**Report on Process Management Simulation Code**

***Introduction***

This report details a C program designed to simulate a basic process management system in an operating system. The program includes functionalities such as process creation, memory management, semaphore-based synchronization, and a round-robin scheduling algorithm. The simulated system enables multiple processes to be loaded, executed, and managed efficiently, providing insights into fundamental OS concepts.

***Methodology***

1. Queue Management: In our program, we've established several key functions to efficiently manage the queues where processes reside. The initQueue function initializes the queues for ready and blocked processes, ensuring they're ready for use. With isQueueEmpty and isQueueFull, we can easily check whether a queue is empty or at full capacity, helping us make informed decisions during process management. The enqueue function adds processes to a queue, while dequeue removes processes from the queue, ensuring a smooth flow of processes through our system.

2. Memory Management: Memory management is a critical aspect of our program, ensuring that processes have the necessary space to execute their tasks. The allocateMemory function plays a pivotal role here, as it allocates memory for a process, allowing it to store instructions and variables. Additionally, storeInstructions enables processes to store their instructions in allocated memory, while storeVariables and retrieveVariable facilitate the management of process variables, ensuring they can be stored and retrieved as needed.

3. Instruction Execution: To execute the various instructions within our simulated environment, we've developed dedicated functions for each instruction type. For instance, instructions like assign, print, writeFile, readFile, printFromTo, semWait, and semSignal are all handled by specific functions tailored to their respective tasks. This modular approach enhances code readability and simplifies the execution of diverse instructions within our system.

4. Process Control: Effective process control is essential for ensuring smooth execution and state transitions within our program. The executeProcess function serves as the core engine, driving the execution of the current instruction for a given process. This function handles all instruction types and manages state transitions seamlessly, ensuring processes progress through their lifecycle as expected. Additionally, executeProgram is responsible for parsing and executing instructions from program files, enabling the seamless integration of external processes into our system.

5. Blocking and Unblocking: In scenarios where processes need to wait for specific resources, functions like blockProcess and unblockProcesses come into play. These functions manage processes waiting for particular resources and handle state transitions accordingly. By efficiently managing the blocking and unblocking of processes, we ensure optimal resource utilization and prevent bottlenecks within our system. Through these carefully crafted functions, our program maintains robust control over process execution and resource allocation, contributing to its overall efficiency and reliability.

The workflow of the program begins with initialization, where queues for ready and blocked processes are set up. Following this, the program moves into the loading phase, where it parses command-line arguments to load program files and establish arrival times for each process. During this step, every process is meticulously created, initialized, and stored for subsequent execution. Once the setup is complete, the program enters the main execution loop, a crucial stage where it continuously monitors for process arrivals. Upon detection, these processes are promptly enqueued into the ready queue, preparing them for execution. Here, the program employs a round-robin scheduling algorithm to manage process execution efficiently. This involves executing instructions, managing context switching using time slices, and ensuring synchronization using semaphores. Throughout this loop, the ready and blocked queues are dynamically updated to reflect the current state of processes. Finally, the program handles output management, where functions like printReadyQueue and printBlockedQueue come into play, offering insights into the status of queues for debugging and visualization purposes. Through this meticulously designed workflow, the program orchestrates the execution and management of processes within an operating system simulation.

***Conclusion***

This program provides a foundational simulation of process management within an operating system, illustrating key concepts such as process scheduling, memory management, and resource synchronization. By implementing the suggested enhancements, the program can be made more robust, scalable, and maintainable, offering richer insights and more reliable performance in process management simulations.

|  |  |
| --- | --- |
| Tareq Osama Ahmed Hassanien | 52-1734 |
| Ahmed Sherif Ahmed Ismail | 50-40004 |
| Ismail Ehab Ismail Aboelfath | 52-15361 |
| Nourhan Ashraf Hussien Abdelaziz Mohamed | 49-3924 |
| Omar Mohamed Abdelwahab Mahmoud Alian | 46-6219 |
| Marwan Tarek Mohamed Abdelfattah Elnady | 46-8998 |

**Team\_99**