**5600**

**Ziyun Chen**

**HW1 - Chapter 4**

**Text

Description automatically generatedQ1:** By running the process-run.py file using the flag ”./process-run.py -l 5:100,5:100”, I got the following result:

From this it seems that the cpu is constantly being used. So, the

percent of time the CPU is in use is 100%.

Text

Description automatically generated with low confidence

Text

Description automatically generated

But adding -c and -p command, it proves that CPU is 100% in use.

**Q2:** When running with “./process-run.py -l 4:100,1:0”, I got this result:

Text

Description automatically generated

But after running it with -c and -p, it turns out:

A screenshot of a computer

Description automatically generated with medium confidence

Process took 4, since it has 4 instructions, while process took 5, plus 1 final step. So the total time is 5+4+1 = 10.

**Q3:** After running the file with: ./process-run.py -l 1:0,4:100, I got:

A screenshot of a computer

Description automatically generated with medium confidence

My guesses was: because from Q2 we know process 0 has 5 instructions in total. So, when process 0 started waiting for IO, it held, and switched to process 1. After process 1 was finished, it returned back to process 0. I assumed the total time should stay the same, which is 10. However, after I run the file with -c -p, I got:

Text

Description automatically generated

The total time was reduced to 6. Once process 0 was waiting for IO, CPU was not used by process 0 anymore, so it started running process1. From 2 to 5, I/O and cpu were running at the same time. Which saved a lot of time. So, the order switch matters.

**Q4:**

Here is the result I got:

A screenshot of a computer

Description automatically generated with medium confidence

Process 1 didn’t start until process 0 was finished, even when process 0 was not using the cpu.

**Q5:**

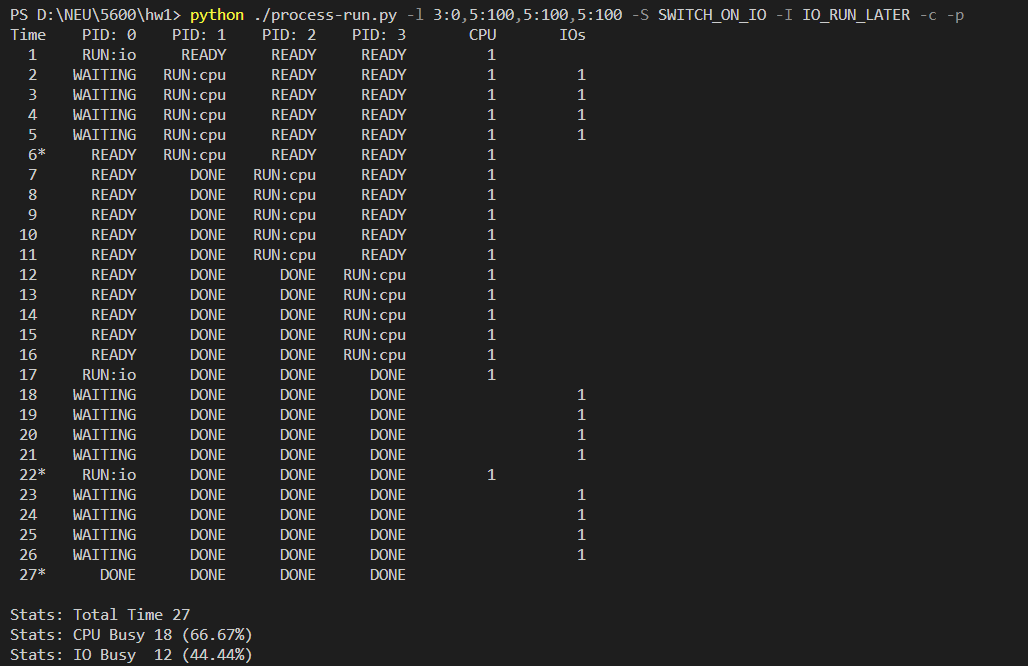
Here is the result I got:

A screenshot of a computer

Description automatically generated with medium confidenceWe achieved the same result as we did from Q3. Process 1 started as soon as process 0 started waiting.

**Q6:**

Here is the result I got,



From this we can tell that, when process 0 first started waiting for IO, process 1 started, after process 1 ended, process 2 started, after process 2 ended, process 3 started, and after process 3, the rest of process 0 continued. And it is not a very efficient way to use system resources, since between the end of process 1 and the start of process 2, we can run process 0 once, and let it wait for IO, while process 2 runs. And do the same thing again between process 2 and process 3, this way it can save us at least 4 running time.

**Q7:** Here is what I got:

A picture containing text

Description automatically generated

The differences between this time and last time are: process 0 got the priority to run first. After process 0 finished one execution, all the other processes needed to wait until process 0 started the next one. It might be a good idea simply because, comparing to last time, this one saved a lot of time, and the cpu was constantly being used. No resource was wasted.

**Q8:**

Here is what I got from random generation:

A computer screen capture

Description automatically generated with medium confidence

From this we can tell that: process 0 is: cpu, io, cpu, and process 1 is: io, io, cpu.

If I use “-I IO\_RUN\_IMMEDIATE”, nothing changes.

A screenshot of a computer

Description automatically generated with medium confidence

If I use “-I IO RUN LATER”,

A computer screen capture

Description automatically generated with medium confidence

Nothing changes as well.

If I use “-S SWITCH\_ON\_IO”, nothing changes.

A computer screen capture

Description automatically generated with medium confidence

If I use ” -S SWITCH\_ON\_END”, when process 0 issued its first IO, process 1 had to wait until it ended.

Text

Description automatically generated