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

Program Schedular

Team Members :

Ahmed H. Mostafa	**58-7370**
Abdelrahman A. Ashry	**58-0567**
Hisham H. Mohammed	**58-7992**
Abdelhamid S. Abdelhamid	**58-1165**
Yahia H. Khattab	**58-1499**
Youssef H. Abdelhamid	**58-0552**
Ali U. Youssef	**58-2854**

Team Number :

Team #63

Under Supervision of :

Dr. Eng. Catherine M. Elias
Dr. Aya Salama

Spring 2025

Contents

Chapter 1

Project Objectives

1.1 Main Objective

Design and implement a modular *mini-OS simulator* that faithfully models process lifecycle management, memory allocation, CPU scheduling, and synchronization, then complete with an interactive GUI for real-time visualization.

1.2 Specific Objectives

1. **Interpreter** – build a custom parser/dispatcher that executes user-defined instruction files one instruction/clock cycle.
2. **Memory Model** – architect a fixed 60-word memory space able to hold every process's *PCB* → *instructions* → *three variables*; each *memory word* is a `(name, data)` pair of strings.
3. **Schedulers** – implement three pluggable algorithms selectable at run-time:
   — First-Come-First-Serve (FCFS),
   — Round-Robin (RR) with user-defined quantum,
   — 4-level Multilevel-Feedback-Queue (MLFQ) with doubling quantums.
4. **Mutual Exclusion** – enforce safe access to the shared resources `userInput`, `userOutput`, and `file` via binary semaphores (`semWait`/`semSignal`) featuring priority-aware unblocking.
5. **Process State Management** – maintain *Ready*, *Running*, *Blocked* and *Terminated* states and guarantee correct state transitions under pre-emptive and non-pre-emptive execution as well as maintaining two queues, ready and blocked.
6. **Graphical User Interface** – provide an intuitive GTK front-end that lets users:
   — load program files and set arrival times,
   — choose the scheduling algorithm and RR quantum,
   — observe memory, queue contents, semaphore status, and execution logs.

Taken together, these goals will give us a compact, hands-on simulator that lets us actually see how an operating system juggles multiple programs, squeezes everything into limited memory, and keeps shared resources from clashing.

Chapter 2

Introduction

2.1 Project Overview

This project implements a simulated operating system scheduler supporting three CPU scheduling algorithms:

- **First-Come-First-Serve (FCFS)**
- **Round Robin (RR)**
- **Multilevel Feedback Queue (MLFQ)**

The system simulates core operating system functionalities with the following key features:

- **Process Control Blocks (PCBs)** containing process metadata and memory boundaries.
- **Three mutex-protected resources:** `userInput`, `userOutput`, and `file`.
- **Cycle-accurate execution**, where each instruction takes exactly **1 clock cycle** to execute.
- **Simulation State Handling**: Whether to start, stop or reset the simulation along with the option to either execute instructions one at a time or keep executing till all processes terminate.
- **Struct(s)** to handle functionalities like simulating the arrival of processes, memory access, mutex state and reading instructions.
- **Global Variables** to store important information needed throughout the simulation.
- **Helper methods** for the GUI to gather needed information.

2.2 Project Goal

This project aims to:

1. Compare the scheduling performance of FCFS, RR, and MLFQ under constrained memory conditions.
2. Tackle several issues that the OS faces like :
   - Race Condition
   - Priority Inversion

Chapter 3

Methodology

3.1 System-Level Workflow

a) **Load Phase** — the user selects as much program text files as he wishes, sets their arrival time and presses on "Load" that calls the function `readProcessAndStore()`.

b) **Process Arrival Handling** — `readProcessAndStore()` is called and stores the process as a pair of `(fileName, arrTime)` in an array of a `struct`, Process, called processes. This array can store up to 5 processes.

c) **Memory Allocation** — Each clock cycle, the array processes is checked if any process should arrive and if so, the method `storeInMemory()` is called and stores the process in the memory as a block: PCB → instructions → three variable slots.
Then it increments the `processCount` and stores the process in the designated queue associated with the choosen algorithm and changes its state to `Ready`.
If the memory is full, nothing will be stored and a message will be printed in the console (The user won't see it while loading but will see it once the simulation starts).

d) **Dispatch & Execution** — The algorithm decides which of the arrived processes should execute, changes its state to `Running` and calls the execution function :

   — `executeProcess()` **(FCFS)**: Executes the whole process one instruction per clock cycle.

