The shell and bash

Rob Hackman and Xiao-bo Li

Winter 2024

University of Alberta

Table of Contents

Operating Systems

You've likely used Mac OS or Windows before, and used a Ubuntu virtual machine in CMPUT 274 - what are the main purposes of an operating system?

Operating Systems

You've likely used Mac OS or Windows before, and used a Ubuntu virtual machine in CMPUT 274 - what are the main purposes of an operating system?

- Manage and provide interfaces for access to your resources (physical hardware)
- Provide environment to run programs safely and securely
- Maintain and provide access to files stored on your computer

What is a Shell?

A shell is a textual interface to allow users to ask their operating system to perform tasks, some example tasks are

What is a Shell?

A shell is a textual interface to allow users to ask their operating system to perform tasks, some example tasks are

- Run a program
- Create, read, or delete a file
- Terminate a program

What is a Shell?

A shell is a textual interface to allow users to ask their operating system to perform tasks, some example tasks are

- Run a program
- Create, read, or delete a file
- Terminate a program

In this course we'll be using Ubuntu as an operating system and the Bourne Again Shell (bash)

You should have multipass or WSL set up from CMPUT 274 to reuse for this term

Sample command

```
$ cat /usr/share/dict/words
```

- `cat` is the program name - we are asking Ubuntu to run the `cat` program
- `/usr/share/dict/words` is a *command-line argument*, additional data we provide which the program expects
  - The operating system passes this command-line argument (which is just a string) to the program for use

The following command would be the same

```
$ cat "/usr/share/dict/words"
```

Sample command

```
$ cat /usr/share/dict/words
```

What does the string `/usr/share/dict/words` represent? This is a filepath - which specifies exactly where a file exists in our operating systems file system.

Sample command

```
$ cat /usr/share/dict/words
```

- In our Ubuntu's file system there is one *root* directory - each file resides somewhere within this directory
- Beginning our path with / specifies that we are starting from that root directory - this is called an *absolute path*
- `/usr` specifies the file `usr` that is immediately within the root directory
- `/usr/share` then specifies the file `share` that is immediately within the `usr` directory
- `/usr/share/dict` specifies the file `dict` that is immediately within the `share` directory
- `/usr/share/dict/words` specifies the file `words` that is immediately within the `dict` directory

Directories

A directory (or folder) is a special type of file that we use to organize other files. In our example on the previous slide `usr`, `share`, and `dict` were all directories.

When working in the shell there is a directory we are currently inside of - that is called the *current working directory*

- The command `pwd` prints out the current working directory
- The command `cd` changes the current working directory
  - If given no arguments sets the current working directory to the logged in users home directory
  - If given a command line argument sets the current working directory to the one specified by the argument

Absolute and Relative Paths

The path `/usr/share/dict/words` was called an absolute path because it started from our root directory.

Any path that does not begin with a forward slash is a relative path and is considered relative to the current working directory

Absolute and Relative Paths

The path `/usr/share/dict/words` was called an absolute path because it started from our root directory.

Any path that does not begin with a forward slash is a relative path and is considered relative to the current working directory

Relative Paths

- `$ cat foo.txt` here `foo.txt` is a relative path to the file with that name in the current working directory
- `$ cat ./foo.txt` the signifier `./` expands to the current directory - so this example is the same as the above
- `$ cat ../foo.txt` the signifier `../` expands to the directory that contains the current directory, so this would be the final `foo.txt` in the directory which contains our current working directory

If the current working directory is `/usr/home/rob/cmput275/slides` can you write the absolute paths each that would be the same as the above relative paths?

Consider `../../../../foo.txt` - what would the absolute path to that file be if the current working directory was as above?

Sample command (again)

The program `cat` expects this command-line argument be provided, and be a path to a file.
What happens if the program `cat` is executed without providing a command line argument?

```
$ cat
```

The program `cat` begins to run and read from the standard input stream. By default, this means it reads what the user types into the keyboard.

