# Lecture Notes for Lecture 3 of CS 5600 (Computer Systems) for the Fall, 2019 session at the Northeastern University Silicon Valley Campus.

## *Shells and System Commands*

Philip Gust,
Clinical Instructor
Department of Computer Science

# Lecture 2 Review

- In lecture 2, we learned that an operating system (OS) is system software that manages computer hardware and  software resources and provides common services for computer programs.
- Early computers had no operating system. Instead, programmers loaded programs directly into memory. When they were done executing, the computer halts.
- As computers became more sophisticated, operating systems enabled a computer to load and run a sequence of programs automatically. The advent of multi-user computers, put new demands on operating systems.

# Lecture 2 Review

- The advent of Unix at AT&T Bell Labs in the early 1970s ushered in a new operating system architecture, and gave rise to a common abstraction known as POSIX that can be implemented by many operating systems.
- Today, many operating systems are POSIX compliant including MacOS, Linux, AIX (IBM), HP-UX (HP). Cygwin provides a largely POSIX-compliant environment for Microsoft Windows.

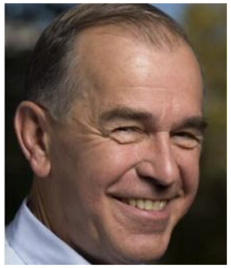
# Lecture 2 Review

- We studied the architecture of a POSIX operating system and saw that it is comprised of nearly concentric layers whose center is the kernel, which abstracts the hardware.
- The outer-most layer includes shells, commands, applications, and services that specify computing tasks to perform.
- The next layer contains libraries that implement common functions such as math, formatting, string manipulation, and other logical operations.
- The layer below that are the system calls that are low-level libraries that interface the layers above it with the operating system kernel.
- The lowest layer is the kernel, which provides managed access to a pool of system resources that are being managed on behalf of programs and services running on the operating system.

# Shells and System Commands

- In this lecture, we will learn about *shells* that run in the outer-most layer of the operating system, and specifically about command line shells.
- Shells are interactive command interpreters that provide access to functionality provided by the libraries and system calls.
- Shells also enable access to commands that implement higher-level functions, and manage the execution of applications and services on behalf of users.
- Finally, shells provide an environment that enables users to create scripts that combine external and built-in commands to automate tasks.

# Shells and System Commands



- The original shell for Unix, by Ken Thompson and then by Stephen Bourne, became known as Bourne shell (**sh**).
- Shells are just regular programs, so several other shells have been created and are are commonly available.
  - C Shell (Bill Joy, 1979) – adopted a more C-like language syntax (**csh**).
  - KornShell (David Korn, 1983) – improved implementation of Bourne shell with extended features (**ksh**)
  - Bash (Brian Fox 1989) – ”bourne-again shell” extended Bourne shell features, fully backward compatible with **sh**. (**bash**)
  - Z shell (Paul Falstad, 1990) includes features from **ksh** and **csh** plus new interactive features. **Zsh** Will be default shell on MacOS Catalina. Can emulate **sh** and **ksh**, but is Incompatible with **bash** and POSIX.
- We will focus on **bash,** the most widely used shell, and currently the default shell for POSIX systems. On Microsoft Windows, It can be accessed from CygWin Terminal program or Linux for Windows.

