CSE 384 (Lecture 7): Common Unix Tools and Shell Scripting

Spring 2020

Overview

- Editors
- Commonly Used Tools
  - Editors
    - vi
  - Grep (regex)
  - Locate
  - Find
  - Sed
  - Awk
  - Tar
  - Gzip

Editors - create/edit text files

- Vi / Vim originally written in 1976 by Bill Joy, a University of California at Berkley student who later went on to co-found Sun Microsystems

- Emacs originally written in 1976 by Carl Mikkelsen, David A. Moon and Guy L. Steele Jr has over 10,000 built-in commands and its user interface allows the user to combine these commands into macros to automate work

- ex - EXtended, is a line editor for Unix systems

- **GNU nano** is a text editor for Unix-like computing systems or operating environments using a command line interface.
- gedit is the default text editor of the GNOME desktop environment and part of the GNOME Core Applications.

VI/VIM - screen-oriented text editor

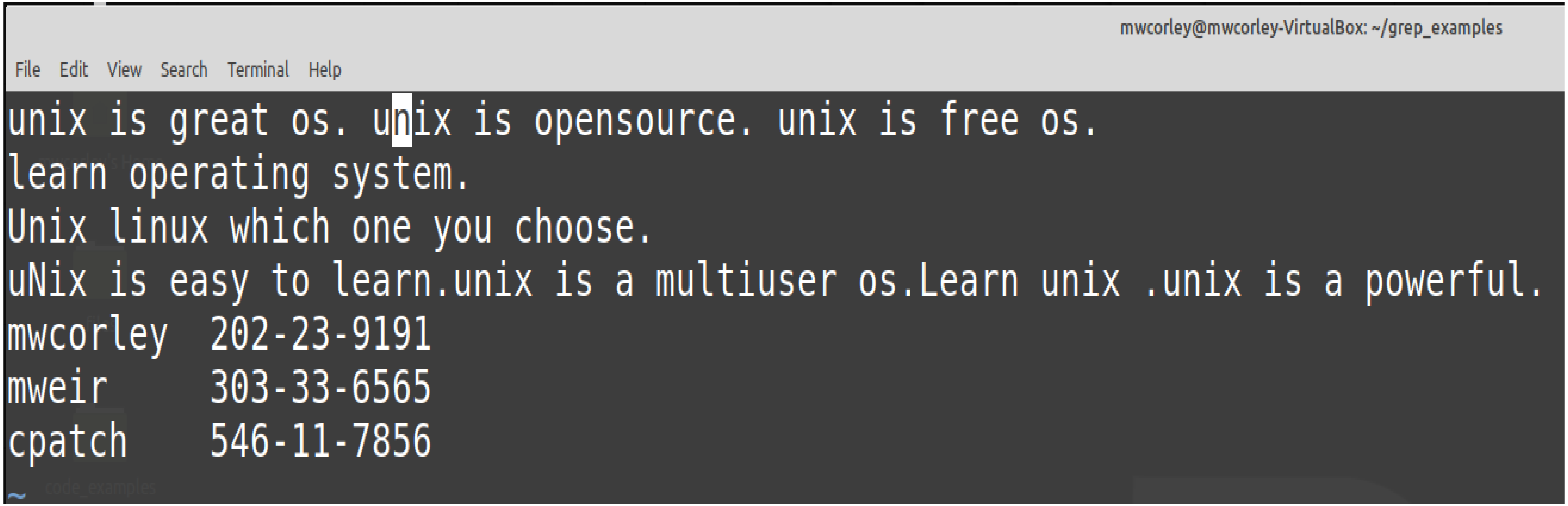
- vi was originally written in 1976 by Bill Joy, a University of California at Berkley student who later went on to co-found Sun Microsystems
  - lightweight and fast, and always available.
  - Important (these days) if the system has no graphical interface
  - POSIX compatibility on Unix systems, requires that vi be present
    - $ vi file.txt
      - *i, o, a*, etc (insert mode), esc (command mode) ~, cw, dw, *dNw, dd,*
      - *Ex – line editor: $:%s/expression/replacement*
      - See vi Cheat sheet
        - *http://www.atmos.albany.edu/daes/atmclasses/atm350/vi_cheat_sheet.pdf*

Class Text: The Linux Command Line (Chapter 12)

Commonly used tools…

- grep – search for patterns in text files
- find – search for files in a directory hierarchy
- locate – find files by name
- AWK – (text processing language) pattern scanning and processing
- sed - stream editor for filtering and transforming text
- tar – (tape) archiving utility
- gzip – (gnu) zip – compress and expand files

