Computer Operating Systems - Homework Assignment-1 BLG312E - Spring 2025

Ada Berfu Kaynak
Student Number: 820210314

1 Introduction

In this assignment for the course **BLG312E – Operating Systems**, a **preemptive, priority-based process scheduler** was implemented using fundamental UNIX process control mechanisms such as `fork()`, `exec()`, `SIGSTOP`, and `SIGCONT`. The scheduler operates using a variation of the Round Robin algorithm in a **multi-level queue** structure, considering *arrival time*, *priority level*, and *remaining execution time* for each job.

The scheduler reads job definitions and the time slice value dynamically from an external `jobs.txt` file. Each job is executed as a separate child process using `exec()`, and process control is achieved through UNIX signals. A **circular array** structure was used for queue management instead of a linked list to increase efficiency and simplify index-based operations.

The `job_runner.c` file was implemented to simulate individual job behavior. All state changes and scheduling events were logged in the required format into `scheduler.log`. The program was tested inside a `Docker` container, and a screen recording was taken to demonstrate correctness and functionality.

2 Code Implementation

2.1 scheduler.c

This part named scheduler.c is designed as a **priority-based scheduler** that reads job informations from the `jobs.txt` file, creates a separate process for each job, and runs these processes based on predefined time slices.

At the beginning of the program, necessary libraries are included and a **circular array** (`job_queue`) is defined to represent the job queue. Queue operations are handled using helper functions such as `enqueue()` and `is_queue_empty()`. Additionally, a `Job` structure is defined to store information for each job, including fields such as `PID`, start status, completion status, priority level, job name, arrival time, and remaining execution time.

Job information is parsed from `jobs.txt` using the `read_jobs()` function. After initialization, the work time and job status fields are reset. The `start_job()` function is used to create a new process with `fork()` and run the specified job program with `execl()`. The child process is initially paused using `SIGSTOP`.

Each process action (start, pause, continue, terminate) is recorded into the `scheduler.log` file via the `log_event()` function. This enables detailed tracking of all job activities.

During the main loop (timer loop), newly arrived jobs are added to the queue according to the current simulation time. The next job to be executed is selected with the `select_next_job()` function. This selection is made based on the job's priority level, arrival time, remaining time, and input order.

If the selected job has not started yet, it is launched using `start_job()`; if it was previously paused, it is resumed with `SIGCONT`. While a job is running for the defined time slice, the system continues to track and queue any newly arrived jobs.

If the job does not complete within its time slice, it is paused using `SIGSTOP` and re-enqueued. If it completes, it is terminated using `SIGTERM` and cleaned up with `waitpid()`. These operations are repeated cyclically until all jobs are completed.

When all jobs finish successfully, the scheduler writes a final message to the log file indicating the end of the timer loop.

2.2 job_runner.c

This file is a helper program that simulates how each job (`jobA`, `jobB`, `jobC`) behaves. It is executed by the scheduler using `exec()`, and identifies which job is running by checking the invoked file name.

The job simulates workload by sleeping in one-second intervals for a predefined duration depending on the job type. This design allows the preemptive scheduling logic of the scheduler to be tested accurately.

If the program is invoked with an unrecognized job name, this is logged to the `scheduler.log` file. The log entry includes the job name and status along with a timestamp. This feature helps with debugging by clearly identifying any unexpected job executions.

Thanks to the symbolic links (`jobA`, `jobB`, `jobC`), all jobs run the same binary, but are treated as distinct processes by the scheduler.

3 Makefile

The `Makefile` is used to compile and build the scheduler program developed for this assignment. The GNU Compiler Collection (`gcc`) is used with the `-Wall` flag enabled to display all warnings during compilation.

When the `make` command is executed, the following targets are generated in order:

- `scheduler` – Compiled from `scheduler.c`, this is the main scheduler executable.
- `job_runner` – Compiled from `job_runner.c`, this program simulates a job.
- `jobA`, `jobB`, `jobC` – These are symbolic links pointing to `job_runner`. Although they execute the same binary, the names allow the scheduler to treat them as separate jobs.

This symbolic linking mechanism enables the scheduler to start jobs by name (e.g., `exec("./jobA")`), while all jobs internally share the same code base from `job_runner`.

Additionally, the `make clean` command removes all compiled binaries and the `scheduler.log` file to reset the workspace.

4 Logging Result and Logging Analysis

The output in the scheduler.log file is visible in the table below. When this table is examined carefully, the processes were run and stopped in accordance with the expected round robin and priority order rules, and the branches worked properly according to these orders.

