Physics 410: Unix

Please report all errors/typos. etc to choptuik@physics.ubc.ca

Last updated September 2000

Index

- Shell Variables
- Environment Variables
- Using C-shell Pattern Matching
- Using the C-shell History and Event Mechanisms
- Standard Input, Standard Output and Standard Error
- Input and Output Redirection
- Pipes
- Regular expressions and **grep**
- Using Quotes: (’ ’, " ", and ‘ ‘)
  - Forward quotes: ’ ’
  - Double quotes: " "
  - Backward quotes: ‘ ‘
- Job control

- Obsolescent commands
  - **telnet**

INTRODUCTION AND MOTIVATION

The main purpose of these notes is to get you familiar with the *interactive* use of Unix for day-to-day organizational and programming tasks. First, recall that Unix is an *operating system* (OS) 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 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, compile and run C and Fortran programs, etc. etc. As many of you are probably aware, various Unix vendors have written GUIs (graphical user interfaces) for their particular versions of Unix. As with similar systems on Macs and PCs, these GUIs largely eliminate the need to issue commands by providing intuitive visual metaphors for most common tasks. However, the command-line approach is still well worth mastering for a variety of reasons, including:

- *Portability:* All Unix systems support the command-line approach, and by sticking with standard features, what you learn on the SGI and Sun machines will be applicable on virtually all Unix systems
- *Speed:* Command-line interfaces minimize the amount of information which needs to be passed from machine to machine when working remotely. If you just want to accomplish a few quick tasks, nothing is more annoying than a sluggish GUI.
- *Power:* Commands can be extended and combined in a straightforward way. Defining new commands using shell programming facilities, or via C and Fortran, is also relatively easy.

When you type commands in Unix, you are actually interacting with the OS through a special program called a *shell* which provides a more user-friendly command-line interface than that defined by the basic Unix commands themselves. I recommend that you use the ‘‘C-shell’’ (**csh**) or the ‘‘tC-shell’’ (**tcsh**) for interactive use. Your SGI accounts are currently set up so that **tcsh** is your default interactive shell, and I recommend that, if possible, you also use the **tcsh** on your **physics** accounts. 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 shell prompt (denoted "% ") will appear in typewriter font. Here’s an example

```
% pwd
/usr2/people/matt
% date
Mon Sep  4 15:25:21 PDT 2000
%
```

FILES AND DIRECTORIES

I assume you are familiar with the notion of a hierarchical organization (tree structure) of files and directories which many modern operating systems employ. If you are not, refer to one of the Unix references or on-line tutorials which I have suggested. There are essentially only two types of files in Unix:

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

Absolute and relative pathnames, working directory: All Unix 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

```
/home5/choptuik/junk
```

refers to a file named ’junk’ which resides in a directory with absolute pathname

```
/home5/choptuik
```

which itself lives in directory

```
/home5
```

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* (hereafter abbreviated WD). The name of the working directory may be printed using the **pwd** command:

```
% pwd
/usr/people/matt
%
```

If you refer to a filename such as

```
foo
```

or a pathname such as

```
dir1/dir2/foo
```

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

```
/usr/people/matt
```

the two previous relative pathnames are identical to the absolute pathnames

```
/usr/people/matt/foo
/usr/people/dir1/dir2/foo
```

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

Home directories: Each user of a Unix 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 [working] directory) with no arguments will always take you to your home directory. On **physics.ubc.ca** you should see something like this

```
% cd
% pwd
/home5/choptuik
```

while on the SGIs (**sgi1.physics.ubc.ca** etc.) it will be something like

```
/usr/people/phys410
```

or

```
/d/sgi1/usr/people/phys410
```

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

```
/usr/people/matt
```

then

```
% cd ~
```

and

```
% cd ~/dir1/dir2
```

are identical to

```
% cd /usr/people/matt
```

and

```
% cd /usr/people/matt/dir1/dir2
```

respectively. (Note that the command **cd** changes the working directory.) The C-shell will also let you abbreviate other users' home directories by prepending a tilde to the user name. Thus, provided I have permission to change to phys410's home directory,

```
% cd ~phys410
```

will take me there.

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

```
% cd ~phys410/hw1
% pwd
/usr/people/phys410/hw1
% cd ..
% pwd
/usr/people/phys410
% cd .
% pwd
/usr/people/phys410
```

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. On most systems (including SGI and Linux systems), the length of a filename cannot exceed 255 characters. Any character except slash ('/)' (for obvious reasons) 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.c  (extension .c)
paper.tex  (extension .tex)
the.longextension (extension .longextension)
noextension (no extension)
```

Note that unlike some other operating systems, extensions are *not* required, and are not restricted to some fixed length (often 3 on other systems). In general, extensions are meaningful only to specific applications, or classes of applications, not to *all* applications. 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
```

The system makes it difficult for you to create a filename which starts with a minus. It is equally difficult to get rid of such a file, so be careful. 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.

COMMANDS OVERVIEW

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

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

where square brackets ('[...]') denote optional quantities. *Options* to Unix 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, a command such as **ls** or **cp** is usually the name of a file which is known to the system to be executable (see the discussion of **chmod** below). 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
```

(Note that the '$' mechanism is the standard way of *evaluating* shell variables and environment variables alike.) On the SGIs, the resulting output should look something like

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

Note that 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
helpme: Command not found.
```

The path variable is typically set in your '~/.login' file and/or (preferably) your '~/.cshrc' file.

