Linux/Unix: The Command Line

Adapted from materials by Dan Hood

Essentials of Linux

- Kernel
  - Allocates time and memory, handles file operations and system calls.
- Shell
  - An interface between the user and the kernel.
- File System
  - Groups all files together in a hierarchical tree structure.

Format of UNIX Commands

- UNIX commands can be very simple one word commands, or they can take a number of additional arguments (parameters) as part of the command. In general, a UNIX command has the following form:

  command options(s) filename(s)

- The *command* is the name of the utility or program that we are going to execute.
- The *options* modify the way the command works. It is typical for these options to have be a hyphen followed by a single character, such as **-a**. It is also a common convention under Linux to have options that are in the form of 2 hyphens followed by a word or hyphenated words, such as **--color** or **--pretty-print**.
- The *filename* is the last argument for a lot of UNIX commands. It is simply the file or files that you want the command to work on. Not all commands work on files, such as **ssh**, which takes the name of a host as its argument.

Common UNIX Conventions

- In UNIX, the command is almost always entered in all lowercase characters.
- Typically any options come before filenames.
- Many times, individual options may need a word after them to designate some additional meaning to the command.

Familiar Commands: scp (Secure CoPy)

- The **scp** command is a way to copy files back and forth between multiple computers.
- Formats for this command:
  - **scp path/local_name username@hostname:path/remote_name**
  - **scp username@hostname:path/remote_name path/local_name**

Familiar Commands: ssh (Secure SHell)

- The **ssh** command is a way to securely connect to a remote computer.
- Formats for this command:
  - **ssh hostname**
  - **ssh username@hostname**
  - **ssh hostname -l username**
- If you do not specify the username, it will assume that you want to connect with the same username that you have on this local computer.
  - Since you have a single username for all computers on the network, you don't need to explicitly enter a username.
  - But you can give a username, and will need to if your local username is different that your network username. This might be the case if you are logging on from home.
- Demo with connecting with different usernames.

Familiar Commands: passwd (change PASSWorD)

- The **passwd** command changes your UNIX password.
- This command is an example of a “no argument” command. Thus, the format of this command is just the command name itself.

  passwd

- passwd is an interactive command, as once we have typed it, we need to interact with it.

Familiar Commands: quota (account QUOTA)

- The **quota** command shows you how much storage space you have left in your account.
- When we use this command, we have to specify the -v option to get the information for *our* account.
- The format of this command is:

quota -v

Man Pages

- The **man** command allows you to access the MANual pages for a UNIX command.
- To get additional help on any of the commands listed below, you can always type **man name_of_command** at the command prompt.
- Examples:
  - **man ssh**
  - **man passwd**

Commands

- **ls :** lists the contents of a directory
  - l : long directory listing
  - a : lists all files, including files which are normally hidden
  - F : distinguishes between directories and regular files
  - h : ? Look it up using **man**
- **pwd** : prints the current working directory
- **cd** : changes directories
  - The difference between relative and absolute paths.
  - Special characters **.**, **..**, and ~.
- **mkdir** : creates a directory
- **rmdir** : removes a directory (assuming it is empty)
  - If you get an error that the directory isn't empty even though it looks empty, check for hidden files.

Commands

- **rm** : removes a file.
  - f : force deletion
  - r : recursive deletion
- **mv** - moves a file, or renames a file
  - f : forces overwrite, if the destination file exists
- **cp** - copies a file, leaving the original intact
  - f : forces overwrite, if the destination file exists
  - r : recursive copying of directories

Commands

- **cat** : shows the contents of a file, all at once
- **more** : shows the contents of a file, screen by screen
- **less** : also shows the contents of a file, screen by screen
- **head** : used to show so many lines form the top of a file
- **tail** : used to show so many lines form the bottom of a file

Commands

- **lpr** : prints a file
- **alias** : creates an alias for a command.
  - Aliases can be placed in your **.cshrc** login script.
  - Example: alias rm ‘rm –i’.
- **date** : shows the date and time on the current system
- **who** : used to print out a list of users on the current system
- **hostname** : prints the hostname of the current computer
- **whoami** : prints your current username

The UNIX Pipe (|)

- The pipe (|) creates a channel from one command to another. Think of the pipe as a way of connecting the output from one command to the input of another command.
- The pipe can be used to link commands together to perform more complex tasks that would otherwise take multiple steps (and possibly writing information to disk).
- Examples:
  - Count the number of users logged onto the current system.
    - The **who** command will give us line by line output of all the current users.
    - We could then use the **wc -l** to count the number of lines...
    - **who | wc –l**
  - Display long listings in a scrollable page.
    - The **lpq** command will give us a list of the waiting print jobs.
    - **lpq | less**

Commands

- **ps** : lists the processes running on the machine.
  - **ps -u** *username* lists only your processes.
  - **ps -a** : lists all processes running on the machine.
  - The PID column of the listing, provides the information required by the kill command.
- **kill** : terminates a process
  - **kill** *process_id* : sends a terminate signal to the process specified by the *process_id* (PID).
  - In cases where the terminate signal does not work, the command "**kill** ***-9*** *process_id*" sends a kill signal to the process.
- **nice** : runs a process with a lower priority.

