Unix Users Introduction Course

Mark O. Stitson

September 29, 1997

Royal Holloway
University of London

Department of Computer Science
Egham, Surrey TW20 0EX, England

Contents

Introduction

I wrote this booklet for a Basic UNIX Users Course. It was not originally intended for self study, but can be used as such. At the end of each section there is a set of exercises. The exercises expect the computer being used to be connected to the Internet, or at least a local area network.

While writing this booklet I was working on a range of different systems and my experience is drawn from all of them. They included Sun SparcStations, HP Workstations, DEC MIPS Workstations, DEC Alpha Servers and PCs running UNIX, DEC VAXes and Alphas running VMS, PCs running DOS and Windows and Apple Macintosh Computers.

All systems have their advantages and disadvantages. This booklet does not try to justify the use of UNIX, the choice of operating system is mainly based on the availability of software needed for the given task.

For those though who want to try UNIX, there are free versions available for most computers including Acorn, Atari, Amiga, Alpha, MIPS, Sparc and the PC.

Writing this course would have been impossible without the help of the systems staff, Jon, Will and Mark, of the Department of Computer Science at Royal Holloway and Laura, Mark and Barry who proof read this booklet countless times to find all my mistakes.

January 1996

The Author

Chapter 1

Why use an Operating System?

1.1 Introduction

Computers without software are like cars without petrol, they do not run. Software can be divided into two groups. The system programs which manage the operation of the computer, and application programs, which do the work for the users. Most users of modern computers will hardly ever use any of the system programs, the most basic of which is the operating system. Everything which involves direct knowledge of the underlying system can be carefully packaged so that in some cases it is impossible to tell which type of computer system is being used. This however has not always been the case.

The first digital computer ever to be designed was the analytical engine by the English mathematician Charles Babbage (1792-1871), this machine was completely mechanical and had to be programmed by changing the way the gear wheels were set up initially. Babbage never got his analytical engine working, although he spent his life and the equivalent cost of 40 steam engines on the project. This machine did not have a operating system.

1.2 The First Generation of Electronic Computers (1930-1955): Vacuum Tubes and Plugboards

After Babbage's unsuccessful attempts little progress was made until the 1930s when Zuse constructed his first computer in his parents living room. He built the first universal binary computer, the Z3, in 1941. Whereas Zuse had constructed his Z3 alone, the first large universal computer ENIAC was constructed by a 50 man team at the University of Pennsylvania and completed in 1945. This computer had to be programmed by wiring up control panels. In the 1950s the wired up control panels were replaced by punched cards. Yet computers were often designed, constructed, programmed, operated and maintained by the same people and operating systems were unheard of.

1.3 The Second Generation of Computers (1955-1965): Transistors and Batch systems

The introduction of transistors in the mid-1950s made it possible to build smaller and more reliable computers. This made it possible to sell computers commercially, separating the designers and builders of the computer from the operators, programmers and maintainers. As computers were so expensive only specially trained operators were allowed into the special, air-conditioned rooms which contained the delicate equipment. Programmers had to write their programs in machine language or FORTRAN[1], then transfer them to punched cards and hand them to the operators to get them processed. Later they would receive back the results. The operators would load the stack of punched cards into the computer and collect the printed output when the calculations had finished. A lot of time was wasted running around with decks of cards or printouts, and ways were found to minimise the inefficient use of the computers.

Simple computers were used to transfer the punched cards to magnetic tape. The tape with several programs was then mounted into the main computer. The operator loaded a special program (the ancestor of today's operating systems) which read jobs from tape and ran them. The output of all programs on the tape was written to a special output tape instead of being printed. When all jobs were finished the operator removed the output tape and took it to a smaller simple computer which printed the contents of the output tape.

1.4 The Third Generation of Computers(1965-1980): ICs and Multiprogramming

The early 1960s saw computer manufacturers maintaining two distinct product lines. On the one hand were the highly mathematical, scientific computers and on the other side were the character-oriented, commercial computers. It was expensive maintaining two product lines and hard for customers to upgrade to more powerful computers, because they had to have all their programs rewritten. This caused IBM to develop a range of computers that could satisfy the needs of all its customers. The IBM System/360 series of computers all

[1]The first high-level computer language, specially designed for mathematical operations.

could run the same software and had the same operating system. It is still in widespread use today by companies who cannot afford to lose computing time by changing their system[2]. To make the operating system good for all sizes of machine, it had to be incredibly complex and needless to say was full of bugs.

The operating system OS/360 introduced many important features of modern operating systems. The most important of which was probably multiprogramming, often mistakenly called multitasking. This allowed the system to run another program, while the first program was waiting for some I/O to complete. For this to work memory was partitioned into several pieces and special hardware was needed to protect other programs from misbehaving programs.

Multiprogramming made possible a second feature which enabled punched cards to be directly read onto disk by the computer while it was processing other programs. These systems were still essentially batch systems, which had a very slow response time. This lead to the development of a system which allowed time-sharing of the CPU , a variant of multiprogramming, in which every user has a on-line terminal on his desk and can use the system whenever he wants to[3]. This lead to the development of MULTICS (MULTiplexed Information and Computing Service), a very powerful operating system, which laid down much operating systems theory, but was too big and complex to become widespread.

Another major development during this phase was the proliferation of minicomputers and the invention of the microprocessor by Intel in 1971. One of the members of the MULTICS team Ken Thompson wrote a stripped down, single user version of MULTICS for a small minicomputer PDP-7 , that no one was using at Bell Labs. Brian Kernighan jokingly called this system UNICS (UNiplexed Information and Computing Service). The system worked so well that Bell Labs invested in a larger minicomputer and Thompson teamed up with Dennis Ritchie, to rewrite the operating system in C, which had been designed by Kernighan and Ritchie. This new operating system was called UNIX and allowed multiple users time-sharing. Bell licensed it to Universities almost for free. Bell gave away the source code to be studied by all licensees, which lead to UNIX being moved or ported to more types of computer than any other operating system.

[2]British airways still use IBM360s for their booking system.

[3]"He" should be read as "he or she" throughout the book.

1.5 The Fourth Generation (from 1980): Personal Computers

The introduction of large scale integrated circuits brought computers onto peoples desks. Minicomputers had enabled departments to buy their own computers, but Personal Computers gave every individual computing power. This drastically changed the way operating systems had to be designed. Instead of the highly trained specialist needed to operate an IBM 360, potentially everybody could now use computers directly. Two operating systems from the early part of this period still exist. MS-DOS which IBM licensed from Microsoft was a operating system containing only the very basic features of a modern operating system. The operating system of the Apple Macintosh on the other hand was the successful attempt to create a user-friendly system which could be operated easily with a mouse using a graphical user interface. Due to its initial high price it has for a long time been relegated to niche applications like graphical design, publishing and advertising.

MS-DOS and UNIX are now trying to catch up with Apple's head start on the graphical user interface. Microsoft has produced Windows 95 which come close to the friendliness of Apple's operating system. Whereas UNIX based workstation manufacturers like Hewlett Packard and IBM with their UNIX variants are trying to catch up by providing graphical system administration tools.

This generation of computers also saw the growth of computer networks which allowed sharing of resources like disks and printers in the simple systems to the sharing of processing power in the advanced systems.

UNIX systems allow users on other computers to use them remotely, whereas MS-DOS and Apple systems can be only used if the user is present directly at the computer.

This also defines the applications UNIX is used for. Whereas MS-DOS and Apple systems are useful for word-processing and spreadsheets, UNIX computers are used for massive database operations and large calculations.

In this context it is important to mention two further high power systems. Digital's VMS is a very secure system originally developed for the VAX series of computers. Microsoft's Windows NT (which has only the user interface in common with Microsoft's Windows 95 or Windows 3.x) is another very secure system, but limited to about 4 processors per machine. The VMS system has a large user base due to its age, whereas the Windows NT system is not so widespread due to its limitations on remote usage and processing power.

Digital which sell all three systems for its Alpha range of machines sells about 60% UNIX systems, 30% VMS systems and 10% Windows NT systems. In this range of high power machines the comparative user-unfriendliness of UNIX is weighed up by the fact that only a small number of experts work with these systems, whereas most users nowadays have desktop computers with Microsoft's Windows or Apple's Macintosh operating system.

Chapter 2

What are the Functions and Components of a Operating System

An operating system has two functions. It acts as a resource manager on the one side and a virtual machine on the other side. The resource manager of the operating system manages allocation of memory, processing time and disk space. The virtual machine should provide a nice user interface, other than the bare hardware would provide.

UNIX for instance has a resource manager, which allocates memory and processing time in a very efficient way. However the virtual machine leaves a lot to be desired on most UNIX systems. The Apple Macintosh in contrast provides a very nice virtual machine with its graphical user interface, but lacks features like protecting programs from each other and fair distribution of processor time. MS-DOS unfortunately provides neither a powerful resource manager nor a friendly virtual machine.

An operating system provides two main services: Filemanagement and Processmanagement.

2.1 Files

Files are used to store data and directories are a way of organising files in a hierarchical structure. What UNIX does differently from DOS and other systems is the way it organises different physical storage devices.

In DOS there are individual drive names (actually letters from A to Z) denoting the different drives. In UNIX there is essentially just one virtual drive. All physical drives are sub-directories of this virtual drive or other drives.

Example: In DOS the first floppy disk drive is always A: and the first hard disk is always C:. When copying a file from the floppy disk directory flop_dir to the hard disk directory hard_dir, we issue the following instruction:

```
copy A:\flop_dir\file C:\hard_dir
```

In UNIX we first have to know where the floppy disk drive and the hard disk drive are attached (mounted) to the system. Say the floppy disk drive is mounted onto the directory floppy, which in turn is a sub-directory of the directory removable-disks. The hard disk is mounted onto the directory hard-disk, which is a sub-directory of the directory fixed-disks. The UNIX copy command now would look like this[1]:

```
cp /removable-disk/floppy/file /fixed-disk/hard-disk
```

The next difference to DOS and other systems is UNIX's handling of other physical devices like the keyboard, screen and printer. Every device is represented as a special file. This makes it possible to copy a normal file to the printer special file. This copying is interpreted by the operating system as an instruction to print the contents of th normal file. Similarly it is possible to copy data from a serial port special file into a normal file, thus reading the serial ports input.

A third important feature of files in UNIX is that every file has an owner, identified by a user identification (UID). The owner of a file can specify the following attributes for a file: read access allowed, write access allowed and execution allowed. These attributes can be separately specified for the owner of the file, a special group of users or everybody. This means a user can set up a file which contains a program to be readable and executable for himself, while others can only execute it, but not read or copy it.

The last feature we will discuss in this section are pipes. These are pseudo-files which can be used to connect processes. The first process redirects its output into a pipe and the second process reads its input from the pipe.

Example: We want a directory listed in alphabetical order.

To do this we can connect the directory command to the sorting command using a pipe. This will produce a sorted directory.

2.2 Processes

A process is basically a program in execution. It consists of a program, its data, its ownership information (UID), the current position at which the computer

[1] The / indicates the top level or root directory as the \ does in DOS.

is in the programs execution and all the other information needed to run the program. The UNIX operating system will occasionally decide to halt a process running and run another process. This gives every process its fair share of processing time. A process is also halted if it is waiting for some I/O operation to finish, like accessing a printer.

When a process is halted the operating system has to store all the information about the processes current state, so that the process can later be restarted exactly where it left off.

Processes can start other processes. These child processes have exactly the same ownership and permissions as the parent processes. Parents can control their children and send them signals. The operating system can also send signals to processes. Their are a wide variety of signals, and most programs react to signals by dying. There are a number of programs though, which are mostly operating system related and take more intelligent action towards signals sent to them.

2.3 Shells

The strict definition of an operating system limits it to only the basic program which runs the computer and provides systems calls to the users' programs. Usually the command interpreter or shell is not considered part of the operating system. But without the shell, which executes users' commands and programs, and is the primary interface between the computer and the user, a computer would be virtually impossible to use.

The basic function of a shell is to allow users to execute programs. Many advanced shells though have features which allow writing of scripts, command-line completion and command-line histories.

Chapter 3

Users & Passwords

The first program encountered when using a UNIX system is the program that checks the user's identity. It first asks the user his username which uniquely identifies the user and then asks the user for his password. The password is compared to the password stored in the system's password file and if it has been entered correctly the user's chosen shell is started.

Each user has a unique user identification (UID) and can be a member of many groups, each identified by a group identification (GID). This determines which files he can access and what privileges he is allowed. Some programs, especially system programs can only be executed by members of a certain group. Very critical programs can only be executed by the super-user, who is the most powerful user in a UNIX system. The super-user is allowed to do everything and can access all files. He always has the UID 0.

A user can change his password at any time. This feature should be used regularly. Even if a secure password has been chosen it is possible to break it and gain illegal access by brute force code breaking within about eight weeks.

As a former systems administrator of this College used to put it: “Passwords are like toothbrushes: You change them frequently and you don't share them.”

To make passwords secure they should not be names or words in a dictionary. Many system administrators will, for the sake of security, occasionally try and break the users passwords by testing them against the spelling dictionary. To make a password secure it should also be something which is not easy to guess. A user's telephone number, registration plate or date of birth are favourite things crackers try first when trying to break into a system. A good password should contain a number or a capital letter in an unusual place. Good passwords are for instance: ”comp5uter” or ”queRy”. Foreign words are also effective as they will be very hard to break using a standard dictionary.

3.1 Changing your password

To change your password type passwd and return. You will then be prompted for your old password, to check it is you and not somebody strolling past your computer. Then you will be prompted for your new password twice. To ensure security, the password you type in is not displayed on the screen. Typing the new password twice ensures you do not mistype.

Chapter 4

Files & Directories

Files and directories are used to store data like a filing cabinet. Files directly store data, whereas directories are used to store files or other directories in a hierarchical structure.

4.1 Directory Contents

Directories store names of files and a reference to the actual file or names of directories and a reference to the other directory. The ls[1] command is used to display the contents of a directory. Given the example in figure 4.1 the ls command will produce the following output:

```
$ls
a         b         c          data      junk      programs
$
```

This gives us just the names of the files and directories. To get a better idea of what are files and what are directories we can use an additional parameter to the ls command:

```
$ls -F
a          b          c           data/      junk/      programs/
$
```

[1] LiSt

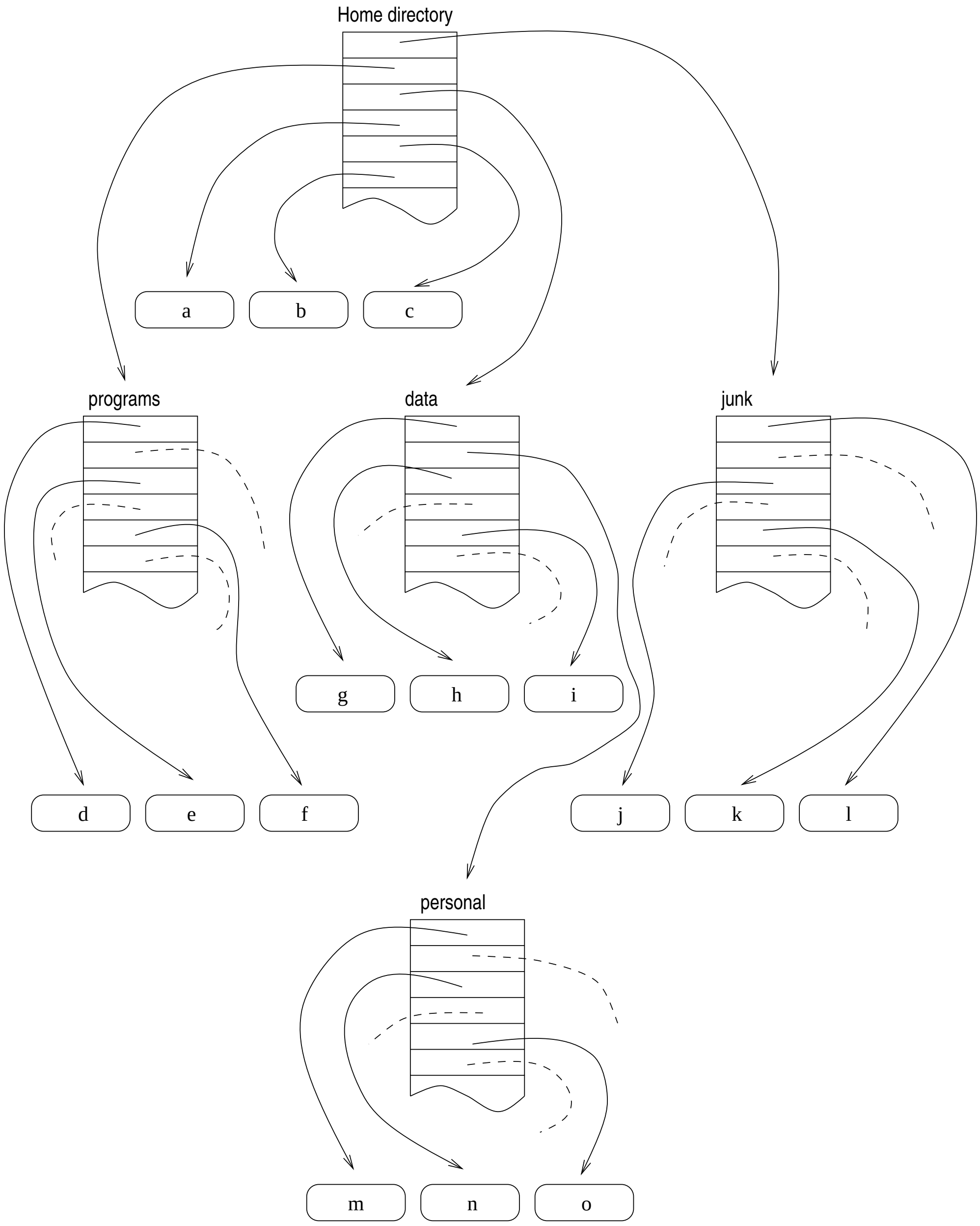


Figure 4.1 A Sample directory sub-tree

We can now see that the subdirectories have all been appended with a '/'. To get even more information we can add another parameter:

```
$ls -Fl
total 3
-rw-------   1 markst   staff           0 Jan 21 17:13 a
-rw-------   1 markst   staff           0 Jan 21 17:07 b
-rw-------   1 markst   staff           0 Jan 21 17:07 c
```

```
drwx------   3 markst   staff        1024 Jan 21 17:11 data/
drwx------   2 markst   staff        1024 Jan 21 17:11 junk/
drwx------   2 markst   staff        1024 Jan 21 17:10 programs/
$
```

This is the most informative directory display available on most UNIX systems. The first column contains the file type and access permissions and type of directory entry. The second column specifies he number of hard links, which will be explained later. The third and fourth column specifies the owner and group. The fifth column shows the file size. The sixth column shows the last modification date and time. The last column finally shows the name of the directory entry.

4.2 How to view other Directories

UNIX has the notion of a current directory. This is the directory the user is currently using. To change the current directory we use the cd[2] command.

Assuming the current directory is the same as above, we can change to the data directory by typing the following:

```
$ls -F
a          b          c          data/      junk/      programs/
$cd data
$ls  -F
g          h          i          personal/
$
```

There are three special directory names in UNIX. The current directory is just called '.', the next directory up the directory tree is called '..' and the user's home directory is called ' ˜ '. Here are three examples:

```
$cd .
$ls -F
g          h          i          personal/
$cd  ..
$ls -F
a          b          c          data/      junk/      programs/
$cd ~
```

[2] Change Directory

```
$ls -F
a          b          c          data/      junk/      programs/
```

To find out what our current directory is we can use the pwd[3] command:

```
$pwd
/home/ugrads/unix_course
```

We can now see that our current directory is called 'unix_course'. It is a subdirectory of 'ugrads', which in turn is a subdirectory of 'home', which is a subdirectory of the root directory '/'.

To get to the directory home we could either use two steps:

```
$cd ..
$cd ..
$ls -F
ftp/     local/   test/    ugrads/
$
```

Alternatively we could use an absolute path name:

```
$cd /home
$ls -F
ftp/     local/   test/    ugrads/
$
```

A third option is to tell the shell to go to the next directory up and then up again in one step:

```
$cd ../..
$ls -F
ftp/     local/   test/    ugrads/
$
```

There are two types of path name being used here. The first method is relative to the current directory. This is a relative path name. The second method is relative to the root directory or absolute. This is a absolute path name.

