CS 350 Lab 2: Simple Process Creation and Control

Spring 2016

Assigned: February 4, 2016
Due: Always consult Blackboard for due dates and late penalties

Overview and Objectives

In Lab 2, you will investigate Linux facilities for process creation, management, and control, both from within C programs and from the command shell. You will build toward writing a program that creates an N-level process tree. Each process in the tree, other than leaf processes, will have M children. Tree parameters N and M will be specified by the user.

Interface, Options, and Parameters

Your program should support the following interface:

```
lab2 [-u] [-N <num-levels>] [-M <num-children>] [-p] [-s <sleep-time]
```

`-u`

Print a usage string for your program on `stderr` and then exit. If `-u` appears anywhere on the command line, the program may ignore all other arguments.

`-N <num-levels>`

Create a process tree with `<num-levels>` levels. You may assume that the user will not enter a negative number. Specifying 0 or 1 for `<num-levels>` should cause your program not to create any child processes (and therefore no additional levels). Your program should not accept values larger than a maximum of 4. The default value for `<num-levels>` is 0.

`-M <num-children>`

Create a process tree wherein every process (other than leaf processes) creates `<num-children>` child processes. You may assume that the user will not enter a negative number or 0. Your program should not accept values larger than 3. The default value for `<num-children>` is 1.

`-p`

Leaf processes should pause by calling `pause()`, immediately upon realizing that they are leaf processes. When this option is specified, the user will clean up processes from the terminal or shell.

`-s <sleep-time>`

Leaf processes should sleep for `<sleep-time>` seconds, immediately upon realizing that they are leaf processes. Do this by calling `sleep()`. Non-leaf processes should not sleep; instead they should wait for child processes to complete, by calling `wait()`.

If the user specifies neither `-p` nor `-s`, then `-s` is assumed (i.e. the leaf processes should call `sleep()`, not `pause()`), and the default value for `<sleep-time>` is 1.

If the user specifies both `-p` and `-s` then notify the user with an error message, print the usage string, and exit.

If the user specifies an option that your program does not support (`-z` for example), your program should print the usage string to standard error (`stderr`), and exit. If the user does not specify an argument where one is expected, the program should likewise report the problem, print the usage string to `stderr`, and then exit.

Output

Before creating any children and before sleeping or pausing, each process in the tree (including the top-level process) should print to `stdout` its level, its own Linux process id, and the process id of its parent, in the following format:

```
ALIVE: Level 2 process with pid=3452, child of ppid=3450.
```

Immediately before exiting, each process should print to standard output a message in the following format:

```
EXITING: Level 2 process with pid=3452, child of ppid=3450.
```

Levels of the tree are numbered from the top down. In other words, the lone root process at the top of a 3-way tree with 3 levels is said to be at Level 2. The 3 children of that root process are said to be at Level 1, and the 9 leaf processes beneath the Level 1 nodes are said to be at Level 0. This is the level numbering that should be reflected in your output statements. *Note that this is backwards from the usual level numbering; we usually say that the root node resides at level 0 of the tree.* I have done this intentionally to make things slightly more straightforward for you (I hope).

Additional Requirements

Important characteristics for your program:

- The user should be able to run your program such that all processes are running concurrently (by choosing to have leaf processes pause, or by having them sleep for long enough for the entire tree to be created).
- All processes must use the same executable, namely `lab2`. You may not have a different executable for different levels of the tree, including leaf processes.
- Every child process must call one of the `exec()` functions. That is, you may not simply bury the `sleep()` or `pause()` inside the loop structure of your program, after returning as a child from `fork()`. That leaf child should instead execute the `lab2` executable.

Initial Steps – Building Toward a Process Tree Program

Some Useful Linux Commands

Begin by running your lab1 program through `strace`, a program that intercepts and reports all system calls that the program makes.

```
$ strace lab1 –g
```

Some of the many system calls that `strace` reports are made by library (e.g. `libc`) functions that you call directly from within your program. Others do not correspond as directly to components of your program. As you use `strace` to investigate the behavior of your programs, you will learn to identify the calls that correspond directly to your code. Keep `strace` in mind, and as needed, please use it to trace your programs throughout the semester (and beyond).

Next, run the `ps` command to list processes.

```
$ ps
```

(If commands do not work exactly as specified in the lab assignment, it could be that your version of a command is "aliased" to some other command. Type "`which ps`" to find the full path to the ps program that your shell will use. You can always specify the full path name to programs; there is a `ps` in `/bin`, so you may use `/bin/ps` if necessary. In general, if something works for a neighbor, but not for you, figure out which program you are *really* running, with `which`.)

Peruse the man page for ps to investigate ps as a utility for listing processes. Also try

```
$ /bin/ps --help all
```

Notice that the operating system maintains lots of information about relationships between processes, much of which you can get at using `ps`.

Try running top, which also shows running processes.

```
$ top
```

Typing 'q' gets you back to a shell prompt.

Write a program that prints a message, calls `sleep()` or `pause()` (try both), then prints another message.

Run your program and practice "suspending" the program and "resuming" the program with Ctrl-Z and with the linux command `fg` (for "foreground"), respectively.

While the program is sleeping, paused, or suspended, "look at it" using `ps`. You may also use `jobs`.

Now check out the "`kill`" command, for killing processes. The `kill` command can take a `pid` as a parameter, and the pid of any running process can be discovered using `ps`.

Create a paused process, figure out its `pid`, then kill it and verify that it is gone.

The `pid` of a process can also be discovered from within a C program, which can also learn the `pid` of its parent process. Investigate the functions `getpid()` and `getppid()`, and include calls to them inside your simple sleep/pause program. Verify that the value that your program reports matches the one that the Linux command utilities show.

Writing Programs that Create Processes

Enter the following program into a file called `simplefork.c`.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main() {
   pid_t pid; /* for a child process */

   pid = fork();
   if (pid < 0) { /* error */
      fprintf(stderr, "Fork failed.");
      return 1;
   }

   if (pid == 00) { /* child process */
      execlp("/bin/ls", "ls", NULL);
   }
   else { /* parent process */
      /* parent will wait for the child to complete */
      wait(NULL);
      printf("Child Complete.");
   }
   return 0;
}
```

Compile and run it.

From this point forward, keep a copy of each working program after every step (just in case).

Alter your program to `exec()` your pause/sleep program that reports the pid and parent pid.

Alter the program to create 4 child processes instead of one.

Alter the program to have the first child create another child. Use two different executables for the two children.

Here's where it gets dangerous....

Use the same executable for the child process and the "grandchild" process. THINK first. Program carefully. Do NOT create a "fork bomb"!

Use a command line argument, passed through to the child via `exec()`, to distinguish the first child from the grand child. Do not do any forking in your code until you are sure that parameter is passed through properly.

Using this code as a building block, you are now ready to tackle the problem of creating an M-way process tree with N levels.

Submission

Tar, gzip, and submit your code to Blackboard exactly as specified in Lab 1 (with the lab number changed where appropriate, of course). Use a makefile that supports "make clean", remove all executable and object code before you tar and gzip, name your executable appropriately, etc.