P3: Shell

Due Tuesday by 11:59pm **Points** 84

Objectives

1. To gain yet more familiarity with programming in C and in Linux
2. To learn how processes are handled with fork(), exec(), and wait()
3. To gain exposure to some of the functionality in shells
4. To understand how redirection of stdout works

1. Overview

This project is to be performed alone. Make sure you read this entire specification before beginning.

In this assignment, you will implement a **command line interpreter**, or **shell,** on top of Linux. 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. More specifically, shells are typically implemented as a simple loop that waits for input and `fork()`s a new child process to execute the command; the child process then `exec()`s the specified command while the parent process `wait()`s for the child to finish before continuing with the next iteration of the loop.

The shell you implement will be similar to, but much 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 `sh` or the shell you are running to learn more about all of the functionality that can be present. For this project, you do not need to implement as much functionality as is in most shells.

Besides the most basic function of executing commands, your shell (`mysh`) must provide the following three features: interactive vs. batch mode, output redirection, and aliasing.

2. Features

2.1) Modes: Interactive and Batch

Your shell can be run in two modes: **interactive** and **batch**. The mode is determined when your shell is started. If your shell is started with no arguments (i.e., `./mysh`) , it will run in interactive mode; if your shell is given the name of a file (e.g., `./mysh batch-file`), it runs in batch mode.

In interactive mode, you will display a prompt (the string `mysh>` , note the space AFTER the `>` character) to `stdout`) and the user of the shell will type in a command at the prompt.

In batch mode, your shell is started by specifying a batch file on its command line; the batch file contains the list of commands (each on its own line) that should be executed. In batch mode, you should **not** display a prompt. In batch mode, you should echo each line you read from the batch file back to the user (`stdout`) before executing it; this will help you when you debug your shells (and us when we test your shell -- most of which will be of batch mode). If the line is empty or only composed of whitespace, you should still echo it; if it is over the 512-character limit then echo at least the first 512 characters (but you may echo more if you want).

In both interactive and batch mode, your shell terminates when it sees the `exit` command on a line or reaches the end of the input stream (i.e., the end of the batch file or the user types 'Ctrl-D').

2.2) 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 redirects standout output to a file with the `>` character; your shell should include this feature.

For example, if a user types `/bin/ls -la /tmp > output` into your shell, nothing should be printed on the screen. Instead, the standard output of the `ls` program should be rerouted to the file `output`. Note that standard error output should not be changed; it should continue to print to the screen. If the `output` file exists before you run your program, you should simply overwrite it (after truncating it, which sets the file's size to zero bytes).

The exact format of redirection is: a command (along with its arguments, if present), followed by any number of white spaces (including none), the redirection symbol `>`, again any number of white space (including none), followed by a filename.

Special cases: Multiple redirection operators (e.g. `/bin/ls > > file.txt`), starting with a redirection sign (e.g. `> file.txt`), multiple files to the right of the redirection sign (e.g. `/bin/ls > file1.txt file2.txt`), or not specifying an output file (e.g. `/bin/ls >`)are all errors. Print exactly: `Redirection misformatted.\n`. If if the output file cannot be opened for some reason (e.g., the user doesn't have write permission or the name is an existing directory), your shell should print exactly `Cannot write to file foo.txt.\n`. In these cases, do not execute the command and continue to the next line.

Do not worry about redirection for built-in commands (`alias`, `unalias`, and `exit`); we will not test these cases.

2.3) Aliases

Many shells also contain functionality for aliases. To see the aliases that are currently active in your Linux shell, you can type `alias`. Basically, an alias is just a short-cut so that the user can type in something simple and have something more complex (or more safe) be executed.

For example, you could set up:

```
mysh> alias ls /bin/ls
```

so that within this shell session, the user can simply type `ls` and the executable `/bin/ls` will be run.

Note that alias is an example of a "built-in" command. A built-in command means that the shell interprets this command directly; the shell does not `exec()` the built-in command and run it as a separate process; instead, the built-in command impacts how the shell itself runs.

There are three ways that alias can be invoked in your shell.

If the user types the word `alias`, followed by a single word (the **alias-name**), followed by a replacement string(s), you should set up an alias between the alias-name and the value (e.g. `alias ll /bin/ls -l -a`). (Special cases: If the alias-name was already being used, just replace the old value with the new value.

If the user simply types `alias`, your shell should display all the aliases that have been set up with one per line (first the alias-name, then a single space, and then the corresponding replacement value, with each token separated by exactly one space).

If the user types `alias` followed by a word, if the word matches a current alias-name, print the alias-name and corresponding replacement value, with each token separated by exactly one space; if the word does not match a current alias-name, just continue.

The user can also unalias alias-names; if the user types `unalias <alias-name>` you should remove the alias from your list. If `<alias-name>` does not exist as an alias, just continue. If the user does not specify `<alias-name>` or there are too many arguments to unalias then print `unalias: Incorrect number of arguments.\n` and continue.

You should be able to handle an arbitrary number of aliases.

