

Released: 1st Oct, 2023
Due: 15th Oct, 2023, 11:59 PM EST

CS-GY 6233 - Fall 2023

Homework #2

Linux Shell Implementation

- | | |
|----------------------|----------|
| 1. Command Execution | [10 pts] |
| 2. I/O Redirection | [30pts] |
| 3. Pipes | [30pts] |
| 4. Summary Questions | [30pts] |
-

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 an exhaustive list of cases that should work.
- When in doubt regarding what needs to be done, ask. Another option is to test it in the real UNIX operating system. Does it behave the same way?
- **TEST your solutions; make sure they work. It's obvious when you didn't test the code.**
- Remember if you are working with a partner, submit your name and NetID# along with your partner's name and NetID# along with your other documents on Brightspace

Assignment Notes

- This assignment does not involve modifying or using xv6 (although the code for Shell.c is adapted from the xv6 shell).
- You must write, compile, and test your code on the Unix shell in your Docker Linux Image (this code won't be running in xv6 this time).
- Because we're not running shell.c in xv6, you are free to use the standard C libraries such as `stdlib.h`, `string.h`, `stdio.h`, etc.
- While it may be tempting to just copy xv6's implementation, there are enough differences between the xv6 APIs and those in Unix OS that doing so would be a bad idea. You can look at how it works for inspiration though.

Overview

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

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

The pipeline shown above takes a text document labeled "*words.txt*" (a file generally installed on UNIX systems that contains a list of English words.), select the words containing the string "cat", and then use *sed* to replace "cat" with "dog", so that, for example, "concatenate" becomes "condogenate". The results are then output to a text file labeled "*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 copying the "*shell.c*" and "*words.txt*" in your docker (unix) environment. This [link](#) provides the syntax for docker copy command. Alternatively, you can also simply boot up your container in VS Code (as instructed in Hello World instructions) and simply create "*shell.c*" and "*words.txt*" in VS Code (and copy the contents). 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 comments labeled with "*//your code here*", which is where you will implement the functionality to make the shell actually work.

Next, you must try to compile the source code to the Unix shell (Again, this will not be using xv6):

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

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

```
:~$ ./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 our shell and the Linux shell. You can quit your shell by typing Ctrl-C or Ctrl-D.

Part 1 - Command Execution (10 Points)

Implement basic command execution by filling in the code inside of the `case` ' ' blocking 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 words.txt
```

Hint: You will notice that there are many variants on `exec(3)`. You should read through the differences between them in the linux manual, 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.

Make sure to explain your program or approach in a few sentences and submit your explanation in a text document on Brightspace (5 points).

Part 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 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 `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
```

Make sure to explain your program or approach in a few sentences and submit your explanation in a text document on Brightspace (5 points)

Part 3 - Pipes (30 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)`, `dup2(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 processes share any file descriptors that were open at the time the fork occurred. You can get a sense for this:

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
int main() {
    int filedес;
    filedес = 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);
}

```

5. If you put that code into a separate file on your system, compile it, and then run the resulting program (as follows), you should see a result like:

```

gcc a.c -o a.out
./a.out

```

This is the parent process, my process ID is *[Random Number X]* and my child is *[Random Number X + 1]*

Hello, I'm in the child, my process ID is *[Random Number X + 1]*

```

cat myfile.txt
Process 2
Process 1

```

6. 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.
7. You may find it helpful to re-read the first chapter of the xv6 manual, which describes in detail how the xv6 shell works. Note that the code shown there will not work as-is -- you will have to adapt it for the Unix environment.
8. Once this is done, you should be able to run a full pipeline:


```

cs6233> cat words.txt | grep cat | sed s/cat/dog/ > doggerel.txt
cs6233> grep con < doggerel.txt

```

Explain your program or approach in a few sentences (10 points).

Part 4 - Summary Questions (30 Points)

1. (10 points) How do we implement the following list of commands successively? How do you ensure that commands are executed regardless of whether each previous command succeeds, and how do you implement them? Also, what is another method to warrant that a command executes only if the previous command succeeds?
 - a. `(cd /XXXdirectory)`
 - b. `(find . -name 'a*' -exec rm {} \)` [removes all files starting with filename a in XXXdirectory]
 - c. `(ls -la)` [lists files in the current directory "XXXdirectory" after removing]
2. (10 points) How would you implement sub shells by implementing "(" and ")".
For example,

- a. `echo "-$(sed 's/cat/lion/g' a.txt)" > hello.txt`
- i. Purpose: First, replace all occurrences of cat with lion in file "a.txt" and echo into hello.txt file.
3. (10 points) How would you implement running commands in the background by supporting "&" and "wait"?

You must submit your modified shell.c in Brightspace. In addition, you must also submit a pdf file to the Brightspace assignment containing your written answers for part 4 and a text file for explaining your approach for part 1, 2 and 3. In total, you will be submitting i)shell.c ii)explanations.txt iii)answers.pdf and iv)partner.txt

Rubric

Part	Description	Deduction
Part 1, 2, and 3	Program does not compile	-90pts
Part 2	Stdin redirection does not work Stdout redirection does not work	-30pts for Stdin redirection -30pts for Stdout redirection
Part 3	Pipe does not work	-40pts for pipe functionality
Part 3	Exec does not work	-30pts for Exec functionality
Part 1, 2, and 3	Bad permissions on redirected output	-10pts
Part 1, 2, and 3	"Not implemented" messages left in	-5pts
Part 1	Missing your written explanation on implementing shell command executions	-5pts
Part 2	Missing your written explanation on implemented shell output redirection	-5pts
Part 3	Missing your written explanation on implemented shell pipe redirection	-5pts
Part 4	Missing your written answers regarding how you would implement shell's unique parameters	-10pts for each of the three questions

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 Prof/TAs. We will be checking for this!

NYU Tandon's Policy on Academic Misconduct:

<http://engineering.nyu.edu/academics/code-of-conduct/academic-misconduct>

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