UNIT-II

PROCESS MANAGEMENT

Syllabus

1. *Process Concept*
2. *Process Scheduling*
3. *Operations*
4. *Inter Process Communication*
5. *Multi Thread Programming Model*
6. *Process Scheduling criteria and algorithms and their evaluation*

Introduction

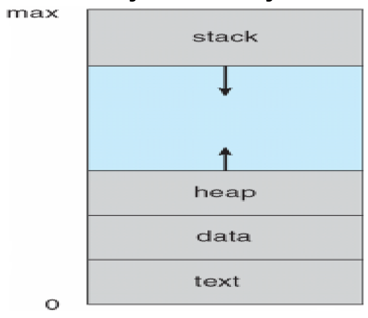
A *process* can be thought of as a program in execution, A process will need certain resources—such as CPU time, memory, files, and I/O devices —to accomplish its task. These resources are allocated to the process either when it is created or while it is executing.

Systems consist of a collection of processes: Operating-system processes execute system code, and user processes execute user code.

The operating system is responsible for the following activities in connection with 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.

2.1.Process Concept

- ❖ An operating system executes a variety of programs:
  - ➢ Batch system – jobs
  - ➢ Time-shared systems – user programs or tasks
- ❖ Textbook uses the terms *job* and *process* almost interchangeably.
- ❖ Process – a program in execution; process execution must progress in sequential fashion.
- ❖ A process includes:
  - ➢ **program counter** and the contents of the processor's registers
  - ➢ **stack -** which contains temporary data (such as function parameters, return addresses, and local variables),
  - ➢ **data section** – which contains global variables.
  - ➢ **heap,** which is memory that is dynamically allocated during process run time.



A program by itself is not a process; a program is a *passive* entity, such as a file containing a list of instructions stored on disk (often called an **executable file),** whereas 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.

2.1.1 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. Each process may be in one of the following states:
• **New.** The process is being created.
• **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).
• **Ready.** The process is waiting to be assigned to a processor.
• **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 *limiting*, however. The state diagram corresponding to these states is presented in below Fig.

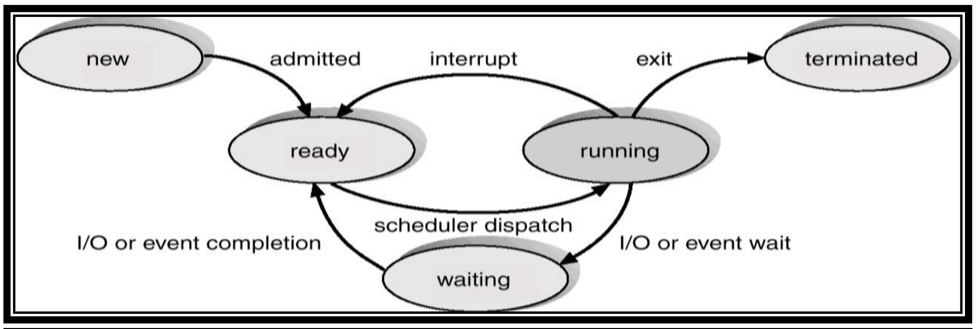
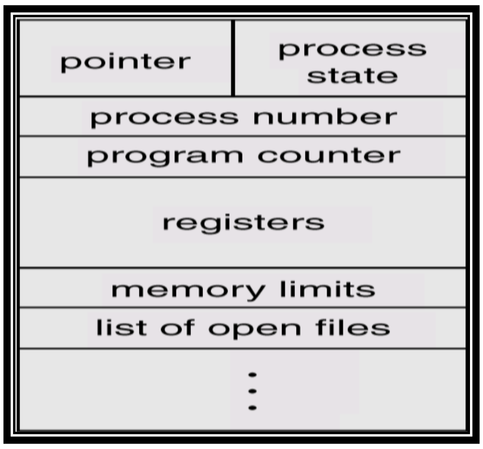


Diagram of process state

2.1.2. 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 3.3.



Process control block (PCB).

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, plus any condition-code information. 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 3.4).
- **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 information as the value of the base and limit registers, 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 members, 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.

2.1.3. Threads

A process is a program that performs a single **thread** of execution. For example, when a process is running a word-processor program, a single thread of instructions is being executed. This single thread of control allows the process to perform only one task at one time. The user cannot simultaneously type in characters and run the spell checker within the same process, for example. Many modern operating systems have extended the process concept to allow a process to have multiple threads of execution and thus to perform more than one task at a time.

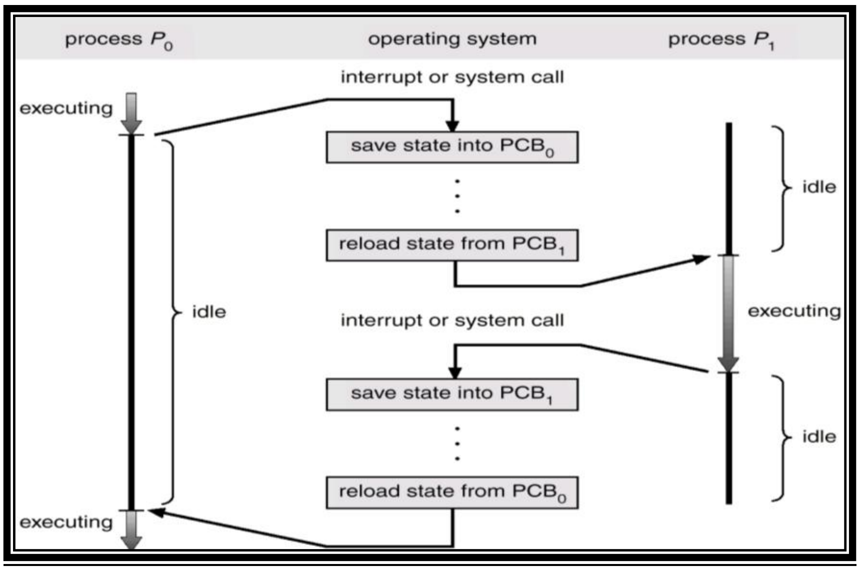


Diagram showing CPU switch from process to process

2.2. Process Scheduling

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. 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. 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.

2.2.1. Scheduling Queues

- **Job queue** – set of all processes in the system.
- **Ready queue** – set of all processes residing in main memory and are ready and waiting to execute. This queue is 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.
- **Device queues** – set of processes waiting for an I/O device.
- Process migration between the various queues.

The process control block in the Linux operating system is represented by the C structure *task_struct*. This structure contains all the necessary information for 'representing a process, including the state of the process, scheduling and memory management information, list of open files, and pointers to the process's parent and any of its children. (A process's *parent* is the process that created it; its *children* are any processes that it creates.) Some of these fields include:

```
pid_t pid;       /* process identifier */
long state;      /* state of the process */
unsigned int time..slice;       /* scheduling information */
struct files_struct *files;     /*list of open files */
struct mm_struct *mm;       /*• address space of this process */
```

For example, the state of a process is represented by the field long state in this structure. Within the Linux kernel, all active processes are represented using a doubly linked list of task_struct, and the kernel maintains a pointer — current — to the process currently executing on the system. This is shown in below Figure.

