Physics 410: Unix

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

Last updated September 6, 2004

Index

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*) that 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, one 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 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 Linux and Sun machines will be applicable on virtually all Unix systems
- *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.
- *Speed:* Command-line interfaces minimize the amount of information that needs to be passed from machine to machine when working remotely. If you just want to accomplish a few quick tasks, few things are more annoying than a sluggish GUI.

The versions of Unix implemented by specific vendors (or programming teams) typically have specific names. In particular, on the **lnx** machines you will be using the **Linux** flavour of Unix, originally coded in large part by Linus Torvald for PCs, and now widely distributed by many different companies and organizations. You will also be using one of Sun Microsystems' implementations of Unix, called SunOS, on **physics**.

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 improved "C-shell", **tcsh**, for interactive use. Your **lnx** accounts are initially set up so that **tcsh** is your default interactive shell, and I suggest that, if possible, you also use **tcsh** on your **physics** accounts.

One of the improvements of **tcsh** upon the original C-shell, **csh**, was the introduction 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, commands that 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
/home/matt
% date
Mon Sep  6 10:40:40 PDT 2004
%
```

If you are going through these notes on-line, then you should have at least one active shell running in which to type sample commands. I will often refer to a window in which a shell is executing as the *terminal*.

FILES AND DIRECTORIES

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

- *Plain files*: that 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 that serve as 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* that begin with / and that describe the path down the file tree to the file in question. Thus

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

refers to a file named **junk** that resides in a directory with absolute pathname

```
/home5/choptuik
```

that itself lives in directory

```
/home5
```

that 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
/home/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

```
/home/matt
```

the two previous relative pathnames are identical to the absolute pathnames

```
/home/matt/foo
/home/matt/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* that 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
/home2/phys410
```

while on the **lnx** machines (**lnx1.physics.ubc.ca** etc.) it will be something like

```
/home/phys410
```

or

```
/d/lnx1/home/phys410
```

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

```
/home/matt
```

then

```
% cd ~
```

and

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

are identical to

```
% cd /home/matt
```

and

```
% cd /home/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
/home/phys410/hw1
% cd ..
% pwd
/home/phys410
% cd .
% pwd
/home/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 Sun 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 that 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 with 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 in contrast to 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
```

You *can* embed blanks in Unix filenames, but it is not recommended.

Unix generally makes it difficult for you to create a filename that starts with a minus. It is also non-trivial 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 ...
```

On Linux systems, many commands also accept options that are longer than a single character; by convention, these options are preceded by *two* minus signs as in:

```
% ls --color=auto -CF
```

Arguments are typically names of files or directories or other text strings that *do not* start with **-** (or **--**). 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 that 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 that 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 that 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 **lnx** machines, the resulting output should look something like

```
. /usr/local/bin /usr/bin /bin /usr/local/pgi/linux86/bin /usr/local/PGI/bin /usr/X11R6/bin /u
```

Note that the **.** in the output indicates that the working directory is in your path. The order in which path-components (first **.**, then /**usr**/**bin**, then /**bin**, 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 for you in a special system file each time a shell starts up (/**etc**/**csh.cshrc** on the **lnx** machines), and it is conventional to modify the default setting via a **set** command in your ~/**.cshrc** file.

IMPORTANT NOTE: See the discussion below on the special structure of the ~/**.cshrc** file for new undergraduate accounts on **physics.ubc.ca** and **lts.physics.ubc.ca**, and observe that you should NOT modify ~/**.cshrc** *per se* on those systems, if you have a new account. Instead all references to ~/**.cshrc** below should be interpreted as references to ~/**.cshrc.solaris** or ~/**.cshrc.linux**, when used in the context of **physics.ubc.ca** or **lts1.physics.ubc.ca**, respectively.

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

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

that 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 that 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**) that is reading input from the terminal. Here's an example using **Mail** on the **lnx** machines:

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

  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** prompts for an optional list of addresses to which the message is to be carbon-copied, but other commands, such as **cat**, will not echo anything. In almost all cases, however, you should be presented with a command prompt, once your have typed ^D. Also, by default, a C-shell exits when it encounters EOF, so if you type ^D at a shell prompt, you may find that you are logged out from the terminal session. If you don't like this behaviour (I don't), put the following line in your ~/**.cshrc**:

  ```
  set ignoreeof
  ```

- *^C: Interrupt*. Type ^C to kill (stop in a non-restartable fashion) commands (processes) that you have started from the command-line. This is particularly useful for commands that are taking much longer to execute or producing much more output to the terminal than you had anticipated.

- *^Z: Suspend*. Type ^Z to suspend (stop in a restartable fashion) commands that you have started from the shell. It is often convenient to temporarily halt execution of a command as will be discussed in job control below.
- *^V: Escape Special Characters:* Type ^V in order to "escape" (protect from shell evaluation) certain other control characters and special keys.

  Example 1: Search for [TAB] characters in file **foo** using **grep**:

  ```
  % grep '^V[TAB]' foo
  ```

  Example 2: Removing "Carriage returns" (**^M**) using vi

  ```
  :%s/^V^M//g
  ```

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 was typically via a single screen (dumb ASCII terminal) and machines were slower, so startup time of shells was an issue. When interacting with Unix via a windowing system, it is easy to start an interactive shell that 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 startup commands in ~/**.cshrc** instead. By default, your accounts on the **lnx** machines do *not* contain a ~/**.login** file.

Notes on ~/**.cshrc on physics and lts1:** Because your accounts on **physics** and **lts1** share a common home directory, the shell startp process must be slightly more involved to take into account the fact that the shell may be running on one of two distinct architectures (machine/OS combinations).

For recently created undergraduate accounts, ~/**.csrhc** should contain the following, and should *NOT* be modified:

```
#################################################################
#         .cshrc file
#################################################################
#
# PATH ##########################################################
#
set ARCH=`/bin/arch`
#
if ( ${?ARCH} != 0 ) then
 if ( ${ARCH} == "sun4" ) then
   #echo $ARCH is sun4
   source .cshrc.solaris
 else
   #echo $ARCH is linux
   source .cshrc.linux
 endif
endif
```

Upon startup this file simply **source**s (executes the commands contained in) either ~/**.cshrc.solaris** or ~/**.cshrc.linux**, depending on the architecture on which the shell is running. Startup commands that would normally go in ~/**.cshrc** should be placed in **.cshrc.linux** and/or **.csrhc.solaris** as appropriate.

