Project 2a: The Unix Shell

Objectives

There are three objectives to this project:

- 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.

Updates

Read these updates to keep up with any small fixes in the specification.

- We have added a diagram to help you understand redirection.
- Duplicates in path are allowed. For example,

```
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> path /a /b /a
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> path
/a
/b
/a
```

- For this project, calling `system()` or `execvp()` is not allowed. You must use `execv()`.

Overview

In this assignment, you will implement a **command line interpreter (CLI)** 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 Unix. 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 the shell you are running (probably `bash`) 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, called `xsh` , is basically an interactive loop: it repeatedly prints a prompt, 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 "exit". The name of your final executable should be **xsh**. The prompt of xsh should look like `"[$PWD]\n$?> "` (note the space after the greater-than sign), where

`$PWD` is the current working directory and `$?` is the exit status of the **previous** command (initially 0). For example:

```
shell% ./xsh
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0>
```

You should structure your shell such that it creates a new process for each new command (note that there are a few exceptions to this, which we discuss below). 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. **However, in this project, you do not have to build any support for running multiple commands at once.**

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 `/bin/ls` with all the given arguments and print the output on the screen. When `ls` exits, your shell should get its exit status and print it in the next prompt.

You might be wondering how the shell knows to run `/bin/ls` (which means the program binary `ls` is found in the directory `/bin`) when you type `ls` . The shell knows this thanks to a **path** variable that the user sets. The path variable contains the list of all directories to search, in order, when the user types a command. We'll learn more about how to deal with the path below.

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

The maximum length of a line of input to the shell is 128 bytes.

You may notice the following when you run your shell:

```
shell% ./xsh
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> lls^[[D
```

Basically, you cannot use the left arrow key on the keyboard to move backwards if you typed something wrong. To fix this, start your shell with a wrapper as follows:

```
shell% ~cs537-1/ta/tools/rlwrap ./xsh
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> lls
```

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 `exit` built-in command, you simply call `exit(0);` in your shell.

So far, you have added your own `exit` built-in command. Most Unix shells have many others such as `cd` , `alias` , `history` , etc. In this project, you should implement **exit**, **cd**, **path**, and **type**.

The formats for **exit** and **cd** are:

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

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

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 an error.

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

```
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> cd /tmp
[/tmp]
0>
```

The format of the **path** built-in command is:

```
[optionalSpace]path[optionalSpace]
[optionalSpace]path[oneOrMoreSpace]dir[optionalSpace] (and possibly
more directories, space separated)
```

A typical usage would be like this:

```
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> path /bin /usr/bin
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> path
/bin
/usr/bin
```

By doing this, your shell will know to look in `/bin` and `/usr/bin` when a user types a command, to see if it can find the proper binary to execute. If the `path` command is not followed by any directories to set, then the shell will print all directories currently in path in their original order. Do NOT set path to empty in this case. Just like in bash, duplicates in path are allowed.

By default, the path should be set to **/bin** so that the user can type a few commands without first setting the path. Type `ls /bin` to see what commands are available.

The format of the **type** built-in command is:

```
[optionalSpace]type[oneOrMoreSpace]command[optionalSpace]
```

The **type** built-in shows how would xsh interpret a certain command. A typical usage would be like this:

```
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> type pwd
pwd is /bin/pwd
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> type exit
exit is a shell builtin
```

The exit status of a built-in command is always 0 or 1, depending on whether it was successful(0) or not(1).

Redirection

Many times, a shell user prefers to send the output of his/her program to a file rather than to the screen. Usually, a 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, but with a slight twist (explained below).

For example, if a user types `"ls -la /tmp > output"` , nothing should be printed on the screen. Instead, the standard output of the `ls` program should be rerouted to the `output.out` file. In addition, the standard error output of the file should be rerouted to the file `output.err` (the twist is that this is a little different than standard redirection).

If the `output.out` or `output.err` files already exists before you run your program, you should simple overwrite them (after truncating). If the output file is not specified (e.g. the user types `ls >`), you should print an error message and not run the program `ls` .

