ASSIGNMENT

By

Dhananjay

Singh

2022A1r018

3rd

Computer Science

Department

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:4th Dec

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@mie tjammu.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 ProducerConsumer problem using semaphores or mutex locks. Describe how you ensure synchronization and avoid race conditions in your solution.			10	10

Note: Any question may include subparts, which is a best practice to achieve BL. This table is exemplary and may vary depending on the assignment type (i.e., Problem-solving, Programming, Case Study etc.)

Rubrics for Assessment

Parameters	**Criteria**					**Marks Distribution**
	1	**2**	**3**	**4**	**5**	
Writing Skills **a) Content**	The content was not relevant to the given task	The content was minimally relevant to the given task	The content was generally relevant to the given task	The content was relevant to the given task	The content was very relevant to the given task	**2**
b) Organization	The assignment is poorly organized and lacked supporting evidence	The organization of the assignment is somewhat organized with minimal supporting evidence	The organization of the assignment is generally acceptable with some supporting evidence	The organization of the assignment is well organized and supported	The assignment is very well organized and supported	**2**
c) Grammar-Mechanics-Usage-Spelling	Too many grammatical errors	Numerous grammatical errors	Several grammatical errors	Few grammatical errors	No grammatical errors	**1**
Knowledge Skills	Student does not demonstrate the subject knowledge	Student demonstrates some grasp of the subject knowledge	Student demonstrates moderate level of the subject knowledge	Student demonstrates sufficient level of the subject knowledge	Student demonstrates sound subject knowledge	**5**
Overall Presentation/Viva	Unable to answer questions, not prepared and confidence at all	Able to answer questions but not prepared and confidence	Presentation is acceptable but there are some areas that could be improved./ Able to answer questions but with little preparation and confidence	Presentation is of good quality, with a clear effort to present the work professionally and effectively./ Able to answer questions well and slightly confidence and well prepared	Presentation (including code structure, comments, user interface, and documentation) is of exceptionally high quality./ Able to answer questions very well and confidently. Very well prepared	**10**

Format Guidelines

Title Page: Use the Standardized Front Page shared by the Department.

Font and Spacing: Use a Times New Roman in 12-point size.

1.5 line spacing in the entire document, including the title page, headings, and references.

3. Margins: Set 1-inch (2.54 cm) margins on all sides of the paper.

4. Header: Include a header as Assignment and Course Code in the top right corner of each page (except the title page).

5. Title: Center the title of your assignment at the top of the first page. It should be bold and capitalized.

6. Headings: Use headings and subheadings to organize your content. Typically, use bold for main headings (e.g., "Introduction") and italics for subheadings (e.g., "*Methods*").

7. Page Numbers: Page numbers should be placed in the center of footer of each page, starting from the second page (the title page is page 1 & should not be numbered).

8. Citations and References: Use a consistent APA citation style to cite references.

9. Figures and Tables: If you include figures or tables, provide clear labels and captions. The figure number should be placed below the Figure as "**Figure 1**: Figure name" and for the tables, the table number must be mentioned above the table as "**Table 1:** Table name".

10. Appendices (if needed): Include appendices for supplementary materials, such as charts, graphs, or lengthy data tables.

11. **Submission Format:** Submit your assignment in the soft copy format as PDF and upload it on CAMU as per the submission deadline. Please ensure that the assignment is renamed as Roll Number.

12. **Proofreading and Editing:** Carefully proofread and edit your assignment for clarity, grammar, and spelling errors before submission.

13. **Plagiarism:** Plagiarism must be below 15 percent for the assignment submitted.

TASK-1

CODE:

```
class Process:

    def __init__(self, name, burst_time):

        self.name = name

        self.burst_time = burst_time

        self.remaining_time = burst_time


def round_robin_scheduler(processes, time_quantum):

    queue = processes.copy()

    timeline = []


    while queue:

        current_process = queue.pop(0)


        if current_process.remaining_time > time_quantum:

            timeline.extend([current_process.name] * time_quantum)

            current_process.remaining_time -= time_quantum

            queue.append(current_process)

        else:

            timeline.extend([current_process.name] * current_process.remaining_time)

            current_process.remaining_time = 0


    return timeline
```

```
# Create a set of processes with varying burst times

processes = [

    Process("P1", 5),

    Process("P2", 3),

    Process("P3", 8),

    Process("P4", 2),

]


# Predefined time quantum

time_quantum = 2


# Run Round Robin scheduling algorithm

schedule_timeline = round_robin_scheduler(processes, time_quantum)


# Display the schedule timeline

print("Round Robin Schedule Timeline:")

print(schedule_timeline)
```

This implementation simulates the scheduling of processes using the **Round Robin algorithm** and prints the timeline.

