A Guide for Linux and the Department Computing Facilities

Amit Jain and John Rickerd

Last Revised: October 15, 2017

1Please send any comments to ajain@boisestate.edu. This guide can be found on the web at http://cs.boisestate.edu/~amit/teaching/handouts/cs-linux/
# Contents

1 Departmental Computing Facilities .................................................. 6

2 Whom to ask for help? ....................................................................... 8

3 Beginner’s Guide ............................................................................. 9
   3.1 Getting started ........................................................................... 10
      3.1.1 Logging in .......................................................................... 10
      3.1.2 Changing your password ...................................................... 10
      3.1.3 Logging out of the system ................................................... 10
   3.2 Some basics ............................................................................... 10
      3.2.1 Correcting your typing ......................................................... 10
      3.2.2 Special keys ......................................................................... 10
      3.2.3 Case sensitivity ................................................................. 11
      3.2.4 How to find information? ..................................................... 11
   3.3 Files and directories ................................................................... 13
      3.3.1 File names ........................................................................... 13
      3.3.2 Creating files and directories ................................................. 14
      3.3.3 Your current directory ......................................................... 14
      3.3.4 Your home directory ........................................................... 14
      3.3.5 Changing directories .......................................................... 14
      3.3.6 Special directories ............................................................... 14
      3.3.7 Special files ......................................................................... 15
      3.3.8 Other important directories ................................................ 15
      3.3.9 Viewing the contents of a text file ........................................ 15
      3.3.10 Listing files and directories ............................................... 16
      3.3.11 Wild-cards and file name completion .................................. 16
      3.3.12 Copying files or directories ................................................ 17
      3.3.13 Renaming a file or directory: .............................................. 17
      3.3.14 Removing(Deleting) files or directories ............................... 17
   3.4 Basic useful commands .................................................................. 17
      3.4.1 Finding the date and the time .............................................. 17
      3.4.2 Obtaining information about users ....................................... 17
      3.4.3 Finding system information ................................................ 18
      3.4.4 Recording a terminal session .............................................. 18
      3.4.5 Sleeping and sequencing .................................................... 20
7.1.4 Aliases .............................................. 38
7.1.5 Customizing ls ..................................... 38
7.1.6 Enhancing cd using a stack ......................... 39
7.1.7 Repeating and editing previous commands ........... 39
7.2 Packing up and backing up your files .................. 40
7.2.1 Archiving files with tar ............................ 40
7.2.2 Compressing files with gzip ....................... 41
7.2.3 Compressing files with bzip2 ....................... 41
7.2.4 Backing up your files ................................ 41
7.3 Recovering lost files .................................. 42
7.4 Text-based Email ...................................... 42
7.4.1 Forwarding email .................................... 42
7.4.2 Text-based mail clients ............................. 42
7.5 Other useful commands ................................. 43
7.5.1 Changing your personal information ................. 43
7.5.2 Changing your login shell ............................ 43
7.5.3 Disk quota .......................................... 43
7.5.4 Checking disk usage ................................. 43
7.5.5 Finding files in your home directory ................. 44
7.5.6 Using find for useful tasks .......................... 44
7.5.7 Spell checking ....................................... 46
7.5.8 Watching a command ................................. 46
7.6 Filters: cool objects ................................... 46
7.6.1 Character transliteration with tr .................... 47
7.6.2 Comparing sorted file with comm ................. 47
7.6.3 Stream editing with sed ............................ 48
7.6.4 String processing with awk ......................... 49
7.7 Processes and Pipes .................................... 52
7.7.1 Input-Output redirection ............................. 52
7.7.2 Processes .......................................... 52
7.7.3 Playing Lego in Linux ............................... 54

8 Shell Scripting .......................................... 56
8.1 Introduction ........................................... 56
8.2 Creating new commands ................................ 56
8.3 Command arguments and parameters .................... 57
8.4 Program output as an argument ......................... 58
8.5 Shell metacharacters ................................... 58
8.6 Shell variables ......................................... 58
8.7 Loops and conditional statements in shell programs .... 59
8.8 Arithmetic in shell scripts ............................. 60
8.9 Interactive programs in shell scripts ................... 61
8.10 Useful commands for shell scripts ..................... 61
8.10.1 The basename command ............................ 61
8.10.2 The test command .................................. 61
8.11 Functions ............................................ 62
8.12 More shell script examples

8.12.1 Printing with proper tab spaces
8.12.2 Simple test script
8.12.3 Changing file extensions in one fell swoop
8.12.4 Replacing a word in all files in a directory
8.12.5 Counting files greater than a certain size
8.12.6 Counting number of lines of code recursively
8.12.7 Backing up your files periodically
8.12.8 Backing up your files with minimal disk space
8.12.9 Watching if a user logs in or logs out

9 Further Exploration
Chapter 1

Departmental Computing Facilities

The MetaGeek lab, (named in honor of a lab sponsorship by a local software company MetaGeek) is located in room CCP 240. The Kount Tutoring center in room CCP 241 (named honor of a lab sponsorship by a local software company Kount) also runs the same software. In addition, there are lab machines in the CCP 242 classroom and in the ENGR 111 lab.

All machines in the lab as well as departmental servers run CentOS 7 Linux. The lab uses a proximity card reader for access with your Boise State identity card. You will also need a login name and a password to use the machines. Normally you should get the login name and password from your instructor. Your instructor should also be able to help you with setting up lab access.

The main server for all of the CS labs is onyx.boisestate.edu and the only one that is on the public Internet, that is, visible from outside the lab. There are 143 workstations in the various CS labs, which are named onyxnode01, onyxnode02, . . . , onyxnode143. The name onyxnode00 is an alias for the main server onyx. The workstations in the labs are connected with a private Gigabit Ethernet switch with the main server and each other.

The lab is especially setup for Computer Science classes. Depending upon the Computer Science class you are taking, you will learn to use various features of the lab. The 143 machines in the labs are clustered together so you can use them as a single machine if need be! Since the machines in the lab are part of a cluster, it is important not to power off any machine or disconnect the network cables. You may disrupt the work of another student.

