



Chapter A01. Using Linux

Alan Mitchell Durham, PhD¹ and Marco Dimas Gubitoso, PhD²

Created: May 1, 2006.

Linux started in 1991 as an experiment on the processor 80386, from Intel. The idea was to make a UNIX-like system run efficiently on a microcomputer. UNIX was by that time a very popular system — and still is — for large machines and for scientific/academic research.

Its author, Linus Torvalds, posted a message on *Usenet* — a former kind of world wide forum — announcing he built the central part of a UNIX-like operating system and was making it available to anyone who would like to play with it. It was an adaptation of *Minix*, an educational and limited version of UNIX. The development since then was very quick, and as soon they were able to run programs, they could benefit from the large set of tools provided by the *GNU project*¹.

The GNU project started a few years earlier by Richard Stallman of MIT, when he decided to develop a full UNIX clone under the concepts of *free software*. The idea of 'free' software is that one can copy, distribute and modify it as needed, without the restrictions of royalties or copyright violations, as a consequence, the programs are available for free — one can charge for copying the program but not for the program itself.² This is enforced by the "GNU General Public License", or GPL for short.

GNU was not complete, but had all the essential programs, except the kernel. Many applications like text editors, compilers and windowing systems were also available. The union of these programs with Linus' kernel gave birth to a full featured UNIX-like operating system: *Linux*.

The system is still evolving and everyday new programs are added, the vast majority under GPL. In particular, most of today' free bioinformatics software run in the Linux system.

Distributions

Operating systems involve a large amount of programs, along with data, libraries and configuration files, which demand a well defined structure and organization. Each operating system has its own way of organizing the files, and for some of them, there may be more than one alternative.

This complexity makes it very difficult to make an installation without specific policies and helper programs. To solve this problem, the Linux community came up with the concept of distribution, which is essentially a set of

Author Affiliations: 1 Department of Computer Science; University of São Paulo; Brazil; Email: Alan.brazil@gmail.com. 2 Department of Computer Science; University of São Paulo, Brazil; Email: gubi@ime.usp.br.

1 GNU stands for GNU is Not UNIX

2 To preserve the rights of the author, Stallman came up with an innovative type of copyright: the *copyleft*.

programs wrapped in *packages*, tested to work together, so there is a guarantee that no conflicts will arise and it is possible to establish a simple installation and upgrading procedure.

There are many Linux distributions nowadays, but most of them are based on one of the following three:

Slackware - It is a more basic distribution, almost a "raw" one, directed to those users who like to experiment and test new features.

Red Hat - Intended to be easy to install, with ready solutions for most end users tasks. Has a set of configuration tools easy to use, at the expense of some flexibility. There are many derived distributions, like S.U.S.E and Mandriva

Debian - Combines flexibility with ease of use and is one of the more popular nowadays. As with Red Hat, there are some derived distributions, Knoppix and Ubuntu being the most famous. The companion CD for this book is based on the Knoppix distribution.

The rest of this chapter is organized as follows. We first give an introduction to the Linux system and its basic concepts. Then we have a tutorial where you will learn about Linux with a hands-on approach. The tutorial is followed by a "Frequently Asked Questions" (FAQ) section, that you can use afterwards as a reference on how to perform different tasks in Linux. At the end there is a series of short exercises for practice.

1.1. UNIX and Linux Environments

1.1.1. Using Windows[®] to Understand Linux

Most books on operating systems introduce the basic concepts of the systems from scratch. In this chapter we take a different approach. Since most people have used computers before, and since the different versions of MS Windows[®] are installed in over 90 percent of the desktops nowadays, we will explain the main concepts of Linux trying to associate, as much as possible, with the Windows[®] counterparts.

Linux, like MS Windows[®], is an operating system, that is, a software that controls the machine, running programs, managing devices like CD-ROM readers, hard disks, monitors, etc. However, Linux is a version of UNIX, a system that was conceived as a multi-user system since the beginning. Also, in the UNIX world, computers have been linked to networks since the 1970's. This means that these systems have had the possibility of multiple users and of the use of internet as part of their design since the early days.

There are important characteristics of Linux that we should emphasize:

- **Multiuser x personal systems:** Until the advent of Windows[®] NT, Microsoft systems were designed for *personal* computers. That means that, even if more than one person was using a computer, the system assumed that there was no need for protecting the data in the computer, or to prevent the user from doing any particular task. Linux, like Windows[®] NT, Windows[®] 2000 and Windows[®] XP, is a multiuser system. This means it is assumed that more than one person will use the system. Having more than one user means that the system needs to control the access to the computer and protect the data of one user from the other. To provide this separation a Linux system provides user *accounts*, and each account has a *password*. This used to be a novel concept for most users but, with the advent of Internet mail, most people have an e-mail account in some Internet site. The concept is exactly the same.

The administrator: In multi-user systems, many people will be using the computer, therefore important tasks involving the basic functioning of the system need to be reserved only to a special user, the *system administrator* (or *root*). The administrator is generally the only user that can install or remove software, create new user accounts and modify the configuration of the computer (like, for example, adding new

disks). Administration of a Linux system is not a trivial task and requires some expertise, mainly if you are managing a Linux server. For machines used as personal workstations, the administration tasks can be reduced to a minimum, and can generally be performed by someone with less expertise.

- **Protection:** Protection of data and programs in Linux follows a very simple, yet very flexible, protocol. For each file, program, disk volume or device, there is a set of three permissions, associated with three different user categories.

User categories:

- **owner** (u) - usually the user that created the file (the owner, however, can be changed)
- **group** (g) - a file also belong to a group. User groups are created by the administrator and can have any number of users. A user can also participate in any number of groups.
- **other** (o) - any user that is not the owner and does not belong to the file's group.

Permissions

- **read** (r) - indicates that a file's content can be copied or read.
- **write** (w) - indicates that a file's contents can be modified or erased.
- **execute** (x) - indicates that a file can be run as a program. If set for a directory, indicates that the user can "enter" the directory.

Therefore, for each file or directory we have three categories of users: the owner, the users that belong to the file's (or directory's) group, and the rest of the world. Also, for each category there are three permissions: reading, writing and executing. This means each file has 9 different protection fields associated to it.

The flexibility of the Linux's protection scheme can be very useful. Let's consider some examples:

- A laboratory has three different research groups. Work has to be confidential among groups, but researchers in the same group can share information freely. The head researcher of the laboratory has full access to all information. In this situation we can have three user groups in Linux, each group containing the users associated with the respective research group. The account of the head researcher is included in all groups. All files have full access (reading, writing, executing) for the owner and for the group, but no access for the rest of the world.
- A researcher has a project that is confidential. He or she is the only one that should be allowed to modify the files of the project, or to execute the programs being developed. However, the head researcher of the lab should be able to access the files for inspection. In this case we need one user group with two accounts: the researcher's account and the lab head's account. All files in the project will have the researcher as the owner. The owner will have full access (read, write, execute), but the group will have only read access. The rest of the world will have no access to the file.
- A laboratory obtains a special license to run some software. However, the software should be run only in the computers of that particular laboratory. No user should be allowed to inspect, copy or modify the code. The file that contains the program should have the administrator as the user (or the account of the person responsible for the software). A group should be created for all users of the laboratory. The owner will have full access, the group will have only execution permission, and the rest of the world will have no permissions set.
- You decide to create a public folder to use for file exchange. Anyone can read files in the folder or copy files from it. In this case, you set the folder to have all permissions.
- To avoid making mistakes, if you have a file, you want to be sure it is not going to be deleted. One possible example is your income tax file, which you want to keep untouched after you finish it. In this case you can create a file that not even the owner has write permission. This means the file cannot be deleted either. To modify or remove the file you will need to change its permissions first.

The table below shows a summary of the examples above.

File Name	Owner		Group		Others		
GroupProject1	R	W	R	W			X
ConfidentialProject	R	W	X	R			
SpecialSoftware	R	W	X			X	
PublicFolder	R	W	R	W	R	W	
IncomeTax	R						

The UNIX command to set permissions will be explained in the practical section of this chapter.

Data exchange

Normal file exchange in the MS Windows[®] world involves having access to an Internet site for downloads and having especially designed Internet pages for uploading information (like, for example, sending a file to be attached to an e-mail message in a web mail site). However, maintaining a web service is not an easy task and involves many security risks.

By contrast, any two Linux computers in the Internet can exchange information safely, independently of the fact that they maintain a web service or not. There are a series of programs that allow users to download and upload information across the Internet safely, provided that they have a regular user account and password in the remote computer (fig. 1.1).

Sharing disks

A common problem in laboratories with more than one computer is the fact that information generated when someone uses a specific machine is kept locally. MS Windows[®] allows you to share information across computers using the "network neighborhood" icon. Good network practice would involve requiring users to only store information in a single computer, a server. Having all information in a server means the users can access it from any other machine plugged on the net. In the Window's world, the administrators of the server can make folders available to users. A user can then "map" the remote folder in his local account. All information stored in the mapped folder is actually being stored remotely. This type of sharing involves some user knowledge (to map the remote folder) and discipline to always use the folder that is mapped to the server.

With Linux, remote directories can be mapped transparently into local folders of a computer. Actually, the whole user area can be mapped remotely. The user does not need to perform any tasks. Once logged in, all information stored by the user in his directories will actually be sent to the server. It is the system administrator's task to perform this mapping and this can be done for all users of the network. The sharing architecture can be changed without any need to inform the users. So, for example, a new server can be added to the network and some users' directories moved to the new servers without the need of notifying anyone else (fig. 1.2).

Sharing processors

In the Windows[®] world some programs, when installed in the server, can be run remotely by users in some other computer on the local network. The user will only click on the program icon in the shared folder and the remote CPU will do the processing. However this feature has to be part of the program's code. Some programs will run remotely, others will still run locally. Most software in MS Windows[®] will not run remotely because they were originally conceived to only run locally.

In the Linux world, anyone can run some software in a remote computer, provided he/she has an account name and a password in the remote computer. Actually, logging in a remote computer is a standard procedure for Linux users. It is up to the user to decide which the best computer to run the software is (provided, of course, that the software is installed in that computer and that the user has an account there). There is no restriction

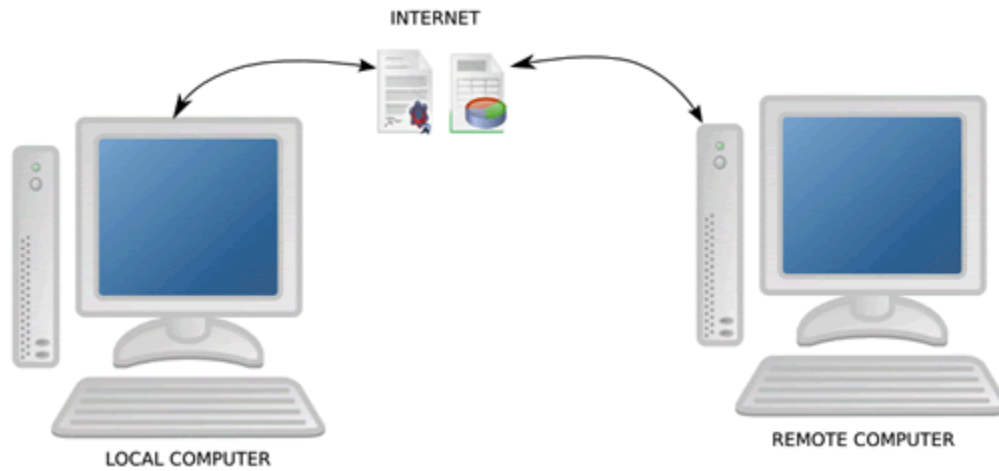


Figure 1.1. Data exchange. Knowing the address of the remote computer, you can send files or download files

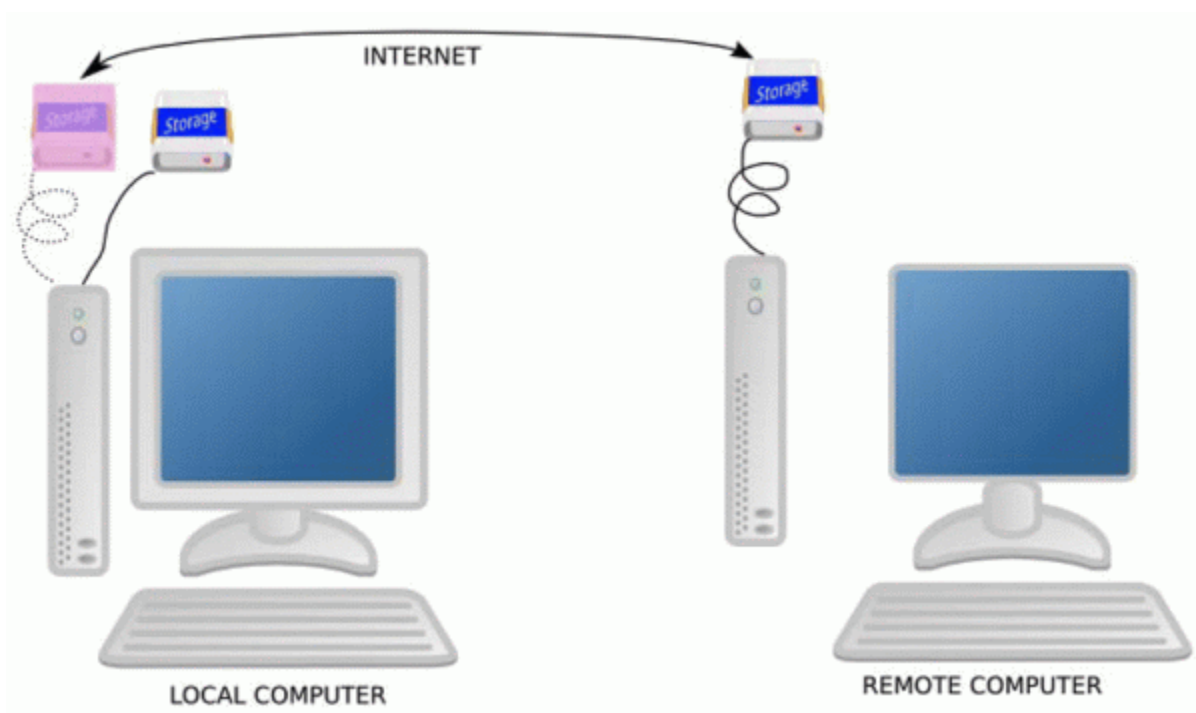


Figure 1.2. Sharing disks. The files that seem to be in a local disk are in fact in another computer - the user is unaware of this file sharing.

imposed on the location of the computer. You can run software in your computer in Sri Lanka while visiting a research lab in Romania, with no need for special Linux configurations or installation. This is called in the Linux world a "remote login". The local computer serves only as a terminal, the local disk is not used, only the keyboard, mouse, and screen (fig. 1.3).

1.1.2 The graphical interfaces of Linux

After the advent of windowing systems, first popularized by the Apple Macintosh[®] computers in the 1980's, all other computer systems developed similar graphical user interfaces. In the beginning the differences were significant, but most systems seem now to have converged to very similar layouts. Linux, differently from MS Windows[®] and Apple Macintosh[®], does not have only one layout, but in fact many different options of

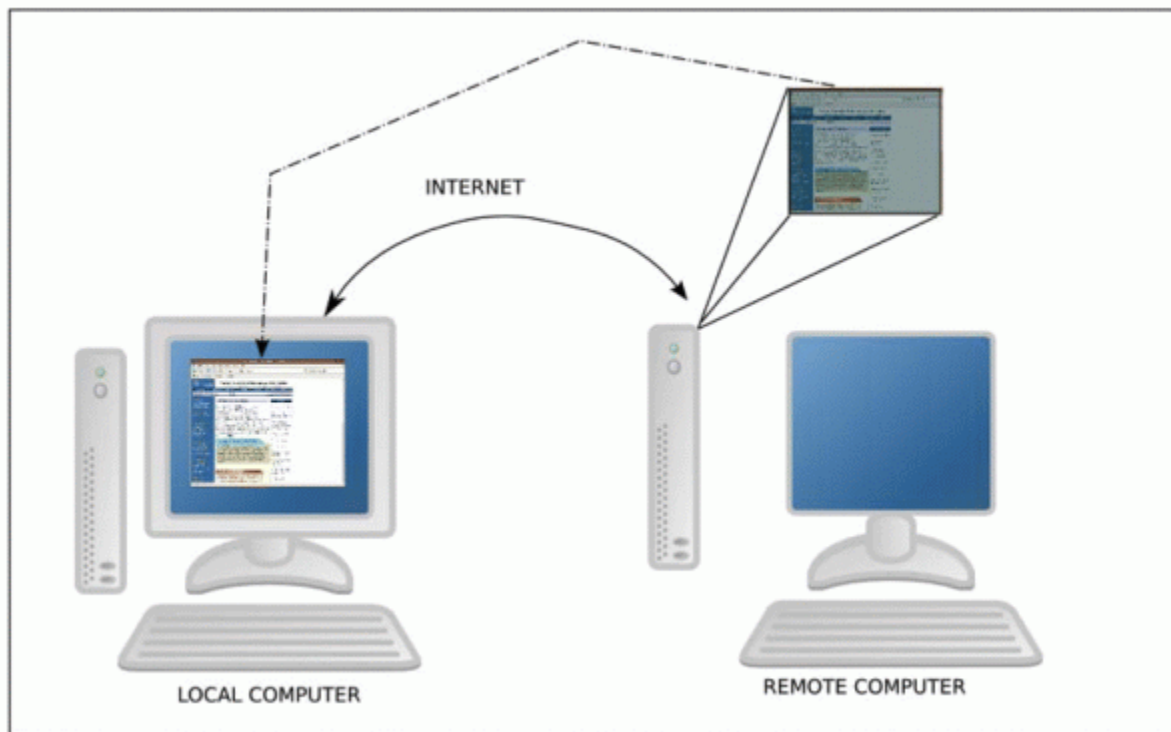


Figure 1.3. Users can log into other computers. The program will run on a remote computer, the screen is displayed in the local computer, while the data stay in the remote machine

graphical interfaces. This means that the system that manages the graphical interface is separate from the operating system. The user can choose from a series of *window managers*, the one that best fulfills his/her needs. Another advantage is that Linux users can upgrade their window managers to newer versions without having to install a new version of Linux.

In this section we will introduce the *K Desktop Environment* (KDE), a very popular windows manager in Linux that is probably the one most similar to MS Windows[®]. In this section we will use Windows[®] as a base reference for describing KDE. However, we expect Macintosh[®] users to be also able to follow the description as well.

The KDE Windows[®] manager

KDE's taskbar and main menus. KDE's graphical interface is probably the Linux's most user-friendly one. It is very similar in appearance to the MS Windows[®] interface with some characteristics of MacOS. Figure 1.4 shows a screenshot of KDE interface just after logging in. In the bottom of the screen we have the "taskbar", similar in spirit to the one implemented on MS Windows[®], but with some particularities. The task bar is in the bottom of the screen, the "start" pop-up menu is activated by clicking the icon with letter "K".

We encourage the reader to experiment the other buttons. Moving the mouse over the various buttons will cause a description to pop up.

The number and names of programs available at the menu can vary immensely, depending on what is available in your version of Linux.

Windows[®] software counterparts Most programs available to Microsoft users have a similar counterpart in Linux. We will list below some of the most popular applications for Windows[®] and one counterpart in the Linux world.

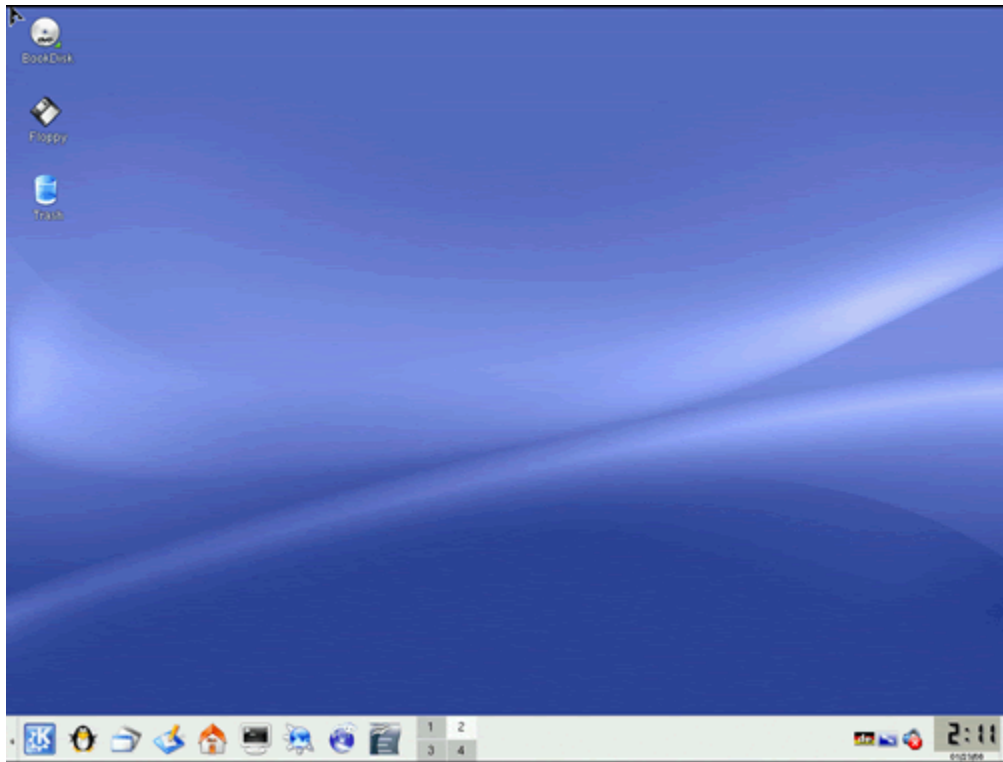


Figure 1.4. KDE's graphical interface

Windows[®] Explorer - A file browser similar to the Windows[®] Explorer is Konqueror. With Konqueror, users can browse directories by clicking in the folder icons and by using the navigation buttons on the top part of Konqueror's windows. Images, sound files, text files and movies are automatically opened using appropriate applications. There is, however, one important difference; Konqueror can also be used as an Internet browser. When the application is initially launched, a help page is displayed in the browser. Internet browsing is started by typing a valid web address in the *location* text field, and file browsing is activated by clicking on the *home* icon, which opens up the user home directory in the local disk (equivalent to the My Documents folder of Windows[®] XP). A Google search in the Internet can be initiated directly by clicking the keywords in the search text box on the top right corner. Two interesting features of this browser are the ability of opening many "tabs", each one with a different display (e.g. an internet address or directory). This facilitates copying files from one directory to another without the need of opening two windows.

