

## School of Computer Science Engineering (SCOPE)

**B. Tech – Computer Science and Engineering**

**OPERATING SYSTEMS**

# Submitted By

**Naveen(22BRS1343)**

**Abubakar(22BRS1203)**

**Dhrrishith(22BAI1458)**

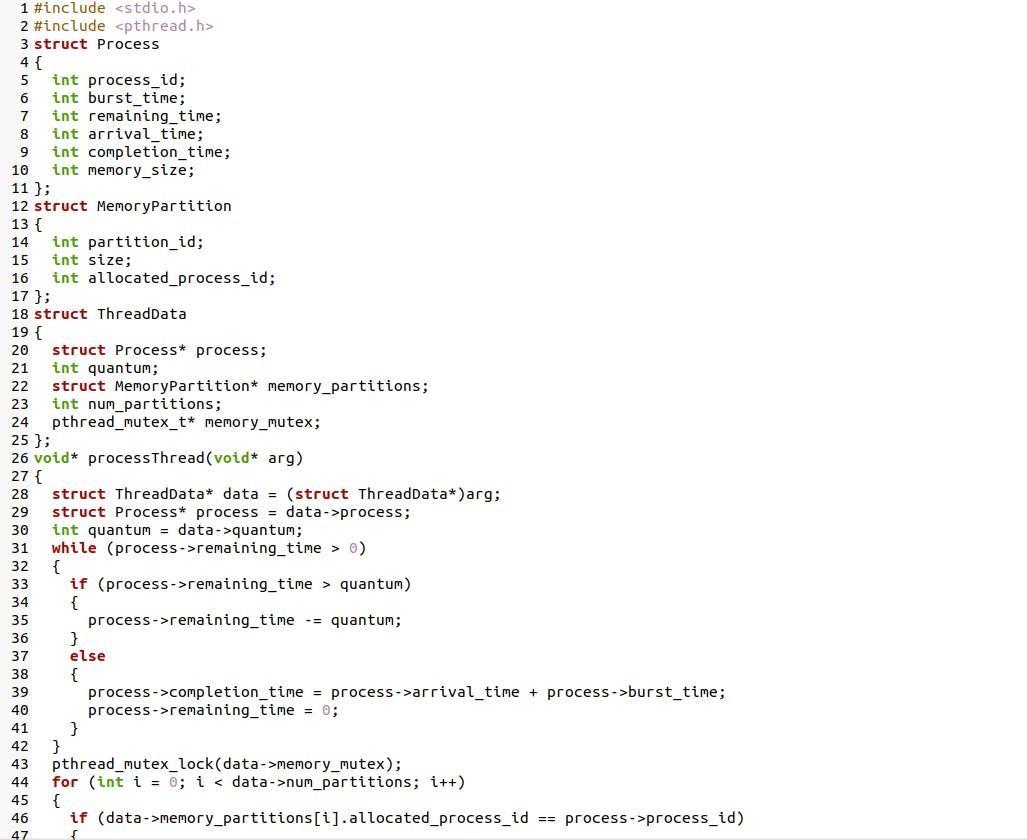
# Submitted To Dr. K.Vallidevi

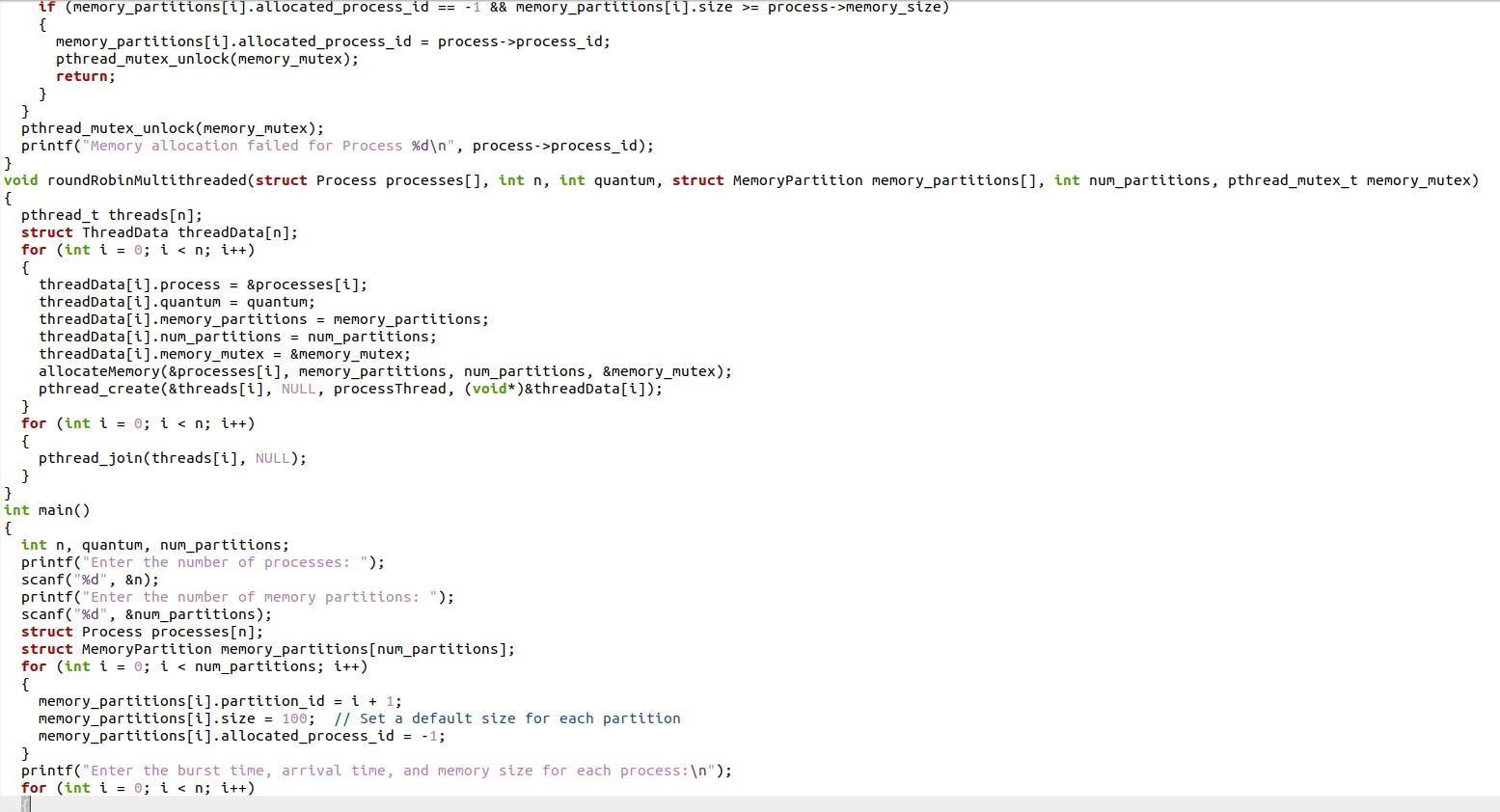
**DATE: 09/11/2023**

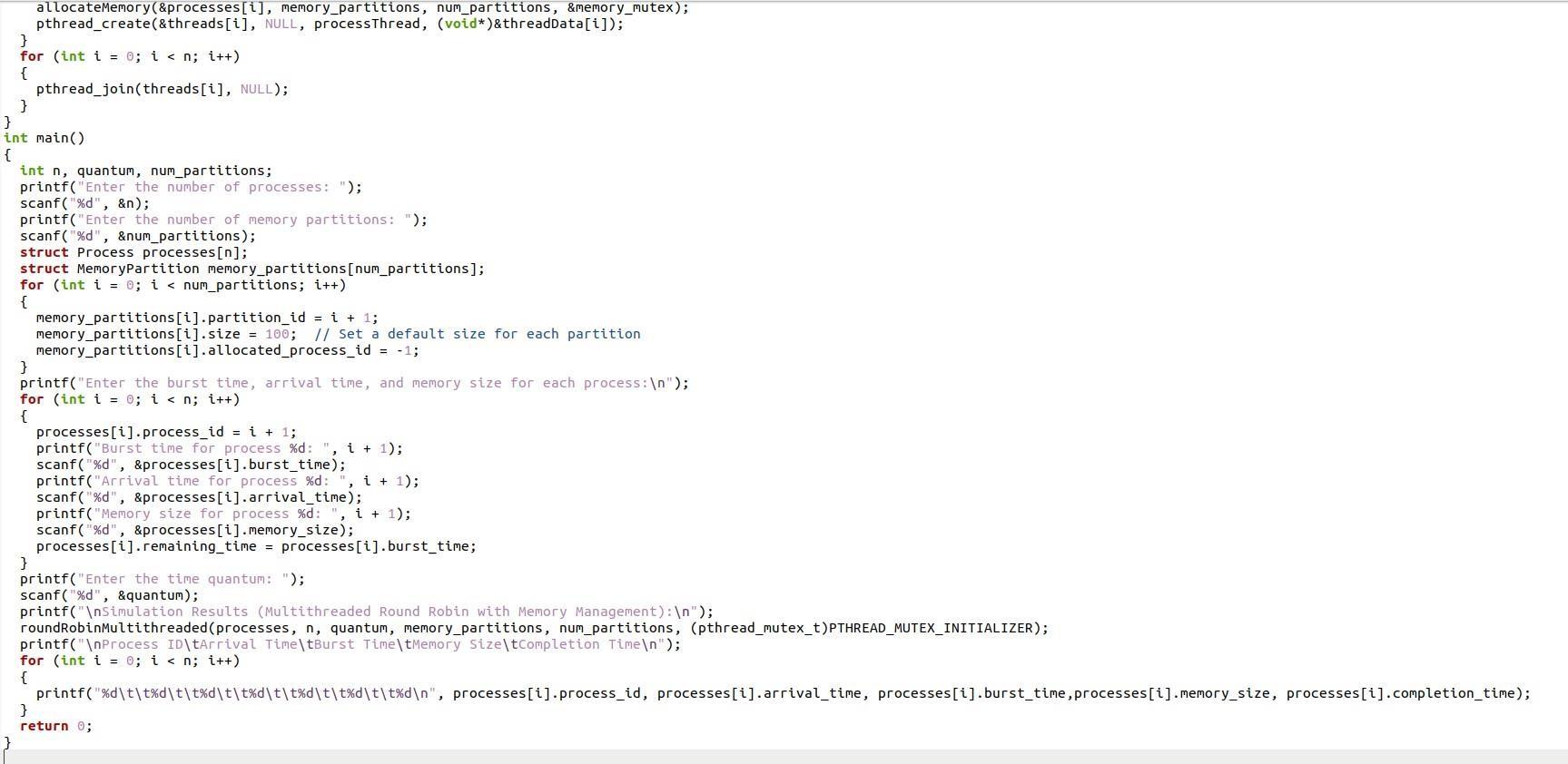
**CPU PROCESS SCHEDULING**

Modern operating systems employ sophisticated mechanisms for process scheduling, multithreading, and memory management to optimize resource utilization and enhance system performance. This combined program integrates these crucial components, offering a comprehensive simulation of a dynamic computing environment. The program incorporates a Round Robin scheduling algorithm for process execution, multithreading to parallelize tasks, and memory management to allocate and deallocate memory for processes. Through user-defined inputs for process attributes, time quantum, and memory partitions, the