Syllabus:

Process Management – Process concept, The process, Process State Diagram , Process control block, Process Scheduling- Scheduling Queues, Schedulers, Operations on Processes, Inter process Communication, Threading Issues, Scheduling-Basic Concepts, Scheduling Criteria, Scheduling Algorithms, Multiple Processor Scheduling.

Process Concept:

A process is an instance of a program that is running in a computer. A process is more than the program code, which is sometimes known as the **text section.** It also includes the current activity, as represented by the value of the **program counter** and the contents of the processor's registers. A process generally also includes the process **stack,** which contains temporary data (such as function parameters, return addresses, and local variables), and a **data section,** which contains global variables. A process may also include a **heap,** which is memory that is dynamically allocated during process run time. The structure of a process in memory is shown in the following figure.

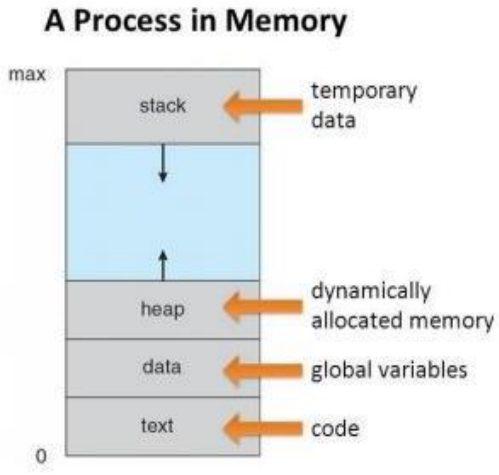
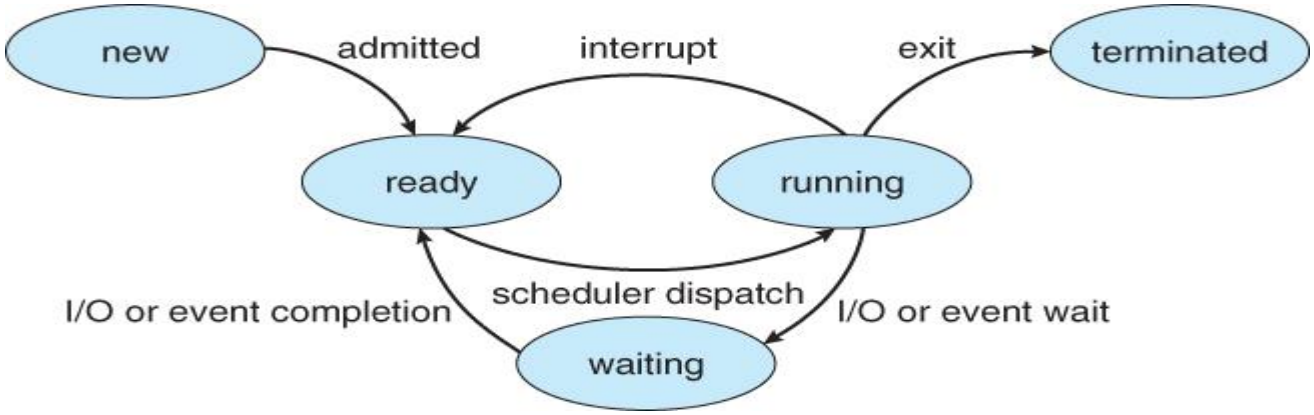


Figure: A process in memory

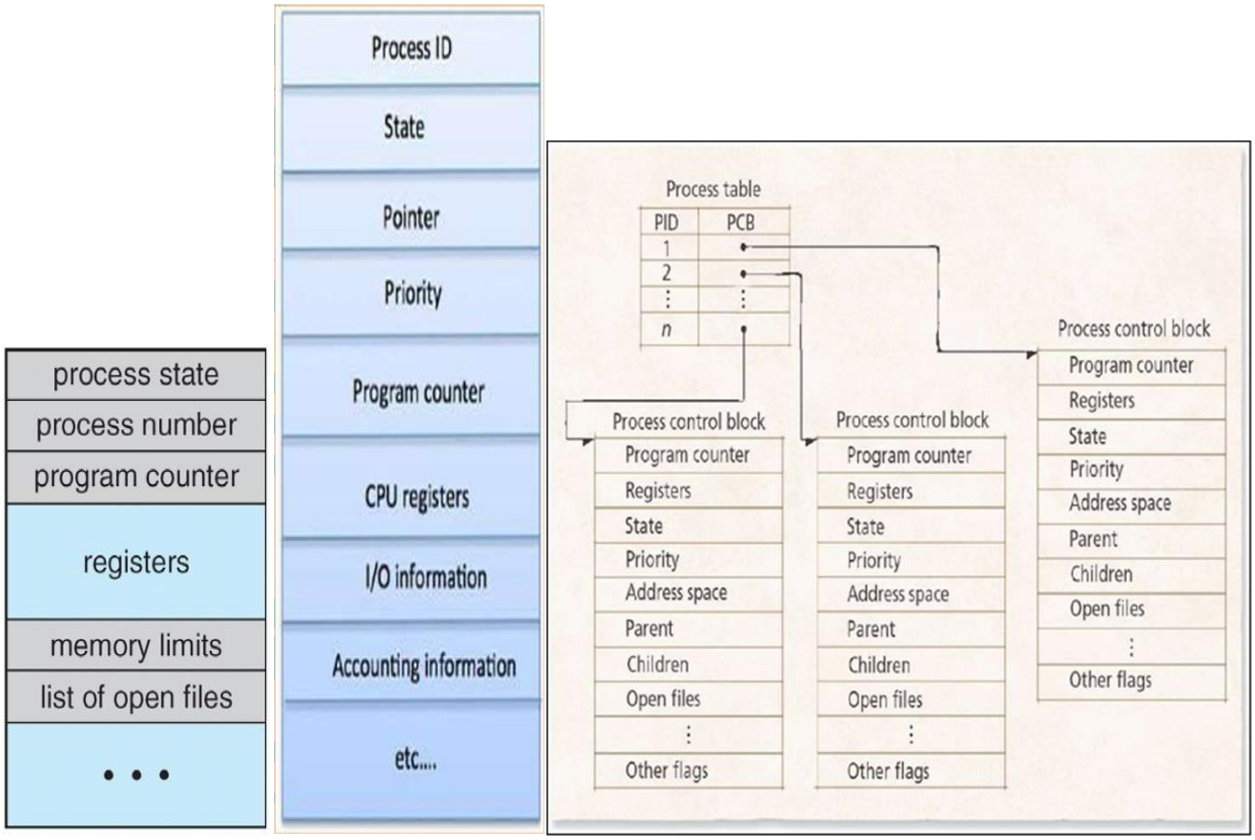
A program is a passive entity, such as a file containing a list of instructions stored on disk (often called an **executable file),** whereas a process is an active entity, with a program counter specifying the next instruction to execute and a set of associated resources. A program becomes a process when an executable file is loaded into memory. Two common techniques for loading executable files are double-clicking an icon representing the executable file and entering the name of the executable file on the command line (as in prog. exe or a. out.)