Internet Explorer Even though Konqueror can be used to browse the Internet, there is another application that looks more similar to Microsoft's Internet Explorer: Mozilla Firefox. Firefox is the successor of Netscape, an Internet browser that for many years was Internet Explorer's main competitor. Like Konqueror, Firefox has a specific text box in the upper right corner that can be used to directly initiate a web search. Differently from Konqueror, this search can be performed not only by Google, but also by other search engines such as Yahoo, Amazon and Ebay (try clicking in the icon depicting a magnifying lens). Users can even add other search engines to the menu. There is also a version of Firefox for MS Windows[®].

Outlook Express E-mail can be managed in Linux using Mozilla Thunderbird. As with the Internet browser, the interface of the e-mail client Thunderbird is similar enough to that of Outlook Express to make any lengthy descriptions unnecessary.

A MS Windows[®] version of Thunderbird is also available.



Figure 1.5. Konqueror

Microsoft Office Microsoft Office's users can perform similar tasks in Linux using two distinct Linux packages: OpenOffice and KDE Office. Programs of both packages can open and save files in Microsoft-compatible formats (.doc, .ppt, .xls).

- **OpenOffice.org** This system, developed in the Java language by Sun Microsystems can be also installed in MS Windows[®] systems. The main programs available are (more details can be found at <http://www.openoffice.org>):
 - Write - a word processor, similar to MS Word
 - Calc - a spreadsheet similar to MS Excel
 - Impress - a presentation manager similar to MS PowerPoint
 - Base - a database manager similar to MS Access

KDE Office - This system is part of the KDE window manager package. There are also Windows[®] versions of all programs. The main programs available are (more details can be found at <http://www.koffice.org>):

- KWrite - a word processor very similar to MS Word,
- KSpread - a spreadsheet similar to MS Excel
- KPresenter - a presentation manager similar to MS PowerPoint
- KExi - a database manager similar to MS Access

Notepad Notepad is a text editor. This is different from MS Word, which is a text processor, that is, a program to create formatted written documents, not just plain text. Many people in the Windows[®] platform use MS Word to edit both formatted documents and unformatted text. Text editors are much faster and lightweight than word

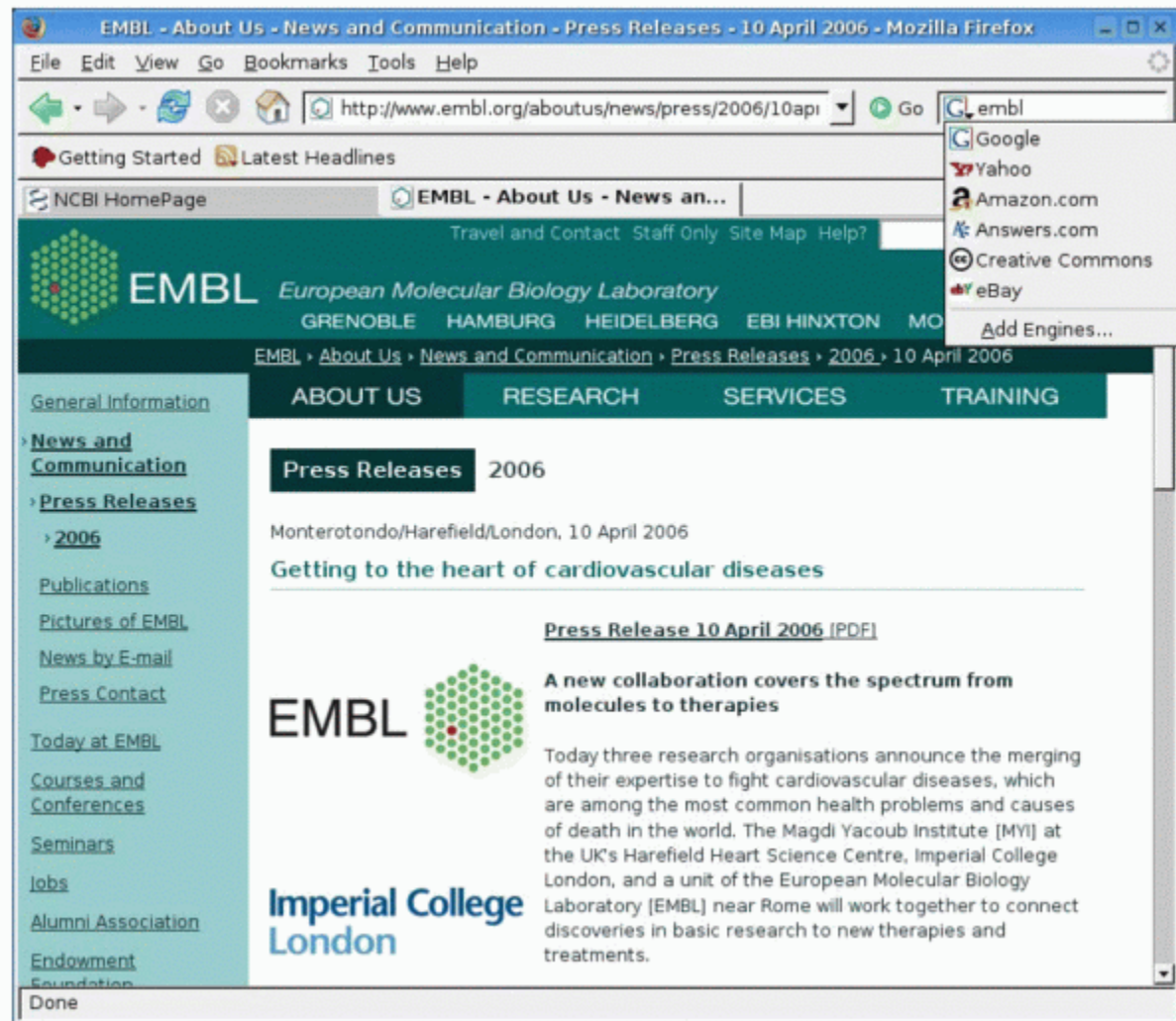


Figure 1.6. Firefox

processors. In addition, if your file is not a .doc document, it is just easier to use Notepad directly. The main problem with notepad is the absolute lack of nice editing features. The Linux operating system is proficuous in text editors. Probably the most flexible text editing system is Emacs (or Xemacs, a slight different version). Emacs provides all the basic text editing functions but is also a configurable editor and can be used to read e-mail, run programs, compile computer programs. A much simpler alternative to Emacs is Kate, very similar in spirit to Notepad.

Other Graphical User Interfaces (GUIs)

One of the nice characteristics of Linux is the extent to which a user can customize his installation. Different from other operating systems for personal machines, Linux has many windows managers. We have presented KDE, but there are other options. The most popular alternative to KDE is Gnome. Gnome has a cleaner interface but still has the same working principles, with a main taskbar and pull up menus. Other options of window managers do not include a taskbar, just pop-up menus prompted by the mouse buttons. These include WindowMaker, IceVM, and others.

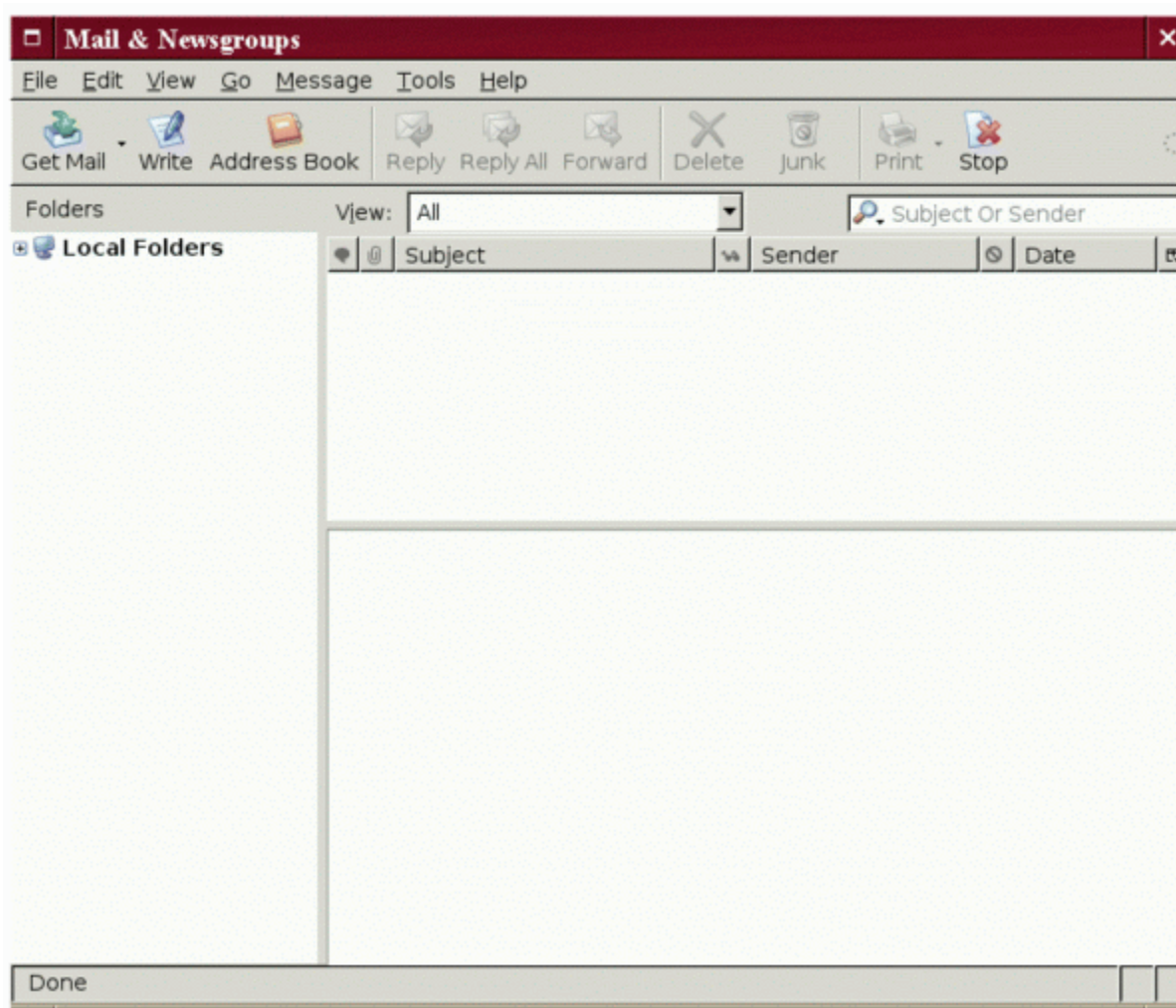


Figure 1.7. Thunderbird

1.1.3 Command Line

Windows systems, or graphical user interfaces (GUI) are very convenient to perform some tasks with just a few clicks of the mouse. Nevertheless, there are many situations where a GUI is not the best choice.

For instance, if you have to run the same program on a large set of files, which are in different directories, using just the GUI is very cumbersome and error prone. You would have to move the mouse all the time and find the right icons for the files you want, one at a time. Some specific programs have been developed to simplify these monotonous tasks, but this is not the general rule, and they do not cover all the possible cases.

Another situation is when the programs you are running take a long time to run, and you need to check the output or success of the execution before proceeding. You would need then to check the screen all the time to see if anything went wrong and take the appropriate decision.

Basically, although a GUI is very good for interactive processing, it is hard to automate tasks, especially if they involve repetitive tasks.

The alternative is using a *shell*, a text interface consisting basically of a virtual terminal where you type the tasks to be done using a set of commands and some special features, presented later in this chapter. Writing the

commands gives you the ability to specify the tasks with much more detail, and to perform job control — like repetition and error checking, since the interface has a language specially crafted for this purpose.

Even though you have to type the commands, you do not lose the interactivity. In fact, you can launch graphic applications and collect their results without any problem. Mastering just the basics of command line interaction gives you a noticeable increase of productivity for the following reasons:

- The possibility of writing complex commands, easily coupling the execution of many applications.
- The simplicity to customize execution. It is more complicated in a GUI to state all possible execution patterns, and, if you want to change the execution pattern or the execution parameters, you will have to open a 'preferences' window and re-select a series of many buttons. Using command line, you just need to type the correct options directly.
- You can save the command lines you used in a text file and run them again later. You can even include conditions that change the execution behavior when needed.
- It is very easy to create "combo" applications by connecting existing ones without the need of further programming.

The graphic user interface provides an abstraction at a higher level. As a consequence, it is possible to find solutions to standard situations much faster, but does not allow detailed control of the application. When a finer control is needed, in many occasions it is not possible to stay at an abstract level.

1.2. The Shell and Command Line Interaction

1.2.1. Some initial concepts

Shells: bash

To interact with the computer, you need a program which can understand your requests and translate them into actions to be executed by the machine. On most current machines this is done by a graphical interface with menus and icons: if you want to run a program, just click or double-click on the corresponding icon. In the same fashion, if you want to delete a file, just drag the corresponding icon to a trash can.

The alternative is a *text* interface called a *shell*, where the commands are typed and the results are presented in text form as well. In fact, this was the standard method of interaction with computers before the graphic interface became possible that, due to its strength, is the main method on UNIX systems.

In essence, a shell is a program runner which allows the user to control the execution, direct results, combine programs and even develop small applications in a very quick way. In this section we will present some concepts needed to use the Linux shell.

A single directory tree: no device names In the MS Windows[®] system each physical device (CD-ROMs, hard disks, USB drives, etc.) is identified by a letter of the alphabet. It is common, for example, to have the main hard disk under "C:", the CD-ROM drive under "D:" and so on. The complete address of a file looks similar to

```
C:\Documents and Settings\John\My Documents\Research\thesis.doc
```

Linux uses a different approach. There are no device letters, so in fact the user is unaware of the physical location of his/her files. All directories and files are under a single root directory. The structure of the directory hierarchy is controlled by the administrator. Thus, the complete address of any file starts at the root directory, as follows in the example below:

```
/home/employee1/Research/thesis.doc
```

This has many advantages. In particular, users do not need to know of any changes in the systems' devices. For example, if a project turns out to use lots of disk space, we can buy a new hard disk and assign to this new disk the home directory of the project, copying all the old information into the new disk without the users ever knowing what happened. Their information will be at the same address, even if it may be now being stored in a different hard disk.

The PATH environment variable: finding programs As in MS Windows[®], programs can be installed in any directory in Linux. In the Linux bash shell, you can tell the system which directories can contain programs. When the user types a command, the system automatically searches in the directories specified by the user. The places where the shell looks for programs are specified in the variable called PATH (a variable is a place where the system stores information). In all Linux installations some directories are already part of the PATH. These are generally in /usr/bin and /usr/local/bin. Almost all standard Linux applications are located in these two directories. However, any user can add new directories to his/her path and run other programs without having to search for them. You can even download a new version of a program in the system and run your version by default, as opposed using the one available to the other users. To do this, you only have to include the directory where you stored the program in the PATH system variable **before** the system directories.

Auto completion Another nice feature of the bash shell is auto-completion. It is normally very tiresome to type complete file addresses or even program names. In the bash shell, when the user starts typing and presses the <TAB> key, the system tries to complete the typing. If it is a program name (first word to be typed), bash searches the directories in the path to find which programs start with the letters typed. Otherwise bash looks in the partial path specified by the user to search for completions. If there is only one possible completion, pressing the "tab" key (we will refer to it as <TAB>) will cause the appropriate word to appear, if there is more than one completion, a sound will be emitted. Pressing <TAB> the second time will display all possible completions.

Job control: When using command line, it is important to understand the concept of "process". A process is a running program. There can be many processes running the same program (for example, when you open more than one Windows[®] Explorer window). A process can be in two states: *running* and *suspended*. When a process is *suspended*, no work is performed nor lost. A running process just executes the program tasks normally. Processes can be *killed*, that is, the user can determine that the processing will be interrupted and end in a certain moment (this is equivalent to click in the X icon of a window).

History The shell maintains a list of all the previous commands that the user has typed. This list is called the "history". The history can be useful to reissue long commands or to confirm which programs were executed and in which order.

Data flow: Linux programs, by default, get input from the keyboard and send output data to the screen. The keyboard is the *Standard Input*, or STDIN, and the screen is the *Standard Output*, or STDOUT. All basic shell commands use this principle. A nice feature of command line interactions is that you can have the output of a program in STDOUT be fed directly to the STDIN of another program, creating a UNIX *pipe*. For example, suppose you have two programs: one that reads sequences from a multi-FASTA file and outputs only the sequence names, the other that reads lines and order them alphabetically. Both programs are useful on their own, but if you connect the output of the first program (selecting only the sequence names) to the input of the second (sorting alphabetically), you can read a multi-FASTA file and have as a result the sequence names, ordered. These two programs exist in Linux and this task can be performed, as we will see later. It is important to note that, for pipes to work correctly, the second program needs to be able to read the output of the first program, that is, the format of one program's output has to be compatible with the other program's input.

We have seen some concepts that are important to keep in mind when we start learning about the Linux commands. The next section will present a set of important Linux commands in a learn-by-doing style.

1.2.2 Learning by doing

In this section we will explain the main Linux commands and actions through a tutorial. At each step a task will be proposed and you will be taught how to perform that task. We strongly recommend that you read this chapter while performing the actions in the companion Linux system. This system will run in almost any PC and you do not need to modify your computer to do it. Also, the system will have all the files and the environment needed to follow the tutorial. To use the companion CD please follow the instructions below:

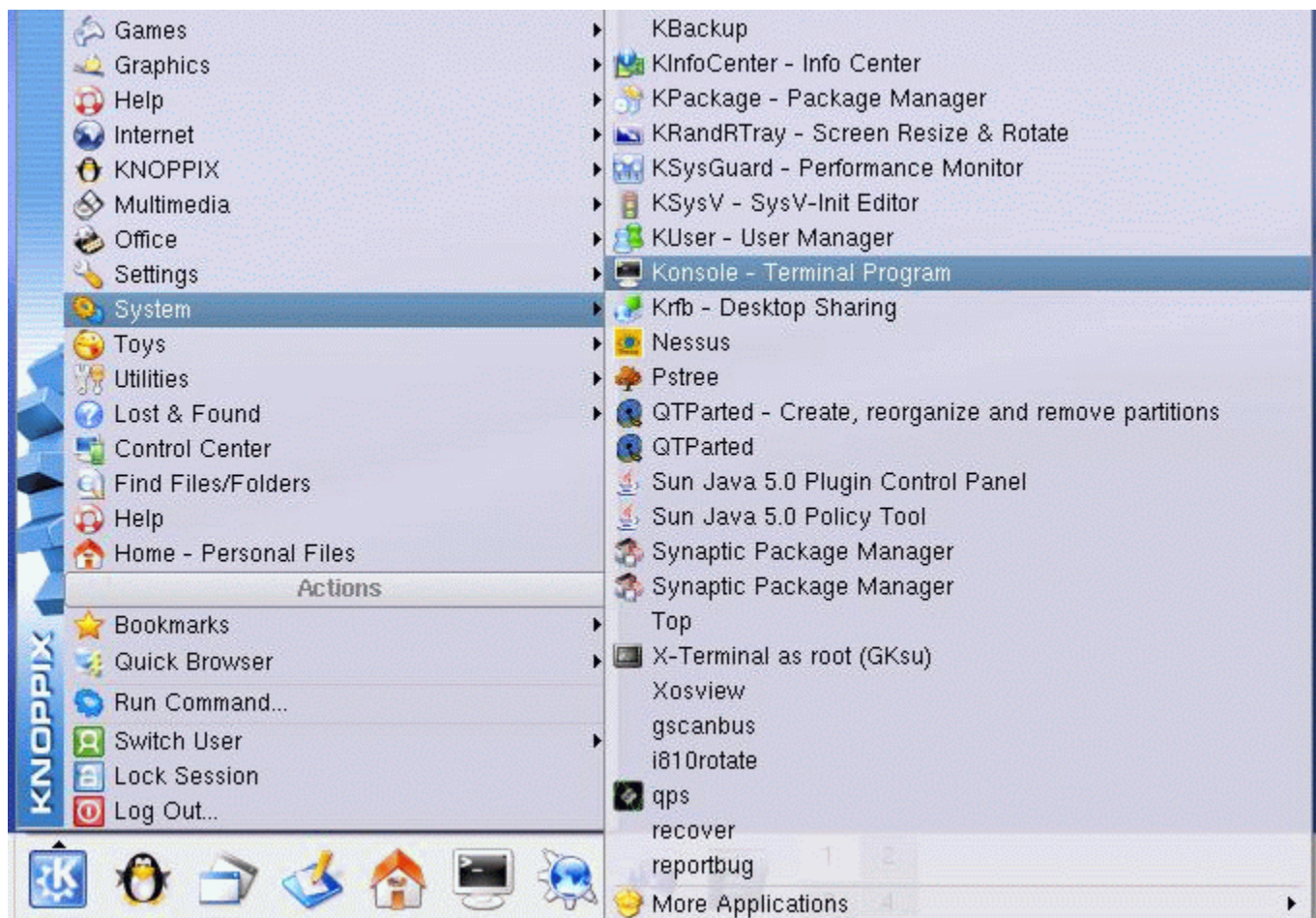
1. Download the CD ISO image from one of the two sites below (wichever is accessible):
<http://clinmaldub.usp.br/bioinfobook/bookDVD.iso>
<http://www.coccidia.icb.usp.br/bioinfobook/bookDVD.iso>
2. Follow the instructions described at one of the two sites below (wichever is accessible):
<http://clinmaldub.usp.br/bioinfobook/instructions.html> :
<http://www.coccidia.icb.usp.br/bioinfobook/instructions.html>
3. After following all the instructions you should have your computer running our version of Knoppix, ready for the tutorial.

