O.S. Lab 1A: Linux Process Creation and Control

Before we start …

It should go without saying that all the work that you will turn in for this and all other labs in this course will be yours. Do not surf the web to get *inspiration* for this assignment, do not include code that was not written by you. You should try your best to debug your code on your own, but it's fine to get help from a colleague if that means getting assistance to identify the problem and doesn't go as far as receiving source code to fix it (in writing or orally).

Set Up

Create a text file called **answers.txt** in which you will write responses to questions throughout this lab. Identify each answer by the number of the problem and the item to which they correspond. For instance: when answering problem 1, item 1, identify your answer as (1.1).

Linux Processes

In this lab, we start experimenting with a few Linux *system calls*. The first one we will look at is **fork**, which is used by a process to spawn an identical copy of itself. When we learn to use a system call or a library function, it is helpful to follow the simple workflow described as follows.

Start by reading the **man** page of the system call or library function in which you are interested. This will help you begin to understand how it works, but it will also show you some practical details that are essential to using it successfully. In the man page, pay close attention to the **SYNOPSIS**; it will tell you:

1. The files you must **#include** in your program.
2. The function prototype(s) with which you will work.

Try it by typing **man fork** from a Linux prompt. From this we learn that any program calling **fork** will need to **#include** the file **unistd.h**. The "angle brackets" indicate that these files reside in an include directory owned by the system (most often **/usr/include**).

We also learn that the **fork** call:

- Returns a value of type **pid_t** (essentially, an integer), and
- Does not take any input parameters, what is indicated by the formal parameter **void**.

Once we have tried our best to understand that information, we should not be so bold as to throw code into a large program to see how things work out. It is often more productive to write a small program just

to test that we have the right understanding about the behavior of the function. Once we have experimented a bit with this program and are convinced that the function does what we expect and that we have learned to use it effectively, we can use it in a larger context.

A first experiment with **fork()** aimed at understanding what a child process inherits from a parent is provided to you in file **fork-test.c**. Looking at this code, you may be inclined to think that you can infer the order of execution of these lines of C code. For instance: you might say that that parent executes first and the child executes next; or you might say that the order of execution is the one in which the program was written.

Don't make the mistake of thinking that you can predict the order of execution of the actions in your processes! The process scheduler in the kernel will determine *what* executes *when* and your code should not rely on any assumptions of order of execution.

Problem 0 (3 points)

Instructions for Problem 0.

1. Create a folder named **Lab1a** to hold all of your work for this lab.
2. Read the manual pages for the following items: **ps** (1), **kill** (1), **fork** (2), **exit (**2), **wait** (2), **waitpid** (2), **execv** (3), **system** (3).
3. Additionally, since you will be working with C file I/O in this lab, you should review the following manual pages to get ready: **open** (2), **read** (2), **write** (2), **close** (2).
4. **[2 points]** In your **Lab1a** directory, create a program called **myprog.c** described below.
5. **[1 point]** Create a **Makefile** to compile **myprog.c**. **You will augment this Makefile to build ALL the deliverables for this lab. Not providing a Makefile that builds all deliverables will assign you a 2 point penalty to the overall assignment.**

When you have completed the problem, add the **Makefile** and **myprog.c** files to your **Lab1a** directory for later submission.

myprog.c Details

For this part of the lab create a program called **myprog.c** that spawns **two child processes**.

- The parent process should run an infinite loop printing to the terminal the string "**parent:**" followed by an increasing counter value, and a new line character.
- The children processes should also run an infinite loop printing to the terminal the string "**child1:**" or "**child2:**" followed by an increasing counter value, and a new line character.

Make sure that your program compiles – don't expect to receive much *(if any)* partial credit if you ever turn in code that does not compile. Additionally, to earn full credit, make sure to debug your code so that it runs as expected. The file **fork.c** used in Problem 1 below may give you a hint as to how to do this.

Important: The parent and the children should have separate counter variables. At the end of each iteration of their respective loops, each process' counter variable gets incremented by one.

The output of the program should *resemble* what appears in the box at right. *Note that the specific values of the counters will be different in your program and that the output will scroll by fast.*

Very important: make sure that **child1** and **child2** are processes spawned by the same **parent** process.

```
parent: 1
parent: 2
parent: 3
parent: 4
…
child1: 1
child1: 2
child2: 1
…
parent: 321
parent: 322
…
child2: 143
child1: 142
…
```

Problem 1 (4 points)

Let's start slowly by investigating what a child process may be inheriting from its parent process. First, let's get this code to compile!

[1 point] a) Take a look at the program given to you in file **fork.c**. Compile and execute the program. Add code to have both the child and the parent print out the value of the *pid* returned by the **fork()** system call.

