read and write files and directories, create and delete them, search them, list file Information, permission management.
    5. **Communications** – Processes may exchange information, on the same computer or between computers over a network Communications may be via shared memory or through message passing (packets moved by the OS)
    6. **Error detection** – OS needs to be constantly aware of possible errors.
        a. May occur in the CPU and memory hardware, in I/O devices, in user program
        b. For each type of error, OS should take the appropriate action to ensure correct and consistent computing
        c. Debugging facilities can greatly enhance the user's and programmer's abilities to efficiently use the system

- Another set of OS functions exists for ensuring the efficient operation of the system itself via resource sharing
    a. **Resource allocation -** When multiple users or multiple jobs running concurrently, resources must be allocated to each of them . Many types of resources - CPU cycles, main memory, file storage, I/O devices.
    b. **Accounting -** To keep track of which users use how much and what kinds of computer resources
    c. **Protection and security -** The owners of information stored in a multiuser or networked computer system may want to control use of that information, concurrent processes should not interfere with each other **Protection** involves ensuring that all access to system resources is controlled
    d. **Security** of the system from outsiders requires user authentication, extends to defending external I/O devices from invalid access attempts.



SYSTEM CALLS

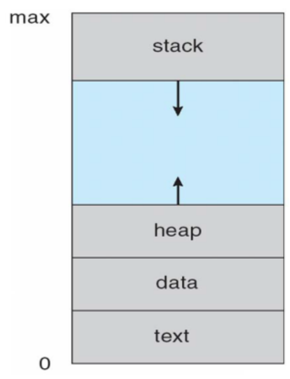
- System calls provide an interface to the services made available by an operating system.
- Typically written in a high-level language (C or C++)
- Mostly accessed by programs via a high-level **Application Programming Interface (API)** rather than direct system call use.

Types of System Calls

1. Process control
    a. create process, terminate process
    b. end, abort
    c. load, execute
    d. get process attributes, set process attributes
2. File management
    a. create file, delete file
    b. open, close file
    c. read, write, reposition
    d. get and set file attributes
3. Device management
    a. request device, release device
    b. read, write, reposition
    c. get device attributes, set device attributes
    d. logically attach or detach devices

PROCESSES

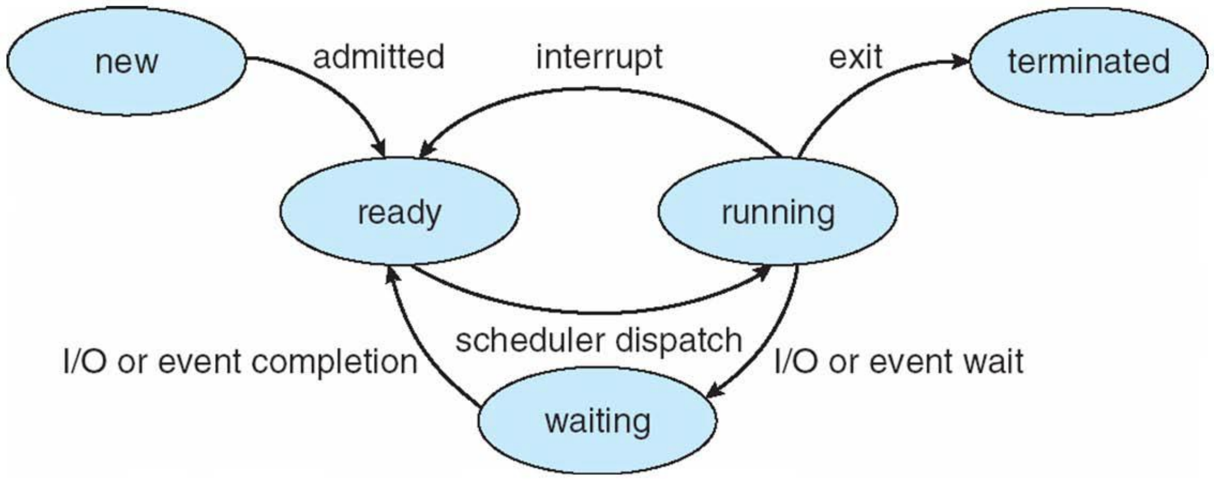
- Program is ***passive*** entity stored on disk (**executable file**), process is ***active.*** Program becomes process when executable file loaded into memory.
- A process contains multiple parts as follows:
  a. The program code, also called text section
  b. Current activity including program counter, processor registers
  c. Stack containing temporary data Function parameters, return addresses, local variables
  d. Data section containing global variables
  e. Heap containing memory dynamically allocated during run time.



