The German University in Cairo (GUC)
Faculty of Media Engineering and Technology
Computer Science and Engineering
Operating Systems - CSEN 602

Milestone 02 Report

Team Members :

58-0464	**Aesha Anwar Sherif**
58-1001	**Walid Moussa Khalil**
58-1034	**Jomana Mahmoud Abdelmigid**
58-1857	**Sara Ahmed Elsawy**
58-2571	**Yehia Hassan Sadek**
58-3703	**Nada Yasser**
58-25160	**Rawan Hossam**

Team Number :

Team 64

Under Supervision of :

Dr. Eng. Catherine M. Elias

Dr. Eng. Aya Salama

Spring 2025

Contents

Chapter 1

Project Objectives

The goal of Milestone 2 is to build a simplified simulation of how an operating system manages processes, memory, and shared resources. This milestone focuses on creating a custom scheduler (First come first serve, Round robin and MLFQ), handling memory allocation, and ensuring proper control over critical sections using mutexes.

More specifically, we aim to:

1. **Develop a Scheduler**
   Create a scheduler that can switch between three different scheduling algorithms: First Come First Serve (FCFS), Round Robin with a user-defined quantum, and a Multilevel Feedback Queue (MLFQ) with increasing quantum values at lower levels.

2. **Execute Custom Programs**
   Build an interpreter that reads and executes code from three provided text files. Each file represents a program, and every instruction is executed in one clock cycle.

3. **Manage Limited Memory**
   Use a fixed-size memory (60 words) to store each process's code, variables, and PCB. Each process must be placed in memory only at its arrival time and must stay within its assigned memory boundaries.

4. **Use PCBs for Process Tracking**
   Maintain a Process Control Block (PCB) for every process to track important data like its ID, state, priority, memory limits, and current instruction.

Chapter 1

5. **Control Shared Resources with Mutexes**
   Implement mutual exclusion using semaphores (semWait and semSignal) to make sure shared resources like files and input/output are accessed safely by one process at a time.

6. **Simulate Concurrent Process Execution**
   Ensure that multiple processes can run in a way that mimics concurrency, while avoiding conflicts over shared resources and keeping the system's behavior predictable.

Chapter 2

Introduction

This milestone simulates the core functionalities of an operating system, including process scheduling, memory management, and resource synchronization. The main objective is to develop a system that can load and execute custom-written programs, allocate them in memory, and manage their execution through different scheduling algorithms.

We implemented a scheduler that supports First Come First Serve (FCFS), Round Robin (RR) with user-defined quantum, and a Multilevel Feedback Queue (MLFQ) algorithm. Each process is assigned a portion of memory, along with a Process Control Block (PCB) that tracks key information such as its state, program counter, and memory limits.

To handle shared resources like files and user input/output, we incorporated mutex mechanisms using semaphores (`semWait` and `semSignal`) to ensure mutual exclusion. The interpreter we developed reads commands from text files and executes them line by line, simulating real process instructions.

Overall, this project provides a simplified but practical demonstration of how operating systems control multiple processes, allocate memory, and prevent race conditions in shared environments.

Chapter 3

Methodology

The system is designed to simulate process scheduling and memory management in an operating system while offering a GUI interface. The system includes process management, memory management, scheduling algorithms, mutexes for resource control, I/O operations, and a graphical representation of the system. The main components are:

- **Process Management**
- **Memory Management**
- **Scheduling (FCFS, Round Robin, Multilevel Feedback Queue)**
- **Mutexes and Resource Management**
- **I/O Management**
- **Graphical User Interface (GUI) using GTK**
- **System Initialization**
- **Instruction Execution**

1. Process Management

- **Process Control Block (PCB)**: Each process is represented by a `PCB` structure, containing information about its state (READY, RUNNING, BLOCKED), program counter (PC), memory bounds, and priority.

Chapter 3

- **Process Creation**: Processes are initialized with a PID, priority, arrival time, and associated instruction file using the method `initializeProcess()`.
- Processes are queued based on arrival time, and their respective program instructions are loaded into memory.

- **Queue Management**: We use a queue-based system to manage processes in different states. There are two primary queues:

    - **Ready Queue (`readyQueue`)**: Stores processes ready for execution.

