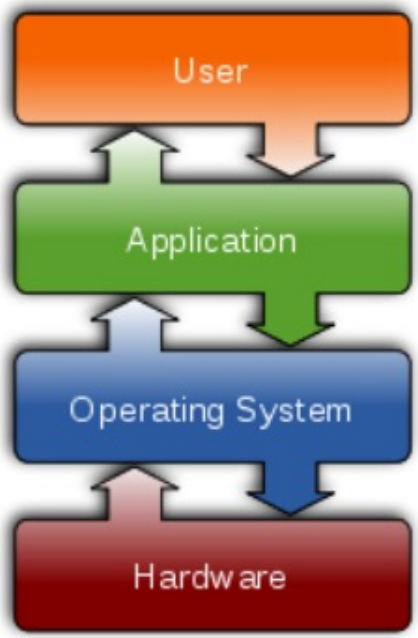
BASH Shell Scripting

Content credits: http://www.cs.washington.edu/391/

Operating systems

- **operating system**: Manages activities and resources of a computer.
  - software that acts as an interface between hardware and user
  - provides a layer of abstraction for application developers

- features provided by an operating system:
  - ability to execute programs (and multi-tasking)
  - memory management (and virtual memory)
  - file systems, disk and network access
  - an interface to communicate with hardware
  - a user interface (often graphical)



- **kernel**: The lowest-level core of an operating system.

Unix

- brief history:
  - Multics (1964) for mainframes
  - Unix (1969)
  - Linus Torvalds and Linux (1992)

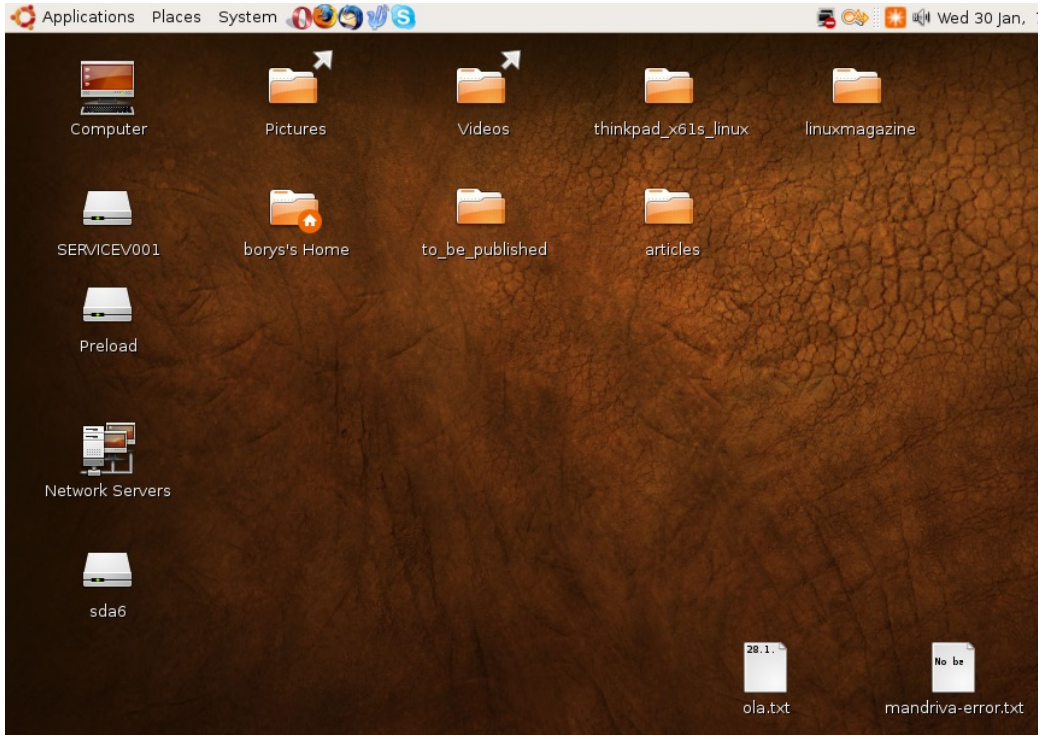
- key Unix ideas:
  - written in a high-level language (C)
  - virtual memory
  - hierarchical file system; "everything" is a file
  - lots of small programs that work together to solve larger problems
  - security, users, access, and groups
  - human-readable documentation included