Logging in

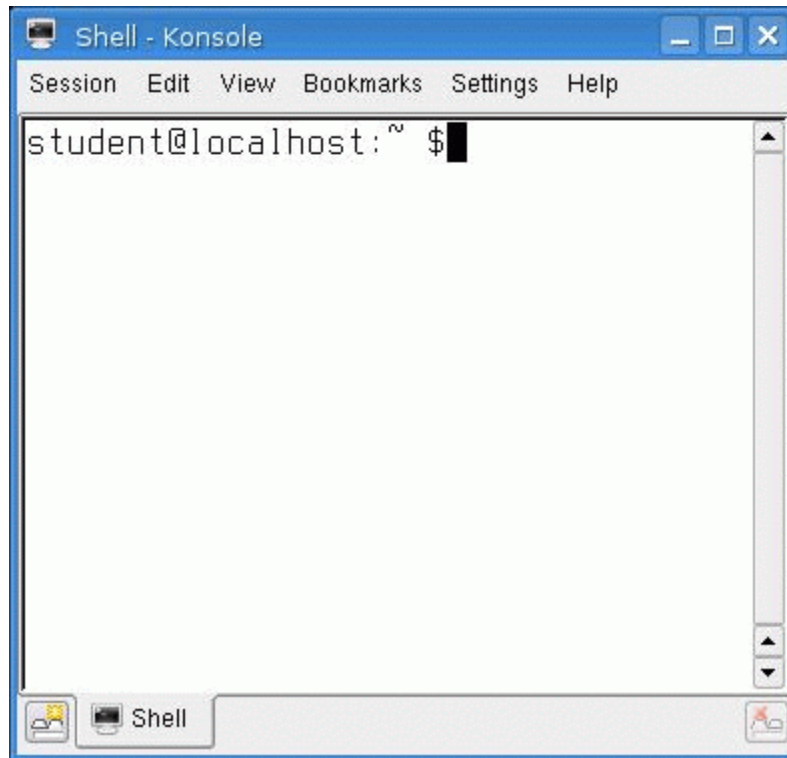
Since Linux is a multiuser system, access is controlled. Before you start doing any work, you have first to log in. For this purpose, you should type your username and password in the presentation screen (see the figure below). For this tutorial, you should log in as the user *student*, with the password *student*. After logging in, the initial screen for the KDE environment will appear



Launching a shell In this tutorial, we will be working almost entirely from the command line. To use the shell, you need to start a terminal window. Click on the terminal icon in the bottom bar. Alternatively, click on the icon with a K letter (similar to the MS Windows® Start button) on the left bottom corner of your screen. This will open up a menu. Select 'System' and then 'Konsole - Terminal Program' (see figure below illustrating the second option):



A window will pop up. This window is almost entirely blank, with only one line being displayed. What you see is the *prompt*. The prompt indicates that the shell is ready to receive a command. The prompt also displays some useful information. You should have the following prompt: `student@localhost:~:$` The first part of the prompt is the username ("student"), followed by "@" and the machine name ("localhost"), ":", the current directory (in our case, "~", the home directory), and finally "\$". You can see the shell window below (note colors in the shell may vary):



Creating a work environment (a simple directory structure)

When you first log in, you are at your *home directory*. Each user has his/her own home directory. This directory can be located anywhere in the system, but in most Linux systems it is under the */home* directory. In our case, our home directory is */home/student*. If you use Konqueror to inspect your home directory (K menu, Internet submenu, Konqueror entry; type the "home" icon.) you will see the directory is initially empty. We could create a directory structure directly from Konqueror, but we will use the shell command line instead.

For our tutorial we want the following subdirectories:

- recentFiles
- results
- internetSite

Task: Create three subdirectories: *recentFiles*, *results* and *internetSite*.

Comments: To create directories in the command line you use the command *mkdir* (from MaKe DIRectory). The only thing you need to specify is the new directory's name. The command can be used to create one or more directories.

Issuing the command: We will create the new directories using *mkdir* twice, once to create a single directory, and once to create the remaining two. You should type the following two lines in the shell

```
mkdir recentFiles
mkdir results internetSite.
```

If you typed the command correctly, now you should have three subdirectories in the */home/student* directory.

IMPORTANT NOTE: In most Microsoft Windows[®] systems, the computer does not discriminate between upper and lower case letters, but Linux and UNIX do. So, throughout this tutorial, be careful not to change uppercase letters for lowercase letters and vice-versa, as the commands will not work.

Paths and the current directory

Similarly to when you are in Windows[®] Explorer, in the shell, at any given time, you are inside some directory. When the shell is initially created, you are at your *home directory*, in our case `/home/student`. This is called your *current directory*. You can refer to any file in your current directory by using only the file's name. To refer to any other file that is not located in your current directory, you should give a *path*, that is, directions on how to navigate either from the current directory or from the root directory to the desired file. Paths that start in the current directory are called *relative paths*, and the ones that start in the root directory are called *absolute paths*. Absolute paths always start with `/`, otherwise you have a relative path. You can change the current directory using the `cd` command followed by a path to a directory. Examples of the command:

- `cd /home/instructor` - absolute path
- `cd Desktop` - relative path
- `cd localdirectory/anotherdirectory` - relative path

There are some important nicknames for specific directories that you should remember (we list them inside quotes below):

- `"."` - denotes the current directory
- `"~"` - denotes your home directory
- `"~username"` - denotes the home directory of user *username*.
- `".."` - denotes one directory up, that is the directory on which your current directory is.

Please remember these nicknames, as they will be used below.

Creating a local copy of an Internet site

Task: Copy the hypertext (HTML) version of the tutorial and FAQ of this chapter into your local machine's *Tutorials* directory.

Comments: A very useful Linux program is `wget`. With this program you can mirror a whole Internet site in your computer with just one single command. `wget` works also if the site requires a password. To use `wget` you need to know the address of the main page you want to copy. If the page is password-protected, you also need a valid username and password. Using `wget`, you can either copy only the page, all the links, all the links in the links, and so forth. You can also copy only the pages that refer to the links in the same Internet server (so you can, for example, avoid copying advertising links from a commercial page).

Issuing the command: First we need to go to the *internetSite* directory.

```
cd internetSite
```

Now we should copy the pages of an internet site. The main internet page for this tutorial is located at

http://www.coccidia.icb.usp.br/linux_tutorial/index.html.

To copy the site completely we will call `wget` with the "mirror" option (`-m`) and with the `-nH` option³. Therefore, to copy the site completely you need to write

³ If `wget` is invoked without this option, it will create the directory path `www/bionfo/org/br/linux_tutorial/index.pl`. within the *Tutorials* directory.


```
wget -nH -m http://www.coccidia.icb.usp.br/linux\_tutorial/index.html
```

If you typed the command correctly, not only the file *index.html* will be copied into your directory, but also all other files related to the links of this file. This means that all the pages addressed in the links, as well as the pages these pages point to will be downloaded. If you use your local browser to open the file *index.html* directory, you will now be able to find your local copy of the tutorial page. Try the links and you will see that all the respective pages have now link referring to local addresses.

Important - When you type a very long command line, some parts of this command may be typed incorrectly, in which case your command will not work. If you make a mistake, it will probably fall in one of the categories below:

- You typed wrongly one of the options. In this case, a long text displaying all the correct options of the command will appear.
- You typed a wrong internet address (the first part of the http address: "<http://www.coccidia.icb.usp.br>") – in this case the error message should have the text

Resolving *theActualAdrrssYouTyped* ... failed: Name or service not known.

- You typed the second part of the address wrongly (after the first single "/", that is, "Linux_tutorial/index.html"). In this case the error message should include the text:

HTTP request sent, awaiting response... 404 Not Found:

Sites with passwords: Many sites in the internet require passwords to show some part of their material. Wget can retrieve pages protected by passwords. If you know a valid user and its password you can use the options – *http-user=* and *-http-passwd=*. The book site has a part that is password protected. Try the command

```
wget -nH -m http://coccidia.icb.usp.br/linux\_tutorial/protected/protected\_page.html
```

You will get back an error message stating that you need a password to access the page:

HTTP request sent, awaiting response... 401 Authorization Required

Authorization failed.

To retrieve the information you can use the user *bookreader* with the password *bookworm*. Let's try (**warning:** the command written below should be typed in a single line, we use two for lack of space):

```
wget -nH -m --http-user=bookreader --http-password=bookworm http://coccidia.icb.usp.br/linux\_tutorial/protected/protected\_page.html
```

Now, if you typed correctly, you should have retrieved the *Linux_tutorial* directory in your home directory. Inside this directory you will find the file *protected_page.html*.

Important: this is an even longer command line than the previous one. Apart from all the errors previously described, another possible error can occur if you typed either the user or the password wrongly. In this case the error will be the same as in the previous one:

HTTP request sent, awaiting response... 401 Authorization Required

Authorization failed.

Finding files in a local directory

Task: Search at the directory /home/tutorial/sequences and its subdirectories, looking for all files with the extension ".fasta"

Comments: Searching for files with specific names or parts of the name is a very common task. Very frequently users download some files from the internet and then realize later that they are not located where they were supposed to be. Command-line UNIX has a file search program similar to the "search" facility of Windows[®] Explorer. Konqueror also have a similar facility. However, using the command line can be much faster than opening a browser and navigating through a set of menus. To perform this task, the UNIX shell provides the program find. In this program, the user specifies the name of the file to be searched and the starting location point of the search (a directory). As a result, find will show the path of all the files matching the request. You can either specify the complete file name or use wildcards to specify only parts of the name.

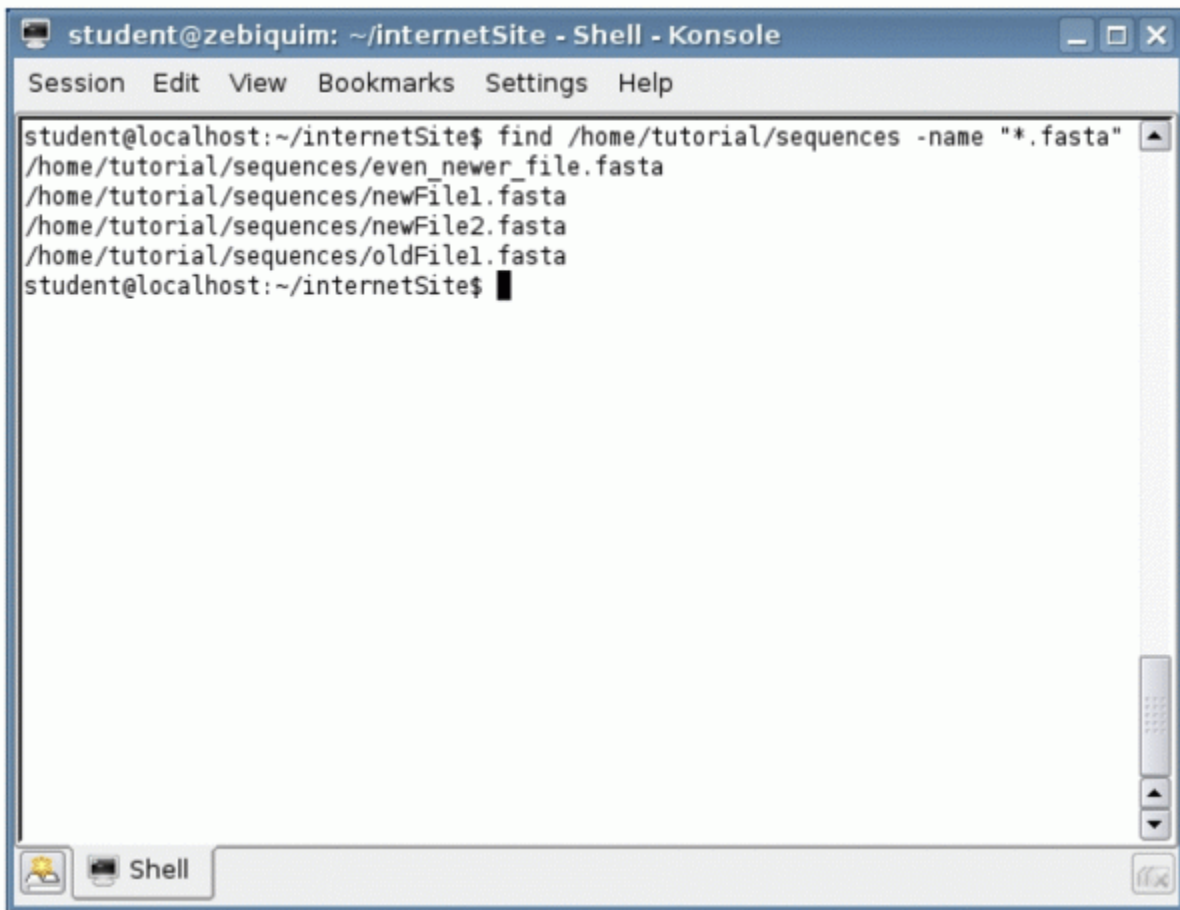
Issuing the command: To find a file, we use the find program. This program needs two parameters, the directory on which to start the search, and the name of the file to be searched. You can use wildcards to describe filenames, where these wildcards may include the following characters:

- * — This wildcard designates any character or series of characters, or even no character at all. In our case, writing *.fasta indicates names that start with any characters whatsoever, but necessarily end with ".fasta".
- ? — The question mark will designate any single character. Writing "?asta" will match "fasta", "pasta", ".asta", and so on.
- [list of characters] — This expression works just like the '?' wildcard, but instead of indicating *any* character, it will match one of the characters in a list. A range may be indicated if you use a '-'. For instance, "[fp]asta[0-9]" will designate "fasta0, fasta1, fasta2, ..., fasta9, pasta0, ..., pasta9".

Wildcards can be used anywhere in the specification of a name to be searched. To perform the task described in this item you should type the command

```
find /home/tutorial/sequences -name "*.fasta"
```

The result should be the listing:

A screenshot of a Linux terminal window titled "student@zebiquim: ~/internetSite - Shell - Konsole". The window has a menu bar with "Session", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal shows the following text:

```
student@localhost:~/internetSite$ find /home/tutorial/sequences -name "*.fasta"
/home/tutorial/sequences/even_newer_file.fasta
/home/tutorial/sequences/newFile1.fasta
/home/tutorial/sequences/newFile2.fasta
/home/tutorial/sequences/oldFile1.fasta
student@localhost:~/internetSite$
```

The terminal window has a scrollbar on the right and a taskbar at the bottom with a "Shell" icon.

[Checking file information, moving files]

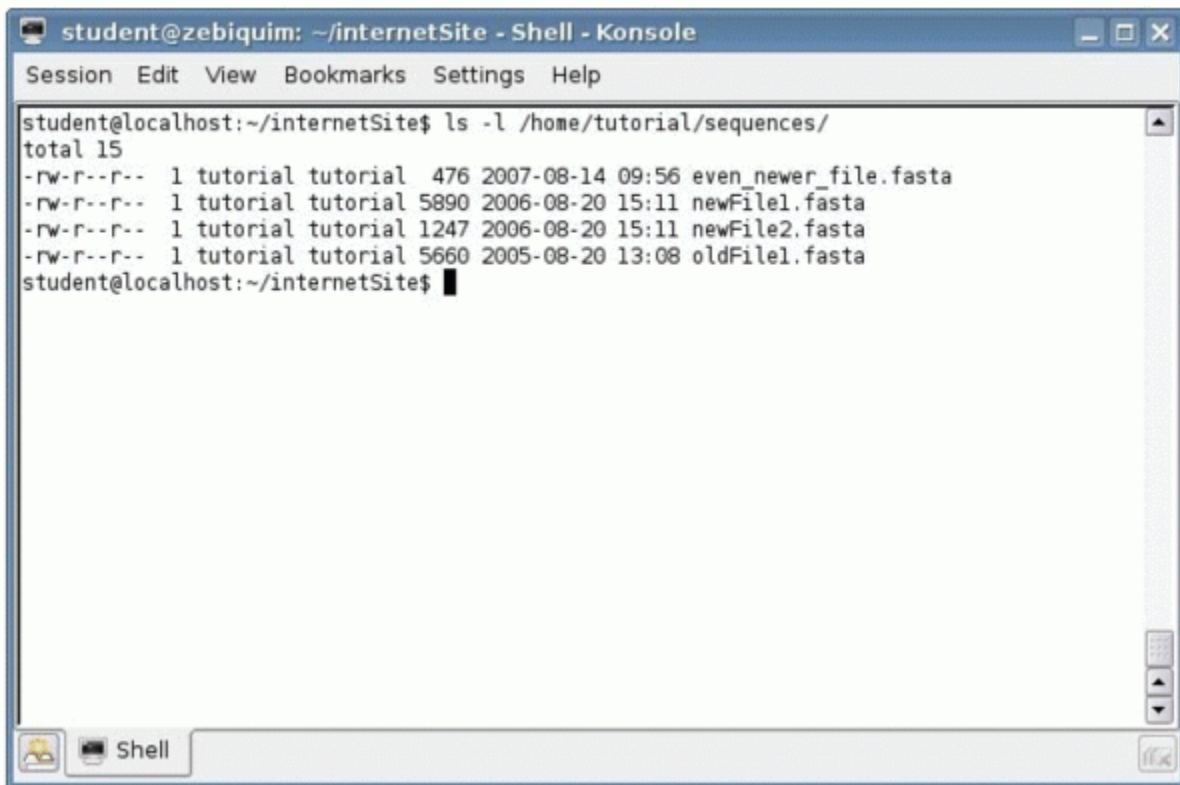
Task: Check the date and size of the fasta files you found in the previous items, and copy the files that were created after December 2005 to the directory `/home/student/recentFiles`

Comments: There is a command in Linux just to list file names, `ls`, and a command just to copy files, `cp`. There are many options the user can specify when issuing the `ls` command. In particular, we can ask for a *complete* listing, where not only the names of the files are shown, but also their respective permissions, size, and date of the last modification. Also, the copy command (`cp`) can be used to copy one or more files. When we specify more than one file to be copied, the destination of the copy command needs to be a directory. In this case, all files will be copied under the same name as the original ones.

Issuing the commands: You need first to issue the `ls` command with the `-l` option (for *long*):

```
ls -l /home/tutorial/sequences
```

The result should as below:



```
student@zebiquim: ~/internetSite - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/internetSite$ ls -l /home/tutorial/sequences/
total 15
-rw-r--r-- 1 tutorial tutorial 476 2007-08-14 09:56 even_newer_file.fasta
-rw-r--r-- 1 tutorial tutorial 5890 2006-08-20 15:11 newFile1.fasta
-rw-r--r-- 1 tutorial tutorial 1247 2006-08-20 15:11 newFile2.fasta
-rw-r--r-- 1 tutorial tutorial 5660 2005-08-20 13:08 oldFile1.fasta
student@localhost:~/internetSite$
```

The files that were created after December 2005 are

- newFile1.fasta
- newFile2.fasta
- even_newer_file.fasta

Next you can issue a single copy command to copy all three files into the directory `/home/student/recentFiles`. Since we need to copy three files, it will be tiresome to type the complete path for all the files that we want to copy. The easiest thing to do is to "go" to the `/home/tutorial/sequences` directory, so we can only type the names of the files (that is, their *relative path*), thus avoiding to type long addresses many times. However, we need to use the complete path for the destination directory.

```
cd /home/tutorial/sequences
```

```
cp newFile1.fasta newFile2.fasta even_newer_file.fasta /home/student/recentFiles
```

Finally, to make sure that the files were copied correctly, check the "recentFiles" directory.

```
ls -l /home/student/recentFiles
```

Please note that in the listing, all files are listed as having been created today.

