ASSIGNMENT

By

Gaurangi Peshin

2022A1R011

3rd Semester

CSE

Model Institute of Engineering & Technology (Autonomous)

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

Jammu, India

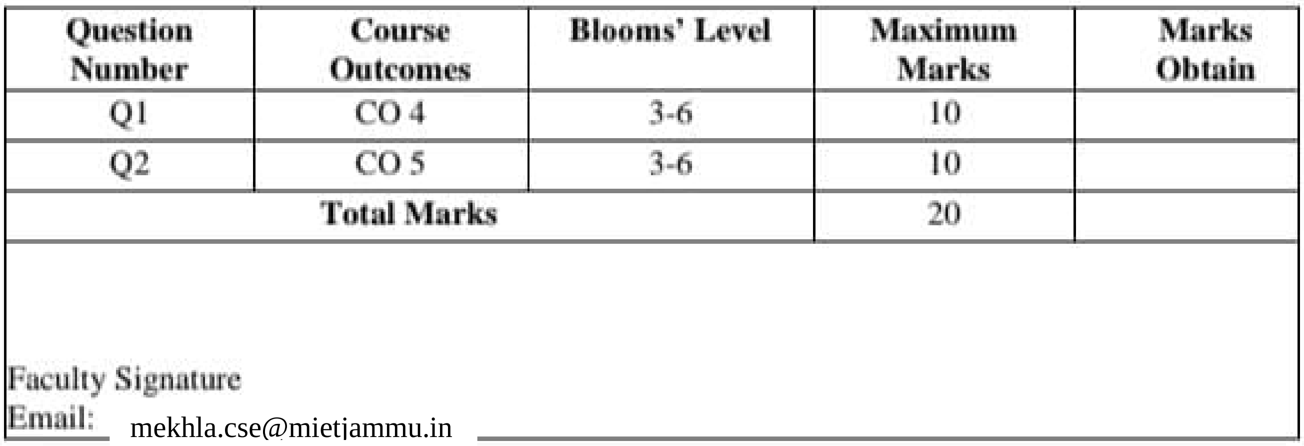
2023

Due Date: 30th November 2023

ASSIGNMENT

Subject Name: Operating Systems
Subject Code: COM-302

Submitted To:
Mrs. Mekhla Sharma
(mekhla.cse@mietjammu.in)



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: mekhla.cse@mietjammu.in

TASKS TO PERFORM

TASK 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.

TASK 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.

TASK-1

Develop a program in a language of your choice to simulate various CPU scheduling algorithms, including First-Come-First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. Conduct a thorough comparison and analysis of the performance of these algorithms using diverse test cases and metrics such as turnaround time, waiting time, and response time.

SOLUTION:

Before delving into the program implementations, it is essential to understand the nature of these scheduling algorithms.

CPU SCHEDULING ALGORITHM:

CPU Scheduling is a critical process that allows one process to utilize the CPU while another process is temporarily delayed (in standby) due to resource unavailability, such as I/O operations. This ensures optimal utilization of the CPU, making the system more efficient, faster, and fairer.

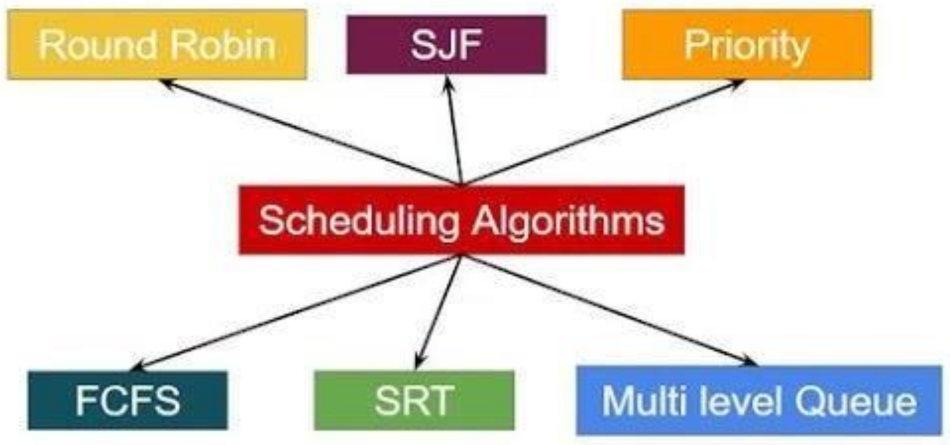
The main types of CPU Scheduling algorithms include :

1) First-Come-First-Served (FCFS)

2) Shortest Job First (SJF)

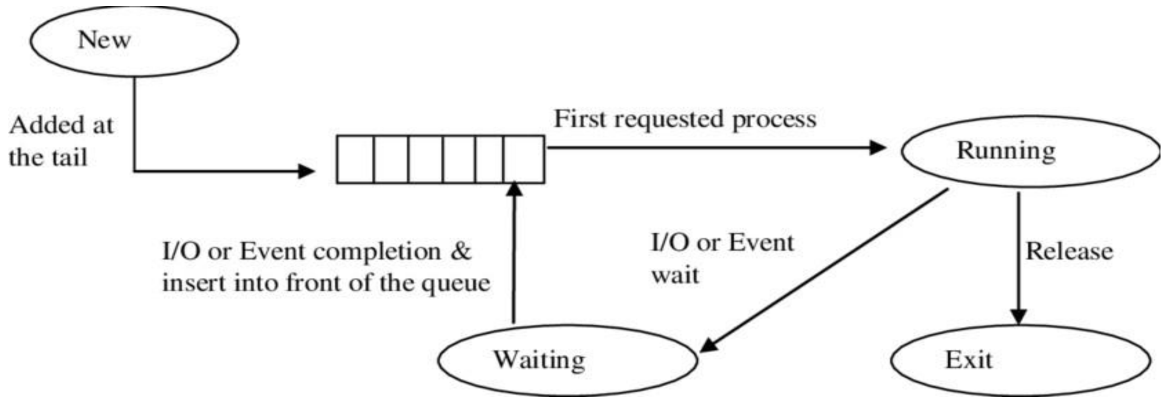
3) Round Robin (RR)

4) Priority Scheduling



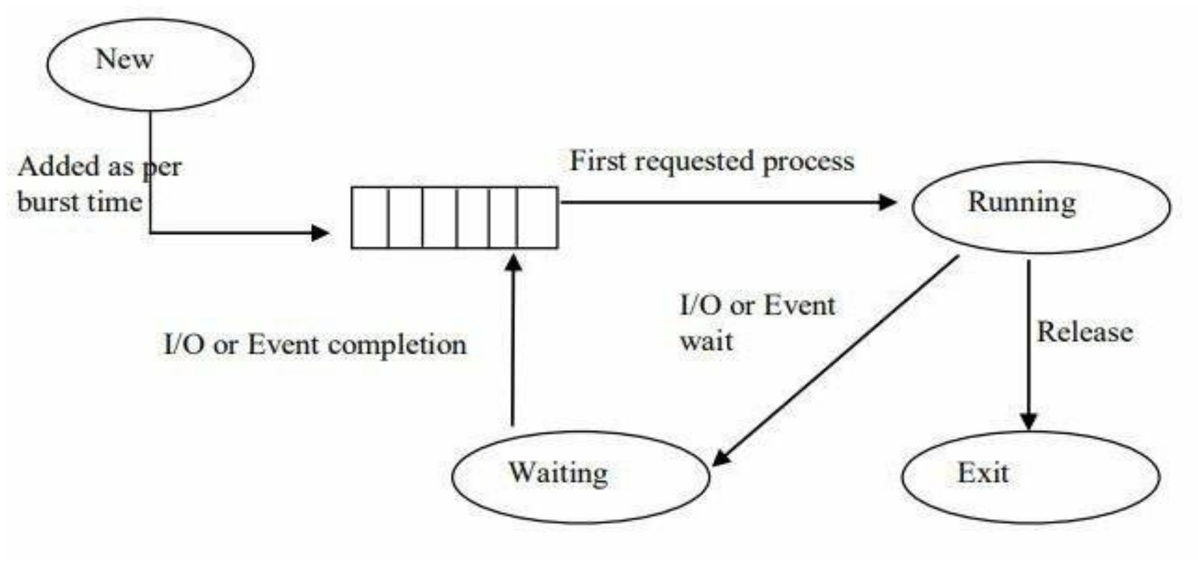
1) First-Come-First-Served (FCFS)

The FCFS scheduling algorithm operates on the principle that the process making the initial CPU request is granted access first. It utilizes a FIFO queue, where the process's Process Control Block (PCB) is linked to the queue's tail upon entering the ready queue. Once the CPU becomes available, it is assigned to the process at the queue's head. Subsequently, the running process is removed from the queue. FCFS is a non-preemptive scheduling algorithm..