Linux

- **Linux**: A kernel for a Unix-like operating system.
  - commonly seen/used today in servers, mobile/embedded devices, ...
- **GNU**: A "free software" implementation of many Unix-like tools
  - many GNU tools are distributed with the Linux kernel
- **distribution**: A pre-packaged set of Linux software.
  - examples: Ubuntu, Fedora
- key features of Linux:
  - **open source software**: source can be downloaded
  - free to use
  - constantly being improved/updated by the community

Linux Desktop

- X-windows
- window managers
- desktop environments
  - Gnome
  - KDE



Shell

- **shell**: An interactive program that uses user input to manage the execution of other programs.
  - A command processor, typically runs in a text window.
  - User types commands, the shell runs the commands
  - Several different shell programs exist:
    - `bash` : the default shell program on most Linux/Unix systems
    - We will use bash
    - Other shells: Bourne, csh, tsch
- Why should I learn to use a shell when GUIs exist?

Why use a shell?

- Why should I learn to use a shell when GUIs exist?
  - faster
  - work remotely
  - programmable
  - customizable
  - repeatable

Basic commands

command	description
echo	It prints the text to the terminal.
date	lists files in a directory
man	brings up the manual for a command

- Shell scripting is case sensitive

```
$ echo $SHELL       → SHELL is a variable
$ echo $HOME        → HOME is a variable
$ echo “Hello Linux!”
$ date
$ date –u
$ date --date="tomorrow"
$ man ls
$ man echo
```

Environment Variables

- Already defined by shell and are part of environment.
- These variables are exported to the processes by shell.
- All the environment variables are written in capital letters.
- Use printenv or env command in the terminal.

```
$ echo $BASH_VERSION   #prints bash version
$ echo $HOME   #current user's parent/home directory
$ echo $USER   #current user
$ echo $PATH   #list of all directories that shell consults to run a command
```

Shell commands

command	description
`exit`	logs out of the shell
`ls`	lists files in a directory
`pwd`	**p**rint the current **w**orking **d**irectory
`cd`	**c**hanges the working **d**irectory
`which`	locates the command and prints its path

```
$ pwd
/home/user_name
$ cd Documents
$ ls
file1.txt file2.txt
$ ls -l
-rw-r--r-- 1 username group 0 2016-03-29 17:45 file1.txt
-rw-r--r-- 1 username group 0 2016-03-29 17:45 file2.txt
$ cd ..
$ which ls
$ exit
```

Relative directories

directory	description
`.`	the directory you are in ("working directory")
`..`	the parent of the working directory (`../..` is grandparent, etc.)
`~`	your **home** directory (on many systems, this is `/home/`***username***)
`~`***username***	***username***'s **home** directory
`~/Desktop`	your desktop

Directory commands

command	description
`ls`	list files in a directory
`pwd`	**p**rint the current **w**orking **d**irectory
`cd`	**c**hanges the working **d**irectory
`mkdir`	create a new directory
`rmdir`	delete a directory (must be empty)

```
$ mkdir dirname
$ rmdir dirname
```

- some commands (cd, exit) are part of the shell ("builtins")
- others (ls, mkdir) are separate programs the shell runs
- Use type command to find out

“Type” command

- type command displays the type of the command.

```
$ type cp      #prints “cp is /bin/cp”
$ type echo    #prints “echo is shell builtin”
$ type while   #prints "while is a shell keyword"
$ type -t cp   #prints "file"
```

- There are five types of commands:
  - Alias, Function, Shell built-in, Keyword, File
- Aliases are shortcuts which exist in the memory.
- Make use of aliases in case of long commands.
- Aliases (with options) can also be created and saved in ~/.bashrc file in the following way:

```
alias mylinux=‘ls -la’
```

