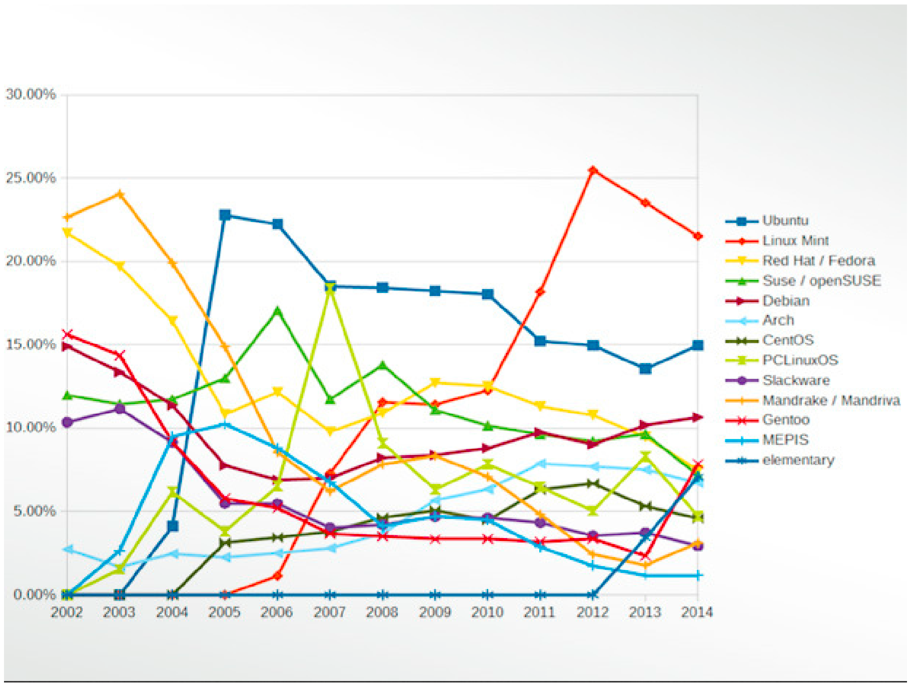
Unix Command-Line
CS3670

Estimated completion time: 45 minutes

This tutorial is meant to help the novice Unix user become more familiar with some of the commonly used commands. Unix is still (and maybe always will be) a very strong command-line environment, although Apple has successfully hidden Unix underneath the Mac OS X graphical user interface (GUI). However, this tutorial will emphasize the command-line, even though many things can be done via some window. The windowing environment can be so different from one "flavor" of Unix to another (and sometimes within the same flavor of Unix) that it is best to stick with those commands that will probably work on all flavors.

I. Historical Background

There are many flavors of the Unix operating system, Linux being just one of several free versions. Two examples of other free versions of Unix include FreeBSD and OpenBSD, which for one reason or another have not captured the public attention like Linux has. CentOS is a *distribution* of Linux, which is the Linux operating system packaged with a lot of additional free and homegrown software to make it useable. There are other Linux distributions, such as Fedora, Ubuntu, Caldera, SuSe and many more. The following graph[1] shows many distributions and their "popularity" over time.



1 http://core0.staticworld.net/images/article/2015/12/slide-08-100633995-orig.jpg [Accessed Dec. 30, 2015]

There are many commercial versions of Unix, such as Sun Solaris (now Oracle Solaris), Hewlett Packard HPUX, IBM AIX, etc. However, the line between commercial and free is blurring, after Sun provided a free download of its OS, and also because the Mac OS is based mostly on FreeBSD and is now free. If you prefer, IBM will also sell you computer hardware with Linux installed instead of their AIX.

Whether free or commercial, the different flavors of Unix are mostly based on the original Unix operating system created in 1971 by a small team at AT&T. However, because AT&T was prohibited from selling operating systems (because AT&T was judged to be an illegal monopoly), they *licensed* the code to various organizations. As a result, some of these organizations modified the operating system to meet their specific needs. Most notable are the modifications made at the University of California at Berkeley (The “B” in BSD). At this point the Unix world was split into two branches: those based on the BSD work, and those based on what was referred to as System III. SunOS, for example, was based on BSD. [Linux is not based on any AT&T code, but was written from scratch instead, and is therefore considered a Unix clone.] Novell obtained the rights to the source code from AT&T in the early 1990s. Even Microsoft at one time marketed a popular brand of Unix they called Xenix.