[3] Path of Working Directory

4.3 Running a program

To run a program it is generally sufficient to type its file name. Going back to our home directory and the to the program directory, we find three files, one of which is annotated by a '*'. This usually means a file is a program. In contrast to the other files it has an x in its access permissions. This stands for eXecutable. We can run it by typing f and return.

```
$cd ~
$cd programs/
$ls -F
d   e   f*
$ls -Fl
total 1
-rw-------   1 markst   staff           0 Jan 21 17:10 d
-rw-------   1 markst   staff           0 Jan 21 17:10 e
-rwx------   1 markst   staff           9 Jan 26 20:47 f*
$f
```

The program will now run and do whatever it does. After it has finished the user will be returned to the prompt. It is useful to know that CTRL-C[4] will usually abort a program.

4.4 Creating & Deleting Directories

A new directory can be created by simply using the mkdir[5] command, and an empty directory can be deleted using the rmdir[6] command. Directories which are not empty cannot be deleted as easily. Say we want to create a subdirectory in our programs directory:

```
$ls -F
d  e  f*
$mkdir p
$ls -F
d   e   f*   p/
```

... and then delete it again:

[4]This means: hold the CTRL key down while typing the c key.
[5]MaKe DIRectory
[6]ReMove DIRectory

```
$rmdir p
$ls -F
d  e  f*
$
```

4.5 Deleting files

Deleting files is as easy as removing directories. To remove a file we use the rm[7] command. Say we want to delete the file 'd' in our programs directory:

```
$ls -F
d  e  f*
$rm d
$ls -F
e  f*
$
```

The file has been deleted without any conformation. As this is potentially dangerous, we can use an additional parameter to inquire whether or not to delete the file:

```
$ls -F
e  f*
$rm -i e
rm: remove ‘e’? y
$ls -F
f*
$
```

We will later find out how we can set up our system to always ask for confirmation when deleting files without having to type rm -i every time we delete a file.

4.6 Displaying and Creating Files

Unix provides a very basic mechanism to display and create files using the cat command. Usually a user will use an editor to create and edit files and a viewer like less to view files. But cat is always available.

[7] ReMove

A file can be created by typing the following (assuming we are still in our programs directory)[8]:

```
$cat > e
This will be the content of
our new file!
$ls -F
e  f*
$
```

The greater symbol signifies that cat will put its output into the file 'e'. If we reverse the greater sign the content of the file is displayed on the screen:

```
$cat < e
This will be the content of
our new file!
$
```

If we do not use any of the two symbols with cat, it will simply display the content of the file:

```
$cat e
This will be the content of
our new file!
$
```

4.7 Moving and Copying Files

4.7.1 Moving Files and Relative Paths

To move files around between directories or rename them we use the mv[9] command. This command takes the name of the source file and the name of the destination as parameters. Say we want to move our file 'e' in the programs directory to the directory above (recall the directory above is denoted by '..'):

```
$ls -F
```

[8] We tell the computer that our input has ended by typing CTRL-D. This is the standard way of telling UNIX our input has ended.

[9] MoVe

```
e   f*
$mv e ..
$ls -F
f*
$ls -F ..
a           b           c           data/
e           junk/       programs/
$
```

Let us now move the file 'e' in the directory above our current directory into the 'data' directory. At this point we will start using relative paths. Relative paths are a method of accessing a file that is not in our current directory. Recall that '..' is the directory above the current directory. File and directory names are always separated by a '/'. If we look at our directory structure in figure 4.1, say we want to give the relative path for the file 'o'. If we are in the start directory the path would be 'data/personal/o', from the directory data it would be 'personal/o', from the directory personal it would just be 'o', from the directory 'junk' or 'programs' it would be '../data/personal/o'. Say we want to give the relative path to the file 'k' from the 'personal' directory, we have to go up two directories and down into the 'junk' directory, this will give the relative path '../../junk/k'. Now remember we are in the 'programs' directory:

```
$mv ../e ../data
$cd ..
$ls -F
a           b           c           data/       junk/       programs/
$ls -F data
e           g           h           i           personal/
$
```

Now let us move the file back to the 'programs' directory and call it 'd':

```
$mv data/e programs/d
$ls -F data
g           h           i           personal/
$ls -F programs
d   f*
$
```

There is one restriction on the mv command; it can only move files on a physical disk, but not across disks. When the mv command fails it is always wise to check if you tried moving files between physical disks.

4.7.2 Copying Files

If we copy a file we create a completely independent copy of the file possibly in a different directory or with a different name. We copy files using the `cp`[10] command. As the `mv` command the `cp` command takes a source and destination as parameters. Let us now create a copy of the file 'd' in the 'programs' directory:

```
$cd programs
$ls -F
d   f*
$cp d e
$ls -F
d   e   f*
$
```

If we want to copy a file to a different directory we can simply specify that directory as the second parameter to the `cp` command:

```
$ls -F ..
a          b          c          data/      junk/      programs/
$cp d ..
$ls -F ..
a          b          c          d          data/      junk/      programs/
$
```

If we want to change the name of the file while copying we just append it to the destination directory name:

```
$cp d ../p
$ls -F ..
a          c          data/      p
b          d          junk/      programs/
$
```

Now let us remove those files and try to copy a whole directory:

```
$cd ..
$ls -F
```

[10] CoPy

```
a           c           data/       p
b           d           junk/       programs/
$rm d  p
$ls -F
a           b           c           data/       junk/       programs/
$cp data  safedata
cp: data: omitting directory
$
```

As we see cp cannot simply copy a directory. We have to explicitly tell it to recursively copy a directory, its files and subdirectories:

```
$cp -R data  safedata
$ls -F
a           b           c           data/       junk/       programs/   safedata/
$ls -F safedata
g           h           i           personal/
$ls -F safedata/personal
m  n  o
$
```

We have created a complete copy of the directory data and all its subdirectories. How do we now get rid of them though?

4.8 Removing Directories and their Contents

To remove a directory and its contents we could erase every individual file in the directory and its subdirectories, then all the subdirectories and finally the directory itself. As this is very cumbersome the rm command can be forced to recursively delete directories and their contents, like the cp command can copy directories recursively:

```
$rm -R safedata
$ls -F
a           b           c           data/       junk/       programs/
$
```

4.9 Crossreferences to Files: Links

We have seen how to create copies of files. Now we will see how to create crossreferences to files. Imagine we have a file that every user needs in his home directory. If we put a copy in every users directory this would use up masses of disk space. To stop wastage like this happening UNIX allows us to define crossreferences to other files: links. We can create a crossreference using the ln[11] command. There are two types of links, hard and soft links. Hard links are just another name for the same file. a hard link can not be distinguished in any way from the original name of the file. The only way to recognise a file which has hard links to it is by inspecting the link counter in the directory listing, if this is greater than one a file has hard links to it. Deleting a file with hard links to it can be problematic, as all hard links to it have to be deleted. Soft links however are much easier and safer. They are clearly distinguishable from the original filename and when the original file is deleted they just point into a void.

4.9.1 Hard Links

To create a hard link we just use the same semantics as with the copy command. we make a link from a source name to a destination name. Let's create a hard link:

```
$ls -Fl
total 3
-rw-------   1 markst   staff         0 Jan 21 17:13 a
-rw-------   1 markst   staff         0 Jan 21 17:07 b
-rw-------   1 markst   staff         0 Jan 21 17:07 c
drwx------   3 markst   staff      1024 Feb  6 20:47 data/
drwx------   2 markst   staff      1024 Jan 21 17:11 junk/
drwx------   2 markst   staff      1024 Feb  6 21:16 programs/
$ln  a d
$ls -Fl
total 3
-rw-------   2 markst   staff         0 Jan 21 17:13 a
-rw-------   1 markst   staff         0 Jan 21 17:07 b
-rw-------   1 markst   staff         0 Jan 21 17:07 c
-rw-------   2 markst   staff         0 Jan 21 17:13 d
drwx------   3 markst   staff      1024 Feb  6 20:47 data/
```

[11] LiNk

```
drwx------   2 markst   staff        1024 Jan 21 17:11 junk/
drwx------   2 markst   staff        1024 Feb  6 21:16 programs/
$
```

The first directory listing shows all files having only one name refering to them. The second listing shows the file with the name 'a' has two hard links to it (including the name 'a' itself), and the file 'd' also has two hard links to it. We cannot however see which other filenames point to them. there is a way however to find this out. Unix assigns every file an inode, which stores the size, location, number of hard links and other information of a file, and is uniquely identified by an inode-number. Directories are just listings of names and associated inode-numbers. To find a file Unix looks up the inode-number associated with the filename requested, finds the inode and from there finds the location. So if we want to know what the other name of a file is, we just have to list all the directories with inode-numbers and find the other name with the same inode-number. This is a very laborious process, which make hard links unattractive.

Another piece of information we can gain from the directory listing is the fact that directories also have hard links to them. junk and programs only have two hard links, but data has three. Each of the directories has a file called '.' in itself which is a hard link to itself. This is not normally shown by ls, but can be forced by the -a (for all) flag on ls. The data directory however has a third link to it; remeber the data directory has a subdirectory called personal. This subdirectory contains a file called '..' which is a hard link to data. this enables us to 'go up' one level in the directory structure.

Now let us look at the directory in full:

```
$ls -Fla
total 5
drwx------   5 markst   staff        1024 Mar  4 20:17 ./
drwx------   4 markst   root         1024 Jan 21 17:07 ../
drwx------   2 markst   staff        1024 Feb  6 21:16 programs/
drwx------   3 markst   staff        1024 Feb  6 20:47 data/
drwx------   2 markst   staff        1024 Jan 21 17:11 junk/
-rw-------   2 markst   staff           0 Jan 21 17:13 a
-rw-------   1 markst   staff           0 Jan 21 17:07 b
-rw-------   1 markst   staff           0 Jan 21 17:07 c
-rw-------   1 markst   staff           0 Jan 21 17:52 .a
-rw-------   2 markst   staff           0 Jan 21 17:13 d
$
```

4.9.2 Soft Links

To create a soft link we use the same command we use to create a hard link, except that we use the -s flag. Again we make a link from a source name to a destination name. Let's create a soft link:

```
$ln -s a e
$ls -F
a          c          data/      junk/
b          d          e@         programs/
$ls -l
total 3
-rw-------   2 markst   staff          0 Jan 21 17:13 a
-rw-------   1 markst   staff          0 Jan 21 17:07 b
-rw-------   1 markst   staff          0 Jan 21 17:07 c
-rw-------   2 markst   staff          0 Jan 21 17:13 d
drwx------   3 markst   staff       1024 Feb  6 20:47 data/
lrwxrwxrwx   1 markst   staff          1 Mar  5 20:03 e -> a
drwx------   2 markst   staff       1024 Jan 21 17:11 junk/
drwx------   2 markst   staff       1024 Feb  6 21:16 programs/
$
```

In the short directory listing our new soft link has been marked with a '', to show it is a soft link. In our long directory listing we can see 'e' is a link by the link attribute in the first column and by the '-> a' which shows where the link goes to.

4.9.3 A last note on Links

Links are not limited to pointing to files in the same directory. They can be in completely different directories. There is the exception that hard links, due to their nature, are limited to one physical disk.

4.10 Hidden files

In the last few directory listings you will have noticed a hidden file, that was revealed when using the -a flag on ls, if you looked carefully. In Unix all files starting with a '.' as the first character of their filename are hidden.

4.11 Access permissions

As we have learnt previously all files have a set of access permissions. Every file is given the user's standard access permissions when it is created. The standard access permissions are set by the umask command:

```
$umask
077
$
```

The standard access permissions are expressed as a 9 bit octal number. As this is not very intuitive we can ask for a symbolic output[12]:

```
$umask -S
u=rwx,g=,o=
$
```

There are access permissions for the user who will own the new file (u), other users in the file's group (g) and other users not in the file's group (o).

There are three types of permission which can be given:

- r Read
- w Write
- x Execute (or access for directories)

In our case the umask means: Do not give group members or others any access, but give the user access for reading and writing for files and reading, writing and accessing for directories.

Permissions can be either set in addition to the existing permissions (+) or removed (-).

Say we want to give all group members read access to our newly created files:

```
$umask g+r
$umask -S
u=rwx,g=r,o=
$
```

[12] This will only work with the bash shell, but not with the csh shell.

To change permissions on already existing files we use the chmod[13] command. The access permissions of existing files are expressed as 12 bit octal numbers, as they contain more information than the standard access permissions. The chmod command can take either a four-digit octal number or a symbolic description of the access permissions. On how to use the octal numbers please refer to appendix A.1.

Whose access is changed is set by 'u' for user, 'g' for group, 'o' for others and 'a' for all simoultaneously.

Permissions can be either set in addition to the existing permissions (+) or removed (-). Additionally the access permissions can be set to only the permissions given on the command line (=).

The following permissions can be given:

- r Read
- w Write
- x Execute (or access for directories)
- X Execute if the file is a directory or some other user already has execute permission for the file.
- s Set user or group ID on execution (eg. passwd)
- t If this flag is set on a directory only the owner of a file in the directory can modify the file, even if other users have write access to that directory. The owner of the directory can however delete all files, even if other users with write access cannot, as they usually can.
- u The same permissions the owner currently has.
- g The same permissions the file's group currently has.
- o The same permissions the other users currently have.

The general syntax for the chmod command is chmod permissions[,permissions] file [file]. So we can set multiple permissions for multiple files. Additionally we can specify the -R flag before the permissions and a directory name as file to apply the permissions the the directory and all files and directories beneath it.

Say we want to change the permissions of a file to allow group read access:

[13] CHange MODe

```
$ls -l a
-rw-------   2 markst   staff          0 Jan 21 17:13 a
$chmod g+r a
$ls -l a
-rw-r-----   2 markst   staff          0 Jan 21 17:13 a
$
```

4.12 Groups

As we know every file has an associated group and every user can be the member of several groups. To find out what groups a user is the member of, we can use the groups command:

```
$groups
staff root
$
```

The current group, files will be created as, is the first in the list (in this case staff). In most systems we can access all files, whose group we are in, regardless of whether or not it is the current one. If we want to change the current group we can use the newgrp [14] command:

```
$newgrp root
```

```
$groups
root staff
$
```

To change a file's group we can use the chgrp[15] command. It takes as its first argument the group name followed by a list of filenames:

```
$ls -l a
-rw-r-----   2 markst   staff          0 Jan 21 17:13 a
$chgrp root a
$ls -l a
-rw-r-----   2 markst   root           0 Jan 21 17:13 a
$
```

[14] NEW GRoup
[15] CHange GRouP

A file's group can only be changed to a group the user is a member of.

4.13 Wildcards

In the previous sections we have seen that some commands like ls, cp, mv, rm, chmod, chgrp, can take a list of filenames as arguments. As it is quite laborious to type out a whole list of filenames UNIX provides wildcards through the shell. There are two wildcards:

The ? can stand for any one character.

EXAMPLE:

?arry can stand for barry, larry, harry, Barry and so on.

??rry con stand for barry, lorry, Barry, Lorry and so on.

The * can stand for any number of characters.

EXAMPLE:

m*m can stand for mm, mum, my_rum and so on.

It is also possible to mix ? and *. The effect of using wildcards is that the shell creates a list of filenames. This is important to know, as the program called with a wildcard expression always sees a list of files. Programs that do not support a list of files on their command line sometimes display weird behaviour, if presented with such a list.

Chapter 5

Piping & Redirection

Piping and redirection are ways of changing where a program puts its output and from where it takes its input. We have already seen an example of redirection using the cat command. In the examples the cat command took its input from a file or put its output into a file. Piping works in a similar way except that the output of one program is used as the input to another program.

To give a few examples we will use the grep[1] program. In its simplest form it will search its input for a string of characters. Say we are looking for a particular file in a very big directory. We know its name contains a 'd'. Using redirection[2] we could first create a file containing all the entries in the directory and then use this file as the input to the grep command:

```
$ls > our_dir
$cat < our_dir
a
b
c
d
data
e
junk
our_dir
programs
$grep d < our_dir
d
data
```

[1] Gnu Regular ExPression search

[2] Remember redirection uses the '<' and '>' symbols to indicate the direction of data flowing to and from the file.

```
our_dir
$rm our_dir
$
```

As this is somewhat cumbersome UNIX also provides us with pipes. These save us having to create a temporary file. To use them we simply put a '|' between the two programs:

```
$ls|grep d
d
data
$
```

Piping and redirection can also be mixed and both kinds of redirection used simultaneously. Also piping is not restricted to two programs, but can involve a whole chain of programs. It is for instance possible to take the input from a file ('<'), then grep some information out, grep some more selected information ('|'), then sort the output ('|') and finally write the output into a file ('>').

Whenever we use the '>' symbol the output file is overwritten with our new information. If we want to keep the original output file and just append our new information to it, we can use two '>' symbols:

```
$cat > z
Hello.
$cat < z
Hello.
$cat >> z
Hello World.
$cat < z
Hello.
Hello World.
$
```

Chapter 6

Process Management

Up till now we have pretended that the UNIX system we were using was only used by one user and only one program was running at a time. In fact the computer you are using can be concurrently used by many users and be appear to be running many programs at the same time.

There are a number of programs running on most UNIX systems designed to ensure smooth operation. These programs will run even if no one is using the system and most of these programs are not of real interest to us as they only help the operating system manage cache, virtual memory and logging of system messages. But some like cron, lpd and xdm have a direct relevance to us. We will later see how.

When we log into the system our shell is started. Every user that logs in has his shell started at login, so there is one shell running for each user simultaneously.

6.1 Listing users

We can check who is currently logged on to the system using the w command. This gives us some basic system statistics and list all users logged on:

```
$w
  8:54pm  up 22 min,  3 users,  load average: 0.00, 0.02, 0.04
User     tty       login@  idle   JCPU   PCPU  what
jasonw   tty5      8:42pm    10                -bash
asti     tty6      8:43pm    11                (bash)
markst   ttyp2     8:32pm                      w
$
```

We can also see when the user logged on, how long he has been idle and what he is currently doing. Two of the users seem to be doing nothing as only their shell appears as their active process. The third user is actually running the w program.

6.2 Listing processes

Although the computer seems to be running many processes simultaneously, it actually only cleverly divides its time between all the processes. The user however is always told that there are many processes running simultaneously and it intuitively seems this way when using the system.

To see which processes are currently running on a UNIX system we can use the ps command. It works slightly different on all systems, but mostly using the command without any flags will list your own processes:

```
$ps
  PID TTY STAT  TIME COMMAND
  107 pp1 S N   0:00 bash
  260 pp1 R N   0:00 ps
$
```

The listing includes the PID or Process Identification, the output device it is associated with, the status, the real processor time used and finally the program name. The status in this case includes the flags S for stopped while ps is R for running and N means a process has a priority which is nice to other processes.

If we want to see all users' normal processes we can use the 'a' flag for ps. As all processes are then displayed there is no way of telling whose' processes they are. We can use the 'u' flag to give us more information:

```
$ps au
USER       PID %CPU %MEM SIZE  RSS TTY STAT START   TIME COMMAND
asti        69  0.0  0.0 1180    0 v06 SWN  20:32   0:00 (bash)
jasonw      68  0.0  4.2 1200  288 v05 S N  20:32   0:00 -bash
markst     107  0.0  5.9 1212  400 pp1 S N  20:32   0:00 bash
markst     108  0.0  5.6 1208  380 pp2 S N  20:32   0:00 bash
markst     156  0.0 23.3 3820 1568 pp1 S N  20:44   0:01 emacs part3.tex
markst     312  0.0  3.2  880  216 pp2 R N  21:14   0:00 ps au
root        67  0.0  1.1  860   80 s00 S N  20:32   0:00 gpm -t ms
$
```

6.3 Virtual Memory

There are some interesting additional fields in this output. The %CPU and %MEM fields just indicate the percentage of CPU time and memory used. The size field indicates the total size of the program and RSS indicates the total resident size of the program. As these values differ the question arises where the rest of the program has gone. The answer is quite simple. As not every program is running at the same time, not all programs have to be in physical RAM at the same time. Also some parts of programs are never used, so they do not have to be in RAM. Other programs wait for user input and do not have to be in RAM until they get some input. These programs are swapped out onto the hard disk to make space for more needy programs and disk caching. In this way a computer with very little physical RAM can appear to have an enormous amount of memory.

