CPTS 360: Programming Assignment 2

CPU Scheduling

Notes:

- This is an individual assignment.
- You should complete the lab on a Linux environment (either a dedicated machine or VM). Using WSL (Windows Subsystem for Linux) on Windows is NOT recommended.
- Your submission will be tested on a Linux environment. You will NOT receive any points if your submitted program does not work on a Linux system.
- Start EARLY. The lab may take more time than you anticipated.
- Read the entire document carefully before you start working on the lab.
- The code and other answers you submit MUST be entirely your own work, and you are bound by the WSU Academic Integrity Policy (`https://www.communitystandards.wsu.edu/policies-and-reporting/academic-integrity-policy/`).
- You MAY consult with other students about the conceptualization of the tasks and the meaning of the questions, but you MUST NOT look at any part of someone else's solution or collaborate with anyone.
- You may consult published references or search online archives, provided that you appropriately cite them in your reports/programs.
- The use of artificial intelligence (AI)–generated texts/code snippets must be CLEARLY DISCLOSED, including the prompt used to generate the results and the corresponding outputs the tool(s) provide.

Good Luck!

1 Introduction

The objective of this assignment is to get yourself familiar with the various CPU scheduling algorithms: *(a)* FCFS (First Come First Serve), *(b)* RR (Round Robin), and *(c)* SJF (Shortest Job First).

2 Overview

In this lab, you will simulate scheduling to see how the time required depends on the scheduling algorithm and the request patterns.

A process is characterized by four numbers A, B, C, and M as follows:

- A is the arrival time of the process.
- C is the total CPU time needed.
- A process contains computation alternating with I/O. We refer to these as CPU bursts and I/O bursts. We simplify the assumption that, for each process, the CPU burst times are uniformly distributed

random integers (UDRIs) in the interval $(0, B]$. To obtain a UDRI t in some interval $(0, U]$, we use the function `randomOS(U)` (see below). If t, the value returned by `randomOS()`, is larger than the total CPU time remaining, we set t to the remaining time.

- The I/O burst time for a process is its preceding CPU burst time multiplied by M.

2.1 Determining CPU Burst Time

We provide an implementation of the function `randomOS(U)` that, given a process id *pid*, reads a random non-negative integer X from a file named `random-numbers` and returns the value $1 + (X \ mod \ U)$. The purpose of standardizing the random burst generation is to ensure that all correct programs will produce the same answers. You must use the provided `randomOS(U)` function to obtain the CPU burst time.

3 Lab Tasks

Your task is to read a file describing n processes (i.e., n triples of numbers) and then simulate the n processes until they all terminate. You will simulate three process scheduling techniques discussed in class:

1. FCFS (First Come First Serve),
2. RR (Round Robin) with a quantum size of 2 and
3. SJF (Shortest Job First).

The SJF variant you will implement is *not* preemptive, but is not run to completion, i.e., switch on I/O bursts. Since SJF is the shortest job first (not the shortest burst first), the time you use to determine priority is the total time remaining (i.e., the input value C minus the number of cycles this process has run).

You should simulate each of the above three scheduling algorithms, assuming, for simplicity, that a context switch takes zero time. You need only do calculations for every *integer* time unit (i.e., you may assume nothing exciting happens at time 11.5).

The way to implement your simulator is to keep track of the state of each process and advance time, making any state transitions needed. At the end of the run, you should first print an identification of the run, including the scheduling algorithm used, any parameters (e.g., the quantum for RR), and the number of processes simulated.

You should then print for each process:

- (A, B, C, M)
- Finishing time
- Turnaround time (i.e., "finishing time" $-A$)
- I/O time (i.e., time in "Blocked" state)
- Waiting time (i.e., time in "Ready" state)

Then, print the following summary data

- Finishing time (i.e., when all the processes have finished)
- CPU Utilization (i.e., percentage of time some job is running)
- I/O Utilization (i.e., percentage of time some job is blocked)
- Throughput, expressed in processes completed per hundred-time units
- Average turnaround time
- Average waiting time

Note: You must use the C programming language for your simulator.