Linux Logo and a Penguin Cluster!
To use the lab over the Internet you must login to onyx.boisestate.edu. Your home directory resides on the main server, which is the file server for the lab. Thus you have the same home directory on all machines in the lab. So in the lab you can work on any workstation and you see the same files in your account.
Chapter 2

Whom to ask for help?

- For login account and access to the lab, contact your instructor.
- For lost or forgotten password, email a request to coenits@cs.boisestate.edu and include your full name, student ID, major as well as the course name and number.
- For help with lab related issues, ask one of the lab tutors.
Chapter 3

Beginner’s Guide

This guide is intended to give you some basic information that you need to start using the Linux operating system. Please see Section 1 for a description of the departmental computing facilities. All the machines in the Computer Science labs run CentOS 7 Linux. Most of the Linux commands that we discuss are common to several other operating systems such as Mac OS X and UNIX systems.

When you login to any of the machines, you will have a graphical desktop known as the K Desktop. To access the full power of Linux, you need to start up a Terminal (also known as a console). In the KDE Desktop, to access a terminal window, go to the desktop background, right click and then select konsole. The konsole terminal lets you 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, in the departmental labs your default shell will be the Bourne Again SHell (or bash).

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@onyx]:. The following shows what output is produced by typing in the command whoami to the shell.

[alice@onyx]: whoami
alice
[alice@onyx]:

The actual prompt that you may see on a particular system might be different. You can also choose your own prompt (See Section 7.1). 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. You don’t actually type the < and > characters!
3.1 Getting started

3.1.1 Logging in

In Linux the procedure for obtaining an authorized use of the system is called “logging in”. When you are on a workstation then you should have a login: prompt on the screen. If the screen is blank, move the mouse to obtain a login window. If there is no response, then the monitor may be turned off. In that case turn on the monitor. Enter your user name at the login: prompt and then enter your password at the password: prompt. Your password will not appear on the screen when typed.

3.1.2 Changing your password

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

3.1.3 Logging out of the system

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

- To log out of a console terminal type logout or exit.
- To log out of an X Windows session, drag the mouse to the background, and click on the right button and select the Leave option in the menu.

3.2 Some basics

3.2.1 Correcting your typing

You 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.

3.2.2 Special keys

Most of the keyboard characters are ordinary displayable characters with the 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 you 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 you 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, you 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 you 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.

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

3.2.3 Case sensitivity

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

3.2.4 How to find information?

There are five sources of information for a Linux system: (1) manual pages, (2) info pages, (3) HOWTOs and documentation for programs, (4) help from programs and (5) the Internet Google search engine. Please bookmark this section and come back to it as needed. Just browse through it first time quickly to see what options you have.

• Man pages: On-line manual pages 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.

  – You 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 you can call from a C/C++ program. Typing in man -a sleep will show you all the man pages for sleep. It will show the first man page. If that is not the one you want, then press q to quit that page and then it will show you the next one on the same command and so on. You can use PageUp and PageDown keys to browse a manual page.

  – You want a one line summary of what a command does. To obtain a one line summary about a command type man -f <command name>. 
– **You want to find a command that does something you want to do.** Use an appropriate keyword that best describes what you are looking for and then type `man -k <keyword>`, which displays one line summaries of the commands that reference that keyword. For example:

```bash
[alice@onyx]: man -k sleep
```

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tcl_Sleep</td>
<td>(3) - delay execution for a given number of milliseconds</td>
</tr>
<tr>
<td>Tcl_Sleep [Sleep]</td>
<td>(3) - delay execution for a given number of milliseconds</td>
</tr>
<tr>
<td>apmsleep</td>
<td>(1) - go into suspend or standby mode and wake-up later</td>
</tr>
<tr>
<td>nanosleep</td>
<td>(2) - pause execution for a specified time</td>
</tr>
<tr>
<td>sleep</td>
<td>(1) - delay for a specified amount of time</td>
</tr>
<tr>
<td>sleep</td>
<td>(3) - Sleep for the specified number of seconds</td>
</tr>
<tr>
<td>usleep</td>
<td>(1) - sleep some number of microseconds</td>
</tr>
<tr>
<td>usleep</td>
<td>(3) - suspend execution for microsecond intervals</td>
</tr>
</tbody>
</table>

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.

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

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

```
man 3 printf
```

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

- **Info:** Info will generally provide a gentler introduction than the man pages. However, it is more complicated to use; `info` gives a topics menu on starting up. A first-time user can type `h` for a tutorial on how to use `info`. Suppose you want information on the topic `<topic>`, then use `info <topic>`. For example, try `info passwd`.

- **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](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 your browser to [www.google.com](http://www.google.com). Get familiar with this search engine and you may save years in your life.

• **KDE Help.** Under the KDE GUI desktop, you can access all the man and info pages in an integrated fashion. Type Alt and F2 together. It will pop up a window. In that window type in `man:cmd` or `info:cmd` to access the man page or info page for the command `cmd`. Try this: Alt and F2 to get a popup window, then type `man:passwd`. In the lab, you can also try `man:submit`.

### 3.3 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 interpret 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.

#### 3.3.1 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 root, 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`.

You can also refer to a file by its **relative pathname** by giving the path from your current position in the tree to the place where the file is. You go up one level in the tree by entering: `cd ..`

For example, if you are currently in the directory `myDir` and you 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`
3.3.2 Creating files and directories

Normally you would use a text editor for creating files. (see Section 6.2 on file editing). However you can use the following command to create an empty file.

```bash
touch <filename>
```

To create a new directory type:

```bash
mkdir <directory_name>
```

3.3.3 Your current directory

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

3.3.4 Your home directory

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

3.3.5 Changing directories

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

3.3.6 Special directories

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

- the current directory: .
- the Parent Directory of the current directory: ..
- your Home Directory: ~
- another User’s Directory: ~<user_name>
- the root directory of the system: /

14
3.3.7 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. See Section 3.3.10 for ways of listing the “dot” files. The “dot” files are used for initializing applications and for customizing your environment (see Section 7.1). For example, when you login, the .bash_profile file sets up your session and terminal characteristics.

