CS/SE/CE/TE 4348 and CS 5348: Operating Systems
Programming Assignment 1

Instructor: Ravi Prakash

Assigned on: June 11, 2015.
Due date: June 22, 2015.

This is an individual project and you are expected to do it on your own.
In this project you will:

1. Use *fork* system call to spawn two child processes from a parent process.
2. Use *execve* system call to replace the executable code for one of the child processes.
3. Use *pipe* system call to communicate between the processes.
4. Use *wait/waitpid* system call in the parent to make sure that the parent does not exit until all the children have finished execution.

Documentation

First, please learn how to use the man pages as that would help you a lot in understanding how fork and other system calls work. Man pages have a section dedicated to system calls. For example you can find more information about fork by doing the following:

```
man 2 fork
```

"2" in the above command refers to the man page section. As we are looking for documentation on system calls, we explicitly specify the man page section. This makes sure that only the system call documentation will be displayed. For some cases, such as read() system call, there is also a command called read, so the above command is helpful in getting documentation for the system call and not for the command.

Description

Part I: Copy-on-Write

You will start with one process and use the *fork* system call to create a child process. Remember that fork creates an exact copy of the parent process. The code that the parent and child execute are exactly the same. However, the child will only execute code that is reachable after the call to fork(). If the parent had some variables that were initialized to certain values before calling fork, the child also would have the exact same variables with the same values.

Linux kernel uses Copy-on-Write semantics for fork. This means that when fork is called, the new processs address space is actually a pointer to the parents address space. However, as soon as some change is made to a variable in the parent or child process, a new address space will be created for the child. So the copy is only performed when a write operation occurs in the parent or child, thus copy-on-write.

After fork() creates a new process, it will return in two different processes, the parent and the child. The return value of fork can be used to distinguish between the two possible scenarios. If you look at the man page of fork, it specifies that fork returns 0 in the child process and pid of the child process in the parent process.

Once you have a parent process and a child process (created using fork), you will use the child process to demonstrate the effects of copy-on-write fork semantics. You will initialize a variable called *cow* to a value of 10. Please make sure that this is done before calling fork(). This is because whatever is present in a processs address space before fork will be considered by fork.

`pipe` is a way to communicate between two processes. You will use the pipe system call to communicate between the parent process and the created child. Again, please take a look at the man page of pipe (man 2 pipe) for more documentation. Basically, the pipe system call expects an integer array of size 2 as an argument. Once pipe system call is executed, the first and second values of the array are filled with valid descriptors for the pipe. This means that if you were to write some data to the descriptor stored in the first array element, you should be able to read the same data from the second descriptor stored in the last element of the array. Let us take a look at an example

```
int    pfd[2];
int    pid;

pipe(pfd);
pid = fork()
if   (pid == 0)
  {
   write(pfd[0], "hi there", 9);
  }
else
  {
   read(pfd[1], buffer, BUFSZ);
  }
```

Note that, in the above example, we invoke the pipe system call before calling fork. This makes sure that fork will copy the pfd variable to the child process, hence the child process would be able to use it directly.

Again, please make it a habit to lookup the man pages for system calls you want to explore. In the above example, I have used read() and write(), so you can find more about them in the man pages.

After calling fork, do the following

1. Display the value of the variable cow in the child and the parent process.
2. Verify that the values are the same.
3. Write a message to the pipe in the child process and read that message in the parent.
4. Assign 1000 to cow in one of the processes (parent or child) and display the value of cow in both the processes.
5. Verify that both values are different.

Part II: execve

In this part, you will create another child process by using fork() again. However, unlike the previous part, you will replace the executable code of the created child by using the execve system call.

While fork() copies the executable code from the parent to the child, execve can be used to replace the executable code of a process after fork()ing. Again, please take a look at the man page of execve. One of the arguments to this system call is the path of the executable that contains the code to be executed. This code would be loaded in the code section of the process that calls execve, thus replacing the executable code that the process would run. For the second child, you will use *execve* to replace the executable code of the child process with that of the 'ls' command. You can find the path to the 'ls' executable by doing the following:

```
> which ls
> /bin/ls
```

Also, note that the 'ls' system command optionally takes an argument, give the path of a directory to list the files in it.

execve can be called as soon as the execution of the second child begins, which is after the fork system call. Note that execve completely replaces the executable code of the calling process, which means that replacing the second

childs code with ls will generate the contents of the directory from which the child started execution, which should be the same as the parent.

Once ls is executed, it will generate a listing of the contents of the current directory.

Termination

Finally, you have to make sure that the parent process does not terminate before the two child processes have finished execution. You can achieve that using the wait system call.

Error handling

Please make sure that you check the return value of all functions. You can find details about the return value of a system call or a function in its man page. In case of an error, your code should display an error message. This can be achieved using the C function perror . You will need to include the header files stdio.h and errno.h. If the error makes further execution meaning less, for example, if pipe returns an error that means you will not be able to communicate between process. You can then display an error message using perror (note that perror will display the message for you, because it reads a global variable errno for error information) and exit with an appropriate code such as EXIT FAILURE. An example of this is the following:

```
if   (pipe(pfd)        <  0)
 {
    /*  Prefix the error messages related to pipe system call with "pipe:" */
   perror("pipe");
   exit(EXIT_FAILURE);   /* Need to include stdlib.h for EXIT_FAILURE  */
 }
```

Submission Information

The submission should be through eLearning in the form of an archive consisting of:

1. File(s) containing the source code.
2. The makefile.

Please do “make clean” before submitting the contents of your directory. This will remove the executable and object code that is not needed for submission.

Your source code must have the following, otherwise you will lose points:

1. Proper comments indicating what is being done
2. Error checking for all function and system calls