Process States (or) Life Cycle of a Process: During execution of a process changes its 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:** A program which is going to be picked up by the OS into the main memory is called a new process.
- **Ready:** Whenever a process is created, it directly enters in the ready state, in which, it waits for the CPU to be assigned. The OS picks the new processes from the secondary memory and put all of them in the main memory. The processes which are ready for the execution and reside in the main memory are called ready state processes. There can be many processes present in the ready state.
- **Block or Wait:** From the Running state, a process can make the transition to the block or wait state depending upon the scheduling algorithm or the intrinsic behavior of the process. When a process waits for a certain resource to be assigned or for the input from the user then the OS move this process to the block or wait state and assigns the CPU to the other processes.
- **Running:** One of the processes from the ready state will be chosen by the OS depending upon the scheduling algorithm. Hence, if we have only one CPU in our system, the number of running processes for a particular time will always be one. If we have n processors in the system then we can have n processes running simultaneously.
- **Terminated:** When a process finishes its execution, it comes in the termination state. All the context of the process (Process Control Block) will also be deleted the process will be terminated by the Operating system.

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 the following figure.



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:** This block indicates type of registers used by the process. The registers vary

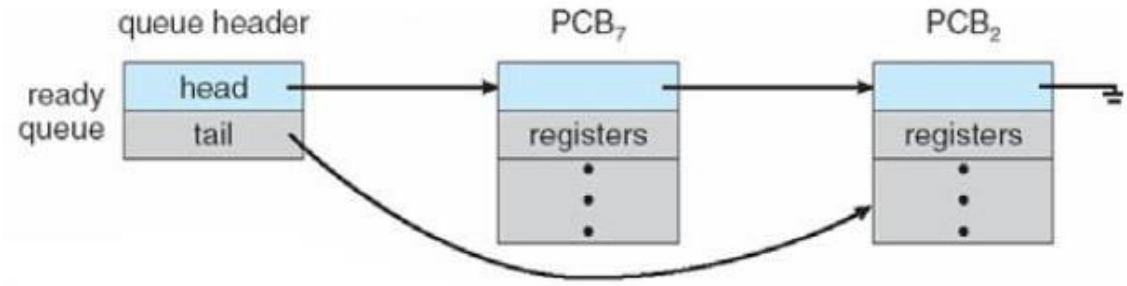
in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information.

- **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.

Process Scheduling: The **process scheduler** selects an available process (possibly from a set of several available processes) for program execution on the CPU. For this processor maintains the following three queues:

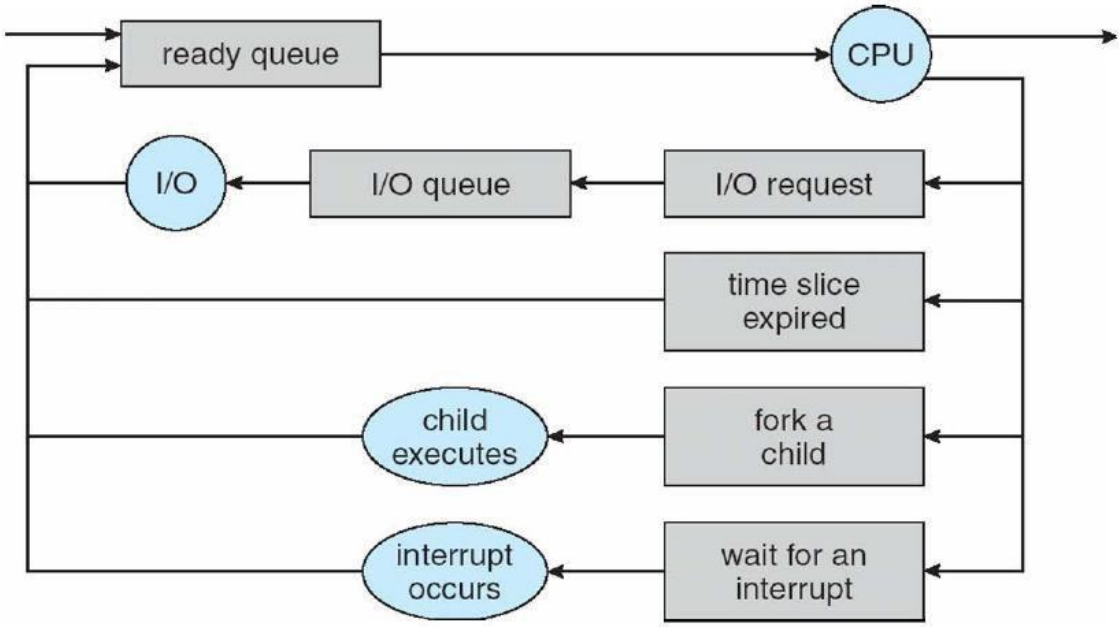
- **Job Queue** - which consists of all processes in the system.
- **Ready Queue**:The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the **ready queue.** This queue is generally stored as a linked list. A ready-queue header contains pointers 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 Queue**: The list of processes waiting for a particular I/O device is called a device **queue.**

Each queue is maintained in the form of linked list as shown below.



A common representation for a discussion of process scheduling is a **queueing diagram** which is shown below. Two types of queues are present: the ready queue and a set of device queues. The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system. A new process is initially put in the ready queue. It waits there tmtil 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 sub process and wait for the sub process'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.



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. There are three different types of schedules:

The **long-term scheduler,** or **job scheduler,** selects processes from this pool and loads them into memory for execution. The long-term scheduler executes much less frequently.

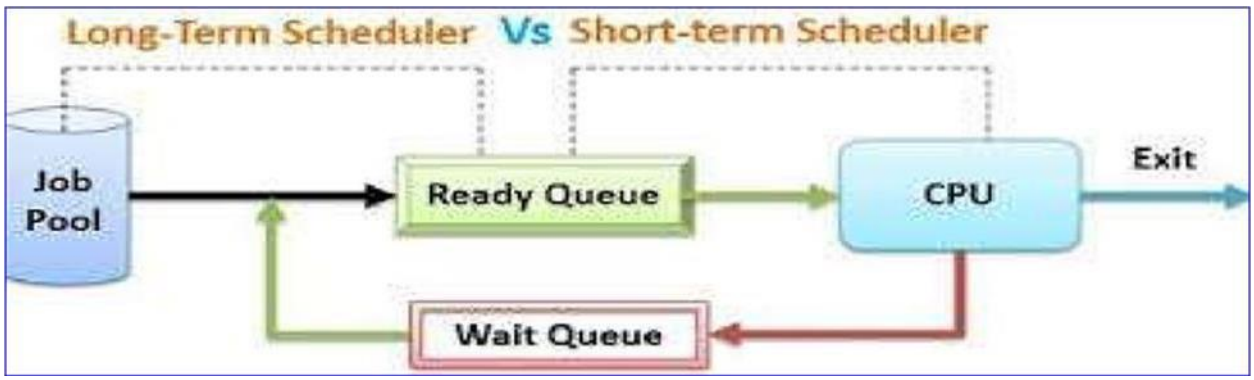
The long-term scheduler controls the **degree of multiprogramming** (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system.

In general, most processes can be described as either I/O bound or CPU bound. A process which spends more amount of time with I/O devices is called as I/O bound process. A process which spends more amount of time with CPU is called as CPU bound process.