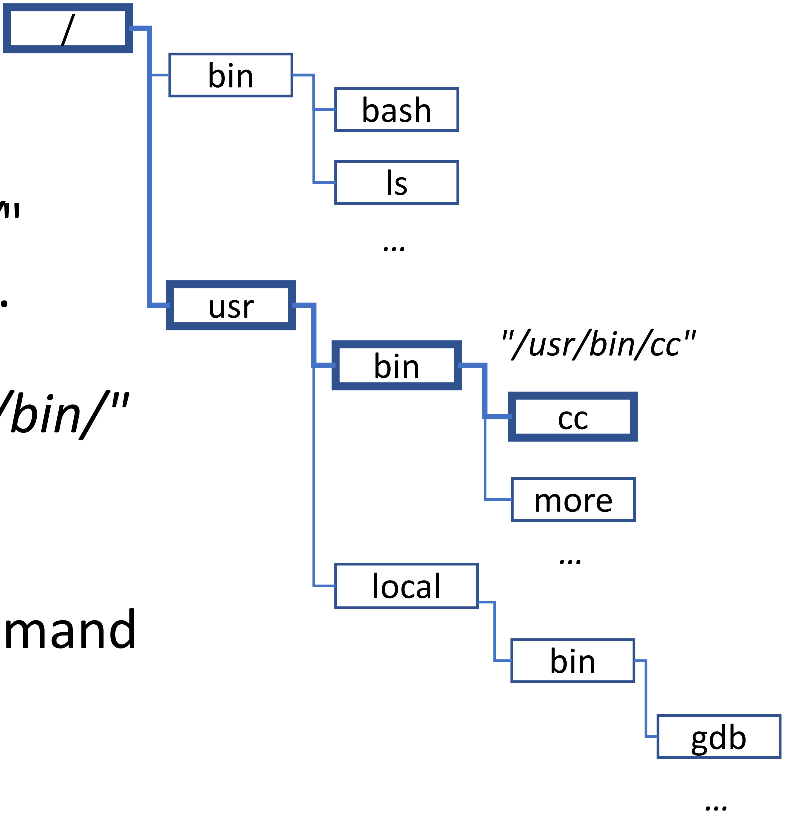
# Shells and System Commands

- When you type in a command at your terminal, the shell interprets the command and runs the program(s) that you specified.
- There are over 250 standard POSIX commands plus numerous others provided through 3rd party software.
- The commands provide many options for customizing their operation and output.
- We will look at some basic POSIX commands, and ways that they can perform useful tasks. In the next lecture, we will see how to use them to create reusable scripts.

# Shells and System Commands

## Accessing Commands from Shells

- Commands are accessed through their *names* and *paths* from the *root directory*. Path elements are separated by a *path separator* "/".
- The root directory is "*/*". The "*/bin/*" directory  contains core commands. The "*/usr/bin/*" directory contains system commands. The "*/usr/local/bin/*" directory contains locally installed commands.
- The full path to the C compiler command is "/usr/bin/cc"



# Shells and System Commands

**Accessing Commands from Shells**

- The shell locates commands by searching for them in "well known" directories. The standard search path includes "/bin", "/usr/bin", and often "/usr/local/bin". Users can configure additional paths.
- When the user runs a command, the shell looks in each directory on the search path until it finds it. The user can also run a utility by specifying its path name.
- The shell remembers a *current working directory* (*cwd*), and commands that operate on files use this no path is given. The cwd can also be specified as "." . The parent directory is "..".
- Paths can be specified either relative to the root directory (*absolute path*) or relative to the current working directory (*relative path*).

# Shells and System Commands

## Listing Directory Content

- Our first the core utility, ”ls”, outputs a listing of files or directories. It provides several options.

| | |
|---|---|
| **ls** | list the contents of the current working directory |
| **ls** -l *name* | the file or the contents of the directory “name” in a long (detailed) format |
| **ls** -al | list all of the contents of the current working directory, including "hidden" files (those that start with a “.”), in a long (detailed format) |
| **ls** *.jpg *.jpeg | list the files ending with “.jpg” or “.jpeg” (presumably JPEG image files) in the current working directory |
| **ls** -t | list the contents of the current working directory, sorted from most recently created first, oldest last |

- Options are specified either by **–optionname** *optionvalue* or as **–optionflag**. A command utility can have multiple options.