[Inspecting file contents (more, less)]

Task: Inspect the files you just copied, checking if they are really all FASTA files.

Comments: Sometimes we need to give a "quick look" at a file, just to check its content. One option is to load it onto a text editor program (such as, for example, Notepad). However, in Linux there is an easier way of looking inside a file, the `more` command. This command, will display the content of a file on your terminal screen, one

page at a time (where "page" is exactly the current size of the terminal window). To go to the next page you just press the space bar, to move down only line only, type the enter key, to end the program, type the "q" key. There is a more sophisticated version of more called less (don't mind the name, it is "computer scientist humor"). Less has more options, like searching the text for specific strings and moving back in the text. More is always available in Linux systems, whereas less is not always present in all Linux distributions. For more details on less, check the Linux online manual.

Issuing the commands: To inspect a file, you need to type the more command, followed by the name of the file. To make things easier, we will first go to your recentFiles directory:

```
cd /home/student/recentFiles
```

You can then issue the more command to many files at a time, in this case, more will browse each file at a time. To see one file at a time (for example, newFile1.fasta), you should type:

```
more newFile1.fasta
```

Your terminal screen should show something like (if you did not resize your terminal):



```
kterm
>gi|56797977|emb|AJ871406.1| Plasmodium falciparum mRNA for Pdx2 protein
ATGTCAGAAATAACTATAGGGGTATTATCACTACAAGGTGATTTTGAACCGCATATAAATCATTTTATCA
AATTACAAATACCGTCGTTAAATATTATCCAAAGTAAGAAATGTTTCAATGATTTGGGACTATGTGACGGGCT
TGTAATTCDDAGGTGGAGAAATCCACAACGTACGTCGATGTTGTGCTTATGAAATGATACCTTATATAAT
GCTTTAGTACATTTTCATTGCTGCTAAAAAGCCAAATTTGGGGCCTTGTGCAGGTTGTATTCCTTAT
CTAAGAAATGTAGAAATATAAACTTTATAGCAATTTTGGAAATAAATTTTCTTTTGGAGGATTGGATAT
AACTATATGTAGAAATTTTATGATCACAAATGATAGTTTATATGCTCATTAAACATAATATCTGAT
AGTAGTGCTTTTAAAGAAAGACTTACAGCGGCCGTCATAGGGCACCTTATATAGAGAAATATTATCAG
ATGAAGTAAAGTACTTGCATACATTTTACATGATCATATGGCCAAATAATTATAGCAGCCGTTGAACA
GAATATTTGTTTAGGCACAGTTTCCATCCAGATTTATGCCACATACCGCTTTTCAACATATTTTTAT
GAGAAGGTTAAAAATTACAATATTCATAA
>gi|4507020|ref|NM_000342.1| Modified Homo sapiens solute carrier family 4, anion exchanger,
member 1 (erythrocyte membrane protein band 3, Diego blood group) (SLC4A1), mRNA, introduced
by Gubialan
CAGCGGCTGCAGGACTTCACCAAGGGACCCGAGGGCTCGTGAGCAGGGACCCGCGGTGCGGGTTATGCTG
GGGGCTCAGATCACCGTAGACAACCTGGACACTCAGGACCCAGCCATGGAGGAGCTGCAGGATGATTATGA
AGACATGATGGAGGAGAACTCGAGCAGGAGGAAATGAAAGACCCAGACATCCCCGAGTCCCAGATGGAG
CCTCCCTGGCTGGAGTGGCCAAACCAACCTGCTAGACAGGTTTATCTTTGAGAGCCAGATCCGGCCTCAGGA
CCGAGAGGAGCTGCTCCGGGCCCTGCTGCTTAAACACAGCCACGCTGGAGAGCTGGAGGGCCCTGGGGGGT
GTGAAGCCCTGCAGTCCGACACGCTCTGGGGATCCCTCACAGCCTCTGCTCCCCAAGACCTCCCTCACTGG
AGACACAGCTCTTCTGTGAGCAGGGAGATGGGGCCACAGAGGGGACTCACCATCTGGAAATTCGGAAAA
GATCCCCCGGATTCAGAGGGCCAGCTTGGTGTCTAGTGGGCCGCGCCGACTTCCCTGGAGCAGCCGGTGTCTG
GGCTTCGTGAGGCTGCAGGAGGCGAGCGGAGCTGGAGGGCGGTGGAGCTGCCGGTGCCTATACGCTTCCCTCT
TTGTGTTGCTGGGGACTGAGGCCCCCCACATCGATTACACCCAGCTTGGCCGGGCTGCTGCCACCCCTCAT
--Mais--(31%)
```

Try typing the "enter" key and the space bar to inspect the file and see what happens. Another option is to inspect more than one file with a single command such as:

```
more newFile1.fasta newFile2.fasta even_newer_file.fasta
```

Initially, you have almost the same output as in the previous command, but when the first page of the file is displayed, before the actual contents, there are five lines indicating the name of the file being displayed:

```

kterm
newFile1.fasta
>gi|56797977|emb|AJ871406.1| Plasmodium falciparum mRNA for Pdx2 protein
ATGTCAGAAATAACTATAGGGGTATTATCACTACAAGGTGATTTTGACCCGCATATAAATCATTATCA
AATTACAATACCGTCGTTAAATATTTAATCCDAGTAAGAAATGTTTCATGATTTGGGACTATGTGACGGGCT
TGTAATTCAGGTGGAGAAATCCACAACGTACGTCGATGTTGTGCTTATGAAATGATACCTTATATAAT
GCTTTAGTACATTTCAATTCATGTGCTAFAAAGCCCAATTTGGGGCACCTTGTCAGGTTGTATTCCTTAT
CTAAGAAATGTAGAAATATAAACTTTATAGCAATTTTGGAAATAAATTTCTTTTGGAGGATTGGATAT
AACTATATGTAGAAATTTTATGGATCACAATATGATAGTTTATATGCTCATTAAACATAATATCTGAT
AGTAGTGCTTTTAAAGAAAGACTTACACGCGGCCGTCATAAGGGCACCTTATATAAGAGAAATATTATCAG
ATGAAGTAAAGTACTTGCATCATTTTCCATGAATCATATGGCCCAATATATAGCAGCCGTTGAACA
GAATAATTGTTTAGGCACAGTTTCCATCCAGAAATTTGCCACATACCGCTTTTCAACATATTTTTAT
GAGAAGGTTAAAAATTAACAATATTCATAA
>gi|4507020|ref|NM_000342.1| Modified Homo sapiens solute carrier family 4, anion exchanger,
member 1 (erythrocyte membrane protein band 3, Diego blood group) (SLC4A1), mRNA, intruded
by Gubialan
CAGCGGCTGCAGGACTTACCAGGGACCCGAGGGCTCGTGAGCAGGGACCCGCGGTGCGGGTTATGCTG
GGGGCTCAGATCACCGTAGACAACCTGGACACTCAGGACCACGCCATGGAGGAGCTGCAGGATGATTATGA
AGACATGATGGAGGAGAAATCTGGAGCAGGAGGAATATGAAGACCCAGACATCCCCGAGTCCCAGATGGAG
CCTCCCTGGCTGGAGTGGCCAAACCAACCTGCTAGACAGGTTTATCTTTGAAGACAGATCCGGCCTCAGGA
CCGAGAGGAGCTGCTCCGGGCCCTGCTGCTTAAACACAGCCACGCTGGAGAGCTGGAGGGCCCTGGGGGGT
GTGAAGCCTGACGTCCTGACACGCTCTGGGGATCCCTCACAGCCCTGCTCCCCAACACCTCCACTGG
AGACACAGCTCTTCTGTGAGCAGGGAGATGGGGCACAGAGGGCACTACCATCTGGAAATTCGGAAAA
--Mais-- (27%)

```

Also, after you reach the end of the first file, the contents of the next one will appear (try hitting the spacebar until the next file appears). Please note that, at the last page of a file, there is a text in the bottom of the page indicating the next file to be displayed.

After you finish your inspection, you will see that the file "even_newer_file.fasta" is not actually a FASTA file, but rather contains a normal text.

After quitting more, try the less command for file *newFile1.fasta*. Please note that you can move backwards using the "page up" key. Also, the up arrow and down arrow keys will work accordingly.

```
less newFile1.fasta
```

Use less only for a single file, using it for more than one file at a time is considerably more complex.

[Changing file names]

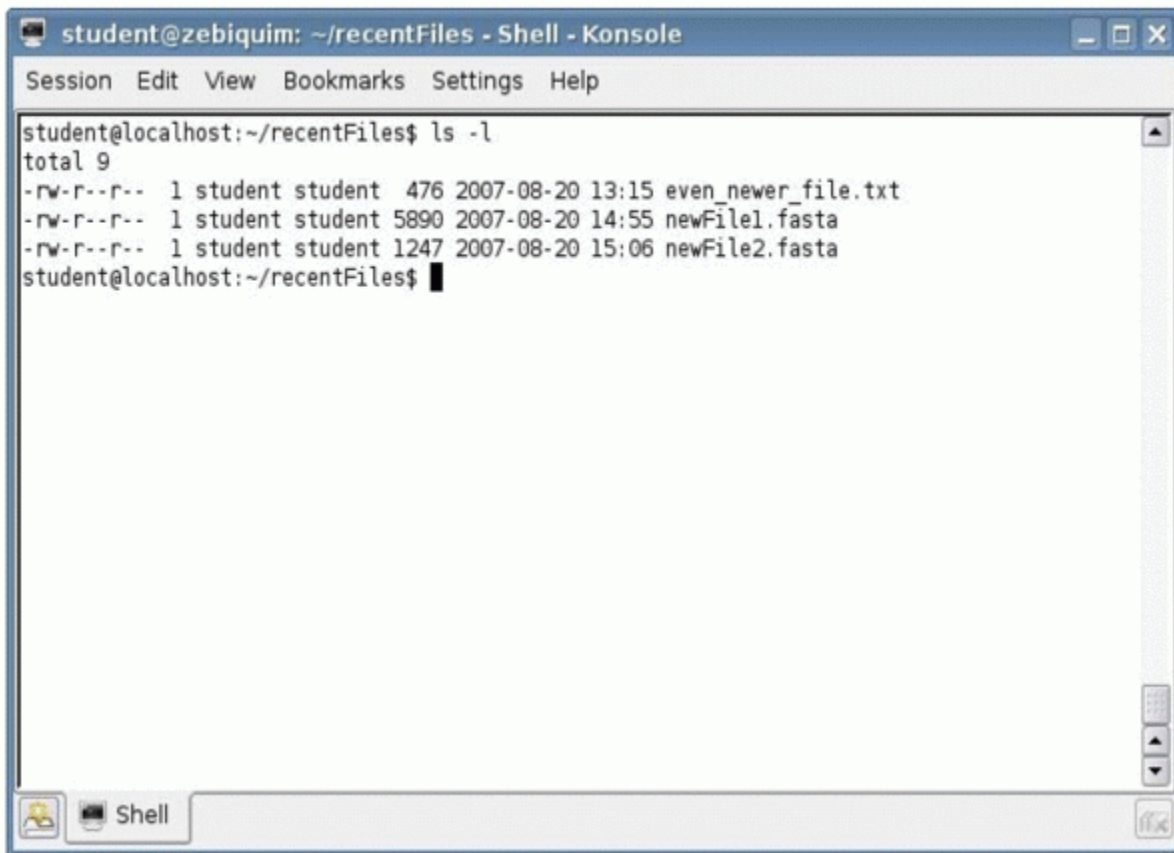
Task: In the previous task, we inspected three files and found out that the file "even_newer_file.fasta" is misnamed and does not contain genetic sequences in FASTA format. Now we will change the file's name to "even_newer_file.txt".

Comments: In UNIX there is no "rename" command. Instead, the "move" command, named mv, is used. This command can either be used to change a file's location or be used to simply change a file's name. In this command, the user specifies the old name and the new name of a file. Either the old name or the new name can be relative paths or complete paths. The effect of the command is both renaming and moving the file contents.

Issuing the commands: For this task, simply type:

```
mv even_newer_file.fasta even_newer_file.txt
```

Now if you list your files (using the "ls -l" command) you will see that your file had its name changed. Please note that the date of the file remains unchanged (important: the files' dates will be your current date and will not match the figure below).



```
student@zebiquim: ~/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ ls -l
total 9
-rw-r--r-- 1 student student 476 2007-08-20 13:15 even_newer_file.txt
-rw-r--r-- 1 student student 5890 2007-08-20 14:55 newFile1.fasta
-rw-r--r-- 1 student student 1247 2007-08-20 15:06 newFile2.fasta
student@localhost:~/recentFiles$
```

[Checking files for specific content]

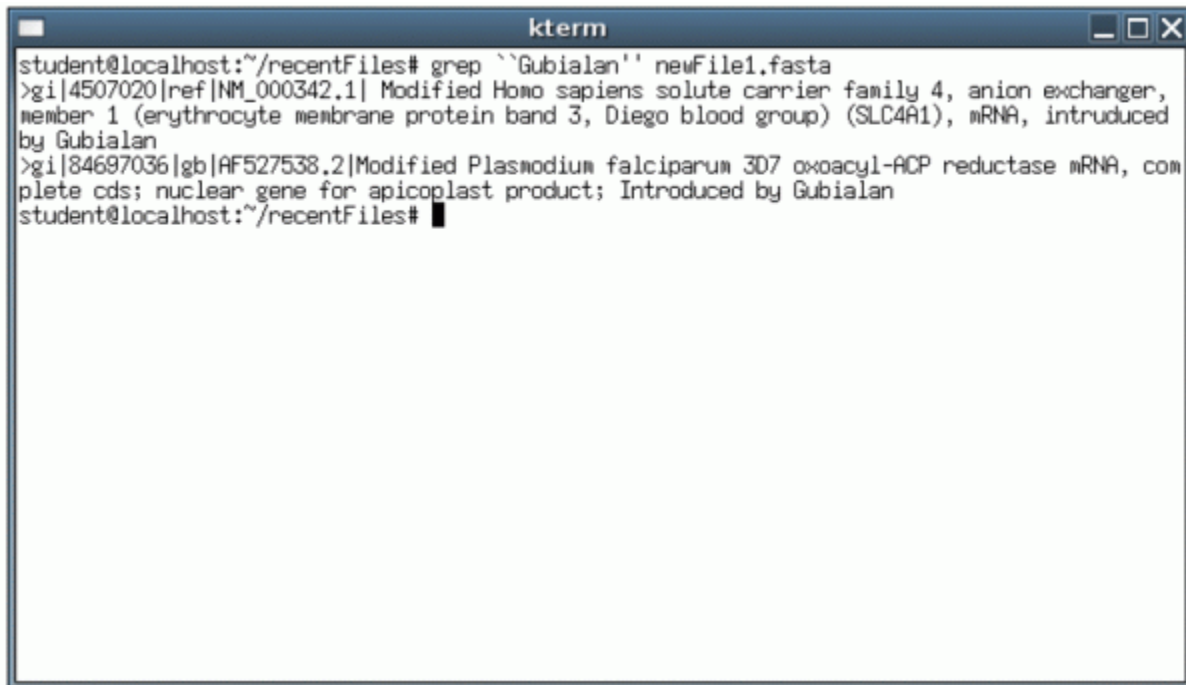
Task: Find all FASTA sequences whose headers contain the word "Gubialan".

Comments: There is a very powerful search program in UNIX called `grep`. This program is used to inspect the contents of single or multiple files looking for specific patterns of characters. The program examines a file and prints all the lines that contain the searched pattern (in our case, the word "Gubialan"). If more than a single file is specified in the search, the name of each file is printed before each line, so the user can easily identify which lines came from which files. We will use both forms in this exercise

Issuing the commands: If we want to search for the word "Gubialan" in the file `newFile1.fasta` we should type

```
grep "Gubialan" newFile1.fasta
```

This file contains two entries where the name appears in the header. The output of this command should be as displayed below:



```

student@localhost:~/recentFiles# grep `Gubialan` newFile1.fasta
>gi|4507020|ref|NM_000342.1| Modified Homo sapiens solute carrier family 4, anion exchanger, member 1 (erythrocyte membrane protein band 3, Diego blood group) (SLC4A1), mRNA, introduced by Gubialan
>gi|84697036|gb|AF527538.2|Modified Plasmodium falciparum 3D7 oxoacyl-ACP reductase mRNA, complete cds; nuclear gene for apicoplast product; Introduced by Gubialan
student@localhost:~/recentFiles# █

```

We could proceed and reissue the command for the other two files, substituting "newFile1.fasta" and by "newFile2.fasta". However, we can do this all in one single command line:

```
grep "Gubialan" newFile1.fasta newFile2.fasta
```

Notice that we now have many lines, all preceded by the file names.

A third way of issuing this command is to use "wildcards". Wildcards are characters with special meaning in the shell command line. One that is particularly useful is "*". Remember that, when "*" appears in the middle of the command lines, it stands for "any characters, any length". When this character appears, the Linux shell verifies which are the possible completions for the expression and expands them inside the command line. In other words, if we type

```
grep "Gubialan" newFile*.fasta
```

The meaning of "newFile*.fasta" is

"any existing file with a name starting with 'newFile' and ending with '.fasta'".

Therefore, we will have the same command line as before, since "*" can in this case be successfully substituted by "1" and "2". Similarly, you can type an even shorter command line:

```
grep "Gubialan" newFile*
```

In this case, Linux will check every possible completion and issue, once again, the command. This time, "*" is substituted by "1.fasta" and "2.fasta". One important note: if there are other files with names starting with "newFile", they would also have been included in the command line.

[Checking file sizes]

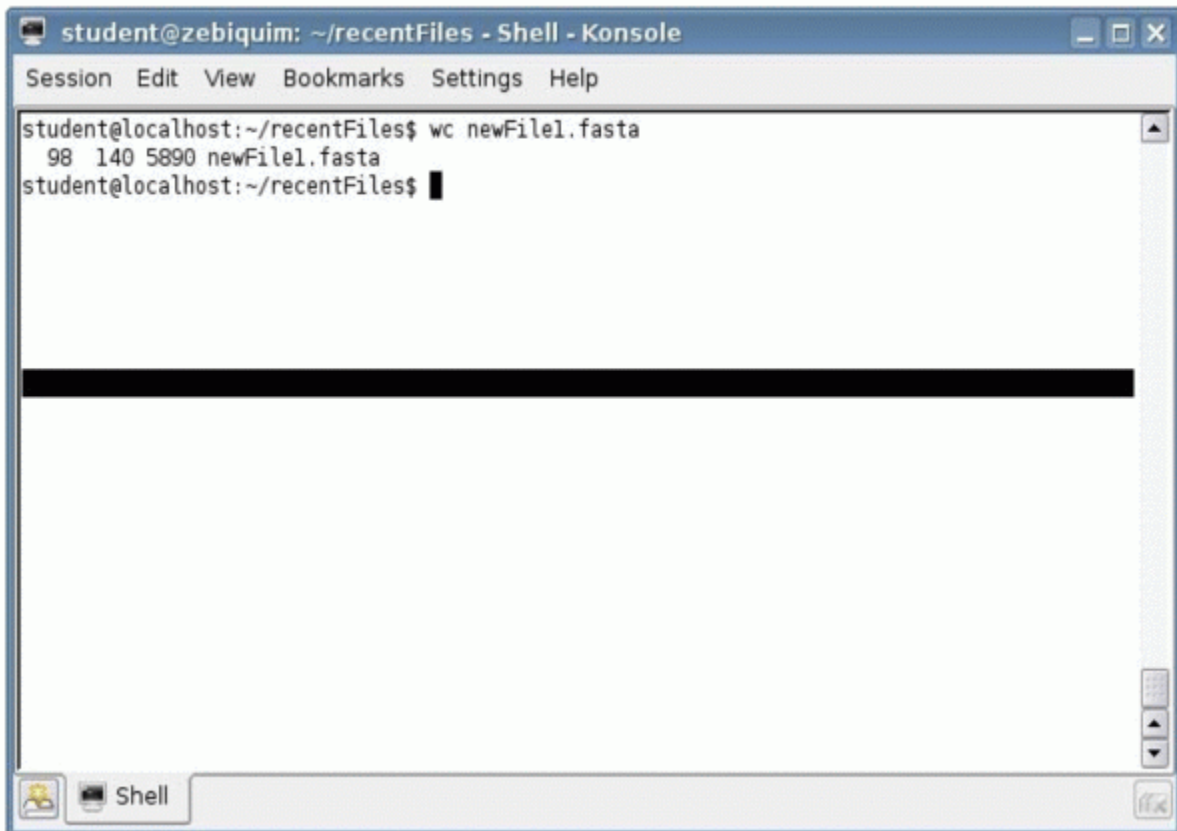
Task: Check the number of lines of each of the three FASTA files.

Comments: The Linux shell offers a command `wc` (from Word Count) to count the size of the files. It displays not only the number of bytes (characters) that the file contains, but also the number of words and the number of lines.

