ASSIGNMENT

By

Gokul kangotra

2022A1R007

3rd SEM

A2

Computer science and Engineering

Model institute of engineering and technology (Autonomous)

(Permanently affiliated with university of Jammu. Accredited with NAAC with 'A' grade)

Jammu, India

2023

OS ASSIGNMENT

GROUP B

COM-302: Operating system

Due date: 4-12-2023

QUESTION NUMBERS	COURSE OUTCOMES	BLOOM'S LEVEL	MAXIMUM MARKS	MARKS OBTAIN
Q1	CO 4	3-6	10	
Q2	CO5	3-6	10	
TOTAL MARKS			20	
Faculty signature: Dr. Mekhla Sharma Email: mekhla.cse@mietjammu.in				

TASK FIRST

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.

CODE

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_PROCESSES 10
typedef struct {
    int id;
    int burst_time;
    int waiting_time;
    int turnaround_time;
    int priority;
    int remaining_time;
} Process;
void fcfs(Process processes[], int n);
void sjf(Process processes[], int n);
void round_robin(Process processes[], int n, int quantum);
void priority(Process processes[], int n);

int main() {
    int n, quantum;
 printf("Enter the number of processes: ");
    scanf("%d", &n);
```

```
if (n > MAX_PROCESSES) {
    printf("Exceeded the maximum number of processes.\n");
    return 1;
}

Process processes[MAX_PROCESSES];

// Generate random processes
srand(123); // Seed for reproducibility
for (int i = 0; i < n; i++) {
    processes[i].id = i + 1;
    processes[i].burst_time = rand() % 20 + 1; // Random burst time between 1 and 20
    processes[i].waiting_time = 0;
    processes[i].turnaround_time = 0;
    processes[i].priority = rand() % 10; // Random priority between 0 and 9
    processes[i].remaining_time = processes[i].burst_time;
}

// Display generated processes
printf("\nGenerated Processes:\n");
printf("ID\tBurst Time\tPriority\n");
for (int i = 0; i < n; i++) {
    printf("%d\t%d\t\t%d\n", processes[i].id, processes[i].burst_time, processes[i].priority);
}

// FCFS
printf("\nFCFS Scheduling:\n");
fcfs(processes, n);

// SJF
printf("\nSJF Scheduling:\n");
sjf(processes, n);

// Round Robin
```

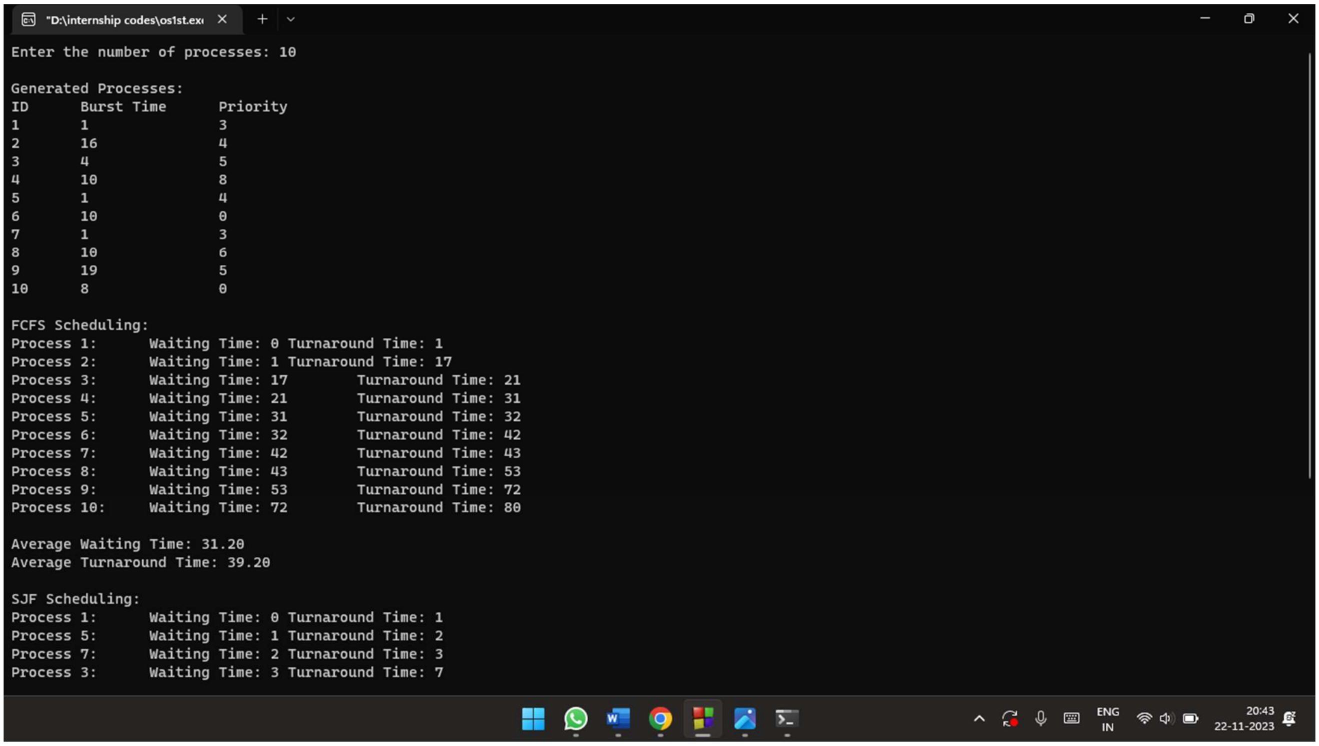
```
    printf("\nEnter time quantum for Round Robin: ");
    scanf("%d", &quantum);
    printf("\nRound Robin Scheduling (Quantum = %d):\n", quantum);
    round_robin(processes, n, quantum);

