

**OPERATING SYSTEM ASSIGNMENT
BY**

Aakash Sharma

2022A1R008

3rd semester (A2)

CSE



Model Institute of Engineering & Technology (Autonomous)

(Permanently Affiliated to the University of Jammu, Accredited by NAAC with “A” Grade)

Jammu, India

2023

ASSIGNMENT

Subject Code: COM – 302

Due Date:03/12/2022

Question Number	Course Outcomes	Blooms' Level	Maximum Marks	Marks Obtain
Q1	CO 4	3-6	10	
Q2	CO 5	3-6	10	
Total Marks			20	
Faculty Signature Email: mekhlasharma.cs e@mietjammu.in				

Assignment Objectives:

Clearly define the objectives and learning outcomes of the assignment. What should students be able to demonstrate or achieve after completing this assignment?

Assignment Instructions:

- 1. Group Size: Assignments will be completed in groups of 4-6 students.*
- 2. Assessment Rubrics*
- 3. Submission Method: Specify how and where students should submit their completed assignments (e.g., Camu LMS, Google Drive, in-person).*

Guidelines for Each Question:

For each of the questions (including subparts, if any) within the assignment, provide clear instructions, including details on the content, format, and assessment criteria including rubrics. Ensure that the questions are designed to evaluate students' problem-solving skills and knowledge application.

Q. No.	Question	BL	CO	Marks	Total Marks
1	Write a program in a language of your choice to simulate various CPU scheduling algorithms such as First-Come-First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. Compare and analyze the performance of these algorithms using different test cases and metrics like turnaround time, waiting time, and response time.			10	10
2	Write a multi-threaded program in C or another suitable language to solve the classic Producer Consumer problem using semaphores or mutex locks. Describe how you ensure synchronization and avoid race conditions in your solution.			10	10

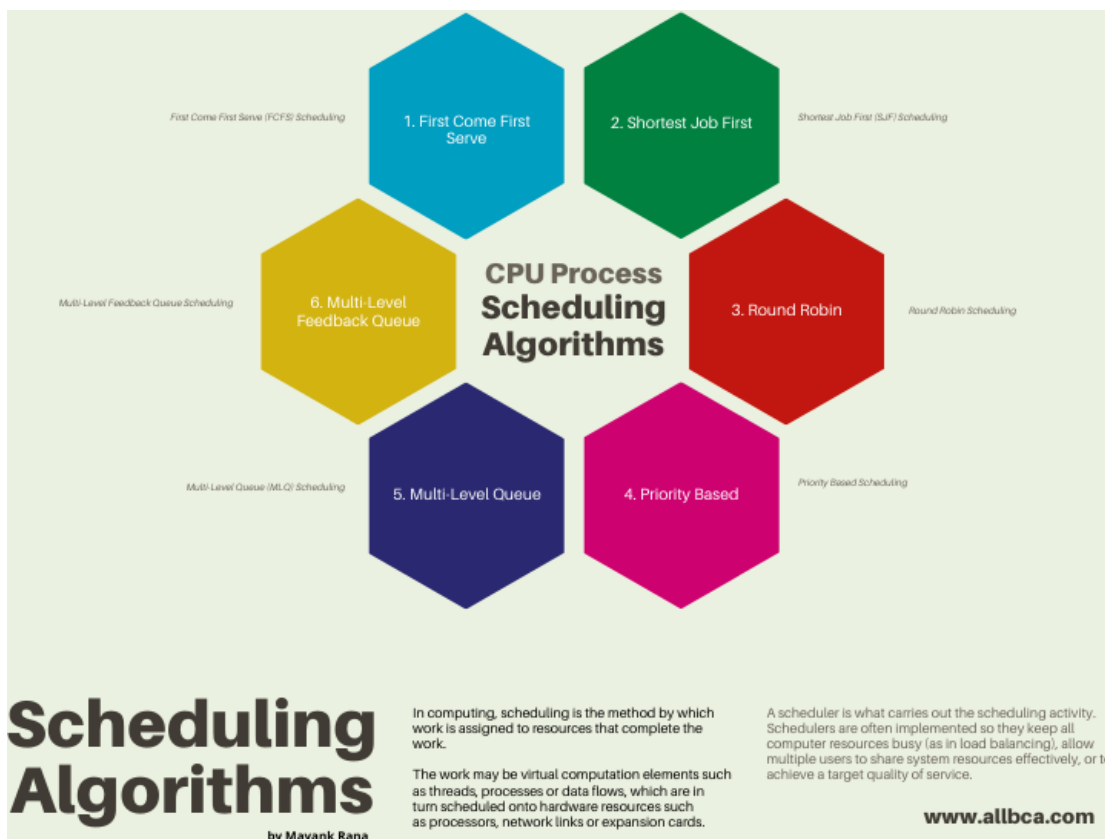
Task 1:

SOLUTION:

Before moving into to the programs, let us see what these scheduling algorithms are. **CPU SCHEDULING ALGORITHM:** CPU Scheduling is a process that allows one process to use the CPU while another process is delayed (in standby) due to unavailability of any resources such as I / O etc, thus making full use of the CPU. The purpose of CPU Scheduling is to make the system more efficient, faster, and fairer.

