Intermediate UNIX Command Line

Mississippi Center for Supercomputing Research
Ben Pharr bnpharr@olemiss.edu

Connecting to a remote server

Most UNIX and Linux systems that are operating as servers run an SSH daemon. Users with valid accounts on that system can then use an SSH client to connect to the server:

- PuTTY: http://www.chiark.greenend.org.uk/~sgtatham/putty/
- Mac OS X and most Linux distributions come with a command-line SSH client (`ssh`) pre-installed.

Today we'll be using the PuTTY to connect to hpcwoods, MCSR's gateway server.

1) Start by opening the PuTTY application.
2) In the "Host Name" field, enter "hpcwoods.olemiss.edu".
3) Click "Open".
4) Depending on whether or not your computer has ever connected to hpcwoods, you may get a dialog asking if you "want to save the new host key to the local database?" Click "Yes". This is so the client can confirm the identity of the server on subsequent connections.
5) If you are an MCSR user, you may use your username in the "User Name" field. Otherwise, use the temporary username given to you by the instructor.
6) Enter your password, or the one given to you by the instructor, in the "Password" field.
7) If successful, you should see a command prompt. Something like:
   ```
   username@hpcwoods:~>
   ```

Environmental Variables

Environmental variables allow a user to customize their command line and its operation to their liking. All the commands in this workshop assume that we're using the bash shell. This can be confirmed by typing:

`echo $SHELL`	Displays the contents of the SHELL environmental variable.

General commands

`env`	Displays all environmental variables.
`echo $USER`	Displays the current username.
`echo $HOME`	Displays the current user's home directory.
`export MYVAR="Testing…"`	Set a new variable.
`echo $MYVAR`	Display the contents of the variable.

Customize your shell prompt

`echo $PS1`	See the current prompt settings.
`export PS1=”\u@\h:\w$ “`	Slight modification to the prompt.
`export PS1="\d \t\n\u@\h:\w(\!)$ "`	A deluxe prompt.
`man bash`	Search for “PROMPTING” for all options.

It is also possible to set certain parts of the prompt to particular colors. See Google for details.

Another environmental variable that needs to be changed a lot is the PATH. The PATH variable determines where the shell looks for commands. For instance, when you type `ls`, the shell looks through each directory in PATH until it finds an executable named `ls`.

`echo $PATH`	Displays the current PATH.
`which ls`	Find out which executable is being used when you type `ls`.
`export PATH=/opt/bin:$PATH`	Prepend a new directory to the PATH.

Shell startup files

The changes that we have been making to environmental variables are not permanent. They go away when we logout. To make them permanent, we need to add them to the shell startup files.

When a user logs in, their shell first reads /etc/profile, allowing system-wide settings to be set. Then it reads ~/.profile or ~/.bash_profile for user settings. Interactive non-login shells (su or xterm, for example) also read ~/.bashrc.

Changes can be made to either file. However, it is not uncommon for the ~/.profile to source, or import, the ~/.bashrc file, meaning it might be preferable to modify the ~/.bashrc file, so the changes would apply in either case.

These files (like all configuration files on UNIX systems) are plain text, meaning they can be modified with your favorite text editor, like nano, vi, or emacs.

Accessing files in other directories

Using files in the current directory is pretty simple, but you can also use files in other directories by using absolute and relative references. This can be demonstrated by copying files from one directory to another. First, let’s create a directory and a file to play with:

`mkdir dir1`	Creates a new directory
`touch dir1/testfile`	Creates a new, empty file inside the directory

The reference `dir1/testfile` is a very simple example of a relative reference. `dir1` is relative to the current directory.

Now we want to create another directory and copy testfile to it. We could make a copy of the entire directory, but that would defeat the purpose of the exercise.

`mkdir dir2`	Create another directory
`cp dir1/testfile dir2`	Copy the file in dir1 to dir2

This is just one way to make a copy of a file in another directory. There are several others, and the "best" method depends on the circumstances.

