Project 1-UNIX Shell

Description and Objectives

In this project, you will write your own UNIX shell that behaves similar to the tcsh shell in linprog. The shell is the part of the OS that processes commands. You will learn how to tokenize strings, create and run built-in commands, run external commands, use file redirection, and expand environment variables.

Details

You will create a shell similar to the tcsh shell in linprog. The tcsh shell is the default shell in linprog. Your shell will be much simpler, but will behave similarly. Your program will essentially execute the following steps:

(1) Print a prompt to the user, waiting for them to type a command and press enter.

(2) Parse the command, ultimately forming the command's arguments.

(3) Execute the command.

(4) Repeat step (1) until the user types the exit command.

Let's go over each of these.

For (1), you will print a prompt with the format [USER]@myshell:[CWD]>, where [USER] represents the user's username and [CWD] represents the current working directory. For example, porter@myshell:/bin>. You will then wait for the user to type a command and press enter. You can assume the command will be at most 80 characters and there will be at most 9 command arguments (including the command name itself). Please make your prompt print with at least one leading newline character. This will help automate testing.

Once you have the command, it's time to do step (2). You will take the command and break it into tokens. For our simplified shell, you will use whitespace as the delimiter. That is, you will assume each token is separated by whitespace. You will then go through each token and do some processing, ultimately forming the arguments of your command.

Next, you will go to step (3) and run the command. There are two types of commands: built-in commands and external commands. Built-in commands are those commands you create. External commands are basically programs that already exist that you simply call on.

Finally, step (4) loops back around to step (1) unless the user chose to exit.

Built-in Commands

Your shell will support three built-in commands:

cd - this command will change the CWD. It will take zero or one argument. If given zero arguments, assume the user wants to go back to their home directory. If given one argument, attempt to change to the directory specified. If you aren't able to change to the directory specified, print an error. Also, if the user provides more than one argument, print an error. Note

that if a new directory is switched to, your prompt will change to reflect that this new directory is your CWD.

echo - this command will print back to the screen what the user types. It takes zero or more arguments. For example, typing echo 1 2 3 will print 1 2 3.

exit - this command will exit your shell.

External Commands

Your shell will be able to execute external commands using the execv() function. External commands are commands like ls, cat, cp, mv, mkdir, etc. that already exist for you to call on. You are not allowed to use any other function to execute external commands. The execv() function requires a command path and arguments. For example, if you type ls -a, both ls and -a are arguments that you will send as the second argument to execv(). However, execv() will need the path to the command for its first argument. If the first argument of your command (i.e. the command name itself) contains a / character, you will assume that you were given the path to the command. Otherwise, you will search the paths listed in the PATH environment variable until you find a location that contains the external command. You will then prepend this path on to your command and pass it as the first argument to execv(). For example, ls might be located in the /bin directory. So, you'll pass "/bin/ls" to execv() for its first argument. In any event, if you can't find the command, print an error.

File Redirection

Your shell will support the file redirection operators < and >. Each of these operators expects a file afterwards. You can assume one will be given (but not necessarily one that exists). You can assume at most one < and at most one > in any one command. You can also assume that redirection will only be used with external commands, not built-in commands. Note that neither the redirection operators nor their respective files are part of a command's arguments. For example, if the user types ls -a > files.txt, you won't send the > and files.txt to the execv() function. Instead, as future notes will show, redirection must be handled by you.

Environment Variables

Your shell will support environment variables. The shell detects an environment variable if something has a $ in front of it. For example, $HOME corresponds to your home directory stored in environment variable HOME. You will have to expand these variables before running commands. That is, you will replace these variables with their values. For example, if I'm running my shell from my own account and I type ls $HOME, I will first replace $HOME with /home/grads/porter before executing the command. Note that /home/grads/porter isn't something I would hard code. In other words, don't hard code values for any environment variable. Instead, get their value by using the proper library function. You can assume whitespace around each environment variable. For example, you won't expect to see $HOME$HOME.

