CS 246E Fall 2021 — Tutorial 1

September 8, 2021

Summary

1 What is a Shell?

A shell is any interface which allows you to interact with your operating system. The graphical user interface for any operating system you've used is a shell. Furthermore, if you have ever used the "command prompt," or "terminal" on your OS before, what you are using is a command-line shell.

In CS246E, we'll be using a specific command line shell called Bash - the Bourne Again Shell, but some scripts you write may also need to be compatible with sh, the Bourne Shell (which has less features).

2 Some Bash Commands

Command-line shells predate modern GUIs for operating systems, but why would we want to use such old technology? The answer is the same as why we write programs to automate tasks for us rather than doing them manually: command-line shells tend to be excellent at doing repetitive tasks with relatively little user guidance, whereas GUI shells are generally not.

At its core, Bash relies on a large number of programs, some of which you've seen before, some which you've seen before, some which you'll see in this tutorial, and many more which you're welcome to discover on your own.

(Note: Almost every command which accepts input from a file will read that input from standard input if a file is not provided)

Some example commands:

- `cat`: prints the contents of a list of files to standard output

  ```
  $ cat file.txt
  These are the contents of file.txt
  ```

  `cat` is useful to quickly view the contents of a file, to concatenate the contents of multiple files together (e.g. `cat file1.txt file2.txt`), or as a very simple text editor (when standard output is redirected to a file).

- `echo`: prints whatever you give it as output
  ```
  $ echo Hello World
  Hello World
  ```
- `ls`: lists all files in the current directory
  ```
  $ ls
  file1 file2 file3
  $ ls file1 file3
  file1 file3
  ```
- `mv`: renames or moves a file
  ```
  $ ls
  file1
  $ mv file1 file2
  $ ls
  file2
  ```
- `cp`: copies a file
  ```
  $ ls
  file1
  $ cp file1 file2
  $ ls
  file1 file2
  ```
- `rm`: deletes a file
  ```
  $ ls
  file1 file2
  $ rm file1
  $ ls
  file2
  ```
- `touch`: creates a new empty file
  ```
  $ ls
  file1
  $ touch file2
  $ ls
  file1 file2
  ```
- `mktemp`: creates a new empty file with a temporary name and gives you the path to that file
  ```
  $ mktemp
  /tmp/tmp.trUfNzfp
  ```
- `mkdir`: makes a new directory
  ```
  $ ls
  file1
  $ mkdir dir1
  $ ls
  file1 dir1
  ```
- `diff`: compares two files and lists any differences
  ```
  $ diff file1 file2
  1c1
  < hello
  ---
  > goodbye
  ```
- `pwd`: gives us the path of our present working directory
  ```
  $ pwd
  /u/jdoe/cs246e/examples/
  ```
- `wc`: counts the number of words, characters, and/or lines in a file
  ```
  $ cat hello.txt
  Hello World
  $ wc -w hello.txt
  2 hello.txt
  ```
- `less`: displays the contents of a file one page at a time, which makes it easier to use than `cat` for large files
- `man`: displays the manual for a command. Each of these commands has many more functions than are listed here, usually specified by special arguments. If you want to see a complete list for a command, use `man`

There are thousands more commands, including `vi` and `emacs` for full-featured text editors, `sed` for regular-expression based text replacement, `find` for complicated searching of the file system, `git` and `svn` for code version control, `clang++` and `g++` for C++ compilers, and so forth.

3 Using the Shell from C++

You can run shell commands easily from within a C/C++ program by including `<cstdlib>` and using the `system` function. `system` actually runs `/bin/sh`, which doesn't have all the features of Bash, but everything we've seen so far will work on both.

```
#include <cstdlib>
int main() {
    // Prints 2 to stdout, assuming helloworld.txt contains "Hello World"
    system( "wc -w helloworld.txt" );
}
```

3.1 Exercise

We're ready to start writing our program. Write a program which accepts a file as a command-line argument and copies it 10 times to files named `copy0`, ..., `copy9`. Use `system` rather than trying to create files using C++ I/O.

4 Exit Statuses and the Test Command

4.1 Exit Statuses

In addition to the output printed by a program to standard out and standard error, every program also returns an *exit status* (or *exit code*) which tells the user whether it was successful or not.

