Navigating Linux Systems

Amit Jain, Luke Hindman, and John Rickerd

Last Revised: July 30, 2024

Acknowledgments

This material is based upon work supported by the National Science Foundation under Award No. 1623189. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation

The authors would especially like to thank Ariel Marvasti and Phil Gore for their proof reading and suggestions.

Material edited for use by College of Western Idaho students (through gracious permission of Boise State University) by Lawrence Sevigny.

Contents

Chapter 1

Beginner's Guide

1.1 Introduction

This chapter is intended to give us some basic information that we need to start using the Linux operating system. The commands discussed in this guide are common to all the Linux variants (such as Ubuntu, Fedora *et al*). Most of the Linux commands that we discuss are common to several other operating systems such as Mac OS X and other UNIX systems.

Microsoft Windows supports a Windows Subsystem for Linux (WSL), that allows us to run Linux under Windows. We can easily find instructions on the internet on how to install WSL (it is a standard part of Windows). Note that for this course, we will be working with a Linux installation as a Virtual Machine under Windows or Mac in order to give us the full power of Linux under our control.

When we login our personal virtual machines (or on any desktop version of Linux), we will see a graphical desktop (aka **G**raphical **U**ser **I**nterface or GUI desktop). Modern Linux desktops are are intuitive to use and customizable. The recommended personal Virtual Machine also comes pre-installed with a variant of Gnome called Budgie.

In this guide, we will focus on the command line interface for Linux for three reasons: (1) it is often the primary way to access a server in the cloud; (2) the command line can be easily automated, which gives it great power; (3) the graphical desktops for Linux are similar to what we see on Microsoft Windows or Macs so we can learn to use them relatively easily.

To access the full power of Linux, we need to start up a *terminal* (also known as a *console*). On your personal Virtual Machine, there should be an icon to start a terminal. In most Linux desktops, we can right click on the desktop background and launch a terminal from the menu.

The terminal lets us enter commands that are run by a special program called a shell. The shell acts as an intermediate between the user and the operating system. Although there are many shells to choose from, the default shell for our machines will be the Bourne Again SHell (or `bash`).

1.2 Notation

All input that a user types and the output produced on the terminal is shown in the `teletype` font. The shell prompt is shown as `alice@ubuntu: $`. The following shows what output is produced by typing in the command `whoami` to the shell.

```
alice@ubuntu:~$ whoami
alice
alice@ubuntu:~$
```

The actual prompt that we may see on a particular system might be different. We can also choose our own prompt (See Section 3.1). Another notation used is to specify a syntactical category. For example, if the user is supposed to provide a filename as an argument to a command, it would be shown as `<filename>`, where the `<` and `>` symbols imply that any valid filename can be specified. We don't actually type the `<` and `>` characters!

1.3 Getting started

1.3.1 Logging in

In Linux the procedure for obtaining an authorized use of the system is called "logging in". When we are on a workstation then we should have a `login:` prompt on the screen. If the screen is blank, move the mouse around to make the monitor come back on. If there is no response, then the monitor may be turned off. In that case turn on the monitor. Enter the user name at the `login:` prompt and then enter your password at the `password:` prompt. The password will not appear on the screen when typed on the command line.

1.3.2 Changing the password

We may change our password after we have logged in to the system. To change the password from the terminal, type `passwd`. Then the system will prompt us for the old password. After we have entered our old password the system will ask us to enter our new password. Choose a password that is difficult to guess by others. The system may reject our password as being too small. If so, select another one; the system will make suggestions about how to select a better password.

1.3.3 Logging out of the system

It is very important to logout of the system. Otherwise our account can be accessed by an unauthorized person who may misuse our account.

- To log out of a console terminal type `logout` or `exit`.
- To log out of the graphical desktop session, go to the start menu (usually on top right or bottom left), and select the *Leave* or *Logout* option in the menu.

1.4 Some basics

1.4.1 Correcting our typing

We can use the `Backspace` (sometimes labeled with the image of an arrow pointing to the left) key for erasing one character at a time and `Ctrl-u` (that is, press `Ctrl` and `u` simultaneously) for killing the entire line.

1.4.2 Special keys

Most of the keyboard characters are ordinary displayable characters with an obvious meaning, but some have special significance to the computer. The `RETURN` (sometimes labeled as `Enter`) key signifies the end of a line of input; the shell echoes it by moving the terminal cursor to the beginning of the next line on the screen. `RETURN` must be pressed before the shell will interpret the characters we have typed.

`RETURN` is an example of a *control character*—an invisible character that controls some aspect of input and output on the terminal. If we press the keys `Ctrl-m` (that is, press `Ctrl` and `m` simultaneously) the effect is the same as pressing the `RETURN` key.

By typing `Ctrl-c`, from the terminal, we can kill most running programs or commands. (The `c` stands for cancel.)

Another important control character is `Ctrl-d`, which tells a program or a command that there is no more input. For example, typing `Ctrl-d` on the terminal will signify to the shell that there is no more input from the user and it will log us out. `Ctrl-d` can also be thought of as an "end of file."

Typing `Ctrl-z` suspends a running program. (The z stands for zzzzzz's or sleep) Type `fg` (for foreground) to restart the sleeping program.

`Ctrl-s` stops our screen from scrolling and `Ctrl-q` resumes scrolling. On most keyboards there is a `Scroll Lock` key for the same purpose.

Here is summary of the control characters.

Control Character	**Action**
`Ctrl-c`	Cancels the program running on the command line
`Ctrl-d`	Tells a program that there is no more input (similar to an End of File)
`Ctrl-z`	Suspends the program running on the command line
`Ctrl-s`	Stops our screen from scrolling
`Ctrl-q`	Resumes scrolling it was stopped

1.4.3 Case sensitivity

All commands and names in Linux are case sensitive. For example, the two commands, `run` and `Run` are *not* the same command.

1.4.4 How to find information?

There are four primary sources of information for a Linux system: (1) manual pages, (2) HOWTOs and documentation for programs, (3) help from programs and (4) Google (or other) search engine. Please bookmark this section and come back to it as needed. Just browse through it the first time quickly to see what options are available.

- **Man pages:** On-line manual pages are available on every Linux system. These are often the best sources of reference information about various programs on the system. Here are some useful pointers about man pages.
  - **We want to know more about a certain command or program**. To find out about a command and a description of what it does, type `man <command_name>`. Sometimes there may be more than one command with the same name. For example `sleep <secs>` command sleeps for the specified number of seconds. There is another man page for the sleep system call that we can call from a C/C++ program. Typing in `man -a sleep` (where `-a` is a command line argument to the `man` command) will show us all the man pages for `sleep`. It will show the first man page. If that is not the one we want, then press `q` to quit that page and then it will show us the next one on the same command and so on. We can use `PageUp` and `PageDown` keys to browse a manual page. On a laptop, we may have to find keys that are equivalent to page up and page down keys.
  - **We want a one line summary of what a command does**. To obtain a one line summary about a command type `man -f <command_name>`.
  - **We want to find a command that does something we want to do.** Think of an appropriate keyword that best describes what we are looking for and then type `man -k <keyword>`, which displays one line summaries of the commands that reference that keyword. For example:

    ```
    alice@ubuntu:~$ man -k sleep
    Tcl_Sleep           (3)  - delay execution for a number of milliseconds
    Tcl_Sleep [Sleep]   (3)  - delay execution for a number of milliseconds
    apmsleep            (1)  - go into standby mode and wake-up later
    nanosleep           (2)  - pause execution for a specified time
    sleep               (1)  - delay for a specified amount of time
    sleep               (3)  - Sleep for the specified number of seconds
    usleep              (1)  - sleep some number of microseconds
    usleep              (3)  - suspend execution for microsecond intervals
    ```

    The numbers in the second column show the Section number for the manual page. So now we see several commands/calls that are related to sleeping. Find out more by doing `man 3 sleep` or `man 1 sleep`, where the number is the section number.
  - Man pages are organized into sections. The following shows the different sections and what they logically contain.

Section	The human readable name
1	User commands that may be started by everyone.
2	System calls, that is, functions provided by the kernel.
3	Subroutines, that is, library functions.
4	Devices, that is, special files in the /dev directory.
5	File format descriptions, e.g. /etc/passwd.
6	Games, self-explanatory.
7	Miscellaneous, e.g. macro packages, conventions.
8	System administration tools that only root user can execute.

For example, if we want to find out more about the `printf` library call, we can simply look for it in Section 3 using the following command:

`man 3 printf`

Otherwise if we say `man printf`, we will get the first man page found, which happens to be in Section 1. This command describes a `printf` that we can use from the shell and is different from the `printf` used in a C program.

- **HOWTOs and program documentation**. Linux has hundreds of excellent HOWTO documents available that describe specific topics. Go to the website: `http://www.tldp.org/docs.html` for a searchable list of HOWTOs. On your local system, look in the directory /usr/share/doc. There are hundreds of subdirectories, one each for specific programs that have more detailed information about that program.

- **Ask a program:** Many programs will display a message describing how to use the program when supplied with the command line option `--help`. For example, try

```
passwd --help
```

- **Google it!** One of the best and fastest sources of information is the Google search engine on the internet. Point the browser to www.google.com. Get familiar with this search engine and it may save countless hours! We can also use other search engines.

1.5 Files and directories

A file is a sequence of bytes. No structure is imposed on a file by the system, and no meaning is attached to its content— that is, the meaning of the bytes depends solely on the programs that process the file. The Linux file system is structured as a tree. The leaves of the tree are ordinary files. The internal nodes of the tree are called directories. A directory is a special file that contains pointers to other files and directories. A subdirectory is a child of another directory.

1.5.1 File names

The name of a file can be any sequence of characters and numbers including even special symbols like .,-,_ etc. and blank space. The top level directory of the file system is called the root directory, and it is represented by a single slash (/). The path to a file is the sequence of directory names starting from the root, and going through various sub-directories to the file. The complete name for any file is given by the path from the root to that file, written from left to right. The complete path to a file is referred to as the absolute pathname.

To specify the absolute pathname, start with a single slash for the root. Then append the names of all directory nodes from the root to the desired file, adding another slash after each directory name. Finally, append the file name itself. For example, the absolute pathname of the file *testfile*, which is a child of directory *myDir*, which is a child of the directory *anotherDir*, which is a child of the *root* directory is: `/anotherDir/myDir/testfile`.

We can also refer to a file by its relative pathname by giving the path from our current position in the tree to the place where the file is. We go up one level in the tree by entering :
`cd ..`
For example, if we are currently in the directory *myDir* and we want to find a file *testfile1* in the directory *yourDir*, which is also a child of directory *anotherDir*, then the relative filename is:
`../yourDir/testfile1`

1.5.2 Creating files and directories

Normally we would use a text editor for creating files. (see Section 4.2.2 on file editing). However we can use the following command to quickly create an empty file.

`touch <filename>`

To create a new directory type:

`mkdir <directory_name>`

1.5.3 Our current directory

When we are using Linux, we are working *in* some directory, which is called our current directory or working directory. We can find out the name of our current directory by using the command `pwd` (for print working directory).

1.5.4 Changing directories

To move to a different directory use the command `cd`. Typing `cd <dir_name>` will take us to the subdirectory `<dir_name>`. We can give either the absolute path of the directory or the relative path from the current directory.

1.5.5 Our home directory

Every user has a home directory. Our home directory is the top of the tree containing all of our files. When we login, we are initially in our home directory. If we type `cd` with no directory name after it, we are moved to our home directory. To find out the pathname of our home directory, type the command `cd` to switch to our home directory and then use the `pwd` command to see the pathname.

1.5.6 Special directories

There are five directories that we will refer to frequently. There are special symbols for referring to them:

- the current directory: .
- the parent directory of the current directory: ..
- our home directory: ~
- another user's home directory: ~<user_name>
- the root directory of the system: /

1.5.7 Special files

Every user has two special files: `.bash_profile` and `.bashrc`, in her home directory. These files and other files whose names start with a "." (dot) do not normally show up in the directory listing (also known as hidden files). See Section 1.5.9 for ways of listing the "dot" files. The "dot" files are used for initializing applications and for customizing our environment (see Section 3.1). For example, when we login, the `.bash_profile` file sets up our session and terminal characteristics.

Many applications have special files that have names starting with a ".", usually located in our home directory. These files are referred to as hidden files (as they don't normally show up in a directory listing). These startup files contain initialization commands for the application. For example, the file `.vimrc` contains the initialization commands for the *Vim* text editor. The suffix "rc" stands for run control.

As mentioned before, the current directory is "." and the parent directory is "..". These two entries are in every directory listing that includes the "dot" files.

1.5.8 Viewing the contents of a text file

A few different ways of listing the contents of a text file are discussed below.

1. `cat <filename>` This will cause the text of the file to scroll off the screen if the text occupies more than one screen of lines. The command `cat` can also be used to concatenate multiple files. For example, `cat <file1> <file2>` will concatenate the two files and then display on the terminal.
2. `more <filename>` This will display the file one screenful at a time; press the space bar to advance to the next screen; press `RETURN` to scroll up one line; and `q` to quit.
3. `less <filename>` The command `less` is similar to the command `more`. However `less` allows us to move backwards in a file using the up/down arrow keys or the `PageUp` and `PageDown` keys. The command `less` is also faster than `more` on large files. (`less` is more, a cliche or not?)
4. Use a text editor. For example: `gedit`, `vim`, `emacs` etc.

1.5.9 Listing files and directories

The command `ls` will list the names of files and directories in our current directory. There are several options in this command. Commonly used ones are as follows.

1. `ls -l` will provide a long listing of the contents of the current directory. This listing includes the file type, permissions, owner and group associated with the file, the time of last modification, size of the file and the file name.
2. `ls -l -h` same as above except the size is shown in human readable units like KB, MB, GB etc.
3. `ls -F` will mark all files that are executable with a star (*) and all directories with a slash (/).
4. `ls --color` will show a colored listing of files. The default is to show directories in blue, executables in green. See Section 3.1.5 for more details. This is the default on Linux.
5. `ls -t` will list files in time order, most recent first.
6. `ls -a` will show all files in the current directory, including the special "dot" files.
7. `ls -R` will list all files recursively in a directory, that is, all files and subdirectories inside it and then inside those subdirectories recursively all the way to all files and subdirectories contain in that directory.

Use `ls --help` to see the other options of the ls command.

1.5.10 Wild-cards and file name completion

The wild card characters are `*` and `?`. The symbol `*` matches any string and `?` matches any single character. For example, the following command lists all files with names starting with the string *hw* in the current directory:

```
ls hw*
```

The following command lists all files ending with `.java` in all subdirectories of the current directory.

```
ls */*.java
```

These characters are simple regular expressions. Regular expressions are a powerful way to express text patterns. We will encounter them again in our explorations in future courses and in life!

The shell has a file name completion feature by which we can avoid typing long file and directory names. Suppose we have a directory named `horriblylongname` and we wanted to `cd` to this directory. We can type `cd horr` and then hit the TAB key and the shell will attempt to find a filename starting with the string `horr`. If there is a unique match, the shell will complete the file name. If there is more than one file or directory that has a name starting with the prefix `horr`, then the shell will beep. Hitting the TAB key again will show us all the files or directories that start with the prefix `horr`. Now we can type a few more characters until the prefix is unique so the shell can complete the filename for us. The file completion feature is very handy and saves the user a lot of typing and errors.

1.5.11 Copying files or directories

- To make a copy of a file type cp <file1> <file2> (if <file2> already exists then it will be overwritten with the contents of <file1>).
- To copy a file into another directory type cp <file1> <dir_name>
- To copy the contents of one directory into another directory use the command: cp -R <dir_name1> <dir_name2>. The option -R (we can also use -r) stands for recursive since the first directory is recursively copied in to the second directory, that is, all subdirectories inside the directory being copied are also copied recursively and so on.

1.5.12 Renaming a file or directory:

- To rename a file, type mv <old_file_name> <new_file_name>.
- To rename a directory, type mv <old_dir> <new_dir>.
- To move a file into another directory, type mv <file1> <dir_name>.

