Lab 3: Process lab (25.1.16)

Goal: In this lab you will write programs using the fork() and wait() system calls.

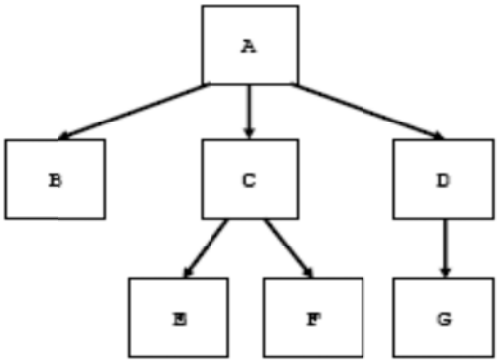
To get checked off on this lab, exercises 1 and 2 must be completed and shown to your lab TA.

- Exercise 1 is due by the end of the current lab.
- Exercise 2 is due at the beginning of the next lab.

If anything is not clear, please ask the TA for help.

Exercise 1

Consider the family tree of processes shown below (where A is the initial parent process):



- This particular family tree can be generated using exactly **SIX** `fork()` calls.
- This family tree can *also* be generated with exactly **FIVE** `fork()` calls.
- This family tree can *also* be generated with exactly **FOUR** `fork()` calls.
- This family tree can *also* be generated with exactly **THREE** `fork()` calls.

Your task is to code the four scenarios. You should explain which processes should fork a child and in what order, to create the tree in the specified number of steps. Use tables similar to that provided below to organize your thoughts. If using the table, you should put in each cell, the PID (corresponding letter) of the process created by the particular parent at that fork call. If a particular process does not create a child, place an X in the appropriate cell. For example, let's assume that only *Process A* creates *Process D* during the first fork, and then only *Process D* creates *Process G* during the second fork. The first two columns of the table should look like the example shown below.

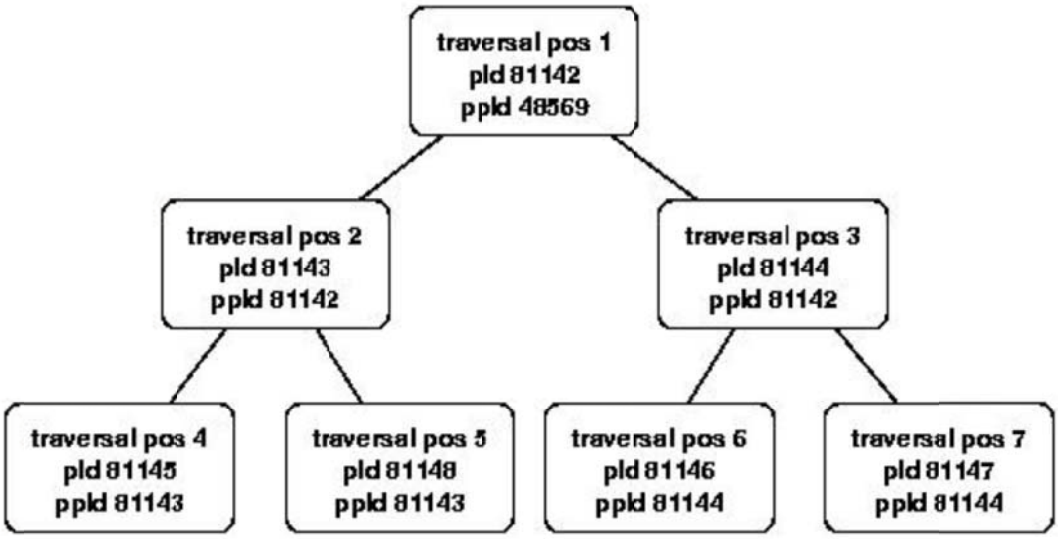
Process	Fork 1	Fork 2	Fork 3	Fork 4	Fork 5	Fork 6
A	D	X				
B	X	X				

C	X	X				
D	X	G				
E	X	X				
F	X	X				
G	X	X				

Caveats: Calling `fork()` inside of a loop counts as multiple fork calls, even though it only appears once in the code. Solutions using forks within loops may be correct, but you need to be very careful in your approach.

Exercise 2

Write a C program that creates a "binary tree" of Unix processes. Call your program proctree.c. Your program should take a single command-line parameter which specifies the number of levels in the binary process tree. Each process should be assigned a number corresponding to its position in a level-order traversal of the tree. Given a height of 3, the tree can be thought of as this binary tree, where the parent-child links are not explicitly stored by your program but are part of the Unix process hierarchy. The tree should look something like this:



You won't draw a graphical representation, but your program's processes should print out the information about the tree, as follows:

```
-> ./proctree 3
[1] pid 81142, ppid 48569
[1] pid 81142 created left child with pid 81143
[1] pid 81142 created right child with pid 81144
 [2] pid 81143, ppid 81142
 [3] pid 81144, ppid 81142
 [2] pid 81143 created left child with pid 81145
 [3] pid 81144 created left child with pid 81146
```

```
 [3] pid 81144 created right child with pid 81147
  [4] pid 81145, ppid 81143
 [2] pid 81143 created right child with pid 81148
  [6] pid 81146, ppid 81144
  [7] pid 81147, ppid 81144
  [5] pid 81148, ppid 81143
 [3] right child 81147 of 81144 exited with status 7
 [3] left child 81146 of 81144 exited with status 6
 [2] right child 81148 of 81143 exited with status 5
 [2] left child 81145 of 81143 exited with status 4
[1] right child 81144 of 81142 exited with status 3
[1] left child 81143 of 81142 exited with status 2
```

Note that each line of output is indented according to the depth of the node in the process tree and begins by printing the traversal position of the process that prints it. Your program's processes should produce output in the following situations:

- When each process is created, it should print its traversal position, its pid (process ID, obtained using `getpid()`) and ppid (parent process ID, obtained using `getppid()`).
- After a process spawns a child process, it should print its own (not the new child's) traversal position, its own pid, and the pid of the newly-spawned child along with an indication of whether this child forms its "left" or "right" subtree.
- When a child exits (using `exit()`), it should provide its traversal position as its exit status. This value should be obtained by the parent when it calls `waitpid()` and printed along with the parent's traversal position, whether the terminated child is a left or right subtree, the parent's pid, the terminated child's pid and the exit status, which should be the child's traversal position.

This program, as you are developing it, has a good chance of becoming a "fork bomb," a program that keep spawning new processes which can render a Linux system nearly useless. To reduce the chances that this happens, you should check the return value of your `fork()` calls and stop if it returns `-1`, which indicates that you were unable to spawn a process. You should also limit your trees to small heights when debugging. Feel free to try larger tree sizes once you're confident that your program is working to see how large a tree you can get before you run out of processes.