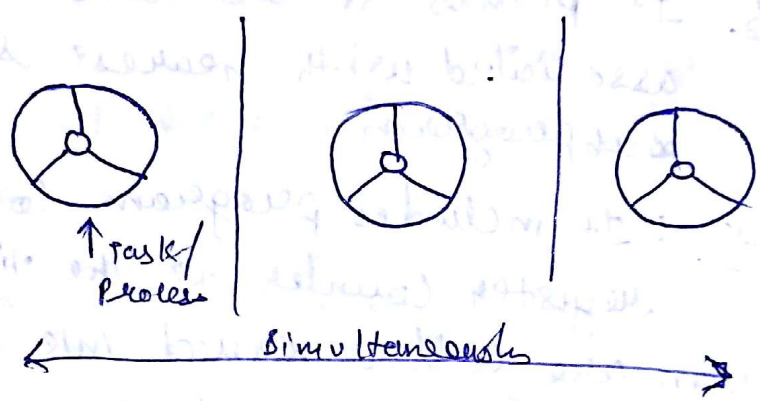
UNIT-III

Concurrent Programming

Two process executes concurrently when their execution time overlaps in time. It is the property of the system in which several computations are execution simultaneously and potentially interacting with each other



Parallel computers: It has several computers each with its own memory & CPU connected with communication links into a network in which each can communicate with others.

Advantages:

1) Speed is increased.
2) Distribution property is Enhanced
3) Scalability is there.
4) Time complexity is reduced.

Scanned by CamScanner

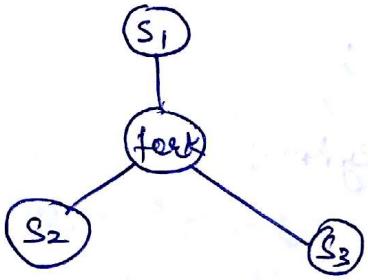
Principle of Concurrent Execution ::

- Call Read process and
- Call write process and
- call execute user program

Language construct for writing concurrent process

i) **fork & join** ⇒ It specifies two concurrent execution in a program

```
    S1
    fork L
begin S2
    :
    :
    L S3;
end
```

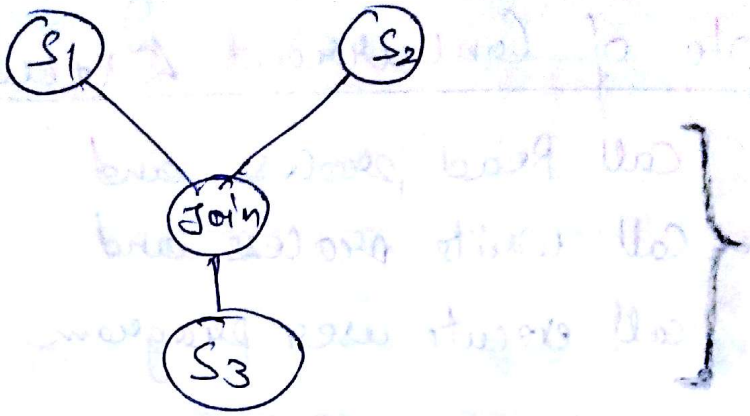


ii) **Code begin** ⇒ It is structured way of denoting the concurrent execution of collection of statements.

```
Code begin
    S1
    S2
    :
    :
    Sn
code end
```

iii) **Join** ⇒ It provide means to combine two Concurrent Computation into one.

