Ch5

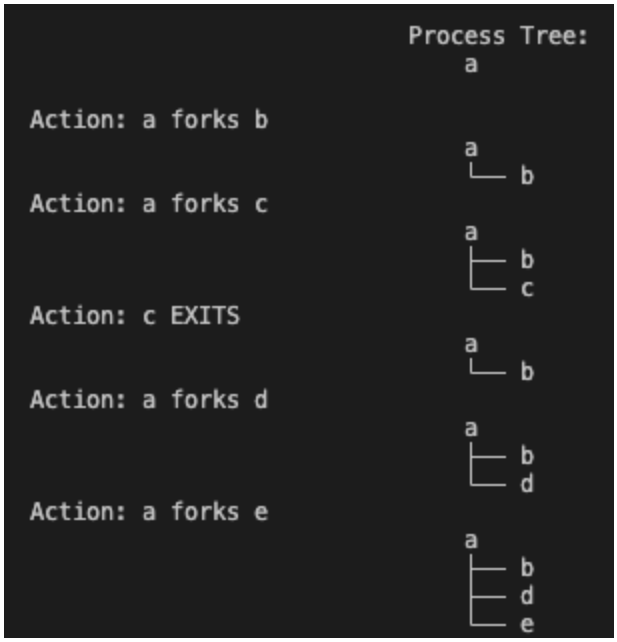
homework(simulator):

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.

```
1e18@Zuojuns-MacBook-Air cpu-api % ./fork.py -s 10

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 False

                        Process Tree:
                            a

Action: a forks b
Process Tree?
Action: a forks c
Process Tree?
Action: c EXITS
Process Tree?
Action: a forks d
Process Tree?
Action: a forks e
Process Tree?
```

Action: a forks b
Action: a forks c
Action: c EXITS
Action: a forks d
Action: a forks e

After the first action ‘a’ forks ‘b’, a is the parent of the process tree. And then ‘a’ forks ‘c’, ‘c’ exits which reduces the tree. Then ‘a’ forks ‘d’ and ‘a’ forks the ‘e’. In the end, the final tree is that ‘a’ is the parent of ‘b’, ‘d’, and ‘e’.



Try the other seed 20:

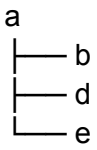
```
1e18@Zuojuns-MacBook-Air cpu-api % ./fork.py -s 20

ARG seed 20
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 False

                        Process Tree:
                            a

Action: a forks b
Process Tree?
Action: b EXITS
Process Tree?
Action: a forks c
Process Tree?
Action: c forks d
Process Tree?
Action: a forks e
Process Tree?
```

Action: a forks b
Action: b EXITS
Action: a forks c
Action: c forks d
Action: a forks e