1.5.13 Removing(Deleting) files or directories

- To remove a file in our current directory type rm <file_name>.
- To remove an empty directory type rmdir <dir_name>.
- To remove a nonempty directory type rm -fR <dir_name> (or rm -fr <dir_name>) . Beware! This will recursively delete everything in that directory. To recover files deleted accidentally, see Section ??.

1.6 Basic useful commands

1.6.1 Finding the date and the time

The command date prints the date and time.

1.6.2 Obtaining information about users

- To see our login name, type whoami
- To see who is presently logged onto the system, type who.
- To see what programs users are running, type w. For an explanation of the column heading, read the man page with the man w command.

- To get information about a specific user type `pinky -l <user_name>`. The program `pinky` gives some basic information about the user and prints out the file `.plan` if the user has such a file in their home directory. So if we make a `.plan` file in our home directory, other users will see it when they use the `pinky` command using our login name. We can use a text editor to create such a file. Try the following on a CWI Computer Science machine:

  `pinky -l lawrence`

We can connect to it with the `ssh` command as shown in Section 1.7.2.

1.6.3 Finding system information

- To find the number of processors available on the system, type `nproc`
- To find details about the processors, type `lscpu`
- To find out the amount of memory on the system, type `free`. It shows the memory in kilobytes, which can be hard to parse. To see the output in more human readable units, type `free -h`
- To see what version of Linux is installed, type `cat /etc/redhat-release`
- To see a summary of system state, use the `top` command. We can change its output to color by typing `z` (lowercase z). Typing `z` again turns the colors off. Note that this shows the top active processes on the system. The first column is the process id. Each process has a unique process id. We can see the load on individual CPUs by pressing '1'. Pressing '1' again takes us back to the view with combined load on all the CPUs. Pressing M sorts the processes by most memory usage. Pressing P sorts the processes by most CPU time used.

Also, try these commands on the CWI Computer Science server for more interesting results! We can connect to it with the `ssh` command as shown in Section 1.7.2.

1.6.4 Recording a terminal session

The command `script <filename>` starts a new session and records all characters input or output to the terminal in the file `<filename>`. Here is an example.

```
alice@ubuntu:~$ script log
Script started, file is log
alice@ubuntu:~$ date
Sat Dec 31 10:51:40 MST 2022

alice@ubuntu:~$ nproc
48

alice@ubuntu:~$ free -h
```

```
              total        used        free      shared  buff/cache   available
Mem:            62Gi       2.2Gi        55Gi       153Mi       4.5Gi        59Gi
Swap:           44Gi          0B        44Gi

alice@ubuntu:~$ exit
exit

Script done, file is log
alice@ubuntu:~$
```

1.6.5 Sleeping and sequencing

The command `sleep` just sleeps for the given number of seconds :-) For example, try the following commands:

```
alice@ubuntu:~$ sleep 2
```

We can try a longer sleep and use Ctrl-c to cancel it. Here is another command to try.

```
alice@ubuntu:~$ date; sleep 10; date
Sat Dec 31 10:50:06 MST 2022
Sat Dec 31 10:50:16 MST 2022
alice@ubuntu:~$
```

Note that the semi-colon allows us to type multiple commands in one line that are executed in sequence. The sleep command is useful in scripting, which we will study in a later module.

1.6.6 Time a command or a program

The command `time <command>` times a `<command>`, where the `<command>` can be a program or a Linux command. However, there are more precise ways of measuring the execution time for parts of a program, which we will learn about in various classes.

For example:

```
alice@ubuntu:~$ time sleep 4

real    0m4.020s
user    0m0.000s
sys     0m0.000s
alice@ubuntu:~$
```

The command time gives the "real" (elapsed) time followed by time spent by the CPU in user mode as well in system mode. Due to programs potentially waiting for input and due to multiple users on a system the *user* plus *system* time is usually less than the *real* time reported by the `time` command.

1.6.7 Counting the number of characters, words and lines

The command `wc <filenames>` counts lines, words and characters for each file. Using `wc -l` counts number of lines only. (`wc` is short for *word count*) For example, the following counts the number of words in the standard dictionary in the system.

```
alice@ubuntu:~$ wc -l /usr/share/dict/words
479828 /usr/share/dict/words
alice@ubuntu:~$
```

1.6.8 Sorting files

The command `sort <filenames>` sorts text files alphabetically by line. Sort considers each line as a record with space-separated fields. By default the first field is used to sort the file. The `sort` command has many options: sorting numerically, sorting in reverse order, sorting on fields within the line etc. Please check the man page using the command `man sort` for more options.

Here are some common options:

```
sort -r file1           reverse normal order
sort -n                 sort in numeric order
sort -rn                sort in reverse numeric order
sort -f                 fold upper and lower case together
sort -k 3,3             sort on the third field
```

For an example usage, see below:

```
alice@ubuntu:~$ cat listing
1040 1680x1050.jpg
   4 log1
   4 log2
  16 logoCOENCS.png
1112 metageek_splash_screen.jpg
   0 testfile1
   0 tt

alice@ubuntu:~$ sort -n -k 1,1 listing
   0 testfile1
   0 tt
   4 log1
   4 log2
  16 logoCOENCS.png
1040 1680x1050.jpg
1112 metageek_splash_screen.jpg

alice@ubuntu:~$
```

1.6.9 Displaying the last few lines in a file

The command `tail <filename>` displays the last 10 lines of the file `<filename>`. The command `tail -<n> <filename>` prints the last `<n>` lines. A neat option is `tail -f <filename>`, which "tails" the end of the file; i.e., as the file grows, tail displays the newest line. This can be handy for monitoring a file being written by a running program.

Try the following command to see the last 10 words in the standard dictionary on our system:

`tail -10 /usr/share/dict/words`

1.6.10 Finding the differences between two text files

The command `diff <file1> <file2>` displays all differences between the two files `<file1>` and `<file2>`. It has many useful options.

Here is an example showing the use of `diff` on two file that differ only in a few lines.

```
alice@ubuntu:~$ cat list1
1040 1680x1050.jpg
   4 log1
  16 logoCOENCS.png
   0 testfile1
   0 listing

alice@ubuntu:~$ cat list2
1040 1680x1050.jpg
   4 log2
1112 metageek_splash_screen.jpg
   0 testfile1
   0 tt

alice@ubuntu:~$ diff list1 list2
2,3c2,3
<    4 log1
<   16 logoCOENCS.png
---
>    4 log2
> 1112 metageek_splash_screen.jpg
5c5
<    0 listing
---
>    0 tt
alice@ubuntu:~$
```

The tag `2,3c2,3` (`c` stands for change) tells the lines numbers from the first file that need to be changed to match the line numbers from the second file.

The `diff` command has several options. It can ignore white space, ignore case, have the output be side by side and others documented on its man page. We can also use the `sdiff` command that shows differences side by side. Two GUI diff programs are `meld` and `kompare` if we are running the Gnome or KDE Desktop. All of these use the `diff` command under the hood.

1.6.11 Finding the differences between two binary files

Use the command `cmp` to check if two files are the same byte by byte. The command `cmp` does not list differences like the `diff` command. However it is handy for a fast check of whether two files are the same or not (especially useful for binary data files).

```
alice@ubuntu:~$ cmp /bin/ls /bin/cat
/bin/ls /bin/cat differ: byte 25, line 1
alice@ubuntu:~$
```

1.6.12 Finding patterns in files using our buddy `grep`

Use `grep <pattern> <filename> [<filename>]`. The `grep` command displays lines in the files matching `<pattern>` as well as the name of the file from which the matching lines come from. The command `grep` has many useful command line options. Please see its man page for more information. Here we will show some examples of a typical usage of `grep`.

The option `-n` makes `grep` display the line number in front of the line that contains the given pattern. For example:

```
alice@ubuntu:~$ grep JButton TimerDemoPanel.java
import javax.swing.JButton;
        private JButton startButton;
        private JButton stopButton;
                startButton = new JButton("Start");
                stopButton = new JButton("Stop");
                        JButton clicked = (JButton) e.getSource();

alice@ubuntu:~$ grep -n JButton TimerDemoPanel.java
6:import javax.swing.JButton;
31:     private JButton startButton;
32:     private JButton stopButton;
78:             startButton = new JButton("Start");
82:             stopButton = new JButton("Stop");
98:                     JButton clicked = (JButton) e.getSource();
alice@ubuntu:~$
```

The option `-v` inverts the search by finding lines that do not match the pattern.

The option `-i` asks `grep` to ignore case in the search string.

A very powerful option is the recursive search option, `-r`, that will make `grep` search recursively in a directory or directories. For example, the following command searches for all files with the string "Crow" in them.

```
alice@ubuntu:~$ grep -r "Crow" C-examples/
C-examples/plugins/plugin1.c:/* Author: Dan Crow
C-examples/plugins/plugin2.c:/* Author: Dan Crow
C-examples/plugins/runplug.c:/* Author: Dan Crow (modified by Amit Jain)
alice@ubuntu:~$
```

Programmers often use this option to quickly search for a declaration of a variable or class in a large project consisting of hundreds or thousands of files in many directories and subdirectories. This is often called *"grepping"* the source! See the following for an example.

```
alice@ubuntu:~$ ls
chap01  chap02  chap03  chap04  chap05  chap06  chap07  extras  file-io
graphics  Makefile  README.md  UML

alice@ubuntu:~$ grep -r JTabbedPane *
chap06/LayoutDemo.java:  import javax.swing.JTabbedPane;
chap06/LayoutDemo.java:         JTabbedPane tp = new JTabbedPane();
alice@ubuntu:~$
```

Note that the asterisk (`*`) matches all directories in the examples folder. We could also have used: `grep -r JTabbedPane .` (using dot for the current directory)

1.6.13 Input-Output redirection

When a command is started under Linux it has three data streams associated with it: standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). The corresponding file streams are numbered are 0, 1 and 2. Initially all these data streams are connected to the terminal.

A terminal is also a type of file in Linux. Most of the commands take input from the terminal and produce output on the terminal. In most cases we can replace the terminal with a file for either or both of input and output. For example:

```
ls >  listfile
```

puts the listing of files in the current directory in the file `listfile`. The symbol `>` means redirect the standard output to the following file, rather than sending to the terminal.

The symbol `>>` operates just as `>`, but appends the output to the contents of the file `listfile` instead of creating a new file. Similarly, the symbol `<` means to take the standard input for a program from the following file, instead of from a terminal. For example:

```
mail -s "program rubric" mary@my.cwi.edu jose@my.cwi.edu < letter
```

mails the contents of the file `letter` to the two users: `mary, and jose`. A Java (or any other) program that reads input from the keyboard can be redirected to read from a file as follows:

```
java GetInput < input.txt
```

To redirect error messages (which are sent on `stderr` with file descriptor 2) see the following:

```
javac BadProgram.java 2> error.log
```

Or if we want both the output and the error messages to go to a file, see the following:

```
javac BadProgram.java > log 2>&1
```

1.6.14 Where do commands live?

Most of the "commands" that we use on the Linux command line are in fact programs (which are stored as files) located typically in the `/bin` or `/usr/bin` directories. Take the `cp` command for instance. We can use the "`which`" command to determine where the `cp` program is located.

```
alice@ubuntu:~$ which cp
/usr/bin/cp
```

Here we can see that the `cp` program is located in the `/usr/bin` directory. When we type a command, the shell searches a list of directories to find the command and runs the first one it finds.

Note that `/` is the top of the file system under Linux. It is also known as the root directory of the file system. The directories `/bin`, `/home`, `/usr` are subdirectories of the root directory of the system. The directories `/home/alice` and `/usr/bin` are subdirectories of `/home` and `/usr` respectively. We will discuss the layout of the file system in a Linux system in more detail in Chapter 2.

1.7 Working on the internet

1.7.1 Hostnames and internet addresses

The *Internet* is a world-wide network of computers. The computers are divided into domains and sub-domains. Each machine has a name and an address that is unique across the internet. To check the name of the machine we are logged on use the command `hostname`. The hostnames and internet addresses of some CWI server machines are:

Hostname	Fully Qualified Domain Name	Internet Address
my	my.cwi.edu	207.70.33.137
selfservice	selfservice.cwi.edu	207.70.33.27

The suffix *cwi.edu* represents the sub-domain (of the .edu domain) comprised of all the machines on campus. To find the internet address of a machine, use the command `host <hostname>`. Here are some examples.

```
alice@ubuntu:~$ host google.com
google.com has address 142.251.215.238
google.com has IPv6 address 2607:f8b0:400a:800::200e
google.com mail is handled by 10 smtp.google.com.
```

```
alice@ubuntu:~$ host my.cwi.edu
my.cwi.edu has address 207.70.33.137

alice@ubuntu:~$ host selfservice.cwi.edu
selfservice.cwi.edu has address 207.70.33.27
```

1.7.2 Remote access using Secure SHell (SSH)

The Secure SHell (SSH) program is a secure encrypted client for login to remote machines. The password as well as the data that is transmitted is encrypted. Note that secure shell software can also be obtained for Microsoft Windows (use the free MobaXTerm software that contains a ssh client in it) and it comes pre-installed with Linux and Mac OS X.

We can login to another machine using the `ssh` command (that is name of the Secure SHell program). The command `ssh` starts a new shell on the remote machine. For example, we can login to the CWI Computer Science server from our local system using the following command:

`ssh USERNAME@63.146.50.133`

where `USERNAME` would be replaced by our login user name on the `63.146.50.133` server (and lab). To logout of the `63.146.50.133` server, use the `exit` command or just type `Ctrl-d`.

1.7.3 Remote file copy using Secure CoPy (SCP)

The secure shell program also comes with a secure copy program that can be used to copy files to/from remote machines. The command to use is `scp`. For simple usage, the `scp` program takes two arguments:

1. the file we want to copy/send
2. the place we want to copy/send to (optionally with a new name)

The basic syntax would be:

`scp sourcepath/filename destinationpath/optionalNewFilename`

Note that the first argument is the FROM part and the second is the TO part.

To copy a directory, use the recursive option `-r` as shown below:

`scp -r sourcepath/directoryname destinationpath/optionalNewDirectoryname`

To specify a remote machine, use the internet hostname of the machine in the path (along with user name on remote machine if different from the local machine). For example:

`scp -r program4 USERNAME@63.146.50.133:workspace/`

copies the local directory named `program4` to under the `workspace` directory on the remote server. Note the required colon ':' after the address. The file path on the remote machine goes after that. If the second argument was `USERNAME@63.146.50.133:` (that is, nothing after the colon), then it would copy the `program4` directory to user's home directory on the remote server. Here the

`USERNAME` is the login user name on the remote server. We can skip this part if the user name on the local system is the same as on the remote `onyx` server.

Note that we would use the `scp` command tyically only on our local system to download file(s) from the remote server or to upload file(s) to the remote server. We cannot use it on the remote server to access our local system for two reasons: (1) our local system is usually behind a firewall and not accessible via secure shell or secure copy (2) the server we are connecting to needs to run the secure shell service to allow clients to connect to it. Our laptop or desktop would typically not have this service enabled (due to security concerns).

How to download file(s) from the remote server to our local machine

A typical example is that we want to copy a project folder from the remote server to our local machine. We will need to know the relative path (relative to our home directory) to the folder on the remote server. Suppose that is `workspace/program4`. Then we can use the following command to copy the `program4` folder to our local machine:

```
scp -r alice@63.146.50.133:workspace/program4 .
```

Here we use the recursive option since it is a directory and not just a file. Note that we must end the remote IP address with a colon (: after the `133`). Then we need to provide the *TO* part. We provide a dot ('.') here to specify the destination as the current directory on our local system. Suppose we wanted to copy it to the `workspace` directory in our home directory on our local system, then we would have used:

```
scp -r USERNAME@63.146.50.133:workspace/program4 ~/workspace
```

Note it both cases above the name of the directory in our local system will be the same as on the remote server. We can change it if we wish as follows:

```
scp -r USERNAME@63.146.50.133:workspace/program4 ~/workspace/newprogram4
```

Now the directory `program4` will be copied as `newprogram4` under the `workspace` directory on our local system.

