Unix

https://harvard-iacs.github.io/2019-CS207/lectures/lecture2/

David Sondak

Harvard University
Institute for Applied Computational Science

9/10/2019

Last Time

- Unix and Linux
- Text editors

Today

- Shell Customization
- Job control
- Unix scripting

Again, some content adapted from Dr. Chris Simmons.

Text Editors and Shell Customization

Text Editors

- For programming and changing of various text files, we need to make use of available Unix text editors
- The two most popular and available editors are vi and emacs
- You should familiarize yourself with at least one of the two
  - Editor Wars
- We will have very short introductions to each

A Brief Text Editor History

- `ed` : line mode editor
- `ex` : extended version of ed
- `vi` : full screen version of ex
- `vim` : Vi IMproved
- `emacs` : another popular editor
- `ed/ex/vi` share lots of syntax, which also comes back in `sed/awk`: useful to know.

`vi` Overview

- The big thing to remember about `vi` is that it has two different modes of operation:
  - Insert Mode
  - Command mode
- The insert mode puts anything typed on the keyboard into the current file
- The command mode allows the entry of commands to manipulate text
- Note that `vi` starts out in the command mode by default

`vim` Quick Start Commands

- `vim <filename>`
- Press `i` to enable insert mode
- Type text (use arrow keys to move around)
- Press `Esc` to enable command mode
- Press `:w` (followed by `return`) to save the file
- Press `:q` (followed by `return`) to exit `vim`

Useful vim Commands

- `:q!` - exit without saving the document. Very handy for beginners
- `:wq` - save and exit
- `/ <string>` - search within the document for text. `n` goes to next result
- `dd` - delete the current line
- `yy` - copy the current line
- `p` - paste the last cut/deleted line
- `:1` - goto first line in the file
- `:$` - goto last line in the file
- `$` - end of current line
- $\wedge$ - beginning of line
- `%` - show matching brace, bracket, parentheses

Here are some `vim` resources: https://vim.rtorr.com/, https://devhints.io/vim, https://vim-adventures.com/, `vimtutor`.

Shell Customization

- Each shell supports some customization.
  - user prompt settings
  - environment variable settings
  - aliases
- The customization takes place in startup files which are read by the shell when it starts up
  - Global files are read first - these are provided by the system administrators (e.g. `/etc/profile`)
  - Local files are then read in the user's `HOME` directory to allow for additional customization

Shell Startup Files

Useful information can be found at the `bash man` page:
https://linux.die.net/man/1/bash

- `~/.bash_profile`
  - Conventionally executed at login shells
  - Conventially only run once: at login
  - MacOS executes it for *every* new window
- `~/.bashrc`
  - Conventionally executed for each new window
  - Can contain similar information as the `.bash_profile`

Decent reference on the difference between `.bash_profile` and `.bashrc`:
Apple Stack Exchange, Scripting OS X

Lecture Exercise

Update your `.bash_profile`

Exercise goals:

- Familiarize with a text editor (like `vim`)
- Create an alias for `ls` (e.g. `ll`) [see https://www.tecmint.com/create-alias-in-linux/]
- Change command line prompt format (see https://www.cyberciti.biz/tips/howto-linux-unix-bash-shell-setup-prompt.html)

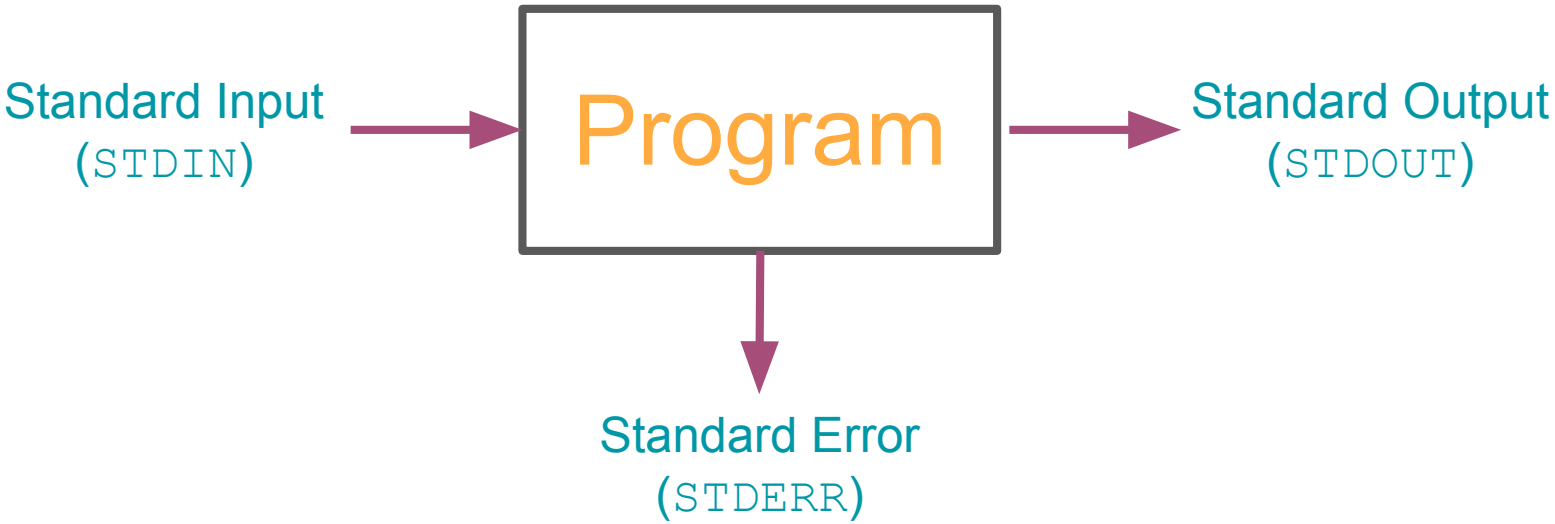
Deliverables:

