ISTANBUL TECHNICAL UNIVERSITY

COMPUTER ENGINEERING DEPARTMENT

BLG 312E
COMPUTER OPERATING SYSTEMS

Homework 1

Ali Emre Kaya
150210097
kayaemr21@itu.edu.tr

SPRING 2025

Contents

1 Introduction

In this assignment, I learn how fork and exec work, and the logic behind operating system scheduling, also I implemented a Round Robin scheduling system which has efficient response time using with C programming language and its valuable fork(), exec() and many other functions.

2 Round Robin Scheduling

Round-robin is a time-slicing scheduling algorithm where each job gets a fixed time slot in a cyclic order. If a job isn't completed within its time slice, it is moved to the end of the queue. This ensures fairness and responsiveness but may lead to higher turnaround time for longer tasks.

$$Tturnaround = Tcompetion \text{ - } Tarrival$$

The cause of high $Tturnaround$ metric in round robin scheduling is understandable from formula; job start to run as soon as possible, bit it finish too later because round robin is fair.

$$Tresponse = Tfirstrun \text{ - } Tarrival$$

But round robin is very effective about response time, because it's fair, when a job comes, it start to run as soon as possible.

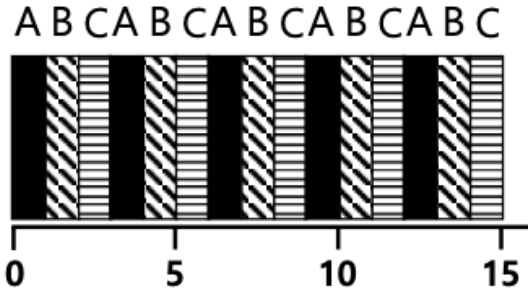

Figure 1: Round Robin

3 Code

First, I prepare a well-structured file structure. Then, I need to take input from *jobs.txt*. I prepare a `Job` struct (in *job.h*) and push the information into a job array. A priority queue structure may be efficient for implementing round-robin, but the given job selection algorithm requires ordered indexes. So, I prefer to preserve the index structure and go with a simple array.

I create *process.c* and *job.c* files to handle round-robin. The process acts like a scheduler and calls jobs.

In my implementation, jobs are added to the queue when their arrival time comes. Then, they wait for the current job's interval time to finish. After that,

the algorithm selects the highest-priority job (there is a basic algorithm detailed in the assignment slide). I implemented this algorithm.

When a new job arrives, a fork occurs. However, when an already forked job selected by algorithm to process, it only resumes execution.

To stop a process, I use `SIGSTOP`; to start it again, I use `SIGCONT`; and to terminate a finished task, I use `SIGKILL`, `SIGTERM` is also usable whn finising a task, but I prefer `SIGKILL` for its robustness.

All logs are saved in *data/output/schedule.log*.

```
[2025-03-21 20:19:15] [INFO] [PRC.c] - RESUME, jobA (PID: 5417)
                                SIGCONT
```

This is an example log. The first bracket represents the time, the third one represents which file generated the log, and then the log information follows.

I run my code in a Docker environment to ensure that everyone can easily execute it. Also, I prepare a default Makefile to make it even easier.

REFERENCES