How to upload file(s) from our local machine to the remote server

A typical example is that we want copy a folder from our local system to the remote server. We would need to know the path on remote server to where we want to copy. Here is an example:

```
scp -r program4 USERNAME@63.146.50.133:workspace/
```

Here the local folder `program4` gets copied to the remote server under the `workspace` folder in the user's home directory on the remote server. Again, note that we must end the remote IP address with a colon (: after the `133`). We can also change the name as we saw in the previous section.

1.7.4 How to use SSH securely without a password

On our Linux VM, create a pair (private/public) pair of authentication keys as follows. Do not enter a passphrase when it asks (simply hit Enter to not have a passphrase)

```
ssh-keygen -t ed25519 -C "your_email@my.cwi.edu"`
```

The keys are generated in the folder `~/ssh` and are in the files `id_ed25519` and `id_ed25519.pub`. Go ahead and take a look at them with the cat command!

```
cat ~/.ssh/id_ed25519.pub
cat ~/.ssh/id_ed25519
```

Now use scp command to copy your public key `~/ssh/id_ed25519.pub` to the remote server as follows:

```
scp ~/.ssh/id_ed25519.pub 63.146.50.133:
```

Then use ssh to login to the remote server and put the public key in the appropriate place.

```
ssh 63.146.50.133
----------on 63.146.50.133---------
cd .ssh
cat ~/id_ed25519.pub >> authorized_keys
chmod 600 authorized_keys
cd ..
rm -f id_ed25519.pub
chmod 700 .ssh
exit
```

If the `.ssh` folder doesn't exist, create it first (`mkdir ~/.ssh`). Now, try using `ssh` to login to the remote server again. You should get in without a password! Note that this is still secure as long as you keep your private key (`~/ssh/id_ed25519`) secure.

1.8 Summary

- Basic commands: `date, who, whoami, w, man, touch, cat, script, sleep, time, wc, sort, tail, diff, cmp, grep, locate`
- File system navigation: `cd, ls, cp, mv, rm, mkdir, pwd`
- Controlling programs using `CTRL-c / CTRL-d / CTRL-z`
- Basic System Monitoring: `top, nproc, lscpu, free`
- Remote access: `scp, ssh`

Chapter 2

Files and File Systems

Everything in a Linux system is a file. Hence, understanding the file system is important in becoming more proficient at navigating the system. In this chapter, we will look further into what files are, how they are represented, and the layout of the file system. This is a continuation of topics introduced earlier in Section 1.5 from Chapter 1. Furthermore, we will look at file security and permissions, some common file system operations and also look under the hood at devices and partitions, where file systems are physically stored.

2.1 Files

A file is a sequence of bytes. A byte is a small chunk of information that is 8 bits long. A bit is one binary digit that is either 0 or 1. Let us create a small file to play around with using a text editor (`gedit`, a simple GUI text editor, should be available on your VM).

```
gedit junk
```

Add the following two lines to the file and save it.

```
I am a file.
Are you a file too?
```

To see the file,

```
alice@ubuntu:~$ cat junk
I am a file.
Are you a file too?
alice@ubuntu:~$
```

We can see a visible representation of all the bytes in the file with command `od` (octal dump):

```
alice@ubuntu:~$ od -c junk
0000000   I       a   m       a       f   i   l   e   .  \n   A   r   e
0000020       y   o   u       a       f   i   l   e       t   o   o   ?
0000040  \n
0000041
alice@ubuntu:~$
```

The option -c means "interpret bytes as characters." Turning on the -b option will show the bytes as well:

```
alice@ubuntu:~$ od -cb junk
0000000   I       a   m       a       f   i   l   e   .  \n   A   r   e
        111 040 141 155 040 141 040 146 151 154 145 056 012 101 162 145
0000020       y   o   u       a       f   i   l   e       t   o   o   ?
        040 171 157 165 040 141 040 146 151 154 145 040 164 157 157 077
0000040  \n
        012
0000041
alice@ubuntu:~$
```

The 7-digit numbers on the side are the position of the next character (in *octal*, or base-8). The character at the end of each line from the file has the octal code `012`, which is the ASCII code for the newline character. Note that most systems and languages use Unicode for characters (such as Java). Linux uses the UTF-8 encoding for Unicode, which makes any ASCII code also a valid Unicode. There is a lot more to Unicode and character encoding but for now this is sufficient to keep us going.

Some other common special characters are backspace (`\b` or `010`), tab (`\t` or `011`), and carriage return (`\r` or `015`). The codes are again given in octal (or base-8). When we type a command on a line and press *Enter*, it generates a newline and the characters typed are processed by the system. That means we can backspace and edit the current line as long we as don't press the *Enter* key.

Note that there is no special character to denote the end of a file. The operating system signifies the end of a file by saying there is no more data in the file. A program will detect this when the next read from the file returns zero bytes.

Typing `Ctrl-d` sends whatever we have typed so far on the command line to the program that is reading it. So if we haven't typed anything, the program will read no characters, and it will look like the end of the file. That is why typing `Ctrl-d` logs us out of the terminal.

2.2 File types

The structure of a file is determined by the programs that use it. Linux doesn't impose any structure on a file and as a result the system cannot tell us the type of file. However, there is a `file` command that makes an educated guess. It does not use the file name, as those are conventions, which are not reliable. Instead it reads the first few hundred bytes from the file and looks for clues to the file type. For example, here is the `file` command on some typical files.

```
alice@ubuntu:~$ file textFile ListFiles.java  ListFiles.class test.data
textFile:         ASCII text
ListFiles.java:   C source, ASCII text
ListFiles.class: compiled Java class data, version 52.0 (Java 1.8)
test.data:        data

alice@ubuntu:~$ file /home/alice /usr/bin /usr/bin/ls
/home/alice:      directory
/usr/bin:         directory
/usr/bin/ls:      ELF 64-bit LSB shared object, x86-64, version 1 (SYSV),
                  dynamically linked, interpreter /lib64/ld-linux-x86-64.so.2,
     for GNU/Linux 2.6.32,
     BuildID[sha1]=8f8149dbcfdd68a9e7d0e8d29115d05c390522d0, stripped
```

Note that the file named `textFile` contains simple text in it. It guesses the Java source file to be either a C program or an ASCII text file. So in this case, the guess isn't quite spot on. It guesses the type of the compiled Java class correctly. A class file is a binary file, unlike a text file. It does not contain ASCII code or newlines in it. The next file is a data file, which is also a binary format determined by the program that created it. The next two are directories, which are also binary files. The last one is the `ls` program, which is a compiled program and it correctly prints out various details for it. A compiled program is also a binary file.

The lack of file formats in the operating system is, in general, an advantage. It allows any system programs to work with any file, with only a few exceptions. Text files are in general more flexible than binary files but they are a bit slower than binary files. However, binary files require specialized programs to access them. A common example of binary files are databases.

To show the structure of a binary file, examine the following Java program that creates a simple binary data file.

```java
import java.io.DataOutputStream;
import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;

public class CreateDataFile {
  public static void main(String[] args) throws IOException {
    File outFile = new File("out.data");
    DataOutputStream dout = new DataOutputStream(new FileOutputStream(outFile));
    dout.writeInt(1);
    dout.writeInt(2);
    dout.writeInt(3);
    dout.writeInt(4);
    dout.close();
  }
}
```

Let us compile and run the above Java program. Then we will examine the output file using `od`. We will use the `-b` option to show the bytes. We also examine the file type for the output file.

```
alice@ubuntu:~$ javac CreateDataFile.java
alice@ubuntu:~$ java CreateDataFile

alice@ubuntu:~$ file out.data
out.data: data

alice@ubuntu:~$ od -b out.data
0000000 000 000 000 001 000 000 000 002 000 000 000 003 000 000 000 004
0000020
```

Note that it shows that there are four integers in the binary data file. Each one takes four bytes. There are no spaces or newline characters in the file as it not a text file.

2.3 Directories

Each user has a login (aka home) directory, for example `/home/alice` for the user *alice*. Each file that alice owns inside her home directory has an unambiguous name, starting with the prefix `/home/alice`. Each running program, known as a process, has a current directory, and all filenames are implicitly assumed to start with the name of that directory unless they begin directly with a slash. The command `pwd` (print working directory) identifies the current directory.

```
alice@ubuntu:~$ pwd
/home/alice
alice@ubuntu:~$
```

The notion of directories is organizational. Related files belong together in a directory. For example, a project in a course, or a set of recipes. For example, the following creates a set of directories for this course.

```
alice@ubuntu:~$ mkdir cpsc-153
alice@ubuntu:~$ cd cpsc-153
alice@ubuntu:~/cpsc-153$ mkdir module01 module02 module03 module04 module05
alice@ubuntu:~/cpsc-153$ cd module01
alice@ubuntu:~/cpsc-153/module01$ touch README.md
alice@ubuntu:~/cpsc-153/module01$ cd ..
alice@ubuntu:~/cpsc-153$ cd module02
alice@ubuntu:~/cpsc-153/module02$ mkdir test1
alice@ubuntu:~/cpsc-153/module02$ cd test1
alice@ubuntu:~/cpsc-153/module02/test1$ echo "I am a test string" > test1
```

The `echo` command shown above prints the string `"I am a test string"` that gets redirected to the file named `test1`. Note that the `ls` command only shows the files in the current directory. The `-R` option shows the directories and files recursively. For example (recall that the dot represents the current directory):

```
alice@ubuntu:~$ ls -R cpsc-153/
cpsc-153/:
module01  module02  module03  module04  module05

cpsc-153/module01:
README.md

cpsc-153/module02:
test1

cpsc-153/module02/test1:
message

cpsc-153/module03:

cpsc-153/module04:

cpsc-153/module05:
```

The structures of directories and files looks like an upside down tree. We can see a more visual representation of the tree structure of a directory with the `tree` command. Try `tree cpsc-153`. If the command isn't found, install it using the package manager, most likely `apt`, `dnf` or `yum` (depending on the Linux distribution installed).

A directory is just a file in Linux as well. However, the system does not let users directly read/write directories. Rather we call the system to do operations on our behalf (for integrity of the file system and for security). This can be done with commands such as `ls`, `mkdir` etc or it can done from inside a program.

Examine the following Java program that recursively lists the size of each file in a directory. Note that the method `listFiles` calls itself recursively!

```java
import java.io.File;

public class ListFiles {
    public static void main(String[] args) {
        if (args.length > 0 ) {
            listFiles(new File(args[0]));
        }
    }
    private static void listFiles(File file) {
        System.out.println(file.getAbsolutePath() + ": " +
                            file.length() + " bytes");
        if (file.isDirectory()) {
            for (File f: file.listFiles()) {
```

```
                listFiles(f);
            }
        }
    }
}
```

Here is its output on the same folder:

```
alice@ubuntu:~$ java ListFiles cpsc-153/
/home/alice/cpsc-153: 4096 bytes
/home/alice/cpsc-153/module04: 4096 bytes
/home/alice/cpsc-153/module02: 4096 bytes
/home/alice/cpsc-153/module02/test1: 4096 bytes
/home/alice/cpsc-153/module02/test1/message: 19 bytes
/home/alice/cpsc-153/module01: 4096 bytes
/home/alice/cpsc-153/module01/README.md: 0 bytes
/home/alice/cpsc-153/module05: 4096 bytes
/home/alice/cpsc-153/module03: 4096 bytes
alice@ubuntu:~$
```

What happens if we keep walking up the file system tree using the `cd` command (assuming we start from within the cpsc-153 directory)?

```
alice@ubuntu:~$ cd cpsc-153; pwd
/home/alice/cpsc-153
alice@ubuntu:~/cpsc-153$ cd ..; pwd
/home/alice
alice@ubuntu:~$ cd ..; pwd
/home
alice@ubuntu:/home$ cd ..; pwd
/
alice@ubuntu:/$ cd ..; pwd
/
alice@ubuntu:/$
```

The directory `/` is called the root directory of the file system. Every file in the system is in the root directory or one of its subdirectories, and the root is its own parent directory.

2.4 Directory hierarchies

The Linux file system is organized as a tree. Table 2.4 shows the file system tree for a typical Linux system starting from the root directory `/`.

Working with the Linux file system is accomplished by using either absolute paths that begin with the root (/) and specify the full path to the file or directory, or by using relative paths that are specified in relation to the current position (directory) within the file system.

Figure 2.1: Linux File System

```
/
|--- bin
|--- boot
|--- dev
|--- etc
|--- home
|    |--- alice
|    |--- hfinn
|--- lib
|--- lib64
|--- media
|--- mnt
|--- proc
|--- root
|--- sbin
|--- tmp
|--- usr
|    |--- bin
|    |--- include
|    |--- lib
|    |--- lib64
|    |--- local
|    |--- sbin
|    |--- share
|    |--- src
|--- var
|    |--- log
     |--- spool
```

The root (top-level) directory of the system is denoted as '/'. Some of the common sub-directories in the root directory are: `home`, `bin`, `sbin`, `usr`, `etc`, `var`, `dev`, `lib`, `proc`, `boot` and `tmp`.

The `home` directory is where the login or home directories for users are kept. For example, `/home/alice` is the name of the home directory for the user `alice`.

The directory `/bin` (short for binary) contains executable programs commonly used by all users. Look at the programs in that directory. If we are curious about any particular program, then we can read the man page for that program with the `man` command. The directory `sbin` contains programs used for system administration.

The directory `/lib` contains shared libraries and drivers. The directory `/var` contains variable data used by several system programs. The directory `boot` has some basic programs that help in booting up the system. The directory `/mnt` contains mount points for mounting a file system temporarily. For example, this is where our USB drives will show up. The `proc` directory is where we can access information about the operating system and the underlying hardware. For example, try `cat /proc/version`.

The directory `/etc` contains system setup information. For example, it contains the file `passwd` that contains the login information about all the users in the system. Under the `usr` directory, there are are many important system directories. For example, the system man pages are kept in the directory `/usr/share/man`.

The directory `/tmp` (short for temporary) is a directory in which any user can write. We can use this directory as a place for storing files temporarily. Usually the `/tmp` directory has much more space than our home directory. The files in this folder are purged periodically by the system administrator. On our personal machine, they will remain there until we remove them.

2.4.1 Symbolic links and hard links

Sometimes it is convenient to have access to a file by multiple names. This can be accomplished by creating a *link*. There are two types of links: hard and symbolic.

- Suppose we have a file named `xyz.java`. We can create a hard link to it with the `ln` command as follows:

  `ln xyz.java xyz.java.save`

  The file `xyz.java.save` is a hard link to the file `xyz.java`. That means if we change the content of either file, the contents of the other file will change as well. If we accidentally delete `xyz.java`, then we still have the file `xyz.java.save`. What is interesting is that these two files are two names for the same data on the disk. Deleting a file merely removes one of the links. The data on the disk is removed only if no links remain to that file. So making a hard link is different than making a copy. It does not make a copy of the data. See the example below.

```
alice@ubuntu:~$ ln xyz.java  xyz.java.save
alice@ubuntu:~$ ls -l xyz.java*
-rw-rw-r--    2 alice     alice         2374 Dec  3 10:50 xyz.java
-rw-rw-r--    2 alice     alice         2374 Dec  3 10:50 xyz.java.save
alice@ubuntu:~$ rm xyz.java
rm: remove regular file ‘xyz.java’? y
alice@ubuntu:~$ ls -l xyz.java.save
-rw-rw-r--    1 alice     alice         2374 Dec  3 10:50 xyz.java.save
alice@ubuntu:~$
```

- Hard links cannot be made to directories. For this purpose, a symbolic link can be used. Symbolic links can be used for files as well. A symbolic link is created with the `ln` command with the `-s` option. A symbolic link acts as a shortcut or pointer to the original file. However, if the original file is removed, the symbolic link is left dangling. See example below.

```
alice@ubuntu:~$ ln -s xyz.java  f1
```

```
alice@ubuntu:~$ ls -l f1 xyz.java
lrwxrwxrwx    1 alice     alice           8 Dec  3 10:57 f1 -> xyz.java
-rw-rw-r--    1 alice     alice        2374 Dec  3 10:57 xyz.java
alice@ubuntu:~$ rm xyz.java
rm: remove regular file 'xyz.java'? y
alice@ubuntu:~$ ls -l f1 xyz.java
ls: xyz.java: No such file or directory
lrwxrwxrwx    1 alice     alice           8 Dec  3 10:57 f1 -> xyz.java
alice@ubuntu:~$ cat f1
cat: f1: No such file or directory
alice@ubuntu:~$
```

Symbolic links are useful for pointing to other directories or files without having to copy them, which would end up in duplicates that waste space and have to be kept consistent.

