UNIT-II

Process and CPU Scheduling, Process Coordination

PART – A

1) Define process. What is the information maintained in a PCB?

Ans:

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

PCB:

Process Control Block (**PCB**, also called **Task Controlling Block**, **Entry of the Process Table**, **Task Struct**, or **Switchframe**) is a data structure in the operating system kernel containing the information needed to manage the scheduling of a particular process. The PCB is "the manifestation of a process in an operating system."

Role

The role of the PCBs is central in process management: they are accessed and/or modified by most OS utilities, including those involved with scheduling, memory and I/O resource access and performance monitoring. It can be said that the set of the PCBs defines the current state of the operating system. Data structuring for processes is often done in terms of PCBs. For example, pointers to other PCBs inside a PCB allow the creation of those queues of processes in various scheduling states ("ready", "blocked", etc.) that we previously mentioned.

2) Define process state and mention the various states of a process?

Ans:

PROCESS STATE:

This is the initial **state** when a **process** is first started/created. 2. Ready. The**process** is waiting to be assigned to a processor. Ready **processes** are waiting to have the processor allocated to them by the **operating system** so that they can run.

STATES OF A PROCESS:

1	**Start** This is the initial state when a process is first started/created.

2	**Ready** The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.
3	**Running** Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.
4	**Waiting** Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.
5	**Terminated or Exit** Once the process finishes its execution, or it is terminated by the operating system, it is moved to the terminated state where it waits to be removed from main memory.

3) Describe context switching?

Ans: A context switch is a procedure that a computer's CPU (central processing unit) follows to change from one task (or process) to another while ensuring that the tasks do not conflict. Effective context switching is critical if a computer is to provide user-friendly multitasking.

In a CPU, the term "context" refers to the data in the registers and program counter at a specific moment in time. A register holds the current CPU instruction. A program counter, also known as an instruction address register, is a small amount of fast memory that holds the address of the instruction to be executed immediately after the current one.

4) Explain the use of job queues, ready queues and device queues?

Ans: Job queue contains the set of all processes in the system and ready queue contains the set of all processes residing in main memory and awaiting execution.

Job queue consists of all the processes where ready queue contains processes which are waiting for execution is the major difference. They are mutually exclusive as a process has to move from job queue to ready queue for execution.

When a new process is created it stays in the job queue and if is ready for execution it then moves to ready queue.

5) Distinguish between thread with process?

Ans: **Threads** are used for small tasks, whereas **processes** are used for more 'heavyweight' tasks – basically the execution of applications. Another **difference between** a **thread** and a **process** is that **threads** within the same **process** share the same address space, whereas **different processes** do not.

6) Explain benefits of multithreaded programming?

- Improved throughput.
- Simultaneous and fully symmetric use of multiple processors for computation and I/O
- Superior application responsiveness.
- Minimized system resource usage.
- Better communication.

7) Explain different ways in which a thread can be cancelled?

Cancellation of a target thread may occur in two different scenarios:
1. **Asynchronous cancellation:** One thread immediately terminates the target thread is called asynchronous cancellation.
2. **Deferred cancellation:** The target thread can periodically check if it should terminate, allowing the target thread an opportunity to terminate itself in an orderly fashion.

8) Distinguish between user threads and kernel threads?

Ans

User threads	**Kernel threads**
User threads are supported above the kernel and are implemented by a thread library at the user level	Kernel threads are supported directly by the operating system
Thread creation & scheduling are done in the user space, without kernel intervention. Therefore they are fast to create and manage	Thread creation, scheduling and management are done by the operating system. Therefore they are slower to create & manage compared to user threads
Blocking system call will cause the entire process to block	If the thread performs a blocking system call, the kernel can schedule another thread in the application for execution

9) Define CPU scheduling?

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

10) List the various scheduling criteria for CPU scheduling?

Ans:

First-Come First-Serve Scheduling, FCFS. ...

Shortest-Job-First Scheduling, SJF. ...

Priority Scheduling. ...

Round Robin Scheduling. ...

Multilevel Queue Scheduling. ...

Multilevel Feedback-Queue Scheduling.

11) Distinguish between preemptive and non-preemptive scheduling techniques?

Ans: The main **differences between preemptive and non-preemptive scheduling** are:

Preemptive scheduling is flexible as it allows any high priority process to access the CPU. On the other hand, a **non-preemptive scheduling** is rigid as the current process continues to access the CPU even if other processes enter the queue.

12) Define turnaround time?

Ans: **Turnaround time** (TAT) means the amount of **time** taken to fulfill a request. ... In computing, **turnaround time** is the total **time** taken between the submission of a program/process/thread/task (Linux) for execution and the return of the complete output to the customer/user.

13) List different types of scheduling algorithms?

Ans: First-Come First-Serve Scheduling, FCFS. ...

Shortest-Job-First Scheduling, SJF. ...

Priority Scheduling. ...

Round Robin Scheduling. ...

Multilevel Queue Scheduling. ...

Multilevel Feedback-Queue Scheduling

14) State critical section problem?

Ans: Informally, a **critical section** is a code segment that accesses shared variables and has to be executed as an atomic action. The **critical section problem** refers to the **problem** of how to ensure that at most one process is executing its **critical section** at a given time.

15) State the requirements that a solution to the critical section problem must satisfy?

Ans:

1. **mutual exclusion**: if process p_i is executing in its critical section, no other process is executing in its critical section
2. **progress**: if no process is executing in its critical section and there exists some processes that wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision of which will enter its critical section next, and this decision cannot be postponed indefinitely
    - if no process is in critical section, can decide quickly who enters
    - only one process can enter the critical section so in practice, others are put on the queue
3. **bounded waiting**: there must exist a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
    - The wait is the time from when a process makes a request to enter its critical section until that request is granted
    - in practice, once a process enters its critical section, it does not get another turn until a waiting process gets a turn (managed as a queue)

16) Define race condition?

Ans: A **race condition** is an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly.

17) Define semaphores. Mention its importance in operating system?

Ans: In computer science, a **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system.

Semaphores are a useful tool in the prevention of race conditions; however, their use is by no means a guarantee that a program is free from these problems. Semaphores which allow an arbitrary resource count are called **counting semaphores**, while semaphores which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** and are used to implement locks.

18) State two hardware instructions and their definitions which can be used for implementing mutual exclusion?

- Ans: TestAndSet

```
boolean TestAndSet (boolean &target)
{
        boolean rv = target;
        target = true;
        return rv;
}
```

- Swap

```
void Swap (boolean &a, boolean &b)
{
        boolean temp = a;
        a = b;
        b = temp;
}
```

19) Explain bounded waiting in critical region?

Ans: There exists a bound, or limit, on the number of times other processes are allowed to enter their critical sections after a process has made request to enter its critical section and before that request is granted. That state is called bounded waiting in critical region.

20) Distinguish between semaphore and binary semaphore?

Ans: In computer science, a semaphore is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent system such as a multitasking operating system.

If there is only one count of a resource, a binary semaphore is used which can only have the values of 0 or 1. They are often used as mutex locks.

21) Define monitor?

Ans: A **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. **Monitors** also have a mechanism for signalling other threads that their condition has been met.

22) Describe entry and exit sections of a critical section?

Ans: The critical section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of the code implementing this request is the entry section. The critical section is followed by an exit section. The remaining code is the remainder section.

23) State the real difficulty with the implementation of the SJF CPU scheduling algorithm?

Ans: Difficulty with the implementation of the SJF CPU scheduling algorithm is to know the length of the next CPU request to be processed.

24) State the factors on which the performance of the Round Robin CPU scheduling algorithm depends?

Ans: The performance of **Round Robin algorithm** depends heavily on the **size of the time quantum.**

If time quantum is too large, RR reduces to the FCFS algorithm

If time quantum is too small, overhead increases due to amount of context switching needed.