I/O Streams

Every program we will run is connected to three *streams*

- standard input (stdin) - by default connected to the text the user types into the keyboard
  - User can signal they are finished entering *all* input by sending the end-of-file (EOF) signal with keys `ctrl+d`
  - When using the `input` function in Python the interpreter would read from `stdin`
- standard output (stdout) - by default connected to the shell's textual output
  - When using the `print` function in Python the interpreter would printing to `stdout`
- standard error (stderr) - by default connected to the shell's textual output
  - Same default location as `stdin` but is not *buffered* (more later)

Redirecting Standard Output

While `stdin`, `stdout`, and `stderr` have locations they are default connected to for execution of any program, the connection can be overridden when asking the operating system to execute a program:

- Redirect `stdout` with the greater than sign (`>`) followed by the filename you'd like to redirect output to
  - Example command:
    `$ wc > outputFile.txt`
  - In the above command when the program `wc` prints to standard output it will send that output to the file `outputFile.txt` instead of the screen
  - The operating system also creates the file `outputFile.txt`, which will overwrite (delete) any existing file with that name in your current location

Redirecting Standard Input

While `stdin`, `stdout`, and `stderr` have locations they are default connected to for execution of any program, the connection can be overridden when asking the operating system to execute a program:

- Redirect `stdin` with the less than sign (`<`) followed by the filename you'd like to redirect input from
  - Example command:
    `$ wc < inputFile.txt`
  - In the above command when the program `wc` reads from standard input it will read that input from the file `inputFile.txt` instead of the screen
  - The operating system opens the file `inputFile.txt` and feeds its contents to the program when it asks for input

Redirecting Standard Error

While `stdin`, `stdout`, and `stderr` have locations they are default connected to for execution of any program, the connection can be overridden when asking the operating system to execute a program:

- Redirect `stderr` with a two followed by the greater than sign (`2>`) followed by the filename you'd like to redirect standard error to
  - Example command:
    `$ wc 2> errorFile.txt`
  - In the above command when the program `wc` prints to standard error it will send that output to the file `errorFile.txt` instead of the screen
  - The operating system also creates the file `errorFile.txt`, which will overwrite (delete) any existing file with that name in your current location

Input versus Command Line Arguments

Consider the following two commands

```
$ wc /usr/share/dict/words
$ wc < /usr/share/dict/words
```

They both count the words in the file `/usr/share/dict/words` but the output is different

This is an important distinction which we'll see more clearly when we start to write our own C and C++ programs to take in command line arguments as well as read input

Input versus Command Line Arguments

In this first example

```
$ wc /usr/share/dict/words
```

The string `/usr/share/dict/words` is passed as a command line argument to the program `wc`

The program `wc` then opens the file `/usr/share/dict/words` and reads its contents (as it has the filepath as a string to be able to open it)

Input versus Command Line Arguments

In this second example example

```
$ wc < /usr/share/dict/words
```

The program `wc` does not receive any additional command line arguments, it then reads from standard input

Since we asked the OS to redirect input from `/usr/share/dict/words` whenever the program `wc` reads from input the operating system instead provides it data from the file

Exit Statuses

When programs finish running they provide an exit status to the operating system

Typically an exit status of 0 indicates no error, and non-zero values indicate errors

However the meaning of exit statuses is entirely up to the author of the program - you can see details about many programs with the manual (`man`) command

```
$ man diff
```

The programs we will write in C and C++ will return exit statuses to the operating system - more later!

Bash Variables - assignment

You can set variables in your shell as well - the syntax is

```
$ x=10
```

Notice there are no spaces between the variable name and the equality sign, nor any spaces between the equality sign and the value the variable is being assigned to

All variables in bash are strings - `x` above is the string that stores 10

Bash Variables - reading values

To read the value of a variable in bash the dollar sign (`$`) is used

```
$ echo $x
$ echo ${x}
```

The curly braces are optional but highly recommended

The text `$x` and `$x` gets expanded into the value of the variable `x` by the bash interpreter, in this case `10`

Note the `echo` command simply prints out all of its command line arguments with spaces between them

Reading the value of a variable not yet set will produce the empty string

Bash Variables - arithmetic

Even though bash variables are strings we can perform arithmetic on them with expressions inside `$(())`

```
$ echo $x+1
$ echo $((x+1))
```

Notice the difference between these two commands - the first does not perform arithmetic and demonstrates how reading a bash variable is just substituting that text literally in place

The first command is as if we literally wrote `$ echo $10+1` as the text we'd like to print out

The second command substitutes 10 for x, and then since the text `"10+1"` is within the `$(())`, that expression is evaluated as an arithmetic expression and evalutes to 11

Bash Variables - special variables

Some special variables already exist in bash, for example

```
$ diff file1.txt file2.txt
$ echo $?
```

The variable read with `$?` in bash is the exit status of the last ran command - in this case it is the exit status of the diff command ran immediately before!