Process States

As a process executes, it changes state .The states of a process are given as follows

i. new: The process is being created
ii. running: Instructions are being executed
iii. waiting: The process is waiting for some event to occur
iv. ready: The process is waiting to be assigned to a processor
v. terminated: The process has finished execution

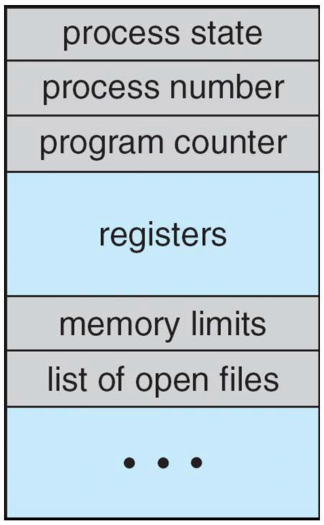


Process Control Block

- Each process is represented in the operating system by a process control block (PCB), also called a task control block.
- It contains many pieces of information associated with a specific process, including these:

a. **Process state.** The state may be new, ready, running, waiting, halted, and so on.
b. **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
c. **CPU registers**. The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.
d. **CPU-scheduling information**. This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
e. **Memory-management information.** This information may include such items as the value of the base and limit registers and the page tables, or the segment tables, depending on the memory system used by the operating system
f. **Accounting information.** This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on. This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

- Thus, the PCB simply serves as the repository for any information that may vary from process to process.



CONTEXT SWITCH

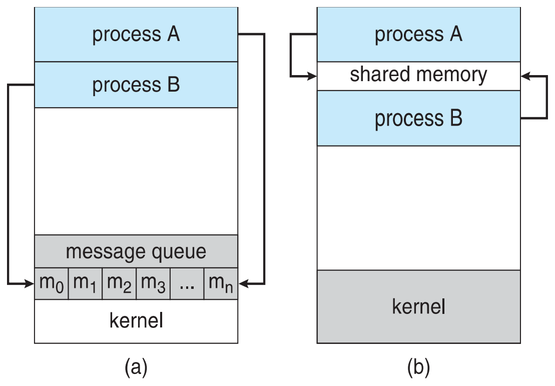
- Interrupts cause the operating system to change a CPU from its current task and to run a kernel routine.
- When an interrupt occurs, the system needs to save the current context of the process running on the CPU so that it can restore that context when its processing is done, essentially suspending the process and then resuming it.
- The context is represented in the PCB of the process. It includes the value of the CPU registers, the process state and memory-management information. We perform a state save of the current state of the CPU, be it in kernel or user mode, and then a state restore to resume operations.
- Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch.
- When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.
- Context-switch time is pure overhead, because the system does no useful work while switching.

INTERPROCESS COMMUNICATION

- Processes executing concurrently in the operating system may be either independent processes or cooperating processes.
- A process is independent if it cannot affect or be affected by the other processes executing in the system.
- Any process that does not share data with any other process is independent.
- A process is cooperating if it can affect or be affected by the other processes executing in the system. Thus, any process that shares data with other processes is a cooperating process.
- There are several reasons for providing an environment that allows process cooperation:
  a. **Information sharing**. Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
  b. **Computation speedup**. If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Such a speedup can be achieved only if the computer has multiple processing cores.
  c. **Convenience**. Even an individual user may work on many tasks at the same time. For instance, a user may be editing, listening to music, and compiling in parallel.

- 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: shared memory and message passing.
- In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region.
- In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.

Message passing   Shared memory.