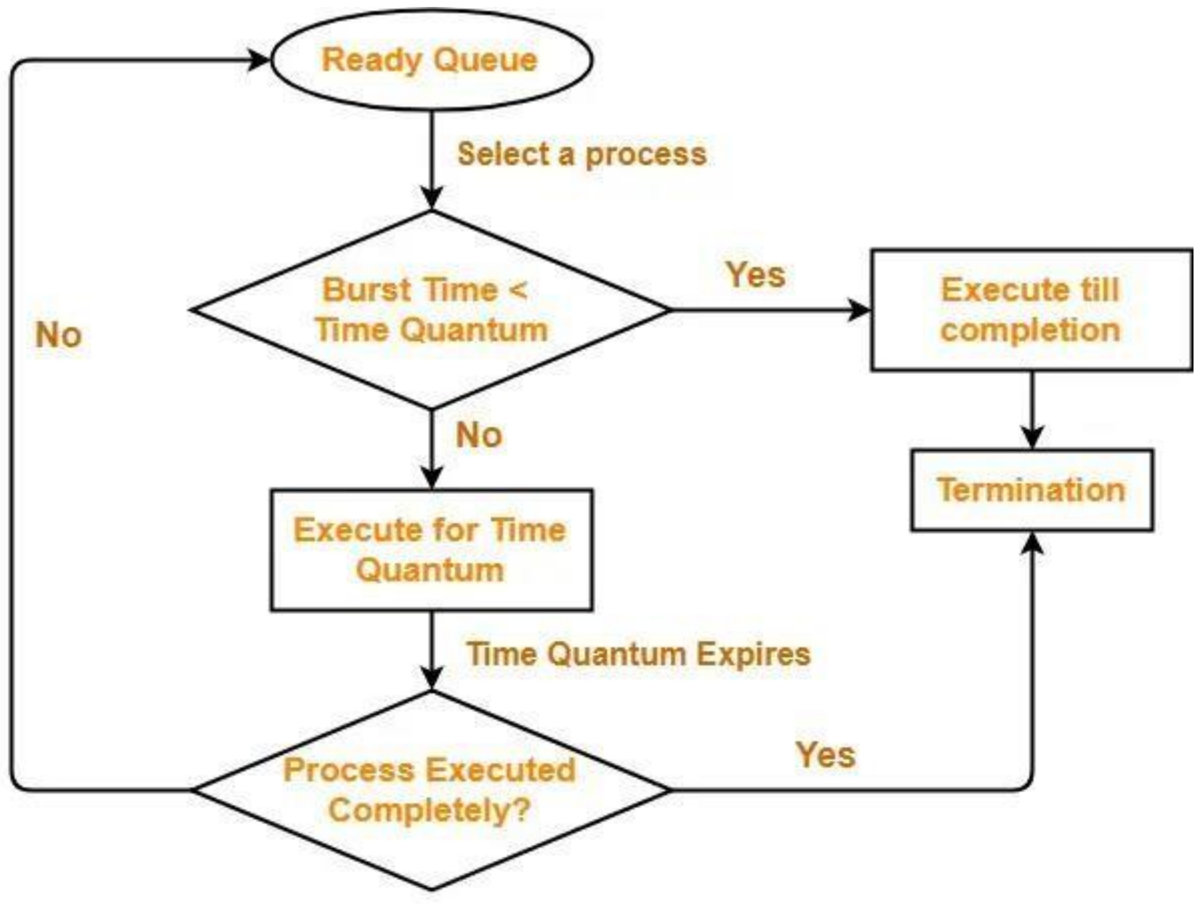
2) Shortest Job First (SJF)

The is a non-preemptive scheduling algorithm, prioritizes processes based on their burst times. In SJF, the process with the smallest burst time is selected for execution first, resulting in minimal overall completion time. This approach aims to reduce waiting times and enhance system efficiency by prioritizing shorter tasks.



3) Round Robin (RR)

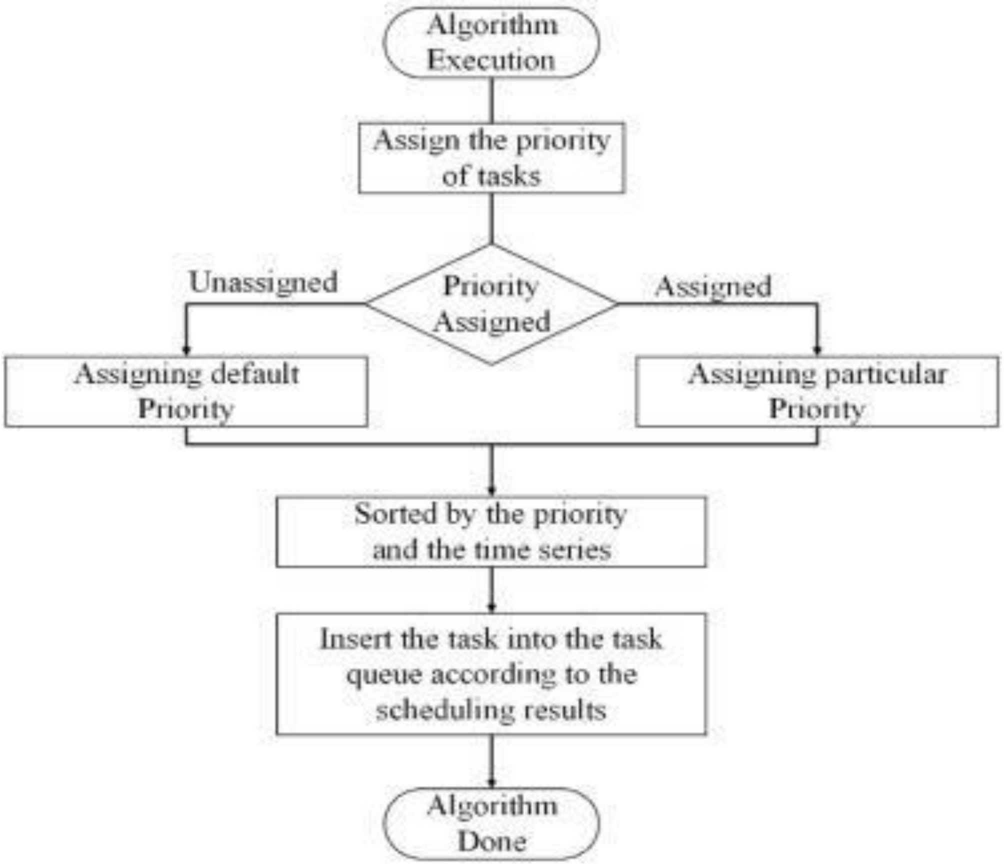
Round Robin serves as a CPU scheduling algorithm that assigns time slices to each process in a cyclic pattern. Each process is granted a small unit of CPU time, referred to as a time quantum, for execution. If a process fails to complete its execution within the allocated time, it is requeued. This approach is designed to ensure fairness, preventing the occurrence of starvation as every process is given an opportunity to run. However, there is a potential drawback of prolonged turnaround time for extended processes and inefficiency in handling tasks with varying execution times.



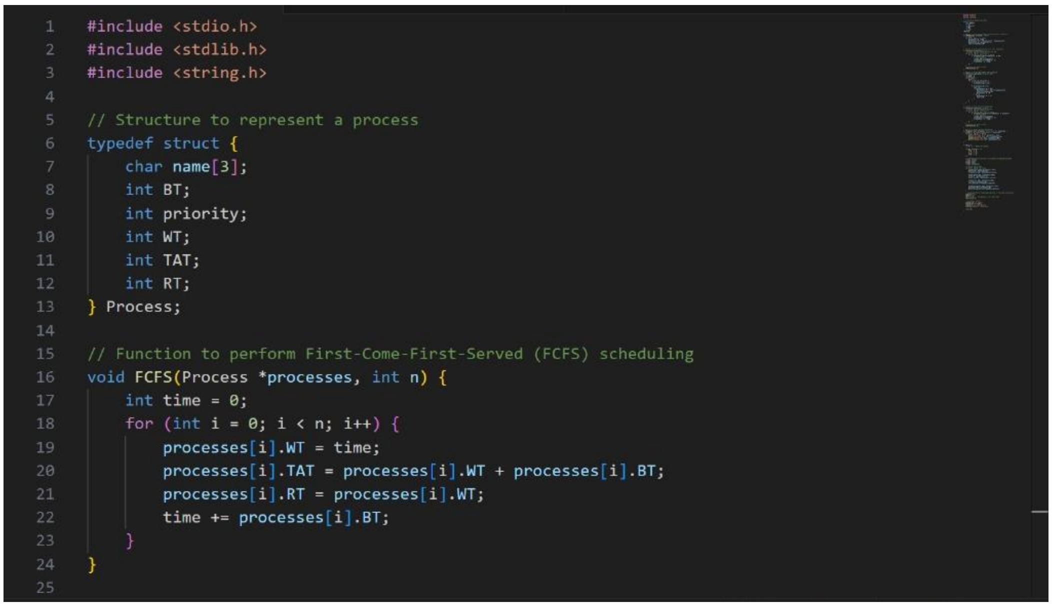
Round Robin Scheduling

4) Priority Scheduling

Priority scheduling is a CPU scheduling algorithm where processes are assigned priorities. The process possessing the highest priority is scheduled for execution first. This algorithm can be either preemptive or non-preemptive, enabling tasks with higher priorities to interrupt those with lower priorities. The objective is to guarantee the completion of higher-priority tasks before lower-priority ones, potentially resulting in an improved system performance.


