An introduction to Linux for bioinformatics

Paul Stothard

January 17, 2016

Contents

1 Introduction

Linux is a free operating system for computers that is similar in many ways to proprietary Unix operating systems. The field of bioinformatics relies heavily on Linux-based computers and software. Although most bioinformatics programs can be compiled to run

on Mac OS X and Windows systems, it is often more convenient to install and use the software on a Linux system, as pre-compiled binaries are usually available, and much of the program documentation is often targeted to the Linux user. For most users, the simplest way to access a Linux system is by connecting from their primary Mac or Windows machine. This type of arrangement allows several users to run software on a single Linux system, which can be maintained by an experienced systems administrator. Although there are other ways for inexperienced users to become familiar with Linux (installing Linux on a PC, using a Live CD to run Linux, running a Linux virtual machine), this document focuses on accessing a remote Linux machine using a text-based terminal. Many powerful statistics and bioinformatics programs can be run in this manner.

2 Getting started

2.1 Obtaining a Linux user account

To gain access to a Linux-based machine you first need to speak a system administrator (sysadmin) to obtain a user name, hostname (or IP address), and password. Once you have this information you can access your account. Alternatively, you can run a Linux virtual machine on top of your present operating system. If you are using a Linux virtual machine you can skip ahead to the section called "Your home directory" after launching the machine and opening a Bash terminal).

2.2 How to access your account from Mac OS X

Mac OS X includes a Terminal application (located in the **Applications** >> **Utilities folder**), which can be used to connect to other systems (Figure 2.2). Launch Terminal and at the command prompt, enter **ssh user@hostname**, replacing "user" and "hostname" with the user name and machine name you have been assigned. Press enter and you should be prompted for a password. The first time you try to connect to your account, a warning message may appear. Ignore the message and allow the connection to be established.

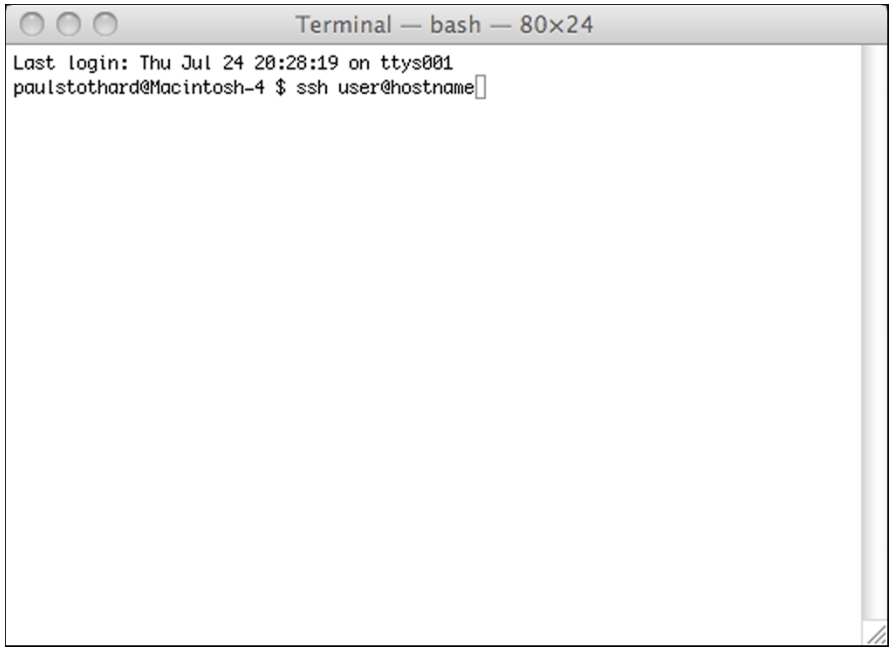

Figure 1: The Mac OS X Terminal application.

2.3 How to access your account from Windows

2.3.1 Using PuTTY

On Windows systems you can use a variety of programs to connect to a Linux system. PuTTY[1] is a free program already available on many Windows machines, including most of the student-accessible computers at the University of Alberta. If PuTTY is not installed you can download an executable from the PuTTY website. Launching PuTTY will open a configuration window resembling the one shown in Figure 2.3.1. Click **Session** in the left pane and then in the **Host Name (or IP address)** text box enter **user@hostname**, replacing "user" and "hostname" with the user name and machine name you have been assigned. Click **Open** to establish a connection with the remote system. The first time you try to connect to your account, a warning message may appear. Ignore the message and allow the connection to be established. A new terminal window will open, and you will be prompted for a password.

[1] http://www.chiark.greenend.org.uk/~sgtatham/putty/

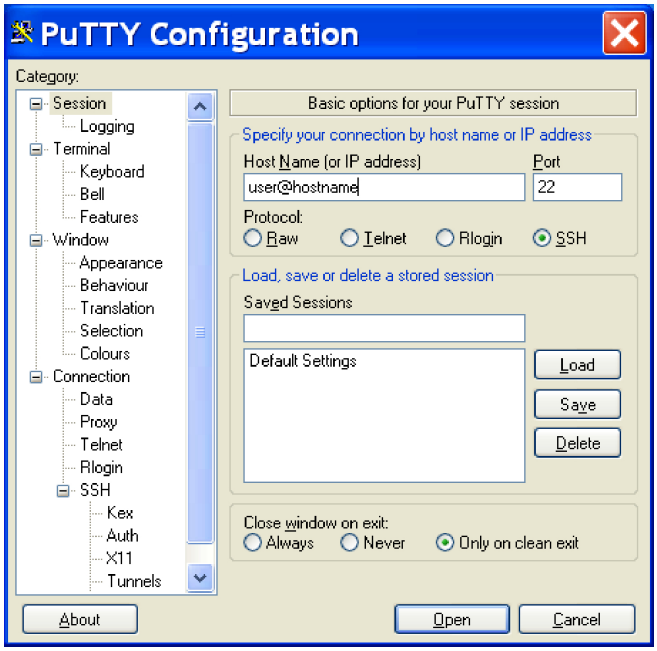

Figure 2: The PuTTY Telnet/SSH client running on Windows.