- Push your `.bash_profile` to your `lectures/L2` directory.
- The `.bash_profile` should have at least three Unix command line aliases.

Note to Windows users: Modify Bash Profile in Windows

Note: The Dracula Theme is pretty fun.

I/O



- File descripters are associated with each stream,
  - `0=STDIN`, `1=STDOUT`, `2=STDERR`
- When a shell runs a program for you,
  - Standard input is the keyboard
  - Standard output is your screen
  - Standard error is your screen
- To end the input, press `Ctrl-D` on a line; this ends the input stream

Shell Stream Redirection

- The shell can attach things other than the keyboard to standard input or output
  - e.g. a file or a pipe
- To tell the shell to store the output of your program in a file, use >,
  - `ls > ls_out`
- To tell the shell to get standard input from a file, use <,
  - `sort < nums`
- You can combine both forms together,
  - `sort < nums > sortednums`

Modes of Output Redirection

- There are two modes of output redirection,
  - `>` — create mode
  - `>>` — append mode
- `ls > foo` creates a new file `foo`, possibly deleting any existing file named `foo` while `ls >> foo` appends the output to `foo`
- `>` only applies to `stdout` (not `stderr`)
- To redirect `stderr` to a file, you must specify the request directly
  - `2>` redirects `stderr` (e.g. `ls foo 2> err`)
  - `&>` redirects `stdout` and `stderr` (e.g. `ls foo &> /dev/null`)
  - `ls foo > out 2> err` redirects `stdout` to `out` and `stderr` to `err`

Wildcards

- The shell treats some characters as special
- These special characters make it easy to specify filenames
- `*` matches anything
- Giving the shell `*` by itself removes `*` and replaces it with all the filenames in the current directory
- `echo` prints out whatever you give it (e.g. `echo hi` prints out `hi`)
- `echo *` prints out the entire working directory!
- `ls *.txt` lists all files that end with `.txt`

Job Control

- The shell allows you to manage jobs:
  - Place jobs in the background
  - Move a job to the foreground
  - Suspend a job
  - Kill a job

Job Control

- The shell allows you to manage jobs:
  - Place jobs in the background
  - Move a job to the foreground
  - Suspend a job
  - Kill a job
- Putting a & after a command on the command line will run the job in the background

Job Control

- The shell allows you to manage jobs:
  - Place jobs in the background
  - Move a job to the foreground
  - Suspend a job
  - Kill a job
- Putting a & after a command on the command line will run the job in the background
- Why do this?
  - You don't want to wait for the job to complete
  - You can type in a new command right away
  - You can have a bunch of jobs running at once
- e.g. `./program > output &`

Job Control

- The shell allows you to manage jobs:
  - Place jobs in the background
  - Move a job to the foreground
  - Suspend a job
  - Kill a job
- Putting a & after a command on the command line will run the job in the background
- Why do this?
  - You don't want to wait for the job to complete
  - You can type in a new command right away
  - You can have a bunch of jobs running at once
- e.g. `./program > output &`
- If the job will run longer than your session use `nohup`
  - `nohup ./program &> output &`

Job Control

- The shell allows you to manage jobs:
  - Place jobs in the background
  - Move a job to the foreground
  - Suspend a job
  - Kill a job
- Putting a & after a command on the command line will run the job in the background
- Why do this?
  - You don't want to wait for the job to complete
  - You can type in a new command right away
  - You can have a bunch of jobs running at once
- e.g. `./program > output &`
- If the job will run longer than your session use `nohup`
  - `nohup ./program &> output &`
- Terminal multiplexers (e.g. tmux or screen) are *great* for this

