LASSONDE SCHOOL OF ENGINEERING | YORK U

Operating System Fundamentals - EECS3221
Project 1 – Fall 2025
Simulating CPU Scheduling Algorithms[1]

Contents

Instructions

- This is an individual project.
- Check eClass for the due date and time.

Description

This project involves implementing several different process scheduling algorithms. The scheduler will be assigned a predefined set of tasks and will schedule the tasks based on the selected scheduling algorithm. Each task is assigned a priority and CPU burst. The following scheduling algorithms will be implemented:

1. **First-Come, First-Served (FCFS):** Tasks are scheduled in the order they request the CPU.
2. **Shortest Job First (SJF):** Tasks are scheduled in order of the length of their next CPU burst.
3. **Priority Scheduling:** Tasks are scheduled based on their priority.
4. **Round-Robin (RR):** Tasks are scheduled in a cyclic order, with each task running for a fixed time quantum (or the remainder of its CPU burst if less than the quantum).
5. **Priority with Round-Robin:** Tasks are scheduled based on priority. For tasks with the same priority, round-robin scheduling is applied using the defined time quantum.

[1] This project is provided as a programming project in Operating System Concepts, 10th Edition, by Silberschatz et al. © 2018 textbook.

Priorities range from 1 to 10, with higher numeric values representing higher priorities.
For RR scheduling, the time **`quantum`** is fixed at **`10`** milliseconds.

The goal of this project is to demonstrate the behaviour and performance of these scheduling algorithms under different conditions.

Scheduling algorithms are discussed in Section 5.3 in the course textbook (numbering may vary slightly by edition).

Implementation

The implementation for this project must be completed in C, with supporting program files provided in the **`StarterKit-Code`** folder. These supporting files read in the schedule of tasks, insert the tasks into a list, and invoke the scheduler.

The schedule of tasks has the form **`[task name] [priority] [CPU burst]`**, with the following example format:

```
T1, 4, 20
T2, 2, 25
T3, 3, 25
T4, 3, 15
T5, 10, 10
```

In this example:

- Task **`T1`** has a priority of **`4`** and a CPU burst of **`20`** milliseconds.
- Task **`T2`** has a priority of **`2`** and a CPU burst of **`25`** milliseconds, and so on.

It is assumed that all tasks arrive at the same time, so your scheduler algorithms do not have to support higher-priority processes pre-empting processes with lower priorities.

Tasks Storage and Organization

There are a few different strategies for organizing the list of tasks:

- Single Unordered List: One approach is to place all tasks in a single unordered list, where the strategy for task selection depends on the scheduling algorithm. For example, Shortest Job First (SJF) would search the list for the task with the shortest CPU burst.
- Ordered List: Alternatively, a list could be ordered according to scheduling criteria (that is, by priority).
- Separate Queues for Each Priority: One other strategy involves having a separate queue for each unique priority, as shown in Figure 5.7.

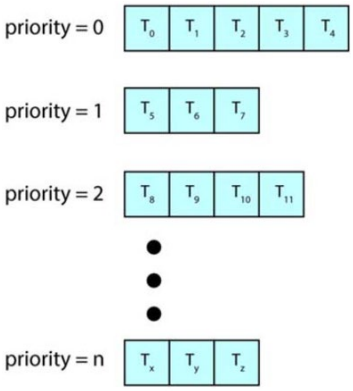


Figure 5.7 Separate queues for each priority.

It is also worth highlighting that we are using the terms **list** and **queue** somewhat interchangeably. However, a queue has very specific FIFO functionality, whereas a list does not have such strict insertion and deletion requirements. You are likely to find the functionality of a general list to be more suitable when completing this project.

Tiebreaking Rules

When scheduling tasks:

- Processes must execute in the order they appear in the input if there are ties in priority or burst time.
- This rule applies to ties during SJF, Priority Scheduling, or Priority with Round-Robin.

See examples of outputs for each algorithm in the `CorrectOutput.txt` file.

StarterKit

The file **`driver.c`** reads in the schedule of tasks, inserts each task into a linked list, and invokes the process scheduler by calling the **`schedule()`** function.

The **`schedule()`** function executes each task according to the specified scheduling algorithm.

As part of your scheduler, you must determine which task should run next. For example, you might implement a **`pickNextTask()`** function to choose a task, and then use the **`run()`** function defined in **CPU.c** to simulate its execution. The **`run()`** function is provided; the task-selection logic is your responsibility.

