Lab 1 – Unix Processes

Due: 9:55 AM, Thursday, February 17, 2005

This week's lab consists of questions to be done on your own (not turned in), questions to be answered and turned in, and a number of programming task. The programming for this lab are considered "laboratory programs" for honor code purposes. You may discuss them with each other, but what you turn in must be your own work.

Practice Questions

You do not need to turn in answers to these questions. Note that the textbook's web page has possible answers to many of the exercises at the end of the chapters.

SG&G 1.10, 1.12, 2.2, 2.9, 3.2, 3.4

Homework Questions

Turn in short answers to these questions. Please place your answers in a file called `lab1.txt`.

SG&G 1.13, 2.14

Practice Using Your `fork()`

Write the program for SG&G question 3.6. Call your program `forkfib.c`. Include a `Makefile` that compiles this program into an executable called `forkfib`.

Notes:

- Do some error checking. If your program is run without a command-line argument, print a "Usage" line. If your program is given a negative length for the Fibonacci sequenace, print an appropriate error message.
- If you store the entire sequence in an array as you generate it, allocate the array of an appropriate size using `malloc(3)`[1]. C has no garbage collection, so any memory you allocate with `malloc()` must be returned to the system with `free()`.
- If you store the values in the sequence with `int` values, you will notice they may overflow the storage capabilities of the data type. You can delay this somewhat by using `long` values. Even then, you will overflow the values with a relatively short sequence. In this case, also print an error message.

[1] This notation means that you can find information about malloc in section 3 of the Unix manual, `man`. Try "man malloc" to see how to use it.

Adding POSIX Shared Memory

Write the program for SG&G question 3.10. Call your program `forkfibshared.c`. Add a rule to your `Makefile` to compile this program into `forkfibshared`.

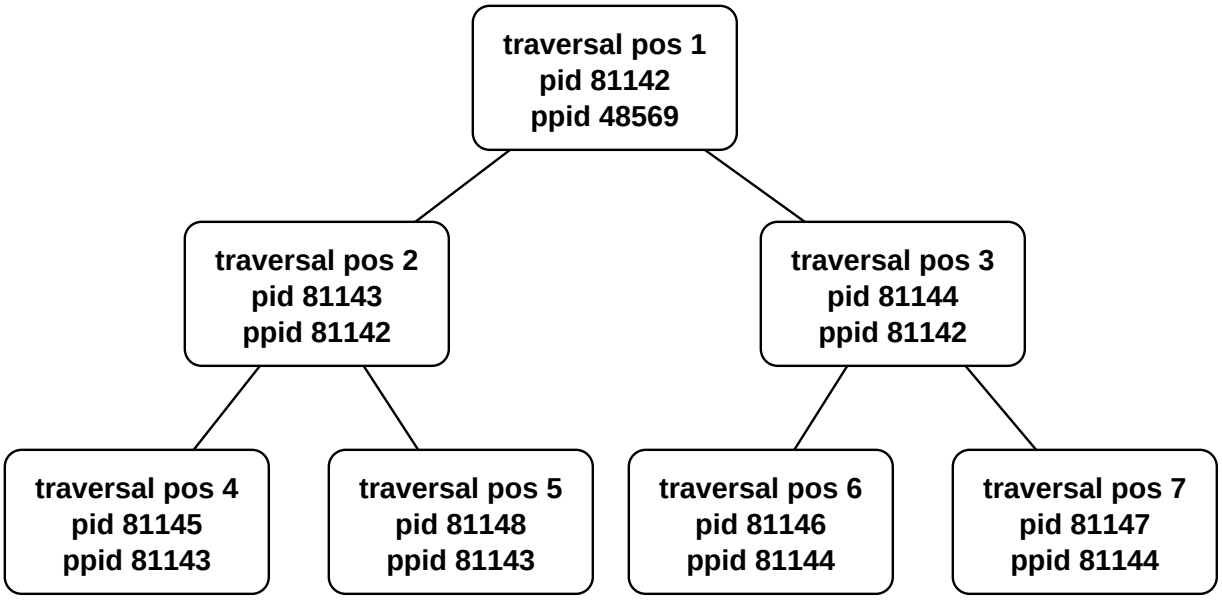
Once your program is working, add a call to `sleep(2)`[2] to your program before you detach and free the shared memory segment. In a separate window, use the `ipcs` command to see that your shared memory segment is listed, and that it goes away when your program terminates.

A Process Tree

Write a C program that creates a "binary tree" of Unix processes. Call your program `proctree.c` and add a rule to your `Makefile` to compile this program into `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
```

[2] Again, the 2 in `sleep(2)` refers to the manual section, not the argument to pass

```
[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," a program that keep spawning new processes which can render a Unix 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. Try it at least on a lab FreeBSD machine, a Solaris server (*bullpen*), and a Linux system (*miltank*) in the Unix lab. You can use ssh to connect to bullpen and miltank. If you have a chance, go down to one of the Mac labs in TCL 216 or TCL 217a and try it on one of the Macs there.

Submission and Evaluation Using the `turnin` utility on a lab FreeBSD system, submit a single tar file called `lab1.tar` that contains all the files you wish to submit. Please include your C source files and Makefile, but not your compiled executables. Make sure your name is in each file.

This lab will be graded out of 25 points. The homework questions are worth 2 points each, `forkfib` is worth 6 points, `forkfibshared` is worth 5 points, and `proctree` is worth 10 points. Programs will be evaulated based on correctness, documentation (have a comment describing the entire program and comments pointing out key parts of the code), and efficiency.