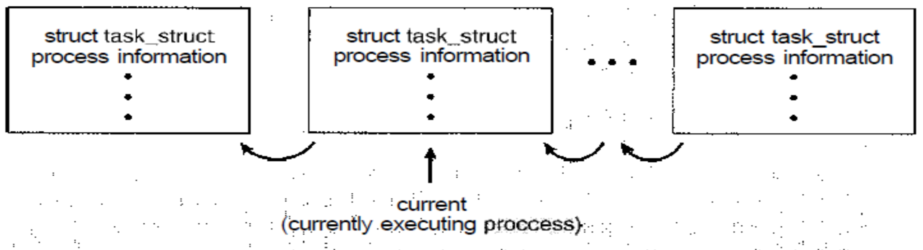
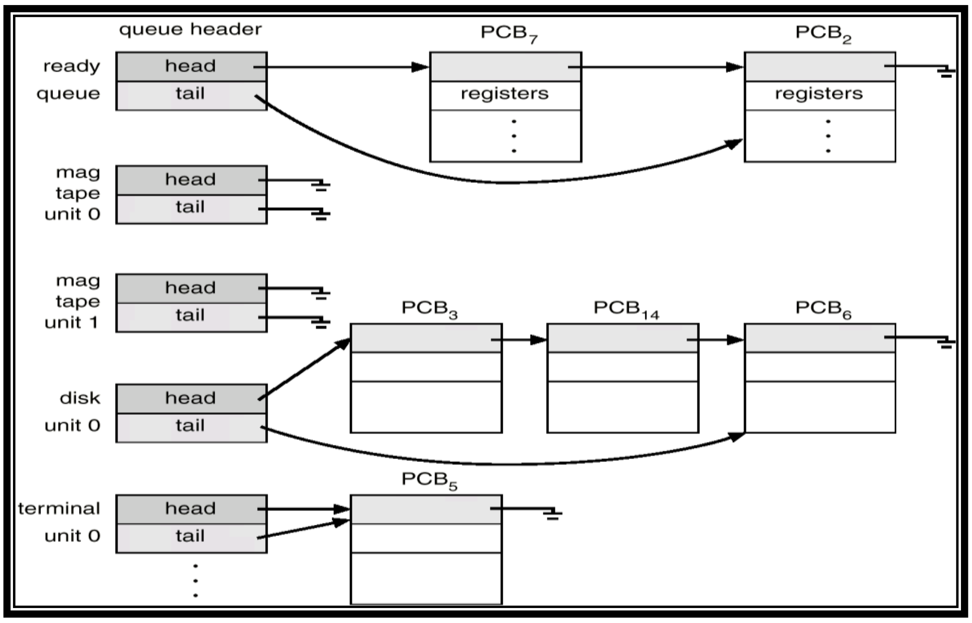


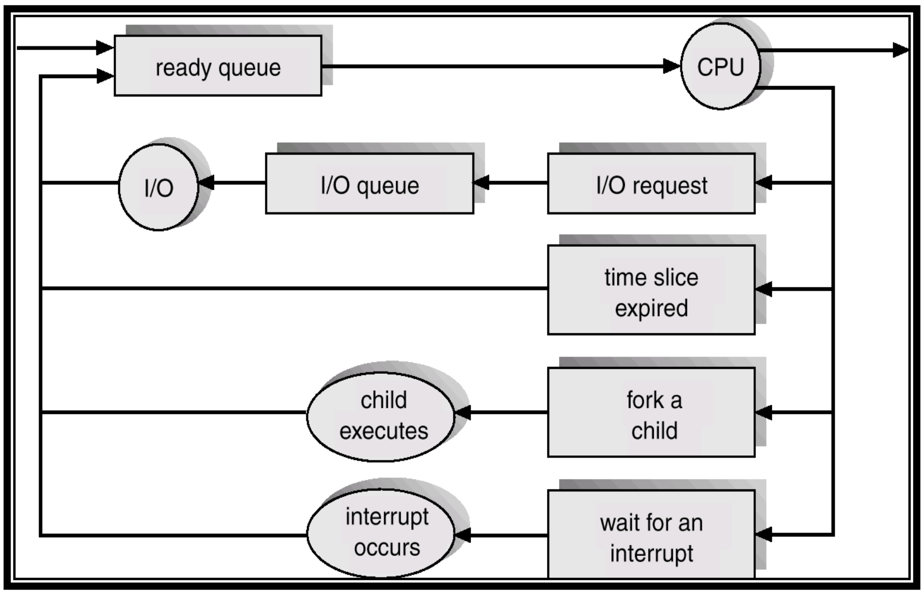
Figure 3,5 Active processes in Linux.

Each device has its own device queue (Below Figure).



The ready queue and various I/O device queues.

A common representation for a discussion of process scheduling is a **queuing diagram.** 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.



Queuing-diagram representation of process scheduling.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or is **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 subprocess and wait for the subprocess'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.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then 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.

2.2.2. Schedulers

A process migrates among the various scheduling queues throughout its lifetime. The operating system must select, for scheduling purposes, processes from these queues in some fashion. The selection process is carried out by the appropriate **scheduler.**

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device (typically a disk), where they are kept for later execution.

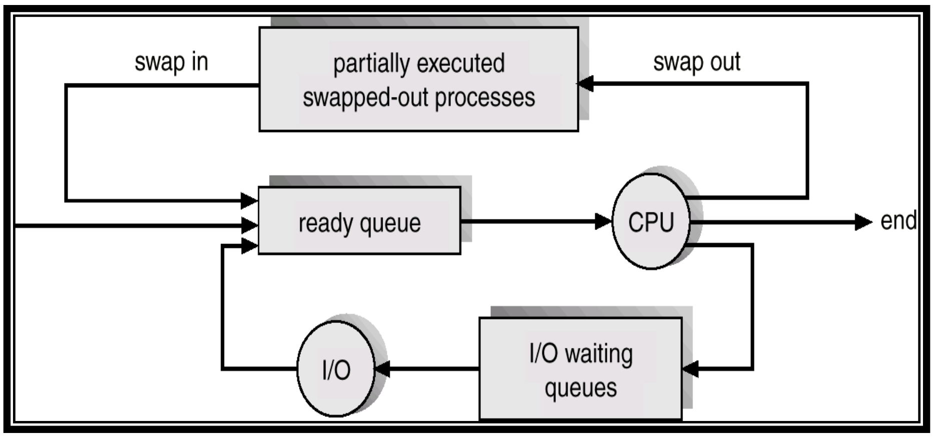
❖ The **long-term scheduler,** or **job scheduler,** selects processes from this pool and loads them into memory for execution.
❖ The **short-term scheduler, or** CPU **scheduler,** selects from among the processes that are ready to execute and allocates the CPU to one of them.
❖ Short-term scheduler is invoked very frequently (milliseconds) ⇒ (must be fast).
❖ Long-term scheduler is invoked very infrequently (seconds, minutes) ⇒ (may be slow).
❖ The long-term scheduler controls the *degree of multiprogramming.*
❖ Processes can be described as either:
   ➢ I/O-*bound process* – spends more time doing I/O than computations, many short CPU bursts.
   ➢ *CPU-bound process* – spends more time doing computations; few very long CPU bursts.

It is important that the long-term scheduler select a good **process mix** of I/O-bound and CPU-bound processes.