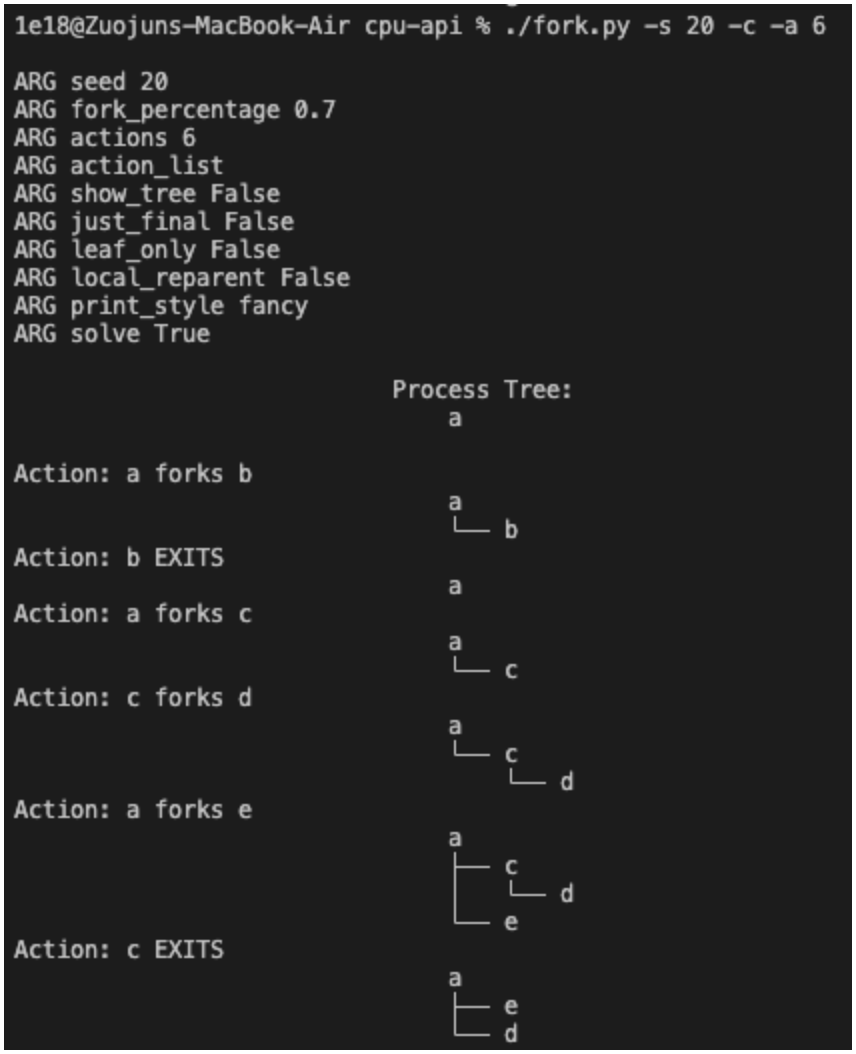
Try different actions: ./fork.py -A a+b,a+c,c+d,c-,a+e

```
1e18@Zuojuns-MacBook-Air cpu-api % ./fork.py -A a+b,a+c,c+d,c-,a+e

ARG seed -1
ARG fork_percentage 0.7
ARG actions 5
ARG action_list a+b,a+c,c+d,c-,a+e
ARG show_tree False
ARG just_final False
ARG leaf_only False
ARG local_reparent False
ARG print_style fancy
ARG solve False

                        Process Tree:
                            a

Action: a forks b
Process Tree?
Action: a forks c
Process Tree?
Action: c forks d
Process Tree?
Action: c EXITS
Process Tree?
Action: a forks e
Process Tree?
```

Action: a forks b
Action: a forks c
Action: c forks d
Action: c EXITS
Action: a forks e
Final Process Tree:



Try -a 6, which means there are 6 actions:

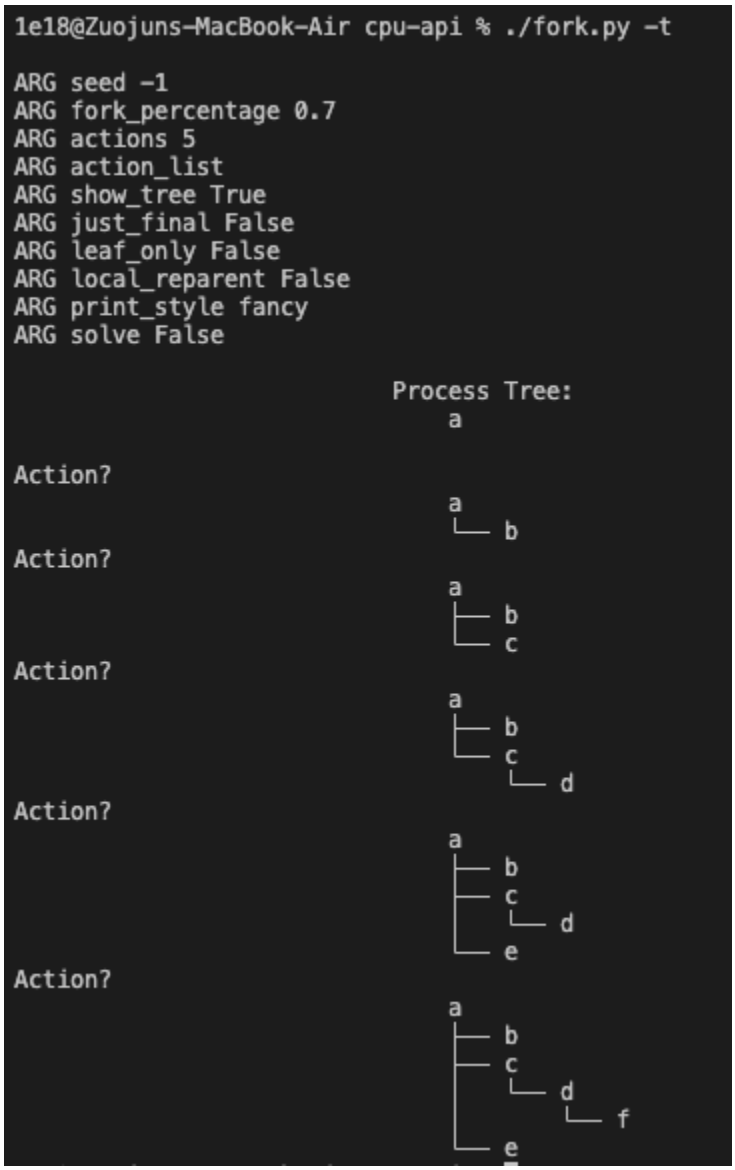


Action: a forks b
Action: b EXITS
Action: a forks c
Action: c forks d
Action: a forks e
Action: c EXITS

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.

I think the percentage of EXIT actions would increase to nearly 100% as the fork percentage decreases to 0.1, while the percentage of EXIT actions would decrease to amost 0% as the fork percentage increases to 0.9.

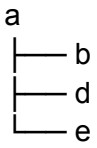
3. Now, switch the output by using the -t flag (e.g., run ./fork.py -t). Given a set of process trees, can you tell which actions were taken?



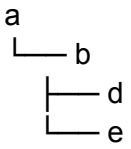
Action: a forks b
Action: a forks c
Action: c forks d
Action: a forks e
Action: d forks f

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.

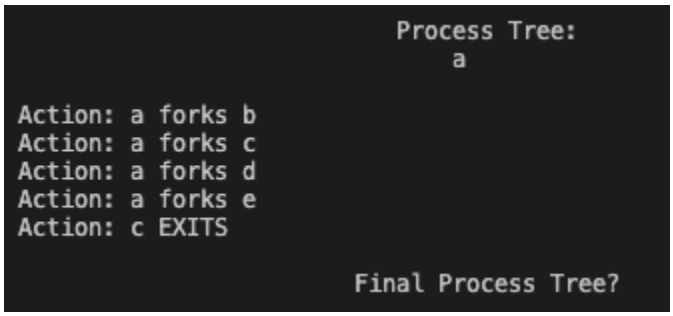
‘d’ and ‘e’ would change their parent from ‘c’ to the parent of the tree ‘a’.
Final Process Tree:



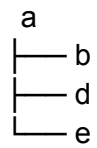
If we add flag -R which means turn the local_reparent on, ‘d’ and ‘e’ would change their parent from ‘c’ to the local parent of them which is ‘c’ ‘s parent ‘b’.
Final Process Tree:

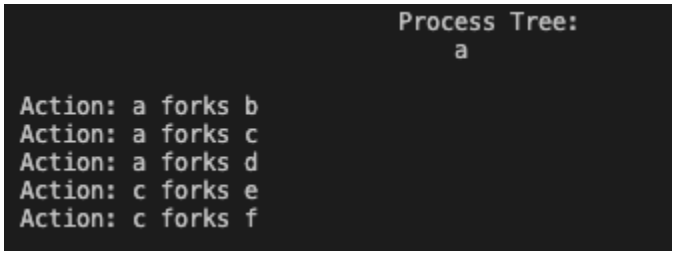


5. One last flag to explore is the -F flag, which skips intermediate steps and only asks to fill in the final process tree. Run ./fork.py -F and see if you can write down the final tree by looking at the series of actions generated. Use different random seeds to try this a few times.