2.5 Security and permissions

2.5.1 File protection

Every file (and directory) on Linux has a mode or protection. A file may be readable (`r`), writeable (deletable) (`w`), and executable (or traversable for a directory) (`x`), in any combination. For a directory to be executable implies that we can traverse the directory and list files in it. In addition, a file can be accessible to a single user (`u`), a group of users (`g`), or all other users (`o`). We are considered the owner of all files and subdirectories in our home directory. This means that we have total, unrestricted access to these files. Use the command `ls -l` to check the current protection settings for a file or a directory.

Consider the following example:

```
alice@ubuntu:~$ ls -l program
-rw-r--r--    1 alice     faculty         0 Oct 25 13:15 program
```

There are ten protection bits. Assume that the bits are numbered 1 through 10 from left to right. Then bits 2, 3 and 4 represent the protection for the user (or the owner). The bits 5, 6 and 7 represent the protection settings for the group and the last three bits represent protection for others (not us or those in our group). Now we can read the above example. The file called `program` can be read by `alice`, anyone in the group faculty as well as any other user on the system. However only `alice` has write access to the file. The first bit has special meaning if it is set (see the man page for `chmod` for more on this special bit).

Consider another example:

```
alice@ubuntu:~$ ls -l wideopen
-rwxrwxrwx    1 alice     faculty         0 Oct 25 13:23 wideopen
```

Everyone on the system has read, write and execute access to the file named `wideopen`. Suppose we want to remove write access from all users except the owner of the file. Then the owner of the file (`alice`) will use the following command.

```
alice@ubuntu:~$ chmod g-w,o-w wideopen
alice@ubuntu:~$ ls -l wideopen
-rwxr-xr-x   1 alice     faculty         0 Oct 25 13:23 wideopen
```

Note that `+` adds access and `-` removes access. See the man page for `chmod` for more details.

Here is an example of protecting a directory from all other users.

```
alice@ubuntu:~$ chmod g-rwx,o-rwx myhw
alice@ubuntu:~$ ls -l myhw
drwxr------   1 alice     faculty         1024 Oct 25 13:23 myhw
```

To make a file executable by all users, use the `chmod` command:

`chmod +x filename`

This is useful for creating our own commands. See Section 4.3 for more on how to create our own commands.

Note that running the command `chmod` on a directory only changes the permission on that directory but does not descend into it recursively to change permissions on all files and subdirectories inside the directory. In order to do that, use the recursive option (`-R`). For example,

`chmod -R g+rw project`

allows the group to have read and write access on all files in the `project` directory.

2.5.2 File ownership

The command `chown` allows the user to change the ownership of a file they own to another user (and group). To use `chown`, a user must have the privileges of the target user. In most cases, only *root* can transfer ownership of a file to another user.

The reason for this restriction is that it would cause security problems if allowed to unprivileged users. Here is an example: If a system has disk quotas enabled, Alice could create a world-writable file under a directory accessible only by her (so no one else could access that world-writable file in the directory), and then run chown to make that file owned by another user Bill. The file would then count under Bill's disk quota even though only Alice can use the file.

However, `chown` is an important operation for the *root* user. For example:

`chown alice file1`

Changes the owner of `file1` to *alice*, but the group is left unchanged.

`chown alice:students file2`

Changes the owner of `file2` to *alice*, and the group to *students*.

```
chown -R alice:students /home/alice
```

Changes recursively the owner of all files in the directory `/home/alice` to *alice*, and the group to *students*.

2.6 Other Common file system operations

2.6.1 Packing up and backing up our files

Archiving files with `tar`

The `tar` command is very useful in bundling up our files and directories. Suppose we want to bundle up the entire directory `cpsc-153` under our home directory. We would use the following command:

```
tar cvf cpsc-153.tar cpsc-153
```

This creates a file, often called a tarball, that contains the entire `cpsc-153` folder along with any subdirectories inside it recursively. The option `c` stands for create, the option `v` for verbose and the option `f` for the name of the tarball to follow.

Then we can copy the tarball to another location on our system, to another system entirely, to a USB drive, to cloud storage, or you could email it to someone. Suppose we have a tarball that we want to unpack. Use the following command:

```
tar xvf cpsc-153.tar
```

Here the option `x` stands for extract.

If we want to list the table of contents for a tarball without extracting any files, use the `t` option.

```
tar tvf cpsc-153.tar
```

Compressing files with `gzip`

The command `gzip` can be used to compress files in order to save space. For example:

```
gzip *.data
```

compresses all files with extension `.data` in the current directory. It adds the extenson `.gz` to files it compresses. To uncompress the files, use:

```
gunzip *.data.gz
```

The `gzip` command is often combined with `tar` by using the `z` option to tar. So we can use:

```
tar czvf cpsc-153.tar.gz cpsc-153
```

to create a tarball that is also compressed. The convention is to name the compressed tarball with the extension `.tar.gz`. To unpack a compressed tarball, we would use the `z` option as well. For example:

```
tar xzvf cpsc-153.tar.gz
```

Another common extension for compressed tar files is `.tgz`. For example:

```
tar czvf cpsc-153.tgz cpsc-153
```

Compressing files with zip

Zip is commonly available compression and archive tool that is available on all operating systems. For example, to archive and compress a directory named `cpsc-153`, we would use:

```
zip -r cpsc-153.zip cpsc-153
```

To unpack an uncompress it, we would use:

```
unzip cpsc-153.zip
```

Note that zip does not preserve all the metadata in Linux and is thus not a preferred way to archive and compress. In the context of Linux file systems, "metadata" is information about a file: who owns it, permissions, file type (special, regular, named pipe, etc) and which disk-blocks the file uses. That's all typically kept in an on-disk structure called an inode, which is short for *index node*. The `tar` command preserves all such metadata and is thus preferred on Linux file systems.

Compressing files with bzip2

The `bzip2` program is another compression program that often does better compression and is becoming quite popular on the Internet. To use it with `tar`, use the `j` option. Here are some example usages.

```
tar cjvf cpsc-153.tar.bz2 cpsc-153
```

The convention is to name the bzipped tarball with the extension .tar.bz2. To unpack a bzipped tarball, we would use the `j` option as well. For example:

```
tar xjvf cpsc-153.tar.bz2
```

Why all these various compression and archiving formats?

To learn more about the various archiving and compressions tools, see the article here:
`https://itsfoss.com/tar-vs-zip-vs-gz/`, which gives a nice explanation and comparison.

Backing up our files

We can backup our home directory using tar and gzip as follows:

```
tar czvf /tmp/home.tar.gz ~
```

The above command creates a compressed tarball of our home directory in the temporary directory of the system. See Section 4.13.8 for a better way of backing up our files.

2.6.2 Checking disk usage

The command `du` is handy in determining how much space is used by a directory and its subdirectories. For example:

```
alice@ubuntu:~/C-examples$ du doublyLinkedLists
220     doublyLinkedLists/bad
100     doublyLinkedLists/library
172     doublyLinkedLists/generic
548     doublyLinkedLists
alice@ubuntu:~/C-examples$
```

By default, `du` reports sizes in units of Kilobytes (1024 bytes). We can ask for human readable units by using the `-h` option. For example:

```
alice@ubuntu:~/C-examples$ du -h doublyLinkedLists
220K    doublyLinkedLists/bad
100K    doublyLinkedLists/library
172K    doublyLinkedLists/generic
548K    doublyLinkedLists
alice@ubuntu:~/C-examples$
```

If we just want the sum total usage of the directory, use the `-s` option. For example:

```
alice@ubuntu:~/C-examples$ du -h -s doublyLinkedLists
548K    doublyLinkedLists
alice@ubuntu:~/C-examples$
```

2.6.3 Locating files in the system

Use the command `locate <substring>` to find all files in the system whose names contains the string `<substring>`. An example: Suppose we want to find a file that has the string "duck" in its name. So we type in the following command to get a list of all files in the system that have "duck" in its name.

```
alice@ubuntu:~$ locate duck
/home/JimConrad/CPSC-221/projects/p1/solution/ducks-animation
/home/JimConrad/CPSC-221/projects/p1/solution/ducks-animation/README
/home/JimConrad/CPSC-221/projects/p1/solution/ducks-animation/TrafficAnimation.java
/home/JimConrad/CPSC-221/projects/p1/solution/ducks-animation/sun.png
/home/alice/CPSC-221/duck.png
/usr/share/kde4/services/searchproviders/duckduckgo.desktop
/usr/share/kde4/services/searchproviders/duckduckgo_info.desktop
/usr/share/kde4/services/searchproviders/duckduckgo_shopping.desktop
alice@ubuntu:~$
```

The `locate` command accesses a database of names of all files on the system to perform the search. That is what makes it fast. The database is usually updated once a day automatically by the system.

If we provide a generic substring, then `locate` may find hundreds or even thousands of files. In this case, it is handy to pipe the output to `grep` to filter out the results of interest (more on pipes in a later module!). For example (note that `|` is the character that represents the pipe connecting the two commands):

```
alice@ubuntu:~$ locate duck | grep alice
/home/alice/CPSC-221/duck.png
alice@ubuntu:~$
```

2.6.4 Finding files in our home directory

The `find` command can be used to find files whose names contain the specified substring. The command `find` works recursively down the directory tree from the specified starting point. Suppose that we want to find all the files named `core` in our home directory. We would use `find` as follows (with example output):

```
alice@ubuntu:~$ find ~ -name "core" -print
/home/alice/public_html/teaching/430/lab/project-ideas/l2h10619/core
/home/alice/res/now/zpl/examples/core
/home/alice/res/bob-katherine/backsearch/core
/home/alice/.gnome-desktop/core
```

Note that `find` is much slower than `locate` since it is actually traversing the directory tree to find the files whereas `locate` uses a pre-built database. However `find` can perform many other functions recursively on a directory tree that `locate` cannot.

2.6.5 Using `find` for useful tasks

The command `find` is a powerful tool that can be used for many tasks that operate on a whole directory.

For example, we can find all Java source files in our home directory (represented by the tilde `~`) with the following use of the `find` command.

```
alice@ubuntu:~$ find ~ -name "*.java" -print
/home/alice/.java
/home/alice/CreateDataFiile.java
/home/alice/ListFiles.java
...
```

However, note that it also matches the file named `.java`, which isn't a valid Java source file. We can skip that by insisting that the pattern must contain at least one letter before the dot. See below for an example.

```
alice@ubuntu:~$ find ~ -name "*[A-Za-z].java" -print
/home/alice/CreateDataFiile.java
/home/alice/ListFiles.java
...
```

Another example: we can find all files that were modified in our home directory in the last 30 minutes with the following use of the `find` command.

```
alice@ubuntu:~$ find ~ -mmin -30 -print
/home/alice
/home/alice/.kde/share/config
/home/alice/.kde/share/config/kdesktoprc
/home/alice/.kde/share/apps/kalarmd
/home/alice/.kde/share/apps/kalarmd/clients
/home/alice/.Xauthority
/home/alice/.viminfo
/home/alice/public_html/teaching/handouts
/home/alice/public_html/teaching/handouts/cs-linux.tex
/home/alice/public_html/teaching/handouts/.cs-linux.tex.swp
/home/alice/public_html/teaching/253/notes
/home/alice/res/qct/pds/sect7/s7ods_orig.tex
/home/alice/res/qct/pds/sect7/s7ods_hash.tex
/home/alice/.alice-calendar.ics
alice@ubuntu:~$
```

One of the most powerful uses of `find` is the ability to execute a command on all files that match the pattern given to `find`. Here are some examples:

- Remove all files with the name `a.out`, starting in the current directory and going in all subdirectories recursively.

  ```
  find . -name "a.out" -exec /bin/rm -f {} \;
  ```

  The characters {} represents a field that gets filled in by the pathname of each instance of a file named `a.out` that the command `find` finds. In the above we have to escape the semicolon so that the shell does not process it. Instead the find command needs to process it, as it represents the end of the command after the `-exec` flag.

- Find all files in our home directory with the extension `.c` and remove execute access from those files.

  ```
  find ~ -name "*.c" -exec chmod -x  {} \;
  ```

- Find all files in our home directory with the extension `.c` or `.h` and remove execute access from those files.

```
find ~ -name "*.[c|h]" -exec chmod -x {} \;
```

The expression `*.[c|h]` is an example of a regular expression. Regular expression are a powerful way of expressing a set of possibilities. See `man 7 regex` for the full syntax of regular expressions.

- Compress all regular files in the directory `dir1`. This can be handy in saving space in our account.

  ```
  find dir1 -type f -exec gzip {} \;
  ```

- Uncompress all regular files in the directory `dir1`.

  ```
  find dir1 -type f -exec gunzip {} \;
  ```

2.7 Devices and partitions

2.7.1 Introduction

One of the neat ideas in Linux (and other similar systems) is that it treats all peripheral devices (such as disk drives, keyboard, console, memory, mouse etc) as a file! These files are contained in the `/dev` directory. For example, a program (with appropriate permissions) can simply read the file `/dev/sda` and it would access the first hard drive on the system. The operating system converts read operations on the `/dev/sda` file into appropriate hardware commands to access the physical hard drive. Thus the program does not need to know about the specific commands needed to talk to the device in question. The system does this behind the scenes using device driver software (also part of the operating system).

For example, here is the listing of all drives on a system that has three physical drives. The drive `/dev/sda` is divided into two partitions, while drives `/dev/sdb` and `/dev/sdc` have one partition each. Partitions allow sections of the storage device to be isolated from each other. This allows storage space to be dedicated for specific areas so that running out of space in one area does not affect the functionality of another area.

```
alice@ubuntu:~$ ls -l /dev/sd*
brw-rw---- 1 root disk 8,  0 Nov 23 05:56 /dev/sda
brw-rw---- 1 root disk 8,  1 Nov 23 05:56 /dev/sda1
brw-rw---- 1 root disk 8,  2 Nov 23 05:56 /dev/sda2
brw-rw---- 1 root disk 8, 16 Nov 23 05:56 /dev/sdb
brw-rw---- 1 root disk 8, 17 Nov 23 05:56 /dev/sdb1
brw-rw---- 1 root disk 8, 32 Nov 23 05:56 /dev/sdc
brw-rw---- 1 root disk 8, 33 Nov 23 05:56 /dev/sdc1
```

Note the “b” in front of the listing: that denotes that this device operates in blocks of data instead of a stream of characters. A terminal device acts as a stream of characters.

When we login, we get a terminal device to which the characters we type and receive are sent. The tty command tells which terminal we are using. For example:

```
alice@ubuntu:~$ whoami
alice
alice@ubuntu:~$ tty
/dev/pts/1
alice@ubuntu:~$ ls -l /dev/pts/1
crw------- 1 alice tty 136, 1 Nov 27 22:25 /dev/pts/1
alice@ubuntu:~$ date > /dev/pts/1
Mon Nov 27 22:25:22 MST 2022
```

Note that only we can read from or write to our terminal. It is often convenient to refer to our terminal in a generic way. The device `/dev/tty` is a synonym for our login terminal, whatever terminal we are actually using.

```
alice@ubuntu:~$ date > /dev/tty
Mon Nov 27 22:26:24 MST 2022
```

Sometimes we want to run a command but don't care what output is produced. We can redirect the output to a special device `/dev/null`, which causes the output to be thrown away. One common use is to throw away regular output so the error messages are more easily visible. For example, the `time` command reports the CPU usage on the standard error, so we can time commands that generate lots of output by redirecting their standard output to `/dev/null`. See below.

```
alice@ubuntu:~$ time sort /usr/share/dict/linux.words  > /dev/null

real    0m0.147s
user    0m0.439s
sys     0m0.021s
```

2.7.2 Creating and working with file systems

The term file system is somewhat overloaded and its meaning depends upon context. In one context, file system refers to the directory hierarchy that a user interacts with when using command line tools or browsing for a file. However, file system also refers to the software layer that manages how files are stored and retrieved from physical storage devices such as hard disk drives, USB thumb drives, solid state drives, etc. In this section we will use the term Linux file system to refer to the Linux directory hierarchy and we will use the term disk file system to the software layer that manages files.

