Lecture 2

Thursday, September 10th, 2020

Shell Customization, I/O, Job control, Environment Variables, Bash scripting

Recap

Last time:

- More Unix commands
- Interacting with the shell
- File attributes

This time:

- Shell customization
- I/O
- Job Management
- Environment variables
- Bash scripting

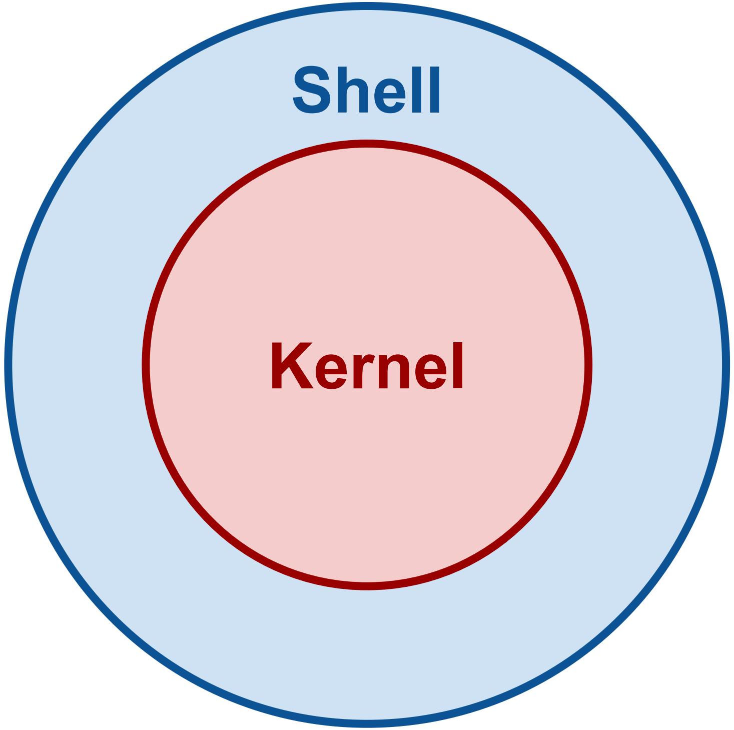
Shell Customization

Shell Recap

Users interact with the shell!

Different types of shells:

- zsh
- bash
- ⋮



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 (I)

- Different startup files are read depending on which shell you are using
- Wikipedia has a nice summary: Shell Configuration Files
- The *bash shell* has two configuration files:
  - `~/.bash_profile` - Read at login
  - `~/.bashrc` - Read at login and when a new window is opened
- The *zsh shell* uses:
  - `~/.zprofile` - Read at login
  - `~/.zshrc` - Read at login and when a new window is opened
- It can sometimes be confusing to keep in mind what all the files do. Here are some refs:
  - Moving to zsh – Scripting OS X
  - What should/shouldn't go in .zshenv, .zshrc, .zlogin, .zprofile, .zlogout?
  - ZSH: .zprofile, .zshrc, .zlogin - What goes where?
  - bash(1) - Linux man page, About bash_profile and bashrc on macOS

Breakout Room: Shell Startup Files

1. Note your breakout room number.
2. Figure out who’s birthday is next.
3. Create an alias for `ls` (e.g. `ll`). Put this in the appropriate startup file!
    a. Example: How to Create and Use Alias Command in Linux
4. Change the command line prompt format. Put it in the appropriate startup file!
    a. Example: How to Change / Set up bash custom prompt (PS1) in Linux