- A code of 0 means "true" or "success"
- A code between 1-255 means "false" or "failure"

The idea is that every successful command is successful in the same way, whereas commands may fail in a number of ways. It is up to each individual program to specify what its exit statuses mean, usually in the program's man pages. For example, `diff` returns 0 if the input files are identical, 1 if they are different, and 2 if an error occurred (such as one file not existing).

The exit status of the most recently run program can be checked using the `$?` shell variable.

4.2 test

The Bash `test` command (which can also be written as paired square brackets) can do all sorts of boolean conditions: see the `man` pages for a comprehensive list. In our case, we'd like to know whether the file the user supplied us as a command line argument actually exists. `test -f` will tell us whether a specifed file exists and is not a directory.

```
$ ls
file1 dir1
$ test -f file1
$ echo $?
0
```

```
$ test -f dir1
$ echo $?
1
$ test -f file2
$ echo $?
1
$ [ -f file2 ] # alternative syntax for test
1
```

4.3 Reading Exit Statuses in C++

You may have noticed that `system` returns an int. However, it doesn't just return the exit status: it also returns some other information we don't care about. The `WEXITSTATUS` function in `<sys/wait.h>` can convert `system`'s return value into an exit status that we can work with in C++.

4.4 Exercise

Using what we have learned so far, verify that the user supplied a path to a regular file which exists. If they did not, print a helpful error message and exit.

5 Using the Results of Commands in Other Commands

You've already seen how we can redirect input from and output to files in the shell. However, sometimes we want to get command-line arguments or input from another command, or equivalently we want to redirect a command's output to be sent to another command.

5.1 Piping

The most basic way to do this is piping. A pipe ties the output of a command to the input of another, so that the first program's output can be read as input by the second program. The syntax is a pipe (|), hence the name.

```
$ echo "Hello World" | wc -w
2
```

5.2 Piping in C

We can also open a pipe in a C program, where one end of the pipe is our program and the other end is another program. The `popen` and `pclose` functions can be used to do this: their syntax is the same as `fopen` and `fclose`, except that the first argument to `popen` should be a shell command rather than a file name. `popen` then returns a `FILE *` which can be used in the same way that the `FILE *` from `fopen` is used.

```
#include <cstdio>
```

```
#include <iostream>

int main() {
    FILE* f = popen( "wc -w helloworld.txt", "r" );
    int words;
    fscanf( f, "\%d", words );
    pclose( f );
    std::cout << words << std::endl;
    return 0;
}
```

5.3 Using Program Output as a Command-line Argument

We can also use the results of a program's output anywhere in Bash we could use a string by using the special `$()` syntax. Simply place the command you wish to execute between the parentheses.

```
$ cat args.txt
-w file.txt
$ cat file.txt
Hello from file.txt
$ wc $(cat args.txt)
3
```

5.4 Exercise

Using what we have learned so far, modify your program so that it copies the specified file until exactly 10 (non-hidden) files total are in the current directory.

6 Suppressing Unwanted Output

Sometimes we don't care about the output of a command: perhaps we know it will succeed, or perhaps we only care about the exit status or a side-effect of the command. One thing we could do is create a temporary file and then delete it:

```
$ mktemp
/tmp/example.file.path
$ mycommand > /tmp/example.file.path
$ rm /tmp/example.file.path
```

However, as you can imagine this gets very cumbersome. Unix provides a better way: there are a number of files in the `devices` folder which behave specially when you interact with them (for example, `/dev/stdin`, `/dev/stdout`, and `/dev/stderr` allow you to interact with I/O as files). In particular, the `/dev/null` file simply ignores whatever is written to it, and can be used as a way to get rid of unwanted output.

```
$ mycommand > /dev/null # Standard output is not printed to screen
```

6.1 Exercise

You now have everything you need to finish your program. Modify your program to use `diff` to find out how many files in the current directory are the same as the supplied file (including the supplied file itself), and copy until there are at least 10. Use `diff`'s exit status to determine whether the files are identical: do not use its output.

You can assume that there are at most 10 files in the current directory, each with a name of length at most 128 and containing no spaces, and no identical file has is named `copy0`,...,`copy9`.