    - **Blocked Queue (`globalBlockedQueue`)**: Stores processes that are blocked waiting for a resource.

2. Memory Management

- **Memory Structure**: Memory is represented as an array of `MemoryCell` structures, where each cell holds an instruction or variable.

- **Memory Allocation**: Each process has a defined memory range (from `memoryLowerBound` to `memoryUpperBound`), where it can store variables and instructions.

- **Execution of Instructions**: Instructions are fetched and executed by the CPU, one at a time. They can manipulate variables stored in memory, interact with I/O (e.g., printing or reading from files), and perform other operations such as assignments and file writing.

Chapter 3

3. Scheduling Algorithms

- **First Come First Serve Scheduling**: The ready queue is managed as a simple FIFO (First In, First Out) queue. Processes are executed based on their arrival time, and the system does not interrupt a running process until it finishes.

- **Round Robin Scheduling**: The system uses a round-robin approach to assign CPU time to processes in the ready queue. The quantum is input, and when a process's time slice is exhausted, the process is preempted and it is moved back to the ready queue.

- **Multilevel Feedback Queue**: This algorithm assigns processes to queues based on priority. The system attempts to prioritize high-priority processes but can demote processes if they consume too much CPU time (Round Robin policy).

4. Mutexes and Resource Management

- **Mutexes**: The system implements mutual exclusion (mutex) to control access to shared resources (user input, file access, and output).

- **Semaphore-based Wait and Signal**: Processes that require a locked resource wait in a blocked queue. When a resource is released (via `semSignal`), the first waiting process is moved back to the ready queue, resuming its execution.

- **Blocked Queue Management**: Processes blocked on a resource are placed in their respective mutex blocked queues. A global blocked queue is used for overall tracking and process management.

Chapter 3

5. I/O Management

- **File I/O Operations**: The system includes file handling functions that allow processes to read from and write to files. `readFile` and `writeFile` operations are implemented to interact with memory, and these operations block processes that are waiting for access to the files.

- **User Input**: Processes can request user input, and the system blocks the process until the input is received.

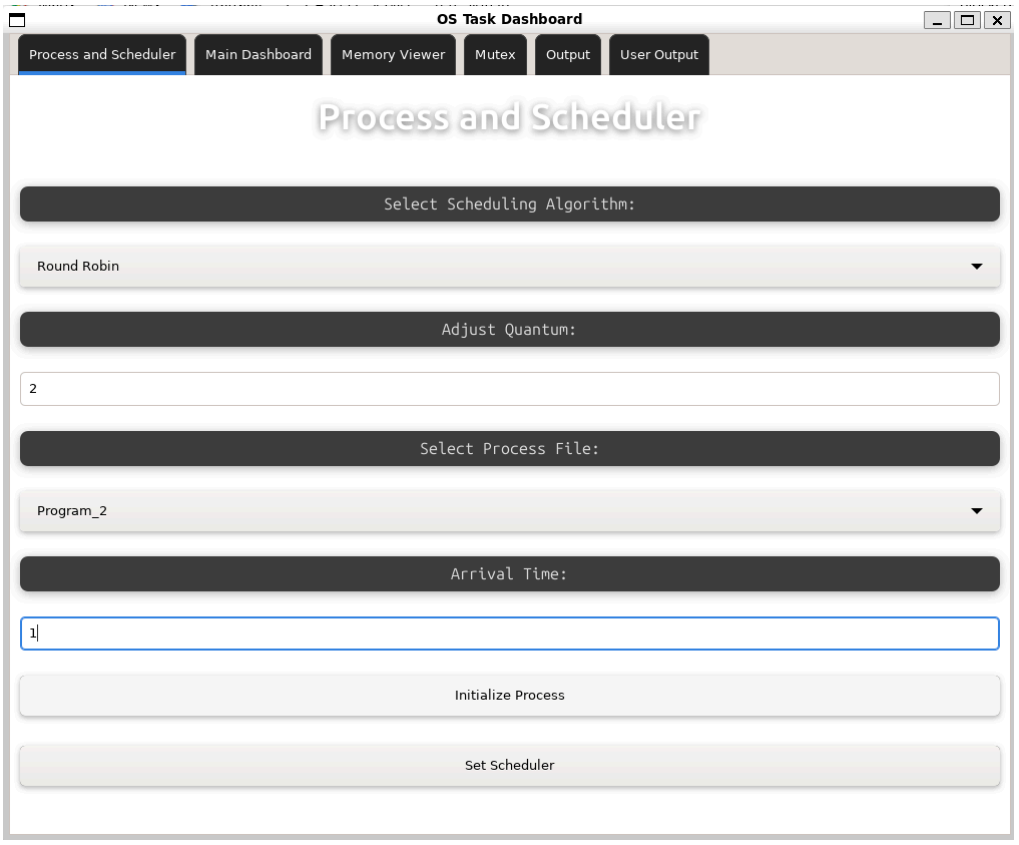
6. Graphical User Interface (GUI) with GTK

