Nicholas Garrett

Professor Fouda

CS 4461

1/28/2022

Homework 1

Chapter 4

4.1.

I would assume that because the script says "System will switch when the current process is FINISHED", that the CPU will have a 100% time use.

```
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%)
```

From the output of the code with -c and -p flags, it appears I was correct in my answer.

4.2.

I would assume that since the flags specify a single process to run with four instructions and a single I/O wait operation, that it would take four ticks to complete the first process and something more than that to finish both processes.

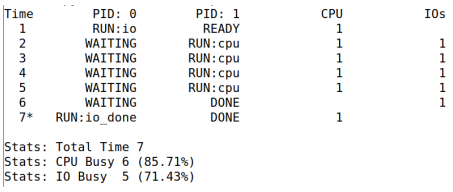
```
Time        PID: 0       PID: 1           CPU           IOs
  1         RUN:cpu       READY             1
  2         RUN:cpu       READY             1
  3         RUN:cpu       READY             1
  4         RUN:cpu       READY             1
  5            DONE      RUN:io             1
  6   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

Stats: Total Time 11
Stats: CPU Busy 6 (54.55%)
Stats: IO Busy  5 (45.45%)
```

From the code output, it appears that there are six busy CPU ticks. I had not considered that the I/O complete would itself take a tick.

4.3

I would assume that because the CPU will set the process checking IO to wait, the four operations will run while the I/O waits. Therefore, I think that this configuration might be able to run in as little as 6 processes.



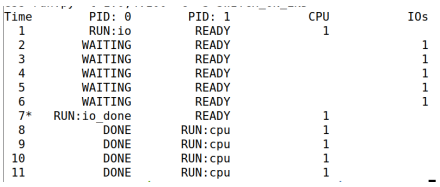
```
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:io         READY             1
  2        WAITING       RUN:cpu            1             1
  3        WAITING       RUN:cpu            1             1
  4        WAITING       RUN:cpu            1             1
  5     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

Stats: Total Time 7
Stats: CPU Busy 6 (85.71%)
Stats: IO Busy  5 (71.43%)
```

From the output, it looks like this assessment was correct, though a single tick occured where the CPU was no longer running the four operations though was still waiting on the I/O operation.

4.4

From the explanation of the flag set, the logical outcome of running the script with its I/O process being followed by four operations is that the number of ticks required is whatever number is required to operate the I/O followed by four process ticks.

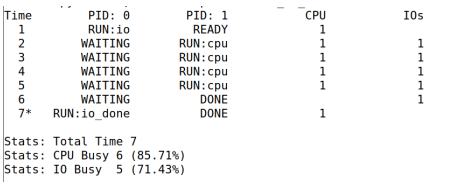


```
Time        PID: 0        PID: 1           CPU           IOs
  1         RUN:io         READY             1
  2        WAITING         READY                         1
  3        WAITING         READY                         1
  4        WAITING         READY                         1
  5        WAITING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

From the code's output, it looks like this assessment is correct.

4.5

Based on the description of how the SWITCH_ON_IO flag operates, I would assume that this call will lead to the four processes starting as soon as the WAIT command occurs on the I/O-dependant process.



```
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        WAITING          DONE                          1
  7*   RUN:io_done          DONE             1

Stats: Total Time 7
Stats: CPU Busy 6 (85.71%)
Stats: IO Busy  5 (71.43%)
```

From the console output, it looks like this assessment was correct.

4.6 From the description of how the flags operate, I think that the first I/O process will start, then WAIT for the I/O response. During this time, the five processes following it will run. And if it is still not done, then the five processes following them will run, etc. After those processes, I expect the other two I/O dependent processes will run.

```
  2       WAITING      RUN:cpu        READY        READY
  3       WAITING      RUN:cpu        READY        READY
  4       WAITING      RUN:cpu        READY        READY
  5       WAITING      RUN:cpu        READY        READY
  6       WAITING      RUN:cpu        READY        READY
  7*        READY         DONE      RUN:cpu        READY
  8         READY         DONE      RUN:cpu        READY
  9         READY         DONE      RUN:cpu        READY
 10         READY         DONE      RUN:cpu        READY
 11         READY         DONE      RUN:cpu        READY
 12         READY         DONE         DONE     RUN:cpu
 13         READY         DONE         DONE     RUN:cpu
 14         READY         DONE         DONE     RUN:cpu
 15         READY         DONE         DONE     RUN:cpu
 16         READY         DONE         DONE     RUN:cpu
 17   RUN:io_done         DONE         DONE        DONE
 18        RUN:io         DONE         DONE        DONE
 19       WAITING         DONE         DONE        DONE
 20       WAITING         DONE         DONE        DONE
 21       WAITING         DONE         DONE        DONE
 22       WAITING         DONE         DONE        DONE
 23       WAITING         DONE         DONE        DONE
 24*  RUN:io_done         DONE         DONE        DONE
 25        RUN:io         DONE         DONE        DONE
 26       WAITING         DONE         DONE        DONE
 27       WAITING         DONE         DONE        DONE
 28       WAITING         DONE         DONE        DONE
 29       WAITING         DONE         DONE        DONE
 30       WAITING         DONE         DONE        DONE
 31*  RUN:io_done         DONE         DONE        DONE

Stats: Total Time 31
```

From the output generated, it looks like my presumption was wrong. It looks like the I/O process started then waited for the I/O return. I had assumed that only one or two of the 5-operation processes would execute before the next I/O process. Though, it looks like they all ran, even after the I/O process was ready.

I say that this is inefficient because it leaves I/O processes to execute afterwards, which leave unused CPU time where it simply waits for I/O return.

4.7

With the new execution flags, it sounds like the I/O process will run then wait. During this time, the five processes following it will run. Then the I/O process will continue to execute, which will run its second I/O call and wait, during which time the next five processes will run, and the same thing will happen again.