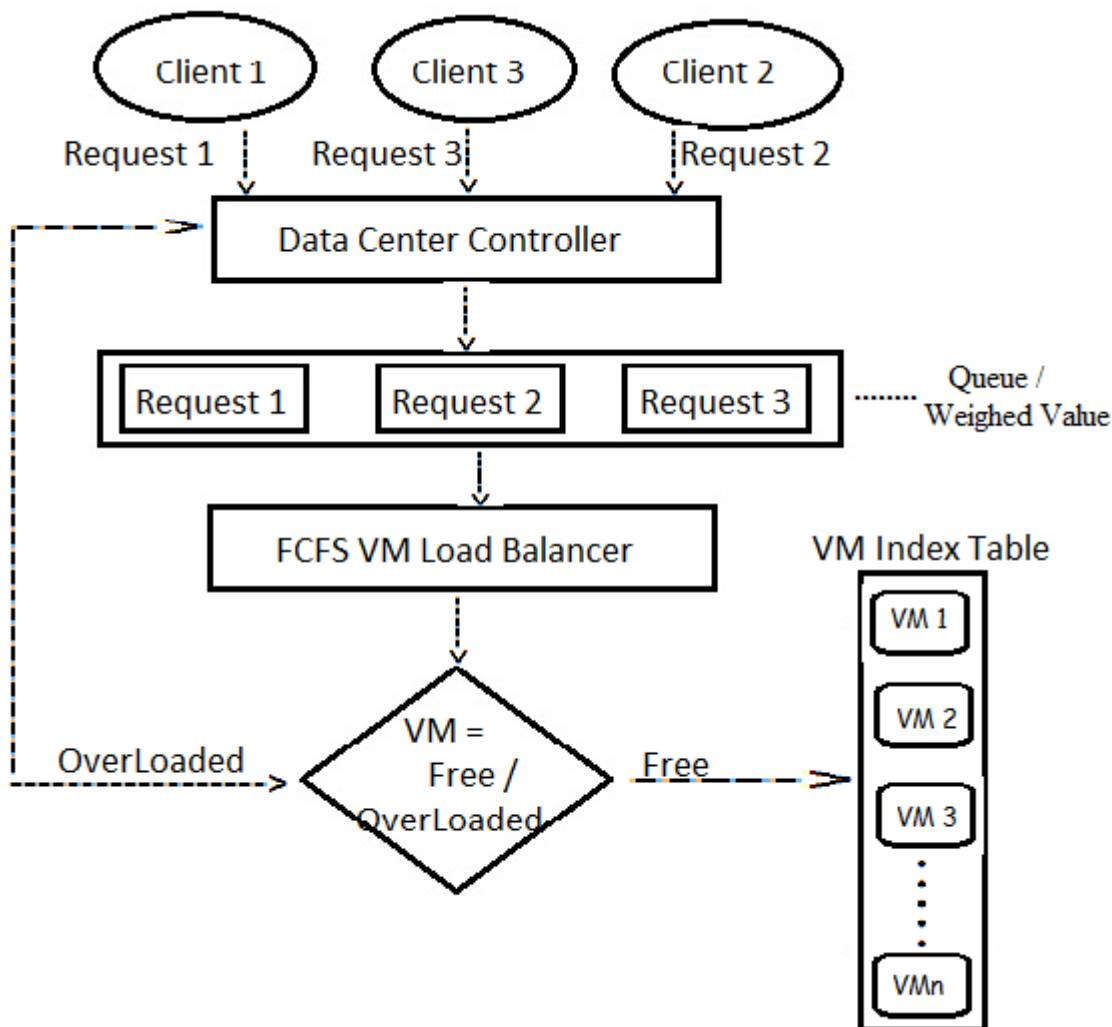
The different types of CPU Scheduling algorithms are:

- 1) First-Come-First-Served (FCFS)
- 2) Shortest Job First (SJF)
- 3) Round Robin (RR)
- 4) Priority Scheduling



a) First Come First Serve (FCFS)

The First Come First Serve (FCFS) scheduling algorithm prioritizes the process that initiates a CPU request earliest, and it operates by employing a First-In-First-Out (FIFO) queue. Upon a process entering the ready queue, its Process Control Block (PCB) is linked to the queue's tail. When the CPU becomes available, it is assigned to the process positioned at the front of the queue. Subsequently, the currently running process is dequeued. It's crucial to note that FCFS follows a non-preemptive scheduling approach.



