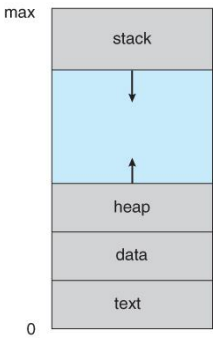
UNIT-2
Process Management

2.1 WHAT IS A PROCESS?

Process is a program in execution. A process is an instance of a program running in a computer. It is close in meaning to task , a term used in some operating systems. In UNIX and some other operating systems, a process is started when a program is initiated (either by a user entering a shell command or by another program).

- On execution, this program may read in some data and output some data.
- Note that when a program is written and a file is prepared, it is still a script (passive entity). It has no dynamics of its own i.e, it cannot cause any input processing or output to happen.
- Once we compile, and later when we run this program, the intended operations take place (active entity). In other words, a program is a text script with no dynamic Behavior.
- When a program is in execution, the script is acted upon. It can result in engaging a processor for some processing and it can also engage in I/O operations.
- It is for this reason a process is differentiated from program.
- While the program is a text script (often called executable file), a program in execution is a process.



2.1.1 THE PROCESS

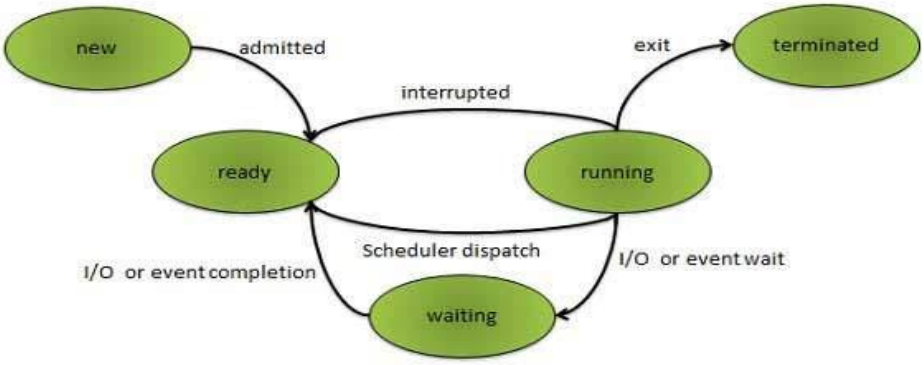
- Process memory is divided into four sections as shown in Figure 3.1 below:
  - The text section comprises the compiled program code, read in from non-volatile storage when the program is launched.
  - The data section stores global and static variables, allocated and initialized prior to executing main.
  - The heap is used for dynamic memory allocation, and is managed via calls to new, delete, malloc, free, etc.
  - The stack is used for local variables. Space on the stack is reserved for local variables when they are declared and the space is freed up when the variables go out of scope. Note that the stack is also used for function return values, and the exact mechanisms of stack management may be language specific.
  - Note that the stack and the heap start at opposite ends of the process's free space and grow towards each other. If they should ever meet, then either a stack overflow error will occur, or else a call to new or malloc will fail due to insufficient memory available.

2.1.2 Process state:

As a process executes, it changes state. The state of a process is defined as the current activity of the process.

Process can have one of the following five states at a time.

S.N.	State & Description
1	**New** The process is being created.
2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run.
3	**Running** Process instructions are being executed (i.e. The process that is currently being executed).
4	**Waiting** The process is waiting for some event to occur (such as the completion of an I/O operation).
5	**Terminated** The process has finished execution.



2.1.3 Process Control Block, PCB:

Each process is represented in the operating system by a process control block (PCB) also called a task control block. PCB is the data structure used by the operating system. Operating system groups all information that needs about particular process.

PCB contains many pieces of information associated with a specific process which are described below.

S.N.	Information & Description
1	**Pointer** Pointer points to another process control block. Pointer is used for maintaining the scheduling list.
2	**Process State** Process state may be new, ready, running, waiting and so on.
3	**Program Counter** Program Counter indicates the address of the next instruction to be executed for this process.
4	**CPU registers** CPU registers include general purpose register, stack pointers, index registers and accumulators etc. number of register and type of register totally depends upon the computer architecture.

5	**Memory management information**: This information may include the value of base and limit registers, the page tables, or the segment tables depending on the memory system used by the operating system. This information is useful for deallocating the memory when the process terminates.
6	**Accounting information** This information includes the amount of CPU and real time used, time limits, job or process numbers, account numbers etc.

Process control block includes CPU scheduling, I/O resource management, file management information etc.. The PCB serves as the repository for any information which can vary from process to process. Loader/linker sets flags and registers when a process is created. If that process gets suspended, the contents of the registers are saved on a stack and the pointer to the particular stack frame is stored in the PCB. By this technique, the hardware state can be restored so that the process can be scheduled to run again.

