UNIT-II

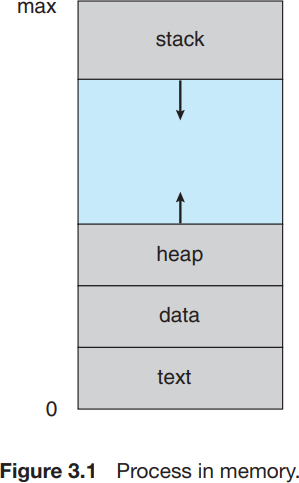
Processes

* A process can be thought of as a **program in execution**. 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.**
* Systems consist of a collection of processes: **operating-system processes execute system code**, and **user processes execute user code**. All these processes may execute concurrently.
* Although traditionally a **process contained** only **a single thread** of control as it ran, most modern operating systems **now support processes that have multiple threads.**
* The operating system is **responsible for** several important aspects of **process and thread management**: the creation and deletion of both user and system processes; the scheduling of processes; and the provision of mechanisms for synchronization, communication, and deadlock handling for processes.

# Process Concept

* + - A process is more than the program code or source code, which is sometimes known as the **text section.**
    - It also includes the current activity(address of the next instruction), as represented by the value of the **program counter** and the contents of the processor’s registers.
    - A process generally also includes the **process stack**, which contains temporary data (such as function parameters, return addresses, and local variables)
    - **Data section** of process contains global variables.
    - A process may also include a **heap**, which is memory that is dynamically allocated during process run time.

The structure of a process in memory is shown in Figure 3.1.



* + - **A program is a passive entity**, such as a file containing a list of instructions stored on disk (often called an executable file).
    - In contrast, 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
    - Two common techniques for loading executable files
      * 1) Double-click on icon representing the executable file and
      * 2) Enter the name of the executable file on the command line (as in prog.exe or a.out).

### Note:

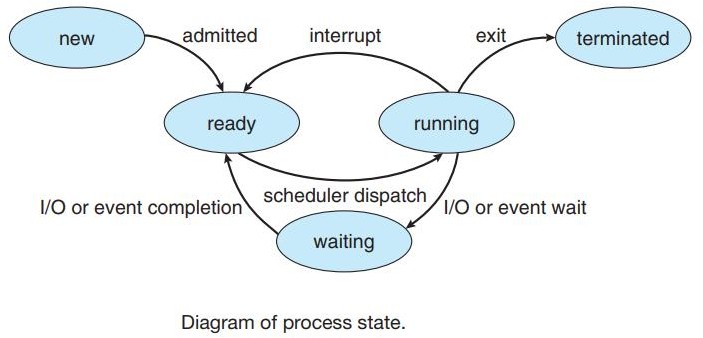
* + - Although two processes may be associated with the same program, they are considered as two separate execution sequences. Each of these is a separate process; have equivalent text sections, but data, heap, and stack sections vary.
    - Process itself can be an execution environment for other code. Java program is executed within the Java virtual machine (JVM). The JVM executes as a process that interprets the loaded Java code
    1. **Process State**

As a process executes, it changes state. The state of a process is defined in part by the current activity of that process.

A process may be in one of the following states:

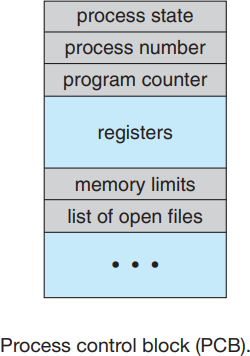
* + - * **New.** The process is being created.
      * **Ready.** The process is waiting to be assigned to a processor.
      * **Running.** Instructions are being executed.
      * **Waiting.** The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
    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

It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting, however. The state diagram corresponding to these states is presented in Figure below



## Process Control Block

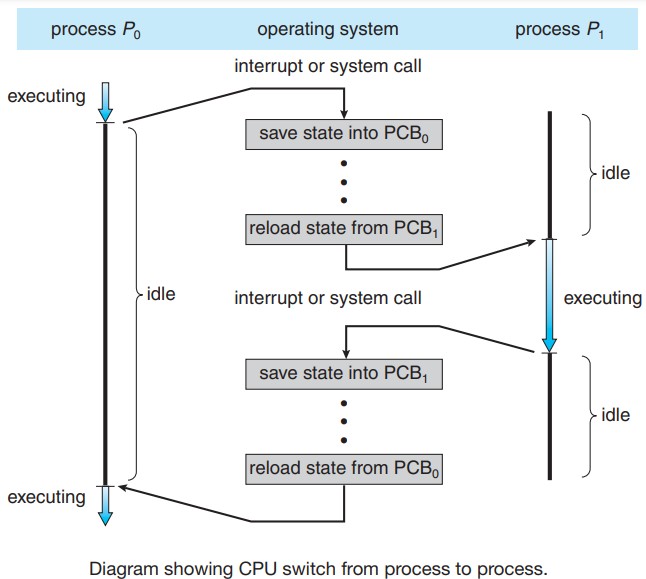
Each **process is represented** in the operating system by a **process control block (PCB)—also called a task control block**. A PCB is shown in Figure below



It contains many pieces of information associated with a specific process, including these:

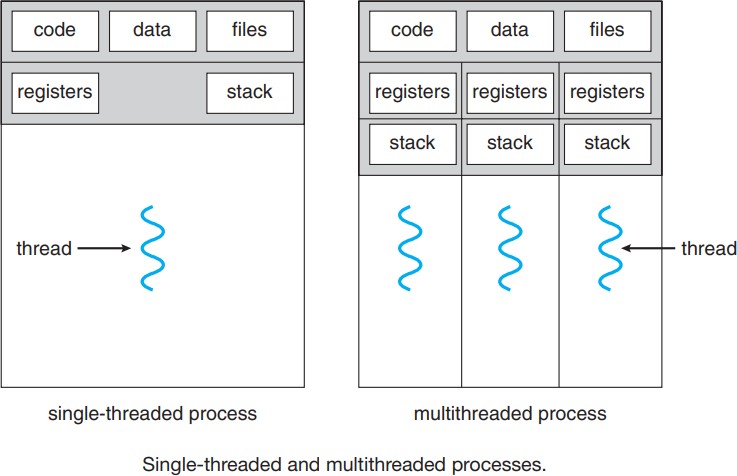
* + - * **Process state**. The state may be new, ready, running, waiting, halted, and so on.
      * **Program counter.** The counter indicates the address of the next instruction to be executed for this process.
      * **CPU registers.** The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers etc. Along with the program counter,

this state information must be saved when an interrupt occurs, to allow the process to be continued correctly afterward (Figure below).