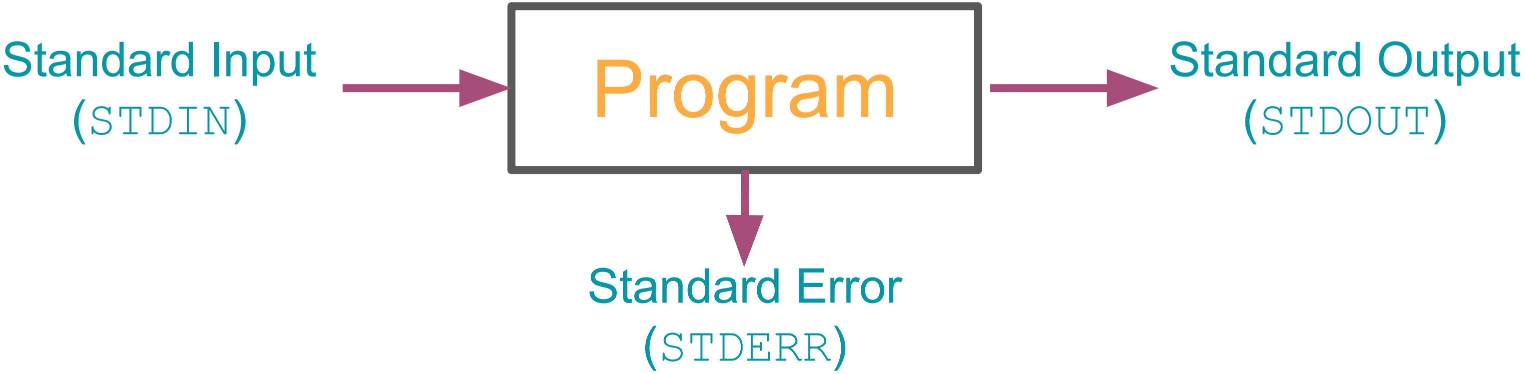
The examples above may use a different shell from yours, but the basic ideas are the same. You will need to figure out where to put the commands.

I like and use this shell theme: https://draculatheme.com/

Windows users may want to check this out if you’re using **Git Bash**: How do I modify my Git Bash profile in Windows?

I/O

I/O



- File descriptors 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
- Use > to tell the shell to store the output of your program in a file
    - `ls > ls_out`
- Use < to tell the shell to get standard input from a file
    - `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 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

Job Control

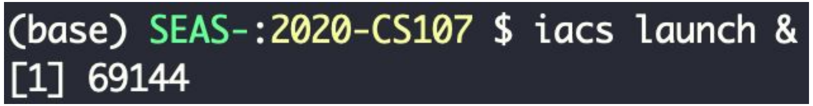
- The shell allows you to manage jobs:
  - Place jobs in the background
  - Move a job to the foreground
  - Suspend a job
  - Kill a job
- Putting an `&` after a command on the command line will run the job in the background
- Why do this?
  - You don't want to wait for the job to complete
  - You can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

Job Control: `nohup` and Terminal Multiplexers

- Use `nohup` if the job will run longer than your session
  - `nohup ./program &> output &`
- `Terminal multiplexers` are *great* for this
  - 6 Best terminal multiplexers as of 2020
  - Screen
  - tmux

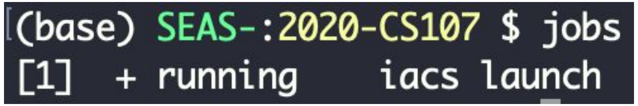
Listing Jobs

The shell assigns a number to each job



```
(base) SEAS-:2020-CS107 $ iacs launch &
[1] 69144
```

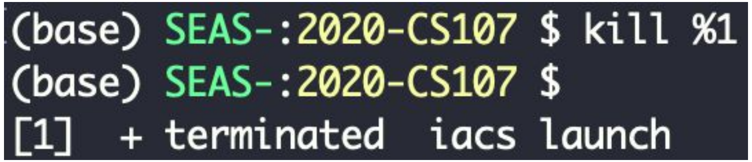
The `jobs` command lists all background jobs



```
(base) SEAS-:2020-CS107 $ jobs
[1]  + running    iacs launch
```

kill the foreground job using `Ctrl-C`

Kill a background job using the `kill` command



```
(base) SEAS-:2020-CS107 $ kill %1
(base) SEAS-:2020-CS107 $
[1]  + terminated  iacs launch
```