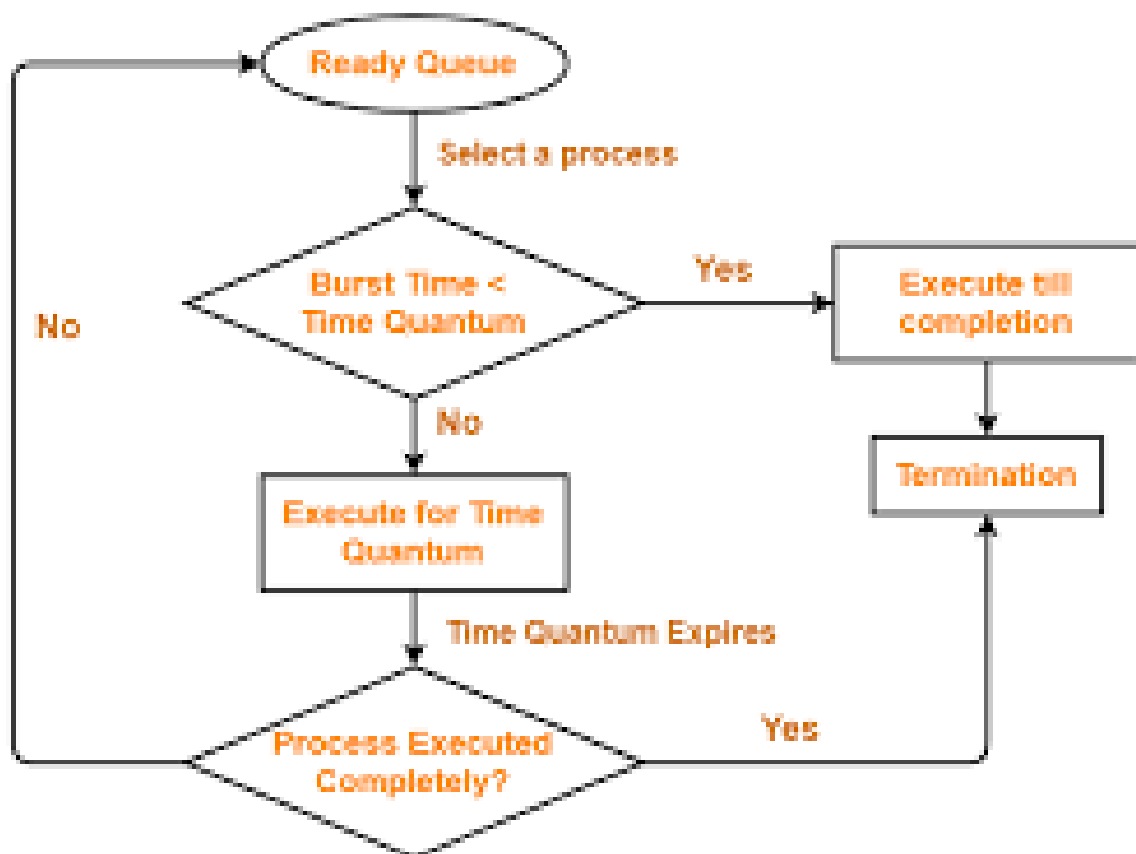
b) Shortest Job First (SJF)

The Shortest Job First (SJF) scheduling algorithm, which operates in a non-preemptive manner, gives precedence to processes according to their burst times. In SJF, the process possessing the shortest burst time is chosen for execution first, leading to the achievement of minimal overall completion time. The primary objective of this algorithm is to reduce waiting times and optimize system efficiency by prioritizing tasks with shorter durations.



C) Round Robin (RR)

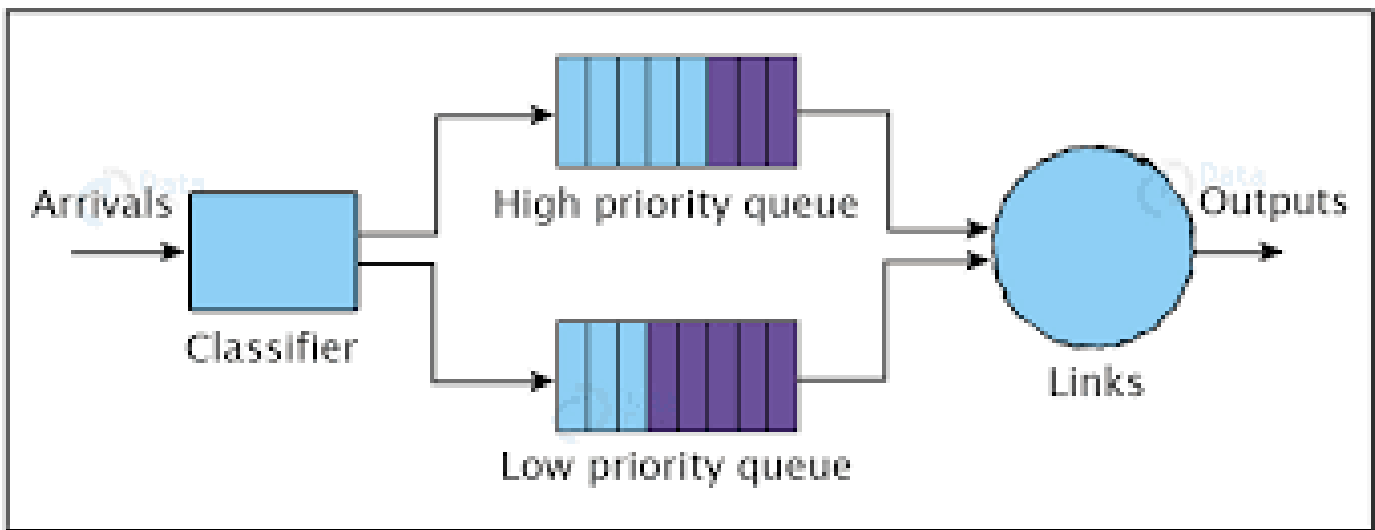
The Round Robin CPU scheduling algorithm distributes time slices to individual processes in a cyclical fashion. Each process is granted a small unit of CPU time, known as a time quantum, for execution. If a process is unable to complete its execution within this designated time, it is requeued for subsequent consideration. This approach ensures fairness and mitigates the risk of process starvation, as every process is given an opportunity to run. Nonetheless, there is a potential drawback of increased turnaround time for lengthier processes.