For the case of older undergraduate accounts, you should find that your home directory contains a file ~/**.csrhc.user**. You should copy that file to **.cshrc.solaris**, then replace the contents of ~/**.cshrc.user** with the those of the ~/**.cshrc** file listed above. Finally, you should copy the contents of ~/**.csrhc** from your **lnx** account to ~/**.cshrc.linux** on your **physics/lts1** account.

Students who have a graduate account can follow the same instructions to effect context-sensitive sourcing of an appropriate startup file, except that the ~/**.cshrc.user** file referred to in the previous paragraph will now simply be ~/**.cshrc**.

Note on ~/**.login on physics and lts1:** All users should add the following lines to the *beginning* of their ~/**.login** on **physics/lts1** (but *not* on the **lnx** machines).

```
if ( `hostname` != physics ) then
   exit
endif
```

This will disable execution of the ~/**.login** commands, which are somewhat specialized for the Sun environment, unless you are logging into **physics**.

Special Files (continued). Note that files whose name begins with a period (**.**) are called *hidden files* 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* that 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 illustrate the basic idea; see the **tcsh** documentation (**man tcsh**) for more complete information:

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

provides an alias for the **ls** command that 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 special characters; it is good defensive programming to always include them.

The following lines define aliases for **rm**, **cp** and **mv** (see below) that will not clobber files without first asking you for explicit confirmation. They are highly recommended for novices and experts alike.

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

The following lines define aliases **RM**, **CP**, and **MV** that act like the "bare" Unix commands **rm**, **cp** and **mv** (i.e. that are *not* cautious). Use them 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.

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

To see a list of all your current aliases, simply type

```
% alias
```

Note that all of the preceding aliases (and a few more) are defined in a file ~/**.aliases** in your **lnx** 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.

You can view the initial contents of your ~/**.cshrc** and ~/**.aliases** files on the **lnx** machines by clicking on the links below:

- **.cshrc**
- **.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 that 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. Examples of **man** invocations include:

```
% man man
```

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

```
% man cp
```

for information on **cp** and

```
% man -k compiler
```

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

Output from **man** will typically look like

```
% man man
man(1)                                                      man(1)

NAME
       man - format and display the on-line manual pages
       manpath - determine user's search path for man pages

SYNOPSIS
       man  [-acdfFhkKtwW]  [-m  system]  [-p  string]  [-C  con
       fig_file] [-M path] [-P pager] [-S section_list] [section]
       name ...

DESCRIPTION
       man  formats  and displays the on-line manual pages.  This
       version knows about the MANPATH and (MAN)PAGER environment
       variables, so you can have your own set(s) of personal man
                        .
                        .
                        .
```

for a specific command and,

```
% man -k 'compiler'
cccp, cpp (1)        - The GNU C-Compatible Compiler Preprocessor.
clisp (1)            - Common Lisp language interpreter and compiler
compile_et (1)       - error table compiler
diagnostics (1)      - Perl compiler pragma to force verbose warning diagnostics
f4rpcgen (1)         - an RPC protocol compiler
                        .
                        .
                        .
```

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 **cpp** in the current example), with another **man** command:

```
% man cpp
```

Communicating with Other Machines:

ssh

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

Note: There are currently *two* major protocol versions of the secure shell, Version 1 and Version 2. The software installed on the **lnx** machines supports both versions. However, protocol Version 1 has security flaws and is not supported on many systems on the internet. Consequently, the discussion that follows is limited to **ssh** Version 2, and is specific to that version.

Typical usage of **ssh** is

```
% ssh lnx1.physics.ubc.ca -l matt
```

which will initiate a remote-login for user **matt** on the machine **lnx1.physics.ubc.ca**. The following commands are equivalent to the above invocation:

```
% ssh matt@lnx1.physics.ubc.ca
% slogin lnx1.physics.ubc.ca -l matt
% slogin matt@lnx1.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:

```
lnx1% ssh matt@vnfe1.physics.ubc.ca date
Mon Sep  6 10:46:06 PDT 2004

lnx1% ssh matt@vnfe1.physics.ubc.ca 'pwd; date'
/home/matt
Mon Sep  6 10:46:17 PDT 2004

lnx1%
```

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 **matt@lnx1.physics.ubc.ca**, I have never used the **ssh** command. However, I *can* and *do* login into **lnx1.physics.ubc.ca** (as **matt**) using the machine's console in the computer lab, and start up a command shell. I can now establish a secure connection to my account on **physics.ubc.ca** via **ssh** as follows:

```
lnx1% ssh choptuik@physics.ubc.ca

Last login: Mon Sep  6 09:49:32 2004 from lnx1.physics.ub
To print one-sided use "lpr -Zsimplex file".
Undergraduate students will need to arrange for a quota.
-----------------------------------------------------------------------------
  type "more /etc/motd"      on physics to re-read this message.
  type "more /etc/motd.full" on physics to read the complete message file
=============================================================================
     Authorized uses only.  All activity may be monitored and reported.
=============================================================================
                            .
                            .
                            .
physics%
```

Note that I've added an occasional blank line to the above output to increase legibility. At this point I am connected to a **tcsh** running on **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 **physics**, I can use the **logout** (or **exit**) command

```
physics% logout
Connection to physics.ubc.ca closed.
lnx1%
```

to return to **lnx1**.

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

```
lnx1% cd ~/.ssh
lnx1% ls
known_hosts
```

The purpose of the **known_hosts** file is to maintain a list of *hostnames* (i.e. things like **physics.ubc.ca**), and identification information, to which I've previously **ssh**'ed. Refer to the **man** page on **ssh** (on the **lnx** machines) for full details on this command.

---

ssh-keygen

Use ssh-keygen to generate authentication information that can be used to enable password-free connection, command execution and file copying (via **scp**) from one Unix system to another (assuming both systems have Secure Shell Version 2 software installed). Note that there are two types of keys supported by the **ssh**: **RSA** and **DSA**. For the purposes of this course, either type will suffice, but for specificity, we will consider the **RSA** case.

Typical usage, continuing with the above example, and assuming that I am logged in as **matt@lnx1.physics.ubc.ca** is