2.3.2 Using Cygwin

Cygwin[2] is a program that can be installed on Windows to provide a Linux-like environment. An advantage of using Cygwin is that you gain access to a lot of the standard Linux utilities, without having to connect to another computer. For example, after launching Cygwin, you can use many of the Linux commands described elsewhere in this guide, such as **pwd** and **mkdir**. You can also edit and run Bash scripts and Perl programs, and you can manage software repositories using subversion (these topics, apart from Bash scripts, are not covered in the tutorial). If you choose to install Cygwin, you can use it to access your remote Linux account by entering **ssh user@hostname** (replace "user" and "hostname" with the user name and machine name you have been assigned) on the command line (Figure 2.3.2).

[2] `http://www.cygwin.com/`

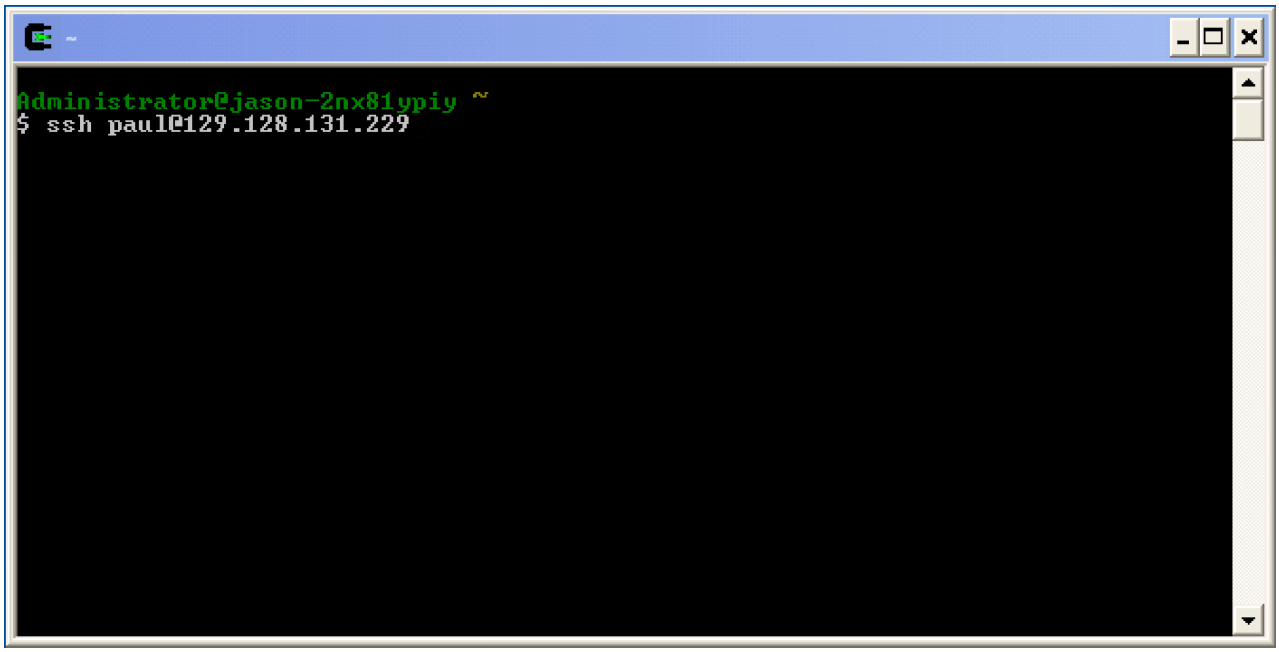

Figure 3: Cygwin, a Linux-like environment for Windows.

2.4 Your home directory

Once you have successfully signed in to your account, you can start exploring the directory structure of the remote computer. In your terminal you should see a command prompt, which usually consists of your user name and the name of the computer on which your account resides.

The examples in this guide will include **$** as the command prompt to illustrate that the commands should be entered into the terminal, and to differentiate the entered commands from the output they return.

When you sign in you will be located in your home directory. To see where this directory is located in the file system, use the **pwd** command:

```
1 pwd
```

In this case the output indicates that the current working directory is **paul** and that the **paul** directory is located inside of the **home** directory. When you enter **pwd** after signing in it should show that you are in a directory with the same name as your user name. The

home directory is located inside the **/** directory, which is also called the "root" directory. If at any time you want to return to your home directory, use the **cd ~** command. As you will see, your home directory is where you can create and delete your own files and directories.

2.5 Some basic commands

As in the previous example, you can see which directory is the current directory by using the **pwd** command:

```
pwd
```

To change to a different directory, use the **cd** command (**cd** means change directory):

```
cd /home
pwd
```

Now you should be in the **home** directory. To see what is inside of this directory, use the **ls** command (**ls** stands for list):

```
ls
```

The folders you see will likely differ from these. Now switch back to your home directory:

```
cd ~
```

In addition to real directory names, you can supply certain alias terms to the **cd** command. One of these is the **~** character, which represents your home directory. Another is **..**, which represents the directory above the current directory. Try the following:

```
cd ~
cd ..
ls
pwd
cd ~
pwd
```

As you can see, **cd**, **ls**, and **pwd** can be used to explore the Linux file system. Don't forget that you can always use **cd ~** to move back to your home directory.

2.6 More commands and command-line options

The **pwd**, **ls**, and **cd** commands point to programs on the Linux system that perform specific tasks and return output. When you enter the commands, the system runs the corresponding program for you. There are many other useful commands available.

To see which user you are signed in as, use the **whoami** command:

```
whoami
```

To see who else is signed in to the same system, use the **who** command:

```
who
```

To see the current time and date, using the **date** command:

```
date
```

To create your own directories use the **mkdir** (make directory) command:

```
cd ~
mkdir seqs
cd seqs
mkdir proteins
cd proteins
pwd
```

To create a new file, use the **touch** command:

```
cd ~/seqs/proteins
touch my_sequence.txt
ls -l
```

As you will see later, there are other ways to create new files (output redirection for example). In the last command above, the **-l** (a lowercase "L", not a "1") option was used with the **ls** command. The **-l** indicates that you want the directory contents shown in the "long listing" format. Most commands accept a variety of options. To see which options are available for a certain command, you can try typing **man** followed by the command name (**man ls** for example to see what options are available for the **ls** command), or the command name followed by **--help** (**ls --help** for example). One of these two methods usually provides information. To see what options can be used with **ls**, enter **man ls**. To get through the list of options that appears, keep pressing Space until the page stops scrolling, then enter "q" to return to the command prompt:

```
man ls
```

To delete a file, use the **rm** (remove) command:

```
cd ~/seqs/proteins
rm my_sequence.txt
ls
```

To remove a directory, use the **rmdir** (remove directory) command:

```
cd ~/seqs
rmdir proteins
ls
```

To copy a file, use the **cp** (copy) command:

```
cd ~/seqs/
touch testfile1
cp testfile1 testfile2
ls
```

To rename a file, or to move it to another directory, use the **mv** (move) command:

```
cd ~
touch testfile3
mv testfile3 junk
mkdir testdir
mv junk testdir
cd testdir
ls
```

The commands covered so far represent a small but useful subset of the many commands available on a typical Linux system [1].