6.4 System usage

If we feel the system might be slow, we can check the system load by using the `uptime` command. It outputs the first line output by the `w` command:

```
$uptime
  1:57pm  up 38 min,  1 user,  load average: 0.00, 0.06, 0.06
$
```

This tells us the current system time, the time the system has been running, the number of users using the system and the load average over the last 5, 10 and 15 minutes. Depending on the system we are using a load of 1 might be high, on other systems however a load of 10 might be high. The load average just represents relative values.

6.5 Running multiple processes

Now we know how to check for multiple processes running, we can start running multiple processes ourselves.

If we want to run a program in background which does not need any user interaction and does not produce screen output we can use the '&' at the end of the command line to do this:

```
$ls -Fla > out_dir &
```

```
[2] 296
$
```

The ls command was told to put its output into a file and run in background. The shell tells us that this is the second job in this shell (the first being the editor this document is written in) and the PID of the job is 296. We are immediately returned to the prompt and can issue further commands while the ls is running in background. If we enter the next command or press return after the ls command has finished we get the following message:

```
$
[2]+  Done                    ls -Fla >out_dir
$
```

This tells us that the command ls has finished.

To see what jobs are currently running and have been started from our shell we can use the jobs command. This has the advantage against the ps command that it only shows us programs started from this shell.

```
$jobs
[1]+  Running                 emacs part2.tex &
$
```

This shows us that job number one is running and the command line used to start it was emacs part2.tex &.

If we are running a program and want to quickly do something else we can stop the current program by typing CTRL-Z. Say we were running the editor program vi. If we stopped it with CTRL-Z we would get the following message:

```
[2]+ Stopped            vi
$
```

Typing jobs would show us the following:

```
$jobs
[1]- Running          emacs part2.tex &
[2]+ Stopped          vi
$
```

The current job is marked + and the previous job is marked -. To return to the current job we can simply type fg[1] or to run the current job in background we can type bg[2]. To access the previous job we can type fg %- or bg %-; any other job can be accessed via its number, eg fg %1, or via its name, eg fg %vi.

So to get back to the vi program, we can type fg or fg %2.

6.6 Sending Signals to Processes

Most processes do not like being sent signals. Their reaction to a signal is usually dying. Some processes however exit gracefully when sent a signal; others have a certain type of individual reaction to a signal, like re-reading their setup files.

We can send a signal to a process using the kill command. The kill command takes an optional signal type as a command line argument and a process ID, which can be found using the ps command. Carrying on from the previous example, lets kill vi:

```
$ps
  PID TTY STAT  TIME COMMAND
 1053 pp1 S N   0:00 bash
 1079 pp1 S N   0:03 emacs part3.tex&
 1209 pp1 T N   0:00 vi
 1211 pp1 R N   0:00 ps
$kill 1209
$ps
  PID TTY STAT  TIME COMMAND
 1053 pp1 S N   0:00 bash
 1079 pp1 S N   0:03 emacs part3.tex&
 1209 pp1 T N   0:00 vi
 1215 pp1 R N   0:00 ps
$
```

Our vi has not terminated itself. This is because we stopped it and although it has received our TERM signal, it cannot react to it. Let's start it running again:

```
$fg
```

[1] ForeGround
[2] BackGround

```
vi
$ps
  PID TTY STAT  TIME COMMAND
 1053 pp1 S N   0:00 bash
 1079 pp1 S N   0:03 emacs part3.tex&
 1226 pp1 R N   0:00 ps
$
```

If we do not specify a signal to be sent, kill automatically sends the TERM signal, which tells programs to terminate themselves. If a program refuses to terminate itself, we can send it the KILL signal, which will externally abort the program. If we had sent our vi the KILL signal, it would have died immediately.

To find out the exact names of all signals we can type kill -l. We can find the signal we want and then send it to a process:

```
$kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGIOT       7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR
$ps
  PID TTY STAT  TIME COMMAND
 1053 pp1 S N   0:00 bash
 1079 pp1 S N   0:03 emacs part3.tex&
 1209 pp1 T N   0:00 vi
 1215 pp1 R N   0:00 ps
$kill -SIGKILL 1209
[2]+  Killed                 vi
$
```

We can specify the name or number of the signal we want to send on the command line.

Chapter 7

Editing Files

7.1 What exactly is editing and an editor

An editor is a program that allows you to edit files. Editing is the process of changing the contents of a file character by character. It is distinct from word processing. Word processing inserts formatting and layout information into a text whereas editing does not. Editors can however have features word processors do not have. They can do indentation, automatic insertion of line breaks for program source code. They can be freely programmed to do a multitude of tasks, emacs even goes as far as offering e mail, news and file manager facilities.

We will look at three common UNIX editors all of which are available for MS-DOS and Windows as well.

Ed is a very basic line editor and available on almost all systems including the most basic. Vi is the classic UNIX full screen editor, it is available on most systems, except the very basic. EMACS is a very more powerful editor that even allows programs to be written with its internal LISP language. A knowledge of these three editors should enable you to do editing on all UNIX systems.

7.2 Using ed

ed is a line-oriented text editor. The reason we learn about it is not only that it is available on nearly all systems, but also that it works on most devices from teleprinters to high end graphics stations. Most workstations only have a line oriented text console in their basic state when no graphics interface is running, so full-screen text editors cannot be used. Line editing means that a whole line has to be typed at a time. This is important to bear in mind when using ed.

ed has two distinct modes: command and input. When first started ed is in command mode. If it is given a file name on the command line it will load that file into its text buffer. All changes made to the text buffer will only be saved to the file when the 'w' command is issued, otherwise all changes are lost. If no file name is given on the command line, the text buffer is empty initially.

In command mode commands are read from standard input and executed, manipulating the contents of the text buffer.

When ed is in command mode an input command like 'a' (append), 'i' (insert) or 'c' (change) will switch ed into input mode. In this mode no commands are available and the standard input is written directly into the text buffer. Input mode is ended by typing a single '.' on a line by itself followed by a return.

When started without a file name ed just sits in command mode without any prompt, waiting for user input; when ed is started with a filename it displays the number of characters in the file and then waits for user input. If we issue the 'P' command, ed will display a prompt whenever it is in command mode. This is useful to distinguish the two modes, even if the prompt is just a '*'.

All ed commands take the same format. They start with an optional range of lines, followed by a command to be applied to that range of lines, which in some cases is followed by a set of parameters. If the range of lines is left out before a command, the command applies to the current line.

To append text after a line we use the 'a' command. To insert text into an empty file we type '0a' at the prompt:

```
$ed
P
*0a
Hello world.
.
*
```

To insert some text before a given line we use the 'i' command:

```
*1i
This is before Hello world.
.
*
```

To list what we have done so far we can use the 'l' command. The first line is always 1 and the last line $:

```
*1,$l
This is before Hello world.$
Hello world.$
*
```

If we want to change a line we use the 'c' command:

```
*1c
Listen everybody:
.
*1,2l
Listen everybody:$
Hello world.$
*
```

To delete lines we use the d command:

```
*1d
*1,$l
Hello world.$
*
```

To write our work to disk we use the w command. If we gave a file on the command line, we can just type w; if we started a new file we have to follow w with the file name to save the text as:

```
*w a
13
*
```

To quit from ed we just use the 'q' command:

```
*q
$
```

ed has a whole range of searching and replacing commands that can be run in very flexible ways, but we will not discuss them here as ed is usually only needed in very extreme cases. For further information see appendix A.2.

7.3 Using vi

vi is a full screen editor, which most people either love or hate. It has a command mode and an input mode similar to ed, but is much easier to use as it is not line oriented, but character oriented. The cursor keys are fully usable, so it is much more intuitive.

To start editing a file we just type vi followed by the filename we want to edit (eg. vi my_file). The following screen will appear:

```
~
~
~
~
~
~
"my_file" [New file]
```

The cursor will be in the top left hand corner. As with ed there are commands for inserting (i) and appending (a) text. If we insert some new text, we enter the input mode by typing i and return to the command mode by typing the ESC key.

```
Hello world.
How are you?
~
~
~
~
~
"my_file" [New file]
```

We can now use the cursor keys to move around the text. To delete a character we can type x. To delete a whole line we can type dd. To search for a string we can type / followed by the expression to search for.

vi also has a subcommand mode, which covers all file functions needed. To issue one of the subcommands we first have to enter the subcommand mode by typing :. A colon will appear on the last line of the screen and we can now enter subcommands. The :w command write the text to the original file. If we append a file name (leaving a space between w and the name), we can write the text to a different file. To start editing a new file we use the :e command followed by a file name. To quit from vi we use the :q command. If we want to leave vi without saving the text we use the q! command.

```
Hello world.
How are you?
~
~
~
~
~
~
~
~
:w
```

These are the basic commands needed to use vi, for further information refer to the appendix **??**.

7.4 Using EMACS

EMACS is a very powerful and large editor. Apart from the normal editor functions, it can automatically format program source code while it is being written, be programmed to do nearly any task in lisp, offers an electronic mail interface and an internet news reader, automatically can compile the programs written with it, provides a file manager, and last but not least offers automated phsychotherapy. The latest X-Windows versions also have pull-down menus.

This power of course also uses a large amount of the computers resources, and some people call it "Eight Megabytes Always Continously Swapping"

In contrast to ed and vi, EMACS does not have distinct command and input modes. When EMACS is started up we can immediately start entering text. All commands are reached via control keys or by calling a lisp function.

The control keys used are the CONTROL and META keys. In the EMACS manual and the following examples keystrokes are shown as follows:

- ◇ C-x i - hold down the CONTROL key, tap x, release the CONTROL key and tap i.

- ◇ C-x C-s - hold down the CONTROL key, tap x, ,tap i and release the CONTROL key OR hold down the CONTROL key, tap x, release the CONTROL key, hold down the CONTROL key, tap i and release the CONTROL.
- ◇ M-t - hold down the META key and tap t.
- ◇ M-! - hold down the META key, hold down the SHIFT key, tap 1.
- ◇ M-x goto-line - hold down the META key and tap d, release the META key and type goto-line followed by RETURN.

EMACS is run by typing emacs and RETURN at the shell prompt. EMACS allows full use of the cursor keys, and the backspace key, so even novices can start editing text without any specialist knowledge. The easiest way to learn about EMACS, is to use the online tutorial, which can be started by typing C-h t.

Here are a few important EMACS commands:

- ◇ C-x C-c - quit from EMACS
- ◇ C-x C-s - save the current file
- ◇ C-x C-f - open a new file for editing
- ◇ C-v - jump forward one page
- ◇ M-v - jump back one page
- ◇ C-k - delete from cursor to end of line and insert into copy buffer
- ◇ C-Space - set a mark
- ◇ C-w - move text into copy buffer
- ◇ M-w - copy text into copy buffer
- ◇ C-y - insert contents of copy buffer

As the use of EMACS is more intuitive as the other editors, it is easiest to learn more about, while using it.

7.5 Conclusion about Editors

ed, vi and EMACS are the most common UNIX editors. ed is available on virtually all systems, vi an the majority and EMACS on nearly all modern systems. There are situations where only ed can be used, which is the only reason to learn its basics. For fast editing vi is the best to use and the majority of system tools use vi, which makes a basic knowledge of it essential. EMACS is a luxury editor which is not essential, but is a lot more user-friendly than ed or vi.

Chapter 8

Printing

Printing in a UNIX system can be done in two ways. Either files are directly copied to the special file associated with the printer or files are submitted to the spooling system.

Copying file directly to the printer has the disadvantage that the output might be jumbled up with somebody else's, if both users are copying their files to the printer at once.

Spooling is far more reliable and also allows the use of remote printers. When the user submits a file for printing, it is placed in a queue. Each printer has one queue, so by selecting the printer with the shortest queue, it is possible to get the printouts quicker, on a higher quality printer or on a printer, which is nearer.

Queues are usually associated directly with one printer, some systems though have one queue for many printers or many queues for one printer. Where there are many printers to one queue the system might have an intelligent way of evenly distributing the jobs onto the printers. Systems with many queues for one printer, may do some pre-processing like adding titles and pagenumbers to simple textfiles, while leaving graphics files as they are. Another use of multiple queues is to have one printer which will print single sided on one queue and double sided on the other queue.

8.1 Printing Files

To print a file we use the lpr command. To print a file to the default printer we can do the following:

```
$lpr our_file
```

```
$
```

If we want the printout on a different printer, we can use the -P flag and specify another printer (eg. raw):

```
$lpr -Praw our_file
$
```

The lpr command can also take input from a pipe:

```
$ls -lF | lpr
$
```

This will print the contents of the directory. The last option we have for the lpr command, is to take our input from the standard input:

```
$lpr
This is just a test print.
$
```

Now that we have submitted our print jobs, how can we check on their progress?

8.2 Monitoring Printing

To check the default print queue, we can use the lpq[1] comand:

```
$lpq
lp is ready and printing
Rank   Owner      Job  Files                                  Total Size
active markst     35   book.ps                                39597 bytes
1st    markst     36   ex1/ex1.dvi                            3400 bytes
$
```

If we want to check another printer queue, we can again add the -P flag:

```
$lpq -Praw
waiting for raw to become ready (offline ?)
Rank   Owner      Job  Files                                  Total Size
1st    markst     0    ex1/ex1.dvi                            3400 bytes
$
```

[1] Line Printer Queue

We also gain some status information from the lpq command. It tells us if a printer is printing, currently off-line, or if data is being sent to a remote printer.

8.3 Stop Printing

If we have mistakenly submitted a file for printing and do not want it printed after all, we can remove it from the print queue as long as printing has not started yet. Carrying on from the previous examples, we can remove jobs from the print queues using the lprm[2] command and the job number given by the lpq command. Again we have the option of specifying a different printer:

```
$lprm -Praw 0
dfA000Aa00255 dequeued
cfA000Aa00255 dequeued
$lpq -Praw
no entries
$
```

8.4 Controlling Printers

We have now seen all essential printer commands, but how can we find out what printers there are and what they are doing.

UNIX provides the lpc[3] command to do exactly that. Additionally it offers the printer administrator the options to start and stop printing and queueing of new jobs. The administrator can also rearrange the order of the jobs in the queue.

The lpc command can either be run as an interactive program or by supplying lpc with commands on the command line:

```
$lpc
lpc> help
Commands may be abbreviated.  Commands are:

abort   enable  disable help    restart status  topq    ?
clean   exit    down    quit    start   stop    up
lpc> status
```

[2] Line Printer ReMove
[3] Line Printer Control

```
ascii:
        queuing is enabled
        printing is enabled
        1 entry in spool area
        waiting for ascii to become ready (offline ?)
lp:
        queuing is enabled
        printing is enabled
        1 entry in spool area
        lp is ready and printing
raw:
        queuing is enabled
        printing is enabled
        no entries
        no daemon present
lpc> quit
$lpc status
ascii:
        queueing is enabled
        printing is enabled
        1 entry in spool area
        waiting for ascii to become ready (offline ?)
lp:
        queueing is enabled
        printing is enabled
        1 entry in spool area
        lp is ready and printing
raw:
        queueing is enabled
        printing is enabled
        no entries
        no daemon present
$
```

We can see which printers there are on the system, if they are printing and accepting jobs, how many jobs there are in the print queue and what the printers status is. The additional commands displayed by the help command are only available to the administrator.

Chapter 9

Shells

Shells are the user's interface to the operating system. Most UNIX shells are text based although there are some graphical shells as well. X-Windows window managers also display a shell like behaviour in some cases, but are not really shells.

We have already seen the most important function of a shell: the starting of programs. Shells do provide other services as well.

9.1 Environment

The environment of a process consists of a set of environment variables. Each of these variables has a name and can store text. There are environment variables telling the system which screen the user is using, where his home directory is, what his current directory is, what type of system is being used, what the prompt looks like and what the search path for programs is. In addition to these standard variables programs can define and use their own variables. Every process inherits the environment from its calling process, which in most cases is the shell. The shell however does not take on changes made by its children to the environment, in the same way children get their looks from their parents, but not vice versa. The shell can be used to create new environment variables, to change existing variables or delete old ones. Every shell has different commands to do this.

The shell we are using is bash. Bash has two commands to manipulate environment variables. The unset command removes variables from the environment. The export command adds or manipulates environment variables. To list all current environment variables we can simply type export:

```
$export
```

```
declare -x DISPLAY="nietzsche:0.0"
declare -x HOME="/home/local/markst"
declare -x HOSTNAME="nietzsche.cs.rhbnc.ac.uk"
declare -x HOSTTYPE="i386"
declare -x LOGNAME="markst"
declare -x OSTYPE="Linux"
declare -x PATH=":/bin:/usr/bin:/usr/X11R6/bin:/usr/X11/bin:."
declare -x PS1="\\h:\`pwd\` > "
declare -x PS2=">> "
declare -x SHELL="/bin/bash"
declare -x TERM="xterm"
declare -x USER="markst"
$
```

We can see what display we are using, what our home directory is, what the name of the computer is we are using, the type of computer, our login name, the operating system's name and the path, which is the directories the shell scans when looking for a program. PS1 is the primary prompt, PS2 is the secondary prompt, then we see what the shell is we are using, what type of terminal we are using and what our user name is. This is just an extract from the environment, giving a few important variables.

If we just want to check the contents of a variable we can use the shell's echo command. This command takes text and prints it out. It can also print the contents of a variable if the variable's name is prefixed with a $:

```
$echo Hello
Hello
$echo The user variable contains: $USER
The user variable contains: markst
$
```

Say we want to delete the USER environment variable:

```
$unset USER
$echo The user variable contains: $USER
The user variable contains:
$
```

To check if the variable has been deleted we can use the export command on its own without a variable name.

Say we now want to add the USER variable again:

```
$export USER=markst
$echo The user variable contains: $USER
The user variable contains: markst
$
```

If we want to create a new variable, which has a space as part of its contents we have to put it in double quotes:

```
$export USER2="Nick Dunkin"
$echo $USER2
Nick Dunkin
$
```

We can also create environment variable, which take their information from other environment variables:

```
$export USER2="$USER2 and $USER"
$echo $USER2
Nick Dunkin and markst
$
```

9.2 Aliases

Aliases are a method of providing shorthands for commonly used commands. Before the shell goes off to search the disk for a program, it will search its alias list to see if one of the aliases has been typed in by the user. In bash[1] aliases are set with the alias command and unset with the unalias command.

One of the most commonly used aliases is to replace ls by ls -F and rm by rm -i:

```
$alias rm="rm -i"
$alias ls="ls -F"
$
```

Typing alias on its own will give a list of current aliases:

[1] csh uses nearly the same commands except that the '=' is replaced by a space.

```
$alias
alias ls='ls -F'
alias rm='rm -i'
$
```

To delete an alias we can use the unalias command:

```
$unalias ls
$alias
alias rm='rm -i'
$
```

9.3 Scripts

Scripts are text files that contain a series of shell commands. When the execute permission is set on such a text file, it can be run like a program and the commands in the file are executed in sequence.

This in it self is no great achievement, yet the extent of the commands provided by most UNIX shells makes these scripts very powerful tools.

Scripts can take arguments as command line parameters. Say we create a script that will display the contents of a file in our home directory and we want to be able to specify the file name without the directory name on the command line. If our script were called homecat we could then run the script to display the file a in our home directory by typing homecat a. The script would look like this:

```
cat ~/$1
```

The $1 denotes the first command line argument and it is possible to specify more than one argument on the command line.