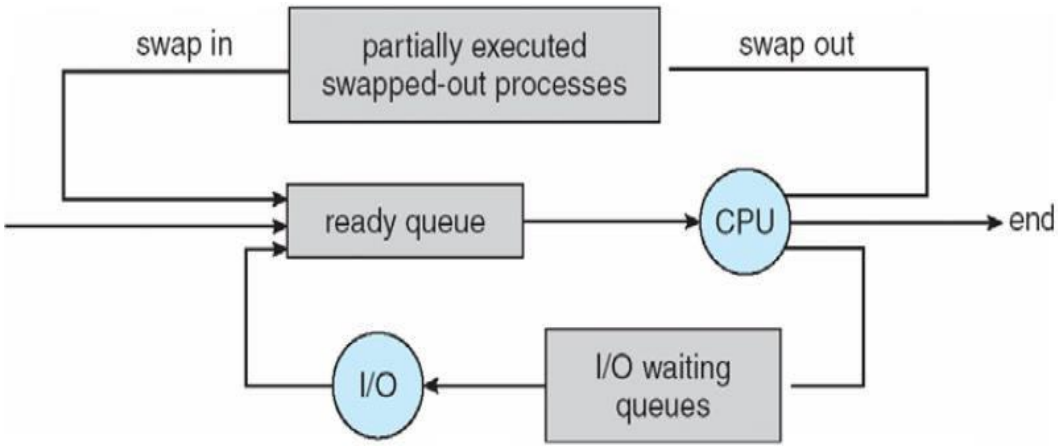
If all processes are I/O bound, the ready queue will almost always be empty. If all processes are CPU bound, the I/O waiting queue will almost always be empty, devices will go unused, and again the system will be unbalanced. The system with the best performance will have a combination of CPU-bound and I/O-bound processes.

The long-term scheduler is also used to control selection of good **Process Mix(Combination of I/O and CPU bound Process)**.

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.



The key idea behind a **medium-term scheduler** is that sometimes it can be advantageous to remove processes from memory and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left off. This scheme is called swapping. The process is swapped out, and is later swapped in, by the medium-term scheduler. Swapping may be necessary to improve the process mix. The following diagram represents swapping process.



Aspects :	Long Term Scheduler	Medium Term Scheduler	Short Term Scheduler
Called as	It is a job scheduler	It is a process swapping	It is a CPU scheduler
Speed	Speed is lesser than short term scheduler	Speed is in between both short and long term scheduler	Speed is fastest among two other scheduler
Multiprogramming	It controls the degree of multiprogramming	It reduces the degree of multiprogramming	It provides lesser control over degree of multiprogramming
Time-sharing system	It is almost absent or minimal in time sharing system	It is a part of Time sharing system	It is also minimal in time sharing system
Processes	It selects processes from pool and loads them into memory for execution	It can reintroduce the process into memory and execution can be continued	It selects those processes which are ready to execute

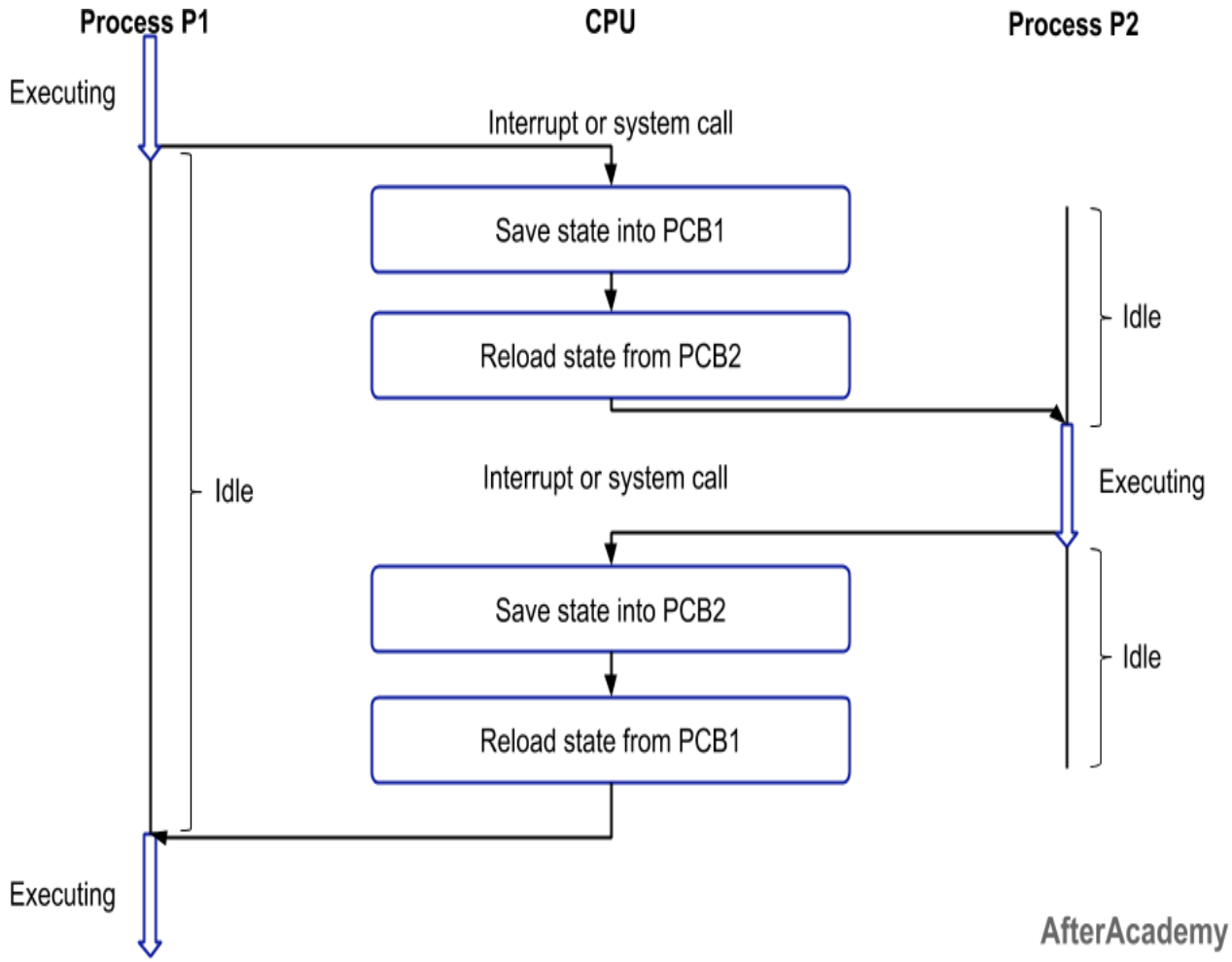
Context Switch: Switching the CPU to another process requires performing a stata save of the current process and a state restore of a different process. This task is known as a **context switch.** When a context switch occurs, the kernel saves the context of the old process in its PCB and loads the saved context of the new process scheduled to run.

The process of context switching involves a number of steps. The following diagram depicts the process of context switching between the two processes P1 and P2.

In the below figure, you can see that initially, the process P1 is in the running state and the

process P2 is in the ready state. Now, when some interruption occurs then you have to switch the process P1 from running to the ready state after saving the context and the process P2 from ready to running state. The following steps will be performed.

1. Firstly, the context of the process P1 i.e. the process present in the running state will be saved in the Process Control Block of process P1 i.e. PCB1.
2. Now, you have to move the PCB1 to the relevant queue i.e. ready queue, I/O queue, waiting queue, etc.
3. From the ready state, select the new process that is to be executed i.e. the process P2.
4. Now, update the Process Control Block of process P2 i.e. PCB2 by setting the process state to running. If the process P2 was earlier executed by the CPU, then you can get the position of last executed instruction so that you can resume the execution of P2.
5. Similarly, if you want to execute the process P1 again, then you have to follow the same steps as mentioned above(from step 1 to 4).



