Computer Science 330
Operating Systems
Siena College
Fall 2020

# Programming Project 1: A Process Tree

**Due: 11:00 AM, Monday, September 14, 2020**

In this programming project you will write a C program that creates a "binary tree" of Unix processes, similar to the class example that does the same using POSIX threads.

You may work alone or in a group of size 1 or 2 on this programming project.

There is not a lot of code to write here (my solution is 115 lines, including comments and blank lines). However, since your experience programming in C, programming in Unix environments, and programming with multiple processes is very limited, it will take some time and you'll have questions. So it's definitely not something that you can do right before the due date.

## Getting Set Up

You will receive an email with the link to follow to set up your GitHub repository, which will be named `proctree-proj-yourgitname`, for this programming project. Only one member of the group should follow the link to set up the repository on GitHub, then others should request a link to be granted write access.

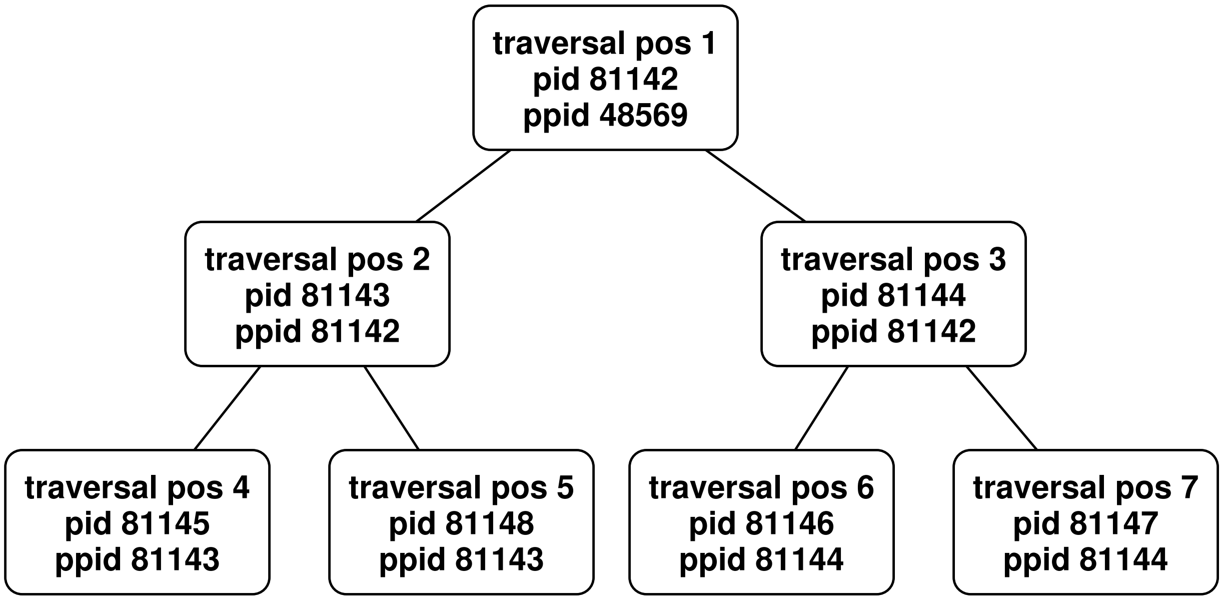
All GitHub repositories must be created with all group members having write access and all group member names specified in the `README.md` file by 11:59 PM, Monday, September 7, 2020. This applies to those who choose to work alone as well!

## Requirements

Call your program `proctree.c` and include a `Makefile` to compile this program into an executable named `proctree`.

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(2)`) and ppid (parent process ID, obtained using `getppid(2)`).
- 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(3)`), it should provide its traversal position as its exit status. This value should be obtained by the parent when it calls `waitpid(2)` 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.

To be able to see what's happening and to reduce the chances that the output of your processes will be interleaved, you should put in calls to `sleep(3)`.

You need not use shared memory for this program. In fact, it will probably confuse things if you try.

This program, as you are developing it, has a good chance of becoming a "fork bomb". 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. Try it at least on a virtual Linux instace under VirtualBox, and on a Mac system (you can use your own or a College Mac). Include these results in the comment at the top of your program. Please be careful if running on our shared system noreaster. It has a lot of responsibilities so please don't crash it!

This program will be graded as a Programming Assignment, according to the guidelines on the course web site.

## Submission

Commit and push!

## Grading

This assignment will be graded out of 25 points.

| Feature | Value | Score |
|---|---|---|
| Correct number of levels created | 4 | |
| Correct total number of processes created | 2 | |
| Correct usage of `fork` system call, including error checks | 2 | |
| Correct usage of `waitpid` system call | 2 | |
| Correct printing of child exit status in parent | 1 | |
| Sleep between successive levels | 1 | |
| Traversal order labels | 3 | |
| Indentation based on tree level | 2 | |
| Largest tree size reported in comment | 1 | |
| Documentation | 4 | |
| Design and style | 2 | |
| `Makefile` | 1 | |
| Total | 25 | |