The History

- Almost every shell stores the previous commands that you have issued.
- Most shells allow you to press the up arrow to cycle through previous commands. These previous commands are what makes up the history.
- You can save some time typing by reusing previous commands. You can execute them exactly as they are or make small alterations as needed.
- The **history** command shows us the history list of previous commands.

```
linux2 [6]# history
1 20:32 ls
2 20:32 cd courses/
3 20:32 ls
```

- You can clear the history using **history –c**.

Bang (!)

- We can access commands in the history and re-execute them using an exclamation mark (!).
  - Many UNIX users refer to this as "bang".
- We can type **!** followed by a history number to re-execute that command:

```
linux2 [4]# history
1 21:47 gcc hello.c
2 21:47 a.out
linux2 [5]# !1
gcc hello.c
```

- We can also type **!** followed by the first character(s) of that command.
  - When there are multiple matches, the shell will always execute the most recently executed match

```
linux2 [7]# history
1 21:47 gcc hello.c
2 21:49 gcc round.c
3 21:49 history
linux2 [8]# !g
gcc round.c linux2
```

- We can also always execute the most recently executed command by issuing the command **!!** (bang bang).

PATH

- The PATH is nothing more than a list of directories in which to look for executable commands.
- Note that if the same command lives in more than one of these places, the first one in the path listing is the one that is used.
- To complicate this matter even more, the versions that are in these different directories may be different, as it the case with gcc (the C compiler) at the time of this writing.

whereis

- The **whereis** command locates all instances of a command that may be in your PATH.
  - There are many bin (binary executable) directories scattered across the system and throughout your PATH.
  - There may be multiple copies (or even worse - versions) of the same command.
- Example:

```
linux2 [23]# whereis bash
cp: /bin/bash /usr/local/bin/bash /usr/share/man/man1/
bash.1.gz
```

  - There are 2 instances of bash, one in "/bin/" and "/usr/local/bin/".
  - Often times you may also get hits for the manpage entries as well (which is the last one that we are seeing).
- There may be multiple versions of the same application in different places.
- Example:

```
linux2 [27]# /bin/bash –version
GNU bash, version 2.05.8(1)-release (i386-pc-linux-gnu)
Copyright 2000 Free Software Foundation
linux2 [28]# /usr/local/bin/bash –version
GNU bash, version 2.03.0(1)-release (i686-pc-linux-gnu)
Copyright 1998 Free Software Foundation
```

which

- **which** tells which instance of a command is being executed when that command is run.
- Typically, the specific instance that is being executed is the one that is first in your PATH.
- However, there are some instances where **which** will return something other than a path name.
  - If commands are aliased or if the command is a built-in shell command, **which** may report that as well.
- Example:

```
linux2 [41]# which gcc
 /usr/local/bin/gcc
linux2 [42]# which mem
 mem: aliased to quota -v
linux2 [43]# which time
 time: shell built-in command
```

UNIX Tools: grep

- The **grep** tool searches for lines matching a specific pattern.
- The general form of the **grep** command is:

  grep pattern files

- Example:

```
linux2 [77]# grep “Jon" *.c
 projaux.c: * Created by: Jon D. Smith
 projaux.c: * Last Modified by: Jon Smith
```

- There are also many flags that you can pass into **grep** (for a complete list, see the man pages).

  -i : searches for the given pattern insensitive to case (matches both uppercase and lowercase)

  -n: displays the numbers of the lines that match the target pattern

  --recursive: searches from this working directory downward

grep Example

- Example:

```
linux2 [78]# grep "Jon" *.c -i
 proj.c: * Created by: JON D. SMITH
 projaux.c: * Created by: Jonathan D. Smith
 projaux.c: * Last Modified by: Jonathan D. Smith

linux2 [79]# grep “Jon" *.c -n
 projaux.c:3: * Created by: Jon D. Smith
 projaux.c:6: * Last Modified by: Jonathan D. Smith

linux2 [85]# grep “Jon" . --recursive
 ./avg.c: * Created by: Jonathan D. Smith
 ./coredump.c: * Created by: Jon D. Smith
 ./dir1/hello.c: * Created by: Jonathan D. Smith
 ./dir3/projaux.c: * Created by: Jonathan Smith
 ./dir3/projaux.c: * Last Modified by: Jonathan Smith
 ./dir3/projaux.h: * Created by: Jon Smith
 ./dir3/projaux.h: * Last Modified by: Jon Smith
 ./foo.c: * Created by: Jonathan D. Smith
```

UNIX Tools: find

- The **find** tool searches for files in a directory hierarchy.
- The general form for the find command is:

  find path conditions

- There are many conditions that you can check for (for a complete list see the man pages).