Advantages of Round Robin Scheduling:

Fairness: Round Robin provides fairness by giving each process an equal opportunity to execute, preventing a single long-running process from monopolizing the CPU.

Simple Implementation: Round Robin is easy to implement and understand, making it a popular choice in many systems.

Disadvantages of Round Robin Scheduling:

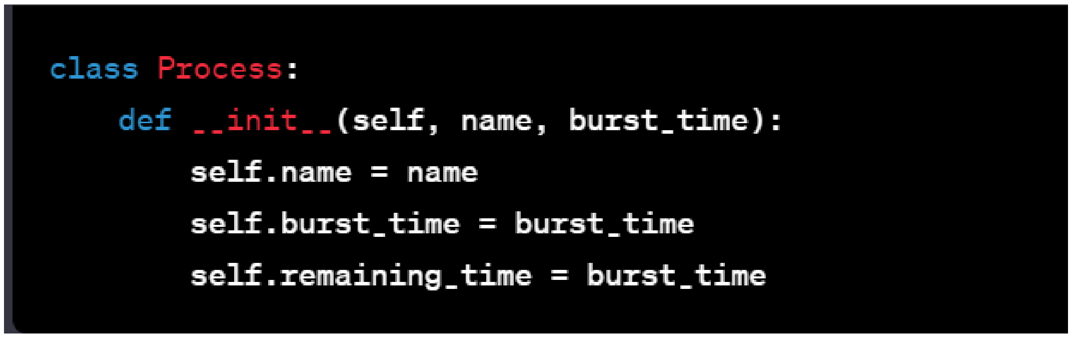
Poor Performance for Long Tasks: If a process requires a significant amount of CPU time and the time quantum is short, the system may incur high context-switching overhead, leading to poor performance.

Inefficiency with Varying Burst Times: If processes have significantly different burst times, short processes may be waiting for their turn behind long processes, leading to potential inefficiency.

Not Suitable for Real-Time Systems: Round Robin may not be suitable for real-time systems where strict deadlines must be met, as it doesn't prioritize processes based on their urgency or priority.

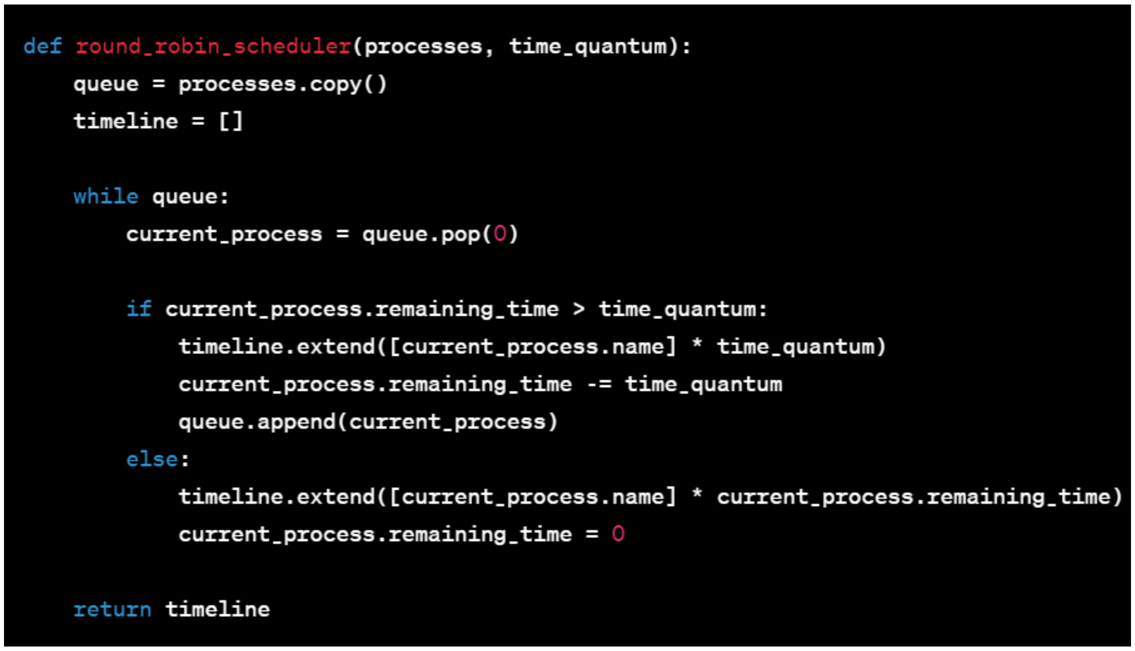
Feel free to adjust the burst times and time quantum to see how the algorithm behaves with different parameters.