Timestamp	**Level**	**Message**
2025-03-26 17:02:10	INFO	Forking new process for jobA
2025-03-26 17:02:10	INFO	Executing jobA (PID: 2208) using exec
2025-03-26 17:02:13	INFO	jobA ran for 3 seconds. Time slice expired – Sending SIGSTOP
2025-03-26 17:02:13	INFO	Forking new process for jobB
2025-03-26 17:02:13	INFO	Executing jobB (PID: 2209) using exec
2025-03-26 17:02:16	INFO	jobB ran for 3 seconds. Time slice expired – Sending SIGSTOP
2025-03-26 17:02:16	INFO	Resuming jobA (PID: 2208) – SIGCONT
2025-03-26 17:02:19	INFO	jobA completed execution. Terminating (PID: 2208)
2025-03-26 17:02:19	INFO	Forking new process for jobC
2025-03-26 17:02:19	INFO	Executing jobC (PID: 2210) using exec
2025-03-26 17:02:22	INFO	jobC ran for 3 seconds. Time slice expired – Sending SIGSTOP
2025-03-26 17:02:22	INFO	Resuming jobB (PID: 2209) – SIGCONT
2025-03-26 17:02:25	INFO	jobB ran for 3 seconds. Time slice expired – Sending SIGSTOP
2025-03-26 17:02:25	INFO	Resuming jobC (PID: 2210) – SIGCONT
2025-03-26 17:02:26	INFO	jobC completed execution. Terminating (PID: 2210)
2025-03-26 17:02:26	INFO	Resuming jobB (PID: 2209) – SIGCONT
2025-03-26 17:02:29	INFO	jobB completed execution. Terminating (PID: 2209)
2025-03-26 17:02:29	INFO	Finished scheduler exiting

Table 1: Log Output

5 Discussion

When I first started developing the scheduler, I preferred a simple structure that only runs processes in order according to their arrival time, which does not fully reflect the priority levels and time slot behavior (this structure is still in my code as comments). However, this approach did not meet the requirements of multi-level scheduling and also prevented the process from being fairly re-selected at the end of the time slot. Therefore, I re-evaluated the process and developed a multi-criteria selection algorithm that re-evaluated processes at the end of each time slot. In this selection mechanism, factors such as priority level, arrival time, remaining work time and the position of the job in the order it was defined were considered together. In this way, both the priority jobs were brought forward and fair selection could be made in cases of equality.

Although I initially considered using a linked list, I preferred a circular array-based queue structure in order to reduce the complexity of the structure and the possibility of errors. In this way, queue management became more controlled and simple. In the management of processes, it was possible to pause and restart the active process using SIGSTOP and SIGCONT signals. This design allowed the scheduler to both prioritize correctly and implement Round Robin behavior effectively. Overall, the structural changes made made the system more stable and suitable for real-time scheduling logic.

6 Results

The requirements requested in the assignment and the information taught in the course have been tested in this assignment and completed successfully. Below you can find the requirements listed step by step.

All files requested to be delivered have been prepared in accordance with the specified rules. The `scheduler.c` and `job_runner.c` source code files are well documented and are located in the project folder. A `Makefile` has been added with these files and all compilation processes have been simplified. Detailed information on how to compile and run the program is explained in the `README.md` file. The main logic of the scheduler is written in the `scheduler.c` file; here, `fork()` for creating jobs, `exec()` for starting external programs, `SIGSTOP` and `SIGCONT` system calls for stopping and continuing jobs are used correctly.

The multi-level scheduling logic part is provided by the `select_next_job()` function, where the processes are selected according to their priority levels, arrival times, remaining work times and entry order. The jobs are dynamically ranked in the queue according to their priorities and are planned fairly by re-evaluating at the end of the time slice. The concept of the `TimeSlice` is read from the `jobs.txt` file and a certain period of time is given to each job accordingly. All events such as starting, stopping, resuming and completing each job

are recorded in the `scheduler.log` file with a time stamp in the desired format.

The explanations including the design decisions are presented in the `report.pdf` file. The answers given to the written questions are also given in detail in the `answers.pdf` file. In addition, a screen recording showing the timer working correctly on the terminal is also included in the folder. All these files are included in a single `.zip` archive named after my student number.

7 Conclusion

The project successfully demonstrates a functional preemptive priority-based scheduler using UNIX process control and multi-level queue logic.