* + - * **CPU-scheduling information.** This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
      * **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.
      * **Accounting information**. This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.
      * **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

## Threads

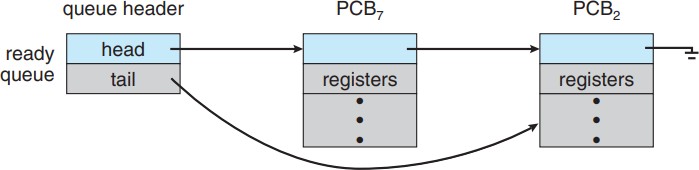
* + - * Process is a program that performs a single thread of execution. For example, when a process is running a word- processor program.
      * This single thread of control allows the process to perform only one task at a time. The user cannot simultaneously type in characters and run the spell checker.
      * Most modern operating systems allow a process to have multiple threads of execution and thus to perform more than one task at a time by, expanding the PCB is to include information for each thread.
      * Figure below illustrate the difference between single thread process and multi thread process

# Process Scheduling

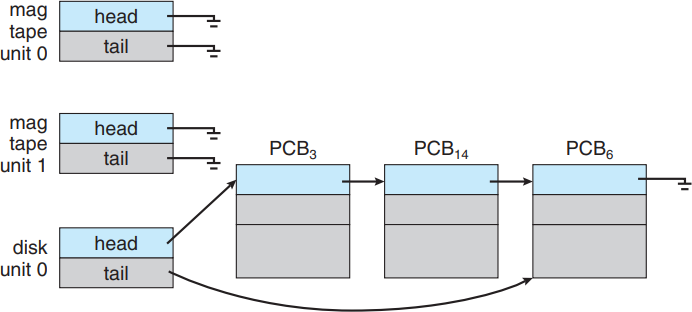
* + - For a **single-processor system, there will never be more than one running process**. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled
    - The objective of **multiprogramming** is to have some process running at all times, to **maximize CPU utilization**.
    - The objective of **time sharing** is to switch the CPU among processes so frequently that users can interact with each program
    - To meet these objectives, the process **scheduler selects an available process** (possibly from a set of several available processes) for program execution on the CPU.

## Process Scheduling

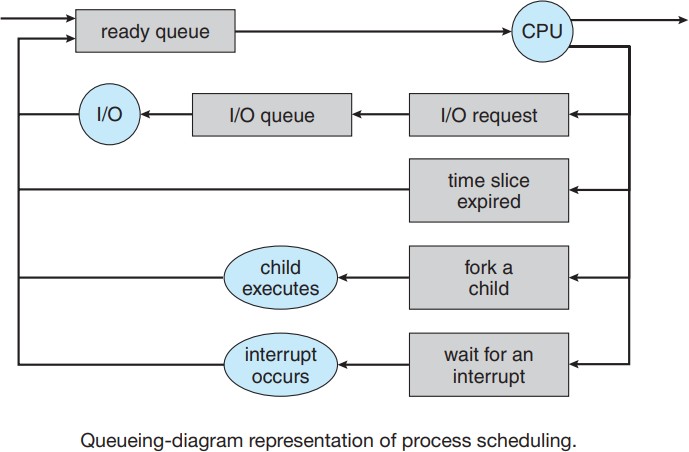
* + - * As processes enter the system, they are put into a **job queue**, which consists of all processes in the system.
      * The **processes that are residing in main memory** and are **ready and waiting to execute** are kept on a list called the **ready queue** (generally **stored as a linked list**)
      * A ready-queue **header contains pointers to the first and final PCBs in the list**. Each PCB includes a pointer field that points to the next PCB in the ready queue



* + - * The system also includes other queues. When a process is allocated the CPU, it executes for a while process may quit, and interrupted, or wait for the occurrence of a particular event, such as the completion of an I/O request.
      * For example, the list of processes waiting for a particular I/O device is called a **device queue**. Each device has its own device queue



* + - * A common representation of process scheduling is a **queueing diagram**, such as that in Figure below.

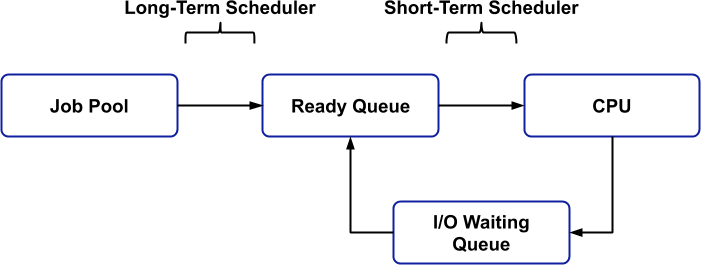


* + - * Each rectangular box represents a queue. **Two types of queues** are present: the **ready queue** and a **set of device queues**.
      * The **circles represent the resources** that serve the queues, and the **arrows indicate the flow of processes** in the system.
      * A **new process** is initially put **in the ready queue**. It waits there until it is selected for **execution, or dispatched.**
      * Once the process is allocated the CPU and is executing, one of several events could occur:
        + The process could issue an **I/O request** and then be placed in an I/O queue.
        + The process could create a new child process and **wait for the child’s termination**.
        + The process could be **removed forcibly from the CPU, as a result of an interrupt**, and be put back in the ready queue.
      * A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

## Schedulers

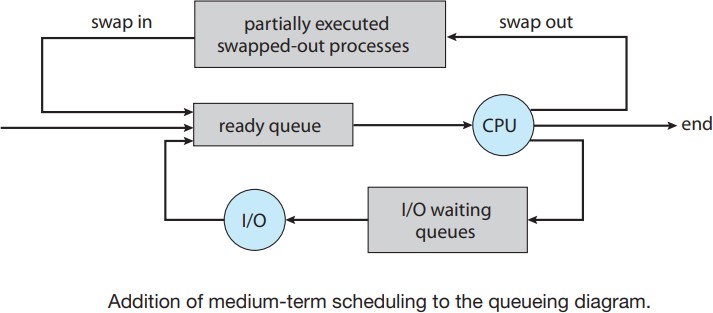
* + - * The selection process for execution is carried out by the scheduler.
      * Processes are **spooled to a mass-storage** device for later execution. **The long-term scheduler, or job scheduler**, selects processes **from this pool** and loads them **into memory for execution.**

