Operating System Principles – Important Questions and Answers in UNIT-2

1) **Consider the following set of processes, with the length of the CPU burst given in milliseconds:**

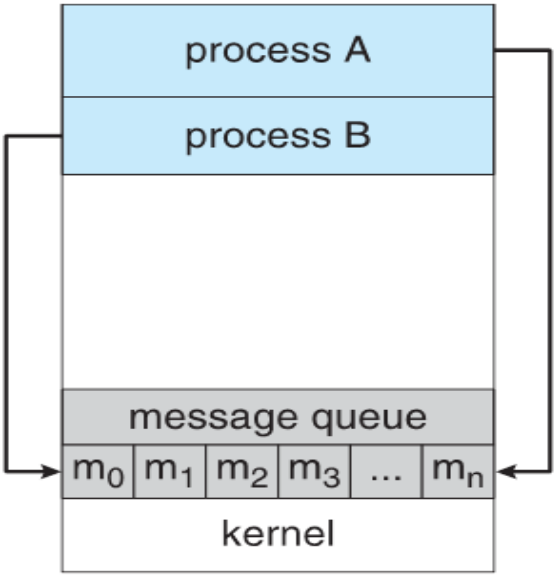
Process	Burst Time	Priority
P1	10	3
P2	1	1
P3	2	3
P4	1	4
P5	5	2

The processes are assumed to have arrived in the order p1, p2, p3, p4, p5, all at time 0. Draw four Gantt charts that illustrates the execution of these processes using the following scheduling algorithms: FCFS, SJF, non-preemptive , priority (a smaller priority number implies a higher priority), and RR (quantum=1). Calculate turnaround time, waiting time of each process and average waiting time for each of these scheduling algorithms.

2) **Demonstrate Inter Process Communication using Message Passing Systems with a neat diagram.**

Cooperating processes require an inter process communication (IPC) mechanism that will allow them to exchange data and Information. Two models of IPC **1. Shared memory 2. Message passing**

Inter process Communication – Message Passing Systems



Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment, where the communicating processes may reside on different computers connected by a network. For example, a chat program used on the World Wide Web could be designed so that chat participants communicate with one another by exchanging messages.

A message-passing facility provides at least two operations: send(message) and receive(message).

Messages sent by a process can be of either fixed or variable size.

1. fixed-sized messages
    - The system-level implementation is straightforward.
    - Makes the task of programming more difficult.
2. variable-sized messages
    - More complex system-level implementation,
    - Programming task becomes simpler.

If processes *P* and *Q* wish to communicate, they need to:

- Establish a ***communication link*** between them
- Exchange messages via send/receive

There are several methods for logically implementing a link and the send()/receive() operations:

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

Direct Communication

- In direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. In this scheme, the send () and receive ()primitives are defined as:
  - send (*P, message*) – send a message to process P
  - receive(*Q, message*) – receive a message from process Q
- Properties of communication link in this case are
  - A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate.
  - A link is associated with exactly two processes.
  - Between each pair of processes, there exists exactly one link.
  - The link may be unidirectional, but is usually bi-directional
- This scheme exhibits *symmetry in addressing; that is, both the sender* process and the receiver process must name the other to communicate. A variant of this scheme employs *asymmetry in addressing. Here, only the sender* names the recipient; the recipient is not required to name the sender.
- In this scheme, the send () and receive() primitives are defined as follows:
  - send(P, message) -Send a message to process P.
  - receive (id, message) -Receive a message from any process; the variable *id is set to the name of the process with which communication has* taken place.
  - The disadvantage in both of these schemes (symmetric and asymmetric) is the limited modularity of the resulting process definitions.
- Changing the identifier of a process may necessitate examining all other process definitions. All references to the old identifier must be found, so that they can be modified to the new identifier.
- In general, any such hard-coding techniques, here identifiers must be explicitly stated, are less desirable than techniques involving indirection,

Indirect Communication

In 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 send() and receive () primitives are defined as follows:

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

Properties of communication link will be

- Link established only if processes share a common mailbox
- A link may be associated with many processes
- Each pair of processes may share several communication links
- Link may be unidirectional or bi-directional

A mailbox may be owned either by a process or by the operating system.

If the mailbox is owned by a process (that is, the mailbox is part of the address space of the process), then we distinguish between the owner (who can only receive messages through this mailbox) and the user (who can only send messages to the mailbox). When a process that owns a mailbox terminates/ the mailbox disappears. Any process that subsequently sends a message to this mailbox must be notified that the mailbox no longer exists.