Operations on Processes: The processes in most systems can execute concurrently, and they may be created and deleted dynamically. The following are different types of operations that can be performed on processes.

Process Creation: A process may create several new processes using fork () system call. 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.

When a process creates a new process, two possibilities exist in terms of execution:

- The parent continues to execute concurrently with its children.
- The parent waits until some or all of its children have terminated.

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.

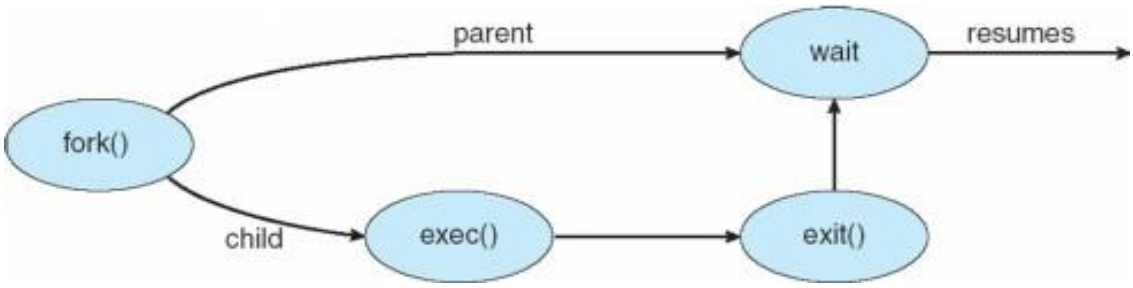


Fig: Process Creation

The following C program illustrates creation of a new process using 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 */
        execlpf"/bin/Is","Is",NULL);
        }
        else
        {       /* parent process. parent will wait for the child to complete */
        wait(NULL);
        printf("Child Complete");
        exit (0) ;
        }
}
```

A new process is created by the **fork ()** system call. The new process consists of a copy of the address space of the original process. This mechanism allows the parent process to communicate easily with its child process. Both processes (the parent and the child) continue execution at the instruction after the fork(), with one difference: The return code for the forkO is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the exec() system call is used after a fork() system call by one of the two processes to replace the process's memory space with a new program. The exec () system call loads a binary file into memory and starts its execution.

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 parent may terminate the execution of one of its children for a variety of reasons, such as these:

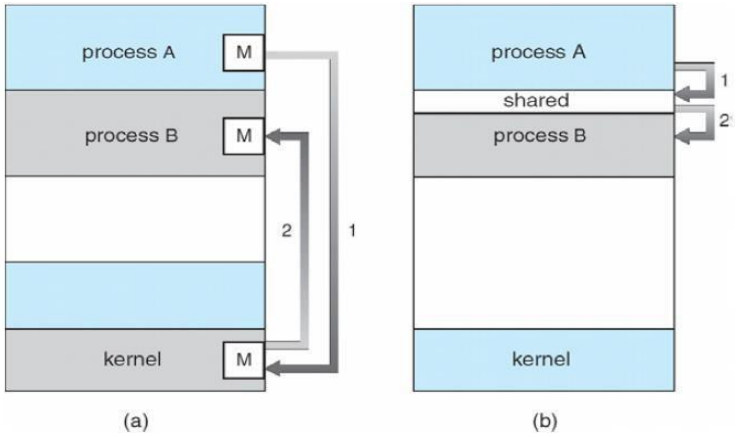
- The child has exceeded its usage of some of the resources that it has been allocated. (To determine whether this has occurred, the parent must have a mechanism to inspect the state of its children.)
- The task assigned to the child is no longer required.
- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

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, sharing provides an environment to allow concurrent access to such information.
- **Computation speedup:** To execute a particular task faster, break it into subtasks, each of which will be executing in parallel with the others.
- **Modularity:** To construct the system in a modular fashion, divide 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** and (2) **message passing.** In the shared-memory model, a region of memory that is shared by cooperating processes is established. Processes can then exchange information by reading and writing data to the shared region. In the messagepassing model, communication takes place by means of messages exchanged between the cooperating processes. The two communications models are contrasted in the following figure.

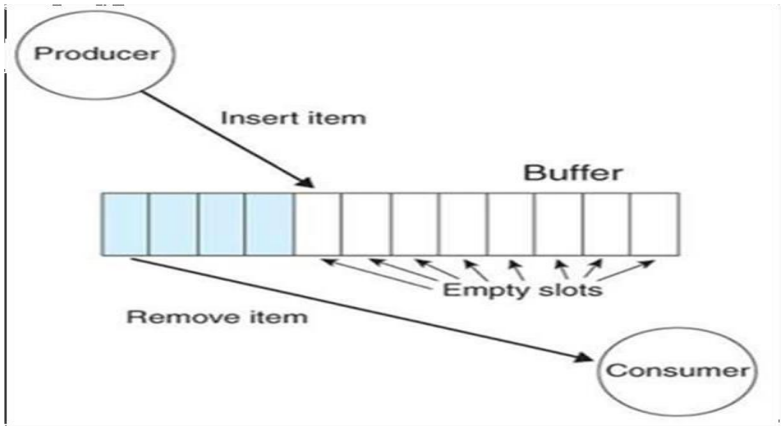


Communications models, (a) Message passing, (b) Shared memory.

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.

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

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, 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.



A producer tries to insert data into an empty slot of the buffer. A consumer tries to remove data from a filled slot in the buffer. As you might have guessed by now, those two processes won't produce the expected output if they are being executed concurrently.

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.

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 buffer is empty when in == out; the buffer is full when ((in + 1) % BUFFER_SIZE) == out.

The code for the producer and consumer processes is shown below.

The Producer process

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

The Consumer process

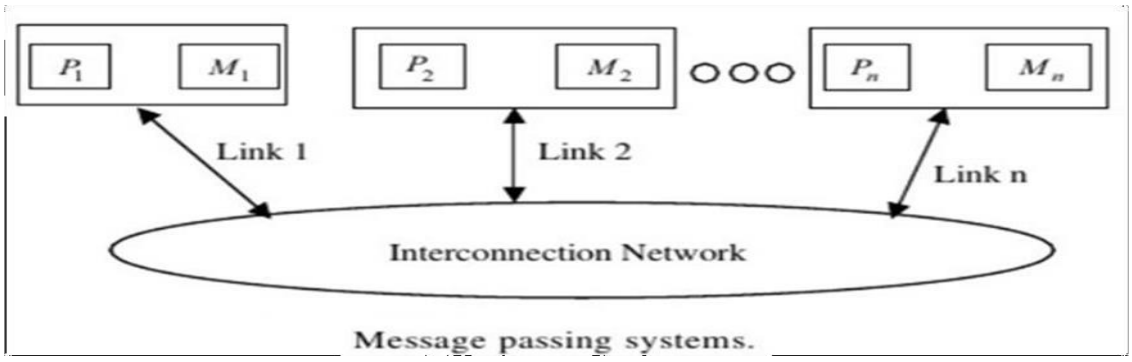
```
item nextConsumed;
while (true)
{
        while (in == out)
        ; //do nothing
        nextConsumed = buffer[out];
        out = (out + 1) % BUFFEFLSIZE;
}
```

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.

Message Passing Systems: Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. A message-passing facility provides at least two operations: send(message) and receive(message). Messages sent by a process can be of either fixed or variable size.

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 in Naming**
- **Synchronous or asynchronous communication**
- **Buffering**



Message passing systems.

- **Direct or indirect communication**: Under **Symmetry direct communication**, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send.0 and receive() primitives are defined as:
  - ❖ send(P, message)—Send a message to process P.
  - ❖ receive (Q, message)—Receive a message from process Q.

A communication link in this scheme has the following properties:

➔ A link is established automatically between every pair of processes that want to

communicate.

➔ A link is associated with exactly two processes.

➔ Between each pair of processes, there exists exactly one link.

In **asymmetry direct communication** 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. 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 sendC) and receive () primitives are defined as follows:

- ❖ send(A, message)—Send a message to mailbox A.