On some systems, the long-term scheduler may be absent or minimal. For example, time-sharing systems such as UNIX and Microsoft Windows systems often have no long-term scheduler but simply put every new process in memory for the short-term scheduler.

Some operating systems, such as time-sharing systems, may introduce an additional, intermediate level of scheduling. This **medium-term scheduler** is diagrammed in below Figure .

The key idea behind a medium-term scheduler is that sometimes it can be advantageous to remove processes from memory (and from active contention for the CPU) 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. The process is swapped out, and is later swapped in, by the medium-term scheduler.



Addition of medium-term scheduling to the queueing diagram.

✓ **Comparison b/w long term, short term and medium term**

S.no	Long term	Short term	Medium term
1.	It is job scheduler	It is CPU scheduler	It is swapping
2.	Speed is less than short term scheduler.	Speed is very fast.	Speed is in between both.
3.	It controls degree of multiprogramming.	Less control over degree of multiprogramming.	Reduce the degree of multiprogramming.
4.	Absent or minimal in time sharing system.	Minimal in time sharing system.	Time sharing system use medium term scheduling
5.	It selects processes from pool and load them into memory for execution.	It selects from among the processes that are ready to execute.	Process can be reintroduced into memory and its execution can be continued.
6.	Process state is (new to ready)	Process state is (ready to running)	-
7.	Select a good process, mix of I/O bound and CPU bound.	Select a new process for a CPU quite frequently.	-

2.2.3. Context Switch

- 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 via a context switch.
- Context of a process represented in the PCB it includes the value of the CPU registers, the process state and memory-management information.
- Context-switch time is overhead; the system does no useful work while switching.
- Time dependent on hardware support.

- ❖ Context-switch times are highly dependent on hardware support. For instance, some processors (such as the Sun UltraSPARC) provide multiple sets of registers.

2.3 Operations on Processes

The processes in most systems can execute concurrently, and they may be created and deleted dynamically. Thus, these systems must provide a mechanism for process creation and termination.

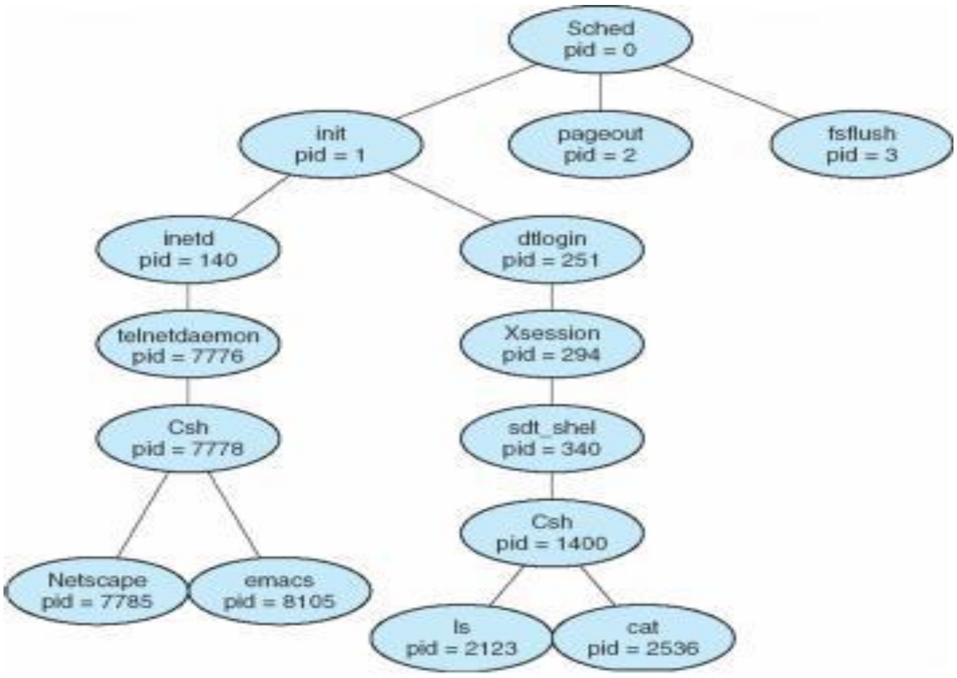
2.3.1 Process Creation

A process may create several new processes, via a create-process system call, during the course of execution. The creating process is called a **parent** process, and the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a **tree** of processes.

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.

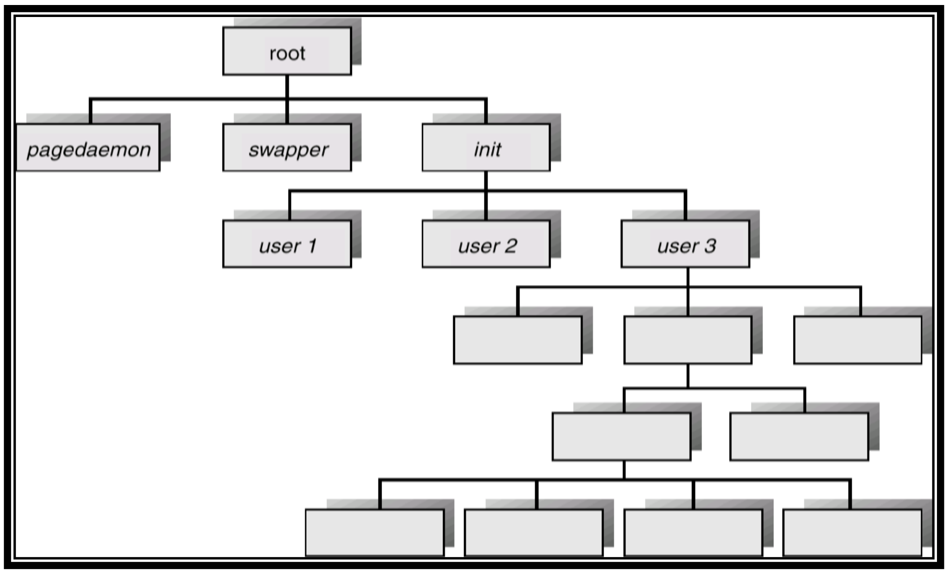
A typical process tree for the Solaris operating system, showing the name of each process and its pid. In Solaris, the process at the top of the tree is the sched process, with pid of 0. The sched process creates several children processes -including *pageout* and *fsflush*. These processes are responsible for managing memory and file systems. The sched process also creates the init process, which serves as the root parent process for all user processes.

We see two children of init — inetd and dtlogin. inetd is responsible for networking services such as t e l n e t and ftp; dtlogin is the process representing a user login screen. When a user logs in, dtlogin creates an X-windows session (Xsession), which in turns creates the sdt_shel process.



A tree of processes on a typical Solaris system

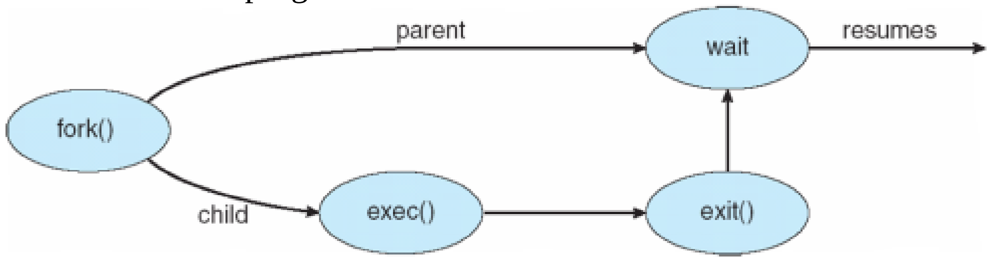
On UNIX, a listing of processes can be obtained using the ps command. For example, entering the command ps -el will list complete information for all processes currently active in the system.



