FIT2070 Operating Systems

Semester 2 2016

Practical 2: Process Creation and Execution

Part I: Pre-lab Preparation

Task 0: Unix commands

Look at the **man** pages of the following commands:

a. **ps**
b. **top**
c. **jobs**
d. **uptime**
e. **pgrep**
f. **pkill**

Task 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 -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:

```
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.

Repeat the above **ps** command, this time taking note of the states of the processes.

Process states:

O Process is running on a processor.
S Sleeping: process is waiting for an event to complete.
R Runnable: process is on the run queue.
T Process is stopped.

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

```
ps -ef | grep your_username
```

For example:

```
ps -ef | grep jojow
```

This command causes **ps** to list all 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 **–u** switch:

```
ps –u your_username
```

For example:

```
ps -u jojow
```

Task 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:

```
ps -eo ppid,pid,user,s,comm > ptree.txt
```

This saves the process listing to a file **ptree.txt**.

What does the **ppid** parameter represent?

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  jojow  S   -bash
```

Use the parent *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 this process until you arrive at **process ID 1** (**init**), thus creating the process tree for your current command interpreter. Show your tutor the process tree.

Task 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, i.e., the shell process, and by the shell when you execute a shell command (e.g., '**ls**'). In this lab you will experiment with the **fork** command to uncover some of its interesting properties.

(**References:** UNIX **man** page for the **fork**.)

Create the following program into a file (call it **fork-ex1.c**), compile it, and run it.

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.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');
            wait(NULL); /* To get all printing done before shell prompt shows*/
      }
      else if (pid == 0)       /* child got here! */
      {
            count(start, 'C');
      }
      else                     /* only if there was a problem with fork */
      {
            printf (" Failed to fork a process \n");
            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);
      }
}
```

Compiling and Running C Programs: To compile C programs, use the program **`gcc`** compiler.

Type the following at the command prompt:

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

The flag '**`-o`**' tells the compiler to put the executable in a file named **`fork`**.
If you left '**`-o fork`**', the executable would be put in a file named **`a.out`**.

To run your program, type the name of the executable at the command prompt. If you compiled your program as described above, you would just type the name **`fork`** (or **`./fork`**) at the command prompt.

Task 4: Basic descriptions for the `fork` command (read this)

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 the system already has a spell checker available. Would not 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 examples and exercises in **Part II**.

Part II: Lab Exercises

Task 5: Understanding the Process Creation

Once again look at the program given in the pre-lab part (**Part I Task 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.

So what is the big deal! Now we have two copies of the same program running – what good does that do?

Well, there is one piece of information you do not have yet. 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 when 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 is *interleaved*. 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 allow us to bundle two program executions into a single program.

Task 6: Another way of delaying with `sleep`

When you look at the example above notice the **`for-loop`** (with 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 (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)?

Summarise the results of your experiments and include 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. If you use the following bit of code:

```
sleep(2);
```

then the process will suspend execution (go into the waiting state) for 2 seconds. 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.

Task 7: Tracing the Process IDs

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 -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.

UID – This is the "user ID" of the user for whom the process was created (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 -el
```

Notice that there are 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. 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 -el");`

Run the program, and look at the output and trace back the sequence of parent process IDs for the child process. This means:

a. Locate the child process in the listing and underline its **PID**;
b. Put a circle around the parent id for that child (it's right next to the **PID**);
c. Draw a line from the circled parent 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**.

Task 8: How to execute another program as the 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. 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. 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, including scripts (see more exercises suggested below.)

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

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

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

Practice more with the following exercises:

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

Instead of running the **`ls`** command in the example given above, write a C or java program or a 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 the 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.

Task 9: Termination (and communication) of Processes

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, is the signal number is **`2`**. The action of this signal is to interrupt the current interactive process.

This signal is generated by pressing <**Control-C**> on the standard input and the current interactive job will be terminated and the terminal with return with the Shell prompt.

The following 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).

```
#!/bin/sh

# trap the signal number 2 and ignore it so that
# when this script is run, it cannot be terminated

trap 'echo you have Pressed Control-C -- I am ignoring it' 2

# perform an infinite loop
while:
do
        echo "Script that ignores the interrupt signal"
        sleep 2
done
```

Modify the above script such that when you press <**Control-C**>, the script displays a message and terminates. (Hint: see the **`man`** page for the "**`exit`**" command.)

Additional Task: Something you should try …

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

- The original process spawns exactly two children processes.
- Each of the children 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:

  ```
  CHILD1: fact(1) = 1
  CHILD2: fact(2) = 2
  CHILD2: fact(2) = 2
  CHILD1: fact(2) = 2
  ...
  ```

- 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)`**.