Introduction to

UNIX

Written by: Luce Skrabanek, ICB, WMC 2004

Reasons for biologists to learn Unix

As biological data sets grow in size, problems become more complex and require more computer power. Computers that can provide this power use Unix.

Brief history

Unix was originally developed at Bell Laboratories as a private research project by a small group of people starting in 1969. Their aims were to design an operating system (OS) which was simple and elegant, written in a high-level language (rather than assembly), and allowed the re-use of code. Unix has a small amount of assembly-level code (the kernel), and the rest is written in C. This means that users could write applications in C, and easily make use of all the OS facilities.

The advantages of Unix are its portability and its popularity.

Unix is popular because it is relatively easy to get Unix on a system (because so little of it is written in low-level language), the application program interface allows many different programs to be easily implementable across different platforms, and it is easy to network multiple systems.

Unix runs on many different systems, so skills learned on one computer can be used with almost any other system. Unix is an inexpensive and flexible computing system.

Unix is now produced by a number of companies for a number of different platforms. This means that Unix is not a single operating system, but a number of OS which differ from each other in small ways. Most commands are common to all systems.

The main task of an OS is to manage all the computer operations and provide a link between the user and the system resources, The Unix OS runs applications, provides access to system resources including hard drives, CD-ROMs, etc, allows multiple users to log on simultaneously, and provides security for the computer and the data stored on it.

Unix functionality is made up of many programs that can be individually modified by anyone with programming experience.

There are three components of the OS:

- The kernel, which is the core of the OS. It manages devices, memory, processes and background processes. It transmits information between the system programs and the system hardware. It schedules and executes all commands issued, and manages functions such as swap space and background processes.
- The shell, which is the interface between the kernel and the user. It acts as an interpreter or translator of commands. The shell accepts commands issued by the user, interprets what the user typed, and then sends this information to the kernel for execution.
- The file system, which is a hierarchy of directories, subdirectories and files.

Different shells, which are basically historical remnants from different development groups, have different sets of commands, and different syntax.

Conventions used in this manual

User-entered commands are written in **`bold Courier font`**. `Normal Courier font` is used for all text which comes up on the screen which has not been user-entered. Anything within '<' and '>' indicates my comment or instruction.

Any comment which is specific to the computers used by the ICB are highlighted by this symbol ✪.

A quick note on terminology

Standard input: what the user types in from the keyboard.
Standard output: the text that results from the process of a command and is outputted to the terminal screen.

The basics

Logging on: The first thing you need is a Unix account on the server that you will be accessing. Each account is protected by a unique username, and a password. When you try to access the system, you will be prompted for your username (or login). Unix usernames are always lowercase. Enter your username, then hit 'Enter'. You will then be prompted for your password. When you type in your password, it will not show up on the screen. Passwords are case-sensitive, so make sure you type in your password exactly as it was given to you. Hit 'Enter'.

Once you have logged on, you will see your cursor beside some text at the right-hand side of your screen. This text is your system prompt, which is where you enter commands which will be interpreted by the shell. The system prompt may include information such as the machine that you are logged on to, your current directory, the current time, or the number of commands that you have entered in this particular session.

✪ The Unix accounts that you have been given use the bash shell. After typing in a command after the prompt, hit 'Enter', so that the shell knows that it should now process that command.

The first time you log in, it is advisable to change your password. You can do this by typing **`passwd`** after the prompt, and hitting 'Enter.' **`passwd`** is a command which tells the system to run the password changing program.

✪ The ICB uses an LDAP system to change passwords. This changes your password on all ICB computers at once.

```
prompt> passwd
changing password for <username>
Old password: <type in the password that you used to access
this account>
New password: <type in your new password, which ideally
should be something that you will remember>
Retype new password: <re-enter your new password>
```

None of the passwords you type in will displayed, for obvious security reasons.

To correct mistyped commands, use the delete or backspace key (this will differ on different machines, depending on how the Unix program has been set up). Some computers require you to enter Control-H (i.e., pressing the Control key and the 'H' key at the same time).

To exit a Unix session, use the **`logout`** command, i.e., type **`logout`** after the prompt. You can also logout by typing Control-D, or **`exit`**. Wait for the message confirming that you have logged out before ending the program that gives you access to the Unix system. Sometimes you will get the message `There are stopped jobs`. In this case, just use any of the above logout commands again.

Unix filesystems, and moving around in them

The Unix filesystem is very similar to the PC or Mac filesystems that you may be familiar with. It is a hierarchical structure. This means that there is one main directory, which contains files (such as executables, or programs), and other directories (termed sub-directories), which in turn can include other directories, and so on. Directories are equivalent to folders on a PC or Mac and are a means of organizing your files. The main directory, similar to your PC/Mac hard disk folder, is called the root directory, and is represented by '`/`'. There are several system directories contained in the root directory, such as `/bin, /etc, /dev, /lib, /usr, /home`.

Any directory or file in the system can be referred to by an absolute pathname, which begins with the root directory, and then lists each of the subsequent directories in the hierarchical tree structure which you have to go through to get to the file or directory you want, each separated by a '`/`', and then finally the filename (or directory) that you want to go to. For example, `/hosts/fulcrum/homes/user` refers to the home directory of a user called 'user'. This directory is contained in a directory called `homes`, which in turn is contained in a directory called `fulcrum`, which is contained in a directory called `hosts`, which is found in the main root directory.