3 Transferring files to and from your Linux account

3.1 Transferring files between Mac OS X and Linux

3.1.1 Using the Terminal application

Recall that Mac OS X includes a Terminal application (located in the **Applications** >> **Utilities** folder), which can be used to connect to other systems. This terminal can also be used to transfer files, thanks to the **scp** command.

Try transferring a file from your Mac to your Linux account using the Terminal application:

1. Launch the Terminal program.
2. Switch to your home directory on the Mac using the command **cd ~**.
3. Create a text file containing your home directory listing using **ls -l > myfiles.txt** (you will learn more about the meaning of **>** later).
4. Now use the **scp** command on your Mac to transfer the file you created to your Linux account. This command requires two values: the file you want to transfer and the destination. Be sure to replace "user" with your user name, and replace hostname with the real hostname or IP address of the Linux system you want to connect to:

```
scp myfiles.txt user@hostname:~/
```

You should be prompted for your user account password. Remember, in the above example you are running the **scp** command on your Mac, not from your Linux account.

Now, delete the **myfiles.txt** file on your Mac, and see if you can use **scp** to retrieve the file from your Linux account:

1. In the terminal on your Mac, switch to your home directory using **cd~**.
2. Delete the **myfiles.txt** file using **rm myfiles.txt**.
3. Use the **scp** command to copy **myfiles.txt** from your Linux account back to your Mac. Remember to replace "hostname" and "user" with the appropriate values when you enter the command:

```
scp user@hostname:~/myfiles.txt ./
```

The above command will prompt you for your Linux account password. Remember that **./** means "current directory". This informs the **scp** program that you would like the file **myfiles.txt** in your home directory on the remote system to be copied to the current directory on your computer.

3.1.2 Using Fugu

For users who prefer to use a graphical interface when transferring files between Mac and Linux, there is the freely available Fugu program.[3] To use Fugu, launch the program and enter the hostname of the computer you wish to connect to in the **Connect to** text area, and enter your Linux account name in the **Username** text area (Figure 3.1.2). Click **Connect** to connect to the remote system. You will be prompted for your Linux account password. Once you are connected to your Linux account you should be able to copy files between systems by dragging files and folders.

3.2 Transferring files between Windows and Linux

The simplest way to transfer files between Linux and Windows is to use the freely available WinSCP program.[4] WinSCP is already installed on many Windows systems, including those provided for student use at the University of Alberta. To use WinSCP, launch the program and enter the appropriate information into the **Host name**, **User name**, and **Password** text areas (Figure 3.2). Click **Login** to connect to the remote system. Once you are connected you should be able to transfer files and directories between systems using the simple graphical interface.

3.3 File transfer exercise

To test your ability to transfer files to your Linux account, download the following file to your Mac or Windows system using a web browser:

`http://www.ualberta.ca/~stothard/downloads/sample_sequences.zip`

[3] `http://rsug.itd.umich.edu/software/fugu/`

[4] `http://sourceforge.net/projects/winscp/`

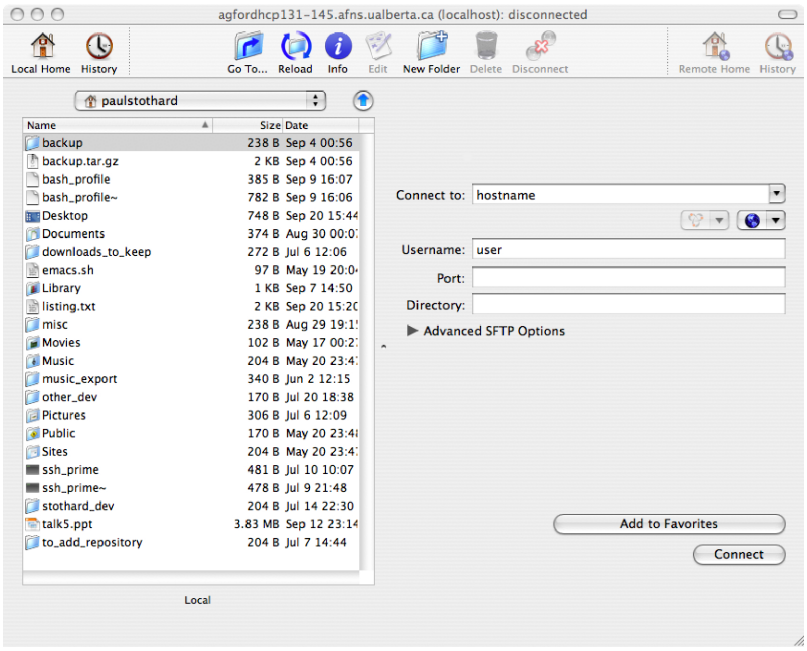

Figure 4: Fugu, a graphical SSH and SCP tool for Mac OS X.

Once the file has been downloaded to your system, use the file transfer methods outlined above to transfer the file to your Linux account. Be sure to perform this exercise, as the sample_sequences.zip file will be used later on in this tutorial.

