FIT2100 Laboraroty #3
Process Creation and Execution
Week 6 Semester 2 2024

July 31, 2024

Revision Status:

Updated by Dr. Charith Jayasekara, Aug 2024.

Contents

1 Background

This Laboraroty aims to extend your knowledge on the concepts of process creation and execution in the Unix/Linux environment.

Before you attempt any of the tasks in this Laboraroty, create a folder named `LAB03` under the FIT2100 folder (`~/Documents/FIT2100`). Save all your source files under this `LAB03` folder.

Before attending the Laboraroty, you should:

- Complete your pre-class preparations (Section 2)
- Attempt the Laboraroty tasks (Section 3)

2 Pre-class Preparation

2.1 Processes

We will start by using the `ps` command to look at the processes running in your Unix/Linux environment.

On the shell command line, type in the following command (this command displays a lot of processes, so the `more` filter is used to break up the output):

```
$ ps -e -f | more
```

Use the `man` command to find out what the `-e` and `-f` switches do to the `ps` command.

(**Note:** `ps -e -l -y` will produce a lengthy output for each of the process.)

Type in the following command (Note: in the third argument, there must be no space after each comma.):

```
$ ps -e -o pid,user,s | more
```

Use `man` to find out how the `-o` option modifies the output of the `ps` command as well as what these options are.

Process States Repeat the above ps command, this time taking note of the states of the processes. Below is a selection of different process states found under Unix (you may see more in the output):

Process State	Description
R	Running or runnable – running on a processor or in the ready queue.
S	Sleeping – blocked while waiting for an event to complete.
T	Stopped – the process has stopped executing due to receiving a signal.
Z	Zombie – the process has terminated but its parent process has not checked (waited) for it yet.

To find out what processes you are running, use the following command line:

```
$ ps -e -f | grep your_username
```

For example:

```
$ ps -e -f | grep student
```

This command causes ps to list all the processes, and then uses the grep filter to restrict the output to just those lines that contain the username `student`.

An alternative is to use the `-u` option. Only processes owned by the user `student` will be listed.

```
$ ps -u student
```

2.2 Process tree

When you execute most commands (e.g. `pwd`, `ls`, `who`, etc.), the shell spawns a new process. The shell is the *parent* process and the created process is the *child*. A collection of parents and children forms a *process tree*. We will investigate the process tree in your Unix/Linux environment using the ps command.

Execute the following command: (Note: again, there must be no space after each comma in the third argument.)

```
$ ps -e -o ppid,pid,user,s,comm > ptree.txt
```

This saves the process listing to a file named `ptree.txt`. What does the `ppid` parameter represent?

Try exploring the process tree yourself:

Open the `ptree.txt` file using a text editor. Find the entry in the file for your current command interpreter (e.g. `bash`). The following is an example of such entry (not necessarily the same values — just an example):

```
...
3806 3807 student S bash
...
```

Use the parent's Process ID to identify the name of the process that created your current command interpreter process (i.e. identify the parent of your `bash` process).

Repeat the above process until you arrive at `process ID 1` (i.e. `systemd` — on some systems, it might be called `init` instead). The process with ID 1 is the first process that runs, when the system starts up. **For your Laboraroty tasks submission, you should submit the `ptree.txt` file that you have created.**

2.3 How to create a new process

The `fork` system call is fundamental to the use and operation of the Unix/Linux operating system.

It is used when you login, to create your execution environment — the shell process; and by the shell when you execute a command (e.g. '`ls`').

Try experimenting with the `fork` system call yourself:

Download the code for `fork.c` program, then compile and run it. Experiment with the program, and try to understand what the code does. You will need this program in the Laboraroty tasks that follow.

To compile this C program and run it, use the following commands:

```
$ gcc fork.c -o fork
$ ./fork
```

(From here, you should read through the next two sections before proceeding to attempt the Laboraroty tasks in Section 3.)

2.4 Basic descriptions for the `fork` system call

The purpose of the `fork` system call is to create a new process. Typically what you want to do is to start running some program.

For example, you are implementing a text editor and want to provide a mechanism for spell checking. You could just implement a function, which provides for spell checking and call it. But suppose that the system already has a spell checker available. Would it be better if we could use the one provided? Fortunately, this option is possible.

