Chapter 1

# The Linux/Unix Operating System

## 1.1 Introduction

The main purpose of this introduction is to get you familiar with the interactive use of Unix/Linux for day-to-day organizational and programming tasks. Unix/Linux is an operating system which we can loosely define as a collection of programs (often called processes) which manage the resources of a computer for one or more users. These resources include the CPU, network facilities, terminals, file systems, disk drives and other mass-storage devices, printers, and many more. During the course, the most common way you will use Unix/Linux is through a command-line interface; you will type commands to create and manipulate files and directories, start up applications such as text-editors or plotting packages, send, compile and run Fortran programs, etc.

When you type commands in Unix/Linux, you are actually interacting with the OS (Operating System) through a special program called a shell which provides a user-friendly command-line interface. These **command-line interfaces** provide powerful environments for software development and system maintenance. Though shells have many commands in common, each type has unique features. Over time, individual programmers come to prefer one type of shell over another. I recommend that you use the 'C-shell' (**csh**), the 'tC-shell' (**tcsh**), or the 'bash-shell' (**bash**) for interactive use. Everything which is described below as being a C-shell feature should work equally as well in tcsh. Note that a significant enhancement of tcsh versus csh is the availability of command-history recall and editing via the 'arrow' keys (as well as 'Delete' and 'Backspace'). After you have typed a few commands, hit the 'up arrow' key a few times and note how you scroll back through the commands you have previously issued. In the following, I assume you have at least one active shell on each system in which to type sample commands, and I will often refer to a window in which as shell is executing as the terminal.

In the following, commands which you type to the shell, as well as the output from the commands and the C-shell prompt (denoted '%' ) will appear in typewriter font. Here is an example:

```
% pwd
/home/p317
% date
Sun Jan 18 19:17:13 PST 2014
```

## 1.2 Files and Directories

There are essentially only two types of files in Unix/Linux:

- Plain files: which contain specific information such as plain text, Fortran source code, object code, executable code, postscript code, a Maple/Mathematica worksheets etc.
- Directories: special files which are essentially containers for other files (including other directories).

**Absolute and relative pathnames, working directory:** All Unix/Linux filesystems are rooted in the special directory called '/'. All files within the filesystem have absolute pathnames which begin with '/' and which describe the path down the file tree to the file in question. Thus

/home/p317/sample.txt

refers to a file named 'sample.txt' which resides in a directory with absolute pathname

/home/p317/

which itself lives in directory

/home

which is contained in the root directory

/

In addition to specifying the absolute pathname, files may be uniquely specified using relative pathnames. The shell maintains a notion of your current location in the directory hierarchy, known appropriately enough, as the working directory. The name of the working directory may be printed using the `pwd` command:

```
% pwd
/home/p317/
```

If you refer to a filename such as

file.txt

or a pathname such as

dir1/dir2/file.txt

so that the reference does not begin with a '/', the reference is identical to an absolute pathname constructed by prepending the working directory followed by a '/' to the relative reference. Thus, assuming that my working directory is

/home/p317/txt

the two previous relative pathnames are identical to the absolute pathnames

/home/p317/txt/file.txt

/home/p317/txt/dir1/dir2/file.txt

Note that although these files have the same filename 'file.txt', they have different absolute pathnames, and hence are distinct files.

**Home directories:** Each user of a Unix/Linux system typically has a single directory called his/her home directory which serves as the base of his/her personal files. The command `cd` (change directory) with no arguments will always take you to your home directory. On my Linux machine I see something like this

```
% cd
% pwd
```

```
/home/p317
```

while on the computer cluster it will be something like

/afs/sdsu.edu/user21/students

When using the C-shell, you may refer to your home directory using a tilde ('~'). Thus, assuming the home directory is

/home/p317

then

```
% cd ~
```

followd by

```
% cd dir1/dir2
```

is identical to

```
% cd /home/p317/dir1/dir2
```

**"Dot" and "Dot-Dot":** Unix/Linux uses a single period ('.') and two periods ('..') to refer to the working directory and the parent of the working directory, respectively:

```
% cd ~p317/hw1
% pwd
/home/p317/hw1
% cd ..
% pwd
/home/p317
% cd .
% pwd
/home/p317
```

Note that

```
% cd .
```

does nothing–the working directory remains the same. However, the '.' notation is often used when copying or moving files into the working directory. See below.