    // Priority
    printf("\nPriority Scheduling:\n");
    priority(processes, n);

    return 0;
}

void fcfs(Process processes[], int n) {
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    for (int i = 0; i < n; i++) {
        if (i > 0) {
            processes[i].waiting_time = processes[i - 1].turnaround_time;
        }

        processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;

        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;

        printf("Process %d:\tWaiting Time: %d\tTurnaround Time: %d\n",
               processes[i].id, processes[i].waiting_time, processes[i].turnaround_time);
    }

    printf("\nAverage Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}
```

```
void sjf(Process processes[], int n) {
    // Sort processes based on burst time
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (processes[j].burst_time > processes[j + 1].burst_time) {
                // Swap
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    // Calculate waiting and turnaround times
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    for (int i = 0; i < n; i++) {
        if (i > 0) {
            processes[i].waiting_time = processes[i - 1].turnaround_time;
        }

        processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;

        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;

        printf("Process %d:\tWaiting Time: %d\tTurnaround Time: %d\n",
               processes[i].id, processes[i].waiting_time, processes[i].turnaround_time);
    }

    printf("\nAverage Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}
```

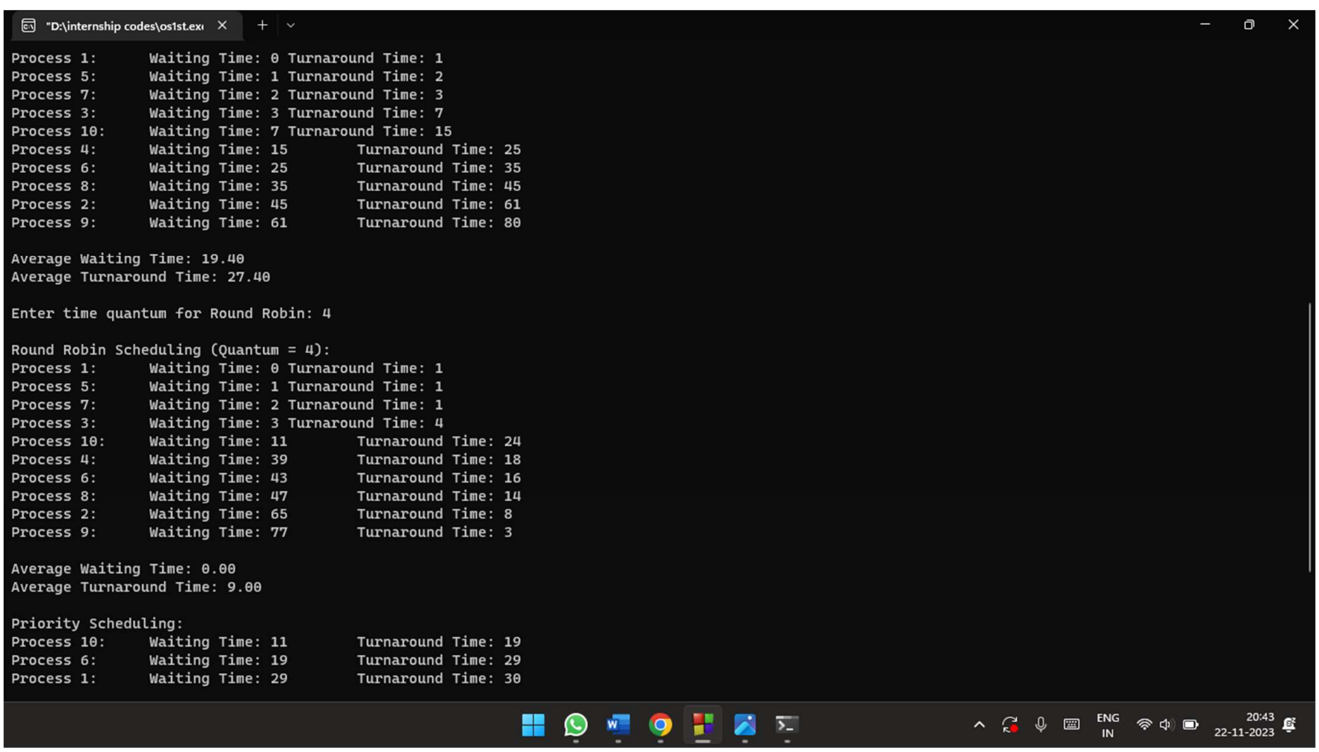
```
void round_robin(Process processes[], int n, int quantum) {
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    int remaining_processes = n;
    int current_time = 0;

    while (remaining_processes > 0) {
        for (int i = 0; i < n; i++) {
            if (processes[i].remaining_time > 0) {
                int execute_time = (processes[i].remaining_time < quantum) ? processes[i].remaining_time :
quantum;

                processes[i].remaining_time -= execute_time;
                current_time += execute_time;

                if (processes[i].remaining_time == 0) {
                    processes[i].turnaround_time = current_time - processes[i].waiting_time;
                    total_turnaround_time += processes[i].turnaround_time;
                    remaining_processes--;

                    printf("Process %d:\tWaiting Time: %d\tTurnaround Time: %d\n",
                           processes[i].id, processes[i].waiting_time, processes[i].turnaround_time);
                } else {
                    processes[i].waiting_time = current_time;
                }
            }
        }
    }

    printf("\nAverage Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}
```

```
void priority(Process processes[], int n) {
    // Sort 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
                Process temp = processes[j];
                processes[j] = processes[j + 1];
                processes[j + 1] = temp;
            }
        }
    }

    // Calculate waiting and turnaround times
    int total_waiting_time = 0;
    int total_turnaround_time = 0;

    for (int i = 0; i < n; i++) {
        if (i > 0) {
            processes[i].waiting_time = processes[i - 1].turnaround_time;
        }

        processes[i].turnaround_time = processes[i].waiting_time + processes[i].burst_time;

        total_waiting_time += processes[i].waiting_time;
        total_turnaround_time += processes[i].turnaround_time;