Many applications have special files that have names starting with a ‘.’, usually located in your home directory. 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.

3.3.8 Other important directories

The root 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.

Suppose we have a user named alice. Let us trace the path to the home directory of alice. Under the home directory, suppose we have a sub-directory called students. Underneath the students directory, suppose we have the directory alice. Then the absolute pathname to the home directory of alice is /home/students/alice.

The directory /bin (short for binary) contains executable programs commonly used by all users. Look at the programs in that directory. If you are curious about any particular program, then read the man page for that program. 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 hep in booting up the system. The directory /mnt contains mount points for mounting a file system temporarily. For example, this is where your CDs will show up (as /mnt/cdrom).

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. You can use this directory as a place for storing files temporarily. Usually the /tmp directory has much more space than your home directory.

3.3.9 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 you 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.

4. Use a text editor. For example: `vim`, `emacs` etc.

### 3.3.10 Listing files and directories

The command **ls** will list the names of files and directories in your 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 7.1.5 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.

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

### 3.3.11 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 shell has a file name completion feature by which you can avoid typing long file and directory names. Suppose you have a directory named `horriblylongname` and you wanted to `cd` to this directory. You 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 you all the files or directories that start with the prefix `horr`. Now you can type a few more characters until the prefix is unique so the shell can complete the filename for you. The file completion feature is very handy and saves the user a lot of typing.

### 3.3.12 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` stands for recursive since the first directory is recursively copied into the second directory, that is, all subdirectories inside the directory being copied are also copied recursively and so on.

### 3.3.13 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>`.

### 3.3.14 Removing(Deleting) files or directories

- To remove a file in your 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>`. Beware! This will recursively delete everything in that directory. To recover files deleted accidentally, see Section 7.3.

### 3.4 Basic useful commands

#### 3.4.1 Finding the date and the time

The command `date` prints the date and time.

#### 3.4.2 Obtaining information about users

- To see your login name, type `whoami`
- To see who is presently logged onto the system, type `who`.
• To see what programs users are running, type `w`.

• To get information about a specific user type `finger <user_name>`.
  The program `finger` gives some basic information about the user and
  prints out the file `.plan` if the user has such a file in her/his home directory.
  So if you make a `.plan` file in your home directory, other users will see it when
  they finger you. You will use a text editor to create such a file. For example,
  you can use `kwrite`.

Also, try these commands on the `onyx` server for more interesting results!
You can connect to it with the `ssh` command as shown in Section 3.5.2.

### 3.4.3 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. You can change its output to color
  by typing Z (uppercase z). 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. You can see the load on
  individual CPUs by pressing '1'. Pressing '1' again takes you back to the view with combined
  load on all the CPUs. Pressing M sorts the processes by most memory usage down. Pressing
  P sorts the processes by most CPU time used down.

Also, try these commands on the `onyx` server for more interesting results!
You can connect to it with the `ssh` command as shown in Section 3.5.2.

### 3.4.4 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>`. For example, this is useful for
keeping a proof of submitting an assignments To stop recording type `Ctrl-d` or `exit`. An example session is shown below.

```
[alice@onyx ~]$ script log
Script started, file is log
[alice@onyx ~]$ date
Sat Oct 14 12:06:11 MDT 2017
[alice@onyx ~]$ nproc
32
[alice@onyx ~]$ free -h
```
Mem: 31G 7.3G 5.3G 533M 18G 22G
Swap: 49G 6.0G 44G

[alice@onyx ~]$ exit
exit
Script done, file is log

Here is another example:

[alice@onyx chap01]$ script cs121-p6-submit.log
Script started, file is cs121-p6-submit.log
[alice@onyx chap01]$ submit alice cs121-2 p6

Right now this program is collecting the following directory
/home/faculty/alice/cs121/programs/chap01
Make sure that the directory is the right one!!!!

If you trying to submit the previous programs(late),
this program will time-stamp your submission.

Here are the files that will be submitted. If that is not correct,
then you can resubmit with the right files in place.
Directory: /home/faculty/alice/cs121/programs/chap01
Files:
./
./Lincoln3.java
./log
./Makefile
./Lincoln2.java
./Lincoln.java

Submission of Programming Assignment p6 is now COMPLETE.
You will be informed of your grade
after the deadline (via e-mail).

Current timestamp: Thu Apr 18 22:29:35 2017

[alice@onyx chap01]$ exit
exit
Script done, file is cs121-p6-submit.log

3.4.5 Sleeping and sequencing

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

```
[alice@onyx ~]$ sleep 2
```

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

```
[alice@onyx ~]$ date; sleep 10; date
Sat Oct 14 13:09:13 MDT 2017
Sat Oct 14 13:09:23 MDT 2017
[alice@onyx ~]
```

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 that we will study in a later module.

3.4.6 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 you will learn about in various classes.

For example:

```
[alice@dslalice cs253]$ time sleep 4
real 0m4.020s
user 0m0.000s
sys 0m0.000s
[alice@dslalice cs253]$
```

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.

3.4.7 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@onyx ~]$ wc -l /usr/share/dict/words
479828 /usr/share/dict/words
[alice@onyx ~]$}
```
3.4.8 Sorting files

The command \texttt{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 \texttt{sort} command has many options: sorting numerically, sorting in reverse order, sorting on fields within the line etc. Please check the man page for more options.

Here are some common options:

\begin{verbatim}
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
\end{verbatim}

For an example usage, see below:

\begin{verbatim}
[alice@onyx ~]$ cat listing
1040 1680x1050.jpg
  4 log1
  4 log2
 16 logoCOENCS.png
1112 metageekSplash_screen.jpg
  0 testfile1
  0 tt

[alice@onyx ~]$ sort -n -k 1,1 listing
  0 testfile1
  0 tt
  4 log1
  4 log2
 16 logoCOENCS.png
1040 1680x1050.jpg
1112 metageekSplash_screen.jpg
\end{verbatim}

3.4.9 Displaying the last few lines in a file

The command \texttt{tail filename} displays the last 10 lines of the file \texttt{filename}. The command \texttt{tail -n filename} prints the last \texttt{n} lines. A neat option is \texttt{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 your system:
\texttt{tail -10 /usr/share/dict/words}
3.4.10 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@onyx ~]$ cat list1
1040 1680x1050.jpg
   4 log1
   16 logoCOENCS.png
   0 testfile1
   0 listing

[alice@onyx ~]$ cat list2
1040 1680x1050.jpg
   4 log2
1112 metageek_splash_screen.jpg
   0 testfile1
   0 tt

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

The `diff` command has several options. It can ignore all white space, ignore case, have the output be side by side and many others documented on its man page.

3.4.11 Finding the differences between two binary files

Use the command `cmp` to check if two binary files are different. The command `cmp` does not list differences like the `diff` command. However it is handy for a fast check of whether two binary files are the same or not.