program simulates the execution of multiple processes in a multithreaded environment, considering memory constraints. The implementation addresses challenges such as race conditions, deadlocks, and memory fragmentation through synchronization mechanisms and appropriate algorithms. The simulation results provide insights into the performance of the system, including process completion times, memory usage, and the impact of different scheduling strategies. This program serves as a valuable tool for understanding the intricacies of operating system concepts and experimenting with various configurations to optimize system behavior**.**

### INPUT



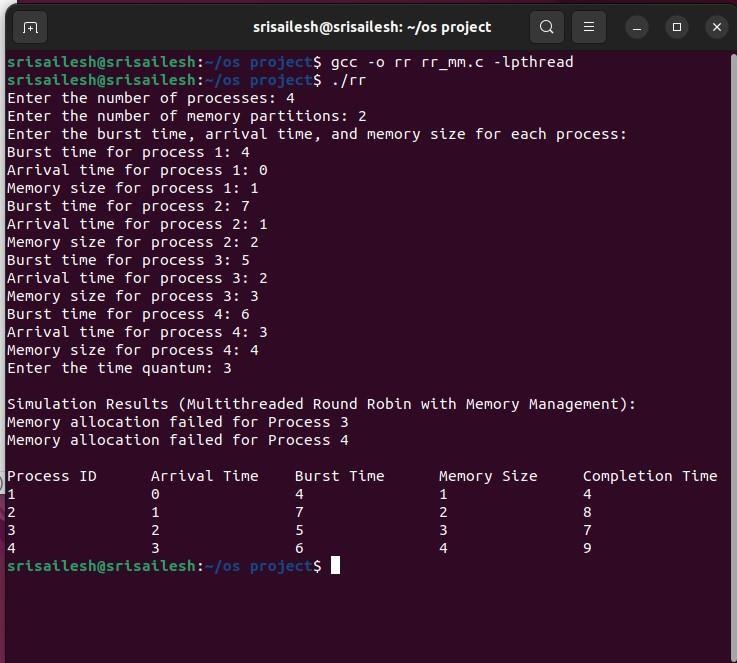




**PSEUDOCODE/ALGORITHM**

1. Define structure for Process.
2. Define structure for Memory Partition.
3. Define structure for Thread Data.
4. Function to simulate the execution of a single process in a thread.
5. Implement Round Robin scheduling for a single process
6. Simulate memory deallocation.
7. Exit thread
8. Function to simulate memory allocation for a process.
9. Function to perform multithreaded Round Robin scheduling with memory management.
10. Create threads for each process.
11. Allocate memory for each process.
12. Create thread
13. Wait for all threads to finish
14. Create the Main function
15. Get user input for the number of processes
16. Get user input for the number of memory partitions
17. Initialize arrays for processes and memory partitions
18. Initialize memory partitions
19. Get user input for burst time, arrival time, and memory size for each process
20. Get user input for time quantum
21. Display simulation results

### OUTPUT



In this program involving process scheduling, multithreading, and memory management, several problem statements may arise. Here are some common issues and their potential solutions:

#### Race Conditions:

**Problem** - Concurrent access to shared resources, such as memory partitions or process data, may lead to race conditions and unpredictable behavior.

