Astronomy 480 || Linux III Tutorial

Pipes

Pipes are extremely handy! Try this to start things off:

`ps -augx | grep` *`yourusername`*

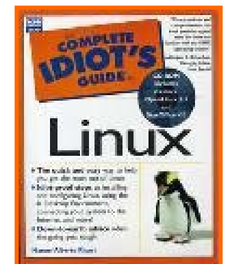
What did you get? Try it without the 'grep.' Type `top` and see what you get. Then, try it again, but pipe it with a 'grep' and your user name. What did 'top' give you? What is it telling you? Type 'q' to get out of 'top.'

I find that the `pipe` command is the most handy when I want to do an `ls -l` but I have so many files and directories that they go by way too fast:

```
ls -l | less
```

Working on Multiple Jobs Using A shell

You will be doing this. It is somehow rewarding to get a really huge computer job going (let's say one that will run for 2 or 3 days of a simulation of the formation of the solar system), and then go home and go to bed, knowing that you are **still** working! How productive is that?



Some command line jobs will take a while to complete. For example, the command

```
ls -R / > filelist.txt 2> error.txt
```

could easily take several minutes before it is done. If you have a lot of files and are listing several network drives, expect it to run for half an hour or so.

With Linux being multitasking, you'd probably want to do something else while the command runs. You can do this very easily by running the command concurrently or in the background. To execute the same command in the background, enter the following:

```
ls -R / ../../ > filelist.txt &
```

The key here is the ampersand at the end of the command line. This tells the shell to run the job in the background. After issuing the command, you'll see something like the following:

```
[alberto@digital alberto]$ ls -R / ../../ > files.txt &

jobs
[1] 872
[alberto@digital alberto]$
```

The job starts and your command line comes back. The number in brackets is the job number or process id assigned to your command. The 872 is the process id for the job in the system-wide process list. The process id for a job is useful in case you need to kill the job (see "Killing a Process: `kill`" in Chapter 23, "System Monitoring: Keeping an Eye on Your System"). Jobs are a little easier to remember than a process id of `872`. When referring to a job number, always precede it with a percent symbol. Both give you a way of controlling the command and to return it back to the front. When the job is completed, you'll see a line like this:

```
   [1]- Done                 ls -R / ../../ >filelist.txt
```

This means your job is done running. Note that programs which produce lots of output on the display should not be run in the background unless the output is redirected. Otherwise, your command prompt will be difficult to use, since output from the background command will continue to appear as you type!

Some programs will require interactive input, so every once in a while you'll see this:

```
   [1]+ Stopped (tty output)    telnet localhost
```

This means that the job needs input, and has paused to wait for keystrokes from you. To bring the job back to the front and enter the required keystrokes, use the command `fg`:

```
[alberto@digital alberto]$ fg %1
```

This will bring job number one (`%1`) back to the foreground. Note that there's a percent symbol before the job number.

To list all jobs started from inside the current shell, use the `jobs` command, like this:

```
[alberto@digital alberto]$ jobs
[1]+ Stopped (tty output)    telnet localhost
```

Suspending a Job

You can suspend a process that is running in the foreground by hitting **Ctrl+Z.** This will put the process to sleep:

```
[alberto@digital alberto]$ ls -R / ../../ > filelist.txt

[1]+ Stopped                 ls -R / ../../ > filelist.txt

[alberto@digital alberto]$ bg %1
[1]+ ls -R / ../../ > filelist.txt 2>errors.txt &
```

In this listing, I started a lengthy process in the foreground. I then put it to sleep (stopped it) by pressing **Ctrl+Z** and revived it in the background with a bg `%1`. `bg` does the opposite of the `fg` command.

Killing a Job

To kill the current foreground process, you can issue a Ctrl+C. This will terminate the process. To kill a job running in the background, you can use the `kill` command followed by its job number or its process id:

```
[alberto@digital alberto]$ kill %1
[alberto@digital alberto]$ kill 872
```

As you can see, the shell both is powerful and complex. Even though we have only barely scratched its surface, what you know can help you to get loads of work done

Review:

Some things you can do with jobs:

kill it: `kill -9` *`jobnumber`* or `kill %1`

If you forgot to make it a background job, and it is tying up your shell, then do a

```
<ctrl>z
```

and then

```
bg
```

There may be other people using the same CPU as you are, and if your job is CPU intensive, then everyone will be fighting for resources. Even if someone wants to do simple tasks, if your job is not 'niced,' you will be despised. You will find this out when we start IRAF, as there will be jobs that get hung up and consume 99% of the CPU time, bringing your work to literally to a complete halt. Not good.