In contrast/ a mailbox that is owned by the operating system has an existence of its own. It is independent and is not attached to any particular process. The operating system then must provide a mechanism that allows a process to do the following:

- create a new mailbox (port)
- send and receive messages through mailbox
- delete a mailbox

The process that creates a new mailbox is that mailbox's owner by default. Initially, the owner is the only process that can receive messages through this mailbox. However, the ownership and receiving privilege may be passed to other processes through appropriate system calls. Of course/ this provision could result in multiple receivers for each mailbox.

Synchronization

- Communication between processes takes place through calls to send () and receive () primitives. There are different design options for implementing each primitive. Message passing may be either blocking or non-blocking also known as synchronous and asynchronous.
- Blocking is considered synchronous
  - Blocking send -- the sender is blocked until the message is received
  - Blocking receive -- the receiver is blocked until a message is available
- Non-blocking is considered asynchronous
  - Non-blocking send -- the sender sends the message and continue
  - Non-blocking receive -- the receiver receives:
    - A valid message, or
    - Null message
- Different combinations possible
  - If both send and receive are blocking, we have a rendezvous

Buffering

messages exchanged by communicating processes reside in a temporary queue. Basically, such queues can be implemented in three ways:

1. **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.(rendezvous). This is a system with no buffering.

2. **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. This is a system with bounded buffering.

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

3) What is a process? How is it different from a program? Illustrate various stages of process life cycle.

Process is a program in execution; process execution must progress in sequential fashion.

The process contains Multiple parts

- ✔ The program code, also called **text section**
- ✔ Current activity including **program counter**, processor registers.
- ✔ **Stack** containing temporary data Function parameters, return addresses, local variables etc.
- ✔ **Data section** containing global variables
- ✔ **Heap** containing memory dynamically allocated during run time

Program is ***passive*** entity stored on disk (**executable file**), process is ***active***

- ✔ Program becomes process when executable file loaded into memory

Execution of program started via GUI mouse clicks, command line entry of its name, etc

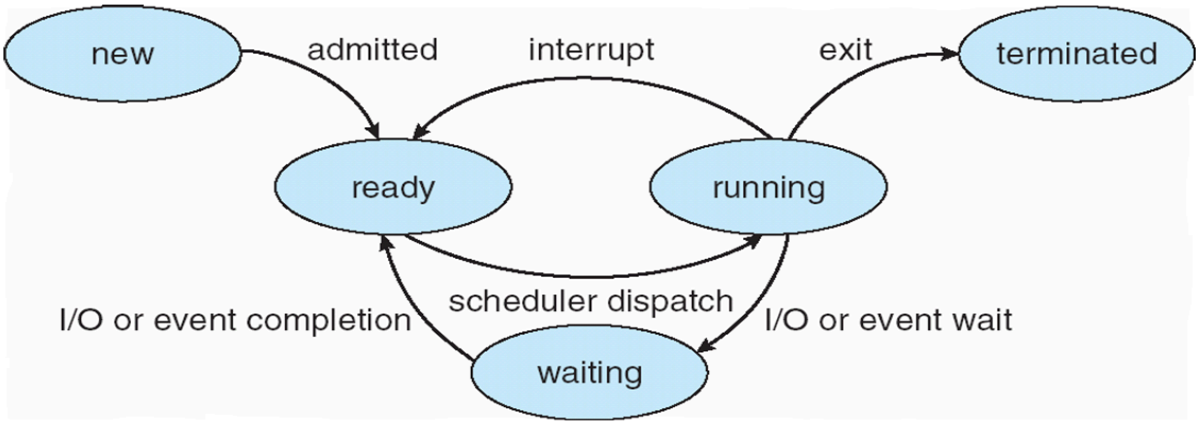
One program can be several processes

- ✔ Consider multiple users executing the same program or the same user may invoke many copies of the web browser program. Each of these is a separate process; and although the text sections are equivalent, the data, heap, and stack sections vary.