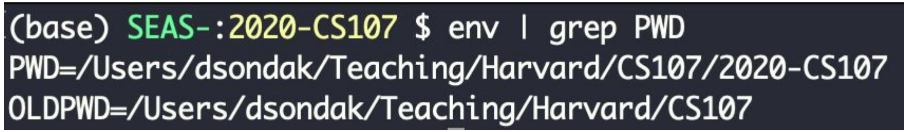
Breakout Room: Practice Listing Jobs

1. Use the `sleep` command to suspend the terminal session for 60 seconds
   a. Hint: If you're never met `sleep` before, type `man sleep` at the command line
2. Suspend the job using `Ctrl-Z`
3. Now list the jobs using the `jobs` command
4. The job isn't in the background; it's just suspended. Send the job to the background with `bg %n` where `n` is the job id that you obtained from the `jobs` command
5. Bring the `sleep` command (your job) back to the foreground with `fg %n`
6. You could let the job finish (since it's only sleeping for 60 seconds after all), or you can kill it. Up to you.

Environment Variables

Environment Variables

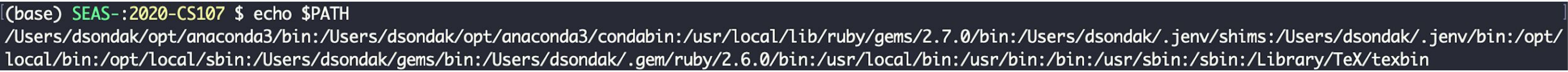
- Unix shells maintain a list of environment variables that have a unique name and value associated with them
  - Some of these parameters determine the behavior of the shell
  - They also determine which programs get run when commands are entered
  - The can provide information about the execution environment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



```
(base) SEAS-:2020-CS107 $ env | grep PWD
PWD=/Users/dsondak/Teaching/Harvard/CS107/2020-CS107
OLDPWD=/Users/dsondak/Teaching/Harvard/CS107
```

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 shell 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



```
(base) SEAS-:2020-CS107 $ echo $PATH
/Users/dsondak/opt/anaconda3/bin:/Users/dsondak/opt/anaconda3/condabin:/usr/local/lib/ruby/gems/2.7.0/bin:/Users/dsondak/.jenv/shims:/Users/dsondak/.jenv/bin:/opt/
local/bin:/opt/local/sbin:/Users/dsondak/gems/bin:/Users/dsondak/.gem/ruby/2.6.0/bin:/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
    - The search terminates once the matching command is found
- 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
  - These settings will be lost if you spawn a new shell
  - To make more lasting changes, alter the login scripts that affect your particular shell (in bash this is `.bashrc`, in zsh this is `.zshrc`)
- An environment variable can be deleted with the `unset` command
  - `unset USE_CUDA`

Unix Scripting

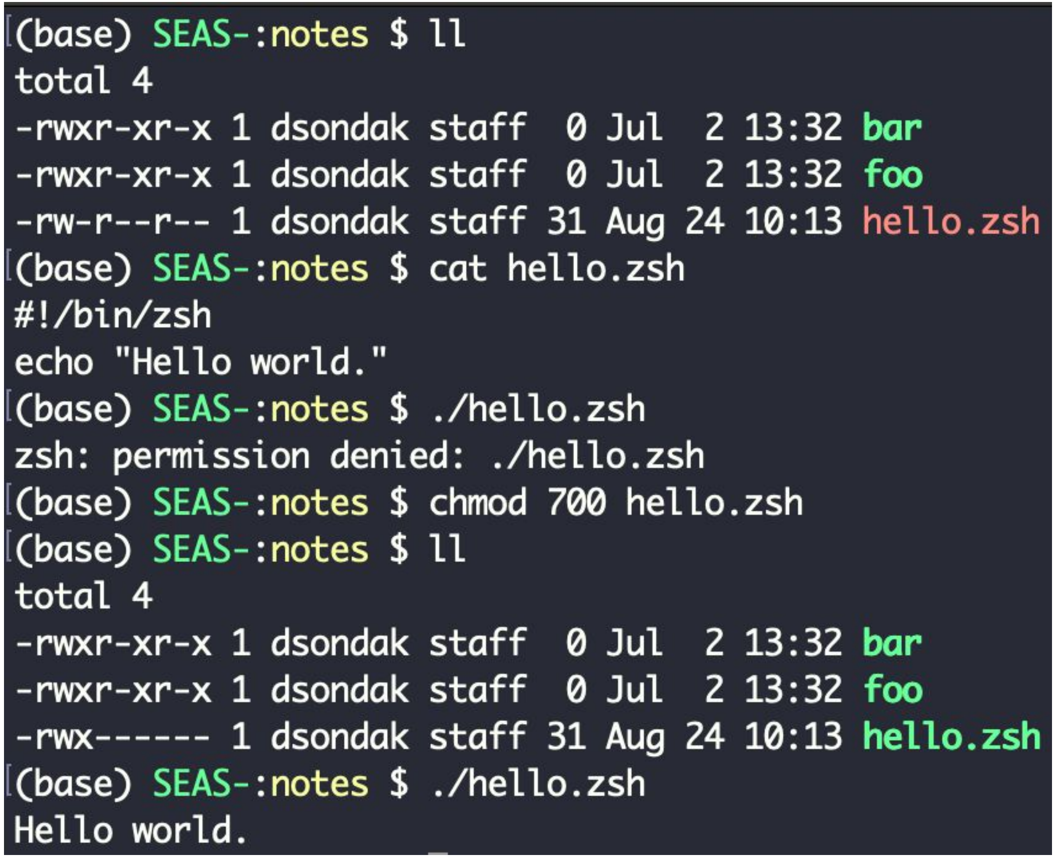
Unix Scripting

- 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
  - Use `#!/bin/zsh` in Zsh
  - These are called *shebang* lines
  - Comment out lines with `#`
- To run a shell script, it must have execute permission
- Excellent resources are available. Here is a recent one:
  - Moving to zsh
  - On the Shebang

Unix Scripting Permissions