Reading and Parsing Commands

The first step in creating your shell is to read and parse the command entered. You will find the fgets() and strtok() functions useful in reading and parsing commands, respectively. Let's look at an example in tokenize.c. This program will continuously ask for a command to be entered. The command is then read using fgets(). fgets() will read characters into a buffer until a newline is encountered or until 80 characters are encountered. Notice I gave it a buffer with 81 slots. This is because you want to reserve

1 slot for the terminating null character. Note that fgets() doesn't get rid of the newline character.

Now, we have our command as entered. However, we need a break it out into individual components separated by whitespace. This is where strtok() comes in. strtok() essentially separates the command based on delimiters. In this case, our delimiters are any of the whitespace characters. This string of delimiters gets passed to strtok() so it knows what to divide on. The function also takes care of both leading and trailing whitespace in my example, something else we want. You'll notice the first time around you pass it the actual string. However, on successive times, you pass it NULL. It will return NULL when it has nothing more to process. Here is an example output:

```
Enter command: ls -a > file.txt

This is the command as entered: ls -a > file.txt


This is the command parsed into whitespace-delimited tokens (one per line):

ls
-a
>
file.txt
```

So, you are now able to read in commands and break them into whitespace-delimited tokens. However, a few questions remain. First, how do you print your username and the current working directory in the command prompt? Your username is stored in the environment variable USER. To get the value of this variable, you use the getenv() function. You pass this function the name of the variable who's value you're looking for and it returns the value. Note that the $ the user types in front of the environment variable is only to indicate an environment variable to your shell. Without it, your shell would think USER had no special meaning. Hence, you will only pass "USER" to this function, not "$USER". As for the CWD, you use another function called getcwd(). I will leave it up to you to experiment with these functions. Note that fgets(), getenv(), and getcwd() all have ways of indicating errors. So, you might want to check for these errors before proceeding.

So, you now know how to read in commands, break them into whitespace-delimited tokens, expand environment variables, and get the current working directory. You're ready to execute your command, right? Not quite. Once you have your command separated into tokens, you will need to go through and process each token and add it to your list of arguments. So, what I recommend is that you create a command structure to pass around to various functions. Inside the structure, you can hold the command arguments in an array, the argument count, and information dealing with input and/or output redirection, such as the file names to redirect to. Here is the basic idea for processing the tokens:

> (1) If the token is the < or > operator, you know that file redirection is needed. You should expect the next token to contain the name of the redirect file. You do not add the < or > operators, nor their respective filenames to the list of command arguments.
>
> (2) If the token starts with a $, you know you have an environment variable. Use getenv() to get its value and add this value to the list of command arguments.
>
> (3) Otherwise, you can simply add the token to the list of command

arguments and go back to step (1) while you still have tokens to process.

I think an example would help. Let's say the user types ls -a $HOME > files.txt. After tokenizing, you are left with:

```
ls
-a
$HOME
>
files.txt
```

Now, you go through each token and decide what to add to the list of arguments. You first add ls. You then add -a. You then notice the $HOME. You get the value of /home/grads/porter (for example) and add this to your list of arguments. You notice the > and files.txt. You record in your command structure that you will redirect output to a file named files.txt. Your final list of arguments looks like this:

```
ls
-a
/home/grads/porter
```

You are now ready to execute your command.

Executing Commands

Now that you have your command stored in some type of structure, you are ready to execute. Remember that there are two main types of commands, built-in and external. The built-in commands are straightforward. The external commands require a little more work.