        printf("Process %d:\tWaiting Time: %d\tTurnaround Time: %d\n",
               processes[i].id, processes[i].waiting_time, processes[i].turnaround_time);
    }
printf("\nAverage Waiting Time: %.2f\n", (float)total_waiting_time / n);
    printf("Average Turnaround Time: %.2f\n", (float)total_turnaround_time / n);
}
```

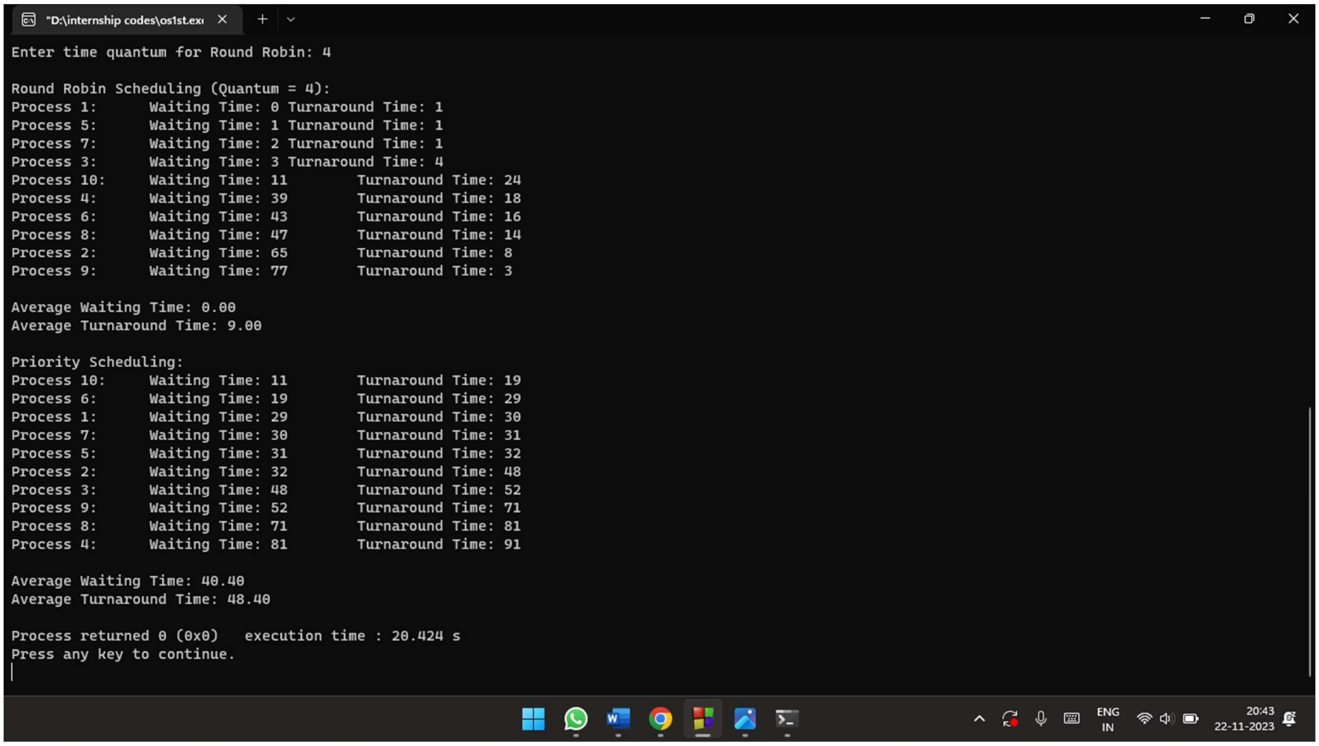
OUTPUT



```
Enter the number of processes: 10

Generated Processes:
ID      Burst Time      Priority
1       1               3
2       16              4
3       4               5
4       10              8
5       1               4
6       10              0
7       1               3
8       10              6
9       19              5
10      8               0

FCFS Scheduling:
Process 1:      Waiting Time: 0 Turnaround Time: 1
Process 2:      Waiting Time: 1 Turnaround Time: 17
Process 3:      Waiting Time: 17        Turnaround Time: 21
Process 4:      Waiting Time: 21        Turnaround Time: 31
Process 5:      Waiting Time: 31        Turnaround Time: 32
Process 6:      Waiting Time: 32        Turnaround Time: 42
Process 7:      Waiting Time: 42        Turnaround Time: 43
Process 8:      Waiting Time: 43        Turnaround Time: 53
Process 9:      Waiting Time: 53        Turnaround Time: 72
Process 10:     Waiting Time: 72        Turnaround Time: 80

Average Waiting Time: 31.20
Average Turnaround Time: 39.20

SJF Scheduling:
Process 1:      Waiting Time: 0 Turnaround Time: 1
Process 5:      Waiting Time: 1 Turnaround Time: 2
Process 7:      Waiting Time: 2 Turnaround Time: 3
Process 3:      Waiting Time: 3 Turnaround Time: 7
```



