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.

CPU utilization will be 100% of the time since both processes never used I/O. This is because all the instructions happen on the CPU. I know it is true because both two processes' second parameter is 100, meaning the CPU will do all the jobs, and they never touch the I/O.

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.

It will take Ten units of time to finish the instruction. This is because the first process has four instructions, which all use the CPU. Then, the first process will use four units of time to finish execution. The second process only has one instruction. It will first read instruction by CPU (take 1 unit of time), then CPU pass to IO. IO will take four units of time to finish. Then IO will pass back to the CPU. Finally, the CPU will take one unit of time to complete the execution. Thus, the second process will take a total 1+4+1 = 6 units times. Notice the first process will execute first. Then the total execution times for both processes is 4+6 =10 units.

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)

Now, the total run time is six units. So, yes, switching does matter. The first process is only for IO, the CPU will first use one unit of time to pass process one to IO, and the CPU starts executing process two's instructions. Meanwhile, IO will begin to handle process one for four units of time. Both processes, one and two, will be executed simultaneously by IO and CPU. Thus, total runtime will be 1 + 4(both process1 and prcess2) + 1 = 6 units. Notice the last unit of time is used to kill both processes.

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?

This time, the CPU will not switch to process two once it passes process one to IO; instead, the CPU will wait for IO to finish process one, then it will switch to process two and start running all the instructions. This way CPU waste a lot of time just waiting for Process one to be finished. CPU will use one unit of time to pass process one to IO and wait four units for process one to finish. Lastly, it will use four units of time to complete process two. Thus, the total run time is nine units.

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.

At begging is the same, the CPU will pass process one to IO. However, instead of waiting for process one to finish and then start running process two, the CPU will immediately begin running process two meanwhile IO handles process one. In that case, the CPU won't just wait; IO and CPU will handle both processes in parallel. The total run time will be six, faster than letting the CPU wait.

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? (Run ./process-run.py -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?

Process one instructions are all about IO. At the start, the CPU will pass process one to IO, then start running process two. Both process one and process two are running in parallel by the CPU and the IO. It takes IO four units to complete one IO instruction. At the time six, process one still had two IO instructions left, but it never had a chance back to the CPU and let the CPU tell IO to do another IO instruction for process one. Process one will wait until all process two and process three to complete, and then the CPU assigns two IO instructions to the IO, which takes eight units of time. This is not effective enough because we let IO waste a lot of time waiting. Instead, we should let process one immediately back to the CPU once it finishes the first IO instruction. Then, the CPU should pause the current process and pass process one back to IO, running the second Instruction. In this way, IO and CPU will be in parallel for a maximum of eight units of time. Notice once the second process and third processes are finished. The CPU also wastes a lot of time waiting for process one back from IO.

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?

This behavior is different because process one can immediately go back to the CPU once IO finishes one IO instruction. Then the CPU can immediately assign another IO instruction of process one to IO. In the previous situation, after completing the first IO instruction, process one must wait for processes two and three complete; then, it will have to go back to the CPU. Then the CPU can assign second IO instruction. Then the CPU will wait four units of time that process one complete second IO instruction. Finally, the CPU will assign the last IO instruction. Running a process that just completes an I/O again would be a good idea because instead let I/O just wait there doing nothing, we can immediately assign another job to IO. Utilize as much I/O time as possible. Moreover, in this way, we can utilize CPU time as well. The last two IO instruction CPU just wait for process one to come back from IO, that waste a lot of time.

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?

The IO_RUN_IMMEDIATE and IO_RUN_LATER will make no difference because all the I/O instruction takes four units of time to run, and there are only three CPU instructions for the other process. Thus, IO run time always finishes later than the CPU.

The SWITCH_ON_IO will be faster than SWITCH_ON_END. Because SWITCH_ON_END will force the CPU to wait for IO to finish, then start running. Thus, the CPU and IO will never have a chance to run in parallel. Either the CPU is running, or IO is running but never both. If we have only one process, this wouldn't be a problem, but once we have two processes and mix with IO instructions and CPU instructions, this will decrease utilization for both CPU and IO.