Shell commands

- many accept **arguments** or **parameters**
  - example: cp (copy) accepts a source and destination file path

- a program uses 3 streams of information:
  - stdin, stdout, stderr (standard in, out, error)

- **input**: comes from user's keyboard
- **output**: goes to console
- **errors** can also be printed (by default, sent to console like output)

- parameters vs. input
  - *parameters*: before Enter is pressed; sent in by shell
  - *input*: after Enter is pressed; sent in by user

Command-line arguments

- most options are a `-` followed by a letter such as `-c`
  - some are longer words preceded by two `-` signs, such as `--count`
- options can be combined: `ls -l -a -r` can be `ls -lar`
- many programs accept a `--help` or `-help` option to give more information about that command (in addition to `man` pages)
  - or if you run the program with no arguments, it may print help info
- for many commands that accept a file name argument, if you omit the parameter, it will read from standard input (your keyboard)

Shell/system commands

command	description
`man or info`	get help on a command
`clear`	clears out the output from the console
`exit`	exits and logs out of the shell

command	description
`date`	output the system date
`cal`	output a text calendar
`uname`	print information about the current system

- "man pages" are a very important way to learn new commands

```
man ls
man man
```

File commands

command	description
`cp*`	copy a file
`mv*`	move or rename a file
`rm*`	delete a file
`touch`	create a new empty file, or update its last-modified time stamp (-m)

```
$ touch file1
$ touch file1 /home/files/file2 file3
$ touch –m Hello.txt
$ rm filename
$ cp source/filename destination/
$ mv source/filename destination/
```

- *Exercise* : Given several albums of `.mp3` files all in one folder, move them into separate folders by artist.

Unix file system

directory	description
`/`	root directory that contains all others (drives do not have letters in Unix)
`/bin`	programs
`/dev`	hardware devices
`/etc`	system configuration files ▪ `/etc/passwd` stores user info ▪ `/etc/shadow` stores passwords
`/home`	users' home directories
`/media`, `/mnt`, …	drives and removable disks that have been "mounted" for use on this computer
`/proc`	currently running processes (programs)
`/tmp`, `/var`	temporary files
`/usr`	user-installed programs

Links

command	description
`ln`	create a link to a file
`unlink`	remove a link to a file

- **hard link**: Two names for the same file.

```
$ ln orig other_name
```

  - the above command links `other_name` as a duplicate name for `orig`
    - if one is modified, the other is too; follows file moves

- **soft (symbolic) link**: A reference to another existing file.

```
$ ln -s orig_filename nickname
```

  - the above command creates a reference nickname to the file orig_filename
    - `nickname` can be used as though it were `orig_filename`
    - but if `nickname` is deleted, `orig_filename` will be unaffected

File examination

command	description
`cat or tac`	output a file's contents on the console
`more or less`	output a file's contents, one page at a time
`head, tail`	output the first or last few lines of a file
`wc`	count words, characters, and lines in a file
`du`	report disk space used by a file(s)
`diff`	compare two files and report differences

```
$ cat > demo.txt                  #enter contents in a file (ctrl+d to save)
$ cat demo.txt                    #displays file contents
$ cat demo1.txt>demo2.txt         #copy contents of demo1 to demo2
```

- Append the content at the end of the file?

```
$ echo "new text at the end">>demo.txt
```

File commands

- tac - Reverse of cat command. It displays the contents of file in reverse order.
  - `$ tac filename`

```
$ head filename        #first 10 lines
$ tail filename        #last 10 lines
$ more filename
$ less filename        #lets you navigate upwards as well as downwards.
$ wc filename  #lines, words and characters/bytes in the file
$ wc –w filename       #words
$ wc –c filename       #count of bytes
$ wc -m filename       #count of characters
$ du                   #disk usage
$ du –h                #human readable
$ du –h –s             #summarize
$ diff file1 file2     #how to change the first file to make it match the
second file.
```

- line numbers corresponding to the first file,
- a letter (**a** for *add*, **c** for *change*, or **d** for *delete*), and
- line numbers corresponding to the second file.