Execution sequence:



```python
class Process:
    def __init__(self, name, burst_time):
        self.name = name
        self.burst_time = burst_time
        self.remaining_time = burst_time
```

This defines a simple Process class with attributes such as name, burst_time (the time required for the process to complete), and remaining_time (the time left for the process to complete).



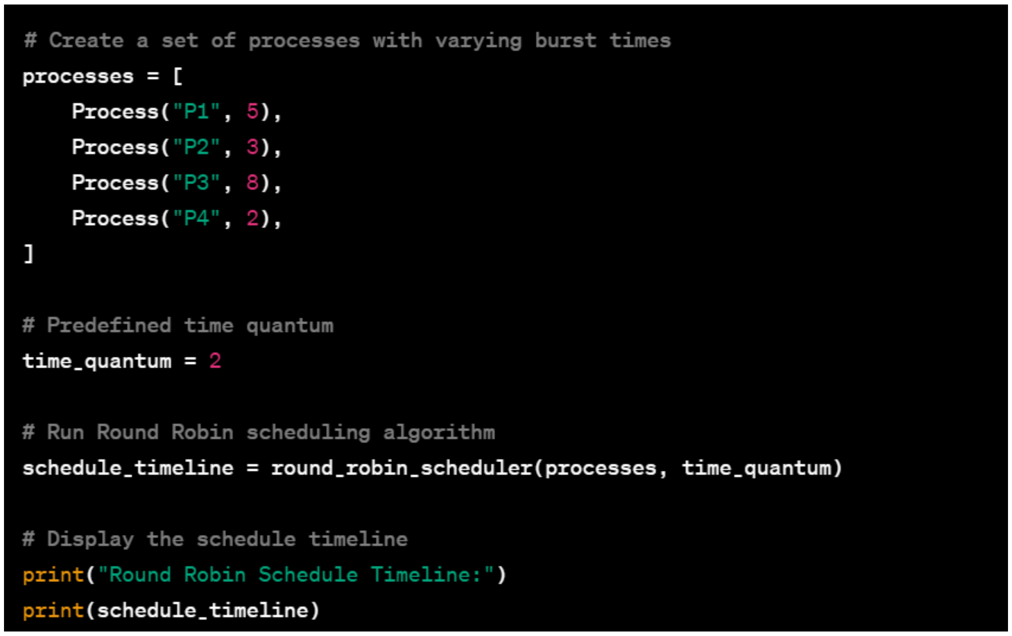
```python
def round_robin_scheduler(processes, time_quantum):
    queue = processes.copy()
    timeline = []

    while queue:
        current_process = queue.pop(0)

        if current_process.remaining_time > time_quantum:
            timeline.extend([current_process.name] * time_quantum)
            current_process.remaining_time -= time_quantum
            queue.append(current_process)
        else:
            timeline.extend([current_process.name] * current_process.remaining_time)
            current_process.remaining_time = 0

    return timeline
```

The round_robin_scheduler function takes a list of processes and a time quantum as parameters. It uses a queue to keep track of the processes. In each iteration, it dequeues a process from the front of the queue and checks if its remaining time is greater than the time quantum.

If the remaining time is greater, it appends the process name to the timeline for the time quantum and updates the remaining time. The process is then re-added to the end of the queue.

