*1. Intro to Processes*

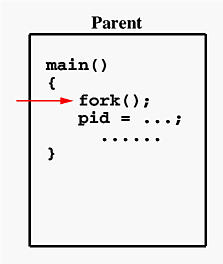
**Fork() system call.**

System call fork() is used to create processes. It takes no arguments and returns a process ID. The purpose of fork() is to create a new process, which becomes the child process of the caller. After a new child process is created, both processes will execute the next instruction following the fork() system call. Therefore, we have to distinguish the parent from the child. This can be done by testing the returned value of fork():

If fork() returns a negative value, the creation of a child process was unsuccessful.fork() returns a zero to the newly created child process.fork() returns a positive value, the process ID of the child process, to the parent. The returned process ID is of type pid\_t defined in sys/types.h. Normally, the process ID is an integer. Moreover, a process can use function getpid() to retrieve the process ID assigned to this process. Therefore, after the system call to fork(), a simple test can tell which process is the child. Please note that Unix will make an exact copy of the parent’s address space and give it to the child. Therefore, the parent and child processes have separate address spaces. Let us take an example to make the above points clear. This example does not distinguish parent and the child processes.

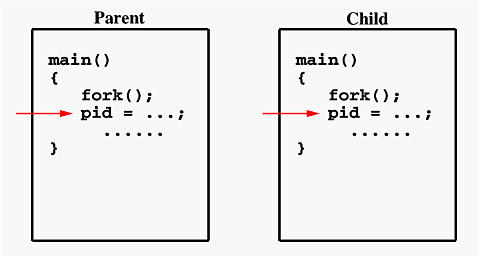
Open up the file **main.c**

Suppose the above program executes up to the point of the call to fork() (marked in red color):



If the call to fork() is executed successfully, Unix will

* make two identical copies of address spaces, one for the parent and the other for the child.
* Both processes will start their execution at the next statement following the fork() call. In this case, both processes will start their execution at the assignment statement as shown below:



Both processes start their execution right after the system call fork(). Since both processes have identical but separate address spaces, those variables initialized before the fork() call have the same values in both address spaces. Since every process has its own address space, any modifications will be independent of the others. In other words, if the parent changes the value of its variable, the modification will only affect the variable in the parent process’s address space. Other address spaces created by fork() calls will not be affected even though they have identical variable names.

What is the reason of using write rather than printf? It is because printf() is “buffered,” meaning printf() will group the output of a process together. While buffering the output for the parent process, the child may also use printf to print out some information, which will also be buffered. As a result, since the output will not be sent to screen immediately, you may not get the right order of the expected result. Worse, the output from the two processes may be mixed in strange ways. To overcome this problem, you may consider to use the “unbuffered” write.

If you run this program, you might see the following on the screen:

................

This line is **from** pid 3456, value 13

This line is **from** pid 3456, value 14

................

This line is **from** pid 3456, value 20

This line is **from** pid 4617, value 100

This line is **from** pid 4617, value 101

................

This line is **from** pid 3456, value 21

This line is **from** pid 3456, value 22

................

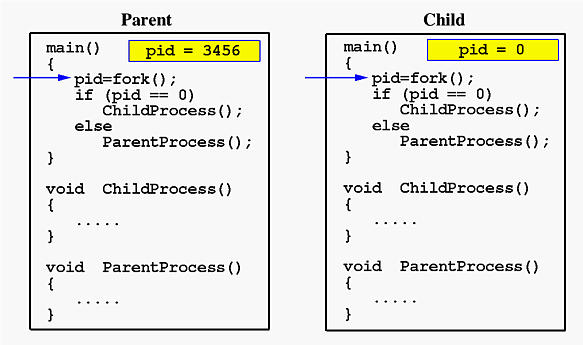
Process ID 3456 may be the one assigned to the parent or the child. Due to the fact that these processes are run concurrently, their output lines are intermixed in a rather unpredictable way. Moreover, the order of these lines are determined by the CPU scheduler. Hence, if you run this program again, you may get a totally different result.

Consider one more simple example, which distinguishes the parent from the child.

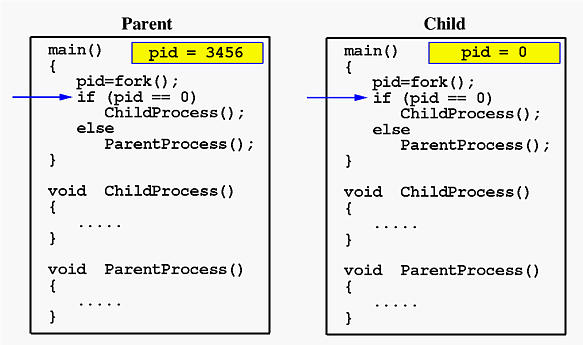
Open up the file **main2.c**

In this program, both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and (2) the value of variable i. For simplicity, printf() is used.

