



FIT2100 Practical #3

Process Creation and Execution

Week 6 Semester 2 2017

Dr Jojo Wong
Lecturer, Faculty of IT
Email: Jojo.Wong@monash.edu
© 2016-2017, Monash University

August 28, 2017

Revision Status:

\$Id: FIT2100-Practical-03.tex, Version 1.0 2017/08/24 17:30 Jojo \$

Contents

1	Background	4
2	Pre-lab Preparation (3 marks)	4
2.1	Processes	4
2.2	Process tree	5
2.3	How to create a new process?	6
2.4	Basic descriptions for the <code>fork</code> command	8
2.5	Understanding the process creation	8
3	Practical Tasks (7 marks)	10
3.1	Task 1: Delaying with the <code>sleep</code> function (2 marks)	10
3.2	Task 2: Tracing the Process IDs (2 marks)	11
3.3	Task 3: Executing another program as the child process (2 marks)	13
3.4	Task 4: Termination and communication of processes (1 mark)	15
3.5	Optional Task: Something you should try	16

1 Background

This practical is aimed 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) that you should complete before attending the lab. The practical tasks specified in Section 3 are to be completed and assessed in the lab.

2 Pre-lab Preparation (3 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):

```
1 $ ps -ef | more
```

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

(Note: `ps -ely` will produce a lengthy output for each of the process.)

Type in the following command:

```
1 $ ps -eo 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 summary of different process states:

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 run queue.
T	The process is stopped.

To find out what processes you are running on your Unix server, use the following command line:

```
1 $ ps -ef | grep your_username
```

For example:

```
1 $ ps -ef | grep jojow
```

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 `jojow`.

An alternative is to use the `-u` option:

```
1 $ ps -u your_username
```

For example:

```
1 $ ps -u jojow
```

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 is a *process tree*. We will investigate the process tree on your Unix server using the `ps` command.

Execute the following command:

```
1 $ ps -eo 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 it yourself (2 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):

```
1 $ 3806 3807 jojow 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. `init`), 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` command is fundamental to the use and operation of the Unix operating system.

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

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

Try it yourself (1 mark):

Create the program presented in the next page and name it as `"fork-ex1.c"`. Compile the program and then run it. **You should submit the source file as part of your pre-lab submission.**

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <unistd.h>
4  #include <sys/types.h>
5
6  void count (int start, char ch);
7
8  int main (int argc, char *argv[])
9  {
10     int pid;
11     int start = 0;
12
13     pid = fork();          /* fork a child process */
14
15     if (pid > 0)           /* parent continues here */
16     {
17         count(start, 'P');
18         /* get all printing done before the shell prompt shows */
19         wait(NULL);
20     }
21     else if (pid == 0)    /* child got here! */
22     {
23         count(start, 'C');
24     }
25     else                 /* only if there was a problem with fork */
26     {
27         perror("Failed to fork a process");
28         exit(1);
29     }
30 }
31
32 void count (int start, char ch)
33 {
34     int i, j;
35
36     for (i = start; i < 10; i++)
37     {
38         for (j = 0; j < 1000000; j++);    /* delay */
39         printf("Message from %c at the %dth iteration\n", ch, i);
40     }
41 }
```

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

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

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` 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 (void) function or other block of code in your program. You will learn about these various possibilities in the practical tasks in Section 3.

2.5 Understanding the process creation

Once again look at the program (`fork-ex1.c`) that you 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, you 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:

```
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.

3 Practical Tasks (7 marks)

3.1 Task 1: Delaying with the sleep function (2 marks)

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:

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

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

The process will suspend the execution (go into the “waiting” state) for 2 seconds. Note that the parameter to the `sleep` function must be an integer, which is interpreted as seconds. You may find this call convenient to use at some time.

3.2 Task 2: Tracing the Process IDs (2 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.

```
1 $ ps -al
```

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.

```
1 $ ps -el
```

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()`:

```
1 system("ps -el");
```

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, as we have seen in Tutorial 3 last week. 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 `exec1-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 `exec1p`.

```
1  exec1p("/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, including the shell scripts (see more exercises suggested below.)

Compile `exec1-ls.c` using the following command line:

```
1  $ gcc exec1-ls.c -o exec1-ls
```

Run the program (`./exec1-ls`) to demonstrate that the child process is created, replaced by `ls`, and then terminated.

Your task:

To get more 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 or a shell script to **do one of the following**:

- (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) Find 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 an executable code. Incorporate the execution overlay in the fork/exec system calls that you did previously and observe the execution of the program.

3.4 Task 4: Termination and communication of processes (1 mark)

Communicating and terminating a process in Unix is performed through *signals*. Processes which receive signals can decide whether they want to act on the signal or ignore it. Signal values are small positive integers and the first 15 signals are pre-defined in all the Unix systems.

The command to send a signal to a process is `kill`. Use `man` page to find the meaning of the first 15 signals.

Of interest, it is the signal number 2. The action of this signal is to “interrupt” the current interactive process. This signal is generated by pressing <Ctrl-C> on the standard input and the current interactive job will be terminated; and the terminal will return with the shell prompt.

The following shell script `trap-example.sh` (available on Moodle) when run interactively will ignore this signal. Hence when the script is run interactively, the terminal will be *locked* forever because of the infinite `while` loop.

The only way to terminate this process is login in another terminal, identify its process ID and send the signal number 9 to that process (or suspend the process and `kill` the suspended process).

```
1  #!/bin/sh
2
3  # trap the signal number 2 and ignore it so that
4  # when this script is run, it cannot be terminated
5
6  trap "echo you have Pressed Control-C — I am ignoring it" 2
7
8  # perform an infinite loop
9  while :
10 do
11     echo "Script that ignores the interrupt signal"
12     sleep 2
13 done
```

Your task: Modify the above script such that when you press <Ctrl-C>, the script displays a message and terminates. (**Hint:** Find out the “`exit`” command in `bash` by using the `help` command.)

3.5 Optional Task: Something you should try

Create a file with name `fork-wait.c` which does the following:

- The original process spawns exactly two child processes.
- Each of the child processes computes the “factorial” of integers between 1 and 10 by recursion, prints the results to the screen, and then terminates. Make sure to print an identifying string for the output of each child process as in:

```
1 CHILD1: fact(1) = 1
2 CHILD2: fact(1) = 1
3 CHILD2: fact(2) = 2
4 CHILD1: fact(2) = 2
5 ...
```

- Don't worry if the order of the output strings produced by the children processes turns out to be different from the example above.
- The parent process waits for the children to terminate before reaching termination itself. You can accomplish this with `wait(2)` or `waitpid(2)`.