Algorithm Execution
Assign the priority of tasks
Priority Assigned
Unassigned
Assigned
Assigning default Priority
Assigning particular Priority
Sorted by the priority and the time series
Insert the task into the task queue according to the scheduling results
Algorithm Done

CODE (In C - Language) with Explanation:

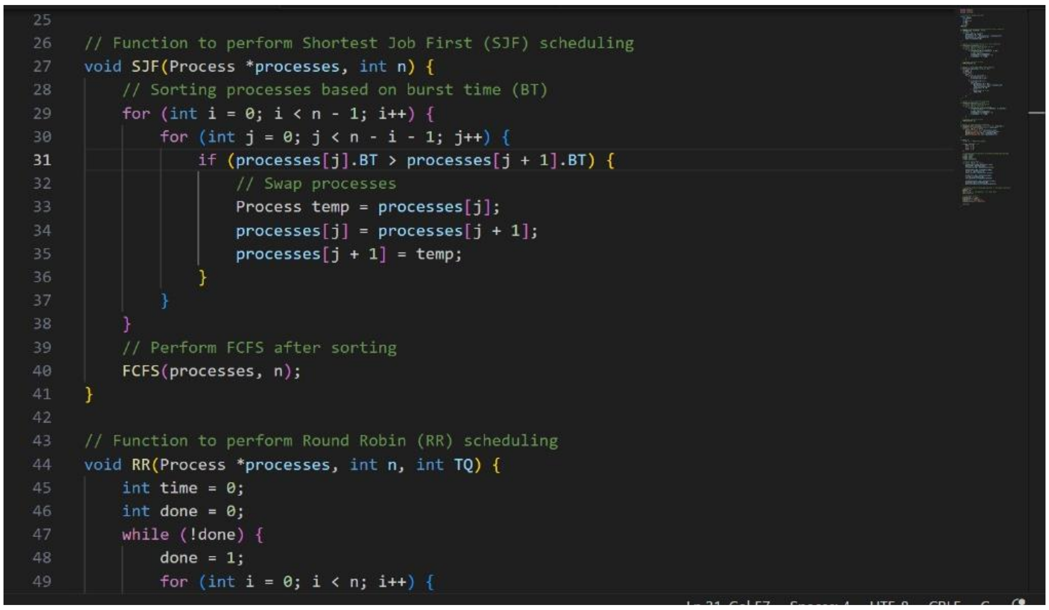


```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Structure to represent a process
typedef struct {
    char name[3];
    int BT;
    int priority;
    int WT;
    int TAT;
    int RT;
} Process;

// Function to perform First-Come-First-Served (FCFS) scheduling
void FCFS(Process *processes, int n) {
    int time = 0;
    for (int i = 0; i < n; i++) {
        processes[i].WT = time;
        processes[i].TAT = processes[i].WT + processes[i].BT;
        processes[i].RT = processes[i].WT;
        time += processes[i].BT;
    }
}

```

To solve this problem the following steps are followed:

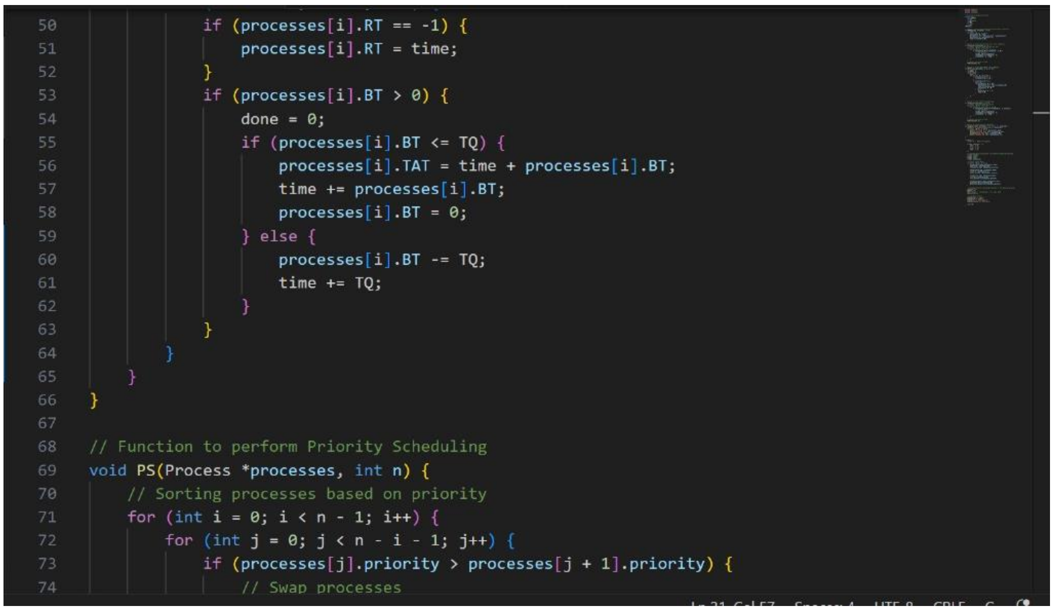
Step 1. **Structure Definition (`typedefstruct):**

In C, a structis used to define a custom data type to represent a process. Thetypedefstatement is used to create an aliasProcess` for this structure.

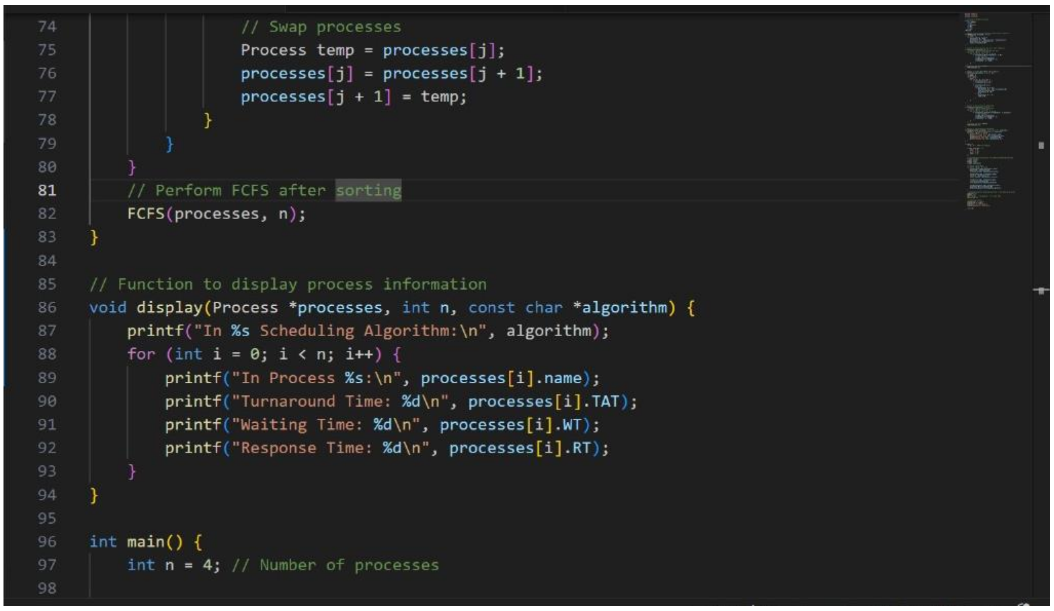


```c

// Function to perform Shortest Job First (SJF) scheduling
void SJF(Process *processes, int n) {
    // Sorting processes based on burst time (BT)
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].BT > processes[j + 1].BT) {
                // Swap processes
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }
    // Perform FCFS after sorting
    FCFS(processes, n);
}

// Function to perform Round Robin (RR) scheduling
void RR(Process *processes, int n, int TQ) {
    int time = 0;
    int done = 0;
    while (!done) {
        done = 1;
        for (int i = 0; i < n; i++) {
```