The Linux file system is organized as a tree. The top of the tree is referred to as the root of the file system and is represented by a single forward slash (/). Working with the Linux file system is accomplished by using either absolute paths that begin with the root (/) and specify the full path to the file or directory, or by using relative paths that are specified in relation to the current position (directory) within the file system.

The Linux file system is made up of one or more disk file systems. These disk file systems are attached to the Linux file system by creating a mapping between a directory and a storage device such as a disk drive. This process is referred to as mounting.

2.7.3 df

The `df` command is used to examine the available free space on all mounted disk file systems. The -h option is commonly utilized to format the output amounts in KB, MB, GB, and TB. Figure 2.2

Figure 2.2: Using the `df` command

```
alice@ubuntu:~$ df -h
Filesystem           Size  Used Avail Use% Mounted on
/dev/mapper/cl-root   17G  5.6G   12G  33% /
devtmpfs             2.0G     0  2.0G   0% /dev
tmpfs                2.0G     0  2.0G   0% /dev/shm
tmpfs                2.0G  9.1M  2.0G   1% /run
tmpfs                2.0G     0  2.0G   0% /sys/fs/cgroup
tmpfs                2.0G   20K  2.0G   1% /tmp
/dev/sda1           1014M  207M  808M  21% /boot
tmpfs                397M  4.0K  397M   1% /run/user/42
tmpfs                397M   48K  397M   1% /run/user/1000
tmpfs                397M     0  397M   0% /run/user/0
```

shows sample output from the `df` command.

As mentioned previously, the Linux file system is a tree. The `df` command shows the disk file systems and where they are attached (mounted) on the tree. Each row in the output displays size of the disk file system and how much space is available. The `df` command includes stats on 'virtual' block devices that only exist in RAM (Random Access Memory - the main memory for a system). These include the `devtmpfs` and `tmpfs` file systems.

2.7.4 lsblk

The `lsblk` command is used to list all the storage devices (block devices) connected to the system. It shows the device name as well as any partitions that may have been created on the device. In addition, it shows the capacity (size) of the device, the device type, and the mount point if it has been attached to the Linux file system.

```
alice@ubuntu:~$ lsblk
NAME          MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
sda             8:0    0   20G  0 disk
|--sda1         8:1    0    1G  0 part /boot
|--sda2         8:2    0   19G  0 part
   |--cl-root 253:0    0   17G  0 lvm  /
```

```
   |--cl-swap 253:1    0    2G  0 lvm  [SWAP]
sdb             8:16   0    8G  0 disk
sr0            11:0    1  1.7G  0 rom
```

The example output shows two disks attached to the Linux system (`sda` and `sdb`). The first disk, `sda`, has two partitions allocated (`sda1` and `sda2`) which contain the root file system (/) and the initial boot file system (`/boot`). The second disk, `sdb`, has no partitions allocated and is not currently mounted as part of the Linux file system.

Notice that the second partition (`sda2`) is further divided into `cl-root` and `cl-swap` using LVM (Logical Volume Manager) and that these LVM devices are in fact the devices that are mounted. LVM is a more powerful and flexible mechanism for managing storage space on block devices, but it is beyond the scope of this guide.

2.7.5 fdisk

The `fdisk` command is an interactive tool that is used to create partitions on a block device. Examples of such devices include hard disk drives, solid state drives, and USB thumb drives. Partitions allow sections of the storage device to be isolated from each other. This allows storage space to be dedicated for specific areas so that running out of space in one area does not affect the functionality of another area.

Partitions also allows for multiples disk file system types to be used on a single storage device. The `fdisk` command allows the user to create/remove partitions, allocate the size of these partitions, and specify the type of file system used on each partition. It needs to be run in the administrative mode.

Figure 2.3: Creating a new partition using `fdisk`

```
alice@ubuntu:~$ sudo fdisk /dev/sdb
[sudo] password for alice:
Welcome to fdisk (util-linux 2.23.2).

Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Device does not contain a recognized partition table
Building a new DOS disk label with disk identifier 0xb73c863a.

Command (m for help): n
Partition type:
   p   primary (0 primary, 0 extended, 4 free)
   e   extended
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-16777215, default 2048):
Using default value 2048
Last sector, +sectors or +size{K,M,G} (2048-16777215, default 16777215):
Using default value 16777215
Partition 1 of type Linux and of size 8 GiB is set

Command (m for help): p

Disk /dev/sdb: 8589 MB, 8589934592 bytes, 16777216 sectors
Units = sectors of 1 * 512 = 512 bytes
Sector size (logical/physical): 512 bytes / 512 bytes
I/O size (minimum/optimal): 512 bytes / 512 bytes
Disk label type: dos
Disk identifier: 0xb73c863a

   Device Boot      Start         End      Blocks   Id  System
/dev/sdb1            2048    16777215     8387584   83  Linux

Command (m for help): w
The partition table has been altered!

Calling ioctl() to re-read partition table.
Syncing disks.
alice@ubuntu:~$
```

In Figure 2.3, see an example `fdisk` session that creates a single partition named `sdb1`. It is named `sdb1` because it is partition `1` on the `sdb` block storage device. Below, we describe step by step how to create a new partition. The figure shows the expected output.

1. Startup `fdisk` in the administrative mode with the command `sudo fdisk /dev/sdb`.
2. Within `fdisk`, use the n command to create a new partition.
3. Then we enter p to specify that the new partition will be a primary partition.
4. Next we select 1 to specify the partition number.
5. The next prompt will ask for the first sector. Just press `Enter` to accept the default value.
6. After that it will ask for the last sector. Just press `Enter` to accept the default value again.
7. Now, use the p command to print the partitions to verify that it is what we are expecting before we modify the disk.
8. Finally, enter the w command to write the partitions out. This step actually creates the new partition.

Each partition is tagged for the type of disk file system that will be used on it. In this case partition `1` is tagged with file system Id `83`, meaning that it will be used for a Linux compatible disk file system. For Linux type partitions, `83` is a generic system id that can be use on partitions that support several different Linux compatible disk file systems including *ext2*, *ext3*, *ext4*, *xfs*, and *reiserfs*.

Figure 2.4: Common `fdisk` commands

```
Command action
   d   delete a partition
   l   list known partition types
   m   print this menu
   n   add a new partition
   p   print the partition table
   q   quit without saving changes
   t   change a partition's system id
   w   write table to disk and exit
```

Figure 2.4 shows the most commonly used `fdisk` commands. The list of options can be accessed by pressing '`m`' within the fdisk console.

2.7.6 mkfs

The `mkfs` tool creates the disk file system on a partition of the block storage device. Note that a partition is just that - a separate physical space on a disk labeled as a partition. The file system is written inside the partition and is contained in it. Linux supports a large number of file systems, each with their own pros and cons. The various disk file systems try to balance read performance, write performance, large file performance, small file performance, reliability, and recoverability. There is not a perfect solution, which is why there are so many options. The *xfs* disk file system is the default for **Red Hat Enterprise**Linux while *ext4* is the default for Ubuntu and Fedora Linux. Both provide a good balance for workloads typically associated with a development workstation.

Figure 2.5: Example mkfs command creating ext4 file system on /dev/sdb1

```
alice@ubuntu:~$ sudo mkfs -t ext4 /dev/sdb1
mke2fs 1.42.9 (28-Dec-2022)
Filesystem label=
OS type: Linux
Block size=4096 (log=2)
Fragment size=4096 (log=2)
Stride=0 blocks, Stripe width=0 blocks
524288 inodes, 2096896 blocks
104844 blocks (5.00%) reserved for the super user
First data block=0
Maximum filesystem blocks=2147483648
64 block groups
32768 blocks per group, 32768 fragments per group
8192 inodes per group
Superblock backups stored on blocks:
        32768, 98304, 163840, 229376, 294912, 819200, 884736, 1605632

Allocating group tables: done
Writing inode tables: done
Creating journal (32768 blocks): done
Writing superblocks and filesystem accounting information: done
```

The example in Figure 2.5 shows how the `mkfs` command can be used to create an *ext4* disk file system on partition 1 of the sdb block storage device. The `-t` option is used to specify the type of disk file system to create. Out of the box, **Red Hat Enterprise**Linux supports the following disk file systems: *btrfs*, *cramfs*, *ext2*, *ext3*, *ext4*, *fat*, *minix*, *msdos*, *vfat*, and *xfs*. The most common file systems that we will deal with are *ext4*, *xfs*, and *btrfs*.

2.7.7 mount

The `mount` command performs two different functions. The primary use of the mount command is to attach (mount) a block storage device to the Linux file system (directory hierarchy). This is accomplished by designating a directory within the hierarchy to be the mountpoint. Then the mount command is used to create a mapping between that mountpoint and the disk file system on a block storage device. This is referred to as mounting a drive.

The example in Figure 2.6 demonstrates how to use the `mkdir` command to create a workspace

Figure 2.6: Example mount command

```
alice@ubuntu:~$ mkdir ~/workspace
alice@ubuntu:~$ sudo mount /dev/sdb1 ~/workspace
```

directory in the user's home drive. This will be the mountpoint. Then mount is used to create a mapping between the workspace directory and the *ext4* disk file system on `/dev/sdb1`.

When executed without command line options, the `mount` command displays all of the block storage devices, both physical and virtual, that are currently attached (mounted) to the Linux file system. In addition, it displays the disk file system type of each block device and any disk file system specific options that were used. On modern systems this output can be quite verbose. The output in Figure 2.7 shows that the *ext4* disk file system on `sdb1` has been mounted to the `/home/alice/workspace` directory in the Linux file system.

Figure 2.7: Example output from mount command (trimmed)

```
sysfs on /sys type sysfs (rw,nosuid,nodev,noexec,relatime,seclabel)
proc on /proc type proc (rw,nosuid,nodev,noexec,relatime)
...
/dev/mapper/cl-root on / type xfs (rw,relatime,seclabel,attr2,inode64,noquota)
/dev/sda1 on /boot type xfs (rw,relatime,seclabel,attr2,inode64,noquota)
/dev/sdb1 on /home/alice/workspace type ext4 (rw,relatime,seclabel,data=ordered)
```

To disconnect (unmount) a block storage device, use the `umount` command. That is not a typo. The command to unmount an attached storage device is called umount. See Figure 2.8.

Figure 2.8: Example umount command

```
alice@ubuntu:~$ sudo umount ~/workspace
```

Storage devices mounted using the mount command are not persistent, meaning that if we reboot the computer they will no longer be attached and must manually be mounted again. A file called the file system table (`fstab`) is used to determine which storage devices are mounted automatically at boot. To add a storage device to the file system table, we will need to open `/etc/fstab` in a text editor (such as `gedit` or `vim`) and add a new entry for the storage device. See Figure 2.9 for details.

Each line in the fstab is white-space delimited (space/tab). The first field is the block storage device containing the disk file system. The second field is the mountpoint (directory). The third field is the specific type of disk file system. The fourth is the disk file system options. The last two relate to when the block storage device is mounted, whether it is required to boot the system, and how frequently it is checked for errors.

Figure 2.9: Example adding storage device to the /etc/fstab file

```
   sudo gedit /etc/fstab
>> Add the following to the end of the file <<
/dev/sdb1               /home/alice/workspace   ext4    defaults        0 0
```

Chapter 3

Advanced User's Guide

3.1 Customizing our shell and improving productivity

The `bash` shell is extensively customizable and fully programmable. In the following subsections, you will see some common ways of customizing the `bash` shell.

3.1.1 Startup or run control (rc) files

For `bash` users, there are two files of interest: `.bashrc` and `.bash_profile`. These are (or should be) located in our home directory. The `.bash_profile` is sourced for each interactive login session, while the .bashrc is sourced for *all* interactive sessions. Normally `.bashrc` contains most of the setup and is invoked from within `.bash_profile`. These files contain settings for a variety of environmental variables, which are visible to all applications.

3.1.2 Changing our shell prompt

The prompt environmental variable is `PS1`. This is configured in the `.bashrc` file. For example, the following sets the prompt:

```
alice@ubuntu:~$ export PS1="[\u@\h]\w $"
```

There are a number of control sequences defined inside the `PS1` variable: for example, `\u` is the username, `\h` is the hostname of the system and `\w` is the current directory. The `bash` man page has many more – search the bash man page for PS1 for a complete list.

3.1.3 Setting the path: how the shell finds programs

When the user types in the name of a program to run, the shell searches a list of directories for the program and invokes the first such program found. The list of directories to search is specified by the `PATH` environment variable.

Normally we don't want to override the system settings; instead, append to the current path as shown below,

```
alice@ubuntu:~$ export PATH="$PATH:/extra/dir"
```

with new entries delimited with a colon. The setting for the PATH variable goes in the .bash_profile or .bashrc file.

Due to security concerns, typically the current directory is not in our path. So we have to specify the path using the dot notation to run programs in the current directory (as shown below):

```
./myprog
```

While that works, we can add the current directory to the PATH so that we do not have to prefix each time with the ./ prefix. Here is the setting.

```
export PATH=$PATH:.
```

Make sure to add . to the end of the path (for security reasons). We add this line at the end of the .bashrc file in our home directory.

3.1.4 Aliases

Aliases are defined with alias <rhs>=<cmd> so that <cmd> is substituted for <rhs> – not unlike a macro substitution. Aliases are usually placed in .bashrc file in our home directory. Some examples:

```
alias rm='rm -i'
alias vi='gvim'
alias ls='ls -F --color=auto'
alias lab5='cd ~/cpsc-121/lab05'
```

Then typing the command lab5 takes us to the directory ~/cpsc-121/lab05. If for some reason we need to remove an alias temporarily in the login session, then use the unalias command. For example: unalias vi.

3.1.5 Customizing ls using aliases

Common usage of the ls command:

```
alias ll="ls -l"
alias la="ls -a"
```

Occasionally, we will see

```
alias l=ls
```

We can also alias commands to extended versions of themselves: alias ls="ls -F --color=auto" This forces ls to color-code files by type, if output is going to a terminal. The colors used by ls can be customized by setting the LS_COLORS environment variable in the .bashrc file. For example, the following is a good setting for terminals with white or light background.

```
# setup for color ls
LS_COLORS='no=00:fi=00:di=01;34:ln=01;35:pi=40;32:so=01;40;35:bd=40;33;01:cd=40;33;01:\
```

```
or=40;31;01:ex=07;32:*.class=01;31:*.tar=01;31:*.tgz=01;31:*.arj=01;31:\
*.taz=01;31:*.lzh=01;31:*.zip=01;31:*.z=01;31:*.Z=01;31:*.gz=01;31:*.deb=01;31:\
*.jpg=01;35:*.gif=01;35:*.bmp=01;35:*.ppm=01;35:*.tga=01;35:*.xbm=01;35:*.xpm=01;35:\
*.tiff=01;35:*.mpg=01;37:*.avi=01;37:*.gl=01;37:*.dl=01;37:*.tex=01;31:'
export LS_COLORS
```

3.1.6 Enhancing cd using a stack

The command `cd` does not allow for a lot of customization, but there are two built-ins that can be used to extend the functionality of `cd`.

- `pushd <new_dir>` will push the current directory onto the directory stack and `cd` to the new directory. This can be aliased as `alias pd=pushd`.
- `popd` is the corresponding pop operation. This will pop the top directory on the stack and `cd` to it. This can be aliased as `alias bd=popd`.

