UNIX For Beginners — Second Edition

Brian W. Kernighan

Modified for Williams by
T. Murtagh and B. Moore

Additional Modifications for Clemson by
D. House

February 7, 2011

Introduction

The UNIX operating system is easy to use, and presents few of the usual impediments to getting the job done. It is hard, however, for the beginner to know where to start, and how to make the best use of the facilities available. The purpose of this document is to help new users get used to the main ideas of the UNIX system and start making effective use of it quickly. It is intended as an introduction to the system rather than as a reference manual. Where appropriate, the reader is directed to other documents describing various details of the system.

This document has four sections:

1. Getting Started: How to log in, how to type, what to do about mistakes in typing, how to log out.
2. Manipulating Files: A brief introduction to the file system and the commands used to create and manipulate files.
3. More About Files: Directories: Discusses the directory structure used to access files.
4. The Shell: Introduces some helpful features of the UNIX command interpreter.

This document assumes that you are reading it while sitting in front of a unix workstation, and have a unix terminal running on your machine. Performing the "experiments" described in the document as you read it will increase your understanding.

1 Getting Started

1.1 Logging In

You must have a UNIX login name. At Clemson, this will be your standard Clemson userid and your password. If you are on your own system, it will be the userid and password that you created for yourself when you built the system.

Normally, UNIX should type

```
login:
```

When you get a `login:` message, type your userid. Follow it by a `RETURN`; the system will not do anything until you type a `RETURN`. If a password is required, you will be asked for it, and printing will be turned off while you type it. Don't forget `RETURN`.

1.2 Typing Commands

Whenever you type something, it goes to the window the mouse is currently in. System commands are accepted by the "xterm" window on your screen. So before typing any of the commands discussed below,

you must make sure that the mouse is in the "xterm" window on your screen.

One you have moved the mouse into your "xterm", try typing

```
date
```

followed by `RETURN`. You should get back something like

```
Mon Jan  3 14:17:10 EST 1991
```

Don't forget the `RETURN` after the command, or nothing will happen. If you think you're being ignored, type a `RETURN` ; something should happen. `RETURN` won't be mentioned again, but don't forget it — it has to be there at the end of each line.

Another command you might try is `finger`, which displays information about users of the system.

`finger` *some-user's-name*

gives something like

```
Login name: lenhart          In real life: Bill Lenhart
Directory: /home/bullseye/lenhart       Shell: /bin/csh
Last login Fri Sep 23 17:29 on ttyp0 from 192.31.45.228
New mail received Mon Jan  2 16:09:19 1989;
  unread since Mon Jan  2 16:09:24 1989
No Plan.
```

In general, `finger` *some-user's-last-name* will display information about that user, including the user's login name.

If you make a mistake typing a command name, and refer to a non-existent command, you will be told. For example, if you type

```
fnger
```

you will be told

```
fnger: not found
```

Of course, if you inadvertently type the name of some other command, it will run, with more or less mysterious results.

1.3 Mistakes in Typing

If you make a typing mistake, and see it before `RETURN` has been typed, there are two simple ways to recover. The `backspace` key erases the last character typed. Successive uses of backspace remove characters back to the beginning of the line (but not beyond). So if you type badly, you can back up as much as you need.

If you have made such a major typing mistake that it would be easiest to discard the current line and start over, you can do this by typing `Control-c`.

1.4 Stopping a Program

The `control-c` character has another important function. You can stop most programs by typing `control-c`. In a few programs, like the mail program, `control-c` stops whatever the program is doing but leaves you in that program.

If you wish to stop a program temporarily so that you can perform some other task and then restart the stopped program later, type `control-z`. The `jobs` command will display a list of all jobs that have been suspended in this way. The command `fg` can be used to restart suspended jobs. Simply typing

```
fg
```

will restart the most recently suspended job. Alternately, a job number may be included as an argument in the `fg` command to request that a particular suspended command be restarted (each job's number is displayed in square brackets by the `jobs` command).