- **GUI Implementation**: The system uses the GTK toolkit for the creation of the graphical user interface. The GUI displays real-time information about processes, memory, and the current scheduling state. It allows users to visualize process states, queues, memory usage, and mutex states in an interactive manner. The interface is implemented in `dashboard.c`,`memory_viewer_tab.c`, `mutex_tab.c` and `process_scheduler_tab.c` .

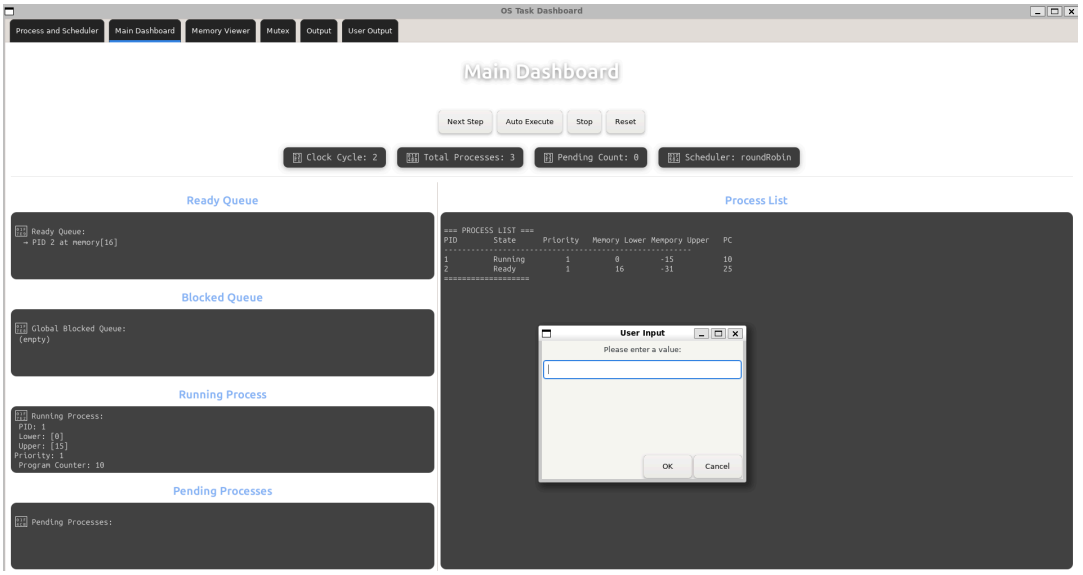
- **The GUI has 5 tabs:**
  - **Process & Scheduler:** To initialize processes with their arrival time and choose the scheduling algorithm.
  - **Main Dashboard:** To display ready and general blocked queue, pending processes and process list. Have Buttons to choose between Auto and Step-by-step execution.
  - **Memory Viewer:** To display the content of 60 cells of memory.