**Filenames:** There are relatively few restrictions on filenames in Unix/Linux. On most systems (including Linux machines), the length of a filename cannot exceed 255 characters. Any character except slash ('/)' and 'null' may be used. However, you should avoid using characters which are special to the shell (such as '(', ')', '*', '?', '$', '!') as well as blanks (spaces). In fact, it is probably a good idea to stick to the set:

a-z A-Z 0-9 _ . -

As in other operating systems, the period is often used to separate the 'body' of a filename from an 'extension' as in:

|  |  |
|---|---|
| program.f | *(extension .f)* |
| paper.tex | *(extension .tex)* |
| document.txt | *(extension .txt)* |

Note that unlike some other operating systems, extensions are not required, and are not restricted to some fixed length (often 3 on other systems). Several standard Unix filename extensions are listed below:

|  |  |
|---|---|
| .c | C language source |
| .f | Fortran 77 language source |
| .f90 | Fortran 90 language source |
| .o | Object code from a compiler |
| .pl | Perl language source |
| .ps | PostScript language source |
| .tex | LaTeXdocument |
| .dvi | LaTeXdevice independent output |
| .gif | CompuServ GIF image |
| .jpg | JPEG |
| .tar | `tar` format archive |
| .Z | Compressed file created with `compress` |
| .tar.Z, .tar.gz | Compressed (gzipped) `tar` archive |
| .a | `ar` library archive file |

The underscore and minus sign are often used to create more human readable filenames such as:

this_is_a_long_file_name
this-is-another-long-file-name

If you accidentally create a file with a name containing characters special to the shell (such as '*' or '?'), the best thing to do is remove or rename (move) the file immediately by enclosing its name in single quotes to prevent shell evaluation:

```
% rm -i 'file_name_with_an_embedded_*_asterisk'
% mv 'file_name_with_an_embedded_*_asterisk' sane_name
```

Note that the single quotes in this example are forward-quotes. Backward quotes have a completely different meaning to the shell.

## 1.3 Commands Overview

General Structure: The general structure of Unix/Linux commands is given schematically by

```
command_name [options] [arguments]
```

where square brackets ('[...]') denote optional quantities. Options to Unix/Linux commands are frequently single alphanumeric characters preceded by a minus sign as in:

```
% ls -l
% cp -R ...
% man -k ...
```

Arguments are typically names of files or directories or other text strings which do not start with '-'. Individual arguments are separated by white space (one or more spaces or tabs):

```
% cp file1 file2
% grep 'a string' file1
```

There are two arguments in both of the above examples; note the use of single quotes to supply the grep command with an argument which contains a space. The command

```
% grep a string file1
```

which has three arguments has a completely different meaning.

**Executables and Paths:** In Unix/Linux, a command such as `ls` or `cp` is usually the name of a file which is known to the system to be executable. To invoke the command, you must either type the absolute pathname of the executable file or ensure that the file can be found in one of the directories specified by your path. In the C-shell, the current list of directories which constitute your path is maintained in the shell variable, 'path'. To display the contents of this variable, type:

```
% echo $path
```

The '$' mechanism is the standard way of evaluating shell variables and environment variables alike. The resulting output may look something like

```
.  /usr/sbin /usr/bsd /sbin /usr/bin /bin /usr/bin/X11 /usr/local/bin
```

The '.' in the output indicates that the working directory is in your path. The order in which path-components (first '.', then '/usr/sbin', then '/sbin', etc.) appear in your path is important. When you invoke a command without using an absolute pathname as in

```
% ls
```

the system looks in each directory in your path–and in the specified order–until it finds a file with the appropriate name. If no such file is found, an error message is printed:

```
% helpme
-bash:  helpme:  command not found
```

The path variable is typically set in your '~/.login' file and/or (preferably) your '~/.cshrc' file. Examine the file '~/.cshrc' in your account. You should see a line like

```
set path=($path /usr/local/bin $HOME/bin)
```

which adds '/usr/local/bin' and '$HOME/bin' to the previous (system default) value of 'path'. Also note the use of parentheses to assign a value containing whitespace to the shell variable. 'HOME' is an environment variable which stores the name of your home directory. Thus

```
set path=($path /usr/local/bin ~/bin)
```

will produce the same effect.

**Control Characters:** The following control characters typically have the following special meaning or uses within the C-shell. You should familiarize yourself with the action and typical usage of each. I will use a caret ('^') to denote the Control (`ctrl`) key. Then

```
% ^Z
```

for example, means depress the z-key (upper or lower case) while holding down the Control key.

- ^D: End-of-file (EOF). Type ^D to signal end of input when interacting with a program (such as `Mail`) which is reading input from the terminal. Here's an example using Mail:

  ```
  % Mail -s "Test Message" fweber@sciences.sdsu.edu
  This is a one line message.
  ^D
  EOT
  %
  ```

  If you try the above exercise, you will notice that the shell does not 'echo' the ^D. This is typical of control characters–you must know when and where to type them and what sort of behaviour to expect. In this case, Mail is gracious enough to echo the characters EOT (end-of-transmission) when you type ^D, but other commands, such as `cat`, will not echo anything. In almost all cases, however, you should be presented with a csh prompt. Also, by default, a C-shell exits when it encounters EOF, so if you type ^D at a csh prompt, you may find that you are logged out. If you don't like this behaviour (I don't), put the following line in '~/.cshrc':

  ```
  set ignoreeof
  ```

- ^C: Interrupt. Type ^C to kill (stop in a non-restartable fashion) commands (processes) which you have started from the command-line. This is particularly useful for commands which are taking much longer to execute or producing much more output to the terminal than you had anticipated. Many commands catch interrupts and you may sometimes have to type more than one to get out. Here's an example, again from Mail,

  ```
  % Mail -s "A message which I decide not to send" fweber@sciences.sdsu.edu
  This is a one line message.
  ^C
  (Interrupt -- one more to kill letter)
  ^C
  %
  ```

- ^Z: Suspend. Type ^Z to suspend (stop in a restartable fashion) commands which you have started from the shell. It is often convenient to temporarily halt execution of a command.

