CSCI 315 Fork Tutorial

1 Introduction

The `fork` function 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., shell process[1], and by the shell when you execute a shell command (e.g., '`ls`').

2 The Basics

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 – wouldn't it be better if you could use the one provided? Fortunately this second 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[2]. 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 below.

2.1 Example 1

Here is a very simple program which uses the `fork` function. You should have created a file containing this code in the prelab exercise. Now compile and run the program.

```
        using namespace std;

1       #include <sys/types.h>
2       #include <sys/wait.h>
3       #include <unistd.h>
4       #include <iostream>

5       void Count(int start, char ch);
```

[1] A process is the object created by an operating system for (managing) the execution of a program.

[2] To be honest, if you wanted to do this you wouldn't use the `fork` command. Instead you would probably use the `exec` command – but the `exec` command uses `fork` to create the process, so `fork` is still the focus of attention.

```
6    int main(int argc, char *arg[])
7    {
8       int pid;
9       int start = 0;
10      pid=fork();
11      if (pid > 0) {          // parent continues here
12          Count(start, 'P');
13          wait(NULL);  // To get printing done before shell prompt shows
14      }
15      else if (pid == 0)      // child got here!
16          Count(start, 'C');
17      else                    // only if there was a problem with fork
18          exit(-1);
19      cout << endl;
20   }

21   void Count(int start, char ch)
22   {
23      int i, j;
24      for (i = start; i < 10; i++) {
25         for (j = 0; j < 1000000; j++) ;
26         cout << endl << ch << ": " << i;
27      }
28   }
```

This actually looks like a pretty simple program – except for the first `if` statement, perhaps[3]. 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 big deal! Now we have two copies of the same program running – what good does that do? Well, there's one piece of information you don't 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)![4] 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.

[3] Actually, the use of `fork` above is not in the traditional form. Traditionally *C* and *C++* programmers would put the assignment on line 10 into the selection condition on line 11 as follows.

```
if ((pid = fork()) > 0)
```

[4] The example clearly contains a third possible scenario: the `fork` command could also return a negative value. This is done when some error occurred during process creation; for example, there is not enough memory to create another process.

That's 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.

2.2 More Examples

In the list that follows each entry is (or ends with) an activity for you to carry out. Write complete responses in each case.

1. 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)?

   Summarize 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.

2. As a second experiment, turn to the original program and replace the last `cout` statement in `main()` with the following:

   ```
   cout << endl << "Now I'm done!" << endl;
   ```

   Re-compile and run the resulting program.

   How many times does the message get printed? What does this tell you about the execution path of each process? To answer this question write down the sequence of numbers of the lines executed by each of the processes.

   [We can insure that only the parent process executes the message at the end by adding a new statement to the end of the child's `else-if` clause. Immediately before the closing '`}`' add the statement `exit(0);` Now compile and execute the program and see what the result is. Notice that in this way the code executed for the child process is limited to that which appears in the child's component of the selection statement. You don't have to hand anything in for this bracketed part.]

3. 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 the command, you will get varing degrees of detail. Execute the following command.

   ```
   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 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:

- 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 include the output as part of your handin. On the printout of your handin file 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.

4. First, comment out the line in your program with the call to `system`. Now change the initial value given the variable `start` to 5 – re-compile and run. What is the effect on the parent and child processes? Now return the initialization to its original form and insert the following line

   ```
   start = 5
   ```

   just *after* line 15, just before the child process calls `Count`. Notice that this reference to line 15 is relative to the listing at the top of this tutorial – not to your own code. So just find the corresponding point in your code.

   Write down what you think will happen (i.e., what sequence of values will be displayed by each process). Now compile and run the program to see what actually happens. What does this tell you about any connection or association between the two processes?

5. The normal way to execute a command is to type the command (along with any arguments) and hit return. The command will execute and when it completes you get the shell prompt back again. But, if you put the '&' symbol at the end of the command line, you get the prompt back right away – i.e., the shell doesn't wait for the child process to finish executing the command.

   Comment out the line

   ```
   wait(NULL);
   ```

   in the parent process branch of `main()`. Re-compile and run the program.

   If the shell process forks a process to execute a command, does our original program simulate executing with or without a trailing '&'? How about the modified program? Explain.

6. Copy the `forkmany.cc` file to your workspace (the file is also available in ~cs315/sunhome/Labs/Lab01/forkmany.cc). This program is similar to the original program except that it creates two children, not just one. In addition, the program is setup so that the second child, and only the second child, will print a message. Study the code and convince yourself that this is the behaviour. You might comment it to reflect those code bits which cause the creation of each child, and the printing by the second child. Notice that the parent creates children, but the children do not.

   Now, modify the code as follows.

   (a) Identify the statement responsible for printing out the process information from the second child. Comment out this statement so that it will no longer display the message. This should be easy to compile and test.

   (b) Modify the code so that each child process also creates two children (these grandchildren shouldn't create any offspring). By putting in judicious print statements, you should be able to convince yourself that all four children are being created.

   (c) Now, add code to the program so that only the very last child created will print the message of the original program – that message is a listing generated by the `ps` command.

   That's the first half of the problem. Once this works you should print out the generated `ps` listing, identify the lines which represent the four grandchildren, then identify the lines of each of the two children, then of the original parent. From the parent, work your way through the listing finding the predecessors of the parent – just follow the chain of PPID's. You might go back and, starting from the top, number the entries in the order of process creation (as indicated by the numeric order of the process ID's (PID's)). Finally, draw a tree with the "granddady" of all the processes as the root and it descendents as the lower level nodes. Each node should be labeled with its PID.

   Hand in the code for each program, making sure that there is sufficient commenting to identify the code executed by each child (grandchild) process. Also handin the output from the final program and the drawing of the tree. Please be neat with the tree drawing.