Chapter 5 Homework (Simulation)

1. Run ./fork.py -s 10 and see which actions are taken. Can you predict what the process tree looks like at each step? Use the -c flag to check your answers. Try some different random seeds (-s) or add more actions (-a) to get the hang of it.

Yes, it is easy to determine the action of each step given the graph.

```
./fork.py -s 10 -c
ARG seed 10
ARG fork_percentage 0.7
ARG actions 5
ARG action_list
ARG show_tree False
ARG just_final False
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve True
                           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
```

2. One control the simulator gives you is the fork percentage, controlled by the -f flag. The higher it is, the more likely the next action is a fork; the lower it is, the more likely the action is an exit. Run the simulator with a large number of actions (e.g., -a 100) and vary the fork percentage from 0.1 to 0.9. What do you think the resulting final process trees will look like as the percentage changes? Check your answer with -c.

With the random seed set to 10 (this is very deterministic, as the result of each run will be exactly the same given the same random seed). In theory, as fork percentage goes down, there will be less child processes to be created. The results are listed as below:

```
./fork.py -s 10 -f 100 -c
ARG seed 10
ARG fork_percentage 100.0
ARG actions 5
ARG action_list
ARG show_tree False
ARG just_final False
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve True
                        Process Tree:
                          a
Action: a forks b
                          a
                          └── b
Action: a forks c
                          a
                          ├── b
                          └── c
Action: c forks d
                          a
                          ├── b
                          └── c
                              └── d
Action: a forks e
                          a
                          ├── b
                          ├── c
                          │   └── d
                          └── e
Action: b forks f
                          a
                          ├── b
                          │   └── f
                          ├── c
                          │   └── d
                          └── e

./fork.py -s 10 -f 0.9 -c
ARG seed 10
ARG fork_percentage 0.9
ARG actions 5
ARG action_list
ARG show_tree False
ARG just_final False
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve True
                        Process Tree:
                          a
Action: a forks b
                          a
                          └── b
Action: a forks c
                          a
                          ├── b
                          └── c
Action: c forks d
                          a
                          ├── b
                          └── c
                              └── d
Action: a forks e
                          a
                          ├── b
                          ├── c
                          │   └── d
                          └── e
Action: b forks f
```

```
                      a
                      ├── b
                      │   └── f
                      ├── c
                      │   └── d
                      └── e

./fork.py -s 10 -f 0.5 -c
ARG seed 10
ARG fork_percentage 0.5
ARG actions 5
ARG action_list
ARG show_tree False
ARG just_final False
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve True
                           Process Tree:
                               a
Action: a forks b
                               a
                               └── b
Action: b EXITS
                               a
Action: a forks c
                               a
                               └── c
Action: c EXITS
                               a
Action: a forks d
                               a
                               └── d


./fork.py -s 10 -f 0.1 -c
ARG seed 10
ARG fork_percentage 0.1
ARG actions 5
ARG action_list
ARG show_tree False
ARG just_final False
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve True
                           Process Tree:
                               a
Action: a forks b
                               a
                               └── b
Action: b EXITS
                               a
Action: a forks c
                               a
                               └── c
Action: a forks d
                               a
                               ├── c
                               └── d
Action: c EXITS
```

```
                     a
                     └── d
```

3. Now, switch the output by using the -t flag (e.g., run ./fork.py -t). Given a set of process trees, can you tell which actions were taken?

Yes, actions are highlighted in yellow:

```
./fork.py -t
ARG seed -1
ARG fork_percentage 0.7
ARG actions 5
ARG action_list
ARG show_tree True
ARG just_final False
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve False
                  Process Tree:
                     a
Action? a fork b
                     a
                     └── b
Action? b EXITS
                     a
Action? a fork C
                     a
                     └── c
Action? c fork d
                     a
                     └── c
                         └── d
Action? c fork e
                     a
                     └── c
                         ├── d
                         └── e
```

4. One interesting thing to note is what happens when a child exits; what happens to its children in the process tree? To study this, let's create a specific example: ./fork.py -A a+b,b+c,c+d,c+e,c-. This example has process 'a' create 'b', which in turn creates 'c', which then creates 'd' and 'e'. However, then, 'c' exits. What do you think the process tree should like after the exit? What if you use the -R flag? Learn more about what happens to orphaned processes on your own to add more context.