4 Understanding Linux

4.1 Paths

Many commands require that you supply a directory or file name. For example, if you enter the command **touch** without specifying a file name, an error message is returned:

```
1 touch
```

Directory and file names like “testdir” and “my_sequences.txt” are called relative paths, since they specify the location of the file or directory in relation to the current working directory. For example, if you are located in your home directory, the command

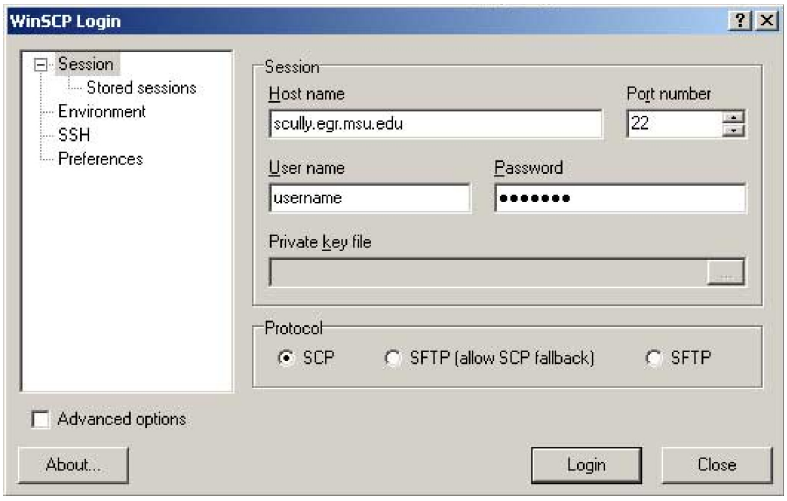

Figure 5: WinSCP.

mkdir some_dir will create a directory called "some_dir" in your home directory.

In the following example two directories are created, and **cd** is used to switch to "dir2" so that a new file can be created there using **touch**:

```
cd ~
mkdir dir1
cd dir1
mkdir dir2
cd dir2
touch somefile
```

Alternatively, by specifying the names of the directories in the paths, the same directory structure and file can be created without using **cd** to switch to the new directories (in this example the **rm -rf dir1** is used to remove the existing "dir1" and all of its contents):

```
cd ~
rm -rf dir1
mkdir dir1
mkdir dir1/dir2
touch dir1/dir2/somefile
```

Relative paths can use **..** to refer to the parent directory (i.e. the directory above the current directory). In the following example, **..** is used twice in the path passed to **touch**,

to create a file called "somefile2" in the directory two levels up from the current directory (which in this case is your home directory):

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

In contrast to relative paths, absolute paths specify the name of a file or directory in relation to the root (top) directory. Absolute paths always begin with a forward slash (the forward slash at the beginning of a path represents the root directory). In the following example, an absolute path is used to instruct **ls** to list the contents of the "etc" directory, which is used by Linux to store configuration files (i.e. the "etc" directory located in the root directory):

```
ls /etc
```

Absolute paths are useful because their interpretation doesn't depend on which directory is the current working directory.

4.2 Typing shortcuts

There are some useful tricks to save typing on the command line. One is to use Tab to complete a command name or a file name. When you press Tab, the system will try to complete the text you have partially entered, based on which characters are found in known command names and file names. If there are multiple possible matches, the system will not guess the matching text, however, if you press Tab twice a list of the possible matches will be given. This method of using Tab on the command line is called "Tab completion".

Try the following:

```
cd ~
mkdir a_new_directory
```

Now begin by typing "cd a" and press Tab instead of entering the full directory name. The full name should appear automatically.

To see what happens when there are multiple possibilities for a command or filename, begin by typing "mk" and press Tab twice. You should see a list of commands starting with the letters "mk".

Another useful command-line shortcut is to use the up and down arrow keys to scroll through commands you have recently used (**ls** is sometimes not stored in this list since it

is easy to type). If you scroll to a command you want to use again, press Enter to execute the command.

4.3 File permissions

Linux uses file permissions to prevent accidental file deletion and to protect data from being manipulated by others. Each file and directory is associated with three types of file permissions: "user", "group", and "other" permissions. The meanings of these terms are discussed below. To see the permission information for a directory or file, use the **ls** command with the **-l** option. Try the following set of commands:

```
cd ~
mkdir somedir
cd somedir
touch somefile
mkdir anotherdir
ls -l
```

The permission information for the directory **anotherdir** and the file **somefile** is given in the file listing. The first column is the file type and the file permissions (**drwxr-xr-x** for example). The third column is the owner of the file or directory (probably you), and the column after that is the group that owns the file. A group is simply a collection of users (groups are created by the sysadmin). A group can be used, for example, to allow different users to collaborate on a particular set of files, while protecting the files from editing by users not in the designated group.

As mentioned above, the first column contains the file type and permission information. The **anotherdir** entry begins with **d**, indicating it is a directory. The **somefile** entry begins with a **-**, which indicates that it is a file. The first three letters after the file type letter are the permissions for the user who owns the file or the directory. The next three letters are the permissions for the group that owns the file or directory. The final three letters define the access permissions for other users. The meanings of the letters are the following:

r (read permission) indicates that the file can be read. In the case of a directory this means that the contents of the directory can be listed.

w (write permission) indicates that the file can be modified. In the case of a directory this means that the contents of the directory can be changed (i.e. create new files, delete existing files, or rename files).

x (execute permission) indicates that the file can be executed as a program. In the case of a directory, the execute attribute means you have permission to enter a directory (i.e. make it the current working directory).

Returning to the example above, the permissions for **somefile** are **rw-r--r--**. This series of characters means that the owner of the file has the read and write permissions (**rw-**). Other users in the group users can read the file but not write or execute it (**r--**). Similarly, all other users can only read the file (**r--**).

To change file permissions, use the **chmod** command. For example, the following changes the permissions associated with **somefile** so that only the owner of the file (**paul** in this case) can read it (along with the **root** user, who will be discussed later):

```
chmod go-r somefile
ls -l
```

The **go-r** portion of the above **chmod** command means "from the group (**g**) and other (**o**) permission sets take away the read permission (**r**)."

To allow everyone to read the file **somefile** you could modify the permissions using the following:

```
chmod a+r somefile
```

The **a** in the above command means "all users". To refer to different types of users separately, use **u** (user who owns the file), **g** (group that owns the file), and **o** (other users).

To see how permissions protect files and directories, try to delete the **/etc** directory, which contains important system files:

```
rmdir /etc
```

Although the full rationale behind permissions may not be apparent to you at this time, it is important to remember that they do exist and that they control who can do what to specific files and directories. These permissions also automatically apply to any program you run on a Linux system. For example, if you run a program that attempts to copy a file for which you do not have the read permission, the program will be denied access to the file and it will not be able to make the copy.