### The short-term scheduler, or CPU scheduler, selects the processes that are ready to execute and allocates the

**CPU** to one of them.

|  |  |  |
| --- | --- | --- |
| **SNO** | **Long-term scheduler** | **Short-term scheduler** |
| 1 | Selects processes from this pool and loads them  into memory for execution. | Selects the processes that are ready to execute and  allocates the CPU to one of them. |
| 2 | The long-term scheduler executes much less frequently(minutes gap) | The short-term scheduler must select a new process for the CPU frequently(millisecond’s gap) |
| 3 | It executes when memory has enough space to  accommodate new process | It executes when CPU is available for allocation |
| 4 | Access Job pool & Ready Queue | Access Ready Queue and CPU |
| 5 | The long-term scheduler controls the degree of multiprogramming (the number of processes in  memory) | The long-term scheduler lesser control over the degree of multiprogramming |
| 6 | Long-term scheduler takes more time to decide which process should be selected for execution. | Short-term scheduler take less time to decide which process should be selected for execution. |

* + - * It is important that the long-term scheduler make a careful selection. In general, most processes can be described as either **I/O bound** or **CPU bound.**
* An **I/O-bound** process is one that spends more of its time doing I/O than it spends doing computations.
* An**CPU-bound** process, in contrast, generates I/O requests infrequently, using more of its time doing computations.
  + - * It is important that the long-term scheduler select a **good process mix of I/O-bound and CPU-bound** processes.
        + If all **processes are I/O bound**, the **ready queue will almost always be empty**, and the short-term scheduler will have little to do.
        + If all **processes are CPU bound**, the **I/O waiting queue will almost always be empty**, devices will go unused
      * Some operating systems, such as time-sharing systems has **medium-term scheduler**



* + - * The key idea behind a medium-term scheduler is that sometimes it can be advantageous to **remove a process from memory** and thus **reduce the degree of multiprogramming**. Later, the **process can be reintroduced into memory**, and its **execution can be continued where it left off**. This scheme is called **swapping**

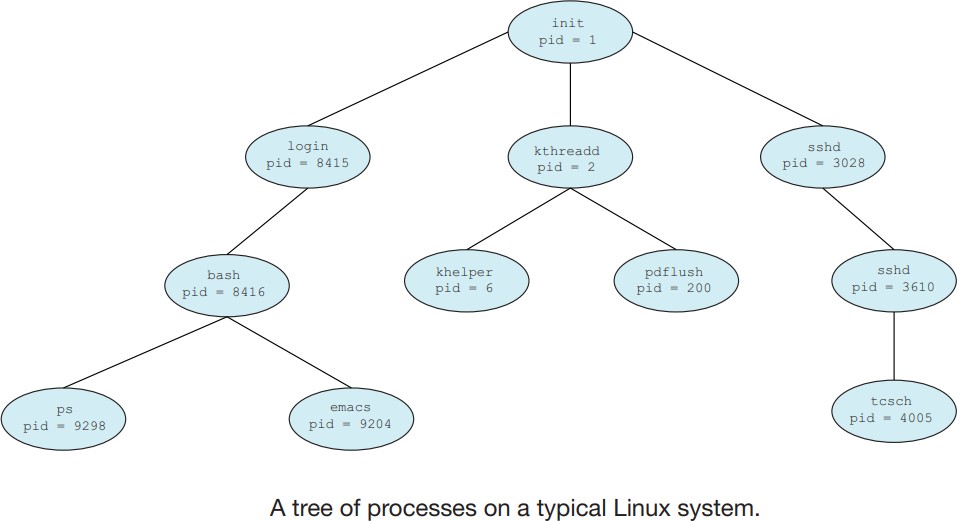
## Context Switch

* + - * 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**
      * 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
      * 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
      * Problems with context switch
        + Context-switch time is **pure overhead**, because the system **does no useful work while switching.**
        + Switching speed **varies from machine to machine**, depending on the memory speed, the number of registers that must be copied.
        + Context-switch times are **highly dependent on hardware support.**

# Operations on Processes

## Process Creation

* + - * During the course of execution, a process may create several new processes.
      * **Each of these new processes may in turn create other processes**, forming a tree of processes.
      * Most operating systems (including UNIX, Linux, and Windows) identify processes according to a **unique process identifier (or pid)**, which is typically an integer number.

* + - * Figure below illustrates a typical process tree for the Linux operating system, showing the name of each process and its pid.
      * The **init process** (which always has a **pid of 1**) **serves as the root parent process for all user processes**. Once the system has booted, the init process can also create various user processes
      * we see **two children of init—kthreadd and sshd**. The **kthreadd process is responsible for creating additional processes** that perform tasks on behalf of the kernel.
      * **The sshd process is responsible for managing clients that connect to the system by using ssh**(which is short for secure shell).
      * The **login process** is responsible for managing clients that directly log onto the system.
      * On UNIX and Linux systems, we can obtain a listing of processes by using the **ps command**. For example, the command

### ps -el

will list complete information for all processes currently active in the system.

* + - * In general, when a process creates a child process, that child process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.
        + A child process may be able to **obtain its resources directly from the operating system**, or
        + It may be constrained to a subset of the resources of the parent process. **The parent may have to partition its resources among its children**, or it may be able to share some resources
      * When a process creates a new process, two possibilities for execution exist:

1. The parent continues to **execute concurrently with its children**.
2. The **parent waits** until some or all of its children have terminated.
   * + - There are also two address-space possibilities for the new process:
3. The **child process is a duplicate of the parent process** (it has the same program and data as the parent).
4. The **child process has a new program loaded** into it. let’s first consider the UNIX operating system.
   * + - A **new process is created by the fork**() system call. The new process **consists of a copy of the address space of the original process.**

### The return code for the fork() is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

* + - * After a fork() system call, one of the two processes typically uses the **exec() system call to replace the process’s memory space with a new program.**
      * The **exec() system call loads a binary file into memory of process** and starts its execution.
      * The **parent can** then create more children; or, it can **issue a wait() system call** to move itself off the ready queue until the termination of the child process.