We can use the shell's echo command to display output on the screen. bash also provides basic arithmetic commands. If we want to write a shell script that adds two numbers, which are passed on the command line, we can use these two functions together:

```
echo $[ $1 + $2 ]
```

Shell can also iterate over a series of items in a list. We could write a little script that printed out the following:

```
Hello Harry !
Hello Sally !
Hello Robert !
Hello Victoria !
```

To do this we have to use the shell's for loop which iterates over a list of words:

```
#!/bin/sh
for i in Harry Sally Robert Victoria; do
  echo Hello $i !
done
```

i is set to a different value each time the for loop is iterated over. This can also be used when renaming files. Say we want to rename all files in a directory, such that they all have a .new appended to them:

```
$ls
m      n      o
$~/rename
$ls
m.new  n.new  o.new
$
```

We cannot use the move command:

```
$mv * *.new
mv: when moving multiple files, last argument must be a directory
$
```

We have to write a script that takes one file at a time and moves it to the new file name. We can tell the for loop to take the list of all files in the current directory, which is '*' and assign the variable i a different file name on each iteration:

```
#!/bin/sh
for i in * ; do
  { mv $i $i.new ; }
done
```

Another useful shell command is the while loop. It allows us to run through a loop while some condition is true. If we wanted to print the numbers 1 to 5 we could use the following script:

```
#!/bin/bash
i=0

while [ $i -le 5 ]; do
  echo $i
  i=$[$i+1]
done
```

The [is actually a command called test. To find out more about it type man test.

We can use the while loop in a similar way to the for loop in renameing files:

```
#!/bin/sh
ls | while read i; do
  { mv $i $i.new ; }
done
```

Here we are reading the output of the ls command into the variable i one line on every iteration. Usually the while loop is used when it is not clear at the start of the loop how many iterations there will be.

Up to now we have not been able to influence the flow of control once a script has started and we have had to execute all commands from start to finish. Sometimes it is useful not to execute all commands. The if statement allows us to do this.

We can put some safety into our rename script by first asking the user if the files should be really renamed:

```
#!/bin/sh
echo "Do you want to rename all files (y/n)"
read i

if [ "$i" = "y" ]; then
  echo "Renaming files..."
  for i in * ; do
    { mv $i $i.new ; }
  done
else
  echo "Aborting..."
fi
```

The case statement is i similar command. It allows the shell script to make decisions of what to do. Say we want to extend our rename script to give us the option the rename a file or not:

```
$~/rename
Do you want to rename the file m (y/n/?)
?
If you answer with 'y' the file m will be renamed.
If you answer with 'n' the file m will not be renamed.
n
Skipping m
Do you want to rename the file n (y/n/?)
n
Skipping n
Do you want to rename the file o (y/n/?)
y
Renameing data
$
```

The shell script needed gets more complicated:

```
#!/bin/sh

for i in * ; do
  j=?
  echo "Do you want to rename the file $i (y/n/?)"
  while [ "$j" = "?" -o "$j"]; do
    read j
    case $j in
      ("y") echo Renameing $i
          mv $i $i.new
          ;;
      ("n") echo Skipping  $i
          ;;
      ("?") echo "If you answer with 'y' the file $i will be renamed."
          echo "If you answer with 'n' the file $i will not be renamed. "
          ;;
    esac
  done
done
```

Shell scripts can be very powerful, but can also be very hard to understand, like the following on which lists all file names and file lengths in a directory:

```
#!/bin/sh
ls -l > /tmp/ls.$$

while read i; do
        fname=`expr "$i" : '.* \(.*\)$'`
        fsize=`expr "$i" : '[-drwxXsStT]*[ ]+[0-9]+
[ ]+[0-9a-zA-Z]+[ ]+[0-9a-zA-Z]+[ ]+
\([0-9]+\)[ ]+.*$'`
        echo -e $fname "\t\t\t" $fsize
done < /tmp/ls.$$

rm /tmp/ls.$$
```

To avoid some of the confusion it is possible to break up shell scripts into functions, though most users only use very basic scripts, and would not be expected to be able to understand scripts like the previous one.

9.4 Initialisation Files

Initialisation files are a special kind of file that is read when a program is started. When a user logs in, the operating system starts some default scripts, which set up the standard path, the user, the home directory and the system's default environment. Each user can have a special file usually called .profile in their home directory, which is run for them at login. In addition the shells also have special scripts that set up aliases and the like for users. For bash the initialisation file is called .bashrc, for csh it is .cshrc. If some alias or path change should be made permanent it should be entered into the shell's initialisation file. If a change is made to the shell's initialisation file it has to be reread by the shell. Simply executing the script will not have any effect as the shell will not inherit the child's (or script's) environment. We have to use the special source[2] command:

```
$source ~/.bashrc


$
```

[2] This works for bash and csh. In addition bash also accepts '.'.

Chapter 10

The Internet

This chapter has been transferred to a separate document.

Chapter 11

X-Windows

11.1 What X-Windows really is

X-Windows provides computers with a graphical user interface. The difference to most graphical user interfaces is, that any program displayed can be run on any other computer, that the displaying computer is connected to. X-Windows is completely separate from the operating system, which means programs can do their work and displaying separately. This is a major difference to graphical user interfaces like MS-Windows, where the working of the program is heavily dependent on the displaying of information.

X-Windows is available on a multitude of computers. There are versions for UNIX, VMS, MS-Windows, the Apple Macintosh and many other platforms. For UNIX and VMS they are the primary graphical user interfaces.

11.2 Window Managers

X-Windows is a bit like an operating system. It provides all functions needed for a graphical user interface except a user interface. This works in the same way the operating system needs a shell to communicate with the user. X-Windows needs a window manager which provides functions like resizing of windows, dragging windows across the desktop, deciding the focus policy and other features.

11.2.1 Things Window Managers do not do

Window managers do not provide applications with a common look and feel. This leaves applications to decide their own user interface. There are applications as diverse as the terminal window (xterm) and the WWW browser Netscape.

Window managers do not control applications. A window manager cannot directly kill off a rogue application, as this is a function of the operating system. The current window selected, does not determine which program is running.

11.2.2 Things Window Managers do

Window managers provide the borders around the windows. Without a window manager windows would simply have no borders. Window managers also allow changing the location and size of windows. They determine in which way the input focus is switched. They can raise or lower windows. Most window managers provide a menu system. Some window managers provide icon management and virtual desktops.

There are a variety of window managers, some of which are listed here:

twm - Tab Window Manager is the most basic window manager provided with nearly all X-Windows implementations[1].

mwm - Motif Window Manager is the window manager you are currently using. It is widespread, but is not free like twm. Newer versions also have virtual desktops. It has a similar look and feel to Microsoft Windows.

fvwm - F(?) Virtual Window Manager is derived from twm but looks very much like mwm. It provides virtual desktops like newer versions of mwm.

olwm - Open Look Window Manager is Sun's window manager, but has been implemented for a variety of machines. It has it's own special look and feel, which is different from all other window managers. There is a version with virtual desktops called olvwm.

Appendix C contains screen shots of the above window managers.

11.2.2.1 Window location

Window managers can change the location of a window. This is usually done by moving the mouse pointer to the title bar of a window, holding down the left mouse button and moving the window, then releasing the mouse button when the window is in the right place.

[1] Suns are often only supplied with olwm or olvwm

11.2.2.2 Window size

If the mouse is moved onto the border, the pointer changes shape and by pressing the left mouse button and moving the mouse it is possible to change the window size. Not all programs allow changing their window size and some programs do not have square windows, which makes changing their size slightly tricky as they do not have borders.

11.2.2.3 Input Focus

The input focus determines which window receives the keyboard input. The way in which your standard window manager has been set up for you is called "click to focus". This means you have to click on the window you want to give keyboard input. The second way of determining the keyboard focus is called "auto keyboard focus". In this mode the window which is currently pointed at by the mouse receives all keyboard input.

To change the focus policy it is necessary to change the window managers resource file.

11.2.2.4 Raising of Windows

Although clicking on a window usuall raises it, when using "auto keyboard focus" this is not necessarily the case. The keyboard focus can be on a window which is partially hidden by another window.

Raising of windows can either be done by clicking on a window or automatically when it receives the input focus. In "auto raise mode" a window is rased automatically after a set number of seconds when the mouse pointer points to it.

Toggling the "auto raise mode" on or off can be done in the window managers resource file.

11.2.2.5 Menus

Window managers provide window specific menus and general menus. The window specific menus determine the state of a window. Mwm provides the following window menu:

Figure 11.1 MWM Window Menu

Further general menus can be specified. These menus appear when pressing a mouse button while the pointer is not in a window:

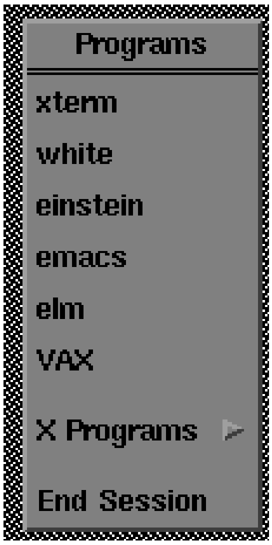


Figure 11.2 MWM Button 1 Menu

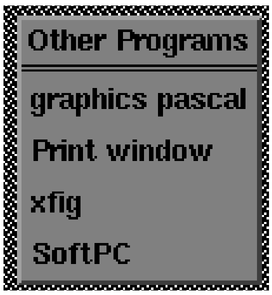


Figure 11.3 MWM Button 2 Menu

Figure 11.4 MWM Button 3 Menu

All the menus can be changed by specifying them in the resource file.

11.2.2.6 Virtual Desktops

Virtual desktops simulate a bigger desktop than is actually physically available. This makes it possible to have the mail program on one part of the virtual desktop, project number one on another part of the desktop and project number two on a third part desktop in such a way that only one of the parts is visible at one time.

11.2.2.7 Icon Management

Most window managers that implement icon management give the user the alternative of either simply placing the icons on the desktop or placing them in a special window called the icon box. Window mangers can also be told to place icons in certain areas of the desktop, e.g. top, bottom, left or right side.

11.3 Authorization

Authorization is concerned with determining who is allowed to use an X display. An X terminal is usually a separate computer from the one running your programs. Authorization mechanisms stop other users on the computer to display their programs on your display.

There are two ways of authorization. The easiest allows any user on a specified computer to display on an X terminal, whereas the other allows only users with a “magic cookie” to display on an X terminal.

11.3.1 xhost - Allowing computers access

The xhost program can disable or enable access control for all computers or certain computers. Owners of "magic cookies" always have access regardless of the setting for computers.

To check which computers have access to your display:

```
$xhost
access control enabled, only authorized clients can connect
$
```

To allow a certain computer to use your display:

```
$xhost +platon
platon being added to access control list
$xhost
access control enabled, only authorized clients can connect
INET:platon
$
```

To allow all computers to use your display, i.e. disable all access control:

```
$xhost +
access control disabled, clients can connect from any host
$xhost
access control disabled, clients can connect from any host
INET:platon
$
```

To cancel access control for all computers except explicitly named ones:

```
$xhost -
access control enabled, only authorized clients can connect
$xhost
access control enabled, only authorized clients can connect
INET:platon
$
```

To cancel access control for named computers:

```
$xhost -platon
platon being removed from access control list
$xhost
access control enabled, only authorized clients can connect
$
```

As giving access control to whole computers leaves a security loophole, access control can be done by giving out "magic cookies".

11.3.2 X Authority - Allowing users access

X authority allows for much finer access control than the xhost mechanism. Everytime a user logs onto an X terminal a "magic cookie" is generated and stored in a file in the users home directory. This "magic cookie" allows access to the X terminal. It is not user specific so it can be passed on to others. The "magic cookie" is also useful when telnetting to another computer. It can be passed to that computer and the remote computer can display its windows on the local X terminal.

There are two sensible ways to make "magic cookies" available to another computer. The first one is trivial. If two computers use the same file space then they will also use the same X authority file and the user has access to the X terminal from both computers. If two computers do not share the same file space the "magic cookie" has to be transfered manually.

11.3.2.1 The xauth program

The xauth program has quite advanced functions for combinig X authority files, but we will only look at the two basic functions.

To list all magic cookies, we can use xauth's list command:

```
$xauth list
einstein:0  MIT-MAGIC-COOKIE-1  4db7c821fc255ba49a7213aa77f7db58
asti:0  MIT-MAGIC-COOKIE-1  4db7c821fc255ba49a7213aa77f7db58
nietzsche:0  MIT-MAGIC-COOKIE-1  4db7c821fc255ba49a7213aa77f7db58
einstein/unix:0  MIT-MAGIC-COOKIE-1  4db7c821fc255ba49a7213aa77f7db58
pg3.cs.rhbnc.ac.uk:0  MIT-MAGIC-COOKIE-1  d563b7dc040abdb0ac4ee90ab81c2cfb
$
```

Usually the first computer is the last one added to the top of the list. It is added when logging in. To find out properly which display we are using we can check the DISPLAY environment variable:

```
$export | grep DISPLAY
declare -x DISPLAY="einstein:0.0"
$
```

Now we can telnet to the other computer and give it the magic cookie:

```
$telnet platon
Trying 134.219.96.1...
Connected to platon.
Escape character is '^]'.

Digital UNIX (platon.cs.rhbnc.ac.uk) (ttypc)

login: markst
Password:

You have mail.
alpha
$ xauth add einstein:0  MIT-MAGIC-COOKIE-1  4db7c821fc255ba49a7213aa77f7db58
$
```

We have now enabled access from the other computer, but we can still not use the remote computer's programs.

11.4 Remote clients

Once access has been granted to a remote client it is allowed to display things on an X terminal, yet because machines can be granted access to many X terminals they have to be told explicitly which one to use[2].

It is possible to tell each individual program which X terminal to use:

```
$xterm -display einstein:0 &
$
```

Note the & at the end of the command line. We still want to be able to use our current window, while opening a new one.

[2]Some modern telnet programs can work this out themselves though. Try checking the display as above.

Alternatively it is possible to set an environment variable, which tells all programs started from that shell to use a certain display:

```
$ export DISPLAY=einstein:0
$ xterm &
```

Any program that use the X-Windows display mechanism can be used in this way. Even window managers can be run on remote machines. Beware though that X-Windows needs a lot of bandwidth and is best not used over dialup lines.

Chapter 12

Other useful UNIX Commands

12.1 Finding out the System Time & Date

The system time and date can be displayed using the date command. In modern UNIX systems the current date and time are directly obtained from a number of atomic clocks via the Internet. The departmental server is syncronized with three atomic clocks and acts as the official campus time server.

12.2 Displaying the Online Documentation

The man program provides an interface to the online documentation, it is available as a text only version and an X-Windows version called xman. To find out more about a command, we can just type man followed by the command's name:

```
$ man man

man(1)                                                          man(1)

NAME
       man - format and display the on-line manual pages
       manpath - determine user's search path for man pages

SYNOPSIS
```

To get information without knowing the command, we can use the keyword option:

```
$ man -k manual
```

```
man (1)              - Format and display the on-line manual pages
whereis (1)          - Locate binary, manual, and or source for program
xman (1)             - Manual page display program for the X Window System
$
```

This gives us a list of applicable manual pages.

12.3 Text Search

UNIX provides a text search command called grep. It searches text for regular expressions. The simplest regular expressions are simply strings of characters. Grep will print out all lines containing the search expression, unless told otherwise:

```
$cat d
This will be the content of
our new file!
$grep our d
out new file!
$
```

We can specify multiple files on the command line:

```
$cat e
This will be the content of
our old file!
$grep our d e
d:our new file!
e:our old file!
$
```

Or we can count the number of occurrences:

```
$grep -c our d e
d:1
e:1
$
```

12.3.1 Regular Expressions

Regular expression are a lot more powerful though. We can search for lines containing the word 'file' not preceeded by 'old'. We can search for empty lines, lines which contain one word or another. To be able to do this it is necessary to understand the syntax and grammar of regular expressions. A regular expression is always tried to be matched against some text. If a match succeeds grep will print that line.

- A '.' can stand for any character
- A list of characters enclosed by '[' and ']' matches any single character in that list. If the first character in the list is a '^', then it matches any single character not in the list.
- The '^' matches the beginning of a line and the '$' matches the end of a line.
- To match one or another regular expression,both regular expressions can be bracketed with '(' and ')' and separated by '|'.
- A regular expression is optional and matched at most once, if it is followed by a '?'.
- A regular expression is matched zero or many times if is followed by a '*'.
- A regular expression is matched one or many times if is followed by a '+'.
- A regular expression is matched n times if is followed by a '{n}'.
- A regular expression is matched n or more times if is followed by a '{n,}'.
- A regular expression is matched m or less times if is followed by a '{,m}'.
- A regular expression is matched at least n times and at most m times if is followed by a '{n,m}'.

In some versions of grep the following characters have to be preceeded by a '\': ?, +, {, |, (,).

Examples

- a - matches exactly one a
- ^ a $ - matches a line containing only a

◇ ^ [ab] $ - matches a line containing one a or one b

◇ a* - matches zero or more a's

◇ a{3,5} - matches 3, 4 or 5 a's

◇ ([ab]c) - matches 'ac' or 'bc'

◇ ([ab]c)—b - matches 'ac', 'bc' or 'b'

◇ [0-9] - matches any single numeral

◇ [a-z] - matches a single lower case alphal character

◇ [a-zA-Z] - matches any single alphabetical character

◇ (very)? happy days - matches 'very happy days' or 'happy days'

Always remember that, if you are in doubt of the ordering in which the regular expression is evaluated,insert brackets.

12.4 Scheduling programs to run unattended

The cron utility allows the user to schedule programs for execution at a later time. Programs can be scheduled to run once using ths at command or at regular intervalls using the crontab command. The cron utility runs the scheduled program and mails its output back to the user.

Programs which are good examples for the use of cron are very large simulation programs, which can be run late at night, or file transfer programs, which might work better at night, when the internet is less used.

When submitting programs to cron it is important to bear in mind, that the user's shell will not be run and no paths etc. will be set up.[1]

12.4.1 at

The at command allows programs to be scheduled at a certain time. The at command takes a time as its argument. After pressing return the commands to be executed can be entered, followed by a CTRL-D on an emty line.

The time can be specified in the form HH:MM[2], followed by an optional date in the form month name day of month, followed by an optional increment. The

[1] Also note that at and crontab work slightly different on different systems.
[2] DEC's UNIX also allows you to specify now instead of a time.

increment is preceeded by a '+' followed by a number or the special wordnext and one of the words: minute[s], hour[s], day[s], week[s], month[s], year[s].

To find out what jobs are scheduled with at we can run at -l. To remove jobs from the queue, we can run at -r followed by the jobs number gained from at -l.

Examples

To run a job at 5 pm we can use: at 17:00

To run a job at 5 pm next week we can use: at 17:00 next week

To run a job at 5 pm on the 27^{th} of March we can use: at 17:00 march 27

To run a job at 5 pm two weeks after the 27^{th} of March we can use: at 17:00 march 27 + 2 weeks

12.4.2 crontab

Crontab is used to run recurring jobs at scheduled times. It is slightly more complicated to use than at, but is a lot more powerful. Every user, who submits jobs with crontab, has a special file which contains all the user's jobs for cron. This file has a special format. It is divided into six columns:

1. Specifies the minute (0-59).
2. Specifies the hour (0-23).
3. Specifies the day of the month (1-31).
4. Specifies the month of the year (1-12).
5. Specifies the day of the week (0-6 for Sunday to Saturday).
6. Specifies the shell command to be executed.

The following values can be specified in the fields that indicate the time:

- An integer (within the appropriate range of values)
- Two integers separated by a dash to indicate an inclusive range.
- A list of integers separated by commas.
- An asterisk to select all possible values.

Note that it is possible to specify the days on which a command is run in two fields (day of month and day of week). It is possible to specify both fields or only one field. To use only one field to specify the days, the other field should contain an asterisk.

To edit your crontab entry run crontab -e. To list your entries run crontab -l. To remove all your crontab entries run crontab -r.

Examples

The following entry will run command at midnight on the first and fifteenth day of each month as well as every Monday:

```
0 0 1,15 * 1 command
```

The following entry will run command every five minutes:

```
0,5,10,15,20,25,30,35,40,45,50,55 * * * * command
```

The following entry will run command at midnight during every day of february:

```
0 0 * 2 * command
```

Appendix A

Manual Pages

A.1 chmod - Changing File Access Permissions

NAME

chmod - change the access permissions of files

SYNOPSIS

chmod [-Rcfv] [--recursive] [--changes] [--silent] [--quiet] [--verbose] [--help] [--version] mode file...