# Shells and System Commands

## Listing Directory Content

- This example specifies output listing for the current directory.

```
$ ls -l
total 19621
drwxrwxr-x  2 phil  staff      4096 Dec 25 09:59 uml
-rw-rw-r--  1 phil  staff      5341 Dec 25 08:38 uml.jpg
drwxr-xr-x  2 phil  staff      4096 Feb 15  2006 univ
drwxr-xr-x  2 phil  staff      4096 Dec  9  2007 urlspedia
-rw-r--r--  1 phil  staff    276480 Dec  9  2007 urlspedia.tar
drwxr-xr-x  8 phil  staff      4096 Nov 25  2007 usr
-rwxr-xr-x  1 phil  staff      3192 Nov 25  2007 webthumb.php
-rw-rw-r--  1 phil  staff     20480 Nov 25  2007 webthumb.tar
-rw-rw-r--  1 phil  staff      5654 Aug  9  2007 yourfile.mid
-rw-rw-r--  1 phil  staff    166255 Aug  9  2007 yourfile.swf
```

File names are in the right-most column. File sizes and last modification dates are to the left of that. The other information identifies the owner and access permissions. Directories are identified with ‘d’. We will learn more about these when we study files later.

# Shells and System Commands

## Viewing Files

- Here are several commands to view the contents of text files:
- **cat** *filename*
  Print the contents of the specified file to the screen (one or more files can be specified and each will be printed to the screen - one after the other.
- **more** *filename*
  Print the contents of the specified file, but pause once the contents have filled the screen before printing more; when paused: ENTER = move one line down, SPACEBAR = page down, q = quit printing
- **less** *filename*
  Like more, but you can move forward and backwards throughout the file using the arrow, page-up, and page-down keys (this command is not available on all systems)

# Shells and System Commands

## Viewing Files

- Here are more commands to view the contents of text files. Each is useful in particular circumstances, often in combination with other commands:
- **head** *filename*
  Print the first few lines of a file
- **head** –n 3 *filename*
  Print the first *3* lines of a file
- **tail** *filename*
  Print the last few lines of a file
- **tail** -n 3 *filename*
  Print the last *count* lines of a file

# Shells and System Commands

**Getting Help: *man***

- We can only scratch the surface of the available commands and features provided by the bash shell and commands.
- Fortunately, almost everything you can do at the command line is well-documented in *manual pages* from within the shell.
- When reading a manual page, navigation is the same as when using the "less" tool described earlier (e.g.: arrow keys, page up/down move around the manual, and 'q' will exit).
- Manual pages are divided into sections. Here some common ones:
  1. commands
  2. system calls
  3. library functions
- Some commands have the same name as system and library functions because the commands are built on the functions.

# Shells and System Commands

**Getting for Help: *man* and *apropos***

- Here are examples of accessing the built-in manual pages :
- **man** bash
  Shows manual page for the "bash" shell (peruse this to get a sense for the bash feature set)
- **man** read
  Shows manual page for the built-in "read" command in section 1
- **man** 2 read
  Shows manual  page for the section 2 read system call
- **man** 1 printf
  Shows manual  page for the section 1 printf command
- **man** 3 printf
  Shows manual  page for the section 3 printf library function

# Shells and System Commands

## Editing Files

- Here are commands for editing text files:
- **ed** *filename*
  Edit *filename* using the **ed** line editor. All POSIX systems have **ed**, but in most cases it is more convenient to use the screen-oriented version, **vi**.
- **vi** *filename*
  Edit *filename* using the **vi** screen editor. All POSIX systems have vi in some form, so it is worth learning this editor. (If using CYGWIN install the "vim" package.)
- **nano** *filename*
  Edit *filename* using the nano screen editor. (Most but not all systems have nano; if using CYGWIN install the "nano" package.)

# Shells and System Commands

## Copying, Moving, and Renaming Files and Directories

- These commands move, rename, and copy files and directories:
- **cp** *file1 file2*
  copy a file (*file1* is copied to a new file named *file2*)
- **cp** -R *dir1 dir2*
  recursively copy one directory to another directory.
- **mv** *file1 file2*
  move or rename a file or directory (*file1* is renamed to *file2*)
- **mv** *file1 ~/dir/*
  move *file1* into sub-directory "*dir"* in your *home directory*, ("~").
  - All POSIX systems, each user has a home directory. On MacOS it is "/Users/" and on Linux and on Cygwin under Windows it is "/home/".
  - Native Windows also provides a location for a home directory for each user: "C:\Users\". (Windows uses "\" as path separator and "C:". as volume name)

# Shells and System Commands

## Removing Files and Directories

- Here are commands used to remove files and directories:
- **rm** *file*
  remove or delete a file (one of more files may be specified)
- **rmdir** dir
  remove an empty directory
- **rm** -R *dir1*
  recursively remove a directory and its contents (one or more directories may be specified) - BE EXTREMELY CAREFUL!