**Special Files:** The following files, all of which reside in your home directory, have special purposes and you should become familiar with what they contain on the systems you work with:

- ~/.cshrc

  Commands in this file are executed each time a new C-shell is started.

- ~/.login

  Commands in this file are executed after those in '~/.cshrc' and only for login shells. When interacting with Unix/Linux via a windowing system, it is easy to start an interactive shell which is not a login shell, but for which you presumably want the same initialization procedure. Consequently, your '~/.login' should be probably be kept as brief as possible and you should put start-up commands in '~/.cshrc' instead.

Note that files whose name begins with a period ('.') are called hidden since they do not normally show up in the listing produced by the '`ls`' command. Use

```
% cd; ls -a
```

for example, to print the names of all files in your home directory. Note that I have introduced another piece of shell syntax in the above example; the ability to type multiple commands separated by semicolons (';') on a single line. There is no guaranteed way to list only the hidden files in a directory, however

```
% ls -d .??*
```

will usually come close.

**Shell Aliases:** The syntax of many Unix/Linux commands is quite complicated and furthermore, the 'bare-bones' version of some commands is less than ideal of interactive use, particularly by novices. The C-shell provides a mechanism called aliasing which allows you to easily remedy these deficiencies in many cases. The basic syntax for aliasing is

```
% alias name definition
```

where 'name' is the name (use the same considerations for choosing an alias name as for filenames; i.e. avoid special characters) of the alias and 'definition' tells the shell what to do when you type 'name' as if it was a command. The following examples should give you basic idea; see the `csh` documentation (`man csh`) for more complete information:

```
% alias ls 'ls -FC'
```

provides an alias for the `ls` command which uses the -F and -C options (these options are described in the discussion of the ls command below). Note that the single quotes in the alias definition are essential if the definition contains white-space. The commands

```
% alias rm 'rm -i'
% alias cp 'cp -i'
% alias mv 'mv -i'
```

define aliases for `rm`, `cp` and `mv` which will not clobber files without first asking you for explicit confirmation. This is highly recommended for novices and experts alike. To see a list of all your current aliases, simply type

```
% alias
```

Aliases are typically defined either in a file '~/.aliases' in your account, or in /etc/profile.local. They can be made available in shells by typing

```
% source ~/.aliases
```

The 'source' command tells the shell to execute the commands in the file supplied as an argument.

## 1.4 Basic Commands

The following list is by no means exhaustive, but rather represents what I consider an essential base set of Unix/Linux commands (organized roughly by topic) with which you should familiarize yourself as soon as possible. Refer to the man pages for additional information.

### Getting Help and Information

**man**
Use `man` (short for 'manual') to print information about a specific Unix/Linux command or to print a list of commands which have something to do with a specified topic (-k option, for keyword). It is difficult

to overemphasize how important it is for you to become familiar with this command. Although the level of intelligibility for commands (especially for novices) varies widely, most basic commands are thoroughly described in their man pages, with usage examples in many cases. It helps to develop an ability to scan quickly through text looking for specific information you feel will be of use. Typical usage examples include:

```
% man man
```

to get detailed information on the `man` command itself,

```
% man cp
```

for information on `cp` and

```
% man -k 'working directory'
```

to get a list of commands having something to do with the topic 'working directory'. The command apropos, found on most Unix/Linux systems, is essentially an alias for `man -k`.

### Communicating with Other Machines

#### ssh

Use `ssh` to securely login into another (remote) machine. Typical usage is

```
% ssh edelweiss.sdsu.edu -l p317
```

or, alternatively,

```
% ssh p317@edelweiss.sdsu.edu
```

which will initiate a remote login for user 'p317' on machine 'edelweiss.sdsu.edu'. ('-l' stands for login-name.)

#### sftp

Use `sftp` to establish a secure connection to another machine for the express purpose of copying files between the two machines. Here's an example illustrating how I might copy files from host 'grizzly.sdsu.edu' to 'rohan.sdsu.edu':

```
% sftp fweber@rohan.sdsu.edu
Connecting to rohan.sdsu.edu...
fweber@rohan.sdsu.edu's password:  xxxxxx
sftp>
sftp> pwd
Remote working directory:  /home/ph/fweber
sftp> ls
hosts mbox public_html
sftp> !pwd
/home/fweber
sftp> !ls
file.ps digiKam kAstro myLinks shellscripts
sftp> put file.ps
uploading file.ps to /home/ph/fweber/
sftp> ls
hosts mbox public_html file.ps
```

`sftp` has a fairly extensive on-line help. Try `sftp> help` as well as

```
sftp> help bin
sftp> help cd
sftp> help lcd
sftp> help put
sftp> help get
sftp> help prompt
sftp> help mget
```

to learn the basics.