If you just want a command's output to pause, for example to keep something critical from disappearing off the screen, type `control-s`. The output will stop almost immediately. When you want to let your program's output continue, type `control-q`.

1.5 Read-ahead

UNIX has full read-ahead, which means that you can type as fast as you want, whenever you want (except during login), even when some command is typing at you. If you type during output, your input characters may appear intermixed with the output characters, but they will be stored away and interpreted in the correct order. So you can type several commands one after another without waiting for the first to finish or even begin.

1.6 On-line Manual

The UNIX *Command Reference Manual* is kept on-line. If you get stuck on something, and can't find an expert to assist you, you can display some manual section that might help on your terminal. This is also useful for getting the most up-to-date information on a command. To see a manual section, type "`man` *command-name*". Thus to read up on the `who` command, type

```
man who
```

and, of course,

```
man man
```

tells all about the `man` command. If the description of a command can not fit in one screen, the `man` command will pause after it fills the screen and display the prompt `--More--`. Depressing the space bar will cause the next page to be displayed. Typing the character `q` (for quit) will terminate the display of the manual information.[1]

If your problem is that you don't know the name of a command, the `man` command provides some limited help. Typing "`man-k` *keyword*" causes the system to display short descriptions of all commands whose short descriptions include the specified keyword. Thus,

```
man -k print
```

will produce a list of commands that have something to do with printing.

1.7 Changing Your Password

At Clemson, you set up your initial password at the *CCIT* Help Desk in the Library. After an initial password has been established, all UserIDs will have a default password expiration setting of one year. Users will be notified by email prior to expiration that their password is going to expire. These notifications will go out 14 and 7 days prior to the password expiring. Two more email notifications will go out as well. One will be on the day of expiration and the last notification will go out 7 days after the password has expired.

After one calendar year with the current password, the user will be prompted to change his/her password on the next successful login. The user will be required to create a password that meets the strong password criteria.(See details below.) Also, the user should make the new password significantly different from the previously used password.

The password change utility can be found at https://login.clemson.edu/changepass.php.

Passwords for network accounts:

- Must be strong.
- Must be changed at least every 365 days for all network accounts.
- May contain upper and lower case alphanumeric characters. However, due to mainframe restrictions, the first 8 characters of a password can contain only alpha and numerical characters, and the following special characters: # $. When accessing the mainframe directly, only use the first 8 characters of the password.

A strong password is achieved by a combination of the following factors:

- At least one letter
- At least one number
- At least one special character
- Are a minimum of eight (8) characters.

1.8 Multiple Command Windows

It is frequently convenient to have several *xterm* windows on your screen at the same time. For example, you might want to use one window to run the `man` command to get a description of a command while using another window to try various examples of the command you are reading about. This is easy to arrange.

You can tell the system to create a new *xterm* window using a "pop-up" menu. If you move the mouse

[1] Actually, the `man` command uses a general purpose UNIX utility named `less` to produce such paginated output. You will learn more about `less` later.

into any part of the screen background and depress the right mouse button, the system will display a menu. Select "Open Terminal".

You can use the new *xterm* just like the original one. Each *xterm* is independent of the others on your screen. You can execute different commands simultaneously in different windows.

If your screen gets too crowded you can tell the system to "kill" some of the windows you have created. To do this to your new *xterm*, place the mouse in the window's title bar and press the left mouse button in the *X* box on the right.

1.9 Logging Out

The remaining sections of this document describe many other important aspects of the system. If you have time you should continue reading them and trying the commands suggested now. When you are done or need to leave, you should performing the following steps to terminate your login session.

Depress the left mouse button in the *System* menu on the screen titlebar. Select *Log Out*.

2 Manipulating Files

Information in a UNIX system is stored in *files*, which are much like ordinary office files. Each file has a name, contents, a place to keep it, and some administrative information such as who owns it and how big it is. A file might contain a letter, or a list of names and addresses, or the source statements of a program, or data to be used by a program, or even a program in its executable form or some other nontextual material.