```
[alice@onyx ~]$ cmp /bin/ls /bin/cat
/bin/ls /bin/cat differ: byte 25, line 1
[alice@onyx ~]$
```
3.4.12 Finding patterns in files using your buddy grep

Use `grep <pattern> <filenames>`. 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 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:

```bash
[alice@onyx chap06]$ 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();
```

```bash
[alice@onyx chap06]$ 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@onyx chap06]$
```

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.

```bash
[alice@onyx examples]: 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@onyx examples]:
```

Programmers often use this option to quickly search for a declaration of a structure 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.

```bash
[alice@onyx examples]$ ls
chap01 chap02 chap03 chap04 chap05 chap06 chap07 extras file-io
```
3.4.13 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 you want to find a file that has the string “duck” in its name. So you type in the following command and see what output you get on your system!

```
[alice@onyx ~]$ locate duck
/home/JimConrad/CS121/projects/p1/solution/ducks-animation
/home/JimConrad/CS121/projects/p1/solution/ducks-animation/README
/home/JimConrad/CS121/projects/p1/solution/ducks-animation/TrafficAnimation.java
/home/JimConrad/CS121/projects/p1/solution/ducks-animation/sun.png
/home/alice/cs121/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@onyx ~]$`
```

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

Note that if you 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:

```
[alice@onyx ~]$ locate duck | grep alice
/home/alice/cs121/duck.png

[alice@onyx ~]$
```

3.5 Working on the Internet

3.5.1 Host-names 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 you are logged on use the command `hostname`. The names of some of the departmental server machines in the labs are:
<table>
<thead>
<tr>
<th>Hostname</th>
<th>Fully Qualified Domain Name</th>
<th>Internet Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>onyx</td>
<td>onyx.boisestate.edu</td>
<td>132.178.227.11</td>
</tr>
<tr>
<td>cs</td>
<td>cs.boisestate.edu</td>
<td>132.178.227.8</td>
</tr>
</tbody>
</table>

The suffix `boisestate.edu` represents the sub-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@onyx ~] : host google
Host google not found: 3(NXDOMAIN)
```

```
[alice@onyx ~] : host google.com
google.com has address 216.239.37.99
google.com has address 216.239.57.99
google.com has address 216.239.39.99
```

```
[alice@onyx ~] $ host cs.boisestate.edu
cs.boisestate.edu has address 132.178.227.8
```

```
[alice@onyx ~] : host onyx.boisestate.edu
onyx.boisestate.edu is an alias for onyx2.boisestate.edu.
onyx2.boisestate.edu has address 132.178.227.11
```

### 3.5.2 Remote access

The program `ssh` is a secure encrypted client for login to remote machines. The password as well as data is transmitted 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 Mac OS X.

You can login to another machine using the `ssh` command. The command `ssh` starts a new shell on the remote machine.

You can also use secure shell to copy files to/from remote machines. The command to use is `scp`. For example, the following command copies the directory `program4` (for login id `alice`) recursively from the server `onyx.boisestate.edu` to your home computer (if you execute this on your home computer).

```
scp -r alice@onyx.boisestate.edu:workspace/program4 .
```

For copying a directory from your system to onyx, you would use something like the following:

```
scp -r program4 alice@onyx.boisestate.edu:workspace/
```

Here the local folder `program4` gets copied to the onyx server under the `workspace` folder in the user’s home directory on onyx.
3.6 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 4

Linux Desktop GUI

4.1 Introduction

Linux uses the X Window system as the underlying engine for graphical display. Two popular desktops built on top of X Windows are the K Desktop Environment (KDE) and Gnome. They are intuitive to use and customizable. Currently the default desktop in the labs is KDE.

KDE comes with an integrated file browser (Dolphin) and an integrated web browser (Konqueror), a simple text editor (kwrite), several multimedia applications and many other applications.

4.2 Using the KDE Desktop

4.2.1 Cut and paste using a mouse

In X Windows you can cut and paste text and commands from one window to any window. To cut, click and drag the left button on the mouse to select the text. To paste, click on the middle button on the mouse. This may be simpler than using the history commands. A quick double click will select a word while a triple click will select an entire line.

4.2.2 Working with the KDE desktop

On your desktop you should see the start menu on the panel at the bottom left. This is the starting point to access all the menus and applications.

Another useful icon is the home directory icon, which is usually labeled with your login name. Clicking on this icon starts up the KDE file browser (the actual file browser program is named konqueror).
4.2.3 Taking a Snapshot of your Desktop

Use the *ksnapshot* program from the console or find it using the search field in the start menu. It lets you take a snapshot of your desktop or any part of it.

4.2.4 Creating directories in the KDE file browser

- Press and hold right mouse button in the file browser background. A menu will popup.
  - Using the right mouse button select *Create New* option in the popup menu and another menu will pop up.
  - Then select *Directory* and release button.
- Type in name of new directory and select *OK*

4.2.5 Creating shortcuts on the KDE desktop

You can create shortcuts on the KDE desktop by pressing the right mouse button in the desktop window (usually on top left), which pops up a menu. Then you would select the type of shortcut you want to create and then fill in the information.