#include <sys/types.h> /\*psfork.c\*/ #include <stdio.h>

#include <unistd.h> int main()

{

pid\_tpid;

/\* fork a child process \*/ pid = fork();

if (pid< 0)

{ /\* error occurred \*/ printf("Fork Failed"); return 1;

}

else if (pid == 0)

{

/\* child process \*/ execlp("/bin/pwd","pwd",NULL); exit(32);

}

else

{

/\* parent process \*/

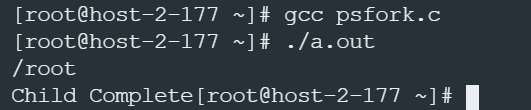
/\* parent will wait for the child to complete \*/

wait(&status); printf("Child Complete");

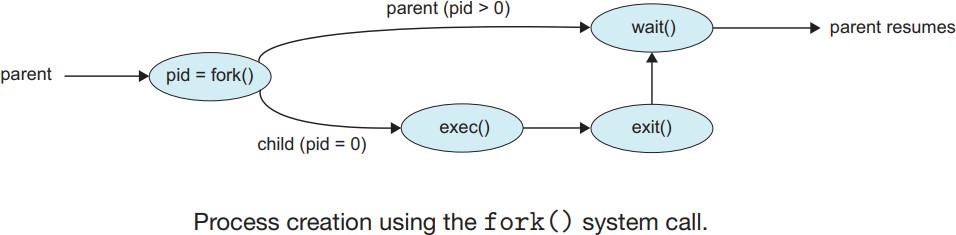
}

return 0;

}

Output:

* + - * When the child process completes, the parent process resumes from the call to wait(), where it completes using the exit() system call. This is also illustrated in the figure below



## Process Termination

* + - * A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.
      * At that point, the process may return a status value (typically an integer) to its parent process (via the wait() )
      * A parent may terminate the execution of one of its children for a variety of reasons, such as these:
        + The child has **exceeded its usage of some of the resources** that it has been allocated
        + The task assigned to the child is no longer required.
        + The parent is exiting, and the operating system does not allow a child to continue if its parent terminates**.**
      * Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates (either normally or abnormally), then all its children must also be terminated. This phenomenon, referred to as **cascading termination**
      * **When a process terminates, its resources are deallocated by the operating system**. However, its entry in the process table must **remain there until the parent calls wait().**A process that has terminated, but whose parent has not yet called wait(), is known as a **zombie process.**
      * If a parent did not invoke wait() and instead terminated, thereby leaving its child processes as **orphan processes**. All orphan process are adopted by **init process.**

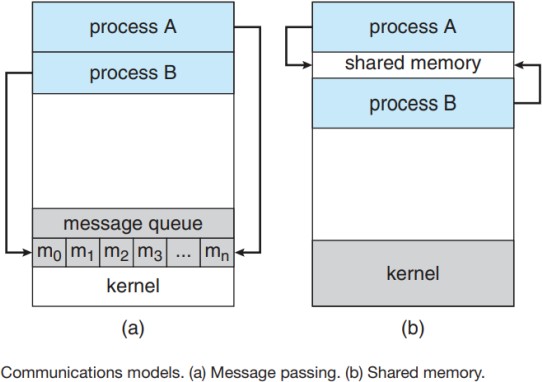
# Inter process Communication

* + - 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. Clearly, any process **that shares data** with other processes is a cooperating process.
    - There are several reasons for providing an environment that allows process cooperation:
      * **Information sharing.** Since several users may be interested in the same piece of information
      * **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.
      * **Modularity:** Dividing the system functions into separate processes or thread
      * **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:

* 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** is useful for exchanging **smaller amounts of data**. Message passing is also **easier to implement**

than shared memory **using system calls.**

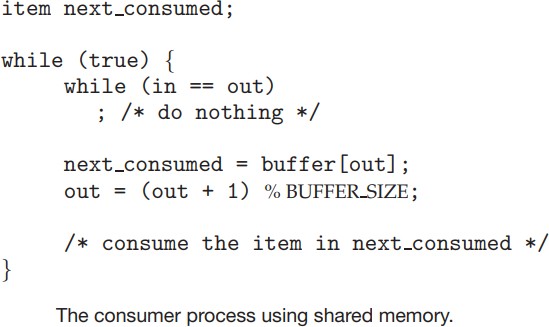
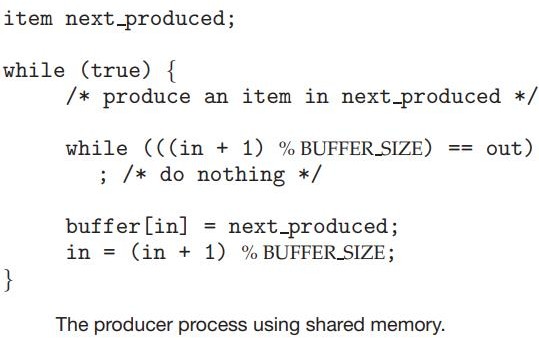
* **Shared memory** can be **faster** than message passing, since **message-passing** systems are typically implemented

### using system calls requires 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 memory accesses, and no assistance from the kernel is required.
  + 1. **Shared-Memory Systems**
* Interprocess communication using shared memory requires communicating processes to establish a region of shared memory.
* Processes that wish to communicate using this shared-memory segment must attach it to their address space.
* They can then exchange/communicate information by reading and writing data in the shared areas. Using shared needs to **remove the restriction “prevent one process from accessing another process’s memory”.**

To illustrate the concept of cooperating processes, let’s consider the **producer–consumer problem**, which is a common paradigm for cooperating processes. **A producer process produces information that is consumed by a consumer process.**

* 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.
* The code for the producer process and the code for the consumer process is shown below



* The shared buffer is implemented as a circular array with two logical pointers: **in and out.**
* The variable **“in”** points to the **next free position** in the buffer; **“out”** points to **the first full position in the buffer**. The **buffer is empty when in == out**; the **buffer is full when ((in + 1) % BUFFER SIZE) == out.**
* The producer process has a local variable **next\_produced** in which the new item to be produced is stored. The consumer process has a local variable **next\_consumed** in which the item to be consumed is stored.
* 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

* In the message-passing model, communication takes place by means of messages exchanged between the cooperating processes.
* Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space.
* 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.
* Here are several methods for logically implementing a link and the send()/receive() operations:
  + Direct or indirect communication
  + Synchronous or asynchronous communication
  + Automatic or explicit buffering

### Direct Communication

* Each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the **send()** and **receive()** primitives are defined as:

### send(P, message)—Send a message to process P. receive(Q, message)—Receive a message from process Q.

* A communication link in this scheme has the following properties:
  + A link is **established automatically** between processes that want to communicate. The processes need to know only **each other’s identity** to communicate.
  + A link is associated with **exactly two processes**.
  + Between each pair of processes, there exists **exactly one link**.
* This scheme exhibits **symmetry in addressing**; that is, both the sender process and the receiver process must name the other to communicate.
* A variant of this scheme employs **asymmetry in addressing**. Here, **only the sender names the recipient**; the **recipient is not required to name the sender**. In this scheme, the send() and receive() primitives are defined as follows:
  + **send(P, message)**—Send a message to process P.
  + **receive(id, message)**—Receive a message from any process. The variable id is set to the name of the process with which communication has taken place.