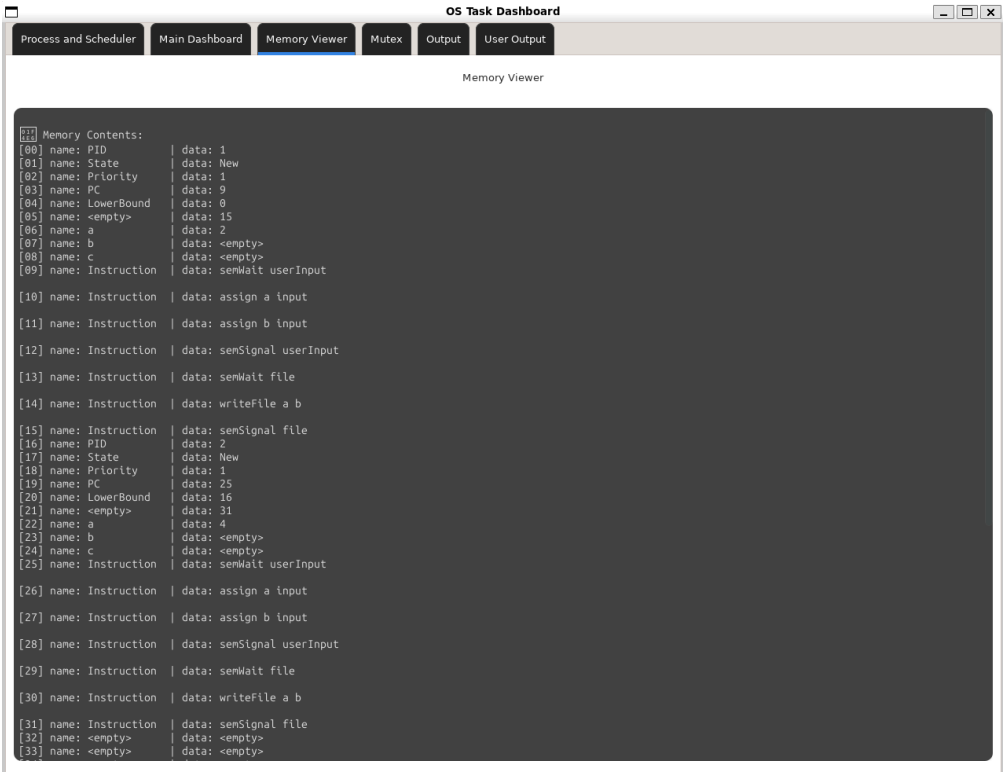
Chapter 3

- **Mutex:** To display resources' blocked queue and status of each mutex.
- **Output:** To display executed instructions and status of processes.
- **User Output:** output messages to be printed according to the instructions of each program.