Eventually, an effort was made to unify the two Unix branches, which became known as System V, release 4 (SVR4), but there were still some subtle differences. Since then, the various flavors have once again drifted apart. These differences, however, are not typically noticeable to the average user of a Unix system. A user on one Unix system can almost transparently move to another, as far as the command-line is concerned (though the windowing environment is another thing). But it can be difficult for a system administrator to move between the two different kinds of Unix.

The remaining sections provide a hands-on tutorial of the Unix command line and scripts. Note that this is just an introduction.

Warning: In the commands given in this instruction, the difference between the number one (‘1’) and the lower-case letter ell (‘l’) can be very slight, if anything. The context for the commands should tell you what it ought to be.

II. Getting Started

Boot your Linux system or VM, log in, and then open a terminal window and start the lab:

```
cd labtainer/labtainer-student
start.py nix-commands
```

III.Basic Commands

Commands in Unix are typed into what is called a *shell*. Each command takes some number of arguments, which are known as “command-line arguments”, “arguments” or “options”. (In Microsoft lingo, these options are referred to as “switches”). **The shell does not execute commands until the Enter key is pressed**. Any errors are reported in the shell window.

When the lab starts, you will be presented with a black virtual terminal, and the current directory is your *home* directory. [Windows uses the name *folder* instead of *directory*.] For the root user, home is at /root on Linux. For regular users, it is often found at /home/*username*, but it could be anywhere. Use the ***pwd*** command (present working directory) to see what your home directory is:

```
pwd
```

List the contents of your home directory by using the ***ls*** (list) command:

```
ls
```

dir is the Windows equivalent of the ***ls*** command. Unlike Windows command-line shells, Unix commands are case-sensitive, meaning that you cannot enter ***LS*** and expect the shell to equate it to the lower-case ***ls***.

Without any arguments, ***ls*** will display the contents of the current directory without much special formatting (except that it is sorted).

In addition to the files you can see, there are other files that are "hidden". In Unix, hidden files are files that you don't necessarily want to see all the time. Therefore, unless specifically requested, they are not shown with a directory listing. Any file or directory that begins with a '.' is hidden, and is referred to as a *dot file*.

List all objects, even the hidden files and directories, by also using the ***–a*** (all) option, as shown below. (Note that there must always be a space between the command and any arguments being passed to it).

```
ls –a
```

The home directory contains many user-level configuration files for modifying how your environment behaves, and they are almost always dot files or dot directories. The ".login" file (if it exists) controls things you always want done when you log in. The ".bashrc" file is used to configure the shell. If these do not exist, then system-wide configuration files are typically used.

Get a bigger picture of the contents of the current directory (such as owner and size) by also using the ***-l*** (long) option, as shown below:

```
ls –al
```

The ***ls –l*** command is used so often that there is often a shortcut for it (***ll***). Try using the shortcut for ***ls –l*** as shown below:

```
ll
```

Without it being the current directory (i.e., the present working directory), the contents of another directory can be listed. List the contents of the bin (binary) directory (where many user-level commands are stored) by doing the following:

```
ls /usr/bin
```

Users may also create new directories. Create a directory from inside your home directory by using the ***mkdir*** command:

```
mkdir temp
ll
```

Change your current directory to the new directory using the ***cd*** (change directory) command:

```
cd temp
pwd
```

List all the contents of this new directory:

```
ll -a
```

Even though a directory is brand new, it is not exactly empty. Every directory has at least two entries: two directories named "." and ".." (called *dot* and *dotdot*). The dot directory is a shortcut for the current directory, while dotdot is a shortcut to the parent directory. Windows has borrowed this philosophy too. (One thing hackers do to hide their own files is to create a directory with three dots, which might be easily overlooked).

Use the dotdot directory to list the contents of the parent directory:

```
ll ..
```

Use the dotdot directory to change your working directory to the parent directory of temp, as shown below:

```
cd ..
pwd
```

(Note that a nice feature of the ***cd*** command is that if it is entered with no arguments, it will always take you back to your home directory).