For example, you can create a shortcut to Eclipse by selecting *Create new → Link to application...* in the menus. Then a properties window will appear, where you can select the name of the shortcut. Then go to the *Application* tab and type in the command in the *Command* field. For Eclipse, the command will be `/usr/local/eclipse/eclipse` the `/dev/cdrom` device. Then go back to the *General* tab and also select an icon to use by clicking on the icon. Hit *Ok* when you are done and you now have a new shortcut.

4.3 Commonly Used Applications

4.3.1 Using CDs and DVDs

The system will automatically mount your CD/DVD and start a file browser on them. To eject a CD/DVD, you may press the right mouse button on the CD/DVD icon on your desktop and select the *eject* option. Alternatively, you can type in the command `eject` in a console to have the same effect. *Remember that as long as you are in the CD folder, the device cannot be ejected. Make sure that you don’t have a file browser open in the directory or its subdirectories on the CD or DVD and then try again to eject.*

4.3.2 Copying a directory to a CD or DVD

On machines that are equipped with a CD/DVD burner, you can copy files to the CD using the program *k3b*. All machines in the lab have CD/DVD burners.
Startup the DVD/CD burner program **k3b** either typing in **k3b** in a console or search from the *K menu → Applications → Multimedia* menu.

### 4.3.3 Burning CDs or DVDs

You can use the **k3b** program for copying CDs or DVDs.

### 4.3.4 Playing a movie

There are two programs available for playing movies in the lab: **xine** and **mplayer**. You can start them from the console or from the *Multimedia* menu under the *K Menu* button in the bottom left of your screen.

### 4.3.5 Viewing a collection of photos

The program **geeqie** is a nice way to see a slide show of photos. Here is a screenshot:
4.3.6 Editing photos

The program **gimp** (Graphical Image Manipulation Program) is a very powerful program for editing photos (rotating photos, adjusting light etc), creating icons, and various other image processing tasks. It is integrated with **gqview** photo viewer so you can edit photos as you watch a slide show. You can also invoke gimp on any photo while you are in the file browser by the following steps.

- Press and hold the right mouse button on a photo
- Move down to **Open With** option and then select **The GIMP** to start the GIMP software for editing photos.

Below we will show some simple uses of **gimp**.

Rotating a photo

- Press and hold right mouse button and select the **Image** menu, then select **Transforms** and then select **Rotate** and then either **90** or **180** or **270** degrees
- Press and hold right mouse button and select the **File** menu and then select **Save**.

Changing brightness of a photo

- Press and hold right mouse button and select the **Image** menu, then select **Colors** and then select **Brightness-Contrast**. Adjust the brightness and contrast

Cropping a photo

Often we may want to cut out (crop) a part of a photo. In **gimp**, you can do this by opening the photo and then using the crop tool from the main tool window. You can also crop by pressing **Shift** and **c**. It activates the crop tool. Then select the region of the photo that you want and press left mouse button on it to crop the photo down to the region you have selected.

Screenshot

Here is a screenshot of **gimp** in action.
4.3.7 Running remote graphical programs

X Windows allows the local display of graphics from a program running remotely on another machine if the remote machine is given permission to display locally. The easiest way to do this is to use the ssh program for remote access (see Section 3.5.2).

[alice@onyx]: ssh -Y server.timbuktu.edu

-->now you are on timbuktu

$ xclock

--> the graphical clock shows up on the screen on your local computer
4.4 Other useful GUI programs

4.4.1 Viewing postscript and PDF files

Use the program \texttt{okular} for viewing postscript or pdf files. You can also use the Adobe Reader \texttt{acroread} for viewing PDF files, which generally have a \texttt{.pdf} extension.
Chapter 5

Filesystem

5.0.2 File protection

Every file on Linux has a mode or protection. A file may be readable, writeable(deletable), and executable, in any combination. In addition, a file can be accessible to a single user (u), a group of users (g), or all other users (o). You are considered the owner of all files and subdirectories in your home directory. This means that you 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@onyx]: 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 yourself or those in your 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@onyx]: 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@onyx]: chmod g-w,o-w wideopen
[alice@onyx]: ls -l wideopen
-rwxr-xr-x 1 alice faculty 0 Oct 25 13:23 wideopen
```
See the man page for `chmod` for more details.

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

```
[alice@onyx]: chmod g-rwx,o-rwx myhw
[alice@onyx]: 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 your own commands. See Section 8.2 for more on how to create your own commands.

### 5.0.3 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 you 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@onyx ~]: ln xyz.java xyz.java.save
[alice@onyx ~]: 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@onyx ~]: rm xyz.java
rm: remove regular file 'xyz.java'? y
[alice@onyx ~]: ls -l xyz.java.save
-rw-rw-r-- 1 alice alice 2374 Dec  3 10:50 xyz.java.save
[alice@onyx ~]:
```

- 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@onyx ~]: rm xyz.java
rm: remove regular file 'xyz.java'? y
[alice@onyx ~]: ls -l xyz.java.save
-rw-rw-r-- 1 alice alice 2374 Dec  3 10:50 xyz.java.save
[alice@onyx ~]:
```
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.
Chapter 6

Text Editors

6.1 Introduction

Learning to be proficient with a text editor is one of the most productive things a user/programmer can do under any operating system. Below we mention two text editors that are very powerful, universally available and extensible. Choose one of these two editors and learn it well!

6.2 The Vim file editor

The editor *vim* is a powerful and universally available screen-oriented editor. It is compatible with the the older *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 UNIX.

Vim has a graphical version that is invoked by typing in *gvim*. This is the recommended editor, especially for programmers. Using the mouse and built-in menus the user can be productive in a few minutes. It also has extensive built-in help. There are two other resources that are 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 *gvim*.

6.3 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.
Chapter 7

Advanced User’s Guide

7.1 Customizing your shell and improving productivity

The bash shell is extensively customizable and fully programmable.

7.1.1 Startup or run control files

For bash users, there are two files of interest: .bashrc and .bash_profile. These are (or should be) located in your 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. A good example to look at is ~amit/.bash_profile and ~amit/.bashrc.