[1 point] b) The variable **num** is declared before the call to **fork()** as shown in this program. After the call to **fork()**, when a new process is spawned, does there exist only one instance of **num** in the memory space of the parent process shared by the two processes or do there exist two instances: one in the memory space of the parent and one in the memory space of the child? Discuss your conclusion in **answers.txt**.

Now, let's experiment with forcing a specific order of termination of the processes. As given to you, the code for this problem makes no guarantee that the child will terminate before the parent does! With the concepts we have covered so far in class, we can use a very basic mechanism to establish order in process creation (with **fork**) and in process termination (with **wait** or **waitpid**).

[2 points] c) Copy **fork.c** to file **fork-wait.c** and modify it so that you can guarantee that the parent process will always terminate **after** the child process has terminated. Your solution cannot rely on the termination condition of the **for** loops or on the use of **sleep**. The right way to handle this is using a *syscall* such as **wait** or **waitpid** – read their man pages before jumping into this task. One more thing: Modify the child process so that it makes calls to **getpid** and **getppid** and prints out the values returned by these calls.

When you have completed the problem, add the **fork-wait.c** file to your **Lab1a** directory for later submission. Remember to edit **Makefile** to accommodate the new program.

Problem 2 (5 points)

This problem will help you remember some material you studied in Computer Systems. Do you remember that every running program (a.k.a. *process*) defines four segments of memory: text, data, stack, and heap?

[1 point] a) Read **fork-data.c** carefully, then compile and run it. In **answers.txt** explain in which segment of your running program the following variables reside: **pid**, **x**, **y**, **i**, and **j**.

[1 point] b) In **answers.txt**, discuss whether running **fork-data.c** allows you to conclude: (1) if the data segment and the stack segment of a parent process are copied over to the child process; and (2) whether changes made to these variables by the child are seen by the parent. *What you discover for (2) will tell you whether parent and child share the same memory for data and stack segments or if they each have their own separate segments.*

Next, this problem will get you to investigate more deeply what a child process may inherit from its parent. This time, we will be working with *files* rather than variables. If a parent process has opened a file and then goes on to spawn a child process, you should wonder if the child will see the same file in open state. That is, will the file descriptor that the parent received after a call to **open** be usable in the child? Furthermore, by reading from a file shared by inheritance, does a process affect the "state" of this file that another process may be reading?

[1 point] c) Copy the file given to you as **fork-data.c** to a new file called **fork-file.c**. Modify your new program so that before the **if/fork** structure, **main** creates and opens a file called **data.txt** and writes into it the string *"this is a test for processes created with fork\nthis is another line"*. Close the file right before the **fork** call so the writing to the file can be complete. Immediately following the **close** call, open the file again for reading. Inside the *parent code* section of the program, have the parent issue a single call to read to get 5 characters from the file and print them to the terminal. Inside the *child code* of the program, have the child process issue a single call to read to get 5 characters from the file and print them to the terminal. Compile your code and run it to observe what happens.

[1 point] d) In case it is not obvious: the file you open in **main** is visible in both child and parent processes. Experiment with your modified **fork-file.c** and write in your **answer.txt** file the answers to the following questions: (1) if one process closes the file, can the other still read from it?; and (2) say the child process reads from the "inherited" file; does that affect what the parent will read from the same file descriptor?

And now for something completely different. Every time you make a syscall or invoke a library function, look at the RETURN VALUE section of its **man** page. If it says something like "*On error, -1 is returned, and errno is set appropriately,"* you should consider *wrapping* that call with a function of your own. By doing so, you can effectively replace calls that your program makes to these services to your customized version of that function, which will react to errors in a standardized manner. In the remainder of this problem, you will write your first wrapper to a syscall.

[1 point] e) Create a new function (outside main) in the file **fork-file.c** to *wrap* the call to **fork** and perform some basic error detection. This function should have the same prototype as the **fork** system call, but its name will start with a capital letter, that is, its prototype will be:

pid_t Fork(void);

In this new function, you will invoke the system call **fork** and check if the return value is **-1**. When that is the case, your function should invoke **perror** to print out a human readable error message and then call the library function **exit** with argument **-1** to abort the program. This will terminate the process that called **Fork** and pass a return code to the creator of that process. (Hint: make sure to read the man pages to **perror**, **exit**, and any other system or library calls used in this lab and to **#include** in your code the header files you need.) **After you create a wrapper for ANY function in this class, be sure that your programs use YOUR wrappers instead of the original functions.**

When you have completed the problem, add the **fork-file.c** file to your **Lab1a** directory for later submission. Remember to edit **Makefile** to accommodate the new program.

