**Overview**

This program, called process-run.py, allows you to see how the state of a process state changes as it runs on a CPU. As described in the chapter, 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 (alas) some other process is

BLOCKED - the process is waiting on I/O

(e.g., it issued a request to a disk)

DONE - the process is finished executing

In this homework, 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:

prompt> ./process-run.py -h

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

prompt> python 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).

prompt> ./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)

prompt>

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:

prompt> ./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:

prompt> ./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:

prompt> ./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.

prompt> ./process-run.py -l 3:0 -L 5

Produce a trace of what would happen when you run these processes:

Process 0

io

io\_done

io

io\_done

io

io\_done

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:

prompt> ./process-run.py -l 3:0 -L 5 -c

Time PID: 0 CPU IOs

1 RUN:io 1

2 BLOCKED 1

3 BLOCKED 1

4 BLOCKED 1

5 BLOCKED 1

6 BLOCKED 1

7\* RUN:io\_done 1

8 RUN:io 1

9 BLOCKED 1

10 BLOCKED 1

11 BLOCKED 1

12 BLOCKED 1

13 BLOCKED 1

14\* RUN:io\_done 1

15 RUN:io 1

16 BLOCKED 1

17 BLOCKED 1

18 BLOCKED 1

19 BLOCKED 1

20 BLOCKED 1

21\* RUN:io\_done 1

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

To handle the completion of the I/O, one more CPU action takes place. Note that a single instruction to handle I/O initiation and completion is not particularly realistic, but just used here for simplicity.

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

Stats: Total Time 21

Stats: CPU Busy 6 (28.57%)

Stats: IO Busy 15 (71.43%)

As you can see, the trace took 21 clock ticks to run, but the CPU was busy less than 30% of the time. The I/O 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 at the back of the chapter to learn more, please.