These are useful when we need to traverse a number of directories but need to return to our current location when done.

If we just need to toggle between two directories, then `cd -` will do the trick.

3.1.7 Repeating and editing previous commands

The command `history` lists previous commands with numbers. We can run any previous command by typing `!` followed by the command number. For example, if the output of the `history` command is as follows:

```
181     ls
182     ls
183     cat /etc/shells
184     cat /etc/hosts
185     w
```

Then we can run the command 183 again as follows:

```
alice@ubuntu:~$ !183
cat /etc/shells
/bin/bash
/bin/sh
/bin/ash
/bin/bsh
/bin/bash2
/bin/tcsh
/bin/csh
```

```
/bin/ksh
/bin/zsh
alice@ubuntu:~$
```

Alternately, we can use `!` followed by a prefix of the command and the shell searches for and executes the last command that started with the given prefix. In the above example, saying `!cat`, will result in the command `cat /etc/hosts` to be executed.

Typing two bang characters is a short-cut for repeating the last command.

```
alice@ubuntu:~$ date
Mon Oct 25 14:29:19 MDT 2022
alice@ubuntu:~$ !!
date
Mon Oct 25 14:29:23 MDT 2022
alice@ubuntu:~$
```

We can use the arrow keys (↑ and ↓) to go up and down the list of commands that we typed into the shell until we reach the desired command. We can use backspace and arrow keys (← and →) to edit the command. Press the ENTER key to execute the command.

We can also set the editor mode for the `bash` shell to be Vi (or Emacs) by placing the following command in our `.bash_profile` file.
`set -o vi`
Vi (actually Vim, which is a super-set of Vi) and Emacs are two powerful text editors. We will be learning Vim in the next chapter.

If we wish to use the more powerful editing commands from `vi`, press the ESC key. Now we can use the `vi` search command `/string` to search for a previous command containing that `string`. Once we get to the desired command we can edit it further using the standard `vi` editing commands. After we are done editing, we just press ENTER to execute the command.

Another common technique is to grep through the history to find the command we had typed earlier.

```
alice@ubuntu:~/cpsc121/examples$ history | grep javac
 8202  javac
 8206  javac WebStats.java
 8211  javac -O WebStats.java
10082  history | grep javac
alice@ubuntu:~/cpsc121/examples$
```

Here the vertical bar symbol `|` is the pipe symbol that connects the two commands `history` and `grep` together. This is an example of object composition!

On a GUI desktop we can use the mouse to cut and paste previous commands.

3.2 Other useful commands

3.2.1 Text-based mail

If we want to send mail to any user then we can use `mail <user_email>`. It will then prompt us for a subject of the mail message. We can leave it blank if we wish. After we have finished typing the message, press `Ctrl-d` to send it. While typing the message we can only edit the current line. However we can invoke the `vim` editor any time by typing ~`v` on a line by itself. This puts our current draft in a temporary file and allows us to edit the message using our default editor. After we are finished typing, exit `vim` and press `Ctrl-d` ends the message.

A mail program with a more convenient interface is `mutt`. It can be installed on all Linux systems. The `mail` and `mutt` programs are useful in scripts, which we will learn more about in Chapter 4.

3.2.2 Changing our personal information

Use the command `chfn` to change our personal information like phone number, office location, our real name etc. The command `chfn` will ask for our password before letting us change our personal information. Use of this command may be restricted on some servers.

3.2.3 Changing our login shell

We can change our default shell by using the following command:

`chsh <newshell>`

where `<newshell>` must be the full path name of one of the shells listed in the file `/etc/shells`. The command `chsh` will ask for our password and then let us specify a new shell.

3.2.4 Spell checking

Use the command `ispell <filename>` to run an interactive spelling checker on a file.

The command `spell` is a non-interactive spelling checker. It just prints out all misspelled words from the input file of words. Hence if we check a file with `spell` and there is no output, then no spelling mistakes were found in the given file. No news is good news!

Both spell and ispell use a dictionary that is located in the file `/usr/share/dict/words`. Here is an example using `spell`.

```
alice@ubuntu:~/simple$ cat test1
dada
dad
mom
father
sun
simpel
```

```
alice@ubuntu:~/simple$ spell test1
dada
simpel
alice@ubuntu:~/simple$
```

The `spell` program by itself is not very useful. However it is useful if used from inside shell scripts (these are covered in Chapter 4).

3.2.5 Watching a command

The command `watch` executes a program periodically, showing output full screen. By default, the program is run every 2 seconds; use `-n` or `--interval` to specify a different interval. For example, we can use watch to see how much memory is being used on our system every 5 seconds with the following command.

```
watch -n 5 free -h
```

3.3 Filters: cool objects

Programs like `sort`, `tail`, `wc`, `grep`, `uniq` read some input, perform some simple transformation on it and write some output. Such programs are called *filters*. Here we will briefly discuss some other well known filters: `tr` for character transliteration, `comm` for comparing files.

The two most used filters are `sed`, which stands for `stream editor` and `awk`, named after its three authors. Both of these are generalizations of grep. Most of this material is borrowed from "Programming in the UNIX Environment" by Kernighan and Pike [1].

3.3.1 Character transliteration with `tr`

The `tr` command transliterates the characters in its input. A common use is for case conversion. For example:

```
cat doc1.txt | tr a-z A-Z
```

converts lower case to upper case, and the following does the reverse:

```
cat doc2.txt | tr A-Z a-z
```

The following example prints one word per line from a normal English text file, where word is any sequence of upper/lower case letters and the apostrophe.

```
tr -cs "A-Za-z'" "[\n*]"
```

3.3.2 Comparing sorted files with `comm`

Given two sorted input files `f1` and `f2`, `comm` prints three columns of output: lines that occur in `f1` only, lines that occur only in `f2` and lines that occur in both files. Any of these columns can be suppressed by an option.

With no options, `comm` produces three-column output. Column one contains lines unique to file f1, column two contains lines unique to file f2, and column three contains lines common to both files.

-1 suppress column 1 (lines unique to file f1)

-2 suppress column 2 (lines unique to file file f2)

-3 suppress column 3 (lines that appear in both files)

For example,

`comm -12 f1 f2`

prints lines in both files, and

`comm -23 f1 f2`

prints lines that are in the first file but not in the second. This is useful for comparing directories. Suppose we have two directories `dir1` and `dir2` that have many files in common but just a few differences. We are interested in the files that are in `dir1` but not in `dir2`. Here is how to accomplish that.

```
alice@ubuntu:~$ ls dir1
f1  f2  f3  f4  f5  f6  f8
alice@ubuntu:~$ ls dir2
f1  f2  f3  f4  f5  f6  f7  f9
alice@ubuntu:~$ ls dir1 > dir1.list
alice@ubuntu:~$ ls dir2 > dir2.list
alice@ubuntu:~$ comm -23 dir1.list  dir2.list
f8
alice@ubuntu:~$ comm -13 dir1.list  dir2.list
f7
f9
alice@ubuntu:~$
```

3.3.3 Stream editing with sed

The basic idea in using sed (Stream EDitor) is to read lines one at a time from the input files, apply commands from the list (placed within single quotes), in order, to each line and write the edited output to the standard output. The syntax looks like the following:

`sed` '*list of ed commands*' *filenames*

Below we will provide a series of useful usages.

- For example, we can change all occurrences of Alice to Amber in a file with the following command:

  `sed 's/Alice/Amber/g' TicTacToe.java > output`

  However the above does not change the file. We have to save the output and rename it to the actual file name, as shown below. Make sure to check the output to verify that the changes have occurred correctly before renaming it!

```
mv output > TicTacToe.java
```

- Here is an example that indents its input one tab stop; it is handy for moving something over to fit better for printing.

  ```
  sed 's/^/\t/' file1
  ```

  where `\t` represents the tab character. We can then just pipe the output to `lpr` to send it to the printer.

  ```
  sed 's/^/\t/' file1 | lpr
  ```

- The following quits after printing the first 3 lines.

  ```
  cat file2 | sed 3q
  ```

- The `sed` program automatically prints each processed line. It can be turned off by the `-n` option so that only lines explicitly printed with the `p` command appear in the output. For example,

  ```
  sed -n '/pattern/p'
  ```

  does the same job as `grep`.

- The following adds a newline to the end of each line, thus double spacing it.

  ```
  sed 's/$/\n/'
  ```

- Some more examples.

  ```
  sed -n '20,30p'  print lines 20 through 30
  sed '1,10d'      delete  lines 1 through 10
  sed '1,/^$/d'    delete up to and including the first blank line
  sed '$d'         delete last line
  ```

3.3.4 String processing with `awk`

The `awk` program is more powerful than even `sed`. The language for `awk` is based on the C programming language. The basic usage is:

```
awk 'program' filenames..
```

but the program is a series of patterns and actions to take on lines matching those patterns.

pattern { *action* }
pattern { *action* }
...

Here we will touch on some simple uses.

The `awk` program splits each line into *fields*, that is, strings of non-blank characters separated by blanks or tabs. The fields are called $1, $2, ..., `$NF`. The variable $0 represents the whole line.

- Let us start with a nice example. Suppose we look at the output of the `who` command.

```
alice@ubuntu:~$ who
aswapna  pts/0        Dec  2 03:41 (jade.boisestate.edu)
jlowe    pts/1        Nov 26 22:34 (24-116-128-35.cpe.cableone.net)
jcollins pts/2        Nov 29 11:30 (eas-joshcollins.boisestate.edu)
tcole    pts/6        Dec  1 11:01 (meteor.boisestate.edu)
ckrossch pts/10       Nov 29 07:35 (masquerade.micron.com)
drau     pts/11       Nov 30 17:53 (sys-243-163-254.nat.pal.hp.com)
tcole    pts/12       Dec  1 12:15 (meteor.boisestate.edu)
alice     pts/13       Dec  2 04:08 (kohinoor.boisestate.edu)
aswapna  pts/15       Dec  1 23:07 (jade.boisestate.edu)
alice     pts/8        Nov 29 17:38 (kohinoor.boisestate.edu)
cwaite   pts/4        Nov 16 07:27 (masquerade.micron.com)
cwaite   pts/7        Nov 16 07:30 (masquerade.micron.com)
alice     pts/20       Dec  1 15:34 (208--714-14694.boisestate.edu)
alex     pts/21       Nov 16 11:10 (144--650-3036.boisestate.edu)
yghamdi  pts/32       Nov 30 22:43 (24-117-243-152.cpe.cableone.net)
jhanes   pts/22       Nov 18 19:41
twhitchu pts/26       Nov 28 16:17
cwaite   pts/29       Nov 30 09:14 (masquerade.micron.com)
kchriste pts/18       Dec  1 16:23 (sys-243-163-254.nat.pal.hp.com)
njulakan pts/15       Dec  1 22:22
alice@ubuntu:~$
```

Let's say we are interested in the name and time of login only. We can select the first and fifth column using `awk`.

```
who | awk '{print $1, $5}'
```

```
alice@ubuntu:~$ who | awk '{print $5, $1}'
jlowe 22:34
jcollins 11:30
tcole 11:01
ckrossch 07:35
drau 17:53
tcole 12:15
alice 04:08
alice 17:38
cwaite 07:27
cwaite 07:30
alice 15:34
alex 11:10
yghamdi 22:43
jhanes 19:41
twhitchu 16:17
cwaite 09:14
kchriste 16:23
njulakan 22:22
alice@ubuntu:~$
```

Now suppose we want to sort them by the time of login. We can do that with the command.

```
 who | awk '{print $5, $1}' | sort
```

```
alice@ubuntu:~$ who | awk '{print $5, $1}' | sort
04:08 alice
07:27 cwaite
07:30 cwaite
07:35 ckrossch
09:14 cwaite
11:01 tcole
11:10 alex
11:30 jcollins
12:15 tcole
15:34 alice
16:17 twhitchu
16:23 kchriste
17:38 alice
17:53 drau
19:41 jhanes
22:22 njulakan
22:34 jlowe
22:43 yghamdi
alice@ubuntu:~$
```

- Using the `-F` option, the delimiter between fields can be changed to any single character. For example, the `/etc/passwd` file contains basic user account information. each line in the file has a number of fields separated by a colon character. The first field is the name of the user. So if we wanted a sorted list of all users on the system, we can use

  ```
  cat /etc/passwd | awk -F: '{print $1}' | sort
  ```

- `awk` keeps track of line numbers with the variable `NR`. So we can use:

  ```
  awk '{print NR, $0}'
  ```

  to add line numbers to any input stream.

- If we need more control on the formatting, we can use `printf` instead of `print`. The `printf` works like the C `printf` function.

  ```
  awk '{printf "%4d\t%s\n", NR, $0}'
  ```

- Data validation examples:

  – Make sure every line has an even number of fields:
    ```
    cat data | awk 'NF%2 != 0'
    ```

  – Print lines that are longer than 72 characters.
    ```
    cat data | awk 'length($0) > 72'
    cat data | awk 'length($0) > 72 {print "Line", NR, "too long:" substr($0,1,60)}'
    ```

  – The `BEGIN` and `END` patterns are two special patterns. The pattern `BEGIN` allows us to do initialization like printing headers, initializing variables before processing input and `END` lets us do post-processing.

    The following example computes the sum and average of the first column in the input.

```
awk '{s = s + $1}\
     END {print s, s/NR}'
```

Here is an example of using the above construct.

```
alice@ubuntu:~/doublyLinkedLists$ wc -l *.c *.h
     26 Job.c
    133 List.c
     42 main.c
     20 Node.c
    131 TestList.c
     12 common.h
     27 Job.h
     46 List.h
     23 Node.h
    460 total
alice@ubuntu:~/doublyLinkedLists$ wc -l *.c *.h | awk '{print $1}'
26
133
42
20
131
12
27
46
23
460
alice@ubuntu:~/doublyLinkedLists$ wc -l *.c *.h | awk '{print $1}' | awk '{s = s+$1}\
> END {print s, s/NR}'
920 92
alice@ubuntu:~/doublyLinkedLists$
```

`awk` has arrays, full programming language statements and much more. Please see the book on AWK [2] to learn more.

3.4 Processes and pipes

3.4.1 Input-Output redirection

When a command is started under Linux it has three data streams associated with it: standard input (`stdin`), standard output (`stdout`), and standard error (`stderr`). The corresponding file numbers are 0, 1 and 2. Initially all these data streams are connected to the terminal.

A terminal is also a type of file in Linux. Most of the commands take input from the terminal and produce output on the terminal. In most cases we can replace the terminal with a file for either or both of input and output. For example:

```
ls >  listfile
```

puts the listing of files in the current directory in the file `listfile`. The symbol `>` means redirect the standard output to the following file, rather than sending to the terminal.

The symbol `>>` operates just as `>`, but appends the output to the contents of the file `listfile` instead of creating a new file. Similarly, the symbol `<` means to take the standard input for a program from the following file, instead of from a terminal. For example:

```
mutt -s "program status report" mary joe tom < letter
```

mails the contents of the file `letter` to the three users: `mary, joe and tom`. The command `mail` can also be used in place of the `mutt` mailer in the above example without any change.

To redirect error messages (which are sent on `stderr` with file descriptor 2) see the following example.

```
javac BadProgram.java 2> error.log
```

Or if we want both the output and the error messages to go to a file, see the following example.

```
javac BadProgram.java > log 2>&1
```

3.4.2 Processes

Creating and managing processes

The shell can also help us in running multiple programs at a time. Suppose we want to run a word count on a large file but we don't want to wait for it to finish. Then we can say:

```
wc hugefile > wc.output &
```

The ampersand `&` at the end of a command line says to the shell to take this command and start executing it in the background and get ready for further commands on the command line. If we don't redirect the output to a file it will show up on our terminal some time later when the `wc` program is done! The command `jobs` lists all background jobs that we have started from the current shell.

If we start a bunch of processes with the ampersand we can use the `jobs` command to list them. Then we can selectively bring one into the *foreground* by the command `fg %n`, where `n` is the job number as listed by the `jobs` command. We can also suspend a running program with `Ctrl-z` and put it in the *background* with the command `bg`.

