Tutorial 2

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

The Process

Introduction

Topics to be covered in this tutorial include:

1. How process states change as programs run and either use the CPU (e.g., perform an add instruction) or do I/O (e.g., send a request to a disk and wait for it to complete).

Points to note about the simulator:

1. This tutorial is based on running a **simulator**, which mimics some aspect of an operating system.
2. The simulator can be used to both **generate problems** and **obtain solutions** for an infinite number of problems. Different random seeds can usually be used to generate different problems; using the `-c` flag computes the answers for you (presumably after you have tried to compute them yourself!).

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.

Pre-lab

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 `process-run.py` simulator

A simulation program, called `process-run.py`, allows you to see how the state of a process state changes as it runs on a CPU. This simulator is located in the directory */public/cs3103/tutorial2*.

Start by copying the simulator to a directory in which you plan to do your work. For example, to copy the file named `process-run.py` in the directory */public/cs3103/tutorial2* to your current directory, enter:

```
$ cp /public/cs3103/tutorial2/process-run.py .
```

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

Introduction to `process-run.py`

This program, `process-run.py`, allows you to see how process states change as programs run and either use the CPU (e.g., perform an add instruction) or do I/O (e.g., send a request to a disk and wait for it to complete).

As described in the lecture, processes can be in a few different states:

RUNNING: the process is using the CPU right now
READY: the process could be using the CPU right now but some other process is
WAITING: the process is waiting on I/O (e.g., it issued a request to a disk)
DONE: the process is finished executing

In this tutorial, we'll see how these process states change as a program runs, and thus learn a little bit better how these things work.

To run the program and get its options, do this:

```
$ ./process-run.py -h
```

If this doesn't work, type "python2" before the command, like this:

```
$ python2 process-run.py -h
```

What you should see is this:

```
Usage: process-run.py [options]

Options:
  -h, --help            show this help message and exit
  -s SEED, --seed=SEED  the random seed
  -l PROCESS_LIST, --processlist=PROCESS_LIST
                        a comma-separated list of processes to run, in the
                        form X1:Y1,X2:Y2,... where X is the number of
                        instructions that process should run, and Y the
                        chances (from 0 to 100) that an instruction will use
                        the CPU or issue an IO
  -L IO_LENGTH, --iolength=IO_LENGTH
                        how long an IO takes
  -S PROCESS_SWITCH_BEHAVIOR, --switch=PROCESS_SWITCH_BEHAVIOR
                        when to switch between processes: SWITCH_ON_IO,
                        SWITCH_ON_END
```

```
  -I IO_DONE_BEHAVIOR, --iodone=IO_DONE_BEHAVIOR
                        type of behavior when IO ends: IO_RUN_LATER,
                        IO_RUN_IMMEDIATE
  -c                    compute answers for me
  -p, --printstats      print statistics at end; only useful with -c flag
                        (otherwise stats are not printed)
```

The most important option to understand is the `PROCESS_LIST` (as specified by the `-l` or `--processlist` flags) which specifies exactly what each running program (or "process") will do. A process consists of instructions, and each instruction can just do one of two things:

- use the CPU
- issue an IO (and wait for it to complete)

When a process uses the CPU (and does no IO at all), it should simply alternate between **RUNNING** on the CPU or being **READY** to run. For example, here is a simple run that just has one program being run, and that program only uses the CPU (it does no IO).

```
$ ./process-run.py -l 5:100
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  cpu
  cpu
  cpu
  cpu

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its
turn)
```

Here, the process we specified is "5:100" which means it should consist of 5 instructions, and the chances that each instruction is a CPU instruction are 100%.

You can see what happens to the process by using the `-c` flag, which computes the answers for you:

```
$ ./process-run.py -l 5:100 -c
Time     PID: 0       CPU       IOs
  1     RUN:cpu        1
  2     RUN:cpu        1
  3     RUN:cpu        1
  4     RUN:cpu        1
  5     RUN:cpu        1
```

This result is not too interesting: the process is simple in the **RUN** state and then finishes, using the CPU the whole time and thus keeping the CPU busy the entire run, and not doing any I/Os.

Let's make it slightly more complex by running two processes:

```
$ ./process-run.py -l 5:100,5:100
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  cpu
  cpu
  cpu
  cpu
Process 1
  cpu
  cpu
  cpu
  cpu
  cpu
Important behaviors:
  Scheduler will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its turn)
```

In this case, two different processes run, each again just using the CPU. What happens when the operating system runs them? Let's find out:

```
$ ./process-run.py -l 5:100,5:100 -c
Time      PID: 0      PID: 1         CPU         IOs
  1      RUN:cpu      READY           1
  2      RUN:cpu      READY           1
  3      RUN:cpu      READY           1
  4      RUN:cpu      READY           1
  5      RUN:cpu      READY           1
  6         DONE    RUN:cpu           1
  7         DONE    RUN:cpu           1
  8         DONE    RUN:cpu           1
  9         DONE    RUN:cpu           1
 10         DONE    RUN:cpu           1
```

As you can see above, first the process with "process ID" (or "PID") 0 runs, while process 1 is READY to run but just waits until 0 is done. When 0 is finished, it moves to the **DONE** state, while 1 runs. When 1 finishes, the trace is done.

