Assignment 2

Writing a Basic Command Shell

Author: Brian Crites, Mike Izbicki

This project must be done in a group of two

Coding Requirements

Develop a command shell called rshell in C++ which is capable of performing the following steps:

1. Print a command prompt (e.g. `$`)
2. Read in a line of command(s) (and connector(s)) from standard input
3. Execute the appropriate commands using `fork`, `execvp`, and `waitpid`

Commands will have the following format (note that the square brackets represent optional portions of an input):

```
$ executable [argumentList] [connector] [executable] ...
```

Where there can be any number of commands (which are composed of executables and argument lists) separated by either `||`, `&&` or `;` which are the only valid connectors. The executable can be any executable program located at one of the PATH environment variable locations and the [argumentList] is a list of zero or more arguments separated by spaces.

> **Note**: The [argumentList] can also be surrounded by quotation marks which you must account for. For instance, the command `echo "hello && goodbye"` would have the `"hello && goodbye"` be the [argumentList] even though without the quotes the `&&` would separate the `goodbye` into its own executable.

You will use the execvp command to run the executable from one of the `PATH` locations. The connector is an optional way you can run multiple commands at once. If a command is followed by `;`, then the next command is always executed; if a command is followed by `&&`, then the next command is executed only if the first one succeeds; if a command is followed by `||`, then the next command is executed only if the first one fails.

> **Note**: you may have questions about how a specific command or set of commands should execute. If you are unsure how a certain combination of true/false executions and connectors will function you should test this on hammer.

Note: Most bash commands are actually executables located in a directory list in the PATH environment variable such as `/bin`, `/usr/bin/` (e.g. `ls`), but some commands are built-in to bash (e.g. `cd`). So while the `ls` command should "just work" in your shell when using execvp, the `cd` command won't and isn't required to for the assignment. Only commands that can be executed through a PATH directory (you can view colon separated list of these directories by running `echo $PATH`) need to be accounted for in this assignment.

The connectors do not impose any precedence and the command line should be execute from left to right. For example:

```
$ ls -a
$ echo hello
$ mkdir test
```

is equivalent to:

```
$ ls -a; echo hello; mkdir test
```

Note: you can assume that there will always be a space after the semi-colon (`;`) connector and before and after the and (`&&`) and or (`||`) connectors.

You are required to use the **composite pattern** when representing the commands and operators in your program. There should be no limit to the number of commands that can be chained together using the connections, and your program must be able to handle any combination of operators. For example, you should be able to handle the command:

```
$ ls -a; echo hello && mkdir test || echo world; git status
```

When executing a line of commands, you will need to account for the following requirements:

1. Execute the commands using the syscalls fork, execvp, and waitpid. Previous cs100 students created two video tutorials a fun cartoon tutorial as well as a more serious explanation, and should refer to the man pages for detailed instructions.
2. Every time you run a syscall, you must check for an error condition and if an error occurs then call perror. For examples on when, how, and why to use perror, see this video tutorial or the official man page.
3. You must have a special built-in command of `exit` which exits your program. This command should also adhere to the connector rules when deciding when/if it should be executed.
4. Anything that appears after a `#` character in the line should be considered a comment. For example, in the command `ls -lR /`, you would execute the program `/bin/ls`

(performed using execvp) passing into it the parameters `-lR` and `/`. But in the command `ls # -lR /`, you would execute `/bin/ls`, but you would not pass any parameters because they appear in the comment section. You should also note that the `#` may or may not be followed by a space before the comment begins

Note: Building the parser for this program is one of the most important parts. There are many ways accidentally introduced bugs into your program and you will be adding more parsing features in future assignments. You can create your own parsing function from scratch or use the strtok function from the C standard libraries or the Tokenizer class provided in the boost library.

Note: Before you start adding any documents or code through a command line interface, make sure that you have correctly setup the git client to attribute your commits to your account (you can get a refresher on how to do this in Lab 01). You will need to do this on **every machine** that you use command line git on for this project.

Important Notes for Hammer

The hammer server is provided by the CS department as a standard environment which you can use to test your code in the same environment that it will be graded in. However, because everyone in the class is using this server to run their assignments there are some imposed limitations around the number of processes you can run to keep the server running well. Please take **very special note** of the following advisories when using the hammer server.

- **No Graphics**: because hammer is a true server you will not have access to any applications which require graphics such as web browsers or graphical editors.
- **One SSH Session**: because the hammer server limits the number of processes you can run at any given time if you are connecting to hammer through multiple sessions (ssh, putty, etc.) this will limit the number of other processes you can run. For this reason you should only use a single connection, if you need multiple windows there are other methods (screen, sleeping processes, etc.) to navigate the system better without multiple connections. You should also properly close your SSH connection so that they do not continue to live on the server side. This is typically done by executing `exit` on the command line you are connected to.
- **Be Careful when Forking**: each time you execute a fork() function you are creating a new process which will execute until it terminates (via exit() or some other program termination). One very common issue is for students to create fork() program that do not properly exit and instead build up over time until they use up all the processes you are alloted on hammer, at which point you will not be able to ssh or perform other normal programs (because you have no available processes).

In order to check for this issue you can use the command line command `ps -uxwwf` and look for a buildup of processes (typically named after the program you are executing like rshell). You can then destroy any accidental processes using the command `kill -KILL <PID>` where you replace `<PID>` with the PID number that displays when you run `ps -uxwwf` .

If your session gets really stuck, you can use `killall -KILL -u <username>` to kill all the processes you own (you must use your own username) which will disconnect you and kill all the processes you have running. Note that by killing all the processes you have running you will lose any changes from files you were editing on the server.

