CSE 374 Lecture 4

Shell Variables and More Scripting

Feel free to ask questions until lecture starts…

Any questions before we get started?

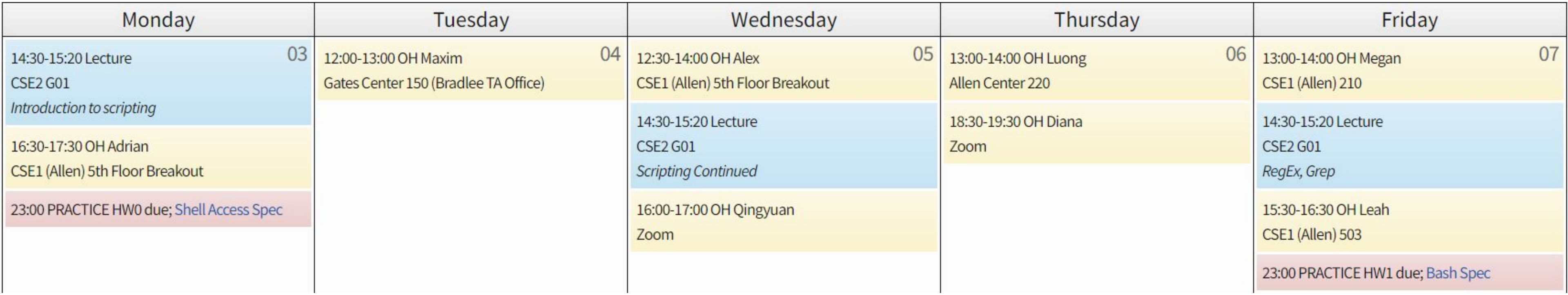
Tc 0

Powered by Poll Everywhere

Today

1. Alias
2. Scripting
3. Source / executable

Office hours this week:



Monday	Tuesday	Wednesday	Thursday	Friday
03	04	05	06	07
14:30-15:20 Lecture CSE2 G01 *Introduction to scripting*	12:00-13:00 OH Maxim Gates Center 150 (Bradlee TA Office)	12:30-14:00 OH Alex CSE1 (Allen) 5th Floor Breakout	13:00-14:00 OH Luong Allen Center 220	13:00-14:00 OH Megan CSE1 (Allen) 210
16:30-17:30 OH Adrian CSE1 (Allen) 5th Floor Breakout		14:30-15:20 Lecture CSE2 G01 *Scripting Continued*	18:30-19:30 OH Diana Zoom	14:30-15:20 Lecture CSE2 G01 *RegEx, Grep*
23:00 PRACTICE HW0 due; Shell Access Spec		16:00-17:00 OH Qingyuan Zoom		15:30-16:30 OH Leah CSE1 (Allen) 503
				23:00 PRACTICE HW1 due; Bash Spec

HW0 & HW1

Please remember that these two assignments should be done using Seaside.

If your homework passes the autograder this is sufficient….

Getting files from the VM to Gradescope: There are options, but, I use the GUI and open Gradescope in the VM web browser.
To move files from seaside, the ‘scp’ command works well. Remember the command is `scp <copyfrom> <copyto>`, and pay attention to which computer you are executing it on.

Passwords, and managing Passwords

Linux systems have consistent password management.

- `/etc/passwd` file contains user info
  - Username
  - Password
  - Userid, groupid
  - Shell
  - Home directory
- `/etc/shadow` stores encrypted passwords