## Creating, Manipulating and Viewing Files (including Directories)

**vi** or **emacs**
It is absolutely crucial that you become facile with one of these standard Unix/Linux editors. Either of these two editors will more than suffice for the creation, modification and viewing of text files at the level required for this course. A brief introduction to `vi` and `emacs` is provided in Chapter 2

**more**
Use more to view the contents of one or more files one page at a time. For example:

```
% more ~/.cshrc
alias rm='rm -i'
alias mv='mv -i'
alias cp='cp -i'
alias dir='ls -aF'
--More--(55%)
```

In this case I have executed the 'more' command in a shell window containing only a few lines. The

```
--More--(55%)
```

message is actually a prompt: hit the spacebar to see the next page, type 'b' to backup a page, and type 'q' to quit viewing the file. Refer to the man page for the many other features of the command.

**lpr**
Use `lpr` to print files. By default, files are sent to the system default printer, or to the printer specified in your PRINTER environment variable. The typical usage is

```
% lpr file_to_be_printed.ps
% lp -d Printername file_to_be_printed.ps
```

which prints a postcript file. If you want to print a regular text file or a fortran program, convert these files first to postscript file using the `enscript` command, Type `man enscript` for more information. The typical usage is

```
% enscript -o file_to_print.ps input_file.txt
```

**cd** and **pwd**
Use `cd` and `pwd` to change and print, respectively, your working directory. We have already seen examples of these commands above. Here's a summary of typical usages (note that I use semi-colons to separate distinct Unix/Linux commands issued on the same line):

```
% cd
% pwd
/home/fweber
% cd ~; pwd
/home/fweber
% cd /tmp; pwd
/tmp
% cd ..; pwd
/
```

Recall that '..' refers to the parent directory of the working directory so that

```
% cd ..
```

takes you up one level in the file system hierarchy.

**ls**

Use `ls` to list the contents of one or more directories. Recall that I advocate the use of the alias

```
% alias ls='ls -FC'
```

which will cause `ls` to (1) append special characters (notably '*' for executables and '/' for directories) to the names of certain files (the -F option) and (2) list in columns (the -C option). Example:

```
% cd
% ls
AnnualReps/ downloads/ Mathematica/ Proposals+Grants/ tmpdir@
```

Note that the files ending with '/' are directories while files marked with '@' are links. To see hidden files, use the '-a' option:

```
% cd ~; ls -a
.gimp-2.2 .profile .xinitrc .xemacs
```

and to view the files in 'long' format, use '-l':

```
% cd ~fweber/tex/; ls -lF
total 224
-rwxr-xr-x 5 fweber users 4096 2005-08-25 19:18 data
lrwxrwxrwx 1 fweber users 17 2007-12-22 14:19 prns -> codes/fcodes/prns/
drwxr-xr-x 2 fweber users 4096 2008-06-20 13:13 ApJ,91/
drwxr-xr-x 6 fweber users 4096 2007-12-03 22:21 Backbending/
```

The output in this case is worthy of a bit of explanation. First observe that `ls` produces one line of output per file/directory listed. The first field in each listing line consists of 10 characters which are further subdivided as follows:

- first character: file type: '-' for regular file, 'd' for directory, 'l' for a link.
- next nine characters: 3 groups of 3 characters each specifying read (r), write (w), and execute (x) permissions for the user (owner of the file), user's in the owner's group and all other users. A '-' in a permission field indicates that the particular permission is denied.

Thus, in the above example, 'data' is a regular file, with read, write and execute permissions enabled for the owner (user 'fweber') and read and execute permissions enabled for member's of group 'users' and all other users. Note that you must have execute (as well as read) permission for a directory in order to be able to cd to it. See chmod below for more information on setting file permissions. Continuing to decipher the file listing, the next column lists the number of links to this file, then comes the name of the user who owns the file and the owner's group. Next comes the size of the file in bytes, then the date and time the file was last modified, and finally the name of the file. If any of the arguments to ls is a directory, then the contents of the directory is listed. Finally, note that the '-R' option will recursively list sub-directories:

```
% cd ~; pwd
/home/fweber
% ls -R
./classes/SDSU/phys-317/course_info:
syllabus.odt
syllabus.pdf
syllabus.ps.gz
```

**mkdir**
Use mkdir to make (create) one or more directories. Sample usage:

```
% cd ~
% mkdir tempdir
% cd tempdir; pwd
/home/fweber/tempdir
```

If you need to make a 'deep' directory (i.e. a directory for which one or more parents does not exist) use the '-p' option to automatically create parents when needed:

```
% cd ~
% mkdir -p a/long/way/down
% cd a/long/way/down; pwd
/home/fweber/a/long/way/down
```

In this case, the mkdir command made the directories

/home/fweber/a, /home/fweber/a/long, /home/fweber/a/long/way,

and, finally

/home/fweber/a/long/way/down

**cp**
Use cp to (1) make a copy of a file, or (2) to copy one or more files to a directory, or (3) to duplicate an entire directory structure. The simplest usage is the first, as in:

```
% cp foo bar
```

which copies the contents of file 'foo' to file 'bar' in the working directory. Assuming that cp is aliased to cp -i as recommended, you will be prompted to confirm overwrite if 'bar' already exists in the current directory. Otherwise a new file named 'bar' is created. Typical of the second usage is

```
% cp foo bar /tmp
```

which will create (or overwrite) files

```
/tmp/foo /tmp/bar
```

with contents identical to 'foo' and 'bar' respectively. Finally, use cp with the '-r' (recursive) option to copy entire hierarchies.