IMPORTANT NOTE: On **physics.ubc.ca** you are **NOT** allowed to modify your '~/.cshrc' file; modify **~/.cshrc.user** instead, and note that all references to '~/.cshrc' below should be interpreted as references to '~/.cshrc.user', when used in the context of **physics.ubc.ca**.

Examine the file '~/.cshrc' in your SGI account. You should see a line like

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

which adds '$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 ~/bin)
```

will produce the same effect.

Control Characters: The following control characters typically have the following special meaning or uses within the C-shell. (If they don't, then your keyboard bindings are ''non-standard'' and you may wish to contact the system administrator about it.) 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" matt@laplace.physics.ubc.ca
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" matt@laplace.physics.ubc.ca
  This is a one line message.
  ^C
  (Interrupt -- one more to kill letter)
  ^C
  %
  ```

  Once more, if you try this example, you should notice that the control sequences are not explicitly echoed by the shell

- ^*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 as I will briefly discus in job control below.

Special Files: The following files, both 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. The existence of separate '~/.cshrc' and '~/.login' files was more useful in the days when interaction with Unix w typically via a single screen (dumb ASCII terminal) and machines were slower so start-up time of shells was an issue. When interacting with Unix 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. At this point it may not be clear to you why this works; if it isn't, you should try to figure it out after you have gone through these notes and possibly looked at the man page for **ls**.

Shell Aliases: As you will discover, the syntax of many Unix commands is quite complicated and furthermore, the ''bare-bones'' version of some commands is less than ideal for 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.

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

Define aliases for **rm**, **cp** and **mv** (see below) which will not clobber files without first asking you for explicit confirmation. Highly recommended for novices and experts alike.

```
% alias RM '/bin/rm'
% alias CP '/bin/cp'
% alias MV '/bin/mv'
```

Define aliases **RM**, **CP**, and **MV** which act like the ''bare'' Unix commands **rm**, **cp** and **mv** (i.e. which are *not* cautious). Use when you are sure you are about to do the correct thing: the presumption being that you have to think a little more to type the upper-case command. To see a list of all your current aliases, simply type

```
% aliases
```

Note that all of the preceding aliases (and a few more) are defined in a file '~/.aliases' in your SGI accounts. As configured, these aliases will be available in *all* interactive shells you start since

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

is in your '~/.cshrc'. (The 'source' command tells the shell to execute the commands in the file supplied as an argument). Although this is not a ''standardized'' approach, I commend it to you as a means of keeping your '~/.cshrc' relatively uncluttered if you define a lot of aliases.

BASIC COMMANDS

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

Getting Help or Information:

man

Use **man** (short for 'manual') to print information about a specific Unix 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 systems, is essentially an alias for **man -k**. Also note that you can use wildcards very similar to those used for filename matching in the shell in the keyword specification:

```
% man -k '*dir*'
```

Note the use of forward quotes in the last two examples; in the first instance the quotes are used to pass a "keyword phrase" to the man command, in the second example, the quotes prevent the '*' from being interpreted by the shell as a filename wildcard. It is instructive to examine the output of the two commands

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

and

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

and to understand *why* the outputs are different.

Output from **man** will typically look like

```
% man man
MAN(1)

NAME
     man - print entries from the on-line reference manuals; find manual
     entries by keyword

SYNOPSIS
     man [-cdwWtpr] [-M path] [-T macropackage] [section] title ...
     man [-M path] -k keyword ...
     man [-M path] -f filename ...

DESCRIPTION
     man locates and prints  . . .
                          .
                          .
                          .
```

for a specific command and,

```
% man -k 'working directory'
cd (1)                   - change working directory
cd (3Tcl)                - Change working directory
chdir, fchdir (2)        - change working directory
getcwd (3C)              - get pathname of current working directory
                         .
                         .
                         .
```

for a keyword-based search. Note that the output from **man -k ...** is a list of commands and brief synopses. You can then get detailed information about any specific command (say **cd** in the current example), with another man command:

```
% man cd
```

Communicating with Other Machines:

ssh (ssh1)

Use **ssh** to establish a secure (i.e. encrypypted) connection from one Unix machine to another. This is the basic mechanism we will use to (1) start a Unix shell on a remote machine and (2) execute one or more Unix commands on a remote machine.

Note: There are currently *two* major versions of the secure shell, Version 1 and Version 2. Since many installations (including the SGI machines), do not yet run Version 2, we will restrict our attention to Version 1. On systems such as **physics.ubc.ca**, which *are* running Version 2, you are strongly advised to alias **ssh** to **ssh1** (and likewise for the **scp**, **slogin** and **ssh-keygen** commmands) by adding the following line to your **~/.cshrc_user** or **~/.aliases** file

```
alias ssh        'ssh1'
alias slogin     'slogin1'
alias scp        'scp1'
alias ssh-keygen 'ssh-keygen1'
```

Re-emphasizing, define these aliases **ONLY** on those systems which have **ssh2** installed.