### Indirect Communication

* With indirect communication, the messages are sent to and received from mailboxes, or ports.
* A **mailbox** can be viewed **abstractly as an object** into which messages can be placed by processes and from which messages can be removed.Each mailbox **has a unique identification.**
* The send() and receive() primitives are defined as follows:
  + send(A, message)—Send a message to mailbox A.
  + receive(A, message)—Receive a message from mailbox A.
* In this scheme, a communication link has the following properties:
  + A link is established between a pair of processes only if both members of the pair have a **shared mailbox.**
  + A link may be associated with **more than two processes.**
  + Between each pair of communicating processes, **a number of different links may exist,** with **each link corresponding to one mailbox.**

### Synchronization

* Communication between processes takes place through calls to send() and receive() primitives. There are different design options for implementing each primitive.

### Message passing may be either blocking or nonblocking— also known as synchronous and asynchronous.

* + **Blocking send.** The sending process is blocked until the message is received by the receiving process or by the mailbox.
  + **Nonblocking send.** The sending process sends the message and resumes operation.
  + **Blocking receive.** The receiver blocks until a message is available.
  + **Nonblocking receive.** The receiver retrieves either a valid message or a null
  + When both send() and receive() are blocking, we have a **rendezvous** between the sender and the receiver.

### Buffering

* Whether communication is direct or indirect, messages exchanged by communicating processes reside in a

**temporary queue(buffer)**. Basically, such queues can be implemented in three ways:

* **Zero capacity.** The queue has a maximum length of zero; thus, the **link cannot have any messages waiting in it.**
* **Bounded capacity.** The queue has finite **length “n”**; thus, **at most“n” messages can reside in it.**
* **Unbounded capacity**. The queue’s length is potentially infinite; thus, **any number of messages can wait in it**. The sender never blocks.

The zero-capacity case is sometimes referred to as a message system with no buffering. The other cases are referred to as systems with automatic buffering.

# Multithreaded Programming

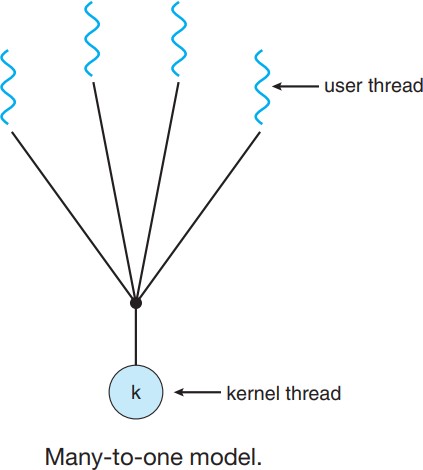
* Threads may be provided either at the user level, for **user threads**, or by the kernel, for **kernel threads.**
* **User threads** are supported above the kernel and are **managed without kernel support**, whereas **kernel threads**

are supported and **managed directly by the operating system**.

* A relationship must exist between user threads and kernel threads.
  + Many-To-One Model
  + One-To-One Model
  + Many-To-Many Model.

## 1.Many-to-One Model

* The many-to-one model maps many user-level threads to one kernel thread.

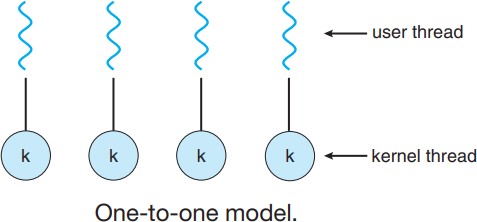
2

* 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

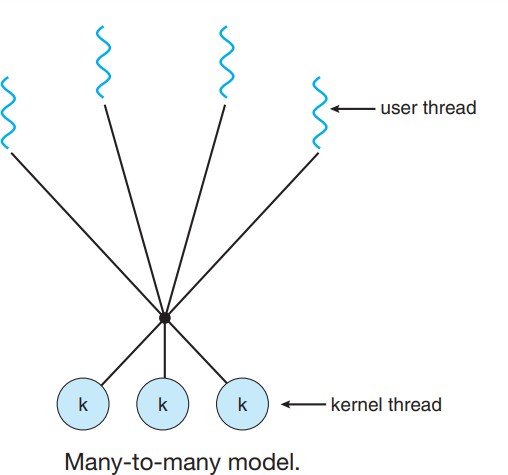
## 2.One-to-One Model

* 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.



## Many-to-Many Model

* The many-to-many model **multiplexes many user-level threads to a smaller or equal number of kernel threads**.



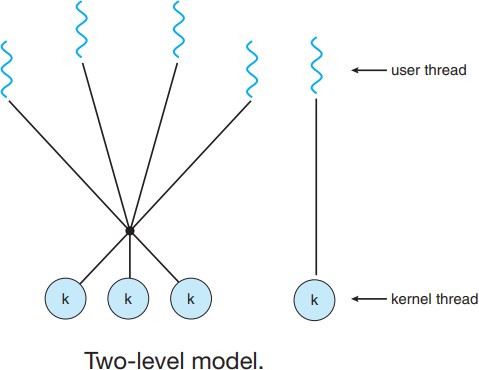
* The number of kernel threads may be specific to either a particular application or a particular machine (an application may be allocated more kernel threads on a multiprocessor than on a single processor).

