CSC209: Lab 6: fork(), pipe(), exec(), dup2()
Due: Monday, July 9 at 11:30PM

Introduction

The purpose of this exercise is to play with fork(), pipe(), exec(), and dup2() and get a feeling for how it works. It should be a fairly long lab.

Run the simplefork program

Open `simplefork.c` in your favourite editor. Read it through to figure out what it is doing. Compile and run it a few times. Recall from lecture that it is up to the operating system to decide whether the parent or the child runs first after the fork call, and it may change from run to run.

Question 1: Which lines of output are printed more than once?

Question 2: Write down all the different possible orders for the output. Note that this includes output orders that you may not be able to reproduce.

Fork in a loop

The program in `forkloop.c` takes one command-line argument, which is the number of iterations of the loop that calls `fork`.

Try running the program first with 1, 2, or 3 iterations.

Notice that the shell prompt sometimes appears in the middle of the output. That happens because the parent process exits before some of the children get a chance to print their output.

Also notice that some of the parent process ids are 1. This happens when the parent process terminates before the child calls `getppid`. (What do we call the child and the parent process when it is in this state?) If you want avoid this situation you can add a `sleep(1);` just before the return call in main. Note that this is really just a hack and if we really want to ensure that a parent does not terminate before its child, we need to use `wait` correctly.

Question 3: How many processes are created, including the original parent, when `forkloop` is called with 2, 3, and 4 as arguments? *n* arguments?

Question 4: If we run `forkloop 1`, two processes are created, the original parent process and its child. Assuming the process id of the parent is 414 and the process id of the child is 416, we can represent the relationship between these processes using the following ASCII diagram:

```
414 -> 416
```

Use a similar ASCII diagram to show the processes created and their relationships when you run `forkloop 3`.

Make the parent create all the new processes

Create a copy of `forkloop.c` called `parentcreates.c`.

In the new file, modify the program so that the new children do not create additional processes, i.e., so that only the original parent calls `fork`.

Keep the `printf` call for all processes.

The resulting diagram will look something like the following when `parentcreates 3` is run.

Note that the child process ids will not necessarily be in sequence.

```
414 -> 416
414 -> 417
414 -> 420
```

Make each child create a new process

Create a copy of `forkloop.c` called `childcreates.c`.

In the new file, modify the program so that each process creates exactly one a new process.

Keep the `printf` call for all processes.

The resulting diagram will look something like the following when `childcreates 3` is called:

```
414 -> 416 -> 417 -> 420
```

Add wait

The information provided by the `getppid` system call may not always give you the information you expect. If a process's parent has terminated, the process gets "adopted" by the `init` process, which has a process id of `1`, which is returned by `getppid`.

A process can *`wait`* for its children to terminate. If a process wants to wait until all its children have terminated, it needs to call wait once for each child. Add the appropriate `wait` calls to both `parentcreates` and `childcreates` to ensure that each parent does not terminate before its children.

Your programs should delay calling `wait` as long as possible.

In other words, if the process has other work to do like creating more children, it should create the children first and then call `wait`.

Understanding the "validate" program

In an application that requires a user to login, the application must be able to read in a user id and password, and validate it to determine whether the login is successful. One approach to validation is to hand the task off to a separate process.

You are given a program that validates user id and passwords, and will apply what you've learned about processes and pipes to create a program that runs this validation program in a child process and reports the result of the validation.

The `validate` program reads the user id and password from `stdin`, because if they were given as command-line arguments they would be visible to programs such as `ps` that inspect the state of the system. You are also given a sample file containing user ids and password combinations, which is used by `validate`.

After reading the comments at the top of `validate.c`, you will want to compile it and try running `validate` directly first.

How many bytes does `validate` expect to read in each read call? Does it require the input to be null-terminated? What happens in each case?

Notice that this program doesn't print any output; the only information it provides comes in the exit code of the program.

Use the shell variable `` `$?` `` to refer to the exit code of the last process run (e.g., by running `` `echo $?` ``).

NOTE: You may not change the validate program.

Your task is to complete `checkpasswd.c`, which reads a user id and password from `stdin`, creates a new process to run the `validate` program, sends it the user id and password, and prints a message to `stdout` reporting whether the validation is successful.

Your program should use the exit status of the `validate` program to determine which of the three following messages to print:

- "Password verified\n" if the user id and password match.
- "Invalid password\n" if the user id exists, but the password does not match.
- "No such user\n" if the user id is not recognized

The exact messages are given in the starter code as defined constants.

Note that in the given password file `pass.txt`, the "killerwhales:swim" has a userid that is too large, and "monkeys:eatcoconuts" has a password that is too long. The examples are expected to fail, but the other cases should work correctly.

You will find the following system calls useful: `` `fork` ``, `` `exec` ``, `` `pipe` ``, `` `dup2` ``, `` `write` ``, `` `wait` `` (along with `` `WIFEXITED` ``, `` `WEXITSTATUS` ``).

You may ***not*** use `` `popen` `` or `` `pclose` `` in your solution.

Important: `` `execl` `` arguments

Week 7 Video 6 "Running Different Programs" demonstrates a version of `execl` that takes only two arguments. The signature for `execl` is:

```
`int execl(const char *path, const char *arg0, ... /*, (char *)NULL */);`
```

In the video, the `execl` call only passed two arguments (`execl("./hello", NULL)`), but that shortcut doesn't work on `teach.cs`. Instead, you need to pass the middle argument (representing `argv[0]`) explicitly: `execl("./hello", "hello", NULL)`.

Let's consider two more examples. If you want to call the executable `./giant` with the arguments `fee fi fo`, you would do it like this:

```
`execl("./giant", "giant", "fee", "fi", "fo", NULL);`
```

If you want to call `./giant` with no arguments you would call it like this:
`execl("./giant", "giant", NULL);`

Submission

Submit your final `parentcreates.c`, `childcreates.c`, and `checkpasswd.c` files to `MarkUs` under the `lab6` folder in your repository. Remember to use `git pull` to first get the starter code. DO NOT submit any other files! *You do not need to submit answers to the questions.*