- ❖ receive(A, message)—Receive a message from mailbox A.

In this scheme, a communication link has the following properties:

➔ A link is established between a pair of processes only if both members of the pair have a shared mailbox.

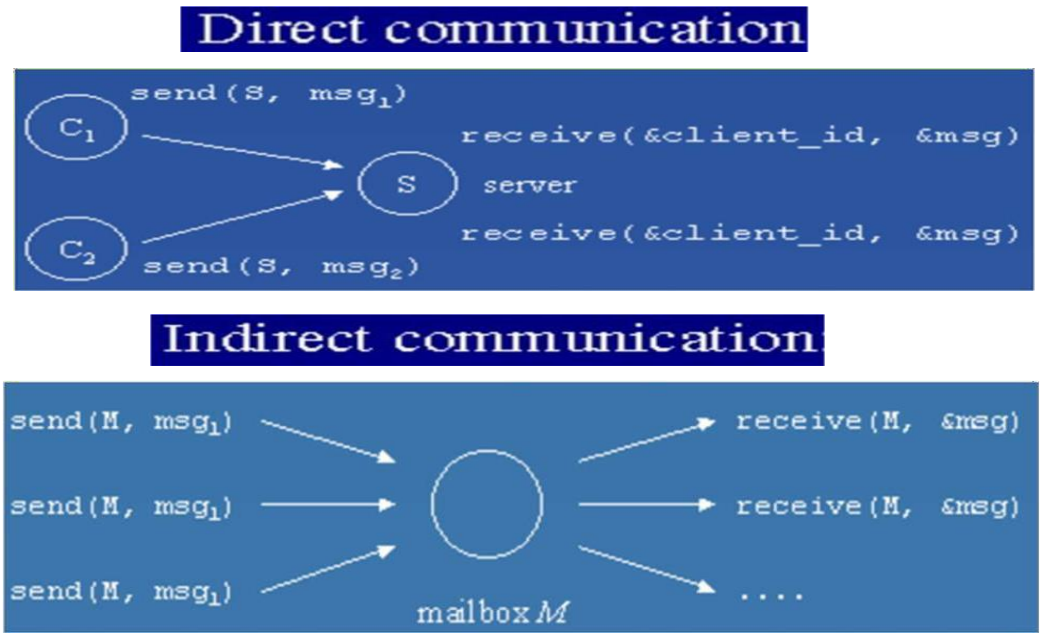
➔ A link may be associated with more than two processes.

➔ Between each pair of communicating processes, there may be a number of different links, with each link corresponding to one mailbox.

A mailbox may be owned either by a process or by the operating system. If the mailbox is owned by a process there can be no confusion about who should receive a message sent to this mailbox(). When a process that owns a mailbox terminates, the mailbox disappears.

In contrast, a mailbox that is owned by the operating system has an existence of its own. 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.

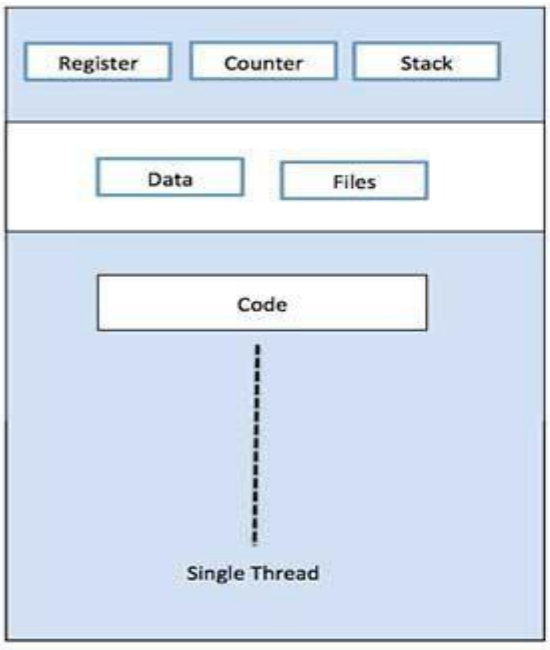


- **Synchronous or asynchronous communication:** Communication between processes takes place through calls to sendO and receive () primitives. There are different design options for implementing each primitive. Message passing may be either **blocking** or **nonblocking**— also known as **synchronous** and **asynchronous.**
  - ❖ **Blocking send.** The sending process is blocked until the message is received by the receiving process or by the mailbox.
  - ❖ **Nonblocking send.** The sending process sends the message and resumes operation.
  - ❖ **Blocking receive.** The receiver blocks until a message is available.
  - ❖ **Nonblocking receive.** The receiver retrieves either a valid message or a null.

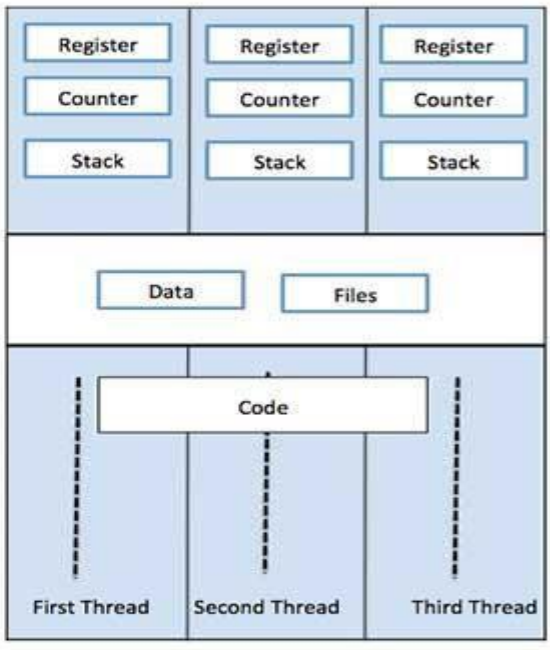
- **Buffering:** Whether communication is direct or indirect, messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:
  - ❖ **Zero capacity:** The queue has a maximum length of zero; thus, the link cannot have any messages waiting 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. If the link is full, the sender must block until space is available in the queue.
  - ❖ **Unbounded capacity:** The queues length is infinite; thus, any number of messages can wait in it. The sender never blocks.

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

Thread: A thread is a flow of execution through the process code, with its own program counter, system registers and stack. A thread is also called a light weight process. Threads provide a way to improve application performance through parallelism. Folowing figure shows the working of the single and multithreaded processes.



Single Process P with single thread



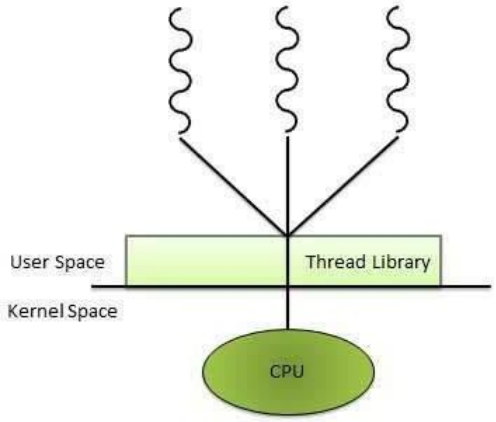
Single Process P with three threads

Difference between Process and Thread:

Process	Thread
Process is heavy weight.	Thread is light weight taking lesser resources than a process.
Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
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.
In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

Types of Threads: Threads are implemented in following two ways:

- **User Level Threads:** User level threads are managed by a user level library. 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. User level threads are typically fast. Creating threads, switching between threads and synchronizing threads only needs a procedure call.



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 done by the Kernel. There is no thread management code in the application area. Kernel threads are supported directly by the operating system. They are slower than user level threads due to the management overhead.

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 multithreaded.

Disadvantages:

- Kernel threads are generally slower to create and manage than the user threads.
- Transfer of control from one thread to another within same process requires a mode switch to the Kernel.

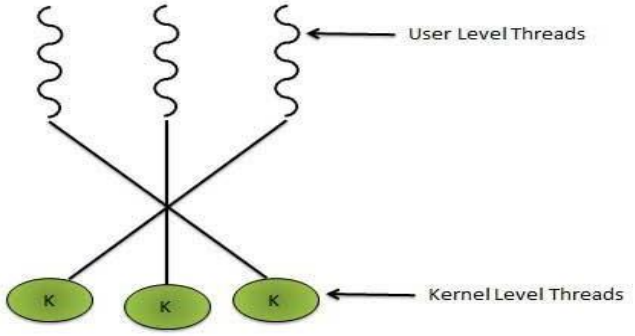
Difference between User-Level & Kernel-Level Thread:

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.

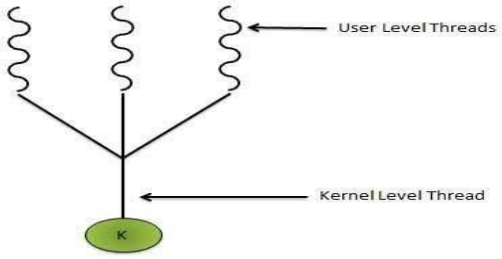
Multi Thread programming Models: Some operating system provide a combined user level thread and Kernel level thread facility. Solaris is a good example of this combined approach. In a combined system, multiple threads within the same application can run in parallel on multiple processors and a blocking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

- Many to Many Model
- Many to One Model
- One to One Model

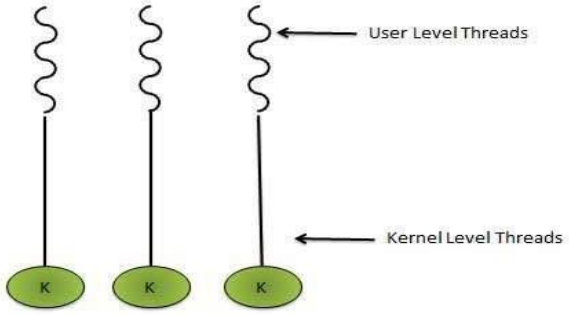
Many to Many Model: In this model, many user level threads multiplexes to the Kernel thread of smaller or equal numbers. The number of Kernel threads may be specific to either a particular application or a particular machine. Following diagram shows the many to many model. In this model, developers can create as many user threads as necessary and the corresponding Kernel threads can run in parallels on a multiprocessor.



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. 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 system does not support them then Kernel threads use the many to one relationship modes.

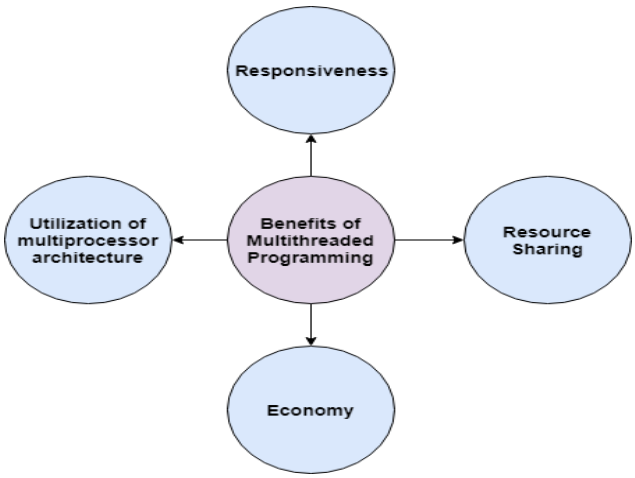


One to One Model: There is one to one relationship of user level thread to the kernel level thread.This model provides more concurrency than the many to one model. It also another thread to run when a thread makes a blocking system call. It support multiple thread to execute in parallel on microprocessors.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