**Solution** - Use synchronization mechanisms like mutexes to control access to shared resources, ensuring that only one thread can modify or access them at a time.

#### Deadlocks:

**Problem** - In a multithreaded environment, deadlocks may occur if processes are waiting for resources held by other processes, creating a circular dependency.

**Solution -** Implement deadlock detection and recovery mechanisms. Avoid circular waiting, and release resources if a process cannot obtain the required ones.

#### Memory Fragmentation

**Problem** - Continuous allocation and deallocation of memory may lead to fragmentation, making it challenging to allocate large contiguous memory blocks.

**Solution -** Implement memory compaction or defragmentation strategies to reduce fragmentation. Use memory allocation algorithms that minimize fragmentation, such as buddy allocation or best-fit allocation.

#### Insufficient Memory:

**Problem** - If the memory requested by a process exceeds the available memory, allocation failures may occur.

**Solution** - Implement memory allocation policies that consider process size and available memory. Use techniques like paging or virtual memory to provide the illusion of larger memory space.

#### Inefficient Scheduling:

**Problem** - The chosen scheduling algorithm may be inefficient or not suitable for the workload.

**Solution** - Experiment with different scheduling algorithms based on the system requirements. Consider factors like turnaround time, waiting time, and response time when selecting a scheduling algorithm.

#### Inaccurate Timing:

**Problem** - The simulation may not accurately represent the timing of real-world systems.

**Solution** - Use more advanced simulation techniques, incorporate system clocks, and consider factors like context switching overhead for a more realistic simulation.

#### Un-optimized Multithreading:

**Problem -** The multithreading implementation may not be optimized, leading to poor performance.

**Solution** - Optimize the multithreading code by considering factors such as thread creation overhead, load balancing, and minimizing thread contention.

#### User Input Errors:

**Problem** - Incorrect or malicious user inputs may lead to unexpected behavior or crashes.

**Solution** - Implement robust input validation to handle erroneous inputs gracefully. Provide meaningful error messages and ensure that the program remains stable in the face of unexpected input.

#### Lack of Error Handling:

**Problem -** Insufficient error handling may make it challenging to identify and recover from unexpected situations.

**Solution** - Implement comprehensive error handling mechanisms to gracefully handle errors, log relevant information, and inform the user about the issue.

Addressing these issues requires careful design, thorough testing, and consideration of various system parameters. It's crucial to strike a balance between complexity and simplicity while ensuring the program meets its intended goals.

**Realtime Examples:**

The provided code, which combines process scheduling, multithreading, and memory management, is a simulation rather than a real-world application. However, the concepts and functionalities it covers can be applied to various scenarios in the context of operating systems. Here are some real-world examples or analogies where similar principles are applied:

1. **Cloud Computing Resource Management:**

- Simulating how a cloud service provider manages multiple virtual machines (processes) with varying resource demands (memory size) using a scheduling algorithm and multithreading.

2. **Parallel Processing in Scientific Computing:**

- Representing a scientific application where complex simulations or calculations are divided into parallel tasks (threads) for efficient execution, while managing memory requirements.

3. **Web Server Handling Multiple Requests:**

- Simulating a web server that handles multiple incoming requests (processes) concurrently by using a scheduling algorithm, ensuring fair access to resources, and managing memory for each request.

4. **Embedded Systems with Limited Resources:**

- Emulating an embedded system scenario where limited memory is allocated for different tasks (processes) and ensuring optimal execution using a scheduling algorithm and efficient memory management.

5. **Multitasking Operating System:**

- Reflecting the operation of a multitasking operating system where multiple applications (processes) run concurrently, each requiring its share of CPU time and memory.

6. **Video Game Engine Optimization:**

- Simulating a video game engine handling various game entities (processes) and their interactions concurrently using multithreading, with memory management for efficient resource usage.

7. **Parallel Database Query Processing:**

- Representing a scenario in a database system where parallel processing and efficient memory allocation are employed to handle multiple queries concurrently, optimizing query execution.

8. **Autonomous Vehicle Systems:**

- Emulating an autonomous vehicle system where different subsystems (processes) operate concurrently, with a scheduling algorithm ensuring timely execution and memory management for efficient data handling.

It's important to note that while the provided code is a simplified simulation, real-world applications often involve more complexity, security considerations, and specific optimizations tailored to the particular use case. The principles demonstrated in the code, however, align with broader concepts found in operating systems and concurrent programming.