```
Process 1:      Waiting Time: 0 Turnaround Time: 1
Process 5:      Waiting Time: 1 Turnaround Time: 2
Process 7:      Waiting Time: 2 Turnaround Time: 3
Process 3:      Waiting Time: 3 Turnaround Time: 7
Process 10:     Waiting Time: 7 Turnaround Time: 15
Process 4:      Waiting Time: 15        Turnaround Time: 25
Process 6:      Waiting Time: 25        Turnaround Time: 35
Process 8:      Waiting Time: 35        Turnaround Time: 45
Process 2:      Waiting Time: 45        Turnaround Time: 61
Process 9:      Waiting Time: 61        Turnaround Time: 80

Average Waiting Time: 19.40
Average Turnaround Time: 27.40

Enter time quantum for Round Robin: 4

Round Robin Scheduling (Quantum = 4):
Process 1:      Waiting Time: 0 Turnaround Time: 1
Process 5:      Waiting Time: 1 Turnaround Time: 1
Process 7:      Waiting Time: 2 Turnaround Time: 1
Process 3:      Waiting Time: 3 Turnaround Time: 4
Process 10:     Waiting Time: 11        Turnaround Time: 24
Process 4:      Waiting Time: 39        Turnaround Time: 18
Process 6:      Waiting Time: 43        Turnaround Time: 16
Process 8:      Waiting Time: 47        Turnaround Time: 14
Process 2:      Waiting Time: 65        Turnaround Time: 8
Process 9:      Waiting Time: 77        Turnaround Time: 3

Average Waiting Time: 0.00
Average Turnaround Time: 9.00

Priority Scheduling:
Process 10:     Waiting Time: 11        Turnaround Time: 19
Process 6:      Waiting Time: 19        Turnaround Time: 29
Process 1:      Waiting Time: 29        Turnaround Time: 30
```



```
Enter time quantum for Round Robin: 4

Round Robin Scheduling (Quantum = 4):
Process 1:      Waiting Time: 0 Turnaround Time: 1
Process 5:      Waiting Time: 1 Turnaround Time: 1
Process 7:      Waiting Time: 2 Turnaround Time: 1
Process 3:      Waiting Time: 3 Turnaround Time: 4
Process 10:     Waiting Time: 11        Turnaround Time: 24
Process 4:      Waiting Time: 39        Turnaround Time: 18
Process 6:      Waiting Time: 43        Turnaround Time: 16
Process 8:      Waiting Time: 47        Turnaround Time: 14
Process 2:      Waiting Time: 65        Turnaround Time: 8
Process 9:      Waiting Time: 77        Turnaround Time: 3

Average Waiting Time: 0.00
Average Turnaround Time: 9.00

Priority Scheduling:
Process 10:     Waiting Time: 11        Turnaround Time: 19
Process 6:      Waiting Time: 19        Turnaround Time: 29
Process 1:      Waiting Time: 29        Turnaround Time: 30
Process 7:      Waiting Time: 30        Turnaround Time: 31
Process 5:      Waiting Time: 31        Turnaround Time: 32
Process 2:      Waiting Time: 32        Turnaround Time: 48
Process 3:      Waiting Time: 48        Turnaround Time: 52
Process 9:      Waiting Time: 52        Turnaround Time: 71
Process 8:      Waiting Time: 71        Turnaround Time: 81
Process 4:      Waiting Time: 81        Turnaround Time: 91