Globbing Patterns

Within a bash command we can use an asterisk character `*` to represent any string within the filenames of a directory

For example consider the commands

```
$ wc *.txt
$ wc a*.txt
```

In the first command the bash interpreter will expand `*.txt` to every filename in the current working directory that ends with `.txt` since the asterisk can represent any string in a file name

In the second command the bash interpreter will expand `a*.txt` to every filname in the current working directroy that begins with `a` and ends with `.txt` since once again the asterisk can represent any string, but in this example it is between the character a and `.txt`

Strings and Bash

We've already seen all values in bash are strings - we can use double or single quotes to make strings with spaces and other special characters in them in our bash commands

```
$ echo "Hello there friend"
$ echo 'Goodbye, again'
```

- Strings with double quotes will allow for all bash expansion except for globbing patterns which will be supressed
- Strings with single quotes will not perform any bash expansion

Bash Expansion in Strings

Run the following series of commands and carefully consider the output of each

```
$ echo *
$ echo "*"
$ echo '*'
$ x=CMPUT
$ echo "$x275 is my favourite class!"
$ echo "${x}275 is my favourite class!"
$ echo '$x275 is my favourite class!'
$ echo '${x}275 is my favourite class!'
```

Combining Commands

Consider the following `head` command

```
$ head -20 alice.txt
```

- `head` reads some number of lines from the start of a file, and receives two arguments here
  - The first argument `-20` specifies how many lines we'd like to grab
  - The second argument `alice.txt` is the file we'd like to grab lines from

How could we count the words in the first 20 lines of Alice in Wonderland?

Combining Commands

Consider the following sequence of commands

```
$ head -20 alice.txt > alice_20.txt
$ wc alice_20.txt
```

- We've redirected the output from the head command to a file `alice_20.txt`
- We've then provided `alice_20.txt` as an argument to `wc`

This isn't ideal though - we've had to create a random file on our computer...

mktemp

Sometimes it can be useful to create a temporary file - we even have a command for that as well

Try the following command

```
$ mktemp
```

mktemp

Sometimes it can be useful to create a temporary file - we even have a command for that as well

Try the following command

```
$ mktemp
```

- It prints out filename

mktemp

Sometimes it can be useful to create a temporary file - we even have a command for that as well

Try the following command

```
$ mktemp
```

- It prints out filename
- It also created that file - a temporary file that will eventually be automatically deleted

mktemp

Sometimes it can be useful to create a temporary file - we even have a command for that as well

Try the following command

```
$ mktemp
```

- It prints out filename
- It also created that file - a temporary file that will eventually be automatically deleted
- But how can we use the output of a command as another piece of a commmand?

Subshells

Just as bash will expand a variable in-place and substitute the variables value we can ask bash to run a command and substitute the output of that command in place

Create a file named `number.txt` that only contains the number 5 in it, then try the following:

```
$ head -$(cat number.txt) alice.txt
```

Subshells

Just as bash will expand a variable in-place and substitute the variables value we can ask bash to run a command and substitute the output of that command in place

Create a file named `number.txt` that only contains the number 5 in it, then try the following:

```
$ head -$(cat number.txt) alice.txt
```

- `$()` forms a subshell wherein we can ask bash to run a command

Subshells

Just as bash will expand a variable in-place and substitute the variables value we can ask bash to run a command and substitute the output of that command in place

Create a file named `number.txt` that only contains the number 5 in it, then try the following:

```
$ head -$(cat number.txt) alice.txt
```

- `$()` forms a subshell wherein we can ask bash to run a command
- The subshell will be replaced in place with the output of the command ran within it

Subshells

Just as bash will expand a variable in-place and substitute the variables value we can ask bash to run a command and substitute the output of that command in place

Create a file named `number.txt` that only contains the number 5 in it, then try the following:

```
$ head -$(cat number.txt) alice.txt
```

- `$()` forms a subshell wherein we can ask bash to run a command
- The subshell will be replaced in place with the output of the command ran within it
- So here our final command resolves to `head -5 alice.txt`

Subshells

Now we can use a subshell and `mktemp` to combine our `head` and `wc` commands

```
$ myTemp=$(mktemp)
$ head -20 alice.txt > ${myTemp}
$ wc ${myTemp}
```

There's a lot going on here - some questions to make sure you understand

- Why did we have to store the result of `mktemp` in a variable?
- Why did we use `$()` when running the `mktemp` command but `$` when reading `myTemp`?

This solution *works* but it's not the best solution available to us...

Pipes

Needing to provide the output of one program as the input to another is a common problem

In fact those who developed the Unix operating system advocate for writing small single-purpose programs which can then be combined with other programs to solve many tasks

So, naturally, Unix has support for exactly this use case - pipes

```
$ head -20 alice.txt | wc
```

Pipes

To use a pipe you simply combine two commands with the pipe character |

The general form of this is

```
$ cmd1 | cmd2
$ cmd1 | cmd2 | cmd3 | cmd4 | ... | cmdn
```

- The output of a command on the left hand side of a pipe will be sent as the input to the command on the right hand side
- You can continue to "chain" pipes

Table of Contents

Programs and the PATH

The commands we've written have been executing programs - but where do these programs exist?