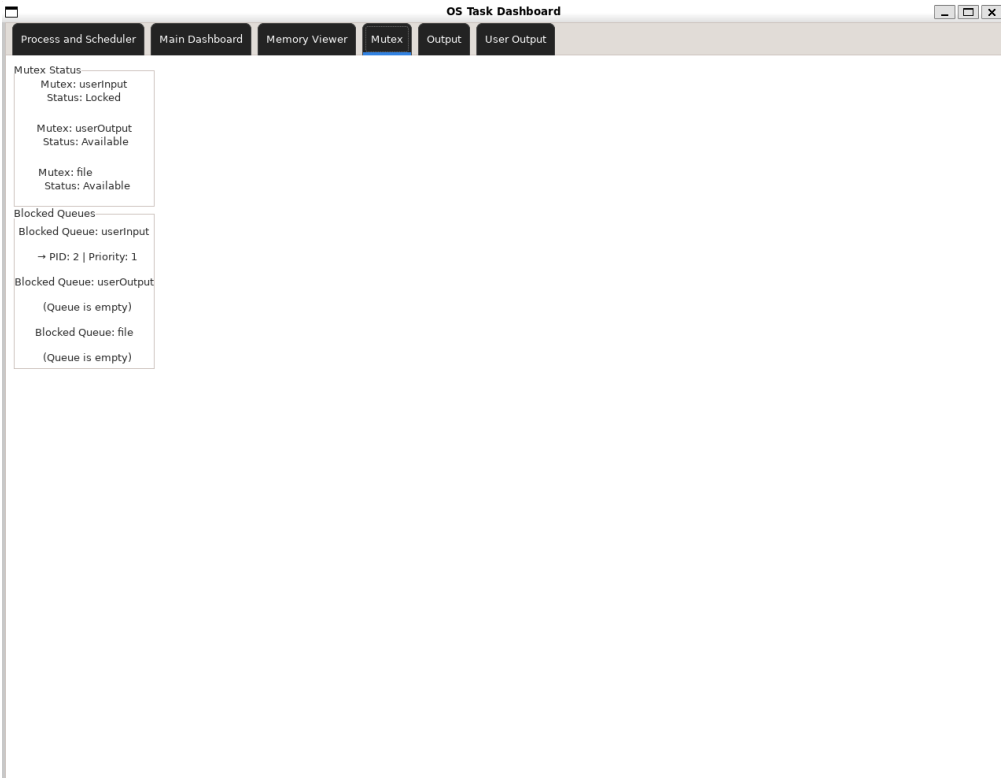


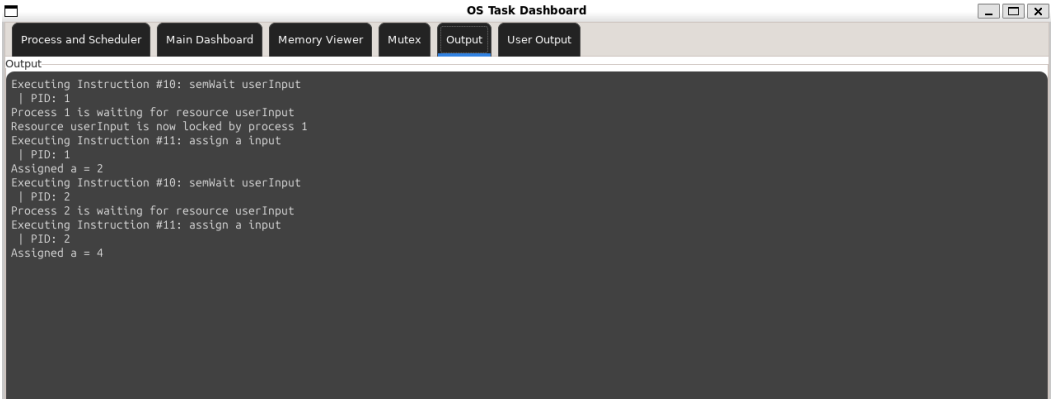
Chapter 3

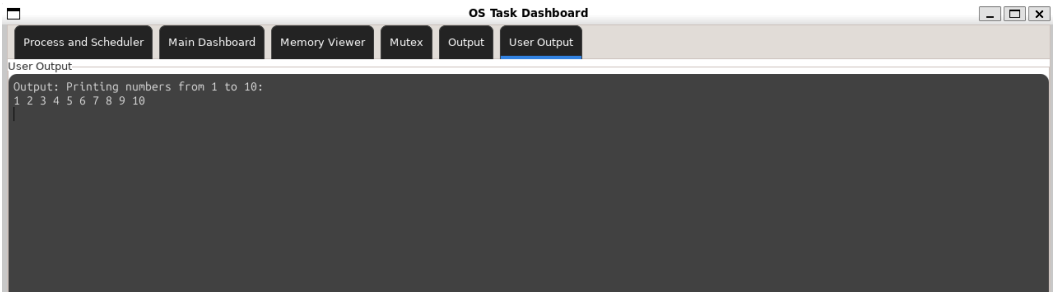

OS Task Dashboard
Process and Scheduler
Main Dashboard
Memory Viewer
Mutex
Output
User Output
Main Dashboard
Next Step
Auto Execute
Stop
Reset
Clock Cycle: 2
Total Processes: 3
Pending Count: 0
Scheduler: roundRobin
Ready Queue
Ready Queue:
→ PID 2 at memory[16]
Blocked Queue
Global Blocked Queue:
(empty)
Running Process
Running Process:
PID: 1
Lower: [0]
Upper: [15]
Priority: 1
Program Counter: 10
Pending Processes
Pending Processes:
Process List
=== PROCESS LIST ===
PID State Priority Memory Lower Memory Upper PC
1 Running 1 0 15 10
2 Ready 1 16 31 25
User Input
Please enter a value:
OK
Cancel