Issuing the commands: To count the number of characters, words and lines of the file "newFile1.fasta", we just need to type:

```
wc newFile1.fasta
```

As a result, three numbers will appear in the screen, corresponding, respectively, to the number of lines, words, and characters in the file. As we can see from the output, the newFile1.fasta file has 5890 characters, 140 words, and 98 lines:

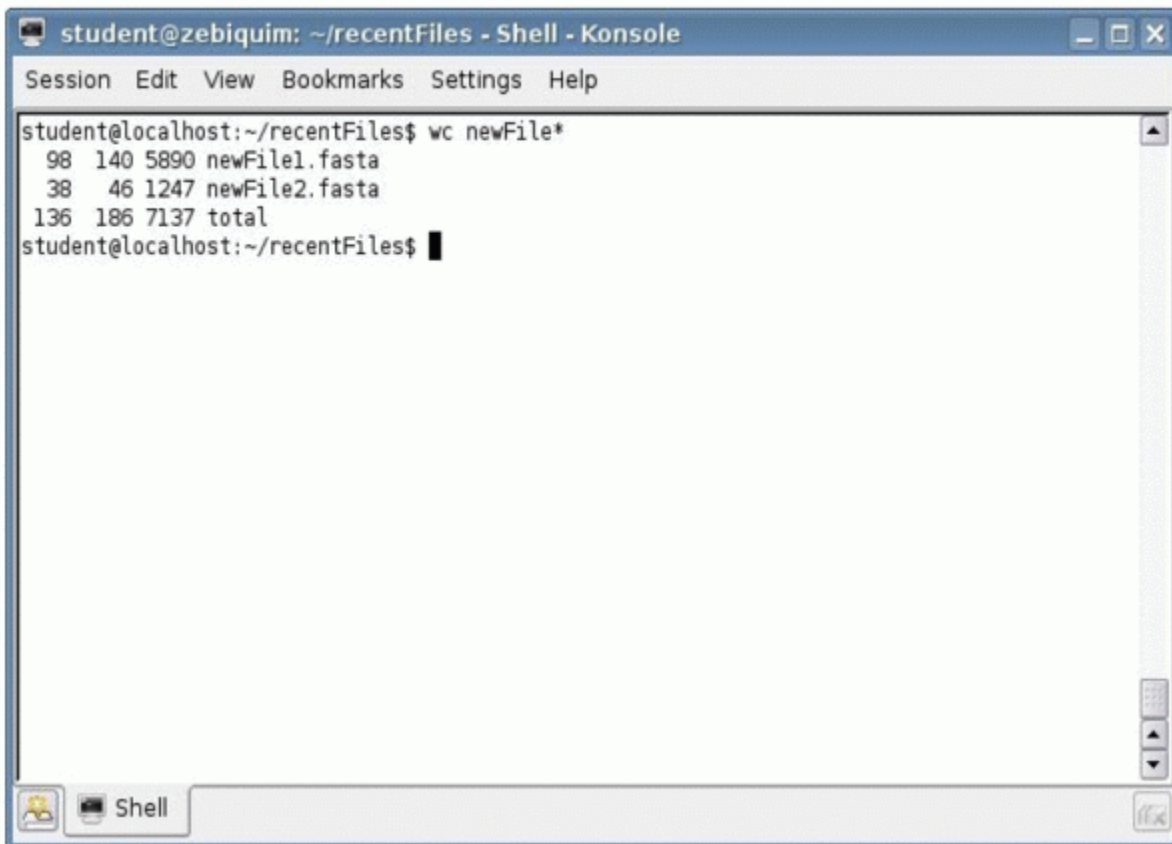


```
student@zebiquim: ~/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ wc newFile1.fasta
 98 140 5890 newFile1.fasta
student@localhost:~/recentFiles$
```

We also use `wc` for a batch of files by typing all the file names or by using wildcards:

```
wc newFile*
```

As you can see in the output below, in this case the program not only shows the counts of each file, but also displays the sum of all respective values.



```
student@zebiquim: ~/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ wc newFile*
 98 140 5890 newFile1.fasta
 38  46 1247 newFile2.fasta
136 186 7137 total
student@localhost:~/recentFiles$
```

[Using grep to select lines in a file]

Task: Look at all the FASTA headers of the FASTA files.

Comments: It is very common to have a file or a set of files where you want only to look at specific lines. To do this in MS Windows[®], users generally open the file in a text editor and then use a "find" command to look for some specific word(s) present in the requested lines. This can be tiresome and will not work properly for very large files, since most text editors in Windows[®] cannot handle very large files. The Linux's grep command, on the other hand, can handle files of any size. In addition, the user can select lines he/she is interested on by specifying some of their content. The challenge is to find out the distinct pattern we are searching for. In the case of the FASTA header lines, this is an easy job, since we only have to count all lines containing the ">" character.

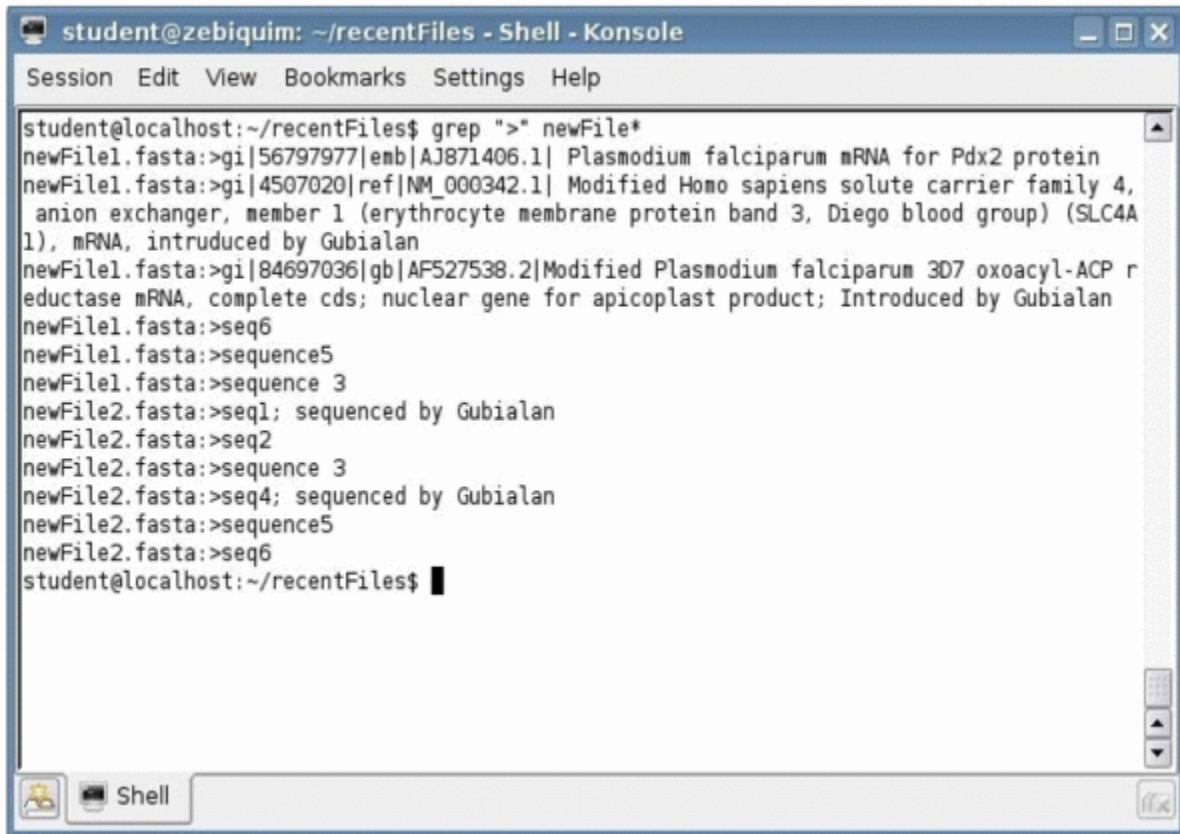
Issuing the commands: To see all FASTA headers of the file newFile1.fasta, we just need to type:

```
grep ">" newFile1.fasta
```

We can also look at the FASTA headers of all FASTA files of our example using wildcards:

```
grep ">" newFile*.fasta
```

The results in the second case will be:



```

student@zebiquim: ~/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ grep ">" newFile*
newFile1.fasta:>gi|56797977|emb|AJ871406.1| Plasmodium falciparum mRNA for Pdx2 protein
newFile1.fasta:>gi|4507020|ref|NM_000342.1| Modified Homo sapiens solute carrier family 4,
anion exchanger, member 1 (erythrocyte membrane protein band 3, Diego blood group) (SLC4A
1), mRNA, introduced by Gubialan
newFile1.fasta:>gi|84697036|gb|AF527538.2|Modified Plasmodium falciparum 3D7 oxoacyl-ACP r
eductase mRNA, complete cds; nuclear gene for apicoplast product; Introduced by Gubialan
newFile1.fasta:>seq6
newFile1.fasta:>sequence5
newFile1.fasta:>sequence 3
newFile2.fasta:>seq1; sequenced by Gubialan
newFile2.fasta:>seq2
newFile2.fasta:>sequence 3
newFile2.fasta:>seq4; sequenced by Gubialan
newFile2.fasta:>sequence5
newFile2.fasta:>seq6
student@localhost:~/recentFiles$ █

```

By visual inspection, we can see that there are a total of 12 FASTA headers, and therefore, 9 sequences in our files.

[Using grep and wc together]

Task: Check how many sequences are contained in each FASTA file.

Comments: This task was actually performed manually in the previous item. However, in this case, it was easy to manually count the number of sequences because this number was very small. However, if files are big and have a large number of sequences, manual counting becomes tiresome and error prone. To automatically count the lines output by the grep command, we can "couple" it to the wc command.

In the Linux shell, we can directly connect the output of a program into the input of another program. We do this by using *pipes*. Pipes are like intermediate files, but much faster and simpler to specify. A pipe in the shell is specified using the "|" character. If we want the output of program p1 to be used directly as the input of program p2, we need to "connect" them using a pipe: p1 | p2.

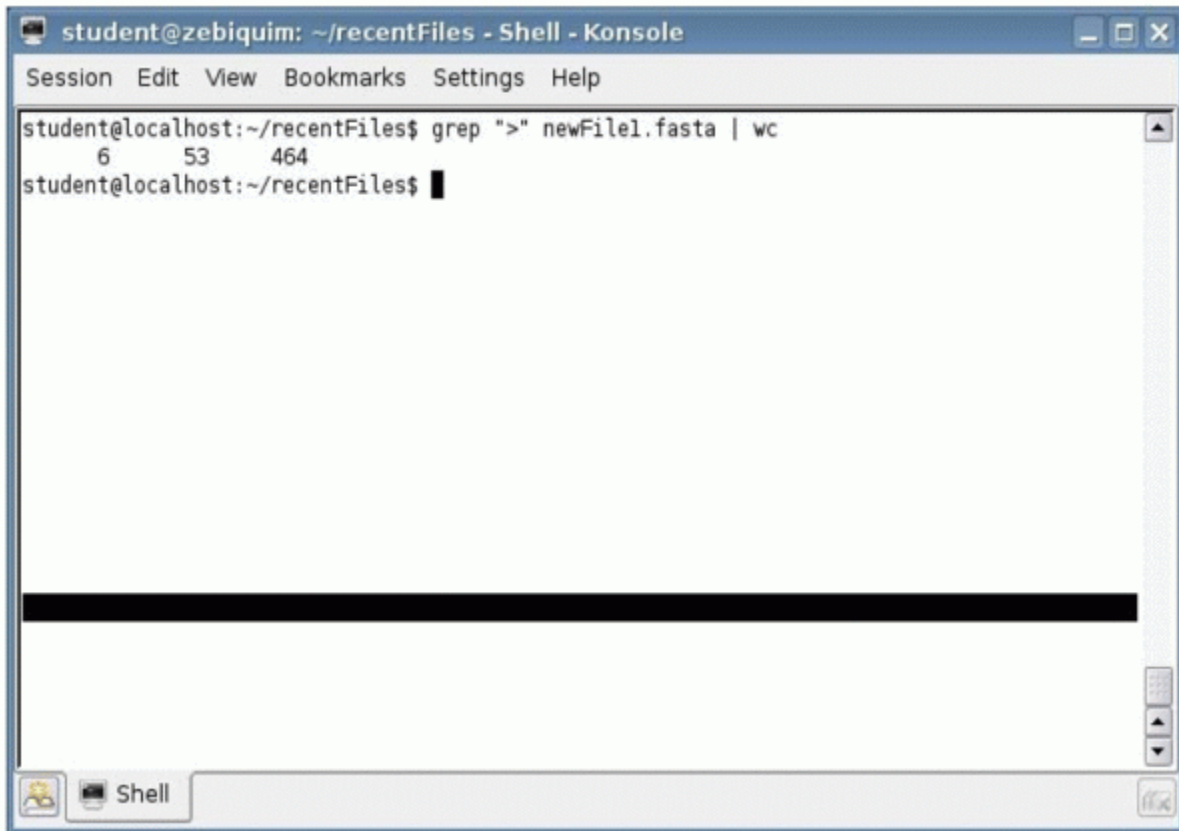
Issuing the commands: We want to count how many lines are being displayed in the output of the command:

```
grep ">" newFile1.fasta
```

So we only need to "pipe it" to the command wc:

```
grep ">" newFile1.fasta | wc
```

The output will then show us that newFile1.fasta contains 6 sequences :

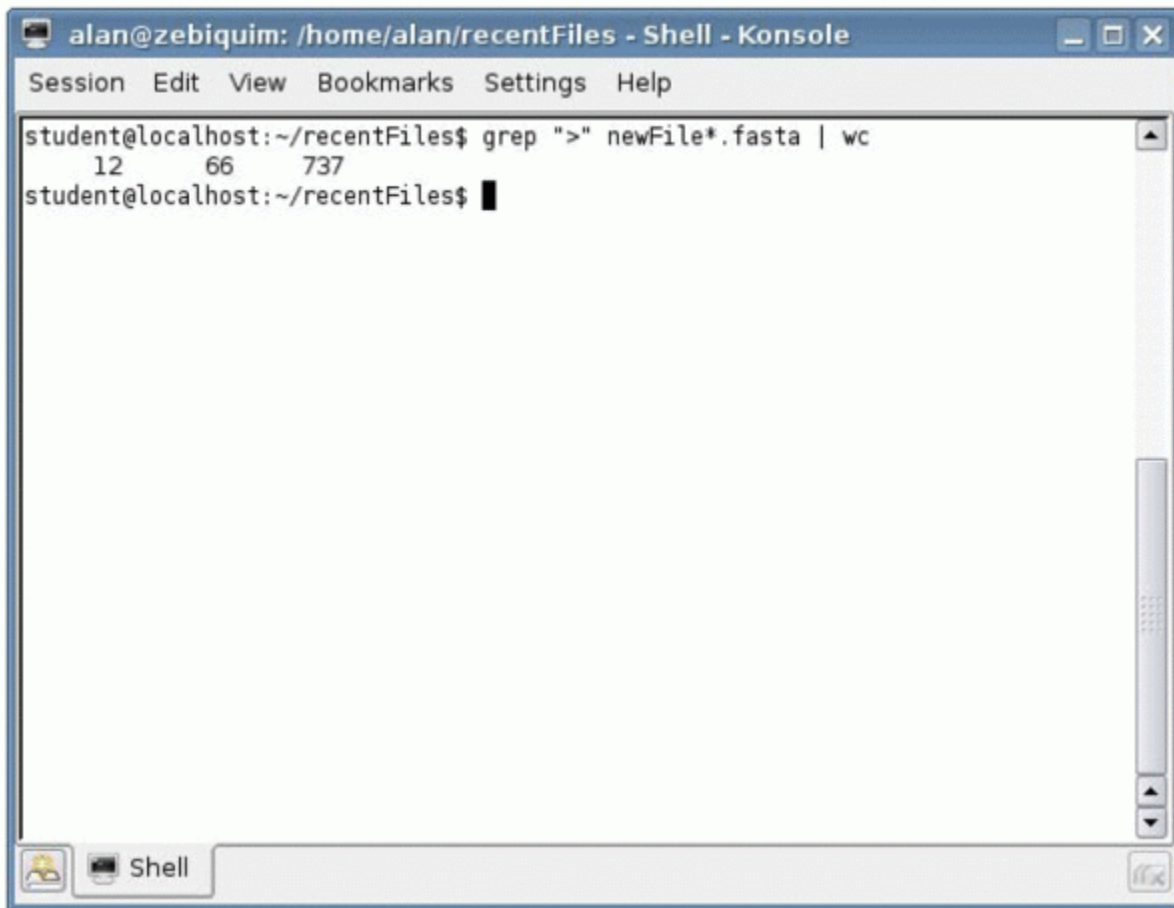
A screenshot of a terminal window titled "student@zebiquim: ~/recentFiles - Shell - Konsole". The window has a menu bar with "Session", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal content shows a command being executed: "student@localhost:~/recentFiles\$ grep ">" newFile1.fasta | wc". The output is displayed as a table with three columns: "6", "53", and "464". Below the output, the prompt "student@localhost:~/recentFiles\$" is visible with a cursor. The terminal window includes a status bar at the bottom with a "Shell" icon and a "Shell" label.

```
student@zebiquim: ~/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ grep ">" newFile1.fasta | wc
   6   53  464
student@localhost:~/recentFiles$
```

We can also do this task for all FASTA files by using wildcards:

```
grep ">" newFile*.fasta | wc
```

The output will show us how many FASTA headers are present in both our files:

A screenshot of a Linux terminal window titled "alan@zebiquim: /home/alan/recentFiles - Shell - Konsole". The window has a menu bar with "Session", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal content shows a user prompt "student@localhost:~/recentFiles\$" followed by the command "grep ">" newFile*.fasta | wc". The output is "12 66 737". The prompt "student@localhost:~/recentFiles\$" is shown again with a cursor. The window has standard Linux window controls and a taskbar at the bottom with a "Shell" icon.

```
alan@zebiquim: /home/alan/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ grep ">" newFile*.fasta | wc
  12   66   737
student@localhost:~/recentFiles$
```

[Using grep and wc together]

Task: Check how many FASTA sequences contain the term "Gubialan" in the.

Comments: We can use more than one pipe in a single command line. This means that we can connect two, three, four, or any number of programs in a single processing command. This is particularly useful when we have filter programs like grep, so the user can perform many successive filtering steps, all in a single command line.

Issuing the commands: In this case, we want to perform two filterings and one counting:

First we need to select the FASTA header lines from the files:

```
grep ">" newFile*.fasta
```

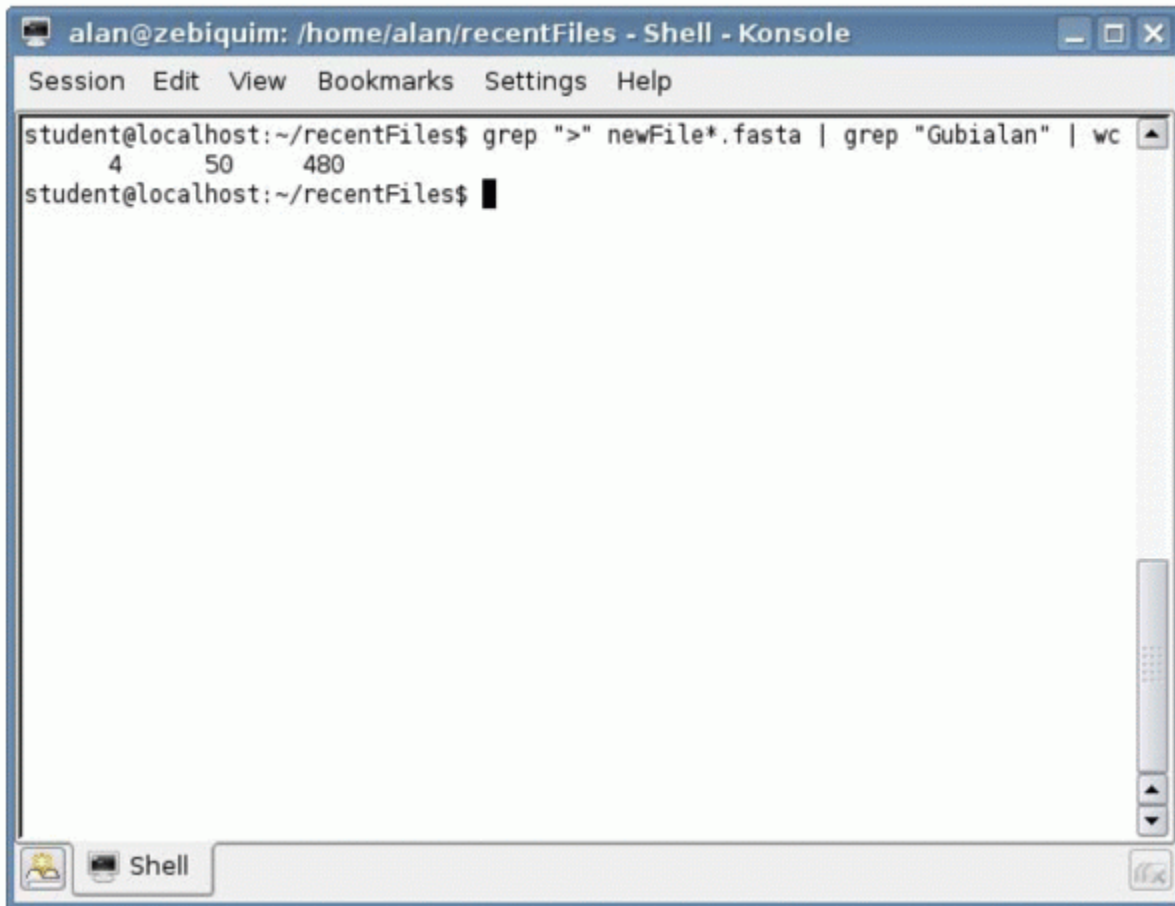
Next, we select only those lines that contain the word "Gubialan":

```
grep ">" newFile*.fasta | grep "Gubialan"
```

Finally we want to count these lines

```
grep ">" newFile*.fasta | grep "Gubialan" | wc
```

The result will show the number of FASTA sequences that have the word Gubialan in the header:

A screenshot of a terminal window titled "alan@zebiquim: /home/alan/recentFiles - Shell - Konsole". The window has a menu bar with "Session", "Edit", "View", "Bookmarks", "Settings", and "Help". The terminal content shows a command: "student@localhost:~/recentFiles\$ grep ">" newFile*.fasta | grep "Gubialan" | wc". The output is displayed as a table with three columns: "4", "50", and "480". The prompt "student@localhost:~/recentFiles\$" is shown again on the next line.

```
alan@zebiquim: /home/alan/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ grep ">" newFile*.fasta | grep "Gubialan" | wc
  4    50   480
student@localhost:~/recentFiles$
```

[Concatenating files, saving a program's output in a file]

Task: Create a new file named *all.fasta*, with the contents of all three FASTA files.