4.4 Redirecting output

Many commands return textual output (i.e. the **ls** command) that is written to the terminal window. You can redirect the output to a file instead, by providing a filename preceded

by the '>' character. Use the following to create a file called **my_listing.txt** containing the output of the **ls -l** command:

```
cd ~
ls -l > my_listing.txt
```

To examine the contents of the **my_listing.txt** file, use the **more** command:

```
more my_listing.txt
```

Note that **more** is useful for viewing the contents of a file, one page at a time. To advance a page press Space. To return to the command prompt, enter "q".

Output redirection is useful for commands that return a lot of output. It is often used so that the output can be used or processed at a later time.

4.5 Piping output

With pipes, which are represented by the | character, it is possible to send the output of one program to another program as input. Consider the following command:

```
cat /etc/services | sort | tail -n 10
```

The above example uses the **cat** command to extract the text from the file **/etc/services**. The text is then piped to the **sort** program, which sorts the lines alphanumerically. Finally, the sorted text is piped to the **tail** program, which displays the last 10 lines of the text. Piping provides a convenient way to perform a series of data manipulations.

4.6 Using locate and find

Occasionally you may want to search a Linux system for a particular file. A simple way to do this is to use the **locate** command:

```
locate blastall
```

In this example the **locate** program was used to search for files or directories matching the name "blastall" (**blastall** is a program that can be used to search DNA and protein sequence databases). The **locate** program does not actually search the Linux file system. Instead it uses a database that is usually updated daily. By using an optimized database, **locate** is able to find items quickly, however you may not obtain results for files recently added to the system.

Another tool for searching for files of interest is **find**. This command accepts several options for specifying the types of files you want. For example, you can search for files based on name, size, owner, modification date, and permissions.

To find files in the **/etc** directory that end with ".conf" and that are more than 10 kilobytes in size you could use this command:

```
find /etc -name "*.conf" -size +10k
```

4.7 Working with tar and zip files

Sometimes you may want to compress a file or a group of files into a zip file or a tar file. Alternatively, you may have a tar or zip file you wish to extract. Note that tar files, which are similar to zip files, are frequently used on Linux-based systems.

To explore the **zip** and **tar** commands, first, create a text file using the following:

```
cd ~
wget www.google.ca -O google.html
```

The **wget** command can be used to download web-based files. In this example it is used to write the google homepage to a file called google.html.

To create a zip file of **google.html** use the **zip** command:

```
zip -r google.zip google.html
rm google.html
```

The **-r** (for recursive) is not necessary in the above example. However, it is needed if you want to if you want to zip the contents of a directory.

To extract this zip file use the **unzip** command:

```
unzip google.zip
```

To create a tar file use the **tar** command:

```
tar -cvf google.tar google.html
```

The **-c** option tells **tar** that you would like to create an archive (as opposed to extract one), and **-v** indicates that you want the **tar** program to be verbose (i.e. print comments and progress messages). **-f** is used to specify the name of the archive you would like to create (**google.tar**). This command is generally used on directory containing multiple files, rather than on a single file as in the previous example.

To extract this tar file use the **tar** command again, this time with the **-x** (extract) option:

```
tar -xvf google.tar
```

To create a tar file that is also compressed (like a zip file), use the **-z** option:

```
tar -cvzf google.tar.zip google.html
```

Note that a directory name can be specified instead of the name of a file. To extract tar.gz or tar.zip files use **tar** with the **-z** option:

```
tar -xvzf google.tar.zip
```

4.8 Wildcard characters

Sign in to your Linux account and locate the **sample_sequences.zip** file you transferred to your home directory (see the **File transfer exercise** section). Extract the file and then use **ls -l** to examine the files that are produced:

```
unzip sample_sequences.zip
ls -l
```

As you can see, several ".fasta" files were extracted from the **sample_sequences.zip** archive. Suppose you want to organize your home directory by placing these new sequence files into a single directory. You can do this easily using the * wildcard character. Try the following:

```
cd ~
mkdir sequences
mv *.fasta sequences
```

The * represents any text. The ***.fasta** instructs the **mv** command to move any file that ends with ".fasta" from the current directory to the **sequences** directory. The * can be used will other commands as a simple way to refer to multiple files with similar names.

4.9 The grep command

A useful command for searching the contents of files is **grep**. **grep** can be used to look for specific text in one or more files. In this example you will use **grep** to examine whether

the fasta files you downloaded contain a properly formatted title line. The title line should start with the **>** character, and there should not be any additional **>** characters in the file. Try the following command:

```
grep -r ">" sequences
```

The **-r** option stands for "recursive" and tells **grep** to examine all the files inside the specified directory. The '>' is the text you want to search for, and **sequences** is the directory you want to search. For each match encountered, **grep** returns the name of the file and the contents of the line containing the match. As you will see when you run the above command, each file contains a single title line as expected.

4.10 The root user

You may have noticed that when you are signed in to your user account you are unable to access many of the files and directories on the Linux system. One way to gain access to these files is to sign in as user **root**. However, you are unlikely to be given the password for the root user, since it is usually reserved for sysadmins, so that they install new programs, create new user accounts, etc. Even sysadmins do not usually sign in as the **root** user, since a small mistake when typing a command can have drastic consequences. Instead, they switch to the **root** user only when they need to perform a specific task that they are unable to perform as a regular user.

4.11 The Linux file system

So far you have worked inside your home directory, which is located in **/home**. You may wonder what the other directories found on the typical Linux system are used for. Here is a short description of what is typically stored in the directories:

/bin contains several useful programs that can be used by the **root** user and standard users. For example, the **ls** program is located in **/bin**.

/boot contains files used during startup.

/dev contains files that represent hardware components of the system. When data is written to these files it is redirected to the corresponding hardware device.

/etc contains system configuration files.

/home the user home directories.

/initrd information used for booting.

/lib software components used by many different programs.

/lost+found files saved during system failures are stored here.

/misc for miscellaneous purposes.

/mnt a directory that can be used to access external file systems, such as CD-ROMs and digital cameras.

/opt usually contains third-party software.

/proc a virtual file system containing information about system resources.