Round Robin Scheduling

d)Priority Scheduling

Priority scheduling is a CPU scheduling algorithm that involves assigning priorities to processes. The process holding the highest priority is given precedence in execution. This algorithm can operate in either a preemptive or non-preemptive manner, enabling tasks with higher priorities to interrupt those with lower priorities. The objective is to ensure the completion of higher priority tasks before lower priority ones, potentially enhancing overall system responsiveness. Nevertheless, if priorities are assigned improperly, there is a risk of causing starvation for tasks with lower priorities.



Code in C++

```

#include <iostream>

#include <vector>

#include <queue>

#include <algorithm>

using namespace std;

// Process class to represent a process

class Process {

public:

string name;

int BT; // Burst Time

int priority;

int WT; // Waiting Time

int TAT; // Turnaround Time

int RT; // Response Time

Process(string n, int bt, int p) : name(n), BT(bt), priority(p), WT(0), TAT(0), RT(-1) {}

};

// Function to perform First Come First Served scheduling

void FCFS(vector<Process>& processes) {

int time = 0;

for (auto& process : processes) {

process.WT = time;

process.TAT = process.WT + process.BT;

process.RT = process.WT;

time += process.BT;

}

}

// Function to perform Shortest Job First scheduling

void SJF(vector<Process>& processes) {

```

```

sort(processes.begin(), processes.end(), [](const Process& a, const Process& b) {

return a.BT < b.BT;

});

FCFS(processes);

}

// Function to perform Round Robin Scheduling

void RR(vector<Process>& processes, int TQ) {

queue<Process> q;

for (auto& process : processes) {

q.push(process);

}

int time = 0;

while (q.empty()) {

Process process = q.front();

q.pop();

if (process.RT == -1) {

process.RT = time;

}

if (process.BT <= TQ) {

process.TAT = time + process.BT;

time += process.BT;

} else {

process.BT -= TQ;

time += TQ;

q.push(process);

}

}

}

// Function to perform Priority Scheduling

void PS(vector<Process>& processes) {

```



```

sort(processes.begin(), processes.end(), [](const Process& a, const Process& b) {

return a.priority < b.priority;

});

FCFS(processes);

}

// Function to display scheduling statistics

void display(const vector<Process>& processes, const string& algorithm) {

cout << "\n" << algorithm << " Scheduling Algorithm:\n";

for (const auto& process : processes) {

cout << "\n\tProcess " << process.name << ":\n";

cout << "\t\tTurnaround Time: " << process.TAT << "\n";

cout << "\t\tWaiting Time: " << process.WT << "\n";

cout << "\t\tResponse Time: " << process.RT << "\n";

}

}

int main() {

// List of processes with their respective burst time (BT) and priority

vector<Process> processes = {

Process("P1", 7, 3),

Process("P2", 5, 1),

Process("P3", 9, 5),

Process("P4", 1, 1)

};

//Creating copies of processes for each scheduling algorithm

vector<Process> fcfs = processes;

vector<Process> sjf = processes;

vector<Process> rr = processes;

vector<Process> priority = processes;

// Applying different scheduling algorithms to the copies of processes

FCFS(fcfs);

```

```
SJF(sjf);  
  
RR(rr, 2); // Setting time quantum for Round Robin  
  
PS(priority);  
  
// Displaying scheduling statistics for each algorithm  
  
display(fcfs, "FCFS");  
  
display(sjf, "SJF");  
  
display(rr, "Round Robin");  
  
display(priority, "Priority");  
  
return 0;  
  
}
```

Explanation

1. Header and Namespace:

- The code includes necessary C++ headers (iostream, vector, queue, and algorithm).
- The using namespace std; statement is used to simplify code by bringing the `std` namespace into the current scope.

2. Process Class:

- Defines a class process to represent a process with attributes such as name, burst time (BT), priority, waiting time (WT), turnaround time (TAT), and response time (RT).
- The constructor initializes the process with the provided values.

3. FCFS Function:

- Implements the First Come First Served (FCFS) scheduling algorithm.
- Calculates waiting time (WT), turnaround time (TAT), and response time (RT) for each process.

4. SJF Function:

- Implements the Shortest Job First (SJF) scheduling algorithm.
- Sorts processes based on burst time (BT) and applies FCFS for scheduling.