As a process executes, it changes **state**

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

Process State Diagram (Stages of Process Life Cycle)



4) **Following is the snapshot of a CPU**

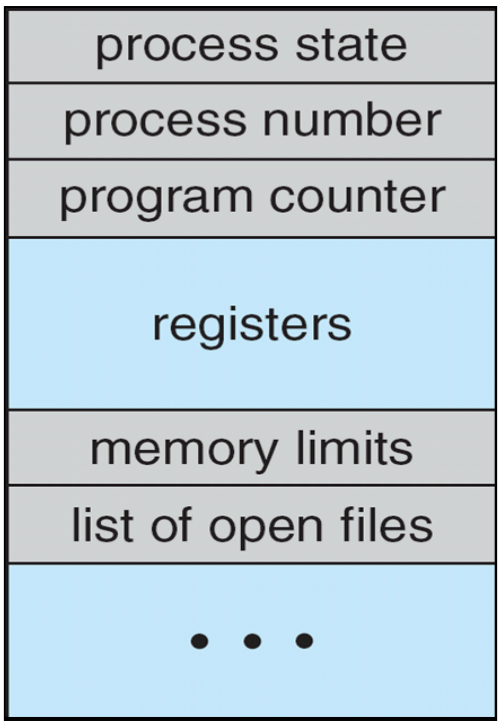
Process	Burst Time	Priority
P1	10	0
P2	29	1
P3	03	2
P4	07	3

Draw the gantt chart and calculate the turnaround time , waiting time , average waiting time of the above processes using FCFS (First come First Serve) ,SJF (Shortest Job First) ,SRTF (Shortest Remaining time first) and RR (Round Robin with time quantum 10) scheduling algorithms

5) **Illustrate process control block with neat sketch.**

Process Control Block

Each process is represented in the operating system by a process control block (PCB)-also called a *task control block.*



A PCB is shown in Figure It contains many pieces of information associated with a specific process, including these:

- **Process state.** The state can be new, ready, running, waiting, halted, etc.
- **Program counter**. The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**. The registers include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter.
- **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 numbers, job or process numbers, and so on.

- **I/O status information.** This information includes the list of I/O devices allocated to the process, a list of open files, and so on.

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

6) Demonstrate Inter Process Communication using Shared Memory Systems.

Cooperating processes require an inter process communication (IPC) mechanism that will allow them to exchange data and Information.

Two models of IPC 1. **Shared memory 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 message passing model, communication takes place by means of messages exchanged between the cooperating processes.

Message passing is easier to implement than is shared memory for inter computer communication.

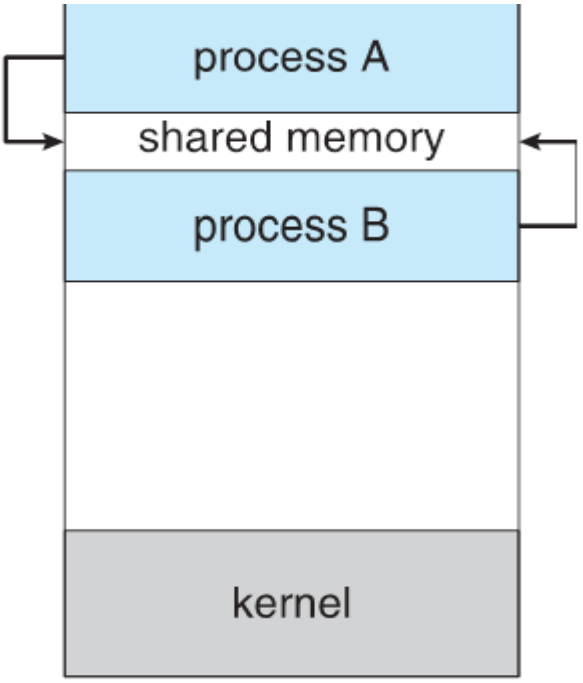
Shared memory is faster than message passing, as message-passing systems are typically implemented using system calls and thus require the more time consuming task of kernel intervention.

In contrast, in shared-memory systems, system calls are required only to establish shared-memory regions. Once shared memory is established, all accesses are treated as routine memory accesses, and no assistance from the kernel is required.

Shared Memory Concept

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.



Recall that, normally, the operating system tries to prevent one process from accessing another process's memory. Shared memory requires that two or more processes agree to remove this restriction. They can then exchange information by reading and writing data in the shared areas. The form of the data and the location are determined by these processes and are not under the operating system's control.