  -name: look for files names that match a given pattern

  -iname: does case-insensitive name matching

- Example:

```
linux2 [95]# find . -name image.jpg ./
  dir1/subdir1/subsubdir/image.jpg
linux2 [96]# find . -name image
```

IO Redirection: stdin, stdout, and stderr

- If you are familiar with C programming, you will remember scanf and printf.
- For now it is sufficient to say:
  - scanf reads from stdin (the keyboard)
  - printf writes out to stdout (the screen via the console)
- UNIX systems provide a facility of redirection that allow us to read from and write to places other than these defaults of the keyboard and the screen.

> Redirection of stdout (overwrite)

- **>** allows us to send the output from stdout to somewhere other than the screen.
- Example: Redirecting the linux date command to a file:

```
linux3-(6:28pm): date > output
linux3-(6:28pm): cat output
Sun Oct 6 18:28:32 EDT 2002
```

- If we redirect stdout using a single > it will overwrite the contents of the file, erasing all previous contents.

```
linux3-(6:28pm): date > output
linux3-(6:28pm): cat output
Sun Oct 6 18:28:44 EDT 2002
```

>> Redirection of stdout (append)

- Redirecting with **>>** appends to a file.

```
linux3-(6:30pm): date >> output
linux3-(6:30pm): cat output
Sun Oct 6 18:30:07 EDT 2002
Sun Oct 6 18:30:19 EDT 2002
```

- Redirection of stdout is helpful if the amount of information printed to the screen is more than the screen can hold.
  - You can redirect the output to a file and then view it using less, more, cat, or the text editor of your choice.
- Redirection of stdout is also useful to save the output of a program.

< Redirection of stdin

- We can redirect stdin from a file.

```
linux3-(6:40pm): gcc -Wall -ansi avg.c -o avg
linux3-(6:40pm): avg
Enter the first integer: 1
Enter the second integer: 2
Average is: 1.500000
```

- Rather than the user typing in the values, lets get them from a file. We will run the program once redirecting 1.dat as the input and again using 2.dat as the input...

```
linux3-(6:40pm): cat 1.dat
1 2
linux3-(6:40pm): avg < 1.dat
Enter the first integer: Enter the second integer: Average is:
1.500000
linux3-(6:40pm): cat 2.dat
1

  2
linux3-(6:41pm): avg < 2.dat
Enter the first integer: Enter the second integer: Average is:
1.500000
```

- scanf is smart enough to skip white space, whether it be a space or newlines. Nothing fancy is needed to handle whitespace.
- Note: that the numbers being read in are not echoed to the screen.

Combining < and >

- We can combine different types of redirection with a single command.
- Example: the program will get input from the file called 1.dat, and redirect all of the program's output to a file called output.

```
linux3-(6:41pm): avg < 1.dat > output
linux3-(6:41pm): cat output
Enter the first integer: Enter the
second integer: Average is: 1.500000
```

>& Redirecting stderr

- Lets examine the following output from the gcc compiler...

```
linux3-(6:42pm): gcc -Wall -ansi avg.c
avg.c:24: unterminated string or character constant
avg.c:19: possible real start of unterminated constant
```

- This is a simple example where gcc finds 2 errors when it tries to compile a buggy version of avg.c. But what if we have so many errors that they all scroll off of the top of the screen and we are unable to see them all? Sound like a job for redirection of stdout to a file...

```
linux3-(6:42pm): gcc -Wall -ansi avg.c > output
avg.c:24: unterminated string or character constant
avg.c:19: possible real start of unterminated constant
linux3-(6:42pm): cat output
linux3-(6:42pm):
```

- What happened? I told the compiler to direct the errors to a file, but they were printed to the screen and not the file like I told it.
- Some programs print to the screen without using stdout. Often times errors and warnings are printed to another output buffer called stderr.
- There are some cases where we may wish to redirect stderr to a file and look at them. Such as when we need to examine them but there are too many. Well to redirect the output we use the > followed by an & sign to tell it to redirect stderr as well...

```
linux3-(6:42pm): gcc -Wall -ansi avg.c >& output
linux3-(6:42pm): cat output
avg.c:24: unterminated string or character constant
avg.c:19: possible real start of unterminated constant
```

