Introduction to Linux and Lubuntu 16.04

modified by Ross Whetten to focus on the bash shell, Ubuntu system architecture, and bioinformatics examples, and based on Unix Tutorial for Beginners by M. Stonebank,

Tutorial Seven

- Installing Debian software packages
- Compiling UNIX software source code
- Download source code
- Extracting source code
- Configuring and creating the Makefile
- Building the package
- Running the software

Tutorial Eight

- UNIX variables
- Environment variables
- Shell variables
- Using and setting variables

Introduction to the UNIX Operating System

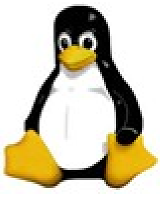
What is UNIX?

UNIX is an operating system which was first developed in the 1970s, and has been under constant development ever since. By operating system, we mean the suite of programs which make the computer work. It is a stable, multi-user, multi-tasking system for servers, desktops and laptops. Eric Raymond has written a historical review of the development of UNIX and Linux operating systems, combined with an interesting overview of philosophical principles he feels are embodied in these operating systems. The book is called The Art of Unix Programming, and is available online at http://catb.org/esr/writings/taoup/html/. A one-page summary of that book is available on the course website as a PDF document.

UNIX systems also have a graphical user interface (GUI) similar to Microsoft Windows which provides an easy to use environment. However, knowledge of UNIX is required for operations which aren't covered by a graphical program, or for when there is no windows interface available, as in some remote login sessions.

Types of UNIX

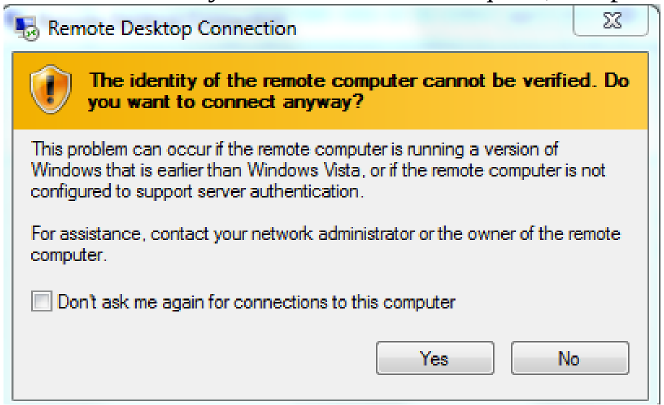
There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, and MacOS X.



For this course we are using Lubuntu GNU/Linux version 16.04. Many other versions, or distributions, of GNU/Linux are available, and they differ enough that it is important to specify which version is of interest when searching online for answers to specific questions. We will use the NC State Virtual Computing Lab to gain access to the Lubuntu computing environment.

Starting a VCL instance and logging in

The NC State Virtual Computing Lab allows users to use “virtual machines” that are based on “machine images” stored at the VCL website. To launch an instance of the machine image used for this class (Biostar_DNASeq), first browse to https://vcl.ncsu.edu and click the Log In link. Sign in using your NC State Unity ID and password, then click the Reservations link. Request a new reservation, and choose the Biostar_DNASeq image from the menu of options. Set the duration of the instance to the amount of time you expect to use the instance; there will be an upper limit to the amount of time you are allowed to request, but please don't request more time than you need.



When the instance is available, click the Connect link on the VCL web page to open a pop-up window with the IP address of the remote computer (your virtual machine). Highlight this using the cursor, copy the address, and then open the Remote Desktop Connection program on your Windows laptop. Paste the IP address into the Remote Desktop Connection window and click Connect – it will give you the warning shown in Figure 1, left.

Figure 1. Warning from Remote Desktop Connection

Click Yes to continue the connection. The next window that opens will be a graphic interface to your virtual machine, with a login prompt asking for a username and password (Fig 2). Enter your NC State Unity ID and password and click OK. The system may then ask if you want to choose a password for a new keyring (click Cancel), or if you want to set up a new panel (click Use Default Panel).

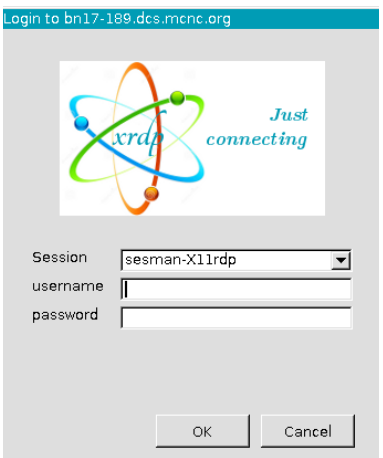


Figure 2. Login window

When the login process is complete, you may see a desktop with icons at the bottom and an Application menu at the upper left corner, or you may just see a desktop with a Trash icon, a File System icon, and a Home folder icon. Right-clicking on the desktop should open a drop-down menu with a link to the Applications menu.(Fig 3).

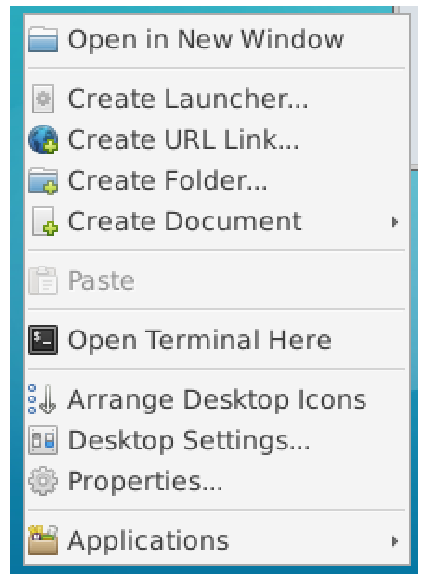


Figure 3. Drop-down menu

Mac users will need to install Microsoft Remote Desktop (available free from the Apple App Store) in order to allow connection to the virtual machine through a remote desktop connection. It is important to note that you must log into the VCL and click the Connect link on the computer from which you plan to connect to the virtual machine, according to instructions on the VCL site.

Starting a terminal session

To open a terminal window in the Lubuntu 16.04 system, click on the Applications menu in the upper left corner of the desktop and choose the Terminal Emulator item, or right-click on the desktop and choose "Open Terminal Here".

The text that appears in the terminal window is called the "system prompt" – this lets you know that the system is ready for you to enter a command. The prompt is made up of your NC State username followed by the @ symbol, a shortened version of the name of the virtual machine you are using, a colon (:), the current directory, and a $. The starting point for a new terminal window may be your home directory, shown as ~, or it may be the desktop if you used the "Open Terminal Here" option. The rest of this document will show the prompt simply as $, to save space, but you will notice that your screen always displays the complete set of information in the prompt: userID@host:directory$, which is also shown at the top center of the terminal window border.

The UNIX operating system

The UNIX operating system is made up of three parts; the kernel, the shell and the programs.

The kernel

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the filestore and communications in response to system calls.

As an illustration of the way that the shell and the kernel work together, suppose a user types **rm myfile** at a command-line prompt (which has the effect of removing the file **myfile**). The shell searches the filestore for the file containing the program **rm**, and then requests the kernel, through system calls, to execute the program **rm** on **myfile**. When the process **rm myfile** has finished running, the shell then returns the UNIX prompt $ to the user, indicating that it is waiting for further commands.

The shell

The shell acts as an interface between the user and the kernel. When a user logs in, the kernel executes a login program to check the username and password, and then start another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands are themselves programs: when they terminate, the shell gives the user another prompt ($ on our systems).

The adept user can customize his/her own shell, and users can use different shells on the same machine. The Lubuntu 16.04 system we are using has the **bash** shell by default. A reference sheet of Linux commands for the bash shell is available on the course website. This is not a complete list of all Linux commands – several online resources are available with more complete lists of Linux commands, but it is important to note that different distributions have different subsets of the universe of possible Linux commands, and no distribution is likely to have all of them.

