

Lab 5

23 Apr 2009

In today's lab, you will write a very rudimentary shell.

1 First, a couple of commands to know about

Type

```
ps
```

This lists all your currently-running processes on this machine. If you instead type

```
ps -aj
```

you'll see processes of other users as well, along with the name of the user and some other process info. (You may need to resize the window to see the whole thing.) Expand that to

```
ps -ajx
```

and you'll see even more processes, including all the system processes deep under the hood and some additional user processes running as daemons (i.e. not attached to a terminal).

Type

```
top
```

and you'll see all the processes in a slightly different format, and updated in realtime (every second by default).

For the remainder of this lab, you should keep one window set aside just for running `ps` and/or `top`, so you can track what's going on with your programs.

One last thing: type

```
cat
```

and hit enter. All this does is read from the keyboard, line-by-line, and echoes it back to the terminal every time you hit enter. Hit `^D` on a line by itself to terminate. This program will be useful to us for testing purposes, when we need a process that will keep running until we tell it to stop.

2 A proto-shell

The first step to writing a shell is writing a program that reads a command from the keyboard, and then executes it without ever returning control to the proto-shell. The key new mechanism here is the system call `execvp`, one of a family of calls that run a program by replacing the program currently running with a different one.

For this task, and for the rest of the lab, we'll assume we're only doing commands with no options or arguments—that will let you hard-code a few things for now.

So your first task is to set up a C program that prints a brief prompt (like “-> ”) and reads in the next line the user types, which should be a single executable file with no spaces in it. (This is straightforward, but test this much and make sure it's working before proceeding.)

Then, pass this to `execvp`. The `execvp` function takes two arguments. The first is a string, the command to be executed. The second is an array of strings, which will be the `argv` for that command's `main` function—BUT NOTE that like strings, and unlike most other arrays we've been using, this array must be null-terminated. Since the `argv` will have one value (the command name), that means that the array you give here should have two elements: the value, and `NULL`.

Test that this works by running your program and typing “`ls`” and hitting enter. The command should run and then return you to your `tcsh` prompt, having completed.

The last piece to write for this section is code to check for bad input. If someone types a word that is not a valid command, `execvp` returns `-1` and sets `errno`, so you can print an error message with `perror` or whatever if that happens, and exit with the failure code.

A couple important things to note at this point: First, anything you put

after the `execvp` will not run unless the `execvp` fails. Feel free to add a `printf` there to prove this to yourself. Second, the command that is run by `execvp` slides right into the process table where the old process used to be. Run your `a.out`, and then in another window use `top` to identify its process id. Back in your proto-shell, type `cat`, and notice that the `top` listing now has `cat` exactly where the `a.out` used to be; rather than running a program in addition to the current one, `execvp` runs a program *in place of* the current one.

But a shell that exits after running a single command wouldn't be very useful. We'll fix that now, but first, save a copy of what you've got so far, for your own future reference.

3 Regaining control

If `execvp` takes over the process it's run in, and we want our program to continue in its existing process, we need to create a new process in which to run `execvp`. This is accomplished with the system call `fork`.

After you have read the user's input, but before your call to `execvp`, print a debugging statement to `stderr` that gives the current process id (which you can get with `getpid()`) and the command you're about to execute; and then make a call to `fork()` and store the return value.

The call to `fork` clones the process, so that you now have the pre-existing process, which we call the parent, and a near-perfect copy of it, which we call the child. *Both of them* think they have just made a call to `fork` (since the child was cloned after the parent called `fork`). The only difference between them is that the child process gets a return value of 0, but the parent process gets a return value that is the process id of the newly-created child process.

So, the next step after a `fork`, always, is to put an `if` to test whether the return value was zero or not. In the child branch, where the return value was zero, you should print another debug message to `stderr` with the current process id and the command you will momentarily execute. Then, the call to `execvp` and the test to see if it errors. In the other branch, the parent branch, where the return value was not zero, for now just print a debug message to that effect that includes the return value; and then let the program terminate. (We'll add the loop later.)

Compile and run this. Try to run `cat`. Use the `top` or `ps` output, along with

the debugging statements you added, to make clear what you're seeing.

One last loose end is that the parent process prints its message right away, and then exits successfully, while the child process is still doing its thing. That's not what we want; we want the parent to wait until the child is done before it continues. This is accomplished with the `wait` system call: it takes the address of an `int` (where it will place some status information), waits for the child process to complete, and then returns the id of the child that completed. So, in the parent branch, prior to the print statement, you should declare an

```
int status = 0;
```

and then pass its address to `wait`. Then, print out the status code in the debug statement.

Compile and run this again. Now, the parent's final message should not appear until after the command completed execution, even if it is something like `cat` that takes a while.

4 Your first true shell, and an Important Notice

At this point, all the heavy lifting is done, and all that is left to make a true (if simple and limited) shell is to loop all of this until the user hits `^D` (which is seen as an end-of-file) or types "`exit`"; the obvious next step is to accept options and arguments to pass along to the command.

Those things will be part of the project you'll be working on for the next couple of weeks, but for now I'd rather you focus on your exam, so at this point just hand in the lab work and you can switch to working on the exam (or leave), after you read one very important last note:

IF you decide to play around with the shell, particularly anything at all to do with `fork`, YOU MUST, you ABSOLUTELY MUST, work on that on a machine in this lab. It's fine if you work remotely, by `ssh`ing from `euclid` to here, but it is vitally important that you not be running any of your own `fork` code on `euclid`, as it is extremely easy to accidentally write code with 'tiny' bugs that will turn out to crash the entire system—and a lot of other people rely on `euclid`. Ever wonder why the room you're sitting in is called the "crash and burn" lab? This is it.