Built-in Commands

cd - remember that this command changes the CWD. In order to do this, you will use the chdir() function. Just pass in the name of the directory you want to change to. Remember that there are various cases where you need to error. First, if the user gives more than one argument, it's an error. Next, the user may give an invalid directory. chdir() will let you know if it failed or not, so you can check for this. Remember, also, that changing the CWD should also change your command prompt. Ex:

```
porter@myshell:/home/grads>cd ..
porter@myshell:/home>cd grads
porter@myshell:/home/grads>
```

echo - remember that this command simply prints to the screen what the user typed after the echo. For example, echo 1 2 3 should print 1 2 3. So, you simply use a normal print statement (like a printf()) to implement this command.

exit - this is the easiest command to implement. You simply exit your program.

External Commands

With external commands, you have to do a little more work. But, the good news is that you don't have to implement any type of special functionality for a command. Instead, you simply call on a pre-existing (presumably)

command. The first step to executing a pre-existing command is to first check if the first argument (the command name itself) contains a /. You can use the strchr() function to do this. If it does, you don't have to do any type of searching for the command. If it doesn't, you do have to search for the command. To do this, you will get the value of the PATH environment variable. This variable contains various paths separated by a :. You will search from left to right until you find a path that contains your command. If you don't, of course you error. We already know how to parse based on whitespace. We can make a simple modification to parse based on a :. Take a look at tokenize2.c. There are some important things to notice. First, all I did was change the whitespace delimiter to a : delimiter. Second, and this is most important, I used the strcpy() function to copy a string into a buffer. You will have to do something similar. The reason is that when you use getenv() to get the value of an environment variable, you are not allowed to modify that actual string. If you do, bad things can happen. So, you'll need to be sure you make a copy of any value from getenv() that you need to modify before using that value. strtok(), of course, modifies the string. Hence, the copy.

So, you now know how to get the possible paths to an external command. Let's say the user types ls -a. We use strchr() on ls to see that there is no /. Remember that we're only looking at the command name itself for a /, not the entire command. strchr() tells us that we're going to have search the PATH environment variable for the ls command. To do so, we get the value of the variable, make a copy of it, then break it into colon-delimited tokens. We then start searching each token. Let's say our first token is /bin. We need to check if /bin contains the ls command. To do this, we append ls on to the path to get /bin/ls. To append, use strcat(). Now, the obvious question is how do we know if /bin/ls exists? Yet another function, of course. The stat() function will let you test for a file's existence. You simply give it the path to the file and a pointer to a structure to hold the file information.

And that's all there is to it...well almost! At this point, you can finally execute the command using execv(). There is another issue, however. execv() will replace the calling process with a new one corresponding to the command you want to execute. Obviously, we don't want to replace our shell process. So, the solution is to create a copy of our main process and let execv() replace this copy. This process, known as a child process, will run and our main process, known as the parent process, will wait on it to finish. Take a look at fork.c. There are a few things to discuss here. First, the fork() function will create a child process that is a copy of the our parent process. Now, both processes will be executing in parallel after the call. So, how do we know which process we are currently in? We use the return value of fork(). If it returns a value less than 0, it failed, so it didn't even make the child process. Usually, this won't happen. Otherwise, it will return a 0 to the child process and something greater than 0 (the child's process ID (PID)) to the parent process. Hence, you only execute your command in the child process case using execv(). Note that execv() shouldn't return. That's why I put an error message beneath it, just in case it does. You can also see an example command structure. First, I have an array of strings, with the final string being NULL. In this case, I'm simply trying to execute the command sleep 5. Now, this array is what you pass to execv() for the second argument. But, remember the first argument must be the path to sleep. So, I hard-coded in /bin/sleep for illustration purposes. Now, the parent process won't receive a 0 return value, hence it will never enter any of that code. Instead, it will wait for the child to finish. To wait, you use the waitpid() function. Essentially, it stops the parent process from doing anything until

the child process finishes. In other words, we want to wait until the child is done sleeping for 5 seconds before proceeding. In the next project, we'll talk about running processes in the background, where we don't have to wait before proceeding. In any event, you must make sure you have the waitpid(), because when the child process terminates, this function is necessary to free up the process entry in the process table. If you don't free it, you'll leave a so-called zombie process running.