Processes Tree on a UNIX System

In general, a process will need certain resources (CPU time, memory, files, I/O devices) to accomplish its task.

- ❖ When a process creates a new process, three possibilities exist in terms of to obtain its resource sharing
    - ➢ Parent and children share all resources.
    - ➢ Children share subset of parent's resources.
    - ➢ Parent and child share no resources.
- ❖ Execution
    - ➢ The parent continues to execute concurrently with its children.
    - ➢ The parent continues to execute concurrently with its children.
- ❖ There are also two possibilities in terms of the address space of the new process:
    - ➢ The child process is a duplicate of the parent process (it has the same program and data as the parent).
    - ➢ The child process has a new program loaded into it.
- ❖ UNIX examples
    - ➢ **fork** system call creates new process
    - ➢ **exec** system call used after a **fork** to replace the process' memory space with a new program.



Process creation

2.3.2 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 ()* system call).
  - ➢ All the resources of the process—including physical and virtual memory, open files and I/O buffers—are deallocated by the operating system.\
- ❖ A process can cause the termination of another process via an appropriate system call (for example, *TerminateProcess ( )* in Win32). Usually, such a system call can be invoked only by the parent of the process that is to be terminated.
- ❖ Note that a parent needs to know the identities of its children. Thus, when one process creates a new process, the identity of the newly created process is passed to the parent.
- ❖ 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.

2.4. 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. 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:

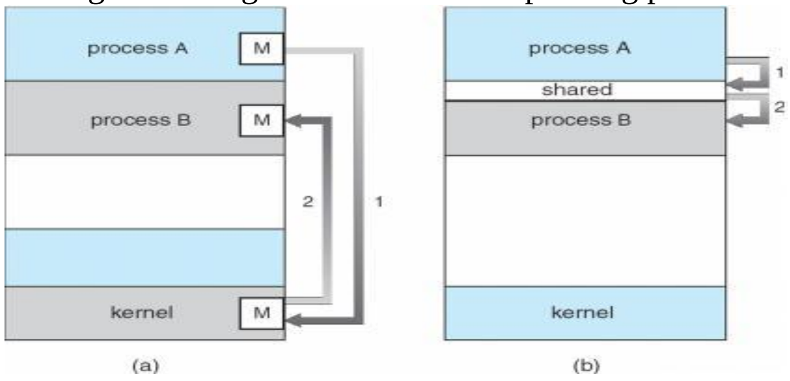
- ❖ **Information sharing.** Since several users may be interested in the same piece of information (for instance, a shared file), we must provide an environment to allow concurrent access to such information.
- ❖ **Computation speedup.** If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others. Notice that such a speedup can be achieved only if the computer has multiple processing elements (such as CPUs or I/O channels).
- ❖ **Modularity.** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- ❖ **Convenience.** Even an individual user may work on many tasks at the same time. For instance, a user may be editing, printing, 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:

(1) Shared memory 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.

(2) **Message passing-** In the message passing model, communication takes place by means of messages exchanged between the cooperating processes.



Communications models, (a) Message passing, (b) Shared memory

2.4.1 Shared-Memory Systems

Interprocess communication using shared memory requires communicating processes to establish a region of shared memory. Typically, 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.

Normally, 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 form of the data and the location are determined by these processes and are not under the operating system's control. The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

To illustrate the concept of cooperating processes, let's consider the producer-consumer problem. A **producer** process produces information that is consumed by a **consumer** process.

- ❖ For example, a compiler may produce assembly code, which is consumed by an assembler. The assembler, in turn, may produce object modules, which are consumed by the loader.

We generally think of a server as a producer and a client as a consumer. For example, a web server produces (that is, provides) HTML files and images, which are consumed (that is, read) by the client web browser requesting the resource.

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.

Let's look more closely at how the bounded buffer can be used to enable processes to share memory. The following variables reside in a region of memory shared by the producer and consumer processes:

```
#define BUFFER_SIZE 10
typedef struct {
                    ……..
                    }item;

item buffer [BUFFER_SIZE] ;
int in = 0 ,-
int out = 0 ;
```

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 producer process has a local variable *nextProduced* in which the new item to be produced is stored. The consumer process has a local variable *nextConsumed* in which the item to be consumed is stored.

This scheme allows at most BUFFER_SIZE - l items in the buffer at the same time.

The code for the producer and consumer processes is shown in Figures respectively.

<u>The producer process</u>

```
item nextProduced;
while (true) {
/* produce an item in nextProduced */
while (((in + 1) % BUFFER-SIZE) == out)
; /* do nothing */
buffer [in] = nextProduced;
in = (in + 1) % BUFFER_SIZE;
}
```

The consumer process.

```
item nextConsumed;
while (true) {
while (in == out)
; //do nothing
nextConsumed = buffer [out];
out = (out + 1) % BUFFEFLSIZE;
/* consume the item in nextConsumed */
}
```

2.4.2 Message-Passing Systems

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network.

❖ For example, a **chat** program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations: *send*(message) and *receive*(message). Messages sent by a process can be of either fixed or variable size. If only fixed-sized messages can be sent, the system-level implementation is straight forward. This makes the task of programming more difficult. Conversely, variable-sized messages require a more complex system-level implementation, but the programming task becomes simpler.

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.

- Direct or indirect communication
- Synchronous or asynchronous communication
- Automatic or explicit buffering

❖ Design characteristics of message systems for IPC:

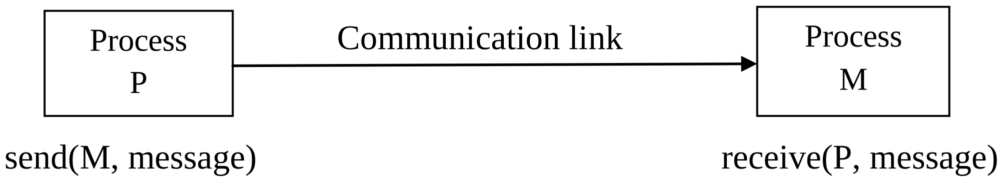
- Naming (Addressing)
- Synchronization
- Buffering
- Message Format

2.4.2.1 Naming

Processes that want to communicate must have a way to refer to each other. They can use either direct or indirect communication.

Under 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.



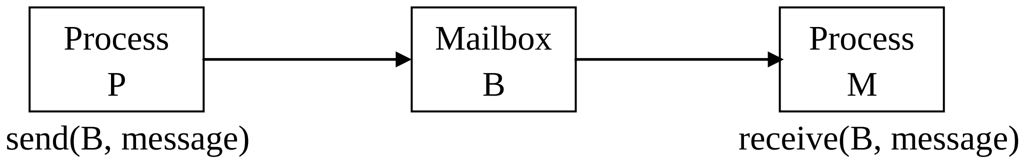
Direct Communication

A communication link in this scheme has the following properties:
• A link is established automatically between every pair of 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.