The UNIX file system is organized so you can maintain your own personal files without interfering with files belonging to other people, and keep people from interfering with you too. There are myriad programs that manipulate files, but for now, we will look only at the most frequently used ones.

2.1 Creating Files — The Editor

If you have to type a paper or a letter or a program, how do you get the information stored in the machine? Most of these tasks are done with a *text editor*. There are many text editors available on the UNIX system. A popular editor is `gedit` which is a mouse-based editor, similar to editors found on GUI based systems like Macintosh and Windows. Other popular editors are `vim`, and `emacs`, which is a fully programmable and customizable editor preferred by many computer scientists. We will not give a detailed introduction, but will merely describe how `emacs` can be used to create a very simple text file. There is an online tutorial to help you learn `emacs`. To take the tutorial, simply type the command `emacs` and follow the instructions that appear. **Pay close attention at the beginning of the tutorial.** Otherwise, you won't know how to get out when you want to.

To create a file called "junk" with some text in it using `emacs`, do the following:

1. Move the mouse into your emacs window. Uncover the window if necessary by clicking the mouse in the title bar.

2. Type `control-x` followed by `control-f`.[2] Emacs should now display a prompt on the bottom line of its window asking for a filename.

3. Type `junk` and press `Return`.

4. Type in several lines of text. Type whatever you want. End each line by typing `Return`.[3]

5. Save the file by typing `control-x` followed by `control-s`.

Now create a second file called "`temp`" in the same manner. You should now have two files, "`junk`" and "`temp`".

[2] This is the standard way to read a file into emacs. If the file doesn't exist it will be created.

[3] If you make a mistake, the backspace character can be used to remove erroneous characters.

2.2 What Files Are Out There?

Now go back to your xterm window (you might have to click on the title bar to raise it). The `ls` (for "list") command lists the names (not contents) of any of the files that UNIX knows about. If you type

```
ls
```

the response will be

```
junk          temp
```

which are indeed the two files just created. The names are sorted into alphabetical order automatically, but other variations are possible. For example, the command

```
ls -t
```

causes the files to be listed in the order in which they were last changed, most recent first. The `-l` option gives a "long" listing:

```
ls -l
```

will produce something like

```
-rw-r--r--  1 tom   41 Jul 22 2:56 junk
-rw-r--r--  1 tom   78 Jul 22 2:57 temp
```

The date and time are of the last change to the file. The 41 and 78 are the number of characters. `tom` is the owner of the file, that is, the person who created it. The `-rw-r--r--` tells who has permission to read and write the file.

Options to `ls` can be combined: `ls -lt` gives the same thing as `ls -l`, but sorted into time order. You can also name the files you're interested in and `ls` will list the information about them only. More details can be found in `ls` (1). [4]

The use of optional arguments that begin with a minus sign, like `-t` and `-lt`, is a common convention for UNIX programs. In general, if a program accepts such optional arguments, they precede any filename arguments but may otherwise appear in any order. But, be warned. UNIX programs are capricious in their treatment of multiple options. For example, the `ps` command will accept the options `ua` as in

```
ps -ua
```

but the command

```
ps -u -a
```

produces an error message.

[4] The notation `ls` (1) means the command `ls` is described in section 1 of the UNIX *Command Reference Manual*.

2.3 What's in a Filename

So far we have used filenames without ever saying what's a legal name, so it's time for a couple of rules. Although you can use almost any character in a filename, common sense says you should stick to ones that are visible, and that you should probably avoid characters that might be used with other meanings. We have already seen, for example, that in the `ls` command, `ls -t` means to list in time order. So if you had a file whose name was `-t`, you would have a tough time listing it by name. Besides the minus sign, there are other characters which have special meaning including: \, >, <, |, &, ?, $, [,], and *. Don't try to put spaces in the middle of a filename (substituting underscores or periods is preferable). File names that begin with a period are treated specially. They are not normally displayed by the `ls` command. Such names are intended to be used for files that one normally does not need to be aware of. The `-a` option forces the `ls` command to list all files — including those whose names start with periods. If you type

```
ls -a
```

you will discover that you have more files than you thought.