/root the root user's home directory.

/sbin essential programs used by the system and by the root user.

/tmp temporary space that can be used by the system and by users.

/usr programs, libraries, and documentation for all user-related programs.

/var contains log files and files created during processes such as printing and downloading.

To see which of these directories is present on the Linux system you are using, perform the following:

```
cd /
ls
```

4.12 Editing a text file using vi

Sometimes you may want to make changes to a text file while signed in to your Linux account. There are several programs available for this purpose, one of which is called **vi**. **vi** is somewhat difficult to operate, since you have to use keyboard shortcuts for all the commands you typically access using menus in other text editing applications. However,

by learning a few key commands you can comfortably edit text files using **vi**. In the example below, you will use **vi** to edit a text file containing multiple DNA sequences.

Many bioinformatics programs, such as **clustalw**, read in multiple sequences from a single file. Each sequence in the file usually needs to be in fasta format, as in the following example:

```
>seq 1
gatattta
>seq2
attatcc
>seq3
etc
```

To combine the p53 sequences in your **sequences** directory into a single file, use the following:

```
cd ~/sequences
cat *p53.fasta > all_p53_seqs.fasta
```

Now examine the **all_p53_seqs.fasta** file using the **more** command. In a fasta file containing multiple sequences, each sequence should have a separate title (titles normally begin with a **>** character). In the current **all_p53_seqs.fasta** file the first sequence record is missing the **>**. To edit this sequence's title, begin by opening the file in **vi**:

```
vi all_p53_seqs.fasta
```

Next, press **i** to enter insert mode. Use the arrows on the keyboard to move the cursor to top left if it isn't already there, and then type ">". Press Esc to leave the insert mode. To save the changes and quit, type **:wq** and press Enter. If you had problems editing the file and wish to quit without saving, press Esc and then type **:q!** and press Enter.

Use **vi** to correct the sequence title in the **bos_taurus_p53.fasta** file too, as we will be using this file in the future.

Note that the goal of this exercise was to introduce you to **vi**. Usually you will not need to edit your sequence files in this manner. However, you may find **vi** useful for making changes to your **.bashrc** file and for creating and modifying Bash scripts (both of these are described below).

5 Bioinformatics tools

5.1 EMBOSS

Now that you have been exposed to several of the built-in Linux commands and the Linux file system, you are ready to use some third party bioinformatics applications. One of these applications is called EMBOSS (The European Molecular Biology Open Software Suite).[5] EMBOSS contains several powerful bioinformatics programs for performing tasks such as sequence alignment, PCR primer design, and protein property prediction [2]. To see whether EMBOSS is installed on the Linux system you are using, try the following:

```
which showalign
```

showalign is one of the programs included with the EMBOSS package. In the above command, **which** is used to look for the **showalign** program on your PATH (the meaning of "PATH" is explained in more detail below). If this command returns something like "/usr/local/bin/showalign", then EMBOSS is likely installed. If instead it returns "no showalign in ..", then talk to your sysadmin.

EMBOSS includes numerous applications. In the following examples you will explore just a few of them. First, switch to your **sequences** directory, which should contain several sequences in fasta format.

```
cd ~/sequences
```

Now, use the EMBOSS **transeq** program to translate the *Bos taurus* p53 nucleotide sequence into a protein sequence (note that the **** below is used to split the command across multiple lines–when typing the command press Enter after the **** or omit the **** and type the entire command on one line):

```
transeq -sequence bos_taurus_p53.fasta -outseq bovine_p53_protein
```

To see the resulting protein sequence use:

```
cat bovine_p53_protein
```

Next, perform a global sequence alignment of two of the p53 sequences using **needle**. Note that when you run this command you will be prompted for some additional information. For this example you can press Enter each time you are prompted for information,

[5] http://emboss.sourceforge.net/

to indicate that you would like to use the default program settings:

```
needle macaca_mulatta_p53.fasta xenopus_laevis_p53.fasta -outfile \
    alignment
```

To examine the alignment that is generated use:

```
more alignment
```

Finally, use the **pepstats** program to obtain protein statistics for the protein sequence you created using **transeq**:

```
pepstats bovine_p53_protein -outfile stats
```

To examine the output use:

```
more stats
```

5.2 Using ClustalW

clustalw[6] is a powerful sequence alignment program that can be used to generate large multiple alignments [3]. To see whether **clustalw** is installed on the Linux system you are using, use the **which** command again:

```
which clustalw
```

This command should return the full path to the **clustalw** program. If it returns "no clustalw in ..", talk to your sysadmin.

The **clustalw** program offers several command-line options for controlling the sequence alignment process. To see these options, enter **clustalw -options**. In the following example **clustalw** is used to align the sequences in the **all_p53_seqs.fasta** file:

```
cd ~
clustalw -infile=sequences/all_p53_seqs.fasta -outfile=alignment -align
```

To view the completed alignment, use **more**:

```
more alignment
```

[6] http://www.ebi.ac.uk/Tools/clustalw2/index.html

5.3 Performing a BLAST search

BLAST[7] is a powerful program for comparing a sequence of interest to large databases of existing sequences [4]. By identifying related sequences you can gain insight into the function and evolution of the genes and proteins you are interested. The BLAST program can be installed on Windows, Mac, and Linux machines. However, to run BLAST on your own computer you also need to download the sequence databases you wish to search. These databases can be very large, and they become outdated quickly, since new sequences are continually added. For these reasons, many users prefer to submit sequences using the web interfaces provided by NCBI. The main drawback of using the web interface is that you can only submit one sequence at a time. If you have a large collection of sequences you wish to analyze, this approach can be very time consuming.