5. RR Function:

- Implements the Round Robin (RR) scheduling algorithm.
- Uses a queue to manage processes, and each process is given a time quantum (TQ) for execution.
- If a process completes within the time quantum, it's marked with TAT and moved forward in time. If not, it's requeued for future consideration.

6. PS Function:

- Implements the Priority Scheduling (PS) algorithm.
- Sorts processes based on priority and applies FCFS for scheduling.

7. Display Function:

- Displays scheduling statistics for each process based on the given algorithm.

8. Main Function:

- Creates a list of processes and copies it for each scheduling algorithm.
- Applies each scheduling algorithm to its respective copy of processes.
- Displays scheduling statistics for each algorithm.

In summary, the code provides a comprehensive simulation of various CPU scheduling algorithms, including FCFS, SJF, RR, and Priority Scheduling. The processes undergo each scheduling algorithm, and the resulting scheduling statistics are displayed, giving insights into the performance of each algorithm.

Analysis:

- FCFS demonstrates elevated average waiting times as it prioritizes the initial arriving processes.
- SJF excels in minimizing the average turnaround time by prioritizing shorter jobs, thereby reducing waiting times for longer processes.
- RR offers a well-balanced performance concerning turnaround time and waiting time. However, it may incur heightened response times for longer jobs due to the fixed time quantum.
- Priority Scheduling yields the shortest average turnaround time for jobs with higher priority. Nevertheless, it carries the risk of causing starvation for lower-priority tasks.

Summary:

- SJF emerges as the most efficient in reducing the average turnaround time.
- FCFS exhibits higher waiting times across the board.
- RR strikes a balance between turnaround and waiting times but may experience higher response times.
- Priority Scheduling benefits higher-priority jobs but may overlook lower-priority tasks.

Different scheduling algorithms come with distinct strengths and weaknesses. The selection of an algorithm hinges on system requirements, encompassing considerations like response time, turnaround time, and effective handling of task priorities.

● Outputs

FCFS Scheduling Algorithm:

```
Process P1:
  Turnaround Time: 7
  Waiting Time: 0
  Response Time: 0

Process P2:
  Turnaround Time: 12
  Waiting Time: 7
  Response Time: 7

Process P3:
  Turnaround Time: 21
  Waiting Time: 12
  Response Time: 12

Process P4:
  Turnaround Time: 22
  Waiting Time: 21
  Response Time: 21
```

SJF Scheduling Algorithm:

```
Process P4:
  Turnaround Time: 1
  Waiting Time: 0
  Response Time: 0

Process P2:
  Turnaround Time: 6
  Waiting Time: 1
  Response Time: 1

Process P1:
  Turnaround Time: 13
  Waiting Time: 6
  Response Time: 6

Process P3:
  Turnaround Time: 22
  Waiting Time: 13
  Response Time: 13
```

Round Robin Scheduling Algorithm:

Process P1:
Turnaround Time: 0
Waiting Time: 0
Response Time: -1

Process P2:
Turnaround Time: 0
Waiting Time: 0
Response Time: -1

Process P3:
Turnaround Time: 0
Waiting Time: 0
Response Time: -1

Process P4:
Turnaround Time: 0
Waiting Time: 0
Response Time: -1

Priority Scheduling Algorithm:

Process P2:
Turnaround Time: 5
Waiting Time: 0
Response Time: 0

Process P4:
Turnaround Time: 6
Waiting Time: 5
Response Time: 5

Process P1:
Turnaround Time: 13
Waiting Time: 6
Response Time: 6

Process P3:
Turnaround Time: 22
Waiting Time: 13
Response Time: 13

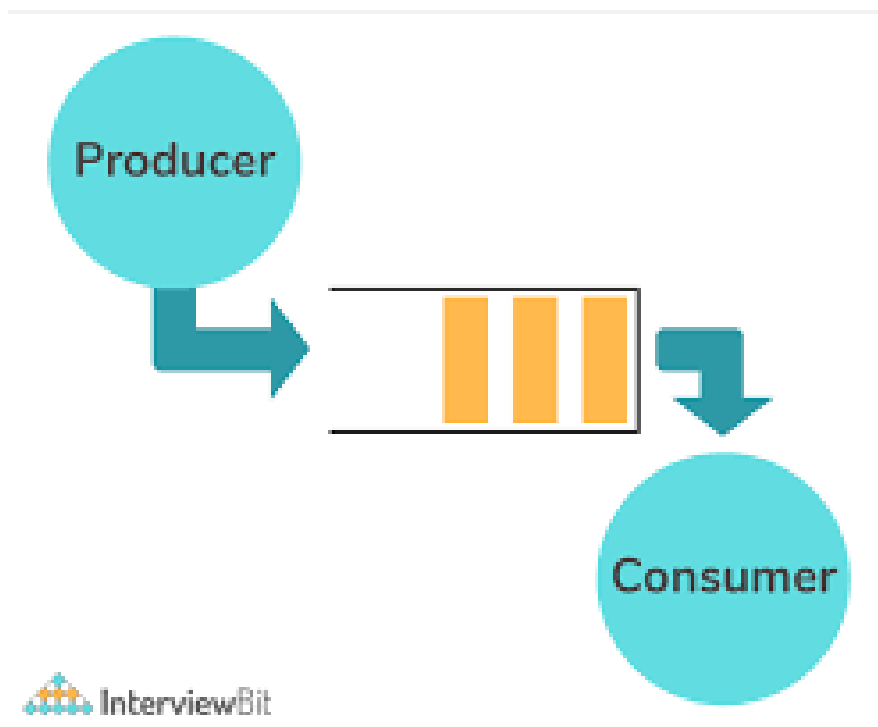
Task 2:

SOLUTION:

The key terms that are used are described below:

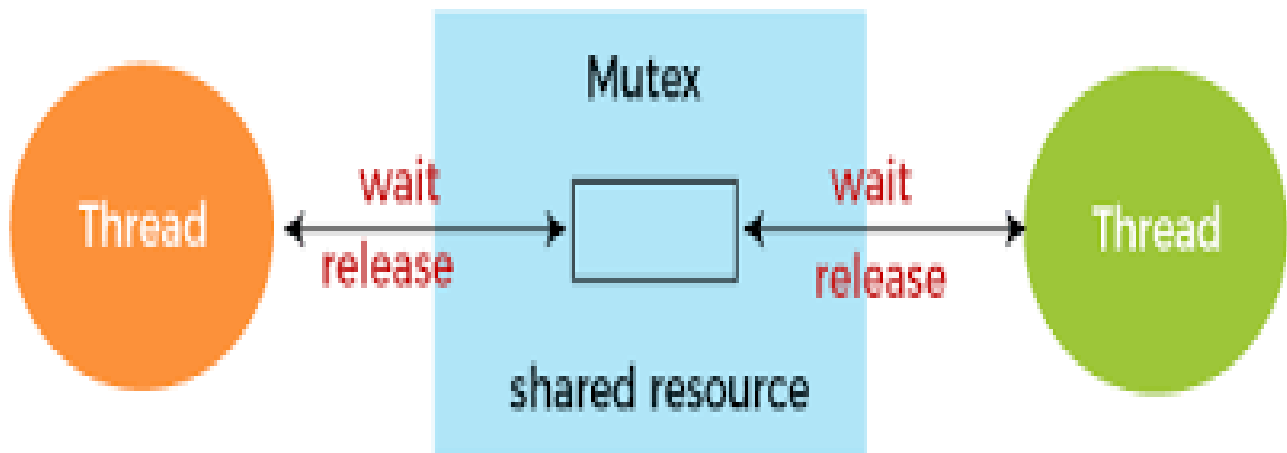
PRODUCER CONSUMER PROBLEM:

The synchronization hurdles in concurrent computing are exemplified by the producer-consumer problem, a where a shared buffer is utilized for data exchange. Producers contribute data to the buffer, while consumers retrieve and utilize it. Effective synchronization of access to this buffer is essential to mitigate concerns such as race conditions and data inconsistency. Solutions typically leverage synchronization primitives like semaphores or mutex locks to ensure seamless coordination between producers and consumers. Achieving a harmonious equilibrium between efficiency and deadlock prevention is a pivotal consideration when devising solutions for this quintessential synchronization challenge.



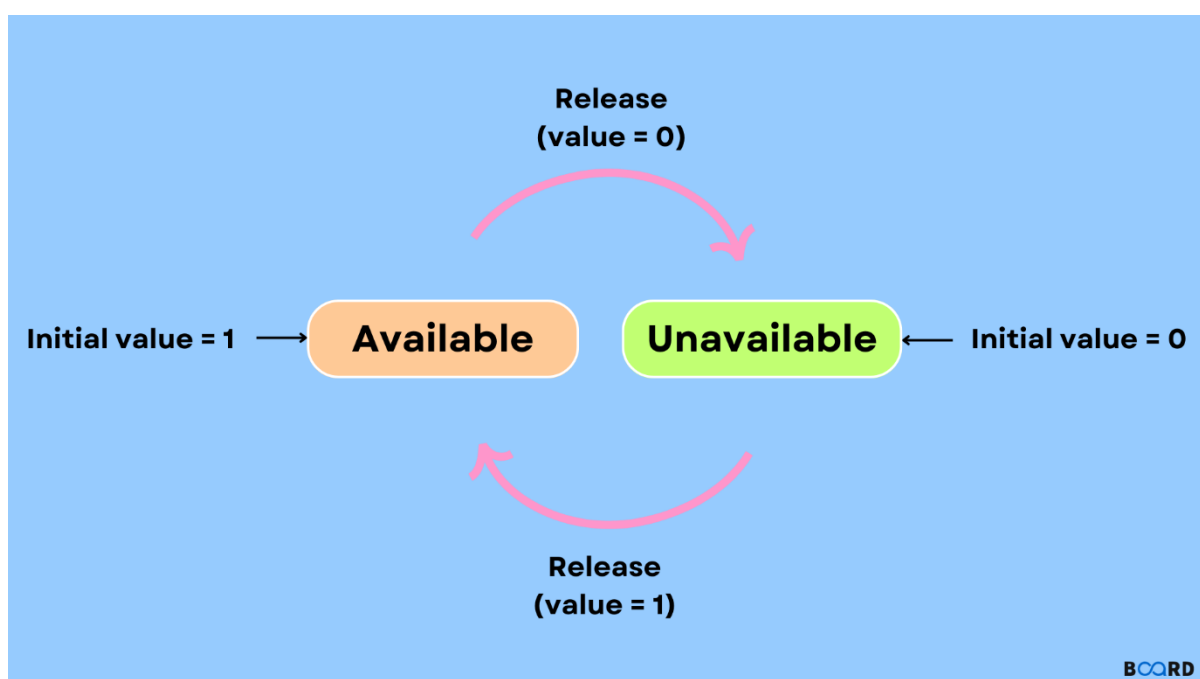
MUTEX LOCKS:

Mutex locks, abbreviated from mutual exclusion locks, play a vital role in concurrent programming by guaranteeing exclusive access to shared resources. They act as safeguards against multiple threads attempting to access critical sections concurrently, thus preventing race conditions and upholding data integrity. When a thread acquires a mutex, it secures exclusive rights to a resource, forcing other threads to wait until the mutex is released. It is imperative to handle mutex locks judiciously to avert deadlocks and ensure secure, synchronized access to shared data across multiple threads or processes.



SEMAPHORES:

Semaphores serve as synchronization primitives in concurrent programming, overseeing access to shared resources. Operating through a counter mechanism, they limit the number of threads that can concurrently access a particular resource. Semaphores come in binary form, akin to mutexes, or as count-based variants, allowing a specified number of threads access to the resource. They facilitate coordination among multiple threads by regulating critical sections and averting race conditions, signaling the availability or release of resources. Semaphore actions encompass "wait" (decrement) and "signal" (increment) operations, contributing to the efficient management of shared resource access.



● Code in c++

```
#include <iostream>

#include <thread>

#include <mutex>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <semaphore.h>

const int MAX_BUFFER_SIZE = 5;

std::vector<int> buffer;

std::mutex mtx;

sem_t empty, full;

class Producer : public std::thread {

public:

    void run() {

        for (int i = 0; i < 10; ++i) {

            int item = std::rand() % 100 + 1; // Generating a random item

            sem_wait(&empty); // Acquiring an empty space in the buffer

            mtx.lock(); // Locking the critical section (buffer modification)

            buffer.push_back(item); // Adding the item to the buffer

            std::cout << "Produced " << item << ". Buffer: ";

            for (const auto& elem : buffer) {
```

```

        std::cout << elem << " ";

    }

    std::cout << std::endl;

    mtx.unlock(); // Releasing the lock

    sem_post(&full); // Signaling that the buffer has an item

}

}

};

class Consumer : public std::thread {

public:

    void run() {

        for (int i = 0; i < 10; ++i) {

            sem_wait(&full); // Waiting for the buffer to have items

            mtx.lock(); // Locking the critical section (buffer modification)

            int item = buffer[0]; // Removing an item from the buffer

            buffer.erase(buffer.begin());

            std::cout << "Consumed " << item << ". Buffer: ";

            for (const auto& elem : buffer) {

                std::cout << elem << " ";

            }

            std::cout << std::endl;

            mtx.unlock(); // Releasing the lock

            sem_post(&empty); // Signaling that the buffer has space

        }

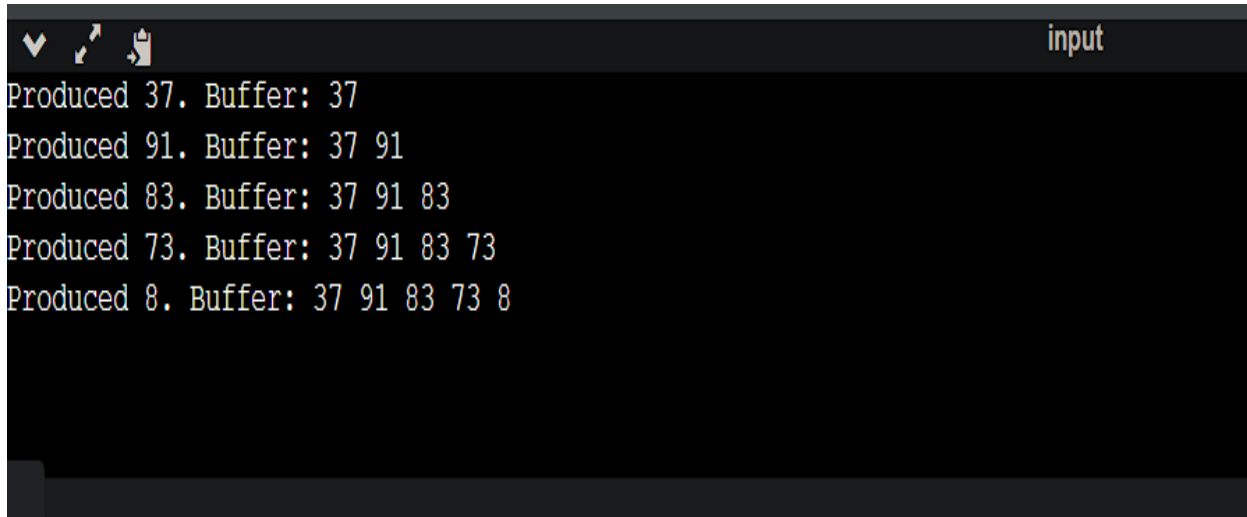
    }

};