Advantage of Multithreaded Programming

Multithreading allows the execution of multiple parts of a program at the same time. These parts are known as threads and are lightweight processes available within the process. So multithreading leads to maximum utilization of the CPU by multitasking.



- **Resource Sharing**

All the threads of a process share its resources such as memory, data, files etc. A single application can have different threads within the same address space using resource sharing.

- **Responsiveness**

Program responsiveness allows a program to run even if part of it is blocked using multithreading. This can also be done if the process is performing a lengthy operation. For example - A web browser with multithreading can use one thread for user contact and another for image loading at the same time.

- **Utilization of Multiprocessor Architecture**

In a multiprocessor architecture, each thread can run on a different processor in parallel using multithreading. This increases concurrency of the system. This is in direct contrast to a single processor system, where only one process or thread can run on a processor at a time.

- **Economy**

It is more economical to use threads as they share the process resources. Comparatively, it is more expensive and time-consuming to create processes as they require more memory and resources. The overhead for process creation and management is much higher than thread creation and management.

Threading Issues:

- There are a variety of issues to consider with multithreaded programming
  - Semantics of fork() and exec() system calls
  - Thread cancellation
    - Asynchronous or deferred
  - Signal handling
    - Synchronous and asynchronous
  - Thread pooling
  - Thread-specific data
    - Create facility needed for data private to thread

Semantics of fork() and exec():

- Recall that when fork() is called, a separate, duplicate process is created
- How should fork() behave in a multithreaded program?
  - Should all threads be duplicated?
  - Should only the thread that made the call to fork() be duplicated?
  - In some systems, different versions of fork() exist depending on the desired behavior
  - Some UNIX systems have fork1() and forkall()
    - fork1() only duplicates the calling thread
    - forkall() duplicates all of the threads in a process
  - In a POSIX-compliant system, fork() behaves the same as fork1()
- **The exec() system call continues to behave as expected**

- Replaces the entire process that called it, including all threads

- **If planning to call exec() after fork(), then there is no need to duplicate all of the threads in the calling process**
  - All threads in the child process will be terminated when exec() is called
  - Use fork1(), rather than forkall() if using in conjunction with exec()

Thread Cancellation:

- **Thread cancellation is the act of terminating a thread before it has completed**

Example - clicking the stop button on your web browser will stop the thread that is rendering the web page

  - **The thread to be cancelled is called the target thread**
- **Threads can be cancelled in a couple of ways**
  - **Asynchronous cancellation** terminates the target thread immediately
    - Thread may be in the middle of writing data ... not so good
  - **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
    - Allows thread to terminate itself in an orderly fashion

Signal Handling:

- Signals are used in UNIX systems to notify a process that a particular event has occurred

- CTRL-C is an example of an asynchronous signal that might be sent to a process

An asynchronous signal is one that is generated from outside the process that receives it

- Divide by 0 is an example of a synchronous signal that might be sent to a process

- A synchronous signal is delivered to the same process that caused the signal to occur
- **All signals follow the same basic pattern:**
  - A signal is generated by particular event
  - The signal is delivered to a process
  - The signal is handled by a signal handler (all signals are handled exactly once)
- **Signal handling is straightforward in a single-threaded process**

- The one (and only) thread in the process receives and handles the signal

- **In a multithreaded program, where should signals be delivered?**
  - Options:
    (1) Deliver the signal to the thread to which the signal applies
    (2) Deliver the signal to every thread in the process
    (3) Deliver the signal only to certain threads in the process
    (4) Assign a specific thread to receive all signals for the process
- **Option 1 - Deliver the signal to the thread to which the signal applies**

- Most likely option when handling synchronous signals (e.g. only the thread that attempts to divide by zero needs to know of the error)

- **Option 2 - Deliver the signal to every thread in the process**

- Likely to be used in the event that the process is being terminated (e.g. a CTRL- C is sent to terminate the process, all threads need to receive this signal and terminate)

Thread Pools:

- **In applications where threads are repeatedly being created/destroyed thread pools might provide a performance benefit**

- Example: A server that spawns a new thread each time a client connects to the system and discards that thread when the client disconnects

- **A thread pool is a group of threads that have been pre-created and are available to do work as needed**