```
lnx1% ssh-keygen -t rsa

Generating public/private rsa key pair.
Enter file in which to save the key (/home/matt/.ssh/id_rsa):
Created directory '/home/matt/.ssh'.
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Your identification has been saved in /home/matt/.ssh/id_rsa.
Your public key has been saved in /home/matt/.ssh/id_rsa.pub.
The key fingerprint is:
ef:f6:04:bd:b4:55:04:01:83:4b:08:db:76:82:a8:26 matt@lnx1.physics.ubc.ca
```

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

```
Enter file in which to save the key (/home/matt/.ssh/id_rsa):
Enter passphrase (empty for no passphrase):
Enter 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, **id_rsa** and **id_rsa.pub**, as can seen in the following output:

```
lnx1% pwd
/home/matt/.ssh

lnx1% ls
id_rsa  id_rsa.pub  known_hosts
```

I can now use the information in ~/**.ssh**/**identity.pub** to enable password-less access to my account(s) on any remote site that is also running **ssh** (Version 2). The basic idea is that each account maintains a database of authorized keys in the file ~/**.ssh**/**authorized_keys**. When a **ssh** request from some *userid* at some *hostname* is received, **ssh** checks the database to see whether a connection from *userid@hostname* should be initiated without entry of a password.

In the above example, no such file exists yet, so if I **ssh** *from* **lnx1** *to* **lnx1**, I still get prompted for a password:

```
lnx1% ssh lnx1
Warning: Permanently added 'lnx1,142.103.238.211' (RSA) to the list of known hosts.
matt@lnx1's password:
```

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

```
lnx1% cd .ssh
~/.ssh
lnx1% cp id_rsa.pub authorized_keys
```

access to **matt@lnx1** from **matt@lnx1** is now password-less.

```
lnx1% ssh lnx1
Last login: Thu Aug 28 11:39:18 2004 from lnx1.physics.ubc.ca
                                 .
                                 .
                                 .
```

A more interesting use of this facility involves exporting a key to a remote machine. Thus, I now **ssh** to my account on **physics**, and add my **matt@lnx1.physics.ubc.ca** public key (the contents of ~/**.ssh**/**id_rsa.pub** on my **lnx** account) to ~/**.ssh**/**authorized_keys** on the remote host.

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

```
lnx1% ssh choptuik@physics.ubc.ca
choptuik@physics.ubc.ca's password:
```

I now modify ~/**.ssh**/**authorized_keys** using, for example, cut-and-paste and my favorite text editor so that one of its lines is *identical* to the contents of my ~/**.ssh**/**id_rsa.pub** on **lnx1**, as is verified by the following:

```
physics% cd .ssh
~/.ssh
physics% grep matt@lnx1 authorized_keys
ssh-rsa AAAAB3NzaC1yc2EAAAABIwAAAIEAsLBXIvI8qmrAnChYLi34xWg ...

physics% exit
```

I can now **ssh** to **physics** from **lnx1** without being asked for a password

```
lnx1% ssh choptuik@physics

Warning: Permanently added 'physics,142.103.236.11' (RSA) to the list of known hosts
Last login: Mon Sep  6 10:48:44 2004 from lnx1.physics.ub
To print one-sided use "lpr -Zsimplex file".
Undergraduate students will need to arrange for a quota.
                        .
                        .
                        .
```

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

Once you understand how the password-less access facility outlined above works, a little thought will convince you that if you duplicate ~/**.ssh**/**id_rsa**, ~/**.ssh**/**id_rsa.pub** and ~/**.ssh**/**authorized_keys** across *all* machines on which you compute, then, assuming that **authorized_keys** contains the contents of **id_rsa.pub** as a single line, you will be able to **ssh** between any and all of the machines that you use without being prompted for a password. This is the strategy that I advocate you employ.

Finally, experience shows that the use of **ssh**, **ssh-keygen** and **scp** can sometimes be confusing to the uninitiated. You are urged to contact me immediately if you have any problems using these commands, since their mastery is a key component of this part of the course.

ftp

Use **ftp** to establish a connection to another machine for the express purpose of copying files (particularly very large files) between the two machines. Here's an example illustrating how I might copy my ~/**.ssh**/**authorized_keys**, ~/**.ssh**/**id_rsa** and ~/**.ssh**/**id_rsa.pub** files file from my **lnx** account to my account on **vnfe1.physics.ubc.ca**:

```
lnx1% pwd
/home/matt/.ssh

lnx1% ftp vnfe1.physics.ubc.ca
Connected to vnfe1.physics.ubc.ca.
220 vnfe1.physics.ubc.ca FTP server (Version wu-2.6.1(1) Thu Nov 29 18:08:28 MST 2001) ready.
530 Please login with USER and PASS.
530 Please login with USER and PASS.
KERBEROS_V4 rejected as an authentication type
Name (vnfe1.physics.ubc.ca:matt): matt

331 Password required for matt.
Password:
230 User matt logged in.
Remote system type is UNIX.
Using binary mode to transfer files.

ftp> bin
200 Type set to I.

ftp> cd .ssh
250 CWD command successful.

ftp> prompt
Interactive mode off.

ftp> mput authorized_keys id_rsa id_rsa.pub
local: authorized_keys remote: authorized_keys
200 PORT command successful.
150 Opening BINARY mode data connection for authorized_keys.
226 Transfer complete.
55031 bytes sent in 0.000479 secs (1.1e+05 Kbytes/sec)
local: id_rsa remote: id_rsa
200 PORT command successful.
150 Opening BINARY mode data connection for id_rsa.
226 Transfer complete.
883 bytes sent in 6e-05 secs (1.4e+04 Kbytes/sec)
local: id_rsa.pub remote: id_rsa.pub
200 PORT command successful.
150 Opening BINARY mode data connection for id_rsa.pub.
226 Transfer complete.
235 bytes sent in 5.2e-05 secs (4.4e+03 Kbytes/sec)

ftp> quit
221-You have transferred 56149 bytes in 3 files.
221-Total traffic for this session was 56918 bytes in 3 transfers.
221-Thank you for using the FTP service on vnfe1.physics.ubc.ca.
```

```
221 Goodbye.
```

```
lnx1%
```

ftp has a rudimentary on-line help facility. Type

```
% ftp
ftp> help
```

to get a full list of available commands, for which *brief* synopses are available via

```
ftp> help bin
```

etc. A useful subset of basic commands is

```
cd, lcd, put, get, prompt, mget, mput, exit
```

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 the 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, including the **lnx** machines as well as **physics** have disabled *inbound* **ftp** due to security concerns. In such cases, *outbound* **ftp** may still be supported, and if so, as long as the destination machine accepts inbound **ftp** connections, files can be transferred to and fro, due to the bi-directional nature of **ftp**.