To avoid pitfalls, you would do well to use only letters, numbers and underscores until you're familiar with the situation. Finally, don't forget that case distinctions matter — `junk`, `Junk` and `JUNK` are three different names.

2.4 Displaying and Printing Files

Now that you've got a file of text, how do you print or display it so people can look at it? There are many programs that do that, probably more than are needed.

One simple way is to use the editor. Go back to your emacs window and type `control-x control-f` (the find file command) and type `junk` at the prompt for a filename. What you typed into the `junk` file will be displayed in the window. Of course, if the file is large, only a small portion of it will be visible on your screen. When you learn more about emacs, you will learn to move about in a file — displaying the parts that interest you.

There are several alternatives to using an editor to display a file. First is `cat`, the simplest of all the programs for displaying files. `cat` simply outputs to the terminal the contents of all the files named in a list. Thus

```
cat junk
```

displays one file, and

```
cat junk temp
```

displays two. The files are simply concatenated (hence the name `cat`) onto the terminal.

`cat` is less than ideal for displaying large files. For example, if you type `cat .bash_profile` the system will display one of the files of commands provided to customize your account when you login. You will have trouble reading the file, however, because it will not all fit on your screen. You could solve this problem by using `control-s` to stop `cat`'s output as described above. The `less` command, however, provides a better solution. If you type

```
less .bash_profile
```

the system will display one screenful of the file on your screen and then wait for input from you before proceeding. Entering a `Return` will cause one more line to be displayed. Entering a space will cause `less` to display the next screenful. Enter a `Q` to exit `less`. In addition, `less` provides a means to advance multiple screenfuls, search for particular strings or examine the file in other ways. See `less` (1) for a complete description.

There are also programs that print files on a printer. The simplest is `lpr`. Typing

```
lpr junk
```

will make UNIX print `junk` on the default printer associated with the machine you are using. Options can be given to the `lpr` command to direct its output to a specific printer. See `lpr` (1) for complete details on using `lpr`.

2.5 Shuffling Files About

Now that you have some files in the file system and some experience in printing them, you can try bigger things. For example, you can move a file from one place to another (which amounts to giving it a new name), like this:

```
mv junk precious
```

This means that what used to be "junk" is now "precious". If you do an `ls` command now, you will get

```
precious     temp
```

Beware that if you move a file to another one that already exists, the already existing contents are lost forever.

If you want to make a copy of a file (that is, to have two versions of something), you can use the `cp` command:

```
cp precious temp1
```

makes a duplicate copy of `precious` in `temp1`.

Finally, when you get tired of creating and moving files, there is a command to remove files from the file system, called `rm`.

```
rm temp temp1
```

will remove both of the files named.

You will get a warning message if one of the named files wasn't there, but otherwise `rm`, like most UNIX commands, does its work silently. There is no prompting or chatter, and error messages are occasionally curt. This terseness is sometimes disconcerting to newcomers, but experienced users find it desirable.

3 Directories

When you first made that file called `junk`, how did the system know that there wasn't another `junk` somewhere else, especially since the person on the computer beside you could also be reading this tutorial? The answer is that generally each user has a personal or *home directory*, which contains only the files that belong to him or her. When you log in, you are "in" your home directory. You may change the directory you are working in — called the *current working directory* — but your home directory is always the same. Unless you take special action, when you create a new file, it is made in your current working directory. Since this is initially your home directory, the file is unrelated to any other file of the same name that might exist in someone else's directory.

A directory can contain other directories as well as ordinary files. An easy way to picture this organization is as a (usually big) tree, with your files located several branches into the tree. It is possible for you to "walk" around this tree, and to find any file in the system, by starting at the root of the tree and walking along the proper set of branches. Conversely, you can start where you are and walk toward the root.