The programs we've used so far have been utilities provided with Ubuntu and are located somewhere on our machine

So when we execute the following command

```
$ wc alice.txt
```

how does the operating system know where to find the program `wc`?

Environment Variables

Our operating systems have defined *environment variables* which are just specifically named variables whose values affect the behaviour of our operating system or shell

For our purposes the most important environment variable is the `PATH` variable, which is a string that represents a list (separated by colons) of filepaths to directories where there are programs we'd like to be able to call by name

You can view the value of your `PATH` variable with the command

```
$ echo $PATH
```

How commands are executed in PATH

When we execute a command

```
$ cmd arg1 arg2 arg3
```

Bash will look in every directory listed in our `PATH` variable for a program with a name matching `cmd`

If Bash does not find a program named `command` it will not be able to run the program

Executing programs not found in PATH

Often in this class we will be executing programs we write - which unless we add them to a directory in our PATH we will not be able to execute simply by name

To execute a program not in your path you must instead give the filepath to that program to execute it - for example if there were a program named `myProgram` in our current working directory I could execute the program with the following command

```
$ ./myProgram
```

Note: Remember what the `./` represents here!

File Permissions

You can't just execute any file as a program by providing the filepath of it to your operating system

Not all files are programs

Also, your operating system is particular about which files are allowed to execute or not - for good reason!

Files have different *permission* which specify who can read that file, write to that file, and even execute that file as a program

File Permissions

You can view the permissions of the files in your current working directory with the command

```
$ ls -l
```

you'll receive an output much like this

```
-rwxr-xr-- 1 rob rob 174392 Jan 9 09:47 alice.txt
drwxr-xr-x 1 rob rob   4096 Jan 9 21:22 c
-rw-r--r-- 1 rob rob      7 Jan 9 10:07 file1.txt
```

What does this mean?

File Permission

```
-rwxr-xr-- 1 rob rob 174392 Jan 9 09:47 alice.txt
drwxr-xr-x 1 rob rob   4096 Jan 9 21:22 c
-rw-r--r-- 1 rob rob      7 Jan 9 10:07 file1.txt
```

Each file has 10 permission bits shown

- The first bit tells you if the file is a directory or not
- The next three bits are the *User* (owner) bits
- The following three bits are the *Group* bits
- The final three bit sare the *Other* bits

File Permission

```
-rwxr-xr-- 1 rob rob 174392 Jan 9 09:47 alice.txt
drwxr-xr-x 1 rob rob   4096 Jan 9 21:22 c
-rw-r--r-- 1 rob rob      7 Jan 9 10:07 file1.txt
```

In each group of three bits each bit represents

- First bit - read privilege (r for read, - otherwise)
- Second bit - write privilege (w for write, - otherwise)
- Third bit - execute privilege (x for executable, - otherwise)

File Permission

```
-rwxr-xr-- 1 rob rob 174392 Jan 9 09:47 alice.txt
drwxr-xr-x 1 rob rob   4096 Jan 9 21:22 c
-rw-r--r-- 1 rob rob      7 Jan 9 10:07 file1.txt
```

So in this example

- The owner of `alice.txt` can read, write, and execute[1] the file
- Members of the group the file `alice.txt` belongs to can read and execute the file
- All other users can read the file `alice.txt`

[1] While permission is granted to execute the file the file is not something that can actually be executed and attempting to run it as such will not work

Changing Permissions

Permissions of a file can be changed with the change mode (`chmod`) command

Use of the `chmod` command is of the form

```
$ chmod modechanges filename
```

Where above `filename` should be replaced with the file you'd like to change the permissions of and `modechange` should be of the form

- x=y for "set permission exactly"
- x-y for "remove permissions"
- x+y for "add permissions"

Where x may be `a` for all, `u` for user, g for group, or o for other
And y must be a string including any combination of r, w, and x

Changing Permissions Examples

Some examples of changing permissions

- `chmod a-x alice.txt` removes the permission to execute file `alice.txt` from everyone (user, group, and other)
- `chmod u+rw foo.txt` adds the permissions to read and write file `foo.txt` to the user of that file
- `chmod g=rx bar.txt` sets the permissions of the group for `bar.txt` to read and execute
  - This would add read and execute permissions to the group if they weren't already set
  - This would remove write permission for the group if they were already set

Knowledge Check - is there a difference between `u=rwx` and `u+rwx`? What about `g=rw` and `g+rw`?