In all cases, **scp** provides a secure replacement for **ftp**. However, the security currently comes at a performance cost due to the encrypted nature of the **scp** connection, and thus on slow networks, or for large files, there is still an argument for using **ftp** at times.

Mail

I assume most of you will read and send your e-mail from a *single* machine/account, probably via your account on **physics.ubc.ca**. (Using the **lnx** machines as your "home base" for reading mail is *not* recommended.)

If you do not yet have a favorite Unix mail system, then I recommend that you use **pine**, which provides a much friendlier user interface than **Mail** that is described briefly here (**pine** has an extensive on-line help facility). 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 that 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.
Cc:
%
```

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 on the **lnx** machines are initially set up so that this alias is defined for you.

.forward

Whichever mailer you use, you should make sure that on every distinct home directory you own (again note that you have *one* home directory on **physics.ubc.ca** and *one* on the **lnx** machines), there is a file called **.forward** that contains your e-mail address on the machine on which you normally read your mail. 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 **lnx** machines (~**matt/.forward**) contains

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

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

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" that, as the name suggests, exits the shell. For login shells, **exit** is effectively a synonym for **logout**.

Creating, Manipulating and Viewing Files (including Directories):

vi or **emacs (xemacs)**

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 references on **vi** or **emacs** for detailed information. **emacs** has an extensive on-line help facility, as does the Linux implementation of **vi**. Also note that a complete on-line *User's Guide* for **xemacs** is available in PDF form **HERE**.

Another option for straightforward editing tasks is **pico**, which is sufficiently simple that the on-line help facility should suffice to quickly make you expert in its usage.

Either **vi** or **emacs** 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 /usr/share/dict/words
Aarhus
Aaron
Ababa
aback
abaft
abandon
abandoned
abandoning
--More--(0%)
```

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--(0%)
```

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. If no options are passed to **lpr**, files are sent to the system-default printer, or to the printer specified by your **PRINTER** environment variable (see below). Typical usage is

```
lpr file_to_be_printed
```

The default printer on the **lnx** machines is the HP LaserJet 4100 in Hennings 205 and, by default, printing will be two-sided (duplex). Should you need to make one-sided hardcopy, print the file from using the **-Psimplex** option:

```
lnx1% lpr -Psimplex file
```

Note that this last form of the **lpr** command is specific to the **physics.ubc.ca** machines.

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 (again note the use of semi-colons to separate distinct Unix commands issued on the same line):

```
% cd
% pwd
/home/matt
% cd ~; pwd
/home/matt
% cd /tmp: pwd
/tmp
% cd ~phys410; pwd
/home/phys410
% cd ..; pwd
/home
% cd phys410; pwd
/home/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. On Linux systems, I advocate the use of the alias

```
% alias ls 'ls --color=auto -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), (2) list in columns (the **-C** option), and (3) color-code the output, again according to the type of the file. Example:

```
% cd ~jdoe
% ls
cmd*  dir1/  dir2/ tmp/
%
```

Note that the file **cmd** is marked executable while **dir1**, **dir2** and **tmp** are directories. To see hidden files and directories, use the **-a** option:

```
% cd ~jdoe; ls -a
./  ../  .Xauthority  .aliases  .cshrc  .ssh/  cmd*  dir1/  dir2/ tmp/
```

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

```
% cd ~jdoe; ls -l
total 12
-rwxr-xr-x    1 jdoe     phys410        67 Aug 30  2001 cmd*
drwxr-xr-x    4 jdoe     phys410      4096 Aug 30  2001 dir1/
drwxr-xr-x    2 jdoe     phys410      4096 Aug 30  2001 dir2/
drwxr-xr-x    2 jdoe     phys410      4096 Sep  5  2001 tmp/
```

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 that 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 **jdoe**) and read and execute permissions enabled for members of group **phys410** and all other users. **dir1**, **dir2** and **tmp** 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 lists 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 that directory are listed. Finally, note that the **-R** option will recursively list sub-directories:

```
% cd ~jdoe; pwd
/home/jdoe

% ls -R
.:
cmd*  dir1/  dir2/

./dir1:
file_1  subdir1/  subdir2/

./dir1/subdir1:
file_2

./dir1/subdir2:
file_3

./dir2:
file_4

./tmp:
```

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
/home/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
/home/matt/a/long/way/down
```

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

```
/home/matt/a   /home/matt/a/long    /home/matt/a/long/way
```

and, finally

```
/home/matt/a/long/way/down
```

cp

Use **cp** to (1) make a copy of a file, (2) copy one or more files to a directory, or (3) 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 ~jdoe; ls -a
./  ../  .Xauthority  .aliases  .cshrc  .ssh/  cmd*  dir1/  dir2/  tmp/
% cd ..; pwd
/home
% cp -r jdoe /tmp
cp: jdoe/.Xauthority: Permission denied
cp: jdoe/.ssh/known_hosts: Permission denied
cp: jdoe/.ssh/random_seed: Permission denied

% cd /tmp/jdoe; ls -a
./  ../  .aliases  .cshrc  .ssh/  cmd*  dir1/  dir2/  tmp/
```

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

```
% cp -r jdoe /tmp
```

was issued. In brief, the directory **/tmp/phys410** was created and all contents of **/usr/people/phys410** (including hidden files) *for which I had read permission* were recursively copied into that new directory: sub-directories of **/tmp/jdoe** were automatically created as 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 particularly 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 that you can modify using the **umask** command. (See **man umask** for more information).

On the **lnx** 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 that 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 **-**. You can also set permissions *absolutely* using an **=**, for example