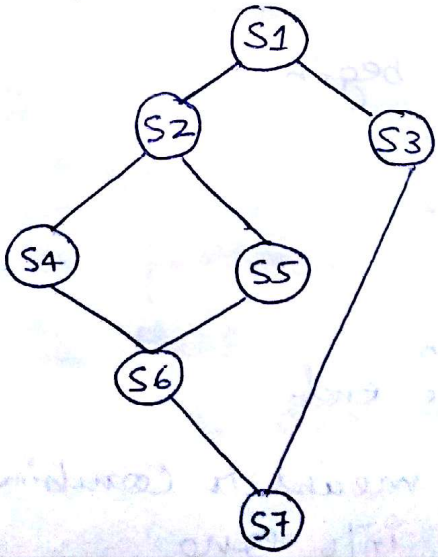
Scanned by CamScanner



Q2 Make a precedence graph for the following program using fork & join command.

```
Codebegin;
S1;
Codebegin;
S3;
end;
begin;
S2;
Codebegin;
S4;
S5;
Code end;
S6;
end;
Code end;
S7;
end;
```

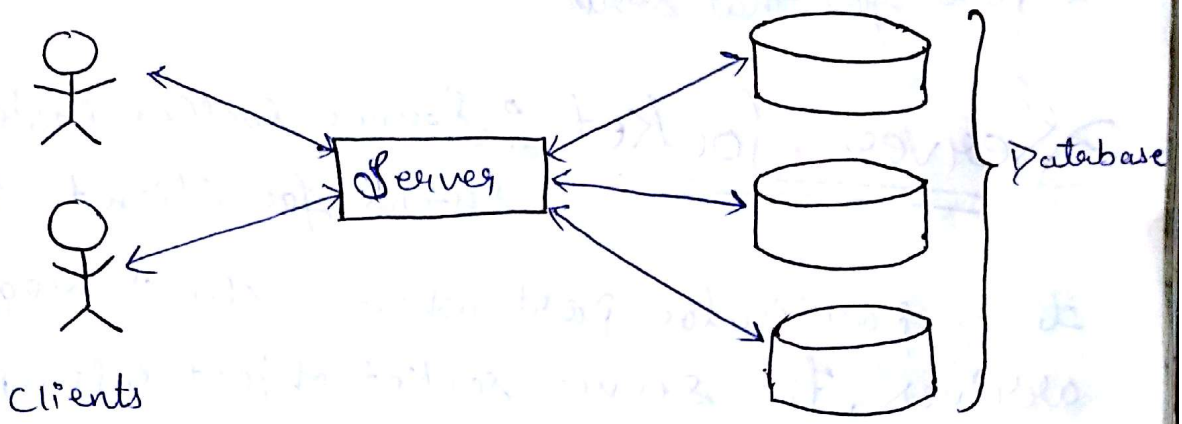
Sol



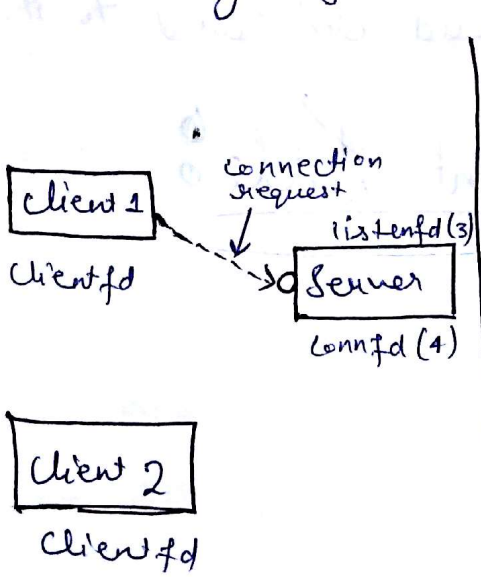
Scanned by CamScanner

Client - Server Computing :

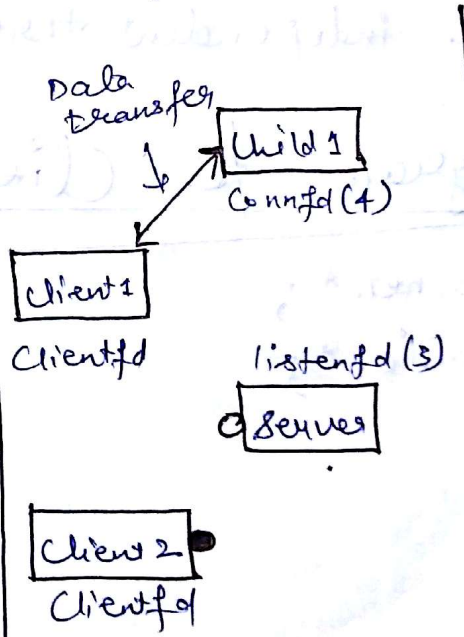
Clients are program running on remote machine that communicate with a program called server that runs at a single sight and respond to request from many clients. Server provide clients with web-pages or database information