To make use of the system's checker you would have your program "**fork**" a process, which runs the system's spell checker. This is exactly the mechanism used by the shell program you use every day. When you type "`ls`", a process running the "`ls`" program is forked — that new process displays its output on the screen and then terminates.

The `fork` function is, in fact, more general. While it can be used to create a process running a particular program in the system, it can also be used to create a process to execute the code of a (e.g. `void`) function or another block of code in your program. You will learn about these various possibilities in the Laboraroty tasks in Section 3.

2.5 Understanding process creation

Once again look at the program (`fork.c`) that you were asked to try out in Section 2.3. Now compile and run the program again.

The program actually looks like a pretty simple program — except for the first `if` statement, perhaps. This is how the `fork` function is always called, and when it completes executing, it returns an integer value. Now what is unique about the `fork` function is that when it is called it creates a new process. But not just any process.

The process created is an exact copy of the process calling `fork` — this means that the entire environment of the calling process is duplicated including run-time stack, CPU registers, etc. In fact, when it creates the copy, it is a copy right down to the value of the program counter; so that when the new process begins execution, it will start right at the return from `fork`, just as its parent. So before the `fork` is executed, there is just the one process; and when the `fork` returns, there are two — and both are executing at the same point in the code.

Now we have two copies of the same program running — *what good does that do?*

It should be noted that the copy is not exactly the same. When `fork` returns in the original process (the *parent*) the return value is some positive integer, which is the process ID of the other process (the *child*). On the other hand, when the `fork` in the child returns, it returns the integer value `0` (zero)!

So in the parent process, the value of `pid` is some positive integer and in the child its value is `zero`. This means that, while the parent process will go on to execute the `if` part of the select statement, the child, will continue on the line with the comment:

```
1  /* child got here! */
```

Once more just to emphasise what goes on. When the child process is created, it is the parent process, which is executing. So when the child process is actually "detached", the program counter for the child process is somewhere in the `fork` code. That means that is where the child process will begin its execution in the `fork` code — just as the parent will do when it continues.

Once we have the two processes, the *parent* (original) and *child* (copy), they are both subject to the scheduling mechanism of the underlying operating system and are forced to take turns on the processor. That is why, when the program executes, the output from the two processes can *interleave*. In fact, if you run the program several times you will see that the order in which output is produced is not always the same.

So what `fork` has done for us, in a sense, is to allow us to bundle two program executions into a single program.

2.6 Executing another program as a child process

The `exec()` system call is used to create a new process that will overlay the process making the `exec()` call. The process calling `exec()` will terminate.

There are a number of variations of this system call. Each will perform the same operation but they accept different parameters. Do lookup the `man` pages related to these system calls and identify the differences. Also note what are the parameters for each of those system calls.

Download the code for the following program, `execl-ls.c`. It has a simple modification of our first program (`fork.c`). Read the program and think about its output. The following statement calls a variation of `exec()` called `execlp`.

```
execlp("/bin/ls", "ls", "-l", NULL);
```

The point here is to use a `ls` process to replace the child process in `fork.c`. You can replace the `ls` process with any other executable programs.

Run the program (`./execl-ls`) to demonstrate that the child process is created, replaced by `ls`, and then terminated:

```
$ gcc execl-ls.c -o execl-ls
$ ./execl-ls
```

3 Assessed Laboratory Tasks

3.1 Task 1: Understanding the process tree (20%)

Try tracing the process tree for a few other processes other than `bash`, in the `ptree.txt` file you created in Section 2.2. What can you observe?

Now, try running the `htop` command as follows:

```
$ htop -t
```

Is the result of running `htop` in agreement with your results from manually tracing the process tree earlier?

Submit the `ptree.txt` file that you have created earlier, together with a brief summary of your findings.

3.2 Task 2: Tracing the Process IDs (20%)

The shell command called `ps` is useful in the context of our current studies. When you execute the `ps` command the system will respond by printing a listing of currently existing processes. Depending on the arguments you give to the command, you will get varying degrees of detail.

Execute the following command at the shell prompt.

```
$ ps -a -l
```

This should generate a listing consisting of several columns of data, with each row being data for a particular process. Here are things to look for.