The **bash** shell has certain features to help the user in entering commands.

Filename Completion - By typing part of the name of a command, filename or directory and pressing the [**Tab**] key, the shell will complete the rest of the name automatically. If the shell finds more than one name beginning with those letters you have typed, it may do nothing – pressing the tab key again will produce a list of all files (including commands and directories) that match the pattern entered so far, so you can see what alternatives are available. Type enough letters to match only one of the alternatives and press tab again to auto-complete the command.

History - The shell keeps a list of the commands you have typed in. If you need to repeat a command, use the arrow keys to scroll up and down the list or type **history** for a list of previous commands.

The programs - files and processes

Everything in UNIX is either a file or a process. Programs are saved as files, but give rise to processes when the program file is executed.

A process is an executing program identified by a unique PID (process identifier).

A file is a collection of data saved to disk or other long-term storage. These files are created by users using text editors, running compilers, or using other tools of the trade. Examples of files:

- instructions comprehensible directly to the machine and incomprehensible to a casual user, such as a compiled executable program or data saved in a binary file
- documents (whether text or in some word-processor format)

- source code of a program written in some high-level programming language and saved as text
- directories, which are files containing information about directory contents, which may in turn be a mixture of other directories (subdirectories) and ordinary files.

The Directory Structure

All the files are grouped together in the directory structure. The file-system is arranged in a hierarchical structure, like an inverted tree. The top of the hierarchy is traditionally called root and written as a forward slash (/). In the Linux system used for this class, we will be using three directories below /: the /home/*<username>* user directory, the /data directory, and the /usr/local/bin directory. The Linux system is accessed through the Virtual Computing Lab, and you will log in using your NC State Unity ID and password, so substitute your NC State user name wherever you see *<username>* in this document. Some of the data and documentation for exercises are in the directories in /data, many of the programs we will be using are in the /usr/local/bin/ directory, and by default, any terminal window you start will begin in your /home/*<username>* user directory. There are several other directories in / in addition to /home, /data, and /usr; these contain system files and should not be modified except under specific conditions when you know what you are doing.

UNIX Tutorial One

1.1 Listing files and directories

`ls` (list)

When you first login, your current working directory is your home directory. Your home directory has the same name as your user-name, which as noted above is your NC State Unity ID, and it is where your personal files and subdirectories are saved. The **prompt**, or symbol that tells you the system is waiting for a command, is **$**. To find out what is in your home directory, type **`ls`** (lowercase L, lowercase S) at the $ prompt, which will look like this:

$ **`ls`**

This command means 'list files in the current directory', although the meaning can be altered by including more **options** and **arguments**. Options are typically given as single letters following a hyphen, such as **-a** or **-n**, and these change what the command does. In some cases, multiple options may be needed; these can be combined into a continuous string after a single hyphen, for example **-alS**. Arguments are additional information provided to a command or to an option, and they can change where the command acts or which files or directories it acts on. We will explore how options and arguments work in more detail in the next steps of the tutorial.

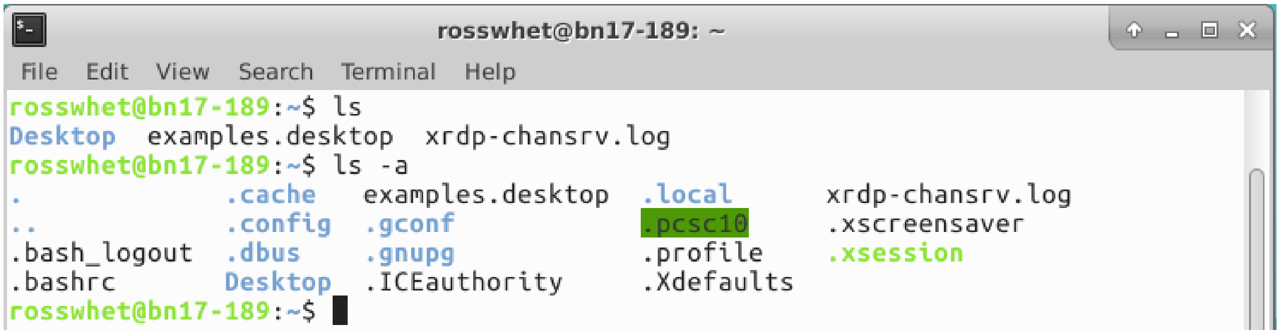
An example result of executing the `ls` command in my home directory.

The `ls` command lists most of the contents of the current user's current working directory. Files with names that begin with a dot (.) are known as hidden files and usually contain important program configuration information. They are hidden because you should not change them unless you are very familiar with UNIX! These files are not listed if you execute the `ls` command with no options.

To list **all** files in your home directory, including those whose names begin with a dot, type

$ **`ls -a`**

As you can see, **ls -a** lists files that are normally hidden.



An example result of executing the `ls -a` command in my home directory.

`ls` is an example of a command which can take options, and **`-a`** is an example of an option. The options change the behavior of the command. There are online manual pages that tell you which options a particular command can take, and how each option modifies the behavior of the command. You can display the manual page (often abbreviated man page) for the `ls` command by typing

$ **`man ls`**

to see an explanation of the options available for use with that command. Look for information on the **`-l`** option (lower-case L, not numeral 1) – what does that do to the output of the **`ls`** command? Type a q to get out of the man page display and back to a terminal prompt. Man pages often contain so much detailed information that it is difficult to find the most important facts. A community-based alternative called tldr (short for "Too Long – Didn't Read") is also available on the machine image, so type `tldr ls` at a terminal prompt to compare the output. The tldr display shows what the command does <u>above</u> the line where the command is displayed, so "list files one per line" is above the command `ls -1` (with a numeral one) which produces that output.

<u>Some key things to understand about commands</u> The system expects the elements of a command to be separated by whitespace characters (one or more spaces or tabs). The first element is the name of the command or program (**`ls`** for the example of the file listing program). The second element consists of any options, which can often be combined, e.g. **`-laS`** for the ls command. The third element is optional, and includes any arguments required by the options. For example, the `ls` command will list the contents of any directory provided as an argument, provided you have permission to view the contents of the directory (more about permissions later). In another example, the `cut` command discussed later has a **`-f`** option that determines which column of data is returned, and this requires an argument (the column number). The fourth element of a command is any argument required by the command itself – for example, an input file to be processed. The final element of the command is a return (the Enter or Return key) – this is the signal to the system that the command is complete and ready to be processed.

1.2 Making Directories

`mkdir` (make directory)

We will now make a subdirectory in your home directory to hold the files you will be creating and using in the course of this tutorial. To make sure you are in your home directory, type

`cd`

at a terminal prompt – when given without any options or arguments, this command means "change to my home directory". To create a subdirectory called `unixstuff` in your current working directory type

$ **`mkdir unixstuff`**

In this case, there are no options, just the command `mkdir` and the argument `unixstuff` (the name of the directory to create). Nothing obvious happens – you just get another terminal prompt. This is typical – successful completion of a command often produces no output to the terminal, while any problem can lead to an error message (which may or may not be informative).

To see the directory you have just created, type

$ **`ls`**

1.3 Changing to a different directory

`cd` (change directory)

The command `cd` *directory* means change the current working directory to *'directory'*. The current working directory may be thought of as the directory you are in, i.e. your current position in the file-system tree. The *'directory'* is an argument to the `cd` command that specifies which directory you want to change into. If you don't specify an argument, the default is your home directory.

To change to the directory you have just made, type

$ **`cd unixstuff`**

Type `ls` to see the contents (which should be nothing, because newly-created directories are empty).