The user's home directory is the one to which the user will automatically be directed to once they log on to the system. If you are currently within your home directory, you can find the absolute pathname of your home directory using the command **`pwd`** (which stands for 'print working directory', and shows the absolute pathname of your current directory).

To move around the Unix filesystem, you can use the command **`cd`** ('change directory'). This command is the first command that we have come across which can take an argument. An argument to a command is something you put after the command itself, which will either modify its behaviour in some way, or may be the name of a file which the command must execute some process on. If you just type **`cd`** on its own, without any argument, you are brought to your home directory. **`cd ..`** (where '`..`' is an argument) brings you to the directory which contains the directory you are currently in, which is referred to as the 'parent' directory. If you are in your home directory, **`cd bin`** (for example) will bring you to the directory called `bin` in your home directory. However, **`cd /bin`** will bring you to the bin dorectory in the root directory. The current directory is referred to in pathnames by a '`.`' (a period). Files and directories may be referred to both by their absolute pathnames (starting from the root directory), or by their relative pathname, which is the pathname relative to your current directory. The relative pathname can include '`..`' to designate a directory one step higher up in the hierarchy. The relative pathname can also include '~' to indicate the user's home directory.

Naming files

Unix is case-sensitive, so `MyFile.txt` is a different file to `myfile.txt` or `MyFiLe.txt`. Characters used in filenames should only be alphanumeric characters, underscores, dashes and periods. There cannot be more than one file with the same name. Any new file created with the same name as an old one will overwrite the old one.

By convention, most filenames start with a lowercase letter, and end with a period followed by an extension. The extension, usually two or three letters, signifies the type of file it is (cf PC extensions), e.g. `.txt` would be a text file, `.fasta` a file containing sequence(s) in FASTA format, `.html` a HTML document. This convention is often not followed, but it is good practice to do so.

Command line syntax

Unix commands usually have flags and arguments associated with them, which may be optional. Flags modify the behaviour of the command, and are usually single-letter abbreviations which are used with a dash in front of them. Some flags also take arguments. An argument to a command is typically a filename. Arguments to flags tend to be pre-defined words (or numbers) which indicate exactly how the behaviour of that command will be affected.

The names of some Unix commands may appear obscure and arcane at first, but most of them do contain within them some description of what they do.

The syntax for Unix commands has a conventional format. Optional flags and arguments are surrounded by square brackets. Below the name of each command given below, the formal syntax of that command is also indicated. All options are usually not given. To see all options for a command, use the **`man`** command (see below).

Some of the more commonly used commands are:

General commands:

`passwd` (PASSWorD)

`passwd`

Allows the user to change their password. No flags or arguments are used by the normal user.

`pwd` (Print Working Directory)

`pwd`

We have already come across this command, which prints the name of the working directory on the screen. This command takes no flags or arguments at all.

`cd` (Change Directory)

`cd` `[directory]`

Changes the directory which we are currently in. This command does not have any flags associated with it, and the argument to the command (i.e. the directory name) is optional. If we do not give this command an argument, we are taken by default to the home directory. Possible arguments to this command are the absolute or relative pathnames of the directory we wish to go to, which can include '**`..`**' to indicate a higher (or parent) directory.

e.g. **`cd ..`**

`cd ~user/bin`

takes us to the `bin` directory in User's home directory.

`ls` (LiSt)

`ls` `[-adlFh1] [names]`

Lists the directories and files in a directory. As for the **`cd`** command, the arguments to this command are optional. With no argument, all the contents of the current directory are listed. For each directory argument, **`ls`** lists the contents of the directory; for each file argument, **`ls`** repeats the filename and any other information requested (specified by the optional flags). With no flags, **`ls`** just prints the plain name(s) of the contents. The more frequently used flags are **`–a`**, **`-l`**, **`-F`**. **`–a`** (All) lists all entries, including those that begin with a period (**`.`**), which are normally not listed. **`-l`** (Long) prints the list in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file. **`–F`** (Filetype) allows instant visualisation of the type of the contents being listed: it puts a slash (`/`) after each filename if the file is a directory, an asterisk (`*`) if the file is executable, an @ if the file is a link. Another useful flag is the **`–1`** (the number one) flag, which will print each filename on a separate line. The **`-d`** (Directory) flag will list the name of a directory without listing its contents, and is often used to check the status of a directory. The **`-h`** (Human-readable) flag is usually used when listing large files, since it will print out the size in an easy-to-read format. The size will consist of the number of units followed by a unit specifier ('b' for bytes, 'K' for kilobytes, 'M' for megabytes, 'G' for gigabytes, etc.). If two or more of these flags are used together, because none of these flags take an argument, they can be typed together:

e.g. **`ls -alF`**

(Note that there are no spaces between the flags, and only one dash. Also be aware that the argument to the **`ls`** command is put after any flags.)

`mkdir` (MaKe DIRectory)

`mkdir` `[-p] dirname . . .`

Creates a directory. In this case, the argument to the **`mkdir`** command is not optional. This command needs to know the name of the directory it must create. It is possible to create a directory along a branch of the hierarchical tree which has not yet been created. In this case, the **`–p`** flag (Parent) creates all the non-existing parent directories first. The '. . .' at the end of the syntax line indicates that more than one argument can be given to this command.

e.g. **`mkdir –p these/directories/have/not/yet/been/created`**

will create not only the directory '`created`', but all the other directories in this tree which do not already exist.

Viewing file contents

`cat` (conCATenate)

`cat` `file . . .`

Reads each file in sequence and writes it to the standard output.

