Chapter 4 HW

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.
A:
CPU utilization is 100% because the processes both use CPU.

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.
A:
It takes 11 clock ticks(4 + 1 + 5 + 1). CPU:6/11; I/O: 5/11

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)
A:
It only takes 7 clock ticks(1 + 5 + 1). During the 5 ticks of blocked due to I/O, CPU can work on the process 2. The order matters since time-sharing works and it's more efficient.

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?
A:
Unlike the question 3, the given command of this problem makes the CPU less efficient because it doesn't allow time-sharing. The clock ticks become 11, like the question2 result, since CPU can't work on the process 2 when the process 1 works on I/O, and the process 2 needs to wait for the process 1 finishing its 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.
A:
The clock ticks become 7, like the question 3 result. CPU works on the process 2 when the process 1 working on I/O, and it's more efficient. Therefore, SWITCH_ON_IO can utilize CPU more efficient.

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?
A:
With the process switching behavior SWITCH_ON_IO, once a process issues I/O, CPU works on other processes that are Ready. Therefore, after the process 1 calls I/O, CPU will switch and work on the remaining three processes. Even the process 1 and 2 finish at the same time, the process 1 still needs to wait for the process 3 and 4 finish. Then process 1 starts to issue the second and the third I/O instructions, now the CPU is idle! Therefore, it takes total 31 clock ticks, but CPU only uses 21/31 and it's not efficient.

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?
A:
Unlike the above behavior, now the process issuing an I/O should use process whenever the I/O finishes its job. Now the process 1 class the second I/O instruction when the process 2 finishes, and the process 1 becomes blocked after issuing I/O and CPU starts to work on the process 3. With the IO done behavior IO_RUN_IMMEDIATE, the process can resume work immediately and doesn't need to wait other processes. Therefore, it takes total 21 ticks and CPU works 21/21!

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?
A:
1. With SWITCH_ON_IO:
IO_RUN_IMMEDIATE has better performance since the process finishing I/O can run the next instruction immediately and it doesn't need to wait for other processes.
IO_RUN_LATER has worse performance if there are many Ready processes, and the process finishes I/O needs to wait for other processes.
Therefore, if there are multiple processes and many processes have higher opportunities using CPU, IO_RUN_IMMEDIATE is better

2. With SWITCH_ON_END:
This behavior limits time-sharing, and CPU can work on the next process only after finishing the previous process. Usually, if has worse performance than SWITCH_ON_IO.