A **`Makefile`** is used to determine the specific scheduling algorithm that will be invoked by the driver. For example, to build the FCFS scheduler, we would enter:

```
make fcfs
```

and would execute the scheduler (using the schedule of tasks `schedule.txt`) as follows:

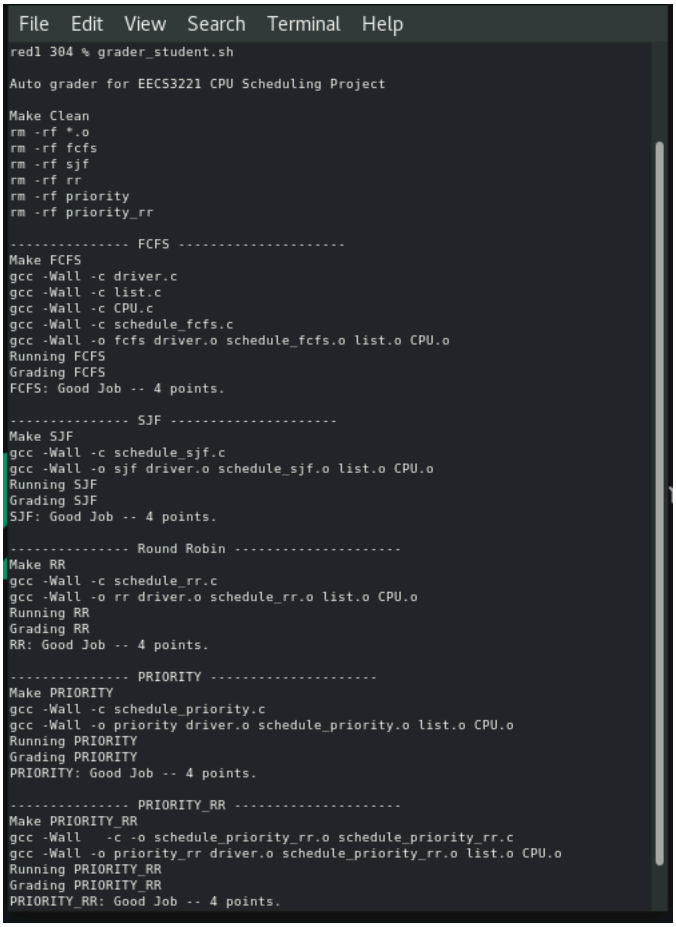
```
./fcfs schedule.txt
```

Before proceeding, be sure to familiarize yourself with the source code provided as well as the **`Makefile`**. Here is a quick tutorial on **`Makefile`** http://www.cs.colby.edu/maxwell/courses/tutorials/maketutor/ in case you are not familiar with it.

Evaluating Your Implementation

The correct outputs for all algorithms for a sample input are shown in the `CorrectOutput.txt` file.

You can use the `grader_student.sh` script to check that your code is working correctly.



Submission

Completing this project will require:

1. Writing the following C files, which invoke the appropriate scheduling algorithm:
    1. `schedule_fcfs.c`
    2. `schedule_sjf.c`
    3. `schedule_rr.c`
    4. `schedule_priority.c`
    5. `schedule_priority_rr.c`

2. Calculating the *average* **turnaround time**, **waiting time**, and **response time** for each of the above scheduling algorithms.
3. Submitting **one** zip file, called A1_cpu_yourfirstname.zip, that includes:
    - The starter code (even if it was not modified)
    - The Makefile (do not change the provided Makefile)
    - Any source files that were added
    - **Do not** submit object (*.o) files or compiled executables
4. Additional requirements:
    - Ensure your project runs correctly on the Department's servers with the required configuration (see Environment Setup).
    - TAs will compile your submission using the provided Makefile; do **not** change the content of the provided Makefile.
    - Grading will be performed using input files with different names. Your algorithms must not be hard-coded to `schedule.txt`; they should work with any input file name.
    - Test your code using the file `grader_student.sh` provided in the `StarterKit`.

Rubric

Each scheduling algorithm is worth **4 points**, making a total of **20 points** for the project. The breakdown for each algorithm is as follows:

1. **Correct Output** (Order and Values): 2.5 points.
2. Deductions for errors in the output:
    - One incorrect line: -0.5 point
    - Two incorrect lines: -1 point
    - Duplicate lines: -0.5 point
    - Three or more incorrect lines: -2.5 points (no credit for this section)
3. **Correct Average Waiting Time**: 0.5 point
4. **Correct Average Turnaround Time**: 0.5 point
5. **Correct Average Response Time**: 0.5 point

Collaboration

Here are the allowed activities while working on this project:

1. **Discussing Problem-Solving Approaches**: You may discuss strategies for solving a problem. However, to preserve the learning experience of others, avoid revealing key solutions directly and instead guide them toward discovering the solutions themselves.
2. **Syntax and Bug Discussions**: You can discuss syntax issues and bugs in your code, provided you do not show your code to others. For example, verbally discussing issues is allowed, but screen sharing via tools like Zoom is not.
3. **Using Small Code Snippets**: Using brief, publicly available code snippets for solving minor problems, such as iterating through an array, is permitted. These must be cited in comments within your code.

