Programming Project 1

Submit Assignment

Due May 1 by 11:59pm **Points** 100 **Submitting** a website url
Available Apr 8 at 5pm - May 2 at 11:59pm 24 days

NOTES

- **We suggest that you do this project in a 2-students group.** You can talk to other teams about the project, but every line of code must be written and understood by you. Of course, you can always ask the TAs and professors for help, too.
- Please feel free to ask any questions about this project on Piazza; and feel free to come to the TAs' office hours for some help on the project.

Objectives

There are three objectives to this assignment:

- To familiarize yourself with the Linux programming environment.
- To learn how processes are created, destroyed, and managed.
- To gain exposure to the necessary functionality in shells.

OVERVIEW

In this assignment, you will implement a **command line interpreter** or, as it is more commonly known, a **shell**. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shells you implement will be similar to, but simpler than, the one you run every day in Linux. You can find out which shell you are running by typing "echo $SHELL" at a prompt. You may then wish to look at the man pages for 'csh' or the shell you are running (more likely tcsh, or bash, or for those few wacky ones in the crowd, zsh or ksh) to learn more about all of the functionality that can be present. For this project, you do not need to implement too much functionality.

PROGRAM SPECIFICATIONS

- **Basic Shell**

Your basic shell is basically an interactive loop: it repeatedly prints a prompt `"520shell> "` (note the space after the percent sign), parses the input, executes the command specified on that line of input, and waits for the command to finish. This is repeated until the user types "bye". The name of your final executable should be **mysh** :

```
prompt> ./mysh
520shell>
```

You should structure your shell such that it creates a new process for each new command. There are two advantages of creating a new process. First, it protects the main shell process from any errors that occur in the new command. Second, it allows for concurrency; that is, multiple commands can be started and allowed to execute simultaneously.

Your basic shell should be able to parse a command, and run the program corresponding to the command. For example, if the user types `"ls -la /tmp"` , your shell should run the program `ls` with all the given arguments and print the output on the screen.

Note that the shell itself does not "implement" `ls` or really many other commands at all. All it does is find those executables and create a new process to run them. More on this below.

The maximum length of a command line your shell can take is 64 bytes (excluding the carriage return).

- **Multiple Commands**

  After you get your basic shell running, your shell is not too fun if you cannot run multiple jobs on a single command line. To do that, we use the `";"` character to separate multiple jobs on a single command line.

  For example, if the user types `"ls; ps; who"` , the jobs should be run all one-by-one in sequential mode. Hence, in our previous example (`"ls; ps; who"`), all three jobs should run: first ls, then ps, then who. After they are done, you should finally get a prompt back.

  Your shell also has a neat parallel mode. For example, if the user types `"ls & ps & who"` , the jobs should be run all at the same time. As before, after they are all done, you should finally get a prompt back.

  Your shell should not accept mixing ``&" and ``;" in one line of command, and should output the generic error message in that situation.

- **Built-in Commands**

  Whenever your shell accepts a command, it should check whether the command is a **built-in command** or not. If it is, it should not be executed like other programs. Instead, your shell will invoke your implementation of the built-in command. For

example, to implement the **bye** built-in command, you simply call `exit(0);` in your C program.

So far, you have added your own `bye` built-in command. Most Unix shells have many others such as `cd` , `echo` , `pwd` , etc. In this project, you should implement **cd**, **pwd**, and **echo**. Your shell users will be happy with this feature because they can change their working directory. Without this feature, your user is stuck in a single directory.

bye, cd, pwd, and echo formats. The formats for **bye** , **cd**, **pwd**, **echo** are:

- ```
  [optionalSpace]bye[optionalSpace]
  ```

  ```
  [optionalSpace]pwd[optionalSpace]
  ```

  ```
  [optionalSpace]cd[optionalSpace]
  ```

  ```
  [optionalSpace]cd[oneOrMoreSpace]dir[optionalSpace]
  ```

  ```
  [optionalSpace]echo[optionalSpace]string[optionalSpace]
  ```

When you run "cd" (without arguments), your shell should change the working directory to the path stored in the $HOME environment variable. Use `getenv("HOME")` to obtain this.

You do not have to support tilde (~). Although in a typical Unix shell you could go to a user's directory by typing "cd ~username", in this project you do not have to deal with tilde. You should treat it like a common character, i.e. you should just pass the whole word (e.g. "~username") to chdir(), and chdir will return error.

Basically, when a user types pwd, you simply call getcwd(). When a user changes the current working directory (e.g. "cd somepath"), you simply call chdir(). Hence, if you run your shell, cd, and then run pwd, it should look like this:

```
% cd
```

```
% pwd
```

```
/home/username
```

```
 % ./mysh
```

```
520shell> pwd
```

```
/home/username
```
```

```
520shell>
```

When you run "echo", the string following it will be printed out, without those white space before or after it. For your shell, you only need to handle one string parameter for "echo" that does not contain any special character like "&" or ";".

