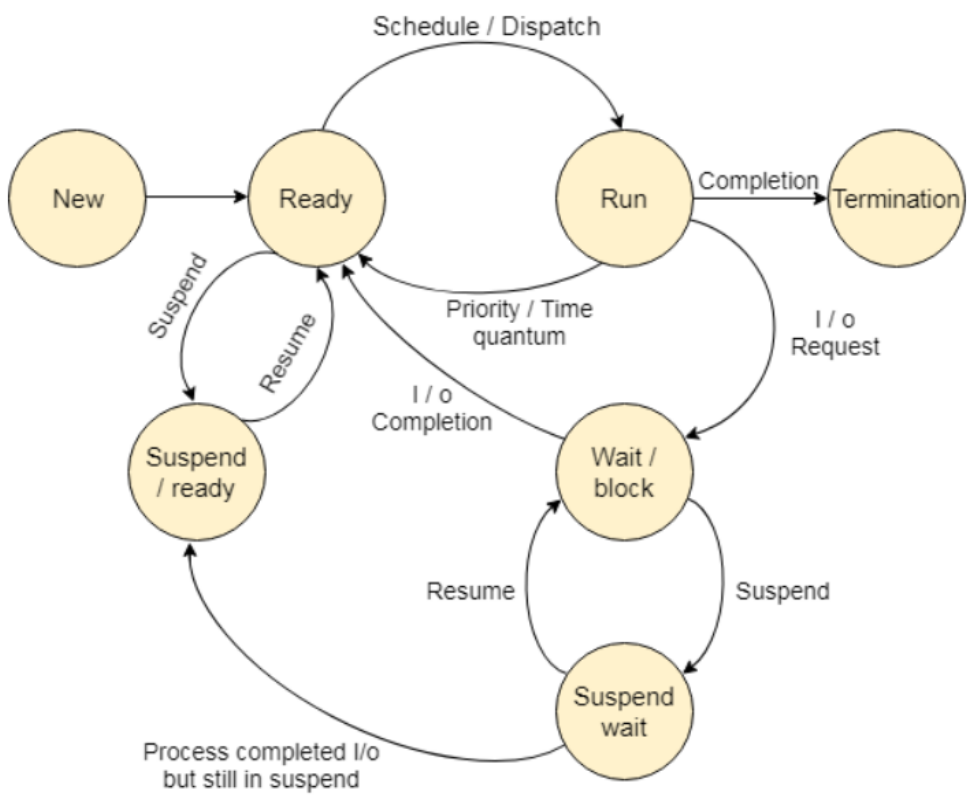
Q 1. What are the typical states of a process? Draw the state transition diagram and briefly state the state and transition details.

State Diagram



States :

1. New

A program which is going to be picked up by the OS into the main memory is called a new process**.**

2. 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.

3. 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.

4. 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.

5. Completion or termination
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.

6. Suspend ready
A process in the ready state, which is moved to secondary memory from the main memory due to lack of the resources (mainly primary memory) is called in the suspend ready state.

If the main memory is full and a higher priority process comes for the execution then the OS have to make the room for the process in the main memory by throwing the lower priority process out into the secondary memory. The suspend ready processes remain in the secondary memory until the main memory gets available.

