OS Project Report

Abdelrahman Elsamalouty(55-0842) T-21

Abdelrahman Elaby(55-0990) T-21

Habiba Hesham(55-0966) T-21

Hazem Mansour(55-1036) T-21

Zeina Mohamed(55-1095) T-21

Ehab Medhat(55-4571) T-21

Team Number: 20

1 Introduction

Our program is a simulation of a simple process scheduler and memory manager, whose main goal is to handle a number of processes, each described by a Process Control Block (PCB) and run under a multi-level feedback queue. Moreover, it provides mutex-based synchronization to resolve resource conflicts of the processes. It even provides a simulation of process generation, execution, and memory management.

2 Methodology

The code uses several core data structures and functions to achieve its functionality:

2.1 Data Structures

- **PCB (Process Control Block)**: Each process is represented by a PCB that carries its ID, state("READY", "RUNNING", "TERMINATED", "BLOCKED"), priority, program counter, and memory allocation information.
- **Queue**: Queues are used to manage PCBs into different states; ready and blocked. Each queue has a fixed size and supports basic operations like enqueue and dequeue.
- **Mutex**: It represents synchronization primitive having a binary state, locked or unlocked, and an owner's processID. Each mutex also contains a queue for processes waiting on the mutex.
- **Element**: It is used to represent a unit of memory and stores the name of the element and the corresponding data. The name of the element could be one of the following, "PCB", "lineOfCode", "variable".

2.2 Functions and Their Functionalities

- **print_PCB**: This prints the information of a given PCB Enqueue and dequeue. These manage PCBs in the queue and add and remove PCBs respectively.
- **semWait** and **semSignal**: These implement semaphore operations in order to control access to resources and block and unblock processes as required.
- **read_program_to_memory**: It reads the given text file into memory locations within PCB boundaries.

- **create_process**: It initializes a new process by assigning it memory and setting its initial state. Once created, the process is enqueued into the first ready priority queue.
- **get_variable** and **store_variable**: It reads and writes variables in a process's memory space.
- **execute_program**: It runs the next instruction of the given process. Tokenizes the instruction string to determine the operation (e.g., "print", "assign", "semWait"). It also checks if the program counter has reached the end of the program. If so, sets the process state to TERMINATED and prints a termination message. Otherwise, increments the program counter to fetch the next instruction in the next execution cycle.
    1. Print: Fetches a variable's value from the PCB and prints it.
    2. Assign: Assigns a value to a variable. If the value is "input", it prompts the user for input. If the value is from a file, it reads the file's content and assigns it to the variable.
    3. Semaphore Operations (semWait, semSignal): Manages access to shared resources like user input, user output, or files using semaphores.
    4. PrintFromTo: Prints numbers in a range specified by two variables.
    5. WriteFile: Writes the content of a variable to a file.
- **run_scheduler**: It applies the scheduling algorithm and selects the next process to be run based on priority and quantum.
- **printMemory**: It prints the current state of memory, showing the PCB, lines of code and variables stored.

2.3 Pseudo Algorithm

```
Initialize global variables and mutexes
Read arrival times for three programs

While there are processes to run or new processes arriving
    Increment cycle counter
    If the current cycle matches the arrival time of a program
        Create a new process
    Run the scheduler to select and execute the next process

Scheduler algorithm:
    If there is a running process
        Keep running as long as quantum hasn't finished
    Else
        Select the highest priority non-empty ready queue
        Dequeue the next process from the selected queue
        Run the selected process
        If the process is still running and its quantum is not finished
            Continue running it
        Else
            Move the process back to an appropriate queue based on its status

Print the final state of memory and processes
```

3 Results

During the implementation and testing of this code, there were several challenges that were faced and fixed:

3.1 Issues and Resolutions

- **Priority**:Changing priority of processes that finished its quantum at the same clock cycle it was blocked in. This was fixed by adding conditions to check if quantum was finished before changing the state of the process to be blocked.
- **Carriage Return Issue**: Text files had a carriage return at the end of each line which we didn't know about causing problems while parsing. This was fixed by using C pre-defined methods that removed the carriage return character at the end.
- **Memory Overflow**: Initially, trying to load programs in memory caused the program to overflow. This was fixed by adding checks so that it does not use more memory than the previously defined `MEMORY_SIZE`.

- **Queue Management**: Errors about queue management were faced, like dequeuing from an empty queue. More error handling was added to those edge cases.

3.2 Final Results

The system finally handled the process creation, execution, and memory operations after the problem was fixed. The processes were scheduled according to their priorities, and the synchronization between the processes was done properly with the usage of mutexes. The memory management system was correctly allocating and deallocating memory for the processes and variables.