Sample test file for examples: geekfile.txt


mwcorley@mwcorley-VirtualBox: ~/grep_examples
File Edit View Search Terminal Help

```
unix is great os. unix is opensource. unix is free os.
learn operating system.
Unix linux which one you choose.
uNix is easy to learn.unix is a multiuser os.Learn unix .unix is a powerful.
mwcorley  202-23-9191
mweir     303-33-6565
cpatch    546-11-7856
~
```

Grep (G/re/p) - “global regular expression print”

- searches text files for the occurrence text matching a specified regular expression and outputs any line containing a match to standard output
    - Regular expression
        - special text string for describing a search pattern. You can think of regular expressions as wildcards on steroids.
    - Resources / Examples
        - Grep
            - https://www.geeksforgeeks.org/grep-command-in-unixlinux/
        - Regex
            - https://www.regular-expressions.info/

        - See Class Text: The Linux Command Line Chapter 19

Grep Examples: Class Text: The Linux Command Line Chapter 19

```
$ grep -i "UNix" geekfile.txt
```

- Search and match lines “UNix” string , disregard case

```
$ grep -ic "UNix" geekfile.txt
```

- Like previous, but count the match occurrences

```
$ grep -i "^UNix" geekfile.txt
```

- Search and match lines beginning with string “UNix”, disregard case

```
$ grep -i '^Unix.*os.$' geekfile.txt
```

- Search and match lines beginning with string “UNix”, followed by any number of characters, and ending with string “os.”

```
$ grep '[0-9]\{3\}-\{0,1\}[0-9]\{2\}-\{0,1\}[0-9]\{4\}' geekfile.txt
```

- Match social security numbers -- assume search applications for PII (personally identifiable information)

```
$ grep -H '[0-9]\{3\}-\{0,1\}[0-9]\{2\}-\{0,1\}[0-9]\{4\}' geekfile.txt
$ grep -l '[0-9]\{3\}-\{0,1\}[0-9]\{2\}-\{0,1\}[0-9]\{4\}' f*.txt
```

Common tools for locating files

- Locate
  - Find files by name
- Find
  - Search for files in a directory hierarchy
- xargs
  - Build and execute command lines from standard input

Chapter 17: Searching for Files

Locate - find files by name

- performs a rapid database search of pathnames, and outputs every name that matches a given substring
- Example
    - find all the programs with names that begin with zip
        - locate bin/zip
        - locate zip
        - locate zip | grep zip$

Chapter 17: Searching for Files (Locate)

Find - search for files in a directory hierarchy

- searches a given directory (and its subdirectories) for files based on a variety of attributes
  - Uses the notion of tests and actions
- Examples

```
$ find ~ | wc -l
$ find ~ -type d | wc -l
$ find ~ -type f -name "*.c" -size +1k
$ find .  -type f -name 'f*'  -exec ls -l '{}' ';'
$ find .  -type f -name 'f*' -ok ls -l '{}' ';'
$ find .  -type f -name 'f*'  -exec  grep -H '[0-9]\{3\}-\{1\}[0-9]\{2\}-\{1\}[0-9]\{4\}' '{}' ';'
$ find .  -type f -name 'f*' | xargs grep '[0-9]\{3\}-\{1\}[0-9]\{2\}-\{1\}[0-9]\{4\}'
```

Chapter 17: Searching for Files

Xargs

- Accepts input from standard in-put and converts it into an argument list for a specified command
  - Example:

```
$ echo 'one two three' | xargs mkdir
$ echo 'one two three' | xargs rmdir
$ find ~ -type f -name 'f*' -print | xargs ls -l
$ time find ~  -type f -name 'f*'  -exec  grep '[0-9]\{3\}-\{1\}[0-9]\{2\}-\{1\}[0-9]\{4\}' '{}' ';'
$ time find ~  -type f -name 'f*' | xargs grep '[0-9]\{3\}-\{1\}[0-9]\{2\}-\{1\}[0-9]\{4\}'
```

Sed – Stream editor

- text editing on a stream of text, either a set of specified files or standard input.
- Use a single editing command (on the command line) or the name of a script file containing multiple commands
  - performs commands upon each line in the stream of text.

