A Quickie Intro to UNIX, Linux, MacOSX

The Filesystem + some tools

For Snooping Around

1) **pwd**
   - print the working directory

2) **ls**
   - list names of the files and subdirectories in the current directory
   - ⋆ **ls -F**
     list directory contents, with terminating marks to indicate subdirectories and executable files.
   - ⋆ **ls -a**
     list all files, even those begining with a '.'
   - ⋆ **ls -l**
     list files in 'long' form giving a lot of information about each one

3) **cd *dirname***
   - change the current working directory to *dirname*
   - ⋆ **cd *data.d***
     change whatever the current working directory is to the directory *data.d*. This example assumes *data.d* is a subdirectory (child) of the current directory.
   - ⋆ **cd *..***
     change working directory to the one immediately above (parent).
   - ⋆ **cd**
     cd with no arguments means go to the default place — your home directory

4) **who**
   - print the list of users currently logged onto this computer

5) **date**
   - print the date and time

6) **cat *filename...***
   - "copy all text" of *filename(s)* to the screen. When more than one filename is given, the files are con**cat**enated end-to-end.

7) **more *filename***
   - will "cat" a file to the screen, one screenfull at a time. Hit <return> to advance by only one line, <space bar> to advance by a screenfull, and <q> to quit.

8) **file *filename***
   - this will make a good guess at what is contained in the file *filename*. Some possible responses are: binary data, ASCII text, C program text, shell commands text.

For Shuffling Files

9) **rm** ***file..***
   - remove the files given in the list *file..*
   - ⋆ **rm** ***data.1 data.2***
     removes the files *data.1* and *data.2*

10) **rmdir** ***dirname***
   - remove a directory
   - ⋆ **rmdir** ***data.d***
     remove the directory *data.d*. The directory to be removed must be empty (see **rm**).

11) **mkdir** ***dirname***
   - make a directory
   - ⋆ **mkdir** ***data.d***
     make the directory *data.d*. The parent will be the current directory.

12) **mv** ***file1 file2***
   - moves *file1* to *file2*. The effect is to change the name of the file *file1* to *file2*.

   mv ***file.. dirname***
   - moves the file(s) into directory *dirname*. Originals "disappear". The moved files retain modification time, ownership and everything else.

13) **cp** ***file1 file2***
   - copy the contents of *file1* into *file2*. If *file2* exists it will be overwritten, so be careful. Copying a file onto itself doen't work.

   cp ***file.. dirname***
   - puts copies of the file(s) into directory *dirname*. Originals remain untouched.

Miscellaneous

14) **lpr** ***file..***
   - copies files to the line printer

15) **echo** ***arguments***
   - echoes all of its arguments, exactly as it sees them
   - ⋆ **echo** ***hi there***
     this should echo `hi there` on your terminal screen.

16) **head** ***filename***
   - print the first 10 lines of the file *filename*

17) **tail** ***filename***
   - print the last 10 lines of the file *filename*

18) **wc** ***filename***
   - "word-count" actually returns three numbers: number of lines, number of 'words' (bunches of characters separated by spaces), and number of characters (includes space characters).

Redirecting Output

Programs usually read input from somewhere and write output to somewhere. By default UNIX commands usually read from the terminal keyboard (standard input) and write to the terminal screen (standard output). But you will often want output to go into files or directly into other commands or programs. The UNIX shell helps you to redirect standard input and standard output very simply.

19) ***program*** **>** ***file***

- redirects standard output into a file. Any UNIX command that writes on standard output can have its output written into a file instead.

⋆ **echo *I like spiders and snakes*** **>** ***critters***
causes the string "I like spiders and snakes" to be entered into the file named "critters". The redirector '>' clobbers or creates — anything already in *critters* would be lost.

20) ***program*** **>>** ***file***

- redirects standard output into a file. Same as a single '>', but does not clobber existent files. Instead the output is appended to the end of the file.

⋆ **echo *Birds are covered with icky feathers*** **>>** ***critters***
causes the string "Birds are covered with icky feathers" to be appended to the end of the file named "critters".