To avoid these issues you can use the **remote_blast_client.pl** program.[8] To download **remote_blast_client.pl** to your Linux account, use the following command:

```
wget http://www.ualberta.ca/~stothard/downloads/remote_blast_client.\
    zip --user-agent=IE
```

Now unzip the file you downloaded (don't forget about Tab completion–you can type "unzip re" and then press Tab to get the full file name):

```
unzip remote_blast_client.zip
```

Change the permissions on the **remote_blast_client.pl** file so that you can execute it:

```
chmod u+x remote_blast_client/remote_blast_client.pl
```

Now use the **remote_blast_client.pl** program to perform a BLAST search for each of the sequences in the **all_p53_seqs.fasta** file you created in your **sequences** directory:

```
cd ~
./remote_blast_client/remote_blast_client.pl -i \
    sequences/all_p53_seqs.fasta -o blast_results.txt -b blastn -d nr
```

The **-i** option in the previous command is used to specify which file contains the sequences you wish to submit and the **-o** is used to specify where you want the results saved. The **-b** and **-d** options are used to specify which BLAST program and database you want to use. The BLAST search may take a few minutes to complete. As the script

[7] `http://blast.ncbi.nlm.nih.gov/Blast.cgi`

[8] Note that NCBI provides a similar tool, called **netblast**, which is available at `ftp://ftp.ncbi.nih.gov/blast/executables/LATEST/`

runs it will give you information about what it is doing. If you wish to cancel the search, use Ctrl-C. Note that Ctrl-C can be used to return to the command prompt for many other programs too.

Once the program has stopped running you can examine the results using **more**. Note that this script returns results in a compact tabular format that does not include alignments.

To perform a BLAST search without relying on NCBI's servers you can use the **blastall** program. First you need to format a sequence database using the **formatdb** program. The following command formats the genomic sequence of E. coli (which was included in the **sample_sequences.zip** file) so that it can be used as a BLAST database:

```
cd ~
formatdb -i sequences/e_coli.fasta -p F
```

To see what the **-i** and **-p** options are used to indicate, try the following:

```
formatdb --help
```

You are now ready to search the E. coli database using any fasta or multi-fasta sequence as the query. The following command compares the two 16S rRNA sequences in **16S_rRNA.fasta** to the E. coli genome:

```
cd ~
blastall -i sequences/16S_rRNA.fasta -d sequences/e_coli.fasta -p \
    blastn -o local_blast_results.txt
```

To examine the results, use **more**:

```
more local_blast_results.txt
```

5.4 Performing a BLAT search

BLAT[9] is a powerful tool for searching for sequences of interest in a completed genome or proteome. It is faster than BLAST, and is much better at aligning cDNA sequences to genomic sequence, because it looks for splice site consensus sequences [5]. BLAT is less sensitive than BLAST, and is thus most useful for comparisons involving sequences from the same species (e.g. a human cDNA vs. the human genome) or closely related species (e.g. a human cDNA vs. the chimp genome).

[9] http://www.kentinformatics.com/

The following command uses the **blat** program to compare a bovine insulin cDNA to bovine chromosome 29:

```
cd ~
blat sequences/bos_taurus_chromosome_29.fasta \
    sequences/bos_taurus_insulin_cDNA.fasta blat_chr_29_output.txt
```

Note that **blat** interprets the first file to be the database, the second to be the query, and the third to be the output. The output returned by **blat** contains the coordinates of similar regions but not a sequence alignment. To generate a sequence alignment from the coordinates, use the **pslPretty** program, which is included with BLAT:

```
cd ~
pslPretty blat_chr_29_output.txt \
    sequences/bos_taurus_chromosome_29.fasta \
    sequences/bos_taurus_insulin_cDNA.fasta blat_chr_29_alignment.txt
```

To view the alignment, use **more**:

```
more blat_chr_29_alignment.txt
```

The **blat** program is usually used to compare sequences to a full genome rather than a single chromosome. The following commands download a complete bovine genome sequence:

```
cd ~
mkdir bovine_genome
cd bovine_genome
wget -c -A "Chr*" -R "ChrY*" ftp://ftp.cbcb.umd.edu/pub/data/assembly/\
    Bos_taurus/Bos_taurus_UMD_3.1/*
gunzip *.fa.gz
```

If you plan on performing several **blat** searches against a genome, you may want to convert the chromosome sequence text files to "2bit" files. The 2bit format is more compact and can lead to faster searches (the **faToTwoBit** program used below is included with **blat**):

```
faToTwoBit Chr1.fa Chr1.2bit
```

To convert all the chromosome text files to 2bit files you can use **find** followed by **xargs**. In the command below, **find** is used to obtain a list of the chromosome sequence files. The file list is passed to **xargs**, which builds a **faToTwoBit** command for each file,

replacing all instances of "{}" with the name of file:

```
cd ~
find bovine_genome -name "*.fa" | xargs -I{} faToTwoBit {} {}.2bit
```

Another way to process multiple files is to use the **-exec** option of **find**. The command placed after **-exec** is run for each file found by **find**. All instances of "{}" become the name of the file when the command is executed:

```
cd ~
find bovine_genome -name "*.fa" -exec faToTwoBit {} {}.2bit \;
```

The **-exec** approach is generally preferred because it works when filenames contain spaces. The resulting 2bit files (one per chromosome) can now be used as the target sequences in **blat** searches. The following uses **find**, **xargs**, and **blat** to compare each bovine chromosome to a single query sequence (**bos_taurus_insulin_cDNA.fasta**):

```
cd ~
find bovine_genome -name "*.2bit" | xargs -I{} blat {} \
    sequences/bos_taurus_insulin_cDNA.fasta {}.insulin.out
```

To quickly examine all the output files produced by **blat** you can use the following:

```
cat bovine_genome/*insulin.out | more
```

To perform searches for multiple queries, first create a text file containing a list of the query sequence files, one filename per line. Pass this list file to **blat** in place of the query. This approach is faster than manually running a separate search for each query, in part because each chromosome sequence needs to be loaded into memory just once. The following creates a file called **query_list.txt** containing the names of all the files in the **sequences** directory that have "bos_taurus" in their title and are less than 1 MB in size. The list file is then used as the query in **blat** searches against each bovine chromosome:

```
cd ~
find ./sequences -name "*bos_taurus*" -size -1000k > query_list.txt
find bovine_genome -name "*.2bit" | xargs -I{} blat {} query_list.txt \
    {}.list.out
```

To convert the **blat** results to alignments, first create a file containing a list of the chromosome sequence files that were searched. This "targets" file can then be passed to **pslPretty** along with the **query_list.txt** file you created earlier. These filename lists are used by **pslPretty** when it obtains the sequences located between the match coordinates

given in the “list.out” files:

```
cd ~
find bovine_genome -name "*.fa" > target_list.txt
find bovine_genome -name "*list.out" | xargs -I{} pslPretty {} \
    target_list.txt query_list.txt {}.pretty
```