   — `executeProcessTillQuantum()` **(RR/MLFQ)**: Executes the process till the quantum expires, the process gets block or the process terminates.

   Both functions call `executeInstruction()` to decode the instruction and execute it.

e) **Synchronisation** — Resource requests invoke `semWait()`; if the semaphore is busy, the process is transferred to the Blocked queue for that resource and to the Blocked queue of the system and its state becomes `Blocked`. Otherwise, the calling process locks the resource to itself.
When finished with a resource, processes invoke `semSignal()` which either unblocks the process with the highest priority waiting for that resource and return it to the algorithm queue (ready queue) or unlocks the resource for future processes to acquire.

f) **Completion / Requeue** — on quantum expiry or unblock, the PID is re-enqueued into the proper ready queue; when its instruction stream ends, the state becomes `Terminated`.

3.1.1 Process Loading & PCB Layout

Upon "*Load*":

— The GUI passes `<file path, arrival time>` to `readProcessAndStore()`.

Upon calling `storeInMemory()`:

— Six memory words are reserved for the PCB fields {*PID, State, Priority, PC, LowerBound, UpperBound*}.

— Program lines (instructions) are copied sequentially; three further words remain blank for run-time variables.

3.1.2 Queue Hierarchy

`processes`	Holds every process file along with its arrival time.
fcfsQueue	Single FCFS queue (non-preemptive).
rrQueue	One circular queue; quantum value supplied by the user through the GUI.
MLF(1-4)Queue	Four feedback levels (Q_1–Q_4) with quantums $\{1, 2, 4, 8\}$; demotion occurs on quantum expiry.

3.1.3 Execution Routines

1. `executeInstruction()` — The execution methods call this method to decode the instruction given as String and execute it.
2. Every instruction has its own method which `executeInstruction()` calls to execute this specific command.
3. The command "assign" is the only one where the method is called later on because it first triggers the GUI to activate the text field "Console Input"

3.1.4 Semaphore Logic

Each semaphore/mutex handles 4 values:

— `name` (userInput/userOutput/file),

— `is_locked` (0=unlocked, 1=locked),

— `locked_by_pid` (The PID of the locking process. -1 if unlocked)

— `blocked_Queue` (PIDs of processes waiting for this resource to get unlocked).

`semWait()` sets `is_locked` to 1 and sets `locked_by_pid` to the running pid; otherwise it adds the running pid to the `blocked_Queue` of the resource and to `all_blocked_queue` and changes the state of this pid to "Blocked".

`semSignal()` either sets `is_locked` to 0 and sets `locked_by_pid` to -1 or leaves `is_locked` as is and looks for the pid of the process with the highest priority contained in `blocked_Queue`. If it finds one, it removes the pid from the `blocked_Queue` and `all_blocked_queue`, sets it as the `locked_by_pid`, changes its state to "Ready" and adds it to the ready queue of the current scheduling algorithm.

3.1.5 GUI Integration & Real-Time Updates

At every clock cycle the GUI updates the following information:

— Process count,
— Current clock cycle,
— Current running pid, current Instruction and time in queue,
— Ready queue(s) and blocked queue(s),
— Mutex states,
— Console Input state,
— Drop down menu (containing existing processes)
— Console log,
— Memory.

3.1.6 Testing & Verification

We designed test suites that:

— Spawn processes with overlapping resource demands to validate semaphore blocking and priority-based unblocking.
— Vary arrival gaps to verify Ready-queue logic under FCFS, RR, and MLFQ.
— Load programs until the 60-word limit to stress the allocator.

All scenarios produced the expected state transitions and GUI traces.

3.1.7 Compile & Run

Use the following commands in order to compile and run our program:

— sudo apt update
— sudo apt upgrade -y
— sudo apt install build-essential gcc g++ make
— sudo apt install libgtk-3-dev pkg-config
— gcc Queue.c OS_MS2.c scheduler_gui2.c -o scheduler_gui ‘pkg-config –cflags –libs gtk+-3.0‘ && sudo ./scheduler_gui

Chapter 4

Results

4.1 System Overview

The operating system simulation successfully recognizes the three processes we test on:

1. **Process 1 (PID 1)**
   Source: `Program_1.txt`
   Memory Allocation: 16 words
   Instructions: 7
   Functionality: Given 2 numbers, the program prints the numbers between the 2 given numbers on the screen (inclusive)
2. **Process 2 (PID 2)**
   Source: `Program_2.txt`
   Memory Allocation: 16 words
   Instructions: 7
   Functionality: Given a filename and data, the program writes the data to the file
3. **Process 3 (PID 3)**
   Source: `Program_3.txt`
   Memory Allocation: 18 words
   Instructions: 9
   Functionality: Given a filename, the program prints the contents of the file on the screen

The memory manager correctly allocated memory for each process's PCB, instructions, and variables, maintaining proper isolation. All mutexes—`userInput`, `userOutput`, and `file`—were initialized to the unlocked state with empty blocked queues.