With -R flag, the child process will find the parent's parent as its new parent if the parent process exits. See parts highlighted in yellow:

```
./fork.py -A a+b,b+c,c+d,c+e,c- -c
ARG seed -1
ARG fork_percentage 0.7
ARG actions 5
ARG action_list a+b,b+c,c+d,c+e,c-
ARG show_tree False
ARG just_final False
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve True
                     Process Tree:
                        a
Action: a forks b
                        a
                        └── b
Action: b forks c
                        a
                        └── b
                            └── c
Action: c forks d
                        a
                        └── b
                            └── c
                                └── d
Action: c forks e
                        a
                        └── b
                            └── c
                                ├── d
                                └── e
Action: c EXITS
                        a
                        ├── b
                        ├── d
                        └── e
./fork.py -A a+b,b+c,c+d,c+e,c- -R -c
ARG seed -1
ARG fork_percentage 0.7
ARG actions 5
ARG action_list a+b,b+c,c+d,c+e,c-
ARG show_tree False
ARG just_final False
ARG leaf_only False
ARG local_reparent True
ARG print_style fancy
ARG solve True
                     Process Tree:
                        a
Action: a forks b
                        a
                        └── b
Action: b forks c
                        a
                        └── b
                            └── c
Action: c forks d
                        a
                        └── b
                            └── c
                                └── d
Action: c forks e
                        a
                        └── b
                            └── c
                                ├── d
                                └── e
Action: c EXITS
                        a
                        └── b
                            ├── d
```

```
                      └── e
```

5. One last flag to explore is the -F flag, which skips intermediate steps and only asks to fill in the final process tree. Run ./fork.py -F and see if you can write down the final tree by looking at the series of actions generated. Use different random seeds to try this a few times.

```
./fork.py -s 10 -f 0.5 -F  -c
ARG seed 10
ARG fork_percentage 0.5
ARG actions 5
ARG action_list
ARG show_tree False
ARG just_final True
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve True
                  Process Tree:
                     a
Action: a forks b
Action: b EXITS
Action: a forks c
Action: c EXITS
Action: a forks d
                Final Process Tree:
                     a
                     └── d
```

6. Finally, use both -t and -F together. This shows the final process tree, but then asks you to fill in the actions that took place. By looking at the tree, can you determine the exact actions that took place? In which cases can you tell? In which can't you tell? Try some different random seeds to delve into this question.

Yes, but there is no guarantee of the ordering of each action, as it could vary.

```
./fork.py -t -F
ARG seed -1
ARG fork_percentage 0.7
ARG actions 5
ARG action_list
ARG show_tree True
ARG just_final True
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve False
                  Process Tree:
                     a
Action? a fork b
Action? b fork c
Action? a fork d
Action? c EXIT
Action? b EXIT
```

```
Final Process Tree:
    a
    └── d
```

Chapter 5 Homework (Code)

1. Write a program that calls fork(). Before calling fork(), have the main process access a variable (e.g., x) and set its value to something (e.g., 100). What value is the variable in the child process? What happens to the variable when both the child and parent change the value of x?

Code in github. Essentially, the child gets a copy of the parent's memory space at the time of fork(). Changes made by the child to its variables do not affect the parent's variables, and vice versa — because they now have separate memory spaces.The memory is shared until either process modifies it, then the OS makes a separate copy.

2. Write a program that opens a file (with the open() system call) and then calls fork() to create a new process. Can both the child and parent access the file descriptor returned by open()? What happens when they are writing to the file concurrently, i.e., at the same time?

Code in github. When a process opens a file (via open()), it gets a file descriptor that refers to an entry in the system-wide open file table.
When the process calls fork(), the child inherits a copy of the file descriptor, but both parent and child refer to the same open file table entry.
If they write concurrently, data may lead to nondeterministic ordering or interleaving.
unless synchronization is used.

3. Write another program using fork(). The child process should print "hello"; the parent process should print "goodbye". You should try to ensure that the child process always prints first; can you do this without calling wait() in the parent?

Code in github. Yes, we can do this without wait(), either with a quick sleep() trick, or more reliably with an IPC like pipe.

4. Write a program that calls fork() and then calls some form of exec() to run the program /bin/ls. See if you can try all of the variants of exec(), including (on Linux) execl(), execle(), execlp(), execv(), execvp(), and execvpe(). Why do you think there are so many variants of the same basic call?