Listing Jobs

- The `jobs` command lists all background jobs

```
dsondak:~/Teaching/Harvard/CS207/2019-CS207
$ jobs
[1]+  Running                 iacs launch &
```

- The shell assigns a number to each job
- *kill* the foreground job using `Ctrl-C`
- Kill a background job using the `kill` command

```
dsondak:~/Teaching/Harvard/CS207/2019-CS207
$ kill %1
[1]+  Terminated: 15          iacs launch
```

- **Try it out:**
  - Use the `sleep` command to suspend the terminal session for 60 seconds
  - Suspend the job using ^-Z
  - List the jobs, send the job to the background with `bg %n`, list the jobs
  - Use the `fg %n` to bring the `sleep` command back to the foreground

Environment Variables

- Unix shells maintain a list of environment variables that have a unique name and value associated with them
  - Some of these parameters determine the behavior of the shell
  - They also determine which programs get run when commands are entered
  - Provide information about the execution environment to programs

Environment Variables

- Unix shells maintain a list of environment variables that have a unique name and value associated with them
  - Some of these parameters determine the behavior of the shell
  - They also determine which programs get run when commands are entered
  - Provide information about the execution environment to programs
- We can access these variables
  - Set new values to customize the shell
  - Find out the value to accomplish a task

Environment Variables

- Unix shells maintain a list of environment variables that have a unique name and value associated with them
  - Some of these parameters determine the behavior of the shell
  - They also determine which programs get run when commands are entered
  - Provide information about the execution environment to programs
- We can access these variables
  - Set new values to customize the shell
  - Find out the value to accomplish a task
- To view environment variables use `env`

```
dsondak:~/Teaching/Harvard/CS207/2019-CS207
$ env | grep PWD
PWD=/Users/dsondak/Teaching/Harvard/CS207/2019-CS207
```

Environment Variables

- Unix shells maintain a list of environment variables that have a unique name and value associated with them
  - Some of these parameters determine the behavior of the shell
  - They also determine which programs get run when commands are entered
  - Provide information about the execution environment to programs
- We can access these variables
  - Set new values to customize the shell
  - Find out the value to accomplish a task
- To view environment variables use `env`

```
dsondak:~/Teaching/Harvard/CS207/2019-CS207
$ env | grep PWD
PWD=/Users/dsondak/Teaching/Harvard/CS207/2019-CS207
```

- Use `echo` to print variables
  - `echo $PWD`
  - The `$` is needed to access the value of the variable

PATH

- Each time you provide the shell a command to execute, it does the following:
  - Checks to see if the command is a built-in shell command
  - If it's not a built-in command, the shell tries to find a program whose name matches the desired command

PATH

- Each time you provide the shell a command to execute, it does the following:
  - Checks to see if the command is a built-in shell command
  - If it's not a built-in command, the shell tries to find a program whose name matches the desired command
- How does the shell know where to look on the filesystem?

PATH

- Each time you provide the shell a command to execute, it does the following:
  - Checks to see if the command is a built-in shell command
  - If it's not a built-in command, the shell tries to find a program whose name matches the desired command
- How does the shell know where to look on the filesystem?
- The `PATH` variable tells the shell where to search for programs

```
$ echo $PATH
/usr/local/opt/ruby/bin:/Users/dsondak/.jenv/shims:/Users/dsondak/.jenv/bin:/opt/local/bin:/opt/
local/sbin:/Users/dsondak/gems/bin:/Users/dsondak/.gem/ruby/2.6.3/bin:/Users/dsondak/anaconda3/b
in:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin
```

PATH

- Each time you provide the shell a command to execute, it does the following:
  - Checks to see if the command is a built-in shell command
  - If it's not a built-in command, the shell tries to find a program whose name matches the desired command
- How does the shell know where to look on the filesystem?
- The `PATH` variable tells the shell where to search for programs

```
$ echo $PATH
/usr/local/opt/ruby/bin:/Users/dsondak/.jenv/shims:/Users/dsondak/.jenv/bin:/opt/local/bin:/opt/
local/sbin:/Users/dsondak/gems/bin:/Users/dsondak/.gem/ruby/2.6.3/bin:/Users/dsondak/anaconda3/b
in:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin
```

- The `PATH` is a list of directories delimited by colons
  - It defines a list and search order
  - Directories specified earlier in `PATH` take precedence
  - Once the matching command is found, the search terminates