Let's try the latter first. Our basic tool is the command `pwd` ("print working directory"), which prints the name of the directory you are currently in. If you type the command `pwd`, it will print something like

```
/home/tom
```

This says that you are currently in the directory `tom`, which is in the directory `home`, which is in the root directory. The character `/` at the start of a file name identifies the root directory. Other occurrences of `/` in a file name separate components naming directories or files.

If you now type

```
ls /home/your-login-id
```

you should get exactly the same list of file names as you get from a plain `ls`. When no arguments are provided, `ls` lists the contents of the current directory. Given the name of a directory, it lists the contents of that directory.

The next step is to try listing the root of the directory tree. Type

```
ls /
```

You should get a response something like this

```
bin         home        sbin        var
cs-locals   kadb        sys         vmunix
dev         lib         tmp         vmunix.old
etc         mnt         usr
```

Most of the items named are directories. `bin` contains the executable files for many of the basic system commands; as mentioned above, `home` contains user files; `usr` contains more system command executables and local software.

Now try

```
cat /home/your-login-id/junk
```

(if `junk` is still around in your directory). The name

/home/*your-login-id*/junk

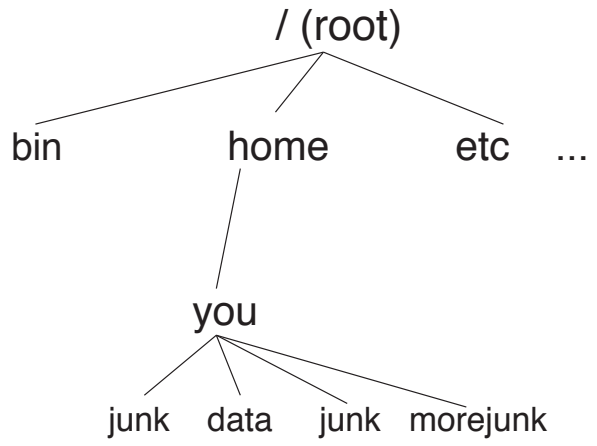
is called the *full pathname* of the file that you normally think of as "`junk`". "Pathname" has an intuitive meaning: it represents the full name of the path you have to follow from the root through the tree of directories to get to a particular file. It is a universal rule in the UNIX system that anywhere you can use an ordinary filename, you can use a full pathname.

.
.
.
.
.
.
.
.
.

Here is a picture which may make this clearer:

This isn't too exciting if all the files of interest are in your own directory, but if you work with someone else or on several projects concurrently, it becomes handy indeed. For example, you can find out what files your neighbor has by saying

```
ls /home/neighbor's-login-id
```



or make your own copy of one of his or her files by typing

```
cp /home/neighbor's id/neighbor's file mycopy
```

If your neighbor doesn't want you poking around in his files, or vice versa, privacy can be arranged. Each file and directory has read-write-execute permissions for the owner, a group, and everyone else, which can be set to control access. See `ls` (1) and `chmod` (1) for details. As a matter of observed fact, most users most of the time find openness of more benefit than privacy. You are expected, however, to follow the Honor Code guidelines with respect to looking in other people's directories. While working on a project, it might be appropriate to deny others access to it.

As a final experiment with pathnames, try

```
ls /bin
```

If you look carefully, some of the names will look familiar. When you run a program by typing its name, such as `ls` or `cat`, after the prompt character, the system simply looks for a file of that name. It normally looks first in your directory, then in a number of special system directories including `/bin`. There is nothing magic about commands like `ls` or `cat`, except that they have been collected into a couple of places to be easy to find and administer.

What if you work regularly with someone else on common information in his or her directory? How can you say "I want to work on his files instead of my own". This is done by changing the directory that you are currently in:

```
cd /home/your-friend's-login
```

Now when you use a filename in something like `cat` or `pr`, it refers to the file in your friend's directory. Changing directories doesn't affect any permissions associated with a file — if you couldn't access a file from your own directory, changing to another directory won't alter that fact. Of course, if you forget what directory you're in, type

```
pwd
```

to find out.