25) Name the algorithms used for foreground and background queue scheduling in a multilevel queue-scheduling algorithm?

Ans:

1. Foreground (interactive) processes: Round Robin
2. Background (batch) processes: FCFS

26) State the assumption behind the bounded buffer producer consumer problem?

Ans: The bounded buffer producer-consumer problem assumes a fixed buffer size. In this case, the consumer must wait if the buffer is full. The buffer may either be provided by the operating system through the use of an interprocess-communication (IPC) facility, or by explicitly coded by the application programmer with the use of shared memory.

It is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

UNIT-II

Process and CPU Scheduling, Process Coordination

PART-B

1) Explain the reasons for process termination?

Ans:

Normal completion: The process executes an OS service call to indicate that it has completed running.

Time limit exceeded: The process has run longer than the specified total time limit. There are a number of possibilities for the type of time that is measured. These include total elapsed time ("wall clock time"), amount of time spent executing, and, in the case of an interactive process, the amount of time since the user last provided any input.

Memory unavailable: The process requires more memory than the system can provide.

Bounds violation: The process tries to access a memory location that it is not allowed to access.

Protection error: The process attempts to use a resource or a file that it is not allowed to use, or it tries to use it in an improper fashion, such as writing to a read-only file.

Arithmetic error: The process tries a prohibited computation, such as division by zero, or tries to store numbers larger than the hardware can accommodate.

Time overrun: The process has waited longer than a specified maximum for a certain event to occur. **I/O failure**: An error occurs during input or output, such as inability to find a file, failure to read or write after a specified maximum number of tries (when, for example, a defective area is encountered on a tape), or invalid operation (such as reading from the line printer).

Invalid instruction: The process attempts to execute a non-existent instruction.

Privileged instruction: The process attempts to use an instruction reserved for the operating system.

Data misuse: A piece of data is of the wrong type or is not initialized.

Operator or OS intervention: For some reason, the operator or the operating system has terminated the process (for example, if a deadlock exists).

Parent termination: When a parent terminates, the operating system may automatically terminate all of the offspring of that parent.

Parent request: A parent process typically has the authority to terminate any of its offspring.

2) Discuss the following process, program, process state, process control block, and process scheduling?

Ans: Process:

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

To put it in simple terms, we write our computer programs in a text file and when we execute this program, it becomes a process which performs all the tasks mentioned in the program.

Program:

A program is a piece of code which may be a single line or millions of lines. A computer program is usually written by a computer programmer in a programming language. For example, here is a simple program written in C programming language.

```
#include <stdio.h>


int main() {
  printf("Hello, World! \n");
  return 0;
}
```

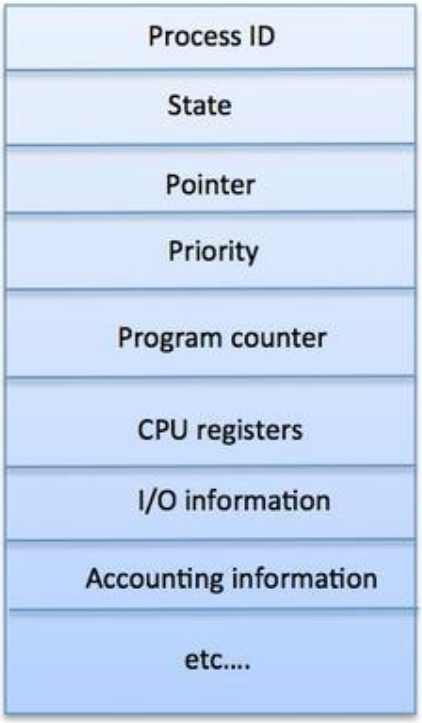
Process Control Block (PCB):

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below in the table −

S.N.	Information & Description
1	**Process State** The current state of the process i.e., whether it is ready, running, waiting, or whatever.
2	**Process privileges** This is required to allow/disallow access to system resources.

3	**Process ID** Unique identification for each of the process in the operating system.
4	**Pointer** A pointer to parent process.
5	**Program Counter** Program Counter is a pointer to the address of the next instruction to be executed for this process.
6	**CPU registers** Various CPU registers where process need to be stored for execution for running state.
7	**CPU Scheduling Information** Process priority and other scheduling information which is required to schedule the process.
8	**Memory management information** This includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
9	**Accounting information** This includes the amount of CPU used for process execution, time limits, execution ID etc.
10	**IO status information** This includes a list of I/O devices allocated to the process.

The architecture of a PCB is completely dependent on Operating System and may contain different information in different operating systems. Here is a simplified diagram of a PCB −



PROCESS SCHEDULING:

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

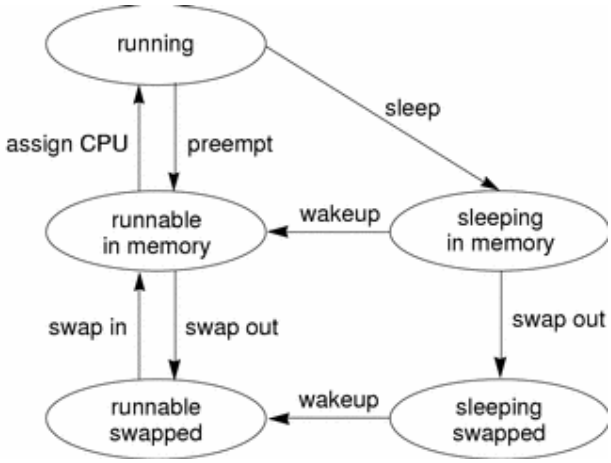
Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

3) Explain the process state transition diagram with examples>

Ans: Process State Transition

Applications that have strict real-time constraints might need to prevent processes from being swapped or paged out to secondary memory. A simplified overview of UNIX process states and the transitions between states is shown in the following figure.

Process State Transition Diagram



An active process is normally in one of the five states in the diagram. The arrows show how the process changes states.

- A process is running if the process is assigned to a CPU. A process is removed from the running state by the scheduler if a process with a higher priority becomes runnable. A process is also pre-empted if a process of equal priority is runnable when the original process consumes its entire time slice.
- A process is runnable in memory if the process is in primary memory and ready to run, but is not assigned to a CPU.
- A process is sleeping in memory if the process is in primary memory but is waiting for a specific event before continuing execution. For example, a process sleeps while waiting for an I/O operation to complete, for a locked resource to be unlocked, or for a timer to expire. When the event occurs, a wakeup call is sent to the process. If the reason for its sleep is gone, the process becomes runnable.

4) Discuss the attributes of the process. Describe the typical elements of process control block?

Ans: **Process Attributes** refer to a variety of descriptive characteristics and parameters of some process to explain how this process is defined and what its current status is. They are used in process management to plan, monitor, measure, and control various processes and their components.

Attributes of a process formulate the type and nature of this process, so they are basic characteristics that let explore how to best manage and operate the process.

Here're basic attributes of a typical process:

- Start time and Finish time
- Goal
- Duration
- Cost
- Requirements
- Milestones

- Performance
- Quality
- Scope

ELEMENTS OF PROCESS CONTROL BLOCK:

- **The process scheduling state**, e.g. in terms of "ready", "suspended", etc., and other scheduling information as well, like a priority value, the amount of time elapsed since the process gained control of the CPU or since it was suspended. Also, in case of a suspended process, event identification data must be recorded for the event the process is waiting for.
- **Process structuring information:** process's children id's, or the id's of other processes related to the current one in some functional way, which may be represented as a queue, a ring or other data structures.
- **Interprocess communication information:** various flags, signals and messages associated with the communication among independent processes may be stored in the PCB.
- **Process privileges,** in terms of allowed/disallowed access to system resources.
- **Process state:** State may enter into new, ready, running, waiting, dead depending on CPU scheduling.
- **Process No.:** a unique identification number for each process in the operating system.
- **Program counter:** a pointer to the address of the next instruction to be executed for this process.
- **CPU registers:** indicates various register set of CPU where process need to be stored for execution for running state.
- **CPU scheduling information**: indicates the information of a process with which it uses the CPU time through scheduling.
- **Memory management information:** includes the information of page table, memory limits, Segment table depending on memory used by the operating system.
- **Accounting information:** includes the amount of CPU used for process execution, time limits, execution ID etc.
- **IO status information:** includes a list of I/O devices allocated to the process.