The resulting alignments can be viewed using the following:

```
cat bovine_genome/*pretty | more
```

6 Streamlining data analysis

6.1 The .bashrc file

When you sign in to your Linux account, a file in your home directory called **.bashrc** is run by the system. This file contains commands that are used to control the behavior of the **bash** program (**bash** is the program that passes the commands you type to the actual programs that do the work). You may not have noticed this file in your home directory, because by default the **ls** command does not show files that start with a “.” character. To see all the files in your home directory, use **ls** with the **-a** option.

```
cd ~
ls -a
```

In the following exercise you will make a few minor changes to your **.bashrc** file using using **vi**. Start by copying your **.bashrc** file so that you can go back to the existing version if the changes you make create problems:

```
cp .bashrc bashrc_backup
```

Now open your **.bashrc** file in **vi**:

```
vi .bashrc
```

Remember to press **i** to enter insert mode. Add the following text below the existing contents:

```
alias rm="rm -i"
alias la="ls -al"
```

Now press Esc to leave insert mode, and then type **:wq** to save your changes and exit **vi**. The first line you added to your **.bashrc** file will tell **bash** (the program that handles the commands you type) to always pass the **-i** option to the **rm** program when you enter the **rm** command. The **-i** option tells **rm** that you want to be warned before any files are actually deleted, and that you want to have the option of canceling the delete process. The second line tells **bash** that you want to use the command **la** to call the **ls** program with the **-a** and **-l** options (show all files and use the long listing format). Defining the **la** command in your **.bashrc** simply saves you the trouble of remembering and typing the full command for listing all files.

Try the new **la** command:

```
la
```

Notice that the **bash** program is saying that it doesn't know what is meant by **la**, even though you defined it in the **.bashrc** file. Remember that the **.bashrc** file is only read when you log in. To tell **bash** to read your **.bashrc** file again, use the **source** command:

```
source .bashrc
```

The **la** command should now be recognized by **bash**.

6.2 Modifying $PATH and other environment variables

Try running the **remote_blast_client.pl** program from your home directory by typing the name of the program:

```
remote_blast_client.pl
```

The **bash** program, which interprets the commands you enter, doesn't know anything about the **remote_blast_client.pl** program. This is the reason you had to enter the exact location of the **remote_blast_client.pl** script when you ran it in the previous example (**./remote_blast_client/remote_blast_client.pl**). Remember that the **./** means the current directory.

Whenever you type a command, **bash** searches for a program with the same name as the command you enter, and for alias commands you specified in your **.bashrc** file. The **bash** program does not however, search the entire file system for a matching program, as this would be very time consuming. Instead, it searches a specified set of directories. The names of these directories are stored in an environment variable called **$PATH**. To see what is currently stored in your path, use the following:

```
echo $PATH
```

Notice that the directory containing **remote_blast_client.pl** is not stored in the **$PATH** variable. You can temporarily add it using the following

```
export PATH=$PATH:~/remote_blast_client
```

To see that it was added, enter **echo $PATH** again. Note that this change to **$PATH** only lasts while you are signed in. To make the change permanent, you could add the above **export** command to the end of your **.bashrc** file using **vi**.

Now that the **$PATH** variable contains information about where to find **remote_blast_client.pl**, try entering the following in your home directory:

```
remote_blast_client.pl
```

The **remote_blast_client.pl** program should start. Press Ctrl-C to return to the command line.

Although the benefits of editing the **$PATH** variable are minor in this case (it isn't difficult to enter the full path to the **remote_blast_client.pl**), understanding environment variables and how to modify them is very important. Indeed, many programs require that you add new environment variables to your **.bashrc** file.

6.3 Writing a simple Bash script

Sometimes you may find it hard to remember what command-line options a program like **remote_blast_client.pl** or **clustalw** requires. Furthermore, you may always use the same options, making all the typing seem quite repetitive. Suppose you want to be able to sign in to your user account and quickly perform an alignment of whatever sequences you have stored in a file called **dna.fasta**. You can do this by writing a simple Bash script that contains the command you want to use.

First create the file that will contain your script:

```
touch align_dna.sh
```

Now edit the file in **vi**:

```
vi align_dna.sh
```

Using **vi**, add the following text (remember to press i to enter insert mode):

```
#!/bin/bash
clustalw -infile=dna.fasta -outfile=dna.alignment -align -type=dna
```

Save your changes and exit **vi**.

Use **chmod** to make your script executable:

```
chmod u+x align_dna.sh
```

To test your script, first create a file called **dna.fasta** in your home directory by copying the fasta file you created previously in your **sequences** directory:

```
cd ~
cp sequences/all_p53_seqs.fasta ./dna.fasta
```

Now execute your script:

```
./align_dna.sh
```

You should see output from **clustalw** appear, and a file called **dna.alignment** should be created. Bash scripts are useful because they help to automate analysis steps, since you do not need to enter a lot of text, and you can be sure the same parameters are used each time. It is possible to build complex scripts consisting of many commands.

7 Summary

This tutorial has provided a brief introduction to the Linux operating system, and in particular the use of the command line. Although the transition to the command line can seem difficult at first, it is well worth the effort if you plan on working with large data sets, such as those arising from high-throughput sequencing projects. If you would like to become even more proficient at analyzing data you should learn a programming language. Linux is well-suited to such an endeavor, because of the wealth of programming tools available. Once you can program you can perform almost any analysis you can imagine.

References

[1] http://ss64.com/bash/

[2] Rice P, Longden I, Bleasby A (2000) EMBOSS: The European Molecular Biology Open Software Suite (2000) Trends Genet 16:276-277.

[3] Chenna R, Sugawara H, Koike T, Lopez R, Gibson TJ, Higgins DG, Thompson JD (2003) Multiple sequence alignment with the Clustal series of programs. Nucleic Acids Res 31:3497-500.

[4] Altschul SF, Madden TL, Schäffer AA, Zhang J, Zhang Z, Miller W, Lipman DJ (1997) Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. Nucleic Acids Res 25:3389-3402.

[5] Kent WJ (2002) BLAT–the BLAST-like alignment tool. Genome Res 12:656-664.