7.1.2 Changing your shell prompt

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

[alice@onyx]: 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.

7.1.3 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 given by the PATH environment variable.

Normally you don’t want to override the system settings; instead, append to the current PATH

[alice@onyx]: export PATH="$PATH:/extra/dir"
with new entries delimited with a colon. The setting for the `PATH` variable goes in `.bash_profile` or `.bashrc` file.

A common situation is that the current directory is not in your path. So you have to specify the path using the dot notation:

```
./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:.
```

Add this line at the end of your `.bashrc` file in your home directory.

### 7.1.4 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 your home directory. Some examples:

```
alias rm="rm -i"
alias vi="gvim"
alias ls="ls -F --color=auto"
alias proj="cd ~/cs453/prog5"
```

Then typing the command `proj` takes you to the directory `~/cs453/prog5`. If for some reason you need to remove an alias, just use the `unalias` command.

### 7.1.5 Customizing ls

Common usage of the `ls` command:

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

Occasionally, you will see `alias l=ls` You 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 being piped to a tty (your 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 pretty nice 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;33:bd=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:*: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:
```

38
7.1.6 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** `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** `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 you need to traverse a number of directories but need to return to your current location when done. Of course, if you just need to toggle between two directories, then `cd ~` will do the trick.

7.1.7 Repeating and editing previous commands

The command `history` lists previous commands with numbers. You 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 you can run the command 183 again as follows:

```
[alice@onyx]: !183
cat /etc/shells
/bin/bash
/bin/sh
/bin/ash
/bin/bash2
/bin/tcsh
/bin/csh
/bin/ksh
/bin/zsh
[alice@onyx]:
```
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@onyx]$: date
Mon Oct 25 14:29:19 MDT 2012
[alice@onyx]$:!!
date
Mon Oct 25 14:29:23 MDT 2012
[alice@onyx]$
```

You can go to the previous command using the built-in editor mode. The default mode setup in your `.bash_profile` file is `vi` editor mode.

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

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

You can also set the editor mode for the `bash` shell to be Emacs by placing the following command in your `.bash_profile` file.

```
set -o emacs
```

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

```
[alice@onyx C-examples]$: history | grep javac
8202 javac
8206 javac WebStats.java
8211 javac -O WebStats.java
10082 history | grep javac
[alice@onyx C-examples]$
```

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!

Under X Windows you can use the mouse to cut and paste previous commands. See Section 4.

### 7.2 Packing up and backing up your files

#### 7.2.1 Archiving files with `tar`

The `tar` command is very useful in bundling up your files and directories. Suppose you want to bundle up the entire directory `cs253` under your home directory. You would use the following
command:

```
tar cvf cs253.tar cs253
```

This creates a file, often called a *tarball*, that contains the entire cs253 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 you can copy the tarball to another location on your system, or copy to another system or copy it to a CD or USB drive or email it to someone. Suppose you have a tarball that you want to unpack. Use the following command:

```
tar xvf cs253.tar
```

Here the option `x` is for extract.

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

```
tar tvf cs253.tar
```

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

```
tar czvf cs253.tar.gz cs253
```

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 cs253.tar.gz
```

### 7.2.3 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 cs253.tar.bz2 cs253
```

```
tar xjvf cs253.tar.bz2
```

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:

### 7.2.4 Backing up your files

You can backup your home directory using *tar* and *gzip* as follows:

```
tar czvf /tmp/home.tar.gz ~
```
The above command creates a compressed tarball of your home directory in the temporary directory of the system. See Section 8.12.8 for a better way of backing up your files.

7.3 Recovering lost files

If you accidentally delete a file then it can be recovered from the previous (daily) backup. For help on recovering deleted files send mail to coenits@cs.boisestate.edu. Always specify the absolute path name for the file and the date to which the file should be restored (and include your full name, student ID and major).

7.4 Text-based Email

7.4.1 Forwarding email

How do I forward my email on onyx so that it automatically goes to my home email?

Suppose that your login on onyx is alice and your home email address is meme@isp.net. Then create a file named .forward in your home directory. The file should have one line in it as follows

```
alice, meme@isp.net
```

Now any email sent to you at alice@onyx.boisestate.edu will still be received on onyx but a copy will also be forwarded to meme@isp.net

7.4.2 Text-based mail clients

When you login to the system, you are informed by the system if you have mail. You can read your mail by simply typing `mail`. The `mail` program will display a numbered list of new messages with their subject headers. To read a message type `n` where `n` is the number of the message you want to read. To delete the message numbered `n` type `d n`. To save a message in a file use the `s` command. To exit out of the mail and save all messages, without deleting, in the file `mbox` in your home directory, use `q`. To reply to a mail use the `Reply` or `R` command. There is also a `help` command in the `mail` program.

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

A mail program with a more convenient interface is mutt. It is available on all Linux systems and, currently, mutt is the favorite text-based mail reader of the authors.
7.5 Other useful commands

7.5.1 Changing your personal information

Use the command \texttt{chfn} to change your personal information like phone number, office location, your real name etc. The command \texttt{chfn} will ask for your password before letting you change your personal information.

7.5.2 Changing your login shell

You can change your default shell by using the following command:

\texttt{chsh}

where shell must be one of the shells listed in the file /etc/shells. The command \texttt{chsh} will ask for your password and then let you specify a new shell.

7.5.3 Disk quota

Each user account may have an associated disk quota that limits the amount of space you can use as well as the number of files you can have in your home directory. The command \texttt{quota \_v} shows you the current use as well as the limit.

\begin{verbatim}
[jcope@onyx jcope]$: quota -v
Disk quotas for user jcope (uid 620):
   Filesystem   blocks  quota  limit  grace   files  quota  limit  grace
/dev/sdb1     18864  25000   30000    1296   2500   3000
[jcope@onyx jcope]$
\end{verbatim}

If you exceed the quota, then the system gives you seven days to remove or compress files to cut down on the disk usage. After seven days, the login privileges are suspended. At that point, you will have to contact the system administrator (coenits@cs.boisestate.edu) to enable your account again.