2.2 PROCESS SCHEDULING:

Definition

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

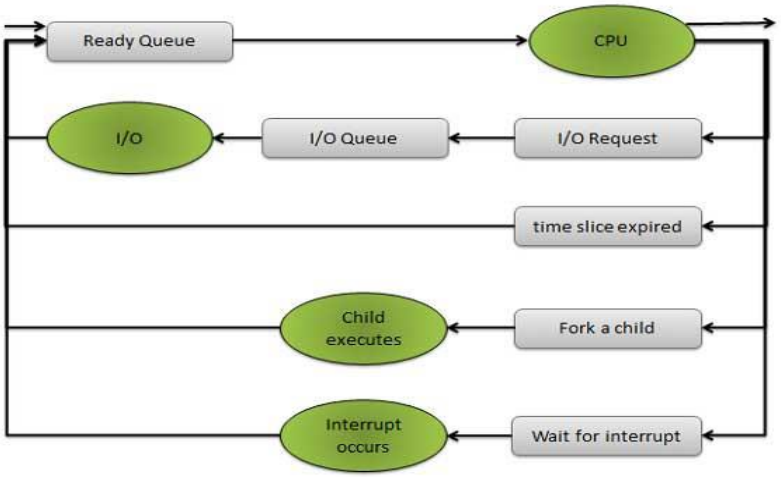
Process scheduling is an essential part of a Multiprogramming operating system. Such operating systems allow more than one process to be loaded into the executable memory at a time and loaded process shares the CPU using time multiplexing.

2.2.1 Scheduling Queues

Scheduling queues refers to queues of processes or devices. When the process enters into the system, then this process is put into a **job queue**. This queue consists of all processes in the system. The operating system also maintains other queues such as **device queue**. Device queue is a queue for which multiple processes are waiting for a particular I/O device. Each device has its own device queue.

This figure shows the queuing diagram of process scheduling.

- Queue is represented by rectangular box.
- The circles represent the resources that serve the queues.
- The arrows indicate the process flow in the system.



Queues are of three types

- Job queue
- Ready queue
- Device queue

A newly arrived process is put in the ready queue. Processes waits in ready queue for allocating the CPU. Once the CPU is assigned to a process, then that process will execute. While executing the process, any one of the following events can occur.

- The process could issue an I/O request and then it would be placed in an I/O queue.
- The process could create new sub process and will wait for its termination.
- The process could be removed forcibly from the CPU, as a result of interrupt and put back in the ready queue.

2.2.2 Schedulers

Schedulers are special system software's which handles process scheduling in various ways. Their main task is to select the jobs to be submitted into the system and to decide which process to run.

Schedulers are of three types

- Long Term Scheduler
- Short Term Scheduler
- Medium Term Scheduler

Long Term Scheduler

It is also called job scheduler. Long term scheduler determines which programs are admitted to the system for processing. Job scheduler selects processes from the queue and loads them into **memory** for execution. Process loads into the memory for CPU scheduling. The primary objective of the job scheduler is to provide **a balanced mix of jobs**, such as I/O bound and processor bound. It also controls the degree of multiprogramming. If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

When process changes the state from new to ready, then there is use of long term scheduler.

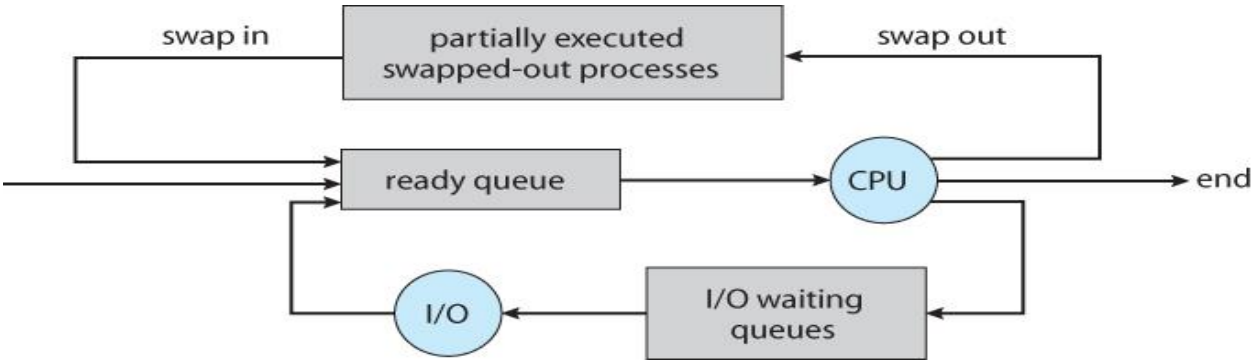
Short Term Scheduler

It is also called CPU scheduler. Main objective is **increasing system performance** in accordance with the chosen set of criteria. It is the change of ready state to running state of the process. CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them.

Short term scheduler also known as dispatcher, execute most frequently and makes the fine grained decision of which process to execute next. Short term scheduler is faster than long term scheduler.