  - sed 's/Unix/windows/gI' geekfile.txt
  - cat geekfile.txt | sed '4s/unix/windows/gI’
  - sed 's/[Ll]inux/Unix/g' geekfile.txt > output.txt

  - https://mwcorley79.github.io/MikeCorley/presentations/TLCL-19.01.pdf#page=325

AWK - interpreted programming language

- AWK is an interpreted programming language, specially designed for text processing. Named after the authors (Alfred Aho, Peter Weinberger, and Brian Kernighan).
  - **Typical uses**
    - Producing formatted text reports,
    - Performing arithmetic operations,
    - Performing string operations, and many more.

Tutorial: https://www.tutorialspoint.com/awk/awk_overview.htm

Wait! What is a Shell again ?

- The **shell** is a program that takes keyboard commands and passes them to the operating system to carry out.
  - Source: The Linux Command Line, Fifth edition, page 2
- The **shell** is the command interpreter in an operating system such as **Unix** or **GNU/Linux**, it is a program that executes other programs.
  - Source: https://www.tecmint.com/different-types-of-linux-shells/
- The **shell** is both an interactive command language and a scripting language, and is used by the operating system to control the execution of the system using shell scripts
  - Source: https://en.wikipedia.org/wiki/Shell_script

Background

- Bourne Shell (***sh***): Steven Bourne – 1977
  - AT&T Bell Labs for V7 UNIX, remains a useful shell today
  - introduced control flows, loops, and variables into scripts, providing a more functional language to interact with the operating system (both interactively and noninteractively).
- C Shell (***csh***): Bill Joy - 1978
  - A graduate student at the University of California, Berkeley, developed for Berkeley Software Distribution (BSD) UNIX systems
  - A key design objectives for the C shell was to create a scripting language that looked similar to the C language.
    - Introduced command history

This material found at: https://developer.ibm.com/tutorials/l-linux-shells/

Background

- Tenex C shell (***tcsh***): Ken Greer, 1983
  - Carnegie Mellon University
  - backward-compatible with csh, but improved its overall interactive features.
- Korn shell (***ksh***): David Korn, 1983
  - used as a scripting language in addition to being backward-compatible with the original Bourne shell.
- Bourne-Again Shell (BASH): Brian Fox, 1989
  - an open source GNU project intended to replace the Bourne shell
  - A superset of the Bourne shell
  - incorporated features from the Korn and C shells
  - One of the most widely used shells

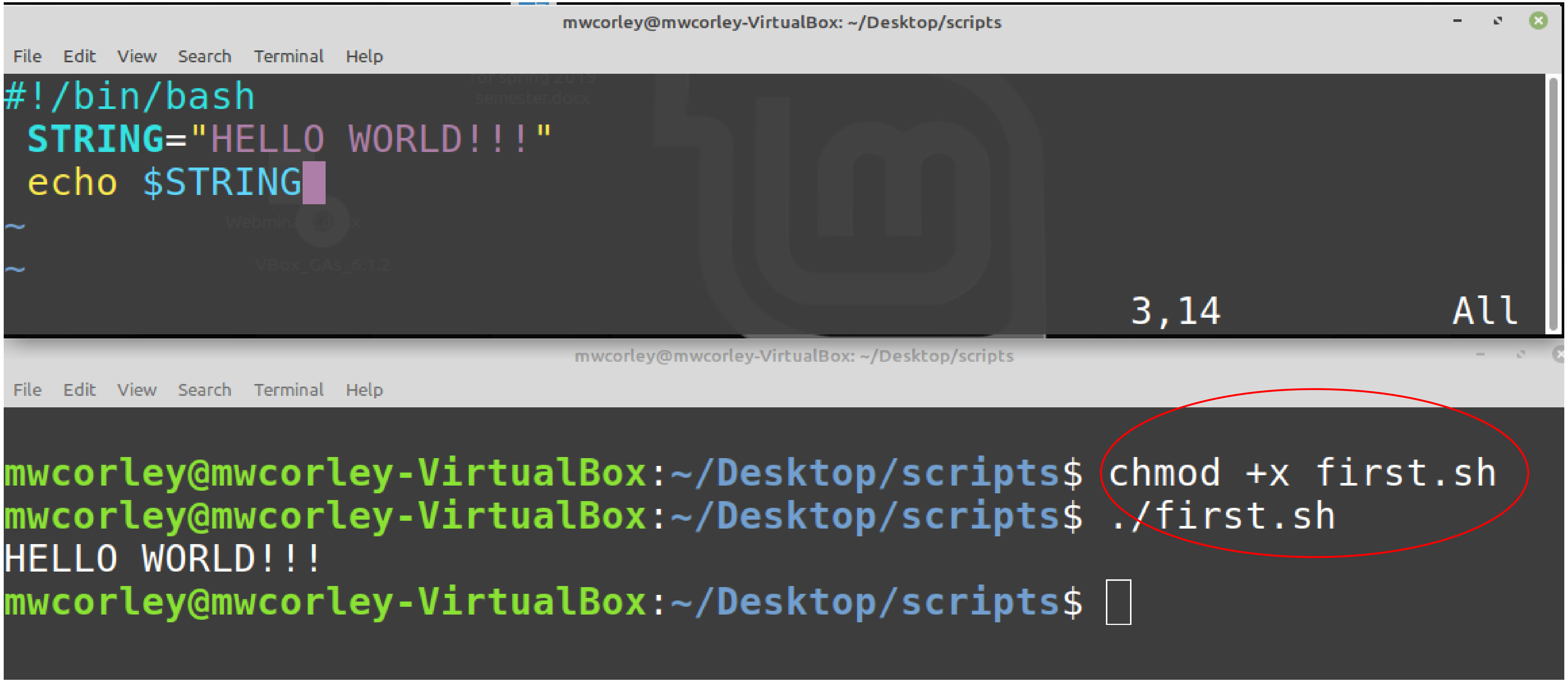
This material found at: https://developer.ibm.com/tutorials/l-linux-shells/