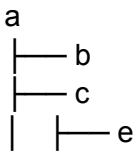


Final Process Tree:



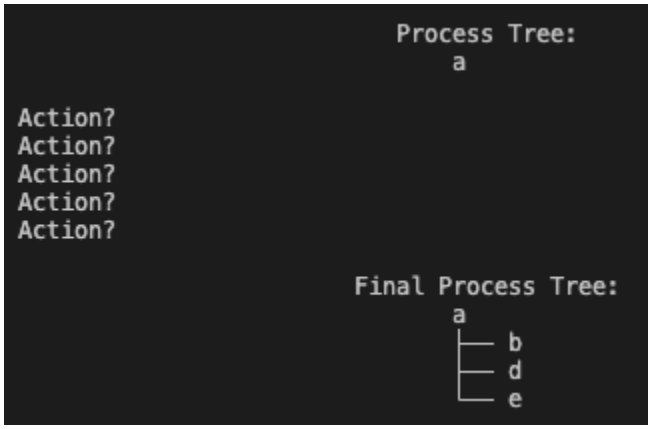


Final Process Tree:

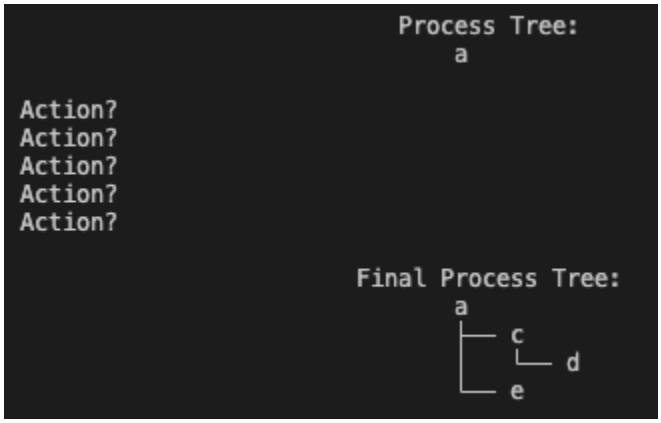


```
|   └── f
└── d
```

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.



For this one, I can't tell. Because it can be:
Action: a forks b
Action: a forks c
Action: c EXITS
Action: a forks d
Action: a forks e
Or it can be:
Action: a forks b
Action: b forks c
Action: a forks d
Action: c forks e
Action: c EXITS
Or other cases. It's not local-reparent case, thus there are multiple situations.



Action: a forks b
Action: b EXITS
Action: a forks c
Action: c forks d
Action: a forks e

Homework(Code)

1

The original value of the variable in the child process is 100.
When both the child and parent process change the value of x, it will not affect the value of the variable in the other process.

```
I am parent process (pid:31980), x = 100
I am parent process (pid:31980), x = 20
I am child process (pid:31981), x = 100
I am child process (pid:31981), x = 10
```

2

Both the parent process and the child process can access the file descriptor.
When they are writing to the file concurrently, the order is not guaranteed.

3

With vfork(), we cant ensure that the child process always prints first.

4

Because in different situations, we need different parameters. Then we need all kinds of system calls of exec() to appropriate the certain situations.

5

Wait returns id of the child process. If use wait() in the child process, it will return -1 because it does not have a child process itself.

```
child pid 14754 wc -1
parent pid 14753 wc 14754
```

6

waitpid() is useful when there are several child processes and we only care about one.

7

It will print nothing. STDOUT_FILENO, possessing an int type, is defined at unistd.h. It's a file descriptor of LINUX system. When we close() it which means redirecting standard output to a file, there is no printing anymore.

8

The first child process writes a message to the pipe and the second child process reads a message from the pipe.

Ch6

homework(simulator):

Measure system call time
I run getPid() a million times and calculate its average time.
The average time for getPid system call is around 100 ns.
Measure context switch time
I have two processes running on one CPU. Process 1 writes to pipe 1 and then gets blocked.
Process 2 reads from pipeline 1 and writes to pipeline 2, then gets blocked, and so on.
The average time for context switch is around 400 ns.

Ch7

homework(simulator):

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

```
1e18@Zuojuns-MacBook-Air cpu-sched % ./scheduler.py -p FIFO -j 3 -l 200,200,200
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 )
```

FIFO scheduler:
Job0: Response time = 0, Turnaround time = 200.
Job1: Response time = 200, Turnaround time = 400.
Job2: Response time = 400, Turnaround time = 600.
Average response time = 200;
Average turnaround time = 400.

```
1e18@Zuojuns-MacBook-Air cpu-sched % ./scheduler.py -p SJF -j 3 -l 200,200,200
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 )
```

SJF scheduler:
Job0: Response time = 0, Turnaround time = 200.
Job1: Response time = 200, Turnaround time = 400.
Job2: Response time = 400, Turnaround time = 600.
Average response time = 200;
Average turnaround time = 400.