Exercise 1a

Make another directory inside the **unixstuff** directory called **backups**

1.4 The directories . and ..

Still in the **unixstuff** directory, type

$ **`ls -a`**

As you can see, in the **unixstuff** directory (and in all other directories), there are two special directories called (**.**) and (**..**)

The current directory (.)

In UNIX, (**.**) means the current directory, so typing

$ **`cd .`** (NOTE: there is a space between the command `cd` and the dot) means "change to the current directory", or (in other words) stay where you are in the **unixstuff** directory.

This may not seem very useful at first, but using (**.**) as the name of the current directory will save a lot of typing, as we shall see later in the tutorial.

The parent directory (..)

(**..**) means the parent of the current directory, so typing

$ **`cd ..`**

will take you one directory up the hierarchy (back to your home directory). Try it now.

Reminder: typing **`cd`** with no argument always returns you to your home directory. This is very useful if you are lost in the file system.

1.5 Pathnames

pwd (print working directory)

Pathnames enable you to work out where you are in relation to the whole file-system. For example, to find out the absolute pathname of your home-directory, type `cd` to get back to your home-directory and then type

$ **pwd**

The full pathname should be `/home/<username>` , which means that your home directory is in the **home** sub-directory, which is in the top-level root directory called "`/`". Recall that <*username*> is short for "your NC State Unity ID".

Exercise 1b

Use the commands **`cd`**, **`ls`** and **`pwd`** to explore the file system, looking at the subdirectories and files in the `/data` and `/usr/local` directories.

(Remember, if you get lost, type **`cd`** by itself to return to your home-directory)

1.6 More about home directories and pathnames

Understanding pathnames

First type **`cd`** to get back to your home-directory, then type

$ **ls unixstuff**

to list the contents of your unixstuff directory. Now

type

$ **ls backups**

You will get a message like this -

backups: No such file or directory

The reason is the **backups** directory is not in your current working directory; instead it is in the `unixstuff` directory. To use a command on a file (or directory) not in the current working directory (the directory you are currently in), you must either `cd` to the correct directory, or specify the path as an argument to the `ls` command. To list the contents of your `backups` directory, you can type either a relative (starting from the current directory) or an absolute path (starting from the root directory). For example, from your home directory, the relative path is:

$ **ls unixstuff/backups**

~ (your home directory)

Home directories can also be referred to by the tilde ~ character, which can be used to specify paths starting at your home directory. So typing

$ **ls ~/unixstuff**

will list the contents of your unixstuff directory, no matter where you currently are in the file system.

What do you think

$ **ls ~**

would list?

What do you think

$ **ls ~/..**

would list?

What do you think

$ **ls /**

would list?

Summary

Command	Meaning
`ls`	list files and directories in the current directory
`ls -a`	list all files (including hidden files) and directories in the current directory
`mkdir`	make a directory
`cd` *directory*	change to named directory
`cd`	change to home-directory
`cd ~`	change to home-directory
`cd ..`	change to parent directory
`pwd`	display the path of the current directory

UNIX Tutorial Two

2.1 Copying Files

cp (copy)

`cp` *file1 file2* is the command which makes a copy of **file1** in the current working directory and calls it **file2**

What we are going to do now is copy a file from the course website, and save the copy to your unixstuff directory, using the `wget` command.

First, `cd` to your **unixstuff** directory.

$ **cd ~/unixstuff**

Then at the UNIX prompt, type,

wget http://www4.ncsu.edu/~rosswhet/BIT815/resources/documents/science.txt

This command downloads the file **science.txt** to the current directory, keeping the same name.

Exercise 2a

Create a backup of your **science.txt** file by copying it to a file called **science.bak**

2.2 Moving files

`mv` (move)

`mv` *file1 file2* moves (or renames) **file1** to **file2**

To move a file from one place to another, use the `mv` command. This has the effect of moving rather than copying the file, so you end up with only one file rather than two.

It can also be used to rename a file, by moving the file to the same directory, but giving it a different name.

We are now going to move the file science.bak to your **backups** directory.

First, change directories to your unixstuff directory (see section 1 for a reminder if you need it). Then, inside the **unixstuff** directory, type

```
$ mv science.bak backups/
```

Type **`ls`** and **`ls backups`** to see if it has worked.

2.3 Removing files and directories

`rm` (remove), `rmdir` (remove directory)

To delete (remove) a file, use the `rm` command. As an example, we are going to create a copy of the **science.txt** file then delete it.

Inside your **unixstuff** directory, type

```
$ cp science.txt tempfile
$ ls
$ rm tempfile
$ ls
```

You can use the `rmdir` command to remove a directory (make sure it is empty first). Try to remove the **backups** directory. You will not be able to since UNIX will not let you remove a non-empty directory with the `rmdir` command.

Exercise 2b

Create a directory called **tempstuff** using `mkdir`, then remove it using the `rmdir` command.

2.4 Displaying the contents of a file on the screen

`clear` (clear screen)

Before you start the next section, you may like to clear the terminal window of the previous commands so the output of the following commands can be clearly understood.

At the prompt, type

```
$ clear
```

This will clear all text and leave you with the `$` prompt at the top of the window.

`cat` (concatenate)

The command `cat` can be used to display the contents of a text file on the screen.

```
$ cat science.txt
```

As you can see, the file is longer than the size of the window, and it scrolls past so quickly that much of it is unreadable.

less

The command `less` writes the contents of a file onto the screen one page at a time. Type

$ **`less science.txt`**

Press the [**space-bar**] or **`f`** to move forward one page in the file, **`b`** to go back a page, and **`q`** if you want to quit reading. As you can see, `less` is used in preference to `cat` for long files.

head

The `head` command writes the first ten lines of a file to the screen.

First clear the screen then type

$ **`head science.txt`**

Then type

$ **`head -5 science.txt`**

What difference did the -5 make to the output of the head command?

tail

The `tail` command writes the last ten lines of a file to the screen.

Clear the screen and type

$ **`tail science.txt`**

Q. How can you view the last 15 lines of the file?

2.5 Searching the contents of a file

Simple searching using less

Using less, you can search through a text file for a keyword (pattern). For example, to search through `science.txt` for the pattern **'science'**, type

$ **`less science.txt`**

then, while you are viewing the file display in less, type a forward slash [/] followed by the word you want to search for:

/science

As you can see, `less` finds and highlights the keyword. Type **n** to search for the next occurrence of the keyword. If you have a mouse with a scroll wheel, you may be able to scroll up and down through the file to look for other occurrences of the search term, or you can use the **f** and **b** keys to move forward and backward in the file. If you want to turn off the highlighting of the search terms found in the file, hit the ESC key then type **u**. When you are finished viewing a file using `less`, type **q** to quit the program and return to the prompt. Use the **man less** command to read the man page about less – there are many more options and keyboard shortcuts available to make the program more useful and more powerful.

grep ("**g**lobal **r**egular **e**xpression **p**rint"**)**

grep is one of many standard UNIX utilities. It searches files for specified words or patterns. First clear the screen, then type

$ **`grep science science.txt`**

As you can see, grep has printed out each line containing the search pattern “science”.

Try typing

$ **`grep Science science.txt`**

The grep command is case sensitive; it distinguishes between Science and science. To ignore upper/lower case distinctions, use the -i option, i.e. type

```
$ grep -i science science.txt
```

To search for a phrase or pattern, you must enclose it in single quotes (the apostrophe symbol) or double quotes, so the space between words is not interpreted as the space between the pattern to search for, and the name of the file to search. For example, to search the science.txt file for the phrase "spinning top", using the case-insensitive option, you would have to type

```
$ grep -i 'spinning top' science.txt
```

Try typing

```
$ grep -i spinning top science.txt
```

to see what happens.