# Shells and System Commands

## Working with Directories

- Here are commands used to create directories
- **mkdir** *dir1*
  create directories (one or more directories may be specified)
- **mkdir** -p *dirpath*
  create the directory path, including all intermediate directories necessary to create the specified directory path

  e.g.: if "*a/b/c*" is specified, the subdirectories "*a*" and *"b*" will be created, if necessary, before creating subdirectory "*c*".

# Shells and System Commands

## Counting Lines, Words, and Characters

- The **wc** command counts lines, words and characters of its input.
- **wc** *file*
  counts the number of lines, words, and characters in the file:
  ```
  12   35   292 file
  ```
- **wc** -l *file*
  counts only the number of lines in the file:
  ```
  12   some_file
  ```
- **wc** -w *file*
  counts only the number of words in the file:
  ```
  12   file
  ```
- **wc** -c *file*
  counts only the number of characters in the file:
  ```
  292 file
  ```

# Shells and System Commands

## Sorting Input

- The **sort** command sorts its input according to specifications. It can treat its input as fields, and can sort them lexically and numerically, in forward or reverse order.
- **sort**
  Sorts lines in ascending lexical order
- **sort** -r
  Sorts lines in descending lexical order
- **sort** –n -r
  Sorts lines in descending numerical order
- **sort** –u
  Sorts lines and eliminates duplicates

# Shells and System Commands

**Command History**

- Bash keeps track of recent commands you have issued.
- Each time you press the up-arrow key, bash will place the next newest command you issued on your command line for you.
- You can then edit the command and re-execute it.
- You can also use the **history** command to list the history
- You an also re-execute a command by specifying its number in the command list. The following re-executes command number 512:

```
!512
```

# Shells and System Commands

**Command Completion**

- Another feature of bash is that you can use the TAB key to complete something you've partially typed (e.g.: a command, filename, environment variable, etc.).
- Suppose you have a file named "constantine-monks-and-willy-wonka.txt" in your directory and want to edit it.
- You can type "**less** will", press the TAB key, and the shell will fill in the rest of the name for you, assuming the completion is unique (i.e.: no other items in the directory begin with "will").
- If the completion is not unique, you can press TAB twice and bash will print out all the possible completions that are available.

# Shells and System Commands

**Pipes**

- The pipe symbol "|" (<shift>+backslash on most keyboards) is used to direct the *standard output* of one command to the *standard input* of another.
- This is an very powerful feature. You can pipe together commands that each do a simple task in ways that allow you to accomplish more complex tasks without writing custom code.
- Most programs take their input from either a specified file, or standard input if no file is specified, and write their output to standard output. Errors are written to standard error.
- By default, standard input in bash is normally the keyboard, and standard output and standard error normally goes to the terminal.

# Shells and System Commands

## Pipes

- Here are some examples:
  - **ls** -l /etc | **more**  
    Pipes the output of the long format directory list command "**ls** -l /etc" through the "**more**" command to view the list a page at a time.
  - **ls** /etc | **sort** | **tail** -n 5  
    Pipes the output of the regular directory list command "**ls**" through the "**sort**" command, then lists the last 5 entries.
  - **ls** /etc | **head –n 3| tail** -n 1  
    Lists the third entry of the regular directory list command "**ls”**.
  - **sort** *file* | **uniq** -c  
    Sorts lines of *file* in ascending order, and outputs a count of unique occurrences of each line.
  - **ls** *dir* | **wc** -l  
    Counts the number of files and directories contained in directory *dir*.