> Try to remember this tidbit: Somehow within the IRAF processes we work through, vim goes out of control and consumes almost all of the CPU effort. Your computer will seem not only sluggish, but totally drunk. We may not have this problem with the new computers, but please be aware that it might happen. If it does, then use the command `top` (explained below) to expose the culprit.

So, let's be nice. Kill your background job if you haven't already. Now, type the following command:

```
nice +19 ls –R ../../ > files.txt &
```

Did you use your up-arrow to save time? Did you get a file-error message? Deal with it. For 'nice' +19 means lowest priority, -20 means 'everyone else be damned.'

Type:

top | grep *yourusername*

Note the **R** followed by the **N** now, indicating how considerate you are.

Logging onto to other machines -- use of **ssh**

You could log on to any other networked machine to which you have access:

ssh -l <*username*> <*computer_name*>

e.g.

```
ssh -l edwinhubble astrolab01.astro.washington.edu
```

Alternatively, this also works:

```
ssh edwinhubble@astrolab01.astro.washington.edu
```

Emacs (an important review)

In order to create and modify the contents of a text file, we use a program generically called an editor. There are a wide variety of editors and preferences run strong. For this class we will learn how to use the one editor that you can be almost sure to find on any UNIX system. It's called "emacs" (two other ubiquitous editors are "vi" - a complete pain for some, and "pico" - the editor used by standard "pine".)

A page with a summary of the basic emacs commands is appended at the end of this document. Commands are entered using the control or alt keys simultaneously with the letter/number, or a sequence. To enter text it's just like Word: you click where you want to type and just type away.

Open a file for editing by typing

```
emacs testfile &
```

You can save the contents of the file and exit the editor by typing:

```
<Ctrl>x <Ctrl>c
```

or, though the icons at the top of the window (!). Emacs always makes a back-up copy of the original version of a file. So if you open testfile again and change something, an ls will show "testfile" and "testfile~", the latter being the original version of "testfile"- it's a very useful feature.

Navigating around in the file can usually be done with the mouse, and a left click at the new position, the arrow keys to go down line by line, <Ctrl>v to page down and <alt>v to page up. Lastly, you can jump to any line (useful for large files) <Ctrl>g and enter the line number in the command buffer at the bottom.

You can search for (and replace) text by typing <alt>% then foo, where foo is the text you're seeking (I'll leave you to use the online help to find out more about that command). There is a similar command in the icons at the top of the page. Emacs has been configured to act much like an editor in Windows or a Mac.

****

IRAF setup

Open up an xterm if you don't have one already. In the xterm window, type

```
xgterm &
```

to bring up an **xgterm** window. You should then see a white-background terminal window. Do a pwd just to make sure you are in your **COURSE** directory. Place the mouse cursor over that window and type

```
setenv iraf /net/iraf/iraf
```

```
$iraf/unix/hlib/mkiraf.csh
```

in order to set up the requisite IRAF initialization file. When prompted for the terminal type, you should answer with `xgterm`.

[In the two command lines above, the first one sets the “word” and the second one uses it. Thus, we could also have typed

```
setenv sneeze /net/iraf/iraf
$sneeze/unix/hlib/mkiraf.csh
```

and it would have done the same thing.]

IRAF setup

Before we can use the IRAF image analysis utility, we need to do some IRAF initialization tasks. There is a command on the Linux system that does this for you, called `mkiraf` (which you just used if you were following directions).

Running the **mkiraf** task produces some new files. One is called “**login.cl**”, in your ***course*** directory. This file contains setup and initialization parameters that IRAF uses when it starts up. It also creates a new directory called “uparm”. The “uparm” directory contains parameter files that will be modified as you use the various tasks in IRAF.

Check that the login.cl file resides in your **course** directory. If not, then go ahead and run `mkiraf`, with the default terminal set to “xgterm”. You can repeat **mkiraf**, but if you do, it will ask if you want to reinitialize your uparm directory. Plus, when you do, it will also overwrite any changes you personally made to your login.cl file.

Take a look at the “login.cl” file with the `less` utility. Lines in login.cl that start with “#” are commented out, i.e. not read by IRAF when it starts up. Lucky for us (since we need practice with the editor!) the defaults that mkiraf creates are inappropriate for our system. We’ll need to do some minor editing on the login.cl file.

