Tutorial 4 **CS3103 Operating Systems**

Student Name: _____

Student No. : | | | | | | | | |

Day: ☐ Monday ☐ Tuesday

Time: ☐ 10:00 - 10:50 ☐ 11:00 - 11:50 ☐ 12:00 - 12:50 ☐ 14:00 - 14:50 ☐ 18:00 - 18:50

CPU Scheduling

Introduction

Topics to be covered in this tutorial include:

- How different schedulers perform under scheduling metrics such as response time, turnaround time, and total wait time.

Acknowledgement

This tutorial was adapted from OSTEP book written by Remzi and Andrea Arpaci-Dusseau at University of Wisconsin. This free OS book is available at http://www.ostep.org.

Getting Started

1. Logging in to the Linux server

- Start the SSH client, e.g., MobaXterm or Xshell.
- Log in to the Linux server using the following details:

  Host Name: gateway.cs.cityu.edu.hk
  User Name: your EID (e.g., cctom2)
  Password: your password

Your password will not be shown on the screen as you type it, not even as a row of stars (******).

NOTE: The shell will always give you a prompt if it is ready to accept commands. The shell prompt normally ends in a `$` sign as we use in this tutorial. Some shell prompts end in `%` or `>` instead. Never copy/type the shell prompt used in this tutorial.

NOTE: Please don't forget to log out (use the `exit` command) after you finish your work.

2. Getting the `scheduler.py` simulator

A simulation program, called `scheduler.py`, is given to allow you to see how different schedulers perform under scheduling metrics, which in the directory */public/cs3103/tutorial4*.

Start by copying them to a directory in which you plan to do your work. For example, to copy tutorial4 directory to your current directory and change to it, enter:

```
$ cp -rf /public/cs3103/tutorial4 .
$ cd tutorial4
```

The last dot/period (.) indicates the current directory as the destination.

Introduction to `scheduler.py` simulator

This program, `scheduler.py`, allows you to see how different schedulers perform under scheduling metrics such as response time, turnaround time, and total wait time. The **turnaround time** of a job is defined as the time at which the job completes minus the time at which the job arrived in the system.

$$T_{turnaround} = T_{completion} - T_{arrival}$$

The **response time** is defined as the time from when the job arrives in a system to the first time it is scheduled.

$$T_{response} = T_{firstrun} - T_{arrival}$$

The wait time is defined as amount of time a process has been waiting in the ready queue.

$$T_{wait} = T_{completion} - \text{job length}$$

Three schedulers are "implemented": FIFO, SJF, and RR. There are two steps to running the program.

First, run without the `-c` flag: this shows you what problem to solve without revealing the answers. For example, if you want to compute response, turnaround, and wait for three jobs using the FIFO policy, run this:

```
$ ./scheduler.py -p FIFO -j 3 -s 100
```

If that doesn't work, try this:

```
$ python2 ./scheduler.py -p FIFO -j 3 -s 100
```

This specifies the FIFO policy with three jobs, and, importantly, a specific random seed of 100. If you want to see the solution for this exact problem, you have to specify this exact same random seed again. Let's run it and see what happens. This is what you should see:

```
$ ./scheduler.py -p FIFO -j 3 -s 100
ARG policy FIFO
ARG jobs 3
ARG maxlen 10
ARG seed 100

Here is the job list, with the run time of each job:
  Job 0 ( length = 2 )
  Job 1 ( length = 5 )
  Job 2 ( length = 8 )

(content removed for brevity, the same hereinafter.)
```

As you can see from this example, three jobs are generated: job 0 of length 2, job 1 of length 5, and job 2 of length 8. As the program states, you can now use this to compute some statistics and see if you have a grip on the basic concepts.

Once you are done, you can use the same program to "solve" the problem and see if you did your work correctly. To do so, use the `-c` flag. The output:

```
$ ./scheduler.py -p FIFO -j 3 -s 100 -c
ARG policy FIFO
ARG jobs 3
ARG maxlen 10
ARG seed 100

Here is the job list, with the run time of each job:
  Job 0 ( length = 2 )
  Job 1 ( length = 5 )
  Job 2 ( length = 8 )

** Solutions **

Execution trace:
  [ time   0 ] Run job 0 for 2.00 secs ( DONE at 2.00 )
  [ time   2 ] Run job 1 for 5.00 secs ( DONE at 7.00 )
  [ time   7 ] Run job 2 for 8.00 secs ( DONE at 15.00 )

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 2.00  Wait 0.00
  Job   1 -- Response: 2.00  Turnaround 7.00  Wait 2.00
  Job   2 -- Response: 7.00  Turnaround 15.00  Wait 7.00

  Average -- Response: 3.00  Turnaround 8.00  Wait 3.00
```

As you can see from the figure, the `-c` flag shows you what happened. Job 0 ran first for 2 seconds, Job 1 ran for 5 seconds, and then Job 2 ran for 8 seconds. Not too hard; it is FIFO, after all! The execution trace shows these results.

The final statistics are useful too: they compute the "response time" (the time a job spends waiting after arrival before first running), the "turnaround time" (the time it took to complete the job since first arrival), and the total "wait time" (any time spent ready but not running). The stats are shown per job and then as an average across all jobs. Of course, you should have computed these things all before running with the `-c` flag!

If you want to try the same type of problem but with different inputs, try changing the number of jobs or the random seed or both. Different random seeds basically give you a way to generate an infinite number of different problems for yourself, and the `-c` flag lets you check your own work. Keep doing this until you feel like you really understand the concepts.

One other useful flag is `-l` (that's a lower-case L), which lets you specify the exact jobs you wish to see scheduled. For example, if you want to find out how SJF would perform with three jobs of lengths 5, 10, and 15, you can run:

```
$ ./scheduler.py -p SJF -l 5,10,15
Here is the job list, with the run time of each job:
  Job 0 (length = 5.0)
  Job 1 (length = 10.0)
  Job 2 (length = 15.0)
```

And then you can use the `-c` flag to solve it again. Note that when you specify the exact jobs, there is no need to specify a random seed or the number of jobs: the jobs lengths are taken from your comma-separated list.

Of course, more interesting things happen when you use SJF (shortest-job first) or even RR (round robin) schedulers. Try them and see!

And you can always run the `-h` flag to get a complete list of flags and options (including options such as setting the time quantum for the RR scheduler).

```
$ ./scheduler.py -h
```

Questions

All questions should be answered on the separate answer sheet provided.

1. Compute the response time and turnaround time when running three jobs of length 200 with the SJF (`./scheduler.py -p SJF -l 200,200,200`) and FIFO (`./scheduler.py -p FIFO -l 200,200,200`) schedulers.

2. Now do the same but with jobs of different lengths: 100, 200, and 300. The commands are (`./scheduler.py -p SJF -l 100,200,300`) and (`./scheduler.py -p FIFO -l 100,200,300`). What if you change the order of the job length? Try different orders to find the difference.

3. Compute the response time and turnaround time when running three jobs of length 3 with the RR scheduler and a time-slice of 1 (`./scheduler.py -p RR -l 3,3,3 -q 1`). Do the same but change the job lengths as 3,2,4.

4. For what types of workloads does SJF deliver the same turnaround times as FIFO?

5. For what types of workloads and quantum lengths does SJF deliver the same response times as RR?

6. What happens to response time with SJF as job lengths increase? Can you use the simulator to demonstrate the trend?

7. What happens to response time with RR as quantum lengths increase? Can you write an equation that gives the worst-case response time, given N jobs?