### Disadvantages of previous model

* The Many to-one model **allows the developer to create as many user threads** as she wishes, it does **not result in true concurrency**, because the kernel can **schedule only one thread at a time.**
* The one-to-one model allows **greater concurrency**, but the developer has to be **careful not to create too many threads within an application**
* 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. **Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.**

## Two-level model

* 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**



# Process Scheduling (CPU Scheduling)

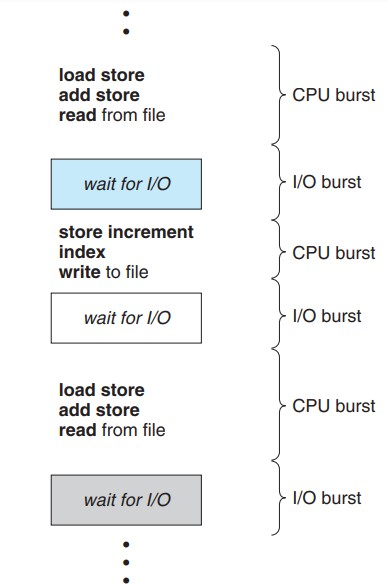
## Basic Concepts

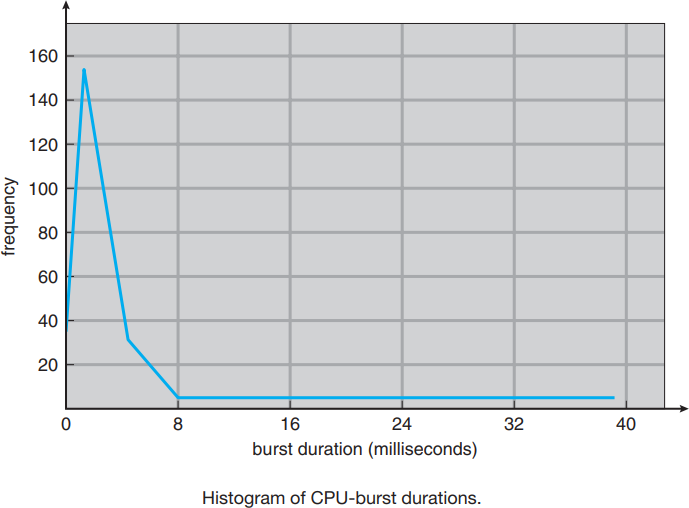
* In a **single-processor system, only one process can run at a time.** Others must wait until the CPU is free and can be rescheduled.
* If that process waits typically for the completion of some I/O request, then CPU then just sits idle. All this waiting time is wasted.
* The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization

With multiprogramming, several processes are kept in memory at one time. If one process has to wait, the operating system takes the CPU away from that process and gives the CPU to another process.This process is called **CPU Scheduling or Process Scheduling**

###  CPU –I/O Burst Cycle

* The success of CPU scheduling depends on an observed property of processes: 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.



* The durations of CPU bursts vary greatly from process to process and from computer to computer, they tend to have a **large number of short CPU bursts** and a **small number of long CPU bursts.**
* An **I/O-bound program typically has many short CPU bursts**. A **CPU-bound program might have a** few long CPU bursts.

## Scheduling Criteria

* Different CPU-scheduling algorithms have different properties, and the choice of a particular algorithm may favor one class of processes over another.
* Many criteria or properties have been suggested for comparing CPU-scheduling algorithms.

### 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).

### 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.

### Turnaround time

* 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.

### Waiting time.

* 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.

### Response time.

* In an interactive system, turnaround time may not be the best criterion.
* Thus, another measure is 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.

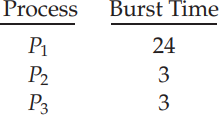
**It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.**

## 2.6.2Scheduling Algorithms

* CPU scheduling deals with the problem of deciding which of the processes in the ready queue is to be allocated the CPU. There are many different CPU-scheduling algorithms.

### 1.First-Come, First-Served Scheduling

* By far the simplest CPU-scheduling algorithm is the first-come, first-served (FCFS) scheduling algorithm. With this scheme, the **process that requests the CPU first is allocated the CPU first.**
* The implementation of the FCFS **easily managed with a FIFO queue**. When a **process enters the ready queue, its PCB is linked onto the tail of the queue**. When the CPU is free, it is allocated to the process at the head of the queue.
* The code for FCFS scheduling is **simple to write and understand**
* Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:



* If the processes arrive in the order P1, P2, P3, and are served in FCFS order, we get the result shown in the following **Gantt chart**, which is a bar chart that illustrates a particular schedule, including the start and finish times of each of the participating processes:



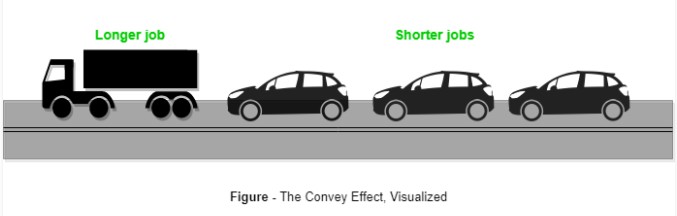
* The waiting time is
  + 0 milliseconds for process P1
  + 24 milliseconds for process P2, and
  + 27 milliseconds for process P3

### Thus, the average waiting time is (0 + 24 + 27)/3 = 17 milliseconds

* If the processes arrive in the order P2, P3, P1, however, the results will be as shown in the following Gantt chart:



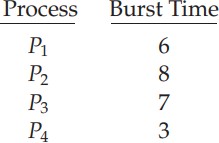
* The **average waiting time is now (6 + 0 + 3)/3 = 3 milliseconds**. This reduction is substantial
* **“Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the processes’ CPU burst times vary greatly”**



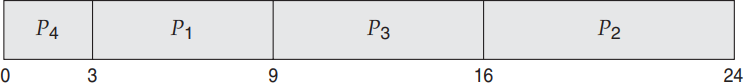
* There is a **convoy effect as all the other processes wait for the one big process to get off the CPU**. This effect results in lower CPU and device utilization than might be possible if the shorter processes were allowed to go first.
* **Note: FCFS scheduling algorithm is nonpreemptive**. Once the CPU has been allocated to a process, that process keeps the CPU until it releases the CPU, either by terminating or by requesting I/O.

### 2.Shortest-Job-First Scheduling (Shortest-next CPU-burst algorithm)

* In this algorithm **CPU is assigned to the process that has the smallest next CPU burst.**
* If the next CPU bursts of two processes are the same**, FCFS scheduling is used to break the tie.**
* As an example of SJF scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:



* Using SJF scheduling, we would schedule these processes according to the following Gantt chart:



* The waiting time is
  + 3 milliseconds for process P1,
  + 16 milliseconds for process P2,
  + 9 milliseconds for process P3, and
  + 0 milliseconds for process P4.