Typical usage of **ssh** is

```
% ssh sgi1.physics.ubc.ca -l matt
```

which will initiate a remote-login for user **matt** on the machine **sgi1.physics.ubc.ca**. The following commands are equivalent to the above invocation:

```
% ssh matt@sgi1.physics.ubc.ca
% slogin sgi1.physics.ubc.ca -l matt
% slogin matt@sgi1.physics.ubc.ca
```

If additional arguments are supplied to **ssh**, they are interpreted as commands to be executed remotely. In this case, control immediately returns to the invoking shell after completion (successful, or otherwise) of the command(s), as seen in the following examples:

```
sgi1% ssh matt@vnfe1.physics.ubc.ca date
Mon Sep  4 14:51:12 PDT 2000

sgi1% ssh matt@vnfe1.physics.ubc.ca 'pwd; date'
/home/matt
Mon Sep  4 14:51:24 PDT 2000

sgi1%
```

Gory Details: In contrast to many of the other commands described here, the behaviour of **ssh** depends crucially on the current *context* for the command, which, by convention, **ssh** stores as a number of files in the directory **~/.ssh** (i.e. as a number of files in a *directory* named **.ssh**, located in your home directory). If **~/.ssh** does not exist (which nominally means that you have yet to issue the **ssh** command from that specific account), it will automatically be created, and certain files within **~/.ssh** will be created and/or modified.

For example, assume that, as **choptuik@physics.ubc.ca**, I have never used the **ssh** command. However, I *can* and *do* login into **physics.ubc.ca** (as **choptuik**) via one of the X-terms in the computer lab, and start up a command shell (an **xterm**, e.g.). I can now establish a secure connection to my account on the Relativity SGI machines via **ssh** -- assuming that I have dutifully aliased **ssh** to **ssh1** as recommended above -- as follows:

```
physics% which ssh
ssh:     aliased to ssh1
```

```
physics% ssh matt@sgi1.physics.ubc.ca

Host key not found from the list of known hosts.
Are you sure you want to continue connecting (yes/no)? yes

Host 'sgi1.physics.ubc.ca' added to the list of known hosts.
Creating random seed file ~/.ssh/random_seed.  This may take a while.

matt@sgi1.physics.ubc.ca's password:

Last login: Mon Sep  4 11:15:22 2000 from laplace.physics.ubc.ca
*******************************************************************************
*   Welcome to sgi1.physics.ubc.ca                                            *
*                                                                             *
*   Report problems to matt@laplace.physics.ubc.ca                            *
*******************************************************************************
                               .
                               .
                               .
sgi1%
```

Note that I've added an occasional blank line to the above output to increase readibility. At this point I am connected to a **tcsh** running on **sgi1.physics.ubc.ca**, and I can now "work" (i.e. issue Unix commands) within a shell executing on that machine.

When I'm done my work on **sgi1**, I can use the **logout** (or **exit**) command

```
sgi1% logout
Connection to sgi1 closed.
physics%
```

to return to **physics**.

Assuming I've done the above, I now see that the directory **~/.ssh** *has* been created, and contains the files **known_hosts** and **random_seed**:

```
physics% cd ~/.ssh
physics% ls
known_hosts       random_seed
```

The purpose of the **known_hosts** file is to maintain a list of "Internet style addresses" (i.e. things like **sgi1.physics.ubc.ca**) to which I've previously **ssh**'ed. Thus, the message and prompt

```
Host key not found from the list of known hosts.
Are you sure you want to continue connecting (yes/no)?
```

to which I *must* answer 'yes' (literally!) -- should I wish to continue to connect -- will *not* appear upon subsequent **ssh** invocations to **sgi1.physics.ubc.ca**.

Thus, continuing with the above example, if I re-connect to **sgi1** from **physics**, I now see

```
physics% ssh matt@sgi1.physics.ubc.ca
matt@sgi1.physics.ubc.ca's password:
```

since **sgi1.physics.ubc.ca** has been deemed a "known", or "trusted", host, and has been recorded as such

in **~/.ssh/known_hosts** :

```
physics% cat ~/.ssh/known_hosts

sgi1.physics.ubc.ca 1024 37 124537240372930070636685397750631200970 70 ...
```

Refer to the **man** page on **ssh**, or **ssh1** for details on the makeup of **known_hosts** entries, and much more than you really need to know!

---

ssh-keygen (ssh-keygen1)

Use ssh-keygen to generate authentication information which can be used to enable password-free connection, command execution and file copying (via **scp**) from one Unix system to another.

Typical usage, continuing with the above example, and assuming that I am logged in as **choptuik@physics.ubc.ca** is

```
physics% which ssh-keygen
ssh-keygen:      aliased to ssh-keygen1

physics% ssh-keygen
Initializing random number generator...
Generating p:  ............++ (distance 176)
Generating q:  ................................++ (distance 534)
Computing the keys...
Testing the keys...
Key generation complete.
Enter file in which to save the key (/home/choptuik/.ssh/identity):
Enter passphrase:
Enter the same passphrase again:
Your identification has been saved in /home/choptuik/.ssh/identity.
Your public key is:
1024 37 124194660003566334173619711485715039312901856333949581811 ...
Your public key has been saved in /home/choptuik/.ssh/identity.pub
```

Note that the **ssh-keygen** command prompts you three times, namely:

```
Enter file in which to save the key (/home/choptuik/.ssh/identity):
Enter passphrase:
Enter the same passphrase again:
```

Be sure to select the default value each time by hitting "Enter" -- and nothing else!