The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

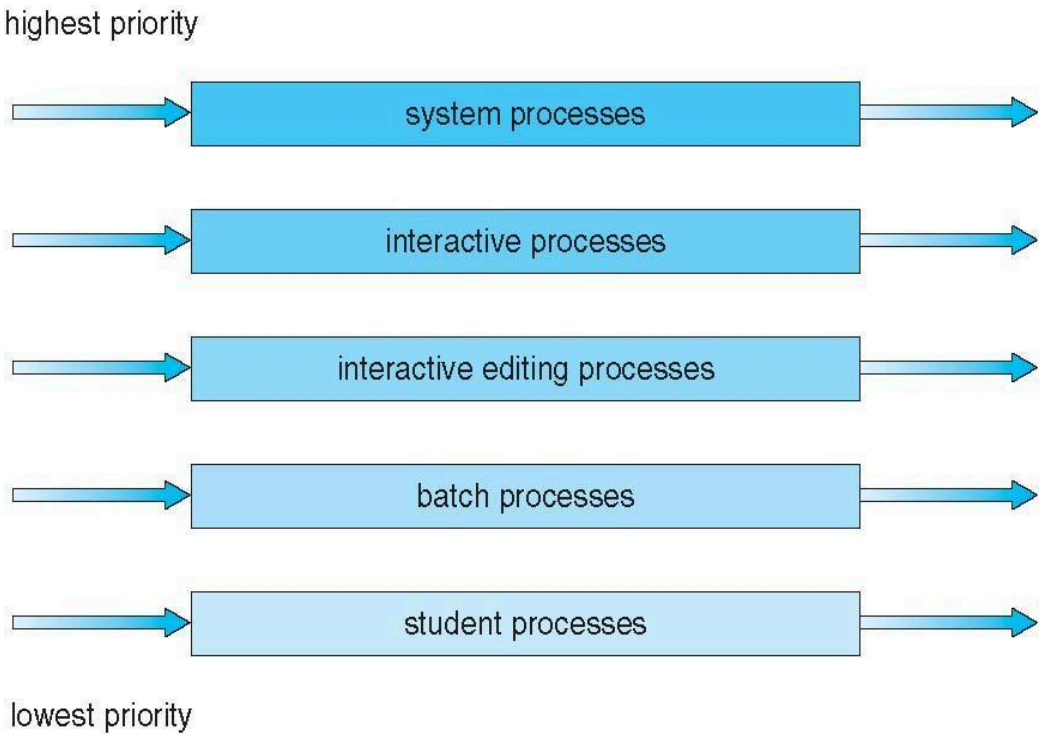
7) **Illustrate Multilevel Queue and Multilevel Feedback Queue Scheduling with neat sketch.**

Multilevel Queue Scheduling

It may happen that processes in the ready queue can be divided into different classes where each class has its own scheduling needs. For example, a common division is a **foreground (interactive)** process

and **background (batch)** processes.These two classes have different scheduling needs. For this kind of situation Multilevel Queue Scheduling is used. Now, let us see how it works.

R**eady Queue** is divided into separate queues for each class of processes. For example, let us take different types of process System processes, Interactive processes and Batch Processes etc. All process have their own queue. Now, look at the below figure.

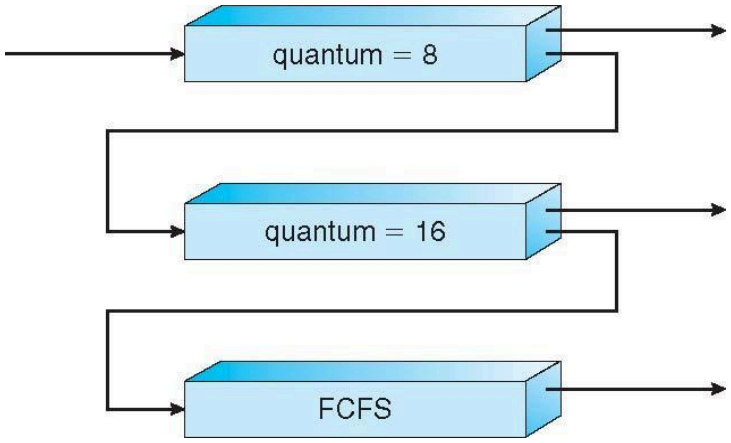


- Each queue has its own scheduling algorithm For example, queue 1 and queue 2 uses **Round Robin** while queue 3 can use **FCFS** to schedule etc
-
- Scheduling must be done between the queues:
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR
  - 20% to background in FCFS

Multi Level Feedback Queue Scheduling

- A process can move between the various queues; aging can be implemented this way
- Multilevel-feedback-queue scheduler defined by the following parameters:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when that process needs service

Example of Multilevel Feedback Queue



- Three queues:
  - Q_0 – RR with time quantum 8 milliseconds
  - Q_1 – RR time quantum 16 milliseconds
  - Q_2 – FCFS

Scheduling

- A new job enters queue Q_0 which is served FCFS
  - When it gains CPU, job receives 8 milliseconds. If it does not finish in 8 milliseconds, job is moved to queue Q_1 .
  - At Q_1 job is again served FCFS and receives 16 additional milliseconds. If it still does not complete, it is preempted and moved to queue Q_2

8) **Describe the concept of process scheduling**.