Project Structure

You must have a directory called `src/` which contains all the source (.cc/.cpp) files for the project. For header files you may either have a folder `header/` which contains your header files or you may keep them in the `src/` directory. You must have a `unit_tests/` directory which should contain all the unit tests you've written using the Google Unit Test framework (the main for the unit tests can be located either in `unit_tests/` or `src/`). You must also have an `integration_tests/` directory which should contain all the integration tests you've written using bash. We recommend using IO redirection for your bash integration tests, and automated validation is not necessary.

Your root directory must have a `CMakeLists.txt` with two targets. The first target is named `rshell` and should build the main executable and the second should be `test` which runs the unit tests that you have created using the Google Unit Test framework.

> **Note:** The file/directory names above are a standard convention. You must use the exact same names in your project, including capitalization. We utilize these names when performing steps of our automated grading process, so any deviation may result in missing points.

The google test code for your unit tests can have any name as long as the files reside in the `unit_tests/` directory and an executable is generated named test in the root directory. The integration test shell scripts that you develop and place into the `integration_tests/` directory must have the following names:

`single_command_tests.sh`	tests primarily for command executions
`multiple_commands_tests.sh`	tests primarily for command and connectors interaction
`commented_command_tests.sh`	tests primarily for comments being respected
`exit_command_tests.sh`	tests primarily for proper exit functionality

Your project should not contain any files not necessary for the project. This includes things like CMake temporary build files used for building the project on your machine such as `CMakeCache.txt` and the `CMakeFiles/` directory as well as executables. We have provided a `.gitignore` in the template repository which will stop may of these files from showing as untracked when you run git status, and you should extend this file with additional temporary and machine specific files.

Submission Instructions

You will also need to add an **annotated** `hw2` tag to the commit that you want to be graded. Annotated tags are described in lab 2 and in the git documentation. `git push` will not automatically push tags to your repository. Use `git push origin hw2` to update your repository to include the `hw2` tag. If you need to update your tag, you can remove the old tag using `git push --delete origin hw2` and push an updated one to your repo.

> **Note**: We will test your project on the hammer server, so while it is not required that you develop on hammer you should verify that your project builds and functions properly on hammer.cs.ucr.edu.

Your project must also contain a names.txt file in the root directory which contains the name, SID, and email of each of the partners in your group. It should have the following format.

```
Brian Crites, 860XXXXXX, bcrit001@ucr.edu
Andrew Lvovsky, 860XXXXXX, alvov001@ucr.edu
```

Do not wait to push your assignment to Github until the project due date. You should be committing and uploading your assignment continuously. If you wait until the last day and can't figure out how to use git properly, then you will get a zero on the assignment or be forced to use the course late policy. No exceptions.

For late assignment submissions, a penalty of 10% will be deducted for every day the assignment is late up to a maximum of 3 days, with exceptions only for documented emergencies. For clarity, this means that an assignment that is 3 days late would incur a 30% penalty to the grade **assessed**. Please submit this form along with your late submission. Otherwise, the team will receive a zero on that assignment.

In addition to the above submission requirements you must update your `README.md` file so that it specifies the current structure of your program. This is especially important since your final program will likely be much different than what you originally designed in Assignment 1.

Testing Your Submission

Your assignment 2 should have approximately the following structure, see the project structure section for more details:

example-project-directory/

```
├── CMakeLists.txt
├── README.md
├── googletest
├── header
        ├── header_file_1.h
        └── header_file_2.h
├── src
        ├── src_file_1.cpp
        └── src_file_2.cpp
└── unit_tests
        ├── test_file_1.cpp
        └── test_file_2.cpp
└── integration_tests
        ├── single_command_tests.sh
        ├── multiple_commands_tests.sh
        ├── commented_command_tests.sh
        └── exit_command_tests.sh
```

You should also have a .gitignore and a .gitmodules hidden file in your root directory. Your root directory should contain no C++ header or source files and your googletest directory should be added to your project as a submodule. While the googletest directory is not required to be present in the root directory it is suggested.

When testing your project we will run (approximately) the following commands on hammer:

```
git clone <assignment-repo-url>
cd <assignment-repo-url>
git checkout tags/hw2
git submodule init
git submodule update
cmake3 .
make
test -e rshell || echo “rshell executable missing, check submission instruction
section of the specifications”
test -e test || echo “test executable missing, check submission instruction
section of the specifications”
```

```
test -d unit_tests || echo "unit_tests/ directory missing, check submission instruction section of the specifications"
test -d integration_tests || echo "integration_tests/ directory missing, check submission instruction section of the specifications"
```

These commands should generate a `test` executable as well as an `rshell` executable in the root directory, otherwise will print that they are missing. **Please run these commands on hammer with a brand new clone** of your GitHub repository to ensure your shell compiles and builds properly before submission.

Collaboration Policy

- You **may not** copy any code found online, doing so will result in failing the course.
- You **may not** look at the source code of any other student.
- You **may** discuss with other students in general terms how to use the unix functions.
- You are **encouraged** to talk with other students about test cases and are allowed to freely share ideas in this regard.

Grading

10	Sufficient Unit Test Cases
10	Sufficient Integration Test Cases
10	Updated README.md
20	Single Command Execution
20	Command and Connectors Execution
20	Command with Comments Execution
10	Commands with Exit Execution
100	**Total**

Note: Your project structure and executable names are not explicitly listed in the grading schedule above but not following the specified file names and location may result in losing points because the automated grading system is unable to process your repository. Loss of points due to an issue specified above are final and failure to build and compile will result in a zero for the assignment.