You can also use an absolute reference. In this example, we'll use the tilde to represent your home directory. The system knows where your home directory is on the filesystem.

```
cp ~/dir1/testfile dir2
```

Another way to use relative references is to use "." to refer to the current directory and ".." to refer to the parent directory of the current directory. For instance, you can copy the file like this:

`cd dir2`	Make `dir2` the current directory
`cp ../dir1/testfile .`	

These "files," as well as other "hidden" "dot" files can be seen by running

```
ls –a
```

The `–a` switch tells `ls` to show all files.

A common use of the `..` file is to go "up" or "back" one directory:

```
cd ..
```

File and directory permissions

A file's permissions determine which users can do what to a particular file. To see the permissions for all files in a directory, type:

```
ls –la
```

The output for each file looks something like this:

```
-rw-r--r--   1 bnp  users         1028 2011-10-14 15:14 .profile
```

The first column contains the permissions. The first dash tell us this is a regular file. If it were a directory, it would have a `d` instead. The next three symbols (`rw–`) represent the permissions for the owner of the file, which is `bnp` in this case.

The next three symbols (`r––`) represent the permissions for the group that owns this file, which is `users` in this case. The final three symbols in the permissions column (`r––`) represent the permission for "world," or everyone else on the system that isn't the owner of the file or a member of the group that owns the file.

In each case, the three symbols tell us whether or not the file can be read (`r`), written to (`w`), or executed (`x`) by that user or group of users. If the letter appears, then that permission in enabled, if not, a dash appears in its place. In the above example, the owner of the file can read and write the file, but the group and world can only read it.

Sometimes you might also hear someone say that a particular "bit is set." For instance, in the above example, it could be said that "the write bit is set for the user." This is because each permission is represented by a binary bit that can be turned on or off.

We can set the permissions of a file using the `chmod` command. It can be done symbolically or numerically. Some examples:

`chmod g+w .profile`	Give the group write privileges.
`chmod g-w .profile`	Revoke the group's write privileges.
`chmod 640 .profile`	Give owner read and write privileges, group read privileges, and no privileges to world.

Text editors

If you bothered to come to this workshop, chances are you spend, or plan to spend, a lot of time logged in to a UNIX system. If that's the case, you'll want to learn to use a text editor other than `nano`.

`vi` is the classic UNIX text editor. It will be on every UNIX system you ever login to, unlike `nano` and `emacs`. However, its commands are a little cryptic and the learning curve is fairly steep.

`Emacs` is another common UNIX text editor. Its learning curve isn't quite as steep as `vi`'s, but it's not exactly user-friendly either.

I personally use `emacs`, though I can use `vi` when the situation calls for it. If I had to pick just one for you to learn, I'd probably have to pick `vi`, because it's everywhere. Either `vi` or `emacs` will give you more text editing power than `nano`. The time you spend learning either one will be well worth it.

Scripts

Often you will need to run a sequence of commands (or a single long command) more than once. You can create a script that contains these commands.

Let's say we want to see the largest files in the current directory. We can do this with the command:

```
ls -sSh | head
```

but we don't want to have to remember that and type it in each time. We can create a script to make it easier on ourselves.

Open a file named `big_files` using your favorite text editor. Add these two lines:

```
#!/bin/bash
ls -sSh | head
```

then save and exit. Before we can execute the script, we need to set the file's execute bits. Running:

```
chmod +x big_files
```

will give everyone on the system permission to execute our script. This is fine in this case.

To execute the script, type:

```
      ./big_files
```

If we want to be able to execute this script from any directory without typing the whole path, we have to put it in our `PATH` environmental variable. It's traditional for each user to have a bin directory in their home directory where they can put programs and scripts they want to execute.