`more` (MORE)

`more` `[file ...]`

Writes the file specified as an argument to standard output, one screenful at a time. The next screenful is displayed by hitting 'Enter'.

head (HEAD)

head `[ -count ] [ file ...]`

head `[ -n count ] [ file ...]`

Prints a specified number of lines from the specified file to standard output. In this case, there is more than one way to use the **head** command. Both syntaxes gives the same output (**-count** and **–n count** both display only the given number (`count`) of lines specified). If the number of lines is not specified, the default number of lines is 10. This is the first command we've come across which takes a commonly used flag which takes an argument.

e.g. **head –20 file1**

will print out the first 20 lines of the file `file1`. The same result is obtained if we type:

head –n 20 file1

tail (TAIL)

tail `[ -f ] [ -c number | -n number ] [ file ]`

Prints the specified file to the standard output beginning at a designated place. With no flags, **tail** writes the last 10 lines of a file to standard output.

sort (SORT)

sort `[-nr] [-k keydef] [files]`

Sorts the file based on some sort criteria. By default, `sort` sorts the lines in the file into ascending order. The **–r** flag invokes reverse sorting. Each line is divided into fields separated by spaces or tabs (the field separators can be changed using the **–t** flag). By default, all fields are used to sort the file. Specific fields can be used to sort the file, chosen by using the **–k** flag, or field `pos1` to field `pos2`. The **–n** flag sorts line numerically.

e.g. **sort –k2 file.txt**

sorts `file.txt` according to the second field

Manipulating files

`mv` (MoVe)

`mv` `file1 [file2 ...] target`

Moves `file1` to the location at `target`. If `target` already exists, it will be overwritten. This command is often used to rename files.

`cp` (CoPy)

`cp` `file1 [file2 ...] target`

Copies `file1` to the location at `target`, so that there are now two copies of the same file (at `file1` and `target`). If `target` already exists, it is overwritten.

Searching file content

`grep` (Global Regular Expression Pattern)

`grep` `[-c] [-n] [file . . .]`

Searches the specified file(s) for a pattern, and prints out the lines that contain that pattern. The pattern can be a character string, or can be a ‘regular expression’ (explained in more detail later). With the **`–c`** flag, the number of lines that contain the pattern is printed out instead of the lines themselves. The **`-n`** flag prints out the line number on which each match is found.

e.g. **`grep hello myfile`**

will print out all the lines that contain ‘hello’ in the file `myfile`.

Other commands

alias (ALIAS)

alias `[alias_name[=string] ...]`

Defines or displays aliases. An alias is commonly set when a command is used often, or is quite long, and the user-defined alias is a short command which 'points' to the original command. If you use the `alias_name` argument on its own, the original command that the alias is pointing to will be written to standard output, if it is defined. To set an alias, after the alias_name that you want to call your new command, enter the original command, in single quotations marks. If no arguments are given, all the aliases currently available during this session will be written to standard output.

e.g. **`alias h20 'head –n 20'`**

will return 20 lines from the start of a file every time you run the command **`h20`**, instead of the default 10.

unalias (UNALIAS)

unalias `alias-name...`

unalias `-a`

Removes alias definitions. With an alias_name argument, unalias will only remove the alias definition for that alias. With the **`–a`** flag, all aliases currently available in the session will be removed.

echo (ECHO)

echo `[ -n ] [ arg ] ...`

Writes its arguments separated by blanks and terminated by a newline (except when **`-n`** is specified) to the standard output.

e.g. **`echo hello my name is elvis`**

prints `hello my name is elvis` to standard output

wc (Word Count)

wc `[ -c|-m ] [ -lw ] [ names ]`

Counts the number of words, lines or characters in a file, or in standard input if no `names` argument is given, and prints the result to standard output. The most commonly used flag is the **`–l`** flag, which counts the number of lines in a file. If no flags are given, the command defaults to **`wc –lwc`**, which outputs the number of lines (**`-l`**), the number of words (**`-w`**), and the number of bytes (**`-c`**). The number of characters is given by **`-m`**. When multiple files are specified, the names of the files will be printed to standard output along with the specified counts.

e.g. **`wc –l myfile`**

tells you how many lines there are in `myfile`.

`ln` (LiNk)

`ln` `[ -sif ] file1 [file2 ...] target`

Links files. If the user wants to have access to a file which is in another directory, but it may be too large to copy (for disk space reasons, perhaps), he can link the `file1` in the other directory to the `target` file in the present directory. If `target` already exists, the **`ln`** command will fail. The **`-s`** flag creates a symbolic link. A symbolic link is a special kind of file whose contents are the name of another file.

e.g. **`ln -s /usr/bin/databases/large/database_file largefile`**

Now `largefile` in the present directory will point to the `database_file` in the `/usr/bin/databases/large/` directory, so that when the user wants to use `database_file`, he can work with `largefile` instead.

`rm` (ReMove)

`rm` `[-i] file ...`

`rm` `-r [-i] dirname . . . [file . . .]`

Removes files. Use this with caution. Once a file has been removed, you CANNOT retrieve it. For safety, some people alias the **`rm`** command to **`rm –i`**, the **–i** flag requiring confirmation of the removal of every file selected. However, in practice, this tends to be tedious. The **`–r`** flag allows you to remove a directory recursively, i.e. delete all the files in that directory and in any subdirectories that directory might contain, as well as the subdirectories themselves.

`rmdir` (ReMove DIRectory)

`rmdir` `dirname . . .`

Removes a directory. Works as for **`rm –r`** above.