Medium Term Scheduler

Medium term scheduling is part of the **swapping**. It removes the processes from the memory. It reduces the degree of multiprogramming. The medium term scheduler is in-charge of handling the swapped out-processes.



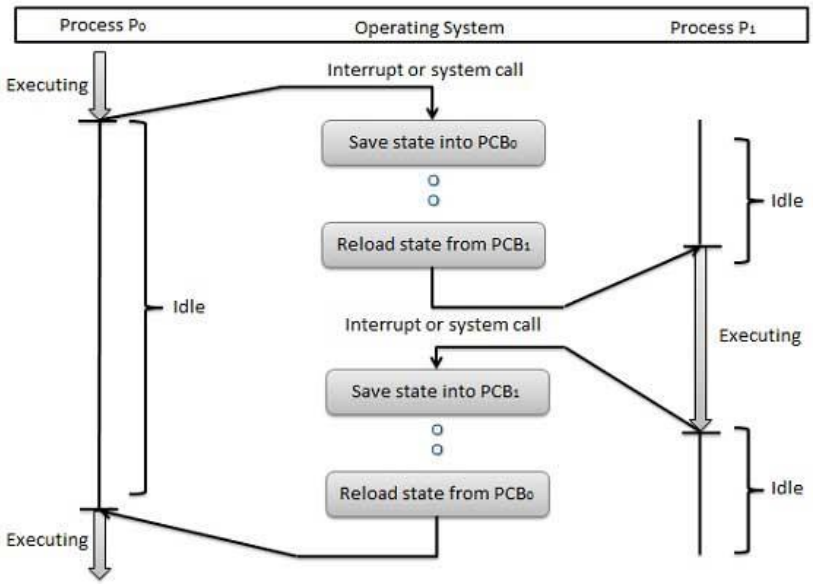
Running process may become **suspended** if it makes an I/O request. Suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the **secondary storage**. This process is called **swapping**, and the process is said to be swapped out or rolled out. Swapping may be necessary to improve the process mix.

2.2.3 Context Switch

A context switch is the mechanism **to store and restore** the **state** or **context** of a CPU in **Process Control block** so that a process execution can be resumed from the same point at a later time. Using this technique a context switcher enables multiple processes to share a single CPU. Context switching is an essential part of a multitasking operating system features.

When the scheduler switches the CPU from executing one process to execute another, the context switcher saves the content of all processor registers for the process being removed from the CPU, in its process descriptor. The context of a process is represented in the process control block of a process.

Context switch time is pure overhead. Context switching can significantly affect performance as modern computers have a lot of general and status registers to be saved. Content switching times are highly dependent on hardware support. Context switch requires (n + m) bxK time units to save the state of the processor with n general registers, assuming b are the store operations are required to save n and m registers of two process control blocks and each store instruction requires K time units.



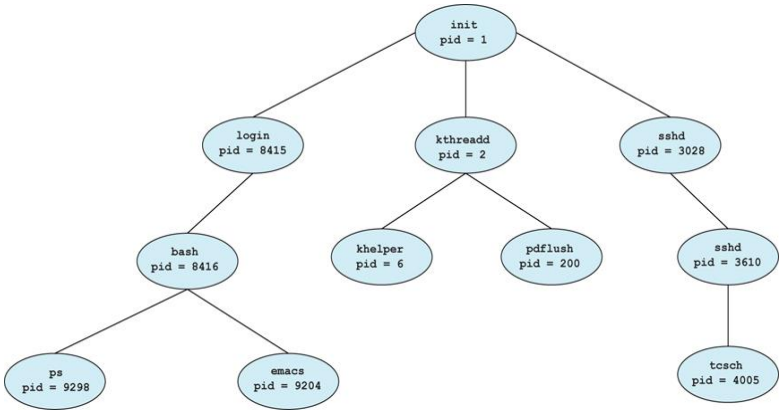
Some hardware systems employ two or more sets of processor registers to reduce the amount of context switching time. When the process is switched, the following information is stored.

- Program Counter
- Scheduling Information
- Base and limit register value
- Currently used register
- Changed State
- I/O State
- Accounting

2.3 Operations on Processes

2.3.1 Process Creation

- Processes may create other processes through appropriate system calls, such as **fork** or **spawn**. The process which does the creating is termed the **parent** of the other process, which is termed its **child**.
- Each process is given an integer identifier, termed its **process identifier**, or PID. The parent PID (PPID) is also stored for each process.
- On typical UNIX systems the process scheduler is termed **sched**, and is given PID 0. The first thing it does at system startup time is to launch **init**, which gives that process PID 1. Init then launches all system daemons and user logins, and becomes the ultimate parent of all other processes. Figure 3.9 shows a typical process tree for a Linux system, and other systems will have similar though not identical trees:



- Depending on system implementation, a child process may receive some amount of shared resources with its parent. Child processes may or may not be limited to a subset of the resources originally allocated to the parent, preventing runaway children from consuming all of a certain system resource.

