Benjamin Pierce (bgp12)
EECS 338
HW 3 Report
Due: 2/6/19

1) simple-shell.c
Sample output:

```
bp0017@bp0017-XPS-15-9550:~/Documents/EECS338/HW3$ ./simple-shell
sh>ls -lrt
total 340
-rw-rw-r-- 1 bp0017 bp0017 233599 Jan 30 18:12 HW3.pdf
-rw-rw-r-- 1 bp0017 bp0017   1438 Jan 30 23:54 execvp_demo.c
-rwxrwxr-x 1 bp0017 bp0017   8664 Feb  4 19:24 demo
-rw-rw-r-- 1 bp0017 bp0017    137 Feb  5 16:30 makefile
-rw-rw-r-- 1 bp0017 bp0017    509 Feb  5 16:47 part3b.c
-rw-rw-r-- 1 bp0017 bp0017    488 Feb  5 16:52 part3a.c
-rw-rw-r-- 1 bp0017 bp0017   1100 Feb  5 17:12 part2a.c
-rw-rw-r-- 1 bp0017 bp0017   1148 Feb  5 17:16 simple-shell.c
-rwxrwxr-x 1 bp0017 bp0017   8744 Feb  5 17:42 simple-shell
-rwxrwxr-x 1 bp0017 bp0017   8568 Feb  5 17:42 part2a
-rwxrwxr-x 1 bp0017 bp0017   8576 Feb  5 17:42 part3a
-rwxrwxr-x 1 bp0017 bp0017   8568 Feb  5 17:42 part2b
-rwxrwxr-x 1 bp0017 bp0017   8576 Feb  5 17:42 part3b
-rw-rw-r-- 1 bp0017 bp0017    590 Feb  5 17:42 part2b.c
-rw-rw-r-- 1 bp0017 bp0017   7388 Feb  5 17:43 bgp12_eecs338_HW3.odt
sh>make
gcc -o simple-shell simple-shell.c
gcc -o part2a part2a.c
gcc -o part3a part3a.c
gcc -o part2b part2b.c
gcc -o part3b part3b.c
sh>exit
bp0017@bp0017-XPS-15-9550:~/Documents/EECS338/HW3$
```

2)
a) part2a.c
Output

```
Process: 0 PID: 28985 PPID: 26584
Process: 1 PID: 28992 PPID: 28985
Process: 2 PID: 28993 PPID: 28985
Process: 3 PID: 28995 PPID: 28992
Process: 4 PID: 28996 PPID: 28992
Process: 5 PID: 28997 PPID: 28993
Process: 6 PID: 28999 PPID: 28993
Process PID: 28995 is done!
Process PID: 28996 is done!
Process PID: 28997 is done!
Process PID: 28999 is done!
Process PID: 28992 is done!
Process PID: 28993 is done!
```

The approach for part a is to explicitly deal with each parent and child with nested if-else statements. This essentially isolates the parents/children and allows each to do their own thing. This approach is conceptually, but kind of confusing due to the verbosity of the code.

b) part2b.c
Output
Root PID: 13730
Process PID: 13739 PPID: 13730
Process PID: 13740 PPID: 13730
Process PID: 13742 PPID: 13739
Process PID: 13743 PPID: 13740
Process PID: 13745 PPID: 13739
Process PID: 13746 PPID: 13740
Process PID: 13742 is done!
Process PID: 13743 is done!
Process PID: 13745 is done!
Process PID: 13746 is done!
Process PID: 13739 is done!
Process PID: 13740 is done!

The approach for part b is a bit different. Instead of chained ifs, we take advantage of the “binary” nature of fork(); creating processes resembles the construction of a binary tree. We also use the fact that child processes can call wait(), but it does nothing. The general methodology is that we place two if statements inside a loop, immediately after a fork(), also inside the loop. The first if statement creates another child for the parent. Then the second if statement checks if two children of the original parent exists; if so, it stops the loop. This continues until the desired depth is reached (In this case, a depth of 3).

3)
a)part3a.c
Number of processes to generate from root process 17718:
4
Process 0 PID: 17718 PPID: 26584
Process 1 PID: 17457 PPID: 17718
Process 2 PID: 17459 PPID: 17457
Process 3 PID: 17461 PPID: 17459
Process 4 PID: 17463 PPID: 17461
Process PID: 17464 is done!
Process PID: 17463 is done!
Process PID: 17461 is done!
Process PID: 17459 is done!
Process PID: 17457 is done!

b) part3b.c

```
Number of processes to generate from parent 26540: 4
Forked process 1 PID: 16067 PPID: 26540
Forked process 2 PID: 16069 PPID: 26540
Forked process 3 PID: 16070 PPID: 26540
Process PID: 16067 is done!
Forked process 4 PID: 16071 PPID: 26540
Process PID: 16069 is done!
Process PID: 16070 is done!
Process PID: 16071 is done!
Parent process 26540 is done
```