PATH

- Each time you provide the shell a command to execute, it does the following:
  - Checks to see if the command is a built-in shell command
  - If it's not a built-in command, the shell tries to find a program whose name matches the desired command
- How does the shell know where to look on the filesystem?
- The `PATH` variable tells the shell where to search for programs

```
$ echo $PATH
/usr/local/opt/ruby/bin:/Users/dsondak/.jenv/shims:/Users/dsondak/.jenv/bin:/opt/local/bin:/opt/
local/sbin:/Users/dsondak/gems/bin:/Users/dsondak/.gem/ruby/2.6.3/bin:/Users/dsondak/anaconda3/b
in:/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/texbin
```

- The `PATH` is a list of directories delimited by colons
  - It defines a list and search order
  - Directories specified earlier in `PATH` take precedence
  - Once the matching command is found, the search terminates
- Add more search directories to your path using `export`:

```
export PATH="$PATH:/Users/dsondak"
```

Setting Environment Variables

- Setting a Unix environment in bash uses the `export` command
  - `export USE_CUDA=OFF`
- Environment variables that you set interactively are only available in your current shell
  - If you spawn a new shell, these settings will be lost
  - To make more lasting changes, alter the login scripts that affect your particular shell (in bash, this is `.bashrc`)
- An environment variable can be deleted with the `unset` command
  - `unset USE_CUDA`

Unix Scripting

Unix Scripting

- Place all the Unix commands in a file instead of typing them interactively
- Useful for automating tasks
  - Repetitive operations on files, etc
  - Performing small post-processing operations
- Shells provide basic control syntax for looping, `if` constructs, etc

More on Unix Scripting

- Shell scripts must begin with a specific line to indicate which shell should be used to execute the remaining commands in the file
  - Use `#!/bin/bash` in BASH
- Comment out lines with #
- To run a shell script, it must have execute permission

Unix Scripting Permissions

```
dsondak:~/Teaching/Harvard/CS207/2019-CS207/content/lectures/lecture1/notes
$ ls -ltr
total 4
-rw-r--r-- 1 dsondak staff  0 Sep  3 17:46 README.md
-r--rwx--x 1 dsondak staff  0 Sep  3 18:37 foo
-r----xrwx 1 dsondak staff  0 Sep  3 18:47 bar
-rw-r--r-- 1 dsondak staff 31 Sep  3 20:58 hello.sh
dsondak:~/Teaching/Harvard/CS207/2019-CS207/content/lectures/lecture1/notes
$ cat hello.sh
#!/bin/bash
echo "hello world"
dsondak:~/Teaching/Harvard/CS207/2019-CS207/content/lectures/lecture1/notes
$ ./hello.sh
-bash: ./hello.sh: Permission denied
dsondak:~/Teaching/Harvard/CS207/2019-CS207/content/lectures/lecture1/notes
$ chmod 700 hello.sh
dsondak:~/Teaching/Harvard/CS207/2019-CS207/content/lectures/lecture1/notes
$ ls -ltr
total 4
-rw-r--r-- 1 dsondak staff  0 Sep  3 17:46 README.md
-r--rwx--x 1 dsondak staff  0 Sep  3 18:37 foo
-r----xrwx 1 dsondak staff  0 Sep  3 18:47 bar
-rwx------ 1 dsondak staff 31 Sep  3 20:58 hello.sh
dsondak:~/Teaching/Harvard/CS207/2019-CS207/content/lectures/lecture1/notes
$ ./hello.sh
hello world
```

Unix Scripting: Conditionals

```
if [ condition_A ]; then
        # code to run if condition_A true
elif [ condition_B ]; then
        # code to run if condition_A false and
        #    condition_B true
else
        # code to run if both conditions false
fi
```

Unix Scripting: String Comparisons

- `string1=string2`: Test identity
- `string1!=string2`: Test inequality
- `-n string`: The length of `string` is nonzero
- `-z string`: The length of `string` is zero

```
today="monday"
if [ "$today" = "monday" ]; then
    echo "Today is Monday!"
fi
```

BASH Integer Comparisons

- `int1 -eq int2`: Test identity
- `int1 -ne int2`: Test inequality
- `int1 -lt int2`: Less than
- `int1 -gt int2`: Greater than
- `int1 -le int2`: Less than or equal
- `int1 -ge int2`: Greater than or equal

```
x=13
y=25
if [ $x -lt $y ]; then
  echo "$x is less than $y"
fi
```

Unix Scripting: Common File Tests

- `-d file`: Test if the file is a directory
- `-f file`: Test if the file is not a directory
- `-s file`: Test if the file has nonzero length
- `-r file`: Test if the file is readable
- `-w file`: Test if the file is writable
- `-x file`: Test if the file is executable
- `-o file`: Test if the file is owned by the user
- `-e file`: Test if the file exists

```
if [ -f foo ]; then
  echo "foo is a file"
fi
```

Lecture Exercises

https://harvard-iacs.github.io/2019-CS207/lectures/lecture2/