FIT2100 Practical #3
Process Creation and Execution
Week 7, Semester 2 2019

September 4, 2019

Revision Status:

`$Id: FIT2100-Practical-03.tex, Version 2.0 2018/08/21 17:30 Jojo $`

`Updated by Daniel Kos, Aug 2019`

Contents

1 Background 3

2 Pre-lab Preparation (4 marks) 3

2.1 Processes 3

2.2 Process tree 4

2.3 How to create a new process 5

2.4 Basic descriptions for the `fork` command 7

2.5 Understanding process creation 7

3 Practical Tasks (6 marks) 9

3.1 Task 1: Delaying with the `sleep` function (1 mark) 9

3.2 Task 2: Tracing the Process IDs (3 marks) 10

3.3 Task 3: Executing another program as the child process (2 marks) 11

1 Background

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

There are some pre-lab preparation (Section 2) tasks that you should complete before attending the lab. The practical tasks specified in Section 3 are also to be assessed in the lab.

Before you attempt any of the tasks in this prac, create a folder named PRAC03 under the FIT2100 folder (~/Document/FIT2100). Save all your source files under this PRAC03 folder.

2 Pre-lab Preparation (4 marks)

2.1 Processes

We will start by using the **ps** command to look at the processes running on your Unix server.

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 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:

```
$ 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
O	The process is running on a processor.
S	Sleeping — the process is waiting for an event to complete.
R	Runnable — the process is on the ready queue.
T	The process is stopped.

To find out what processes you are running on your Unix server, 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:

```
$ ps -u student
```

Only processes owned by user: `student` will be listed.

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 on your Unix server using the ps command.

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

```
$ 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?

Pre-class exercise (3 marks):

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. `/sbin/init`) (`init` is the original process when the system starts up; ancestor of every other process), thus creating the process tree for your current command interpreter. **You should submit the process tree that you created as part of the pre-lab submission.**

2.3 How to create a new process

The `fork` system call is fundamental to the use and operation of the Unix 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`').

Throughout this practical, you will experiment with the `fork` function to uncover some of its interesting properties.

Pre-class exercise (1 mark):

Download the following code for `fork-ex1.c`. Compile the program and then run it. You will need this program in the lab tasks.

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

void count (int start, char ch);

int main (int argc, char* argv[]) {
    int pid;
    int start = 0;

    pid = fork();              /* fork a child process */

    if (pid > 0) {             /* parent continues here */
        count(start, 'P');
        /* get all printing done before the shell prompt shows */
        wait(NULL);
    } else if (pid == 0) {     /* child got here! */
        count(start, 'C');
    } else {                   /* only if there was a problem with fork */
        perror("Failed to fork a process");
        exit(1);
    }
}

void count (int start, char ch) {
    int i, j;

    for (i = start; i < 10; i++) {
        for (j = 0; j < 1000000; j++);     /* delay */
        printf("Message from %c at the %dth iteration\n", ch, i);
    }
}
```

To compile this C program, use gcc compiler with the following command:

```
$ gcc fork-ex1.c -o fork-ex1
```

To run your program, type the name of the executable file at the command prompt. If you compiled your program as described above, you should type `./fork-ex1` at the command prompt.

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

2.4 Basic descriptions for the `fork` command

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

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 practical tasks in Section 3.

2.5 Understanding process creation

Once again look at the program (`fork-ex1.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?*

Recall from last week's tutorial, 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 `then` part of the select statement, the child, will continue on the line with the comment:

```
  /* 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.

3 Practical Tasks (6 marks)

3.1 Task 1: Delaying with the `sleep` function (1 mark)

Refer to the program (`fork-ex1.c`) given in the pre-lab preparation, Section 2.3. You should notice the `for-loop` with the loop variable '`j`'. This is a *delay* loop, to slow down the processing so that we can see the independence of the parent and child processes. (Execute the program again.)

Now to see the effect of the delay loop, change the limit value by dropping one zero (to make it `100000`). Re-compile and run the program. What has happened to the pattern of executions of the two processes?

Change the value to `500000` and repeat the experiment. Can you change the value so that one process can print out four values without losing control (but no more than 4 values)?

Your task: Summarise the results of your experiments and document your explanation for the changes in the execution patterns.

By the way, there is an alternative (less precise but simpler) method for waiting — but in integer chunks of *seconds*.

Your task: Change the waiting code in the program with the following bit of code:

```
   sleep(2);
```

Re-compile and run the program again. What do you notice?

The process 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.

3.2 Task 2: Tracing the Process IDs (3 marks)

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 UID's 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.

Your task: Make the following changes to your program and name this modified program as `fork-ex2.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 finding 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`.

3.3 Task 3: Executing another program as the child process (2 marks)

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. Please read the `man` pages related to these system calls and identify the differences. Also note what are the parameters for each of those system calls.

Look at the `execl-ls.c` that is available on Moodle. It has a simple modification of our first program (`fork-ex1.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-ex1.c`. You can replace the `ls` process with any other executable programs.

Compile `execl-ls.c` using the following command line:

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

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

Your task:

To get a better handle on how to create a process and *overlay* it with an executable code of a program and to execute it, do the following.

Instead of running the `ls` command in the example given above, write a C program to **do one of the following**:

(a) A program that reads a sequence of integers from the terminal and computes the average, min, max and standard deviation and prints the result. You can assume that these numbers are positive integers.

(b) A program that finds the next prime number after 100.

(c) Given three positive integers, the program identifies whether these numbers can form the sides of a triangle. If so, whether the triangle is an equilateral or isosceles or right angle triangle.

Compile your C program into executable code. Incorporate the execution overlay in the `fork/exec` system calls that you did previously and observe the execution of the program.