Step 2. Function Declarations and Definitions: The code defines several functions (FCFS, SJF, RR, PS, display, main) to perform different scheduling algorithms, display process information, and execute the main program**.**

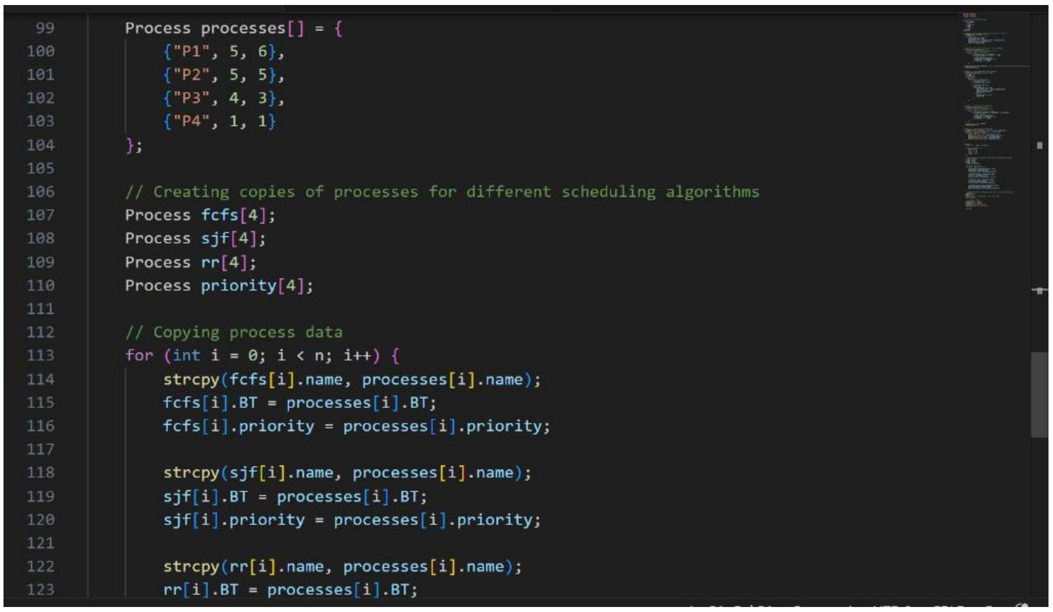


```c
            if (processes[i].RT == -1) {
                processes[i].RT = time;
            }
            if (processes[i].BT > 0) {
                done = 0;
                if (processes[i].BT <= TQ) {
                    processes[i].TAT = time + processes[i].BT;
                    time += processes[i].BT;
                    processes[i].BT = 0;
                } else {
                    processes[i].BT -= TQ;
                    time += TQ;
                }
            }
        }
    }
}

// Function to perform Priority Scheduling
void PS(Process *processes, int n) {
    // Sorting processes based on priority
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].priority > processes[j + 1].priority) {
                // Swap processes
```



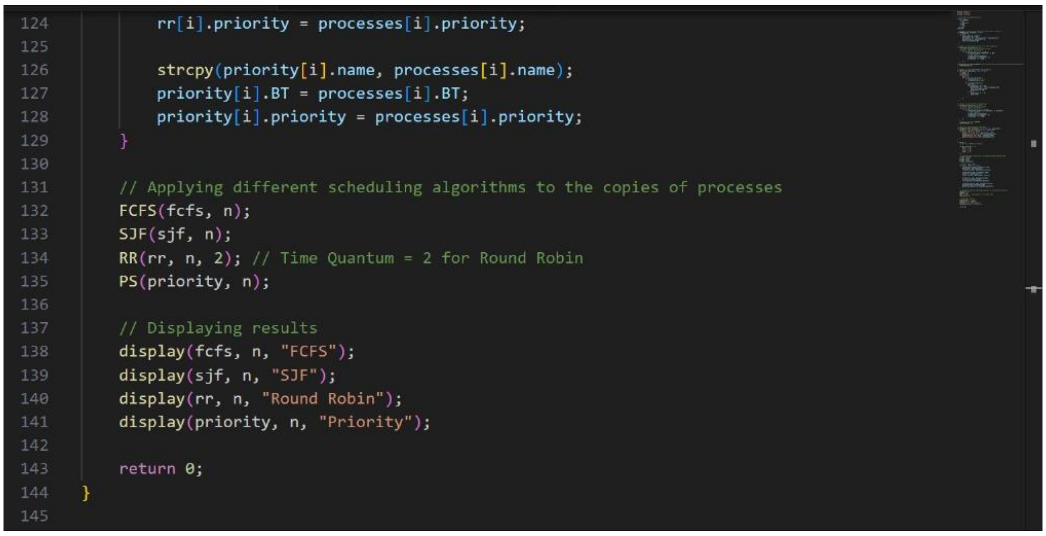
```c
                    // Swap processes
                    Process temp = processes[j];
                    processes[j] = processes[j + 1];
                    processes[j + 1] = temp;
                }
            }
        }
        // Perform FCFS after sorting
        FCFS(processes, n);
}

// Function to display process information
void display(Process *processes, int n, const char *algorithm) {
    printf("In %s Scheduling Algorithm:\n", algorithm);
    for (int i = 0; i < n; i++) {
        printf("In Process %s:\n", processes[i].name);
        printf("Turnaround Time: %d\n", processes[i].TAT);
        printf("Waiting Time: %d\n", processes[i].WT);
        printf("Response Time: %d\n", processes[i].RT);
    }
}

int main() {
    int n = 4; // Number of processes

```

Step 3. Sorting Algorithms: The sorting algorithms used (Bubble Sort for SJF and Priority Scheduling) are straightforward. They compare elements based on burst time or priority and swap them accordingly.

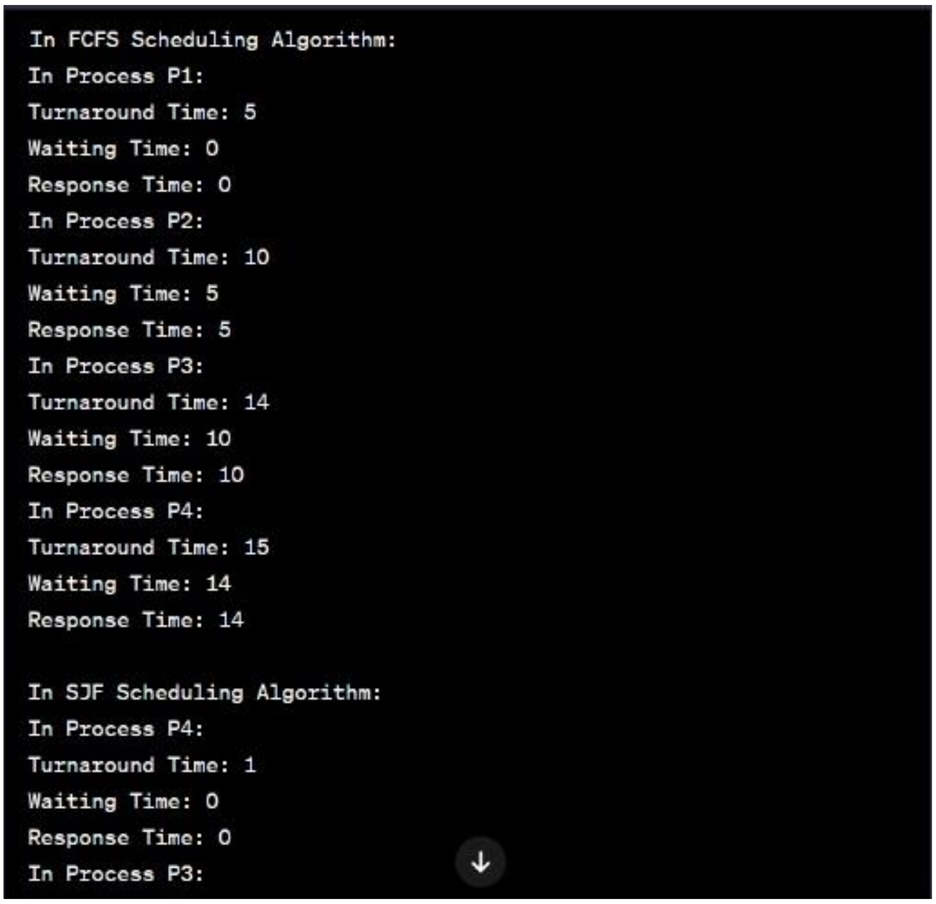


```c
    Process processes[] = {
        {"P1", 5, 6},
        {"P2", 5, 5},
        {"P3", 4, 3},
        {"P4", 1, 1}
    };

    // Creating copies of processes for different scheduling algorithms
    Process fcfs[4];
    Process sjf[4];
    Process rr[4];
    Process priority[4];

    // Copying process data
    for (int i = 0; i < n; i++) {
        strcpy(fcfs[i].name, processes[i].name);
        fcfs[i].BT = processes[i].BT;
        fcfs[i].priority = processes[i].priority;

        strcpy(sjf[i].name, processes[i].name);
        sjf[i].BT = processes[i].BT;
        sjf[i].priority = processes[i].priority;

        strcpy(rr[i].name, processes[i].name);
        rr[i].BT = processes[i].BT;
```