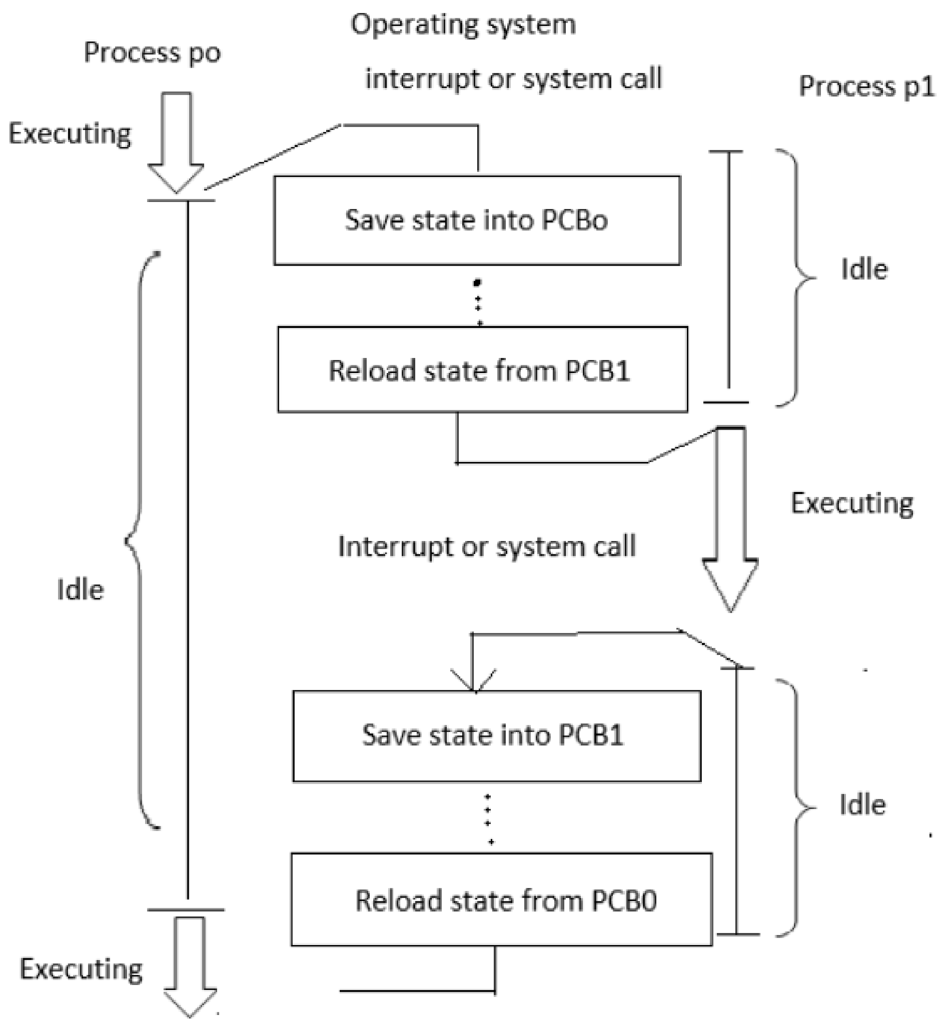
7. Suspend wait
Instead of removing the process from the ready queue, it's better to remove the blocked process which is waiting for some resources in the main memory. Since it is already waiting for some resource to get available hence it is better if it waits in the secondary memory and make room for the higher priority process. These processes complete their execution once the main memory gets available and their wait is finished.

Q 2. Describe the actions taken by a kernel to context-switch between processes. What are the actions in case of a switch between threads?

When an interrupt occurs, the system need to save the current context of the process/threads running on the CPU, so that it can later restore when needed.

Switching the CPU to another process/threads required performing a state save of the current process/threads and state restore of different process/threads and this task is known as context switching

Action taken by kernel to context switch between processes.



1. Process Switching:
Process switching is a type of context switching where we switch one process with another process. It involves switching of all the process resources with those needed by

a new process. This means switching the memory address space. This includes memory addresses, page tables, and kernel resources, caches in the processor.

2. Thread Switching:
Thread switching is a type of context switching from one thread to another thread in the same process. Thread switching is very efficient and much cheaper because it involves switching out only identities and resources such as the program counter, registers and stack pointers.

The cost of thread-to-thread switching is about the same as the cost of entering and exiting the kernel.

Question 3 : Write a pseudocode for a parent process P to create two child processes C1 and C2. C1 uses a pipe() call to send "hello" string to C2. Provide sufficient details of all the system calls used in your code.