7.5.4 Checking disk usage

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

\begin{verbatim}
[alice@onyx C-examples]: du doublyLinkedLists
220 doublyLinkedLists/bad
100 doublyLinkedLists/library
172 doublyLinkedLists/generic
548 doublyLinkedLists
[alice@onyx C-examples]:
\end{verbatim}
By default, `du` reports sizes in units of Kilobytes (1024 bytes). You can ask for human readable units by using the `-h` option. For example:

```
[alice@onyx C-examples]: du -h doublyLinkedLists
220K  doublyLinkedLists/bad
100K  doublyLinkedLists/library
172K  doublyLinkedLists/generic
548K  doublyLinkedLists
[alice@onyx C-examples]:
```

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

```
[alice@onyx C-examples]: du -h -s doublyLinkedLists
548K  doublyLinkedLists
[alice@onyx C-examples]:
```

### 7.5.5 Finding files in your 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 you want to find all the files named `core` in your home directory. You would use `find` as follows (with example output):

```
[alice@onyx]: 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.

### 7.5.6 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, you can find all files that were modified in your home directory in the last 30 minutes with the following use of the `find` command.

```
[alice@onyx docs]: find ~ -mmin -30 -print
/home/alice
/home/alice/.kde/share/config
```
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 {} \;
  ```

  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.

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

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

- Find all files in your 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 your account.

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

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

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

Use the command `ispell <filename>` to run an interactive spelling checker on a file. You can also use `ispell` from inside the mailer `mutt` with the `i` option before sending an email.

The command `spell` is a non-interactive spelling checker. It just prints out all misspelled words from the input file of words. Hence if if you check a file with and there is not 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@onyx simple]: cat test1
dada
dad
mom
father
sun
simpel
[alice@onyx simple]: spell test1
dada
simpel
[alice@onyx simple]:
```

The `spell` program by itself is not very useful. However it is useful if used from inside shell scripts. More about shell scripts in Section 8.2

7.5.8 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, you can use `watch` to see how much memory is being used on your system every 5 seconds with the following command.

```
watch -n 5 free
```

7.6 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].

46
7.6.1 Character transliteration with tr

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

\begin{verbatim}
cat doc1.txt | tr a-z A-Z
\end{verbatim}

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

\begin{verbatim}
cat doc2.txt | tr A-Z a-z
\end{verbatim}

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.

\begin{verbatim}
tr -cs "A-Za-z'" "[\n*]
\end{verbatim}

7.6.2 Comparing sorted file with \texttt{comm}

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

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

- \texttt{-1} suppress column 1 (lines unique to file \texttt{f1})
- \texttt{-2} suppress column 2 (lines unique to file \texttt{f2})
- \texttt{-3} suppress column 3 (lines that appear in both files)

For example,

\begin{verbatim}
comm -12 f1 f2
\end{verbatim}

prints lines in both files, and

\begin{verbatim}
comm -23 f1 f2
\end{verbatim}

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

\begin{verbatim}
[alice@onyx comm]: ls dir1
f1 f2 f3 f4 f5 f6 f8
[alice@onyx comm]: ls dir2
f1 f2 f3 f4 f5 f6 f7 f9
[alice@onyx comm]: ls dir1 > dir1.list
[alice@onyx comm]: ls dir2 > dir2.list
[alice@onyx comm]: comm -23 dir1.list dir2.list
f8
[alice@onyx comm]: comm -13 dir1.list dir2.list
\end{verbatim}
7.6.3 Stream editing with sed

The basic idea is to read lines one at a time from the input files, apply commands from the list, in order, to each line and write the edited output to the standard output.

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

Below we will provide a series of useful usages.

- For example, we can change all occurrences of Amit to JHack in all .h and .c files with the following command
  
  ```
  sed 's/Amit/JHack/g' *.c *.h > output
  ```

  However the above does not change the files. We need to use some shell programming to complete the job. See the example in Section 8.12.

- 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
  ```
7.6.4 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:

```bash
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@onyx alice]$ 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)
  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
  twitchu 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@onyx alice]$
  ```

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

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

  ```
  jlowe 22:34
  jcollins 11:30
  tcole 11:01
  ckrossch 07:35
  drau 17:53
  ```
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
```

```
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
```

- 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:
  ```awk 'NF%2 != 0'```  
- Print lines that are longer than 72 characters.
  ```awk 'length($0) > 72'
  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.

```bash  
[alice@onyx 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@onyx doublyLinkedLists]: wc -l *.c *.h | awk '{print $1}'
  26
  133
   42
   20
  131
   12
   27
   46
   23
  460
[alice@onyx doublyLinkedLists]: wc -l *.c *.h | awk '{print $1}' | awk '{s = s+$1} > END {print s, s/NR}'
  920 92
[alice@onyx doublyLinkedLists]:
```

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

51
7.7 Processes and Pipes

7.7.1 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.

```
gcc BadProgram.c 2> error.log
```

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

```
gcc BadProgram.c > log 2>&1
```

7.7.2 Processes

Creating and managing processes

The shell can also help you in running multiple programs at a time. Suppose you want to run a word count on a large file but you don’t want to wait for it to finish. Then you 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 you don’t redirect the output to a file it will show up on your terminal some time later when the wc program is done! The command jobs lists all background jobs that you have started from the current shell.

If you start a bunch of processes with the ampersand you can use the jobs command to list them. Then you can selectively bring one into the foreground by the command fg %n, where n is the job number as listed by the jobs command. You can also suspend a running program with Ctrl-z and put it in the background with the command bg.
Each running program is 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@onyx]: date
Mon Oct 25 14:40:52 MDT 2012
[alice@onyx]: wordfreq Encyclopedia.txt > output &
[1] 19027
[alice@onyx]: jobs
[1]+ Running wordfreq Encyclopedia.txt >output &
[alice@onyx]: 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
```

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:

```
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 -I /tmp/cluster.pbs.611
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 20818 0.0 0.0 1688 808 ttyp0 S  01:08 0:00 /usr/share/pvm3/lib/LINUXI386/pvmd3 -nws00
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
```

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

Next time you login, you can use the `ps` command to check whether the job has finished. Then check for the output file.