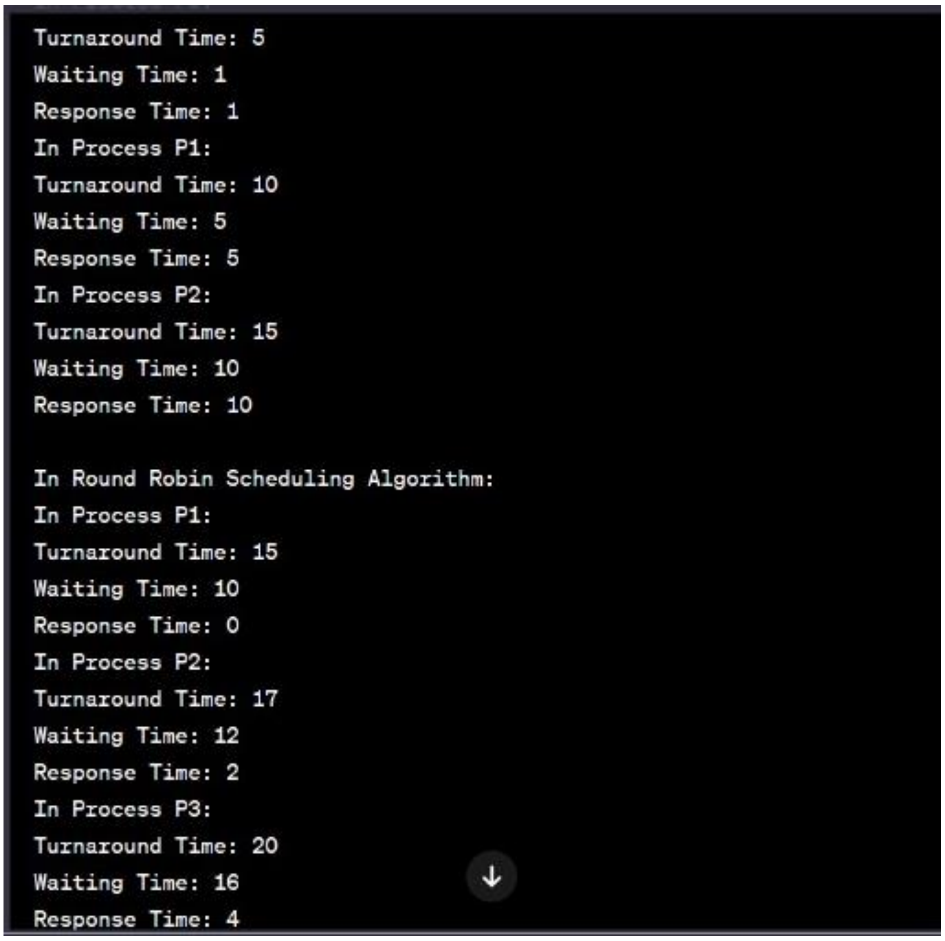
Step 4. Main Function (main): The main function initializes the original processes array, creates copies for each scheduling algorithm, applies the scheduling algorithms, and displays the results.


```c
            rr[i].priority = processes[i].priority;

            strcpy(priority[i].name, processes[i].name);
            priority[i].BT = processes[i].BT;
            priority[i].priority = processes[i].priority;
        }

        // Applying different scheduling algorithms to the copies of processes
        FCFS(fcfs, n);
        SJF(sjf, n);
        RR(rr, n, 2); // Time Quantum = 2 for Round Robin
        PS(priority, n);

        // Displaying results
        display(fcfs, n, "FCFS");
        display(sjf, n, "SJF");
        display(rr, n, "Round Robin");
        display(priority, n, "Priority");

        return 0;
    }
```

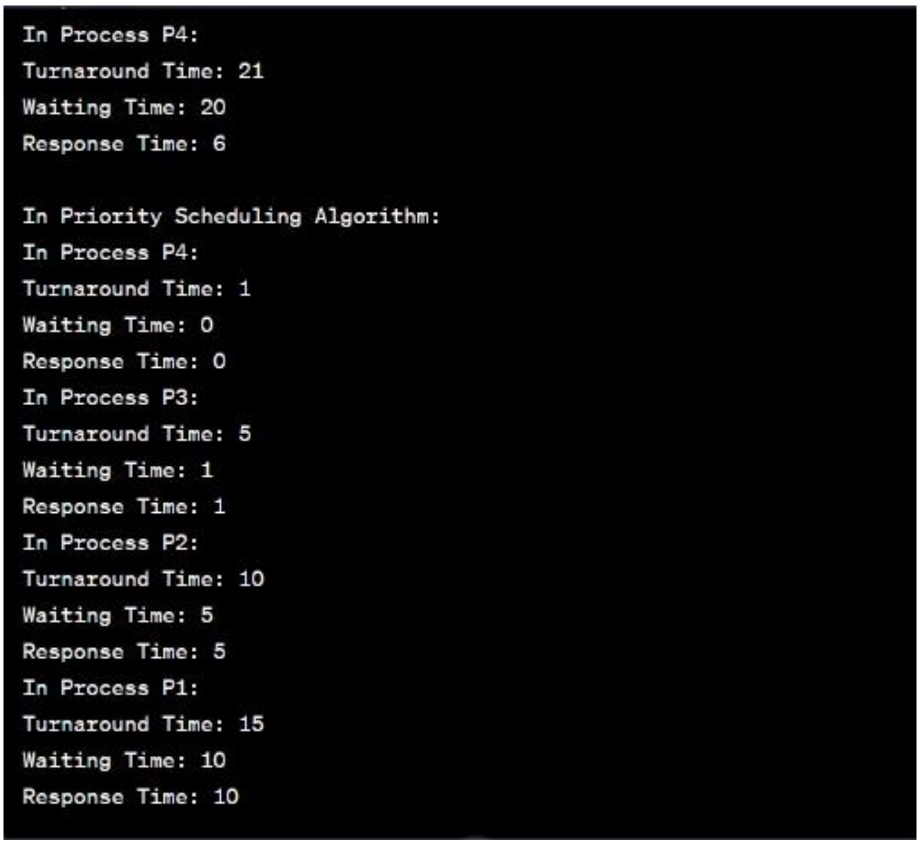
OUTPUT



```
In FCFS Scheduling Algorithm:
In Process P1:
Turnaround Time: 5
Waiting Time: 0
Response Time: 0
In Process P2:
Turnaround Time: 10
Waiting Time: 5
Response Time: 5
In Process P3:
Turnaround Time: 14
Waiting Time: 10
Response Time: 10
In Process P4:
Turnaround Time: 15
Waiting Time: 14
Response Time: 14

In SJF Scheduling Algorithm:
In Process P4:
Turnaround Time: 1
Waiting Time: 0
Response Time: 0
In Process P3:
```



```
Turnaround Time: 5
Waiting Time: 1
Response Time: 1
In Process P1:
Turnaround Time: 10
Waiting Time: 5
Response Time: 5
In Process P2:
Turnaround Time: 15
Waiting Time: 10
Response Time: 10

In Round Robin Scheduling Algorithm:
In Process P1:
Turnaround Time: 15
Waiting Time: 10
Response Time: 0
In Process P2:
Turnaround Time: 17
Waiting Time: 12
Response Time: 2
In Process P3:
Turnaround Time: 20
Waiting Time: 16
Response Time: 4
```



```
In Process P4:
Turnaround Time: 21
Waiting Time: 20
Response Time: 6

In Priority Scheduling Algorithm:
In Process P4:
Turnaround Time: 1
Waiting Time: 0
Response Time: 0
In Process P3:
Turnaround Time: 5
Waiting Time: 1
Response Time: 1
In Process P2:
Turnaround Time: 10
Waiting Time: 5
Response Time: 5
In Process P1:
Turnaround Time: 15
Waiting Time: 10
Response Time: 10
```

ANALYSIS ON THE BASIS OF PERFORMANCE

FCFS (First Come First Serve)

Simple and easy to implement, but may lead to long waiting times, especially for processes with longer burst times. It may not be optimal in terms of turnaround time.

SJF (Shortest Job First):

Can lead to optimal turnaround time by selecting the shortest job first. However, it may suffer from the "starvation" problem, where longer processes may wait indefinitely if shorter processes keep arriving.