`man` (MANual)

This command prints out the description of a command, its uses, and syntax. Some manual pages can be a little dry.

`apropos` (APROPOS)

`apropos [keyword]`

Prints out the list of commands that include this keyword in their description. This is useful when you don't know what command to use for some particular job.

Useful tidbits

Wildcards (`*, ?`)

An asterisk (`*`) is used to denote a wildcard. As the name suggests, a wildcard is a symbol that can stand for any number of characters (zero or more). The **`?`** wildcard can stand for any one single instance of any character.

e.g. **`ls text*`**

lists all files in the current directory which begin with the string `text`, such as `text`, `text.txt, text3.txt, text2`.

`ls text?`

Will only list `text2`.

Redirection of output (>)

When we run a command, we usually get some sort of output written to standard output. Some commands have an option where we can specify an output file. In cases where this option is not available, and we would like to save the output that appears in the standard output. We can 'redirect' the output of the command to a specified file, using the '**>**' operator.

e.g. **`cat myfile.* > allmyfiles`**

will print out the contents of all files which begin with 'myfile.' (in list order), (e.g. `myfile.txt, myfile.bak, myfile.rubbish`, but not `myfile` itself, because it is not followed by a period, nor will it list `myfile3.txt)`, and save the result into a new file called `newfile`.

Another way that **`cat`** can be used with the redirection operator is:

`cat > newfile`

Press return, then type whatever you want to be put into the file **`newfile`**, and then once you have finished, press return again, and `Ctrl-D`.

Piping (|)

Sometimes we may want to use the output from one command as input for another. In these cases, we can save the output from the first program into a file, and then invoke that file as an argument for the second command, or we can utilize a procedure known as 'piping'. This involves entering the first command, followed by '**|**' followed by the second command. This only works when the output from the first command is directed to standard output, and the second command can take input from standard input.

e.g. **`ls –1 * | wc -l`**

which will list all the files in the current directory, one per line, and then the **`wc –l`** command will count number of lines. In effect, this tells you the number of files in the current directory.

Jobs

Any command that is being processed is called a job. While a command is being processed, it is said to be 'running.'

Commands can take variable amounts of time to run, depending on their complexity, ranging from a few milliseconds to days. While a command is running, the shell that is processing that command cannot be accessed. In effect, what this means is that if you type a command after the prompt, and start it running, the prompt will only reappear (and you will only be able to enter further commands) once that process has been completed. When this happens, there are two ways of getting back a prompt and entering further commands: you can cancel the job, usually by typing Ctrl-C (this may differ according to the way the system is set up), or you can background the job. What this means is that you leave the job running in one shell, but you open another shell on your terminal, where you can now enter further commands. When the job finishes, you will get a notification telling you that the job has finished, and the shell that was running that job is closed.

You can background jobs in one of two ways.

1. If you know that the job might take some time, you can type an ampersand (&) after the command, before hitting 'Enter.' This automatically creates a shell for the job to run in, and gives you a shell in which to enter commands.
2. If the job is taking more time than you thought, you can pause or 'suspend' the job by typing Ctrl-Z, the prompt reappears, and then you can enter 'bg' (BackGround), which opens a new shell for the job to run in the background.

`ps` (Process Status)

`ps` `[options]`

This prints to the standard output information about the current processes associated with the session. This information includes the process ID, terminal identifier, cumulative execution time (how much time the process has taken so far) and the command name.

`jobs` (JOBS status)

`jobs` `[ -l | -p][job_id...]`

Lists and displays the status of currently running jobs in the current shell environment.

`kill` (KILL)

`kill` `[-signal] pid. . .`

`kill` `-signal -pgid. . .`

`kill` `-l`

Terminates a process. The process to be terminated should be identified by its PID (process ID). This can be obtained using either the **`ps`** or the **`jobs`** commands. By default, **`kill`** sends a termination signal which causes the process to end. Some processes protect themselves, so to make sure that a process is killed, use

`kill –9 <pid>`

Permissions

Every file in a Unix system is owned by somebody. Permissions include read, write, execute, and permissions indicate who is allowed to do what with any given file. To see the current permissions of a file, type **`ls –l filename`**. At the beginning of the line describing this file are a series of letters or hyphens, e.g. `–rwxrwxrwx filename` The letters r, w, and x represent ‘read’, ‘write’ and ‘execute’ permissions, respectively. There are three sets of these letters, the first set indicating the permissions that the user himself has for this file, the second group showing the permissions that the group of user that the user belonngs to has, and the third showing the permissions for all other users who have accounts on the system. Permissions can be referred to either symbolically or absolutely. The symbolic representation of the permissions is that shown above, with letters indicating read, write and execute permissions. The absolute representation uses numbers. For every group of three, r has the value of 1, w is 2 and x is 4. Adding these numbers gives the absolute representation of permissions for that group, e.g. `734 filename` means that the user has read, write, execute (1+2+4) permissions, the group has read and write permissions (1+2) and all other users have execute (4) permissions only.

`chmod` (CHange permissions MODe)

`chmod` `[-R] mode file ...`

`chmod` `[-R] [ugoa]{+|-|=}[rwxXstl] file ...`

Changes the permissions mode of a file or directory. Changes can be made using either the symbolic or absolute representations. If symbolic representations are used, the group(s) whose permissions are being changed must be specified (the groups are indicated using letters: u: user; g: group; o; others; a: all, i.e. user, group and others) as well as whether the permissions are being added or removed.