There are two possibilities for the parent process after creating the child:

1. Wait for the child process to terminate before proceeding. The parent makes a wait( ) system call, for either a specific child or for any child, which causes the parent process to block until the wait( ) returns. UNIX shells normally wait for their children to complete before issuing a new prompt.

2. The parent waits until some or all its children have terminated

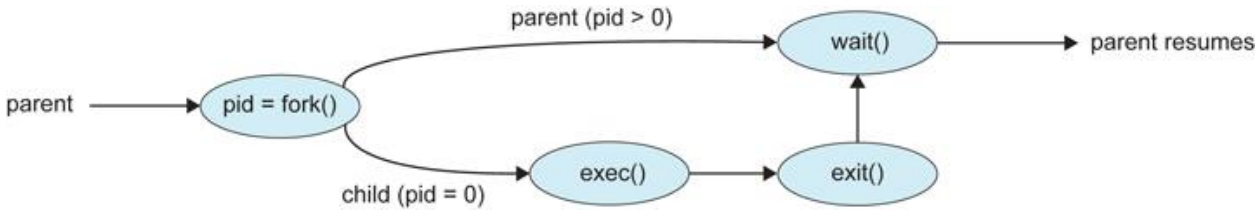
Run concurrently with the child, continuing to process without waiting. This is the operation seen when a UNIX shell runs a process as a background task. It is also possible for the parent to run for a while, and then wait for the child later, which might occur in a sort of a parallel processing operation. (E.g. the parent may fork off a number of children without waiting for any of them, then do a little work of its own, and then wait for the children.)

Two possibilities for the address space of the child relative to the parent:

- The child may be an exact duplicate of the parent, sharing the same program and data segments in memory. Each will have their own PCB, including program counter, registers, and PID. This is the behavior of the fork system call in UNIX.
- The child process may have a new program loaded into its address space, with all new code and data segments. This is the behavior of the spawn system calls in Windows. UNIX systems implement this as a second step, using the exec system call.

Figures below shows the fork and exec process on a UNIX system. Note that the fork system call returns the PID of the processes child to each process - It returns a zero to the child process and a non-zero child PID to the parent, so the return value indicates which process is which.

Process IDs can be looked up any time for the current process or its direct parent using the getpid( ) and getppid( ) system calls respectively.



Process creation using the fork ( ) system call

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

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

    if (pid < 0) {/* error occurred */
      fprintf(stderr, "Fork Failed");
      exit(-1);
    }
    else if (pid == 0) {/* child process */
      execlp("/bin/ls","ls",NULL);
    }
    else {/* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete");
      exit(0);
    }
}
```

Figure 3.10 C program forking a separate process.

2.3.2 Process Termination

Processes may request their own termination by making the exit ( ) system call, typically returning an int. This int is passed along to the parent if it is doing a wait ( ), and is typically zero on successful completion and some non-zero code in the event of problems.

Child code:

```
int exitCode;

exit( exitCode );  // return exitCode; has the same effect when executed from main( )
```

parent code:

```
pid_t pid;

int status

pid = wait( &status );