**mv**

Use mv to rename files or to move files from one directory to another. Again, I assume that mv is aliased to mv -i so that you will be prompted if an existing file will be clobbered by the command. Here's a 'rename' example:

```
% ls
thisfile
% mv thisfile thatfile
% ls
thatfile
```

The following sequence illustrates how several files might be moved up one level in the directory hierarchy:

```
% pwd
/tmp/lev1
% ls
lev2/
% cd lev2
% ls
file1 file2 file3 file4
% mv file1 file2 file3 ..
% ls
file4
% cd ..
% ls
file1 file2 file3 lev2/
```

**rm**

Use rm to remove (delete) files or directory hierarchies. The use of the alias rm -i for cautious removal is highly recommended. Once you have removed a file in Unix/Linux there is essentially *nothing* you can do to restore it other than restoring a copy from backup tapes (assuming the system is regularly backed up). Examples include:

```
% rm thisfile
```

to remove a single file,

```
% rm file1 file2 file3
```

to remove several files at once, and

```
% rm -r thisdir
```

to remove all contents of directory 'thisdir', including the directory itself. Be particular careful with this form of the command and note that

```
% rm thisdir
```

will not work. Unix/Linux will complain that 'thisdir' is a directory.

**chmod**

Use `chmod` to change the permissions on a file. See the discussion of `ls` above for a brief introduction to file-permissions and check the man pages for `ls` and `chmod` for additional information. Basically, file permissions control who can do what with your files. Who includes yourself (the user 'u'), users in your group ('g') and the rest of the world (the others 'o'). What includes reading ('r'), writing ('w', which itself includes removing/renaming) and executing ('x'). When you create a new file, the system sets the permissions (or mode) of a file to default values which you can modify using the

```
% umask
```

command. (See `man umask` for more information). On Linux machines, your defaults should be such that you can do anything you want to a file you have created, while the rest of the world (including fellow group members) normally has read and, where appropriate, execute permission. As the man page will tell you, you can either specify permissions in numeric (octal) form or symbolically. I prefer the latter. Some examples which should be useful to you include:

```
% chmod go-rwx myFirstFortranProgram.f
```

which removes all permissions from 'group' and 'others'. A `% ll -l` therefore produces on the following terminal output,

```
-rw------- 1 p317 users 3233 2011-01-20 13:09 myFirstFortranProgram.f
```

To make a file executable by everyone ('a' stands for all and is the union of user, group and other) type

```
% chmod a+x executable_file.o
```

and

```
% chmod u-w file_to_write_protect.txt
```

to remove the user's (your) write permission to a file to prevent accidental modification of particularly valuable information. Note that permissions are added with a '+' and removed with a '-'.

## 1.5 More on the C-Shell

Shell Variables: The shell maintains a list of local variables, some of which, such as 'path', 'term' and 'shell' are always defined and serve specific purposes within the shell. Other variables, such as 'filec' and 'ignoreeof' are optionally defined and frequently control details of shell operation. Finally, you are free to define you own shell variables as you see fit (but beware of redefining existing variables). By convention, shell variables have all-lowercase names. To see a list of all currently defined shell variables, simply type

```
% set
```

To print the value of a particular variable, use the Unix/Linux `echo` command plus the fact that a '$' in front of a variable name, causes the evaluation of that variable:

```
% echo $path
```

To set the value of a shell variable use one of the two forms:

```
% set thisvar=thisvalue
% echo $thisvar
```

```
thisvalue
```

or

```
% set thisvarlist=(value1 value2 value3)
% echo $thisvarlist
value1 value2 value3
```

Shell variables may be defined without being associated a specific value. For example:

```
% set somevar
% echo $somevar
```

The shell frequently uses this 'defined' mechanism to control enabling of certain features. To undefine a shell variable use **unset** as in

```
% unset somevar
% echo $somevar
somevar - Undefined variable
```

Following is a list of some of the main shell variables (predefined and optional) and their functions:

- path: Stores the current path for resolving commands.
- prompt: The current shell prompt–what the shell displays when it is expecting input.
- cwd: Contains the name of the (current) working directory.
- term: Defines the terminal type. If your terminal is acting strangely, the command
  `% set term=vt100; resize`
  often provides a quick fix.
- noclobber: When set, prevents existing files from being overwritten via output redirection (see below).
- filec: When set, enables file-completion. Partially typing a file name (using an initial sequence which is unique among files in the working directory), then typing 'TAB' will result in the system doing the rest of the typing of the file-name for you.
- shell: Which particular shell you are using
- ignoreeof: When set, will disable shell 'logout' when ^D is typed.