DESCRIPTION

This manual page documents the GNU version of chmod. chmod changes the permissions of each given file according to mode, which can be either a symbolic representation of changes to make, or an octal number representing the bit pattern for the new permissions.

The format of a symbolic mode is '[ugoa...][[+-=][rwxXstugo...]...][,...]'. Multiple symbolic operations can be given, separated by commas.

A combination of the letters 'ugoa' controls which users' access to the file will be changed: the user who owns it (u), other users in the file's group (g), other users not in the file's group (o), or all users (a). If none of these are given, the effect is as if 'a' were given, but bits that are set in the umask are not affected.

The operator '+' causes the permissions selected to be added to the existing permissions of each file; '-' causes them to be removed; and '=' causes them to be the only permissions that the file has.

The letters 'rwxXstugo' select the new permissions for the affected users: read (r), write (w), execute (or access for directories) (x), execute only if the file is a directory or already has execute permission for some user (X), set user or group ID on execution (s), save program text on swap device (t), the permissions that the user who owns the file currently has for it (u), the permissions that other users in the file's group have for it (g), and the permissions that other users not in the file's group have for it (o).

A numeric mode is from one to four octal digits (0-7), derived by adding up the bits with values 4, 2, and 1. Any omitted digits are assumed to be leading zeros. The first digit selects the set user ID (4) and set group ID (2) and save text image (1) attributes. The second digit selects permissions for the user who owns the file: read (4), write (2), and execute (1); the third selects permissions for other users in the file's group, with the same values; and the fourth for other users not in the file's group, with the same values.

chmod never changes the permissions of symbolic links; the chmod system call cannot change their permissions. This is not a problem since the permissions of symbolic links are never used. However, for each symbolic link listed on the command line, chmod changes the permissions of the pointed-to file. In contrast, chmod ignores symbolic links encountered during recursive directory traversals.

OPTIONS

-c, --changes
: Verbosely describe only files whose permissions actually change.

-f, --silent, --quiet
: Do not print error messages about files whose permissions cannot be changed.

-v, --verbose
: Verbosely describe changed permissions.

-R, --recursive
: Recursively change permissions of directories and their contents.

--help
: Print a usage message on standard output and exit successfully.

--version

Print version information on standard output then exit successfully.

A.2 ed - The Line Editor

```
NAME
       ed, red - text editor

SYNOPSIS
       ed [-] [-Gs] [-p string] [file]

       red [-] [-Gs] [-p string] [file]

DESCRIPTION
       ed  is a line-oriented text editor.  It is used to create,
       display, modify and otherwise manipulate text files.   red
       is  a restricted ed: it can only edit files in the current
       directory and cannot execute shell commands.

       If invoked with a file argument, then a copy  of  file  is
       read  into  the editor's buffer.  Changes are made to this
       copy and not directly to file itself.  Upon  quitting  ed,
       any  changes  not explicitly saved  with a 'w' command are
       lost.

       Editing is done in two distinct modes: command and  input.
       When  first  invoked, ed is in command mode.  In this mode
       commands are read from the standard input and executed  to
       manipulate  the  contents of the editor buffer.  A typical
       command might look like:

              ,s/old/new/g

       which replaces all occurences of the string old with  new.

       When  an input command, such as 'a' (append), 'i' (insert)
       or 'c' (change), is given, ed enters input mode.  This  is
       the primary means of adding text to a file.  In this mode,
       no commands are available; instead, the standard input  is
       written  directly  to the editor buffer.  Lines consist of
       text up to and including a newline character.  Input  mode
       is  terminated by entering a single period  (.) on a line.

       All ed commands operate on whole lines or ranges of lines;
```

e.g., the ‘d’ command deletes lines; the ‘m’ command moves lines, and so on. It is possible to modify only a portion of a line by means of replacement, as in the example above. However even here, the ‘s’ command is applied to whole lines at a time.

In general, ed commands consist of zero or more line addresses, followed by a single character command and possibly additional parameters; i.e., commands have the structure:

```
[address [,address]]command[parameters]
```

The address(es) indicate the line or range of lines to be affected by the command. If fewer addresses are given than the command accepts, then default addresses are supplied.

OPTIONS

-G Forces backwards compatibility. Affects the commands ‘G’, ‘V’, ‘f’, ‘l’, ‘m’, ‘t’, and ‘!!’.

-s Suppresses diagnostics. This should be used if ed’s standard input is from a script.

-p string
Specifies a command prompt. This may be toggled on and off with the ‘P’ command.

file Specifies the name of a file to read. If file is prefixed with a bang (!), then it is interpreted as a shell command. In this case, what is read is the standard output of file executed via sh(1). To read a file whose name begins with a bang, prefix the name with a backslash (\). The default filename is set to file only if it is not prefixed with a bang.

LINE ADDRESSING

An address represents the number of a line in the buffer. ed maintains a current address which is typically supplied to commands as the default address when none is specified. When a file is first read, the current address is set to the last line of the file. In general, the current address is set to the last line affected by a command.

A line address is constructed from one of the bases in the list below, optionally followed by a numeric offset. The offset may include any combination of digits, operators (i.e., +, - and ^) and whitespace. Addresses are read from left to right, and their values are computed relative to the current address.

One exception to the rule that addresses represent line numbers is the address 0 (zero). This means "before the first line," and is legal wherever it makes sense.

An address range is two addresses separated either by a comma or semicolon. The value of the first address in a range cannot exceed the value of the the second. If only one address is given in a range, then the second address is set to the given address. If an n-tuple of addresses is given where n > 2, then the corresponding range is determined by the last two addresses in the n-tuple. If only one address is expected, then the last address is used.

Each address in a comma-delimited range is interpreted relative to the current address. In a semicolon-delimited range, the first address is used to set the current address, and the second address is interpreted relative to the first.

The following address symbols are recognized.

. The current line (address) in the buffer.

```
$       The last line in the buffer.

n       The nth, line in the buffer where n is a number in
        the range [0,$].


-

^       The previous line.  This is equivalent to  -1  and
        may be repeated with cumulative effect.


-n

^n      The  nth  previous line, where n is a non-negative
        number.


+       The next line.  This is equivalent to +1  and  may
        be repeated with cumulative effect.


+n

whitespace n
        The  nth next line, where n is a non-negative num-
        ber.  Whitespace followed by a number n is  inter-
        preted as +n.


,

%       The  first through last lines in the buffer.  This
        is equivalent to the address range 1,$.


;       The current through  last  lines  in  the  buffer.
        This is equivalent to the address range .,$.


/re/    The  next  line  containing the regular expression
```

re. The search wraps to the beginning of the buffer and continues down to the current line, if necessary. // repeats the last search.

?re? The previous line containing the regular expression re. The search wraps to the end of the buffer and continues up to the current line, if necessary. ?? repeats the last search.

'lc The line previously marked by a 'k' (mark) command, where lc is a lower case letter.

REGULAR EXPRESSIONS

Regular expressions are patterns used in selecting text. For example, the ed command

```
g/string/
```

prints all lines containing string. Regular expressions are also used by the 's' command for selecting old text to be replaced with new.

In addition to a specifying string literals, regular expressions can represent classes of strings. Strings thus represented are said to be matched by the corresponding regular expression. If it is possible for a regular expression to match several strings in a line, then the left-most longest match is the one selected.

The following symbols are used in constructing regular expressions:

c Any character c not listed below, including '{', '}', '(', ')', '<' and '>', matches itself.

\c A backslash-escaped character c other than '{',

’}’, ‘(’, ‘)’, ‘<’, ‘>’, ‘b’, ’B’, ‘w’, ‘W’, ‘+’, and ‘?’ matches itself.

Matches any single character.

[char-class]

Matches any single character in char-class. To include a ‘]’ in char-class, it must be the first character. A range of characters may be specified by separating the end characters of the range with a ‘-’, e.g., ‘a-z’ specifies the lower case characters. The following literal expressions can also be used in char-class to specify sets of characters:

```
[:alnum:]  [:cntrl:]  [:lower:]  [:space:]
[:alpha:]  [:digit:]  [:print:]  [:upper:]
[:blank:]  [:graph:]  [:punct:]  [:xdigit:]
```

If ‘-’ appears as the first or last character of char-class, then it matches itself. All other characters in char-class match themselves.

Patterns in char-class of the form:

```
[.col-elm.] or,   [=col-elm=]
```

where col-elm is a collating element are interpreted according to locale(5) (not currently supported). See regex(3) for an explanation of these constructs.

[^char-class]

Matches any single character, other than newline, not in char-class. char-class is defined as above.

^ If ‘^’ is the first character of a regular expression, then it anchors the regular expression to the beginning of a line. Otherwise, it matches itself.

$ If '$' is the last character of a regular expression, it anchors the regular expression to the end of a line. Otherwise, it matches itself.

\(re\) Defines a (possibly null) subexpression re. Subexpressions may be nested. A subsequent back-reference of the form '\n', where n is a number in the range [1,9], expands to the text matched by the nth subexpression. For example, the regular expression '\(a.c\)\1' matches the string 'abcabc', but not 'abcadc'. Subexpressions are ordered relative to their left delimiter.

* Matches the single character regular expression or subexpression immediately preceding it zero or more times. If '*' is the first character of a regular expression or subexpression, then it matches itself. The '*' operator sometimes yields unexpected results. For example, the regular expression 'b*' matches the beginning of the string 'abbb', as opposed to the substring 'bbb', since a null match is the only left-most match.

\{n,m\}

\{n,\}

\{n\} Matches the single character regular expression or subexpression immediately preceding it at least n and at most m times. If m is omitted, then it matches at least n times. If the comma is also omitted, then it matches exactly n times. If any of these forms occurs first in a regular expression or subexpression, then it is interpreted literally (i.e., the regular expression '\{2\}' matches the string '{2}', and so on).

\<

\> Anchors the single character regular expression or subexpression immediately following it to the beginning (\<) or ending (\>) of a word, i.e., in ASCII, a maximal string of alphanumeric characters, including the underscore (_).

The following extended operators are preceded by a backslash (\) to distinguish them from traditional ed syntax.

\‘
\’ Unconditionally matches the beginning (\‘) or ending (\’) of a line.

\? Optionally matches the single character regular expression or subexpression immediately preceding it. For example, the regular expression ‘a[bd]\?c’ matches the strings ‘abc’, ‘adc’ and ‘ac’. If \? occurs at the beginning of a regular expressions or subexpression, then it matches a literal ‘?’.

\+ Matches the single character regular expression or subexpression immediately preceding it one or more times. So the regular expression ‘a+’ is shorthand for ‘aa*’. If \+ occurs at the beginning of a regular expression or subexpression, then it matches a literal ‘+’.

\b Matches the beginning or ending (null string) of a word. Thus the regular expression ‘\bhello\b’ is equivalent to ‘\<hello\>’. However, ‘\b\b’ is a valid regular expression whereas ‘\<\>’ is not.

\B Matches (a null string) inside a word.

\w Matches any character in a word.

\W Matches any character not in a word.

COMMANDS

All ed commands are single characters, though some require additonal parameters. If a command's parameters extend over several lines, then each line except for the last must be terminated with a backslash (\).

In general, at most one command is allowed per line. However, most commands accept a print suffix, which is any of 'p' (print), 'l' (list) , or 'n' (enumerate), to print the last line affected by the command.

An interrupt (typically ^C) has the effect of aborting the current command and returning the editor to command mode.

ed recognizes the following commands. The commands are shown together with the default address or address range supplied if none is specified (in parenthesis).

(.)a — Appends text to the buffer after the addressed line, which may be the address 0 (zero). Text is entered in input mode. The current address is set to last line entered.

(.,.)c — Changes lines in the buffer. The addressed lines are deleted from the buffer, and text is appended in their place. Text is entered in input mode. The current address is set to last line entered.

(.,.)d — Deletes the addressed lines from the buffer. If there is a line after the deleted range, then the current address is set to this line. Otherwise the current address is set to the line before the deleted range.

e file — Edits file, and sets the default filename. If file is not specified, then the default filename is used. Any lines in the buffer are deleted before the new file is read. The current address is set to the last line read.

e !command
: Edits the standard output of '!command', (see !command below). The default filename is unchanged. Any lines in the buffer are deleted before the output of command is read. The current address is set to the last line read.

E file
: Edits file unconditionally. This is similar to the e command, except that unwritten changes are discarded without warning. The current address is set to the last line read.

f file
: Sets the default filename to file. If file is not specified, then the default unescaped filename is

printed.

(1,$)g/re/command-list
: Applies command-list to each of the addressed lines matching a regular expression re. The current address is set to the line currently matched before command-list is executed. At the end of the 'g' command, the current address is set to the last line affected by command-list.

Each command in command-list must be on a separate line, and every line except for the last must be terminated by a backslash (\). Any commands are allowed, except for 'g', 'G', 'v', and 'V'. A newline alone in command-list is equivalent to a 'p' command.

(1,$)G/re/
: Interactively edits the addressed lines matching a regular expression re. For each matching line, the line is printed, the current address is set, and the user is prompted to enter a command-list. At the end of the 'G' command, the current address is set to the last line affected by (the last) command-list.

The format of command-list is the same as that of the 'g' command. A newline alone acts as a null command list. A single '&' repeats the last non-null command list.

H — Toggles the printing of error explanations. By default, explanations are not printed. It is recommended that ed scripts begin with this command to aid in debugging.

h — Prints an explanation of the last error.

(.)i — Inserts text in the buffer before the current line. Text is entered in input mode. The current address is set to the last line entered.

(.,.+1)j
Joins the addressed lines. The addressed lines are deleted from the buffer and replaced by a single line containing their joined text. The current address is set to the resultant line.

(.)klc — Marks a line with a lower case letter lc. The line can then be addressed as 'lc (i.e., a single quote followed by lc) in subsequent commands. The mark is not cleared until the line is deleted or otherwise modified.

(.,.)l — Prints the addressed lines unambiguously. If invoked from a terminal, ed pauses at the end of each page until a newline is entered. The current address is set to the last line printed.

(.,.)m(.)
Moves lines in the buffer. The addressed lines are moved to after the right-hand destination address, which may be the address 0 (zero). The current address is set to the last line moved.

(.,.)n Prints the addressed lines along with their line numbers. The current address is set to the last line printed.

(.,.)p Prints the addressed lines. If invoked from a terminal, ed pauses at the end of each page until a newline is entered. The current address is set to the last line printed.

P Toggles the command prompt on and off. Unless a prompt was specified by with command-line option -p string, the command prompt is by default turned off.

q Quits ed.

Q Quits ed unconditionally. This is similar to the q command, except that unwritten changes are discarded without warning.

($)r file
Reads file to after the addressed line. If file is not specified, then the default filename is used. If there was no default filename prior to the command, then the default filename is set to file. Otherwise, the default filename is unchanged. The current address is set to the last line read.

($)r !command
Reads to after the addressed line the standard output of '!command', (see the !command below). The default filename is unchanged. The current address is set to the last line read.

(.,.)s/re/replacement/
(.,.)s/re/replacement/g
(.,.)s/re/replacement/n
Replaces text in the addressed lines matching a regular expression re with replacement. By

default, only the first match in each line is replaced. If the ‘g’ (global) suffix is given, then every match to be replaced. The ‘n’ suffix, where n is a postive number, causes only the nth match to be replaced. It is an error if no substitutions are performed on any of the addressed lines. The current address is set the last line affected.

re and replacement may be delimited by any character other than space and newline (see the ‘s’ command below). If one or two of the last delimiters is omitted, then the last line affected is printed as though the print suffix ‘p’ were specified.

An unescaped ‘&’ in replacement is replaced by the currently matched text. The character sequence ‘\m’, where m is a number in the range [1,9], is replaced by the mth backreference expression of the matched text. If replacement consists of a single ‘%’, then replacement from the last substitution is used. Newlines may be embedded in replacement if they are escaped with a backslash (\).

(.,.)s Repeats the last substitution. This form of the ‘s’ command accepts a count suffix ‘n’, or any combination of the characters ‘r’, ‘g’, and ‘p’. If a count suffix ‘n’ is given, then only the nth match is replaced. The ‘r’ suffix causes the regular expression of the last search to be used instead of the that of the last substitution. The ‘g’ suffix toggles the global suffix of the last substitution. The ‘p’ suffix toggles the print suffix of the last substitution The current address is set to the last line affected.

(.,.)t(.)

Copies (i.e., transfers) the addressed lines to after the right-hand destination address, which may be the address 0 (zero). The current address is set to the last line copied.

u
: Undoes the last command and restores the current address to what it was before the command. The global commands 'g', 'G', 'v', and 'V'. are treated as a single command by undo. 'u' is its own inverse.

(1,$)v/re/command-list
: Applies command-list to each of the addressed lines not matching a regular expression re. This is similar to the 'g' command.

(1,$)V/re/
: Interactively edits the addressed lines not matching a regular expression re. This is similar to the 'G' command.

(1,$)w file
: Writes the addressed lines to file. Any previous contents of file is lost without warning. If there is no default filename, then the default filename is set to file, otherwise it is unchanged. If no filename is specified, then the default filename is used. The current address is unchanged.

(1,$)wq file
: Writes the addressed lines to file, and then executes a 'q' command.

(1,$)w !command
: Writes the addressed lines to the standard input of '!command', (see the !command below). The default filename and current address are unchanged.

(1,$)W file

Appends the addressed lines to the end of file. This is similar to the 'w' command, expect that the previous contents of file is not clobbered. The current address is unchanged.

(.)x Copies (puts) the contents of the cut buffer to after the addressed line. The current address is set to the last line copied.

(.,.)y Copies (yanks) the addressed lines to the cut buffer. The cut buffer is overwritten by subsequent 'y', 's', 'j', 'd', or 'c' commands. The current address is unchanged.

(.+1)zn Scrolls n lines at a time starting at addressed line. If n is not specified, then the current window size is used. The current address is set to the last line printed.

!command

Executes command via sh(1). If the first character of command is '!', then it is replaced by text of the previous '!command'. ed does not process command for backslash (\) escapes. However, an unescaped '%' is replaced by the default filename. When the shell returns from execution, a '!' is printed to the standard output. The current line is unchanged.

(.,.)# Begins a comment; the rest of the line, up to a newline, is ignored. If a line address followed by a semicolon is given, then the current address is set to that address. Otherwise, the current address is unchanged.

($)= Prints the line number of the addressed line.

(.+1)newline

Prints the addressed line, and sets the current address to that line.

FILES

/tmp/ed.*	Buffer file
ed.hup	The file to which ed attempts to write the buffer if the terminal hangs up.

LIMITATIONS

ed processes file arguments for backslash escapes, i.e., in a filename, any characters preceded by a backslash (\) are interpreted literally.

If a text (non-binary) file is not terminated by a newline character, then ed appends one on reading/writing it. In the case of a binary file, ed does not append a newline on reading/writing.

per line overhead: 4 ints

DIAGNOSTICS

When an error occurs, if ed's input is from a regular file or here document, then it exits, otherwise it prints a '?' and returns to command mode. An explanation of the last error can be printed with the 'h' (help) command.

Attempting to quit ed or edit another file before writing a modified buffer results in an error. If the command is entered a second time, it succeeds, but any changes to the buffer are lost.

ed exits with 0 if no errors occurred; otherwise >0.

A.3 vi - Visual editor

NAME

vi, vedit, view - Edits files with a full screen display

SYNOPSIS

```
vi [-ls] [-R] | [-r] [-c subcommand] [-t tag] [-wnumber] [+subcommand] [-]
     [file ...]

view [-l] [-t tag] [-c subcommand] [-wnumber] [+subcommand] [-] [file ...]

vedit [-l] [-R] | [-r] [-t tag] [-wnumber] [-] [file ...]
```

The vi command is a display editor that is based on an underlying line editor (ex).

FLAGS

-c subcommand
: Executes the specified ex subcommand (command) before displaying the file for which the editor was invoked.

-l
: Indents appropriately for LISP code, and accepts the (,), {, }, [, and] characters (parentheses, braces, and brackets) as text rather than interpreting them as vi subcommands. The LISP modifier is active in open or visual modes.