The grep program interprets this as a command to search for the pattern “spinning” in the file “top”, then in the file “science.txt”. The shell returns an error telling you that it cannot find a file “top”, then prints the lines from the science.txt file that contain the pattern “spinning”. This result shows that grep interprets the first “word” (set of characters surrounded by spaces) as the pattern to search for, and each subsequent “word” as the name of a file to search. The space character is a delimiter that separates parts of the command that are interpreted in different ways. As you can imagine, this means that filenames that contain spaces are not interpreted correctly by the bash shell unless they are surrounded by quotes, as in the first “spinning top” example above. The web page http://faculty.salina.k-state.edu/tim/unix_sg/shell/metachar.html has a list of shell metacharacters; these are all characters used to convey specific kinds of information to the shell that go beyond the simple meaning of the character itself. The difference between single and double quotes is that no shell metacharacters are interpreted inside single quotes, but some metacharacters are interpreted inside double quotes.

The -E option tells grep to use "extended format regular expressions”, which are ways of describing search patterns that use ?, +, {, |, (and) as metacharacters to allow more flexible and powerful searches. Such extended format regular expressions are commonly enclosed in double-quotation marks so they are not interpreted by the shell, which uses some of the same metacharacters in different ways.

```
$ grep -E "this|that" science.txt
```

This searches for either the word “this” or the word “that” – the | is interpreted as a “meta-character” that can means either pattern can match. This is a very simple example of a regular expression; this topic is covered in more detail in the man page for grep, and in a separate RegularExpressions.pdf document available on the course website.

Some of the other options of grep are:

-v display those lines that do NOT match the search pattern
-n precede each matching line with the line number
-c print only the total count of matched lines

Try some of these options and see the different results. Don't forget, you can use more than one option at a time. For example, the number of lines that do NOT contain words matching the pattern `".and"` (ie, any character except newline, followed by “and”) is given by:

```
$ grep -vc ".and " science.txt
```

wc (word count)

The `wc` command (short for word count) is a handy utility. To do a word count on `science.txt`, type

`$ wc -w science.txt`

To find out how many lines the file has, type

`$ wc -l science.txt` (remember, this option uses a lower-case letter L)

`file` (determine file type)

To find out what type of file (text, image, or binary data) the `science.txt` file is, type

`$ file science.txt`

In the case of the `science.txt` file, the ".txt" extension suggests this is a text file, but such informative file extensions are not required in Unix or Linux systems, so the `file` command is useful for finding the file type of files with uninformative names.

Summary

Command	Meaning
`cp` *file1 file2*	copy file1 and call it file2
`mv` *file1 file2*	move or rename file1 to file2
`rm` *file*	remove a file
`rmdir` *directory*	remove an empty directory
`cat` *file*	display a file
`less` *file*	display a file a page at a time
`head` *file*	display the first few lines of a file
`tail` *file*	display the last few lines of a file
`grep` *'keyword' file*	search a file for the exact word "keyword"
`wc` *file*	count number of characters/words/lines in file
`file` *file*	determine what type of file this is

UNIX Tutorial Three

3.1 Redirection

Most processes initiated by UNIX commands write to the standard output (that is, they write to the terminal screen), and many take their input from the standard input (that is, they read it from the keyboard). There is also a standard error output, where processes write their error messages, by default to the terminal screen.

We have already seen one use of the `cat` command to write the contents of a file to the screen.

Now type `cat` without specifying a file to read

`$ cat`

Then type a few words on the keyboard and press the [**Return**] key.

Finally hold the [**Ctrl**] key down and press [**d**] (written as **^D** for short) to end the input. What has happened?

If you run the `cat` command without specifying a file to read, it reads the standard input (the keyboard), and on receiving the 'end of file' (**^D**), copies it to the standard output (the screen).

In UNIX, we can redirect both the input and the output of commands.

3.2 Redirecting the Output

We use the > symbol to redirect the output of a command. For example, to create a file called **list1** containing a list of fruit, type

$ **`cat > list1`**

Then type in the names of some fruit. Press [**Return**] after each one.

```
pear
orange
banana
apple
```

`^D` {this means press [Ctrl] and [d] to stop}

What happens is the `cat` command reads the standard input (the keyboard) and the > redirects the output, which normally goes to the screen, into a file called **list1**

To read the contents of the file, type

$ **`cat list1`**

Exercise 3a

Using the above method, create another file called **list2** containing the following fruit: orange, plum, mango, grapefruit. Display the contents of **list2** to the screen to confirm that the file exists.

3.2.1 Appending to a file

The form >> appends standard output to a file. So to add more items to the file **list1**, type

$ **`cat >> list1`**

Then type in the names of more fruit

```
peach
grape
orange
plum
```

`^D` (Control D to stop)

To read the contents of the file, type

$ **`cat list1`**

You should now have two files. One contains eight items, the other contains four.

We will now use the cat command to join (concatenate) **list1** and **list2** into a new file called **biglist**. Type

$ **`cat list1 list2 > biglist`**

What this does is read the contents of **list1** and **list2** in turn, then redirect the combined text to the file **biglist**. It is useful to think of this is as a stream of data, flowing out of the files **list1** then **list2**, concatenated by the `cat` command, and then flowing into the file **biglist**. Much of the power of the Unix or Linux command line comes from the ability to carry out a series of operations on such streams of data.

To read the contents of the new file, type

$ **cat biglist**

3.3 Redirecting the Input

We use the < symbol to redirect the input of a command.

The command sort alphabetically or numerically sorts a list. Type

$ **sort**

Then type in the names of some animals. Press [Return] after each one.

dog
cat
bird
ape
^D (control d to stop)

The output will be ape
bird
cat dog

Using < you can redirect the input to come from a file rather than the keyboard. For example, to sort the list of fruit, type

$ **sort < biglist**

and the sorted list will be output to the screen. To

output the sorted list to a file, type,

$ **sort < biglist > slist**

Use cat to read the contents of the file **slist**

3.4 Pipes

Pipes are a means of transferring the output of one command directly to another command as input, without creating an intermediate file – moving the stream of data from one command to another, so to speak. For a bioinformatics perspective on the value of pipes in managing and analyzing large datasets, see Vince Buffalo's blog post on the topic.
For example, one method to get a sorted list of file names in the current directory is to type,

$ **ls > names.txt**
$ **sort < names.txt**

This is a bit slow and you have to remember to remove the temporary file `names.txt` when you have finished. What you really want to do is connect the output of the **ls** command directly to the input of the **sort** command. This is exactly what pipes do. The symbol for a pipe is the vertical bar | character, which is the shift character on the key above the Enter key on a US standard keyboard.

For example, typing

$ **ls | sort**

will give the same result as above, but faster, and without creating the intermediate file.

To find out how many files are in the current directory, type

$ **`ls | wc –l`**

Several options are available for the `sort` command; three useful ones are the `-n`, `-r`, and `-k` options. The `-n` option specifies sorting in numerical order (as opposed to 'lexical' order, which corresponds to the order of characters in the local character encoding scheme), the `-r` option specifies sorting in reverse order (descending values rather than ascending values), and the `-k` option allows specification of one or more fields (columns of values) to be used as the key on which sorting occurs. To see a list of all files in the /data directory on your VCL instance, sorted in order of decreasing file size, use the command

`ls –l /data | sort –nrk 5,5`

Listing files in order of decreasing file size is such a useful tool that there is a much easier way to do this – read the man page for the `ls` command to find out what it is. This is not uncommon in Linux systems – if a particular task arises frequently, there is often a specific command or command option that can carry out that task with a minimum of effort.

Exercise 3b

Using pipes, display all lines of **list1** and **list2** containing the letter 'g', and sort the result.

3.5 More commands: `cut`, `uniq`, `rev`, `fold`, and `tr`

The command **`cut`** is used to extract a column of values from a table of values. The default delimiter separating the columns is a tab character, but a different delimiter can be specified using the `-d` option, e.g. `-d" "` to specify a space as the delimiter. The column to be extracted is specified by the `-f` option (for "field") – columns are numbered beginning with 1 on the left side of the file.