**Environment Variables:** Unix/Linux uses another type of variable–called an environment variable–which is often used for communication between the shell (not necessarily a C-shell) and other processes. By convention, environment variables have all uppercase names. In the C-shell, you can display the value of all currently defined environment variables using

```
% env
```

Some environment variables, such as 'PATH' are automatically derived from shell variables. Others have their values set (typically in '~/.cshrc' or '~/.login' ) using the syntax:

```
% setenv VARNAME value
```

Note that, unlike the case of shell variables and set, there is no '=' sign in the assignment. The values of individual environment variables may be displayed using printenv or echo:

```
% printenv HOME
/home/fweber
% echo $HOME
/home/fweber
```

Observe that, as with shell variables, the dollar sign causes evaluation of an environment variable. It is particularly notable that the values of environment variables defined in one shell are inherited by commands (including C and Fortran programs, and other shells) which are initiated from that shell. For this reason, environment variables are widely used to communicate information to Unix/Linux commands (applications). The DISPLAY environment variable, which every X-application checks to see which display it should use for output is a canonical example. Following is a list of some of standard environment variables with their functions:

- DISPLAY: Tells X-applications which display (screen) to use for output. Typically set on remote machines so that output appears on the local screen. For example, assuming I am remote logged into 'darwin' from the console of 'grizzly', then, at my 'darwin' prompt, I can type:

  ```
  %setenv DISPLAY grizzly.phys.sdsu.edu:0.0
  ```

  after which all X-applications started on 'darwin' will display on the 'grizzly' console. If you encounter problems displaying windows from a remote application on your local console, try typing `xhost +` at any shell prompt on the local machine. See `man xhost` for more information.

- HOME: Asks the shell to substitute the environment variable HOME here. For example,

  ```
  % cd $HOME/tex
  ```

  allows you to change from any (sub) directory directly to `/tex`, which resides in your home directory. This command is equivalent to

  ```
  % cd ~/tex
  ```

- PRINTER: Defines default printer for use with `lpr` or `lp`, `enscript`, `firefox` etc. Check your '~/.cshrc' which sets this variable.

**Using C-shell Pattern Matching:** The C-shell provides facilities which allow you to concisely refer to one or more files whose names that match a given pattern. The process of translating patterns to actual filenames is known as *filename expansion* or *globbing*. Patterns are constructed using plain text strings and the following constructs, known as *wildcards:*

? Matches any single character.

* Matches any string of characters.

[a-z] Matches any single character contained in the specified range (the match set)–in this case lower-case 'a' through lower-case 'z'.

[^a-z] Matches any single character not contained in the specified range.

Match sets may also be specified explicitly, as in [02468]. Examples are:

```
% [^b-z,A-Z]*
assignment1.tex assigment1.pdf
```

list all files and folders whose names begin with an 'a' such as assignment.tex and assignment.pdf.

```
% ls ????
x.log
```

lists all regular (not hidden) files and directories whose names contain precisely *four* characters such as x.log.

```
% cp a* /tmp
```

copies all files whose name begins with 'a' to the temporary directory '/tmp'.

```
% mv *.f ../newdir
```

moves all files whose names end with '.f' to directory '../newdir'. Note that the command

```
% mv *.f *.for
```

will not rename all files ending with '.f' to files with the same prefixes, but ending in '.for', as is the case on some other operating systems. This is easily understood by noting that expansion occurs before the final argument list is passed along to the mv command. If there are not any '.for' files in the working directory, '*.for' will expand to nothing and the last command will be identical to

```
% mv *.f
```

which is not at all what was intended.

**Using the C-shell History and Event Mechanisms:** The C-shell maintains a numbered history of previously entered command lines. Because each line may consist of more than one distinct command (separated by ';'), the lines are called events rather simply commands. Type

```
% history
```

after entering a few commands to view the history. Although 'bash' (which I assume you are using) allows you to work back through the command history using the up-arrow and down-arrow keys, the following event designators for recalling and modifying events are still useful, particularly if the event number forms part of the shell prompt as it does for the initial set-up on a Linux machine:

!! Repeat the previous command line

!21 Repeat command line number 21

!a Repeat most recently issued command line which started with an 'a'. Use an initial sub-string of length $> 1$ for more specificity.

!?b Repeat most recently issued command line which contains 'b'; any string of characters can be used after the '?'

Unix/Linux users often refer to an exclamation point ('!') as 'bang'.

**Standard Input, Standard Output and Standard Error:** A typical Unix/Linux command (process, program) reads some input, performs some operations on, or depending on, the input, then produces some output. It proves to be extremely powerful to be able to write programs which read and write their input and output from 'standard' locations. Thus, Unix/Linux defines the notions of

- standard input: default source of input
- standard output: default destination of output
- standard error: default destination for error messages and diagnostics

Many Unix/Linux commands are designed so that, unless specified otherwise, input is taken from standard input (or stdin), and output is written on standard output (or stdout). Normally, both stdin and stdout are attached to the terminal. The cat command with no arguments provides a canonical example (see man cat if you can't understand the example):

```
% cat
foo
foo
bar
bar
^D
```

Here, cat reads lines from stdin (the terminal) and writes those lines to stdout (also the terminal) so that every line you type is 'echoed' by the command. A command which reads from stdin and writes to stdout is known as a *filter*.

