Project #1 - RUSH

Due Feb 14 by 11:59pm **Points** 100 **Available** Jan 25 at 1pm - Feb 14 at 11:59pm

This assignment was locked Feb 14 at 11:59pm.

Rapid Unix SHell

In this project, you'll build a simple, efficient Unix shell. A shell is a *command line interpreter (CLI)* that operates in the following way: when you type in a command (in response to a prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when the child process has finished

Program Specification

Your basic shell, called `rush` (Rapid Unix SHell), is basically an interactive loop: it repeatedly prints a prompt `rush>` (note the space after the greater-than 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 `exit`. The name of your final executable should be `rush`.

The shell must be invoked with no arguments:

```
prompt> ./rush
rush>
```

At this point, `rush` is running, and ready to accept commands.

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's something called the shell **path**; more on this below).

Structure

1. Basic Shell

The shell runs in a while loop, repeatedly asking for input to tell it what command to execute. It then executes that command. The loop continues indefinitely, until the user types the built-in command `exit`, at which point it exits. Once you print the prompt `rush>` to the standard output, use `fflush()` to make sure your solution prints everything in the expected order.

For reading lines of input, you should use `getline()`. All command lines used for testing your program will be provided in a single line with at most 255 characters. To parse the input line into constituent pieces, you might want to use `strsep()`. Read the man page for more details.

To execute commands, look into `fork()`, `exec()`, and `wait()`/`waitpid()`. Read the man page for more details. If `fork()` fails, report as an error.

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()` library function call to run a command. 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. Read the man page for more details.

2. Paths

In our example above, the user typed `ls` but the shell knew to execute the program `/bin/ls`. How does your shell know this?

It turns out that the user must specify a **path** variable to describe the set of directories to search for executables; the set of directories that comprise the path are sometimes called the *search path* of the shell. The path variable contains the list of all directories to search, in order, when the user types a command.

Important: Note that the shell itself does not implement `ls` or other commands (except built-ins). All it does is find those executables in one of the directories specified by `path` and create a new process to run them.

To check if a particular file exists in a directory and is executable, consider the `access()` system call. For example, when the user types `ls`, and path is set to include both `/bin` and `/usr/bin`, try `access("/bin/ls", X_OK)`. If that fails, try "/usr/bin/ls". If that fails too, report as an error.

Your initial shell path should contain one directory: `/bin`

Note: You **do not** have to worry about **absolute paths** (`/bin/ls`) or **relative paths** (`./main`) in the program name. The location of the programs (e.g. `ls` and `main`) will only be specified through the shell path in the test cases.

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 must invoke your implementation of the built-in command. For example, to implement the `exit` built-in command, you simply call `exit(0);` in your source code, which then will exit the shell.

In this project, you should implement `exit`, `cd`, and `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 arguments to `exit`.
- `cd`: `cd` always take 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.
- `path`: The `path` command takes 0 or more arguments, with each argument separated by whitespace from the others. A typical usage would be like this: `rush> path /bin /usr/bin`, which would add `/bin` and `/usr/bin` to the search path of the shell. If the user sets path to be empty, then the shell should not be able to run any programs (except built-in commands). The `path` command always overwrites the old path with the newly specified path.

4. 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.

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

If the `output` file exists before you run your program, you should simple overwrite it (destroy previous content). If the file cannot be opened, report as an error.

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 arguments to the right of the redirection sign are errors.

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

5. Parallel Commands

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

```
rush> 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 for them to complete. After all processes are done, return control to the user as usual.

Note #1: Parallel commands can use redirection.

Note #2: Do not worry about built-in commands being used in parallel with other commands (built-in or not).

6. Program Errors (in blue in the specification)

The one and only error message. 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), as shown above. Once you print the error message, use `fflush()` to make sure your solution prints everything in the expected order.

After most errors, your shell simply *continue processing* after printing the one and only error message. However, if the shell is invoked with any argument, it should exit by calling `exit(1)`.

Note: Empty command lines are not errors.

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 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 (rather, the program will print its own error messages and exit).

7. Other requirements

Make sure your code is robust to white space of various kinds, including spaces (`' '`) and tabs (`'\t'`). The user should be able to put variable amounts of white space before and after commands, arguments, and various operators; even command-line operators (redirection and parallel commands) must be separated from other arguments through white spaces.

8. Miscellaneous Hints

Do not try to solve the entire project at once. Start from easier functionalities, and make sure they work before adding new functionalities (e.g. no need to worry about parallel commands if you can't even run one command yet).

Test your own code! Throw lots of different inputs at it and make sure the shell behaves well.

Submission instructions

The correctness of your submissions will be automatically evaluated by Gradescope. You must prepare a file named `rush.zip` containing a folder named `src`. This folder must contain your source code in C language (other languages are not allowed) and a `Makefile` to build the executable named `rush`. This code must be buildable and runnable in a Linux environment with a recent version `gcc` (more precisely, gcc version 11.4.0 with Ubuntu 11.4.0-1ubuntu1 22.04). This is the sequence of commands that will be executed by Gradescope to build your code:

```
$ unzip rush.zip
$ cd src
$ make
$ if [ -f rush ]; then echo SUCCESS; fi
SUCCESS
$
```

Make sure this sequence works in your testing environment. You must submit your zip file through Gradescope!

Test cases

In the test cases below, the prompt from the original shell is shown as `$` while the prompt for your shell is shown as `rush>`.

Example #1

```
$ rush with arguments
An error has occurred
$
```

Example #2

```
$ rush
rush> exit
$
```

Example #3

```
$ rush
rush> ls
list
of
philes
rush> ls > tmp.txt
rush> ls > tmp.txt > tmp.txt
An error has occurred
rush> ls
list
of
philes
tmp.txt
rush>
rush> exit with arguments
An error has occurred
rush> exit
$
```

Example #4

```
$ rush
rush> ls /abc
ls: /abc: No such file or directory
rush> ls & ls
list
list
of
philes
of
philes
rush> exit
$
```