21) ***program***$_1$ **|** ***program***$_2$

- redirects standard output into another program, or command. This '|' is called a "pipe", and the sequence of linked programs is a "pipeline". Many programs serve as filters: they acquire input, change the data somehow and write the results on standard output. The UNIX pipeline allows you to easily combine filter 'tools', each with a clearly defined function, into a sequence to perform a particular, more complex task. This is one of the features that makes UNIX especially nice to use.

⋆ **echo *pipelines are versatile*** **|** ***lpr***
Prints "pipelines are versatile" on the line printer. Wow! A pipeline can be a lot longer than this, with more than two linked commands or programs.

The 'Toolbox', Metacharacters and Permissions

The success of the UNIX operating system is due largely to its flexibility which, in turn, springs mainly from its philosophy of software modularity — its 'toolbox' approach to computing utilities. These 'tools' are so varied, and so flexible, that it is difficult to conceive of a text or data manipulation problem that cannot be accomplished using them. Programming in languages such as C or Java is almost never needed for such tasks.

The commands **man** and **man -k** make it easy to find and use these program tools. If you don't know the name of the command that might solve your problem, then use the **man -k** command. Type:
man -k *topic*

where topic is a word that you guess, and that is relevant to your problem. **man -k** will give you a list of the commands and command-descriptions that contain your word *topic* somewhere within them. This is a crude procedure, but it is often effective.

Once you know the command name, and you want to know more about it, or how to use it, look it up in the UNIX Programmer's Manual. An on-line copy of the UPM is available through the **man** command. To find out more about it, use it on itself by typing
man *man*

File Permissions

give you total control over who can do what with your files. There are 3 kinds of permissions (read, write, execute) for 3 different groups of people (you, your group, the world). To check the permissions on *thisfile*, issue the command
ls -l *thisfile*
and inspect the string of characters at the beginning of the line. It potentially can be
`rwxrwxrwx`
which means that all three categories of people have read, write, and execute permision. Most files are created by default as
`rw-r--r--`
meaning that the owner has read and write permission, and all other people can only read it. Read permission means that a file can be copied and printed. Write permission allows modification or removal of the file. If a file has execute permission, it can be invoked as a command simply by typing its name.

File permission "modes" are changed using the **chmod** command.

Metacharacters

or "wild card characters" are short cuts to save typing time and hassle. Let's say you are in a directory that contains the following files (as revealed by **ls**):

```
file1 file2 filethree filex junkola
```

***** matches any pattern. Examples:
rm * (would remove all of the files! *BEWARE!*)
rm j* (removes "junkola", only)
rm *h* (removes "filethree", only)

? matches any single character. Example:
rm *file?* (removes files "file1", "file2", and "filex")

[*list of possibles*]Examples:
rm *file[2x]* (removes only "file2" and "filex")

[c1-c2] matches any character in the range between character "c1" and character "c2", inclusive. Possible ranges are [0-9], [A-Z], [a-z], and subranges within them. Examples:
rm *file[1-5]* (removes only "file1" and "file2")
rm *file[a-z]* (removes only "filex")

So LOOK OUT for metacharacters in your command arguments. If you want to avoid their special meanings, precede them with a backslash (\), or surround the argument with single quotes. Examples:

echo ***, and **echo** ' *' both print a * character.
echo * will print all the filenames in the directory, because it is those names that the shell tries to match patterns with, and * matches any pattern.

22) **grep *pattern filename***
- prints out all of the lines in file *filename* which include the pattern *pattern*.
- ⋆ **grep *gizmo ****
  prints all the lines with *gizmo* in them from all files in the current directory.
- ⋆ **grep *fish references***
  is the command you would use if you had a file containing all of your references, and you wanted a list of all the articles dealing with fish. This command would print out all of the lines in the file *references* which had the word *fish* on it.

More On Redirection

Yesterday we looked at output redirection, and saw that the stream of information coming out of a program (on "standard output") could be deflected away from the screen (the default) and into a file with '>', onto the end of a file with '>>', and fed to another program as input with '|'. Sometimes it is nice to have output come to the screen AND go into a file, at the same time. So we use **tee. tee** is always used in a pipeline. Example:

who | tee *whoison* prints the output of **who** on the screen and puts the output of **who** into the file *whoison*.