Writing Shell Scripts (Overview)

- We’ve covered the first 9+ chapters of the Linux Command Line
  - We’ve discussed many of the basic commands for navigating, and managing a Linux system.
    - foundational ideas including, Files, I/O redirection and pipelines, a
    - handful of the important tools to accomplish useful work.
    - However, using the shell by supplying command lines (one a time). Granted this is useful,
    - By joining our tools together into programs of our
- The shell can carry out complex sequences of tasks with the command line alone, but shell scripts unlock the power of the Linux system.

Writing Shell Scripts http://linuxcommand.org/lc3_writing_shell_scripts.php#contents

First BASH script: first.sh



mwcorley@mwcorley-VirtualBox: ~/Desktop/scripts
File Edit View Search Terminal Help

```
#!/bin/bash
 STRING="HELLO WORLD!!!"
 echo $STRING
~
~
                                                    3,14          All
```

mwcorley@mwcorley-VirtualBox: ~/Desktop/scripts
File Edit View Search Terminal Help

```
mwcorley@mwcorley-VirtualBox:~/Desktop/scripts$ chmod +x first.sh
mwcorley@mwcorley-VirtualBox:~/Desktop/scripts$ ./first.sh
HELLO WORLD!!!
mwcorley@mwcorley-VirtualBox:~/Desktop/scripts$ 
```

Setting up your environment

- Consider putting your scripts in local bin folder: ~/bin and adding that to your PATH variable: *export PATH=~/bin:"$PATH“*
  - *Update the interactive shell by sourcing it. $ . .bashrc*
  - *Cause the shell to re-read the .bashrc*
    - *.bashrc is shell script that runs when BASH is started*
      - *initializes the interactive shell session*
  - *A second, and better option to update the PATH variable from .bashrc (or the .profile)*
  - If your don’t set your PATH, then no problem, you’ll just have to proceed every script/exe with “./ “ as in ./script_name.sh

Shebang (#!) - or the “bang” line

- The #! syntax is used in scripts to indicate an interpreter for execution under UNIX / Linux operating systems.
- Most script starts with the following line: *#!/bin/interpreter*
  - absolute path to the interpreter need to run the program
  - For BASH scripts
    - #!/bin/bash
  - Python
    - #!/usr/bin/python