The `~` (tilde) is an abbreviation for accessing home directories. To change to your home directory, for example you can `cd ~`. You can see a file in your home directory with `cat ~/filename`. You can access someone else's home directory with a `~` followed by their login name. For example, `ls ~tom` will list the files in Tom's home directory.

It is usually convenient to arrange your own files so that all the files related to one thing are in a directory separate from other projects. For example, when you take Computer Science 237, you might want to keep all your assignments in a directory called `cs237`. So make one with

```
mkdir cs237
```

then go to it with

```
cd cs237
```

If you then put a file `program.1` in this directory, it is located in

```
/home/your-login-id/cs237/program.1
```

To remove the directory `cs237`, remove all the files in the directory (using `rm`) and then leave the directory with

```
cd ..
```

This will change to the directory above the one you are currently in. (In this case it will return you to your home directory.) Then type

```
rmdir cs237
```

Alternately, you can remove the directory and all its contents by going to your home directory and typing the single command

```
rm -r cs237
```

The `-r` option instructs `rm` to recursively remove a directory and its contents (including sub-directories). Needless to say, this is a potentially dangerous option which should be used with great care.

You can always go up one level in the tree of files by saying

```
cd ..
```

"`..`" is the name of the parent of whatever directory you are currently in. For completeness, "`.`" is an alternate name for the directory you are in. Finally, typing `cd` with no file name will get you back to your home directory.

4 The Shell

When the system prints the prompt and you type commands that get executed, it is not the UNIX kernel that is talking to you but a go-between called the *shell* or command interpreter. The shell is just an ordinary program like `date` or `emacs`. In fact, there are several shells available that you can choose from. On our system, most accounts are initialized to use a program named `bash` (for Bourne Again SHell) as a shell. Other, older UNIX shells are called `sh` and `csh`.

In addition to allowing you to execute simple commands, the shells can assist you in other ways. In particular, they support:

1. Filename shorthands: Instead of typing a long list of file names as arguments to a command, you can specify a pattern for the file names in the command — the shell will find all files that match the pattern and substitute them for the pattern in the command.

2. Input-output redirection: You can arrange for the output of any program to go into a file instead of onto the terminal, and for the input to come from a file instead of from the terminal. Furthermore, you can even arrange to have the output of one program sent as input to another program.

3. Personalizing your environment: You can define your own commands and shorthands.

In addition, bash provides a history mechanism that allows you to re-use all or parts of the text of previous commands, filename completion, and many other features.

4.1 Filename Shorthands

Suppose you're typing a large document like a book. Logically this divides into many small pieces, like chapters and perhaps sections. Thus you should type the document as a number of files. You might have a separate file for each chapter, called

```
chap1
chap2
etc...
```

Or, if each chapter were broken into several files, you might have

```
chap1.1
chap1.2
chap1.3
etc...
chap2.1
chap2.2
etc...
```

You can now tell at a glance where a particular file fits into the whole.

There are advantages to a systematic naming convention which are not obvious to the novice UNIX user. What if you wanted to print the whole book? You could say

```
lpr chap1.1 chap1.2 chap1.3 ......
```

but you would get tired pretty fast, and would probably even make mistakes. Fortunately, there is a shortcut. You can say

```
lpr chap*
```

The * means "anything at all," so this translates into "print all files whose names begin with chap", listed in alphabetical order.

This shorthand notation is not a property of the pr command. It is a service of the shell that can be used in any command. Therefore, you can list the names of the files in the book by typing:

```
ls chap*
```

The * is not limited to the last position in a filename — it can be anywhere and can occur several times. Thus

```
rm *junk* *temp*
```

removes all files that contain junk or temp as any part of their name. As a special case, * by itself matches every filename, so

```
rm *
```

removes **all** files. (You had better be *very* sure that's what you wanted to say!)

The * is not the only pattern-matching feature available. Suppose you want to print only chapters 1 through 4 and 9. Then you can say

```
lpr chap[12349]*
```

The [...] means to match any of the characters inside the brackets. A range of consecutive letters or digits can be abbreviated, so you can also do this with

```
lpr chap[1-49]*
```

Letters can also be used within brackets: [a-z] matches any character in the range a through z.