5) Explain the principles of concurrency and the execution of concurrent processes with a simple example?

Ans: Concurrency is the interleaving of processes in time to give the appearance of simultaneous execution. Thus it differs from parallelism, which offers genuine simultaneous execution. However the issues and difficulties raised by the two overlap to a large extent:

1 • sharing global resources safely is difficult;

• optimal allocation of resources is difficult;

• locating programming errors can be difficult, because the contexts in which errors occur cannot always be reproduced easily.

Parallelism also introduces the issue that different processors may run at different speeds, but again this problem is mirrored in concurrency because different processes progress at different rates.

A Simple Example

The fundamental problem in concurrency is processes interfering with each other while accessing a shared global resource. This can be illustrated with a surprisingly simple example:

```
chin = getchar();

chout = chin;

putchar(chout);
```

Imagine two processes P1 and P2 both executing this code at the "same" time, with the following interleaving due to multi-programming.

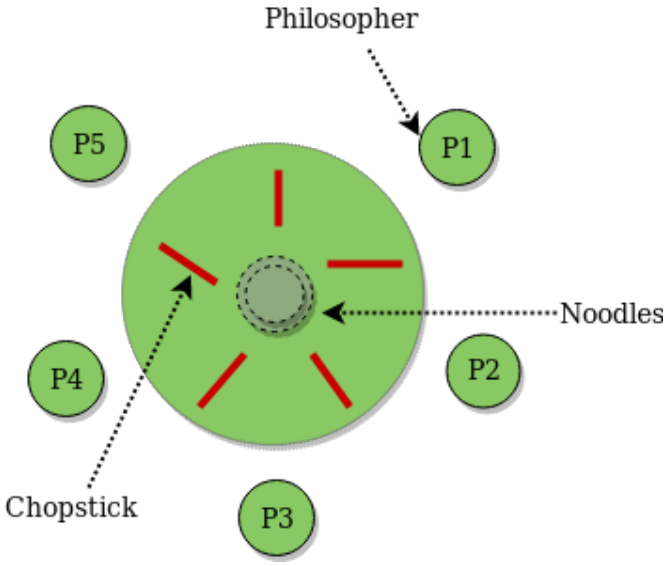
1. P1 enters this code, but is interrupted after reading the character x into chin.

2. P2 enters this code, and runs it to completion, reading and displaying the character y.

3. P1 is resumed, but chin now contains the character y, so P1 displays the wrong character. The essence of the problem is the shared global variable chin. P1 sets chin, but this write is subsequently lost during the execution of P2. The general solution is to allow only one process at a time to enter the code that accesses chin: such code is often called a critical section. When one process is inside a critical section of code, other processes must be prevented from entering that section. This requirement is known as **mutual exclusion**.

6) Describe dining-philosophers problem? Devise an algorithm to solve the problem using semaphores?

Ans: **The Dining Philosopher Problem –** The Dining Philosopher Problem states that K philosophers seated around a circular table with one chopstick between each pair of philosophers. There is one chopstick between each philosopher. A philosopher may eat if he can pickup the two chopsticks adjacent to him. One chopstick may be picked up by any one of its adjacent followers but not both.



Semaphore Solution to Dining Philosopher –
Each philosopher is represented by the following pseudocode:

```
process P[i]
 while true do
  {  THINK;
    PICKUP(CHOPSTICK[i], CHOPSTICK[i+1 mod 5]);
    EAT;
    PUTDOWN(CHOPSTICK[i], CHOPSTICK[i+1 mod 5])
  }
```

There are three states of philosopher : **THINKING, HUNGRY and EATING**. Here there are two semaphores : Mutex and a semaphore array for the philosophers. Mutex is used such that no two philosophers may access the pickup or putdown at the same time. The array is used to control the behavior of each philosopher. But, semaphores can result in deadlock due to programming errors.

Code –

```
#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };
```

```
sem_t mutex;
sem_t S[N];

void test(int phnum)
{
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        // state that eating
        state[phnum] = EATING;

        sleep(2);

        printf("Philosopher %d takes fork %d and %d\n",
                    phnum + 1, LEFT + 1, phnum + 1);

        printf("Philosopher %d is Eating\n", phnum + 1);

        // sem_post(&S[phnum]) has no effect
        // during takefork
        // used to wake up hungry philosophers
        // during putfork
        sem_post(&S[phnum]);
    }
}

// take up chopsticks
void take_fork(int phnum)
{

    sem_wait(&mutex);

    // state that hungry
    state[phnum] = HUNGRY;

    printf("Philosopher %d is Hungry\n", phnum + 1);

    // eat if neighbours are not eating
    test(phnum);

    sem_post(&mutex);

    // if unable to eat wait to be signalled
    sem_wait(&S[phnum]);

    sleep(1);
}

// put down chopsticks
```

```
void put_fork(int phnum)
{

    sem_wait(&mutex);

    // state that thinking
    state[phnum] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n",
           phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
    test(RIGHT);

    sem_post(&mutex);
}

void* philospher(void* num)
{

    while (1) {

        int* i = num;

        sleep(1);

        take_fork(*i);

        sleep(0);

        put_fork(*i);
    }
}

int main()
{

    int i;
    pthread_t thread_id[N];

    // initialize the semaphores
    sem_init(&mutex, 0, 1);

    for (i = 0; i < N; i++)

        sem_init(&S[i], 0, 0);

    for (i = 0; i < N; i++) {
```

```
        // create philosopher processes
        pthread_create(&thread_id[i], NULL,
                philospher, &phil[i]);

        printf("Philosopher %d is thinking\n", i + 1);
    }

    for (i = 0; i < N; i++)

        pthread_join(thread_id[i], NULL);
}
```

7) Explain the infinite buffer producer/consumer problem for concurrent processing which uses binary semaphores?

Ans: In computing, the **producer–consumer problem** (also known as the **bounded-buffer problem**) is a classic example of a multi-process synchronization problem. The problem describes two processes, the producer and the consumer, who share a common, fixed-size buffer used as a queue. The producer's job is to generate data, put it into the buffer, and start again. At the same time, the consumer is consuming the data (i.e., removing it from the buffer), one piece at a time. The problem is to make sure that the producer won't try to add data into the buffer if it's full and that the consumer won't try to remove data from an empty buffer.

The solution for the producer is to either go to sleep or discard data if the buffer is full. The next time the consumer removes an item from the buffer, it notifies the producer, who starts to fill the buffer again. In the same way, the consumer can go to sleep if it finds the buffer to be empty. The next time the producer puts data into the buffer, it wakes up the sleeping consumer.

The solution can be reached by means of inter-process communication, typically using semaphores. An inadequate solution could result in a deadlock where both processes are waiting to be awakened. The problem can also be generalized to have multiple producers and consumers.