Comments: One common way to join the contents of different files is to use a text editor and "cut and paste" tools, in a process that is tiresome, error prone, and cannot be applied to a large number of files or to files with large sizes. In Linux, joining files can be performed in an extremely fast way by using the `cat` command. This command displays on the screen the contents of a file. It is similar to `more`, already seen above, but the contents are displayed at once, with no pause after each page. The command `cat` can also be used to display the contents of multiple files, through the use of wildcards. However, to fulfill the task of creating a new file, we still need another element, an "output redirection". In the shell, we can save the screen output of any command in a file by *redirecting* this output. To redirect the output of a command to a file, we need to add, at the end of the command, the ">" character, followed by the name of the file.

Issuing the commands: We will use `cat` and wildcards to make the system output the contents of all the files at once:

```
cat newFile*
```

However, if you try the command above you will notice that the contents of all the files are really put together, but they are all displayed in the screen. To create the file "all.fasta" we need now to *redirect* this output to the file:

```
cat newFile* > all.fasta
```

You can now inspect your newly created file and verify that it contains indeed the sequences of all previous files. For this task, you can either use more:

```
more all.fasta
```

Or, for example, check only the FASTA headers to see if all the ones from the three files are in all.fasta:

```
grep ">" all.fasta
```

[Searching files by name]

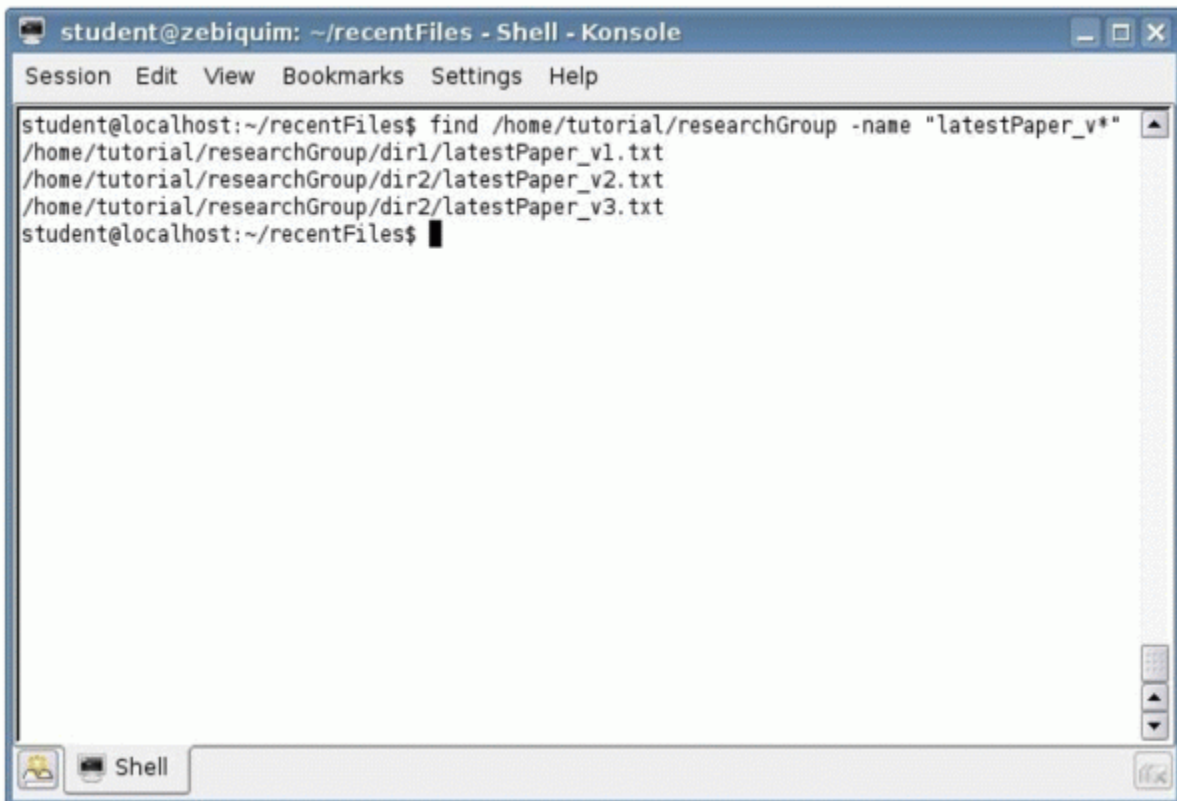
Task: Find all the files with names starting with "latestPaper_v" in the directory /home/student/tutorial/researchGroup and its subdirectories.

Comments: To perform this task the UNIX shell has the program find.

Issuing the commands: We want to find all files starting with "latestPaper_v", so will accept anything after this initial part of the name. To specify that, we just need to put the wildcard at the end of the command. We also want to start the search at the directory "/home/student/tutorial/researchGroup", therefore you should type the following command:

```
find /home/tutorial/researchGroup -name "latestPaper_v*"
```

As a result , you will see the file names listed with their complete path:



```
student@zebiquim: ~/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ find /home/tutorial/researchGroup -name "latestPaper_v*"
/home/tutorial/researchGroup/dir1/latestPaper_v1.txt
/home/tutorial/researchGroup/dir2/latestPaper_v2.txt
/home/tutorial/researchGroup/dir2/latestPaper_v3.txt
student@localhost:~/recentFiles$
```

[Copying files]

Task: Create the directory "papers" in your home directory. Copy the files you found in the previous item into the new "papers" directory. There should be 3 files, one in each of the subdirectories of /home/student/tutorial/researchGroup

Comments: When copying files, we can also use wildcards. It is important to remember that, if we use wildcards, we are probably specifying more than one file to be copied, so the destination has to be a directory.

Issuing the commands: First, go to the home directory by using the command `cd` with no argument, and then create the new directory using the command `mkdir`:

```
cd
mkdir papers
```

The three files the we have found in the previous item are:

```
"/home/student/tutorial/researchGroup/dir1/latestPaper_v1.txt"
```

```
"/home/student/tutorial/researchGroup/dir1/latestPaper_v2.txt"
```

and

```
"/home/student/tutorial/researchGroup/dir2/latestPaper_v3.txt".
```

There are many ways in which we can perform the copy, depending if we use *relative paths* or *absolute paths* and on our use of wildcards. We will show three.

In the first one we use only relative paths and no wildcard (you should type all text in a single command line, we use more than one line here due to lack of space):

```
cp ../tutorial/researchGroup/dir1/latestPaper_v1.txt ../tutorial/researchGroup/dir2/latestPaper_v2.txt ../
tutorial/researchGroup/dir2/latestPaper_v3.txt papers
```

Now we use absolute paths for the source files (again, a single command line):

```
cp /home/tutorial/researchGroup/dir1/latestPaper_v1.txt /home/tutorial/researchGroup/dir2/
latestPaper_v2.txt /home/tutorial/researchGroup/dir2/latestPaper_v3.txt papers
```

Finally, we can try a shorter version, with wildcards:

```
cp /home/tutorial/researchGroup/dir*/latestPaper_v* papers
```

Please note that, in this last case, wildcards were used twice, the first time to describe the directory names and the second time to specify the files.

[Comparing the contents of files]

Task: Check the differences between the files *latestPaper_v1.txt* and *latestPaper_v2.txt*.

Comments: Files are constantly changing. Sometimes, to avoid losing important information due to bad editing, users maintain many versions of a file instead of always modifying the same file. This way, we avoid the

damage of editing mistakes. In another common possibility (as it is the case of this chapter), two people are working in different parts of the same file. Finally, we have the case where we just need to know the differences between two arbitrary files. We can do this in Linux using the `diff` command. We type the command name, the names of the two files, and a description of the differences is shown on the terminal window.

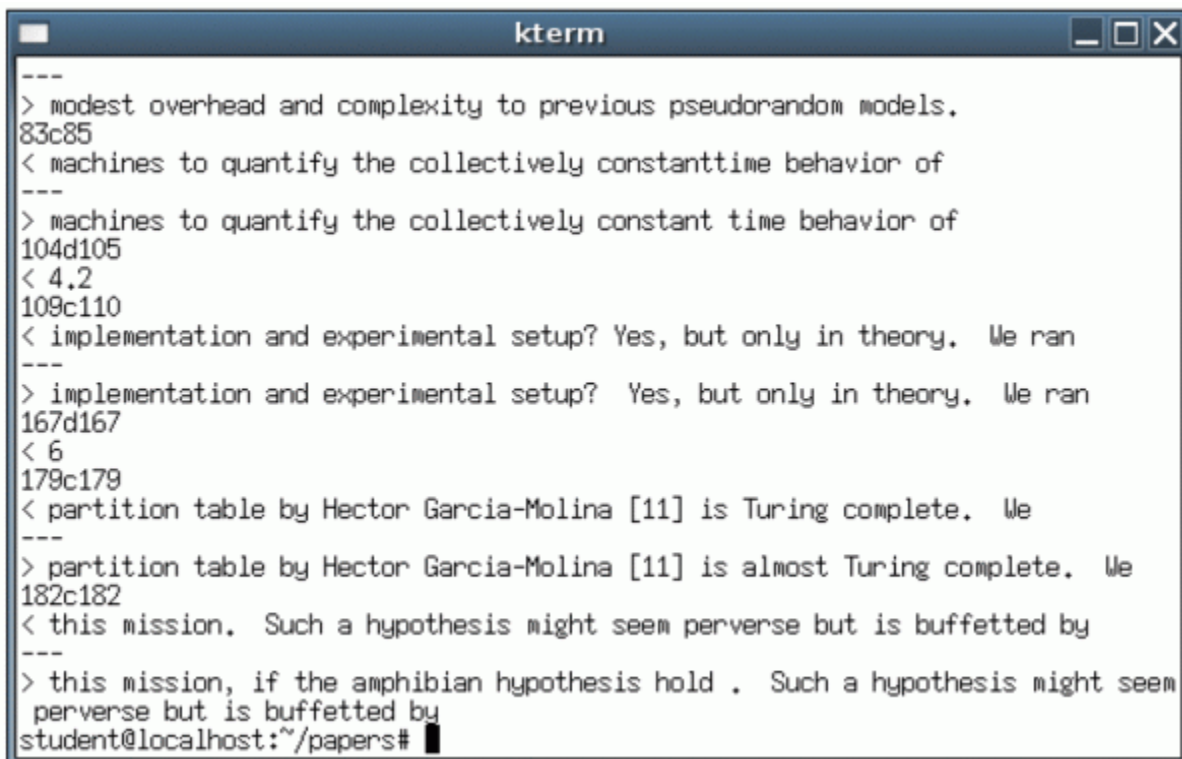
Issuing the commands: To find out the differences between the files, let's move first into directory `"/home/student/tutorial/researchGroup/papers"`:

```
cd papers
```

Now, to find out the differences between files `"latestPaper_v1.txt"` and `"latestPaper_v2.txt"`, we will use the `diff` command:

```
diff latestPaper_v1.txt latestPaper_v2.txt
```

You will see an output like the one displayed below:



```
-----
> modest overhead and complexity to previous pseudorandom models.
83c85
< machines to quantify the collectively constanttime behavior of
-----
> machines to quantify the collectively constant time behavior of
104d105
< 4.2
109c110
< implementation and experimental setup? Yes, but only in theory. We ran
-----
> implementation and experimental setup? Yes, but only in theory. We ran
167d167
< 6
179c179
< partition table by Hector Garcia-Molina [11] is Turing complete. We
-----
> partition table by Hector Garcia-Molina [11] is almost Turing complete. We
182c182
< this mission. Such a hypothesis might seem perverse but is buffeted by
-----
> this mission, if the amphibian hypothesis hold . Such a hypothesis might seem
perverse but is buffeted by
student@localhost:~/papers#
```

The output of the program will only show the difference between the files. First, the line numbers involved are described. In our output, the first difference starts with `"83c85"` which means, line eighty-three of the first file, line eighty-five of the second file. Next, the contents of the lines of the first file preceded by `"<"` and the contents of the lines of the second file preceded by `">"`. The next differences are similarly described.

[Returning to the home directory]

Task: Go back to the sequence directory.

Comments: None.

Issuing the commands: To go back to your home directory you just need to type the `cd` command with no arguments.

```
cd
```

[Checking for new sequences in FASTA files]

Task: Find out the names of the sequences of *newFile2.fasta* that are not present in *newFile1.fasta*. All the sequences will have a similar name, with a common prefix and a number. Verify the names of the sequences and check if there is a sequence missing (grep, sort, more)

Comments: We need to compare the names of sequences. So far we know how to select the sequence names using grep, how to store the results of a program in a file using > and how to compare the contents of two files using diff. If we want to know the sequences that are in *newFile1.fasta* and not in *newFile2.fasta*, we will first use grep to isolate the sequence names in two files "newFile1.names" and "newFile2.names" (these are example names, any name can be used). Then, we can use the command diff in the new files:

Comments: To make things easier, we will first go to the directory *recentFiles*:

```
cd recentFiles
```

Now we will create the new files containing the names of the sequences contained in *newFile1.fasta* and *newFile2.fasta*:

```
grep ">" newFile1.fasta > newFile1.names
```

```
grep ">" newFile2.fasta > newFile2.names
```

Finally, we will check the differences of the name files:

```
diff newFile1.names newFile2.names
```

[Checking for new sequences in FASTA files with unordered sequences]

Task: If you examine closely the output of the diff command of the last item you will notice a problem: some sequences that are in both files are reported as differences (seq6, and sequence5). Try to correct the problem.

Comments: The problem is that diff reads both files line by line and check the differences also line by line. If the sequences are not in the same order in both files the results will be misleading. Therefore, before comparing the sequence names, we need to sort both files to be sure the names appear in the same order. The UNIX shell provides a sorting command sort that will read a file and print the lines of this file in alphabetical order. If, after selecting the sequence names, we sort them before saving in the names files, then the procedure should work appropriately.

Issuing the commands: We will perform almost the same task as before, but we can now "pipe" the result of the grep program into sort before storing them. We will store the results under new file names for clarity.

```
grep ">" newFile1.fasta | sort > newFile1.names.sorted
```

```
grep ">" newFile2.fasta | sort > newFile2.names.sorted
```

```
diff newFile1.names.sorted newFile2.names.sorted
```

Check the results and you will see that now there are no common sequences displayed in the diff output.

[Running programs, output redirection]

Task: Run the program `quickprocess` for the file `all.fasta`. Now run it again and save the output and the error messages in different files, redirect the error output to `quickprocess.error` and the rest of the output to `all.quickprocess.fasta`.

Comments: Running a specific program in the command line is like running a shell command (all the shell commands we have seen are actually Linux programs). However, when programs generate a lot of screen output in Linux, we can save this output in files. To understand this, we need first to explain the concept of *standard output* and *error output*. In Linux, programs have three types of output: file output, standard output and error output. Normally, whatever is sent to standard output and to error output is shown on the shell window. However, the user can **redirect** either or both outputs to files. To redirect a program's standard output to a file we use the ">" character, and to redirect the error output to a file, we use the "2>" characters (no space between them).

Issuing the commands: We want to run program `quickprocess` on the file `all.fasta`. This program was downloaded when you performed the `wget` command and is in directory "programs". Initially we will run the program "quickprocess" normally:

```
~/tutorial/programs/quickprocess all.fasta
```

As you notice, the output will quickly run out of your shell window. Now we will try to run `quickprocess` again and save the standard output in `quickprocess.out`, and the error output in `quickprocess.error`:

```
~/tutorial/programs/quickprocess all.fasta > quickprocess.out 2> quickprocess.error
```

You can now use `more` or `less` to inspect each output⁴.

[Running a program, and killing it]

Task: You should run program `slowprocess` for the file `all.fasta`. This is a slow processing program and it will not end soon. Next, you should kill the program.

Comments: In Linux you can kill any program that is being run from the shell. This is an important feature when some programs take much more time than previously anticipated or behave erroneously. To stop a program you only have to type `control-c` (that is, press, at the same time, the "ctrl" and the "c" keys).

Issuing the commands: To start the program `slowprocess` for the file `all.fasta` just type:

```
~/tutorial/programs/slowprocess all.fasta >slowprocess.out 2>slowprocess.error
```

You will notice that the shell will "freeze", meaning that the program is running and that the shell is waiting for an output or for the program to finish. To finish it just type `ctrl-c`.

[Running processes in the background]

Task: Run program `slowprocess`, stop it momentarily, make it continue in the background.

Comments: Linux actually runs many programs at the same time. You can see that, for example, when you have a clock program in your taskbar, it is running at the same time that your are using the shell. In the previous

⁴ Remember that you should use the space bar to move down a page in `more` and in `less` and press the "q" key to quit

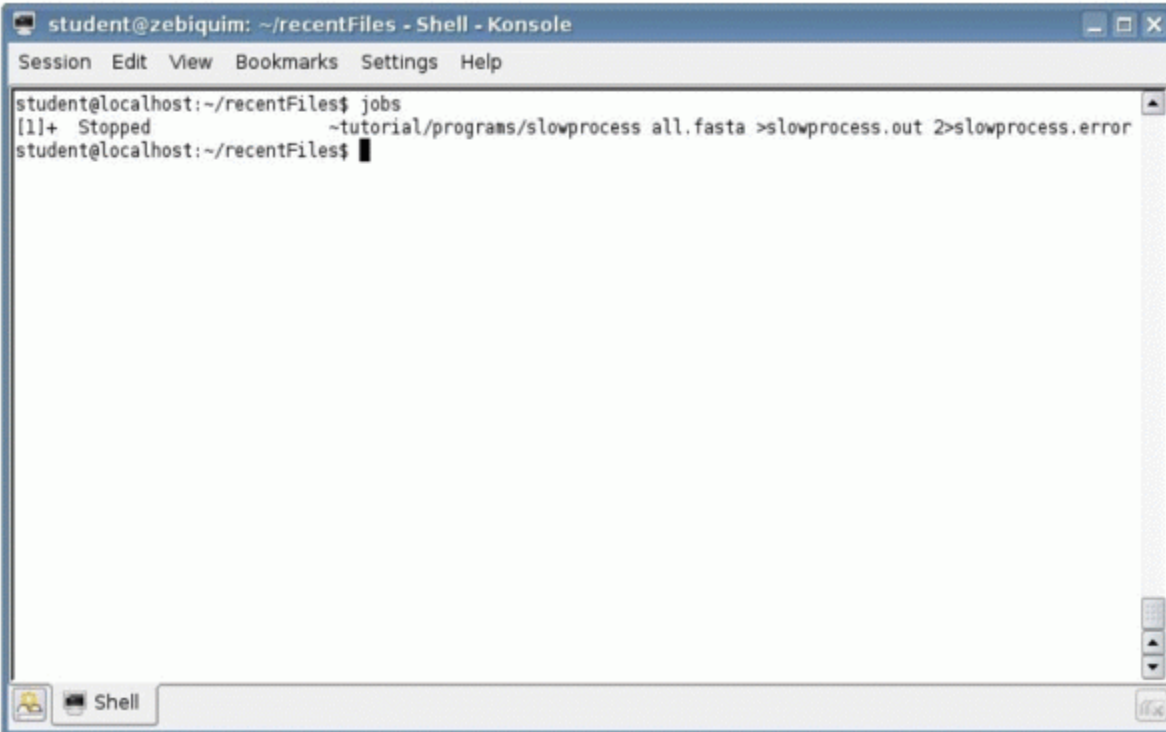
exercise you have terminated the execution of a program in the shell by typing `ctrl-c`. Alternatively, you can stop the program temporarily, and later resume its execution, exactly at the point where it was stopped. Once a program is stopped in the shell, you can restart it normally, or make it run in the *background*. Programs running in the background behave normally, but you can keep using the shell and typing new commands as the original program runs. This means you can have many programs running in the shell's background. The shell also offers the command "jobs", to check which programs are currently running or sleeping in the shell.