// pid indicates which child exited. exitCode in low-order bits of status

// macros can test the high-order bits of status for why it stopped
```

Processes may also be terminated by the system for a variety of reasons, including:

- The inability of the system to deliver necessary system resources.
- In response to a KILL command, or other un handled process interrupt.
- A parent may kill its children if the task assigned to them is no longer needed.
- If the parent exits, the system may or may not allow the child to continue without a parent.
- When a process terminates, all of its system resources are freed up, open files flushed and closed

2.3 **Inter process Communication:**

There are two types of processes

1. Independent Processes

2. Cooperating Processes

- Independent Processes operating concurrently on systems are those that can neither affect other processes or be affected by other processes.
- Cooperating Processes are those that can affect or be affected by other processes.

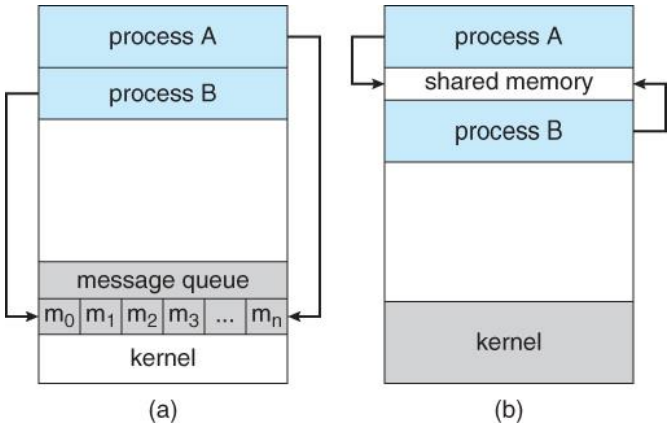
There are several reasons why cooperating processes are allowed:

- **Information Sharing** - There may be several processes which need access to the same file for example. (e.g. pipelines.)
- **Computation speedup** - Often a solution to a problem can be solved faster if the problem can be broken down into sub-tasks to be solved simultaneously (particularly when multiple processors are involved.)
- **Modularity** - The most efficient architecture may be to break a system down into cooperating modules. (E.g. databases with a client-server architecture.)
- **Convenience** - Even a single user may be multi-tasking, such as editing, compiling, printing, and running the same code in different windows.

Cooperating processes require some type of inter-process communication, which is most commonly one of two types:

Shared Memory systems and Message Passing systems.

Figure illustrates the difference between the two systems:



- Shared Memory is faster once it is set up, because no system calls are required and access occurs at normal memory speeds. However it is more complicated to set up, and doesn't work as well across multiple computers. Shared memory is generally preferable when large amounts of information must be shared quickly on the same computer.

- Message Passing requires system calls for every message transfer, and is therefore slower, but it is simpler to set up and works well across multiple computers. Message passing is generally preferable when the amount and/or frequency of data transfers is small, or when multiple computers are involved.

➢ 2.3.1 **Shared-Memory Systems**

In general the memory to be shared in a shared-memory system is initially within the address space of a particular process, which needs to make system calls in order to make that memory publicly available to one or more other processes.

Other processes which wish to use the shared memory must then make their own system calls to attach the shared memory area onto their address space.

Generally a few messages must be passed back and forth between the cooperating processes first in order to set up and coordinate the shared memory access.

Producer-Consumer Example Using Shared Memory

This is a classic example, in which one process is producing data and another process is consuming the data. (In this example in the order in which it is produced, although that could vary.)

The data is passed via an intermediary buffer, which may be either unbounded or bounded. With a bounded buffer the producer may have to wait until there is space available in the buffer, but with an unbounded buffer the producer will never need to wait. The consumer may need to wait in either case until there is data available.

```
#define BUFFER_SIZE 10

typedef struct {

    . . .

} item;



item buffer[ BUFFER_SIZE ];

int in = 0;