```
% chmod a=r file_for_all_to_read
```

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 **lnx1** and that **choptuik@physics.ubc.ca:~/.ssh/authorized_keys** contains a line duplicating the contents of **matt@lnx1.physics.ubc.ca:~/.ssh/id_rsa.pub**. Then the command

```
lnx1% scp ~/.cshrc choptuik@physics.ubc.ca:~/lnx_cshrc
```

will copy my **~/.cshrc** file on **lnx1** to the file **~/lnx_cshrc** on **physics**. Similarly, the command

```
lnx1% scp choptuik@physics.ubc.ca:~/.cshrc ~/sun_cshrc
```

will copy my **~/.cshrc** file on **physics** to the file **~/sun_cshrc** on **lnx1**.

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" that 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), as it is by default on your **lnx** accounts.

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 of your 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 that 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 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.

Individual environment variables may be viewed using **printenv** or **echo**:

```
% printenv HOME
/home/matt
% echo $HOME
/home/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) that are initiated from that shell. For this reason, environment variables are widely used to communicate information to Unix commands (applications).

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

- **HOME**: Stores the user's home directory;

  ```
  cd $HOME/dir1
  ```

  is equivalent to

  ```
  cd ~/dir1
  ```

- **PRINTER**: Defines default printer for use with **lpr**, **netscape** etc.

- **DISPLAY**: Tells X-applications what 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 appropriate setting of **DISPLAY** automatically. Notify me if you have any problems apparently related to mis-settings of **DISPLAY**.

Using C-shell Pattern Matching: The C-shell provides facilities that 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 (including no
        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 than simply commands. Type

```
% history
```

(which I personally alias as **hi**) 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 in your initial set-up on the **lnx** machines.

```
!!       Repeat the previous command line

!21      (Example) Repeat command line number 21

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

!?b      (Example) Repeat most recently issued command line
         that 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 /hmoe/matt
cp: cannot create regular file '/hmoe/matt': No such file or directory
% ^mo^om
cp foo /home/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, job, task, application) 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 that 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 that 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 that 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, 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
```

had 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 that 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** that 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 that 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 that 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 input 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) that 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 interactively building up multi-stage pipelines.

Regular Expressions and grep: Regular expressions may be formally defined as those character strings that 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 as specifications for rather general *patterns* that we will wish to detect, usually in the contents of files. Although there are similarities in the Unix specification of regular expressions to C-shell wildcards (see above), there are important differences as well, so be careful. We begin with regular expressions that 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 that 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 regular expressions 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 that contain the string 'the'. If one or more file arguments are supplied along with the regular expression, then **grep** will search those files for lines matching the regular expression, 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 that 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** that contain 'the' or 'The' or 'tHe' etc. Second, the **-v** option instructs grep to print all lines that *do not* match the pattern; thus

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

will print all lines of **text** that *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** that 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 regular expression that contains characters such as '*', '?', '[', '!' ..., that are special to the shell, the regular expression 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 regular expression in single quotes.

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

prints all lines of **file1** that contain exactly 5 characters (not counting the "newline" at the end of each line):

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

prints all lines of **file1** that 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** that *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 that 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 C-shell 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 that 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 that contains whitespace, or to protect command arguments that 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 that 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  6 10:55:49 PDT 2004

% set thedate=`date`
% echo $thedate
Mon Sep 6 10:55:56 PDT 2004

% which true
/bin/true

% file `which true`
/bin/true: ELF 32-bit LSB executable, Intel 80386 ...

% file `which true` `which false`
/bin/true:  ELF 32-bit LSB executable, Intel 80386 ...
/bin/false: ELF 32-bit LSB executable, Intel 80386 ...
```

Note that the **file** command attempts to guess what type of contents its file arguments contain and **which** reports full path names for commands that 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 6 10:56:13 PDT 2004

% 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]' that identifies the task at the shell level, and a '1299' (the process id) that 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
Suspended
% 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]' (that 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 Linux, there is a **killall** command, which allows 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.

Another useful, though Linux-specific, command is **pstree**, which shows processes currently running on the host machine in the form of a tree. If you want to limit the output to your own processes (and not, for example, **root**'s), use

```
% pstree -u your_userid
```

BASIC SHELL PROGRAMMING

For the novice user a Unix shell can be viewed primarily as a command interpreter. However, shells are actually fully functional programming languages and it is extremely useful to know at least a little about shell programming, also known as writing *shell scripts*, for the following reasons (not an exhaustive list!):

1. Scripts can be used to customize or extend Unix commands in a more powerful and robust fashion than the aliasing mechanism discussed above.
2. Scripts can be used to automate sequences of Unix commands, with the possibility of changing one or more of the arguments to one or more of the commands. If you find yourself typing a series of commands several times, it takes very little time to create a script to accomplish the task, after which the execution of a single command does the trick. This has the added bonus that the script *per se* provides documentation for the job you are doing.
3. Many tasks that are cumbersome to perform in the context of a general purpose programming language, such as **C** or **Fortran**, are easy to accomplish using a script. This particularly applies to issues involving file and directory manipulation, or the processing of output from a number of programs.

Although it is entirely possible to write **tcsh**-scripts, the Bourne-shell, **sh**, (or on Linux and many other Unix systems, **bash**, for Bourne-again-shell) tends to be used more often for scripting, and thus will be our focus here. Time constraints preclude anything but a cursory overview of shell programming; if you wish to become a wizard of this particular craft, I suggest you consult the classic text, *The UNIX Programming Environment*, by Kernighan and Pike, cited in the following as reference [1]. In addition, should you find yourself in need of *complex* scripts, you may wish to consider using **perl**, which is an extremely powerful scripting language that has become very popular in the Unix community over the past decade or so.

We start with a very simple example. Consider the problem of "swapping" the names of two files, which arises more often in practice than one might expect, and which cannot be accomplished with a standard Unix command. Assuming that no file **t** exists in the working directory, the command sequence

