# Process Control

This is an individual homework, not a group effort. This homework has four parts.

## Part 1 – Using fork() and exec()

/  
This part should be relatively straight forward. Following the numerous templates, I’ve already give you, simple write a program that calls fork(), the child will execute the “/bin/ls” program given a command line argument of “-l /usr/bin”. The parent will wait for this to finish, and then print the return code, and then exit.

If the exec() function fails, the child should print a message “Error during execution\n”;

This executable should be called “part1”.

### Sample Output

(not my /usr/bin directory)

$ ./part1  
total 28  
-rw-r--r-- 1 tbriggs tbriggs 49 Jan 28 14:30 Makefile  
-rwxr-xr-x 1 tbriggs tbriggs 17040 Jan 28 14:31 part1  
-rw-r--r-- 1 tbriggs tbriggs 597 Jan 28 14:31 part1.c  
Child exited: 0

## Part 2 – Handling SIGCHLD

Whenever a child dies, the parent process is notified with a “SIGCHLD” message. Use the “signal()” system call to register a function to handle the death of a child by printing a message, which includes the input argument to the signal function:

\*\*\* CHILD DIED 33\*\*\*

To be clear, the signal() function expects a function that matches the following definition:

typedef void (\*sighandler\_t)(int);

This is a C notation for a “function pointer” to a void function that takes a single integer argument. You can get the “address of a function” by simply using the name of the function.

Spawn the “ls” program as in part 1. This time, the message should appear before the parent exits.

This executable should be called “part2”

### Sample Output

(still not my /usr/bin directory)

$ ./part2  
total 52  
-rw-r--r-- 1 tbriggs tbriggs 49 Jan 28 14:30 Makefile  
-rwxr-xr-x 1 tbriggs tbriggs 17120 Jan 28 15:36 part2  
-rwxr-xr-x 1 tbriggs tbriggs 17040 Jan 28 14:31 part1  
-rw-r--r-- 1 tbriggs tbriggs 597 Jan 28 14:31 part1.c  
-rw-r--r-- 1 tbriggs tbriggs 724 Jan 28 15:36 part2.c  
\*\* CHILD DIED 17\*\*  
Child exited: 0

## Part 3 – Handling Child’s Standard Output

Whenever a child is spawned, it is connected to the same standard output (c.f. part 1, the output of ls came to your terminal). But, we can “fix” that. Start with a copy of part 1’s source code. In the child’s logic, before calling exec():

1. use low-level “close()” function to close the file id for stdout
2. use the low-level “open()” function to open a new file called “output.txt” for writing, and create / truncate if it doesn’t exit / does exit.

Then, do the execute(). Meanwhile, the parent will wait() for the child, then exit. Finally, check the contents of the file, it should be the output of ls.

Note, if you read the manual page, you’ll see that the open function *can* have 3 arguments, the third being the permissions. Use the number “0777” for the permissions that it will use to create the new file. If you don’t, it will be created without any permissions and you won’t be able to read it.

### Sample Output

(and yes, this is not my /usr/bin directory).

$ rm -f output.txt

$ ./part3  
Child exited: 0  
$ cat output.txt  
total 76  
-rw-r--r-- 1 tbriggs tbriggs 49 Jan 28 14:30 Makefile  
-rwxr-xr-x 1 tbriggs tbriggs 17120 Jan 28 15:36 a.out  
-rwxr-xr-x 1 tbriggs tbriggs 0 Jan 28 15:42 output.txt  
-rwxr-xr-x 1 tbriggs tbriggs 17040 Jan 28 14:31 part1  
-rw-r--r-- 1 tbriggs tbriggs 597 Jan 28 14:31 part1.c  
-rw-r--r-- 1 tbriggs tbriggs 724 Jan 28 15:36 part2.c  
-rwxr-xr-x 1 tbriggs tbriggs 17128 Jan 28 15:42 part3  
-rw-r--r-- 1 tbriggs tbriggs 733 Jan 28 15:42 part3.c

### Sample Output

## Part 4 – Handling Standard Input

As in part 3, we start with the code in part 1 as a template. This time, however, **before** calling fork(), the parent will create a pair of **ordinary pipes**. Read the documentation about this step, its well documented. Then, the parent will fork() (as before).

In the child, **before** exec()’ing, we **close()** stdin (similar, yet different, to what we just did in part 3). Then, **after that**, but **before exec**, you will need to use the “dup()” or “dup2()” system call to duplicate the *read end* of the ordinary pipe, this will connect the standard input of the child process to the pipe. One other change in the child is to run the “cat” program instead of “ls”.

Meanwhile, the parent can write messages to the pipe. Since strings are just bytes, we can “print” to the pipe by first using “sprintf()” and the using “write()” to the *write side* of the pipe.

The child (the “/usr/bin/cat”) program will continue reading from the pipes as long as *any* writers are still open. Because the pipes were created **before** the fork(), there are **two** writers – the parent and the child. Therefore, before calling exec(), the child should close its copy of the write end of the pipe. Otherwise, the “cat” command will continue waiting for itself to write data….

Your main logic will send three strings to the pipe, each will be terminated by new-line:

“hello world\n”

“Watson, come quickly\n”

“it worked!\n”

Then, your main logic will close() the write side of the pipe, and then wait for the child to terminate.

$ ./part4  
hello word  
Watson, come quickly  
it worked!  
Child exited: 0

# Deliverables

Zip up the “Makefile” and all of the C files, and submit them to the Autograder in Gradescope for this problem. Of course, you can also use GitHub and use that to post your solutions too!

As before, I will be grading your source code as well:

* Style / Readability
* Proper use of system calls – including relevant error handling
* Proper functional decomposition