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. For example, POSIX message queues use an integer value to identify a mailbox. In this scheme, a process can communicate with some other process via a number of different mailboxes.

Two processes can communicate only if the processes have a shared mailbox, however. 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.



Indirect communication

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, there may be a number of different links, with each link corresponding to one mailbox.

In contrast, a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:
• Create a new mailbox.
• Send and receive messages through the mailbox.
• Delete a mailbox.

2.4.2.2 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 blocking or **nonblocking-** also known as s**ynchronous** 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.

2.4.2.3 Buffering

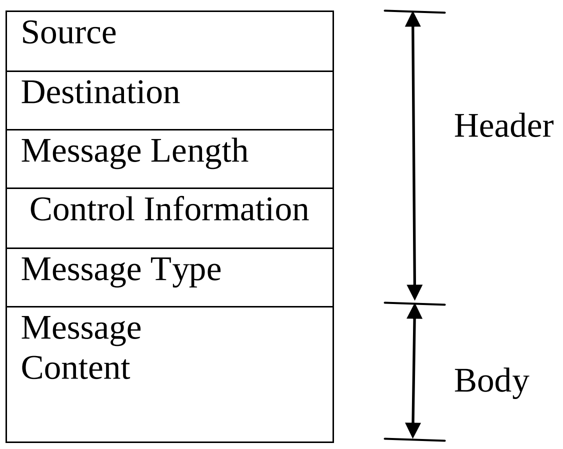
Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

• **Zero capacity.** The queue has a maximum length of zero; thus, the link cannot have any messages waiting in it. In this case, the sender must block until the recipient receives the message.

- **Bounded capacity.** The queue has finite length *n;* thus, at most *n* messages can reside in it. If the queue is not full when a new message is sent, the message is placed in the queue and the sender can continue execution without waiting. The links capacity is finite, however. If the link is full, the sender must block until space is available in the queue.

- **Unbounded capacity.** The queues length is potentially infinite; thus, any number of messages can wait in it. The sender never blocks.

2.4.2.4 Message Format



Typical Message Format

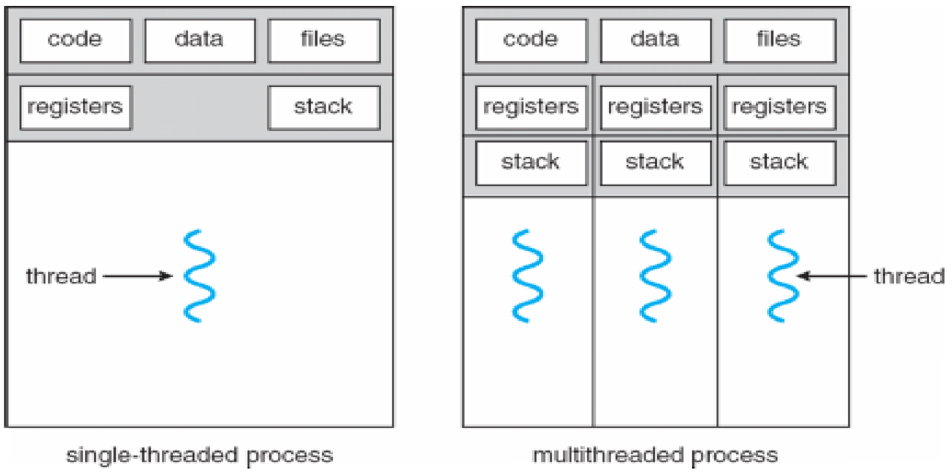
The above message format for operating system that support variable length messages. Os supports both variable length and fixed length messages.

Message format is divided into two parts: Header and body. Header contains source address, destination address, message length, and message type and control information. Message content is the body part.

2.5 Multi Thread Programming Model

Thread

- ❖ A thread is a basic unit of CPU utilization; with its own thread ID, a program **counter, a register set, and a stack.** A thread is sometimes called as **light weight process**.
- ❖ Thread represents a software approach to improving performance of O.S by reducing the over head of process switching.
- ❖ 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. Each thread represents a separate flow of control.
- ❖ Finally, many operating system kernels are now multithreaded; several threads operate in the kernel, and each thread performs a specific task, such as managing devices or interrupt handling. For example, Solaris creates a set of threads in the kernel specifically for interrupt handling; Linux uses a kernel thread for managing the amount of free memory in the system
- ❖ Below Figure illustrates the difference between a traditional **single-threaded** process and a **multithreaded** process.



✓ <u>Differences b/w *Process* and *Thread*</u>

S.No	Process	Thread
1.	Process is called heavy weight process	Thread is called light weight process
2.	Process switching needs interface with operating system.	Thread switching does not need to call a os and cause an interrupt to the kernel.
3.	In multiple process implementation each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4.	If one server process is blocked no other server process can execute until the first process unblocked.	While one server thread is blocked and waiting, second thread in the same task could run.
5.	Multiple redundant process uses more resources than multiple threaded.	Multiple threaded process uses fewer resources than multiple redundant process.
6.	In multiple process each process operates independently of the others.	One thread can read, write or even completely wipe out another threads stack.

<u>Benefits</u>

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

1. Responsiveness.
2. Resource sharing.
3. Economy
4. Utilization of multiprocessor architectures

<u>2.5.1 Multithreading Models</u>

Support for threads may be provided either at

1. User level – for user threads
2. Kernel level – for kernel threads

1. User level - User threads are supported above the kernel and are managed without kernel support. The thread library contains code for creating and destroying threads, for passing messages, and data between threads, for scheduling thread execution and for saving and restoring thread contexts.

- ❖ The application begins with a single thread and begins running in that thread.
- ❖ User level threads are generally fast to create and manage.

Advantages of user level thread over kernel level thread.

1. Thread switching does not require kernel mode privileges.
2. User level thread can run on any OS.
3. Scheduling can be application specific.
4. User level threads are fast to create and manage.

Disadvantages of user level threads:

1. In a typical OS , most system calls are blocking.
2. Multithreaded application can not take advantages of multiprocessing.

2. Kernel level

kernel level threads are supported and managed directly by the operating system. Virtually all contemporary operating systems—including Windows XP, Linux, Mac OS X, Solaris, and Tru64 UNIX (formerly Digital UNIX)—support kernel threads.

- ❖ There is no thread management code in the application area.
- ❖ Any application can be programmed to be multithreaded.
- ❖ All of the threads with in an application are supported within a single process.
- ❖ The kernel performs thread creation, scheduling and management in kernel space.
- ❖ Kernel threads are generally slower to create and manage than the user threads.

Advantages of kernel level threads

- ❖ Kernel can simultaneously schedule multiple threads from the same process on multiple processes.
- ❖ If one thread in a process is blocked, the kernel can schedule another thread of the same process.
- ❖ Kernel routines themselves can be multithreaded.

Disadvantages of kernel threads

- ❖ Thread minimizes context switch time.
- ❖ Use of threads provides concurrency within a process.
- ❖ Efficient communication.
- ❖ Economy – it is more economical to create and context switch threads.
- ❖ Utilization of multiprocessor architecture – The benefits of multithreading can be greatly increased in a multiprocessor architecture.