Average Waiting Time: 40.40
Average Turnaround Time: 48.40

Process returned 0 (0x0)   execution time : 20.424 s
Press any key to continue.
```

ANYALASIS REPORT

Overview:

The program simulates four CPU scheduling algorithms: First-Come-First-Served (FCFS), Shortest Job First (SJF), Round Robin (RR), and Priority Scheduling. It generates random processes with burst times and priorities for testing.

Strengths:

1. **Modular Design:**
    - The code is modular, with each scheduling algorithm implemented as a separate function (**fcfs**, **sjf**, **round_robin**, and **priority**).
    - This modular design enhances readability and allows for easy modification or extension.
2. **User Interaction:**

- The program interacts with the user to input the number of processes and the time quantum for Round Robin scheduling.
- This makes the program flexible and adaptable to different scenarios.

3. **Random Process Generation:**
   - Random processes are generated, providing a dynamic testing environment.
   - This allows for the evaluation of scheduling algorithms under various conditions.

Weaknesses:

1. **Limited Error Handling:**
   - The program lacks extensive error handling. It assumes valid input, and errors may occur if the user provides unexpected or incorrect input.
2. **Fixed Maximum Processes:**
   - The code defines a maximum number of processes (**MAX_PROCESSES**). If the user enters more processes, the program displays an error.
   - A more dynamic approach, such as dynamic memory allocation, could be considered to handle any number of processes.

3. **Incomplete Round Robin Implementation:**

- The Round Robin implementation is incomplete. The **total_waiting_time** and **total_turnaround_time** variables are initialized but not used.
- Additionally, the average waiting and turnaround times are not correctly calculated for Round Robin scheduling.

Suggestions for Improvement:

1. **Error Handling:**
    - Implement robust error handling to handle invalid user inputs and unexpected situations gracefully.
2. **Dynamic Memory Allocation:**
    - Consider using dynamic memory allocation for processes to handle any number of processes entered by the user.
3. **Complete Round Robin Implementation:**