My **~/.ssh** directory now contains new files, **identity** and **identity.pub**, as can seen in the following output:

```
physics% pwd
/home5/choptuik/.ssh
physics% ls
identity       identity.pub   known_hosts    random_seed
```

I can now use the information in **~/.ssh/identity.pub** to enable password-less access to my account(s) on any remote site which is also running **ssh1**. The basic idea is that each account maintains a database of

authorized keys in the file **~/.ssh/authorized_keys** .

In the above example, no such file exists yet, so if I **ssh** back to physics, I still get prompted for a password:

```
physics% ssh physics
Host key not found from the list of known hosts.
Are you sure you want to continue connecting (yes/no)? yes
Host 'physics' added to the list of known hosts.
choptuik@physics's password:
```

However, if I "create" the authorized keys database via

```
physics% cd .ssh
~/.ssh
physics% cp identity.pub authorized_keys
```

access to **choptuik@physics** is now password-less.

```
physics% ssh physics
Last login: Mon Sep  4 16:33:09 2000 from physics
Sun Microsystems Inc.   SunOS 5.6       Generic August 1997
==============================================================================
                                .
                                .
                                .
```

A more interesting use of this facility involves exporting a key to a remote machine. Thus, I now **ssh** to my account on **sgi1**, and add my **choptuik@physics.ubc.ca** key (the contents of **~/.ssh/identity.pub**) to **~/.ssh/authorized_keys** on the remote host.

Again, the initial **ssh** results in a prompt for a password:

```
physics% ssh matt@sgi1.physics.ubc.ca
matt@sgi1.physics.ubc.ca's password:
```

I now modify **~/.ssh/authorized_keys** using cut-and-paste and my favorite text editor so that it contains a *single line* which is identical to the contents of my **~/.ssh/identity.pub** on **physics**, as is verified by the following:

```
sgi1% cd .ssh
~/.ssh
sgi1% grep choptuik@physics authorized_keys
1024 37 124194660003566334173619711485715039312901856333949581811 ...

sgi1% exit
```

I can now **ssh** to **sgi1** from **physics** without specifying a password

```
physics% ssh matt@sgi1
Host key not found from the list of known hosts.
Are you sure you want to continue connecting (yes/no)? yes
Host 'sgi1' added to the list of known hosts.
Last login: Mon Sep  4 14:45:20 2000 from warp.physics.ubc.ca
******************************************************************************
```

```
*   Welcome to sgi1.physics.ubc.ca                                          *
                                .
                                .
                                .
```

Here, I've also illustrated the little twist that **ssh** distinguishes "aliased" hosts -- in this case **sgi1** is an alias (via DNS lookup) for **sgi1.physics.ubc.ca**. The use of such aliases is, as usual, a matter of convenience.

Clearly the mastery of **ssh** and **ssh-keygen** is a non-trivial matter. You are urged to contact the instructor **IMMEDIATELY** if you have any problems using these commands, or if you wish to see a strategy for maintaining and distributing a single *global* key list, which is handy if you compute on a large number of hosts.

ftp

Use **ftp** to establish a connection to another machine for the express purpose of copying files between the two machines. Here's an example illustrating how I might copy my '~/.ssh/authorized_keys' file from my SGI account to my **physics** account:

```
physics% pwd
/home5/choptuik/.ssh

physics% ftp sgi1.physics.ubc.ca
Connected to sgi1.physics.ubc.ca.
220 sgi1.physics.ubc.ca FTP server ready.
Name (sgi1.physics.ubc.ca:choptuik): matt
331 Password required for matt.
Password:
230 User matt logged in.
ftp> bin
200 Type set to I.
ftp> cd .ssh
250 CWD command successful.
ftp> get authorized_keys
200 PORT command successful.
150 Opening BINARY mode data connection for 'authorized_keys' (57454 bytes).
226 Transfer complete.
local: authorized_keys remote: authorized_keys
57454 bytes received in 0.0084 seconds (6685.03 Kbytes/s)
ftp> quit
221 Goodbye.
physics%
```

ftp has fairly extensive on-line help. Try

```
% ftp
ftp> help
```

as well as

```
ftp> help bin
ftp> help cd
ftp> help lcd
ftp> help put
ftp> help get
```

```
ftp> help prompt
ftp> help mget
```

to learn the basics. It is usually advisable to use ‘‘BINARY mode’’ to transfer files. Some installations support *anonymous ftp*. This means that *anyone* can ftp to the site using username ’anonymous’. In such cases you are usually requested to type your full name or e-mail address as your password. In most cases you will only be able to **get** (and not **put**) files from such sites.

Note: Many sites (such as **physics.ubc.ca**) do not allow *inbound* **ftp** connections. In such instances, **scp** provides a replacement, particularly for smaller files and/or directory hierarchies. Such sites do, however, often still permit *outbound* **ftp** requests, which, due to the bi-directional nature of the connection, also provides a replacement. In particular, as illustrated by the above example, you can use **ftp** from **physics.ubc.ca** *TO* **sgi1.physics.ubc.ca** to transfer files between **physics** and **sgi1**, in either direction.