Files or directories can be moved or simply renamed by using the ***mv*** (move) command. Rename the temp directory to temp2 by doing the following:

```
mv temp temp2
ll
```

Copying files is done with the ***cp*** (copy) command. Copy one of your hidden files into the temp2 directory:

```
cp .bashrc temp2
ll -a temp2
```

You can, of course, rename the file while copying. Copy the hidden file again, renaming it in the process:

```
cp .bashrc temp2/.bash
ll –a temp2
```

No space!

Try to delete the temp2 directory using the ***rmdir*** (remove directory) command:

```
rmdir temp2
```

It should have failed because files still exist in that directory. Delete the files in the temp2 directory by using the ***rm*** (remove) command:

```
rm temp2/.bas*
ll –a temp2
```

Notice the use of the wild card "*" symbol in the ***rm*** command. The command was interpreted to mean: delete all the files starting with ".bas".

Now remove the directory:

```
rmdir temp2
ll
```

To display the contents of a text file to the screen, the ***cat*** (concatenate) command can be used. Display the contents of the password file:

```
cat /etc/passwd
```

Display text on the screen by using the ***echo*** command:

```
echo "hello world"
```

The echo command may seem meaningless, but it comes in handy, such as when writing scripts (to be covered later).

IV. Pipes and Redirection

Outputs from commands are almost always directed to the shell window, immediately following the line where the command was typed. If the output turns out to be quite long, it is useful to have a way to view it before it speeds by. One of the commands from the last section scrolled by too fast to see it all:

```
ls /usr/bin
```

One way to slow it down is to “pipe” it into another command. The pipe symbol is the “|” character (shift “\”). (Microsoft also borrowed this concept).

Pipe the previous command through the ***more*** command (as shown below), which will display one screen at a time:

```
ls /usr/bin |  more
```

To see the next page of output, press the space bar. To see one line at a time, press Enter. To quit at any time, press ‘q’.

This is traditionally the way complicated commands are done in Unix: stringing successive commands together with pipes. The philosophy has been to keep commands simple.

Another way to deal with a lot of output is through redirection. The output of a command can be redirected to a file using the “>” redirection symbol. Microsoft also borrowed this concept.

Redirect the directory listing into a file, as shown in the following command:

```
ls /usr/bin  >  listing
ll
```

This file can now be viewed using ***more*** or ***cat*** or your favorite editor.

If the file named “listing” already exists, it will be overwritten with a redirection. It is possible, however, to append the contents of an existing file by using “>>”:

```
echo "testing"  >> listing
cat listing
```

The displayed output should first list the contents of the /usr/bin directory, followed by the single word “testing”.

V. Help

The command-line method for getting help is with the ***man*** command (short for “manual”). Enter the following to get more information about the ***mkdir*** command:

```
man mkdir
```

The output is piped through the ***more*** command. (Press ‘q’ to quit). You can even get information about the man command itself by entering the following:

```
man man
```

VI. Searching

The command for searching the contents of a file for a given string is ***grep*** (global regular expression pattern). (In Windows, this command is called ***find***).

Search for the string “student” in all the files in the /etc directory:

```
grep  student  /etc/*
```

The first argument of the ***grep*** command is the string to be searched for (student), while the last argument is the file, or files, to look in. The /etc directory is where many configuration files are kept. The output of ***grep*** consists of the file name(s) the string was found in, and sometimes the line in the file where it was found. If the search string has spaces, then it needs to be quoted.

There were a lot of errors reported from the previous ***grep*** command, so you can tell ***grep*** to be silent about those errors in order to have a cleaner output:

```
grep -s student  /etc/*
```

The Unix ***find*** command is like the Swiss army knife of Unix commands, and is also very useful for finding things. Its syntax is somewhat complicated, and it can do much more than can be shown in this tutorial.

One basic use of ***find*** is to locate a file with a known name. Use ***find*** to locate a file called “hosts”, using the following command:

```
find /etc –name hosts -print
```

find recursively checks all the directories from the starting point down. In the above example, it looked everywhere at and below the “/etc” directory. The ”-name hosts” tells ***find*** to search for all files/directories whose name is “hosts”. The “-print” tells ***find*** what to do when it finds the requested file(s). In this case, it prints out the path where it is located.

