CSEN 602 Operating Systems: Milestone 2 Project Report

TEAM 5

May 8,2025

Mennatulla Ahmed Elsabagh 56-29279

Alaa Abdelnaser 58–22615

Hoor Haytham 58-10710

Esraa Ahmed 58-14014

Malak Hisham Sallam 58-0665

Mennatullah Shaaban Ramzy Amer 58-23237

Menna ZeinEldin 58-14518

1-Introduction

Milestone 2 aims to develop a simulated operating system that manages multiple processes, applies various scheduling algorithms, enforces mutual exclusion for shared resources, and includes a user-friendly GUI for interaction. Operating on a fixed 60-word memory, the system manages process creation and executes instructions from specified program files. This report outlines the system architecture, key component implementations, encountered challenges, and GUI features designed to fulfill project requirements.

2-System Architecture

The operating system is implemented in C, with the core components organized as follows:

- **Process Control Block (PCB):** Stores process data, including process ID, state (READY, RUNNING, BLOCKED, TERMINATED), priority, program counter, memory boundaries, instruction count, int time_in_ready and int time_in_blocked.
- **Memory Management:** A 60-word memory array where each word holds a name (e.g., variable or instruction identifier) and data (e.g., instruction string or variable value).
- **Scheduler:** Supports FCFS, RR, and MLFQ algorithms to manage process execution.
- **Mutexes:** Three mutexes (userInput, userOutput, file) ensure mutual exclusion for shared resources.
- **Interpreter:** Parses and executes instructions from program files.
- **GUI:** Visualizes system state, process details, memory allocation, and resource usage.

3-Implementation Details

3.1 Process Management

Processes are created according to arrival times defined in the `ProcessInfo` structure. The `create_process` function reads program files, loads instructions into memory, initializes three variables per process (`a`, `b`, and one additional variable), and sets up the PCB. The PCB is allocated in memory after the instructions and variables, using seven memory words to store the process ID, state, priority, program counter, memory boundaries, and instruction count.

Listing 1: Snippet of create*processfunction*

```
1  int create_process( int process_id , char* program_file , int start_index) {
2      FILE * file = fopen ( program_file , " r");
3      if  (! file ) {
4          printf(" Error: Could not open file % s\ n", program_file );
5          return  -1;
6      }
7      char ** instructions = malloc (15 * sizeof( char *));
8      int instruction_count = 0;
9      char line [256];
```

```
10          while ( fgets( line , sizeof( line ), file )) {
11                  char* trimmed = trim ( line );
12                  if ( strlen ( trimmed ) == 0) continue ;
13-                 instructions[ instruction_count ]  =  strdup ( trimmed );
14                  instruction_count ++;
15          }
16          fclose ( file );
17          store_instructions ( start_index , instructions ,
                instruction_count );

18          char* var_names [] = {" a", " b", " "};

19          init_variables( start_index ,  instruction_count ,  3 ,  var_names);
20          PCB * pcb = malloc( sizeof( PCB ));
21          pcb - > process_id =  process_id ;
22          pcb - > memory_lower_bound  =  start_index;
23          pcb - > memory_upper_bound = start_index + instruction_count + 3
                + 7 - 1;
24          pcb - > program_counter  =  0;
25          pcb - > state = READY ;
```

3.2 Memory Management

Memory is structured as a fixed array of 60 `MemoryWord` structures, each comprising a `name` and `data` field. Processes receive contiguous memory segments starting from a specified index. The `store_instructions`, `init_variables`, and `store_pcb` functions handle memory allocation for instructions, variables, and PCBs, respectively. Variable storage and retrieval within a process's memory bounds are managed by the `set_variable` and `get_variable_value` functions.

3.3 Scheduling Algorithms

Three scheduling algorithms were implemented:

- **FCFS:** Selects the first process in the ready queue (queue 0) and runs it to completion or until it blocks. Implemented in `schedule_fcfs`.
- **Round Robin:** Uses a user-defined quantum to preempt processes. Processes are enqueued back to the ready queue after their quantum expires no preemption during the quanta. Implemented in `schedule_round_robin`.
- **MLFQ:** Manages four priority queues with quanta of 1, 2, 4, and 8 cycles, respectively. Processes start at priority 1 and are demoted to lower priorities upon quantum expiration. The lowest priority (level 4) uses RR scheduling. Implemented in `schedule_mlfq`.

Listing 2: Snippet of MLFQ Scheduler

```
PCB* schedule_mlfq(int* time_run) {
    static int quanta[4] = {1, 2, 4, 8};
    if (os_system.running_process && os_system.running_process->
       state == RUNNING) {
        int level = os_system.running_process->priority - 1;
        if (*time_run < quanta[level]) {
            bool higher_priority_exists = false;
            for (int i = 0; i < level; i++) {
                if (os_system.ready_queues[i].count > 0) {
                    higher_priority_exists = true;
                    break;
                }
            }
            if (!higher_priority_exists) {
                return os_system.running_process;
            }
        }
        if (level < 3) {
            os_system.running_process->priority++;
        }
        os_system.running_process->state = READY;
        int new_queue_idx = os_system.running_process->priority -
            1;
        enqueue(&os_system.ready_queues[new_queue_idx], os_system
           .running_process);
        store_pcb(os_system.running_process->memory_lower_bound,
                  os_system.running_process->instruction_count,
                     3,
                  os_system.running_process);
        os_system.running_process = NULL;
        *time_run = 0;
    }
    for (int i = 0; i < 4; i++) {
        if (os_system.ready_queues[i].count > 0) {
            PCB* next_process = dequeue(&os_system.ready_queues[i
               ]);
            if (next_process) {
                next_process->state = RUNNING;
                os_system.running_process = next_process;
                *time_run = 0;
                store_pcb(next_process->memory_lower_bound,
                   next_process->instruction_count, 3,
                           next_process);
                return next_process;
            }
        }
    }
    return os_system.running_process;
}
```