So what about INPUT redirection? It can be done also. Most UNIX commands allow you to name the input file as one of the command arguments. If don't name an input file, most UNIX commands will look to the terminal keyboard for their input. From the terminal you then type the 'input file', and then end your input file with a <control>d. Example:

cat > junk

is missing the name of the file to be **cat**-ed into the file *junk*. UNIX stares at you without giving a prompt. So you type:

socks shoes
undies pants
shirt pants
pants shoes
<control>d

And now you will have your familiar UNIX prompt back, and *junk* will contain the stuff you typed.

Commands that normally seek input from the terminal can be told to take their input from a file with a '<'. You will need this more often for your own programs than for standard UNIX commands. Anyway,

tsort < *junk* > *getdressed*

works. Now you know how to get dressed in the morning. Wondering what **tsort** does? Use the **man** command! This was just a weird example of the many different UNIX tools available on any mechine running UNIX, Linux, Mac OSX, Irix, Solaris, FreeBSD, Cray OS ...

A very brief intro to a handy editor present on all of the above operating systems can be found on the next page.

vi — the visual editor

WARNING: Some people *hate* **vi**.

Other people really like this editor. In an effort to steer beginners toward this latter category (liking **vi**), here is a short introduction.

What drives some people crazy about **vi** is that it has two modes: *command* mode and *insert* mode. In command mode, keystrokes are interpreted as orders for moving around the file, for deleting pieces of the file, or for entering insert mode. While in insert mode, all keystrokes are entered into the file (even control characters) with one exception . . . *escape*. Escape returns you to command mode.

So what some people hate is that every once in a while they issue a command while insert mode, and find their command entered into the file rather than acted upon! **vi** afficionados point out that pressing the <esc> key to leave insert mode before issuing a command soon becomes habit. And why put up with two modes? Because it means that most commands can be mnemonic — making them easy to remember, and *avoiding* the need for drop-down menus or other help to remind you of weird character sequences needed to accomplish routine operations. As examples, **dw** deletes a word, **3dw** deletes 3 words, **f***x* moves you forward along a line to character *x*, and **:wq** writes your entire file and quits vi.

The following small subset of **vi** commands will allow you to do just about anything. When you've mastered these, Google for slightly more extensive cheat sheets for faster and faster ways.

vi *file* calls vi and starts you off in command mode.
Use <space>, <backspace>, and arrow keys to move the cursor around.

0 put cursor at the beginning of the line.
$ put cursor at end of line.
/pat scan for your chosen pattern *pat*.
x delete a character, i.e. cross it out!
dd delete entire line on which the cursor sits (sorry, not mnemonic)
u undo. Will undo the last change you made to the file.
:wq *write and quit*, i.e. save the file with the modifications just made
:q! quit without changing/creating the file

a append. Insert text *after* cursor. Enter insert mode.
i insert. Insert text *before* cursor. Enter insert mode.
o open line *below* the line containing the cursor. Enter insert mode.
O open line *above* the line containing the cursor. Enter insert mode.
<esc> escape (leave) insert mode, enter command mode. Use when in doubt.

vi (and *vim*, "vi improved") is a standard editor on UNIX systems of all sizes. It is also available on virtually all other operating systems (e.g. gettable for free even for Windows).

50 Nifty Commands

at	queue jobs for later execution
awk / gawk	pattern-directed scanning and processing (re-format data)
cal	displays a calendar
chmod	change file modes
cmp	compare two files
comm	select or reject lines common to two files
crontab / launchctl	schedule commands to run automatically
date	display or set date and time
df	display free disk space
diff	find differences between two files
du	display disk usage statistics
echo	write arguments to the standard output
expr	evaluate expression
file	determine file type
find	walk a file hierarchy
grep, egrep, fgrep	print lines matching a pattern (or not)
gzip, gunzip, zcat	compress or expand files
head	display first lines of a file
join	relational database operator
kill	terminate or signal a process
locate	find filenames quickly
lpr	print a file (line-printer / laser printer?)
man	format and display the on-line manual pages
mkdir	make directories

more / less	pager
open	like double-clicking on files/ directories
passwd	modify a user's password
paste	merge corresponding or subsequent lines of files
pbcopy, pbpaste	provide copying and pasting to the Clipboard (Mac)
ps	process status
pwd	return working directory name
rm	remove directory entries
rmdir	remove directories
scp	secure copy (remote file copy program)
screen	screen manager for background jobs
script	make typescript of terminal session
sed	stream editor
sh	command interpreter (shell)
sort	sort lines of text files
ssh	SSH client (remote login program)
tail	display the last part of a file
tar	tape archiver; manipulate tar archive files
tee	clone/split standard output
test	condition evaluation utility
top	display updating information about running processes
tr	translate characters
uniq	report or filter out repeated lines in a file
vi	text editors, along with ex and view
w	"what's up", summary of who is on, and system activity
wc	word, line, character, and byte count