# Shells and System Commands

## Pipes

- Most commands that accept a filename as an argument can also read data piped from another program as standard input.
  - The commands
    **head** -n 1 *file*

    and
    **cat** *file* | **head** -n 1

    result in the same output: the first line of "*file*". The first form does not require running another program.

# Shells and System Commands

## Pipes

- Suppose a file contains file names separated by whitespaces:

  ```
  /a/file1
  /a/b/file2  /a/c/e/file2
  /a/f/g/h/file4
  ```

- The *xargs* command can be used to run another command with the file names as command line arguments.
  - **cat** *file* | **xargs ls** –l
    Display a long list of the files named in *file*.
  - **cat** file | **head** –n 1 | **xargs wc** –w
    List the number of words in the first file named in *file*.
  - **cat** file | **xargs**
    Echoes the files named in *file* on a single line (echo by default)
- Caution: needs special treatment for file names with whitespaces.

# Shells and System Commands

## Redirection

- The *redirection* directives, ">" and ">>" can be used on the output of most commands to redirect their standard output to a file.
  - **head** -n 10 *file* > *new_file*  
    Redirects the standard output of **head** to a file *new_file*
  - **head** -n 10 *file* >> *existing_file*  
    Redirects the standard output of **head** and appends to the end of *existing_file*, which will be created if it does not already exist
- The directives can also be written as "1>" and "1>>" because the index 1 is associated with standard output.

# Shells and System Commands

## Redirection

- The *redirection* directives, "2>" and "2>>" can be used to redirect a errors written the *standard error* of a command to a file. The index 2 is associated with standard error
  - **cat** *file* 2> *err_file*
    Redirects the standard error of **cat** to a file *err_file* to capture the error message if *file* is not found or cannot be read.
  - **cat** *file* 2>> *err_file*
    Appends the standard error of **cat** to a end of *err_file* to capture the error message if *file* is not found or cannot be read.
  - **echo** "error message" 1>&2
    Redirects the standard output of **echo** to standard error.
  - **cat** file1 file2 file3 2>> /dev/null
    Redirects standard error reporting non-existent file2 to the special device file "/dev/null" which discards all characters sent to it.

# Shells and System Commands

## Redirection

- The redirection directive "<" can be used to direct the contents of a file to standard input of a command.
  - **head** -n 10 < *file*
    Redirects input to the **head** command from *file*
  - **tr** -s " \t\n" " " < *file*
    Transliterates whitepace characters (space, tab, newline) from *file* into spaces, and "squeezes" multiple spaces into single space
- The directives can also be written as "0<" because the index 0 is associated with standard input.