### Thus, the average waiting time is (3 + 16 + 9 + 0)/4 = 7 milliseconds.

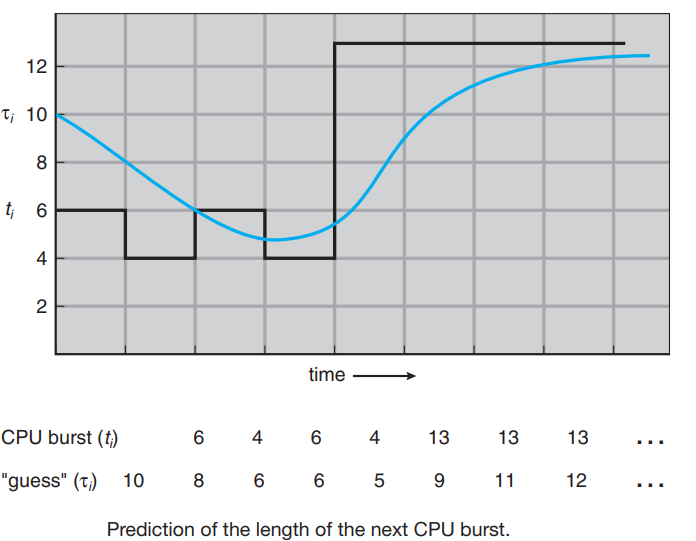
* By comparison, if we were using the FCFS scheduling scheme, the average waiting time **would be milliseconds.**
* The SJF scheduling algorithm is provably **optimal**, in that **it gives the minimum average waiting time** for a given set of processes.

### The real difficulty with the SJF algorithm knows the length of the next CPU request.

* **SJF algorithm is used long term scheduling**. We can use the **process time limit that a user specifies** when he submits the job.
* SJF algorithm **cannot be implemented at the level of short-term CPU scheduling**. With short-term scheduling, **there is no way to know the length of the next CPU burst**. One **approach to this problem is to try to approximate SJF scheduling.**
* **We expect that the next CPU burst will be similar in length to the previous ones**. **By computing an approximation** of the length of the next CPU burst, we can pick the process with the shortest predicted CPU burst.
* The next CPU burst is generally predicted as an **exponential average** of the measured lengths of previous CPU bursts. We can define the exponential average with the following formula



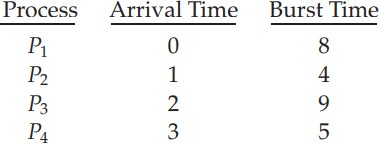
* Let **tn** be the length of the nth CPU burst. The value of **tn** contains our most recent information
* while stores the past history, let be our predicted value for the next CPU burst
* The parameter controls the relative weight of recent and past history in our prediction.Then for 1
* Figure below shows an exponential average **with = 1/2 and = 10.**



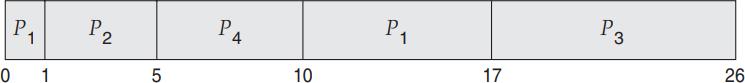
* If = 0, then = , and recent history has no effect (current conditions are assumed to be transient).
* If = 1, then = tn, and only the most recent CPU burst matters (history is assumed to be old and irrelevant).
* The SJF algorithm can be either **preemptive** or **nonpreemptive.**
* The choice arises when a new 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.

### A preemptive SJF algorithm will preempt the currently executing process

* **A non preemptive SJF algorithm will allow the currently running process to finish its CPU burst.**
* Preemptive SJF scheduling is sometimes called **shortest-remaining-time-first scheduling.**
* As an example, consider the following four processes, with the length of the CPU burst given in milliseconds:



* If the processes arrive at the ready queue at the times shown and need the indicated burst times, then the resulting preemptive SJF schedule is as depicted in the following Gantt chart:

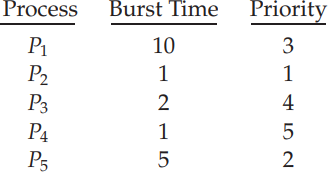


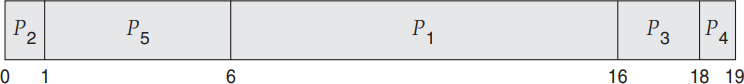
* Process P1 is started at time 0, since it is the only process in the queue. Process P2 arrives at time 1. The remaining time for process P1 (7 milliseconds) is larger than the time required by process P2 (4 milliseconds), so process P1 is preempted, and process P2 is scheduled.

### The average waiting time for this example is [(10 − 1) + (1 − 1) + (17 − 2) + (5 − 3)]/4 = 26/4 = 6.5 milliseconds.

* **Nonpreemptive SJF scheduling** would result in an average waiting time of **milliseconds.**

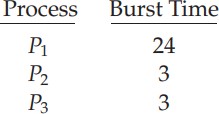
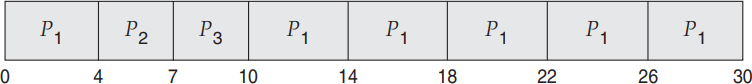
### 3.Priority Scheduling

* The SJF algorithm is a special case of the general 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 in FCFS order.
* Note that we discuss scheduling in terms of high priority and low priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority.
* As an example, consider the following set of processes, assumed to have arrived at time 0 in the order P1, P2, ···, P5, with the length of the CPU burst given in milliseconds:
* Using priority scheduling, we would schedule these processes according to the following Gantt chart:



* The average waiting time is 8.2 milliseconds
* Priority scheduling can be either preemptive or non preemptive.
  + A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.
  + A nonpreemptive priority scheduling algorithm will simply put the new process at the head of the ready queue.
* A major problem with priority scheduling algorithms is **indefinite blocking, or starvation**. A priority scheduling algorithm can leave some low priority processes waiting indefinitely.
* A solution to the problem of indefinite blockage of low-priority processes is **aging**. Aging involves gradually increasing the priority of processes that wait in the system for a long time.