```
(base) SEAS-:notes $ ll
total 4
-rwxr-xr-x 1 dsondak staff  0 Jul  2 13:32 bar
-rwxr-xr-x 1 dsondak staff  0 Jul  2 13:32 foo
-rw-r--r-- 1 dsondak staff 31 Aug 24 10:13 hello.zsh
(base) SEAS-:notes $ cat hello.zsh
#!/bin/zsh
echo "Hello world."
(base) SEAS-:notes $ ./hello.zsh
zsh: permission denied: ./hello.zsh
(base) SEAS-:notes $ chmod 700 hello.zsh
(base) SEAS-:notes $ ll
total 4
-rwxr-xr-x 1 dsondak staff  0 Jul  2 13:32 bar
-rwxr-xr-x 1 dsondak staff  0 Jul  2 13:32 foo
-rwx------ 1 dsondak staff 31 Aug 24 10:13 hello.zsh
(base) SEAS-:notes $ ./hello.zsh
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

You may want to use these in your conditional statements sometimes.

Test identity

```
string1=string2
```

Test inequality

```
string1!=string2
```

The length of string is nonzero

```
-n string
```

The length of string is zero

```
-z string
```

String Comparisons: Example

```
#!/bin/zsh

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

Integer Comparisons

Test identity

```
int1 -eq int2
```

Test inequality

```
int1 -ne int2
```

Less than

```
int1 -lt int2
```

Greater than

```
int1 -gt int2
```

Less than or equal

```
int1 -le int2
```

Greater than or equal

```
int1 -ge int2
```

Integer Comparisons: Example

```
#!/bin/zsh

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

Common File Tests (I)

Sometimes you want to check the state of files and directories:

Test if the file is a directory

```
-d file
```

Test if the file is not a directory

```
-f file
```

Test if the file has nonzero length

```
-s file
```

Test if the file is readable

```
-r file
```

Test if the file is writable

```
-w file
```

Test if the file is executable

```
-x file
```

Common File Tests (II)

Test if the file is owned by the user

```
-o file
```

Test if the file exists

```
-e file
```

```
#!/bin/zsh

foo=$1 # read from command line

if [ -f $foo ]; then
    echo "$foo exists and is a file."
fi
```

Recap

Today we covered:

- Shell customization - have fun making your terminal be useful and look great
- Redirecting the I/O streams
- Controlling jobs: Putting them in the background / foreground, cancelling them, suspending them
- Quick tour through Unix environment variables
- Some of the essentials of Unix scripting