  - Threads may be created when the process starts
  - A thread may be kept in a queue until it is needed
  - After a thread finishes, it is placed back into a queue until it is needed again
  - Avoids the extra time needed to spawn new threads when they're needed

- Advantages of thread pools:
  - Typically faster to service a request with an existing thread than create a new thread (performance benefit)
  - Bounds the number of threads in a process
- The only threads available are those in the thread pool
- If the thread pool is empty, then the process must wait for a thread to re-enter the pool before it can assign work to a thread
- Without a bound on the number of threads in a process, it is possible for a process to create so many threads that all of the system resources are exhausted

Thread-Specific Data:

- **Thread-specific data - in some applications it may be useful for each thread to have its own copy of data**
  - May also be referred to as Thread-local storage or Thread-static variables

Important CPU scheduling Terminologies:

- **Burst Time/Execution Time:** It is a time required by the process to complete execution. It is also called running time.
- **Arrival Time:** when a process enters in a ready state
- **Finish Time:** when process complete and exit from a system
- **Multiprogramming:** A number of programs which can be present in memory at the same time.
- **Jobs:** It is a type of program without any kind of user interaction.
- **User:** It is a kind of program having user interaction.
- **Process:** It is the reference that is used for both job and user.
- **CPU/IO burst cycle:** Characterizes process execution, which alternates between CPU and I/O activity. CPU times are usually shorter than the time of I/O.

Scheduling Criteria: Many criteria have been suggested for comparing CPU scheduling algorithms. The criteria include the following:

- **CPU Utilization:** In general, 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).
- **Throughput:** One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.
- **Turnaround time:** The interval from the time of submission of a process to the time of completion is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.
- **Waiting time:** Waiting time is the sum of the periods spent waiting in the ready queue.
- **Response time:** 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.

CPU 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 as shown below.

1. **First Come First Serve (FCFS) Scheduling:** In this scheduling, 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.

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Arrival Time	Execute Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	8
P3	3	6	16

Process	Wait Time : Service Time - Arrival Time
P0	0 - 0 = 0
P1	5 - 1 = 4
P2	8 - 2 = 6
P3	16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

This reduction is substantial. Thus, the average waiting time under an FCFS policy is generally not minimal and may vary substantially if the process's CPU burst times vary greatly.

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.

Advantages: Suitable for batch system. It is simple to understand and code

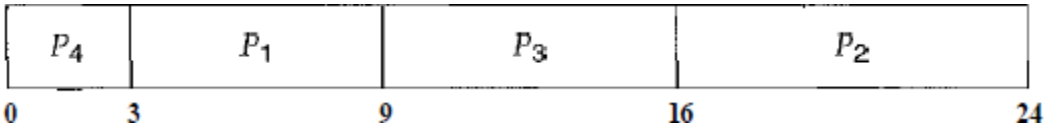
Disadvantages:

- Waiting time can be large if short requests wait behind the long ones.
- It is not suitable for time sharing systems where it is important that each user should get the CPU for an equal amount of time interval.
- A proper mix of jobs is needed to achieve good results from FCFS scheduling.

2. **Shortest-Job-First (SJF) Scheduling (***shortest-next-CPU-burst algorithm***):** A different approach to CPU scheduling is the **shortest-job-first (SJF) scheduling algorithm.** 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. Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time
P_1	6
P_2	8
P_3	7
P_4	3

Using SJF scheduling, the following Gantt chart occurred.

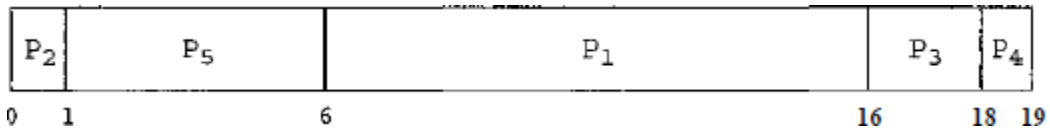


The waiting time is 3 milliseconds for process *P1,* 16 milliseconds for process *P2,* 9 milliseconds for process *P3,* and 0 milliseconds for process P4. Thus, the average waiting time is (3 + 16 + 9 + 0)/4 - *7* milliseconds.

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. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU burst. The larger the CPU burst, the lower the priority. 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, the following Gantt chart will be occurred. The average waiting time is 8.2 milliseconds.



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 process that is ready to run but waiting for the CPU can be considered blocked. 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. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes.

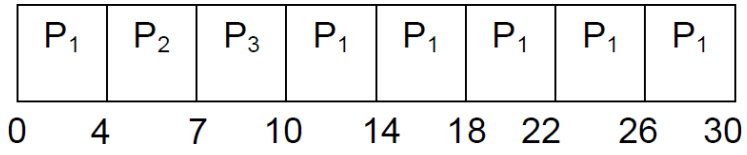
4. **Round Robin (RR) Scheduling:** The **round-robin (RR) scheduling algorithm** is designed especially for timesharing systems. It is similar to FCFS scheduling, but preemption is added to switch between processes. A small unit of time, called a **time quantum** or time slice, is defined. A time quantum 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, 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*. Since process Pi does not need 4 milliseconds, it quits before its time quantum expires. The CPU is then given to the next process, process P3. Once each process has received 1 time quantum, the CPU is returned to process Pi for an additional time quantum. The resulting RR schedule is as shonw below.