The login.cl file in `/homes/larson` has the entries that are appropriate for our system. There is an easy way to find differences between the login.cl file you just created and the one in the above directory. Try this:

```
diff login.cl ~larson/login.cl
```

The diff command lists differences between two text files. It might be useful to keep a record of the differences, so try

```
diff login.cl ~larson/login.cl > diffs
```

which redirects the differences into a file called “diffs”. Now, let’s open up a new window and use it as we edit the login.cl file. To do this, you can invoke the xgterm command, which will open up a new window. Type

```
xgterm &
```

Move the mouse cursor over that window and activate it. Now in the new window (check to see in which directory it opens up) type

```
less diffs
```

which will allow you to move down through the file of differences. A space bar gives you a full page, a carriage return gives you the next line. You can get out of the `less` utility in the xgterm window by typing `q`.

In the original xterm window (the one in which you started), type

```
emacs login.cl &
```

to start editing the file. Look over at the comparison file and make appropriate changes to bring your login.cl file into conformity with one that is known to work. In particular be sure that you set

set home =“/net/projects/Astro_480/spring-06/*yourusername/*” including the quotation marks and the trailing slash symbol
set imdir =”home$images/”

where *username* should be your login user name. The “imdir” is the name of the directory where you should store your images. [CHECK: do you actually have a subdirectory named ***images***?] Be sure to include the trailing slashes in the directory names.

Then set up the default image type, over-write mode, and standard image display by setting :

set	imtype	=”fits”
set	clobber	= no
set	stdimage	= imt2048.

Then update and quit emacs by saving via the icon or by typing `<Ctrl>x <Ctrl>c.` Run the diff task again (use up-arrow!) and check the result.

Basic environment setups and setting useful aliases

If you recall earlier on in this exercise, you set an environment variable in order to run IRAF. Rather than do that each time you log on (and in every shell) you can set such variables in your ENV/.cshrc.personal. In addition this is the place to set up any command aliases you'd like.

For instance I find it much safer to make the delete command `'rm'` interactive, so that you cannot accidentally delete a file. The same goes for the copy `'cp'`, and move `'mv'` commands. As further editing practice take a look in my ~larson/ENV/.cshrc.personal and edit yours as you please. It is standard operating procedure to steal other people's ideas for these things! The grad students probably have the best files.

Source this file (`source .cshrc.personal`) or exit all terminals and open them up again, and test these aliases (if it doesn't work try putting the same lines into your ENV/.bash_profile.personal). The handiest alias alive is the one where you alias a long tree of directories. I've used ast480 as an alias for

```
cd /net/projects/Astro_480/spring-06/larson
```

Note: if you see double or single quotes in the environment files, then these are absolutely necessary.

Information Transfer: File Compression

It is often useful to reduce the size of a data file, in order to reduce the storage space needed and to reduce the time it takes to transfer the file across the net. There are two basic classes of compression algorithms: lossy and lossless. Lossless compression retains all the information that exists in the original file, and basically packs it into fewer bits. Lossy compression, on the other hand, achieves a smaller data file but at the expense of information.

In general, astronomers shy away from lossy compression algorithms for data files. It is usually a struggle to attain the desired signal-to-noise ratio, so why give it up?

When considering whether to just transfer a file as is, or to implement a sequence of compress/transfer/uncompress you need to consider the trade-off between reduced transfer speed vs. increased computational load on the two ends of the transfer pipe. In my experience there is not a unique optimum decision: it depends on the power of the two machines and the overall throughput of the network connection.

A commonly used lossless compression scheme under Linux is gzip (and the inverse operation, gunzip). Typical text files compress by about 50% when gzip'd. The syntax for gzip is pretty simple. To compress a file called foo.dat you would type

```
gzip foo.dat
```

And this would produce a file called foo.dat.gz

Compressed files generally have suffixes of .gz or .Z. Beware of files with such names, as trying to view them on the screen will likely lead to a variety of control characters being sent to the terminal, often sending it into some kind of trance.

You can uncompress a file with the gunzip command, so to undo what was done above you would type

`gunzip foo.dat.gz` (or just `gunzip foo.dat`)

Notice that the suffix .gz is implicit, you don't need to type it.

Tab completion and mouse copying

We learned about tab completion earlier. Mouse copying is another handy, time-shaving tool for cut-and-paste operations. You can click the left mouse button and drag to select some text. This is particularly effective when you're moving things between windows. You can then move the mouse to the desired window, and activate it with one right click. If you then hit the middle mouse button (or the wheel- press down) the text will be pasted in at the cursor position. (This may be window-manager sensitive – e.g. KDE versus Gnome.)

Information Transfer: tar archives.

Imagine you've just finished an extended observing run and have generated literally hundreds of images, each residing in an individual data file. This is a common occurrence in high data rate astronomical observing systems. Rather than deal with the bookkeeping and labor of transferring each file, one at a time, to a remote system (or onto a CD) it's much easier to deal with a single bundle that contains all the files of interest. The Linux utility "tar" accomplishes this.