Absolute: **`chmod 741 filename`**

Changes the permissions of `filename` to read, write and execute for the user, execute only for the group, and read only for all others.

Symbolic: **`chmod ug+wx filename`**

Gives the user and group write and execute permission.

Stringing commands together `(;)`

You can have more than one command on the same line, separating them by using ‘**`;`**’. This finishes the first command, then executes the second (and third, etc). You can have as many commands on the same line as you want, separated by semi-colons.

e.g. **`ls –al * | wc –l > myfile; chmod 700 myfile; ls –l myfile`**

This lists all the files in the current directory in long format, counts the number of lines generated this way, and saves the number to a file called `myfile`. We then change the permissions on `myfile`, and do list the long format information on `myfile` to make sure that the permissions are what we want them to be.

Regular expressions (REs)

Regular expressions are patterns that can match a family of character strings. They are used in a number of Unix commands (e.g., **`grep`**, **`egrep`**) and can be very powerful.

Regular expressions are composed of regular characters, and characters that have a special meaning within the context of the regular expression. The special characters are:

1. `.` matches any single character.
2. `*` if this follows a character, matches zero or more occurrences of the character.
3. `^` matches at the beginning of a line only.
4. `$` matches at the end of a line only.
5. `[ ]` matches any one occurrence of the characters enclosed within the brackets. If a hyphen is included, this indicates a range. All of the above special characters lose their special meaning within the brackets. `^` takes on another special meaning within these brackets. If `^` is the first character within the brackets, this indicates that the string should match any character except those in brackets.
6. \ makes all of the above special characters lose their special meaning (including itself). Its special effect is also lost within brackets, as above.

The regular expression will match the longest expression it can in the piece of text that you are looking for the regular expression in.

e.g.,

1. **`l.*g`** means match any patterns that contain an `l`, zero or more characters, followed by `g`. Matches `along, algorithm, log, long, loving, leading`.
2. **`\.`** matches a period
3. **`e.$`** means match a word at the end of a line that ends with an `e` followed by one instance of any other character. Matches ‘`He saw her in the`

tree', 'The man was me.', 'Well met', but not 'Did you know her?'

4. **[^a-m]nd** matches a string with nd, preceded by onr instance of any uppercase letter or lowercase n-z. Matches 'And then I saw her face' 'What a wonderful world it is'

Text processing using **awk**

awk (named after its developers Aho, Weinberger, Kernighan)

awk [-F re] [-v var=value] ['prog'] [file. . .]

awk [-F re] [-v var=value] [-f progfile] [file. . .]

The awk utility is a very useful and highly comprehensive application, almost being a programming language in its own right. The more immediately usable features are described here, but it is recommended that the interested user take a look at some of the extensive manuals and tutorials on this.

awk works by scanning each line in a file and performs certain commands for lines matching specific criteris. Each line is split up into fields, the default field separators being spaces or tabs. The field separators can be user-defined as any regular expression, using the -F flag. Every field in a line canbe specified by its numbered position, i.e., the first field in a line is called by $1. The built-in variable NF is the number of fields on a given line, so the last field in a line is called using $NF. The whole line is referred to by $0. Any action to be performed can either be put on the command line (as all out examples will be) or put into a file, which can then be specified on the command line using the –f flag. The most commonly used command is print.
e.g. **awk '{print NF, $1, $NF}' file.txt > file.out**
scans file.txt, and for every line prints the number of fields that line has, the first field and the last field to a file called file.out. The "action" that the awk command performs is put into single quotes and curly brackets. The commas after the print

statement indicate that a space should be inserted after the number of fields (`NF`) and the first field (`$1`).

If our file contained lines such as:

```
P08100 (human) R98765 (rat) U56567 (elephant)
```

and we wanted to take all the organisms out of the file, and list them all on a separate line, we could use an `awk` command such as:

```
awk –F "[()]" '{print $2"\n" $4 "\n" $6}' ac.txt > 
organism.txt
```

where we use the regular expression `[()]` (enclosed in double quotations marks to escape the metacharacters '`(`' and '`)`') to denote the range of characters that fit our field separators. The organism name is then every second field, which we print out, followed by a newline character ("`\n`"), so that `organism.txt` now looks like:

```
human
rat
elephant
```

We can specify more complicated actions, that include logical and conditional statements.

e.g., **`awk '{if (NF < 10) print $0}' file.txt`**

prints out only those files whose lines have less than 10 fields.

e.g., **`awk '{for (i = NF; i >= 1; i--) print $i}' file.txt`**

prints each line in a file with the fields in reverse order, one field per line.

Text editors

There are two text editors which are widely used: `vi` and `emacs`. Which one you use is a matter of preference. Every Unix system has some version of `vi` installed, and most system administrators install `emacs`. In this course, we will go into some detail in the use of `vi`.

vi (VIsual display editor)

There are three 'modes' in `vi`: a 'surfing' mode, in which you can move around the file and delete text; an 'input' mode, where you can insert new text into the file; and 'command' mode, where you can substitute text, search for text, delete text, access the shell, and save changes to the file.

From 'surfing' mode, you can access either 'input' mode, or 'command' mode. To exit input mode, pressing 'Escape' brings you back to surfing mode. Command mode is only active for one command, after which you are returned to surfing mode.

To open an existing file `myfile` in `vi`, type **`vi myfile`.** The file will be opened in surfing mode. The cursor is initially set at line1, character1 of your text.