Shared-Memory Systems

- Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. A shared-memory region resides in the address space of the process creating the shared-memory segment.
- Other processes that wish to communicate using this shared-memory segment must attach it to their address space. The operating system tries to prevent one process from accessing another process's memory.
- Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas.
- The producer–consumer problem, which is a common paradigm for cooperating processes. A producer process produces information that is consumed by a consumer process.
- One solution to the producer–consumer problem uses shared memory.
- To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.
- This buffer will reside in a region of memory that is shared by the producer and consumer processes.
- A producer can produce one item while the consumer is consuming another item. The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.
- Two types of buffers can be used. The unbounded buffer places no practical limit on the size of the buffer. The consumer may have to wait for new items, but the producer can always produce new items.
- The bounded buffer assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

Message-Passing Systems

- Another way to achieve the same effect is for the operating system to provide the means for cooperating processes to communicate with each other via a message-passing facility.
- Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. It is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.
- A message-passing facility provides at least two operations:
  send(message)   receive(message)

- If processes P and Q want to communicate, they must send messages to and receive messages from each other: a communication link must exist between them. This link can be implemented in a variety of ways.

THREADS

- 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.
- A traditional (or ***heavyweight***) process has a single thread of control. If a process has multiple threads of control, it can perform more than one task at a time.

BENEFITS:

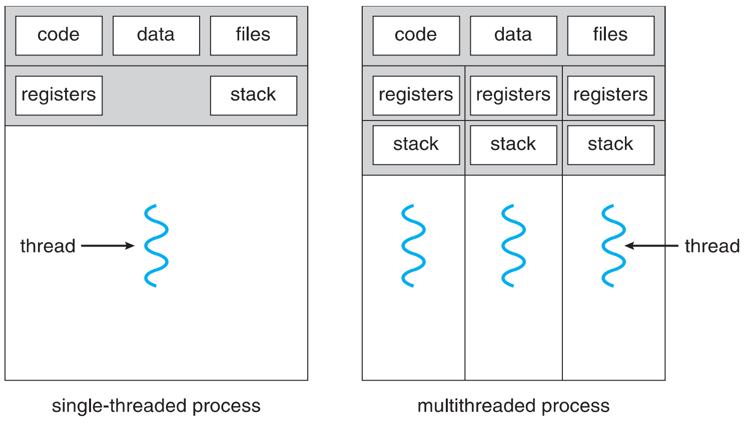
The benefits of multithreaded programming can be broken down into four major categories:

Responsiveness :- Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.

Resource sharing:- Threads share the memory and the resources of the process to which they belong by default. The benefit of sharing code and data is that it allows an application to have several different threads of activity within the same address space.

Economy :- Allocating memory and resources for process creation is costly. Because threads share the resources of the process to which they belong, it is more economical to create and context-switch threads.

Scalability. The benefits of multithreading can be even greater in a multiprocessor architecture, where threads may be running in parallel on different processing cores. A single-threaded process can run on only one processor, regardless how many are available.

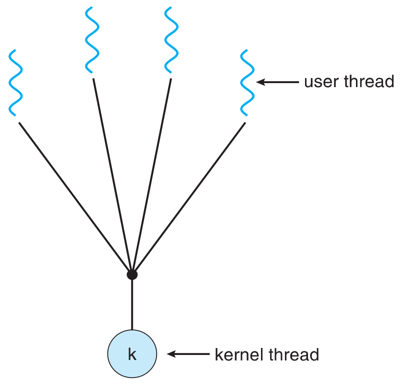


Single-threaded and multithreaded processes.

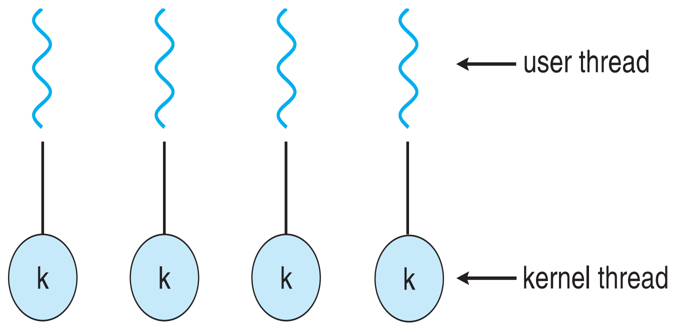
Multithreading Models

- 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.
- A relationship must exist between user threads and kernel threads. Three common ways of establishing such a relationship: the many-to-one model, the one-to-one model, and the many-to-many model.