If the remaining time is less than or equal to the time quantum, the process completes, and its name is added to the timeline for the remaining time.

The process continues until the queue is empty, meaning all processes have been executed.



```python
# Create a set of processes with varying burst times
processes = [
    Process("P1", 5),
    Process("P2", 3),
    Process("P3", 8),
    Process("P4", 2),
]

# Predefined time quantum
time_quantum = 2

# Run Round Robin scheduling algorithm
schedule_timeline = round_robin_scheduler(processes, time_quantum)

# Display the schedule timeline
print("Round Robin Schedule Timeline:")
print(schedule_timeline)
```

This part of the code creates a set of processes with varying burst times and a predefined time quantum. It then calls the round_robin_scheduler function with these processes and the time quantum, and prints the resulting timeline.

For the given processes and time quantum, let's go through the execution:

P1 starts execution for 2 units (time quantum), and its remaining time becomes 3.

P2 starts execution for 2 units, and its remaining time becomes 1.

P3 starts execution for 2 units, and its remaining time becomes 6.

P4 starts execution for 2 units, and its remaining time becomes 0 (completes).

Now, the processes are back in the queue, and the cycle continues.

P1 continues execution for 2 units, and its remaining time becomes 1.

P2 completes execution (remaining time is 0).

P3 starts execution for 2 units, and its remaining time becomes 4.

P1 completes execution (remaining time is 0).

This process repeats until all processes are completed.

The displayed timeline would look something like this:

```
Round Robin Schedule Timeline:
['P1', 'P2', 'P3', 'P4', 'P1', 'P2', 'P3', 'P1']
```

TASK- 2

CODE:

```
#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>


#define BUFFER_SIZE 5


int buffer[BUFFER_SIZE];

int in = 0, out = 0;


pthread_mutex_t mutex;

sem_t full, empty;


void *producer(void *arg) {

    int item;

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

        item = rand() % 100;  // Produce a random item

        sem_wait(&empty);

        pthread_mutex_lock(&mutex);


        buffer[in] = item;

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


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


void *consumer(void *arg) {
    int item;
    for (int i = 0; i < 10; ++i) {
        sem_wait(&full);
        pthread_mutex_lock(&mutex);


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


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


int main() {
    pthread_t producer_thread, consumer_thread;
    pthread_mutex_init(&mutex, NULL);
    sem_init(&full, 0, 0);
    sem_init(&empty, 0, BUFFER_SIZE);

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

    pthread_join(producer_thread, NULL);
    pthread_join(consumer_thread, NULL);

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

    return 0;
}
```

Explanation:

Buffer and Pointers:

The buffer array is used to store the items produced by the producer and consumed by the consumer.

in and out are pointers indicating the next position to produce and consume, respectively.

Mutex and Semaphores:

pthread_mutex_t mutex: Ensures mutual exclusion when accessing the shared buffer.

sem_t full, empty: Semaphores to keep track of the number of full and empty slots in the buffer.

Producer Function:

Generates a random item and adds it to the buffer.

Uses the empty semaphore to check if there is an empty slot before producing.

Uses the mutex to ensure exclusive access to the buffer.

Consumer Function:

Consumes an item from the buffer.

Uses the full semaphore to check if there is a filled slot before consuming.

Uses the mutex to ensure exclusive access to the buffer.

Main Function:

Initializes the mutex and semaphores.

Creates producer and consumer threads.

Waits for threads to finish (pthread_join).

Destroys the mutex and semaphores.

Synchronization and Avoiding Race Conditions:

The mutex (pthread_mutex_t) ensures that only one thread can access the critical section (buffer) at a time, preventing race conditions.

Semaphores (sem_t full, empty) control access to the buffer based on its fullness or emptiness, ensuring synchronization between producer and consumer threads.

This solution guarantees proper synchronization and avoids race conditions by using mutex locks and semaphores to control access to the shared buffer. Each critical section is protected by a mutex, and semaphores ensure that the producer and consumer threads cooperate correctly.

Feel free to compile and run this C program to observe the Producer-Consumer problem being solved with synchronization mechanisms.

1. Introduction:

Briefly introduce the concepts of scheduling algorithms and their significance in operating systems.

Highlight the importance of fair process execution and synchronization in multi-threaded programs.