The ? pattern matches any single character, so

```
ls ?
```

lists all files which have single-character names, and

```
ls -l chap?.1
```

lists information about the first file of each chapter (chap1.1, chap2.1, etc.).

Of these niceties, * is certainly the most useful, and you should get used to it. The others are frills, but worth knowing.

4.2 Input-output Redirection

Most of the commands we have seen so far produce output on the terminal; some also take their input from the terminal. It is universal in UNIX systems that the terminal can be replaced by a file for either or both input and output. As one example,

```
ls
```

makes a list of files on your terminal. But if you say

```
ls > filelist
```

a list of your files will be placed in the file filelist (which will be created if it doesn't already exist, or overwritten if it does). The symbol > means "put the output in the following file, rather than on the terminal." Nothing is produced on the terminal.

As another example, you can combine several files into one by capturing the output of cat in a file:

```
cat f1 f2 f3 > temp
```

This is in fact where cat's name comes from — it can be used to concatenate several files into one.

The symbol >> operates very much like > does, except that it means "add to the end of." That is,

```
cat f1 f2 f3 >>temp
```

means to concatenate f1, f2 and f3 to the end of whatever is already in temp, instead of overwriting the existing contents. As with >, if temp doesn't exist, it will be created for you.

In a similar way, the symbol < means to take the input for a program from the following file, instead of from the terminal. Thus, to send a letter to several users, you can use emacs to prepare the text of the letter in a file let, then send it to several people by typing the commands

```
mail adam <let
mail eve  < let
mail mary< let
```

(In fact, mail accepts several addresses on one line so you could also type

```
mail adam eve mary < let
```

).

4.3 Pipes

One of the novel contributions of the UNIX system is the idea of a *pipe*. A pipe is simply a way to connect the output of one program to the input of another program, so the two run as a sequence of processes — a pipeline.

For example, suppose that you want to print a list of all the files in your directory on the printer. You could direct the output of `ls` to a file and then print the file using the commands

```
ls > filelist
lpr filelist
rm filelist
```

What you really want, however, is to have the output of `ls` given as input to `lpr`. This can be done by using a pipe as in the single command

```
ls | lpr
```

which has the same effect as the three commands shown above. The vertical bar (`|`) means to take the output from `ls`, which would normally have gone to the terminal, and provide it to `lpr` as input.

There are many other examples of pipes. The program `wc` counts the number of lines, words and characters in its input. Thus

```
ls | wc
```

tells how many files are in your directory.

Any program that reads from the terminal can read from a pipe instead; any program that writes on the terminal can drive a pipe. Many UNIX programs are written so that they will take their input from one or more files if file arguments are given; if no arguments are given they will read from the terminal, and thus can be used in pipelines.

You can have as many elements in a pipeline as you wish. Thus,

```
ls | wc | lpr
```

will print a count of the number of files you have on a printer.

4.4 Processes

The shell is capable of running more than one command for you at the same time. This occurs when you use the pipe symbol. Rather than waiting for the first command in a pipeline to complete before starting the second, the shell lets all the programs whose names you include in a pipeline run at the same time (although obviously some programs may need to wait for earlier programs to produce the input they need).

There is another way you can specify the simultaneous execution of several commands. For example, if you are doing something time-consuming, like sorting a large list of names, and you don't want to wait around for the results before starting something else, you can say

```
sort namelist > sortednames &
```

(`sort` is the name of the UNIX sort utility — see `sort` (1) for more details). The ampersand at the end of a command line says "start this command running, then take further commands from the terminal immediately," that is, don't wait for it to complete. Thus the sort program will begin, but you can do something else at the same time. Commands started in this way are said to run in background. If you do not redirect the output of a command ended with an `&`, the output may interfere with the work you are doing while the command executes.