OS Task Dashboard
Process and Scheduler
Main Dashboard
Memory Viewer
Mutex
Output
User Output
Memory Viewer
Memory Contents:
[00] name: PID | data: 1
[01] name: State | data: New
[02] name: Priority | data: 1
[03] name: PC | data: 9
[04] name: LowerBound | data: 0
[05] name: <empty> | data: 15
[06] name: a | data: 2
[07] name: b | data: <empty>
[08] name: c | data: <empty>
[09] name: Instruction | data: semWait userInput
[10] name: Instruction | data: assign a input
[11] name: Instruction | data: assign b input
[12] name: Instruction | data: semSignal userInput
[13] name: Instruction | data: semWait file
[14] name: Instruction | data: writeFile a b
[15] name: Instruction | data: semSignal file
[16] name: PID | data: 2
[17] name: State | data: New
[18] name: Priority | data: 1
[19] name: PC | data: 25
[20] name: LowerBound | data: 16
[21] name: <empty> | data: 31
[22] name: a | data: 4
[23] name: b | data: <empty>
[24] name: c | data: <empty>
[25] name: Instruction | data: semWait userInput
[26] name: Instruction | data: assign a input
[27] name: Instruction | data: assign b input
[28] name: Instruction | data: semSignal userInput
[29] name: Instruction | data: semWait file
[30] name: Instruction | data: writeFile a b
[31] name: Instruction | data: semSignal file
[32] name: <empty> | data: <empty>
[33] name: <empty> | data: <empty>

Chapter 3


OS Task Dashboard
Process and Scheduler
Main Dashboard
Memory Viewer
Mutex
Output
User Output
Mutex Status
Mutex: userInput
Status: Locked
Mutex: userOutput
Status: Available
Mutex: file
Status: Available
Blocked Queues
Blocked Queue: userInput
→ PID: 2 | Priority: 1
Blocked Queue: userOutput
(Queue is empty)
Blocked Queue: file
(Queue is empty)


OS Task Dashboard
Process and Scheduler
Main Dashboard
Memory Viewer
Mutex
Output
User Output
Output
Executing Instruction #10: semWait userInput
| PID: 1
Process 1 is waiting for resource userInput
Resource userInput is now locked by process 1
Executing Instruction #11: assign a input
| PID: 1
Assigned a = 2
Executing Instruction #10: semWait userInput
| PID: 2
Process 2 is waiting for resource userInput
Executing Instruction #11: assign a input
| PID: 2
Assigned a = 4


OS Task Dashboard
Process and Scheduler
Main Dashboard
Memory Viewer
Mutex
Output
User Output
User Output
Output: Printing numbers from 1 to 10:
1 2 3 4 5 6 7 8 9 10

Chapter 3

7. System Initialization

- The system is initialized by setting up the memory and process management structures. Each process is registered and placed into the pending process list before being created. A mutex initialization function ensures that all resources are ready for allocation and use by processes.

8. Instruction Execution

- **Instruction Parsing and Execution**: Instructions are parsed, and based on the type (e.g., `assign`, `print`, `writeFile`, etc.), the appropriate action is performed. For example:
    - **Assign**: Assigns a value to a variable.
    - **Print**: Outputs the value of a variable.
    - **WriteFile**: Writes a variable value to a file.
    - **ReadFile**: Reads data from a file and assigns it to a variable.

Chapter 5

Conclusion

The project successfully demonstrated how different scheduling algorithms impact process execution and system performance. By implementing and analyzing **First-Come-First-Served (FCFS)**, **Round Robin (RR)**, and **Multilevel Feedback Queue (MLFQ)**, we gained valuable insights into process management in operating systems and the importance of choosing the right scheduling strategy based on system requirements. Each algorithm has its own strengths and weaknesses, and the choice of algorithm plays a critical role in achieving optimal performance in a multi-tasking environment.

Additionally, this project emphasized the importance of **memory management** and **mutual exclusion (mutexes)** in process synchronization. By implementing **mutexes** to manage shared resources (such as user input/output and file access), we ensured that only one process could access these resources at a time, preventing race conditions and ensuring data consistency. This aspect of process synchronization is crucial for the stability and reliability of multi-threaded applications.