<u>Surfing mode.</u>

The movement keys are:

- up: **`k`**
- down: **`j`**
- left: **`h`**
- right: **`l`**
- scroll up a page: **`Ctrl-b`** (Back)
- scroll down a page: **`Ctrl-f`** (Forward)
- last line of the file: **`Shift-g`** (G).
- advance a word: **`w`**

You can put the cursor anywhere in the text in this fashion (a bit like clicking your mouse button at any given position on a word processor like Word).

To move to a certain word in the file, use **`/`** (if the word is further down in the text) or **`?`** (if the word is further up in the text) followed by the word/string/regular expression you want the cursor to be at. If the word appears more than once, you can go to all occurrences of the word by repeatedly entering **`/`** (or **`?`**)**.** There is no need to retype the word you are looking for after the first time, just like the 'Find' and 'Find Again' commands in a word processor.

You can also 'Cut', 'Copy' and 'Paste' text in surfing mode: (x refers to a single character, w to a word, and doubling the command refers to a line):

> **`x`**: deletes the character under the cursor.
> **`4x`**: deletes the character under the cursor and the next three characters.
> **`dd`**: deletes the whole line that the cursor is on.
> **`dw`**: deletes the word that the cursor is in.
> **`5dd`**: deletes the current line, and the next four lines.
> **`Shift-d`** (D) deletes the line from the cursor position to the end.

If a line is deleted like this, it is kept in memory until the next command (any keystroke, except for moving around), like a 'Cut' command in a word processor. To put the deleted line(s) somewhere else in the text, type '**`p`**', and the line is inserted under the line you are currently on (like 'Paste' in a word processor). Uppercase P will insert the deleted line above the current line. If you deleted characters or words instead of lines, then the text will be placed just after your cursor position. To 'Copy' characters, words or lines, use '**`y`**' (yank) instead of '**`d`**'.

Other miscellaneous commands in surfing mode:

`r` (replace) allows you to replace the character your cursor is positioned over (e.g., **`rt`** will replace the current character with `t`).

`Shift-j` (J) joins the line the cursor is on, and the line below it.

`0` moves to the beginning of the line

`$` moves to the end of the line

<u>Input mode:</u>

There are three main ways to insert text:

`a` (append) allows you to start typing after the cursor position;

`i` (insert) allows you to start typing just before the cursor position;

`o` (open line) makes a new line below the line you are currently on, and allows you to start typing at the beginning of that new line.

`Shift-a` (A) allows you to start typing at the end of the current line;

`Shift-i` (I) allows you to start typing at the beginning of the current line;

`Shift-o` (O) opens a new line for typing above the line you are currently on.

To delete a character you have just inserted, most systems will allow you to use the 'Delete' key, but on some systems you may have to use **`Ctrl-H`**.

<u>Command mode:</u>

To enter command mode, type '**`:`**'. The '**`:`**' will appear at the bottom of your editor screen. There are five main commands that are used here:

Moving to a specific line

`:<number>` takes you to that line in the file. You can find out what line the cursor is on using **`Ctrl-G`** in surfing mode.

Inserting another file

`:r <filename>` copies `filename` into the file being edited, under the line the cursor is on.

Deleting lines

`:<line1>,<line2>d` deletes all lines from `line1` to `line2`. To refer to the last line, use **`$,`** e.g. **`:89,$d`** deletes all lines from 89 to the end of the file. To delete from the current line to the end of the file, you can use **`:.,$d`**

Substituting text

`:<line1>,<line2>s/<pattern1>/<pattern2>/[g]` substitutes `pattern2` for `pattern1` in the file between lines `line1` and `line2`. `Pattern1` can be a string or a regular expression. The (optional) '`g`' at the end indicates that this substituution shoud be done globally, which means that if `pattern1` appears more than once on any line, we should replace all occurrences of it on that line with `pattern2`. If we only wanted to change the first occurrence of `pattern1` on any line with `pattern2`, we would leave out the '`g`'. This can also be used for deleting text. If we leave `pattern2` blank, then `pattern1` is replaced with nothing, in effect deleting it.

Saving and exiting from `vi`

`:w` (write) saves the file

`:w <filename>` saves the file to `filename`

`:<line1>,<line2>w <filename>` saves the range of lines indicated to `filename`

`:wq` (write and quit) saves the file and exits `vi`

`:q!` exits `vi` without saving any changes

Typing **`u`** (undo) at any point will undo the last command.

emacs

`emacs` is a popular Unix full-screen editor from Free Software Foundation, installed on most Unix systems. It allows the use of the delete key besides the arrows (unlike `vi`). It also automatically saves the original copy with a different name, so if the changes turn out to be unneeded or unwanted, you can go back to the previous version.

`emacs` is useful because a) lines longer than ~4800 characters are uneditable in `vi`; and b there is a useful utility called `xemacs` which uses menus so that the user does not have to remember the various key-stroke commands.

`emacs` will not be covered in this course. If you wish to learn how to use it, there is a useful tutorial available.

Shell scripts:

Any series of shell commands may be stored inside a regular text file for later execution. A file that contains shell commands is called a shell script. Before you can run a script, you must give it execute permission (**`chmod +x`**). Then to run it, you only need to type its name. Scripts are useful for storing commonly used sequences of commands, and can range in complexity from simple one-liners to complete programs. Any command which can be executed at the Unix prompt can be executed within a shell script.