Each running program is known as a *process*. The number printed by the shell for each running program is a unique *process-id* that the operating system assigns to the process when it is created. To check for running processes use the `ps` command (`ps` is for process status). A sample session is shown below.

```
alice@ubuntu:~$ date
Mon Oct 25 14:40:52 MDT 2022
alice@ubuntu:~$ wordfreq Encyclopedia.txt  > output &
[1] 19027
alice@ubuntu:~$ jobs
[1]+  Running                 wordfreq Encyclopedia.txt >output &
alice@ubuntu:~$ ps
  PID TTY          TIME CMD
19018 ttyp1    00:00:00 bash
```

```
19027 ttyp1    00:00:00 wordfreq
19033 ttyp1    00:00:02 sort
19034 ttyp1    00:00:00 uniq
19035 ttyp1    00:00:00 sort
19036 ttyp1    00:00:00 wc
19037 ttyp1    00:00:00 ps
alice@ubuntu:~$
[1]+  Done                    wordfreq Encyclopedia.txt >output
alice@ubuntu:~$ date
Mon Oct 25 14:41:20 MDT 2022
alice@ubuntu:~$
```

To see all processes on the system, use the command `ps augxw`. To search for processes owned by a user `bcatherm`, use grep as shown below:

```
ps augxw | grep bcatherm
```

Here is a sample output

```
alice@ubuntu:~$ ps augxw | grep bcatherm
bcatherm 14322  0.0  0.0  6760 2020 ?        S    Dec01   0:02 /usr/sbin/sshd
bcatherm 14328  0.0  0.0  4396 1480 pts/14   S    Dec01   0:00 -bash
bcatherm 16659  0.0  0.0  6792 2032 ?        S    00:10   0:00 /usr/sbin/sshd
bcatherm 16667  0.0  0.0  4392 1472 pts/0    S    00:10   0:00 -bash
bcatherm 20128  0.0  0.0  4152 1072 pts/0    S    00:53   0:00 /bin/sh /usr/local/bin/pbsget -4
bcatherm 20134  0.0  0.0  1584  656 pts/0    S    00:53   0:00 qsub -v DISPLAY -q interactive
bcatherm 20140  0.0  0.0  4392 1472 ttyp0    S    00:53   0:00 -bash
bcatherm 20141  0.0  0.0  1456  312 ttyp0    S    00:53   0:00 pbs_demux
bcatherm 20901  0.4  0.0  8360 2868 pts/14   S    01:09   0:00 vim control/pvm/stage3.c
alice     20907  0.0  0.0  3592  628 pts/19   S    01:09   0:00 grep bcatherm
alice@ubuntu:~$
```

If we start a background job with the ampersand `&` and logout, normally the background job is terminated. However, we can ask the shell to let the background job continue running after we log out by using the prefix `nohup`. For example:

```
nohup mylongrunningprogram &> output &
logout
```

Next time we login, we can use the `ps` command to check whether the job has finished. Then check for the output file. Note - `&> output` is a short-hand for `> output 2>$1`

Killing processes

To kill a process use `kill process-id`, where `<process-id>` is as shown by the `ps` command, or `kill %n`, where `n` is the job number as reported by the `jobs` command. If we feel lazy about finding the process id (laziness can be a good trait for a programmer!), then we can use the `killall` command, which kills by the name of the process. For example, we can use `killall wordfreq`, which kills all processes that have the string `wordfreq` as a part of their name.

If a process has gone amok and does not respond to the `kill` command, we can give the `-9` option (which is the same as the `SIGKILL` signal, a signal that will almost surely kill the process).

```
killall -9 wordfreq
```

3.4.3 Playing Lego in Linux

Running commands in series

The semicolon is interpreted by the shell as a command separator. So we can type multiple commands separated by semicolons on a line and the shell will execute them serially in the order we typed the commands. For example,

```
sleep 300; echo "Tea is ready"
```

the above command will output the string "Tea is ready" after 300 seconds (5 minutes).

Combining commands using pipes

A *pipe* is a way to connect the output of one program to the input of another program without any temporary file; a *pipeline* is a connection of two or more programs through pipes. The symbol for a pipe is `|`. For example:

`who | wc -l`	*Count users*
`who | grep mary | wc -l`	*Count how many times Mary is logged in*
`cat file1 file2 file3 | spell | less`	*Concatenate three files, run the spelling checker on the output and then display the results one page at a time with the `less` program.*

Let's play with pipes and processes. The command `last` prints out a list of users that have logged in to the system since the last date the log file has been kept. For example, the following shows the partial output of `last` on a system.

```
gcook    pts/86       132.178.175.169  Mon Apr  1 10:25 - 12:41  (02:16)
znickel  pts/85       et238-1164.boise Mon Apr  1 10:23 - 11:21  (00:58)
mlukes   pts/82       obsidian.boisest Mon Apr  1 10:22 - 15:01  (04:39)
lroutled pts/81       mg122-9.boisesta Mon Apr  1 10:20 - 11:47  (01:27)
aolson   pts/76       mg122-6.boisesta Mon Apr  1 10:19 - 11:45  (01:26)
dornelas pts/62       node19.boisestat Mon Apr  1 10:17 - 14:14  (03:56)
dornelas pts/61       node19.boisestat Mon Apr  1 10:03 - 10:39  (00:35)
dornelas pts/60       node19.boisestat Mon Apr  1 10:01 - 14:14  (04:12)
rgeetha  pts/59       65-129-50-248.bo Mon Apr  1 09:37 - 11:38  (02:00)
rgeetha  pts/49       65-129-50-248.bo Mon Apr  1 09:37 - 15:38  (06:01)
mmartin  pts/45       75-92-191-73.war Mon Apr  1 09:32 - 11:46  (02:13)
mvail    pts/43       69-92-71-108.cpe Mon Apr  1 09:19 - 11:21  (02:02)
mvail    pts/39       69-92-71-108.cpe Mon Apr  1 09:19 - 11:22  (02:03)
whieb    pts/25       masquerade.micro Mon Apr  1 09:15 - 11:20  (02:04)
tford    pts/43       216.190.60.34    Mon Apr  1 07:58 - 07:58  (00:00)
```

```
aolson   pts/39       cls-busn-206a.bo Mon Apr  1 07:36 - 08:44  (01:08)
mvail    pts/25       69-92-71-108.cpe Mon Apr  1 07:35 - 08:38  (01:03)
aibrahim pts/39       65-129-56-197.bo Mon Apr  1 05:00 - 05:00  (00:00)
aibrahim pts/25       65-129-56-197.bo Mon Apr  1 04:59 - 05:14  (00:15)
...
wtmp begins Mon Apr  1 04:59:35 2022
```

Suppose we want to make a list of all the users who have been on the system and arrange the list by how often they have logged in to the system. So the only useful information for us is the first column of the output. We will use the filter `awk` to extract the first column as follows:

```
last | awk '{print $1}'
```

Next we want to sort this list of names so that duplicates are brought together.

```
last | awk '{print $1}' | sort
```

Next we will use the command `uniq -c` that eliminates duplicates from a list of words, replacing each set of duplicates by one instance of the word prefixed by a count of how many instances of that word were found in the list.

```
last | awk '{print $1}' | sort | uniq -c
```

Now we have a list of users prefixed with how many times each user has logged in to the system. Next we sort this list by the numeric count in reverse order (so that larger counts show up first).

```
last | awk '{print $1}' | sort | uniq -c | sort -rn
```

Try this command on your system and see what results you get! If you want to learn more about pipes and filters, see the book *The UNIX Programming Environment* [1].

3.5 Data Science, anyone?

Combining `grep`, `sed`, `awk` with pipes and other filters can be used to perform many common data science tasks!

Chapter 4

Shell Scripting

4.1 Introduction

The shell has a complete programming language interpreter built into it. The shell supports a wide variety of iterative and branching structures (that is, loops and if's) that are covered in the `man` page for the `bash` shell.

Shell programming comes in two flavors: shell scripts and shell functions. Shell scripts, once defined, can be executed just like any other executable. Shell functions are similar to shell scripts but are defined in the environment. This simply means that they load much faster than shell scripts and can change the current environment.

4.2 Text editors

4.2.1 Introduction

Learning to be proficient with a command-line, text editor is one of the most productive things a user/programmer can do under any operating system. Often, when we login in to remote servers, there is no graphical desktop (and therefore no GUI editors nor IDEs) so text editors are all we can use. For example, we will not be able to use `gedit` nor VS Code. For this reason, everyone should know one command-line, text editor reasonably well. Below we mention two text editors that are very powerful, universally available and extensible. Choose one of these two editors and learn it well!

4.2.2 The Vim file editor

The Vim editor is a powerful and universally available screen-oriented editor. It is compatible with the older text-based `vi` editor. The Vim editor has extensive online documentation, and has its home page at the web address http://www.vim.org. The Vim editor is available for all kinds of machines and operating systems including Mac OS X, MS Windows and all variants of Linux and UNIX.

Vim has both a command-line version (which we are detailing here) that is invoked by typing in `vim` as well as a graphical version that is invoked by typing in `gvim`. The command-line version is the recommended text editor for non-GUI servers, especially for programmers. Using the mouse and built-in menus the user can be productive in a few minutes. However, the real power of Vim is accessed through the commands that can be used inside it. It has extensive built-in help. There is another resource that is helpful for learning editing in `vim`.

- Use the command `vimtutor` for a 30 minute tutorial on effective editing using `vi`.

The editor of choice of the authors is `vim`.

4.2.3 The GNU `emacs` file editor

GNU Emacs is a powerful editor that is also available on most Linux/UNIX machines. The editor is invoked by typing `emacs` and has a built-in tutorial.

4.3 Creating new commands

A new command can be created by writing a shell script. A shell script is just a text file containing a sequence of shell commands. Almost anything accepted at the command line can go into a shell script file. However, the first line is unusual; called the *shebang*, it holds the path to the command interpreter, so the operating system knows how to execute the file: `#!/path/to/interp flags`. This can be used with any interpreter (such as `python`), not just shell languages. Here is a simple example of a shell script:

```
#!/bin/sh
STR="Hello world!"
echo $STR
```

Open a file, say `hello.sh`, and type in the commands shown above. Then save the file and set the executable bit as follows.

```
chmod +x hello.sh
```

Now run the script. It is customary for shell scripts to have an `.sh` extension but not required. Here is what it will look like.

```
alice@ubuntu:~$ chmod +x hello.sh
alice@ubuntu:~$ ./hello.sh
Hello world!
alice@ubuntu:~$
```

As it stands, `hello.sh` only works if it is in our current directory (assuming that our current directory is in our `PATH`). If we create a shell script that we would like to run from anywhere, then move the shell script to the directory `~/bin`. Create that folder if it doesn't already exist. This is typically where users store their own custom scripts. Then we will be able to invoke the shell script from anywhere if the `~/bin` directory is on the shell `PATH`.

It is not a good idea to name a script `test` – there is a built in by the same name, and this can cause no end of debugging problems. Similarly, it is usually a good idea (for security) to use the full path to execute a script; if it is in our current directory, then `./script` will work.

Finally, for debugging scripts, `bash -x script.sh` will display the commands as it runs them.

Frequently, we will write scripts by entering commands on the command line until things work. Then, we open an editor and retype all the commands. However, we can use the history mechanism to our advantage: recall that `history` will display the history file, and `fc` will load selected commands into the editor. However, `fc <first> <last>` will load the lines from the `<first>` command to the `<last>` command into our editor. Simply make the edits, and write them to a new file.

4.4 Command arguments and parameters

When a shell script runs, the variable `$1` is the first command line argument, `$2` is the second command line argument and so on. The variable `$0` stores the name of the script. The variable `$*` gives all the arguments separated by spaces as one string. The variable `$#` gives the number of command line arguments. The use of a dollar sign means to use the value of a variable.

The following example script counts the number of lines in all files given in the command line.

```
#!/bin/sh
# lc.sh

echo $# files
wc -l $*
```

Note the # sign is a comment in the second line of the script. The # comment sign can be used anywhere and makes the rest of the line a comment. The command `echo` outputs its arguments as is to the console (the $# gets converted to the number of command line arguments by the shell). Go ahead and type the above script in a file named lc.sh to try it out. Don't forget that shell scripts need to be set executable – `chmod +x <script>`.

Here is an example usage:

```
alice@ubuntu:~/shell-examples$ lc.sh *
3 files
      5 hello.sh
      6 lc.sh
      3 numusers
     14 total
alice@ubuntu:~/shell-examples$
```

The simple script assumes that all names provided on the command line are regular files. However, if some are directories, then `wc` will complain. A better solution would be to test for regular files and only count lines in the regular files. For that we need an `if` statement, which we will cover in a later section.

4.5 Program output as an argument

The output of any program can be placed in a command line (or as the right hand side of an assignment) by enclosing the invocation in back quotes: `` `cmd` `` or using the preferred syntax `$(cmd)`. However back quotes are still widely used in scripts.

For example, we can use `ls` and use its output as an argument to our line counting script from the previous section.

```
alice@ubuntu:~/shell-examples$ lc.sh $(/bin/ls)
3 files
      5 hello.sh
      6 lc.sh
      3 numusers
     14 total
alice@ubuntu:~/shell-examples$
```

Note that we specified `/bin/ls` instead of `ls` because the `ls` command was aliased to `ls --color`, which would not work because the color options adds special characters in the listing. By using the full pathname of the command, we are bypassing the alias. Alternately, we could have unalias'd `ls` before using it.

4.6 Shell metacharacters

Some important metacharacters in the shell:

`` `...` ``	run command in `` `...` `` and replace with output
\	escape, for example \c take character c literally
`'...'`	take literally without interpreting the contents
`"..."`	take literally after processing $, `` `...` `` and \

4.7 Shell variables

Shell variables are created when assigned. The assignment statement has strict syntax. There must be no spaces around the `=` sign and assigned value must be a single word, which means it must be quoted if necessary. the value of a shell variable is extracted by placing a $ sign in front of it. For example,

```
side=left
```

creates a shell variable `side` with the value `left`.

Some shell variables are predefined when we log in. Among these are the `PATH`, which we have discussed in Section 3.1.3. The variable `HOME` contains the full path name of our home directory, the variable `USER` contains our user name.

Variables defined in the shell can be made available to shell scripts and programs by exporting them to be *environment* variables. For example,

```
export HOME=/home/alice
```

defines the value of `HOME` variable and exports it to all scripts and programs that we may run after this assignment. Try the following script:

```
#!/bin/sh
#test.sh
echo "HOME="$HOME
echo "USER="$USER
echo "my pathname is " $0
prog=$(basename $0)
echo "my filename is " $prog
```

Note that $0 gives the pathname of the script as it was invoked. If we are interested in the filename of the script, then we need to strip the leading directories in the name. The command `basename` does that for us nicely. Also by using `$(basename)`, we can take the output from `basename` and store it in our variable.

Other useful pre-defined shell variables. The variable `$$` gives the process-id of the shell script. The value `$?` gives the return value of the last command. The value `$!` gives the process id of the last command started in the background with &.

4.8 Loops and conditional statements in shell programs

4.8.1 for loop

The simplest for loop iterates over a list of strings, as shown below:

```
for variable in list of words
do
    commands
done
```

Here is a sample that creates five folders.

```
alice@ubuntu:~/sandbox$ ls
alice@ubuntu:~/sandbox$ for f in 1 2 3 4 5
> do
> mkdir folder$f
> done
alice@ubuntu:~/sandbox$ ls
folder1  folder2  folder3  folder4  folder5
```

We can also write the `for` loop in one line by using semicolon separators.