While working on this project, it is not allowed to be:

1. **Viewing Another Student's Code**: You may not view another student's project code to understand an idea or solve a problem, nor share your own code for this purpose.
2. **Possessing Unauthorized Code**: You may not possess project solution code you didn't write yourself, nor another student's code, in any form (electronic, physical, etc.), even for debugging purposes. Sharing such code is equally prohibited.

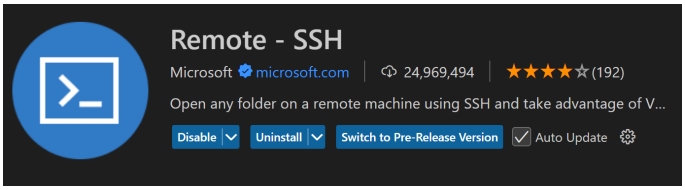
Important Note: Your code will be run through code plagiarism checkers.

Environment Setup

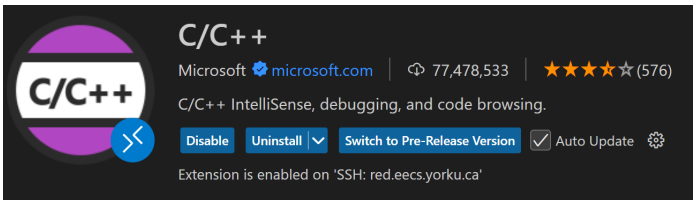
This section provides guidance for setting up EECS servers for remote development in C. You will need to configure the server and set up a client IDE (e.g., Visual Studio Code) to use the server as a backend for development.

You can use any EECS department server (e.g., red, red1, crimson, etc.) to write your code. There are several ways you can use the servers:

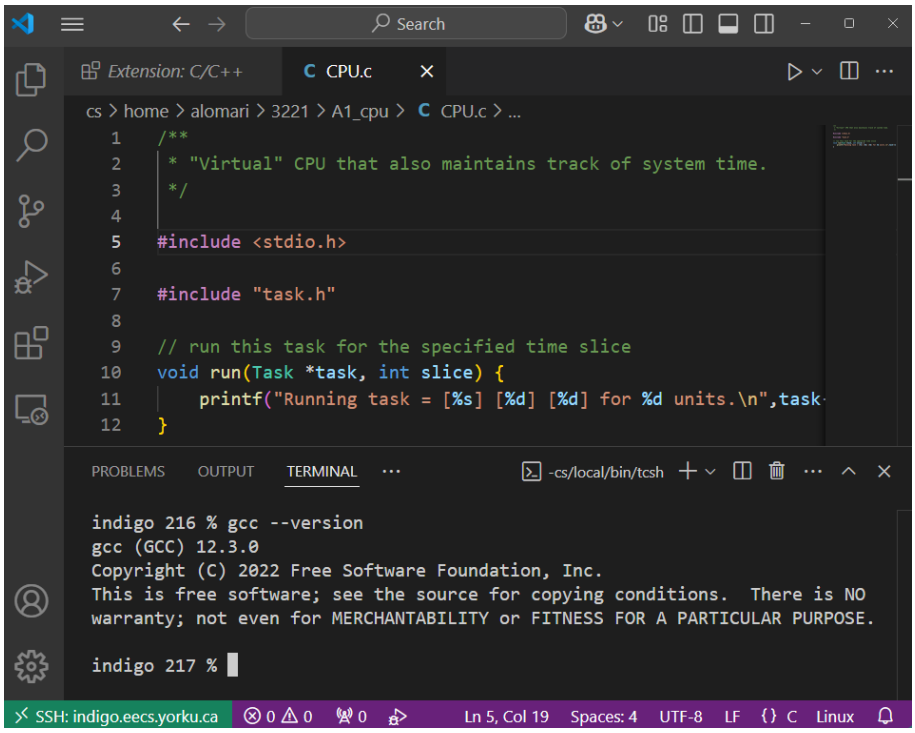
1. Physically visit one of the EECS labs, log in locally to the machine and start developing your code.
2. If you can't be in the lab, you can still log in remotely to https://remotelab.eecs.yorku.ca and start developing your code.
3. You can set up your local IDE to connect to the remote server following the example steps below:
    1. Download and install Visual Studio Code on your local machine.
    2. Install Required Extensions:
        - Remote - SSH Extension: Allows remote connections to servers.



        - C/C++ Extension: Enables C/C++ development.



    3. Create a remote connection to your, e.g., red1, server.
    4. Open the terminal inside VSCode and verify the versions of GCC and make.



You are all set to start the development of your project code.
If you encounter login or configuration issues, contact EECS IT support.