-r[file]
: Recovers file after an editor or system crash. If you do not specify a file, vi displays a list of all saved files.

-s
: Invokes vi in open mode. (Silent mode.)

-R
: Sets the readonly option to protect the file against overwriting.

-t tag
: Edits the file containing the tag and positions the editor at its definition. To use this flag, you must first create a database of function names and their locations using the ctags command.

-v Invokes the visual editor. When the -v flag is specified, an enlarged set of subcommands is available, including screen editing and cursor movement features.

-wnumber
Sets the default window size to number. This is useful when you use the editor over a low-speed line.

- Suppresses all interactive user feedback. If you use this flag, file input/output errors do not generate an error message.

+[subcommand]
Performs the ex subcommand before editing begins. If you do not specify subcommand, the cursor is placed on the first line of the file.

DESCRIPTION

The ex editor subcommands can be used within the vi editor, because vi is based on ex. The view command is a read-only version of vi; the readonly option is set to protect files during browsing. The vedit command is a version of vi intended for beginners. In it, the report option is set to 1, the showmode option is set, and the novice option (making vedit a line editor, rather than a screen editor) is set. (For more information on these options, see Setting Options.)

The file argument specifies the files to be edited. If you supply more than one file on the command line, vi edits each file in the order specified.

When you use vi, changes you make to a file are reflected on your display. The position of the cursor on the display indicates its position within the file. The subcommands affect the file at the cursor position.

Limitations of the vi Editor

The maximum limits of the vi editor are as follows:

+ 2048 bytes per line

+ 256 bytes per global command list

- 128 bytes in the previous inserted and deleted text
- 128 bytes in a shell escape command
- 128 bytes in a string-valued option
- 30 bytes in a tag name
- 128 map macros with 2048 bytes total.

Editing Modes

The vi editor has the following operational modes:

Command mode

When you start the vi editor, it is in Command mode. Any subcommand can be entered from this mode, except commands that can only be used in the Text Input mode (those subcommands that make corrections during text insertion). When subcommands and the other modes end, vi returns to Command mode. Pressing the <Esc> key cancels a partial subcommand.

Text Input mode

Entered by the a, A, i, I, o, O, cx (where x represents the scope of the subcommand), C, s, S, and R subcommands. After entering one of these commands, you can enter text into the editing buffer at the current cursor position. To return to Command mode, press <Esc> for normal exit or press the Interrupt key sequence to end abruptly.

Last Line mode

Some subcommands (those with the prefix : (colon), / (slash), ? (question mark), or !!) read input on a line displayed at the bottom of the screen. When you enter the initial character, vi places the cursor at the bottom of the screen, where you enter the remaining characters of the command. Press <Return> to perform the subcommand and enter the Interrupt key sequence to cancel it.

When !! is used, the cursor moves only after the two exclamation points are entered. When : is used, special meaning is given to the following characters if used as counts before a command:

% All lines, regardless of cursor position

$ Last line

. Current line

Setting Options

The vi editor allows you to customize options so that you can use the editor for a specific task. Use the set command to set or change an option. To view the current setting of options, enter :set all while in vi Command mode.

Some options are set to a string or a number value; other options are simply turned on or off. To change an option that is set to a value, enter a command in the form :set option=value. To toggle an option that can be set to on or off, enter a line of the form :set option to set it to on or :set nooption to set it to off.

Options can be abbreviated in a set command. The following table lists some of vi's options, along with abbreviations and descriptions:

Option	Abbreviation	Description
[no]autoindent	ai	Indents automatically in Text mode to the indentation on the previous line by using the spacing between tab stops specified by the shiftwidth option. The default is noai. To back the cursor up to the previous tab stop, type <Ctrl-d>. This option is not in

		effect for global commands.
[no]autoprint	ap	Prints the current line after any command that changes the editing buffer. The default is ap. This option applies only to the last command in a sequence of commands on a single line, and is not in effect for global commands.
[no]autowrite	aw	Writes the editing buffer to the file automatically before the :n, :ta, <Ctrl-a>, and ! subcommands if the editing buffer was changed since the last write command. The default is noaw.
[no]beautifying text	bf	Prevents user from entering control characters (except for tab, newline, and formfeed) in the editing buffer

		during text entry. The default is nobf. This option does apply to command input.
directory	dir=	Displays the directory that contains the editing buffer. The default is dir=/var/tmp.
[no]edcompatible	ed	Retains global (g) and confirms (c) subcommand suffixes during multiple substitutions and causes the read (r) suffix to work like the r subcommand. The default is noed.
[no]errorbells	eb	Precedes error messages with an <Alert> character. Setting this option off (noeb) does not suppress the alerting in visual mode. The default is noeb.
[no]exrc	"!	

		If not set, ignores any .exrc file in the current directory during initialization, unless the current directory is that named by the HOME variable. The default is noexrc.
[no]flash	fl	Uses visual flash rather than audible bell. The default is fl.
hardtabs	ht=	Tells vi the distance between the hardware tab stops on your display. The default is ht=8.
[no]ignorecase	ic	Ignores the distinction between uppercase and lowercase while searching for regular expressions. The default is noic.
[no]lisp	lisp	Enters vi in LISP

		mode. In this mode, vi appropriately indents for LISP code and the (,), {, }, [[, and]]. The default is nolisp.
[no]list	list	
		Displays text with tabs and the end of lines marked. Tabs are displayed as ^I and the end of lines as $. The default is nolist.
[no]magic	magic	
		Treats the characters ., [, and * as special characters in scans. In Off mode, only the (,), and $ characters retain special meanings; however, special meaning of other characters can still be invoked by preceding the character with a \ (backslash). The default is magic.
[no]modeline	modeline	
		Runs an editor command line if found

in the first five and the last five lines of the file. An editor command line may be anywhere in a line. To be recognized as a command line, it must contain a space or a tab followed by the string ex: or vi:. The command line is ended by a second : (colon). The editor tries to interpret any data between the first and second : as editor commands. The default is nomodeline.

[no]number	nu	Displays lines prefixed with their line numbers. The default is nonu.
[no]optimize	opt	Speeds up the operation of terminals that do not have cursor addressing. The default is noopt.
paragraphs	para=	

		Defines macro names that start paragraphs. The default is para=IPLPPPQPP LIpplpipnpbp. Single letter nroff macros, such as .P must include the space as a quoted character if respecifying a paragraph.
[no]prompt	prompt	
		Prompts for command mode input with a : (colon). When not set, no prompt is displayed. The default is on.
readonly	"	
		Allows writing to a different file. In addition, the write can be forced by using the ! (exclamation point) character (see the editor command write). The default is off, unless the file lacks write permission or the -R flag is specified.
[no]redraw	re	

		Simulates a smart display on a dumb display. The default is nore.
[no]remap	remap	
		Enables following of map keys. For example, if remap is set, map u k map r u causes r to map to k. If noremap is set, r maps to u. The default is remap.
report	report=	
		Sets the number of repetitions of a command before a message is displayed. For subcommands that can produce a number of messages, such as global subcommands, the messages are displayed when the command is completed. The default is report=5.
scroll	scr=	
		Sets the number of lines to be scrolled when you scroll the screen

		up or down. The default scroll is one-half the size of the screen.
sections	sect=	
		Defines macro names that start sections. The default is sect=NHSHH HUuhsh+c. Single letter nroff macros, such as .P must include the space as a quoted character if respecifying a paragraph.
shell	sh=	
		Defines the shell for ! or :! commands. The default is the value of the SHELL environment variable.
shiftwidth	sw=	
		Sets the distance for the software tab stops used by autoindent, the shift commands (> and <), and the text input commands (<Ctrl-d> and <Ctrl-t>) to allow the editor to

		indent text and move back to a previous indentation. The default is sw=8.
[no]showmatch	sm	
		Shows the matching open parenthesis (or open brace { as you type the close parenthesis) or close brace }. The default is nosm.
[no]showmode	smd	
		Displays mode indicator at the bottom of the screen when in the insert or replace mode. The default is nosmd.
[no]slowopen	slow	
		Postpones updating the display during inserts. The default is noslow.
[no]sourceany		
		Allows the use of the source command on a file that a user does not own. The default is nosourceany.

tabstops	ts=	
		Sets the distance between tab stops when a file is displayed. The default is ts=8.
taglength	tl=	
		Determines length of tag.
[no]tags	tag	
		Specifies a list of possible filenames of tag files. The default is tags /usr/lib/tags.
term	term=	
		Sets the kind of display you are using. The default is term=$TERM, where $TERM is the value of the TERM shell variable.
[no]terse	terse	
		Allows vi to display the short form of messages. The default is noterse.
[no]timeout	to	

		Sets a time limit of 2 seconds on entry of characters. This limit allows the characters in a macro to be entered and processed as separate characters when timeout is set. To resume use of the macro, set notimeout. The default is to.
ttytype	tty=	Same as term.
[no]warn	warn	Displays a warning message before the ! subcommand executes a shell command if this is the first time you issued a shell command after a given set of changes were made in the editing buffer, but not written to a file. The default is warn.
window	wi=	Sets the number of lines displayed in one window of text. The default is

dependent on the baud rate at which you are operating: 600 baud or less / 8 lines, 1200 baud / 16 lines, higher speeds / full screen minus 1.

wrapmargin wm=

Sets the margin for automatic wordwrapping from one line to the next. A value of 0 indicates no wordwrapping. The default is wm=0.

[no]wrapscan ws

Allows string searches to wrap from the end of the editing buffer to the beginning. The default is ws.

wraptype wt=

Causes words to be wrapped in Japanese style. wraptype=general causes general-purpose wrap on word breaks, where word break is defined as whitespace or space between two non-ASCII characters. general is a combination of word and flexible. wraptype=word causes wrap on words. wraptype=rigid causes wrap on column and before closing punctuation. wraptype=flexible causes wrap on column, but closing punctuation may extend past the margin.

[no]writeany	wa	Turns off the checks usually made before a write command. The default is nowa.

Defining Macros

If you use a subcommand or sequence of subcommands frequently, you can create a macro that issues the subcommand or sequence when you call a macro. To create a macro, enter the sequence of subcommands into an editing buffer named with a letter of the alphabet. When used as buffer names, lowercase ASCII letters a through z overlay the contents of the buffer, while uppercase ASCII letters A through Z append text to the previous contents of the buffer, allowing the building of a macro piece by piece.

To invoke the macro, enter @x, where x is the letter name of the buffer. Enter @@ to repeat the last macro you invoked.

Mapping Keys

You can use the map command to set a keystroke to a subcommand or a sequence of subcommands. To set a key mapping, enter :map key subcommand where key is the key to which you want to assign a subcommand or sequence of subcommands and subcommand is the subcommand or sequence of subcommands. For example, to set @ to delete lines, enter:

```
:map @ dd
```

In this example, @ is the key to which the subcommand is assigned and dd is the subcommand.

In the next example, a subcommand sequence is mapped to a key:

```
:map * {>}
```

The * (asterisk) is the key to which the subcommand sequence is assigned and {>} is the subcommand sequence. The { (open brace) moves the cursor to the beginning of the paragraph and the > (right angle bracket) indents the paragraph to the next shiftwidth.

To display the list of the current key mappings while you are in Command mode, enter the :map command. You can also remove a key mapping. To remove a key mapping, enter :unmap string or :unmap! string where string is the string used after the :map command to set the key and subcommand sequence. For example, to remove key mapping for the previous example, enter:

```
:unmap *
```

If function keys are defined for your terminal, they can be put in a map or unmap command by typing <Ctrl-v> then pressing the desired key. In this way, function keys that are unused during editing can be mapped to useful editing subcommand sequences.

If the ! (exclamation point) character is appended to the command name map (map!), the mapping is effective during input mode rather than during visual mode.

Abbreviations

You can define abbreviations for long phrases that you use often. vi then automatically expands these abbreviations whenever you enter them in insert mode. Only single-byte characters can appear in abbr_string; abbr_string cannot contain Asian multi-byte characters.

To define an abbreviation, enter:

```
:abbr abbreviation phrase
```

abbreviation is the abbreviation you specify for the longer text specified by phrase. For example, to specify the abbreviation imho for the phrase In my humble opinion, enter:

```
:abbr imho In my humble opinion
```

Keeping a Customized Change

When you customize vi from the vi command line, the customized editor is in effect until you exit the editor. If you want to reuse such things as option settings and key mappings, you must put them in the .exrc file in your home directory or define the EXINIT environment variable. The vi editor processes the commands given in the EXINIT variable or reads the .exrc file each time you invoke it. Here is an example of an .exrc file:

```
set ai aw
set wm=5
map  @  dd
```

Users with both a .exrc file and an EXINIT environment variable will find that the vi editor no longer reads the .exrc file. This change was made to meet the specifications of XPG4.

The standard provides for an approximation of the old behavior. A new variable, named exrc, is defined. When this variable is set by the commands in the EXINIT environment variable, the editor reads .exrc in the current directory for additional startup commands. For example:

```
setenv EXINIT 'set ai terse magic bf wm=1 exrc'
```

Additionally, the editor refuses to read the .exrc file if its mode grants write permission to anyone other than the owner; this restriction prevents certain security breaches. No overt indication is given when such a refusal occurs.

vi Character Sets

The collation sequence, as defined by the value of the LC_COLLATE environment variable, defines the alphanumeric set used by your system. This table affects the performance of vi macros and subcommands.

The vi editor uses the collation sequence to distinguish between a small word and a big word. A small word is bounded by letters or numbers as defined in the collation table. For example, isn't is two small words. The ' (apostrophe) is not a number or an alphabetic character, and it bounds both the small word t and the small word isn. A big word is bounded by spaces, tabs, or newline indicators. For example, stop is a big word. For more information, see the section Moving to Words.

Subcommand Syntax

[named_buffer] [operator] [number] argument

Surrounding brackets indicate optional items.

[named_buffer]
    A temporary text storage area.

[operator]
    Specifies the subcommand or action; instructs the vi editor.

[number]
    A whole decimal value that specifies either the extent of the action or a line address. The vi editor interprets this number in one of the following ways:

1. Go to line number:
   5G
   10z<Return>

2. Go forward number columns.

```
25<Space>
```

3. Scroll number of lines:

```
10<Ctrl-d>
10<Ctrl-u>
```

4. Delete number lines:

```
6dd
```

5. % means all. To yank all lines:

```
%y
```

(The preceding command replaces 1,$y.)

argument
: Specifies what to act on. This can be a text object (a character, word, sentence, paragraph, section, or character string) or a text position (a line, position in the current line, or screen position).

Moving Within a File

Enter the following subcommands in Command mode. You can cancel an incomplete subcommand by pressing the <Esc> key.

<Left Arrow>, h, <Ctrl-h>
: Moves the cursor one character to the left.

<Down Arrow>, j, <Ctrl-j>, <Ctrl-n>
: Moves the cursor down one line, remaining in the same column.

<Up Arrow>, k, <Ctrl-p>
: Moves the cursor up one line, remaining in the same column.

<Right Arrow>, l <Space>
: Moves the cursor one character to the right.

Long lines: Lines over one screen width are wrapped but not broken. When using the Up Arrow or Page Up key, @ lines are added at the bottom of the screen when too few physical lines are available to display the complete line. The Down Arrow key moves the entire line off the screen at once.

Character Positioning Within a Line

Enter the following subcommands in Command mode.

^ Moves the cursor to the first nonspace character.

0 Moves the cursor to the beginning of the line.

$ Moves the cursor to the end of the line.

fx Moves the cursor to the next x character.

Fx Moves the cursor to the last x character.

tx Moves the cursor to one column before the next x character.

Tx Moves the cursor to one column after the last x character.

; Repeats the last f, F, t, or T subcommand.

, Repeats the last f, F, t, or T subcommand in the opposite direction.

number<Space>
Moves the cursor to the specified column.

Moving to Words

Enter the following subcommands in Command mode.

w Moves the cursor to the next small word.

b Moves the cursor to the previous small word.

e Moves the cursor to the end of the next small word.

W   Moves the cursor to the next big word.

B   Moves the cursor to the previous big word.

E   Moves the cursor to the end of the next big word.

Moving by Line Positioning

Enter the following subcommands in Command mode.

G   Moves to the line number given as preceding argument, or the end of the file if no preceding count is given.

H   Moves the cursor to the top line on the screen.

L   Moves the cursor to the last line on the screen.

M   Moves the cursor to the middle line on the screen.

+   Moves the cursor to the next line, at its first nonspace character.

-   Moves the cursor to the previous line, at its first nonspace character.

<Return>
    Moves the cursor to the next line, at its first nonspace character.

Moving to Sentences, Paragraphs, or Sections

Enter the following subcommands in Command mode. You can cancel an incomplete subcommand by pressing the <Esc> key.

(Places the cursor at the beginning of the previous sentence (or the previous S-expression if you are in LISP mode).

)   Places the cursor at the beginning of the next sentence (or the next S-expression if you are in LISP mode).

{   Places the cursor at the beginning of the previous paragraph (or at the next list if you are in LISP mode).

}   Places the cursor at the beginning of the next paragraph, at the next section if you are in C mode, or at the next list if you are in LISP mode.

| Requires a count; the cursor is placed in that column (if possible).

]] Places the cursor at the next section, or function if you are in LISP mode.

[[Places the cursor at the previous section, or function if you are in LISP mode.

Paging and Scrolling

<Ctrl-u>
Scrolls up (default 12 lines).

<Ctrl-d>
Scrolls down (default 12 lines).

<Ctrl-f>
Pages forward one screen.

<Ctrl-b>
Pages backward one screen.

<Ctrl-y>
Scrolls the window up one line.

<Ctrl-e>
Scrolls the window down one line.

<Ctrl-m>
Moves to the first non-white character in the next line. A count specifies the number of lines to go forward.

<Ctrl-t>
Inserts shiftwidth white space in input mode, if at the beginning of the line or preceded only by white space. This inserted space can only be backed over using <Ctrl-d>.

<Ctrl-[>

Cancels a partially formed command; sounds the bell if there is none.

In input mode, terminates input mode.

When entering a command on the bottom line of the screen (ex command line or search pattern with \ or ?), terminates input and executes command.

z+ Pages up.

z^ Pages down.

Searching for Patterns

The following commands allow you to search for patterns within a file. Patterns can be regular expressions as described for grep.

/pattern
Places the cursor at the next line containing pattern.

?pattern
Places the cursor at the next previous line containing pattern.

n Repeats the last search for pattern in the same direction.

N Repeats the last search for pattern in the opposite direction.

/pattern/+number
Places the cursor at the numberth line after the line matching pattern.

?pattern?-number
Places the cursor at the numberth line before the line matching pattern.

% Finds the parenthesis or brace that matches the one at the current cursor position.

Marking and Returning

Enter the following subcommands in Command mode. You can cancel an incomplete subcommand by pressing the <Esc> key.

‘‘ Moves the cursor to the same cursor position of the previous current line.

’’ Moves the cursor to the beginning of the previous current line.

mx Marks the current position with the letter specified by x.

‘x Moves the cursor to the same cursor position of line marked x.

’x Moves the cursor to the beginning of the line marked x.

Adjusting the Screen

Enter the following subcommands in Command mode. An incomplete subcommand can be canceled by pressing the <Esc> key.

<Ctrl-l>
Clears and redraws the screen.

<Ctrl-r>
Redraws the screen and eliminates blank lines marked with a @.

z<Return>
Redraws the screen with the current line at the top of the screen.

z- Redraws the screen with the current line at the bottom of the screen.

z. Redraws the screen with the current line at the center of the screen.

/pattern/z-
Redraws the screen with the line containing pattern at the bottom.

znumber<Return>
Makes the window number lines long.

Adding Text to a File--Text Input

The following subcommands are entered in Command mode and bring the vi editor into Text Input mode to allow you to add text to your file. End Text Input mode by pressing the <Esc> key.

atext
    Inserts text after the cursor.

Atext
    Adds text to the end of the line.

itext
    Inserts text before the cursor.

Itext
    Inserts text before the first nonspace character in the line.

o   Adds an empty line below the current line.

O   Adds an empty line above the current line.

Changing Text While in Input Mode

Use the following commands only while in Text Entry mode. They have different meanings in Command mode.

