Missing Semester of CS Notes

Sean Wu

April 6, 2020

Contents

1 The Shell - Bash

1.1 Paths

- Cmd line arguments separated by whitespace
- Use quotes `" "` or escape the space `\`

environment variable: variable set whenever shell starts (not every run of shell)

- ex. home dir, username, `PATH` variable
- Comments in bash start with `#`

```
echo $PATH # all file paths that bash will search for programs
# OUTPUT: colon-separated list
```

- Whenever name of program (ex. `echo`) is typed, bash will search through this list in `PATH` , looking in each directory for the program matching the command

```
which echo # tells you where file for command is located (ex. echo)
```

paths: way to name location of file on computer

- Paths separated by forward slashes `/` for UNIX and backslashes `\` for Windows

`/` root; top of file system

- On UNIX, everything is under the root `/` namespace
- i.e. all absolute paths start with `/`
- On Windows, there is one root for every partition
- ex. `C:\`, `D:\`
- i.e. separate file system path hierarchies for each drive

absolute path: fully determines location of file
relative path: path relative to your current working directory
`.` current directory
`..` parent directory
`~` home directory
`-` directory you were just in

1.2 Flags and Options

- Flags and options specified after the program name
- The short form is usually with single slashes `-<char>` and the long form is usually with double dashes `--<word>`
- ex. `-v` and `--version` tell you the version of the program
- ex. `-h` and `--help` give you a quick help guide for the program
- Running command with `--help` flag gives you the `usage` in the following format

```
usage: ls [OPTION] ... [FILE] ...
# [] means optional
# ... means 1 or more of the previous thing
```

flag: doesn't take a value (usually)
option: takes a value (usually)

1.3 File Permissions

- Get file permissions by running `ls -a`
- Permissions specified in 3 groups of 3 (r, w, x)

1. 1st group of 3 permissions is for owner of file
2. 2nd group of 3 permissions is for the group of people owning the file
3. 3rd group of 3 permissions is for everyone else

- Note: if you have write access on a file but read access on a directory, you cannot directly delete a file (can only empty it)

For files:

- don't have that permission
r read access
w write access
x execute acess

For folders:

- don't have that permission
r can see files inside directory
w can rename, create, remove files

x can search this directory (i.e. enter directory with `cd`)

`chmod` : command to change file modes or Access Control Lists (i.e. change permissions)

1.4 Deleting things

`rm` : removes a file

- By default, `rm` is **not** recursive on UNIX (i.e. cannot remove a directory)
- Add a `-r` (recursive) flag to delete a directory
- Recursive delete removes everything under the path you give it

`rmdir` : deletes a directory only if it is empty (a safe delete)
`cmd L` : clears terminal output to previous mark
`cmd K` : clears terminal to start

1.5 Input and Output Streams

- Each program has 2 primary streams

1. Input stream: terminal by default
2. Output stream: terminal by default

`<` : rewire input of previous program to be the contents of this file on the right
`>` : rewire output of previous program into this file
`>>` : appends to the end of a file instead of overwriting

```
echo hello > hello.txt # writes string "hello" into file hello.txt
```

`|` : a **pipe**; takes the output of program on left and makes it the input of the program on the right. **Input program does not know about output program and vice versa** . The programs just read and write to those spots.

1.6 Root User (UNIX)

- Acts like admin user on Windows

- Has user id 0
- Has all permissions (Superuser)

`sudo` : does the following command as superuser (root user)

kernel: core of computer

`sysfs` : file system for kernel parameters of computer

- Need to be admin to change kernel params of a computer
- Note: if using `sudo` with pipes and redirects, `sudo` only applies to one portion (because input and output programs don't know about each other)

`$` indicates that you are **not** running as root

`#` indicates that you are running as root

```
sudo echo 500 > brightness
# does not work because brightness doesn't know about sudo
```

`sudo su` gives you a shell as superuser (shell runs as root now)

`exit` allows you to exit out of superuser shell mode

1.7 Misc. Helpful Commands

`man` gives you the manual pages for a program

`tail` gives you the last n lines of a file

```
tail -n5 # gives you the last 5 lines of a file
```

`tee` writes to output and to terminal output

```
echo 1000 | sudo tee brightness # changes brightness
# Note: this can be run without using superuser terminal
```

`xdg-open` opens file (Linux)

`open` opens file (macOS)

1.8 Executable and UNIX Shebang

shebang: a character sequence involving `#!` at the beginning of a script

- A shebang `#!` indicates that a file is an executable in UNIX

```
#!/bin/sh
curl --head --silent https://missing.csail.mit.edu

# First line indicates that program loader should run the
# program /bin/sh, passing path/to/script (name of this file)
# as the first argument.
```