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

```
1e18@Zuojuns-MacBook-Air cpu-sched % ./scheduler.py -p FIFO -j 3 -l 100,200,300
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 )
```

FIFO scheduler:

Job0: Response time = 0, Turnaround time = 100.
Job1: Response time = 100, Turnaround time = 300.
Job2: Response time = 300, Turnaround time = 600.
Average response time = 133.33;
Average turnaround time = 333.33.

```
1e18@Zuojuns-MacBook-Air cpu-sched % ./scheduler.py -p SJF -j 3 -l 100,200,300
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 )
```

SJF scheduler:
Job0: Response time = 0, Turnaround time = 100.
Job1: Response time = 100, Turnaround time = 300.
Job2: Response time = 300, Turnaround time = 600.
Average response time = 133.33;
Average turnaround time = 333.33.

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

```
1e18@Zuojuns-MacBook-Air cpu-sched % ./scheduler.py -p RR -j 3 -l 100,200,300 -q 1
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 )
```

Time slice = 1 means that one job would switch to another after it runs for 1 millisecond.
As a result,
RR scheduler:
Job0: Response time = 0, Turnaround time = 100 + 99 + 99 = 298.
Job1: Response time = 1, Turnaround time = 298 + 1 + 99 + 1 = 499.
Job2: Response time = 2, Turnaround time = 600.
Average response time = 1;
Average turnaround time = 465.67

4. For what types of workloads does SJF deliver the same turnaround times as FIFO?
When the order of jobs in the job list is in the order from smallest length to largest length. For example, the workloads are 100, 200, 300.

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

When the workload is same with the quantum length and all of the jobs have the same workload. For example, the workloads are 200, 200, 200 and the quantum length is 200.

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

The response time would increase as job lengths increase.

```
1e18@Zuojuns-MacBook-Air cpu-sched % ./scheduler.py -p SJF -j 3 -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
```

```
1e18@Zuojuns-MacBook-Air cpu-sched % ./scheduler.py -p SJF -j 3 -l 200,300,400 -c
ARG policy SJF
ARG jlist 200,300,400

Here is the job list, with the run time of each job:
  Job 0 ( length = 200.0 )
  Job 1 ( length = 300.0 )
  Job 2 ( length = 400.0 )


** Solutions **

Execution trace:
  [ time   0 ] Run job 0 for 200.00 secs ( DONE at 200.00 )
  [ time 200 ] Run job 1 for 300.00 secs ( DONE at 500.00 )
  [ time 500 ] Run job 2 for 400.00 secs ( DONE at 900.00 )

Final statistics:
  Job   0 -- Response: 0.00  Turnaround 200.00  Wait 0.00
  Job   1 -- Response: 200.00  Turnaround 500.00  Wait 200.00
  Job   2 -- Response: 500.00  Turnaround 900.00  Wait 500.00

  Average -- Response: 233.33  Turnaround 533.33  Wait 233.33
```

We can see the average response time increased by 100 as the job length increased by 100 for each job.

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?

The average response time is always equal to the quantum length.
If Q = quantum length;
When q is always less than any turnaround time of jobs, the response time = Σ i * q/ N. (0 <= i < N).

Job i	Response time
0	0
1	q
2	q + q
…	…
N - 2	(N - 2) * q
N - 1	(N - 1) * q

The total response time should be
q + 2q + 3q + … + (N-2) * q + (N-1) * q
= q(1 + 2 + 3 + … + (N-1))
= q * (N - 1) * N/2
The average response time
= The total response time / N
= q * (N - 1) /2 (0 <= i < N)

When q is larger or equal than any turnaround time of jobs, it turns to the same question as FIFO. The worst case is that we have a decreasing order of workload for jobs, the response time = Σ response time of the first(N-1) jobs/ N.

Job i	Workload length	Response time
0	L_0	0
1	L_1	L_0
2	L_2	$L_0 + L_1$
…	…	…

i	L_i	$\sum_{j=0}^{j=i-1} L_j$
…	…	…
N - 1	L_{N-1}	$\sum_{i=0}^{i=N-2} L_i$