Round Robin:

Ensures fairness and prevents starvation by allowing each process to run for a fixed time quantum. However, it may lead to higher turnaround time for longer processes and inefficiency with varying execution times.

Priority Scheduling:

Prioritizes processes based on priority. It can be effective in ensuring that high-priority tasks are completed first, but it may suffer from the "starvation" problem if lower-priority tasks keep arriving.

The choice of scheduling algorithm depends on the specific requirements and characteristics of the system. Each algorithm has its strengths and weaknesses, and the best choice depends on the workload and performance goals.

TASK-2

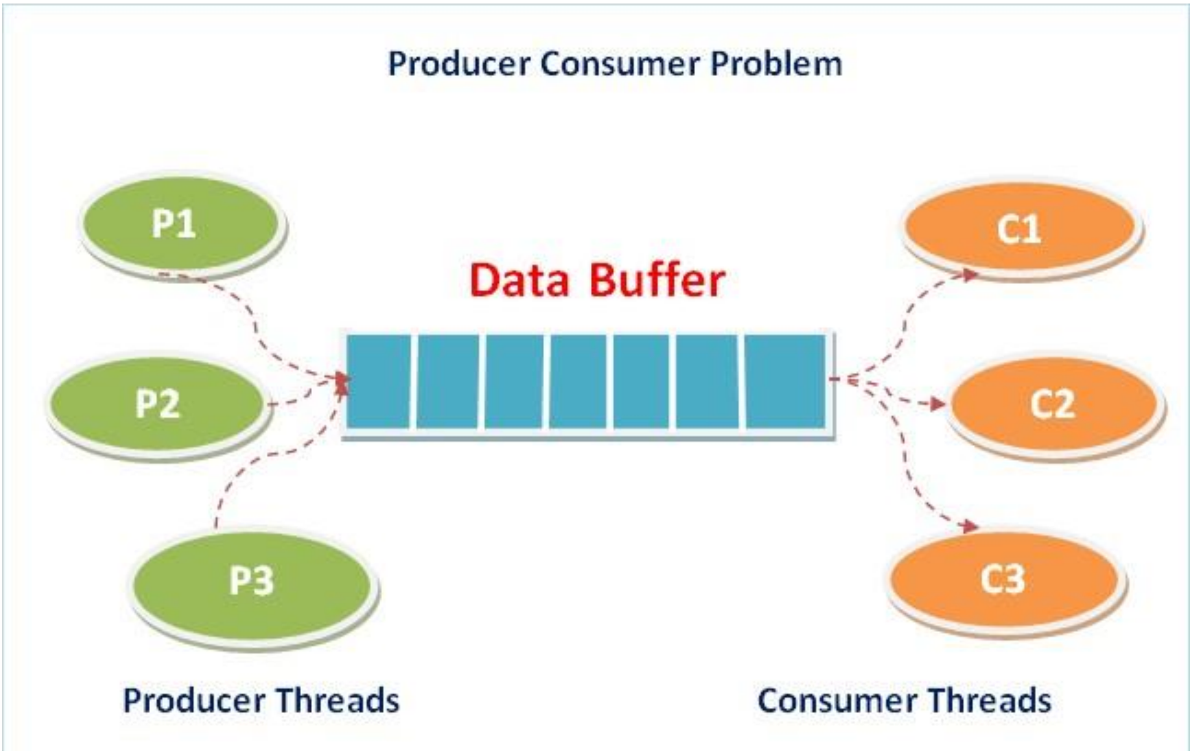
Write a multi-threaded program in C or another suitable language to solve the classic Producer-Consumer problem using semaphores or mutex locks. Describe how you ensure synchronization and avoid race conditions in your solution.

SOLUTION:

The essential terms used in the solution are outlined below:

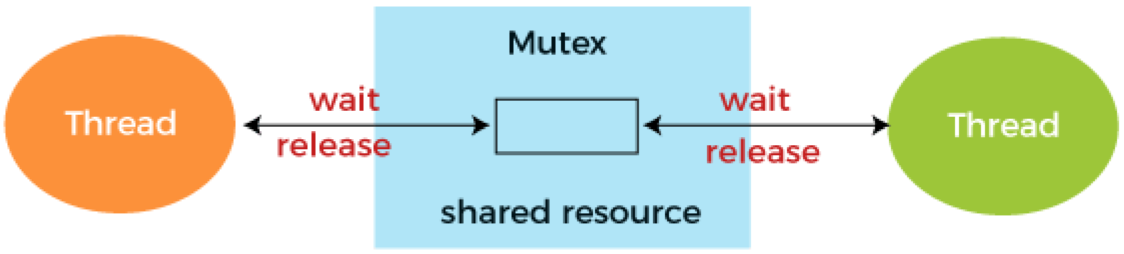
PRODUCER CONSUMER PROBLEM:

The producer-consumer problem represents synchronization challenges in concurrent computing. It revolves around a shared buffer where producers add data, and consumers retrieve and utilize it. Ensuring synchronized access to this buffer is vital to prevent issues like race conditions or data inconsistency. Solutions commonly utilize synchronization primitives such as semaphores or mutex locks to guarantee proper coordination between producers and consumers. Striking a balance between efficiency and avoiding deadlock is a crucial aspect of designing effective solutions for this classic synchronization problem.



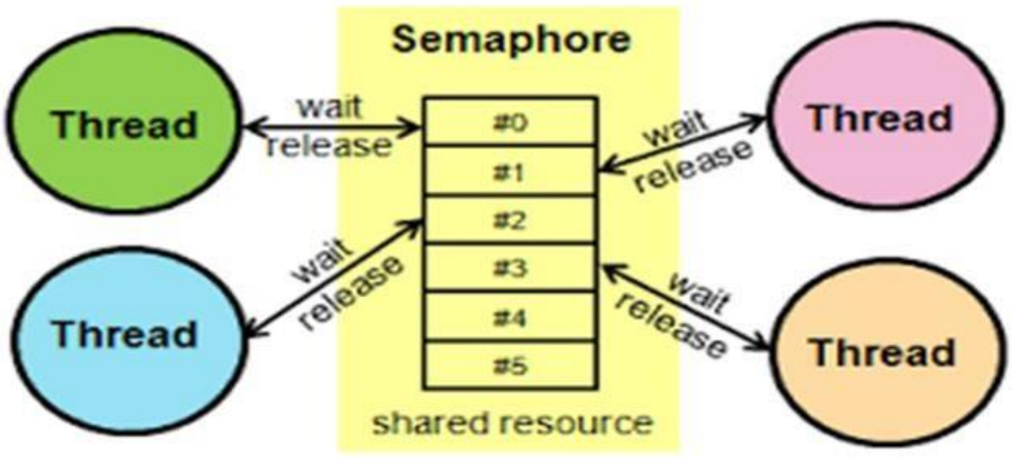
MUTEX LOCKS:

Mutex locks, abbreviated for mutual exclusion locks, guarantee exclusive access to shared resources in concurrent programming. They prevent multiple threads from concurrently accessing critical sections, preventing race conditions and preserving data integrity. Acquiring a mutex provides a thread with exclusive rights to a resource, while other threads wait until it is released. Proper management of mutex locks is essential to prevent deadlocks and ensure secure, synchronized access to shared data among multiple threads or processes.

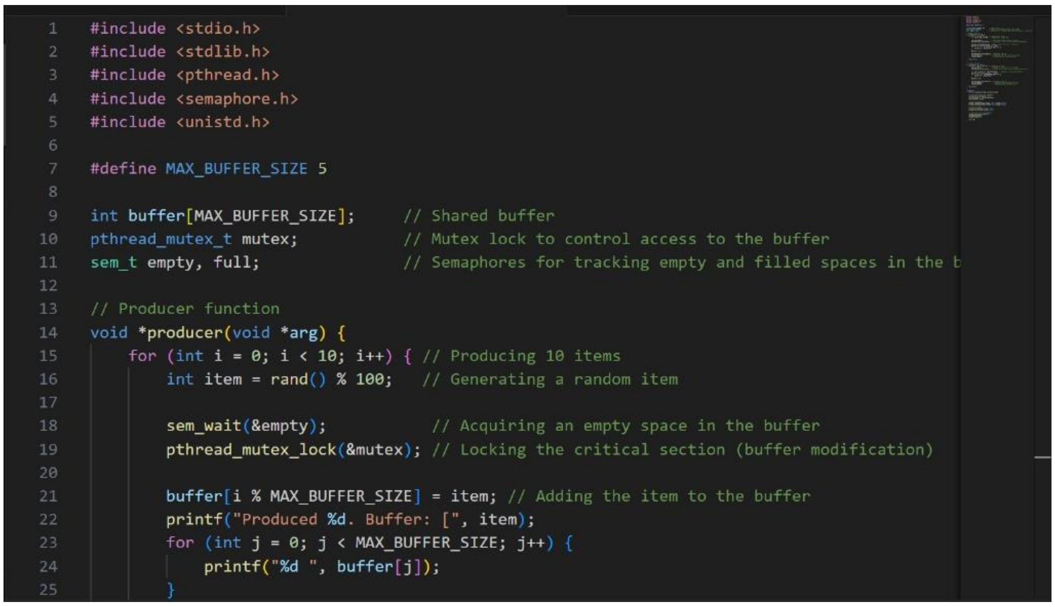


SEMAPHORES :