File Redirection

File redirection is the final piece of the puzzle. If you're keeping some type of command structure, by the time you go to execute in the child process, you will be able to decide if input should come from a file and/or output should go to a file. Let's look at redirection.c. It looks fairly simple. It simply redirects output to a file (instead of to the screen like it normally would go). The key functions are open() and dup2(). open() will try to open a particular file. You can read about the various flags to give it. In this case, I am telling it to open a file for output, creating it if it doesn't exist. Now, to understand dup2(), you have to understand file descriptors. Basically, file descriptors correspond to files. By default, UNIX assigns a few for you. One of them is stdout, which corresponds to the screen...usually. stdout corresponds to the file descriptor 1. What dup2() will do is allow you to change where stdout will map to. In this case, I give it the file descriptor of the file I just opened. So, from this point on, when my program tries to write to stdout, it will go to the file instead of the screen. That's why the print statement at the end shows up in output.txt instead of on the screen. Reading from a file is simliar, except you open the file for reading purposes instead of writing purposes. Then, you use dup2() with 0, which corresponds to stdin in UNIX.

Ok, this should get you going in the right direction for the project. I don't want to give away everything, so there are some aspects that you may need to research. However, you have the basic ideas and links to the functions you will want to use. Good luck!

Sample Runs

Here are some sample runs that you can use to compare your output to. Note that your output doesn't have to match exactly, but it couldn't hurt.

Built-in Commands

cd

```
porter@myshell:/home/grads/porter>cd ..
porter@myshell:/home/grads>cd ..
porter@myshell:/home>cd ..
porter@myshell:/>cd .
porter@myshell:/>cd
porter@myshell:/home/grads/porter>cd /
porter@myshell:/>cd / /
cd: Too many arguments.
porter@myshell:/>cd
porter@myshell:/home/grads/porter>cd Fun
porter@myshell:/home/grads/porter/Fun>cd ..
porter@myshell:/home/grads/porter>cd FunAgain
FunAgain: No such file or directory.
porter@myshell:/home/grads/porter>cd ./Fun
porter@myshell:/home/grads/porter/Fun>cd
```

```
porter@myshell:/home/grads/porter>cd /home/grads/porter/Fun
porter@myshell:/home/grads/porter/Fun>cd $HOME
porter@myshell:/home/grads/porter>
```

echo

```
porter@myshell:/home/grads/porter>echo

porter@myshell:/home/grads/porter>echo Hello There!
Hello There!
porter@myshell:/home/grads/porter>echo 1 2 3
1 2 3
porter@myshell:/home/grads/porter>echo $HOME
/home/grads/porter
porter@myshell:/home/grads/porter>echo $USER
porter
porter@myshell:/home/grads/porter>echo $FUN
$FUN: Undefined variable.
porter@myshell:/home/grads/porter>echo $USER is at $HOME
porter is at /home/grads/porter
```

exit

```
porter@myshell:/home/grads/porter>exit
porter@linprog2.cs.fsu.edu:~>
```