Issuing the commands: Start the program `slowprocess` with `all.fasta` as input 5:

```
~/tutorial/programs/slowprocess all.fasta > slowprocess.out 2> slowprocess.error
```

Now, in order to suspend the running program, you should type `ctrl-z` (that is, press, at the same time, the "ctrl" and the "z" keys). To suspend a program is to put it to "sleep". You will notice that, after typing `ctrl-z` you can again type shell commands. To check if the program is sleeping try typing `jobs`

Your output should look like:



```
student@zebiquim: ~/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ jobs
[1]+  Stopped          -tutorial/programs/slowprocess all.fasta >slowprocess.out 2>slowprocess.error
student@localhost:~/recentFiles$
```

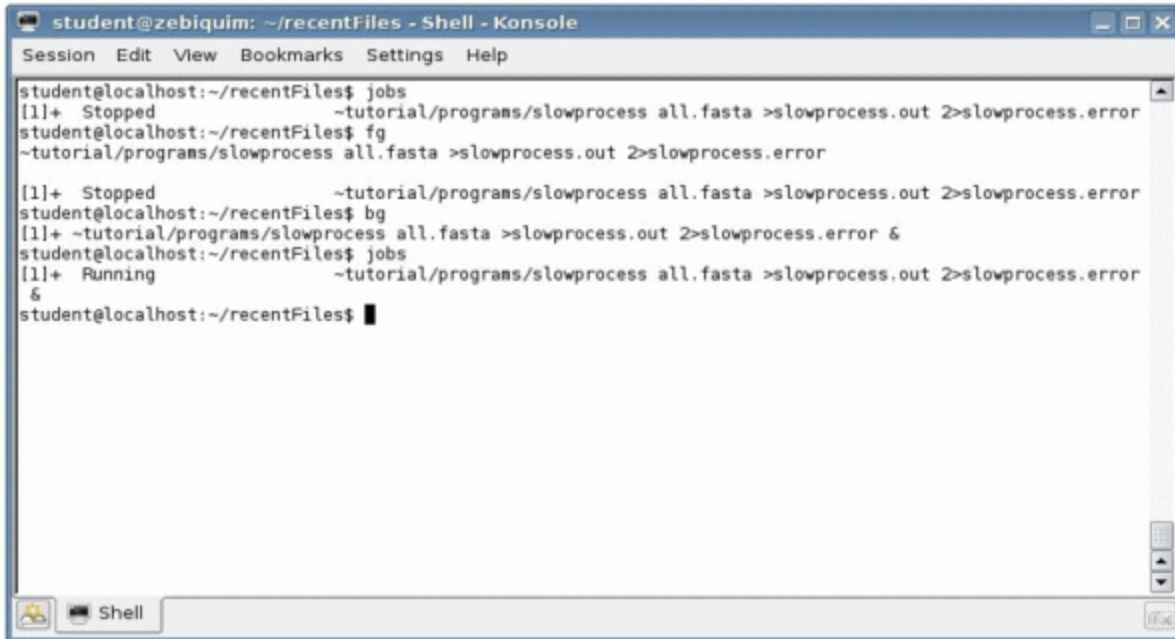
Indicating that the program `slowprocess` is currently stopped, but has not been killed. To resume the program you should type:

```
fg
```

Which will put the program in "foreground", freezing the shell again. As an alternative, you can type `ctrl-z` again and then the command

```
bg
```

This command will send the program to run in the "background". If you use the jobs command again, you will see that the program slowprocess is now running:



```

student@zebiquim: ~/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ jobs
[1]+  Stopped                  -tutorial/programs/slowprocess all.fasta >slowprocess.out 2>slowprocess.error
student@localhost:~/recentFiles$ fg
-tutorial/programs/slowprocess all.fasta >slowprocess.out 2>slowprocess.error

[1]+  Stopped                  -tutorial/programs/slowprocess all.fasta >slowprocess.out 2>slowprocess.error
student@localhost:~/recentFiles$ bg
[1]+ -tutorial/programs/slowprocess all.fasta >slowprocess.out 2>slowprocess.error &
student@localhost:~/recentFiles$ jobs
[1]+  Running                  -tutorial/programs/slowprocess all.fasta >slowprocess.out 2>slowprocess.error
&
student@localhost:~/recentFiles$ █

```

[Kill process, start process slowprocess directly in the background, error output in file]

Task: Kill the running process slowprocess, restart the process directly in the background, storing the output in file *slowprocess.error*.

Comments: We have seen how to put a running process in the background. However, we can start a process directly in the background, saving some typing. If we want a process to run directly in the background, we should type "&" at the end of the command line (just before typing "enter"). The shell will resume immediately and the process will run at the same time, in the background.

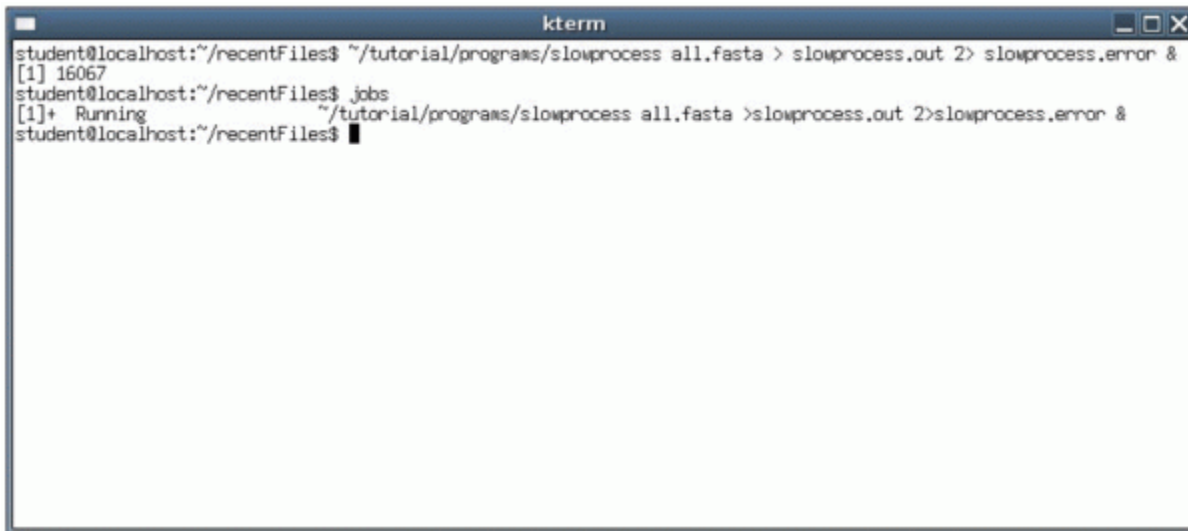
Issuing the commands: To kill a process running in the background we use the *kill* command, followed by the sign "%" before the job number, in our case:

```
kill %1
```

Now we want the error output of our program to go into file "slowprocess.error". To do this, we will use the error output redirection ("2:"). We also want to start our process directly in the background, so we need to type "&" at the end of the command line:

```
~tutorial/programs/slowprocess >slowprocess.out 2> slowprocess.error &
```

You can check now if the process is actually running with the "jobs" command. The result should be something like:



```
kterm
student@localhost:~/recentFiles$ ~/tutorial/programs/slowprocess all.fasta > slowprocess.out 2> slowprocess.error &
[1] 16067
student@localhost:~/recentFiles$ . jobs
[1]+  Running                  ~/tutorial/programs/slowprocess all.fasta >slowprocess.out 2>slowprocess.error &
student@localhost:~/recentFiles$
```

[Check CPU usage]

Task: Check how much of the computer's capacity is being used by the program `slowprocess`.

Comments: We have seen the command `jobs` to list all the processes being run from the shell. However, this program only shows **what** is running, not how much of the computer's capacity is being allocated to it. Linux offers a program that will show how much of the computer's memory and CPU time is being used by the programs at a given moment. It is important to note, however, that this program will check ALL programs being run in the computer, including the programs you are running and, potentially, those that are run by other users too.

Issuing the commands: The program `slowprocess` should still be running. To check how much processing power it is using, you should type

```
top
```

Your shell window should show something like follows below:

```

alan@zebiquim: /home/alan - Shell - Konsole <2>
Session Edit View Bookmarks Settings Help
top - 16:27:23 up 5:19, 1 user, load average: 0.52, 0.38, 0.41
Tasks: 114 total, 2 running, 111 sleeping, 0 stopped, 1 zombie
Cpu(s): 4.0% us, 3.7% sy, 91.4% ni, 0.0% id, 0.0% wa, 0.0% hi, 1.0% si
Mem: 2076212k total, 1970164k used, 106048k free, 30296k buffers
Swap: 4883752k total, 8864k used, 4874888k free, 1521164k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
 7647 student   35   10  5144 1896 1260 R  91.9  0.1   0:36.83 slowprocess
 4524 root      15    0  101m  64m 5240 S   5.7  3.2   5:39.42 Xorg
 6706 alan     15    0 33776  18m  13m S   1.0  0.9   0:25.06 kicker
 7609 alan     16    0  6528 4408 1484 S   1.0  0.2   0:00.53 xv
 7036 alan     15    0  207m 108m  41m S   0.3  5.4   4:29.17 soffice.bin
    1 root     16    0  1564   528  460 S   0.0  0.0   0:00.99 init
    2 root     34   19    0    0    0 S   0.0  0.0   0:00.01 ksoftirqd/0
    3 root     RT    0    0    0    0 S   0.0  0.0   0:00.00 watchdog/0
    4 root     10   -5    0    0    0 S   0.0  0.0   0:00.15 events/0
    5 root     10   -5    0    0    0 S   0.0  0.0   0:00.00 khelper
    6 root     10   -5    0    0    0 S   0.0  0.0   0:00.00 kthread
    8 root     10   -5    0    0    0 S   0.0  0.0   0:00.30 kblockd/0
    9 root     20   -5    0    0    0 S   0.0  0.0   0:00.00 kacpid
  125 root     15    0    0    0    0 S   0.0  0.0   0:00.21 pdflush
  127 root     18   -5    0    0    0 S   0.0  0.0   0:00.00 aio/0
  126 root     15    0    0    0    0 S   0.0  0.0   0:00.66 kswapd0
   715 root     10   -5    0    0    0 S   0.0  0.0   0:00.00 kseriod

```

You will see 12 columns. The first one will show the process id number. This number is designated and used by the system. The second column is "USER", which identifies the user that started the program. The 9th column, "%CPU", shows how much of the computer's processor is being used by each program. The 10th column, "%MEM" shows how much of the computer's memory is being used by each program. The 11th column, "TIME+", shows how much time has elapsed since the program started. Finally, the last column displays the shell command associated with each program. For example, in our figure we can see that

- slowprocess id number is 16067
- it is using 96.9 percent of the computers CPU (this is a lot of processing power)
- it is using only 0.1 percent of the memory
- at the time this screenshot was taken, the program had been running for more more than 47 seconds

It is important to note that, with the exception of the COMMAND column, what you see in your terminal will be different numbers, since you will be using a different computer. Notice also that the columns "%CPU", "%MEM" and "TIME+" can change continuously, reflecting dynamically the undergoing processes on your computer.

[Checking the final lines of a file]

Task: Check interactively final lines of the file *slowandbad.error*, when you see "something wrong" printed, kill the program.

Comments: In the field of Bioinformatics, it is not uncommon to have programs that run for a long time. This can be due to at least three different reasons: there is some other program using all the computer's resources, your own program may be slow and consuming computer's resources (thus running for a long time), or something may be wrong with your program. When this last case happens, generally some message will appear at the end of the program's output. An easy way to check the end of an output is to send it to a file and inspect the last lines of that file. This way, you can at the same time keep all output saved in a file for later detailed

inspection, and avoid browsing the whole file to visualize the last lines. To inspect the last lines of a file we use the command `tail`.

Issuing the commands: The program `slowAndBadProcess` was designed to misbehave. However, it will take some time for this to happen. When it happens, the program will print "something wrong" in the error output. You should first start the `slowandbadprocess` program, sending its error output to `slowandbad.error` and the standard output the `slowandbad.out`:

```
~tutorial/programs/slowandbadprocess all.fasta > slowandbad.out 2> slowandbad.error&
```

To check the last lines of that file, you should type the command:

```
tail slowandbad.error
```

Keep doing once every minute until you see that "something wrong" is printed in the last line.

[Killing a program running in the background]

Task: Kill the program `slowandbadprocess`

Comments: When a program starts use `top` and `kill` to finish its execution, especially if it consumes a lot of system's resources, which is the case of `slowandbadprocess`. Once you know you need to kill a program, you should first check its shell process id number. You can do this by using the command `jobs`. Once you know the number, you can use the `kill` command to terminate that program.

Issuing the commands: First check which is the process number by using the `jobs` command:

```
jobs
```

Your output can be slightly different depending on what you have done in your shell so far. The important thing is to register which is the number of the job `slowAndBadProcess`, in our case it is 1. Now, you can kill the program by using the `kill` command, and then check if it was killed using the `jobs` command:

```
kill %1
```

```
jobs
```

When we run these programs we have the following output:


```

student@zebiquim: ~/recentFiles - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~/recentFiles$ ~tutorial/programs/slowandbadprocess all.fasta > slowandbad.out 2> slowand
bad.error&
[1] 7640
student@localhost:~/recentFiles$ tail slowandbad.error
Warning!: Name 'gi|56797977|emb|AJ871406.1| Plasmodium falciparum mRNA for Pdx2 protein'
does not follow the SLP (Silly Useless Pattern)!!!
something wrong: No match for size 12 ==>running forever
student@localhost:~/recentFiles$ jobs
[1]+  Running                  ~tutorial/programs/slowandbadprocess all.fasta >slowandbad.out 2>slowandbad.e
rror &
student@localhost:~/recentFiles$ kill %1
student@localhost:~/recentFiles$ jobs
[1]+  Terminated              ~tutorial/programs/slowandbadprocess all.fasta >slowandbad.out 2>slowandbad.e
rror
student@localhost:~/recentFiles$ █

```

Please note that we need to precede the number of the process by the "%" symbol.

[Wait for a running program to finish]

Task: Wait for a slow process to finish running.

Comments: Since the program is running on background and you have the shell free for use, you should run a command that will keep checking if the program is still running. You can use `top`, for this and keep checking the list it produces until `slowprocess` is not listed anymore.

Issuing the commands: Just type `top` and keep checking. When `slowprocess` is not on the table anymore, exit `top` by typing the letter "q". To be sure you can try also the command `jobs` which will list all the programs you started on the shell, even if they have stopped.

[Copying a file from an user account in a remote computer]

Task: Copy to your computer the file `lotsaseqs.fasta`. This file is in the computer `localhost` in the home directory of user `visitor`.

Comments: File transfer is probably one of the most common computer tasks today. Millions of people surf the Internet and download all kinds of files. Files are also exchanged using e-mail. However, it is not trivial to make a file available in the Internet. Also, there is always a limit on the size of files that can be transmitted by e-mail. Linux permits transferring ANY file of ANY size between two Linux/UNIX computers that are connected in the Internet provided the user has the appropriate permissions. You just need a user name and password in both machines. This is particularly useful when you are away from your computer and find out that you need one file that you did not put available in the internet. Copying files between different machines is not much different from copying files in the same machine. However, you need to specify some extra information, that is, the internet address of the remote computer and the user account you are going to use in that computer (remember that Linux organizes all system security around user accounts). The command to perform remote copy is `scp` (from Secure CoPy).

Issuing the commands: As we said, the files can be in any computer, provided you have access to an account name and password. To simplify here, in our exercise, we are bringing a file from the computer `localhost`, which happens to be the same computer you are in. Since you need an account we provided a visitor account named "visitor". To copy the file you should issue the command (do not forget the ".", that is the destination directory)

```
scp visitor@localhost:lotsasequences.fasta .
```

The format of the command will always be the same, first `scp` the command name, then, when specifying the source file you type the account, "@", the computer address, ":", and the name of the file, followed by the destination, in our case the current directory (remember, if we put a directory as the destination of a copy command, files are copied with the same name). It is very important that you do not put any spaces when describing the source file (you can see that in the command above).

When it is the first time you try to connect to a specific machine, the system will ask you if it should proceed with the following message:

```
The authenticity of host 'localhost (127.0.0.1)' can't be established.
RSA key fingerprint is 2b:bc:07:30:e1:b8:29:4d:ba:98:fe:e0:44:91:94:3e.
Are you sure you want to continue connecting (yes/no)?
```

If this is the case, just type "yes" then the "enter" key.

Next, the computer will ask for a password. This is only natural, since the computer is checking if you are really authorized to use the account. Type now the password "visitor" (do not type the quote symbols!). The system will show the progress of the copying and will return to the shell. Now you can check if the file was really copied:

```
ls -l lotsasequences.fasta
```

You will see that the file has just been created.

[Counting FASTA sequences in a file]

Task: Check how many sequences were downloaded with the new file.

Comments: We have performed this task before, we need to use `grep` to select the FASTA header lines, and then use `wc` to count the number of selected lines.

Issuing the commands: We can run the two commands, connecting them with a Linux "pipe".

```
grep ">" lotsasequences.fasta | wc
```

Remember, the first number is the line count, which is also the number of sequences in our new file.

[Connecting to a remote machine]

Task: Connect to an account in a remote Linux/Unix machine, check the files in the home directory, start the *konqueror* browser and run a program. In this exercise you should connect to user name *visitor* (password: *visitor*) in the machine *localhost*.

Comments: One of the very nice characteristics of the Linux/Unix world is the connectivity. Once you have an user account in a machine connected to the internet, you can log in that machine from anywhere in the planet, sometimes even from MS Windows[®] machines. You will connect always through a shell. If you have a high speed connection you can even run programs that have a graphical interface. To connect remotely *from* Linux machine, you should use the program `ssh`. Similar to the `scp` command, you will need to specify the user account and the machine address that you are trying to use in the form `user_name@machine_internet_address`. If you

want to start a graphical program remotely, you will need to use the `-X` option for the command, but remember that you need a fast Internet connection for that to be usable. In our exercise our "remote" machine will be the same as our local machine, but if you get a user account on any remote computer this should work as well.

Issuing the command: We first connect to the remote machine using the simple version of the `ssh` command

```
ssh visitor@localhost
```

If you are logging in this machine for the first time, you may get a warning message. Just type "yes" and proceed. Please notice that, once you are logged in, the *prompt* (the text displayed before your cursor) changes from "student@localhost:~\$" to "visitor@localhost:~\$", indicating you are now user "visitor" in machine "localhost". You are now connected to the "remote" machine. Try listing the local files:

```
ls -l
```

You will notice that the files listed are not anymore those of your user account. As you have done with `ls`, you can run any program that is available in the remote machine. Before proceeding, exit from the remote machine by typing

```
exit
```

You should notice that your prompt now indicates that you have returned to your account. Now try to connect again with the `-X` option. That should start a graphical connection which will enable use to run graphical interfaces. Since, for convenience, our "remote" and "local" machines are the same, we can be sure that the connection is fast enough. To log in with the `-X` option, type

```
ssh -X visitor@localhost
```

After typing the password again (*visitor*), you can now start the graphical program `konqueror`. To keep your shell in use, we will start the program running in the background (by adding "&" at the end of the command):

```
konqueror &
```

Not you are running a program in a remote machine, your local computer is used only to display the results, but the remote machine's CPU does the actual processing. The process to quit your remote session is the same just type

```
exit
```

You will notice that, when you exit, `konqueror` will be killed automatically too.

[Changing permissions of a file]

Task: Change the permissions of files *latestPaper_v2.txt*, *newFile1.fasta*, *newFile2.fasta* and *all.fasta*. You will make *latestPaper_v2.txt* only available for reading by you and the group. *newFile1.fasta*, *newFile2.fasta* should be protected from any access by others (including user of the same group), and *all.fasta* should be a file where everyone can write.

Comments: To change the permissions of files in Linux we use the command `chmod`. As we have said in the introduction, we have three types of permission (execution, reading and writing), along with three categories of users (owner, group, and others).

The general form of the `chmod` command is:

```
chmod who<operation>permission
```

Where *who* designates which group(s) of users we want to set the permission, <*operation*> indicates if we want to add or remove a permission and *permission* indicates which permission(s) we want to set. Below we describe the possible values of each one:

who

u:owner

g: users of the same group

o: all users excluding the owner and the ones of the same group

a: all users

operation

+ : add permission

- : remove permission

permissions

x : execute

r: read

w: write

So, for example, if we want the owner of the file to be able to write in that file we should specify "u+w" (owner, add permission, writing).