4.2 First-Come-First-Served (FCFS) Scheduling Results

Arrival times of the 3 processes set to 0,2 and 4 respectively.

Execution Timeline

— clock cycle 0: Process 1 arrives and starts execution

— clock cycle 0 - 6: Process 1 completes execution

— clock cycle 2: Process 2 arrives and appears in the ready queue

— clock cycle 4: Process 3 arrives and appears in the ready queue

— clock cycle 7 - 13: Process 2 completes execution

— clock cycle 14 - 22: Process 3 completes execution

4.3 Round Robin Scheduling Results

Arrival times of the 3 processes set to 0,2 and 4 respectively.

Small Quantum (Q = 1)

— clock cycle 0: Process 1 arrives, starts execution, locks UserInput and finishes its quantum

— clock cycle 1: Process 1 executes, locks UserInput and finishes its quantum

— clock cycle 2: Process 2 arrives, starts execution and gets blocked from UserInput

— colock cycle 3: Process 1 executes and finishes its quantum

— clock cycle 4: Process 3 arrives, starts execution and gets blocked from UserInput

— clock cycle 5: Process 1 executes, unlocks UserInput (to process 2) and finishes its quantum

— clock cycle 6: Process 2 executes and finishes its quantum

— clock cycle 7: Process 1 executes, locks UserOutput and finishes its quantum

— clock cycle 8: Process 2 executes and finishes its quantum

— clock cycle 9: Process 1 executes and finishes its quantum

— clock cycle 10: Process 2 executes, unlocks UserInput (to process 3) and finishes its quantum

— clock cycle 11: Process 1 executes, unlocks UserOutput and completes execution

— clock cycle 12: Process 3 executes and finishes its quantum

— clock cycle 13: Process 2 executes, locks file and finishes its quantum

— clock cycle 14: Process 3 executes, unlocks UserInput and finishes its quantum

— clock cycle 15: Process 2 executes and finishes its quantum

— clock cycle 16: Process 3 executes and gets blocked from file

— clock cycle 17: Process 2 executes, unlocks file (to process 3) and completes execution

— clock cycle 18: Process 3 executes and finishes its quantum

— clock cycle 19: Process 3 executes, unlocks file and finishes its quantum

— clock cycle 20: Process 3 executes, locks UserOutput and finishes its quantum

— clock cycle 21: Process 3 executes and finishes its quantum

— clock cycle 22: Process 3 executes, unlocks UserOutput and completes execution

Medium Quantum (Q = 3)

— clock cycle 0: Process 1 arrives and starts execution

— clock cycle 0 - 2: Process 1 executes, locks UserInput and finishes its quantum

— clock cycle 2: Process 2 arrives and appears in the ready queue

— clock cycle 3: Process 2 starts execution and gets blocked from UserInput

— clock cycle 4: Process 3 arrives and appears in the ready queue

— colock cycle 4 - 6: Process 1 executes, unlocks UserInput (to process 2), locks UserOutput and finishes its quantum

— clock cycle 7: Process 3 starts execution and gets blocked from UserInput

— clock cycle 8 - 10: Process 2 executes, unlocks UserInput (to process 3) and finishes its quantum

— clock cycle 11: Process 1 executes, unlocks UserOutput and completes execution

— clock cycle 12 - 14: Process 3 executes, unlocks UserInput, locks file and finishes its quantum

— clock cycle 15: Process 2 executes and gets blocked from file

— clock cycle 16 - 18: Process 3 executes, unlocks file (to process 2), locks UserOutput and finishes its quantum

— clock cycle 19 - 20: Process 2 executes, unlocks file and completes execution

— clock cycle 21 - 22: Process 3 executes, unlocks UserOutput and completes execution

Large Quantum (Q = 9)

— This behaves exactly like the FCFS run since the largest number of instructions is 9 (Program 3)

4.4 Multilevel Feedback Queue (MLFQ) Scheduling Results

Arrival times of the 3 processes set to 0,2 and 4 respectively.