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

```
ls > out1 out2
ls > out1 out2 out3
ls > out1 > out2
```

Note: don't worry about redirection for built-in commands (e.g., we will not test what happens when you type `path /bin > file`). Empty command with redirection (`> file`) will not be tested either.

The `">"` operator will be separated by spaces. Valid input may include the following:

```
ls
ls > a
ls > a
```

But not this (it is ok if this works, it just doesn't have to):

```
ls>a
```

Similarly, the standard input of a program can also be redirected to read from a file, rather than from the user interactively. The syntax is to use < instead of >. For example, **bc** is a simple command calculator.

```
shell% bc
2*3
6
shell% cat ~cs537-1/ta/tests/2a/ctests/bc/input
2*3
shell% bc < ~cs537-1/ta/tests/2a/ctests/bc/input
6
```

But sometimes we may not want to create a separate file. Why can't the input be passed like a command line argument? Here comes the here-string syntax

```
shell% bc <<< 2*3
6
```

In this example, no file was created. The standard input of **bc** connects to a pipe created by the shell, and the string `2*3\n` is sent from the shell to **bc**. From **bc**'s perspective, it looks like the string comes from standard input.

In your shell, the only input redirection mechanism required is here-string. Input redirection from a file is not required. Formally, there are no spaces between the three arrows in "<<<", and there will be one or more spaces around this operator. Here-string can appear alone or after output redirection, and there must be exactly one word after the "<<<" operator.

Note that **bc** is in **/usr/bin**. You will need to set the path before you can test the here-string in your own shell:

```
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> path /usr/bin
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> bc <<< 2*3
6
```

Defensive Programming and Error Messages

Defensive programming is 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.

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). Besides, shell-related errors set the exit status in prompt to 1, while program-related errors set it to whatever the program returns. For example,

```
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> invalid-command
An error has occurred
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
1> ls
lint  tests  tools  xv6
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
0> ls /bad
/bin/ls: cannot access '/bad': No such file or directory
[/afs/cs.wisc.edu/p/course/cs537-gerald/ta]
2>
```

Here the shell cannot find `invalid-command`, so the only error message is printed and the exit status is set to 1. However in the third command, the shell starts `ls` with argument `/bad` successfully, but `ls` reports an error and calls `exit(2);`

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 128 bytes).

Your shell should also be able to handle the following scenarios below, which are **not errors.**

- An empty command line. The exit status should not change as no external or built-in command is executed.
- Multiple white spaces on a command line.
- Tabs are used in place of spaces.

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. 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.

Basic Shell

Prompting: To get the current working directory in the prompt, take a look at the **getcwd()** function. The exit status should be from the execution of the previous command. If that was from an external program, you can get it with **wait()/waitpid()**. Do not worry about programs that exit with negative numbers.

Parsing: For reading lines of input, once again check out **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 whitespaces).

Executing Commands: Look into **fork, execv,** and **wait/waitpid.** See the man pages for these functions, and also read book chapter here.