Mail

I assume most of you will read and send your e-mail from a *single* machine/account which may not be **physics.ubc.ca** or one of the SGI machines. If you *do* wish to use one of the ‘‘course machines’’ for mail, I recommend that you use **pine** which provides a much friendlier user interface than **Mail** which is described briefly here. The advantage of knowing a bit about **Mail** is that it is almost universally available, and is good for firing-off short messages, or for sending material which has been pre-prepared in a file.

Here’s a quick example showing how to use Mail to send a message:

```
% Mail -s "this is the subject" choptuik@physics.ubc.ca
This is a one line test message.
^D
```

Note that multiple recipients can be specified on the command line. Another form involves redirection from a file.

```
% Mail -s "sending a file as a message" matt@laplace.physics.ubc.ca < message
```

sends the contents of file ’message’.

If you wish to read mail using **Mail**, type **Mail** (when you *have* mail) then type **help** to figure out how to continue. The **Mail** sub-commands:

- **h** to display message headers
- **p** to print the next message
- **s** to save a message
- **d** to delete a message
- **r** to reply to a message
- **q** to quit **Mail**

should get you going. Finally note that I advocate using the alias

```
% alias mail Mail
```

so that you never invoke the unfriendly ‘‘bare’’ **mail** command. Your accounts were initially set up so that this alias was automatically defined.

.forward

Whatever mailer you use, you should make sure that on every distinct home directory you own (recall that you have *one* home directory on **physics.ubc.ca** and *one* on the SGIs), there is a file called ’.forward’ which contains an e-mail address. That way, mail sent to your username at any machine will be automatically forwarded to the specified address. You can verify for yourself that my .forward on both **physics.ubc.ca** (’~choptuik/.forward’) and on the SGIs (’~matt/.forward’) contains

```
matt@laplace.physics.ubc.ca
```

Of course, your ’~/.forward’ should contain your own preferred e-mail address.

xterm

Use the **xterm** command to start a new shell which runs as an X-application, and thus displays on your current X-display (presumably your local console or X-term). For example, assume you have logged onto **physics** from one of the Computer Lab X-terms, and then, from there, have **ssh**-ed into **sgi1**.

```
physics% ssh matt@sgi1.physics.ubc.ca
                .
                .
                .
sgi1%
```

On **sgi1** you can start an **xterm**

```
sgi1% xterm &
```

At this point, a new shell window should appear on your X-term. *Note that the window is displayed on the X-term, but that the shell running in that window is executing on **sgi1***. Also note that it is conventional to run **xterm**s in the background (by appending an ‘&‘ to the command-line---see job control) so that control returns to the shell prompt, and you can continue to type commands.

Some vendors have "souped-up" (and typically somewhat propietary) versions of **xterm**. On the SGIs for example, the "souped-up" version is called **winterm**.

Finally, note that this description of **xterm** is intended mostly to familiarize you with what is arguably the quintessential X-application. You will probably rarely find yourself *explicitly* issuing **xterm** commands as illustrated above---more commonly the **xterm**s you use will be started automatically when you login, or will be created from a menu-selection.

Logging out:

logout

Type **logout** to terminate a Unix session:

```
% logout
```

If there are suspended jobs (see job control below), you will get a warning message, *and you will not be logged out*.

```
% logout
There are stopped jobs
```

If you then type **logout** a second time (with no intervening command), the system assumes you have decided you don't care about the suspended jobs, and will log you out. Alternatively, you can deal with the suspended jobs, and then **logout**.

exit

The **exit** command is actually a C-shell "built-in" which, as the name suggests, exits the shell. For login shells (including shells running in an **xterm**), **exit** is effectively a synonym for **logout**.

Creating, Manipulating and Viewing Files (including Directories):

vi *or* **emacs**

It is absolutely crucial that you become facile with one of these standard Unix editors and because each editor itself has many sub-commands I refer you to suitably general texts on Unix, or specific books on **vi** or **emacs** for detailed information. Either of these two editors will more than suffice for creation, modification and viewing of text files at the level required for this course.

more

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

```
% more ~/.cshrc
umask 022

limit -h coredumpsize > /dev/null

setenv LANG          C

setenv HOMEMWC       /d/laplace/usr2/people/matt


--More--(1%)
```

In this case I have executed the **more** command in a shell window containing only a few lines (i.e. my pages are short). The

```
--More--(1%)
```

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. Note that the

output from **man** is typically piped through **more**.