Shell Scripts

Shell scripts

- are just text files containing UNIX commands
- can be used as commands once they are made executable by **chmod +x *script***
- use a different "language" depending on the particular shell that will execute them

Different shells

- Shells themselves are programs that read from standard input and write to standard output. The function of a shell is that of a *command interpreter* to help you specify, easily, what work you would like done. For different purposes you may use whatever shell program you think is most convenient. Some popular shells:
- **sh**
  the original UNIX shell written by Bourne. Unless you specify otherwise, UNIX systems will assume that your shell script is written in the syntax of the Bourne shell.
- **csh**
  written by Bill Joy (the vi guy), the C-shell was intended to feel more like the C language and thus more familiar to many programmers. Its innovative interactive features made it the default *interactive* shell (the one you get when you login) on all UNIX systems. But **sh** is easier to program.
- **bash**
  from the GNU project, this is the "Bourne again shell". It programs like sh and has the interactive features of csh, plus more. It is standard on Linux systems.
- you can change shells any time just by running the one you want
- you can change your default shell with **chsh**.

Programming with programs

- Multiple commands on a line are separated by a semi-colon (;)
- Split up a line by typing a backslash (\) just before you hit a <return>
- Standard output generated by commands within *backquotes* (` ) stays right there between the backquotes. So the output of **echo The date and time are: ` date`** produces:
  ```
  The date and time are Mon 21 Feb 2011 17:06:50 PST
  ```
- You can do more than just list UNIX commands or chain them together in a linear pipeline (as totally cool as pipelines are). Shell programs have all of the control structures of high level computer languages, such as **if-then**, **for-loop**, **while-loop** and even a **case** statement.
- The *condition* part of an **if** or **while** statement is not a boolean value. It is a *command*. If the command returns an exit status of 0 (meaning "I worked") then it functions as a "true".

Interactive UNIX

Configuring Your UNIX Environment

The files ".cshrc" and ".login" in your home directory contain instructions for setting up your computing environment (assuming your default shell is **csh**). The file ".cshrc" is read *any* time you start a csh. This happens more often than you may think. For example, if you run a UNIX command from the command line in **vi**, it is run in a separate shell. The ".login" file is run after ".cshrc" when you first log in. By customizing these files you can tailor UNIX to your own tastes. Shell variables and aliases are the features you will want to alter most frequently.

SHELL VARIABLES can be "set" or defined with the statement

set *varname=expression*

Now the shell variable *varname* can be used anywhere except within single quotes simply by preceding it with a $ sign. (So, if you don't want the shell to try substituting when it sees a $ sign, use single quotes. A backslash before the dollar sign also works). Suppose we save our large database files in the directory /db/demo, and often have to go there. If the line

set *g=/usr/games*

appeared in our .login file, then any time we wanted to go to /usr/games we could just type

cd $*g*

and we would be there! Note that several shell variables are set in .login . The command

set

with no arguments, will print all of the currently defined shell variables. If you just want to check the value of a particular one, type:

echo $*varname*

where *varname* is the name of the shell variable you are curious about.

ALIASES provide a way of customizing the commands you commonly use. For example, I always like to have the information that is provided by the **-F** flag when I use the **ls** command. So my .login file contains the line

alias ls ls -F and every time I type **ls** UNIX behaves as though I had typed **ls -F**. People longing for DOS (not a practical example) might want

alias dir ls so that they can type **dir** to get a list of their files. To check all of your aliases, type

alias and to check a particular one, e.g. **ls**, type

alias ls

For more information about any shell use the **man** command. Remember that the shell is just a program, and that variations exist. The default interactive shell at many installations is the "C-shell" (**csh**), the one we have been using. The most common shell to write "shell-scripts" in is the original "Bourne shell" (**sh**).