Queue Structure

— Level 1: Highest Priority (`quantum = 1`)

— Level 2: Medium-High (`quantum = 2`)

— Level 3: Medium-Low (`quantum = 4`)

— Level 4: Lowest Priority (`quantum = 8, RR`)

Execution Timeline

— clock cycle 0: Process 1 arrives, starts execution, locks UserInput and finishes its quantum (demoted to queue 2)

— clock cycle 1 - 2: Process 1 executes and finishes its quantum (demoted to queue 3)

— clock cycle 2: Process 2 arrives and appears in queue 1

— clock cycle 3: Process 2 executes, gets blocked from UserInput and finishes its quantum (demoted to queue 2)

— clock cycle 4: Process 3 executes, gets blocked from UserInput and finishes its quantum (demoted to queue 2)

— clokc cycle 5 - 8: Process 1 executes, unlocks UserInput (to process 2), locks UserOutput, unlocks it and completes execution

— clock cycle 9 - 10: Process 2 executes and finishes its quantum (demoted to queue 3)

— clock cycle 11 - 14: Process 2 executes, unlocks UserInput (to process 3), locks file, unlocks it and completes execution

— clock cycle 15 - 16: Process 3 executes, unlocks UserInput and finishes its quantum (demoted to queue 3)

— clock cycle 17 - 20: Process 3 executes, locks file, unlocks it, locks UserOutput and finishes its quantum (demoted to queue 4)

— clock cycle 21 - 22: Process 3 executes, unlocks UserOutput and completes execution

Please note that in all algorithms the program terminates at clock cycle 23 to check if any process is not done yet.

Chapter 5

Conclusion

5.1 Summary of Findings

This project successfully implemented a process scheduler and resource management system within a simulated operating system environment. The system included support for three CPU scheduling algorithms—**First Come First Serve (FCFS)**, **Round Robin (RR)**, and **Multi-level Feedback Queue (MLFQ)**—alongside mutex-based synchronization for shared resources.

Through this implementation, we demonstrated how different scheduling strategies manage process execution, resolve resource contention, and perform priority-based unblocking.

Key observations include:

— **FCFS** ensured simple execution logic but lacked flexibility when handling processes of varying lengths.

— **RR** offered balanced time-sliced execution but introduced context-switching overhead, especially with smaller quantum values.

— **MLFQ** dynamically adjusted process priorities, favoring short tasks in higher queues while preventing starvation for longer ones.

— **Mutexes** successfully ensured mutual exclusion for critical sections, with blocked processes prioritized based on their scheduling priority.

5.2 Key Insights

1. Scheduling Trade-offs

— FCFS is easy to implement but inefficient for systems with diverse workloads.

— RR performance heavily depends on quantum size; smaller quantum improves fairness but adds overhead.

— MLFQ adapts to process behavior and combines time-slicing with priority-based demotion/promotion.

2. Resource Management

— Mutexes provided robust mutual exclusion for user input, output, and file access.

— Priority-based unblocking mechanisms ensured high-priority processes were not starved.

3. Memory Management

— The memory module, limited to 60 words, was effectively used to store PCBs, instructions, and variables.

— Memory fragmentation could become a limitation as more processes are loaded.

5.3 Technical Considerations and Challenges

— **Circular Queue Implementation:** Maintaining correct front and rear pointers to prevent overflow/underflow was critical.

— **Priority Handling in MLFQ:** Fairly demoting and promoting processes between levels required careful logic to avoid starvation and ensure responsiveness.

— **Blocked Process Management:** Synchronizing multiple mutex-specific queues with the global blocked queue added complexity.

— **Instruction Execution:** Executing commands such as `assign`, `print`, `semWait`, and `semSignal` required dynamic string parsing and process state updates.

— **Memory Constraints:** With limited space, memory optimization was essential to accommodate process data without exceeding bounds.

5.4 Final Thoughts

This project provided in-depth exposure to fundamental operating system concepts, including process scheduling, memory management, and synchronization. It highlighted the real-world trade-offs in selecting scheduling algorithms and designing fair resource management systems.

The experience serves as a practical foundation for understanding the intricacies of OS-level concurrency and multitasking.