You do not need to worry about aliases to other aliases, aliases that involve redirection, or redirection of aliases. There are only three words that cannot be used as alias-names: `alias`, `unalias`, and `exit`. For example, if the user types `alias alias some-string`, `alias unalias some-string`, or `alias exit some-string`, your shell should print to `stderr` `alias: Too dangerous to alias that.\n` and continue.

To actually use an alias, the user types the alias just as they would type any other command:

```
mysh> alias ll /bin/ls -l
mysh> ll
```

Running an alias with additional arguments (e.g. `ll -a` where `ll` is an alias-name) is undefined behavior. We will not test this. All alias calls will consist of only the alias-name.

3. Program Specifications

Your C program must be invoked exactly as follows:

```
mysh [batch-file]
```

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

- batch-file: an **optional** argument. If present, your shell will read each line of the batch-file for commands to be executed. If not present, your shell will run in interactive mode by printing a prompt to the user at `stdout` and reading the command from `stdin`.

For example, if you run your program as

```
mysh file1.txt
```

then your program will read commands from `file1.txt` until it sees the `exit` command.

Defensive programming is an important concept in operating systems: an OS can't simply fail when it encounters an error; it must check all parameters 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 an understandable error message and either continue processing or exit, depending upon the situation.

You should consider the following situations as errors; in each case, your shell should print a message using `write()` to `STDERR_FILENO` and **exit** gracefully with a return code of 1:

- An incorrect number of command line arguments to your shell program. Print exactly `Usage: mysh [batch-file]\n` (with no extra spaces)
- The batch file does not exist or cannot be opened. Print exactly `Error: Cannot open file foo.\n`(assuming the file was named `foo`).

For the following situation, you should print a message using `write()` to `STDERR_FILENO` and **continue** processing:

- A command does not exist or cannot be executed. Print exactly `job: Command not found.\n` (assuming the command was named `job`).

Optionally, to make coding your shell easier, you may print an error message and continue processing in the following situation (you can handle this however you want, as long as you can continue processing with the next line):

- A very long command line (for this project, over 512 characters).

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

- An empty command line.
- White spaces include tabs and spaces.
- Multiple white spaces on an otherwise empty command line.
- Multiple white spaces between command-line arguments, including before the first command on a line and after the last command.

- Batch file ends without `exit` command or user types 'Ctrl-D' as a command in interactive mode.

All of these requirements will be tested!

We will not test the `exit` command with extra arguments.

4) C Implementation Hints

This project is not as hard as it may seem at first reading; in fact, the code you write will be much, much smaller than this specification. Writing your shell in a simple manner is a matter of finding the relevant library routines and calling them properly. Your finished programs will probably be under 500 lines, including comments. If you find that you are writing a lot of code, it probably means that you are doing something wrong and should take a break from hacking and instead think about what you are trying to do.

Your shell is basically a loop: it repeatedly prints a prompt (if in interactive mode), 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 ends their input.

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 easy concurrency; that is, multiple commands can be started and allowed to execute simultaneously (although we won't be using that easy concurrency in this project).

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 (using the Unix command `man`). You will also find man pages useful for seeing which header files you should include.

4.1) Printing

You may use the `write()` system call for all printing (including prompts, error messages, and job status), whether to `stdout` or to `stderr`. Why should you use `write()` instead of `fprintf` or `printf`? The main difference between the two is that `write()` performs its output immediately whereas `fprintf()` buffers the output temporarily in memory before flushing it. As a result, if you use `printf()` you will probably see output from your shell intermingled in unexpected ways with output from the jobs you `fork()`; you will fail our tests if your output is intermingled. If you decide to use `printf()` make sure you ALWAYS call `fflush()` immediately after the call to `printf()`.

4.2) 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.) You may find the `strtok()` routine

useful for parsing the command line (i.e., for extracting the arguments within a command separated by ' '). A warning about `strtok()`: It modifies the input string, and the char pointers it returns point to various indices within the input string. You may find `strdup()` handy if you want to preserve the input string, but be careful about memory leaks.

4.3) Executing Commands

Look into `fork()`, `execv()`, and `waitpid`. You will note that there are a variety of commands in the exec family; **for this project, you must use** `execv()`.

The fork system call creates a new process. After this point, two processes will be executing within your code. You will be able to differentiate the child from the parent by looking at the return value of fork; the child sees a 0, the parent sees the pid of the child.

Note that `execv()` does not search your `PATH` environment variable and so `mysh` will require that full paths are specified for all commands. For example, unless the executable `ls` exists in the directory you started `mysh` in, calling `execv()` with `ls` will fail, but calling `execv()` with `/bin/ls` should succeed. (If you want to find where an executable lives in your `PATH` so that you can specify its full path within `mysh`, you can probably use `which` in the shell you normally use. For example, `which ls` returns `/bin/ls`. On a related note, you will notice that `cd` will not work within your shell. `cd` is a built-in command for most shells that simply changes an environment variable reflecting the "current working directory" for all the processes that are created by this shell.)