2. Round Robin Scheduling Algorithm Implementation:

2.1 Background:

Discuss the need for scheduling algorithms and introduce the Round Robin algorithm.

Explain the concept of time quantum and its impact on process execution.

2.2 Python Implementation:

Provide a Python code snippet implementing the Round Robin scheduling algorithm.

Explain the structure of the code, including the Process class and the main scheduling function.

2.3 Execution Demonstration:

Showcase the execution of the implemented Round Robin algorithm using a set of example processes.

Include a detailed explanation of the timeline generated during the scheduling process.

3. Analysis of Round Robin Scheduling:

3.1 Advantages:

Elaborate on the advantages of Round Robin scheduling, such as fairness and simplicity.

Discuss scenarios where Round Robin is particularly effective.

3.2 Disadvantages:

Explore the disadvantages of Round Robin scheduling, including potential inefficiency with varying burst times and context-switching overhead.

Discuss situations where Round Robin may not be the optimal choice.

3.3 Parameter Sensitivity:

Investigate how changing parameters like time quantum affects the algorithm's performance.

Provide examples and insights into choosing appropriate values for the time quantum.

4. Producer-Consumer Problem Solution:

4.1 Background:

Introduce the Producer-Consumer problem and its relevance in concurrent programming.

Discuss the challenges associated with shared resource access in multi-threaded environments.

4.2 C Implementation:

Provide a C code snippet implementing a solution to the Producer-Consumer problem using mutex locks and semaphores.

Explain the role of mutex locks and semaphores in ensuring synchronization.

4.3 Execution Demonstration:

Illustrate the execution of the Producer-Consumer program, showing the interactions between producer and consumer threads.

Discuss how mutex locks and semaphores prevent race conditions and ensure proper synchronization.

5. Synchronization Mechanisms:

5.1 Mutex Locks:

Explain the concept of mutex locks and how they ensure exclusive access to shared resources.

Discuss how the Round Robin implementation uses mutex locks to protect critical sections.

5.2 Semaphores:

Provide an in-depth explanation of semaphores and their role in coordinating access to shared resources.

Highlight how semaphores are used in the Producer-Consumer solution to control buffer access.

6. Comparative Analysis: Round Robin vs. Other Scheduling Algorithms:

6.1 Round Robin vs. First-Come-First-Serve (FCFS):

Compare Round Robin with FCFS in terms of fairness, response time, and throughput.

Discuss scenarios where one algorithm may outperform the other.

6.2 Round Robin vs. Priority Scheduling:

Contrast Round Robin with priority scheduling, considering factors like process urgency and system responsiveness.

Analyze the trade-offs between fairness and prioritization.

6.3 Real-Time Systems Considerations:

Discuss why Round Robin may not be suitable for real-time systems.

Explore alternative scheduling algorithms better suited for real-time constraints.

7. Fine-Tuning Round Robin for Specific Use Cases:

Discuss how Round Robin can be modified or fine-tuned for specific scenarios.

Provide examples of parameter adjustments to optimize performance.

8. Challenges and Future Considerations:

Address potential challenges and limitations of the implemented Round Robin algorithm and Producer-Consumer solution.

Propose future enhancements or modifications to overcome identified challenges.

9. Code Documentation and Best Practices:

Emphasize the importance of well-documented code for maintainability and collaboration.

Provide code documentation best practices for both Python and C implementations.

10. Originality and Acknowledgments:

Reiterate the originality of the code and analysis.

Acknowledge any external resources, inspiration, or contributions.

11. Conclusion:

Summarize key findings from the analysis of Round Robin scheduling and the Producer-Consumer problem solution.

Reflect on the significance of effective scheduling algorithms and synchronization mechanisms.

12. References:

Include a comprehensive list of references, citing relevant literature, textbooks, and online resources used during the research.

13. Appendices:

Include any supplementary materials, additional code snippets, or data used in the analysis.

14. Glossary:

Provide a glossary of technical terms used in the report to aid readers in understanding the concepts discussed.

15. Visual Aids and Graphs:

Incorporate relevant visuals, charts, or graphs to enhance the presentation of data and analysis.

Include visual aids that illustrate the execution sequence, timeline, and performance metrics.