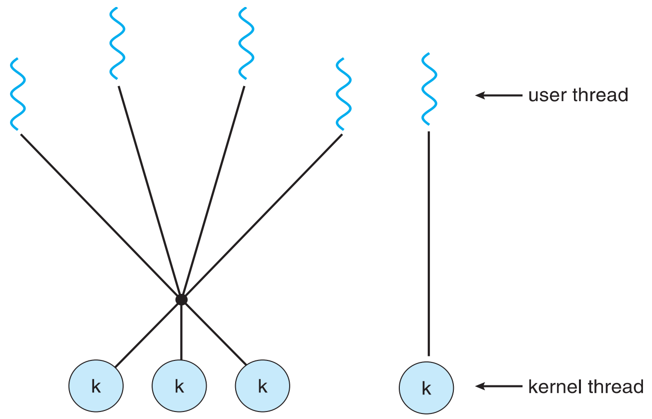
1. **Many-to-One Model**
   The many-to-one model maps many user-level threads to one kernel thread. Thread management is done by the thread library in user space, so it is efficient. However, the entire process will block if a thread makes a blocking system call.
   Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multicore systems.



2. **One-to-One Model**
   The one-to-one model maps each user thread to a kernel thread. It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call. It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.



3. **Many-to-Many Model**
The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads. The number of kernel threads may be specific to either a particular application or a particular machine. One variation on the many-to-many model still multiplexes many user level threads to a smaller or equal number of kernel threads but also allows a user-level thread to be bound to a kernel thread. This variation is sometimes referred to as the two-level model.



Thread Libraries

- ➢ A thread library provides the programmer with an API for creating and managing threads. There are two primary ways of implementing a thread library.
- ➢ The first approach is to provide a library entirely in user space with no kernel support. All code and data structures for the library exist in user space. This means that invoking a function in the library results in a local function call in user space and not a system call.
- ➢ The second approach is to implement a kernel-level library supported directly by the operating system. In this case, code and data structures for the library exist in kernel space. Invoking a function in the API for the library typically results in a system call to the kernel.
- ➢ Three main thread libraries are: POSIX PThreads, Windows, and Java.

a. PThreads, the threads extension of the POSIX standard, may be provided as either a user-level or a kernel-level library.
b. The Windows thread library is a kernel-level library available on Windows systems.
c. The Java thread API allows threads to be created and managed directly in Java programs

Threading Issues
Some of the issues to consider in designing multithreaded programs are as follows:

1. **The fork() and exec() System Calls:**
The fork() system call is used to create a separate, duplicate process. The semantics of the fork() and exec() system calls change in a multithreaded program. If one thread in a program calls fork(), does the new process duplicate all threads, or is the new process single-threaded? Some UNIX systems have chosen to have two versions of fork(), one that duplicates all threads and another that duplicates only the thread that invoked the fork() system call. The exec() system call typically works in the same way that is, if a thread invokes the exec() system call, the program specified in the parameter to exec() will replace the entire process—including all threads.

2. **Signal Handling**

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. All signals, whether synchronous or asynchronous, follow the same pattern:

a. A signal is generated by the occurrence of a particular event.
b. The signal is delivered to a process.
c. Once delivered, the signal must be handled.

A signal may be handled by one of two possible handlers:

a. A default signal handler
b. A user-defined signal handler

Every signal has a default signal handler that the kernel runs when handling that signal. This default action can be overridden by a user-defined signal handler that is called to handle the signal.

3. **Thread Cancellation**

Thread cancellation involves terminating a thread before it has completed. A thread that is to be canceled is often referred to as the target thread.

Cancellation of a target thread may occur in two different scenarios:

a. Asynchronous cancellation. One thread immediately terminates the target thread.
b. Deferred cancellation. The target thread periodically checks whether it should terminate, allowing it an opportunity to terminate itself in an orderly fashion.