```
Time        PID: 0       PID: 1       PID: 2       PID: 3
  1         RUN:io         READY        READY        READY
  2       WAITING      RUN:cpu        READY        READY
  3       WAITING      RUN:cpu        READY        READY
  4       WAITING      RUN:cpu        READY        READY
  5       WAITING      RUN:cpu        READY        READY
  6       WAITING      RUN:cpu        READY        READY
  7*  RUN:io_done         DONE        READY        READY
  8         RUN:io         DONE        READY        READY
  9       WAITING         DONE     RUN:cpu        READY
 10       WAITING         DONE     RUN:cpu        READY
 11       WAITING         DONE     RUN:cpu        READY
 12       WAITING         DONE     RUN:cpu        READY
 13       WAITING         DONE     RUN:cpu        READY
 14*  RUN:io_done         DONE        DONE        READY
 15        RUN:io         DONE        DONE        READY
 16       WAITING         DONE        DONE     RUN:cpu
 17       WAITING         DONE        DONE     RUN:cpu
 18       WAITING         DONE        DONE     RUN:cpu
 19       WAITING         DONE        DONE     RUN:cpu
 20       WAITING         DONE        DONE     RUN:cpu
 21*  RUN:io_done         DONE        DONE        DONE

Stats: Total Time 21
Stats: CPU Busy 21 (100.00%)
Stats: IO Busy  15 (71.43%)
```

From the output, it looks like this assessment is correct. The executed code runs much more efficiently this time, keeping the CPU busy at 100%. Running the processes that had just made an I/O cal again helps to improve efficiency if they call multiple I/O returns.l

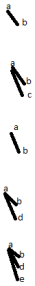
4.8

From what I have learned while doing this assignment, I think that the first process will run, then wait on its I/O. During this time, the second process will run and wait on its I/O. The first process will run next as its I/O call should finish first and will finish, followed by the second process.

**********uncompleted**********

Chapter 5

5.1 Below is plotted what I expect he tree would look like, looking at the actions.



And here we have the actual process tree. It looks like I was able to interpret the information correctly.

```
                                    Process Tree:
                                        a

Action: a forks b

                                        a
                                        └── b
Action: a forks c

                                        a
                                        ├── b
                                        └── c
Action: c EXITS

                                        a
                                        └── b
Action: a forks d

                                        a
                                        ├── b
                                        └── d
Action: a forks e

                                        a
                                        ├── b
                                        ├── d
                                        └── e
```

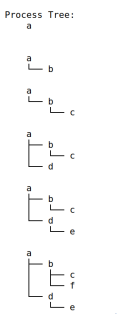
.

5.2

As the percentage likelyhood for a fork increases, the number of forks on each of the processes also increases. At 0.9, every process has a fork.

5.3

From the process tree:



it looks like the actions were:

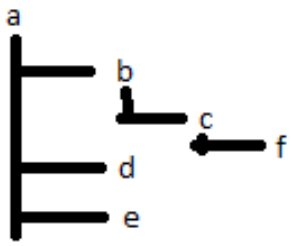
a forks b
b forks c
a forks d
d forks e
b forks f

5.4

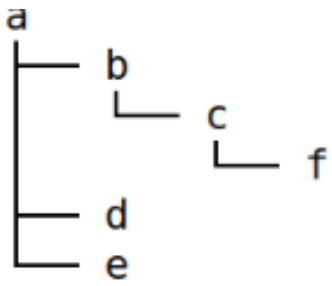
From the prompt, I assumed that when c was exited, that all the forked children of c would either dissapear of link to the parent of c, or b. However, it looks like this only occurs on the -R flag, and otherwise the children link from a.

5.5

From the operations, I drew the following process tree.



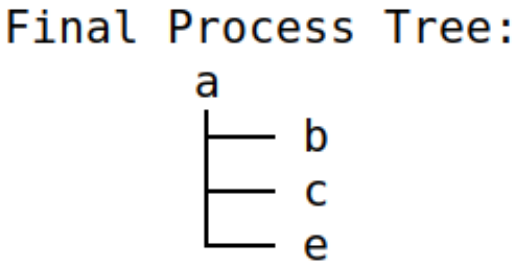
And here is the process tree that was returned with the -c flag:



.

5.6

Running the execution command with the relavent flags, I got:



To the left, there were five action tags, representing that five actions took place.

I would guess that the processes were along the lines of:

a forks b
a forks c
a forks d
a forks e
d exits

Unfortunately, all I can really determine is that d was removed and the fork for e occurred after d was created.

5.1
code in varAfterFork.c
The variable does not change its value between the parent and child processes. Then, when each changes the value for x, the change does not affect the variable's value in the other process.

5.2
**********uncompleted**********

5.3
code in helloGoodbye.c I can not guarantee that the child process will complete first, though I force the parent to wait by having it run 100000 iterations which should slow it long enough for child to finish.

5.4
code in forkedLS.c
5.5 5.6 5.7 5.8