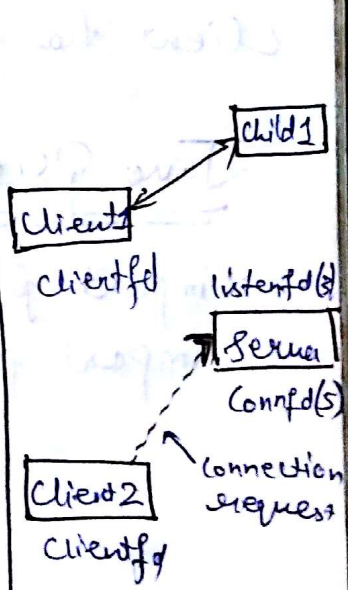
Clients are web browser run by millions of individual users and **Server** are many web hosting systems running at host sites on web.



Server accepts connection request from client.



Server forks a child process to service the client.



server accepts another connection request.

Scanned by CamScanner

In socket based client server System, a server listens to a particular port for client application sending request for connecting with server.
Socket class is provided in java that allows for a server to monitor and answer such request for connection. The client sends a request for a connection by creating a socket with hostname & port for that server

Server Socket : Server socket instance listens for client to connect to a particular port when a client request arrives, the server socket object sets up a socket instance for the connection and then spins of a new thread to interact with clients via socket Many clients can therefore be served since each client has an independent thread dedicated to it.

Scanned by CamScanner

I

```
// File Name GreetingClient.java
import java.net.*;
import java.io.*;
public class GreetingClient {
 public static void main(String [] args) {
      String serverName = args[0];
      int port = Integer.parseInt(args[1]);
      try {
         System.out.println("Connecting to " + serverName + " on port " + port);
         Socket client = new Socket(serverName, port);
          System.out.println("Just connected to " + client.getRemoteSocketAddress());
         OutputStream outToServer = client.getOutputStream();
         DataOutputStream out = new DataOutputStream(outToServer);
          out.writeUTF("Hello from " + client.getLocalSocketAddress());
         InputStream inFromServer = client.getInputStream();
         DataInputStream in = new DataInputStream(inFromServer);
          System.out.println("Server says " + in.readUTF());
         client.close();
       }catch(IOException e) {
         e.printStackTrace();
       }
   }
}
```

[CLIENT SIDE]

II

```
// File Name GreetingServer.java
import java.net.*;
import java.io.*;
public class GreetingServer extends Thread {
   private ServerSocket serverSocket;
   public GreetingServer(int port) throws IOException {
      serverSocket = new ServerSocket(port);
      serverSocket.setSoTimeout(10000);
   }
 public void run() {
```

[SERVER SIDE]