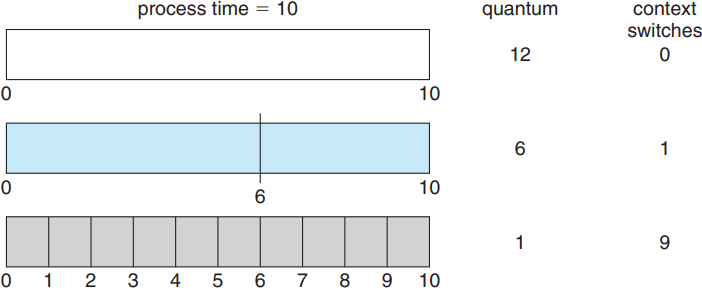
### 5.Round-Robin Scheduling

* The round-robin (RR) scheduling algorithm is designed especially for timesharing systems.
* It is similar to FCFS scheduling, but preemption is added to enable the system to switch between processes.
* A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds in length.
* The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to **1 time quantum.**
* To implement RR scheduling, new processes are added to the tail of the ready queue. **The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process.**
* One of two things will then happen.
  + The process may have a CPU burst of less than 1 time quantum. In this case, the process itself **will release the CPU voluntarily**. The CPU scheduler will then select the next process in the ready queue.
  + If the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an **interrupt to the process** will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue.
* Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:
* If we use a time quantum of 4 milliseconds
* Process P1 gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process P2
* Process P2 does not need 4 milliseconds, so it quits before its time quantum expires. The CPU is then given to the next process, process P3.
* Once each process has received 1 time quantum, the CPU is returned to process P1 for an additional time quantum. The resulting RR schedule is as follows:
* Let’s calculate the average waiting time for this schedule
  + P1 waits for 6 milliseconds (10 - 4)
  + P2 waits for 4 milliseconds, and
  + P3 waits for 7 milliseconds.
  + Thus, the average waiting time is 17/3 = 5.66 milliseconds.
* In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process’s CPU burst exceeds 1 time quantum, that process is preempted and is put back in the ready queue. **The RR scheduling algorithm is thus preemptive.**
* If there are **n** processes in the ready queue and the time quantum is **q**, Each process must wait no longer than

### (n − 1) × q time units until its next time quantum.

**Performance of RR scheduling algorithm**

* The performance of the RR algorithm depends heavily on the size of the time quantum
  + If the time quantum is extremely large, the RR policy is the same as the FCFS
  + In contrast, if the time quantum is extremely small (say, 1 millisecond), the RR approach can result in a large number of context switches.



* Assume, for example, that we have only one process of 10 time units.
* If the quantum is 12 time units, the process finishes in less than 1 time quantum, with no overhead.
* If the quantum is 6 time units, however, the process requires 2 quanta, resulting in a context switch.
* If the time quantum is 1 time unit, then nine context switches will occur, slowing the execution of the process.
* *Thus, we want the time quantum to be large with respect to the context switch time. If the context-switch time is approximately 10 percent of the time quantum, then about 10 percent of the CPU time will be spent in context switching.*

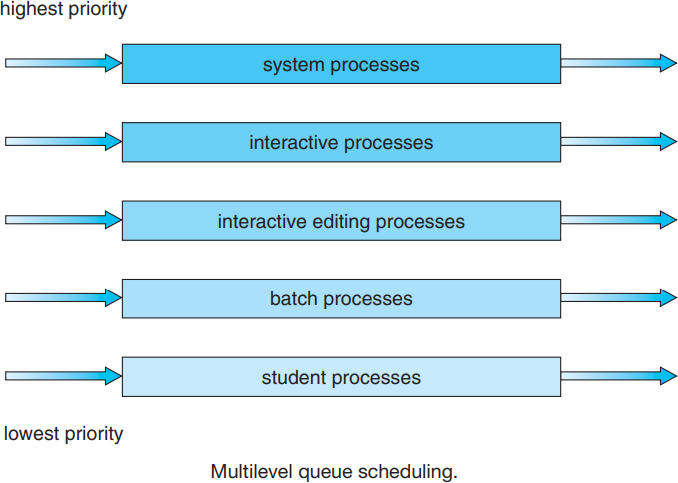
### Turnaround time also depends on the size of the time quantum.

* Although the time quantum should be large compared with the context switch time, it should not be too large. As we pointed out earlier, if the time quantum is too large, RR scheduling degenerates to an FCFS policy.

### A rule of thumb is that 80 percent of the CPU bursts should be shorter than the time quantum.

**6.Multilevel Queue Scheduling**

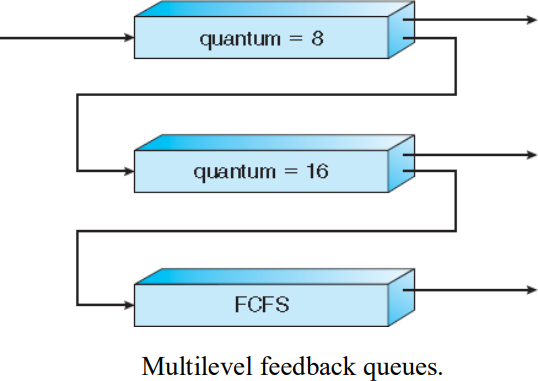
* Another class of scheduling algorithms has been created for situations in which processes are easily classified into different groups.
* **For example**, a common division is made between **foreground (interactive) processes and background (batch) processes**. These two types of processes have different 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.
  + The foreground queue might be scheduled by an RR algorithm,
  + While the background queue is scheduled by an FCFS algorithm.
* Let’s look at an example of a multilevel queue scheduling algorithm with five queues, listed below in order of priority:
  + - * 1. System processes
        2. Interactive processes
        3. Interactive editing processes
        4. Batch processes
        5. Student processes
* In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.



* If an interactive editing process entered the ready queue while a batch process was running, the batch process would be preempted.

### 7.Multilevel Feedback Queue Scheduling

* Normally, when the multilevel queue scheduling algorithm is used, processes are permanently assigned to a queue when they enter the system.
* If there are separate queues for foreground and background processes, for example, processes do not move from one queue to the other, since processes do not change their foreground or background nature.
* This setup has the advantage of low scheduling overhead, but it is inflexible.
* The **multilevel feedback queue scheduling algorithm**, in contrast, 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.



* For example, consider a multilevel feedback queue scheduler with three queues, numbered from 0 to 2 (above Figure).

*The scheduler first executes all processes in queue 0.

*Only when queue 0 is empty will it execute processes in queue 1.

*Similarly, processes in queue 2 will be executed only if queues 0 and 1 are empty.

* A process that arrives for queue 1 will preempt a process in queue 2.
* A process in queue 1 will in turn be preempted by a process arriving for queue 0.
* A process entering the ready queue is put in queue 0.
* A process in queue 0 is given a time quantum of 8 milliseconds. If it does not finish within this time, it is moved to the tail of queue 1.
* If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 milliseconds. If it does not complete, it is preempted and is put into queue 2.
* Processes in queue 2 are run on an FCFS basis but are run only when queues 0 and 1 are empty
* This scheduling algorithm gives highest priority to any process with a CPU burst of 8 milliseconds or less. Such a process will quickly get the CPU, finish its CPU burst, and go off to its next I/O burst.
* Processes that need more than 8 but less than 24 milliseconds are also served quickly, although with lower priority than shorter processes.
* Long processes automatically sink to queue 2 and are served in FCFS order with any CPU cycles left over from queues 0 and 1.
* 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
* This algorithm can be configured to match a specific system under design. **Unfortunately, it is also the most complex algorithm.**