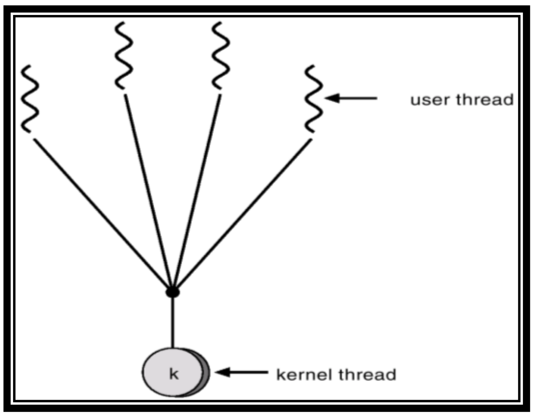
✓ Differences b/w user level and kernel level

S.no	User level threads	Kernel level threads
1	User level threads are faster to create and manage.	Kernel level threads are slower to create and manage.
2.	Implemented by a thread library at the user level.	Os support directly to kernel threads.
3.	Ult can run on any OS.	Klt are specific to the OS.
4.	Support provided at the user level called user level thread.	Support may be provided by kernel is called kernel level threads.
5.	Multithread application can not take advantage of multiprocessing.	Kenel routines themselves can be multithreaded.

- ❖ Some OS provide a combined user level thread and kernel level thread facility. **Solaris** is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process.
- ❖ There must exist a relationship between user threads and kernel threads. There are three common ways of establishing this relationship.
  1. Many to one relationship
  2. One to one relationship
  3. Many to many relationship

1. Many to one relationship

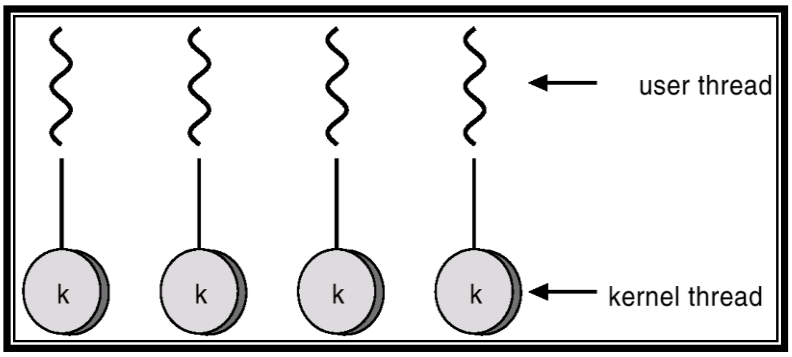
- ❖ The many-to-one model maps many user-level threads to one kernel thread.
- ❖ Thread management is done by the thread library in user space.
- ❖ It is efficient; but the entire process will block if a thread makes a blocking system call.
- ❖ Only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- ❖ If the user level thread libraries are implemented in the OS, that system does not support kernel threads use the many to one relationship.



Many to one model

2. <u>One to one relationship</u>

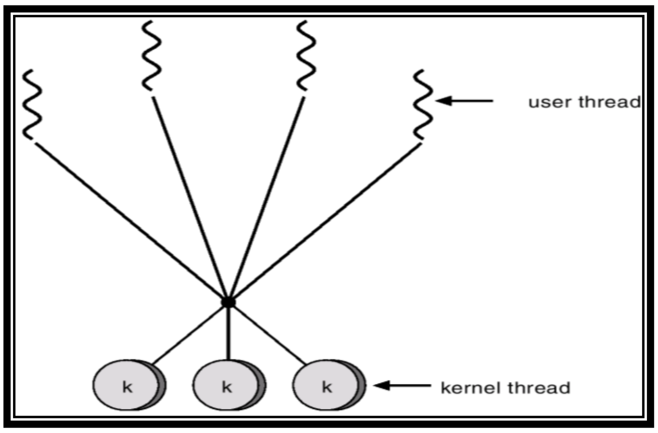
- ❖ The one-to-one model maps each user thread to a kernel thread.
- ❖ It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call; it also allows multiple threads to run in parallel on multiprocessors.
- ❖ The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- ❖ performance of an application, most implementations of this model restrict the number of threads supported by the system.
- ❖ Eg: Windows 95, 98, NT, 2000, and XP- implement the one-to-one model.



<u>One-to-one model</u>

3. <u>Many to many relationship</u>

- ❖ The many-to-one model many user-level threads to the kernel level thread of less or equal numbers.
- ❖ The number of kernel threads may be specific to either a particular application or a particular machine.



<u>Many-to-many model</u>

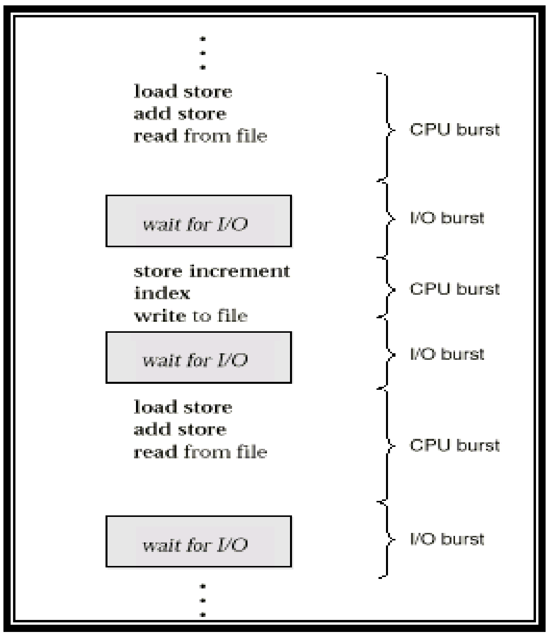
In many-to-one model allows the developer to create as many user threads as she wishes, true concurrency is not gained because the kernel can schedule only one thread at a time.

2.7 Process Scheduling criteria and algorithms and their evaluation

CPU scheduling is the basis of multiprogrammed operating systems. By switching the CPU among processes, the operating system can make the computer more productive.

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.

CPU-I/O Burst Cycle – 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.



Alternating sequence of CPU and I/O bursts.

2.7.1 CPU Scheduler

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.

Note that the ready queue is not necessarily a first-in, first-out (FIFO) queue. As we shall see when we consider the various scheduling algorithms, a ready queue can be implemented as a FIFO queue, a priority queue, a tree, or simply an unordered linked list.

<u>*Preemptive Scheduling*</u>

❖ CPU scheduling decisions may take place when a process:

1. Switches from running to waiting state.
2. Switches from running to ready state.
3. Switches from waiting to ready.
4. Terminates.

For situations 1 and 4, there is no choice in terms of scheduling. A new process (if one exists in the ready queue) must be selected for execution. There is a choice, however, for situations 2 and 3.

When scheduling takes place only under circumstances 1 and 4, we say that the scheduling scheme is **nonpreemptive** or **cooperative;** otherwise, it is **preemptive.** 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.

<u>*Dispatcher*</u>

Another component involved in the CPU-scheduling function is the **dispatcher.** The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler. This function involves the following:

• Switching context
• Switching to user mode
• Jumping to the proper location in the user program to restart that program

The dispatcher should be as fast as possible, since it is invoked during every process switch. The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency.**

<u>2.7.2 Scheduling Criteria</u>

Different CPU scheduling algorithms have different properties. Many criteria have been suggested for comparing CPU scheduling algorithms.

