CSC 458: Operating Systems

Program 1: Mines Shell

February 2, 2024

In this project, you'll build a simple Unix shell. The shell is the heart of the command-line interface, and thus is central to the Unix/C programming environment. Mastering use of the shell is necessary to become proficient at using any OS. Knowing how the shell itself is built is the focus of this project. The objectives of this assignment are

- to further familiarize yourself with the Unix/Linux programming environment,
- to learn how processes are created, destroyed, and managed, and
- to gain exposure to the minimum functionality provided by shells.

1 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 another command. The shell you implement will be similar to, but simpler than, the one you run every day in Unix. If you don't know what shell you are running, it's probably the Bourne Again Shell (bash). One thing you should do on your own time is learn more about your shell, by reading the man pages or other online materials.

2 Program Specifications

Your basic shell, called `mish` (short for MInes SHell, naturally), performs a simple loop. On each iteration, it reads a line of input, 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` or until end of file. The name of your final executable should be `mish`. The shell can be invoked with either no arguments or a single argument. It should exit and report an error if given more than one argument. When started with no arguments, the shell is in interactive mode, and allows the user to type commands directly.

The shell also supports non-interactive mode which reads input from a shell script file. Here is how you run the shell with a script named `batch.sh`:

```
prompt> ./mish batch.sh
```

In interactive mode, a prompt is printed between each command. In non-interactive mode, no prompt should be printed.

You should structure your shell such that it creates a process for each new command (the exception are built-in commands, discussed below). 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 the given arguments `-la` and `/tmp`. How does the shell know to run `/bin/ls`? It uses something called the *path,* which is described below.

3 Basic Shell

The shell is very simple (conceptually): it runs in a loop, repeatedly asking for input to tell it what command to execute. It then executes that command. For reading lines of input, you can use `getline()` or you may prefer to use the GNU Readline Library. Either approach allows you to obtain arbitrarily long input lines with ease. Readline is more powerful.

Generally, the shell will be run in interactive mode, where the user types a command (one at a time) and the shell acts on it. However, your shell will also support non-interactive mode, in which the shell is given an input file containing the commands commands. In this case, the shell should not read user input from `stdin` but rather from the script file to get the commands to execute. In either mode, if you hit the end-of-file marker (EOF), you should call `exit(0)` to exit gracefully.

3.1 Basic Parsing

To parse the input line into constituent pieces, you might want to use `strtok()` (or, if doing nested tokenization, use `strtok_r())`. Read the man page (carefully) for more details. To execute commands, look into `fork()`, `exec()`, and `wait()/waitpid()`. See the man pages for these functions, and also read the relevant book chapter for a brief overview. You will note that there are a variety of commands in the `exec` family. For this project, you must use `execvpe()`. You are not allowed to use the `system()` library function to run a command. Remember that if `execvpe()` is successful, it will not return. if it does return, there was an error. The most challenging part is getting the arguments correctly specified.

3.2 Environment Variables

In our example above, the user typed `ls` but the shell knew to execute the program `/bin/ls`. The shell uses the `PATH` environment variable to accomplish this. The `PATH` variable to provides the list of all directories to search, in order, when the user types a command. Note that the shell itself does not implement `ls` or most other commands, except for a few built-in commands. All it does is find those executables in one of the directories specified by the path and create a new process to run them.

Most shells allow you to specify a specific binary without using a search path, using either absolute paths or relative paths. For example, a user could type the absolute path `/bin/ls` to execute the `ls` program without performing a search for the program. A user could also specify a relative path which starts with the current working directory and specifies the executable directly.

For example, `./main` would run the program named `main` that resides in the current directory. The `execvpe()` system call will automatically perform the search. You just have to provide a way to set environment variables in your shell.

3.3 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; in your mish source `exit` built-in command, you simply call the `exit(0)` system call. In this project, you should implement `exit`, `cd`, and environment variable assignment (.e.g. `PATH=`) as built-in commands.

exit When the user types `exit`, your shell should simply call the exit system call with 0 as a parameter. It is an error to pass any command-line arguments to the `exit` command.

cd The `cd` command always takes one argument (0 or > 1 args should be signaled as an error). To change directories, use the `chdir()` system call with the argument supplied by the user. If `chdir` fails, that is also an error.

<var>= The <var>= command assigns a value to the given environment variable. If the variable does not exist, then it is created. The `PATH` environment variable should be inherited from the parent, but can be changed. The `PATH` environment variable is a list of directories, separated by colons `:` that specify what directories to examine when searching for an executable. A typical usage would be like this:

```
mish> PATH=/bin:/usr/bin}
```

which would tell the shell to look for commands in the `/bin` and `/usr/bin` directories. If the user sets the path to be empty, then the shell should not be able to run any programs, except built-in commands and commands that are specified using the complete path or a relative path. The `PATH=` command always overwrites the old path with the newly specified path.