# Shells and System Commands

## Redirection

- The directive << *marker* can be used to direct the multiple lines following to standard input, delimited by matching markers.

  -
    ```
    wc -l  << EOF
    This ls line 1
    This is line 2
    This is line 3
    EOF
    ```

    Counts the three following lines, up to but not including the trailing *EOF* marker. The *EOF* marker can be any character sequence that does not appear in the input lines

# Shells and System Commands

**Redirection**

- The directive <<< can be used to direct the content of a string to standard input.
  - **wc** -w  <<< "word1 word2 word3"
    Counts 3 words in a string separated by whitespace characters
- This is often more efficient than the equivalent piped operation because it does not require running another command:
  - **echo** "word1 word2 word3" | **wc** -w
    Counts 3 words in a string separated by whitespace characters

# Shells and System Commands

## Searching for Files by Name

- To find files and directories within the file system use the **find** command. Searches recursively for names specified by options:
  - **find** . -name *name*
    Find files and directories named *name* in current directory recursively
  - **find** . -maxdepth 2 -name *name*
    Finds files and directories named *name* only in current directory and immediate subdirectories
  - **find** . -type f -name '**cunit**'
    Find only regular files that contain "cunit" in the current directory recursively (wildcard characters evaluated by **find** instead of **bash**).
  - **find** * –type d -exec **echo** {} +
    Echoes subdirectories in current directory recursively on one line
  - **find** * –type d -print0 | xargs -0
    Echoes subdirectories in current directory recursively on one line

# Shells and System Commands

**Searching for Strings in Files**

- To search within the contents of a file use the "grep" command. Here are some examples with literal strings:
  - **grep** "find me" *some_file*
    prints all the lines in *some_file* that contain "find me"
  - **grep** "find me" *some_directory/*.txt*
    prints all lines of text files in *some_directory* that contain "find me"
  - **grep** -R "find me" *some_directory*
    prints all the lines of files in *some_directory* or any child directories recursively that contain "find me"

# Shells and System Commands

## Searching for Strings in Files

- **grep** can also match based on regular expression patterns. In fact, ‘grep’ stands for “global regular expression print”.
- A regular expression has  a number of pattern matching characters and expressions. Here are examples of *extended patterns* (use -E) :
    - ^ matches the beginning of string
    - $ matches the end of string
    - . matches a single character
    - * matches zero or more occurrences of the preceding character or pattern
    - + matches one or more occurrences of the preceding character or pattern
    - {3} or {2-5} or {2,} matches exactly 3, between 3 and 5, or at least 2 occurrences of the preceding character or pattern
    - [0-9] or \d matches a digit
    - [a-zA-Z] or [[:alpha:]] matches an ASCII letter or Unicode alpha character
    - [^[:alpha:]] matches any character that is not a Unicode alpha character

# Shells and System Commands

## Searching for Strings in Files

- Here are some examples matching with extended patterns:
  - **grep** –E '^ {3}'
    match lines that start with three spaces
  - **grep** –E 'edu$'
    match lines the end in “edu”
  - **grep** –E '^[[:alpha:]]*$'
    match lines that contain only alpha characters
  - **grep** –E '^\d+ +\d+ +\d+$'
    match lines that have three columns of digits separated by blanks
  - **grep** -E '^[a-zA-Z.-_]+@[a-zA-Z.-_]+[.](edu|com|org)$’
    match email addresses

# Shells and System Commands

## Transforming Strings in Files

- The program, **sed** (stream editor) can transform strings in the input stream, matching literal strings or regular expression patterns using the same syntax as the *vi* text editor
  - **sed** 's/search/replace/'
    replaces the first occurrence of ”search” on each line with “replace”
  - **sed** 's/search/replace/g '
    replaces every occurrence of ”search” on each line with “replace”
  - **sed** 's/search//g'
    delete every occurrence of “search” on each line
  - **sed** -E 's/([0-9a-fA-F]+)/0x\1/ '
    prefix hexadecimal numbers by “0x”
  - **sed** -E ' /^# /d '
    delete lines that begin with “# “

# Shells and System Commands

**Processing Fields Files**

- The ‘cut’ command line utility is for cutting sections from each line of files and writing the result to standard output.
- It can be used to cut parts of a line by delimited, byte position, or by a separator character.
- Suppose we have a comma-separated CSV file “names.csv”:

```
John,Smith,34,London
Arthur,Evans,21,Newport
George,Jones,32,Truro
```

- The command ***cut*** *–d ', ' –f 1,4 names.csv* outputs columns 1 and 4 separated by the input delimiter:

```
John,London
Arthur,Newport
George,Truro
```

# Shells and System Commands

**Processing Fields Files**

- The ‘awk’ utility provides a powerful language for operating on input lines that can process fields of information using patterns and field specifiers. It is very useful in shell scripts.
- ‘awk’ was named for the initials of its three authors: Al Aho, Peter Weinberger, and Brian Kernighan.
    - **awk** '{ print $2, $1 } '
      prints the first two space-separated fields in opposite order
    - **awk** 'BEGIN { FS = "," } { print $2 }’
      prints the second field with ‘,’ as the field separator
    - **awk** 'BEGIN { count=0 } { count += $1 } END { print count } '
      computes sum of numbers in input
    - **awk** '/^[0-9]+/{ print $2 }'
      prints the second field of any line that begins with digits