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization.

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running.

To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU.

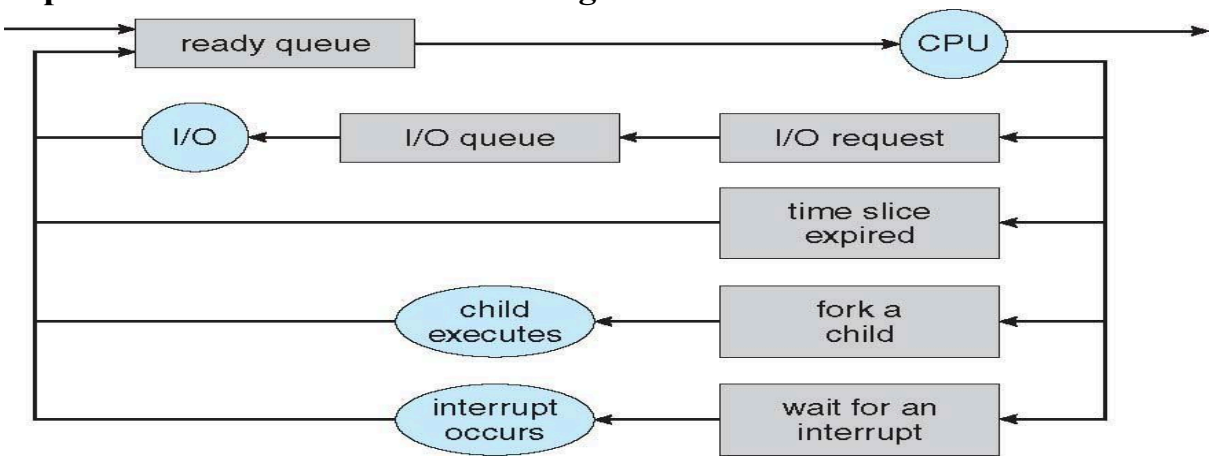
For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled

Process scheduler selects among available processes for next execution on CPU. It Maintains **scheduling queues** of processes

a. **Job queue** – set of all processes in the system
b. **Ready queue** – set of all processes residing in main memory, ready and waiting to execute
c. **Device queues** – set of processes waiting for an I/O device

Processes migrate among the various queues

Representation of Process Scheduling



A common representation for a process scheduling is a queueing diagram, such as that in Figure.

Each rectangular box represents a queue. Two types of queues are present: the ready queue and a set of device queues.

The circles represent the resources that serve the queues, and the arrows indicate the flow of processes in the system.

A new process is initially put in the ready queue. It waits there until it is selected for execution, or is dispatched. Once the process is allocated the CPU and is executing, one of several events could occur:

The process could issue an I/O request and then be placed in an I/O queue.

The process could create a new subprocess and wait for the subprocess's termination.

The process could be removed forcibly from the CPU, as a result of an interrupt, and be put back in the ready queue.

In the first two cases, the process eventually switches from the waiting state to the ready state and is then put back in the ready queue. A process continues this cycle until it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

9) Discuss the operations performed on processes.

The main operations performed on processes are 1. Process Creation
2. Process Termination

Process Creation

A process may create several new processes, via a create-process system call during the course of execution. The creating process is called a parent process, and the new processes are called the children of that process. Each of these new processes may in turn create other processes, forming a tree of processes. Most operating systems (including UNIX and the Windows family of operating systems) identify processes according to a unique **process identifier (or pid),** which is typically an integer number.

In general, a process will need certain resources (CPU time, memory, files, 1/0 devices) to accomplish its task. When a process creates a subprocess, that subprocess may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.

The parent may have to partition its resources among its children, or it may be able to share some resources (such as memory or files) among several of its children. Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.