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 (see below). Typical usage is

```
lpr file_to_be_printed
```

More details concerning printing will be supplied as the course progresses.

cd *and* **pwd**

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

```
% cd
% pwd
/usr2/people/matt
% cd ~; pwd
/usr2/people/matt
% cd /tmp: pwd
/tmp
% cd ~phys410; pwd
/usr/people/phys410
% cd ..; pwd
/usr/people
% cd phys410; pwd
/usr/people/phys410
```

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

```
% cd ..
```

takes you up one level (closer to the root) 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 ~phys410
% ls
cmd*   hw1/   util/
%
```

Note that the file 'cmd' is marked executable while 'hw1' and 'util' are directories. To see hidden files and directories, use the **-a** option:

```
% cd ~phys410; ls -a
./            .aliases      .login        cmd*
../           .cshrc        .profile      hw1/
.Xauthority   .exrc         .ssh/         util/
```

and to view the files in ‘‘long’’ format, use **-l**:

```
% cd ~phys410; ls -l
total 1
total 16
-rwxr-xr-x    1 phys410  user          35 Sep  4 16:00 cmd*
drwxr-xr-x    4 phys410  user          31 Sep  4 10:19 hw1/
drwxr-xr-x    2 phys410  user        4096 Sep  4 12:05 util/
```

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.
- 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, ’cmd’ is a regular file, with read, write and execute permissions enabled for the owner (user ’phys410’) and read and execute permissions enabled for member’s of group ’user’ and all other users. ’hw1’ and ’util’ are seen to be directories with the same permissions. 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 long file listing, the next column list the number of links to this file (if you don’t know what a *link* is, don’t worry) 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 are listed. Finally, note that the **-R** option will recursively list sub-directories:

```
% cd ~phys410; pwd
/usr/people/phys410
% ls -R
cmd*   hw1/   util/

./hw1:
q4/  q5/

./hw1/q4:
input

./hw1/q5:
First.c   first.f

./util:
Makefile      p410f.f       pp2d_demo.h   sphplot.h
fsleep.c      p410fsa.f     pp2d_input    tfsleep.c
fsleep.h      pp2d_demo.c   sphplot.c
```

Note how each sub-listing begins with the relative pathname to the directory followed by a colon. For kicks, you might want to try

```
% cd /
% ls -R
```

which will list essentially all the files on the system which you can read (have read permission for). Type '^C' when you get bored.

mkdir

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

```
% cd ~
% mkdir tempdir
% cd tempdir; pwd
/usr/people/matt/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
/usr/people/matt/a/long/way/down
```

In this case, the **mkdir** command made the directories

```
/usr/people/matt/a   /usr/people/matt/a/long    /usr/people/matt/a/long/way
```

and, finally

```
/usr/people/matt/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:

```
% cd ~phys410; ls
cmd*   hw1/   util/
% cd ..; pwd
/usr/people
% cp -r phys410 /tmp
% cd /tmp/phys410; ls -a
./            .aliases      .login        cmd*
../           .cshrc        .profile      hw1/
.Xauthority   .exrc         .ssh/         util/
```

Study the above example carefully to make sure you understand what happened when the command

```
% cp -r phys410 /tmp
```

was issued. In brief, the directory '/tmp/phys410' was created and all contents (including hidden files) of '/usr/people/phys410' were recursively copied into that new directory: sub-directories of '/tmp/phys410' were automatically created when required.

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
```

while 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 lev1/
```

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've removed a file in Unix 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 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 the SGI machines, your defaults should be such that you can do anything you want to a file you've 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 file_or_directory_to_hide
```

which removes all permissions from 'group' and 'others', effectively hiding the file/directory,

```
% chmod a+x executable_file
```

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

```
% chmod u-w file_to_write_protect
```

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 '-'.

scp

Use **scp** (whose syntax is an extension of **cp**) to copy files or hierarchies from one Unix system to another. **scp** is part of the **ssh** distribution and uses the same authentication for password-less access

described in the **ssh** section above.

For example, assume I am logged into **sgi1** and that 'choptuik@physics.ubc.ca:~/.ssh/authorized_keys' contains a line duplicating the contents of 'matt@sgi1.physics.ubc.ca:~/.ssh/identity.pub'. Then the command

```
sgi1% scp ~/.cshrc choptuik@physics.ubc.ca:/tmp/sgi_cshrc
```

will copy my '~/.cshrc' file on **sgi1** to the file '/tmp/sgi_cshrc' on **physics**. Similarly, the command

```
sgi1% scp choptuik@physics.ubc.ca:~/.cshrc /tmp/sun_cshrc
```

will copy my '~/.cshrc' file on **physics** to the file '/tmp/sun_cshrc' on **sgi1**.

Be very careful using **scp**, particularly since there is no **-i** (cautious) option. Also note that there *is* a **-r** option for remote-copying entire hierarchies.

On some machines, a default mode for **scp** includes a "statistics trace" which can be useful if you are **scp**ing large files over slow connections. Printing of these statistics may be disabled by setting the **SSH_NO_SCP_STATS** environment variable. For instance

```
% setenv SSH_NO_SCP_STATS on
```

will do the trick, and you may wish to have such a line in your **~/.cshrc** file(s).

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 **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. See discussion above.
- `prompt`: The current shell prompt---what the shell displays when it’s expecting input.
- `cwd`: Contains the name of the (current) working directory.
- `term`: Defines the terminal type. If your terminal is acting strangely

  ```
  % 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’re using
- `ignoreeof`: When set, will disable shell ’logout’ when ^D is typed.