int out = 0;
```

- **2.3.2 Message-Passing Systems**

Message passing systems must support at a minimum system calls for "send message" and "receive message". A communication link must be established between the cooperating processes before messages can be sent.

There are three key issues to be resolved in message passing systems as further explored in the next three subsections:

- Direct or indirect communication (naming)
- Synchronous or asynchronous communication
- Automatic or explicit buffering.

2.3.2.1 Direct or indirect communication (Naming):

- With direct communication the sender must know the name of the receiver to which it wishes to send a message.
- There is a one-to-one link between every sender-receiver pair.
- For symmetric communication, the receiver must also know the specific name of the sender from which it wishes to receive messages.
- For asymmetric communications, this is not necessary.

2.3.2.2 Indirect communication: it uses shared mailboxes, or ports.

- Multiple processes can share the same mailbox or boxes.

- Only one process can read any given message in a mailbox. Initially the process that creates the mailbox is the owner, and is the only one allowed to read mail in the mailbox, although this privilege may be transferred.
- The OS must provide system calls to create and delete mailboxes, and to send and receive messages to/from mailboxes.

2.3.2.3 Synchronization:

Either the sending or receiving of messages (or neither or both) may be either blocking or non-blocking.

Blocking send: the sending process is blocked until the message is received by the receiving process or the mail box.

Non blocking send: the sending process sends the message and resumes operation.

Blocking receive: the receiver blocks until a message is available.

Non blocking receiver: the receiver receives either a valid message or a null

2.3.2.4 Buffering: whether communication is direct or indirect message exchange by communicating processes resides in temporary queue., which may have one of three capacity configurations:

- Zero capacity - Messages cannot be stored in the queue, so senders must block until receivers accept the messages.
- Bounded capacity- There is a certain pre-determined finite capacity in the queue. Senders must block if the queue is full, until space becomes available in the queue, but may be either blocking or non-blocking otherwise.
- Unbounded capacity - The queue has a theoretical infinite capacity, so senders are never forced to block.

2.4 Multi thread programming model:

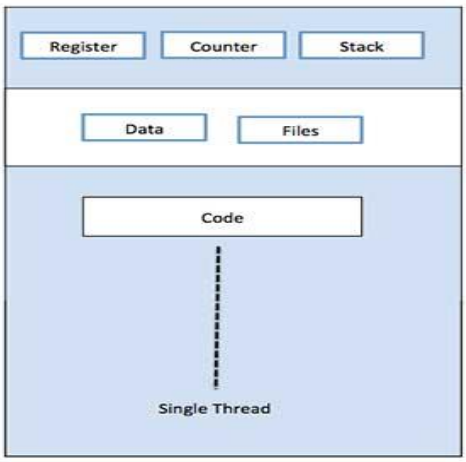
What is Thread?
A thread is a flow of execution through the process code, with its own program counter that keeps rack of which instruction to execute next, system registers which hold its current working variables and stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment and open files. When one thread alters a code segment memory item, all other threads see that.

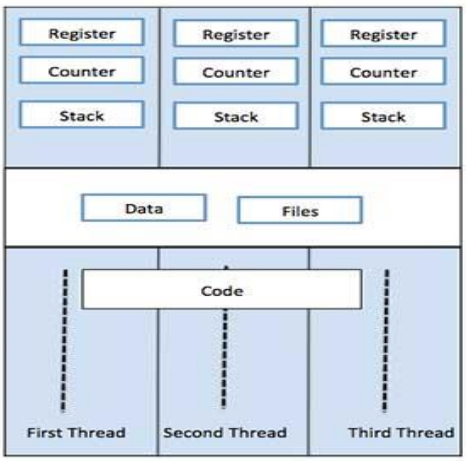
A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Threads represent a software approach to improving performance of operating system by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads has been successfully used in implementing network servers and web server. They also provide a suitable foundation for

parallel execution of applications on shared memory multiprocessors. Following figure shows the working of the single and multithreaded processes.



Single Process P with single thread



Single Process P with three threads

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments 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 process is blocked then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, second thread in the same task can run.
5	Multiple processes without using threads use more Resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates Independently of the others.	One thread can read, write or change another thread's data.

Advantages of Thread:

- Thread minimizes context switching 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 architectures to a greater scale and efficiency.

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

1. **Responsiveness**

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

2. **Resource sharing**

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

3. **Economy**

Allocating memory and resources for process creation is costly. Because threads share resources of the process to which they belong, it is more economical to create and context-switch threads. Empirically gauging the difference in overhead can be difficult, but in general it is much more time consuming to create and manage processes than threads.

In Solaris, for example, creating a process is about thirty times slower than is creating a thread, and context switching is about five times slower.