To create this directory, copy the script to your bin directory, and add the bin directory to your `PATH`, follow these steps (from your home directory):

```
   1) mkdir bin
   2) cp big_files bin
   3) export PATH=~/bin:$PATH
```

You can now execute the big_files script (from anywhere on the system) by typing

```
      big_files
```

However, if you want this change to your `PATH` to be permanent, you'll need to put the command to change it in one of your shell startup files.

Process management

When a program is executed, a process is created. Each process is assigned a process ID (PID).

The `top` utility can be used to see the system's processes, sorted by CPU utilization, as well as the general status of the system. At the top, it shows system uptime, number of users logged in, load average (a measure of how busy the system is), the number of processes in various states, and system memory usage.

Below that, processes are shown, one per row. For each process, it lists the PID, owner of the process, priority, memory usage information, CPU and memory utilization, time the process has been running, and the name of the program. Pressing `?` allows you to change how processes are shown.

To see a list of all processes, type:

```
      ps -ef
```

The list can be filtered using the `grep` command. For instance, to get a list of all `ssh` processes, type:

```
      ps -ef | grep ssh
```

To get a list of all processes not owned by root (the administrative user), use `grep`'s `-v` (invert) option:

```
      ps -ef | grep -v root
```

Under normal conditions, processes do their job and go away, not needing any attention from us. But occasionally they get hung up for one reason or another. If the process is still tied to your terminal, you can attempt to kill it by pressing Control-c. If it is not tied to your terminal, or if that doesn't work, you'll need to find its PID using top or ps and kill it using the kill command:

```
      kill pid
```

usually works, but when it doesn't

```
      kill -9 pid
```

usually does. If that doesn't work you need to reboot the system.

Searching files

You can search the contents of a text file by using the `grep` utility. To search through `/etc/passwd` for the phrase "bnp", run:

```
grep bnp /etc/passwd
```

To search through a directory of files (/etc) for files containing the phrase "bnp", use the recursive option to grep:

```
grep –r bnp /etc
```

The will almost certainly result in several errors, because we don't have permission to read all the files in /etc. To get rid of these errors, we can pipe standard error to /dev/null, leaving just the useful output on standard error.

```
grep –r bnp /etc 2> /dev/null
```

By default, `grep` does a case sensitive search. To make it case insensitive, you can add the `–i` switch. `grep` has lots of switches, which you can see by reading it's man page.

Uploading and Downloading Files

Many SSH clients have a feature that allow you to drag and drop files to and from your computer to the remote server. (In SSH Secure Shell Client, click on the yellow folder with blue dots in the toolbar.) There is also a standalone application called FileZilla that will allow you to do this. It is available for all platforms.

There is also a command line utility called scp (Secure Copy) that allows you to transfer files to and from hpcwoods. For instance, if user `bnp` wanted to transfer a file called file.txt to his home directory on hpcwoods, he could run:

```
scp file.txt bnp@hpcwoods.olemiss.edu:~
```

It can also be used in the opposite direction to download files from hpcwoods:

```
scp bnp@hpcwoods.olemiss.edu:~/file.txt .
```

hpcwoods, sequoia, and catalpa all share the same home directories and /ptmp directories, so you upload a file to hpcwoods, it will also be available on sequoia and catalpa.

Finding a file

Find files using a database that is updated once a day

`locate` *`filename`* (Not installed on hpcwoods.)

Find all files in your home directory with *.txt extension

```
find ~ –name “*.txt”
```

Show all files in the current directory that have been changed in the past day

```
find . -mtime -1
```

Compressing files

`tar –zcvf example.tar.gz directory` (Archive and compress)

`tar –zxvf example.tar.gz` (Decompress and unarchive)

Command Line Shortcuts and Tricks

`!find` (Run the `find` command with the same arguments as last time)

```
mkdir long_dir_name
```

`cd !$` (Reuse the last argument from the previous command)

Use `Control-r` to do a reverse, incremental search for previous commands

Other interesting commands

Lookup information about a user

`finger` *`username`*

Show disk usage in the current directory

```
du –h
```

Download a file from the web

`wget` *`url`*