```
520shell> echo hello
```

```
hello
```

```
520shell> echo hello
```

```
hello
```

```
520shell> echo hi hello
```

```
hi
```

- **Redirection**

  Many times, a shell user prefers to send the output of his/her program to a file rather than to the screen. The UNIX shell provides this nice feature with the `">"` character. Formally this is named as redirection of standard output. To make your shell users happy, your shell should also include this feature.

  For example, if a user types `"ls -la /tmp > output"` , nothing should be printed on the screen. Instead, the output of the `ls` program should be rerouted to the `output` file.

  If the `"output"` file already exists before you run your program, you should simple overwrite the file (after truncating it). If the output file is not specified (e.g. `"ls > "`), you should also print an error message.

  Here are some redirections that should **not** work:

  ```
  ls > out1 out2
  ```

  ```
  ls > out1 out2 out3
  ```

  ```
  ls > out1 > out2
  ```

- **Pipeline**

As a new twist, your shell will also have the ability to feed the output of a program into the input of another program. The format for this is as follows:

> prompt> ls | grep ".c"

When your shell encounters this line, it has to do a couple of things. It has to create two new processes (in the example, one to execute `ls` and one to execute `grep ``.c''`). It needs to use the `pipe()` system call to connect the two together, thus sending the output of `ls` into the input of `grep` .

Note that the output of each command can only be redirected once, either to a file through ``>'' or to another command through ``|''. Your shell should not allow this: `ls > file | grep ".c"` or anything similar.

- **Program Errors**

  The one and only error message. In summary, you should print this one and only error message whenever you encounter an error of any type:

  ```
  char error_message[30] = "An ERROR has occurred\n";
  ```

  ```
  write(STDERR_FILENO, error_message, strlen(error_message));
  ```

  The error message should be printed to stderr (standard error). Also, do not add whitespaces or tabs or extra error messages.

  There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there is any program-related errors (e.g. invalid arguments to `ls` when you run it, for example), let the program prints its specific error messages in any manner it desires (e.g. could be stdout or stderr).

- **White Spaces**

  Zero or more spaces can exist between a command and the shell special characters (i.e. `";"` and `">"`). All of these examples are correct.

  ```
  520shell> ls;ls;ls
  ```

  ```
  520shell> ls ; ls ; ls
  ```

  ```
  520shell> ls>a; ls > b; ls> c; ls >d
  ```

- **Batch Mode**

So far, you have run the shell in interactive mode. Most of the time, testing your shell in interactive mode is time-consuming. To make testing much faster, your shell should support **batch mode** .

In interactive mode, you display a prompt and the user of the shell will type in one or more commands at the prompt. In batch mode, your shell is started by specifying a batch file on its command line; the batch file contains the same list of commands as you would have typed in the interactive mode.

In batch mode, you should **not** display a prompt. You should print each line you read from the batch file back to the user before executing it; this will help you when you debug your shells (and us when we test your programs). To print the command line, **do not use printf** because printf will buffer the string in the C library and will not work as expected when you perform automated testing. To print the command line, use write(STDOUT_FILENO, ...) this way:

> write(STDOUT_FILENO, cmdline, strlen(cmdline));

In both interactive and batch mode, your shell should terminates when it sees the `bye` command on a line or reaches the end of the input stream (i.e., the end of the batch file).

To run the batch mode, your C program must be invoked exactly as follows: `mysh [batchFile]`

The command line arguments to your shell are to be interpreted as follows.

> batchFile: an optional argument (often indicated by square brackets as above). If present, your shell will read each line of the batchFile for commands to be executed. If not present or readable, you should print the one and only error message (see **Error Message** section below).

Implementing the batch mode should be very straightforward if your shell code is nicely structured. The batch file basically contains the same exact lines that you would have typed interactively in your shell. For example, if in the interactive mode, you test your program with these inputs:

> ```
> % ./mysh 520shell> ls ; who ; ps
> ```
>
> ```
> some output printed here
> ```
>
> ```
> 520shell> ls > /tmp/ls-out;;;; ps > /non-existing-dir/file;
> ```
>
> ```
> some output and error printed here
> ```
>
> ```
> 520shell> ls-who-ps
> ```

```
some error printed here
```

then you could cut your testing time by putting the same input lines to a batch file (for example myBatchFile):

```
ls ; who ; ps
```

```
ls > /tmp/ls-out;;;; ps > /non-existing-dir/file;
```

```
ls-who-ps
```

and run your shell in batch mode:

```
% ./mysh myBatchFile
```

In this example, the output of the batch mode should look like this:

```
ls ; who ; ps
```

```
some output printed here
```

```
ls > /tmp/ls-out;;;; ps > /non-existing-dir/file; some output and error printed
 here ls-who-ps some error printed here
```

Important Note: To automate grading, we will heavily use the batch mode . If you do everything correctly except the batch mode, you could be in trouble. Hence, make sure you can read and run the commands in the batch file. Soon, we will provide some batch files for you to test your program.

Defensive Programming and Error Messages:

As in the first project, in this project defensive programming is also required. Your program should check all parameters, error-codes, etc. before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner; by "reasonable", we mean print the error message (as specified in the next paragraph) and either continue processing or exit, depending upon the situation.

Since your code will be graded with automated testing, you should print this *one and only error message* whenever you encounter an error of any type:

```
char error_message[30] = "An ERROR has occurred\n";
```

```
write(STDERR_FILENO, error_message, strlen(error_message));
```

For this project, the error message should be printed to *stderr* Also, do not attempt to add whitespaces or tabs or extra error messages.

You should consider the following situations as errors; in each case, your shell should print the error message to stderr and **exit** gracefully:

- An incorrect number of command line arguments to your shell program.

For the following situation, you should print the error message to stderr and **continue** processing:

- A command does not exist or cannot be executed.
- A very long command line (over 64 characters, excluding the carriage return)

Your shell should also be able to handle the following scenarios below, which are **not errors** . The best way to check if something is not an error is to run the command line in the real Unix shell.

- An empty command line.
- An empty command between two or more ';' characters.
- Multiple white spaces on a command line.
- White space before or after the ';' character or extra white space in general.

All of these requirements will be tested extensively.

HINTS

Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly. To simplify things for you in this assignment, we will suggest a few library routines you may want to use to make your coding easier. (Do not expect this detailed of advice for future assignments!) You are free to use these routines if you want or to disregard our suggestions. To find information on these library routines, look at the manual pages (using the Unix command **man**).

1. **Basic Shell**

   Parsing: For reading lines of input, you may want to look at **fgets().** To open a file and get a handle with type **FILE *** , look into **fopen().** Be sure to check the return code of these routines for errors! (If you see an error, the routine **perror()** is useful for displaying the problem. *But do not print the error message from perror() to the screen. You should only print the one and only error message that we have specified above*). You may find the **strtok()** routine useful for parsing the command line (i.e., for extracting the arguments within a command separated by whitespace or a tab or ...).

   Executing Commands: Look into **fork** , **execvp** , and **wait/waitpid** . See the UNIX man pages for these functions, and also read the Advance Programming in the UNIX Environment, **Chapter 8** (specifically, 8.1, 8.2, 8.3, 8.6, 8.10). Before starting this project, you should definitely play around with these functions.

Pipes: the **pipe()** system call is needed here. **Chapter 15** of the APUE book may be useful (specifically, about pipes).

You will note that there are a variety of commands in the `exec` family; for this project, you must use **execvp** . You should **not** use the **system()** call to run a command. Remember that if `execvp()` is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified. The first argument specifies the program that should be executed, with the full path specified; this is straight-forward. The second argument, `char *argv[]` matches those that the program sees in its function prototype:

```
int main(int argc, char *argv[]);
```

Note that this argument is an array of strings, or an array of pointers to characters. For example, if you invoke a program with:

```
foo 205 535
```

then argv[0] = "foo", argv[1] = "205" and argv[2] = "535".

Important: the list of arguments must be terminated with a NULL pointer; that is, argv[3] = NULL. We strongly recommend that you carefully check that you are constructing this array correctly!

2. **Multiple Commands**

   If you get your basic shell running, supporting multiple commands should be straightforward. The only difference here is that you need to wait for the previous process to finish before creating a new one. To do that, you simply use waitpid() again.

3. **Built-in Commands**

   For the 'bye' built-in command, you should simply call 'exit();'. The corresponding process will exit, and the parent (i.e. your shell) will be notified.

   For managing the current working directory, you should use **getenv** , **chdir** , and **getcwd** . The `getenv()` call is useful when you want to go to your HOME directory. **You do not have to manage the PWD environment variable.** getcwd() system call is useful to know the current working directory; i.e. if a user types pwd, you simply call getcwd(). And finally, chdir is useful for moving directories. Fore more, read the Advanced UNIX Programming book **Chapter 4.22 and 7.9** .

4. **File Redirection**

   Redirection is probably the most tricky part of this project. For this you **only need dup2()** .

   The idea of using `dup2()` is to intercept the byte stream going to the standard output (i.e. your screen), and redirect the stream to your designated file. dup2 uses file descriptors, which implies

that you need to understand what a file descriptor is. You should read the Advanced UNIX Programming book **Chapter 1.5** to get an initial understanding of what a file descriptor is.

With file descriptor, you can perform read and write to a file. Maybe in your life so far, you have only used **fopen()** , **fread()** , and **fwrite()** for reading and writing to a file. Unfortunately, these functions work on **FILE*** , which is more of a C library support; the file descriptors are hidden. Hence, it is impossible for you to use dup2 with these particular functions.

To work on file descriptor, you should use **creat()** , **open()** , **read()** , and **write()** system calls. These functions perform their works by using file descriptors. To understand more about file I/O and file descriptors you should read the Advanced UNIX Programming book **Section 3** (specifically, 3.2 to 3.5, 3.7, 3.8, and 3.12). Before reading forward, at this point, you should get yourself familiar with file descriptor.

The idea of redirection is to make the stdout descriptor point to your output file descriptor. First of all, let's understand the STDOUT_FILENO file descriptor. When a command `"ls -la /tmp"` runs, the ls program prints its output to the screen. But obviously, the ls program does not know what a screen is. All it knows is that the screen is basically pointed by the STDOUT_FILENO file descriptor. In other words, you could rewrite `printf("hi")` in this way: `write(STDOUT_FILENO, "hi", 2)` .

To give yourself a practice, create a simple program where you create an output file, intercept stdout, and call printf("hello"). When you creat your output file, you should get the corresponding file descriptor. To intercept stdout, you should call `"dup2(output_fd, STDOUT_FILENO);"` . If you run your program, you should not see "hello" printed on the screen. Instead, the word has been redirected to your output file.

In short, to intercept your `ls` output, you should redirect stdout before you execute `ls` , i.e. make the dup2() call before the exec('ls') call. Alternately, you can close(STDOUT_FILENO) and open() a new file; that file will be assigned the lowest available file descriptor and hence will be assigned to standard output.

5. **Miscellaneous Hints**

Remember to get the **basic functionality** of your shell working before worrying about all of the error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as "ls"). Then try adding more arguments.

Next, try working on multiple commands. Make sure that you are correctly handling all of the cases where there is miscellaneous white space around commands or missing commands. Finally, you add built-in commands and redirection supports.

We strongly recommend that you check the return codes of all system calls from the very beginning of your work. This will often catch errors in how you are invoking these new system calls. And, it's just good programming sense.

Beat up your own code! You are the best (and in this case, the only) tester of this code. Throw lots of junk at it and make sure the shell behaves well. Good code comes through testing -- you must

run all sorts of different tests to make sure things work as desired. Don't be gentle -- other users certainly won't be. Break it now so we don't have to break it later.

Keep versions of your code! Each of you get a phoenixforge directory, which provides a SVN code repository for you. Minimally, when you get a piece of functionality working, check in this version of code. By keeping older, working versions around, you can comfortably work on adding new functionality, safe in the knowledge you can always go back to an older, working version if need be. **We will put up a post on Piazza about how to use your phoenixforge directories.**

HANDIN

To ensure that we compile your C correctly for the demo, you will need to create a simple **makefile;** this way our scripts can just run `make` to compile your code with the right libraries and flags. If you don't know how to write a makefile, you might want to look at the man pages for `make`, or read this **tutorial (http://pages.cs.wisc.edu/~remzi/OSTEP/lab-tutorial.pdf)**.

The name of your final executable should be `mysh` , i.e. your C program must be invoked exactly as follows:

```
prompt> ./mysh
prompt> ./mysh inputTestFile
```

Please also submit a README file (i.e. filename: "README"). Your README file should (1) list the members of your team; and (2) describe what functionalities are not working (if you think your code is perfect, simply write "All good").

Keep your source code (including the README and Makefile) in the **p1** sub-directory of your phoenixforge directory: https://phoenixforge.cs.uchicago.edu/svn/-mpcs52030-spr-20/p1 The TA will look at the time-stamp and use the last version before deadline as your submission.

Put all of your .c source files into the above directory. Do **not** put any .o files in the SVN. Make sure that your code runs correctly on linux machines (e.g., try linux.cs.uchicago.edu).

If you work in a 2-people group, please only use **one** person's phoenixforge code repository (i.e., do **not** copy code around and keep two code repositories). Just keep in mind that (1) the owner of a code repository can and should give full access permission to your team partner; and (2) specify both the team members in the README.

GRADING

We will run your program on a suite of test cases, some of which will exercise your programs ability to correctly execute commands and some of which will test your programs ability to catch error conditions. Be sure that you thoroughly exercise your program's capabilities on a wide range of test suites, so that you will not be unpleasantly surprised when we run our tests.

Thank Professor Remzi A.-D. for providing the project materials.