Resource sharing options

1. Parent and children share all resources
2. Children share subset of parent's resources
3. Parent and child share no resources

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

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

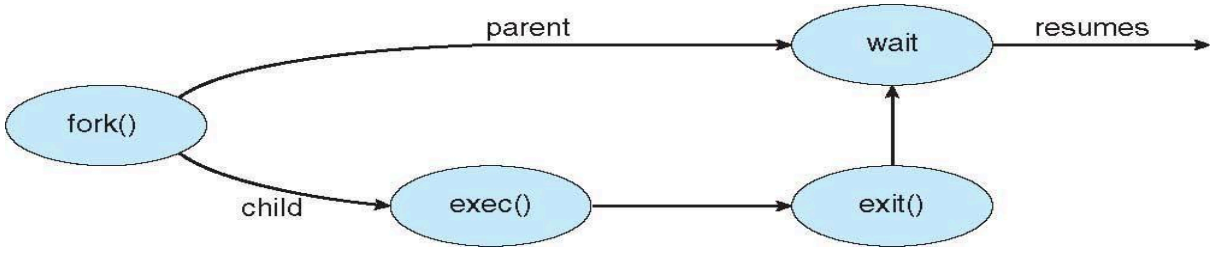
There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process (it has the same program and data as the parent).
2. The child process has a new program loaded into it.

UNIX examples

fork() system call creates new process

exec() system call used after a **fork()** to replace the process' memory space with a new



```c
#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");
       return 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");
    }

    return 0;
}
```

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.

Termination can occur in other circumstances as well. A process can cause the termination of another process using abort() system call , Usually, such a system call can be invoked only by the parent of the process that is to be terminated.

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

• The child has exceeded its usage of some of the resources that it has been allocated.
• The task assigned to the child is no longer required.
• The parent is exiting, and the operating system does not allow a child to continue if its parent terminates.

Some operating systems do not allow child to exists if its parent has terminated. If a process terminates, then all its children must also be terminated.

- **cascading termination.** All children, grandchildren, etc. are terminated.
- The termination is initiated by the operating system.

The parent process may wait for termination of a child process by using the **wait()**system call**.** The call returns status information and the pid of the terminated process

pid = wait(&status);

If no parent waiting (did not invoke **wait()**) process is a **zombie**
If parent terminated without invoking **wait** , process is an **orphan**

10) Consider the following set of processes, with the length of the CPU burst given in milliseconds:

Process	Burst Time	Arrival Time
P1	3	0
P2	4	3
P3	3	7
P4	5	9
P5	7	12

Draw Gantt charts that illustrates the execution of these processes using the following scheduling algorithms: FCFS, SRTF and RR (quantum=3). Calculate average waiting time for each of these scheduling algorithms and identify the algorithm that has minimum average waiting time.

11) Analyze process utilities in Linux Operating System with syntax and examples.

Following is the snapshot of a CPU

Process	CPU Burst	Arrival Time
P1	75	0
P2	40	10
P3	25	10
P4	20	80
P5	45	85

Draw the Gantt chart and calculate the turnaround time waiting time , average waiting time and response time of the jobs for FCFS (First Come First Served), SJF (Shortest Job First), SRTF (Shortest Remaining Time First) and RR (Round Robin with time quantum 15) scheduling algorithms.

12) Discuss About Different Types Of Schedulers

Schedulers are special system software which handle 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 a **job scheduler**. A long-term scheduler determines which programs are admitted to the system for processing. It 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.

On some systems, the long-term scheduler may not be available or minimal. Time-sharing operating systems have no long term scheduler. When a process changes the state from new to ready, then there is use of long-term scheduler.

Short Term Scheduler

It is also called as **CPU scheduler**. Its main objective is to increase 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 a process among the processes that are ready to execute and allocates CPU to one of them.

Short-term schedulers, also known as dispatchers, make the decision of which process to execute next. Short-term schedulers are faster than long-term schedulers.

Medium Term Scheduler

Medium-term scheduling is a part of **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.

A running process may become suspended if it makes an I/O request. A suspended processes cannot make any progress towards completion. In this condition, to remove the process from memory and make space for other processes, 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.

Comparison among Scheduler

S.N.	Long-Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler.
2	Speed is lesser than short term scheduler	Speed is fastest among other two	Speed is in between both short and long term scheduler.

3	It controls the degree of multiprogramming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprogramming.
4	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of Time sharing systems.
5	It selects processes from pool and loads them into memory for execution	It selects those processes which are ready to execute	It can re-introduce the process into memory and execution can be continued.