External Commands

```
porter@myshell:/home/grads/porter>ls
a.out background.c fork.c Fun myshell myshell.c redirection.c tokenize2.c
tokenize.c
porter@myshell:/home/grads/porter>/bin/ls
a.out background.c fork.c Fun myshell myshell.c redirection.c tokenize2.c
tokenize.c
porter@myshell:/home/grads/porter>/ls
/ls: Command not found.
porter@myshell:/home/grads/porter>./ls
./ls: Command not found.
porter@myshell:/home/grads/porter>ls -al
total 160
drwx------ 7 porter CS-Grads 4096 Dec 14 20:26 .
drwxr-xr-x 296 root CS-Grads 12288 Sep 13 11:22 ..
-rw------- 1 porter CS-Grads 1673 Nov 3 18:20 .abbrev_defs
-rw------- 1 porter CS-Grads 577 Nov 10 21:41 .alias
-rwx------ 1 porter CS-Grads 14074 Dec 14 20:17 a.out
-rw-r--r-- 1 porter CS-Grads 1794 Dec 3 00:52 background.c
-rw------- 1 porter CS-Grads 207 Nov 26 18:01 .bash_history
drwxr-xr-x 2 porter CS-Majors 4096 Jan 18 2006 .bin
-rw-r--r-- 1 porter CS-Majors 570 Oct 28 2007 .cshrc
-rw------- 1 porter CS-Grads 365 Oct 12 21:13 .emacs
-rw-r--r-- 1 porter CS-Grads 797 Dec 10 19:07 fork.c
-rw------- 1 porter CS-Grads 15 Sep 21 03:23 .forward
drwx------ 3 porter CS-Grads 4096 Dec 6 21:58 Fun
-rw------- 1 porter CS-Grads 6373 Dec 10 19:56 .history
-rw------- 1 porter CS-Grads 35 Nov 26 18:01 .lesshst
drwx------ 2 porter CS-Grads 4096 Sep 26 2007 .lisp
-rw-r--r-- 1 porter CS-Majors 1131 Aug 24 2005 .login
```

```
-rwx------ 1 porter CS-Grads 14898 Dec 6 21:56 myshell
-rw-r--r-- 1 porter CS-Grads 13709 Dec 14 20:16 myshell.c
-rw-r--r-- 1 porter CS-Grads 637 Dec 10 19:28 redirection.c
drwxr-xr-x 2 porter CS-Majors 4096 Dec 16 2004 .scripts
-rw-r--r-- 1 porter CS-Majors 1348 Sep 25 2006 .setup
-rw------- 1 porter CS-Grads 8 Sep 15 18:02 .sh_history
drwx------ 2 porter CS-Majors 4096 Sep 10 2005 .ssh
-rw-r--r-- 1 porter CS-Majors 898 Nov 23 19:24 .tcshrc
-rw-r--r-- 1 porter CS-Grads 751 Dec 10 18:41 tokenize2.c
-rw-r--r-- 1 porter CS-Grads 1073 Dec 8 00:23 tokenize.c
-rw------- 1 porter CS-Grads 777 Dec 14 20:17 .Xauthority
porter@myshell:/home/grads/porter>date
Tue Dec 14 20:27:39 EST 2010
porter@myshell:/home/grads/porter>cat myshell.c
I'M NOT THAT CRAZY!
porter@myshell:/home/grads/porter>ls > output.txt
porter@myshell:/home/grads/porter>cat output.txt
a.out
background.c
fork.c
Fun
myshell
myshell.c
output.txt
redirection.c
tokenize2.c
tokenize.c
porter@myshell:/home/grads/porter>ls < output.txt
a.out background.c fork.c Fun myshell myshell.c output.txt redirection.c
tokenize2.c tokenize.c
porter@myshell:/home/grads/porter>ls < output2.txt
output2.txt: No such file or directory.
porter@myshell:/home/grads/porter>ls < output.txt > output2.txt
porter@myshell:/home/grads/porter>more output2.txt
a.out
background.c
fork.c
Fun
myshell
myshell.c
output2.txt
output.txt
redirection.c
tokenize2.c
tokenize.c
porter@myshell:/home/grads/porter>ls $KDEDIR
bin games java JMF lib libexec lost+found share tmp
etc include javasrc kerberos lib64 local sbin src X11R6
porter@myshell:/home/grads/porter>
```

Grading

Your code uses good programming practices (30%).

Your shell supports external commands (30%).

Your shell properly prints the command prompt (10%).

Your shell supports the three built-in commands (10%).

Your shell supports file redirection (10%).

Your shell supports environment variables (10%).

Submitting

Submit two files: myshell.c containing your source code, and makefile which builds an executable named myshell.x. Do this with ~cop4610p/submitscripts/proj1submit.sh.