<Ctrl-h>
    Erases the last character.

<Ctrl-w>
    Erases the last small word. (For more information about small words, see the section vi Character Sets.)

\   Quotes the Erase and Kill characters.

<Esc>
    Ends insertion, back to Command mode.

Quit key sequence
    Interrupts, terminates insert or <Ctrl-d>.

<Ctrl-d>
    Goes back to the previous autoindent stop.

^<Ctrl-d>

Ends autoindent for this line only.

0<Ctrl-d>
Moves the cursor back to the left margin.

<Ctrl-v>
Quotes a nonprinting character.

Changing Text from Command Mode

Use the following subcommands in Command mode. An incomplete subcommand can be canceled by pressing the <Esc> key.

C Changes the rest of the line (c$).

c Must be followed by a movement command. Deletes the specified region of text and enters input mode to replace it with the entered text. If more than part of a single line is affected, the deleted text is saved in the numeric buffers. If only part of the current line is affected, the last character to be deleted is marked with a $. A count is passed through to the move command. If the command is cc, the whole of the current line is changed.

cc Changes a line.

cw Changes a word.

D Deletes the rest of the line (d$) and puts it into the undo buffer.

d Must be followed by a movement command. Deletes the specified region of text. If more than part of a line is affected, the text is saved in the numeric buffers. A count is passed through to the move command. If the command is dd, the whole of the current line is deleted.

dd Deletes a line and puts it into the undo buffer.

dw Deletes a word and puts it into the undo buffer.

J Joins lines.

rx Replaces the current character with the character specified by x.

R Overwrites characters.

s Substitutes characters (cl).

S Substitutes lines (cc).

u Undoes the previous change.

x Deletes a character.

X Deletes characters before cursor (dh).

<< Shifts one line to the left.

<L Shifts all lines from the cursor to the end of the screen to the left. (The < character describes a range upon which the L subcommand acts.)

>> Shifts one line to the right.

>L Shifts all lines from the cursor to the end of the screen to the right. (The > character describes a range upon which the L subcommand acts.)

~ Changes the letter at the cursor to the opposite case.

Copying and Moving Text

Use the following subcommands in Command mode. An incomplete subcommand can be canceled by pressing <Esc>.

p (P)
Puts back text in the undo buffer after (before) the cursor.

"xp (xP)
Puts back text from the buffer x after (before) the cursor. You must precede the character x with a double quote.

"xdobject
Deletes object into the buffer x. You must precede the character x with a double quote.

yobject
    Yanks object into the undo buffer (for example, yw to yank a word).

"xyobject
    Yanks object into buffer x. You must precede the character x with a double quote.

Y   Places the line in the undo buffer.

Restoring and Repeating Changes

Use the following subcommands in Command mode. An incomplete subcommand can be canceled by pressing the <Esc> key.

u   Undoes the last command.

U   Restores the current line if the cursor has not left the line since the last change.

.   Repeats the last change or increments the np command.

    Note that this command is not meant for use with a macro. Enter @@ to repeat a macro.

"np Retrieves the nth last delete of a complete line or block of lines. You must precede the character n with a double quote.

Saving Changes to a File

Use the following subcommands in Command mode. An incomplete subcommand can be canceled by pressing <Esc>. If you are using these subcommands within the ex editor, you do not need to type the : (colon).

:w Writes the editing buffer contents to the original file.

:w file
    Writes the editing buffer contents to the named file.

:w! file
    Overwrites file with the editing buffer contents.

Interrupting, Cancelling, and Exiting vi

Q Enters the ex editor in Command mode.

ZZ Exits vi, saving changes, if any were made.

:q Quits vi. If you have changed the contents of the editing buffer, vi displays a warning message and does not quit.

:q! Quits vi, discarding the editing buffer with no warning.

:sh Runs a shell. You can return to vi by pressing <Ctrl-d>.

:w !command
: Runs the file through the specified shell command (causes no change to the file).

:!command
: Runs command, then returns.

:!! Repeats the last :!command command.

n!!command
: Executes the shell command identified by command and replaces the number of lines specified by n with the output of command. If n is not specified, the default is 1. If command expects standard input, the lines specified are used as input. (10!!sort sorts the next 10 lines.)

!linescommand
: Works like n!!command, except that lines is a line address (for example, !Gsort sorts the rest of the file).

Quit key sequence
: Interrupts a subcommand.

Editing a Second File

Enter the following subcommands in Command mode. An incomplete subcommand can be canceled by pressing the <Esc> key.

:e file

Edits file. If you are using this subcommand from the ex editor, you do not need to type the : (colon).

:e! Reedits the current file and discards all changes.

:e + file
Edits file, starting at the end.

:e +number
Edits file, starting at the line number.

:e # and <Ctrl-a>
Edits the alternate file. The alternate file is usually the previous current filename. However, if changes are pending on the current file when a new file is called, the new file becomes the alternate file.

:r file
Reads the file into the editing buffer by adding new lines below the current line. If you are using this subcommand from the ex editor, you do not need to type the : (colon).

:r !command
Runs the shell command identified by command and places its output in the file by adding new lines below the current cursor position.

Using a Tags File

The ctags command makes a tags file from the specified C, Pascal, FORTRAN, yacc, lex, or LISP source files. A tags file gives the locations of function and type definitions in a group of files. To use the next three commands, you must first create a database of function names and their locations (called a tags file) using the ctags command.

:ta tag
Edits a file containing tag at the location of tag. If the tag is in another file and the current file has been changed (and noaw is set), a warning is posted. If you are using this subcommand from the ex editor, you do not need to type the : (colon).

<Ctrl-]>
Finds the word at the cursor in the tags file then displays the proper

file, placing the cursor at the tag. If the tag is in the current file, moves cursor to it. Same as :ta, but the tag is the word to the right of the cursor.

<Ctrl-t>
Returns to the file and line where the cursor was positioned when <Ctrl-]> was entered.

Editing a List of Files

Enter the following subcommands in Command mode.

:n Edits the next file in the list entered on the command line.

:n file ...
Specifies a new list of files to edit.

Displaying File Information

Enter the following subcommand in Command mode to show the current filename, the current line number, the number of lines in the file, and the percentage of lines of the file that are before the cursor:

<Ctrl-g>

RELATED INFORMATION

Commands: ctags(1), ed(1)/red(1), edit(1), ex(1), grep(1)/fgrep(1)/egrep(1), nroff(1)

A.3.1 Short Command Reference for vi

Command	Description
Screen Control Commands	
<**CTRL/L**>	Reprints current screen.
<**CTRL/R**>	Reprints current screen and eliminates @@ lines.
z<**RETURN**>	Moves current line to top of screen.
z-	Moves current line to bottom of screen.
z.	Moves current line to center of screen.
/*pattern*/**z-**	Moves line containing *pattern* to bottom of screen
z*n***.**	Sets screen size to *n* lines.
<**CTRL/Y**>	Exposes one more line at top of screen.
<**CTRL/E**>	Exposes one more line at bottom of screen.
n<**CTRL/E**>	Exposes *n* more lines at bottom of screen.
n<**CTRL/Y**>	Exposes *n* more lines at top of screen.
Paging Commands	
<**CTRL/F**>	Pages forward one screen.
<**CTRL/B**>	Pages back one screen.
<**CTRL/D**>	Pages down half screen.
<**CTRL/U**>	Pages up half screen.
Cursor Positioning Commands	
j	Moves cursor down one line, same column
*n***j**	Moves cursor down *n* lines, same column
k	Moves cursor up one line, same column
*n***k**	Moves cursor up *n* lines, same column
h	Moves cursor back one character.
*n***h**	Moves cursor back *n* characters.
l	Moves cursor forward one character.
*l***h**	Moves cursor forward *n* characters.
+	Moves cursor to beginning of next line.
<**RETURN**>	Moves cursor to beginning of next line.
-	Moves cursor to beginning of previous line.
ˆ	Moves cursor back to first nonblank space on current line.
0	Moves cursor to beginning of current line.
$	Moves cursor to end of current line.
<**SPACE**>	Moves cursor forward one character.
n|	Moves cursor to column *n*.
w	Moves cursor forward one word.
*n***w**	Moves cursor forward *n* words.
b	Moves cursor back one word.
*n***b**	Moves cursor back *n* words.
e	Moves cursor to end of current word. When repeated, moves cursor to end of next word.
H	Moves cursor to beginning of first line on screen.
L	Moves cursor to beginning of last line on screen.
	continued on next page

continued from previous page

Command	Description
M	Moves cursor to beginning of middle line on screen.
n**G**	Moves cursor to beginning of line n. Default is last line of file.
fx	Moves cursor forward on current line to next occurrence of x.
Fx	Moves cursor back on current line to previous occurrence of x.
tx	Moves cursor forward on current line to character before x.
Tx	Moves cursor back on current line to character before x.
;	Repeats previous f, F, t, or T command.
,	Reverses direction of f, F, t, or T command.
/*pattern*	Moves cursor forward to next occurrence of *pattern*.
?*pattern*	Moves cursor back to previous occurrence of *pattern*.
n	Repeats last / or ? pattern search.
N	Reverses direction of last / or ? pattern search.
%	Finds matching ( ), { }, or [ ] if cursor on either one of the pair.
“	Moves cursor to previous context. Functional only after altering text or searching a pattern.
”	Moves cursor to beginning of previous context line. Functional only after altering text or searching a pattern.
mx	Marks current position with letter x [a-z].
‘x	Moves cursor to position previously marked x.
’x	Moves cursor to beginning of line containing position previously marked x.
]]	Moves cursor to next section (for text containing formatting macros).
[[	Moves cursor to previous section (for text containing formatting macros).
)	Moves cursor to beginning of next sentence.
}	Moves cursor to beginning of next paragraph.
(	Moves cursor back to previous sentence.
{	Moves cursor back to previous paragraph.
Text Insertion Commands	
a	Appends text after cursor until stopped by pressing the escape key.
A	Appends text at end of current line until stopped by pressing the escape key. Same as $a.
i	Inserts text before cursor until stopped by pressing the escape key.
I	Inserts text at beginning of current line until stopped by pressing the escape key. Same as ˆa.
o	Opens new line below current line for text insertion until stopped by pressing the escape key.

continued on next page

continued from previous page

Command	Description
O	Opens new line above current line for text insertion until stopped by pressing the escape key.
<CTRL/D>	Backs out text one shift width. Auto indent must be set.
<CTRL/0D>	Backs out text to left edge of screen. Auto indent must be set.
<CTRL/H>	Overwrites last character during text insertion.
<DELETE>	Overwrites last character during text insertion.
<CTRL/T>	Indents text one shift width. shift width must be set.
<CTRL/W>	Deletes previous word during text insertion.
<ESC>	Stops text insertion.
<CTRL/V>	Inserts and displays following control character (next argument).
Text Deletion Commands	
dw or **dW**	Deletes current word.
x	Deletes current character.
X	Deletes previous character.
*n***x**	Deletes *n* characters.
dw	Deletes *n* words. Default is current word or remainder of current word.
de	Deletes current word, but leaves punctuation.
dd	Deletes current line.
*n***dd**	Deletes *n* lines.
*n***dj**	Deletes current line plus next *n* lines.
D	Deletes from cursor to end of current line.
d/*patern*	Deletes all text up to *pattern*.
df*x*	Deletes text through the given *x*.
Text Change Commands Change commands work with objects, put you into insert mode, shift text to right or left to fit, and must be ended with the escape key.	
cc	Changes characters on current line until stopped with escape key.
*n***cc**	Changes *n* (number) of lines.
cw or **cW**	changescharacters of current word until stoppedwith escape key.
*n***cw**	Changes characters on *n* next words.
c$	Changes text up to end of line.
ct*x*	Changes text up to the given letter *x*.
C	Changes remaining text on current line until stopped by pressing the escape key.
~	Changes case of current character.
xp	Transposes the current and the following character.
J	Joins current line with next line.
>>	Moves current line one shift width to the right.

continued on next page

continued from previous page

Command	Description
<<	Moves current line one shift width to the left.
>**L**	Moves all lines between cursor and end of screen one shift width to the right.
<**L**	Moves all lines between cursor and end of screen one shift width to the left.
n>> or *n*<<	Moves *n* number of lines one shift width to left or right respectively.
Text Replacement Commands Replacement commands work on a single character or line and can be repeated, put you into insert mode, shift text from right or left to fit, and must be ended with the escape key.	
s	Substitutes text for current character until stopped by pressing the escape key.
S	Substitutes text for current line until stopped by pressing the escape key.
:s	Substitutes new word(s) for old. Written as: **:<addr range> s/old/new/g**, where addr range is a range of line numbers. Example: **:1,$ s/car/truck/g**.
&	Repeats last substitution (**:s**) command.
Undo and Redo Commands These commands apply to all text alteration activities.	
u	Undoes last change made.
U	Restores current line.
.	Repeats last change.
Buffer Usage Commands	
*n***Y**	Yanks *n* lines to unnamed buffer. Default is current line.
yy	Yaks current line to unnamed buffer.
*n***yy**	Yanks *n* lines to unnamed buffer. Default is current line.
"*n***p**	Puts back text from *n*th previous delete (1-9).
yw	Yanks curent word to unnamed buffer.
y*n***w**	Yanks *n* words to unnamed buffer.
p	Puts yanked text line(s) after current line or yanked words after cursor.
P	Puts yanked text line(s) before current line or yanked words before cursor.
"*xn***Y**	Yanks *n* lines to buffer *x* (a-z). Default is current line.
"*xn***L**	Yanks text from cursor to end of screen into buffer *x* (a-z).
"*x***y**/*pattern*	Yanks text from cursor to *pattern* into buffer *x* (a-z).
"*xn***dd**	Deletes *n* lines and saves them in buffer *x* (a-z). Default is current line.
"*x***p**	Puts text from buffer *x* (a-z) before current line.
"*x***P**	Puts text from buffer *x* (a-z) after current line.

continued on next page

continued from previous page

Command	Description
You may also yank, copy, or delete a vi command into a named buffer and then execute the command with **@@**x, where x is the buffer in which you have put your command.	
File Manipulation Commands	
:w*file*	writes changes to *file*. default is current file.
:wq	Writes changes to current file and quits edit session.
:w!*file*	Overwrites file. Default is to overwrite current file.
:q	Quits edit session (no changes made).
:q!	Quits edit session and discards changes.
:e*file*	Edits *file*.
:e!	Discards changes and reedits current file.
:e+*n file*	Edit *file* and place cursor at line n. Default is to place cursor at end of *file*.
:e#	Edits alternate file. Allows shifting of text between files by using named buffers.
:sh	Executes shell and the returns to edit session.
:!*command*	Escapes to executes *command* and then returns to edit session.
:n	Edits next file in argument list.
:f*name*	Changes name of current file to *name*. Default is to print name of file and current line number.
<CTRL/G>	Displays current file name and line number.
:r*file*	Reads contents of *file* into current file at cursor position.
:so*file*	Sources given ex or vi command file.
:set nu	Numbers each text line.
Command Filters A command filter takes input from the buffer, modifiesit, and inserts its output back into the buffer in place of the input text.	
!*x command*	Uses text of cursor positioning command x buffer as standard input to the given *command* and replaces the buffer text with output from the *command*. Example: **!Gsort** will sort all text to the end of the file, then replace those lines with the sorted output.
!!*command*	Performs the *command* on the current line, then replaces the current line with the output from the *command*. Example: **4!!sort** will sort the next four lines and replace those lines with the sorted output.
:r !*command*	Reads in the output of the command after the current line. Example: **:r !date** will add the date after the current line.
:w !*command*	Sends the contents of the buffer to a *command* without affecting them. Examples: **:w !wc** will count the buffer contents; **:w !lpr** will print the buffer contents; **:1,25w !lpr** will print lines 1 through 25.

continued on next page

<table>
<tr><td colspan="2">continued from previous page</td></tr>
<tr><td>Command</td><td>Description</td></tr>
<tr><td colspan="2">Startup Commands
Map commands and named buffers can be set up in the EXINIT variable located in the .bashrc file in your home directory. This creates your editing environment whenever vi is invoked. Unused keys available for vi command mapping are K, V, g,q, v, *, =. It is also possible to redefine any built-in vi command keys, as shown in the following example (E, s and S).
Example .bashrc file EXINIT setting:
export EXINIT=”set ai aw ic sw=4 redraw terse|map g G|map v~~~~—map E :e#<CTRL/M> |map s eas<esc> |map K G:r \!spell %<CTRL/M>”</td></tr>
<tr><td>ab abbr full</td><td>Defines an abbreviation abbr for the word(s) full. When typing abbr it is automatically replaced by full.</td></tr>
<tr><td>ai</td><td>Auto indent</td></tr>
<tr><td>aw</td><td>Auto write</td></tr>
<tr><td>ic</td><td>Ignore case on searches</td></tr>
<tr><td>sw=n</td><td>Shift width equals n spaces (for <, >, <CTRL/D>, <CTRL/T>).</td></tr>
<tr><td>redraw</td><td>Redraw screen after deletes.</td></tr>
<tr><td>terse</td><td>Terse error messages.</td></tr>
<tr><td>wm</td><td>wrap margin (spaces from right edge of screen). Automatic line splitting.</td></tr>
<tr><td>ws</td><td>Wrap scan around end of buffer on searches.</td></tr>
<tr><td>map</td><td>Define your own commands.</td></tr>
<tr><td>:mapx command</td><td>Map operator where xis any letter, command is any vi/ex command or combination of commands. Examples: :map g G causes g command to act like G; :map v ~~~~ causes letter v to change the case of the next four letters.</td></tr>
<tr><td colspan="2">Source Command Files
Editor options and key mappings that only affect the current invocation of vi can be specified in a file that you source after invoking vi.
Examples:
A file named .dialexrc containing commands set noredraw slow would be useful on a dial-in terminal.
A file named .textexrc containing these commands could be useful when edting English text:
set ic wm=10
ab U UNIX
ab UEG UNIX Engineering Group
Invoke this command from within vi as follows:
:so ~/.textexrc</td></tr>
</table>

GNU Emacs Reference Card

(for version 19)

Starting Emacs

To enter GNU Emacs 19, just type its name: `emacs`

To read in a file to edit, see Files, below.

Leaving Emacs

suspend Emacs (or iconify it under X)	`C-z`
exit Emacs permanently	`C-x C-c`

Files

read a file into Emacs	`C-x C-f`
save a file back to disk	`C-x C-s`
save **all** files	`C-x s`
insert contents of another file into this buffer	`C-x i`
replace this file with the file you really want	`C-x C-v`
write buffer to a specified file	`C-x C-w`

Getting Help

The Help system is simple. Type `C-h` and follow the directions. If you are a first-time user, type `C-h t` for a **tutorial**.

remove Help window	`C-x 1`
scroll Help window	`ESC C-v`
apropos: show commands matching a string	`C-h a`
show the function a key runs	`C-h c`
describe a function	`C-h f`
get mode-specific information	`C-h m`

Error Recovery

abort partially typed or executing command	`C-g`
recover a file lost by a system crash	`M-x recover-file`
undo an unwanted change	`C-x u` or `C-_`
restore a buffer to its original contents	`M-x revert-buffer`
redraw garbaged screen	`C-l`

Incremental Search

search forward	`C-s`
search backward	`C-r`
regular expression search	`C-M-s`
reverse regular expression search	`C-M-r`
select previous search string	`M-p`
select next later search string	`M-n`
exit incremental search	`RET`
undo effect of last character	`DEL`
abort current search	`C-g`

Use `C-s` or `C-r` again to repeat the search in either direction. If Emacs is still searching, `C-g` cancels only the part not done.