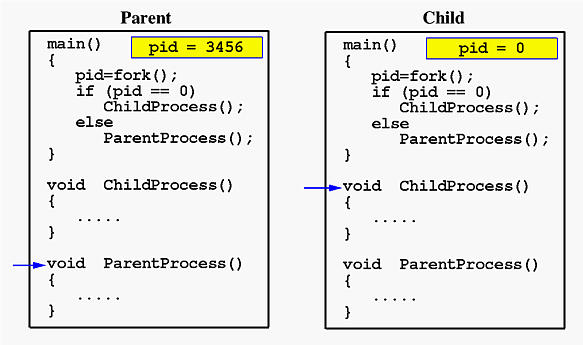
When the main program executes fork(), an identical copy of its address space, including the program and all data, is created. System call fork() returns the child process ID to the parent and returns 0 to the child process. The following figure shows that in both address spaces there is a variable pid. The one in the parent receives the child’s process ID 3456 and the one in the child receives 0.



Now both programs (i.e., the parent and child) will execute independent of each other starting at the next statement:



In the parent, since pid is non-zero, it calls function ParentProcess(). On the other hand, the child has a zero pid and calls ChildProcess() as shown below.



Due to the fact that the CPU scheduler will assign a time quantum to each process, the parent or the child process will run for some time before the control is switched to the other and the running process will print some lines before you can see any line printed by the other process. Therefore, increase the value of MAX\_COUNT to be large enough so that both processes will run for at least two or more time quanta. If the value of MAX\_COUNT is too small the process can finish in one time quantum, and you will see two groups of lines, each of which contains all lines printed by the same process.

Current layout: 2 Panels with tree

**2. Process Lab**

**Implement a program using Processes**

The Objective of this lab is for you to implement a program that creates two child processes. A total of 3 processes will exist after the creation (i.e. 1 for the parent process, and 2 child processes).

**Child Process**

Both child processes will loop for some random number of iterations, but no more than 30, while the parent process will wait for both to terminate. Open the terminal and use the man pages to lookup the random() function in C (e.g. man random). You will have to make sure you include the proper #include. Also, in order to use the random number generator you must seed it correctly using srandom(). If the man pages are not enough, use Google!

Within the loop, each child should print a message *“Child Pid: <pid> is going to sleep!”*, then call the sleep() function for a random amount of time, but no more than 10 secs (i.e. use the random()). When the child wakes up, it should print the message *“Child Pid: is awake!\nWhere is my Parent: <ppid>?”*. Use the getpid() to obtain the <pid> of the current running process, and getppid() to get the <ppid> of the parent. Use the man pages to lookup getpid() and getppid(). If the man pages are not enough, use Google!

Once the loop has terminated the child process should calll exit(0) to terminate.

**Parent Process**

After forking the two child processes, the parent process must call the wait() function to wait for both child processes to complete. As each processes completes, the parent should print out the message “Child Pid: <pid> has completed” using the wait(&status). Use the man pages to lookup the wait() and its usage. If the man pages are not enough, use Google!

**Makefile**

Create the makefile to compile the program. Use the prior labs as examples. You can call your executable anything you want. If those are not enough, use Google!

**Running your program**

Bring up another terminal. In the new terminal, lets view the status of the currently running processes by executing the command ps -ef on the command line. ps is the command to show a listing of the process status table. You can get more information about what is ps by viewing the man pages. After typing ps, notice the list of all the processes running in the background such as bash, sshd, init…etc. These are the programs for the command line shell, terminal, and the parent process of all processes init. There probably are others there that are providing some OS management services as well. Notice in the 2nd and 3rd columns are the PID and parent PPID of each of the processes. In the 4th column is the total time the processes have been running. In your coding terminal start your program you just compiled by typing in its executable name. Return to the other terminal that you executed ps. Lets view the status of all of your processes again after running your program. Notice you now have three processes with the same name as your executable. Look at the PID and PPID and remember their values. Now switch to the terminal running your code. Look at the data being printed out by your program. Notice the pid and ppid values. They should be the same.

**Terminating your program**

You can wait for the program to complete, but if you want to terminate the program early, you will need to send the program a signal to terminate. Ctrl-C will bring back the command line, but your program will still be running in the background (i.e. that includes the parent, and two child processes) just like the other OS management services. In order to terminate the program (i.e. process), we need to send a signal using the kill() system call. Look at the process status table and identify the two child processes. They will both have the same process name (i.e. the executable you gave it in your gcc command line in the Makefile ), and also have the same number in the process status table for their parent pid PPID. There should also be another process with the executable name that has its PID equal to the PPID of each of the child processes. You must kill the parent process, and the parent will kill the child processes. On the command line type the following.

kill -9 \<PPID\>

where <PPID> should be replace by the number shown in the process status table after executing the ps command.

In your spare time, play around with the code. Increase the nunber of child processes, increase or decrease the sleep time, or random numbers that are generated.

When complete push your code to github, and post a link to your github in Canvas.

Mark as CompletedBack to dashboard