Searching and sorting

command	description
`grep`	search a file for a given string (useful options: –v and –i)
`sort`	convert an input into a sorted output by lines
`uniq`	strip duplicate (adjacent) lines
`find`	search for files within a given directory
`locate`	search for files on the entire system
`which`	shows the complete path of a command

```
$ grep "ab" sample.txt
$ sort sample.txt      #what happens with -r
$ find /sample_directory -name filename
$ find . -name "*.pdf"
$ locate filename      #searches in all possible locations; faster
$ uniq filename
```

- *Exercise* : Given a text file `names.txt`, display the students arranged by the reverse alphabetical order of their names.

Keyboard shortcuts

^***KEY*** means hold `Ctrl` and press ***KEY***

key	description
`Up arrow`	repeat previous commands
`^R` *`command name`*	search through your history for a command
`Home/End or ^A/^E`	move to start/end of current line
`"`	quotes surround multi-word arguments and arguments containing special characters
`*`	"wildcard" , matches any files; can be used as a prefix, suffix, or partial name
`Tab`	auto-completes a partially typed file/command name
`^C or ^\`	terminates the currently running process
`^D`	end of input; used when a program is reading input from your keyboard and you are finished typing
`^Z`	suspends (pauses) the currently running process
`^S`	don't use this; hides all output until ^Q is pressed

Shell History

- The shell remembers all the commands you've entered
- Can access them with the `history` command
- Can execute the most recent matching command with !
  - Ex: `!less` will search backwards until it finds a command that starts with `less`, and re-execute the entire command line
- Can execute also execute a command by number with !

```
165  19:36   ls
166  19:37   cat test.txt
167  19:38   pwd
168  19:40   history
```

Ex: !166 will execute: "`cat test.txt`"

Programming

command	description
`javac` ***`ClassName`*`.java`**	compile a Java program
`java` ***`ClassName`***	run a Java program
`python, perl, ruby, gcc, sml, ...`	compile or run programs in various other languages

- *Exercise* : Write/compile/run a program that prints "Hello, world!"

```
$ javac Hello.java
$ java Hello
Hello, world!
$
```

Output redirection

command `>` ***filename***

- run ***command*** and write its output to ***filename*** instead of to console;
  - think of it like an arrow going from the command to the file...
  - if the file already exists, it will be overwritten (be careful)
  - `>>` appends rather than overwriting, if the file already exists

- Example: `ls -l > myfiles.txt`
- Example: `python Foo.py >> Foo_output.txt`
- Example: `cat > somefile.txt`

Input redirection

command* < *filename

- run ***command*** and read its input from ***filename*** instead of console
  - whenever the program prompts the user to enter input, it will instead read the input from a file
  - some commands don't use this; they accept a file name as an argument

- Example: `python Guess < input.txt`
- *Exercise*: run hello world with the input stream as a file instead of the console
- *Exercise*: Also change the output stream to write the results to file

- again note that this affects *user input*, not *parameters*
- useful with commands that can process standard input or files:
  - e.g. `grep`, `more`, `head`, `tail`, `wc`, `sort`, `uniq`, `write`

Combining commands

command1 | ***command2***

- run ***command1*** and send its console output as input to ***command2***
- very similar to the following sequence:

```
command1 > filename
command2 < filename
rm filename
```

- Examples:

```
diff students.txt names.txt | less
sort names.txt | uniq
```

Misusing pipes and cat

- Misuse of `cat`
  - bad: `cat input_filename | command`
  - good: `command < input_filename`
  - bad: `cat filename | more`
  - good: `more filename`
  - bad: `command | cat`
  - good: `command`

Commands in sequence

command1 ; command2

- run ***command1*** and then ***command2*** afterward (they are not linked)

command1 && command2

- run ***command1***, and if it succeeds, runs ***command2*** afterward
- will not run ***command2*** if any error occurs during the running of 1

- Example: Make directory `songs` and move my files into it.

```
mkdir songs && mv *.mp3 songs
```