When you initiate a command with `&`, the system replies with two numbers called the job number and process number, which identify the command in case you later want to stop it. The first number (surrounded by square brackets) is the job number; the second number is the process number. If you want to stop the command, you can say

```
kill process-number
```

or

```
kill %job-number
```

If you forget the process number, the command `ps` will tell you about everything you have running. The numbers in the commands output under the heading `PID` are the process numbers. If you forget the job

number, the `jobs` command will list all outstanding jobs.

Another alternative provided by the shell is the ability to specify the sequential execution of several commands in one line. You can run two programs with one line by separating the commands with a semicolon; the shell recognizes the semicolon and breaks the line into two commands. Thus

```
date; ls
```

executes the `date` command and then the `ls` command before returning with a prompt character.

You can say

```
( command-1; command-2 ; command-3) &
```

to start three commands in the background. In this case, the shell will not start command i until command $i-1$ has completed. Alternately, you can start a background pipeline with

```
command-1 | command-2 &
```

Just as you can tell `mail` or some similar program to take its input from a file instead of from the terminal, you can tell the shell to read a file to get commands. (Why not? The shell, after all, is just a program, albeit a clever one.) For instance, suppose you want to set tabs on your terminal, and find out the date and who's on the system every time you log in. Then you can put the three necessary commands (`tabs`, `date`, `who`) into a file, let's call it `startup`, and then run it with

```
bash startup
```

This says to run the shell with the file `startup` as input. The effect is as if you had typed the contents of `startup` on the terminal.

If this is to be a regular thing, you can eliminate the need to type `bash`: simply type, once only, the command

```
chmod +x startup
```

and thereafter you need only say

```
startup
```

to run the sequence of commands. The `chmod` command marks the file executable; the shell recognizes this and runs it as a sequence of commands. (See `chmod` (1) for details.)

If you want `startup` to run automatically every time you log in, there is a file in your home directory called `.bash_profile`, Place a line containing the word `startup` in this file. When the shell first gains control when you log in, it looks for the `.bash_profile` file and does whatever commands it finds in it.

Every time you start a new copy of the shell (which happens when you log in and each time you create a new window), the shell processes the commands in another file named `.bashrc`.[5] Even if you don't want to change them, looking at the `.bash_profile` and `.bashrc` files can be an educational (but potentially confusing) experience.

4.5 The History Mechanism[6]

bash stores the commands you type and provides access to them through its *history mechanism*. This makes it possible to repeat a previous command, include portions of a previous command in the current command or re-execute a previous command after fixing a typing error.

The list of all commands currently saved by the history mechanism is displayed by executing the `history` command. The `history` commands numbers the old commands it displays. Typing

```
!n
```

will re-execute the command numbered n.

If you would rather not lookup the number of a previous command, you can type `!` followed by the first characters of the command. For example,

```
!ls
```

will execute the last command that began with ls.

Another way to avoid looking up the number is to use `control-p`. Pressing `control-p` will display

[5] As the files name suggests, only the `bash` shell performs the actions described in this paragraph.

[6] Everything in this section applies only to `bash and tcsh`.

your previous command, which you can then edit. Repeatedly pressing `control-p` will go back to earlier commands.

If the command you want to execute is similar to a previous command, or you want to fix a typing mistake, you can `control-p` back to the command, then edit it using `control-b` to move the cursor back a character, `control-f` to move forward a character, `control-a` to move to the left edge of the line, and `control-d` to delete a character. `Backspace` and `delete` work as usual. Anything you type is inserted where the cursor is located.

Another useful function the history mechanism provides is the ability to re-use the arguments of the previous command. The characters `!*` occurring in a command are replaced by the arguments to the previous command. Thus, one can first check the contents of a file by typing

```
less somelongfilename
```

and then print the file by typing

```
lpr !*
```

You can avoid typing somelongfilename by just typing enough of it so that the shell can recognize it, (for example "somel" for the filename "somelongfilename") and pressing tab. bash will type the rest of the filename for you.

Finally, you can fix simple typing mistakes in a command by typing

^*old-string*^*new-string*

This causes the shell to re-execute the last command after first replacing the first occurrence of *old-string* by *new-string*.