Environment Variables: Unix 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
/d/laplace/usr2/people/matt
% echo $HOME
/d/laplace/usr2/people/matt
```

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 commands (applications). The HOME environment variable, which stores the full pathname of your home directory, provides a canonical example.

Following is a list of some of standard environment variables with their functions:

- HOME: Stores home directory;

  ```
  cd $HOME/dir1
  ```

  is equivalent to

  ```
  cd ~/dir1
  ```

- PRINTER: Defines default printer for use with **lpr**, **netscape** etc. Check your '~/.cshrc' file on the SGIs for a line which sets this variable.

- DISPLAY: Tells X-applications which display (screen) to use for output. Generally set on remote shells so that output appears on a local screen. In the past, this was typically accomplished manually, or in a user's '~/.cshrc' or '~/.login'.

  ssh now handles apporpriate setting of DISPLAY automatically. Notify the instructor if you have any problems apparently related to mis-settings of DISPLAY.

Using C-shell Pattern Matching: The C-shell provides facilities which allow you to concisely refer to one or more files whose names 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]   (Example) Matches any single character contained in the
        specified range (the match set)---in this case lower-case 'a'
        through lower-case 'z'
[^a-z]  (Example) Matches any single character not contained
        in the specified range
```

Match sets may also be specified explicitly, as in

```
[02468]
```

Examples:

```
ls ??
```

lists all regular (not hidden) files and directories whose names contain precisely two characters.

```
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 aren't 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 **tcsh** (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 your initial set-ups on the Linux and Sgi machines:

```
!!       Repeat the previous command line
!21      (Example) Repeat command line number 21
!a       (Example) Repeat most recently issued command line which started
         with an 'a'.  Use an initial sub-string of length > 1 for
         more specificity.
!?b      (Example) Repeat most recently issued command line which contains
         'b'; any string of characters can be used after the '?'
```

(Note that Unix users often refer to an exclamation point ('!') as ''bang''.) The following constructs are useful for recycling command arguments:

```
!*       Evaluates to all of the arguments supplied to the previous
         command
!$       Evaluates to the last argument supplied to the previous command
```

Finally, the following construct is useful for correcting small typos in command lines:

```
^old_string^new_string
```

This changes the *first* occurrence of *old_string* in the previous command to *new_string* then executes the modified command. Example:

```
% cp foo /usr2/poeple/matt
Cannot create /usr2/poeple/matt
No such file or directory
```

```
% ^oe^eo
cp foo /usr2/people/matt
```

Note that whenever any of the above constructs are used, the shell echoes the effective command before it is executed.

Standard Input, Standard Output and Standard Error: A typical Unix 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 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 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_cat' already 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 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: Part of the "Unix programming philosophy" is to keep input and output to and from commands in "machine-readable" form: this usually means keeping the input and output simple, structured and devoid of extraneous information which, while informative to humans, is likely to be a nuisance for other programs. Thus, rather than writing a command which produces output such as:

```
% pgm_wrong
Time = 0.0 seconds  Force = 6.0 Newtons
Time = 1.0 seconds  Force = 6.1 Newtons
Time = 2.0 seconds  Force = 6.2 Newtons
```

we write one which produces

```
% pgm_right
0.0   6.0
1.0   6.1
2.0   6.2
```

The advantage of this approach is that 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 (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 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.

Regular Expressions and grep: Regular expressions may be formally defined as those character strings which are recognized (accepted) by *finite state automata*. If you haven't studied automata theory, this definition won't be of much use, so for our purposes we will define regular expressions (or *regexps* for short) as specifications for rather general *patterns* which we will wish to detect, usually in the contents of files. Although there are similarities in the Unix specification of regexps to C-shell wildcards (see above), there are important differences as well, so be careful. We begin with regular expressions which match a single character:

```
a         (Example) Matches 'a', any character other than the special
          characters: . * [ ] \ ^ or $ may be used as is
\*        (Example) Matches the single character '*'.  Note that
          '\' is the "backslash" character.  A backslash may be
          used to "escape" any of the special characters listed above
          (including backslash itself)
.         Matches ANY single character.
[abc]     (Example) Matches any one of 'a', 'b' or 'c'.
[^abc]    (Example) Matches any character which ISN'T an 'a', 'b'
          or 'c'.
[a-z]     (Example) Matches any character in the inclusive range
          'a' through 'z'.
[^a-z]    (Example) Matches any character NOT in the inclusive range
          'a' through 'z'.
^         Matches the beginning of a line.
$         Matches the end of a line.
```

Multiple-character regexps may then be built up as follows:

```
ahgfh     (Example) Matches the string 'ahgfh'.  Any string of
          specific characters (including escaped special characters)
          may be specified in this fashion.
a*        (Example) Matches zero or more occurrences of the character
          'a'.  Any single character expression (except start and
          end of line) followed by a '*' will match zero or more
          occurrences of that particular sequence.
.*        Matches an arbitrary string of characters.
```

All of this is may be a bit confusing, so it is best to consider the use of regular expressions in the context of the Unix **grep** command.

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 case 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 (not counting the "newline" at the end of each line):