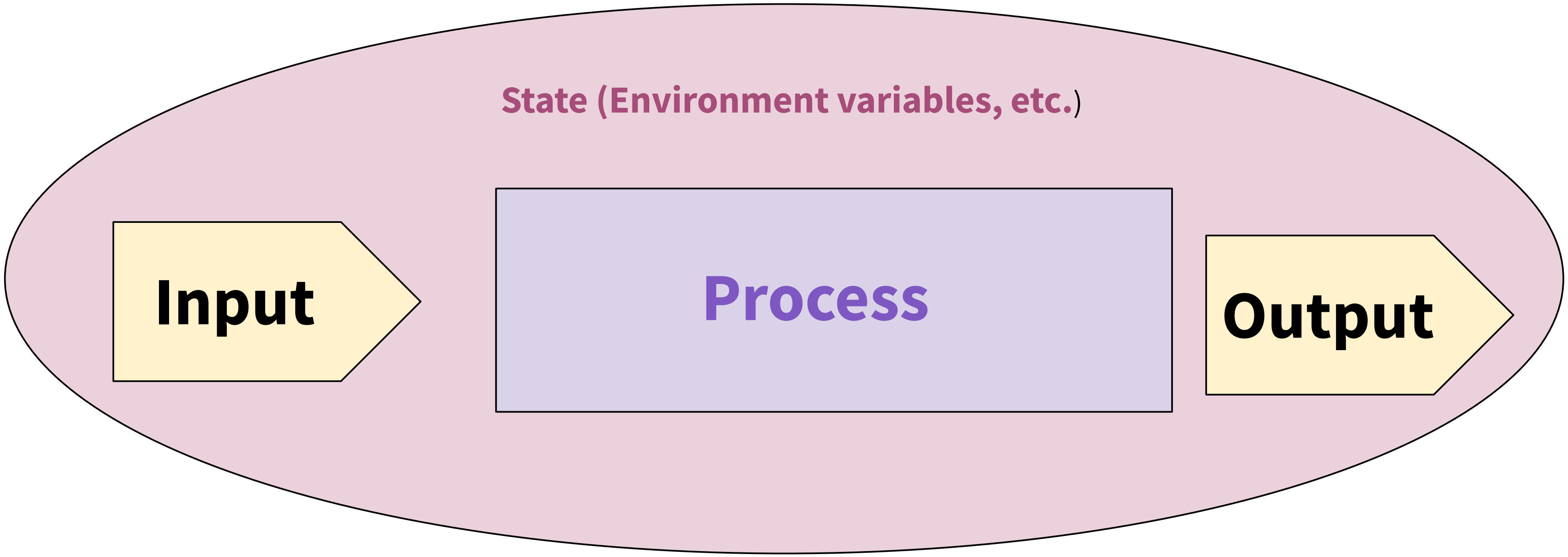
Change your password on Linux:

```
> passwd
```

Prompts for previous password, then new password

Passwd also has facilities for those with sudo access to update other user accounts and password management

Seaside is a little different - passwords are obtained from the UWNetID servers (no /etc/passwd entries).
Passwd will work, and propagate changes through UWNetID servers.



BASH applies its own processing to the I/O text - ‘globbing’

Towards Scripts

- Shell has a state (working directory, user, aliases, history, streams)
- Can expand state with variables
- ‘Source’ runs a file and changes state

Variables & Alias

Define variable

i=15

Access variable

$i

Undefined variable is empty string

Alias cheer=”echo yahoo\!”

Defines a shortcut, or ‘alias’ to a command

Essentially a super simple script.

.bashrc

Alias

Defines a shortcut or ‘alias’ to a command.

(Essentially a really easy script)

Also, ‘alias’

.bashrc

.bash_profile

- Executed for login shells
- Use for commands run once
  - Changing $PATH

.bashrc

- Executed for non-login shells
- Use for commands that are re-run
  - Aliases & functions

Emacs (text editor)

```
C-x C-s #save

C-x C-c # quit

C-e # go to end of line

C-a # go to beginning of line

C-x C-f # find a file

C-g #exit menu

C-x C-k # kill a buffer
```

You can use any text editor you like. Emacs is amazingly powerful, and highly customizable with lisp scripts. It is probably worth learning.