Inadequate implementation[edit]

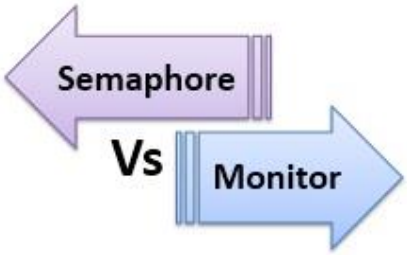
To solve the problem, some programmer might come up with a solution shown below. In the solution two library routines are used, `sleep` and `wakeup`. When sleep is called, the caller is blocked until another process wakes it up by using the wakeup routine. The global variable `itemCount` holds the number of items in the buffer.

```
int itemCount = 0;

procedure producer()
{
    while (true)
    {
        item = produceItem();
```

```
        if (itemCount == BUFFER_SIZE)
        {
            sleep();
        }

        putItemIntoBuffer(item);
        itemCount = itemCount + 1;

        if (itemCount == 1)
        {
            wakeup(consumer);
        }
    }
}

procedure consumer()
{
    while (true)
    {

        if (itemCount == 0)
        {
            sleep();
        }

        item = removeItemFromBuffer();
        itemCount = itemCount - 1;

        if (itemCount == BUFFER_SIZE - 1)
        {
            wakeup(producer);
        }

        consumeItem(item);
    }
}
```

8) Define monitor? Distinguish between monitor and semaphore. Explain in detail a monitor with notify and broadcast functions using an example?

Ans: A **monitor** is a synchronization construct that allows threads to have both mutual exclusion and the ability to wait (block) for a certain condition to become true. **Monitors** also have a mechanism for signaling other threads that their condition has been met.



Semaphore and Monitor both allow processes to access the shared resources in mutual exclusion. Both are the process synchronization tool. Instead, they are very different from each other. Where **Synchronization** is an integer variable which can be operated only by wait() and signal() operation apart from the initialization. On the other hand, the **Monitor** type is an abstract data type whose construct allow one process to get activate at one time. In this article, we will discuss the differences between semaphore and monitor with the help of comparison chart shown below.

BASIS FOR COMPARISON	SEMAPHORE	MONITOR
Basic	Semaphores is an integer variable S.	Monitor is an abstract data type.
Action	The value of Semaphore S indicates the number of shared resources availabe in the system	The Monitor type contains shared variables and the set of procedures that operate on the shared variable.
Access	When any process access the shared resources it perform wait() operation on S and when it releases the shared	When any process wants to access the shared variables in

BASIS FOR COMPARISON	SEMAPHORE	MONITOR
	resources it performs signal() operation on S.	the monitor, it needs to access it through the procedures.
Condition variable	Semaphore does not have condition variables.	Monitor has condition variables.

9) List out the various process states and briefly explain the same with a state diagram?

Ans: The various States of the Process are as Followings:-

1) New State : When a user request for a Service from the System , then the System will first initialize the process or the System will call it an initial Process . So Every new Operation which is Requested to the System is known as the New Born Process.

2) Running State : When the Process is Running under the CPU, or When the Program is Executed by the CPU , then this is called as the Running process and when a process is Running then this will also provides us Some Outputs on the Screen.

3) Waiting : When a Process is Waiting for Some Input and Output Operations then this is called as the Waiting State. And in this process is not under the Execution instead the Process is Stored out of Memory and when the user will provide the input then this will Again be on ready State.

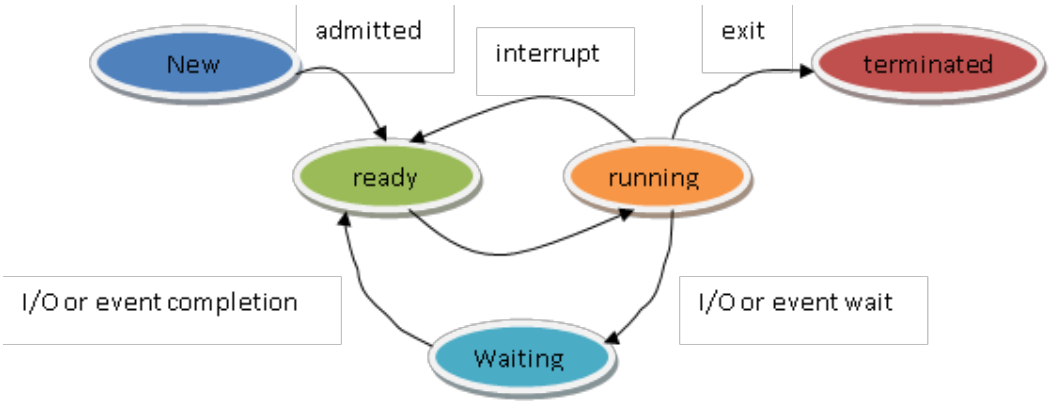
4) Ready State : When the Process is Ready to Execute but it is waiting for the CPU to Execute then this is called as the Ready State. After the Completion of the Input and outputs the Process will be on Ready State means the Process will Wait for the Processor to Execute.

5) Terminated State : After the Completion of the Process , the Process will be Automatically terminated by the CPU . So this is also called as the Terminated State of the Process. After Executing the Whole Process the Processor will Also deallocate the Memory which is allocated to the Process. So this is called as the Terminated Process.

As we know that there are many processes those are running at a Time, this is not true. **A processor can execute only one Process at a Time**. There are the various States of the Processes those determined which Process will be executed. The Processor will Execute all the processes by using the States of the Processes, the Processes those are on the Waiting State

will not be executed and CPU will Also divides his time for Execution if there are Many Processes those are Ready to Execute.

When a Process Change his State from one State to Another, then this is also called as the **Process State Transition**. In this a Running Process may goes on Wait and a ready Process may goes on the Wait State and the Wait State can be goes on the Running State.



10) a) Describe process scheduling? Explain the various levels of scheduling.

Ans:

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

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

There are two levels of scheduling as followed:

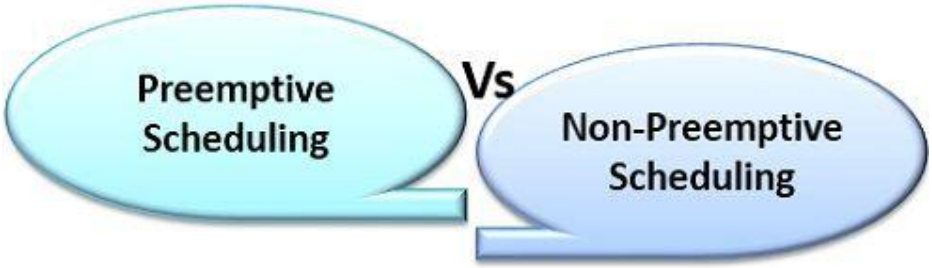
High Level Scheduling:

- High level scheduling is when a computer system chooses which jobs, tasks or requests to process.
- In this model, tasks or requests are prioritized and scheduled to complete based on the maximum amount of work or tasks the system can handle at once.
- Tasks for which the system does not have enough bandwidth to address are scheduled for later when other tasks are completed and free up the required bandwidth for the specific tasks.
- High level scheduling is also sometimes called "long-term scheduling."

Low Level Scheduling:

- Low level scheduling is when a system actually assigns a processor to a task that is ready to be worked on.
- In other words, in low level scheduling, a system assigns a specific component or internal processor to a specific task based on priority level, required bandwidth and available bandwidth.
- Low level scheduling determines which tasks will be addressed and in what order.
- These tasks have already been approved to be worked on, so low level scheduling is more detail oriented than other levels.

b) Distinguish pre-emptive and non-pre-emptive scheduling algorithms?



Ans:

BASIS FOR COMPARISON	PREEMPTIVE SCHEDULING	NON PREEMPTIVE SCHEDULING
Basic	The resources are allocated to a process for a limited time.	Once resources are allocated to a process, the process holds it till it completes its burst time or switches to waiting state.
Interrupt	Process can be interrupted in between.	Process can not be interrupted till it terminates or switches to waiting state.
Starvation	If a high priority process frequently arrives in the ready queue, low priority process may starve.	If a process with long burst time is running CPU, then another process with less CPU burst time may starve.
Overhead	Preemptive scheduling has overheads of scheduling the processes.	Non-preemptive scheduling does not have overheads.
Flexibility	Preemptive scheduling is flexible.	Non-preemptive scheduling is rigid.
Cost	Preemptive scheduling is cost associated.	Non-preemptive scheduling is not cost associative.