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). In this case, the child process should call `_exit()` to terminate itself (see more below). 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:

```
/bin/foo 205 535
```

then `argv[0] = "/bin/foo"`, `argv[1] = "205"` and `argv[2] = "535"` (where each string is null-terminated). Note the list of arguments must also 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! You should be able to handle up to 99 tokens for each shell command - 100 including the necessary NULL terminator.

The `waitpid()` system call allows the parent process to wait for one of its children. Your shell must wait for its child process to finish before it prints the next prompt or executes the next command

4.4) Standard Output Redirection

You can elegantly perform output redirection in your shell by taking advantage of several properties:

- Applications use the known file descriptor, `stdout`, to write to standard output (`stdout` has the value 1)
- Your shell can set up the file descriptors in the child process (after `fork()`) as desired (see below) and then those file descriptors remain unchanged in the child process after the call to `execv()`

Thus there are two ways your shell can setup the file descriptors for redirection.

The first approach relies on the fact that Linux uses the lowest available file descriptor for a process when it opens a new file (this is guaranteed). Thus, by closing `stdout` and then immediately opening another file, `foo.txt`, your shell is guaranteed that the file descriptor returned for accessing `foo.txt` is the same as what `stdout` used to be (i.e., 1). Thus, when this process writes to `stdout` (or the file descriptor 1), it will actually write to the file `foo.txt`.

The second approach uses the system call `dup2()` so that two different file descriptors can point to the same open file and can be used interchangeably. Read the man pages for more details!

4.5) File Descriptors and Fork

When you call `fork()` the open files from the parent process are also accessible to the child process. So if your batch file (or even `stdin`) is open at the time `fork()` is called, the child process will also have access to the file. This can become a problem if the call to `execv()` in the child process fails. In this case you will need to call `exit()` from the child process so that it terminates instead of having two instances of the shell running at once. The slight problem with `exit()` as implemented by glibc (the C standard library used on the lab machines) is that it calls `lseek()` on open files before closing them (this effectively rewinds the file so it starts reading from the beginning). Since the parent and child share access to the file, the parent will be affected by this call, and may continue to read it from the beginning, creating an infinite loop. There are at least two possible solutions:

- Use `_exit()` instead of `exit()` which does not perform the file cleanup before terminating
- Use `fclose()` to close the file in the child process before calling `exit()`

4.6) 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 focus on interactive mode, and get a single command working (probably first a command with no arguments, such as `ls`); then make sure you can handle commands arguments (e.g., `ls -l`). Then, add in the functionality to work in batch mode (most of our test cases will use batch mode, so make sure this works!). Handling file redirection is probably your next step, followed by aliases. Finally, make sure that you are correctly handling all of the cases where there is miscellaneous white space around commands or missing commands.

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.

You will have to implement some sort of data structure for your aliases. We recommend using a doubly-linked LinkedList because it allows for easy addition and removal. For your future benefit, it may be worth implementing a generic LinkedList.

5. Grading

To ensure that we compile your C correctly, 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. Otherwise, you can start with the sample `Makefile` available here:

```
~cs537-1/projects/shell/Makefile
```

The first rule of your Makefile should build `mysh` so that running `make` without any arguments will build `mysh`.

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

```
./mysh
./mysh inputTestFile
```

Copy all of your `.c` (and `.h`) source files into the appropriate handin directory. Do **not** submit any `.o` files. Make sure that your code runs correctly on the linux machines in the 13XX labs.

We have provided test cases to help you test your code. Use this command to list all the test cases: `~cs537-1/tests/p3/list-tests.sh` Use this command to run all the test cases (including the linter and Valgrind): `~cs537-1/tests/p3/run-tests.sh` (The `run-tests.sh` command should be run in the directory containing your code and Makefile.)

The majority of your grade for this assignment will depend upon how well your implementation works. We will run your program on a suite of about 20 test cases. 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.

We will again verify that your code passes lint and valgrind tests, as in Project 1.

6. Handing in your Code

- **Handing it in: Copy your source files to `~cs537-1/handin/login/p3/ontime` where login is your CS login.** Do NOT use this handin directory for your workspace. You should keep a separate copy of your project files in your own home directory and then simply copy the relevant files to this handin

directory when you are done. The permissions to this handin directory will be turned off promptly when the deadline passes and you will no longer be able to modify files in that directory.

- You can use up to 3 slip (late) days across all the projects throughout this semester; for example, you can use 1 slip day on three separate assignments or 3 slip days on a single assignment (or other combinations adding up to a total of 3 days). If you are using slip days, you must create a file called `slip_days` with the full pathname `~cs537-1/handin/login` `/p3/slip_days`. The file should contain a single line containing the integer number of slip days you are using; for example, you can create this file with `echo 1 > slip_days`. You must then copy your code into the corresponding slip directory: `~cs537-1/handin/login/p3/slip1`, `~cs537-1/handin/login/p3/slip2`, *or* `~cs537-1/handin/login/p3/slip3`.