```
% mv a t
% mv b a
% mv t b
```

will exchange the names of files **a** and **b**. Building on this sequence, here's a script called **swap** that, naturally enough, "swaps" the names of an *arbitrary* pair of files:

```
#! /bin/sh

# Bare-bones script to swap names of two files

# Usage: swap file1 file2

mv $1  t
mv $2 $1
mv t  $2
```

The first line of the script

```
#! /bin/sh
```

is an important bit of Unix magic that tells the shell that when the name of the file containing the script is used as a command, the shell should start up a *new* shell (in this case a Bourne-shell, **sh**) and execute the remaining contents of the script in the context of that new shell. *Every* shell-script that you write should start with this incantation.

Lines that begin with a hash ("number sign") **#** (excluding the magic first line) such as

```
# Bare-bones script to swap names of two files

# Usage: swap file1 file2
```

are comments, and are ignored by the shell.

The final three lines of the script

```
mv $1  t
mv $2 $1
mv t  $2
```

do all the work. The constructs **$1** and **$2** evaluate to the first and second arguments, respectively, which are supplied to the script. In general, one can access the first nine arguments of a script using **$1**, **$2**, ..., **$9**, and, if more than nine arguments need to be parsed (!), using **${10}$**, **${11}**, etc. If a specific argument is missing, the corresponding construct will evaluate to the null string, i.e. to "nothing".

Having created a file called **swap** containing the above lines, I set execute permission on the file with the **chmod** command

```
% chmod a+x swap
% ls -l swap
-rwxr-xr-x    1 phys410  phys410       114 Aug 31  2001 swap*
```

and the script is ready to use:

```
% ls
f1  f2  swap*
% cat f1
This is the first file.
% cat f2
This is the second file.
% swap f1 f2
% cat f1
This is the second file.
% cat f2
This is the first file.
```

When developing and debugging a shell program, it is often very useful to enable "tracing" of the script. This is done by adding the **-x** option to the header line:

```
#! /bin/sh -x
```

Having made this modification, I now see the following output when I invoke **swap** a second time:

```
% swap f1 f2
+ mv f1 t
+ mv f2 f1
+ mv t f2
```

Note how each command in the script is echoed to standard error (with a **+** prepended) as it is executed. Also observe that the **mv** command used in this instance is the "bare bones" version; i.e. any aliases that I have defined for use in interaction with **tcsh** will *not* be in effect since the commands are being executed by a *different* shell.

Although **swap** as coded above is reasonably functional, it is not very robust and can potentially generate undesired "side-effects" if used incorrectly. Observe, for example, what happens when the script is invoked without any arguments (tracing has now been disabled by removing the **-x** option in the header)

```
mv: missing file argument
Try `mv --help' for more information.
mv: missing file arguments
Try `mv --help' for more information.
mv: missing file argument
Try `mv --help' for more information.
```

or, worse, with one argument

```
% swap f1
mv: missing file argument
Try `mv --help' for more information.
mv: missing file argument
Try `mv --help' for more information.
% ls
f2  swap*  t
```

Here's a second version of **swap** that fixes several of the shortcomings of the naive version, and that also illustrates a few more shell programming features:

```
#! /bin/sh

# Improved version of script to swap names of two files

# Set shell variable 'P' to name of script
P=`basename $0`

# Set shell variable 't' to name of temporary file
t=.swap.tempfile.3141

# Usage function
usage () {
cat << END
usage: $P file1 file2

       Swaps filenames of file1 and file2
END
exit 1
}

# Function that is invoked if temporary file already exists
```

```
t_exists () {
cat << END
$P: Temporary file '$t' exists.
$P: Remove it explicitly before executing this script.

/bin/rm -f $t
END
exit 1
}

# Function that checks that its (first) argument is an
# existing file
check_file () {
if [ ! -f $1 ]; then
   echo "$P: File '$1' does not exist"
   error="yes"
fi
}

# Argument parsing---script requires exactly 2 arguments
case $# in
2) file1=$1; file2=$2 ;;
*) usage;;
esac

# Check that the arguments refer to existing files
check_file $1
check_file $2

# Bail out if either or both arguments are invalid
test "X${error}" = X || exit

# Ensure that temporary file doesn't already exist
test -f $t && t_exists

# Do the swap
mv $file1 $t
mv $file2 $file1
mv $t     $file2

# Normal exit, return 'success' exit status
exit 0
```

Let us examine this new version of **swap** in detail.

As the comment indicates, the command

```
# Set shell variable 'P' to name of script
P=`basename $0`
```

sets the shell variable **P** to the name of the script, i.e. to **swap** in this case. This happens as follows. First, **$0** is a special shell-script variable that always evaluates to the invocation name of the script---i.e. what the user actually typed in order to execute the script. Second, as **man** tells us, the **basename** command deletes any prefix ending in / from its argument and prints the result on the standard output. Third, the backquotes around the **basename** invocation capture the standard output of the command, which is then assigned to the shell variable **P** via the assignment statement.

Note the syntax for setting shell variables in **sh**, which is slightly different than that for **tcsh** variables:

```
% var=value
```

There is no **set** in this case, but, again, *there can be no white space before or after the equals sign.*

We use **basename** here so that if someone invokes our script using its full path name, perhaps

```
% /home/phys410/shellpgm/ex2/swap f1 f2
```

the shell variable **P** will still be assigned the value **swap**. The value of **P** is subsequently used in diagnostic messages, to make the origin of the messages clear to the user. Use of this mechanism can save some typing if one is writing a script that prints many such messages. In addition, if the script is subsequently used as a basis for a *new* shell program, a minimum of changes (perhaps none) are necessary in order that the new script output the "correct" diagnostics.

The next assignment sets the shell variable **t** to the name of a temporary file that, under normal circumstances, should never exist in the directory in which **swap** is executed. This isn't the most bullet-proof of strategies, but it's better than using **t** itself for the name for the temporary file!

```
# Set shell variable 't' to name of temporary file
t=.swap.tempfile.3141
```

The next section of code defines a shell function, called **usage**, which can be invoked from anywhere in the script. When called, the function will print a message to standard output informing the user of the proper usage of the command, and then exit (stop execution of the script).