There were a lot of permission problems with the last execution of the ***find*** command. Gain root privilege by entering the this command:

```
sudo su
```

The “student” account on this computer is a member of the “sudo” group, allowing you to work with root privileges. (Note that for convenience, most Labtainer labs are configured such that the “sudo” command does not require you to provide a password. Typical systems require use of a password when using sudo.) You can usually tell when you are executing with root privileges in the shell, because the conventional prompt is the ‘#’ character. All other processes and windows are still executing on behalf of the regular user.

Re-execute the last ***find*** command with root privileges:

```
find /etc –name hosts -print
```

Wild card characters can be used, but they must be quoted. For example, to find all the files ending with “.h”, the following could be used (**typed on one line**):

```
find /usr/include -name
"*.h" -print
```

An even more basic use of ***find*** is to display the path of **every** file it sees. In the following example, ***find*** is told to look in the entire hierarchy starting with “/usr/local”. When it finds a file, it prints out the location. In other words, it will display all the file and directory names in the hierarchy.

```
find /usr/local –print
```

Return to the privilege of a regular user:

```
exit
```

You should still have a shell window open, but the prompt should have returned to what it was before ***su*** was entered ($).

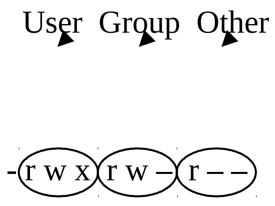
VII. Access Control

Linux has the traditional simple Unix form of Discretionary Access Control (DAC) known as *permission bits*. Every file has an owner, and every file belongs to a group. The owner controls access to a file; the owner of a file assigns his/her own permissions, permissions for members of the owning group, and others (everyone else). These three entities (referred to as user, group and other) may be given one or more of the following permissions to files or directories: read, write, and execute.

Display the contents of your home directory:

```
cd
ll -a
```

The permissions are displayed on the far left-hand side of the output of each object. The permissions are broken down by user (i.e., owner), group and others, as follows:



Each section is divided into one of three permissions: read (r), write (w), and execute (x). If a permission has been granted then it appears, otherwise a '-' indicates the permission is not granted. In the above example, the *user* (or file owner) has all three permissions, while the *group* has read and write permissions, and *other* has only read permission.

For the curious, the leading '-' signifies a file, while a 'd' indicates a directory.

To change permissions on a file or directory, the ***chmod*** (change mode) command is used. There are multiple ways to use the interface, but these examples will stick to what may be the easiest to remember and understand. (The GUI is the easiest of all.)

To indicate that the permissions of the user need to be changed, use 'u'. For the group, use 'g', and for other use 'o'. For read, write and execute, use 'r', 'w' and 'x', respectively. For example, change the permissions on your .bashrc file so that everyone can write to it:

```
ll .bashrc
chmod  o+w  .bashrc
ll .bashrc
```

The "o+w" means "Add the write permission to other". Now remove the permission:

```
chmod  o-w  .bashrc
ll .bashrc
```

Multiple changes can be made at one time. Do the following to make multiple changes at once:

```
chmod  ugo+rwx  .bashrc
ll .bashrc
```

where user, group and other are all given read write and execute permissions to the file.

To change the permissions so the group and other only have read access, do the following:

```
chmod go=r .bashrc
ll .bashrc
```

VIII. Process Management

To display a list of the processes currently executing, the p***s*** (process status) command is used. Enter the following:

```
ps
```

Without any arguments, the default output is a list of the processes associated with your command-line session. Among other things, it shows the unique process ID (PID), the amount of CPU time used, and the program name.

To display all the processes currently running (even those not associated with your terminal), enter the following:

```
ps ax
```

If a process ever gets hung and will not die, the PID displayed for that process can be used to terminate it. **For example**, if a process with PID of 11076 needed to be terminated, the following is used:

```
kill –9 11076
```

If the process needing termination is running from the command-line, then maybe something as simple as CTRL-C will kill it.

To see who is currently logged into the system, the following is used:

```
who
```

Since you are the only one logged in, it is not very useful right now. But because Unix can support a lot of users simultaneously, it can be very helpful on a production system. For example, if a system administrator wants to reboot a system, it would be wise to see if everyone has logged out first.