**Input and Output Redirection:** The power and flexibility of the stdin/stdout mechanism becomes apparent when we consider the operations of input and output redirection which are implemented in the C-shell. As the name suggests, redirection means that stdin and/or stdout are associated with some source/sink other than the terminal. Input redirection is accomplished using the '<' (less-than) character which is followed by the name of a file from which the input is to be extracted. Thus the command-line

```
% cat < input_to_cat
```

causes the contents of the file 'input_to_cat' to be used as input to the cat command. In this case, the effect is exactly the same as if

```
% cat input_to_cat
```

has been entered.

Output redirection is accomplished using the '>' (greater than) character, again followed by the name of a file into which the (standard) output of the command is to be directed. Thus

```
% cat > output_from_cat
```

will cause cat to read lines from the terminal (stdin is not redirected in this case) and copy them into the file 'output_from_cat'. Care must be exercised in using output redirection since one of the first things which will happen in the above example is that the file 'output_from_cat' will be clobbered. If the shell variable 'noclobber' is set (recommended for novices), then output will not be allowed to be redirected to an existing file. Thus, in the above example, if 'output_from_catálready existed, the shell would respond as follows:

```
% cat > output_from_cat

output_from_cat:  File exists
```

and the command would be aborted.

The standard output from a command can also be appended to a file using the two-character sequence '>>' (no intervening spaces). Thus

```
% cat >> existing_file
```

will append lines typed at the terminal to the end of 'existing_file'.

From time to time it is convenient to be able to throw away the standard output of a command. Unix/Linux systems have a special file called '/dev/null' which is ideally suited for this purpose. Output redirection to this file, as in:

```
verbose_command > /dev/null
```

will result in the stdout from the command disappearing without a trace.

**Pipes:** It is then often possible to combine commands (programs) on the command-line so that the standard output from one command is fed directly into the standard output of another. In this case we say that the output of the first command is *piped* into the input of the second. Here's an example:

```
% ls -1 | wc
10 10 82
```

The -1 option to `ls` tells 'ls' to list regular files and directories one per line. The command 'wc' (for word count) when invoked with no arguments, reads stdin until EOF is encountered and then prints three numbers: (1) the total number of lines in the input, (2) the total number of words in the input, and (3) the total number of characters in the input (in this case, 82). The pipe symbol '|' tells the shell to connect the standard output of 'ls' to the standard input of 'wc'. The entire `ls -1 | wc` construct is known as a `pipeline`, and in this case, the first number (i.e., 10) which appears on the standard output is simply the number of regular files and directories in the current directory.
Pipelines can be made as long as desired, and once you know a few Unix/Linux commands and have mastered the basics of the C-shell history mechanism, you can easily accomplish some fairly sophisticated tasks by building up multi-stage pipelines.

**grep**
`grep` (which loosely stands for (g)lobal search for (r)egular (e)xpression with (p)rint) has the following general syntax:

```
grep [options] regular_expression [file1 file2 ...]
```

Note that only the 'regular_expression' argument is required. Thus

```
% grep the
```

will read lines from stdin (normally the terminal) and echo only those lines which contain the string 'the'. If one or more file arguments are supplied along with the regexp, then grep will search those files for lines matching the regexp, and print the matching lines to standard output (again, normally the terminal). Thus

```
% grep the *
```

will print all the lines of all the regular files in the working directory which contain the string 'the'. Some of the options to `grep` are worth mentioning here. The first is '-i' which tells grep to ignore upper/lower cases when pattern-matching. Thus

```
% grep -i the text
```

will print all lines of the file 'text' which contain 'the' or 'The' or 'tHe' etc. Second, the '-v' option instructs grep to print all lines which *do not* match the pattern; thus

```
% grep -v the text
```

will print all lines of text which do not contain the string 'the'. Finally, the '-n' option tells grep to include a line number at the beginning of each line printed. Thus

```
% grep -in the text
```

will print, with line numbers, all lines of the file 'text' which contain the string 'the', 'The', 'tHe' etc. Note that multiple options can be specified with a *single* '-' followed by a string of option letters with no intervening blanks.
Here are a few slightly more complicated examples. Note that when supplying a regexp which contains characters such as '*', '?', '[', '!' ..., which are special to the shell, the regexp should be surrounded by single quotes to prevent shell interpretation of the shell characters. In fact, you won't go wrong by always enclosing the regexp in single quotes.

```
% grep '^.....$' file1
```

prints all lines of 'file1' which contain exactly 5 characters.

```
% grep 'a' file1 | grep 'b'
```

prints all lines of 'file1' which contain at least one 'a' and one 'b'. (Note the use of the pipe to stream the stdout from the first grep into the stdin of the second.)

```
% grep -v '^#' input > output
```

extracts all lines from file 'input' which `do not` have a '#' in the first column and writes them to file 'output'.
Pattern matching (searching for strings) using regular expressions is a powerful concept, but one which can be made even more useful with certain extensions. Many of these extensions are implemented in a relative of `grep` known as `egrep`. See the man page for `egrep` if you are interested.