2 Shell Tools and Scripting

2.1 Defining Variables

```
foo=bar # make var foo store the value bar
echo $foo # OUTPUT: bar (the value of the foo)
foo = bar # will not work bec of spaces
# interprets as foo being the command with = and bar being args
# Note: spaces reserved in bash for separating CLI args
```

2.2 Defining Strings

```
echo "Hello" # OUTPUT: Hello
echo 'World' # OUTPUT: World (literal string for '')
# Note: for literal strings, double "" and single quotes ''
# are equivalent
```

```
echo "value is $foo" # OUTPUT: value is bar
# variable $foo will be expanded in string for double quotes ""
echo 'value is $foo' # OUTPUT: value is $foo
# outputs string characters as displayed for single quotes ''
# doesn't expand $foo
```

2.3 Defining Functions

```
# mcd.sh, a command to make a new dir and switch to it
mcd () {
  mkdir -p "$1" # $1 is a special var for 1st CLI arg
  cd "$1"
}
```

```
source mcd.sh # executes the script mcd.sh
# new mcd function has been defined in shell
# can now do
mcd test
```

2.4 Special Bash Variables

`$0` : name of script

`$1` : 1st CLI arg

`$2 to $9` : 2nd to 9th arg

`$@` : expands to all args

`$#` : number of args given to current command

`$?` : gets error code from previous command

`$_` : last arg of previous command

`!!` : **bang bang**; Entire last command, including arguments. Usually used when you don't have permission (expands to previous command)

`$$` : Process Identification number for the current script

```
mkdir /mnt/new # Permission denied
sudo !! # becomes equivalent to
sudo mkdir /mnt/new
```

2.5 Commands and Exit Codes

- Commands often return output using `STDOUT`, errors through `STDERR` and a Return Code to report errors in a more script friendly manner
- Return code or exit status are used by scripts/commands to communicate how execution went

`0` : no issue; everything went OK

`1` or any number: error or issue with running command

```
echo "Hello" # OUTPUT: Hello
echo $? # OUTPUT: 0
```

```
grep foobar mcd.sh # no output
echo $? # OUTPUT: 1
# bash tried to search for foobar string in mcd script but it
# wasn't there (an error occurred)
```

2.6 Boolean Logic

- Note: `true` and `false` always have `0` and `1` error codes

```
true
echo $? # OUTPUT: 0
false
echo $? # OUTPUT: 1
```

2.7 Logical Operators

- Exit codes can be used to conditionally execute commands using `&&` and `||`

`||` : **OR operator**; executes 1st commmand and if it fails, it executes the (i.e. 1st command did not have a 0 error code) 2nd command

`&&` : **AND operator**; will only execute the 2nd command if the 1st one runs w/out error codes (i.e. 1st command had a 0 error code)

```
false || echo "oops fail" # OUTPUT: oops fail
# bash ran 2nd command bec the 1st command has an error code of 1
true || echo "Will not be printed" # no output
# bash didn't run the 2nd command bec the 1st command has an
# error code of 0
```

```
true && echo "Things went well" # OUTPUT: Things went well
false && echo "This will not print"
```

`;` can concatenate commands in the same line with a semicolon `;`

```
false; echo "This always prints" # OUTPUT: This always prints
```

2.8 Command Substitution

- Command substitution is used to get the output of a command as a variable

`$(cmd)` : will execute `cmd`, get the output of the command (stored in a **variable**) and substitute it in place.

```
foo=$(pwd) # gets output of pwd and stores it in foo variable
echo $foo
```

```
echo "We are in $(pwd)" # OUTPUT: We are in /Users/admin/Documents
# Note: $(pwd) is expanded because we are using double quotes ""
```

2.9 Process Substitution

- Process substitution is useful when commands expect values to be passed by file instead of by STDIN

`<(cmd)` : will execute `cmd` and place the output in a **temporary file** and substitute the `<()` with that file's name

```
cat <(ls) <(ls ..) # OUTPUT: prints files in current dir and then
# files in parent dir
# ls-ing both current and parent directories and then storing
#output in temp file using process substitution <(cmd)
# cat then reads the output of the temp file
```

`/dev/null` : special UNIX null register used to discard data that we do not care about

`>` : redirects standard output `STDOUT`

`2>` : redirects standard error `STDERR`

```
#!/bin/bash

echo "Starting program at $(date)" # Date will be substituted

echo "Running program $0 with $# arguments with pid $$"

for file in $@; do
    grep foobar $file > /dev/null 2> /dev/null
```

```
    # When pattern is not found, grep has exit status 1
    # We redirect STDOUT and STDERR to a null register since we do
    # not care about them
    if [[ $? -ne 0 ]]; then
        echo "File $file does not have any foobar, adding one"
        echo "# foobar" >> "$file"
        # appends # foobar to end of file as a comment
    fi
done
```

- To see equality test flags, run `man test`
- When performing comparisons in bash try to use double brackets `[[ ]]` in favor of simple brackets `[ ]`. Chances of making mistakes are lower although it won't be portable to `sh`