```
# Usage function
usage () {
cat << END
usage: $P file1 file2

        Swaps filenames of file1 and file2
END
exit 1
}
```

The general form of a function definition is

```
routinename () {
   command
   command
   ...
}
```

The parentheses pair after *routinename* tells the shell that a function is being defined, while the braces enclose the body of the function.

Within the **usage** routine appears the construct

```
cat << END
   ...
END
```

known as a "here document". Here documents can be used anywhere in a script to provide "in-place" input for the standard input of a command. You can refer to the **man** page on **sh** for full details, but the idea and mechanics are simple. To provide "in-place" input to an arbitrary command, append **<< END** after the command name and all of its arguments. *Be certain that there is no white space after the token* **END**. Subsequent lines are then the standard input to the command. A line that *exactly* matches the string **END** (i.e. **grep '^END$'** succeeds) signals end-of-file (so be sure you have such a line in your script!). Again, beware of leading or trailing white space in the end-of-file marker. Finally, note that the string **END** is arbitrary; you can use essentially any string you wish as long as you use the identical string in both contexts. **END** is simply my convention.

An interesting and useful feature of here-documents is that they are partially interpreted by the shell *before* being fed into their destination command. In particular, shell-variable-evaluations

```
$var
```

are executed, as are

```
`command [arguments]`
```

constructs. Thus, when the **usage** function is executed, the message

```
usage: swap file1 file2

        Swaps filenames of file1 and file2
```

will appear on standard output, after which the execution of

```
exit 1
```

will return control to the invoking shell. Here, the argument to the **exit** command is an exit code indicating a completion status for the script. Since there is generally only one way for a command to succeed, but often many ways it can fail, a exit status of **0** indicates success in Unix, while any non-zero value (**1** in this case), indicates failure.

All Unix commands return such codes (scripts that terminate without an explicit **exit**, implicitly return success to the invoking shell) and they can be used in the context of shell-control structures such as **if**, **while** and **until** statements.

The function **t_exists** is very similar in construction to **usage**, and is used in the unlikely event that a file named **.swap.tempfile.3141** *does* exist in the directory in which the script is invoked.

```
# Function that is invoked if temporary file already exists
t_exists () {
cat << END
$P: Temporary file '$t' exists.
```

```
$P: Remove it explicitly before executing this script.

/bin/rm -f $t
END
exit 1
```

Function **check_file** illustrates the use of function arguments, as well as the shell **if** statement.

```
# Function that checks that its (first) argument is an
# existing file
check_file () {
if [ ! -f $1 ]; then
   echo "$P: File '$1' does not exist"
   error="yes"
fi
```

As with arguments to the script itself, function arguments are accessed *positionally*, via **$1, $2, ...** . Note, then, that the evaluation of **$1**, for example, depends crucially on context (or scope): within a function, **$1** evaluates to the first argument to the routine, while outside of any function it evaluates to the first argument to the script.

For our purposes, a suitably general form of the shell **if** statement is

```
if command a; then
   commands 1
elif command b; then
   commands 2
elif command c; then
   commands 3
   ...
else command last
   commands n
fi
```

All clauses apart from the first are optional, as is apparent from the **if** statement in the **check_file** routine. The evaluation of the **if** statement begins with the execution of *command a*. If this command succeeds (returns exit status 0), then *commands 1* are executed (commands must appear on separate lines, or be separated by semicolons) and control then passes to the command following the end of the **if** statement (i.e. after the **fi** token). Otherwise, *command b* is executed; if it succeeds, *commands 2* are performed, otherwise *command c* is executed, and so on.

The **if** statement in our **check_file** routine

```
if [ ! -f $1 ]; then
   echo "$P: File '$1' does not exist"
   error="yes"
fi
```

uses the Unix **test** command, for which **[** is essentially an alias (the **]** is "syntactic-sugar" and does nothing but make the expression "look right"). Thus an equivalent form is

```
if test ! -f $1; then
   echo "$P: File '$1' does not exist"
   error="yes"
fi
```

test accepts a general expression *expr* as an argument, evaluates *expr* and, if its value is true, sets a zero exit status (success); otherwise, a non-zero exit status (failure) is set. **test** accepts many different options for performing a variety of tests on files and directories, and implements a fairly complete set of logical operations such as negation, or, and, tests for string equality/non-equality, integer equal-to, greater-than, less-than etc.; see **man test** for full details.

In the current case, the **-f $1** option returns true if the first argument to the routine is an existing regular file (i.e. not a directory or other type of special file). The **!** is the negation operator, so the overall **test** command returns success (true) if the first argument is *not* an existing regular file.

The next section of code introduces the shell **case** statement:

```
# Argument parsing---script requires exactly 2 arguments
case $# in
2) file1=$1; file2=$2 ;;
*) usage;;
esac
```

A general **case** statement looks like

```
case word in
pattern) commands ;;
```

```
pattern) commands ;;
...
esac
```

Starting from the top, and using essentially the same pattern-matching rules used for filename matching, the **case** statement compares *word* to each *pattern* in turn, until it finds a match. When a match is found the corresponding commands (and only those commands) are executed, after which control passes to the statement following the end of the **case** statement (i.e. after the **esac** token). Note that the commands associated with each case must be terminated with a double semi-colon.

In our current example, we match on the built-in shell variable **$#**, which evaluates to the number of arguments that were supplied to the shell. The first set of actions

```
2) file1=$1; file2=$2 ;;
```

is evaluated if precisely two arguments have been supplied. If the script has been invoked with anything *but* two arguments, **$#** is then matched against *****, which will *always* succeed; i.e.