    - Complete the Round Robin implementation by correctly calculating average waiting and turnaround times.
4. **Additional Metrics:**
    - Include additional metrics like response time and consider displaying a Gantt chart for a visual representation of the scheduling.
5. **Code Comments:**
    - Add comments to explain complex sections of the code, making it more understandable for someone reading the code for the first time.

Conclusion:

The provided program serves as a good starting point for simulating CPU scheduling algorithms. Addressing the mentioned weaknesses and incorporating suggestions for improvement would enhance the program's reliability, flexibility, and completeness. Additionally, further testing with a variety of scenarios and edge cases would help ensure the correctness and robustness of the implementation.

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.

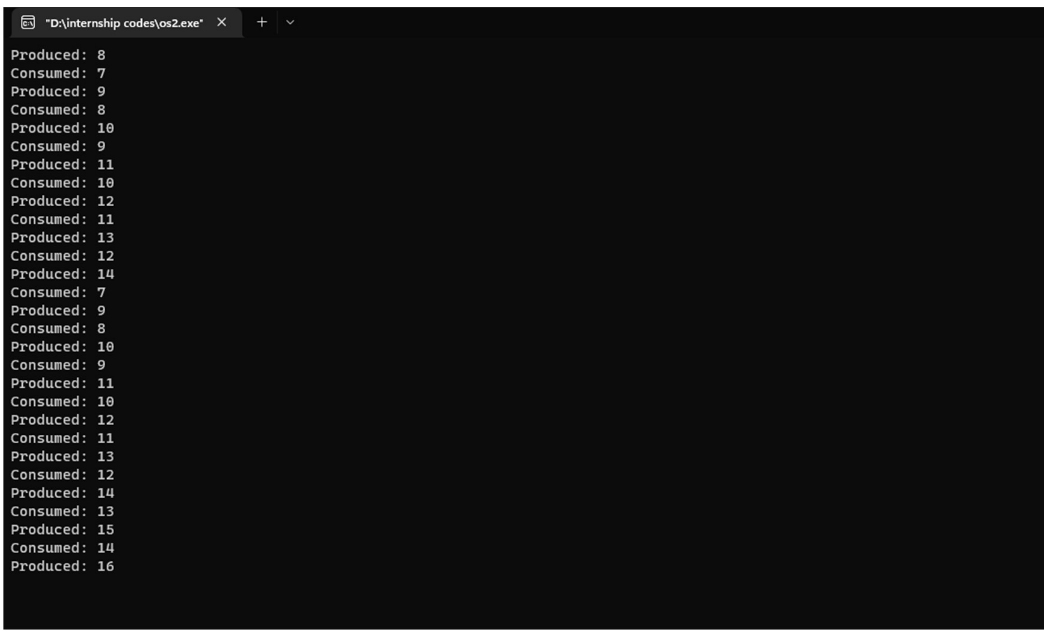
CODE

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

#define BUFFER_SIZE 5
#define MAX_ITEMS 20

int buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
sem_t mutex, empty, full;
int produced_items = 0;

void *producer(void *arg) {
    int item = 1;
    while (produced_items < MAX_ITEMS) {
        sem_wait(&empty);
        sem_wait(&mutex);

        buffer[in] = item;
        printf("Produced: %d\n", item++);
        in = (in + 1) % BUFFER_SIZE;
```

```
        produced_items++;

        sem_post(&mutex);
        sem_post(&full);
    }
    pthread_exit(NULL);
}

void *consumer(void *arg) {
    while (produced_items < MAX_ITEMS) {
        sem_wait(&full);
        sem_wait(&mutex);

        int item = buffer[out];
        printf("Consumed: %d\n", item);
        out = (out + 1) % BUFFER_SIZE;

        sem_post(&mutex);
        sem_post(&empty);
    }
    pthread_exit(NULL);
}

int main() {
    pthread_t producer_thread, consumer_thread;

    sem_init(&mutex, 0, 1);
    sem_init(&empty, 0, BUFFER_SIZE);
    sem_init(&full, 0, 0);

    pthread_create(&producer_thread, NULL, producer, NULL);
    pthread_create(&consumer_thread, NULL, consumer, NULL);

    pthread_join(producer_thread, NULL);
```

```
    pthread_join(consumer_thread, NULL);


    sem_destroy(&mutex);
    sem_destroy(&empty);
    sem_destroy(&full);


    return 0;
}
```