The tar command will take a list of files and produce a single file that contains the files in question. This tar archive, a "tarball" can then be transferred around as easily as a single file.

To bundle a group of files into a tar archive, use

```
tar -cvf  <tarfile> <file list>
```

For example, if I wanted to make a tar file called `images.tar` that contained all the files that end in .fits (a common extension for images) I would type

```
tar -cvf  images.tar *.fits
```

Note that the tar command does not conform to the syntax that is typical for Linux commands (command source destination). In order to unravel a tar archive and recover the constituent files, you would type

```
tar -xvf  <tarfile>
```

Check out the man page on the tar command (try `man tar`) to see the mind-numbing variety of options that the "tar" command posses. It's common to first make a tar file, and then compress the tar file in order have a convenient bundle of files that are a minimum size. It's common to see files that end in ".tar.gz".

The options listed above, however, are almost always the only ones you need. These 4 do the following:

-c: create

-v: verbose (print to the screen what is being (un)tarred)
-x: extract
-f: create a file, don't tar onto a tape (a bit of arcane linuxness that you may ignore)

Information Transfer: Passing files between machines.

Now that we know how to produce compact tar files of data, the next step is to learn how to transfer them between machines. There are a number of options, with a range of security properties. If you have access to an account on each end of the file transfer then the preferred method is to use the "scp" (for secure copy) utility. This will allow you to move a file without ever sending a password in the clear, but requires authentication.

To move a file called "mydata.tar.gz" from a hypothetical machine called hubble at the Apache Point Observatory (with network node hubble.apo.nmsu.edu) to my home directory on the astrolab cluster, I would type

```
scp larson@hubble.apo.nmsu.edu:~/mydata.tar.gz  ~
```

The syntax is

```
scp user@node:path/sourcefile    user@node:path/destfile
```

and sourcefile and destfile refer to the source and destination file names, respectively, and the paths are pathnames to those files. You should try this out. Transfer a file or two between the machine you're logged into, and another machine in the astrolab cluster. You'll be prompted for a password on each file transfer. The protocol does support wildcards (i.e. *), etc., but only if you enclose the full file name in quotes e.g. `user@node`:'path/sourcefile' , but often you will still want to know the names of the file(s) you are transferring.

Other archaic information transfer protocols such as telnet and ftp (except anonymous ftp run by a careful system administrator) are now forbidden, as they are totally insecure.

Information Transfer: File Permissions (review)

Since Linux is a multiuser operating system, some file protection protocols have been established. There are 3 categories of users: the entire world, your "group", and the individual user. The authority to read, write and (in the case of a program) execute a file can be granted or denied to each of these classes of users. When you do an "ls –l" listing of a directory, the information on the far left hand side of each line pertains to file permission characteristics. All files are owned by individual users, each of whom is a member of a group.

The first column shows whether the object is a directory or not. Directories are listed with a "d" in the first column. The next 9 columns show whether the world, group, and file owner have read, write and execute privileges for the file in question. A dash means they lack the relevant permission, whereas the letters r, w, and x in the relevant columns show that the respective action is allowed.

In moving data around and between Linux systems, file permission characteristics can be a major inconvenience. For example, you may wish for some foreign colleague to "scp" over your hot-off-the-telescope images, but the instrument data files might be generated with conservative permission properties, that deny non-group members the ability to read/copy the file.

You can overcome most of these permission problems by changing the permission properties of the data files to allow open access. This is done with the "chmod" command, for example

```
chmod 777 *.fits
```

would allow all users all privileges on all the files ending in .fits that reside in the current directory. The chmod command takes two arguments. The first is a number that is used as a byte mask that establishes the permissions for the file that is named as the second argument. See the man pages on chmod and ls for more details.

Some emacs Commands

Inserting text

left click at position — Insert, starting at the cursor
"insert" key — toggles between insert and overwrite modes
<Ctrl>x i then enter *file* — inserts contents of file at cursor position
<Ctrl>g then enter *line number* — goes to given line in file

Deleting text

delete — deletes character preceding the cursor
<Ctrl>k — deletes current line of text to right of (and under) cursor
highlight block of text and press "delete" — deletes the highlighted section

Undoing something

<Ctrl>- — that is press control key and hyphen simultaneously

Opening/closing subwindows

<Ctrl>x 2 — divides emacs window in 2 horizontally (can repeat to further subdivide)
<Ctrl>x 3 — divides emacs window in 2 vertically
<Ctrl>x 1 — Ctrl

Exiting emacs

<Ctrl>x <Ctrl>s — writes a new version of the file
<Ctrl>x <Ctrl>c — quits the emacs program, with options to save text if needed