PART 2
PROCESS SCHEDULING

Scheduling Criteria

Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms.
The criteria include the following:

a. CPU utilization. We want to keep the CPU as busy as possible. Conceptually, CPU utilization can range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily loaded system).

b. Throughput. If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be ten processes per second.

c. Turnaround time. From the point of view of a particular process, the important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

d. Waiting time. 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. Waiting time is the sum of the periods spent waiting in the ready queue.

e. Response time. The time from the submission of a request until the first response is produced. This measure, called response time, is the time it takes to start responding, not the time it takes to output the response.

Note: For Scheduling Algorithms study all the examples during classwork.

Multilevel Queue Scheduling

- A common division is made between foreground (interactive) processes and background (batch) processes.
- These two types of processes have different response-time requirements and so may have different scheduling needs.
- A multilevel queue scheduling algorithm partitions the ready queue into several separate queues. The processes are permanently assigned to one queue, generally based on some property of the process, such as memory size, process priority, or process type. Each queue has its own scheduling algorithm.
- For example, separate queues might be used for foreground and background processes. The foreground queue might be scheduled by an RR algorithm, while the background queue is scheduled by an FCFS algorithm.

Multilevel Feedback Queue Scheduling

- The multilevel feedback queue scheduling algorithm, allows a process to move between queues.
- The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue. This scheme leaves I/O-bound and interactive processes in the higher-priority queues.
- In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority queue. This form of aging prevents starvation.
- In general, a multilevel feedback queue scheduler is defined by the following parameters:
    - The number of queues
    - The scheduling algorithm for each queue
    - The method used to determine when to upgrade a process to a higher priority queue
    - The method used to determine when to demote a process to a lower priority queue
    - The method used to determine which queue a process will enter when that process needs service
- The definition of a multilevel feedback queue scheduler makes it the most general CPU-scheduling algorithm. It can be configured to match a specific system under design.

Thread Scheduling

- User-level threads are managed by a thread library, and the kernel is unaware of them. To run on a CPU, user-level threads must ultimately be mapped to an associated kernel-level thread, although this mapping may be indirect and may use a lightweight process (LWP).
- The various scheduling issues involving user-level and kernel-level threads are

  Contention Scope:

    - One distinction between user-level and kernel-level threads lies in how they are scheduled. On systems implementing the many-to-one and many-to-many models, the thread library schedules user-level threads to run on an available LWP. This scheme is known as **process contention scope** (**PCS**), since competition for the CPU takes place among threads belonging to the same process. To decide which kernel-level thread to schedule onto a CPU, the kernel uses **system-contention scope** (**SCS**). PCS is done according to priority—the scheduler selects the runnable thread with the highest priority to run. User-level thread priorities are set by the programmer and are not adjusted by the thread library, although some thread libraries may allow the programmer to change the priority of a thread.

    - PThreads:

  Pthread API allows specifying PCS or SCS during thread creation. Pthreads identifies the following contention scope values:

    - PTHREAD SCOPE PROCESS schedules threads using PCS scheduling.
    - PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

  On systems implementing the many-to-many model, the PTHREAD_SCOPE_PROCESS policy schedules user-level threads onto available LWPs.
  The PTHREAD_SCOPE_SYSTEM scheduling policy will create and bind an LWP for each user-level thread on many-to-many systems, effectively mapping threads using the one-to-one policy.

Multiple-Processor Scheduling

If multiple CPUs are available, load sharing becomes possible—but scheduling problems become correspondingly more complex.
One approach to CPU scheduling in a multiprocessor system has all scheduling decisions, I/O processing, and other system activities handled by a single processor—the master server.
The other processors execute only user code. This asymmetric multiprocessing is simple because only one processor accesses the system data structures, reducing the need for data sharing.
A second approach uses symmetric multiprocessing (SMP), where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its own private queue of ready processes.
On SMP systems, it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor.
Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system. There are two general approaches to load balancing: push migration and pull migration.
With push migration, a specific task periodically checks the load on each processor and—if it finds an imbalance—evenly distributes the load by moving (or pushing) processes from overloaded to idle or less-busy processors.
Pull migration occurs when an idle processor pulls a waiting task from a busy processor.

****