`for variable in` *list of words*; `do` *commands*; `done`

Note that we can use wild card expressions in the for loop list as shown in the example below:

```
alice@ubuntu:~/sandbox$ for f in folder*; do ls -ld $f; done
drwxrwxr-x 2 alice alice 4096 Dec  5 14:17 folder1
drwxrwxr-x 2 alice alice 4096 Dec  5 14:17 folder2
drwxrwxr-x 2 alice alice 4096 Dec  5 14:17 folder3
drwxrwxr-x 2 alice alice 4096 Dec  5 14:17 folder4
drwxrwxr-x 2 alice alice 4096 Dec  5 14:17 folder5
```

We can also execute a program inline and take its output as the list of strings that a for loop iterates over. See the example below. In this example, the first command creates an empty file in the `/tmp` directory. Then the for loop uses the `find` command to list the full path to any file with the `.txt` extension in the user's home directory. The cat command with `>>` concatenates all such files into the one file in `/tmp` directory!

```
alice@ubuntu:~/sandbox$ echo > /tmp/all-my-text-in-one-file.txt
alice@ubuntu:~/sandbox$ for name in $(find ~ -name "*.txt" -print)
> do
>   cat $name >> /tmp/all-my-text-in-one-file.txt
> done
```

We can also write for loops that look more like in Java using the following syntax:

`for ((expr1; expr2; expr3)) do` *commands*; `done`

See below for an example:

```
alice@ubuntu:~/sandbox$ for ((i=0; i<10; i++))
> do
>   echo $i
> done
0
1
2
3
4
5
6
7
8
9
```

We can also use `printf` with bash. It uses formatting similar to `printf` in Java (and C).

```
alice@ubuntu:~/sandbox$ for ((i=0; i<20; i++)); do printf "%02d\n" $i; done
00
01
02
03
04
05
06
07
08
09
10
11
12
13
14
15
16
17
18
19
```

Try the above loop using `echo $i` instead of the `printf` and notice the difference.

We can also nest for loops. For example:

```
alice@ubuntu:~/sandbox$ for i in 1 2 3
> do
>   for j in 1 2
>   do
>     echo $i "*" $j "=" $[i*j]
>   done
> done
1 * 1 = 1
1 * 2 = 2
2 * 1 = 2
2 * 2 = 4
3 * 1 = 3
3 * 2 = 6
alice@ubuntu:~/sandbox$
```

4.8.2 if statement

Here are the various forms of the `if` statement:

`if` *command*
`then`

```
    commands
fi
```

Here `fi` denotes the end of the if statement.

```
if command
then
    commands
else
    commands
fi
```

```
if command
then
    commands
elif
    commands
else
    commands
fi
```

```
if command; then commands; [ elif command; then commands; ] ...  [ else commands; ]
fi
```

4.8.3 case statement

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

Here `esac` denotes the end of the `case` statement. It is case spelt backwards!

Let's create a small script that expects one command line argument.

```
alice@ubuntu:~/sandbox$ vim check.sh
alice@ubuntu:~/sandbox$ cat check.sh
#!/bin/bash

case $# in
0) echo "Usage: " $0 " <foldername>";;
esac
alice@ubuntu:~/sandbox$ chmod +x check.sh
alice@ubuntu:~/sandbox$ ./check.sh
Usage:  ./check.sh  <foldername>
```

Note that `$#` is the number of command line arguments and `$0` is the name of the script as invoked.

4.8.4 while loop

```
while command
do
    commands
done
```

`while command; do commands; done`

For example:

```
alice@ubuntu:~$ while true
> do
>   sleep 2
>   date
> done
Tue Dec  5 14:54:55 MST 2022
Tue Dec  5 14:54:57 MST 2022
Tue Dec  5 14:54:59 MST 2022
^C
alice@ubuntu:~$
```

Here is another example (that checks every 2 seconds how many times a user is logged in on the `onyx` server):

```
alice@ubuntu:~$ while true
> do
>   sleep 2
>   who | grep amit | wc -l
> done
1
1
1
^C
alice@ubuntu:~$
```

4.8.5 until loop

```
until command
do
    commands
done
```

`until command; do commands; done`

4.9 Arithmetic in shell scripts

Normally variables in shell scripts are treated as strings. To use numerical variables, enclose expressions in square brackets. For example, here is a code snippet that adds up the integers from 1 to 100.

```
alice@ubuntu:~$ for ((i=0; i<=100; i++))
> do
>   sum=$[sum + i]
> done
alice@ubuntu:~$ echo $sum
5050
```

Here is a script that adds up the first column from a text data file:

```
#!/bin/sh
# addData.sh
sum=0
for x in $(cat data.txt | awk '{print $1}')
do
    sum=$[sum+$x]
done
echo "sum =" $sum
```

Here is a sample run:

```
alice@ubuntu:~/sandbox$ cat data.txt
100 3.5
200 3.5
300 3.5
400 3.5
500 3.5
600 3.5
alice@ubuntu:~/sandbox$ ./addData.sh
sum = 2100
```

4.10 Interactive programs in shell scripts

If a program reads from its standard input, we can use the "here document" concept in shell scripts. It is best illustrated with an example. Suppose we have a program `p1` that reads two integers followed by a string. We can orchestrate this in our script as follows:

```
#!/bin/s''
```

```
p1 <<END
12 22
string1
END
```

where END is an arbitrary token denoting the end of the input stream to the program p1.

4.11 Useful commands for shell scripts

4.11.1 The basename command

Many times it is useful to extract the filename out of a pathname. For example, is the pathname is /usr/local/bin/cdisks, then we want to strip off all directories and forward slashes and come up with cdisks, which is the actual file name. The command basename does that for us nicely.

```
alice@ubuntu:~/guide$ basename /usr/local/bin/cdisks
cdisks
alice@ubuntu:~/guide$
```

The basename command can also be used to remove the extension of a file. For example:

```
alice@ubuntu:~/guide$ basename xyz.txt .txt
xyz
alice@ubuntu:~/guide$
```

4.11.2 The test command

The test command is widely used in shell scripts for conditional statements. See the man page for test for all possible usages. For example we can use it to test two strings:

```
if test "$name" = "alice"
then
    echo "yes"
else
    echo "no"
fi
```

Note that there must be a space around the = in the test command. We can also use it to check if a file is a regular file or a directory. For example.

```
for f in *
do
    if test -f "$f"
    then
        echo "$f is a regular file"
    else

        echo "$f is a directory"
    fi
done
```

We can also use it to compare numbers. For example.

```
if test "$total" -ge 1000
    then
        echo "the total is >= 1000"
    fi
done
```

Note that bash also allows the syntax `[...]`, which is almost equivalent to the `test` command. It also has a newer variant `[[ ... ]]`, which is recommended but it isn't part of the POSIX shell standard. See man page for bash for more details.

4.12 Functions

Generally, shell functions are defined in a file and sourced into the environment as follows:

```
$ . file
```

They can also be defined from the command line. The syntax is simple:

```
name () {
commands;
}
```

Parameters can be passed, and are referred to as `$1, $2`, and so on. `$0` holds the function name, while `$#` refers to the number of parameters passed. Finally, `$*` expands to all parameters passed. Since functions affect the current environment, we can do things like this:

```
tmp () {
cd /tmp
}
```

This will `cd` to the /tmp directory. We can define similar functions to `cd` to directories and save ourselves some typing. This can't be done in a shell script, since the shell script is executed in a subshell. That is, it will `cd` to the directory, but when the subshell exits, we will be right where we started.

Here is a function that uses arguments:

```
add () {
    echo $[$1 + $2];
}
```

To use the function:

```
$ add 2 2
4
$
```

The following example shows that the notion of arguments is context-dependent inside a function.

```
#!/bin/bash
#functionArgs.sh

echoargs ()
{
    echo '=== function args'
    for f in $*
    do
        echo $f
    done
}

echo --- before function is called
for f in $*; do echo $f; done

echoargs a b c

echo --- after function returns
for f in $*; do echo $f; done
```

Try the above out by creating a script and running it!

4.13 Extended shell script examples

Here we show some extended examples of shell scripts.

4.13.1 Printing with proper tab spaces

Suppose, we want a command called print that expands tabs to four spaces and then prints it on the default printer. The program `expand -4` expands tabs to 4 spaces. So we create a file called `print.sh` that contains the following.

```
#!/bin/sh
expand -4 $1 | lpr
```

Here `$1` denotes the first command line argument passed to the script `print.sh`, the name of the file to print. Then we set the executable bit and move the print script to our `bin` directory.

```
chmod +x print
mv print ~/bin/
```

Now we have the print command available from anywhere. Note that we will need to have a directory named `bin` in our home directory for the above sequence of commands to work.

4.13.2 Simple test script

Suppose we have a Java program, say *MySort*, that we want to test for several input sizes. We can write a script to automate the testing as follows

```
#!/bin/sh
for n in 10000 20000 30000 40000 50000 60000
do
        echo "Running MySort for " $n " elements---"
        java MySort $n
        echo
done
```

4.13.3 Changing file extensions in one fell swoop

Suppose we have hundreds of files in a directory with the extension `.cpp` and we need to change all these files to have an extension `.cc` instead. The following script `mvall` does this if used as following.

`mvall cpp cc`

```
#!/bin/sh
# simple/mvall
prog=`basename $0`
case $# in
0|1) echo 'Usage:' $prog '<original extension> <new extension>'; exit 1;;
```

```
esac

for f in *.$1
do
  base=$(basename $f .$1)
  mv $f $base.$2
done
```

The for loop selects all files with the given extension. The `basename` command is used to extract the name of each file without the extension. Finally the `mv` command is used to change the name of the file to have the new extension.

4.13.4 Replacing a word in all files in a directory

Here is a common problem. A directory has many files. In each of these files we want to replace all occurrences of a string with another string. On top of that we want to only do this for regular files.

```
#!/bin/sh
# sed/changeword

prog=`basename $0`
case $# in
0|1) echo 'Usage:' $prog '<old string> <new string>'; exit 1;;
esac

old=$1
new=$2
for f in *
do
        if test "$f" != "$prog"
        then
            if test -f "$f"
            then
                sed "s/$old/$new/g" $f > $f.new
                mv $f $f.orig
                mv $f.new $f
                echo $f done
            fi
        fi
done
```

First the case statement checks for proper arguments to the script and displays a help message if it doesn't have the right number of command line arguments.

The for loop selects all files in the current directory. The first if statement makes sure that we do not select the script itself! The second if tests to check that the selected file is a regular file. Finally we use `sed` to do the global search and replace in each selected file. The script saves a copy of each original file (in case of a problem).

4.13.5 Counting files greater than a certain size

For the current directory we want to count how many files exceed a given size. For example, saying `bigFile.sh 100`, counts how many files are greater than or equal to 100KB size.

```
#!/bin/sh
#bigFile.sh

case $# in
0) echo 'Usage: ' $prog '<size in K>'; exit 1;;
esac

limit=$1
count=0
for f in *
do
    if test -f $f
    then
        size=$(ls -s $f| awk '{print $1}')
        if  test $size -ge $limit
        then
            count=$[count+1]
            echo $f
        fi
    fi
done
echo $count "files bigger than " $limit"K"
```

For each selected file, the first if checks if it is a regular file. The we use the command `ls -s $f | awk 'print $1'`, which prints the size of the file in Kilobytes. We pipe the output of the `ls` to `awk`, which is used to extract the first field (the size). Then we put this pipe combination in back-quotes to evaluate and store the result in the variable `size`. The if statement then tests if the size is greater than or equal to the limit. If it is, then we increment the `count` variable. Note the use of the square brackets to perform arithmetic evaluation.

4.13.6 Counting number of lines of code recursively

The following script counts the total number of lines of code in `.c` starting in the current directory and continuing in the subdirectories recursively.

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

```
# countlines.sh
total=0
for currfile in $(find . -name "*.c" -print)
do
    total=$[total+($(wc -l $currfile| awk '{print $1}'))]
    echo -n 'total=' $total
    echo -e -n '\r'
done
echo  'total=' $total
```

If we want to be able to count `.h`, `.cc` and `.java` files as well, modify the argument `-name "*.c"` to `-name "*.[c|h|cc|java]"`

4.13.7 Backing up our files periodically

The following script periodically (every 15 minutes) backs up a given directory to a specified backup directory. We can run this script in the background while we work in the directory. An example use may be as shown below.

`backup1.sh cpsc208 /tmp/cpsc208.backup &`

```
#!/bin/sh
# backup1.sh

prog=`basename $0`
case $# in
0|1) echo 'Usage:' $prog '<original dir> <backup dir>'; exit 1;;
esac

orig=$1
backup=$2
interval=900 #backup every 15 minutes

while true
do
    if test -d $backup
    then
        /bin/rm -fr $backup
    fi
    echo "Creating the directory copy at" `date`
    /bin/cp -pr $orig $backup
    sleep $interval
done
```

4.13.8 Backing up our files with minimal disk space

A simple backup script that creates a copy of a given directory by using hard links instead of making copies of files. This results in substantial savings in disk space. Since the backup file has hard links, as we change our files in the working directory, the hard links always have the same content. So if we accidentally removed some files, we can get them from the backup directories since the system does not remove the contents of a file until all hard links to it are gone. Note that hard links cannot span across file systems.

```
#!/bin/sh
# backup2.sh

prog=`basename $0`
case $# in
0|1) echo 'Usage:' $prog '<original dir> <backup dir>'; exit 1;;
esac

orig=$1
backup=$2
if test -d $backup
then
    echo "Backup directory $backup already exists!"
    echo -n "Do you want to remove the backup directory $backup? (y/n)"
    read answer
    if test "$answer" = "y"
    then
        /bin/rm -fr $backup
    else
        exit 1
    fi
fi

mkdir  $backup
echo "Creating the directory tree"
find $orig -type d -exec mkdir $backup/"{}" \;

#make hard links to all regular files
echo "Creating links to the files"
find $orig -type f -exec ln {} $backup/"{}" \;

echo "done!"
```

4.13.9 Watching if a user logs in or logs out

The following script watches if a certain user logs in or out of the system. An example use:

```
watchuser hfinn 10
```

which will watch if the user `hfinn` logs in or out every 10 seconds.

```
#!/bin/sh
# watchuser.sh

case $# in
0) echo 'Usage: ' $prog '<username> <check interval(secs)>'; exit 1;;
esac

name=$1
if test "$2" = ""
then
    interval=60
else
    interval=$2
fi

who | awk '{print $1}' | grep $name >& /dev/null
if test "$?" = "0"
then
    loggedin=true
    echo $name is logged in
else
    loggedin=false
    echo $name not logged in
fi

while true
do
    who | awk '{print $1}' | grep $name >& /dev/null
    if test "$?" = "0"
    then
        if test "$loggedin" = "false"
        then
            loggedin=true
            echo $name is logged in
        fi
    else
        if test "$loggedin" = "true"
        then
            loggedin=false
            echo $name not logged in
        fi
    fi
```

```
    sleep $interval
done
```

Here is another version, written using functions for improved modularity:

```
#!/bin/bash
# watchuser-with-fns.sh
check_usage() {
    case $# in
    0) echo 'Usage: ' $prog '<username> <check interval(secs)>'; exit 1;;
    esac
}
check_user() {
    who | awk '{print $1}' | grep $name >& /dev/null
    if test "$?" = "0"
    then
        if test "$loggedin" = "false"
        then
            loggedin=true
            echo $name is logged in
        fi
    else
        if test "$loggedin" = "true"
        then
            loggedin=false
            echo $name not logged in
        fi
    fi
}
check_usage $*
name=$1
if test "$2" = ""
then
    interval=60
else
    interval=$2
fi
loggedin=false
check_user $name

while true
do
    check_user $name
    sleep $interval
done
```

Chapter 5

Further Exploration

We highly recommend working through the first five chapters of *The UNIX Programming Environment* [1] to further deepen your knowledge of scripting and power usage of the shell.

Bibliography

[1] *The UNIX Programming Environment* by B. W. Kernighan and R. Pike, Prentice Hall. Written by some of the original designers of UNIX. Despite the many changes in UNIX, this book remains a classic. The first five chapters are highly recommended as a follow up reading.

[2] *The AWK Programming Language* by Alfred V. Aho, Brian W. Kernighan, Peter J. Weinberger, Addison Wesley.

[3] *The Linux Home Page.* `http://www.linux.org`. Lots of useful information, news and documentation about Linux.