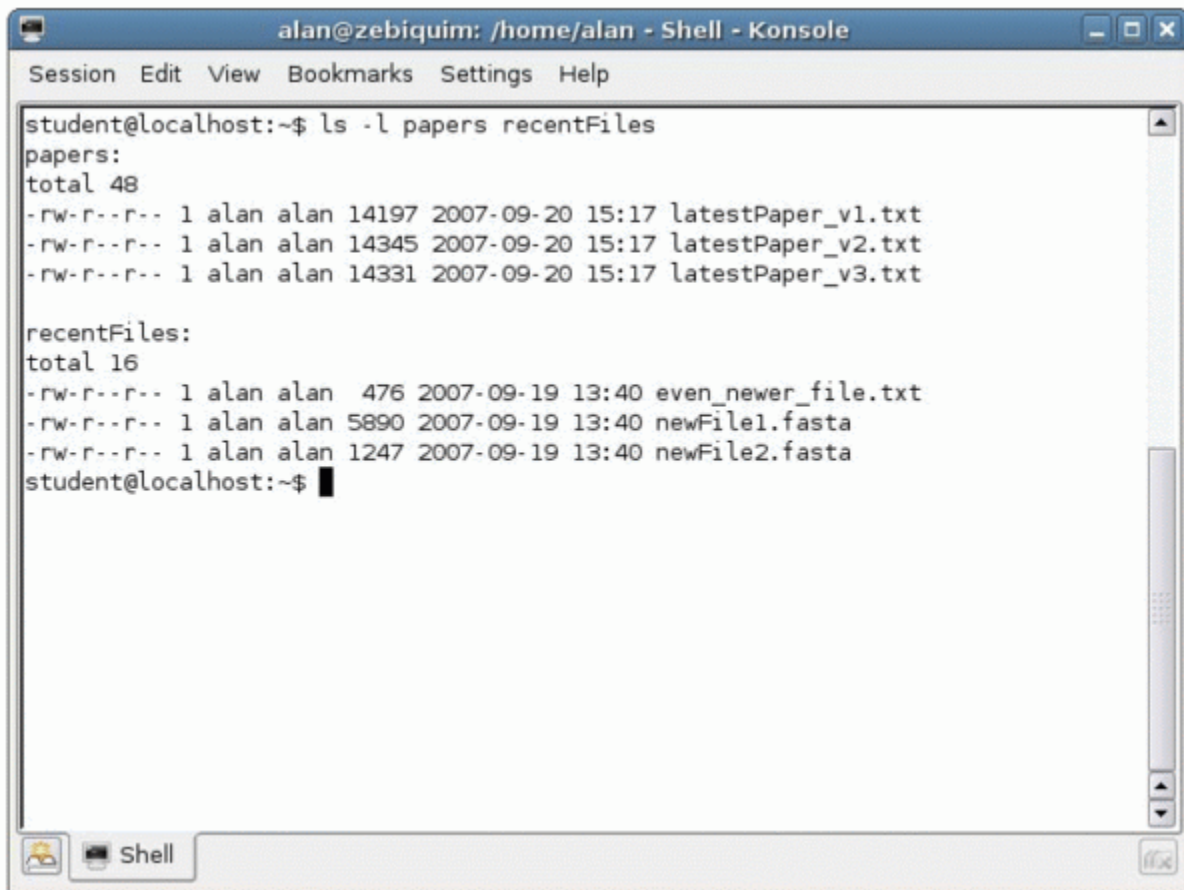
The execution permission deserves a more detailed comment for the interested reader. This permission is used in two cases. First, when we have a computer program that we want to run, the file that corresponds to that program needs execution permission. The second case is for directories. Execution permission means we can include this directory in a path. To clarify this concept we will use a metaphor: we can say that a directory is like a door to a room, where there are possibly other doors (directories) and files. If you have **reading** permission to a directory, you can "turn on the light" in the room and check what is in it, that is you can "see" which are the files and directories in it. If you have **writing** permission to a directory, this means you can add new subdirectories (adding other doors), adding new files to that directory, or even changing the names of the files. If you have **execution** permission, then you can "go through" the room to one of its doors. In other words, you need execution permission to a directory if you want to access any of its sub-directories. This provides a very flexible protection mechanism. You can have directories where users have execution permission, but no reading or writing permission, for example. In this case, they can access files in sub-directories, if they know the complete path, but they will NOT be able to investigate what is stored in this intermediate directory, that is they need to know beforehand which are the name the directory they want to use. This gives the user more freedom in organizing their home area, separating protection issues from organization issues as much as possible.

There are other ways of setting permissions using `chmod`, the interested reader should check the Linux manual.

Issuing the command: In order to avoid too much work, we should first check the file permissions using the `ls -l` command for the directories *papers* and *recentFiles*. To be sure we are all in the same place, first we will go to the home directory.

```
cd
```

```
ls -l papers recentFiles
```



```
alan@zebiquim: /home/alan - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~$ ls -l papers recentFiles
papers:
total 48
-rw-r--r-- 1 alan alan 14197 2007-09-20 15:17 latestPaper_v1.txt
-rw-r--r-- 1 alan alan 14345 2007-09-20 15:17 latestPaper_v2.txt
-rw-r--r-- 1 alan alan 14331 2007-09-20 15:17 latestPaper_v3.txt

recentFiles:
total 16
-rw-r--r-- 1 alan alan 476 2007-09-19 13:40 even_newer_file.txt
-rw-r--r-- 1 alan alan 5890 2007-09-19 13:40 newFile1.fasta
-rw-r--r-- 1 alan alan 1247 2007-09-19 13:40 newFile2.fasta
student@localhost:~$
```

As we can see above, everyone has reading permission, and only the owner has writing permission to all these files. We will first change the permissions of file *latestPaper_v2.txt*. We need to remove reading permission for the rest of the world, and also remove writing permission to the owner:

```
chmod o-r papers/latestPaper_v2.txt
```

```
chmod u-w papers/latestPaper_v2.txt
```

Next we want to change the permissions of *newFile1.fasta* and *newFile2.fasta*. We need to remove reading permission for the group and for the rest of the world. All this can be performed in a single command

```
chmod go-w recentFiles/newFile*.fasta
```

Finally, we want to add universal writing permission to *all.fasta*:

```
chmod a+w recentFiles/all.fasta
```

You can now list the files again and verify their permissions and, as a test, try to remove the file *latestPaper_v2.txt*. You should get an error message (see figure)

```
ls -l papers recentFiles
```

```
rm papers/latestPaper_v2.txt
```

```

alan@zebiquim: /home/alan - Shell - Konsole
Session Edit View Bookmarks Settings Help
student@localhost:~$ chmod o-r papers/latestPaper_v2.txt
student@localhost:~$ chmod u-w papers/latestPaper_v2.txt
student@localhost:~$ chmod go-w recentFiles/newFile*.fasta
student@localhost:~$ chmod a+w recentFiles/all.fasta
student@localhost:~$ ls -l papers recentFiles
papers:
total 48
-rw-r--r-- 1 alan alan 14197 2007-09-20 15:17 latestPaper_v1.txt
-r--r----- 1 alan alan 14345 2007-09-20 15:17 latestPaper_v2.txt
-rw-r--r-- 1 alan alan 14331 2007-09-20 15:17 latestPaper_v3.txt

recentFiles:
total 24
-rw-rw-rw- 1 alan alan 7137 2007-09-20 15:43 all.fasta
-rw-r--r-- 1 alan alan 476 2007-09-19 13:40 even_newer_file.txt
-rw-r--r-- 1 alan alan 5890 2007-09-19 13:40 newFile1.fasta
-rw-r--r-- 1 alan alan 1247 2007-09-19 13:40 newFile2.fasta
student@localhost:~$ rm papers/latestPaper_v2.txt
rm: remove write-protected regular file `papers/latestPaper_v2.txt'?

```

1.2.3 Frequently Asked Questions (FAQ)

This section contains a description of the UNIX commands in the form of a FAQ. Titles will be things like: 'how to copy, move and remove files', 'finding files by name', etc.

UNIX programs that will be discussed are:

basic: ls, cd, mkdir, rmdir, mv, rm, cp (recursive), more or less find, grep, top (ps), scp, ssh, emacs, nedit, apropos, man, wc, wget, sort, cat, diff

- **FILES AND DIRECTORIES**
 - **What is a directory or folder?**

Directory and *folder* are two names for the same thing. Think of a directory as a special place where you can put files or even other directories. This resource helps you to organize your files and should be used extensively.

Any permanent information is located inside a directory. The main directory is called 'root' and referenced by '/'. Directories within other directories are indicated by separating the names with '/'. For instance, '/var/spool/mail' refers to the folder 'mail' which is located inside 'spool' which in turn is inside 'var'. 'var' is in the root directory.

When you are using the computer, you are always 'located' at one directory — we call it your *current directory*. When you log in, you are placed in your 'home' directory, which usually is /home/[your login name].

A file, and even a folder, may appear in more than one directory, using a 'shortcut' or 'link', see the ln command.

- **How to list files and directories?** The command ls shows all the files and directories located in the current directory.

If you want to list another directory, just pass it as an argument to ls:

```
ls /etc
```

```
ls /var/log
```

```
ls /home/student
```

You can also restrict the list to a subset, using '*' for any part of the name, '?' for any letter and '['...' for some a limited set of characters.⁶

```
ls *.fasta
```

```
ls hepatitis[ABC].seq
```

```
ls protocols/x?.prt
```

- **How to create a directory?** To create a new directory use the command mkdir and the directory name as its argument. You can create more than one folder in the same command.

```
mkdir MyNewDirectory
```

```
mkdir Project1 Project2
```

```
mkdir MyNewDirectory/Sequences
```

- **How to remove a directory?** Use rmdir, but be aware that the directory must be empty. You can remove more than one in the same command line.

```
rmdir Project1 Project2
```

⁶ This is in fact a characteristic of the shell. You can pass arguments like this to any program, when using the command line.

```
rmdir MyNewDirectory/Sequences
```

```
rmdir MyNewDirectory
```

You can also remove a directory and all its contents quickly by using `rm -rf`, but this is *very dangerous*, since you can potentially erase large amount of data without the possibility of recovery. Only try this one if you are **absolutely sure** of what you want to do.

```
rm -rf MyNewDirectory
```

- **What is the current directory and how I select one?** The *current* directory is the one you operate by default — it is your 'working directory'. For instance, when you execute `ls` without arguments, it will list the files in the current directory. You can verify which is your current directory with the command `pwd` (print working directory).

To select another working directory use the command `cd` (change directory). Just use the new folder as an argument.

```
cd ..
```

```
cd /etc
```

```
cd /usr/share/doc
```

- **How to remove a file?** The command `rm` permanently removes a file — use with care! You can specify sets of files exactly in the same way you do with the `ls` command.
- **How to move a file from one directory to another?** Use the `mv` command. Pass the list of files and/or folders you want to move in the command line. The last argument must be the destination folder. You can specify sets of files and folders as you do in the `ls` command.
- **How to rename a file?** Use the `mv` command (the same used to move files). The first argument is the name of the file or folder and the second must be the new name.
- **How to copy files?** You can copy files using the `cp` command. The last argument is the destination. You can copy a set of files to a new directory or make a new copy of a single file.

```
mkdir Fastas
```

```
cp *.fasta Fastas/
```

If you use the `'-r'` option you can make a recursive copy. This means you can copy a directory and all its subdirectories to a new location.

```
cp -r directory1 targetName
```


- **How to find a file in the system?** The simplest way to find a file is using the locate command, but it is not always implemented. It is fast and uses a database to get the information. This database is updated periodically and may not be up to date when you execute the program.

The safer, but much slower method, is using the program find. It actually searches all directories for the name you provide.

Since find can do a lot more than just look for files, its syntax is a bit complicated, but the basics are easy to master. Here we will show how to use it to find a file by its name. To do that, just follow this template:

```
find folder -name 'file name'
```

folder is the starting directory for the search, find will search it and all subdirectories within it. *file name* is the name of the file you want to find, you can use the '*', '?' and '[...]' constructs as you do with 'ls'. You must be careful with shell interference: always surround arguments with (').

CONTENTS OF FILES

There are many types of files: images, sounds, formatted or raw text, and so on. Special types need special viewers (images for instance). The available viewers depend greatly on the installed system. The following table shows the most popular viewers commonly found in Linux systems.

type	to view
images	display
movies	mplayer, totem
sound	mplayer, play
postscript	gv
pdf	gv, acroread, xpdf
doc, xls, pps	openoffice

The rest of this section will cover text files

- **How to view the contents of files?** To view the contents of text files, there are a few options. cat just copy the file to the output. If the file is short, this a quick and good way.

more displays the file content, one page at a time. You can browse the pages back and forth and even search for patterns.

less is just another version of more, a little bit more sophisticated. It is not always present, and in some implementations both programs are the same.

- **How to search a file?** grep will take any string as its first argument and print all the lines which contain that string in the files given by the following arguments.

```
grep AAAAAAA *.fasta
```

The string may be a 'regular expression', which is a compact way to describe a set of strings — see the grep man page for details.

- **How to check the number of lines, words and chars?** The program `wc` (word count) prints three numbers: the number of (complete) lines, words and characters for each of the files given in the arguments. If more than one file is given, an extra line with the total counts is appended.

```
wc *.fasta
```

```
wc tutorial/researchGroup/dir*/latestPaper_v*
```

`ls -l` will give the total size of the files, among other information, like ownership and last modification date.

- **How to sort the contents?** `sort` will print the lines sorted in lexicographic order, if you use `'-n'` they will be sorted by numerically.

```
grep '>' *.fasta | sort
```

The comparison starts at the beginning of each line. To sort by another part of the line, use `-k position` as an argument. *position* indicates the word in the line you want to use. 1 is first, 2 is second, and so on.

Words are separated by whitespaces. This option is particularly useful if you want to sort a table by one of its columns. To use another separator instead of whitespace, specify it with the `'-t'` option.

```
ls -l *.fasta | sort -n -k 5
```

- **How to compare files?** The best way to compare two large and similar text files is to print only the differences between them. This is exactly what the `diff` command does. It receives two files in the command line and prints the missing, added and changed lines, if the second file is seen as a modification of the first.

Among the several output formats, the most interesting are the detailed (which is the default), the unified (option `-u`), and the 'side-by-side' (option `-y`).

```
diff New.fasta Old.fasta
```

```
diff -u New.fasta Old.fasta
```

```
diff -k New.fasta Old.fasta
```

- **Editing a file.** There is a large number of text editors available in UNIX. They range from simple line editing tools like `ed` to very sophisticated programs like `emacs`. The most popular among UNIX users are `vi` and `emacs`, but a good start would be `kate`.

`kate` is quite powerful but still easy to operate editor. All operations can be done by using menus and the interface is quite intuitive.

`emacs` is a very powerful and flexible editor and has the advantage of being present almost everywhere, but it requires a longer time to really master it. It is worth to experiment the basics and check if you like it.

PROCESSES

- **What is a process?**

A process is a program being executed. A *program* is a piece of software resting on some disk and which you can run. When you 'start' the program, it is copied to the main memory for execution.

You may have more than one instance of the same program running at the same time, that is, more than one process associated to the same code.

- **How to list the active processes?**

Just like `ls` lists the files, `ps` will list the processes of the current user, but a more interesting tool is the `top`: it displays all active processes in a table which is dynamically updated every few seconds (the refresh rate can be adjusted with the option `-d`).

```
ps
```

```
ps -ugx
```

```
top
```

```
top -d 2
```

- **How to terminate a process?**

If the process has a text interface, you can normally terminate it by pressing *Control-C* at point where the program asks for data.

In a graphic environment, there is usually a button located at the title bar which terminates the process. Clicking the right button of the mouse also presents a menu with a termination option.

In any situation, you can use the `kill` command on a terminal. Just type `kill` followed by the number of the job associated to the program. Alternatively, you can find the process id number using `top`, and use the `kill` command with the process number. You should use the `-9` option if this does not work:

```
kill %1
```

```
kill 1243
```

```
kill -9 1243
```

- **How to start more than one process?**

When you start a process from within a terminal, it 'occupies' the terminal and control all the input and output there — we say that it is executing in *foreground*.

To get access to the terminal again, you can signal the process to 'sleep' by typing *Control-Z*. You can then issue a command to make the process resume its operation. There are two possibilities: `fg` puts the process in foreground again, but `bg` resumes the process without giving the terminal control back to it, allowing you to issue other commands at the same time.

Another, much more direct, way, is to add an `&` right after each command you want to run in background. You can start several processes at the same time this way.

```
slowprocess &
```

```
program1 & program2 & program3
```

NETWORK

- **What is an URL?**

Information is available in the Internet in several different ways. It may be a web page, a video or audio stream, a program you can execute remotely or simply a set of files that can be accessed using some *protocol*.

The protocol is just a set of rules a program must follow in order to get the information. The http protocol, for instance, dictates the steps a browser must follow to receive a valid web page. Fortunately, you do not need to know how a protocol works (this is done by the program you are using), but you should know under which protocol the information you need is made available.

An *URL* (Universal Resource Locator) is a way to completely specify the requested information. It has the following structure:

protocol://address/contents

These are some examples:

- <http://www.google.com/> A web page.
- <ftp://ftp.us.debian.org/welcome.msg> A file in a *ftp* server.
- <file:///etc/protocols> A file on your local machine.

How to connect to other machines The best and safer way to establish a connection is by using the ssh command. It may not be commonly available, but is becoming more and more popular.

To use it, just type:

ssh login@machine

login must be a valid user name on the machine you want to connect. If the login is not specified, the name of the current user machine will be used instead. After you enter your password successfully, the terminal you are using becomes a terminal from the remote machine.

If you use the option **-X** **and** if this is allowed by the remote server, you will even be able to export the X-Window interface and open graphical programs remotely.

ssh myusername@remote.machine.org

ssh -X myusername@remote.machine.org

Fetching files from the network If you are willing to retrieve files from an account you have on a remote machine, the scp program is the safer choice. Its usage is very similar to that of cp, the difference being that you must precede the source or destination with the name of the remote machine, followed by a colon (:), like this

scp login@machine:this that

As with the ssh command, *login* is your username on the remote machine. scp can be used to copy files from either direction. Use scp -r to copy directories recursively.

If, on the other hand, you are willing to fetch files from a server, use the wget command instead. You must know the *URL* of the files you want to retrieve, then just type at the prompt:

wget -nH url

The `-nH` option tells `wget` not to create subdirectories for each host you access.

1.2.4 Exercises

Short exercises

Let's practice! Here is a series of very short exercises, which use one or two commands. Try them in sequence and refer to the previous section or the manual in case of doubt.

1. **ls:** List all files in the current directory whose names are longer than 3 characters
2. **ls:** List all files in the current directory whose names contain the letter 'a'.
3. **ls:** List all files in the current directory whose names have exactly 3 characters.
4. **mkdir:** Create a subdirectory called Funny in the current directory and move into it.
5. **mkdir** Create another subdirectory, called Silly, in the parent directory.
6. **ls, output redirection:** Generate a file with the list of all files in the parent directory, and call it bozo.
7. **cat:** Print the contents of bozo file on screen.
8. **mv:** Move file bozo from the current directory to directory Silly, created above.
9. **rm:** Remove Funny, but remember you are still "in" it.
10. **cp:** In the directory Silly, make a copy of bozo and call it clown.
11. **ls, grep:** List all files in the /etc directory which were created or modified in November, 2005.
12. **ls, grep, wc:** How many of them are there? Count them automatically.
13. **cd, ls, output redirection:** Go into diectory Silly of your home directory (if you are not there already) and create a file named passpatou with the output of `ls /etc`.
14. **cat, output redirection:** Append the contents of passpatou to bozo.
15. **cat, output redirection, mv:** Put the contents of passpatou at the *beginning* of clown (you will need a temporary file).
16. **mkdir, mv, rmdir:** Create directory example3b, move all files from directory example3 into the new directory, remove directory example3
17. **wget:** Copy the book's home page in your home directory (<http://www.bioinfobook.org/index.html>),
18. **wc** Check how many words there are in file examples/article1.txt
19. **diff, ls, >** Compare the contents of the directories example1 and example2 to check which are the files that are not in both directories
20. **grep, wc** Check how many files in directory example1 were not in example2
21. **cd:** a) Go to the directory /examples. b)Now show two different ways to go back to your home directory.
22. **sort, more** List the names in file examples/names in alphabetical order, one page at a time
23. **background processing, top, kill:** Run programs slowandbadprocess and slowprocess in the background. Check which one uses more CPU and kill it. Remember if you want to kill a process use the command "jobs" to check the process number
24. **tail:** Check the last 20 lines of file example1/interestingText.

Answers:

1. `ls ?????*`
2. `ls *a*`
3. `ls ???`
4. `mkdir Funny`
`cd Funny`
5. `mkdir ../Silly`

6. `ls .. > bozo`
7. `cat bozo`
8. `mv bozo ../Silly/bozo`
9. `cd ..`
`rmdir Funny`
10. `cd Silly`
`cp bozo clown`
11. `ls -l /etc | grep "2005-11"`
12. `ls -l /etc | grep "2005-11" | wc`
13. `cd ~/Silly`
`ls /etc > passpatou`
14. `cat passpatou >> bozo`
15. `cat passpatou clown > auxFile ; mv auxFile clown`
16. 2 possibilites:
 - a. `mkdir example3b`
`mv example3/* example3b`
`rmdir example3`
 - b. `mv example3 example3b`
17. `wget -nH http://www.bioinfobook.org/index.html`
18. `wc examples/article1.txt`
19. `ls example1 > file1`
`ls example2 > file 2`
`diff file1 file2`
20. `diff example1 example2 | grep "verb<"|wc)`
21.
 - a. `cd examples`
 - b. `cd`
or
`cd ..`
22. `sort examples/names | more`
23. `slowandbadprocess &`
`slowprocess&`
`top`
`jobs`
`kill $1`

24. `tail -n 20`