Attribute	Description
`UID`	This is the "user ID" of the user for whom the process was created (i.e. this should be your user ID).
`PID`	This is the "process ID".
`PPID`	This is the process ID of the parent.
`CMD`	This is the name of the executing program.

There are other entries, but not of interest at this time. These are not all the processes running on your machine. Try the following command and you will see a much longer list.

```
$ ps -e -l
```

Notice that there may be more UIDs in this listing. One thing you can do is to pick a process and then follow the trail of PPID-PID through a succession of parents, grandparents, etc.

We want to use this command to give us a snapshot of the active processes while one of our child processes is still active.

Make the following changes to your program and name this modified program as `fork2.c`:

- At the start of the program, add the line: `#include <stdlib.h>`
- In the branch for the child process, add the following line before the call to `count()`:

```
system("ps -e -l");
```

Run the program, and look at the output and trace back the sequence of parent process IDs for the child process. **Document your findings for the following**:

(a) Locate the child process in the listing and underline its PID;

(b) Put a circle around the parent's process ID for that child (it's right next to the PID);

(c) Draw a line from the circled parent's ID to the PID for that process (find it on a previous line);

(d) Repeat this for the parent, and its parent, etc., until you reach the process with PID 1.

In addition to your findings above, are there any differences between using `exec()` and `system()`? If there are differences, can you list two of them?

3.3 Task 3: Understanding process scheduling (30%)

Refer to the program (`fork.c`) which you have downloaded in the pre-class preparation, Section 2.3. Notice the `for-loop` with the loop variable '`j`'. This is a *delay* loop, to make the parent and child processes take a longer time to complete their work. If you haven't already done so, try running the `fork` program several times. **What can you say about the output?**

Now, to see the effect of the delay loop, change the limit value by dropping one zero (to make it `100,000`). Re-compile and run the program. **What has happened to the pattern of executions of the two processes?** Continue to drop one zero from the limit value until the value of the limit is 100. **Are there any changes?**

Now, what if we added more zeroes to initial limit value of `1,000,000` instead, so that the value of the limit is `10,000,000`, or `100,000,000`? **Does this result in anything different?**

Can you **find a value for the limit** so that one process can print out four values at a time without losing control (but no more than 4 values)?

By the way, there is an alternative (less precise but simpler) method for waiting — but in integer chunks of *seconds*. Now, change the waiting code in the program with the following bit of code:

```
1  sleep(2);
```

Re-compile and run the program again. **What differences in the output do you notice?**

The process will be blocked from executing for 2 seconds. Note that the parameter to the `sleep` function must be an integer, which is interpreted as seconds. This is a useful function to remember.

Submit a concise summary and explanation of your observations of the output that were produced when you made the various changes to the value of the limit, and before/after adding the `sleep` function as described above.

3.4 Task 4: Implementing a Multi-Process Factorial Calculator using `fork()` (30%)

Write a C program that creates multiple child processes using the `fork()` system call. Your application must perform the following tasks. You should use the provided template file `task4.c`, which can be downloaded from Moodle, and complete it according to the specifications below.

Parent Process:

- **Input:** Ask the user for the number of child processes (N) to create.
- **Create Children:** Create N child processes.
- **Wait:** Wait for all child processes to complete.
- **Terminate:** Print a message stating that all children have terminated and then exit.

Child Processes:

- **Print Information:** Print its PID and the PID of its parent process.
- **Calculation:** Calculate the factorial. Child one should calculate 1! (factorial of 1), child 2, 2! and child N, N!.
- **Result:** Print the result of the calculation.
- **Terminate:** Exit properly.

Requirements:

- The program should create child processes using the `fork()` system call.
- Include error handling for `fork()`.
- Use the `wait()` system call in the parent process to ensure that all child processes have terminated before the parent process exits.
- The code should be well-commented to explain the functionality.

3.5 Wrapping Up

Please upload all your work to the Moodle submission link, which should include your .c source files and your responses to any non-coding tasks. Please note that non-code answers should be provided in plain text files. There is no need to include the compiled executable programs. Ensure to upload all the necessary files before the conclusion of your lab class. Any delay in submitting these files to Moodle will result in a late penalty.