**Using Quotes (' ', " ", and ` `):** Most shells, including the `csh` and the Bourne-shell, use the three different types of quotes found on a standard keyboard

| | |
|---|---|
| ' ' | known as forward quotes, single quotes, or just quotes, |
| " " | known as double quotes, |
| ` ` | known as backward quotes or just back-quotes |

for distinct purposes.

**Forward quotes: ' '** We have already encountered several examples of the use of forward quotes which inhibit shell evaluation of any and all special characters and/or constructs. Here's an example:

```
% set a=100
% echo $a
100

% set b=$a
% echo $b
100

% set b='$a'
% echo $b
$a
```

Note how in the final assignment, set b='$a', the $a is protected from evaluation by the single quotes. Single quotes are commonly used to assign a shell variable a value which contains whitespace, or to protect command arguments which contain characters special to the shell (see the discussion of grep for an example).

**Double quotes: " "** Double quotes function in much the same way as forward quotes, except that the shell 'looks inside' them and evaluates (a) any references to the values of shell variables, and (b) anything within back-quotes (see below). Example:

```
% set a=100
% echo $a
100

% set string="The value of a is $a"
% echo $string
The value of a is 10
```

**Backward quotes: ' '** The shell uses back-quotes to provide a powerful mechanism for capturing the standard output of a Unix/Linux command (or, more generally, a sequence of Unix/Linux commands) as a string which can then be assigned to a shell variable or used as an argument to another command. Specifically, when the shell encounters a string enclosed in back-quotes, it attempts to evaluate the string as a Unix/Linux command, precisely as if the string had been entered at a shell prompt, and returns the standard output of the command as a string. In effect, the output of the command is substituted for the string and the enclosing back-quotes. Here are a few simple examples:

```
% date
Thu Jan 20 21:16:22 PST 2011
% set thedate=`date`
% echo $thedate
Thu Jan 20 09:59:59 EST 2011
```

## 1.6 Summary of Basic Unix/Linux Commands

| Topic | Command | Example |
|---|---|---|
| List file names | ls | ls -l *.c, ls -a, ls -F, ls -alF |
| Move files or directories | mv | mv temp.txt newfile.txt |
| | | mv temp.txt ../new/list.txt |
| Copy files or directories | cp | cp temp.txt newfile.txt |
| | | cp temp.txt ../new/list.txt |
| Remove a file or directory | rm | rm temp.txt |
| | | rm -i temp.txt |
| | | rm -rf directory |
| Look the MANual pages for a command | man | man rm |
| The '-k' option searches man pages for keyword | | man -k xterm |
| Make a directory | mkdir | mkdir newdir |
| Remove a directory | rmdir | rmdir newdir |
| Change directory | cd | cd texdir |
| Print working directory | pwd | pwd |
| Send file to a printer | lpr, lp | lp -Pprintername filename |
| List content of file | cat | cat file1 |
| | more | more file1 |
| | less | less file1 |
| Print string or variable | echo | echo $USER |
| | | echo "hello, world" |
| To see list of recent commands | history | history |
| Set protection of a file | chmod | chmod 755 file |
| Set owner of a file | chown | chown smith file |
| Make a link (alias) to a path | ln | ln -s ~/classes/phys-317 phys-317 |
| Find out disk quota | quota | quota -v |
| Find out disk usage | du | du |
| Create archive file tarfile.tar from list of files (can be a directory) | tar -cvf | tar -cvf tarfile.tar list |
| Create gzipped archive file from list of files | tar -czvf | tar -czvf tarfile.tar.tgz list |
| Extracts files from archive file tarfile.tar | tar -xvf | tar -xvf tarfile.tar |
| Extract files from a gzipped tarfile | tar -xzvf | tar -xzvf tarfile.tar.gz |
| Zip filename (can be tar file) into compressed file filename.gz | gzip | gzip filename |
| Unzip filename from filename.gz | gunzip | gunzip filename.gz |
| Another file compression | bzip2 | bzip2 filename |
| Decompressing files | bunzip2 | bunzip2 filenem.bz2 |
| Convert text files to PostScript | enscript | enscript -o file.ps file |
| Format files for printing on a PostScript printer | a2ps | a2ps -o code.ps code.f |
| Printing and pagination filter for text files | pr | pr program.f90 > program.f90.pr |
| For transferring files between computers, use 'scp' (secure copy) or 'sftp' (secure ftp) | scp | scp yourname@host:file file |
| | | scp yourname@host:file . |
| | | scp file yourname@host:. |
| | | scp file yourname@host:file |
| | sftp | username@host |
| | | pwd, cd subdir, ls, !ls, put, get, quit |
| To log on remotely, the preferred protocol is 'ssh' | ssh | yourname@host |

Notes ........................................................................................................................................

Notes ..............................................................................................................................................

Notes ........................................................................................................................................

Notes ......................................................................................................................................