The command **`uniq`** is used to recover a subset of items from a sorted list; depending on the options used, the subset can include only items that appear more than once, only items that appear exactly once, or one copy of every item that appears at least once. Use the command

`man uniq`

at the command line to display the manual page for the command **`uniq`** to learn how to use these options. One important note – the **`uniq`** command only compares elements in a list to adjacent elements in order to determine which are repeated or unique, so it is essential to sort the list before using the command. Compare the output of the following commands:

`uniq -d slist`

`uniq -c slist` versus **`uniq -c slist | sort –`**

`nrk1,1 uniq -u slist`

The command **`rev`** is used to reverse the order of characters in a text stream. If a file of many lines is provided in the stream, the order of characters on each line is reversed.

The command **`fold`** is used to introduce line breaks in text – for example, some FASTA-format DNA or protein sequence files are stored without line breaks in the sequence, so that each record occupies only two lines in the file – one for the header line, the other for the sequence. Piping such a file through the **`fold`** command is one way to introduce line breaks. The default action is to break after column 80, but other positions can be specified with the -w option.

The command **`tr`** is used to translate characters in the text stream, which can be useful for many tasks. For example, consider the process of writing the reverse-complement of a DNA sequence. Each character in the set {A, C, G, T} is translated into the corresponding character from the set {T, G, C, A} to create the complementary sequence, and the order of characters is then reversed to create the reverse complement. To demonstrate this, enter the command

```
echo AATGCATAGGG | tr ACGT TGCA | rev
```

A useful option for the **`tr`** command is the `-s` option, short for "squeeze-repeats" – this replaces a repeating string of each character listed with a single copy of the same character. This is helpful if you want to use the **`cut`** command to extract a column from a space-separated text stream in which variable numbers of space characters are used to separate columns. For example,

```
ls -l /data/chipseq | sort -nrk5,5 | cut -d" " -f5
```

does not return the expected column of file sizes sorted in decreasing order, because the **`cut`** command treats each space character as a separate delimiter (unlike the `sort` command, which merges adjacent spaces into a single delimiter). Including the "squeeze-repeats" option of the **`tr`** command solves this problem, and produces the desired result:

```
ls -l /data/chipseq | sort –nrk5,5 | tr –s " " | cut – d" " -f5
```

Summary

<table>
<tr><th>Command</th><th>Meaning</th></tr>
<tr><td>command > file</td><td>redirect standard output to a file</td></tr>
<tr><td>command >> file</td><td>append standard output to a file</td></tr>
<tr><td>command < file</td><td>redirect standard input from a file</td></tr>
<tr><td>command1 | command2</td><td>pipe the output of command1 to the input of command2</td></tr>
<tr><td><code>cat</code> file1 file2 > file0</td><td>concatenate file1 and file2 to file0</td></tr>
<tr><td><code>sort</code></td><td>sort data</td></tr>
<tr><td><code>cut</code></td><td>extract a column of values from tabular data</td></tr>
<tr><td><code>rev</code></td><td>reverse the order of characters on a line</td></tr>
<tr><td><code>tr</code></td><td>translate one character set to another</td></tr>
<tr><td><code>uniq</code></td><td>identify unique or repeated values in a sorted list</td></tr>
</table>

Answer to Exercise 3b: `cat list1 list2 | grep g | sort` is one way to do this; another way is to take advantage of the fact that the grep function accepts multiple filename arguments, and just use

`grep g list1 list2 | sort` – the output is slightly different, because grep returns the name of the file where the pattern was found as the first item in each line, if it reads files directly, but not if it receives input from a pipe.

UNIX Tutorial Four

4.1 Wildcards

The * wildcard

The character ***** is called a wildcard or meta-character, and will match against none, one, or more of any character(s) in a file (or directory) name. A more detailed explanation of wildcard characters is given in the FileGlobbing.pdf document from the course website.

For example, in your **unixstuff** directory, type

```
$ ls list*
```

This will list all files in the current directory starting with **list....**

Try typing

```
$ ls *list
```

This will list all files in the current directory ending with **....list**

The ? wildcard

The meta-character ? will match exactly one character.
So **?ouse** will match files like **house** and **mouse**, but not **grouse**. Compare the output from the *list command (above) to the output of

```
$ ls ?list
```

4.2 Filename conventions

We should note here that a directory is merely a special type of file. So the rules and conventions for naming files apply also to directories.

In naming files, meta-characters (those with special meanings to the shell, such as **/ * & $**) , should be avoided. Also, avoid using spaces within names, because the shell interprets spaces as boundaries between commands, options, and arguments. The safest way to name a file is to use only alphanumeric characters, that is, letters and numbers, together with _ (underscore) and . (dot). Reminder: http://faculty.salina.k-state.edu/tim/unix_sg/shell/metachar.html has a list of meta-characters and how they are used to convey special information to the shell.

Good filenames	**Poor filenames**
project.txt	project text
my_big_program.c	my big program.c
fred_dave.doc	fred & dave.doc

File names conventionally start with a lower-case letter, and may end with a dot followed by a group of letters indicating the contents of the file. For example, all files consisting of C code may be named with the ending **.c**, for example, **prog1.c** . Then in order to list all files containing C code in a particular directory, you need only type **ls *.c** in that directory.

As a general rule, any time you create a set of files that are part of the same project and will be analyzed together, it is good practice to use names that fit into a single pattern, so that all filenames can be matched easily. For example, sequence files derived from samples in 96-well plate format can be named using the plate number, row letter, and column number, but it is wise to include leading zeros for numbers less than ten. For example, it is easier to write a pattern that matches both p01A01 and p10H12 than it is to match p1A1 and p10H12, and it is also easier to extract specific items of information (such as the row identifier) from a set of names if the position within each name is always the same.

4.3 Getting Help

On-line Manual Pages

There are on-line manual pages that give information about most commands. The manual pages tell you which options a particular command can take, and how each option modifies the behavior of the command. Type `man` *`command`* to read the manual page for a particular command, and *q* to leave the man page and return to the terminal prompt. Man pages use the `less` pageviewing program, so all the search functions and other tools of `less` are available for man pages as well.

For example, to find out more about the **wc** (word count) command, type

$ **`man wc`**

Alternatively

$ **`whatis wc`**

gives a one-line description of the command, but omits any information about options etc. The program tldr is installed on the virtual machine image; this gives practical examples of how to use options for many common commands, but is not yet as comprehensive as man pages.

Apropos

When you are not sure of the exact name of a command,

$ **`apropos keyword`**

will give you the commands with “keyword” in their manual page header. For example, try typing

$ **`apropos copy`**

Summary

Command	Meaning
`*`	match any number of characters
`?`	match one character
`man` *command*	read the online manual page for a command
`whatis` *command*	brief description of a command
`apropos` *keyword*	match commands with keyword in their man pages

UNIX Tutorial Five

5.1 File system security (access rights)

In your **unixstuff** directory, type

$ **`ls -l`** (lower-case letter L, for long listing!)

You will see that you now get lots of details about the contents of your directory. Each file (and directory) has associated access rights, which may be found by typing **ls -l**. This option also gives additional information as to which user (*lubuntu* in this example) and group (*bit815* in this example) owns the file:

```
-rwxrw-r-- 1 lubuntu bit815 2450 Sept29 11:52 file1
```

In the left-hand column is a 10 symbol string consisting of an initial character followed by three groups that can have the symbols r, w, x, or -. The initial character can be a -, d, l, or sometimes s or S. If d is present, it indicates a directory, and l indicates a link; if - is the starting symbol of the string, it indicates a file.

The 9 remaining symbols indicate the permissions, or access rights, for three categories of users.