```
*) usage ;;
```

serves as a "default" case, and the **usage** function will thus be called if we've used an incorrect number of arguments.

Using the **check_file** function, the script then ensures that each argument names an existing regular file:

```
# Check that the arguments refer to existing files
check_file $file1
check_file $file2
```

Note that if either or both of the checks fail, then the shell variable **error** (all variables are global in a shell script unless explicitly declared **local**, see **man sh** for more information) will be set to **yes**. The calls to **check_file** are followed by a command list that tests whether **error** has been set, and exits the script if it has:

```
# Bail out if either or both arguments are invalid
test "X${error}" = X || exit
```

The expression

```
"X${error}" = X
```

illustrates a little shell trick that tests whether a shell variable has been defined. If **error** has been set to **yes** by **check_file**, then **"X${error}"** evaluates to **Xyes**; otherwise it evaluates to **X**. The binary operator **||** (double pipe) can be used between any two Unix commands (or, more generally, pipelines):

```
command 1 || command 2
```

and has the following semantics: *command 1* is executed, and *if and only if* the command *fails* (returns a non-zero exit status), *command 2* is executed. Thus, the sequence is equivalent to

```
if [ ! command 1 ]; then
   command 2
fi
```

Similarly, the next piece of the script

```
# Ensure that temporary file doesn't already exist
test -f $t && t_exists
```

illustrates the use of the binary operator **&&** (double ampersand), which also can be used between any two commands:

```
command 1 && command 2
```

In this case *command 1* is executed, and *if and only if* the command *succeeds* (returns a 0 exit status), *command 2* is executed. Thus, an equivalent form is

```
if [ command 1 ]; then
   command 2
fi
```

In the current example, if the temporary file **.swap.temp.3141** *does* exist, the function **t_exists** is called to print the diagnostic message and exit.

Finally, if we've made it past all of the error-checking, it's time to actually swap the filenames, and have the script return a "success" exit status to the invoking environment:

```
# Do the swap
mv $file1 $t
```

```
mv $file2 $file1
mv $t     $file2

# Normal exit, return 'success' code
exit 0
```

We can now test our improved version of swap, exercising in particular all of the error-checking features that have been incorporated. Here again is a contents-listing of the directory containing the script:

```
% ls
f1  f2  swap*
% more f1 f2
::::::::::::::
f1
::::::::::::::
This is the first file.
::::::::::::::
f2
::::::::::::::
This is the second file.
```

We start with a no-argument invocation:

```
% swap
usage: swap file1 file2

       Swaps filenames of file1 and file2
```

followed by single-argument execution:

```
% swap f1
usage: swap file1 file2

       Swaps filenames of file1 and file2
```

In both cases **swap** dutifully prints the usage message to standard output as desired.

We now invoke **swap** in a "normal" fashion, and verify that it is working properly:

```
% swap f1 f2
% more f1 f2
::::::::::::::
f1
::::::::::::::
This is the second file.
::::::::::::::
f2
::::::::::::::
This is the first file.
```

Supplying **swap** with two arguments that are *not* names of files in the working directory results in appropriate error messages:

```
% swap a1 a2
swap: File 'a1' does not exist
swap: File 'a2' does not exist
```

as does an invocation where *one* of the arguments is invalid:

```
% swap a1 f2
swap: File 'a1' does not exist
```

Finally, after (perversely) creating **.swap.tempfile.3141**,

```
% touch .swap.tempfile.3141
```

execution of **swap** with valid arguments triggers the **t_exists** routine:

```
swap: Temporary file '.swap.tempfile.3141' exists.
swap: Remove it explicitly before executing this script.

/bin/rm -f .swap.tempfile.3141
```

Removing the file as instructed, the script once again silently performs its job:

```
% /bin/rm -f .swap.tempfile.3141
```

```
% swap f1 f2
```

We will conclude our whirlwind tour of shell programming with a description of a few more control structures, some additional niceties concerning shell variable evaluation, and a glimpse at a command useful for writing scripts that "interact" with the user.

The shell provides three structures for looping. The first is a **for** loop:

```
for var in word list; do
   commands
done
```

Here, for each word (token) in *word list*, the *commands* in the body of the loop are executed, with the shell variable *var* being set to each word in turn. As usual, an example makes the semantics clear:

```
% cat for-example
#! /bin/sh

# Illustrates shell 'for' loop

for i in foo bar 'foo bar '; do
        echo "i -> $i"
done

% for-example
i -> foo
i -> bar
i -> foo bar
```

Note that a "word" can contain white space if it has been quoted, as is the case for **'foo bar '**

In many instances, a for loop in a script will loop over all of the arguments supplied to the script. The built-in shell variable **$*** evaluates to the argument list, so we can write

```
for i in $*; do
   commands
done
```

but the shell also has a shorthand for this particular case, namely:

```
for i; do
   commands
done
```

In addition to **for** iterations, there are also **while** loops:

```
while command; do
   commands
done
```

and **until** loops:

```
until command; do
   commands
done
```

For these iterations, the body of the loop is repetitively executed as long as *command* succeeds or fails, respectively.

The following table summarizes some built-in shell variables that are particularly useful for script writing [1]:

Variable	Evaluates to
$#	number of arguments
$*	all arguments
$?	return value of last command
$$	process-id of the script

Also observe that a script inherits all of the *environment variables* (such as **$HOME**, **$PATH**, ...) that have been set in the invoking shell (be it a **tcsh**, **sh**, **bash** or whatever). Within a Bourne-shell, a syntax differing from the C-shell case must be used to set an environment variable:

```
var=value
export var
```

The **export** command tells the shell that *var* is to be identified as an environment variable. Also, the "Bourne-again" shell, **bash** (which is the same as **sh** on Linux systems), allows the above two statements to be combined into a single one:

```
export var=value
```

As shown in the next table [1], we can also use some tricks in the evaluation of shell variables to make writing scripts a little easier at times:

Expression	Evaluates to
$*var*	value of ***var***, nothing if ***var*** undefined
${*var*}	same as above; useful if alphanumerics follow variable name
${*var*-thing}	value of ***var*** if defined; otherwise **thing**; **$*var*** unchanged.
${*var*=thing}	value of ***var*** if defined; otherwise **thing**; if undefined **$*var*** set to **thing**
${*var*+thing}	**thing** if ***var*** defined; otherwise nothing
${*var*?message}	if defined, **$*var***; otherwise print **message** and exit shell.

Finally, the **read** command can be used to interactively provide input to a script. Here's an example

```
% cat read-example
#! /bin/sh

echo "Hello there!  Please type in your name:"
read name
echo "Pleased to meet you, $name"

% read-example
Hello there!  Please type in your name
Matthew Choptuik
Pleased to meet you, Matthew Choptuik
```

References

[1] Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment*, Prentice Hall, 1984