Code and explanation in github.

5. Now write a program that uses wait() to wait for the child process to finish in the parent. What does wait() return? What happens if you use wait() in the child?

Code in github. If the child process has its own child process, then it will wait for its child. If not, it will return -1, which raises an error. Only parent processes can wait for their children.

6. Write a slight modification of the previous program, this time using waitpid() instead of wait(). When would waitpid() be useful?

Code in github. waitpid() allows us to wait for a specific child (given it PID). When we have multiple children and want to wait for a particular one. This allows us to have more control and aching non-blocking wait.

7. Write a program that creates a child process, and then in the child closes standard output (STDOUT FILENO). What happens if the child calls printf() to print some output after closing the descriptor?
Code in github.

1. **The printf() call silently fails - No output appears on screen**
2. **printf() returns a negative value indicating an error**
3. **The program doesn't crash - It continues running normally**
4. **stderr still works - You can still output to standard error**

8. Write a program that creates two children, and connects the standard output of one to the standard input of the other, using the pipe() system call.
Code in github.

Chapter 6 Homework (Measurement)

Code in github.

Chapter 7

1. Compute the response time and turnaround time when running three jobs of length 200 with the SJF and FIFO schedulers.

The results will be identical, as job lengths are the same (200). Essentially, there is no difference between SJF and FIFO.

```
./scheduler.py -p SJF -l 200,200,200 -c
ARG policy SJF
ARG jlist 200,200,200
Here is the job list, with the run time of each job:
  Job 0 ( length = 200.0 )
  Job 1 ( length = 200.0 )
  Job 2 ( length = 200.0 )
** Solutions **
Execution trace:
  [ time   0 ] Run job 0 for 200.00 secs ( DONE at 200.00 )
  [ time 200 ] Run job 1 for 200.00 secs ( DONE at 400.00 )
  [ time 400 ] Run job 2 for 200.00 secs ( DONE at 600.00 )
Final statistics:
  Job   0 -- Response: 0.00  Turnaround 200.00  Wait 0.00
  Job   1 -- Response: 200.00  Turnaround 400.00  Wait 200.00
  Job   2 -- Response: 400.00  Turnaround 600.00  Wait 400.00
  Average -- Response: 200.00  Turnaround 400.00  Wait 200.00
```

```
./scheduler.py -p FIFO -l 200,200,200 -c
ARG policy FIFO
ARG jlist 200,200,200
Here is the job list, with the run time of each job:
  Job 0 ( length = 200.0 )
  Job 1 ( length = 200.0 )
  Job 2 ( length = 200.0 )
** Solutions **
Execution trace:
  [ time   0 ] Run job 0 for 200.00 secs ( DONE at 200.00 )
  [ time 200 ] Run job 1 for 200.00 secs ( DONE at 400.00 )
  [ time 400 ] Run job 2 for 200.00 secs ( DONE at 600.00 )
Final statistics:
  Job   0 -- Response: 0.00  Turnaround 200.00  Wait 0.00
  Job   1 -- Response: 200.00  Turnaround 400.00  Wait 200.00
  Job   2 -- Response: 400.00  Turnaround 600.00  Wait 400.00
  Average -- Response: 200.00  Turnaround 400.00  Wait 200.00
```

2. Now do the same but with jobs of different lengths: 100, 200, and 300.

```
./scheduler.py -p SJF -l 100,200,300 -c
ARG policy SJF
ARG jlist 100,200,300
Here is the job list, with the run time of each job:
  Job 0 ( length = 100.0 )
  Job 1 ( length = 200.0 )
  Job 2 ( length = 300.0 )
** Solutions **
Execution trace:
  [ time   0 ] Run job 0 for 100.00 secs ( DONE at 100.00 )
  [ time 100 ] Run job 1 for 200.00 secs ( DONE at 300.00 )
  [ time 300 ] Run job 2 for 300.00 secs ( DONE at 600.00 )
Final statistics:
  Job   0 -- Response: 0.00  Turnaround 100.00  Wait 0.00
  Job   1 -- Response: 100.00  Turnaround 300.00  Wait 100.00
  Job   2 -- Response: 300.00  Turnaround 600.00  Wait 300.00
  Average -- Response: 133.33  Turnaround 333.33  Wait 133.33
```

```
./scheduler.py -p FIFO -l 100,200,300 -c
ARG policy FIFO
ARG jlist 100,200,300
Here is the job list, with the run time of each job:
  Job 0 ( length = 100.0 )
  Job 1 ( length = 200.0 )
  Job 2 ( length = 300.0 )
** Solutions **
Execution trace:
  [ time   0 ] Run job 0 for 100.00 secs ( DONE at 100.00 )
  [ time 100 ] Run job 1 for 200.00 secs ( DONE at 300.00 )
  [ time 300 ] Run job 2 for 300.00 secs ( DONE at 600.00 )
Final statistics:
  Job   0 -- Response: 0.00  Turnaround 100.00  Wait 0.00
  Job   1 -- Response: 100.00  Turnaround 300.00  Wait 100.00
  Job   2 -- Response: 300.00  Turnaround 600.00  Wait 300.00
  Average -- Response: 133.33  Turnaround 333.33  Wait 133.33
```