2.10 Manipulating Files

`*` **globbing**; 0 or multiple character wildcard. When used with partial file name will expand to all files matching that pattern

`?` single character wildcard; only replaces 1 character (not 0 or more like with globbing)

`{}` used when you have a common substring that you want to expand automatically. Like for writing files with similar names but different extensions

```
ls *.sh # lists all files with .sh extension
```

```
# given files foo, foo1, foo2, foo10 and bar
rm foo? # deletes foo1 and foo2
rm foo* # deletes all except for bar
```

```
convert image.png image.jpg
convert image.{png,jpg} # equivalent to above line
# Rememeber: NO SPACES or else bash treats them as separate args
```

```
touch foo{,1,2,10}
touch foo foo1 foo2 foo10
```

```
# can also combine everything and at multiple levels
touch project{1,2}/src/test{1,2,3}.py

# globbing techniques can also be combined like this
mv *{.py,.sh} folder
# Will move all *.py and *.sh files
```

`..` expands into a range. `1..5` ⟶ `1,2,3,4,5`

```
touch {foo,bar}/{a..j}
# expands into foo/a to foo/j and same with bar/a and bar/j
diff <(ls foo) <(ls bar) # compares output of 2 ls commands
```

2.11 Bash and Python Scripting

`#!` **shebang**; indicates that file is an executable and specifies which interpreter to use

- Can add shebang to python to make it executable from the shell

```
#!/usr/local/bin/python
# above line tells shell to use python as the interpreter
import sys
for arg in reversed(sys.argv[1:]):
  print(arg)
```

```
# can run above python file script.py as executable in shell
./script.py a b c # a,b,c are arguments passed to the script
```

```
# to avoid assuming where python is located, we can use the
# env command in python file

#!/usr/local/bin/env python
# give python as argument to env command
# output of env (location of python) becomes the interpreter
# specified by the shebang
import sys
for arg in reversed(sys.argv[1:]):
  print(arg)
```

`shellcheck` : useful CLI program to debug shell scripts; native shell doesn't give much useful error/debug statements

`tldr` : useful CLI program to get short documentation and examples for commands instead of using `man`

2.12 Shell Functions vs Scripts

1. Functions have to be in the same language as the shell, while scripts can be written in any language (ex. `python`)

- This is why including a shebang for scripts is important

2. Functions are loaded once when their definition is read. Scripts are loaded every time they are executed.
   - This makes functions slightly faster to load but whenever you change them you will have to reload their definition
3. Functions are executed in the current shell environment whereas scripts execute in their own process
   - Thus, functions can modify environment variables, e.g. change your current directory, whereas scripts can't.
4. Scripts will be passed by value environment variables that have been exported using `export`

2.13 Finding Files

`find` UNIX CLI tool that recursively searches thru all the files that match a certain pattern

`locate` uses a database updated using `cron` that is a a faster way of searching for files. To manually update database, run `updatedb` (Linux) or `sudo /usr/libexec/locate.updatedb` from root `/` for MacOS

- Tradeoff between `find` and `locate` is **speed vs freshness**
- Database may contain out of date info and needs to be updated

```
# Find all directories named src
find . -name src -type d
# Find all python files with a folder named test in their path
find . -path '**/test/**/*.py' -type f
# Find all files modified in the last day
find . -mtime -1
# Find all zip files with size in range 500k to 10M
find . -size +500k -size -10M -name '*.tar.gz'
```

```
# Delete all files with .tmp extension
find . -name '*.tmp' -exec rm {} \;
# Find all PNG files and convert them to JPG
find . -name '*.png' -exec convert {} {.}.jpg \;
```

2.14 Searching Within Files

`grep` UNIX CLI tool used for searching or matching patterns from input text

`rg` **ripgrep**; a CLI tool that improves `grep` by ignoring .git folders, using multi CPU support, etc.

Useful `grep` and `rg` flags

`-C n` gives n lines of **C**ontext around the matched string
`-v` in**v**erts the match, i.e. print all lines that do not match the pattern
`-R` **R**ecursively go into directories and look for text files for the matching string.

```
# Find all python files where I used the requests library
rg -t py 'import requests'
# Find all files (including hidden files) without a shebang line
rg -u --files-without-match "^#!"
# Find all matches of foo and print the following 5 lines
rg foo -A 5
# Print statistics of matches (# of matched lines and files )
rg --stats PATTERN
```

2.15 Searching Previous Shell History

`up arrow` : goes through previous commands line by line. Inefficient for very old commands
`history` : command that prints out most recent commands
`ctrl r` : backwards search fo previous command history and execute in place. Repetitive typing of `ctrl r` will give you next previous command

```
history 1 # prints all results since beginning of time
history 1 | grep convert
# search all history for commands using convert
```

2.16 Directory Structure

`tree` : pretty prints the directory structure