4 Intermediate Features

Your shell can now run basic commands. To get full credit, you must add some features to make your shell more powerful.

4.1 Input and Output Redirection

Many times, a shell user prefers to send the output of a 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. For example, if a user types `ls -la /tmp > output`, nothing should be printed on the screen. Instead, the standard output of the `ls` should be redirected to a file named `output` in the current working directory. If the output file exists before you run your program, you should simple overwrite it (after truncating it). The exact format of redirection is a command (and possibly some arguments) followed by the redirection symbol followed by a filename. Multiple redirection operators or multiple files to the right of the redirection sign are errors. Note: don't worry about redirection for built-in commands.

Sometimes, a shell user prefers to run a program and have it read from a file rather than from the keyboard (stdin). Usually, a shell provides this nice feature with the < character. Formally this is named as redirection of standard input. To make your shell users happy, your shell should also include this feature. For example, if a user types `sort < input`, then the sort program should be run with the file named `input` replacing stdin. Likewise, the command `sort < unsorted > sorted`, would invoke the `sort` program to read from the file named `unsorted` and write the sorted data to the file named `sorted`.

4.2 Parallel Commands

Your shell will also allow the user to launch multiple commands in parallel. This is accomplished with the ampersand operator as follows:

```
mish> cmd1 & cmd2 args1 args2 & cmd3 args1
```

In this case, instead of running `cmd1` and then waiting for it to finish, your shell should run `cmd1`, `cmd2`, and `cmd3` (each with whatever arguments the user has passed to it) in parallel, before waiting for any of them to complete. Then, after starting all such processes, you must make sure to use `wait()` or `waitpid()` to wait until *all* of them them to complete. After all processes are done, return control to the user as usual (or, if in batch mode, move on to the next line). You must also be able to perform input and output redirection on individual parallel commands.

4.3 Pipes

Your shell will also allow the user to launch multiple commands in parallel and connect the output of each command to the input of the next command using a pipe. This is accomplished with the pipe operator as follows:

```
mish> cmd1 | cmd2 args1 args2 | cmd3 args1 > result
```

In this case, your shell should create two unnamed pipes, *a* and *b*, then fork to create the three processes, as if it were doing parallel commands. Before calling `exec`, each child will replace its stdin and/or stdout with the correct pipe read or write descriptor. In this case, note that the final output is redirected to a file. Likewise, the initial input to the first command could use input redirection. All of the commands run in parallel, as before, but now data passes from the first process to the second, and so on. After starting all such processes, you must make sure to use `wait()` or `waitpid()` to wait until *all* of them them to complete. After all processes are done, return control to the user as usual (or, if in batch mode, move on to the next line).

5 Program Errors

When errors are encountered, you should print an informative message. The `perror()` function in the C standard library can be helpful for this. Your shell should produce an error message whenever it encounters an error of any type. The error message should be printed to stderr. After most errors, your shell simply continue processing. However, if the shell is invoked with more than one file, or if the shell is passed a bad batch file, it should exit by calling `exit(1)`.

Your shell must enforce the syntax specified in this project assignment. If the syntax of the command looks perfect, you simply run the specified command. Otherwise, print an informative message. If there are any program-related errors (e.g., invalid arguments to `ls` when you run it, for example), the shell does not have to worry about that. The program will print its own error messages and exit.

6 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. Next, add built-in commands. Then, try working on redirection. Finally, think about parallel commands and pipes. Each of these requires a little more effort on parsing, but each should not be too hard to implement.

At some point, you should make sure your code is robust to white space of various kinds, including spaces and tabs. In general, the user should be able to put variable amounts of white space before and after commands, arguments, and various operators. However, the operators (redirection and parallel commands) do not require whitespace. 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. It's also 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 different inputs at it and make sure the shell behaves well. Good code comes through testing; you must run many different tests to make sure things work as desired. Don't be gentle – other users certainly won't be. Finally, 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.

Make sure you compile your program as follows:

```
mish>  gcc -o mish -Wall -Werror -O mish.c
```

There should not be any error messages and warning during compilation. Create your own test cases and test thoroughly. Your program may end up creating too many processes than intended due to a bug (fork bomb). To prevent your self to be locked out of the system limit the number of processes you can create by adding this line to your .bashrc file in your home directory.

```
ulimit -u 100
```

In case you reach the limit of 100, then login through a different terminal and kill mish and all the processes created by mish. Use commands `ps -u` to find its process ID and `kill <pid>` to kill it.

7 Extra Credit

If you satisfactorily complete the shell as specified above, you can get extra credit by implementing command line completion, command history, or other features of a modern shell.