Report for Round Robin Scheduler Assignment BLG312E - Spring 2025

Batuhan Karakus
150210040

March 26, 2025

Abstract

This report presents the design, implementation, and analysis of a preemptive, priority-based process scheduler implemented in C. The scheduler simulates process scheduling by reading job details from an external file (`jobs.txt`), managing process creation and control via `fork()`, `exec()`, `SIGSTOP` and `SIGCONT` signals, and logging all state transitions with timestamps. The report also discusses scheduling fairness, potential edge cases, and the deployment of the scheduler within a Docker container.

1 Introduction

This scheduler is designed as a preemptive, priority-based process scheduler that uses a variant of round-robin scheduling. It reads job information from an external file (`jobs.txt`) and creates child processes to simulate job execution. The entire implementation is contained in a single C source file (`scheduler.c`), which supports two modes:

- **Scheduler Mode:** Reads `jobs.txt` and schedules jobs.
- **Job Mode:** Simulates a single job's execution when run with the `job` argument (e.g., `./scheduler job jobA 6`).

Additionally, the project is deployed in a Docker container to ensure a consistent and isolated environment.

2 Implementation Overview

2.1 Scheduler Mode

In this mode, the scheduler:

- Reads the time slice and job entries from `jobs.txt`. Each job entry includes a job name, arrival time, priority, and execution time.
- Maintains job attributes such as remaining execution time, process ID, and input order.

- Uses a scheduling algorithm that sorts jobs based on:
  - Priority (lower numeric value indicates higher priority),
  - Arrival time,
  - Remaining execution time,
  - Input order (if other criteria are equal).
- Creates a child process for each job using `fork()` and executes the same binary in "job" mode via `execl()`.
- Manages preemption by sending a `SIGSTOP` signal when a job's time slice expires, and later resumes it with `SIGCONT` if necessary.
- Logs every process state transition with a timestamp in `scheduler.log`.

2.2 Job Mode

In job mode (invoked with the `job` argument), the program:

- Simulates the execution of a single job by decrementing its execution time every second.
- Supports pausing and resuming via `SIGSTOP` and `SIGCONT` signals.
- Exits upon completion of the job's execution time.

3 Design Decisions

3.1 Single Source File Structure

Integrating both scheduler and job functionalities in one file simplifies the build process and ensures code consistency. The same binary is used in two different modes based on command-line arguments.

3.2 Process Control and Signal Management

- **Process Creation:** The scheduler uses `fork()` to spawn child processes and `execl()` to invoke the binary in job mode.
- **Preemption:** A running job is preempted using `SIGSTOP` when its allocated time slice expires. If the job's remaining execution time is less than or equal to the time slice, it is allowed to finish.
- **Resumption:** Preempted jobs are resumed using `SIGCONT` when selected for execution again.
- **Logging:** Every critical event (fork, exec, preemption, resumption, termination) is logged with a timestamp for debugging and verification purposes.

3.3 Scheduling Algorithm

The scheduler implements a round-robin strategy enhanced with priority:

- Jobs are enqueued based on their arrival time.
- The ready queue is sorted by priority, arrival time, remaining execution time, and input order.
- The scheduler avoids immediate re-selection of the last preempted job when other jobs are available.

4 Scheduling Fairness Analysis

4.1 Round-Robin Mechanism

Using a fixed time slice ensures that every job gets a fair share of CPU time. Jobs are preempted if they exceed their allotted time, which prevents any single process from monopolizing resources.

4.2 Priority, Arrival Time, and Remaining Execution Time

Jobs with higher priority (lower numerical value) are scheduled first. In cases of equal priority, the job that arrived earlier and/or has a smaller remaining execution time is given preference. This multi-factor approach mitigates the risk of starvation.

4.3 Potential Enhancements

Additional fairness can be achieved by:

- Implementing an **aging mechanism** that gradually increases the priority of waiting processes.
- Adjusting time slices dynamically based on system load or process behavior.

5 Edge Cases and Failure Scenarios

5.1 Unresponsive Processes

If a process fails to respond to `SIGSTOP` or `SIGCONT`, the scheduler's control may be compromised. Although the current implementation assumes standard behavior, introducing timeouts or additional error handling could mitigate this risk.

5.2 Starvation

High-priority processes could potentially starve lower-priority ones. The round-robin mechanism, combined with multi-criteria scheduling, minimizes this risk. Nonetheless, an aging mechanism could further safeguard against starvation.

5.3 Input File Errors

The scheduler performs basic validation of `jobs.txt` entries. Malformed lines are skipped with an error message. Enhanced error reporting and robust parsing could further improve the system.

5.4 Deadlocks and Resource Contention

Non-blocking waits (e.g., `waitpid` with `WNOHANG`) help avoid deadlocks. However, simultaneous resource requests may still lead to contention, suggesting that more granular process control might be beneficial.

6 Docker Implementation

To ensure consistent deployment and testing, the scheduler was implemented and executed within a Docker container. This approach guarantees that all dependencies (GCC, Make, etc.) are available and that the environment is consistent across different systems.

Below is an example `Dockerfile` used for this project:

```
FROM ubuntu:20.04

RUN apt-get update && \
    apt-get install -y gcc make pandoc

COPY . /app
WORKDIR /app

RUN make

CMD ["./scheduler"]
```

The Docker container can be built with:

```
docker build -t scheduler .
```

And run using:

```
docker run --rm scheduler
```

7 Conclusion

The developed scheduler meets the assignment requirements by effectively implementing process creation, preemption, and resumption using `fork()`, `exec()`, `SIGSTOP`, and `SIGCONT`. It accurately reads job details from `jobs.txt`, logs all process state changes, and uses a fair scheduling algorithm that combines round-robin and priority-based strategies. The integration with Docker ensures a consistent execution environment.

8 Future Improvements

Future enhancements could include:

- Implementing an aging mechanism to further prevent process starvation.
- Introducing dynamic time slicing based on system load.
- Improving error handling and logging for greater robustness.
- Refining the simulation of time to achieve more precise scheduling behavior.

9 References

- Operating Systems Concepts and Process Scheduling literature.
- BLG312E - Spring 2025 Assignment Guidelines.
- C Standard Library Documentation for process control and signal handling.