4. **Scalability:**

The benefits of multithreading can be greatly increased in a multiprocessor architecture, where threads may be running in parallel on different processors. A single threaded process can only run on one CPU, no matter how many are available.

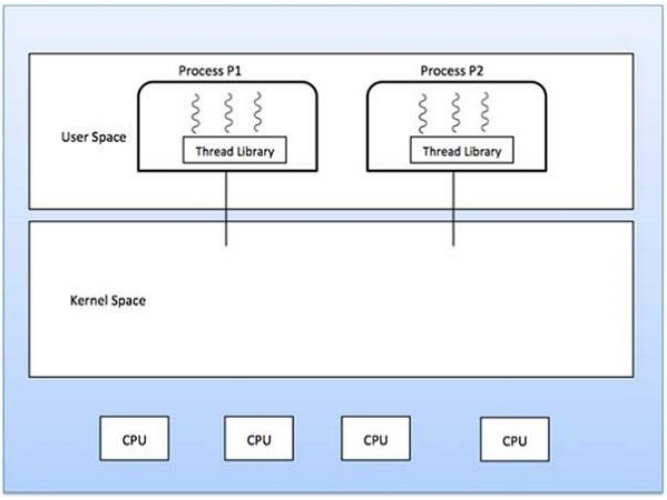
Types of Thread

Threads are implemented in following two ways −

- **User Level Threads** − User managed threads.
- **Kernel Level Threads** − Operating System managed threads acting on kernel, an operating system core.

User Level Threads

In this case, the thread management kernel is not aware of the existence of threads. The thread library contains code for creating and destroying threads, for passing message and data between threads, for scheduling thread execution and for saving and restoring thread contexts. The application starts with a single thread.



Advantages

- Thread switching does not require Kernel mode privileges.
- User level thread can run on any operating system.
- Scheduling can be application specific in the user level thread.
- User level threads are fast to create and manage.

Disadvantages

- In a typical operating system, most system calls are blocking.
- Multithreaded application cannot take advantage of multiprocessing.

Kernel Level Threads

In this case, thread management is done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. Any application can be programmed to be multithreaded. All of the threads within an application are supported within a single process.

The Kernel maintains context information for the process as a whole and for individuals threads within the process. Scheduling by the Kernel is done on a thread basis. 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

- 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

- Kernel threads are generally slower to create and manage than the user threads.

- Transfer of control from one thread to another within the same process requires a mode switch to the Kernel.

Multithreading Models

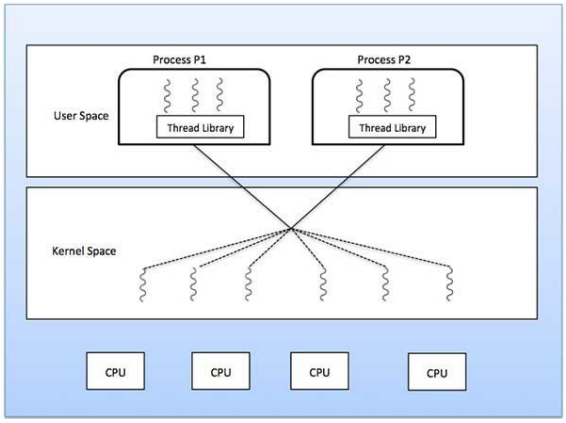
Some operating system provides a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking system call need not block the entire process. Multithreading models are three types

- Many to many relationship.
- Many to one relationship.
- One to one relationship.

Many to Many Models:

The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads.

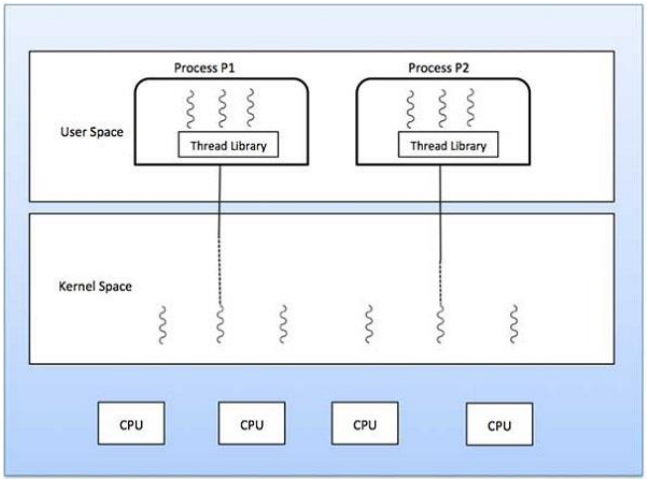
The following diagram shows the many-to-many threading model where 6 user level threads are multiplexing with 6 kernel level threads. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallel on a multiprocessor machine. This model provides the best accuracy on concurrency and when a thread performs a blocking system call, the kernel can schedule another thread for execution.



Many to One Model:

Many-to-one model maps many user level threads to one Kernel-level thread. Thread management is done in user space by the thread library. When thread makes a blocking system call, the entire process will be blocked. Only one thread can access the Kernel at a time, so multiple threads are unable to run in parallel on multiprocessors.

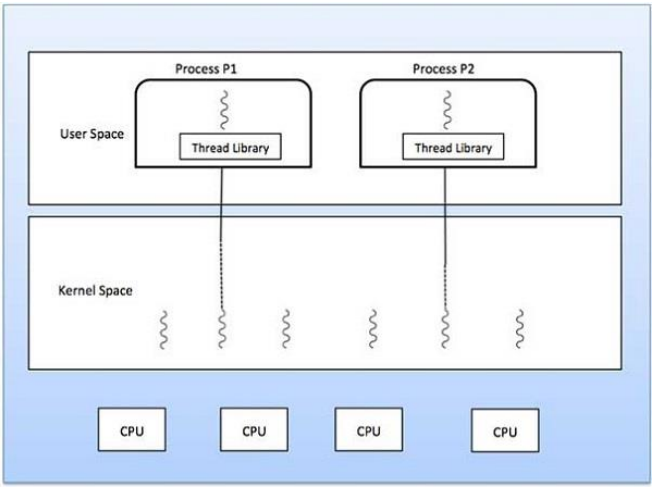
If the user-level thread libraries are implemented in the operating system in such a way that the system does not support them, then the Kernel threads use the many-to-one relationship modes.



One to One Model:

There is one-to-one relationship of user-level thread to the kernel-level thread. This model provides more concurrency than the many-to-one model. It also allows another thread to run when a thread makes a blocking system call. It supports multiple threads to execute in parallel on microprocessors.

Disadvantage of this model is that creating user thread requires the corresponding Kernel thread. OS/2, windows NT and windows 2000 use one to one relationship model.



Difference between User-Level & Kernel-Level Thread

S.N.	User-Level Threads	Kernel-Level Thread

1	User-level threads are faster to create and manage.	Kernel-level threads are slower to create and manage.
2	Implementation is by a thread library at the user level.	Operating system supports creation of Kernel threads.
3	User-level thread is generic and can run on any operating system.	Kernel-level thread is specific to the operating system.
4	Multi-threaded applications cannot take advantage of multiprocessing.	Kernel routines themselves can be multithreaded.

2.5 CPU Scheduling

CPU scheduling is a process which allows one process to use the CPU while the execution of another process is on hold(in waiting state) due to unavailability of any resource like I/O etc, thereby making full use of CPU. The aim of CPU scheduling is to make the system efficient, fast and fair.

Scheduling criteria:

There are many different criteria’s to check when considering the "best" scheduling algorithm :

- **CPU utilization**

  To make out the best use of CPU and not to waste any CPU cycle, CPU would be working most of the time(Ideally 100% of the time). Considering a real system, CPU usage should range from 40% (lightly loaded) to 90% (heavily loaded.)

- **Throughput**

  It is the total number of processes completed per unit time or rather say total amount of work done in a unit of time. This may range from 10/second to 1/hour depending on the specific processes.

- **Turnaround time**

  It is the amount of time taken to execute a particular process, i.e. The interval from time of submission of the process to the time of completion of the process(Wall clock time).

- **Waiting time**

The sum of the periods spent waiting in the ready queue amount of time a process has been waiting in the ready queue to acquire get control on the CPU.

- **Response time**

  Amount of time it takes from when a request was submitted until the first response is produced. Remember, it is the time till the first response and not the completion of process execution (final response).

In general CPU utilization and Throughput are maximized and other factors are reduced for proper optimization.

Scheduling algorithms:

We'll discuss four major scheduling algorithms here which are following :

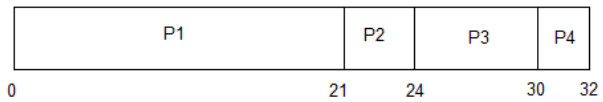
1. First Come First Serve(FCFS) Scheduling
2. Shortest-Job-First(SJF) Scheduling
3. Priority Scheduling
4. Round Robin(RR) Scheduling
5. Multilevel Queue Scheduling

First Come First Serve (FCFS) Scheduling:

- Jobs are executed on first come, first serve basis.
- Easy to understand and implement.
- Poor in performance as average wait time is high.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The average waiting time will be = (0 + 21 + 24 + 30)/4 = 18.75 ms



This is the GANTT chart for the above processes

Shortest-Job-First (SJF) Scheduling:

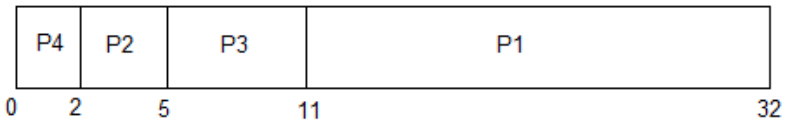
Shortest job next (**SJN**), also known as **shortest job first** (**SJF**) is a scheduling policy that selects the waiting process with the smallest execution time to execute next.

- Best approach to minimize waiting time.
- Actual time taken by the process is already known to processor.
- Impossible to implement.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

In Shortest Job First Scheduling, the shortest Process is executed first.

Hence the GANTT chart will be following :



Now, the average waiting time will be = (0 + 2 + 5 + 11)/ 4 = 4.5 ms

Priority Scheduling:

- Priority is assigned for each process.
- Process with highest priority is executed first and so on.
- Processes with same priority are executed in FCFS manner.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement.

PROCESS	BURST TIME	PRIORITY
P1	21	2
P2	3	1
P3	6	4
P4	2	3

The GANTT chart for following processes based on Priority scheduling will be,

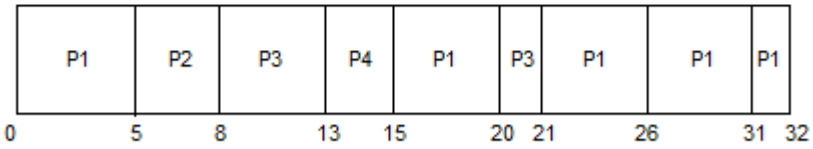
The average waiting time will be, (0 + 3 + 24 + 26)/4 = 13.25 ms

Round Robin(RR) Scheduling:

- A fixed time is allotted to each process, called **quantum**, for execution.
- Once a process is executed for given time period that process is preemptied and other process executes for given time period.
- Context switching is used to save states of preemptied processes.

PROCESS	BURST TIME
P1	21
P2	3
P3	6
P4	2

The GANTT chart for round robin scheduling will be,



The average waiting time will be, 11 ms.

Multilevel Queue Scheduling:

Multi-level queue scheduling algorithm is used in scenarios where the processes can be classified into groups based on property like process type, CPU time, IO access, memory size, etc.

One general classification of the processes is foreground processes and background processes. In a multi-level queue scheduling algorithm, there will be 'n' number of queues, where 'n' is the number of groups the processes are classified into.

Each queue will be assigned a priority and will have its own scheduling algorithm like Round-robin scheduling or FCFS. For the process in a queue to execute, all the queues of priority higher than it should be empty, meaning the process in those high priority queues should have completed its execution. In this scheduling algorithm, once assigned to a queue, the process will not move to any other queues

- Multiple queues are maintained for processes.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

Multilevel feedback queue:

Unlike multilevel queue scheduling algorithm where processes are permanently assigned to a queue, multilevel feedback queue scheduling allows a process to move between queues. This

movement is facilitated by the characteristic of the CPU burst of the process. 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 also helps to prevent starvation of certain lower priority processes.