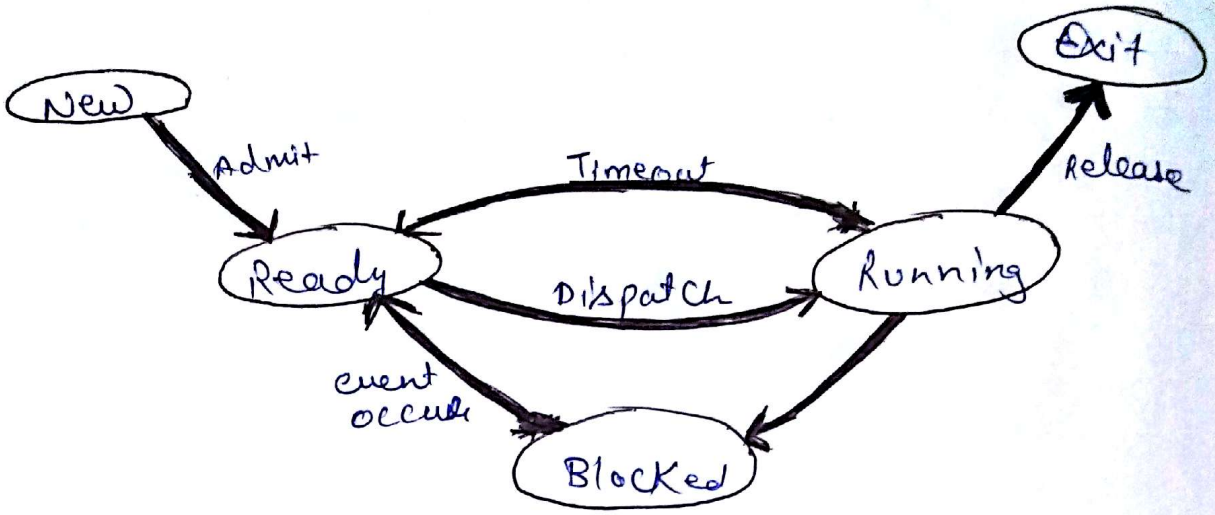
Scanned by CamScanner

```
      while(true) {
         try {
            System.out.println("Waiting for client on port " +
               serverSocket.getLocalPort() + "...");
            Socket server = serverSocket.accept();
             System.out.println("Just connected to " +
server.getRemoteSocketAddress());
            DataInputStream in = new DataInputStream(server.getInputStream());

            System.out.println(in.readUTF());
            DataOutputStream out = new DataOutputStream(server.getOutputStream());
            out.writeUTF("Thank you for connecting to " +
server.getLocalSocketAddress()
               + "\nGoodbye!");
            server.close();
            }catch(SocketTimeoutException s) {
            System.out.println("Socket timed out!");
            break;
         }catch(IOException e) {
            e.printStackTrace();
            break;
         }
      }
   }
public static void main(String [] args)
int port = Integer.parseInt(args[0]);
     try {
        Thread t = new GreetingServer(port);
        t.start();
     }catch(IOException e) {
        e.printStackTrace();
     }
  }
}
```

Scanned by CamScanner

Process: Program in execution is termed as a process.



In Concurrent execution, resources must be scheduled.

Types of schedular.

1) Long-term schedular
2) Short-term schedular
3) Medium-term schedular

i) **Long-term schedular**

It allows limited no. of process ~~es~~ in ready queue and this allows processes beyond certain limits.