https://bash.cyberciti.biz/guide/Shebang

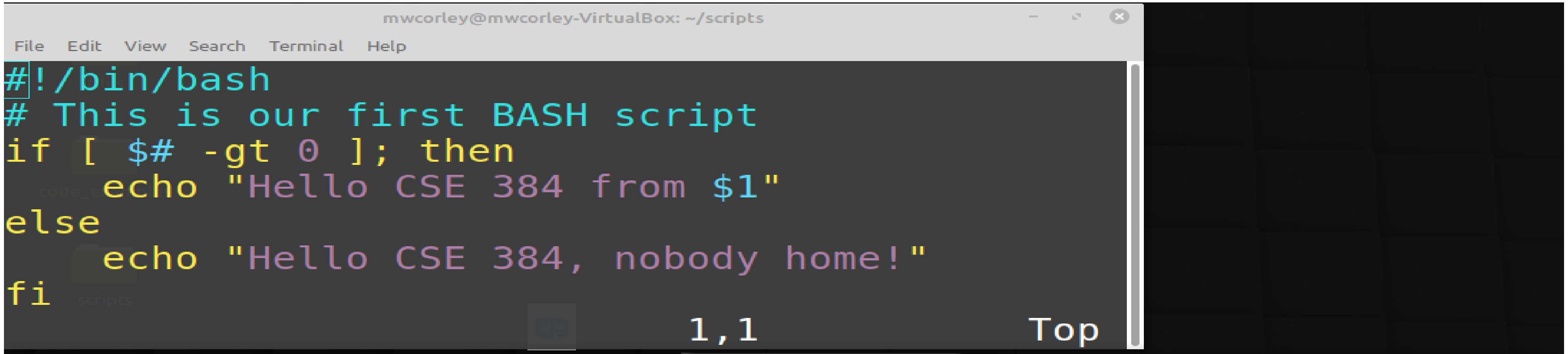
How does #! Work?

- #! - human readable magic number consisting of the byte string 0x23 0x21
  - The kernel uses the *exec()* family of functions (system calls) to determine whether the executable file to be executed is a script (interpreted) or a binary
    - Need to set the execute bit for *exec()* to work
    - exec() replaces a process image with a new process image: consider a shell: *fork()* -> new process -> *exec(new process, process image)* -> *child process*

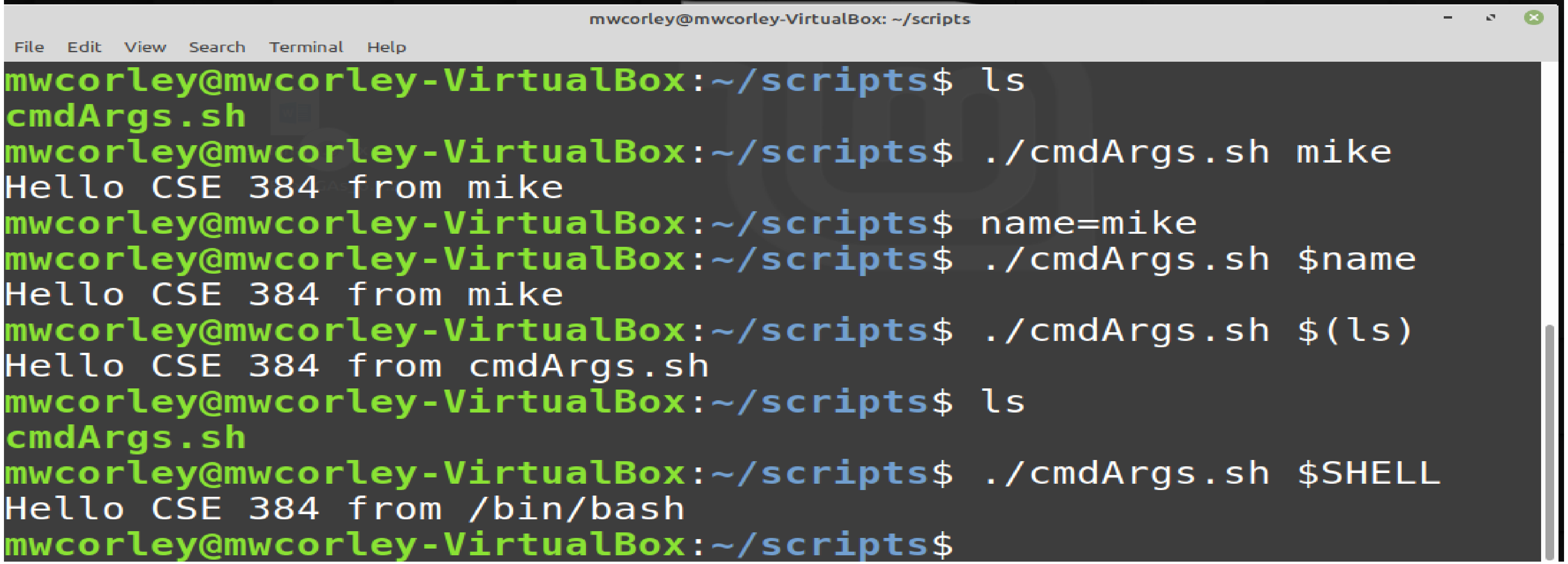
Exec() reads the magic number of executable file, if the “#!” is present the executable present after #! Is run instead

