process-run.py protocol

Josef Müller, Ismail Zeybek

October 7, 2019

Contents

1 process-run.py tasks 1-4

1.1 Task 1

Run the Programm with the following flags:

```
./process-run.py -l 5:100, 5:100
```

What should the CPU utilization be? Why do you know this? Use the -c and -p flags to see if you were right.

```
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
```

Answer:

> The CPU utilization should be 100. This is because of the parameters we run the python script with. The first parameter is the number of instructions that process should run, the second parameter the chances (0 - 100) that instruction will use the CPU or issue an I/O

Solution:

```
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

Stats: Total Time 10
Stats: CPU Busy 10 (100.00)
Stats: IO Busy  0 (0.00)
```

1.2 Task 2

Now run with these flags:

```
./process-run.py -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.

```
Produce a trace of what would happen when you run these processes:
Process 0
  cpu
  cpu
  cpu
  cpu
  cpu

Process 1
  io
```

Answer:

> It takes about 4 ticks to complete the first process. After that there will be a context switch to the second process. It'll take about 4+1 ticks to issue an I/O and wait for it. After that there's still one tick to complete

Solution:

```
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        DONE     RUN:io          1
  6        DONE    WAITING                    1
  7        DONE    WAITING                    1
  8        DONE    WAITING                    1
  9        DONE    WAITING                    1
 10*       DONE       DONE

Stats: Total Time 10
Stats: CPU Busy 5 (50.00)
Stats: IO Busy  4 (40.00)
```

1.3 Task 3

Now switch the order of the processes:

```
./process-run.py -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)

Answer:

> It won't take as long as in the 2. question because after issuing an I/O the process is going to switch to his waiting state and the second process is performing his CPU instructions. This means that while the first process is in a waiting state the second process can do his CPU tasks. After having finished with that the second process will be completed and the first can return back and complete too.

Solution:

```
Time     PID: 0     PID: 1        CPU        IOs
  1      RUN:io      READY          1
  2     WAITING    RUN:cpu          1          1
  3     WAITING    RUN:cpu          1          1
  4     WAITING    RUN:cpu          1          1
  5     WAITING    RUN:cpu          1          1
  6*       DONE       DONE

Stats: Total Time 6
Stats: CPU Busy 5 (83.33)
Stats: IO Busy  4 (66.67)
```

1.4 Task 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, one doing I/O and the other doing CPU work?

```
-l 1:0,4:100 -c -S SWITCH_ON_END
```

Answer:

> It will take a great amount of time to complete both processes because there'll be no concurrency. Both processes will be run like in a FIFO without any sort of parallelism. The first process is going to issue an I/O and the second process is going to rest in its ready state instead of being switched.

Solution:

```
Time     PID: 0     PID: 1        CPU        IOs
  1      RUN:io      READY          1
  2     WAITING      READY                    1
  3     WAITING      READY                    1
  4     WAITING      READY                    1
  5     WAITING      READY                    1
  6*       DONE    RUN:cpu          1
  7        DONE    RUN:cpu          1
  8        DONE    RUN:cpu          1
  9        DONE    RUN:cpu          1
```