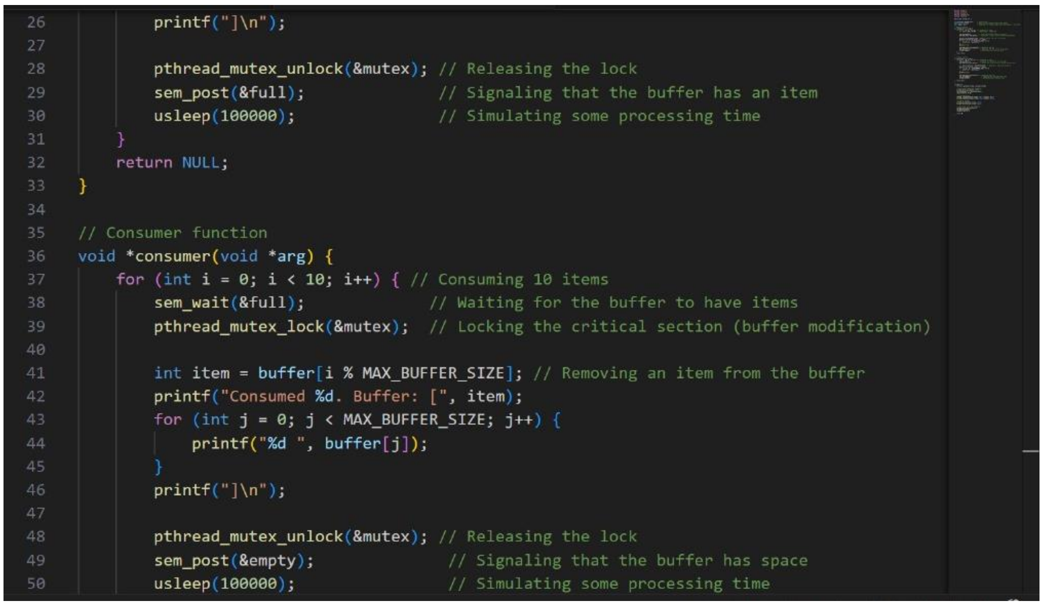
Semaphores are synchronization primitives utilized in concurrent programming to manage access to shared resources. They maintain a counter that limits the number of threads that can access a resource concurrently. Semaphores can be binary (similar to mutex) or count-based, allowing a specific number of threads access to a resource. They facilitate coordination among multiple threads, regulating critical sections and preventing race conditions by signaling when resources are available for use or when they become free. Semaphore operations include "wait" (decrement) and "signal" (increment) actions that aid in efficiently managing access to shared resources.



CODE (In C - Language)



```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>

#define MAX_BUFFER_SIZE 5

int buffer[MAX_BUFFER_SIZE];     // Shared buffer
pthread_mutex_t mutex;           // Mutex lock to control access to the buffer
sem_t empty, full;               // Semaphores for tracking empty and filled spaces in the b

// Producer function
void *producer(void *arg) {
    for (int i = 0; i < 10; i++) { // Producing 10 items
        int item = rand() % 100;   // Generating a random item

        sem_wait(&empty);           // Acquiring an empty space in the buffer
        pthread_mutex_lock(&mutex); // Locking the critical section (buffer modification)

        buffer[i % MAX_BUFFER_SIZE] = item; // Adding the item to the buffer
        printf("Produced %d. Buffer: [", item);
        for (int j = 0; j < MAX_BUFFER_SIZE; j++) {
            printf("%d ", buffer[j]);
        }
```



```c
        printf("]\n");

        pthread_mutex_unlock(&mutex); // Releasing the lock
        sem_post(&full);              // Signaling that the buffer has an item
        usleep(100000);               // Simulating some processing time
    }
    return NULL;
}

// Consumer function
void *consumer(void *arg) {
    for (int i = 0; i < 10; i++) { // Consuming 10 items
        sem_wait(&full);             // Waiting for the buffer to have items
        pthread_mutex_lock(&mutex);  // Locking the critical section (buffer modification)

        int item = buffer[i % MAX_BUFFER_SIZE]; // Removing an item from the buffer
        printf("Consumed %d. Buffer: [", item);
        for (int j = 0; j < MAX_BUFFER_SIZE; j++) {
            printf("%d ", buffer[j]);
        }
        printf("]\n");

        pthread_mutex_unlock(&mutex); // Releasing the lock
        sem_post(&empty);              // Signaling that the buffer has space
        usleep(100000);                // Simulating some processing time
```

EXPLANATION

1. ****Mutex Lock (pthread_mutex_t):****

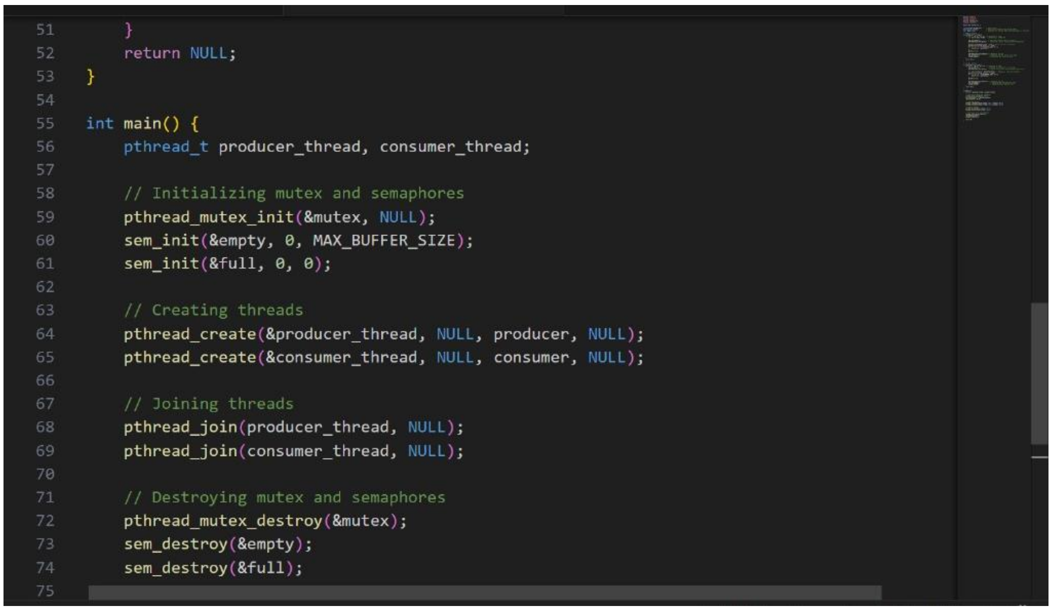
   Mutex locks are used to ensure exclusive access to the shared buffer, preventing race conditions during modifications.

2. ****Semaphores (sem_t):****

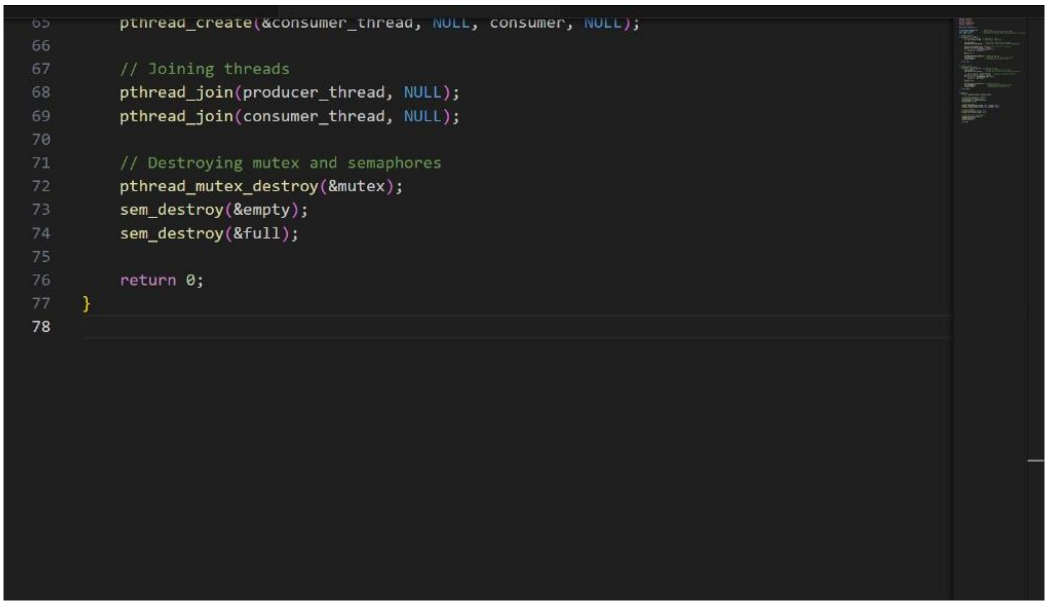
   Semaphores are used for synchronization. empty semaphore tracks empty spaces in the buffer, and full semaphore tracks filled spaces.