3.4-Mutual Exclusion

Three mutexes (userInput, userOutput, file) are implemented to manage resource access. The *semWait* function attempts to acquire a mutex, blocking the process if

the resource is locked. The semSignal function releases the mutex and unblocks the highest-priority process from the mutexs blocked queue. Unblocked processes are placed in the ready queue corresponding to their priority, in mutex the process that ocks will unlock

Listing 3: Snippet of semSignal Function

```
1     resource ? resource : " NULL");
2             return ;
3         }
4         PCB * pcb = os_system . running_process;
5         if  (! pcb )  {
6             printf(" sem Signal error: no  running  process\ n");
7             return ;
8         }
9         if ( mutex - > is_locked && mutex - > holding_process == pcb ) {
10            mutex - > is_locked = 0;
11            mutex - > holding_process  =  NULL;
12            printf(" Process % d released % s\ n", pcb - > process_id , resource );
131           snprintf( buffer , sizeof( buffer), " Process % d released % s\ n", pcb - >
                  process_id , resource );
14            Add Log Message (& gui_state ,  buffer);
15            if ( mutex - > blocked_queue . count > 0) {
16                PCB * unblocked_pcb = dequeue_highest_priority (& mutex
                      - > blocked_queue );
17                if ( unblocked_pcb ) {
18                    unblocked_pcb - > state  =  READY ;
19                    unblocked_pcb - > priority = unblocked_pcb - >
                          blocked_priority ;
20                    unblocked_pcb - > program_counter ++;
21                    int queue_idx = unblocked_pcb - > priority - 1;
22                    if ( queue_idx < 0 | | queue_idx >= 4) queue_idx = 0;
23                    enqueue (& os_system . ready_queues[ queue_idx],
                          unblocked_pcb );
24                    printf(" Process % d unblocked for % s, returned to queue % d,
                          PC =% d\ n",
```

```
25                          unblocked_pcb - > process_id , resource ,
                                queue_idx , unblocked_pcb - >
                                program_counter );
26                  snprintf( buffer , sizeof( buffer), " Process % d unblocked for % s,
                        returned to queue % d, PC =% d\ n",
27                          unblocked_pcb - > process_id , resource ,
                                queue_idx , unblocked_pcb - >
                                program_counter );

28                  Add Log Message (& gui_state ,  buffer);

29                  store_pcb ( unblocked_pcb - > memory_lower_bound ,

                        unblocked_pcb - > instruction_count , 3 , unblocked_pcb );

30                  mutex - > is_locked = 1;

31                  mutex - > holding_process  =  unblocked_pcb ;
32                  printf(" Process % d acquired % s upon unblocking \ n"
                        , unblocked_pcb - > process_id , resource );
33                  snprintf( buffer , sizeof( buffer), " Process % d acquired % s
                        upon unblocking \ n", unblocked_pcb- > process_id ,
                        resource );


37


38
```

4-Interpreter

The readProgramSyntax function parses program instructions and dispatches them to appropriate handlers:

- print: Outputs a variables value.
- assign: Assigns a value to a variable, supporting user input or file reads.

- writeFile: Writes data to a file.
- readFile: Reads file contents into a variable.
- printFromTo: Prints numbers between two values.
- semWait/semSignal: Manages mutex operations. Each instruction executes in one clock cycle, as specified.

5-GUI Implementation

GUI Integration via `GUIState` Structure:

- **Main Dashboard:** Displays a system overview, including the process list (ID, state, priority, memory bounds, program counter) and queue states.
- **Scheduler Control:** Allows selection between FCFS, RR, and MLFQ scheduling algorithms, with adjustable quantum for RR. Provides start, stop, reset, and step controls.
- **Resource Management:** Displays mutex status and the list of blocked processes.
- **Memory Viewer:** Visualizes the allocation of the 60-word memory space.
- **Log Panel:** Provides real-time logging of instructions, process states, and resource events.
- **Process Creation:** Enables adding new processes with user-specified file paths, arrival times, and memory start indices.
- **Logging and Execution:** `AddLogMessage` function ensures real-time logging, while `run_system_step` function facilitates step-by-step execution.

6-Challenges and Solutions

- Mutex Priority Handling: To manage unblocked processes based on priority, we maintained the blocked priority field in the PCB by saving the priority before blocking and restoring it after unblocking.
- MLFQ Preemption: Preemption of lower-priority processes by higher-priority ones was achieved by checking higher-priority queues in the schedule_mlfq function.
- Memory Allocation: To prevent memory overlaps, predefined start indices were assigned to processes, ensuring contiguous allocation within the 60-word limit.

- GUI Integration: Synchronizing GUI updates with system state was handled by logging significant events using AddLogMessage

7-Testing and Validation

The system was tested with three program files:

- **Program 1:** Prints numbers between two user-input values, using printFromTo.
- **Program 2:** Writes data to a file, using writeFile.
- **Program 3:** Reads and prints file contents, using readFile.

Tests verified correct scheduling, mutex operations, memory allocation, and GUI updates. The step-by-step mode was used to trace execution, ensuring each instruction completed in one clock cycle.

8-Conclusion

Milestone 2 successfully implements a simulated operating system with effective process management, scheduling, memory allocation, mutual exclusion, and a comprehensive GUI. The solution fulfills all project requirements, including three scheduling algorithms, mutex-based resource management, and real-time system visualization. Potential enhancements include dynamic memory allocation and support for more advanced program instructions.