Towards Scripts

- Shell has a state (working directory, user, aliases, history, streams)
- Can expand state with variables
- ‘Source’ runs a file and changes state
- Can run a file without changing state by running script in new shell.
- Allows for repeatable processes and actions

Variables useful in a script

$# stores number of parameters (strings) entered

$0 first string entered - the command name

$N returns the Nth argument

$? Returns state of last exit

$* returns all the arguments

$@ returns a space separated string with each argument

(* returns one word with spaces, @ returns a list of words)

Variables

Shell has a state, which includes shell variables

All variables are strings (but can do math, later)

White space matters - not spaces around the ‘=’

Create: myVar= or myVar=value

Set: myVar=value

Use: $myVar

Remove: unset $myVar

List variables (use ‘set)

Special Variables

Common variables which set shell state:

$HOME - sets home directory. $HOME=~/CSE374 would reset your home directory to always be CSE374

$PS1 - sets prompt

$PATH - tells shell where to look for things. Often extended:
$PATH=$PATH:~/CSE374

Show current state: `printenv`

Export Variables

Use: export myVar

To make variable available in the initial shell environment.

If a program changes the value of an exported variable it does not change the value outside of the program

: export -n remove export property

Variables act as though passed by value

Special Characters

! > < & | * ~ [] “ ‘ ` $ /

\ is escape character

“string”

‘string’

What do they all mean?

Would substitute things like $VAR

Suppresses substitutions

Okay, lets make a script!

1. First line of file is #!/bin/bash (specifies which interpreter to execute)
2. Make file executable (chmod u+x)
3. Run a file ./myNewScript
4. Shell sees the shell program (/bin/bash) and launches it to run the script
5. Can include
   a. String tests (string returns true if non-zero length, string < string, etc.)
   b. Logic (&&,||,!) - use double brackets
   c. File tests (-d : is directory, -f: is file, -w: file has write permission etc.)
   d. Math - use double parens

Script Arguments & Errors

Script refers to ith argument at $i ; $0 is the program

Use ‘shift’ to move arguments towards left ($i become $i-n)

Exit your shell with 0 (normal) or 1 (error)

Exit Codes

Command ‘exit’ exits a shell, and ends a shell-script program.

Exit with no error:

Use `exit` or `exit 0`

Exit with error:

User `exit 1` or.. {1-255}

Quoting Variables

In order to retain the literal value of something use ‘single quotes’

In order to retain all but $, ` , \ use “double quotes”

Put $* and $@ in quotes to correctly interpret strings with spaces in them.

Arithmetic

Variables hold strings, so we need a way to tell the shell to evaluate them numerically:

`K=$i+$j` **does not add the numbers**

Use the shell function ((

```
k=$(( $i+$j ))
```

Or `let k=”$i+$j”`

The shell will automatically convert the strings to the numbers

Functions and local variables

Yes, possible

Generally, a script's variables are global

```
name () compound-command [ redirections ]
or
function name [()] compound-command [
redirections ]

Ex:

func1()
{
    local var='func1 local'
    func2
}
```

Stuff to watch out for

White space: spacing of words and symbols matters

Assign WITHOUT spaces around the equal, brackets are WITH SPACES

Typo on left creates new variable, typo on right returns empty string.

Reusing variable name replaces the old value

Must put quotes around values with spaces in them

Non number converted to number produces ‘0’

Conditionals

Binary operators: -eq -ne -lt -le -gt -ge

Can use the [[shell command to use < , > , ==

Syntax is a little different, but commands works as expected

Flow control

```
if test; then
      commands
fi

while test; do
      commands
done

for variable in words; do
      commands
done
```

test *expression* or [*expression*]

```
if [ -f .bash_profile ]; then
    echo "You have a .bash_profile.
Things are fine."
else
    echo "Yikes! You have no
.bash_profile!"
fi
```

http://linuxcommand.org/lc3_man_pages/testh.html

Shell-scripting Notes

Bash Scripting	Java Programming
Interpreted	Compiled
Esoteric variable access	Highly structured, Strongly typed
Everything is a string	Strings have library processing
Easy access to files and program	Data structures and libraries
Good for quick & interactive programs	Good for large complex programs

Scripting Style Guide

Do one thing and do it well.

Scripts should generally be <200 lines

Always use spaces, not tabs (indent line with two spaces)

Comment code with ‘#’

https://google.github.io/styleguide/shell.xml