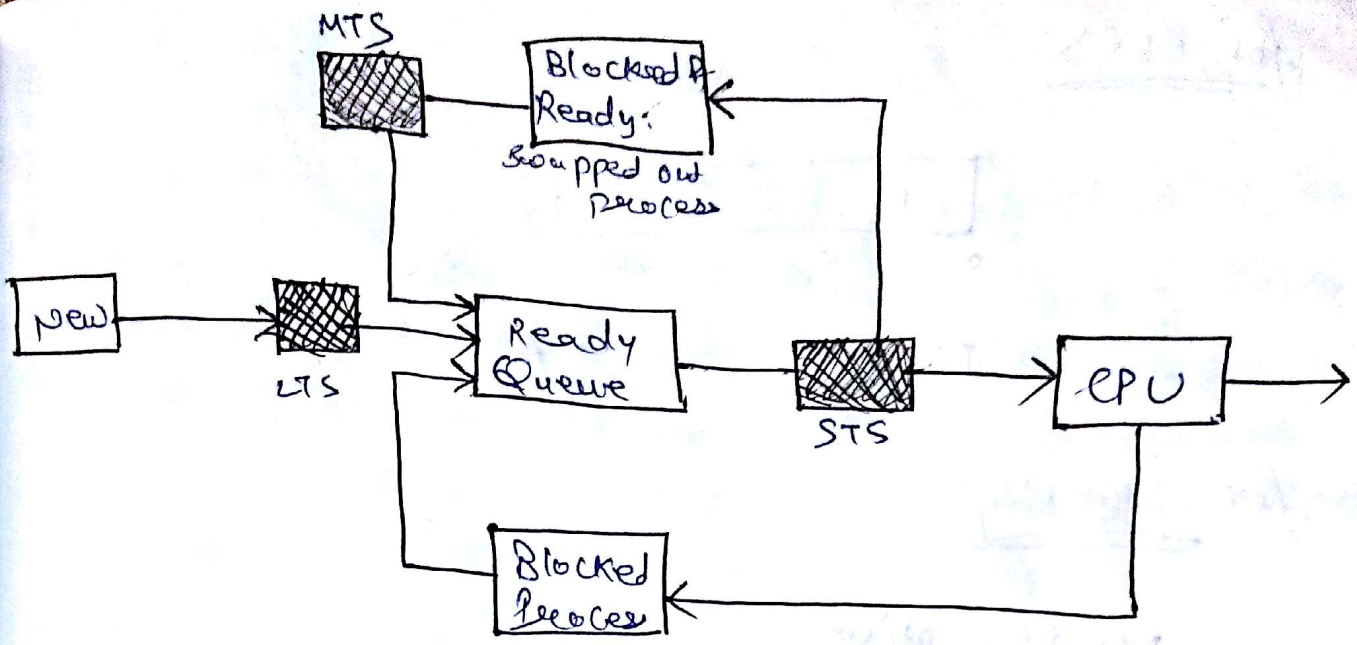
ii) **Short-term schedular**

Besides which of the ready processes to be scheduled next

iii) **Medium-term schedular**

It schedules the swapped out blocked queue.

Scanned by CamScanner



Scheduling Philosophy:

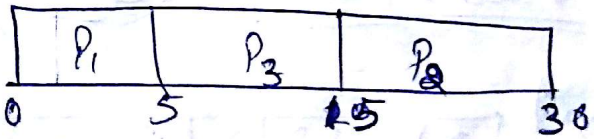
1) Pre-emptive.
2) Non-preemptive.

Q

Process	AT	BT	Priority
P1	0	5	2
P2	1	15	3
P3	2	10	1

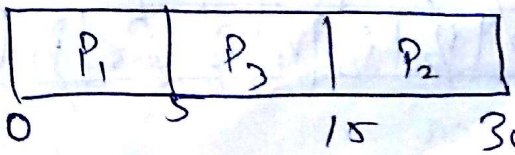
Sol/ for SJF

Non-preemptive.



$WT = \frac{0+13+4}{3} = 17/3$ $\quad TAT = \frac{5+13+29}{3} = 47/3$

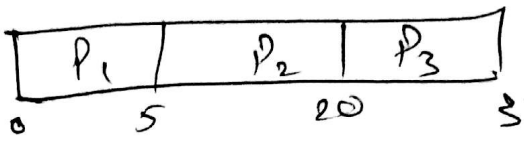
Preemptive.



$WT = \frac{0+13+4}{3} = 17/3$ $\quad TAT = \frac{5+13+29}{3} = 47/3$

Scanned by CamScanner

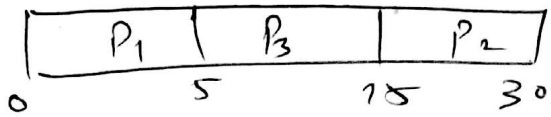
for FCFS



$WT = \frac{22}{3}$ $\quad TAT = 52/3$

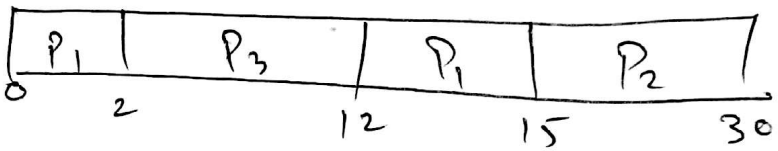
for Priority

Non-preemptive:



$WT = 17/3$ ms $\quad TAT = 47/3$ ms

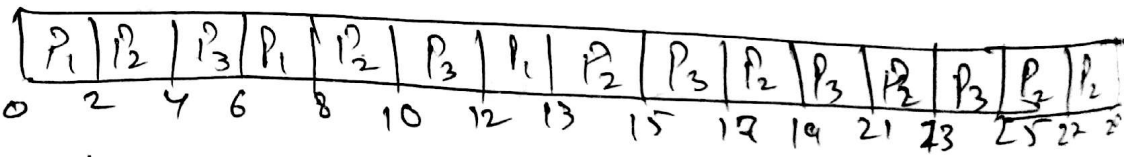
Preemptive:



$WT = 24/3$ ms $\quad TAT = 54/3$ ms.

for Round Robin: (Q = 2)

Gantt



Ready -

$WT = \frac{0+4+4+(2-1+4+3+2+2+2)+(4-2+4+3+2}{3}$

$= 35/3$

$TAT = \frac{13+29+23}{3} = 65/3$

Scanned by CamScanner

Process Synchronization:

Concurrent process come into conflict with each other when they are competing for use of same resource. Thus, synchronization is management that if one process is using the resource the other needs to wait when P_1 exits, it sends the signal to P_2 to use the resource.

Critical Section: Each process has a segment code called critical section When one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

```
do {
    entry section
    critical section
    Exit section
    Remainder section
} while (true);
```

Requirements of critical section:

1) **Mutual Exclusion:** If one process is executing in critical section then no other process can be executing in its critical section.

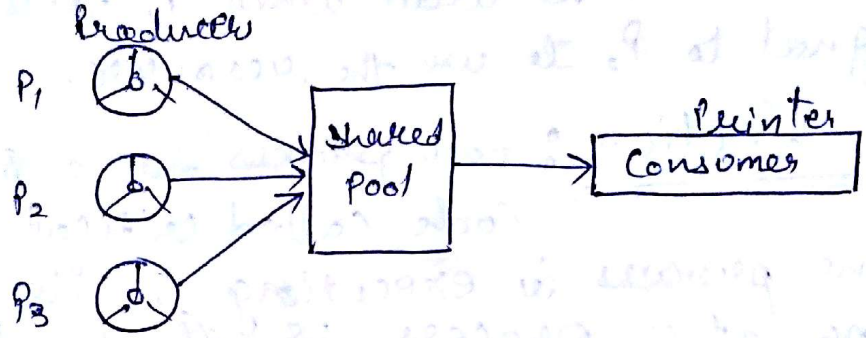
2) **Progress:** If no process is executing in critical section then only those process can enter CS who are not executing in their remainder section.

3) **Bounded waiting:** There exist a bound on no. of time a process can enter its critical section.

Scanned by CamScanner

Producer - Consumer Problem:

It states that if there is one or more producers generating some type of data and placing them in a buffer then there should be a single consumer taking out the items out of the buffer one at a time.



Conditions:

1) Only one process either producer or consumer may access the buffer at any one time.
2) Producer won't try to add into a full buffer and consumer should not try to remove anything from an empty buffer.

```
Producer()
while(1)
{
    while(counter == buffer_size);
    buffer[in] = next produced;
    in = (in+1) % buffer_size;
    counter++;
}
```

```
Consumer()
while(1)
{
    while(counter == 0);
    ~~buffer[out] = next consumed~~
    ~~in = (out+1) %~~
    next consumed = buffer[out];
    out = (out+1) % buffer_size;
    counter--;
}
```

Scanned by CamScanner

Synchronization is achieved by three processes-

1) Semaphores: It is a variable that has an integer value on which three inst. are performed —

a) initialize to a non-negative value.
b) Sem wait decrements semaphore value.
c) Sem signal increments semaphore value.

```
struct semaphore
{
    int count;
    queue type queue;
}
void semwait (semaphore S)
{ S.count--; if(S.count<0)
  {place process in queue;
    block this process;
  }
}
void semsignal (semaphore S)
{
    S.count++;
    if (S.count<=0)
    { remove a process P from queue;
      place queue in ready list}
    }
}
```

Scanned by CamScanner