OUTPUT



ANALAYSIS REPORT

Program Overview:

The program is a multi-threaded implementation of the classic Producer-Consumer problem in C. It uses three semaphores (**mutex**, **empty**, and **full**) and a shared circular buffer to synchronize the producer and consumer threads.

Key Components:

1. **Circular Buffer:**
    - The circular buffer (**buffer**) is used to store produced items, and it is shared between the producer and consumer threads.
    - Indices **in** and **out** are used to manage the insertion and removal of items from the buffer in a circular manner.
2. **Semaphores:**
    - **mutex**: Protects the critical section when accessing the shared buffer.
    - **empty**: Represents the number of empty slots in the buffer. The producer waits on this semaphore when the buffer is full.
    - **full**: Represents the number of filled slots in the buffer. The consumer waits on this semaphore when the buffer is empty.
3. **Producer Thread:**
    - The producer thread runs an infinite loop, producing items and adding them to the buffer.

- It uses semaphores to control access to the shared buffer and to signal the availability of new items.

4. **Consumer Thread:**
    - The consumer thread also runs an infinite loop, consuming items from the buffer.
    - It uses semaphores to control access to the shared buffer and to signal when the buffer is not empty.
5. **Main Function:**
    - Initializes semaphores and creates the producer and consumer threads.
    - The main function waits for both threads to complete using **pthread_join**.

Synchronization and Race Condition Avoidance:

1. **Mutex Lock:**
    - The **pthread_mutex_t mutex** is used to ensure mutual exclusion when accessing the shared buffer.
    - It protects critical sections to avoid race conditions.
2. **Semaphores:**
    - **empty** and **full** semaphores are used to signal when the buffer has empty slots or filled slots, respectively.
    - These semaphores help in proper synchronization between the producer and consumer threads.
3. **Circular Buffer Indices:**
    - The **in** and **out** indices are manipulated in a way that allows the buffer to be used in a circular manner.

- This prevents overwriting items or accessing empty slots incorrectly.

Analysis:

1. **Correctness:**
    - The program demonstrates correct synchronization between the producer and consumer threads.
    - Mutex locks and semaphores are appropriately used to prevent race conditions and ensure proper coordination.
2. **Resource Management:**
    - The program efficiently uses semaphores to manage the availability of empty and filled slots in the buffer, preventing both overproduction and overconsumption.
3. **Infinite Loops:**
    - The threads run in infinite loops, which might not be suitable for all scenarios. Consideration should be given to introducing exit conditions or signals for a more controlled termination.
4. **Buffer Size:**
    - The buffer size is set to 5, but this can be adjusted based on the specific requirements of the application.

Conclusion:

The program provides a well-structured solution to the Producer-Consumer problem, employing mutex locks and semaphores for synchronization. Further enhancements could include introducing exit conditions for the threads and parameterizing the buffer size for flexibility.

GROUP PICTURE