11) Discuss about following? a) Process b) Components of process c) Program versus process d) Proces sstates

Ans: a) PROCESS:
It contains the program code and its current activity. Depending on the operating system (**OS**), a **process** may be made up of multiple threads of execution that execute instructions concurrently. ... Each CPU (core) executes a single task at a time.

b)

S.N.	Component & Description
1	**Stack** The process Stack contains the temporary data such as method/function parameters, return address and local variables.
2	**Heap** This is dynamically allocated memory to a process during its run time.
3	**Text** This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.
4	**Data** This section contains the global and static variables.

c) A **process** is an instance of a computer **program** that is being executed. It contains the **program** code and its current activity. Depending on the operating system (OS), a **process** may be made up of multiple threads of execution that execute instructions concurrently.

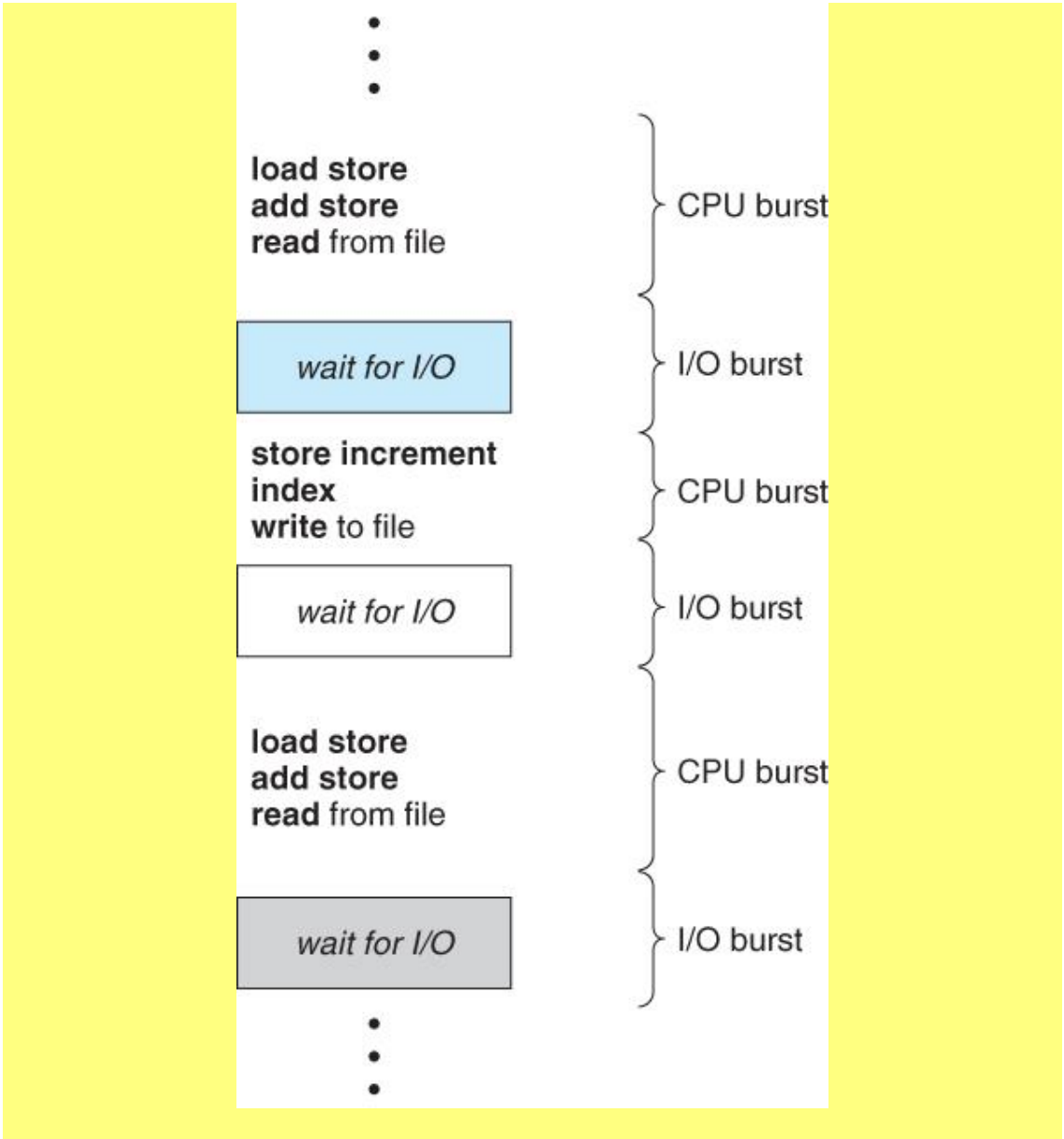
d) The current state of the process i.e., whether it is ready, running, waiting, or whatever is called as a process state.

12) Discuss the following? a) CPU-I/O burst cycle b) CPU schedule c) Pre-emptive and non-preemptive scheduling d) Dispatcher

Ans: a) Almost all processes alternate between two states in a continuing ***cycle***, as shown in Figure 6.1 below ***:***

A CPU burst of performing calculations, and

An I/O burst, waiting for data transfer in or out of the system.



b) **The scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them.**

- Dispatcher. ...
- Types of CPU Scheduling. ...
- Non-Preemptive Scheduling. ...
- Preemptive Scheduling. ...
- Scheduling Criteria. ...
- Scheduling Algorithms. ...
- First Come First Serve(FCFS) Scheduling. ...
- Shortest-Job-First(SJF) Scheduling.
-

c) **Preemptive scheduling:** The preemptive scheduling is prioritized. The highest priority process should always be the process that is currently utilized.Preemptive algorithms are driven by the notion of prioritized computation. The process with the highest priority should always be the one currently using the processor. If a process is currently using the processor and a new process with a higher priority enters, the ready list, the process on the processor should be removed and returned to the ready list until it is once again the highest-priority process in the system

Non-Preemptive scheduling: When a process enters the state of running, the state of that process is not deleted from the scheduler until it finishes its service time.Non-preemptive algorithms are designed so that once a process enters the running state(is allowed a process), it is not removed from the processor until it has completed its service time (or it explicitly yields the processor). context_switch() is called only when the process terminates or blocks

D) **Dispatcher:** The **dispatcher** is the module that gives control of the CPU to the process selected by the short-time scheduler(selects from among the processes that are ready to execute). The function involves : Switching context. Switching to user mode. Jumping to the proper location in the user program to restart that program.

13) Explain the concept of multi-threading? Discuss the following multithreading models. a) Many-to-one b) One-to-one c) Many-to-many d) Two-level

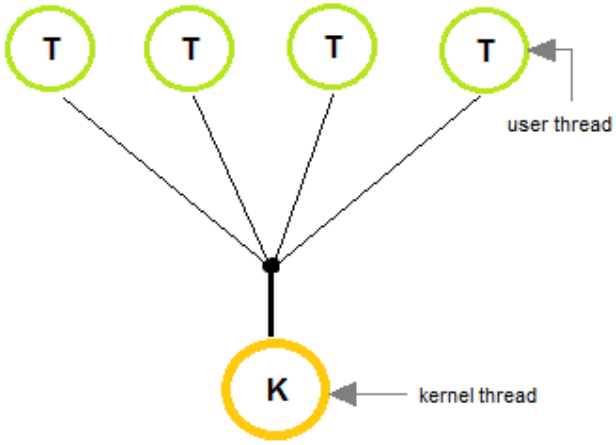
Ans: MULTITHREADING:

Multithreading is a type of execution model that allows **multiple** threads to exist within the context of a process such that they execute independently but share their process resources.