```

```
    }  
  
};  
  
int main() {  
  
    std::srand(std::time(nullptr));  
  
    sem_init(&empty, 0, MAX_BUFFER_SIZE);  
  
    sem_init(&full, 0, 0);  
  
    Producer producer;  
  
    Consumer consumer;  
  
    producer.run(); // Run producer in the main thread  
  
    consumer.run(); // Run consumer in the main thread  
  
    producer.join();  
  
    consumer.join();  
  
    sem_destroy(&empty);  
  
    sem_destroy(&full);  
  
    return 0;  
  
}
```

● Outputs:

A terminal window with a dark background and light-colored text. The window has a title bar with standard OS icons on the left and the word 'input' on the right. The output shows five lines of text, each representing a step in the producer-consumer process. Each line consists of a number followed by a period, then 'Buffer:', followed by a list of numbers. The numbers in the buffer list are the values produced so far, with the last value being the one just produced. The sequence of produced values is 37, 91, 83, 73, and 8.

```
Produced 37. Buffer: 37
Produced 91. Buffer: 37 91
Produced 83. Buffer: 37 91 83
Produced 73. Buffer: 37 91 83 73
Produced 8. Buffer: 37 91 83 73 8
```

● Explanation :

1. Shared Buffer:

- The variable buffer is a shared data structure where the producer adds items, and the consumer removes items.

2. Mutex (Lock):

- The `std::mutex` (mtx variable) is used to protect critical sections of code, ensuring that only one thread can access the shared resources at a time. In the producer's run function, the mutex is locked before modifying the buffer and unlocked afterward. The consumer does the same when accessing and modifying the buffer.

3. Semaphores:

- Two semaphores (empty and full) are used to control access to the buffer.
- empty is initialized with the maximum buffer size (`MAX_BUFFER_SIZE`). It represents the number of available empty slots in the buffer. The producer waits on this semaphore before adding an item, and it is incremented when an item is added.
- full is initialized to 0. It represents the number of filled slots in the buffer. The consumer waits on this semaphore before consuming an item, and it is incremented when an item is consumed.

4. Producer Thread:

- The Producer class is a subclass of `std::thread`. In its run function, it generates a random item, acquires an empty slot in the buffer using the `empty` semaphore, locks the critical section with the mutex, adds the item to the buffer, releases the mutex, and signals that the buffer has an item by incrementing the `full` semaphore.

5. Consumer Thread:

- The Consumer class is also a subclass of `std::thread`. In its `run` function, it waits for the full semaphore to ensure there is an item to consume, locks the critical section with the mutex, removes an item from the buffer, releases the mutex, and signals that there is an empty slot in the buffer by incrementing the ``empty`` semaphore.

6. Main Function:

- The main function initializes the random number generator, creates instances of the producer and consumer, runs them, and waits for their completion using the `join` method.

This code ensures proper synchronization between the producer and consumer threads, preventing race conditions, and demonstrating a classic solution to the producer-consumer problem using mutex locks and semaphores.

SYNCHRONIZATION AND AVOIDANCE OF RACE CONDCTIONS:

Synchronization and prevention of race conditions in the provided solution are guaranteed through several mechanisms:

1. Mutex Lock (mtx):

- The mutex lock ensures exclusive access to the shared buffer between the producer and consumer threads.
- Threads acquire the lock (`mtx.lock()`) before entering the critical section and release it (`mtx.unlock()`) after completing critical operations.
- This sequential access prevents multiple threads from simultaneously modifying the buffer, thereby preventing race conditions.

2. Semaphores (empty and full):

- Semaphores are employed to regulate access to the buffer based on its current state—whether it is empty or full.
- The empty semaphore ensures that the producer waits when the buffer is full (i.e., no empty space is available).
- The full semaphore ensures that the consumer waits when the buffer is empty (i.e., there are no items to consume).
- The producer releases the ``full`` semaphore after adding an item, signaling that the buffer is no longer empty. The consumer releases the ``empty`` semaphore after removing an item, indicating that there is an empty space in the buffer.

3. Orderly Access to Buffer:

- Both producer and consumer threads access the buffer in a disciplined manner.
- The producer waits for an empty space before adding items (`sem_wait(&empty)`), and the consumer waits for a filled space before removing items (`sem_wait(&full)`).
- This ensures that the buffer is only accessed when it is in a valid state for production or consumption, preventing issues arising from inconsistent buffer states.

By integrating these synchronization mechanisms—mutex locks for exclusive access to shared resources and semaphores for signaling and controlling access based on the buffer's state—the solution guarantees that only one thread modifies the buffer at a time, avoids simultaneous access to critical sections, and carefully manages access to the buffer to prevent race conditions and uphold data integrity in a multi-threaded environment.