Running Programs

Now we know how to run a file that is executable and how to make a file executable — but which files can we actually execute?

Any file that is an executable binary built for our computer[1] *or* any file written in a language we have an interpreter

We've written plenty of programs in Python and we have an interpreter for Python — how can we execute our Python scripts as if it were an executable program?

[1] We'll talk more about what this means when we get to C

Executing Python Scripts

Previously to run our Python programs we'd run the interpreter and provide the filename of our program as an argument to the interpreter

```
$ python my_script.py
```

It doesn't change functionality but I'd like to be able to run `my_script.py` as such:

```
$ ./my_script.py
```

Shebang Line

If we place at the start of our program the following:

```
#!/usr/bin/python
```

When we execute this text file as a program our operating system[1] will use the interpreter found at `/usr/bin/python` to execute our program

This line is called the *shebang* line

[1] This is a feature of Unix-based operating systems

Running our script as a program

Now that we've added the shebang line pointing to a Python interpeter to `my_script.py` let's try and execute it as a program

```
$ ./my_script.py
```

We get an error:

```
-bash: ./my_script.py: Permission denied
```

We forgot to add the execute permission — whoops!

Running our script from anywhere

In order to be able to execute `my_script.py` I must be in the directory that stores it or know the path to it - I'd like to be able to run it easily from *anywhere*

Let's create a new directory `cmput275_bin` - first navigate to where we'd like to put it (for me in my `cmput275` directory)

```
$ mkdir cmput275_bin
```

Running our script from anywhere

Now move the script into the `cmput275_bin` folder with the `mv` command

```
$ mv sourceFilePath targetFilePath
```

Let's also use `mv` to change the file name to a nicer command-like name — let's call it `myRand`

Read the `man` pages on `mv` for more details

Warning: the `mv` command will overwrite any file at `targetFilePath` with the file from `sourceFilePath` so be careful!

Adding a directory to our path

Now that we have our `cmput275_bin` directory to store our executable tools, we're going to add that directory to our path so that our operating system will search that directory for commands we issue

```
$ PATH="${PATH}:/home/rob/cmput275/cmput275_bin"
```

This is the absolute path to my `cmput275_bin` directory — yours is probably different!

After doing this we can now run `myRand` as a command from anywhere![1]

[1] You may notice the next time you start up your shell this is no longer true - more on that shortly!

Table of Contents

The Bash Interpreter

Our shell, bash, *is* an interpreter — it has a set of rules for how to read our commands and what procedures to follow

Your bash interpter is likely located at `/bin/bash`

Since we have a bash interpreter then we could start a text file with the shebang line:

```
#!/bin/bash
```

Bash Scripts

A *bash script* is a text file that contains a series of bash commands, meant to be executed in order - such as the following file

```
#!/bin/bash
echo "Hello today $(whoami)!"
cal
echo "What shall we do today?"
```

Bash Scripts

Bash scripts are incredibly useful for when we have a process we want to do over and over again

They can be made even more powerful by learning some of the language features of the bash

Bash Parameters

Within a bash script we can access command line arguments that were passed to the script with the variable names $1, $2, $3[1]...
Consider the following file `args.sh`

```
#!/bin/bash
echo "First arg $1, second arg $2, third arg $3"
```

Now try the following command:

```
$ ./args.sh Fee Fi Fo
```

[1] The variable $0 is always the currently running command itself!

Bash Conditionals

```
#!/bin/bash
if [ $1 -eq 5 ]; then
  echo "Number is 5"
fi
```

Spacing is important here — will not work if you don't have the correct spacing in condition line

Many conditions exist — read about them online!

Bash For Loops

```
#!/bin/bash
for x in One Two Three; do
  echo "word - $x"
done
```

Here we've literally written `One Two Three` — that could also be any bash expression that evaluates to a series of strings

Try using `$(cat alice.txt)`, `*`, `*.txt` instead

Again many powerful tools - read more online!

Bash While Loops

```
#!/bin/bash
x=$1
while [ $x -le 10 ]; do
  echo $x
  x=$((x+1))
done
```

Bash Functions

```
#!/bin/bash
sayArgs() {
  echo "Function arg1: $1, Function arg2: $2"
}

sayArgs foo bar
sayArgs bar foo
```

Note - withing functions args variables $1, $2, etc. refer to the arguments provided to that *function*

Bash - Your Best Productivity Tool

There are many more very useful built-in tools to bash, as well as several other features

Bash scripts can be very useful little tools for repeated processes you run often - consider learning more about how you can use bash scripts and the built-in tools to improve your productivity!