- The left group of 3 gives the file permissions for the user that owns the file or directory (the *lubuntu* user has read, write, and execute permissions on file1 in the above example);
- The middle group gives the permissions for the group of people to whom the file or directory belongs (the *bit815* group has read and write permissions on file1 in the above example);
- The rightmost group gives the permissions for all others (other users have only read permission on file1).

The symbols r, w, and x have slightly different meanings depending on whether they refer to a simple file or to a directory.

Access rights on files.

- r (or -), indicates read permission (or lack of it), that is, the presence or absence of permission to read and copy the file
- w (or -), indicates write permission (or lack of it), that is, the permission (or otherwise) to change a file
- x (or -), indicates execution permission (or lack of it), that is, the permission to execute a file, where appropriate

Access rights on directories.

- r allows users to list files in the directory;
- w means that users may delete files from the directory or move files into it;
- x means the right to access files in the directory. This implies that you may read files in the directory provided you have read permission on the individual files.

  So, in order to read a file, you must have execute permission on the directory containing that file, and hence on any directory containing that directory as a subdirectory, and so on, up the tree.

Some examples

`-rwxrwxrwx`	a file that everyone can read, write and execute (and delete).
`-rw-------`	a file that only the owner can read and write - no-one else can read or write and no-one has execution rights (e.g. your mailbox file).

5.2 Changing access rights

chmod (changing a file mode)

Only the owner of a file or the root user can use `chmod` to change the permissions of a file. The options of chmod are as follows.

Symbol	Meaning
u	user
g	group
o	other
a	all
r	read – also expressed as octal value 4
w	write (and delete) – also expressed as octal value 2
x	execute (and access directory) – also expressed as octal value 1
+	add permission
-	take away permission

For example, to remove read write and execute permissions on the file **biglist** for the group and others, type

```
$ chmod go-rwx biglist
```

This will leave the other permissions unaffected.

To give read and write permissions on the file **biglist** to all,

```
$ chmod a+rw biglist
```

Octal values are a more compact way of setting the permissions for all three classes (user, group, and others) in a single command. The permissions for each class are expressed as the sum of the permitted operations (read=4, write=2, execute=1), so `chmod 755 biglist.txt` sets the permissions on the file biglist.txt to rwx for user and r-x for group and others.

Exercise 5a

Try changing access permissions on the file **science.txt** and on the directory **backups** to allow only read access for all three classes.

Use `ls -l` to check that the permissions have changed, and try different operations to see what is allowed and what is not allowed for files and directories with r-- permissions set.

5.3 Processes and Jobs

A process is an executing program identified by a unique PID (process identifier). To see information about processes active in your terminal session, with their associated PID and status, type

```
$ ps
```

A process may be in the foreground, in the background, or be suspended. In general the shell does not return the UNIX prompt until the current process has finished executing.

Some processes take a long time to run and hold up the terminal. Backgrounding a long process has the effect that the UNIX prompt is returned immediately, and other tasks can be carried out while the original process continues executing.

Running background processes

To background a process, type an **&** at the end of the command line. For example, the command sleep waits a given number of seconds before continuing. Type

```
$ sleep 10
```

This will wait 10 seconds before returning the command prompt $. Until the command prompt is returned, you can do nothing except wait.

To run sleep in the background, type

```
$ sleep 10 &
```

[1] 6259 (The process ID number returned by your system will be different)

The **&** runs the job in the background and returns the prompt straight away, allowing you to run other programs while waiting for that one to finish.

The first line in the above example is typed in by the user; the next line, indicating job number and PID, is returned by the machine. The user is being notified of a job number (numbered from 1) enclosed in square brackets, together with a PID and will be notified again when a background process is finished.

Backgrounding a current foreground process

At the prompt, type

```
$ sleep 1000
```

You can suspend the process running in the foreground by typing **^Z**, i.e.hold down the [**Ctrl**] key and type [**z**]. Then to put it in the background, type

```
$ bg
```

Note: do not background programs that require user interaction, such as the command-line text editor **vi**

5.4 Listing suspended and background processes

When a process is running, backgrounded or suspended, it will be entered onto a list along with a job number. To examine this list, type

```
$ jobs
```

An example of a job list could be [1]

Suspended sleep 1000
[2] Running netscape
[3] Running matlab

To restart (foreground) a suspended processes, type

$ `fg $` *jobnumber*

For example, to restart sleep 1000, type

```
$ fg $ 1
```

Typing fg with no job number foregrounds the last suspended process.

5.5 Killing a process

kill (terminate or signal a process)

It is sometimes necessary to kill a process (for example, when an executing program is in an infinite loop)

To kill a job running in the foreground, type **^C** (control c). For example, run

```
$ sleep 100
```

then hold down the Ctrl key and hit the C key. The screen will show ^C , and the process should end.

To kill a suspended or background process, type

`$ kill $`*jobnumber*

For example, run

```
$ sleep 100 &
$ jobs
```

If it is job number 4, type

```
$ kill %4
```

To check whether this has worked, examine the job list again to see if the process has been removed.

ps (process status)

Alternatively, processes can be killed by finding their process numbers (PIDs) and using kill *PID_number*

```
$ sleep 1000 &
$ ps
```

PID TTY TIME COMMAND
20077 pts/5 0:05 sleep 1000
21563 pts/5 0:00 bash
21873 pts/5 0:25 ps

Note – the PIDs in the first column should be different if you run this on your system, and the running processes may be different as well, but the format will be similar.

To kill off the process **sleep 1000**, type

`$ kill <PID>` (using the process id number returned on your system for the *sleep* process) and then type ps again to see if it has been removed from the list.

If a process refuses to be killed, uses the **-9** option, i.e. type

```
$ kill -9 <PID>
```

Note: Only the root user can kill other users' processes – ordinary users cannot do this.

Summary

Command	Meaning
`ls -la`	list access rights for all files
`chmod` [*options*] *file*	change access rights for named file
command &	run command in background
`^C`	(Ctrl key + c key) kill the job running in the foreground
`^Z`	(Ctrl key + z key) suspend the job running in the foreground
`bg`	background the suspended job

`jobs`	list current jobs
`fg 1`	foreground job number 1
`kill %1`	kill job number 1
`ps`	list current processes
`kill 26152`	kill process number 26152

UNIX Tutorial Six

Other useful UNIX commands

df

The `df` command reports on the space left on the file system. For example, to find out how much space is left on the local drive of your virtual machine instance, type

```
$ df .
```

du

The du command outputs the number of kilobytes used by each subdirectory. This is useful if you want to find out which directory has the most files. In your home-directory, type

```
$ du -s *
```

The **-s** flag will display only a summary (total size) and the * means all files and directories.

gzip

This reduces the size of a file, thus freeing valuable disk space. For example, you can create a gzipped tar archive of all the list files in the unixstuff directory named **list.tgz** by executing

```
$ cd ~/unixstuff
```

```
$ tar –czf list.tgz *list*
```

The -c option tells the `tar` command to create an archive, the -z option specifies that the archive is to be gzip-compressed after creation, and the -f specifies the file name of the archive. The `*list*` pattern is a file glob that matches all files in the current directory that contain “list” anywhere in the file name. You can calculate the total size of these files using the `bc` command, which is a command-line basic calculator. First change to the unixstuff directory (if you are not already there), and execute `ls -l *list*` to see the sizes of the four files **biglist**, **list1, list2**, and **slist**:

```
$ ls –l *list*
```

To send the values of the file sizes (found in column 5) to the bc calculator with + symbols inserted, execute the following command:

```
$ ls –l *list* | tr –s " " | cut -d" " –f5 | paste –sd+ | bc
```

The `translate` and `cut` commands were described in section 3.5; the `paste` command joins lines (by default) or characters on a line (using the `-s` option) together, using the character after the `-d` option as the delimiter. You can see the output from the series of commands starting with ls and ending with paste by removing the final `| bc` from the command. Compare the sum of the file sizes with the size of the **list.tgz** compressed archive.