Motion

entity to move over	**backward**	**forward**
character	`C-b`	`C-f`
word	`M-b`	`M-f`
line	`C-p`	`C-n`
go to line beginning (or end)	`C-a`	`C-e`
sentence	`M-a`	`M-e`
paragraph	`M-{`	`M-}`
page	`C-x [`	`C-x ]`
sexp	`C-M-b`	`C-M-f`
function	`C-M-a`	`C-M-e`
go to buffer beginning (or end)	`M-<`	`M->`

scroll to next screen	`C-v`
scroll to previous screen	`M-v`
scroll left	`C-x <`
scroll right	`C-x >`
scroll current line to center of screen	`C-u C-l`

Killing and Deleting

entity to kill	**backward**	**forward**
character (delete, not kill)	`DEL`	`C-d`
word	`M-DEL`	`M-d`
line (to end of)	`M-0 C-k`	`C-k`
sentence	`C-x DEL`	`M-k`
sexp	`M-- C-M-k`	`C-M-k`

kill **region**	`C-w`
copy region to kill ring	`M-w`
kill through next occurrence of *char*	`M-z` *char*
yank back last thing killed	`C-y`
replace last yank with previous kill	`M-y`

Marking

set mark here	`C-@` or `C-SPC`
exchange point and mark	`C-x C-x`
set mark *arg* **words** away	`M-@`
mark **paragraph**	`M-h`
mark **page**	`C-x C-p`
mark **sexp**	`C-M-@`
mark **function**	`C-M-h`
mark entire **buffer**	`C-x h`

Query Replace

interactively replace a text string	`M-%`
using regular expressions	`M-x query-replace-regexp`

Valid responses in query-replace mode are

replace this one, go on to next	`SPC`
replace this one, don't move	`,`
skip to next without replacing	`DEL`
replace all remaining matches	`!`
back up to the previous match	`^`
exit query-replace	`ESC`
enter recursive edit (`C-M-c` to exit)	`C-r`

Multiple Windows

delete all other windows	`C-x 1`
delete this window	`C-x 0`
split window in two vertically	`C-x 2`
split window in two horizontally	`C-x 3`
scroll other window	`C-M-v`
switch cursor to another window	`C-x o`
shrink window shorter	`M-x shrink-window`
grow window taller	`C-x ^`
shrink window narrower	`C-x {`
grow window wider	`C-x }`
select buffer in other window	`C-x 4 b`
display buffer in other window	`C-x 4 C-o`
find file in other window	`C-x 4 f`
find file read-only in other window	`C-x 4 r`
run Dired in other window	`C-x 4 d`
find tag in other window	`C-x 4 .`

Formatting

indent current **line** (mode-dependent)	`TAB`
indent **region** (mode-dependent)	`C-M-\`
indent **sexp** (mode-dependent)	`C-M-q`
indent region rigidly *arg* columns	`C-x TAB`
insert newline after point	`C-o`
move rest of line vertically down	`C-M-o`
delete blank lines around point	`C-x C-o`
join line with previous (with arg, next)	`M-^`
delete all white space around point	`M-\`
put exactly one space at point	`M-SPC`
fill paragraph	`M-q`
set fill column	`C-x f`
set prefix each line starts with	`C-x .`

Case Change

uppercase word	`M-u`
lowercase word	`M-l`
capitalize word	`M-c`
uppercase region	`C-x C-u`
lowercase region	`C-x C-l`
capitalize region	`M-x capitalize-region`

The Minibuffer

The following keys are defined in the minibuffer.

complete as much as possible	`TAB`
complete up to one word	`SPC`
complete and execute	`RET`
show possible completions	`?`
fetch previous minibuffer input	`M-p`
fetch next later minibuffer input	`M-n`
regexp search backward through history	`M-r`
regexp search forward through history	`M-s`
abort command	`C-g`

Type `C-x ESC ESC` to edit and repeat the last command that used the minibuffer. The following keys are then defined.

previous minibuffer command	`M-p`
next minibuffer command	`M-n`

GNU Emacs Reference Card

Buffers

select another buffer	C-x b
list all buffers	C-x C-b
kill a buffer	C-x k

Transposing

transpose **characters**	C-t
transpose **words**	M-t
transpose **lines**	C-x C-t
transpose **sexps**	C-M-t

Spelling Check

check spelling of current word	M-$
check spelling of all words in region	M-x ispell-region
check spelling of entire buffer	M-x ispell-buffer

Tags

find a tag (a definition)	M-.
find next occurrence of tag	C-u M-.
specify a new tags file	M-x visit-tags-table
regexp search on all files in tags table	M-x tags-search
run query-replace on all the files	M-x tags-query-replace
continue last tags search or query-replace	M-,

Shells

execute a shell command	M-!
run a shell command on the region	M-|
filter region through a shell command	C-u M-|
start a shell in window *shell*	M-x shell

Rectangles

copy rectangle to register	C-x r r
kill rectangle	C-x r k
yank rectangle	C-x r y
open rectangle, shifting text right	C-x r o
blank out rectangle	M-x clear-rectangle
prefix each line with a string	M-x string-rectangle

Abbrevs

add global abbrev	C-x a g
add mode-local abbrev	C-x a l
add global expansion for this abbrev	C-x a i g
add mode-local expansion for this abbrev	C-x a i l
explicitly expand abbrev	C-x a e
expand previous word dynamically	M-/

Regular Expressions

any single character except a newline	. (dot)
zero or more repeats	*
one or more repeats	+
zero or one repeat	?
any character in the set	[...]
any character not in the set	[^ ...]
beginning of line	^
end of line	$
quote a special character *c*	*c*
alternative ("or")	\\\|
grouping	\\(... \\)
*n*th group	*n*
beginning of buffer	\\`
end of buffer	\\'
word break	\\b
not beginning or end of word	\\B
beginning of word	\\<
end of word	\\>
any word-syntax character	\\w
any non-word-syntax character	\\W
character with syntax *c*	\\s*c*
character with syntax not *c*	\\S*c*

Registers

save region in register	C-x r s
insert register contents into buffer	C-x r i
save value of point in register	C-x r SPC
jump to point saved in register	C-x r j

Info

enter the Info documentation reader	C-h i
Moving within a node:	
scroll forward	SPC
scroll reverse	DEL
beginning of node	. (dot)
Moving between nodes:	
next node	n
previous node	p
move **up**	u
select menu item by name	m
select *n*th menu item by number (1–5)	*n*
follow cross reference (return with l)	f
return to last node you saw	l
return to directory node	d
go to any node by name	g
Other:	
run Info **tutorial**	h
list Info commands	?
quit Info	q
search nodes for regexp	s

Keyboard Macros

start defining a keyboard macro	C-x (
end keyboard macro definition	C-x)
execute last-defined keyboard macro	C-x e
append to last keyboard macro	C-u C-x (
name last keyboard macro	M-x name-last-kbd-macro
insert Lisp definition in buffer	M-x insert-kbd-macro

Commands Dealing with Emacs Lisp

eval **sexp** before point	C-x C-e
eval current **defun**	C-M-x
eval **region**	M-x eval-region
eval entire **buffer**	M-x eval-current-buffer
read and eval minibuffer	M-ESC
re-execute last minibuffer command	C-x ESC ESC
read and eval Emacs Lisp file	M-x load-file
load from standard system directory	M-x load-library

Simple Customization

Here are some examples of binding global keys in Emacs Lisp. Note that you cannot say "\M-#"; you must say "\e#".

```
(global-set-key "\C-cg" 'goto-line)
(global-set-key "\C-x\C-k" 'kill-region)
(global-set-key "\e#" 'query-replace-regexp)
```

An example of setting a variable in Emacs Lisp:

```
(setq backup-by-copying-when-linked t)
```

Writing Commands

```
(defun command-name (args)
  "documentation"
  (interactive "template")
  body)
```

An example:

```
(defun this-line-to-top-of-window (line)
  "Reposition line point is on to top of window.
With ARG, put point on line ARG.
Negative counts from bottom."
  (interactive "P")
  (recenter (if (null line)
              0
            (prefix-numeric-value line))))
```

The argument to `interactive` is a string specifying how to get the arguments when the function is called interactively. Type `C-h f interactive` for more information.

For copies of the GNU Emacs manual, write to the Free Software Foundation, Inc., 675 Massachusetts Ave, Cambridge MA 02139.

A.5 grep - Text Search

```
NAME
       grep, egrep, fgrep - print lines matching a pattern

SYNOPOSIS
       grep [ -[[AB] ]num ] [ -[CEFGVBchilnsvwx] ] [ -e ] pattern
       | -ffile ] [ files...  ]

DESCRIPTION
       Grep searches the named input files (or standard input  if
       no files are named, or the file name - is given) for lines
       containing a match to the given pattern.  By default, grep
       prints the matching lines.

       There  are three major variants of grep, controlled by the
       following options.
       -G     Interpret pattern as  a  basic  regular  expression
              (see below).  This is the default.
       -E     Interpret pattern as an extended regular expression
              (see below).
       -F     Interpret pattern as a list of fixed strings, sepa-
              rated by newlines, any of which is to be matched.
       In  addition,  two  variant  programs  egrep and fgrep are
       available.  Egrep  is  similiar  (but  not  identical)  to
       grep -E, and is compatible with the historical Unix egrep.
       Fgrep is the same as grep -F.

       All variants of grep understand the following options:
       -num   Matches will be printed with num lines  of  leading
              and  trailing  context.   However,  grep will never
              print any given line more than once.
       -A num Print num lines of trailing context after  matching
              lines.
       -B num Print  num lines of leading context before matching
              lines.
       -C     Equivalent to -2.
       -V     Print the version number of grep to standard error.
              This  version  number should be included in all bug
              reports (see below).
       -b     Print the byte offset within the input file  before
```

each line of output.

-c   Suppress normal output; instead print a count of matching lines for each input file. With the -v option (see below), count non-matching lines.

-e pattern
       Use pattern as the pattern; useful to protect patterns beginning with -.

-f file
       Obtain the pattern from file.

-h   Suppress the prefixing of filenames on output when multiple files are searched.

-i   Ignore case distinctions in both the pattern and the input files.

-L   Suppress normal output; instead print the name of each input file from which no output would normally have been printed.

-l   Suppress normal output; instead print the name of each input file from which output would normally have been printed.

-n   Prefix each line of output with the line number within its input file.

-q   Quiet; suppress normal output.

-s   Suppress error messages about nonexistent or unreadable files.

-v   Invert the sense of matching, to select non-matching lines.

-w   Select only those lines containing matches that form whole words. The test is that the matching substring must either be at the beginning of the line, or preceded by a non-word constituent character. Similarly, it must be either at the end of the line or followed by a non-word constituent character. Word-constituent characters are letters, digits, and the underscore.

-x   Select only those matches that exactly match the whole line.

REGULAR EXPRESSIONS

A regular expression is a pattern that describes a set of strings. Regular expressions are constructed analagously to arithmetic expressions, by using various operators to combine smaller expressions.

Grep understands two different versions of regular expression syntax: ``basic'' and ``extended.'' In GNU grep, there is no difference in available functionality using either syntax. In other implementations, basic regular expressions are less powerful. The following description applies to extended regular expressions; differences for basic regular expressions are summarized afterwards.

The fundamental building blocks are the regular expressions that match a single character. Most characters, including all letters and digits, are regular expressions that match themselves. Any metacharacter with special meaning may be quoted by preceding it with a backslash.

A list of characters enclosed by [and] matches any single character in that list; if the first character of the list is the caret ^ then it matches any character not in the list. For example, the regular expression [0123456789] matches any single digit. A range of ASCII characters may be specified by giving the first and last characters, separated by a hyphen. Finally, certain named classes of characters are predefined. Their names are self explanatory, and they are [:alnum:], [:alpha:], [:cntrl:], [:digit:], [:graph:], [:lower:], [:print:], [:punct:], [:space:], [:upper:], and [:xdigit:]. For example, [[:alnum:]] means [0-9A-Za-z], except the latter form is dependent upon the ASCII character encoding, whereas the former is portable. (Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket list.) Most metacharacters lose their special meaning inside lists. To include a literal] place it first in the list. Similarly, to include a literal ^ place it anywhere but first. Finally, to include a literal - place it last.

The period . matches any single character. The symbol \w is a synonym for [[:alnum:]] and \W is a synonym for [^[:alnum]].

The caret ^ and the dollar sign $ are metacharacters that respectively match the empty string at the beginning and end of a line. The symbols \< and \> respectively match the empty string at the beginning and end of a word. The symbol \b matches the empty string at the edge of a word, and \B matches the empty string provided it's not at the edge of a word.

A regular expression matching a single character may be followed by one of several repetition operators:

?	The preceding item is optional and matched at most once.
*	The preceding item will be matched zero or more times.
+	The preceding item will be matched one or more times.
{n}	The preceding item is matched exactly n times.
{n,}	The preceding item is matched n or more times.
{,m}	The preceding item is optional and is matched at most m times.
{n,m}	The preceding item is matched at least n times, but not more than m times.

Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated subexpressions.

Two regular expressions may be joined by the infix operator |; the resulting regular expression matches any string matching either subexpression.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole subexpression may be enclosed in parentheses to override these precedence rules.

The backreference \n, where n is a single digit, matches the substring previously matched by the nth parenthesized subexpression of the regular expression.

In basic regular expressions the metacharacters ?, +, {, |, (, and) lose their special meaning; instead use the backslashed versions \?, \+, \{, \|, \(, and \).

In egrep the metacharacter { loses its special meaning; instead use \{.

DIAGNOSTICS

Normally, exit status is 0 if matches were found, and 1 if no matches were found. (The -v option inverts the sense of the exit status.) Exit status is 2 if there were syntax errors in the pattern, inaccessible input files, or other system errors.

BUGS

Email bug reports to bug-gnu-utils@prep.ai.mit.edu. Be sure to include the word ‘‘grep’’ somewhere in the ‘‘Subject:’’ field.

Large repetition counts in the {m,n} construct may cause grep to use lots of memory. In addition, certain other obscure regular expressions require exponential time and space, and may cause grep to run out of memory.

Backreferences are very slow, and may require exponential time.

Appendix B

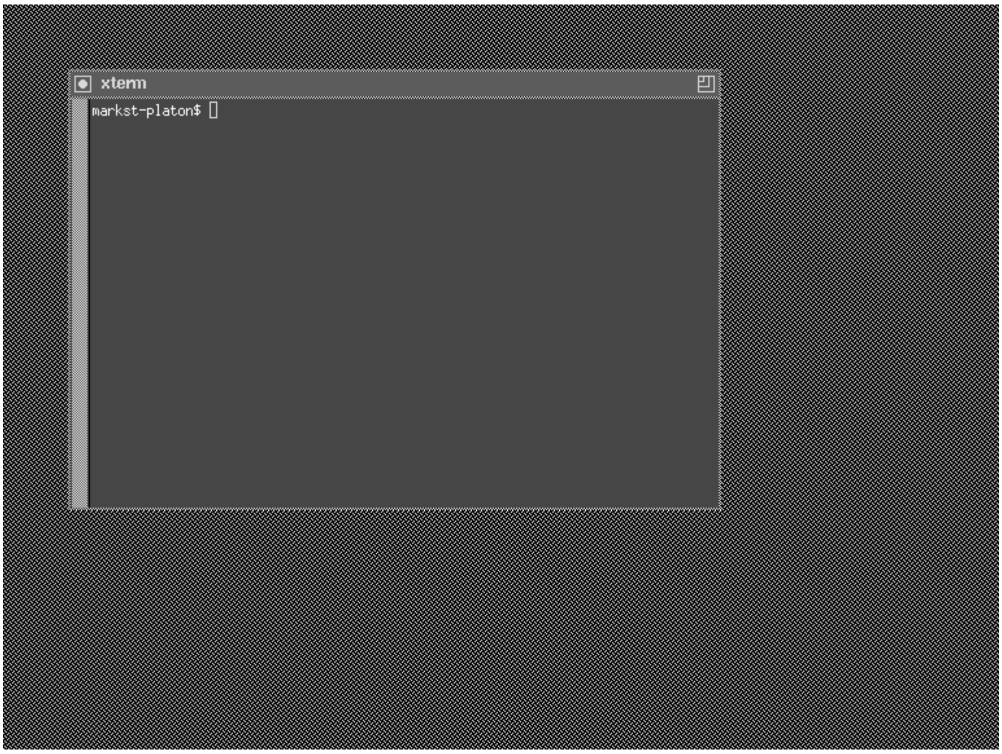
Some FTP and WWW Sites

Company	ftp	WWW
3COM	ftp.3com.com	www.3com.com
Adaptec	ftp.adaptec.com	www.adaptec.com
Adobe	ftp.adobe.com	www.adobe.com
Apple	ftp.austin.apple.com	www.apple.com
Ati	ftp.atitech.ca	www.atitech.ca
Borland	ftp.borland.com	www.borland.com
CA-Associates	mf.cai.com	www.cai.com
Compaq	ftp.compaq.com	www.compaq.com
Creative Labs	ftp.creaf.com	www.creaf.com
Dell	dell1.dell.com	www.dell.com
Diamond	ftp.diamondmm.com	www.diamondmm.com
Hewlett-Packard	ftp.cup.hp.com	www.hp.com
IBM	ftp.austin.ibm.com	www.ibm.com
Id_Software	ftp.idsoftware.com	www.idsoftware.com
Intel	ftp.intel.com	www.intel.com
Lotus	ftp.ccmail.com	www.ccmail.com
Maxtor	ftp.maxtor.com	www.maxtor.com
Micron	ftp.micron.com	www.micron.com
Microsoft	ftp.microsoft.com	www.microsoft.com
NEC	ftp.nec.com	www.nec.com
Novell	ftp.novell.com	www.novell.com
Oracle	ftp.oracle.com	www.oracle.com
Seagate	ftp.seagate.com	www.seagate.com
Supra	ftp.supra.com	www.supra.com
Symantec	ftp.symantec.com	www.symantec.com
Toshiba	ftp.tais.com	www.tais.com
US-Robotics	ftp.usr.com	www.usr.com
Western Digital	ftp.wdc.com	www.wdc.com

Appendix C

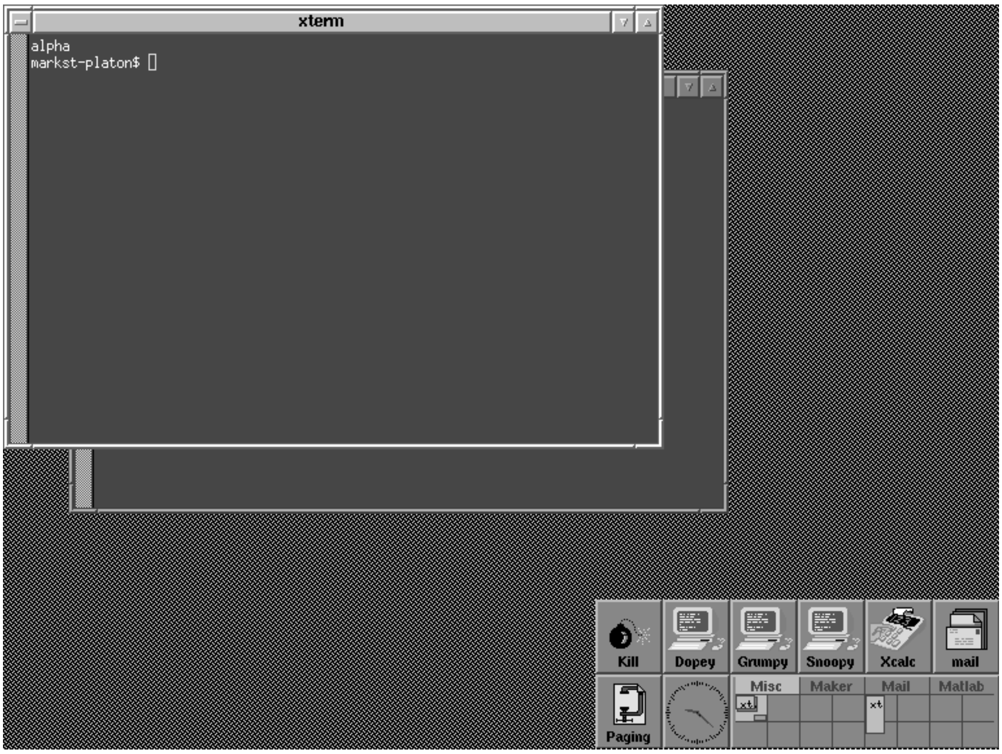
Window Managers

C.1 Tab Window Manager



C.2 Motif Window Manager

C.3 F Virtual Window Manager



C.4 Open Look Virtual Window Manager