3. *Producer and Consumer Functions:*

These functions represent the producer and consumer actions. They acquire and release semaphores and lock/unlock the mutex for proper synchronization.



```
    }
    return NULL;
}

int main() {
    pthread_t producer_thread, consumer_thread;

    // Initializing mutex and semaphores
    pthread_mutex_init(&mutex, NULL);
    sem_init(&empty, 0, MAX_BUFFER_SIZE);
    sem_init(&full, 0, 0);

    // Creating threads
    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Joining threads
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Destroying mutex and semaphores
    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

```



```
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    // Joining threads
    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

    // Destroying mutex and semaphores
    pthread_mutex_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);

    return 0;
}

```

4. *Main Function:*

It initializes mutex and semaphores, creates producer and consumer threads, and then joins the threads to wait for their completion.

5. **Simulated Processing Time (usleep):**

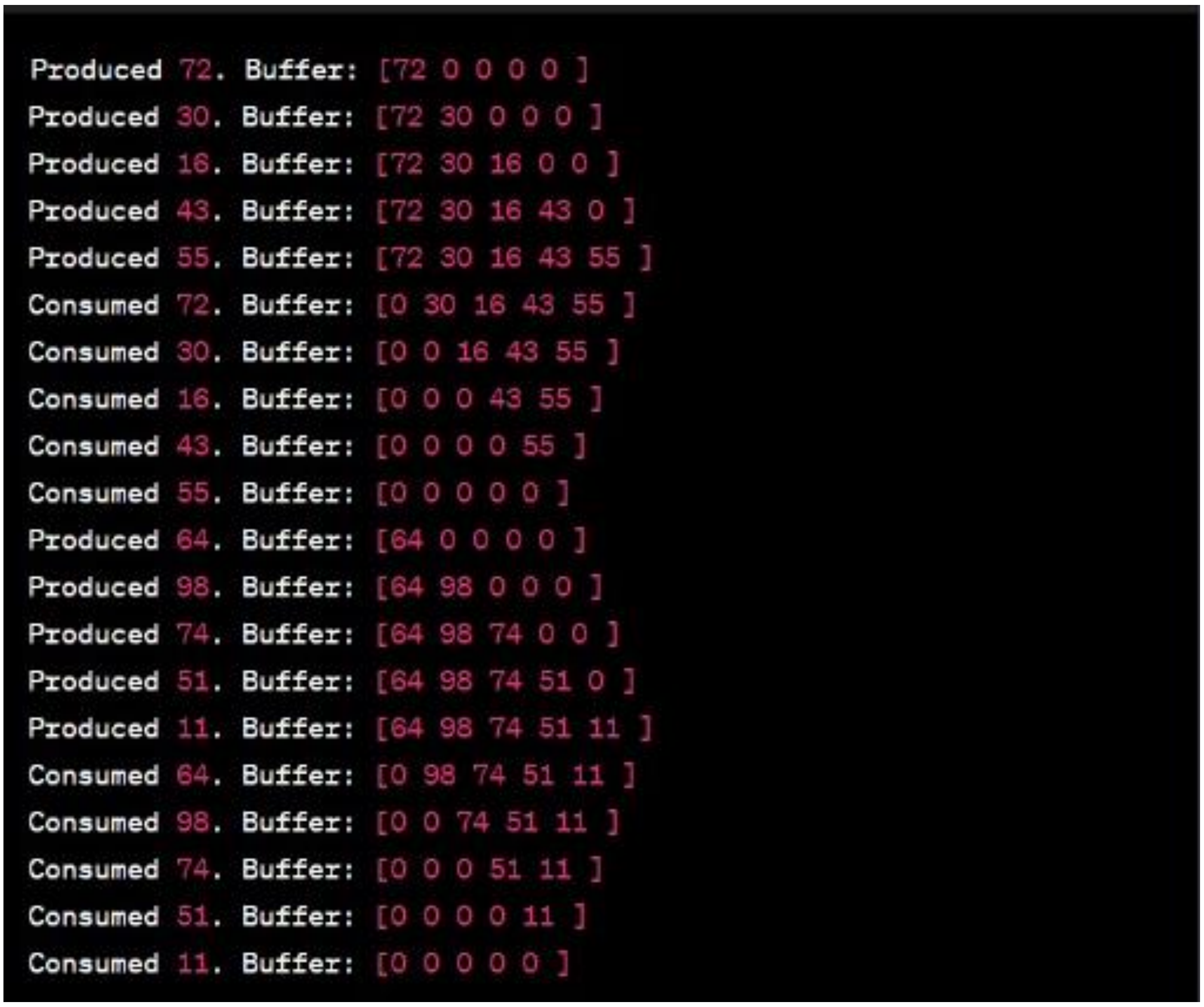
It introduces a delay to simulate some processing time between producing and consuming items.

6. *Thread Creation and Joining:*

Threads are created using pthread_create, and the pthread_join function is used to wait for the threads to finish execution.

This C code provides a multi-threaded solution to the Producer-Consumer problem with proper synchronization using mutex locks and semaphores to avoid race conditions.

OUTPUT



```
Produced 72. Buffer: [72 0 0 0 0 ]
Produced 30. Buffer: [72 30 0 0 0 ]
Produced 16. Buffer: [72 30 16 0 0 ]
Produced 43. Buffer: [72 30 16 43 0 ]
Produced 55. Buffer: [72 30 16 43 55 ]
Consumed 72. Buffer: [0 30 16 43 55 ]
Consumed 30. Buffer: [0 0 16 43 55 ]
Consumed 16. Buffer: [0 0 0 43 55 ]
Consumed 43. Buffer: [0 0 0 0 55 ]
Consumed 55. Buffer: [0 0 0 0 0 ]
Produced 64. Buffer: [64 0 0 0 0 ]
Produced 98. Buffer: [64 98 0 0 0 ]
Produced 74. Buffer: [64 98 74 0 0 ]
Produced 51. Buffer: [64 98 74 51 0 ]
Produced 11. Buffer: [64 98 74 51 11 ]
Consumed 64. Buffer: [0 98 74 51 11 ]
Consumed 98. Buffer: [0 0 74 51 11 ]
Consumed 74. Buffer: [0 0 0 51 11 ]
Consumed 51. Buffer: [0 0 0 0 11 ]
Consumed 11. Buffer: [0 0 0 0 0 ]
```

SYNCHRONIZATION AND AVOIDANCE OF RACE CONDTIONS

Synchronization and avoidance of race conditions in the provided solution are ensured through several mechanisms:

1. Mutex Lock (`mutex`): The `mutex` lock is used to ensure exclusive access to the shared `buffer` between the producer and consumer threads. Whenever a thread wants to access or modify the buffer, it acquires the lock (`mutex.acquire()`) before entering the critical section and releases it (`mutex.release()`) after finishing the critical operations. This prevents multiple threads from simultaneously modifying the buffer, thus avoiding race conditions.

2. Semaphores (`empty` and `full`): Semaphores are used to control access to the buffer based on its state—whether it's empty or full.

- `empty` semaphore ensures that the producer waits if the buffer is full (i.e., there's no empty space).

- `full` semaphore ensures that the consumer waits if the buffer is empty (i.e., there are no items to consume).

- When the producer adds an item to the buffer, it releases the `full` semaphore to signal that the buffer is no longer empty. When the consumer removes an item from the buffer, it releases the `empty` semaphore to signal that there is now an empty space in the buffer.

3. Orderly Access to Buffer: Both producer and consumer threads access the buffer in an orderly manner. The producer waits for an empty space before adding items (`empty.acquire()`), and the consumer waits for a filled space before removing items (`full.acquire()`). This ensures that the buffer is only accessed when it's in a valid state for production or consumption, preventing issues that arise from accessing an inconsistent buffer state.

By combining 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 ensures that only one thread at a time modifies the buffer, prevents simultaneous access to critical sections, and regulates access to the buffer to prevent race conditions and maintain data integrity in a multi-threaded environment.