To gzip a single file, use the `gzip` command:

```
$ gzip biglist
```

To expand a gzipped file, use the `gunzip` command

`$ gunzip biglist.gz` Note that the biglist.gz file disappears if you use gunzip on it.

zcat

zcat will read gzipped files without needing to uncompress them first. First compress biglist, then view the file using zcat:

```
$ gzip biglist
```

`$ zcat biglist.gz` What happens to the biglist.gz file if you use zcat on it?

This file is small, so you won't need to pipe the output through less, but you can, just to see that the pipe does what you would expect. What happens if you use cat instead of zcat?

```
$ zcat biglist.gz | less
```

Use the `file` command to see what kind of file biglist.tgz is.

zgrep

The task of working with gzip-compressed files is common enough on Linux systems that specific commands such as `zcat` and `zgrep` were written to allow display or searching of compressed files without having to decompress them and save a copy of the decompressed file. For small files like the examples we have been working with, the difference is trivial, but for genome data files that may contain many gigabytes of data, this is very important. Try searching the biglist.gz file to determine the number of occurrences of the regular expression pattern ".range".

```
zgrep –c ".range" biglist.gz
```

file

file classifies the named files according to the type of data they contain, for example ascii (text), pictures, compressed data, etc.. To report on all files in your home directory, type

```
$ file *
```

diff

This command compares the contents of two files and displays the differences. Suppose you have a file called **file1** and you edit some part of it and save it as **file2**. To see the differences type

`$ diff` *file1 file2*

Lines beginning with a < denote material in file1 but not file2, while lines beginning with a > denote material in file2 but not file1.

find

This searches through the directories for files and directories with a given name, date, size, or any other attribute you care to specify. It is a simple command but with many options - you can read the manual by typing `man find`.

To search for all files with the extension **.txt**, starting at the current directory (.) and working through all sub-directories, then printing the name of the file to the screen, type

```
$ find . -name "*.txt" -print
```

To find files over 1Mb in size, and display the result as a long listing, type

```
$ find . -size +1M -ls
```

free

This command reports the total amount of random-access memory (RAM) and swap space (hard drive space allocated for use as temporary storage) on the system, along with a breakdown of how much is used for active processes, cached information, or free. This is useful for evaluating how much of the system resources are in use before you start a memory-intensive process. The `-m` option displays the output in units of megabytes instead of bytes, to make evaluating it easier. The column labeled "free" shows unused memory or swap space resources, and the column labeled "available" is the total amount of available RAM. Any space occupied by cached or buffered information can be freed if the system needs the memory, but "used" memory is not available for any new process that starts.

UNIX Tutorial Seven

7.1 Installing Software as Ubuntu packages

Many public domain software packages are already installed on the virtual machine system. Linux users often need to download and install software packages to add new capabilities to their systems or to try new methods. Many programs are available as pre-compiled and ready-to-install packages. If these are available in the "repositories", or central servers that make Ubuntu software available, then they can be installed using the command:

$ **`sudo apt install`** *package-name*

We will install a package from the Ubuntu repositories that are required for compiling and installation of the STACKS software package. Packages that are needed in order for other programs to run are called "dependencies", because the program of interest (STACKS, in this case) depends on the presence of the required packages.

From a terminal prompt, type the command

$ **`sudo apt install libsparsehash-dev`**

Not all software is available in the Ubuntu repositories, however. One example is the program RStudio, which has already been installed on the Biostar_DNASeq machine image. The RStudio package was downloaded from https://www.rstudio.com/products/rstudio/download/ as a Debian installation package called **rstudio-1.1.383-amd64.deb**. Lubuntu 16.04 is a member of the Debian family of Linux systems, so installable program packages are identified by a **.deb** extension. The program was installed using the command-line program **`dpkg`**, with the **`–i`** option to specify that we are installing rather than removing a package. The **`dpkg –i`** command is issued as root user using the **`sudo`** command, which tells the system to install those programs in the search path so every user on the system will have access to them. You need not install RStudio now, because a recent version is already installed, but the software is updated regularly, so you will need to install an updated version in the future if you want to keep the system current.

7.2 Compiling UNIX software source code

Not all software is available pre-packaged into installable Debian or Ubuntu packages. Many programs are available only as source code, which must be compiled in order to produce a usable program on your local computer.

There are a number of steps needed to compile source code for software.

- Locate and download the source code (which is usually compressed in a gzipped Tape ARchive or **tar** file with the extension **.tar.gz** or **.tgz**)
- Unpack the source code

- Compile the code
- Install the resulting executable
- Set paths to the installation directory (if installation is not done as root user)

Of the above steps, compiling can be the most difficult, but many tools exist to make this process easier.

Compiling Source Code

All high-level language code must be converted into a form the computer understands. For example, C language source code is converted into a lower-level language called assembly language. The assembly language code made by the previous stage is then converted into object code which the computer understands directly. The final stage in compiling a program involves linking the object code to code libraries which contain certain built-in functions. This final stage produces an executable program.

To do all these steps by hand is complicated and beyond the capability of the ordinary user. A number of utilities and tools have been developed for programmers and end-users to simplify these steps.

make and the Makefile

The **`make`** command allows programmers to manage large programs or groups of programs. It aids in developing large programs by keeping track of which portions of the entire program have been changed, compiling only those parts of the program which have changed since the last compile.

The **`make`** program gets its set of compile rules from a text file called **Makefile** which resides in the same directory as the source files. It contains information on how to compile the software, such as the optimization level, or whether to include debugging information in the executable. It also contains information on where to install the finished compiled binaries (executables), manual pages, data files, dependent library files, configuration files, etc.

Some packages require you to edit the Makefile by hand to set the final installation directory and any other parameters. However, many packages are now being distributed with the GNU configure utility.

configure

As the number of UNIX variants increased, it became harder to write programs which could run on all variants. Developers frequently did not have access to every system, and the characteristics of some systems changed from version to version. The GNU configure and build system simplifies the building of programs distributed as source code. All programs are built using a simple, standardized, two-step process. The program builder need not install any special tools in order to build the program.

The configure shell script attempts to guess correct values for various system-dependent variables used during compilation. It uses those values to create a **Makefile** in each directory of the package.

The simplest way to compile a package is:

1. `cd` to the directory containing the package's source code and read the README file, if there is one.
2. Type `./configure` to configure the package for your system, including any options that are appropriate, as outlined in the README file for the software package. For STACKS v 1.48, the recommended form is `./configure --enable-sparsehash`
3. Type `make` to compile the package.

4. Optionally, type `make check` to run any self-tests that come with the package. The STACKS v1.48 package does not include any self-tests, so running this command will produce an error message stating that no checks are available.
5. Type `sudo make install` to install the programs and any data files and documentation in the appropriate directories in the search path. There is not much point in doing this on the VCL instance, because changes are not saved, but you can try it to see what happens.
6. Optionally, type `make clean` to remove the program binaries and object files from the source code directory.

The configure utility supports a wide variety of options. You can usually use the **`--help`** option to get a list of interesting options for a particular configure script.

The only generic options you are likely to use are the **`--prefix`** and **`--exec-prefix`** options. These options are used to specify the installation directories.

The directory named by the **`--prefix`** option will hold machine independent files such as documentation, data and configuration files.

The directory named by the **`--exec-prefix`** option, (which is normally a subdirectory of the **`--prefix`** directory), will hold machine dependent files such as executables.

7.3 Downloading source code

For this example, we will compile the source code for the STACKS program written by Julian Catchen to analyze RAD-seq data.