You will note that there are a variety of commands in the `exec` family; for this project, you must use **execv.** You should **not** use the **system()** call to run a command. Remember that if `execv()` 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
```

and assuming that you find `foo` in directory `/bin` , then argv[0] = "/bin/foo", argv[1] = "205" and argv[2] = "535".

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

To check if a particular program exists in a directory, use the **stat()** system call. For example, when the user types `ls` , and path is set to include both `/bin` and `/usr/bin` , try `stat("/bin/ls")` . If that fails, try `stat("/usr/bin/ls")` . If that fails too, print the **only error message.**

Built-in Commands

For the `exit` built-in command, you should simply call `exit()`.

For managing the current working directory, you should use **getenv** and **chdir**. The `getenv()` call is useful when you want to go to your HOME directory. **You do not have to manage the PWD**

environment variable. And `chdir()` is useful for moving directories. For more information on these topics, read the man pages or the Advanced Unix Programming book **Chapters 4 and 7.**

Redirection

Redirection is relatively easy to implement. For example, to redirect standard output to a file, **open()** a file and use **dup2()** to connect it to stdout. More on this below.

With a 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.

To work on a file descriptor, you should use **open()** , **read()** , and **write()** system calls. These functions perform their work 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), or just read the man pages. Before reading forward, at this point, you should become more familiar with file descriptors.

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)`.

For here-string redirection, use **pipe()** instead of **open()** to get the file descriptor for redirection of STDIN_FILENO.

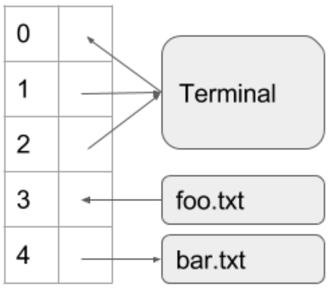

Figure 1: A normal program that reads from foo.txt and writes to bar.txt

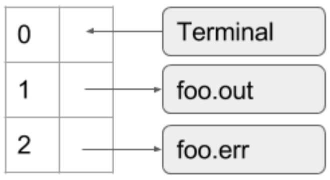

Figure 2: A program with its stdout and stderr redirected to files on disk

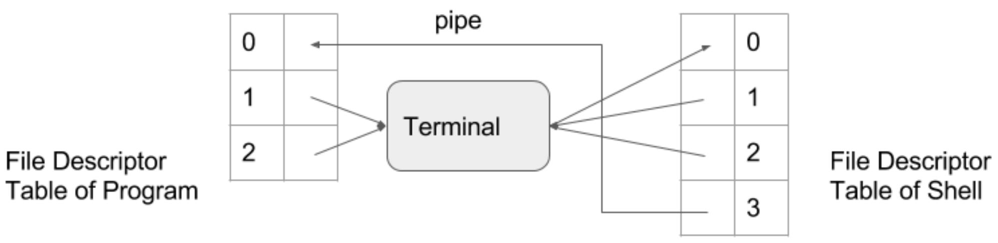
pipe
0
1
2
Terminal
0
1
2
3
File Descriptor Table of Program
File Descriptor Table of Shell

Figure 3: A program with its stdin redirected from a pipe with the shell

Figure 1 is the file descriptor table for a program without I/O Redirection. When it calls `printf` or `write(1, ...)`, the output goes to file descriptor 1, which is the terminal. Similarly, `scanf` or `read(0, ...)` or `fgets(..., stdin)` goes to file descriptor 0, which is also the terminal by default.

The program in Figure 2 redirects its standard out and stardard err to two files. As a result, when it calls `printf` or `write(1, ...)`, the output goes to `foo.out`. Note that from the program's perspective, it is doing the same function calls with the same arguments as in Figure 1.

Figure 3 involves two programs and a pipe between them. Anything written by the shell with `write(3, ...)` can be read by the user program with `read(0, ...)` or `fgets(..., stdin)`. Again, the user program is not aware of this.

To manipulate with the file descriptor table, use `open`, `close`, `dup2`, and `pipe`.

Test Along The Way

It is a good habit to get basic funtionalities working before moving to advanced features. You can run an individual test with the following command:

```
shell% ~cs537-1/ta/tests/2a/runtests testname
```

1. The easiest test is named **exit**. You only need to print the prompt and implement the **exit** built-in command.
2. Add the ability to parse and execute an external program. The following tests should pass at this stage: exec, badexec, stress, line, badline, badarg, whitespace
3. Support the built-in commands **cd** and **path**: cd, badcd, path, path2
4. The **type** built-in command: typebin, typebuiltin, typeorder, badtype
5. Output Redirection: rdr, rdr2, badrdr, badrdr2
6. Here-String: herestr, herestr2, badherestr, badherestr2
7. No memory leak in all features above: valgrindtest

Miscellaneous Hints

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. More advanced programmers will use a source control system such as git. Minimally, when you get a piece of functionality working, make a copy of your .c file (perhaps a subdirectory with a version number, such as v1, v2, etc.). 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.