When a script is run, the kernel determines which shell the script was written for, and then executes the shell using the script as its standard input. The Unix kernel determines which shell the script was written for by examining the first line of the script. In our case, we are only going to be looking at the Bourne Again shell (bash). The first line can either read **`#!/bin/bash`** or there can be no standard first line at all, and the kernel will interpret the script as a bash script. However, to get into good habits, it's a good idea to use `#!/bin/bash` as the first line. Another good practice is to name your bash scripts with the suffix `.bash`.

A simple example of a bash script is:

`example.bash`

```
#!/bin/bash

echo you are
id
echo "The directory you are in is"
pwd
echo "The files in this directory are"
ls
echo "The date is"
date
```

(**`id`** is a command which prints the user's name and ID, and name of the group the user belongs to and that group's ID; **`date`** outputs the current date and time)

This script outputs:

```
prompt> example.bash
You are
uid=1367(luce) gid=1000(physbio)
The directory you are in is
hosts/fulcrum/home/luce/tmp
The files in the directory are
example.sh
The date is
Mon Apr 8 12:00:06 EDT 2002
```

If you wanted to output the results of the commands on the same line as the `echo` statement, you could either use `echo –n` (suppresses the newline), or enclose the command in backwards quotes. The following example illustrates these uses.

```
#!/bin/bash

echo you are `id`
echo –n “The directory you are in is ” <you must put double
quotes around the echo statement when using the –n flag to
ensure that you get a space between the statement and the
result of the subsequent command>
pwd
echo “The files in this directory are ”
ls
echo The date is `date`
```

<u>Variables</u>

Variables are useful tools within shell scripts. Variables are names that have a value that can vary (hence ‘variable’). Variables can be environment variables, or local variables. Some of the more common environmental variables are: $HOME, $PATH, $USER,

$SHELL. Local variables are variables that are created by the user, usually only lasting within a particular session, or within a shell script. The value of a variable can change during the course of the execution of a shell script.

When the value of the variable needs to be used, the variable name is called, preceded by a **`$.`** This tells the shell to replace the variable name with the variable value. In the Bourne shell, we assign a variable like this:

```
variable=value
```

There are NO whitespace characters around the '`=`'. To use that variable, we can then type **`$variable`**. The shell then interprets this as being `value`. If the variable name is immediately followed by other characters that could be interpreted by the shell as being part of the variable name, the variable name itself is enclosed in curly brackets, e.g.

```
verb=paint
echo I like ${verb}ing
```

Output:

```
I like painting
```

But if you tried:

```
echo I like $verbing
```

the output would be:

```
I like
```

because the variable name `verbing` would have no value (unless it had been previously assigned).

You can make a local variable an environmental variable (for that session) by using the export command.

```
export variable
```

makes `variable` an environmental variable. Note that you use the variable name alone (i.e., without the **`$`**).

Reading text from the terminal

The command to read text from the terminal is **`read`**. This command tells the script to wait until the user enters some text from the keyboard, then reads one line from the terminal, one line defined as ending with a newline character, and assigns each word to a variable. Any words that are left over are all assigned to the last variable.

e.g.

```
name.bash
#!/bin/bash
echo enter your name
read name
echo Your name is $name

prompt> name.bash
enter your name
luce
Your name is luce

name2.bash
#!/bin/bash
echo enter your full name
read firstname lastname
echo Your first name is $firstname
echo Your last name is $lastname

prompt> name2.bash
enter your full name
luce skrabanek
Your first name is luce
Your second name is skrabanek
```

Command line arguments

You can also pass information in to a shell script in much the same way as you do with the commands that we have seen before. Just as a Unix command such as `mkdir` takes an argument, so you shell script can take an argument too. Command line arguments are identified by variables whose names indicate their position after the command itself. Thus, $1 refers to the first argument after the command (the value of $0 is the command name itself), $2 is the variable of the second argument, and so on, up to $9. All command line arguments can be referred to by the single variable $*.

e.g.

```
arguments.bash
#!/bin/bash
echo 1 $1
echo 2 $2
echo $3 3
echo $4 4

prompt> arguments.bash have a nice day
1 have
2 a
nice 3
day 4
```

Expression evaluation (**expr**)

This command is used to calculate arithmetic functions. **expr** evalutates expressions which can be mathematical or logical. It also has some string operators. Once it has evaluated the expression, it sends the result to standard output.

e.g. **`expr 1 + 1`**

Each component of the **expr** expression has to be separated by whitespace. All shell metacharacters (`(,),<,>,|,&,;,*`) must be 'escaped' by a back slash (\). Expressions can be grouped using **`\( expr \)`**.

e.g.,

```
expr.bash
#!/bin/bash
echo Enter three numbers
read a b c
echo `expr \($a + $b \) \* $c`

prompt> expr.bash
Enter three numbers
1 2 3
9
```

The numerical operators supported by `expr`, in order of precedence, are:

multiplication (*), division (/), modulus/remainder (%)

addition (+), subtraction (-)

comparison operators (=>, >, =, <=, !=)

logical operators: and (&), or (|)

The numerical operators work on integer values only.

The string operators supported by `expr` are:

`match <string> <regular expression>`, also written as `<string> : <regular expression>`

returns the length of `string` if `string` and `regular expression` match, otherwise 0; e.g., **`echo `expr "fred" : ".*r.*"``** returns 4 (the length of fred).