Problem 3 (4 points)

Before you get into this problem, let's talk about a process' *termination status*. In Problem 2, you were asked to check the return value of the call to **fork()** and to terminate the parent process when it fails. The mechanism for termination we suggested was to invoke the **exit** library call with argument **-1**. *By convention,* when a Linux process terminates without error, it returns or exits with status zero. When the termination status is different from zero, this convention indicates that an error condition arose.

You can see how this works out in practice with a little experiment that you can run from your **bash** shell. (This is the default shell used on knuth. To figure out which shell you are running type **ps -p $$** at the command prompt.) The **cat** system utility, which resides in the **/bin** directory, can be used to display the content of a text file on the terminal. Try this out:

yourname@knuth2:~$ /bin/cat fork-file.c

Note that we are using the *absolute path* to invoke the **cat** utility and we are doing it just so that you don't execute any other program with the same name in your PATH. If the file you passed to **cat** via the command line (that is, **fork-file.c**) is in your current working directory, its contents will appear on your terminal screen. When that is the case, the program cat terminates successfully and exits with status zero. You can learn what was the terminations status of the last program you executed by inspecting a shell variable, as follows:

yourname@knuth2:~$ echo $?

When all goes well in your execution of **cat**, this **echo** command will show zero for termination status. Now, try to **cat** a file that doesn't exist:

yourname@knuth2:~$ /bin/cat bogus.nuthin

You don't have a file with that weird name, hopefully! See how you got a termination status different from zero? It was probably 1, in this case. Anyway, what this value in **$?** indicates is that something went wrong in the execution of the previous program. From now on, remember to use the termination status of a program to your advantage: make your processes use **exit** or have the main function in your programs **return** a non-zero value when things go so awfully wrong that you must abort them.

You will practice exactly that in this problem. And you will also learn to create processes that are not identical clones of their parent. This latter part means you will practice using a function from the **exec** family.

[2 points] a) Create a file called **catcount.c** in which you will write a program that receives one command line argument of type *string*: the name of a text file. Your program will work as follows:

- Start out by spawning a child process.
- The child process calls **execlp** to run **/bin/cat** with the command line argument that the parent received. Read the man page of this library call to learn what arguments to pass to it. This replaces the binary executable on the child process, but the parent goes on with the code you wrote.
- The parent process calls **wait** so that it blocks until the child terminates and passes back its termination status.
- If the child process terminates without error, the parent spawns another child and, again, calls **wait** so that it can block until the child terminates.
- The new child calls **execlp** again, but this time it runs **/usr/bin/wc** on the same argument that the parent received from the command line (the file name passed to **cat** previously).
- Once the parent learns that the child has terminated, it goes on to terminate also. If the parent gets to this point, it's because all has gone well, so its termination status should be 0.

We wrap up this assignment with a little bit of practice in working with a double pointer (that is, a *pointer-to-pointer* kind of variable). If you read the man page for **execlp**, you will notice in the SYNOPSIS section the definition of a variable that your programs will see when they **#include** the appropriate header file. The variable is defined in **unistd.h** as:

char **environ;

To use it in your program, you will need to define the variable as "extern", as indicated below:

extern char **environ;

The keyword **extern** tells the linker that this variable is defined in a separately compiled module (in a library, in this case). Because of C's duality between pointers and arrays, you can view **environ** as an *array of pointers*, where each element is a string that matches the following format:

KEYWORD=VALUE

For this problem, we don't care about the format of each individual string. We are interested in being able to write code to print all these strings to the terminal. To achieve this goal, we must traverse the array

from first to last element printing each string we find followed by the "\n" (newline) character. Since this is a variable size array, the pointer value **NULL** is used as sentinel to mark the end of the array.

[2 points] b) In your **catcount.c** program, create a function with the following prototype:

void print_environment(void);

Have your program call this function at the very start of **main**. The code of the function should be a simple loop to iterate through the array of strings **environ** up until its last element, printing to the terminal each of the elements it finds (again, follow each of these strings with a "\n").

When you have completed the problem, add the **catcount.c** file to your **Lab1a** directory for later submission. Remember to edit **Makefile** to accommodate the new program.

Hand In

As instructed throughout the lab create a **Lab1a** folder and submit there the following files: **Makefile**, **answers.txt**, **myprog.c**, **fork-wait.c**, **fork-file.c**, and **catcount.c**.

Before turning in your work for grading, create a text file in **Lab1a** called **submission.txt**. In this file, provide a list to indicate to the grader, problem by problem, if you completed the problem and whether it works to specification.

Finally zip your **Lab1a** folder and submit the compressed file to Brightspace for grading. Total of **16 points**.

Last updated 1.28.2022 by T. O'Neil. This lab was originally developed by Prof. L. Felipe Perrone (Bucknell) based on materials created by Prof. Phil Kearns for CSCI 315 at The College of William & Mary.