Let's look at one more example before getting to some questions. In this example, the process just issues I/O requests. We specify here that I/Os take 5 time units to complete with the flag `-L`.

```
$ ./process-run.py -l 3:0 -L 5
Produce a trace of what would happen when you run these processes:
Process 0
  io-start
  io-start
  io-start

Important behaviors:
  System will switch when the current process is FINISHED or ISSUES AN IO
  After IOs, the process issuing the IO will run LATER (when it is its
turn)
```

What do you think the execution trace will look like? Let's find out:

```
$ ./process-run.py -l 3:0 -L 5 -c
Time     PID: 0        CPU        IOs
  1  RUN:io-start         1
  2     WAITING                    1
  3     WAITING                    1
  4     WAITING                    1
  5     WAITING                    1
  6* RUN:io-start         1
  7     WAITING                    1
  8     WAITING                    1
  9     WAITING                    1
 10     WAITING                    1
 11* RUN:io-start         1
 12     WAITING                    1
 13     WAITING                    1
 14     WAITING                    1
 15     WAITING                    1
 16*       DONE
```

As you can see, the program just issues three I/Os. When each I/O is issued, the process moves to a **WAITING** state, and while the device is busy servicing the I/O, the CPU is idle.

Let's print some stats (run the same command as above, but with the `-p` flag) to see some overall behaviors:

```
Stats: Total Time 16
Stats: CPU Busy 3 (18.75%)
Stats: IO Busy  12 (75.00%)
```

As you can see, the trace took 16 clock ticks to run, but the CPU was only busy less than 20% of the time. The IO device, on the other hand, was quite busy. In general, we'd like to keep all the devices busy, as that is a better use of resources.

There are a few other important flags:

- **-s SEED, --seed=SEED**: the random seed. This gives you way to create a bunch of different jobs randomly.
- **-L IO_LENGTH, --iolength=IO_LENGTH**: this determines how long IOs take to complete (default is 5 ticks).
- **-S PROCESS_SWITCH_BEHAVIOR, --switch=PROCESS_SWITCH_BEHAVIOR**: when to switch between processes: SWITCH_ON_IO, SWITCH_ON_END. This determines when we switch to another process: SWITCH_ON_IO, the system will switch when a process issues an IO; SWITCH_ON_END, the system will only switch when the current process is done.
- **-I IO_DONE_BEHAVIOR, --iodone=IO_DONE_BEHAVIOR**: type of behavior when IO ends: IO_RUN_LATER, IO_RUN_IMMEDIATE. This determines when a process runs after it issues an IO: IO_RUN_IMMEDIATE: switch to this process right now; IO_RUN_LATER: switch to this process when it is natural to (e.g., depending on process-switching behavior).

Now go answer the questions to learn more.

Questions

1. Run `process-run.py` with the following flags: `-l 5:100,5:100`. What should the CPU utilization be (e.g., the percent of time the CPU is in use?) Why do you know this? Use the `-c` and `-p` flags to see if you were right.

2. Now run with these flags: `-l 4:100,1:0`. These flags specify one process with 4 instructions (all to use the CPU), and one that simply issues an I/O and waits for it to be done. How long does it take to complete both processes? Use `-c` and `-p` to find out if you were right.

3. Switch the order of the processes: `-l 1:0,4:100`. What happens now? Does switching the order matter? Why? (As always, use `-c` and `-p` to see if you were right)

4. We'll now explore some of the other flags. One important flag is `-S`, which determines how the system reacts when a process issues an I/O. With the flag set to `SWITCH_ON_END`, the system will **NOT** switch to another process while one is doing I/O, instead waiting until the process is completely finished. What happens when you run the following two processes (`-l 1:0,4:100 -c -S SWITCH_ON_END`), one doing I/O and the other doing CPU work?

5. Now, run the same processes, but with the switching behavior set to switch to another process whenever one is **WAITING** for I/O (`-l 1:0,4:100 -c -S SWITCH_ON_IO`). What happens now? Use `-c` and `-p` to confirm that you are right.

6. One other important behavior is what to do when an I/O completes. With `-I IO_RUN_LATER`, when an I/O completes, the process that issued it is not necessarily run right away; rather, whatever was running at the time keeps running. What happens when you run this combination of processes? (`-l 3:0,5:100,5:100,5:100 -S SWITCH_ON_IO -I IO_RUN_LATER -c -p`) Are system resources being effectively utilized?

7. Now run the same processes, but with `-I IO_RUN_IMMEDIATE` set, which immediately runs the process that issued the I/O. How does this behavior differ? Why might running a process that just completed an I/O again be a good idea?

8. Now run with some randomly generated processes: `-s 1 -l 3:50,3:50` or `-s 2 -l 3:50,3:50` or `-s 3 -l 3:50,3:50`. See if you can predict how the trace will turn out. What happens when you use the flag `-I IO_RUN_IMMEDIATE` vs. `-I IO_RUN_LATER`? What happens when you use `-S SWITCH_ON_IO` vs. `-S SWITCH_ON_END`?