The average waiting time is 17/3 = 5.66 milliseconds. In the RR scheduling algorithm, no process is allocated the CPU for more than 1 time quantum in a row (unless it is the only runnable process). If a process's CPU burst exceeds 1 time quantum, that process is *preempted* and is put back in the ready queue. The RR scheduling algorithm is thus preemptive.

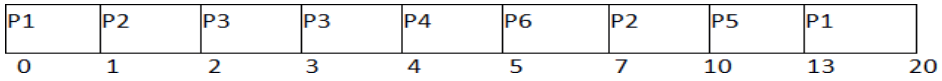
5. **Shortest Remaining Time First (SRTF) Scheduling Algorithm:**

This Algorithm is the **preemptive version** of **SJF scheduling**. In SRTF, the execution of the process can be stopped after certain amount of time. At the arrival of every process, the short term scheduler schedules the process with the least remaining burst time among the list of available processes and the running process.

Once all the processes are available in the **ready queue**, No preemption will be done and the algorithm will work as **SJF scheduling**. The context of the process is saved in the **Process Control Block** when the process is removed from the execution and the next process is scheduled. This PCB is accessed on the **next execution** of this process.

Example, there are five jobs P1, P2, P3, P4, P5 and P6. Their arrival time and burst time are given below in the table.

Process ID	Arrival Time	Burst Time	Completion Time	Turn Around Time	Waiting Time	Response Time
1	0	8	20	20	12	0
2	1	4	10	9	5	1
3	2	2	4	2	0	2
4	3	1	5	2	1	4
5	4	3	13	9	6	10
6	5	2	7	2	0	5



Avg Waiting Time = 24/6

Starvation:-

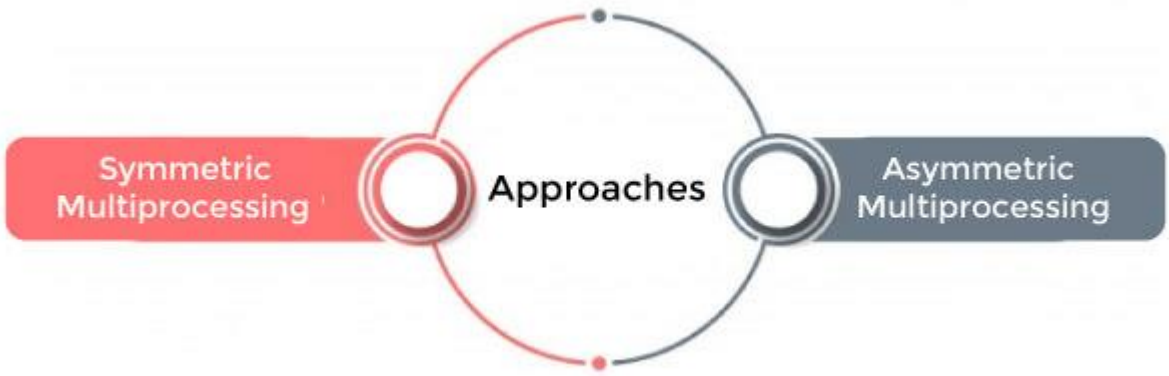
lower priority process never executes. or wait for the long amount of time because of lower priority or highest priority process taking a large amount of time.

Multiple Processor Scheduling:

A multi-processor is a system that has more than one processor but shares the same memory, bus, and input/output devices. In multi-processor scheduling, more than one processors (CPUs) share the load to handle the execution of processes smoothly.

Approaches to Multiple Processor Scheduling

There are two approaches to multiple processor scheduling in the operating system: Symmetric Multiprocessing and Asymmetric Multiprocessing.

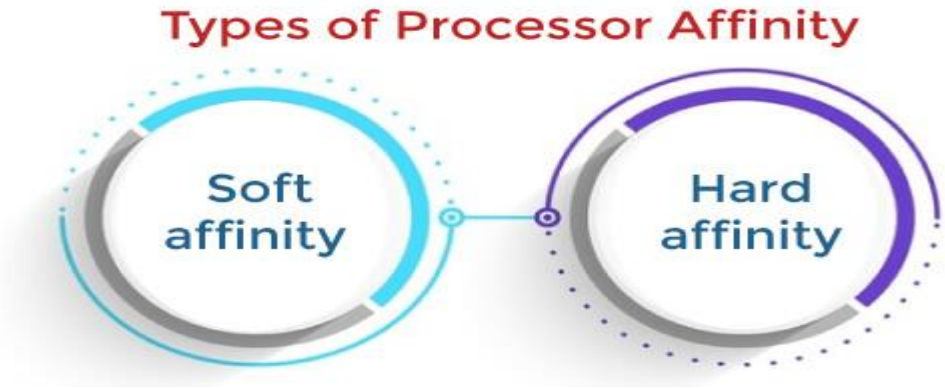


Symmetric Multiprocessing: It is used where each processor is self-scheduling. All processes may be in a common ready queue, or each processor may have its private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

Asymmetric Multiprocessing: It is used when all the scheduling decisions and I/O processing are handled by a single processor called the Master Server. The other processors execute only the user code. This is simple and reduces the need for data sharing, and this entire scenario is called Asymmetric Multiprocessing.

Processor Affinity:

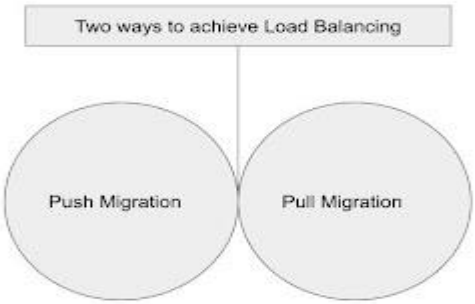
Processor Affinity means a process has an affinity for the processor on which it is currently running. There are two types of processor affinity.



Soft Affinity: When an operating system has a policy of keeping a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.
Hard Affinity: Hard Affinity allows a process to specify a subset of processors on which it may run. Some Linux systems implement soft affinity and provide system calls like sched_setaffinity() that also support hard affinity.

Load Balancing;

Load Balancing is the phenomenon that keeps the workload evenly distributed across all processors in an SMP system. There are two general approaches to load balancing.



Push Migration: In push migration, a task routinely checks the load on each processor. If it finds an imbalance, it evenly distributes the load on each processor by moving the processes from overloaded to idle or less busy processors.
Pull Migration: Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Multi-core Processors:

In multi-core processors, multiple processor cores are placed on the same physical chip. Each core has a register set to maintain its architectural state and thus appears to the operating system as a separate physical processor. SMP systems that use multi-core processors are faster and consume less power than systems in which each processor has its own physical chip.

FREQUENTLY ASKED QUESTIONS

1. What is a Process? Explain about various fields of Process Control Block?
2. With a neat diagram, explain various state of a process?
3. What is scheduler? Explain various types of schedulers and their roles with help of process state diagram?
4. Describe the differences among short term, medium term, and long term scheduling?
5. Explain the following operations on processes: Process Creation and Process Termination
6. What are the advantages of Inter Process Communication? How communication takes place in a Shared memory environment? Explain.
7. Write and explain various issues involved in message passing systems?
8. Define a Thread? Give the benefits of multithreading. Differentiate process and Thread?
9. Explain about different types of multithreading models?
10. Define thread. What are the differences between user level and kernel level thread?
11. What are the criteria for evaluating the CPU scheduling algorithms? Why do we need it.
12. Explain Round Robin Scheduling algorithm with an example?
13. Distinguish between preemptive and non-preemptive scheduling. Explain each type with an example?
14. What are the parameters that can be used to evaluate algorithms? Also explain different algorithmic evaluation methods with advantages and disadvantages?