UNIX Tools: tar

- The **tar** command is used for creating an archive of a directory hierarchy.
- **tar** archives are a handy way of sending a bunch of files (or a program distribution) across the network or posting them on the internet.
  - Begin by creating a tar archive of the files.
  - Transmit that tar archive over the network or post it online.
  - Untar the files where you want them.
- Usage:
  - Creating a tar archive:

    tar –cvf <archive_name>.tar <files>
  - Viewing the contents of an archive:

    tar –tvf <archive_name>.tar
  - Extracting a tar archive to the current directory:

    tar –xvf <archive_name>.tar

tar Examples

- Create a tar archive of your home directory and place it in your working directory:

  tar –cvf myhome.tar home/
- View the contents of the tar archive:

  tar –tvf myhome.tar
- Extract the tar archive to your current working directory:

  tar –xvf myhome.tar

- Creating, viewing, and extracting a tar archive

```
linux3[6]% tar -cf archive.tar file*.txt
file1.txt
file2.txt
linux3[7]% tar -tvf archive.tar
-rw-r--r-- eeaton/rpc 4298 2005-09-26 10:19:02 file1.txt
-rw-r--r-- eeaton/rpc 4441 2005-09-26 10:20:12 file2.txt
linux3[8]% mkdir temp
linux3[9]% cd temp
linux3[10]% tar -xvf ../archive.tar
file1.txt
file2.txt
linux3[11]% ls
file1.txt  file2.txt
```

Advanced Command Chaining

- We can use combinations of parenthesis, semicolons, i/o redirection, and pipes to create powerful commands:
- Example: Copying a set of files while preserving timestamps:

  (cd ~/courses; tar -cvf - CMSC121) |

  (cd ~/../pub/courses/; tar -xvf -)
- Note that this is a fancy way of doing what **cp** with the **-r** and **--preserve** flags already does.

Running Background Jobs

- Jobs (a.k.a. commands) can be told to be run in the background by issuing the command followed by the ampersand symbol (&).
- This starts the execution of the command, but allows immediate input from the terminal for other purposes (such as compilation).
- When a command is issued with the ampersand, the shell prints out some information about the job.
    - First is the job number which is shown in square brackets, followed by the process ID number.
    - You may also see a line of information about that job.
    - Lastly the system prompt will be redisplayed for your next command.
    - When the job is completed, the shell will tell you that it is done. This typically does not happen immediately, but when the shell next gets to draw the prompt (after the next command).

```
linux1 [21]# emacs foo.c &
[1] 16819
linux1 [22]# gcc foo.c
foo.c: In function `main':
foo.c:19: parse error before `return‘
linux1 [23]# gcc foo.c
linux1 [24]#
[1] Done   emacs foo.c
```

Managing Processes

- **ps** : lists the processes running on the machine.
  - ps -u username lists only your processes.
  - ps -a : lists all processes running on the machine.
  - The PID column of the listing, provides the information required by the kill command.
- **top** : a more detailed method of observing processes.
- **nice** : runs a process with a lower priority.
  - ALWAYS use this if you are running a process that will take a long while (hours or days).
  - **nice** doesn't slow down your process much, but allows the interactive aspects of the computer (GUI, etc) to take priority. This makes system administrators and other users happy.
- **nohup** : keeps a process running even after you log out

Killing Processes

- **^c** : terminates a foreground process

```
linux2 [18]# xcalc
(I pressed ^C here)
linux2 [19]#
```

- **kill** : terminates a process

```
linux2 [29]# xcalc &
[1] 27131 linux2
```

  - **kill *process_id*** : sends a terminate signal to the process specified by the *process_id* (PID).

```
linux2 [21]# kill 26118
[1]    + Terminated xcalc
```

  - **kill %*job_num*** : sends a terminate signal to the specified job

```
linux2 [30]# kill %1
[1] Terminated xcalc
```

  - In cases where the terminate signal does not work, the command **kill *-9 process_id*** sends a kill signal to the process.

xkill - Killing Graphical (X window) Processes

- The **xkill** command allows you to select a window and will kill off the process associated with that window. Use this when a graphical program stops responding.   The **xkill** command accepts mouse input to determine which process to kill:

```
linux2 [39]# xkill
Select the window whose client you wish to kill with
  button 1....
xkill:       killing creator of resource 0x1e00015
X connection to linux2.gl.umbc.edu:10.0 broken (explicit
  kill or server shutdown).
[1]   + Exit 1           xcalc
linux2 [40]#
```

- Note: xkill terminates a process and not a window. What does this mean? Well one process may be associated with multiple windows of the same application. So if Netscape hangs in a window and you xkill that window, everything associated with Netscape will be killed off, as they all originate from the same PID.