History

The **csh** introduced various means of re-issuing and/or modifying previously executed commands. They are collectively referred to as "history mechanisms". Here are some handy ones to know:

!! "bang-bang" is replaced by everything you typed on the previous command-line. You can add to the end of the command. For example if the previous command simply wrote to standard

output, you could type "**!! > filename**" to re-execute the last command but redirect its output into the file **filename**.

!$ is replaced by the last thing on the previous command line. If you had just redirected output into file **bigfilename** then you could peek at the first 3 lines of your output by typing
head -3 !$

history a long word to be typing in UNIX, but it will list the commands you have executed. And you could always make a shorter alias for it if you want to.

!12 repeats command number 12, or whatever number you choose. The numbers are those you'd see in a **history** listing, or perhaps in your UNIX prompt, it you are using a prompt with the command-number in it.

!a repeats the last command starting with an "**a**". Use whatever letter you want, and if you don't want the last command starting with an "**a**" but instead want the last command starting with "**aw**" then just type as many character as you like to make the specification unique. This one is very handy.

Job Control

Since UNIX is a multitasking operating system, you can run many tasks or 'jobs' at the same time. When you start running large programs, you might get tired of waiting for them, and want to work on something else while the computer is crunching. Or you might want to stop in the middle of a program, do something else, then restart it. No problem:

&

- is a command terminator that causes the given command to run in the 'background'. Your terminal will be freed up, giving you another prompt from the shell, so you do not have to wait for the command to terminate. Example:
  - ⋆ **grep *secret hugefile* > *stash***
    will go to work finding all the lines with the pattern *secret* in the file *hugefile*, and putting them in the file *stash*. Meanwhile, I could do anything else rather than wait for it to complete. If a program sends a lot of output to the screen, it is usually not convenient to run it in the background and do something else.

<control>z

- stops the program that is currently running in the 'foreground'. The program can later be put back in the foreground, or in the background.

jobs

- prints out your currents list of jobs, and whether they are running, stopped, or done (completed). Background jobs have a & at the end.

bg % n

- puts stopped job number *n* (in the list obtained by typing **jobs**) in the background and starts it running. With no argument it applies to the job just stopped (with <control>z).

fg % n

- starts stopped job number *n* running in the foreground.

Background jobs save you waiting time, and the job control commands allow you to change your mind a lot! If you want to scrap a particular job altogether, use

kill % n

- terminates job number *n* (in the list obtained by typing **jobs**).

Using a remote UNIX machine

ssh youracct@remote.machine.ca

This is what you use to sign on, a “secure shell". Unlike the older Telnet program, your password will not go over in the clear and, in fact, everything sent in both directions will be encrypted. Check your accountname on your current machine by typing **whoami**. If it is the same as the accountname on the remote machine, you can omit the *youracct@* part. If your current machine is actually on the .machine.ca network, you may only need to type **ssh remote**. When you sign on you will have a standard terminal window and UNIX shell.

scp filehere youracct@remote.machine.ca:.
scp youracct@remote.machine.ca:filethere .

This is the fast way to transmit a file to the home directory on the remote machine, or from there to the current directory on your local machine. It works exactly the same ways as the standard **cp** command except that you can prepend acct@host: to any pathname.

ssh-keygen -t rsa

This will generate a pair of encryption keys you can use to avoid typing your password when using **ssh** or **scp**. The keys will be put in the “.ssh" directory under your home directory on the local machine. Take the key that is the file *id_rsa.pub* and put it in (append it to) the file *authorized_keys* in the .ssh directory of the remote machine.

* * * * * * * *

To check the load and what is running on the remote machine, or any UNIX machine, use the commands **w** and **top**.

If it is easy to direct all output from the program you are running on the remote machine into a file, then you can simply start it in the background, or start it then put it in the background, sign off, and come back later to check on it. Example:

./netlogo-headless.sh –threads 4 –model lemming.nlogo –experiment discts05 &

Otherwise you can use the program **screen** to capture all of the output within a virtual screen . . . which can be detached to run in the background for later. Example:

screen
start the program
ctrl-a, d (to detach the virtual screen)
start another one, or log off to go out and play
later, ssh back in
screen -R (to re-attach), or
screen -ls (to list the choices if you left multiple screens, then screen -R thisone)
ctrl-d (to terminate the virtual screen, which is a shell)