3.1 Starter Code and Sample Input/Output

We provide a skeleton C code and a few helper functions, including `randomOS(U)`. For each scheduling algorithm, there are several runs with different process mixes. A mix is a value of n followed by $n(A,B,C,M)$ quadruples. The first two input sets are given below.

```
1 (0 1 5 1)   about as easy as possible
2 (0 1 5 1) (0 1 5 1)   should alternate with FCFS
```

Note that the comments (texts) are not part of the input. Your parser must have a way to discard comments.

We provide a set of sample inputs and corresponding outputs (under `sample_io` directory). For each sample output, we also provide a detailed trace (see under `trace_and_summary` directory) that contains tick-based event traces to ease your debugging. **Note:** you are *not* required to generate detailed trace.

We also provide a `makefile`. You can compile to the code with `make` and then test each sample input as `make <test-number>`. For example, to compile and run `input-4`, run:

```
make
make test04
```

3.2 Breaking Ties

There are two places where the above specification is not deterministic, and different choices can lead to different answers. To standardize the answers, we make the following choices.

1. A running process can have three events occur during the same cycle:
   (a) It can terminate (remaining CPU time goes to zero);
   (b) It can block (remaining CPU burst time goes to zero);
   (c) It can be preempted (e.g., the RR quantum goes to zero).

   They should be considered in the above order. For example, consider the process terminates if all three occur in one cycle.

2. Many jobs can have the same "priority." For example, in RR, jobs can become ready at the same cycle, and you must decide in what order to insert them onto the FIFO (First-In-First-Out) ready queue. Ties should be broken by favoring the process with the earliest arrival time *A*.

   If the arrival times are the same for two processes with the same priority, then favor the process listed earliest in the input.

We break ties to standardize answers. We use this particular rule for two reasons. *First,* it does break all ties. *Second,* it is very simple to implement (although it is not especially wonderful and is not used in practice).

3.3 Running Your Program

Your program must read its input from a file whose name is given as a command line argument (from the `makefile`). The input format is shown above (but the "comments" may or may not be present in the input traces).

Be sure you can process the sample inputs. Do not assume you can change the input by adding spaces or commas or removing parenthesis. Your code will be tested on sample inputs plus a few additional process sets with similar formats.

Your program must send its output to the screen (terminal). We provide a few helper functions to print out output and summary data.

4 Deliverable

- Complete the implementation of the C code.

  [90 Points]

  - The point split for your implementation is as follows:
    - **80 points** for correctness (process all inputs and generate correct outputs as formatted by the files in `sample_io/output/summary`), and
    - **10 points** for good code organization, indentation, and proper comments.

- A PDF document containing a summary of how you implemented each of the three algorithms, including how you incorporated tie-breaking rules in your code (see below for formatting details).

  [10 Points]

  - Your report should be **no more than two pages**; use standard **letter paper size** ($8\frac{1}{2}$ by 11 inches) with **11 points Times font** and **1-inch margin**.
  - Your PDF filename should be `pa2_lastname.pdf`, where `lastname` is your surname.
  - Include your **full name** and **WSU ID** inside the report.
  - Include your report on the GitHub repository.
  - We will **deduct 5 points** if the filename/contents do not follow this convention and/or you use a different font/margin than that listed above.

5 Submission Guidelines

Commit and push your work (source and report) on the GitHub Classroom Programming Assignment 2 repository. **Make sure you can successfully push commits on GitHub Classroom — *last minute excuses will not be considered***. Paste the URL of your GitHub repo to the corresponding lab assignment section of Canvas. Check the course website for additional details.

Note: Your work on GitHub Classroom will not be considered for grading unless you submit the link to the GitHub repository URL on Canvas. Hence, **You will NOT receive any points if you fail to submit your URL on Canvas.** Submissions received after the due date will be graded based on the late submission policy as described in the course syllabus.

Acknowledgement. This assignment was initially developed by the NYU Operating Systems course staff and has been customized for CPTS 360 by the instructor.