Many-To-One Model

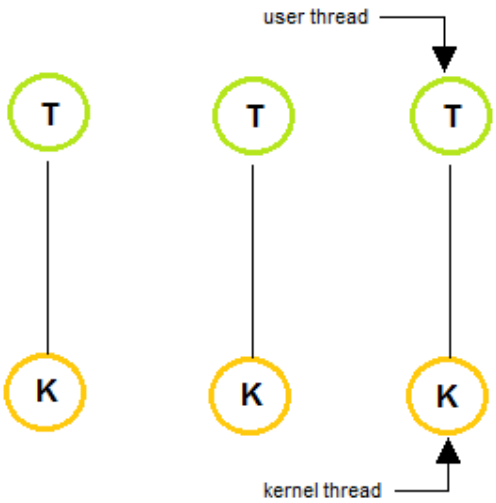
- In the many-to-one model, many user-level threads are all mapped onto a single kernel thread.

- Thread management is handled by the thread library in user space, which is efficient in nature.



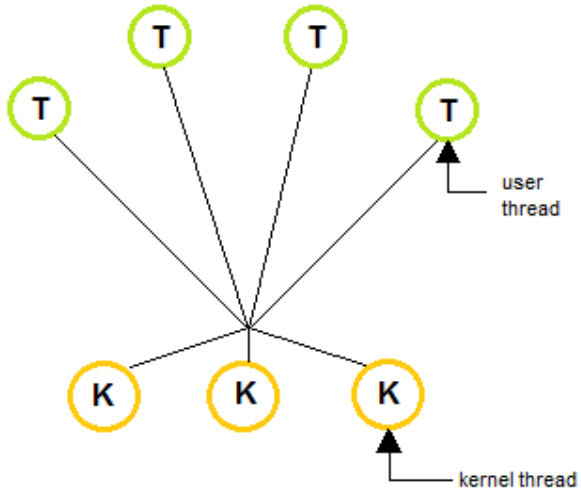
One-To-One Model

- The one-to-one model creates a separate kernel thread to handle each and every user thread.
- Most implementations of this model place a limit on how many threads can be created.
- Linux and Windows from 95 to XP implement the one-to-one model for threads.



Many-To-Many Model

- The many-to-many model multiplexes any number of user threads onto an equal or smaller number of kernel threads, combining the best features of the one-to-one and many-to-one models.
- Users can create any number of the threads.
- Blocking the kernel system calls does not block the entire process.
- Processes can be split across multiple processors.



TWO LEVEL:

There are two levels of threading

User level threads

Kernel level threads

14) Explain the issues that may rise in multi-threading programming. Discuss about each in detail?

1. **Thread Cancellation**.

   Thread cancellation means terminating a thread before it has finished working. There can be two approaches for this, one is **Asynchronous cancellation**, which terminates the target thread immediately. The other is **Deferred cancellation** allows the target thread to periodically check if it should be cancelled.

2. **Signal Handling**.

   Signals are used in UNIX systems to notify a process that a particular event has occurred. Now in when a Multithreaded process receives a signal, to which thread it must be delivered? It can be delivered to all, or a single thread.

3. **fork() System Call**.

   fork() is a system call executed in the kernel through which a process creates a copy of itself. Now the problem in Multithreaded process is, if one thread forks, will the entire process be copied or not?

4. **Security Issues** because of extensive sharing of resources between multiple threads.

15) Discuss the following CPU scheduling algorithms a) Round robin b) Multilevel- queue scheduling c) Multi-level feedback queue scheduling

Ans:

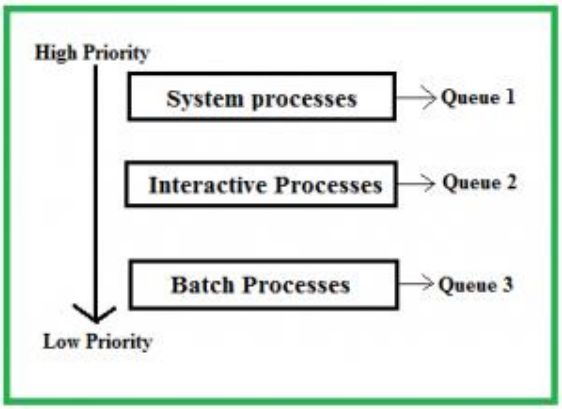
a) Round robin: **Round-robin** (RR) is one of the algorithms employed by process and network schedulers in computing.[1][2] As the term is generally used, time slices (also known as time quanta)[3] are assigned to each process in equal portions and in circular order, handling all processes without priority (also known as cyclic executive). Round-robin scheduling is simple, easy to implement, and starvation-free. Round-robin scheduling can also be applied to other scheduling problems, such as data packet scheduling in computer networks. It is an operating system concept.

The name of the algorithm comes from the round-robin principle known from other fields, where each person takes an equal share of something in turn.

b) 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.

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



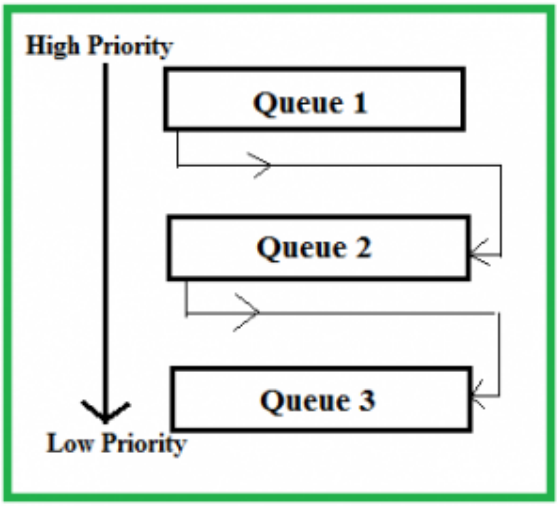
All three different type of processes have there own queue. Each queue have its own Scheduling algorithm. For example, queue 1 and queue 2 uses **Round Robin** while queue 3 can use **FCFS** to schedule there processes.

Scheduling among the queues : What will happen if all the queues have some processes? Which process should get the cpu? To determine this Scheduling among the queues is necessary. There are two ways to do so –

1. **Fixed priority preemptive scheduling method –** Each queue has absolute priority over lower priority queue. Let us consider following priority order **queue 1 > queue 2 > queue 3**.According to this algorithm no process in the batch queue(queue 3) can run unless queue 1 and 2 are empty. If any batch process (queue 3) is running and any system (queue 1) or Interactive process(queue 2) entered the ready queue the batch process is preempted.
2. **Time slicing** – In this method each queue gets certain portion of CPU time and can use it to schedule its own processes.For instance, queue 1 takes 50 percent of CPU time queue 2 takes 30 percent and queue 3 gets 20 percent of CPU time.

c) Multilevel feedback queue:

This Scheduling is like Multilevel Queue(MLQ) Scheduling but in this process can move between the queues. **Multilevel Feedback Queue Scheduling (MLFQ)** keep analyzing the behavior (time of execution) of processes and according to which it changes its priority. Now, look at the diagram and explanation below to understand it properly.



Now let us suppose that queue 1 and 2 follow round robin with time quantum 4 and 8 respectively and queue 3 follow FCFS.One implementation of MFQS is given below –

1. When a process starts executing then it first enters queue 1.
2. In queue 1 process executes for 4 unit and if it completes in this 4 unit or it gives CPU for I/O operation in this 4 unit than the priority of this process does not change and if it again comes in the ready queue than it again starts its execution in Queue 1.
3. If a process in queue 1 does not complete in 4 unit then its priority gets reduced and it shifted to queue 2.
4. Above points 2 and 3 are also true for queue 2 processes but the time quantum is 8 unit.In a general case if a process does not complete in a time quantum than it is shifted to the lower priority queue.
5. In the last queue, processes are scheduled in FCFS manner.
6. A process in lower priority queue can only execute only when higher priority queues are empty.
7. A process running in the lower priority queue is interrupted by a process arriving in the higher priority queue.