The criteria include the following:

<u>*CPU utilization.*</u> 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 used system).

<u>*Throughput.*</u> 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.*

<u>*Turnaround time:*</u>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.

<u>*Waiting time:*</u> 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.

*Response time.*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. The turnaround time is generally limited by the speed of the output device.

2.7.3 Scheduling 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.

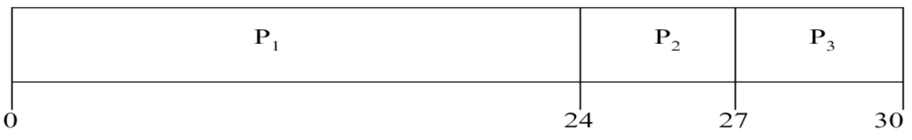
1. First-Come, First-Served Scheduling
2. Shortest-Job-First Scheduling
3. Priority Scheduling
4. Round-Robin Scheduling
5. Multilevel Queue Scheduling
6. Multilevel Feedback-Queue Scheduling

2.7.3.1 First-Come, First-Served Scheduling

- **First-come, first-served (FCFS) scheduling algorithm is** the simplest CPU-scheduling algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first.
- The implementation of the FCFS policy is 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 running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand.
- The average waiting time under the FCFS policy, however, is often quite long.
- Example : Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

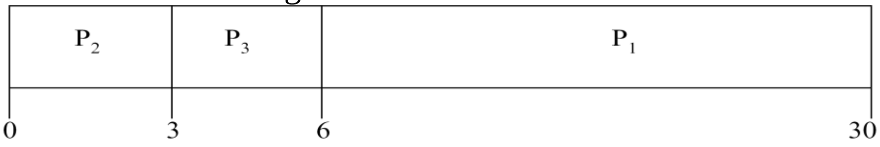
Process	Burst Time
P1	*24*
P2	*3*
P3	*3*

❖ 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:**



- Waiting time for *P1*= 0; *P2*= 24; *P3* = 27
- Average waiting time: (0 + 24 + 27)/3 = 17

❖ Suppose that the processes arrive in the order P_2 , P_3 , P_1 , we get the result shown in the following **Gantt chart:**



- Waiting time for $P_1 = 6$; $P_2 = 0$; $P_3 = 3$
- Average waiting time: (6 + 0 + 3)/3 = 3

The 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.

2.7.3.2 Shortest-Job-First Scheduling

This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

Note that a more appropriate term for this scheduling method would be the *shortest-next-CPU-burst algorithm*, because scheduling depends on the length of the next CPU burst of a process, rather than its total length.

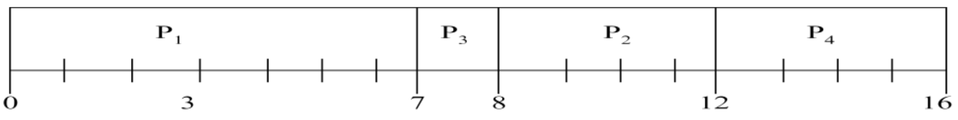
➢ Two schemes:
  - nonpreemptive – once CPU given to the process it cannot be preempted until completes its CPU burst.
  - preemptive – if a new process arrives with CPU burst length less than remaining time of current executing process, preempt. This scheme is know as the Shortest-Remaining-Time-First (SRTF).

➢ SJF is optimal – gives minimum average waiting time for a given set of rocesses.

❖ As an example of ***Non-Preemptive*** scheduling, consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

➢ we would schedule these processes according to the following Gantt chart:

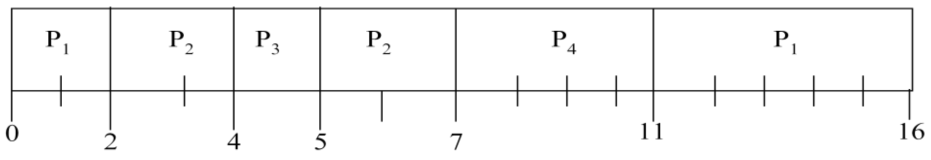


- ➢ Average waiting time = (0 + 6 + 3 + 7)/4 = 4

❖ Example of *Preemptive SJF.*

Process	Arrival Time	Burst Time
P_1	0.0	7
P_2	2.0	4
P_3	4.0	1
P_4	5.0	4

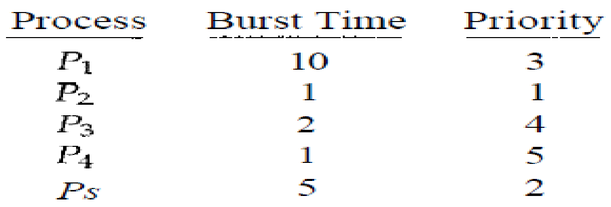
- ➢ we would schedule these processes according to the following Gantt chart:



- ➢ Average waiting time = (9 + 1 + 0 +2)/4 - 3
- ➢ SJF algorithm is optimal one. It gives the minimum average waiting time for a given set of processes. SJF can not be implemented at the level of short term CPU scheduling. There is no way to know the length of the next CPU burst.

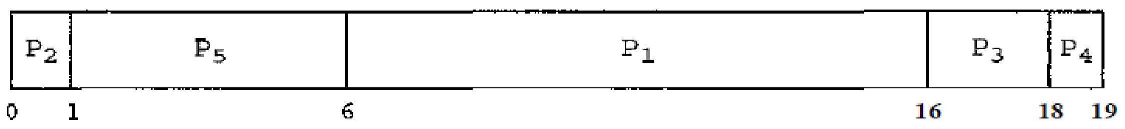
2.7.3.3 Priority Scheduling

- ➢ 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. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. we assume that low numbers represent high priority.
- ➢ As an example, consider the following set of processes, assumed to have arrived at time 0, in the order Pi, P2, • • -, *P5,* with the length of the CPU burst given in milliseconds:



Process	Burst Time	Priority
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

- ➢ Using priority scheduling, we would schedule these processes according to the following Gantt chart:



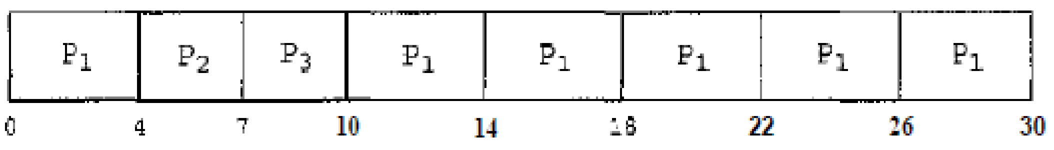
- Priority scheduling can be either preemptive or nonpreemptive. When a process arrives at the ready queue, its priority is compared with the priority of the currently running process.
- A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher 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 is a technique of gradually increasing the priority of processes that wait in the system for a long time.

