CSC 4730 Fall 2023 Project 1

Fork, exec, pipes and redirection

Objectives

You will master several key operating systems services. These are:

service	short description
fork	how new processes are created
exec	how new programs are loaded
pipe	a means of interprocess communication
redirection	how standard input and output(s) can be manipulated

Deadlines

You and your assigned partner have 7 days to complete this assignment plus one grace day. If you have not turned anything in by 11:59 PM on the eighth day, you will win a grade of zero. Therefore, to receive partial credit, hand in something before the project expires!

Overview

You use the shell *all the time*. It may seem special, but it isn't. It's a user program just like all of yours. It's purpose is to accept commands from you and, interpreting these, launch other programs.

Rather than build a simple shell utilizing a shell-like command syntax, you will use command line options. For example:

```
$ myshell -t /bin -1 ls -2 wc -o /tmp/output.txt
```

This command line will run the program `ls` piping its output to `wc` which writes its standard output into /tmp/output.txt. The directory in which these commands should execute is /bin.

In a real shell the above command line would look like this:

```
$ pushd /bin; ls | wc > /tmp/output.txt; popd
```

Coding this project using real shell syntax would mean many hours of your time would be devoted to parsing the shell syntax and **not** learning operating system concepts.

Again, do not be put off by the command line options in this program - this method of controlling your simple shell is way easier than parsing more realistic command lines.

Review of what's underneath `cin`, `cout` and `cerr`

`cin` is a wrapper for `STDIN`.

`cout` is a wrapper for `STDOUT`.

`cerr` is a wrapper `STDERR`.

The underlying input / output paths are simply the first three entries in a process's *open file* table.

Index 0 is `STDIN`.

Index 1 is `STDOUT`.

Index 2 is `STDERR`.

The table size is at least 20 long in a typical full featured Unix-like OS. Being an array that you do not have direct access to, you specify where input / output goes using file descriptors - a fancy name for the indices of the array keeping track of open files and file-like devices.

Review of indirection

The shell, and by extension *every* program, reads and writes via the open file table. All they know is that they're writing to index 1 or 2 and reading from index 0. The program almost never knows and doesn't care what's on the other end.

So:

```
$ ls
```

Writes its output to the console. That's where index 1 goes by default.

But:

```
$ ls | wc
```

`ls` is **still** writing to index 1. But this time, index 1 of `ls` is hooked up to index 0 of `wc`. `wc` reads from index 0 just like always - it has no idea that it isn't hooked up to the keyboard.

And:

```
$ ls > foo.txt
```

Once again, `ls` writes to index 1. It has no idea that index 1 is ultimately fill in data in a file.

Let's consider this line:

```
$ ls > foo.txt 2>&1
```

`ls` will still write to index 1 but index 1 will be routed to a file. Index 2 is going to be routed to the same place as index 1.

System calls `dup` and `dup2`

It is possible to copy the contents of open file table cells indicated by a file descriptor to other locations within the open file table. This is the job of the `dup()` and `dup2()` system calls. You will need to use one of these (depending how you choose to code things up).

Using `xv6` to Learn More Deeply

Let's drill down in the `xv6` source code to see how `dup()` is implemented. **You should do this for `fork()` and `exec()`**. I have already modelled how to learn other people's code here.

Starting with `sys_dup()` in `sysfile.c` you'll see it calls `argfd()` which fetches a file descriptor from user land, vets it, then returns the file descriptor AND a pointer to the cell in the open file table corresponding to it.

Then, it uses `fdalloc()` to locate an empty cell in the open file table. Notice `fdalloc()` starts at index 0 and marches forward. After assigning the cell to point to the open file, `filedup()` updates a reference count.

`fork()`

The system call `fork()` is very special. One process calls the function. Two processes return from the function. That is, `fork()` is how new processes are made. Note we're *not* talking about threads - but processes. Both the parent process and the child process are identical with the exception of the return value from `fork()`.

In the child, the return value of `fork()` is 0.

In the parent, the return value of `fork()` is the process ID of the child.

`exec()`

The `exec()` *family* of system calls are also special. These calls cause the operating system to locate an executable file on disk, load its contents into memory **OVERLAYING** the memory of the original process. Then, the OS launches the entry point (i.e. `main()`) of the new code.

Relationship between `fork()` and `exec()`

If P1 forks, you get two P1's. If P1 keeps forking itself, all you'll have are P1's. How do you ever get to run a program other than P1?

`exec()` works hand-in-hand with `fork()`.

- P1 `forks()`
- Now there are two P1's

- The parent P1 knows it is the parent because it gets a positive non-zero return code from `fork()`
- The child P1 knows it is the child because it gets a zero return code from `fork()`
- The child P1 potentially does some bookkeeping then calls `exec()` for P2
- `exec()`, if it works, does not return - rather P2 begins executing

`fork()` and calling `exec()` in this project

Your program will, based upon various options, set up certain file descriptors to non-standard values (for example to support redirection) and may be required to create a pipe (if 2 programs are specified). Your program will have to perform at least one fork - maybe two. Your program will have to do various plumbing tasks to make sure the right outputs go to the right inputs. Your program will have to call `exec()` one or two times, depending on options.

Two of the ancient *pptx* slide decks might be helpful here.

- `one`
- `two`

Helpful videos

Here is a previous year's lecture on youtube.

- 9/8/2020

Here is a video specifically on setting up pipes:

- 9/11/2020

Here is a video specifically about file descriptors:

- FD

Required command line options

Command line options can occur in any order. You must implement the following:

- `-t` followed by a path to a directory - the command or sequence of commands should start in the named directory. It is an error if the directory does not exist or the path does not lead to a directory. `t` is for `t`emporary directory.
- `-i` followed by a path to a file - this file will be wired into the standard input of the first program in the sequence. This is optional. If not given, the standard input is left alone. It is an error if the file path is not valid or the file fails to open. This is the real shell equivalent of '<'.