16) A scheduling mechanism should consider various scheduling criteria to realize the scheduling objectives? List out all the criteria

Ans: Scheduling Criteria

Scheduling criteria is also called as scheduling methodology. Key to multiprogramming is scheduling.Different CPU scheduling algorithm have different properties .The criteria used for comapring these algorithms include the following:

- **CPU Utilization:**

  Keep the CPU as busy as possible. It range from 0 to 100%. In practice, it range from 40 to 90%.

- **Throughput:**

  Throughput is the rate at which processes are completed per unit of time.

- **Turnaround time:**

  This is the how long a process takes to execute a process. It is calculated as the time gap between the submission of a process and its completion.

- **Waiting time:**

  Waiting time is the sum of the time periods spent in waiting in the ready queue.

- **Response time:**

  Response time is the time it takes to start responding from submission time.It is calculated as the amount of time it takes from when a request was submitted until the first response is produced.

- **Fairness:**

  Each process should have a fair share of CPU.

17) Define semaphore? Explain the method of application of semaphore for process synchronization?

Ans: In computer science, a **semaphore** is a variable or abstract data type used to control access to a common resource by multiple processes in a concurrent **system** such as a multiprogramming **operating system**.

Semaphores are commonly use for two purposes: to share a common memory space and to share access to files. Semaphores are one of the techniques for interprocess communication (IPC). The C programming language provides a set of interfaces or "functions" for managing semaphores.

The goal of this tutorial is explain how semaphores can be used to solved synchronization problems, which arise through cooperation between processes. The tutorial will start with the basics on creating and setting-up semaphores, then tackle the most basic use of semaphores, to protect *critical sections* of code. Finally, the *Bounded Buffer Problem*, described in the tutorial on **Cooperation**, will be tackled.

The Critical Section Problem

I assume you have read the tutorial on cooperating processes, and Interprocess Communication (IPC) facilities provided by the Operating System for process cooperation. The most synchronization problem confronting cooperating processes, is controling access to shared resource. Suppose two processes share access to a

file, or shared memory segment (or when we discuss threads in a couple of weeks, we'll see threads share the same memory, so they must synchronize their actions), and at least one of these processes can modify the data in this shared area of memory. That part of the code of each program, where one process is reading from or writing to a shared memory area, is a *critical section* of code, because we must ensure that only one process execute a critical section of code at a time. The Critical Section Problem is to design a protocol that the processes can use to coordinate their activities when one wants to enter its critical section of code.

Suppose we have two processes that share a memory segment of four bytes which stores an integer value. Let this value be named by the variable `v`. Process 1 (P1) and Process 2 (P2) have a section of code with the following lines:

```
if (0 < v)
  v--;
```

These lines of code will be part of a *critical section* of code, because it is important that each process be allowed to execute all of them without interference from the other process. Here is an example of how the processes can interfere with each other:

1. `v` has the value 1
2. P1 tests `0 < v`, which is true
3. P1 is removed from the CPU and replaced by P2
4. P2 tests `0 < v`, which is true
5. P2 executes `v--`, so `v` has the value 0
6. P1 given back the CPU and starts where it last left off: executing `v--`, so `v` has the value -1

By interfering with each other during the execution of the lines of code, the two processes caused the value of the common variable `v` to drop below 0, which they were trying to prevent from happening.

The protocol developed for solving the Critical Section Problem involved three steps:

1. **Entry section**: Before entering the critical section, a process must request permission to enter
2. **Critical section**: After permission is granted, a process may execute the code in the critical section. Other processes respect the request, and keep out of their critical sections.
3. **Exit section**: The process acknowledges it has left its critical section.

The problem that remains is how to effectively implement this protocol. How can the processes communicate their requests and grant their permissions so that only one process at a time is in a critical section.

The Solution

The Dutch scientist E.W. Dijkstra showed how to solve the *Critical Section Problem* in the mid-sixties, and introduced the concept of a semaphore to control synchronization. A *semaphore* is an integer variable which is accessed through through two *special* operations, called `wait` and `signal`. Why we need *special* operations will be discussed shortly. Originally, Dijkstra called the two special operations `P` (for *proberen*, the dutch word for test) and `V` (for *verhogen*, the dutch word to increment). Let `S` be an integer variable, our semaphore, then the classical definition of `wait` is given by

```
wait(S) {
  while(S ≤ 0);
  S--;
}
```

and `signal` is given by

```
signal(S) {
  S++;
}
```

We can use the semaphore to synchronize our processes:

```
// Entry Section
wait(S);

// Critical Section
if (0 < V)
   V--;

// Exit Section
signal(S);
```

1. Process P calls `wait(S)`
2. While `S` is 0, P waits.
3. When `S` is 1, P is allowed to enter its critical section
4. While P is in its critical section, the value of `S` is zero, blocking other processes from entering their critical section.
5. When P is finished and ready to leave its critical section, it executes `signal` resetting `S` to 1 and allowing another process to enter.

A semaphore which can take the value zero or one is called a *binary* semaphore, or *mutex*, for *mut*ually *ex*clusive.

18) Explain the Readers and Writers problem and its solution using the concept of semaphores?

Ans: Readers Writer Problem

Readers writer problem is another example of a classic synchronization problem. There are many variants of this problem, one of which is examined below.

Problem Statement:

There is a shared resource which should be accessed by multiple processes. There are two types of processes in this context. They are **reader** and **writer**. Any number of **readers** can read from the shared resource simultaneously, but only one **writer** can write to the shared resource. When a **writer**is writing data to the resource, no other process can access the resource. A **writer** cannot write to the resource if there are non zero number of readers accessing the resource.

Solution:

From the above problem statement, it is evident that readers have higher priority than writer. If a writer wants to write to the resource, it must wait until there are no readers currently accessing that resource.

Here, we use one mutex **m** and a semaphore **w**. An integer variable **read_count** is used to maintain the number of readers currently accessing the resource. The variable **read_count** is initialized to 0. A value of 1 is given initially to **m** and **w**.

Instead of having the process to acquire lock on the shared resource, we use the mutex **m** to make the process to acquire and release lock whenever it is updating the **read_count** variable.

The code for the writer process looks like this:

```
while(TRUE) {
  wait(w);
  /*perform the
write operation */
  signal(w);
}
```

The code for the reader process looks like this:

```
while(TRUE) {
  wait(m);   //acquire lock
  read_count++;
  if(read_count == 1)
        wait(w);
  signal(m);  //release lock
  /* perform the
    reading operation */
  wait(m);   // acquire lock
  read_count--;
  if(read_count == 0)
        signal(w);
  signal(m);  // release lock
}
```

Code Explained:

- As seen above in the code for the writer, the writer just waits on the **w** semaphore until it gets a chance to write to the resource.
- After performing the write operation, it increments **w** so that the next writer can access the resource.
- On the other hand, in the code for the reader, the lock is acquired whenever the **read_count** is updated by a process.
- When a reader wants to access the resource, first it increments the **read_count** value, then accesses the resource and then decrements the **read_count** value.
- The semaphore **w** is used by the first reader which enters the critical section and the last reader which exits the critical section.
- The reason for this is, when the first readers enters the critical section, the writer is blocked from the resource. Only new readers can access the resource now.

- Similarly, when the last reader exits the critical section, it signals the writer using the **w** semaphore because there are zero readers now and a writer can have the chance to access the resource.