IX. Editors

The traditional command-line editors in Unix are ***vi*** and ***emacs***. It is not advisable to try to learn the ***vi*** editor unless there is a professional need, such as programming or an expectation of administering systems. ***emacs*** works better for a novice but it consumes 500 MB of disk, and is not typically included in these labs. In the old days "Religious wars" were waged over which was better, though the advantage of ***emacs*** was somewhat weakened when windowing environments appeared.

For most novices, windowing editors will suffice because are easy to learn. However, windowing is not always available, and you sometimes may need to modify a file using a text based editor. Labs in this course will include the "leafpad" editor. This can be invoked by typing ***leafpad*** at the command prompt.

X. History

By default, most Unix shells keep track of the commands you have entered. Enter the following to see the commands you have entered as the student user:

```
history
```

There are several benefits of keeping track of your commands. One benefit is being able to reuse previous commands without retyping them. One way to do that is to use the up and down arrow keys. Once a command is found using this approach, the line can be modified before the Enter key is pressed to execute the command.

Use the up arrow key until the last ***grep*** command is displayed, then press Enter to re-execute it.

An even quicker way to run the last ***grep*** command is by using the '!' character as shown below:

```
   !grep
```

Of course, the more general use of '!' is to search for the last command that started with whatever follows "!", such as "`!ps`".

The more advanced uses of these features will not be covered here.

XI. Shell Scripts

One way of automating repetitive command-line tasks is to put those commands into a file, make that file executable (using the ***chmod*** command), and execute it when needed. Such files are called *shell scripts* or just *scripts*. Microsoft calls them *batch files*. Very complicated tasks can be performed with shell scripts.

For this simple example you will be using the diagnostic ***ping*** command. ***Ping*** is mostly used to determine whether a remote system is responding to low-level network activity. Issue the following ***ping*** command:

```
   ping  google.com
```

The above ***ping*** command determines whether the machine with the given IP address is working. When you have seen enough, enter ***Ctrl–C*** to kill the pinging.

Assume a simple example: every morning the first thing you do is verify that your most important servers are accessible. You might perform the following commands (**but not now**):

```
ping  mail
ping  payroll
ping  printer
ping  router
```

It would be nice if one command could be entered that would perform all your pings at once. But before we do something like that, let's alter the ***ping*** commands to be somewhat friendly for a script. **Enter the following**:

```
ping  -c 1  -w 1
google.com
```

Instead of pinging continuously until interrupted by the user, the above command will ping only once (-c 1), and will only wait one second (-w 1) for the reply. That is an improvement, but you don't really want to see all that output. So try the following:

```
ping  -c 1  -w 1
google.com > /dev/null
```

You redirected all the output into a black hole from which nothing returns. So we have removed all the output, but now we don't know if the ping was successful. So, try the following addition:

```
ping -c 1  -w 1
google.com > /dev/null  &&
echo Up
```

That looks more like it. If ***ping*** is successful, the command after "&&" is executed, otherwise it is not. In this case "Up" is displayed on the screen if the ping is successful.

Now you can write a script. Start up an editor, such as ***leafpad***, and enter the following lines:

```
echo
echo "Trying Google"
ping -c 1  -w 1
google.com > /dev/null &&
echo Up

echo
echo "Trying Bing"
ping -c 1  -w 1  bing.com
> /dev/null && echo Up

echo
echo "Trying NPS"
ping -c 1  -w 1  nps.edu >
/dev/null && echo Up
```

Save the file in your home directory with the name of "pinger", and then exit the editor.
Make the file executable and try it out by doing the following:

```
ll
chmod u+x pinger
ll
./pinger
```

Notice how you had to enter a "./" before `pinger`? Why is that? The shell is configured to look in given locations for the commands entered by the user. Rarely is the command in the current directory, so if you want to execute something outside the usual places, you must be explicit. The "./" tells the shell to look in the current directory. If you only enter `pinger` then you'll get a "command not found" error.

That is the end of this tutorial. You will learn more about Unix commands in other lab assignments.

XII. Finish Up

After finishing the lab, go to the terminal on your Linux system that was used to start the lab and type:

```
      stop.py nix-commands
```