The **stacks-1.48.tar.gz** source code archive has been downloaded from http://catchenlab.life.illinois.edu/stacks/ and saved in the `/data` directory of the virtual machine image. Many of the other programs installed on the machine image were obtained from either https://sourceforge.net or from https://github.com, which are both sites that host source code for a wide variety of open-source software projects including many related to bioinformatics.

7.4 Extracting the source code

Go into your home directory and list the contents of the `/data` directory.

```
$ cd
$ ls –l /data
```

As you can see, two of the filenames end in tar.gz. The `tar` (short for Tape ARchive) command turns several files and directories into one single tar file. This can then be compressed using the `gzip` command (to create a tar.gz or .tgz file), or the `bzip2` command (to create a tar.bz2 file).

You can unzip the file using the `gunzip` command, then unpack the tar file using the `tar -xf` command ("extract file"), but using the -z option to `tar` allows decompressing the zip file and unpacking the archive in one step. Type **man tar** at the command prompt to read more information about the tar command and its options.

```
$     tar -xzf /data/stacks-1.48.tar.gz
```

Again, list the contents of your home directory using ls -l, then change to the **stacks-1.48** sub-directory.

```
$ cd stacks-1.48
```

7.5 Configuring and creating the Makefile

The first thing to do is carefully read the **README** and **INSTALL** text files (use the `less` command). These contain important information on how to compile and run the software.

The package uses the GNU configure system to compile the source code.

Run the configure utility, using the `--enable-sparsehash` option so the compiler knows that library is available.

```
$ ./configure –-enable-sparsehash
```

If configure has run correctly, it will have created a Makefile with all necessary options. You can view the Makefile if you wish (use the less command), but do not edit the contents of this file.

7.6 Building the package

Now you can go proceed to build the package by running the make command.

```
$ make
```

After several minutes (depending on the speed of the computer), the executables will be created. You can check to see everything compiled successfully by typing

```
$ make check
```

If everything is okay, you can now install the package.

```
$ sudo make install
```

This will install the files into the appropriate directories in the search path, so you can run the program without the need to specify the complete path to the executable programs. This change won't be saved, but I will make sure the program will be installed and available by the next class period.

7.7 Running the software

You are now ready to run the software (assuming everything worked).

If you list the contents of the stacks-1.48 directory, you will see a number of subdirectories: config, php, scripts, sql, and src.

In addition to these directories (shown in blue in the terminal display), you will also see executable programs (shown in green in the terminal display) called ustacks, pstacks, cstacks, sstacks, populations, and genotypes (among others). To run any of these programs, just type the name of the program at the command prompt and supply the required options. For a list of the options, type the program name with no options – for example, to see options for the ustacks program, type:

```
$  ustacks
```

Note that the ustacks program has options and takes arguments, just as with the command-line utilities we have been using so far. The output from these programs can be re-directed to a file, just as we learned with the example of the `cat` command.

UNIX Tutorial Eight

8.1 UNIX Variables

Variables are a way of passing information from the shell to programs when you run them.

Programs look in the local “environment” for particular variables and if they are found will use the values stored. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.

Standard UNIX variables are split into two categories, environment variables and shell variables. In broad terms, shell variables apply only to the current instance of the shell and are used to set short-term working conditions; environment variables have a farther reaching significance, and those set at login are valid for the duration of the session. By convention, environment variable names are often written in UPPER CASE; the value of the variable is denoted by adding a $ to the beginning of the name.

8.2 Environment Variables

An example of an environment variable is the OSTYPE variable. The value of this is the current operating system you are using. Type

`$ echo $OSTYPE`

More examples of environment variables are

- USER (your login name)
- HOME (the path name of your home directory)
- HOST (the name of the computer you are using)
- ARCH (the architecture of the computers processor)
- DISPLAY (the name of the computer screen to display X windows)
- PRINTER (the default printer to send print jobs)
- PATH (the directories the shell should search to find a command)

Finding out the current values of these variables.

The **`printenv`** command displays the current values of all ENVIRONMENT variables, which is typically a fairly long list. The value of individual variables can be displayed using the **`echo`** command. For example,

$ **`echo`** `$PATH`

will print to the screen the list of directories in which the shell will search to find programs or commands typed at the prompt.

Environment variables are set in the **bash** shell using the **export** command, as follows:

$ **`export`** `VAR=value`

sets the value of the environmental variable VAR to ‘value’. Note that no space is allowed either before or after the equal sign, and the $ is not used in the assignment. The variable VAR will have the value 'value' only for the duration of the current login session, but all child processes initiated from the current shell session inherit this variable with this value.

8.3 Shell Variables

SHELL variables are set using a simple assignment, and the current value is displayed using echo.

$ `VAR=value`

$ echo $VAR

Note that the variable name does not include the $ when the value is assigned (the $ at the beginning of that line is the prompt), but must include the $ when the value is retrieved. The value of shell variables is not inherited by child processes, but instead is only available within the shell session in which the value was assigned. The exception to this is variables assigned in the `.bashrc` file (see below), which are inherited in all bash shell sessions. A shell variable can be exported to the environment after it is created, if the need arises, by executing the command

$ **export** VAR

8.4 Using and setting variables

Each time you login to a UNIX host, the system looks in your home directory for initialization files. Information in these files is used to set up your working environment. The **bash** shell uses two files called **`.login`** and **`.bashrc`** (note that both file names begin with a dot).

At login the shell first reads **`.bashrc`** followed by **`.login`**

`.login` is to set conditions which will apply to the whole session and to perform actions that are relevant only at login.

`.bashrc` is used to set conditions and perform actions specific to the shell and to each invocation of it.

The guidelines are to set ENVIRONMENT variables in the **`.login`** file and SHELL variables in the **`.bashrc`** file.

WARNING: NEVER put commands that run graphical displays (e.g. a web browser) in your **`.bashrc`** or **`.login`** file.

8.5 Setting shell variables in the .bashrc file

For example, to change the number of shell commands saved in the history list, you need to set the shell variable HISTSIZE. It is set to 1000 by default, but you can change this if you wish.

$ HISTSIZE=200

Check this has worked by typing

$ echo $HISTSIZE

However, this has only set the variable for the lifetime of the current shell. If you open a new Terminal window, it will only have the default history value set. To PERMANENTLY set the value of history, you will need to add the set command to the `.bashrc` file using a text editor.

A programming text editor called SciTE is installed in the virtual machine image. This program can be started from the command line, or using the graphic interface used to start the Terminal, but it is easiest to edit hidden files by starting from the command line.

$ **scite ~/.bashrc**

Modify the following line:

```
HISTSIZE=1000
```

to another value, save the file and force the shell to reread the `.bashrc` file by using the shell `source` command.

$ source .bashrc

Test to see if this has worked by typing

```
$ echo $HISTSIZE
```

8.6 Setting the path

When you type a command, your PATH variable defines in which directories the shell will look to find the command you typed. If the system returns a message saying "command: Command not found", this indicates that either the command doesn't exist at all on the system or it is simply not in your path.

For example, if the **sudo make install** command had not been used to install the STACKS package into the search path, you would either need to directly specify the path to the program you wanted to run (**~/stacks-1.48/ustacks**), or you would need to have the directory **~/stacks-1.48** in your path.

You could add it to the end of your existing path (the **$PATH** represents this) by issuing the command:

```
$  PATH=$PATH:~/stacks-1.48
```

To add this path permanently to your path for all future login sessions, you could add that line to your .bashrc AFTER the list of other commands. THIS IS NOT NECESSARY in our case because we used the **sudo make install** command after compiling the STACKS programs, to ensure that the programs are installed in the directories that are already listed in the path.

To find where a particular program is installed, you can use the **locate** command. This command relies on a database of files, so in order to ensure that the database is up-to-date, it is wise to always first execute the **updatedb** command, which must be run as root user. For example, try

$ **sudo updatedb**

$ **locate** ustacks

to see where the ustacks program has been installed.