---

19) Explain the uses of the following: a. Mutex object b. Semaphore object c. Waitable timer object

Ans: a) A **mutex object** is a synchronization **object** whose state is set to signaled when it is not owned by any thread, and nonsignaled when it is owned. ... After writing to the shared memory, the thread releases the **mutex object**. A thread **uses** the CreateMutex or CreateMutexEx function to create a **mutex object**.

b) A semaphore is an interprocess synchronization object that maintains a count between zero and a maximum value.
The object state is signaled when its count is greater than zero, and nonsignaled when its count is zero.
A semaphore is used as a counting resource gate, limiting the use of a resource by counting threads as they pass in and out of the gate.
Each time a waiting thread is released because of the signaled state of a semaphore, the count of a semaphore is decremented.
Use the CreateSemaphore function to create a named or unnamed semaphore object.
c) A *waitable timer object* is a synchronization object whose state is set to signaled when the specified due time arrives. There are two types of waitable timers that can be created: manual-reset and synchronization. A timer of either type can also be a periodic timer.

Object	Description
manual-reset timer	A timer whose state remains signaled until **SetWaitableTimer** is called to establish a new due time.
synchronization timer	A timer whose state remains signaled until a thread completes a wait operation on the timer object.
periodic timer	A timer that is reactivated each time the specified period expires, until the timer is reset or canceled. A periodic timer is either a periodic manual-reset timer or a periodic synchronization timer.

20) Write short notes about the following: a. Binary Semaphores b. Bounded Waiting

Ans: Binary Semaphores:

Semaphores which allow an arbitrary resource count are called counting **semaphores**, while **semaphores** which are restricted to the values 0 and 1 (or locked/unlocked, unavailable/available) are called **binary semaphores** and are used to implement locks.

Binary semaphores are sometimes easier to implement. than counting semaphores. We will not describe implementations of binary semaphores in terms of low-level or OS constructs (these would be similar to the implementations of counting semaphores). Instead we show how counting semaphores can be implemented by binary semaphores, which demonstrates that binary sempahores are as powerful as counting semaphores.

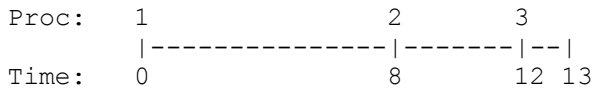
Bounded waiting:

There exists a **bound**, or limit, on the number of times other processes are allowed to enter their critical sections after a process has made request to enter its critical section and before that request is granted.

Process	Arrival Time	Burst Time
P1	0.0	8
P2	0.4	4
P3	1.0	1

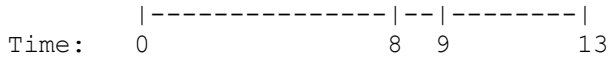
Sol:

FCFS Gantt Chart



Average Turnaround Time: ((8-0)+(12-0.4)+(13-1.0)) / 3 = 10.53

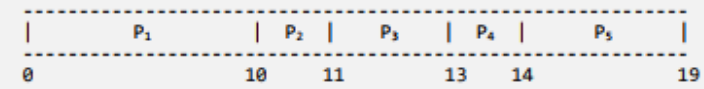
SJF Gantt Chart



Average Turnaround Time: ((8-0)+(13-0.4)+(9-1.0)) / 3 = 9.53

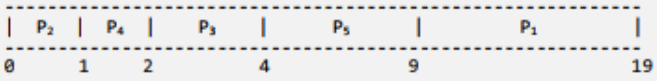
Consider the following set of processes, with the length of the CPU burst given in milliseconds: Process Burst Time Priority P1 10 3 P2 1 1 P3 2 3 P4 1 4 P5 5 2 The processes are assumed to have arrived in the order P1, P2, P3, P4, P5 all at time 0. a. Draw four Gantt charts that illustrate the execution of these processes using the following scheduling algorithms: FCFS, SJF, nonpreemptive priority (a smaller priority number implies a higher priority), and RR (quantum = 1). b. What is the turnaround time of each process for each of the scheduling algorithms in part a? c. What is the waiting time of each process for each of these scheduling algorithm?

Answer: 1. FCFS:



Performance statistics:

Process	*Arrival Time*	*Burst Time*	*Priority*	*Finish Time*	*Turnaround Time*	*Waiting Time*
P_1	0	10	3	10	10	0
P_2	0	1	1	11	11	10
P_3	0	2	3	13	13	11
P_4	0	1	4	14	14	13
P_5	0	5	2	19	19	14
Average					13.4	9.6

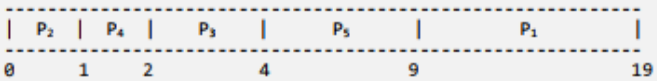


Performance statistics:

Process	*Arrival Time*	*Burst Time*	*Priority*	*Finish Time*	*Turnaround Time*	*Waiting Time*
P_1	0	10	3	19	19	9
P_2	0	1	1	1	1	0
P_3	0	2	3	4	4	2
P_4	0	1	4	2	2	1
P_5	0	5	2	9	9	4
Average					7	3.2

3. SJF (preemptive, Shortest-next-CPU-burst):

In this case the processes all arrive at the same time. This method turned out to be the same as nonpreemptive SJF.



Performance statistics:

Process	*Arrival Time*	*Burst Time*	*Priority*	*Finish Time*	*Turnaround Time*	*Waiting Time*
P_1	0	10	3	19	19	9
P_2	0	1	1	1	1	0
P_3	0	2	3	4	4	2
P_4	0	1	4	2	2	1
P_5	0	5	2	9	9	4
Average					7	3.2

1) **Consider three processes (process id 0, 1, 2 respectively) with compute time bursts 2, 4 and 8 time units. All processes arrive at time zero. Consider the longest remaining time first (LRTF) scheduling algorithm. In LRTF ties are broken by giving priority to the process with the lowest process id. The average turn around time is:**

Sol:

Let the processes be p0, p1 and p2. These processes will be executed in following order.

```
2)   p2  p1  p2  p1  p2  p0  p1   p2   p0   p1   p2
3) 0   4   5   6   7   8   9   10    11   12   13   14
```

4) Turn around time of a process is total time between submission of the process and its completion.
   Turn around time of p0 = 12 (12-0)
   Turn around time of p1 = 13 (13-0)
   Turn around time of p2 = 14 (14-0)
5) Average turn around time is (12+13+14)/3 = 13.

Consider three CPU-intensive processes, which require 10, 20 and 30 time units and arrive at times 0, 2 and 6, respectively. How many context switches are needed if the operating system implements a shortest remaining time first scheduling algorithm? Do not count the context switches at time zero and at the end.

Let three process be P0, P1 and P2 with arrival times 0, 2 and 6 respectively and CPU burst times 10, 20 and 30 respectively. At time 0, P0 is the only available process so it runs. At time 2, P1 arrives, but P0 has the shortest remaining time, so it continues. At time 6, P2 arrives, but P0 has the shortest remaining time, so it continues. At time 10, P1 is scheduled as it is the shortest remaining time process. At time 30, P2 is scheduled. Only two context switches are needed. P0 to P1 and P1 to P2.

Explain the following process state transition diagram for a uniprocessor system, assume that there are always some processes in the ready state

Sol: life cycle of a thread.

Explain Four jobs to be executed on a single processor system arrive at time 0 in the order A, B, C, D. their burst CPU time requirements are 4, 1, 8, 1 time units respectively. The completion time of A under round robin scheduling with time slice of one time unit is?

Sol: Here is the sequence

| A | B | C | D | A | C | A | C | A | {Consider each block takes one time unit)

Completion time of A will be 9.

Explain Which scheduling algorithm allocates the CPU first to the process that requests the CPU first?

Sol: first come,first serve scheduling.