Pseudo Code:

```
#define msgsize 16;
char *msg="hello";

int main()
{
        int fd[2];
        pipe(fd);
        char inbuf[msgsize];
        c1=fork(); //child 1 of parent created
        if(c1==1)
        {
                c2=fork(); //child2 of parent created.
        }
        //child 1 writing hello in pipe
        if(c1==0)
        {
                close(fd[0]);
                write(fd[1],msg,msgsize);
                close(fd[1]);
                wait(NULL);
        }
        //child2 reading hello from pipe
        if(c2==0)
        {
                close(fd[1]);
                read(fd[0],inbuf,msgsize);
                printf(inbuf);
                close(fd[0]);
        }
        return 0;
}
```

Q 4. Which of the following scheduling algorithms could or couldn't result in starvation? Clearly explain your answer.

a. First-come, first-served
b. Shortest job first
c. Round robin
d. Priority

a. In First Come First Serve (FCFS) if a process with a very large Burst Time comes before other processes, the other process will have to wait for a long time but it is clear that other process will definitely get their chance to execute, so it will not suffer from starvation.
b. In Shortest Job First (SJF) [pre-emptive] if process with short process time keep on coming continuously then process with higher burst time will do wait and suffer from starvation. [non-preemptive] Similar to won't suffer from starvation.
c. In Round Robin there is a fixed time quant and every process will get their chance to be executed, so no starvation is here.
d. In Priority based scheduling if higher priority process keeps on coming then low priority process will suffer from starvation.

Question 5: A semaphore implementation that doesn't use busy waiting is preferable in many cases, Assume that an operating system provides sleep() and wakeup(P). The sleep() operation suspends the process that invokes it.The wakeup(P) operation resumes the execution of a suspended process P. Using these, define an appropriate semaphore variable S and write the pseudo code to implement wait(*S) and signal(*S) that doesn't use busy-wait.

```
struct semaphore {
          Int value;

        Queue<process> q;

} wait(semaphore s)
{
        if (s.value == 1) {
                s.value = 0;
        }
        else {
                // add the process to the waiting queue
                q.push(P) sleep();
        }
}
Signal(Semaphore s)
{
        if (s.q is empty) {
                s.value = 1;
        }
        else {

                // select a process from waiting queue
                Process p = q.front();
                // remove the process from waiting as it has been
                // sent for CS
                q.pop();
                wakeup(p);
        }
```

}

Q 6. Consider a banking system that maintains an account balance with two functions:
deposit(amount) and withdraw(amount).These two functions are passed the amount that is to be deposited or withdrawn from bank account balance. Assume user A and B share the amount. Concurrently, A calls the withdraw() function, and B calls depost(). Show how a racing condition can occur using specific example.

Definition:It is a situation in which concurrently executing processes accessing the same shared data item may affect the consistency of the data item. Outcome of the execution depends the particular order in which the statements are executed.

Two conditions causes race conditions:
1. Shared variable used by two processes
2. There is non-atomicity in execution of the process.

pseudocode :

```
method withdrawal(amount)
    1. read(amount)
    2. amount=amount-value
    3. store(amount)
end withdrawal

 method deposit(amount)
    1. read(amount)
    2. amount=amount+value
    3. store(amount)
end deposit
```

Here is an example of the race condition:

Case 1:
Suppose the initial value of the amount is ₹ 500. A’s program calls withdrawal(value=₹ 100) and accesses the amount of ₹500. But, before the new amount is stored, preemption occurs and control goes to execute the B’s program. It calls deposit(value=₹100) and accesses the Current Balance of ₹ 500. Then, A's program changes the shared variable amount to ₹ 400, followed by the B's program changing the amount variable to ₹ 600. With a withdrawal(₹ 100) and a deposit(₹ 100), the amount should again be ₹ 500, but it is ₹ 600 (note: it could have gone the other way).

Case 2:
Suppose initial value of amount is ₹ 500. B’s program calls Deposit(value=₹ 100) and accesses the amount of ₹500. But, before the new amount is stored, preemption occurs and control goes to execute the A’s program. It calls Withdrawal(value=₹100) and accesses the Current Balance of ₹ 500. Then, A's program changes the shared variable amount to ₹ 600, followed by the B's program changing the amount variable to ₹ 400. With a withdrawal(₹ 100) and a deposit(₹ 100), the amount should again be ₹ 500, but it is ₹ 400(note: it could have gone the other way).