2.7.3.4 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 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. 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, we keep the ready queue as a FIFO queue of processes. 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.
- The average waiting time under the RR policy is often long.
- ❖ Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	24
P_2	3
P_3	3

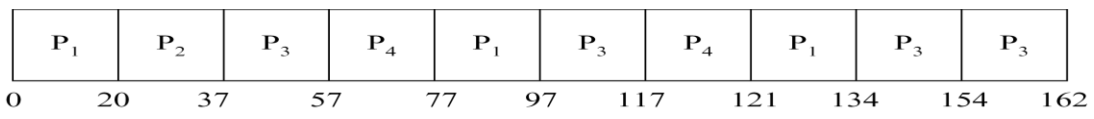
If we use a time quantum of 4 milliseconds, then process Pi gets the first 4 milliseconds. Since it requires another 20 milliseconds, it is preempted after the first time quantum, and the CPU is given to the next process in the queue, process *P2*.



❖ Example of RR with Time Quantum = 20

Process	Burst Time
P_1	53
P_2	17
P_3	68
P_4	24

The Gantt chart is:



➢ Typically, higher average turnaround than SJF, but better *response*.

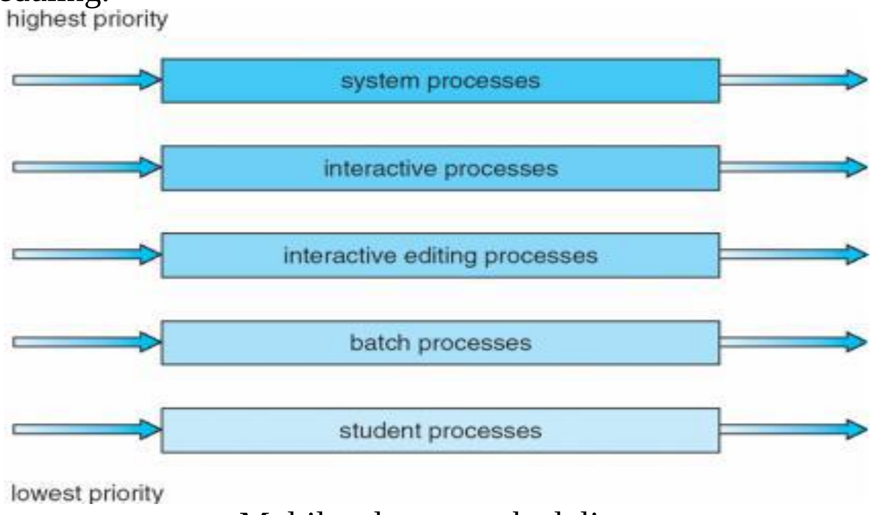
2.7.3.5 Multilevel Queue Scheduling.

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 (Figure 5.6). 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. In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.



Multilevel queue scheduling

Another possibility is to time-slice among the queues. Here, each queue gets a certain portion of the CPU time, which it can then schedule among its various processes.

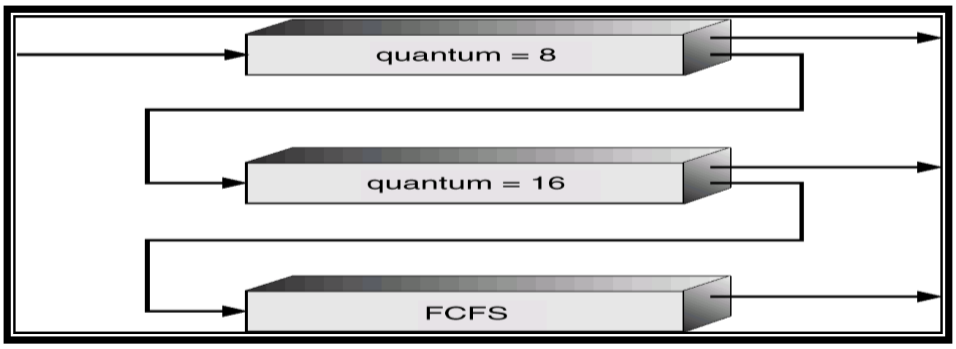
For instance, the foreground queue can be given 80 percent of the CPU time for RR scheduling among its processes, whereas the background queue receives 20 percent of the CPU to give to its processes on an FCFS basis.

2.7.3.6 Multilevel Feedback-Queue Scheduling

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.

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 (below 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 only be executed 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.



Multilevel feedback queues.

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 higherpriority queue
• The method used to determine when to demote a process to a lowerpriority queue
• The method used to determine which queue a process will enter when that process needs service

UNIT-II-Revised Questions

NOV-2012

2. (a) Explain in detail round robin scheduling algorithm .With an example show how a smaller time quantum increases context switches.
(b) Describe the differences between short-term, medium-term and long-term scheduling.

2. (a) Define process and explain with a neat diagram about process states and process control block. Discuss briefly system calls.
(b) Discuss in detail schedulers. Describe the difference between different types of scheduling.

2. (a) The benefits of multithreaded programming can be broken down into four major Categories. Explain briefly each of these categories.
(b) What are the scheduling algorithms? Discuss briefly about the priority scheduling algorithm and the round-robin scheduling algorithm.

2. (a) What are multi-threaded models? Explain types of multi-threaded model in detail.
(b) Explain briefly the procedure to predict the next CPU burst in the shortest-job first algorithm.

MAY-2012

2. a) What is multiprogramming? Explain.
b) What are the responsibilities of OS in file management?
c) What is critical section problem? Write a solution to the Bounded-buffer producer - consumer problem using semaphores. [4+4+8]

2. a) Explain the multithreaded models for user and kernel threads.
b) Explain the Readers-Writers problem and give the solution for synchronization using semaphores. [8+8]

2. a) What is meant by CPU scheduling? Explain the criteria for comparing CPU scheduling algorithms.
b) Explain the common approaches for authenticating a user identity. [8+8]

2. a) Discuss about revocation of access rights in detail.
b) Describe the differences among short-term, medium-term, and long-term scheduling. [8+8]

MAY-2011

2. a) Consider the following set of processes, with the length of the CPU-burst time given in milliseconds:

Process	Burst Time
P1	5
P2	1
P3	3
P4	1
P5	7

The processes are assumed to have arrived in the order P1,P2, P3, P4, P5 all at time 0.
i) Draw four Gnatt charts illustrating the execution of these processes using SJF, and RR(quantum = 1) scheduling.
ii)What is the waiting time of each process for each of the scheduling algorithms in part (i)?
b) Explain about the interprocess communication [8+8]

2. List and explain the different CPU scheduling algorithms with example [16]

2. (a) Describe the differences among short-term, medium-term, and long-term scheduling
(b) What is thread? What are the benefits of multithreaded programming? [8+8]

2. (a) What is a process? Explain the process states and PCB.
(b) What is a thread? What are the benefits of multi threaded programming? [8+8]

MAY-2010

2. (a) Explain Priority scheduling with the help of an example. What are Starvation and aging?
(b) What are the situations in which Round robin scheduling is preferred and situations in which it results in overhead. [12+4]

2. (a) What are the different types of Scheduling Queues? What are the different types of schedulers?
(b) What is the difference between Preemptive and non-preemptive scheduling?
(c) What are the criteria for evaluating scheduling algorithms? Evaluate any one scheduling algorithm using those criteria. [5+4+7]

2. (a) What is the difference between process and program? How they are related.
(b) How and what information about a process is represented using PCB. [8+8]

2. (a) What is system call? What is the difference between system call and system program. What are the different types of system calls? [10+6 M]
(b) With the help of a diagram, explain the different states of a process.