3. Now do the same, but also with the RR scheduler and a time-slice of 1.

```
./scheduler.py -p RR -l 100,200,300 -q 1 -c
ARG policy RR
ARG jlist 100,200,300
Here is the job list, with the run time of each job:
  Job 0 ( length = 100.0 )
  Job 1 ( length = 200.0 )
  Job 2 ( length = 300.0 )
Final statistics:
  Job   0 -- Response: 0.00  Turnaround 298.00  Wait 198.00
  Job   1 -- Response: 1.00  Turnaround 499.00  Wait 299.00
  Job   2 -- Response: 2.00  Turnaround 600.00  Wait 300.00
  Average -- Response: 1.00  Turnaround 465.67  Wait 265.67
```

4. For what types of workloads does SJF deliver the same turnaround times as FIFO?

a. Jobs arrive in increasing order of runtime
   If jobs naturally arrive sorted from shortest to longest, FIFO will execute them in the same order SJF would choose.
   Example:

   Job A arrives at time 0, runtime = 1
   Job B arrives at time 1, runtime = 2
   Job C arrives at time 2, runtime = 3

   Both FIFO and SJF execute: A B C

b. All jobs have the same runtime
   When all jobs take equal time, the order doesn't matter for turnaround time.

c. Only one job in the system
   With a single job, there's no scheduling decision to make.

5. For what types of workloads and quantum lengths does SJF deliver the same response times as RR?

SJF delivers the same response times as RR when: (1) the quantum length is greater than or equal to the longest job, causing RR to behave exactly like SJF since every job completes in its first time slice without any rotation. (2) all jobs have the same runtime and that runtime is less than or equal to the quantum, making the scheduling order irrelevant since all jobs get immediate response time of 0. (3) there's only a single job in the system.

6. What happens to response time with SJF as job lengths increase? Can you use the simulator to demonstrate the trend?

In SJF, longer jobs must wait for ALL shorter jobs to complete before they get any CPU time. As job lengths increase, the longer jobs experience increasingly poor response times.

```
Execution trace:
  [ time   0 ] Run job 0 for 100.00 secs ( DONE at 100.00 )
  [ time 100 ] Run job 1 for 1000.00 secs ( DONE at 1100.00 )
  [ time 1100 ] Run job 2 for 10000.00 secs ( DONE at 11100.00 )
  [ time 11100 ] Run job 3 for 100000.00 secs ( DONE at 111100.00 )
  [ time 111100 ] Run job 4 for 1000000.00 secs ( DONE at 1111100.00 )
Final statistics:
  Job   0 -- Response: 0.00  Turnaround 100.00  Wait 0.00
  Job   1 -- Response: 100.00  Turnaround 1100.00  Wait 100.00
  Job   2 -- Response: 1100.00  Turnaround 11100.00  Wait 1100.00
  Job   3 -- Response: 11100.00  Turnaround 111100.00  Wait 11100.00
  Job   4 -- Response: 111100.00  Turnaround 1111100.00  Wait 111100.00
```

7. What happens to response time with RR as quantum lengths increase? Can you write an equation that gives the worst-case response time, given N jobs?

Response time gets worse as quantum length increases, especially for jobs that arrive later in the queue, until the Q is equal and greater to the job length (there will be no further effect to the response time).

Response time (worst case) = $(N - 1) \times Q$

N = number of jobs in the system

Q = quantum length