- `-o` followed by a path to a file - this file will be wired into the standard output of whatever program is last in the sequence. This is optional. If not given, the standard output is left alone. It is an error if the file path is not valid or the file fails to open. The file is opened in overwrite mode. Be careful. This is the real shell equivalent of '>'.
- `-a` followed by a path to a file - this file will be wired into the standard output of whatever program is last in the sequence. This is optional. If not given, the standard output is left alone. It is an error if the file path is not valid or the file fails to open. This differs from -o in that the file is to be opened in append mode. This is the real shell equivalent of '»'.

NOTE: if both -o and -a are given, whichever comes last takes precedence.

- `-1` followed by a path of a program to run. This **must** be provided. It is an error if the path does not lead to an executable program.
- `-2` followed by a path of a program to run. This is optional. If given, it receives its input from a pipe from the preceding program. It is an error if the path does not lead to an executable program.
- `-D` this command line option does not take an argument. It is optional. If given, it prints the current working directory. See the getcwd function.

Optional command line options

- `-v` this command line option does not take an argument. It is optional. If given, it turns on any additional debugging print out you might want. I will not test this and you are not required to implement it. This is for your use.

Mode of operation

If more than one executable program is specified, your program must use the `pipe` system call to wire together standard out of the first program to the standard in of the second.

Your program must use fork and some variety of exec (specifically execvp) appropriately either one or two times.

Your program must correctly wire up the standard input of the first executable program if requested.

Your program must correctly wire up the standard output of the last executable program if requested.

Your program must wait for the programs you've run to exit and print their return codes.

Sample output

```
$ ./a.out
Missing required command line option -1
$

$ ./a.out -1 ls
a.out  main.cpp  main.d  main.o  makefile  README.md  wkspc.code-workspace
Child 1: 12191 returns: 0
$

$./a.out -1 ls -2 wc
      7       7      69
Child 2: 12194 returns: 0
Child 1: 12193 returns: 0
$

$ ./a.out -i main.cpp -1 wc
 196  653 4860
Child 1: 12252 returns: 0
$

$ # Notice difference between -a and -o.
$ # The -o and -a options are dangerous. Who among you
$ # will be first to lose their source code? We'll see!
$
$ ./a.out -i main.cpp -1 wc -2 wc -o danger.txt
Child 2: 12656 returns: 0
Child 1: 12655 returns: 0
$ cat danger.txt
      1       3      15
$ ./a.out -i main.cpp -1 wc -2 wc -a danger.txt
Child 2: 12672 returns: 0
Child 1: 12671 returns: 0
$ cat danger.txt
      1       3      15
      1       3      15
$ ./a.out -i main.cpp -1 wc -2 wc -o danger.txt
Child 2: 12686 returns: 0
Child 1: 12685 returns: 0
$ cat danger.txt
      1       3      15
$
$ ./a.out -1 ls
a.out  danger.txt  main.cpp  main.d  main.o  makefile  README.md  wkspc.code-workspace
Child 1: 12755 returns: 0
```

```
$ ./a.out -1 ls -t ~
courses  Desktop  Documents  Downloads  Music  Pictures  Public  src  Templates  Videos
Child 1: 12758 returns: 0
$

## Some errors

$ # some errors
$ ./a.out -1 doesnotexist
Prog 1 exec failed.
Child 1: 12760 returns: 1
$ ./a.out -i doesnotexist -1 wc
Could not open redirected input file: No such file or directory
Child 1: 12762 returns: 1
$
```

Nothing in this specification says you must pass custom command line arguments to either program 1 or program 2. You are not to do so, however you must correctly pass a minimal argc and argv.

Hint

In a standard shell, execute

```
sleep 1 | sleep 1 | sleep 1 | sleep 1 | ps -l
```

(the last command is ps dash lowercase L).

Examine the process ID columns and parent process ID columns. This may suggest to you how to structure your code governing who forks who.

To put the potato on the fork, this is wrong: parent forks child 1 - child 1 forks child 2.

Another Hint

You'll need to use at least these system calls:

```
fork
execvp
pipe
dup and / or dup2
close
open
getcwd
```

You will need to use:

```
getopt
```

One more hint

- Start implementing the `-1` option.
- Then, I would implement the `-i` option.
- Next, the `-o` and `-a` options - watch out that you don't blow away something important when testing these options.
- Finally, implement the `-2` option - here is where you will need pipes and some more forking around.

What to Hand In

Zip your makefile and all of your .c and / or .cpp source files and any header files into a single zip file containing NO sub-folders. If your zip file contains a subfolder, I will remove 5 points.

Partners

- You must use only the partner I assign you.
- Only 1 person should hand in code. The code should clearly state who the partners are in a comment at the top of the main source file.
- The non-code-submitting partner must submit a text file "partner.txt" that states who the partners are.
- Failure to list partners correctly as described above removes 5 points from your grade.

Software Kills

So you forgot one step. So what? Suppose your implementation sits in a chain storing or processing log information. There's something of vital importance to the automated driving system you're working on that some other programmer screwed up. But nobody notices because log entries that would have told engineers what went wrong wasn't written. People die in wreck after wreck. You didn't write the code that caused the fault that caused the deaths, but you're responsible for dead people non-the-less.

Grading

The class will receive a rubric when grading is complete that describes errors and penalties.

Both partners get the same grade without exception.