```
% 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, quotes
    " " ->  Known as double quotes
    ‘ ‘ ->  Known as backward quotes, 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 100
```

Backward quotes: ‘ ‘ The shell uses back-quotes to provide a powerful mechanism for capturing the standard output of a Unix command (or, more generally, a sequence of Unix 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 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
Mon Sep  4 16:17:02 PDT 2000

% set thedate=‘date‘
% echo $thedate
Mon Sep 4 16:17:14 PDT 2000

% which true
/usr/bin/true

% file ‘which true‘
/usr/bin/true:  /sbin/sh script text

% more ‘which true‘
#!/sbin/sh
#Tag 3840
#       Copyright (c) 1984, 1986, 1987, 1988, 1989 AT&T
#         All Rights Reserved

#       THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T
#       The copyright notice above does not evidence any
#       actual or intended publication of such source code.

#ident  "@(#)true:true.sh       1.4"
#ident  "$Revision: 1.1.1.1 $"

% file ‘which true‘ ‘which false‘
/usr/bin/true:  /sbin/sh script text
/usr/bin/false:  /bin/sh script text
```

Note that the **file** command attempts to guess what type of contents its file arguments contain and **which** reports full path names for commands which are supplied as arguments. Observe that in the last example, multiple back-quoting constructs are used on a single command line.

Finally, here’s an example illustrating that back-quote substitution is *enabled* for strings within double quotes, but *disabled* for strings within single quotes:

```
% set var1="The current date is ‘date‘"
% echo $var1
The current date is Mon Sep 4 16:18:22 PDT 2000

% set var2=’The current date is ‘date‘’
% echo $var2
The current date is ‘date‘
```

Job Control: Unix is a multi-tasking operating system: at any given time, the system is effectively running many distinct processes (commands) simultaneously (of course, if the machine only has one CPU, only one process can run at a specific time, so this simultaneity is somewhat of an illusion). Even within a single shell, it is possible to run several different commands at the same time. *Job control* refers to the shell facilities for managing how these different processes are run. It should be noted that job control is arguably less important in the current age of windowing systems than it used to be, since one

can now simply use multiple shell windows to manage several concurrently running tasks.

Commands issued from the command-line normally run in the *foreground*. This generally means that the command ‘‘takes over’’ standard input and standard output (the terminal), and, in particular, the command must complete before you can type additional commands to the shell. If, however, the command line is terminated with an ampersand: ’&’, the job is run in the *background* and you can *immediately* type new commands while the command executes. Example:

```
% grep the huge_file > grep_output &
[1] 1299
```

In this example, the shell responds with a ’[1]’ which identifies the task at the shell level, and a ’1299’ (the process id) which identifies the task at the system level. You can continue to type commands while the **grep** job runs in the background. At some point **grep** will finish, and the next time you type ’Enter’ (or ’Return’), the shell will inform you that the job has completed:

```
[1]    Done   grep the huge_file > grep_output
```

The following sequence illustrates another way to run the same job in the background:

```
% grep the huge_file > grep_output
^Z
Stopped
% bg
[1] grep the huge_file > grep_output &
```

Here, typing ’^Z’ while the command is running in the foreground stops (suspends) the job, the shell command **bg** restarts it in the foreground. You can see which jobs are running or stopped by using the shell **jobs** command.

```
% jobs
[1] + Stopped    grep the huge_file > grep_output
[2]   Running    other_command
```

Use

```
% fg %1
```

to make run the job labeled ’[1]’ (which may either be stopped or running in the background), run in the foreground. You can **kill** a job using its job number (%1, %2, etc.)

```
% kill %1
[1]  Terminated    grep the huge_file > grep_output
```

You can also **kill** a job using its process ID (PID) which you can obtain using the Unix **ps** command. See the man pages for **ps** and **kill** for more details. On many Unix systems, including the SGIs, there is a **killall** command, which allow you to kill processes by name. Finally, the shell will complain if you try to **logout** or **exit** the shell when one or more jobs are stopped. Either explicitly kill the jobs (or let them finish up if that’s appropriate) or type **logout** or **exit** again to ignore the warning, kill all stopped jobs, and exit.

OBSELESCENT COMMANDS

The following command has largely been superseded and/or rendered superflous by the **ssh** command. It is described here primarily for "compatibility" with previous versions of the course.

telnet

Use **telnet** to establish a connection with another system (not necessarily Unix). Typical usage is

```
% telnet physics.ubc.ca
Trying 142.103.236.1...
Connected to physics.ubc.ca.
Escape character is '^]'.

The programs and data stored on this system are lawfully available only to
authorized users.  Unauthorized access to any program or data on this system
is a criminal offense.

This system may be monitored at any time for the purpose of ensuring system
integrity.  Report any unusual activity with your account or data to the
system administrators (sysadmin@physics.ubc.ca)
================================================================================
++++++++  As of January 1, 2000 all cleartext logins to physics and mach will
+ NOTE +  be disallowed.  This means that normal telnet and ftp will not work.
+ NOTE +  You will have to use SRP enabled clients or SSH.  Please see
+ NOTE +  http://www.physics.ubc.ca/ComputerFacilities/secure-login.html
++++++++  for more information.
================================================================================

Warning:  This session is not using secure authentication.


UNIX(r) System V Release 4.0 (physics)

login:
```

at which point you login as usual. If a session established this way ‘‘hangs’’ on you, or you otherwise appear to have lost control of your terminal, try typing

```
% ^]
```

(i.e. ’Control-]’) at which point you should get a

```
telnet>
```

prompt. Type **quit** at the the prompt to exit **telnet**.