`substr <string> <start> <length>`

returns the portion of `string` that starts at position `start` and is `length` characters long;

`index <string> <character list>`

returns the integer index of the first character in `string` that matches any of the characters in `character list`;

`length <string>`

returns the length of `string`

Expression testing (**`test`**)

The **`test`** command evaluates various expressions or determines whether they are true or false. There are a number of expressions that can be used, some of which are listed below. They are often used in the conditional loops that are detailed below. Expression tests return integers. When an expression is true, a value of 0 is returned, otherwise it returns a non-zero integer. The spaces used in the list below are important. Only a selection of possible tests is given.

`-l <string>`	Returns the length of `string`
`string1 = string2`	Returns true (0) if `string1` is equal to `string2`
`string1 != string2`	Returns true if `string1` is not equal to `string2`
`integer1 –eq integer2`	Returns true if `integer1` = `integer2`
`integer1 –ne integer2`	Returns true if `integer1` != `integer2`
`integer1 –gt integer2`	Returns true if `integer1` > `integer2`
`integer1 –ge integer2`	Returns true if `integer1` ≥ `integer2`
`integer1 –lt integer2`	Returns true if `integer1` < `integer2`
`integer1 –le integer2`	Returns true if `integer1` ≤ `integer2`
`! expr`	Returns true if `expr` is false
`expr1 –a expr2`	Returns true if both `expr1` and `expr2` are true
`expr1 –o expr2`	Returns true if either `expr1` or `expr2` is true

e.g.

`test 10 –gt 5`

returns true (or 0)

and

`test 10 –le 5`

returns false

Conditional loops (**case...esac**)

The **case** command executes various lists of commands based on the value of a single string. It compares the value of the variable to a predefined list of values. The set of commands it executes are those where the value of the variable and one of the predefined values are the same.

e.g.

```
case.bash
#!/bin/bash
echo Enter your favorite color
read color
case $color in          <compares the value of $color to:>
   "blue"|"green")      <blue and green. If $color is
either blue or green, executes this list of commands>
      echo "That's a lovely color."
      ;;
   "purple")            <$color is compared to purple>
       echo "That's my favorite color too!"
       ;;
   "red")               <$color is compared to red>
       echo "My bedroom is painted that color."
       ;;
   *)                   <all other user inputs>
       echo "Oh, that's strange."
       ;;
esac

prompt> case.bash
Enter your favorite color
purple
That's my favorite color too!
```

```
prompt> case.bash
Enter your favorite color
turquoise
Oh, that's strange.
```

Conditional statements (**if...then...elif...else...fi**)

The **if** command processes a list of commands if some condition is met. This loop structure can become very complex, as you can have any number of conditions, and different sets of commands that must be processed depending on whether the conditions are met or not.

e.g.

```
if1.bash
#!/bin/bash
echo "I'm thinking of a number. Can you guess it?"
read number
if test $number –eq 23
   echo "Well done! You're amazing!"
else
   echo "Nope. That wasn't it."
fi
```

```
prompt> if1.bash
I'm thinking of a number. Can you guess it?
23
Well done! You're amazing!
```

```
prompt> if1.bash
I'm thinking of a number. Can you guess it?
191
Nope. That wasn't it.
```

```
if2.bash
#!/bin/bash
echo Enter two numbers
read first second
if test $first –gt 10
then
   if test $second –gt 10
      echo “You like big numbers.”
   else
      echo “Up and down – I suppose you think you’re
unpredictable.”
   fi
elif test $first –eq 10
then
   if test $second –eq 10
      echo “Looks like you like 10s.”
   else
      echo “Good choices.”
   fi
elif test $second –lt 10
      echo “You like small numbers.”
   else
     echo “You should be a random number generator.”
   fi
fi
```

```
prompt> if2.bash
Enter two numbers
10 34
You should be a random number generator.
```

Iterative loops (**`for...do...done`**)

The **`for`** command allows a list of commands to be executed several times, using a different loop variable in each iteration.

e.g.

```
for.bash
#!/bin/bash
for animal in lion pig dog zebra cat <for every word in
this list, do the list of commands in the do...done loop.
In each successive loop, the variable animal takes on the
value of the next word>
do
   echo I like ${animal}s
done

prompt> for.bash
I like lions
I like pigs
I like dogs
I like zebras
I like cats

rename.bash
#!/bin/bash
ls
for file in `ls –1`
do
   mv $file new$file
done
ls
```

```
prompt> rename.bash
test1.txt      <lists the files in the current directory>
test2.txt
newtest1.txt   <lists the renamed files in the directory>
newtest2.txt
```

<u>Iterative loops (**`until...do...done`**)</u>

The **`until`** command repeats a list of commands until some condition is met.

e.g.

```
until.bash
#!/bin/bash
echo Enter a number
read number
until test $number –gt 6
do
   echo Piggy number $number went to the butcher
   number=`expr $number + 1`
done
```

```
prompt> until.bash
Enter a number
3
Piggy number 3 went to the butcher
Piggy number 4 went to the butcher
Piggy number 5 went to the butcher
Piggy number 6 went to the butcher
```

Iterative loops (**`while...do...done`**)

The **`while`** command repeats a list of commands as long as some condition is met.

e.g.

```
while.bash
#!/bin/bash
x=1
while test $x –le $1
do
   y=1
   while test $y –le $1
   do
     echo –n `expr $x \* $y` "\t"  <inserts a tab>
     y=`expr $y + 1`
   done
   echo                            <prints a new line>
   x=`expr $x + 1`
done
```

```
prompt> while.bash 3
1    2    3
2    4    6
3    6    9
```