Run: *xxd (hexdump) somescript, man exec, man fork*

First script with command line argument


mwcorley@mwcorley-VirtualBox: ~/scripts
File Edit View Search Terminal Help

```
#!/bin/bash
# This is our first BASH script
if [ $# -gt 0 ]; then
    echo "Hello CSE 384 from $1"
else
    echo "Hello CSE 384, nobody home!"
fi
                                1,1           Top
```


mwcorley@mwcorley-VirtualBox: ~/scripts
File Edit View Search Terminal Help

```
mwcorley@mwcorley-VirtualBox:~/scripts$ ls
cmdArgs.sh
mwcorley@mwcorley-VirtualBox:~/scripts$ ./cmdArgs.sh mike
Hello CSE 384 from mike
mwcorley@mwcorley-VirtualBox:~/scripts$ name=mike
mwcorley@mwcorley-VirtualBox:~/scripts$ ./cmdArgs.sh $name
Hello CSE 384 from mike
mwcorley@mwcorley-VirtualBox:~/scripts$ ./cmdArgs.sh $(ls)
Hello CSE 384 from cmdArgs.sh
mwcorley@mwcorley-VirtualBox:~/scripts$ ls
cmdArgs.sh
mwcorley@mwcorley-VirtualBox:~/scripts$ ./cmdArgs.sh $SHELL
Hello CSE 384 from /bin/bash
mwcorley@mwcorley-VirtualBox:~/scripts$
```

Special variables to be aware of… (*demoSpecialVars1.sh*)

- **$0** - The name of the Bash script.
- **$1, $2, $3,…** - user (command line) arguments supplied to the Bash script.
- **$#** - number arguments supplied to the Bash script.
- **$@** - All the arguments supplied to the Bash script.
  - When quoted expand to separate strings
- **$*** - All the arguments supplied to the Bash script
  - When quoted expand to single string
- **$?** - The exit status of the most recently run process.
- **$$** - The process ID of the current script.
- **$USER** - The username of the user running the script.
- **$HOSTNAME** - The hostname of the machine the script is running on.
- **$SECONDS** - The number of seconds since the script was started.
- **$RANDOM** - Returns a different random number.
- **$LINENO** - Returns the current line number in the Bash script.

Source: https://ryanstutorials.net/bash-scripting-tutorial/bash-variables.php
https://www.bogotobogo.com/Linux/linux_shell_programming_tutorial3_special_variables.php

BASH If Statement: (*demoIf.sh*)

```
if [ <some test> ]
then
<commands>
fi
```

Tests

- brackets ([ ]) reference to the command **test**.
  - All of the operators that test supports can be used in the if statement
  - Sets exit status of 0 (true) or 1 (false) depending on the evaluation of EXPR.
- Run *man test* to see all of the possible operators
  - http://linuxcommand.org/lc3_man_pages/testh.html

Source: https://ryanstutorials.net/bash-scripting-tutorial/bash-if-statements.php

Bash Functions: (demoFunctionsLoops.sh)

- functions don’t allow return a values
  - return value is the exit status of the last statement executed in the function,
    - 0 for success
    - non-zero decimal number in the 1 - 255 range for failure
    - return status can be specified by using the return keyword
      - assigned to the variable $?.
      - return statement terminates the function. You can think of it as the function’s exit status.
- See example
  - Source: https://linuxize.com/post/bash-functions/
  - Source https://ryanstutorials.net/bash-scripting-tutorial/bash-loops.php

BASH arrays: (demoArrays.sh)

- There a number of ways to declare an use arrays, this is one way..

  ```
  distro=("redhat" "debian" "gentoo" "ubuntu" "mint")
  ```

- Source: https://www.cyberciti.biz/faq/finding-bash-shell-array-length-elements/

Examples: examples.tar.gz

- readExample.sh
  - demo BASH keyboard input and output
- demoSpecialVars.sh
  - Demonstrate $@,
- demoFunctionsLoops.sh
  - Demo BASH loops, functions and return values
- demoArrays.sh
  - A quick little array demo
- demoMenu.sh
  - Demonstrate to create menu prompt with case statement
- demoCountWords.sh
  - Test various commands with in a shell script
- GetLectures.sh
  - Demonstrate reading URLs from the text file, and downloading (curl) content from the web