Homework 2: Shell Implementation

Due Date: March 22nd, 2019 at 11:55 pm

Academic Honesty

Aside from the narrow exception for collaboration on homework, all work submitted in this course must be your own. Cheating and plagiarism will not be tolerated. If you have any questions about a specific case, please ask the instructor or the TAs. Also, we will be testing for plagiarism.

NYU's Policy on Academic Misconduct:
http://engineering.nyu.edu/academics/code-of-conduct/academic-misconduct

Homework Notes

General Notes

- Read the assignment carefully, including what files to include.
- Don't assume limitations unless they are explicitly stated.
- Treat provided examples as just that, not exhaustive list of cases that should work.
- When in doubt regarding what needs to be done, ask. Another option is test it in the real UNIX operating system. Does it behaves the same way?
- **TEST** your solutions, make sure they work. It's obvious when you didn't test the code.

Notes

- This assignment does not involve modifying or using xv6 (although the code for Shell.c is adapted from the xv6 shell).
- You should write, compile, and test your code on macOS or Linux, rather than in QEMU.
- While it may be tempting to just copy xv6's implementation, there are enough differences

between the xv6 APIs and those in Linux/macOS that doing so would be a bad idea. You

can look at how it works for inspiration though.

Rubric

Here are **some** of the things we know we will test, but these are not the **only** things we will test. Therefore make sure to test your program thoroughly and thoughtfully.

Total: 100 points

- Program does not compile (-90)
- stdin redirection not working (-30)
- stdout redirection not working (-30)
- pipe not working (-40)
- exec not working (-30)
- bad permissions on redirected output (-10)
- "not implemented" messages left in (-5)

In this assignment, you will implement pieces of a UNIX shell, and get some familiarity with some UNIX library calls and the UNIX process model. By the end of the assignment, you will have a shell that can run complex pipelines of commands, such as:

```
$ cat /usr/share/dict/words | grep cat | sed s/cat/dog/ >
doggerel.txt
```

The above pipeline takes `/usr/share/dict/words` (a file generally installed on UNIX systems that contains a list of English words), selects out the words containing the string "cat", and then uses sed to replace "cat" with "dog", so that, for example, "concatenate" becomes "condogenate". The results are output to "doggerel.txt". (You can find detailed descriptions of each of the commands in the pipeline by consulting the

manual page for the command; e.g.: "man grep" or "man sed".)

Start by downloading the `shell.c` skeleton file attached to this homework. You don't have to understand how the parser works in detail, but you should have a general idea of how the flow of control works in the program. You will also see the `"// your code here"` comments, which is where you will implement the functionality to make the shell actually work.

Next, try to compile the source code to the shell:

```
$ gcc shell.c -o shell
```

You can then run and interact with the shell by typing `./shell :`

```
user@cs6233:~$ ./shell
cs6233> ls
exec not implemented
cs6233>
```

Note: The command prompt for our shell is set to `cs6233>` to make it easy to tell the difference between it and the Linux/macOS shell. You can quit your shell by typing Control-C or Control-D.

Problem 1: Command Execution (30 points)

Implement basic command execution by filling in the code inside of the `case''` blockin the `runcmd` function. You will want to look at the manual page for the exec(3) function by typing

"man 3 exec" (Note: throughout this course, when referring to commands that one can look up in the man pages, we will typically specify the section number in parentheses -- thus, since exec

is found in section 3, we will say exec(3)).

Once this is done, you should be able to use your shell to run single commands, such as

```
cs6233> ls
cs6233> grep cat
/usr/share/dict/words
```

Hint:

1. You will notice that there are many variants on exec(3). You should read through the differences between them, and then choose the one that allows you to run the commands above -- in particular, pay attention to whether the version of exec you're using requires you to enter in the full path to the program, or whether it will search the directories in the PATH environment variable.

Problem 2: I/O Redirection (30 points)

Now extend the shell to handle input and output redirection. Programs will be expecting their input on standard input and will write to standard output, so you will have to open the file and then replace standard input or output with that file. As before, the parser already recognizes the '>' and '<' characters and builds a `redircmd` structure for you, so you just need to use the information in that `redircmd` to open a file and replace standard input or output with it.

Hints:

1. Look at the dup2(2) and open(2) calls.
2. The file descriptor the program is currently using for input or output is available in

   `rcmd->fd`

.

3. If you're confused about where `rcmd->fd` is coming from, look at the `redircmd` function and remember that 0 is standard input, 1 is standard output.
4. Be careful with the `open` call; in particular, make sure you read about the case when you pass the `O_CREAT` flag.

When this is done, you should be able to redirect the input and output of commands:

```
cs6233> ls > a.txt
cs6233> sort -r < a.txt
```

Problem 3: Pipes (40 points)

The final task is to add the ability to pipe the output of one command into the input of another.

You will fill out the code for the '|' case of the switch statement in runcmd to do this.

Hints:

1. The parser provides the left command in `pcmd->left` and the right command in `pcmd->right`.
2. Look at the fork(2), pipe(2), close(2) and wait(2) calls.
3. If your program just hangs, it may help to know that reads to pipes with no data will block until all file descriptors referencing the pipe are closed.
4. Note that fork(2) creates an exact copy of the current process. The two process share any file descriptors that were open at the time the fork occurred. You can get a sense for this behavior by looking at a small test program like this one:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include
<sys/stat.h>

int main() {
int filedes;
filedes = open("myfile.txt", O_RDWR | O_CREAT, S_IRUSR |
S_IWUSR);
int rv;
rv = fork();
if (rv == 0) {
     char msg[] = "Process 1\n";
     printf("Hello, I'm in the child, my process ID is
%d\n", getpid());
       write(filedes, msg, sizeof(msg));
     }

    else {

    char msg[] = "Process 2\n";
    printf("This is the parent process, my process ID is %d
and my child is %d\n", getpid(), rv);
    write(filedes, msg, sizeof(msg));
     }

    close(filedes);
 }
```

If you put that code into a file, compile it, and then run the resulting program, you should see a result like:

```
user@cs6233:~$ ./a.out

This is the parent process, my process ID is 56968 and my
child is 56969 Hello, I'm in the child, my process ID is 56969
user@cs6233:~$ cat myfile.txt

Process 2

Process 1
```

You can see that both the parent and child process both got a copy of "filedes", and that writes to it from each process went to the same underlying file.

5. You may find it helpful to re-read the first chapter of the xv6 book, which describes in detail how the xv6 shell works. Note that the code show there will not work as-is -- you will have to adapt it for the Linux/macOS environment.

Once this is done, you should be able to run a full pipeline:

```
cs6233> cat /usr/share/dict/words | grep cat | sed s/cat/dog/
> doggerel.txt
cs6233> grep con < doggerel.txt
```

You can now submit your modified `shell.c` on NYU Classes.

Credits: This assignment is adapted from a homework by Brendan Dolan-Gavitt