**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 you feel lazy about finding the process id (laziness can be a good trait for a programmer!), then you can use the `killall` command, which kills by the name of the process. For example, you 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, you 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
```

### 7.7.3 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
who | grep mary | wc -l
cat file1 file2 file3 | spell | less
```

*Count users*  
*Count how many times Mary is logged in*  
*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)
```
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 \texttt{awk} (a string processing program) to extract the first column as follows:

\begin{verbatim}
last | awk '{print \$1}'
\end{verbatim}

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

\begin{verbatim}
last | awk '{print \$1}' | sort
\end{verbatim}

Next we will use the command \texttt{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.

\begin{verbatim}
last | awk '{print \$1}' | sort | uniq -c
\end{verbatim}

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).

\begin{verbatim}
last | awk '{print \$1}' | sort | uniq -c | sort -rn
\end{verbatim}

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 \textit{The UNIX Programming Environment} [1].
Chapter 8

Shell Scripting

8.1 Introduction

The shell has a complete programming language built in 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.

8.2 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.
As it stands, **hello.sh** only works if it is in your current directory (assuming that your current directory is in your **PATH**). If you create a shell script that you would like to run from anywhere, then move the shell script to the directory `~/bin/`. Then you will be able to invoke the shell script from anywhere since that directory is on the shell **PATH**.

Don’t forget that shell scripts need to be set executable – `chmod +x <script>`. Also, it is not a good idea to name your 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 your current directory, then `./<script>` will work.

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

Frequently, you will write scripts by entering commands on the command line until things work. Then, you open an editor and retype all the commands. However, you can use the history mechanism to your 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 your editor. Simply make your edits, and write them to a new file.

## 8.3 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.

```bash
#!/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). Here is an example usage:

```bash
[alice@onyx shell-examples]: lc.sh *
3 files
```
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.

### 8.4 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@onyx shell-examples]: lc.sh $(/bin/ls)
3 files
  5 hello.sh
  6 lc.sh
  3 numusers
14 total
[alice@onyx 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 unaliased `ls` before using it.

### 8.5 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 `

### 8.6 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,

\[
\text{side} = \text{left}
\]

creates a shell variable \text{side} with the value \text{left}.

Some shell variables are predefined when you log in. Among these are the \text{PATH}, which we have discussed in Section 7.1.3. The variable \text{HOME} contains the full path name of your home directory, the variable \text{USER} contains your user name.

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

\[
\text{export HOME}=/\text{home/alice}
\]

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

\[
#!/\text{bin/sh}
\text{#test.sh}
\text{echo "HOME}=$\text{HOME}
\text{echo "USER}=$\text{USER}
\text{echo "my pathname is } $0
\text{prog}='\text{basename} \text{0}'
\text{echo "my filename is } $\text{prog}
\]

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 \texttt{basename} does that for us nicely. Also by using back quotes we can take the output from \texttt{basename} and store it in our variable.

\textbf{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 &.

\section{8.7 Loops and conditional statements in shell programs}

- \textbf{for loop}

  \[
  \text{for var in list of words}
  \text{do}
  \text{commands}
  \text{done}
  \]

  \[
  \text{for var in list of words; do commands; done}
  \]

  \[
  \text{for (( expr1; expr2; expr3)) do commands; done}
  \]

- \textbf{if statement}
if command
then
   commands
else
   commands
fi

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

• case statement
  case word in
    pattern) commands;;
    pattern) commands;;
    ...
  esac

• while loop
  while command
do
    loop body as long as command returns true
done
while command; do commands; done

• until loop
  until command
do
    loop body as long as command returns false
done
until command; do commands; done

8.8 Arithmetic in shell scripts

Normally variables in shell scripts are treated as strings. To use numerical variables, enclose expressions in square brackets.

#!/bin/sh
sum=0
for x in 'cat data'
do
   sum=$[sum+$x]
done
echo "sum=" $sum
8.9 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:

```bash
#!/bin/sh

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

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

8.10 Useful commands for shell scripts

8.10.1 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@onyx guide]: basename /usr/local/bin/cdisks
    cdisks
[alice@onyx guide]:
```

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

```
[alice@onyx guide]: basename xyz.txt .txt
    xyz
[alice@onyx guide]:
```

8.10.2 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:

```bash
if test "$name" = "alice"
    then
        echo "yes"
    else
        echo "no"
fi
```
We can also use it to check if a file is a regular file or a directory. For example.

```bash
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.

```bash
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.

### 8.11 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, you can do things like this:

```
tmp () {
    cd /tmp
}
```
This will `cd` to the `/tmp` directory. You can define similar functions to `cd` to directories and save yourself 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, you will be right where you started.

Here is a function that uses arguments:

```bash
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.

```bash
#!/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
}
```

8.12 More shell script examples

Here we show some more examples of shell scripts.
8.12.1 Printing with proper tab spaces

Suppose, you 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` that contains the following.

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

Here `$1` denotes the first command line argument passed to the script `print`, 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.

8.12.2 Simple test script

Suppose you have a 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---'
    MySort $n
    echo
done
```

8.12.3 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.

8.12.4 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
        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).

8.12.5 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 `countsize 100`, counts how many files are greater than or equal to 100K size.

```bash
#!/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}macenoes $limit
      then
        count=$((count+1))
        echo $f
      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.

8.12.6 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.

```bash
#!/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 ''
done
echo 'total=' $total
```

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

### 8.12.7 Backing up your files periodically

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

```
backup1.sh cs253 /tmp/cs253.backup &
```

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

prog='basename $0'

# $1 is the original directory
# $2 is the backup directory

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
```
8.12.8 Backing up your 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 you change your files in the working directory, the hard links always have the same content. So if you accidentally removed some files, you 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 filesystems.

```bash
#!/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!"
```

8.12.9 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 alice 10

which will watch if alice 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

fi
sleep $interval
done

Here is another version, written using functions:

#!/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 9

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.
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.
