**Chapter 1**

**UNIX Operating System**

**1. Overview**

The UNIX operating system was designed to let a number of programmers access the computer at the same time and share its resources. The operating system coordinates the use of the computer's resources, allowing one person, for example, to run a spell check program while another creates a document, lets another edit a document while another creates graphics, and lets another user format a document -- all at the same time, with each user oblivious to the activities of the others.

The operating system controls all of the commands from all of the keyboards and all of the data being generated, and permits each user to believe he or she is the only person working on the computer. This real-time sharing of resources makes UNIX one of the most powerful operating systems ever.

Although UNIX was developed by programmers for programmers, it provides an environment so powerful and flexible that it is found in businesses, sciences, academia, and industry. Many telecommunications switches and transmission systems also are controlled by administration and maintenance systems based on UNIX. While initially designed for medium-sized minicomputers, the operating system was soon moved to larger, more powerful mainframe computers. As personal computers grew in popularity, versions of UNIX found their way into these boxes, and a number of companies produce UNIX-based machines for the scientific and programming communities.

**2. Uniqueness of UNIX**

The features that made UNIX a hit from the start are:

* Multitasking capability
* Multi-user capability
* Portability
* UNIX programs
* Library of application software

**2.1. Multitasking**

Many computers do just one thing at a time, as anyone who uses a PC or laptop can attest. Try logging onto our company's network while opening the browser while opening a word processing program. Chances are the processor will freeze for a few seconds while it sorts out the multiple instructions. UNIX, on the other hand, lets a computer do several things at once, such as printing out one file while the user edits another file. This is a major feature for users, since users don't have to wait for one application to end before starting another one.

**2.2. Multi-user**

The same design that permits multitasking permits multiple users to use the computer. The computer can take the commands of a number of users -- determined by the design of the computer -- to run programs, access files, and print documents at the same time. The computer can't tell the printer to print all the requests at once, but it does prioritize the requests to keep everything orderly. It also lets several users access the same document by compartmentalizing the document so that the changes of one user don't override the changes of another user.

**2.3. System portability**

A major contribution of the UNIX system was its portability, permitting it to move from one brand of computer to another with a minimum of code changes. At a time when different computer lines of the same vendor didn't talk to each other -- yet alone machines of multiple vendors that meant a great savings in both hardware and software upgrades. It also meant that the operating system could be upgraded without having all the customer's data inputted again. And new versions of UNIX were backward compatible with older versions, making it easier for companies to upgrade in an orderly manner.

**2.4. UNIX tools**

UNIX comes with hundreds of programs that can divided into two classes:

* Integral utilities that are absolutely necessary for the operation of the computer, such as the command interpreter, and
* Tools those aren’t necessary for the operation of UNIX but provide the user with additional capabilities, such as typesetting capabilities and e-mail.

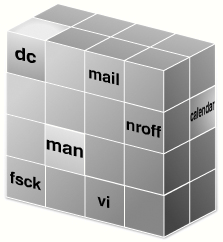


Figure 1.1 - UNIX Modular Structure

Tools can be added or removed from a UNIX system, depending upon the applications required.

**2.4.1. UNIX Communications**

E-mail is commonplace today, but it has only come into its own in the business community within the last 10 years. Not so with UNIX users, who have been enjoying e-mail for several decades. UNIX e-mail at first permitted users on the same computer to communicate with each other via their terminals. Then users on different machines, even made by different vendors, were connected to support e-mail. And finally, UNIX systems around the world were linked into a world wide web decades before the development of today's World Wide Web.

**2.5. Applications libraries**

UNIX as it is known today didn't just develop overnight. Nor were just a few people responsible for it's growth. As soon as it moved from Bell Labs into the universities, every computer programmer worth his or her own salt started developing programs for UNIX. Today there are hundreds of UNIX applications that can be purchased from third-party vendors, in addition to the applications that come with UNIX.

**3. How UNIX is organized**

The UNIX system is functionally organized at three levels:

* The kernel, which schedules tasks and manages storage;
* The shell, which connects and interprets users' commands, calls programs from memory, and executes them; and
* The tools and applications that offer additional functionality to the operating system

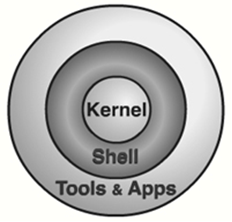


Figure 1.2 - Parts of the UNIX System

The three levels of the UNIX system:

* Kernel
* Shell
* Tools & Applications

**3.1. The kernel**

The heart of the operating system, the kernel controls the hardware and turns part of the system on and off at the programmer’s command. If we ask the computer to list (ls) all the files in a directory, the kernel tells the computer to read all the files in that directory from the disk and display them on our screen.

**3.2. The shell**

There are several types of shell, most notably the command driven Bourne Shell ($sh) and the C Shell($csh) (no pun intended), and menu-driven shells that make it easier for beginners to use. Whatever shell is used, its purpose remains the same -- to act as an interpreter between the user and the computer. The shell also provides the functionality of "pipes," whereby a number of commands can be linked together by a user, permitting the output of one program to become the input to another program.

**C Shell**

The C shell is a command processor that's typically run in a text window, allowing the user to type commands which cause actions. The C shell can also read commands from a file, called a script. Like all Unix shells, it supports filename wildcarding, piping, here documents, command substitution, variables and control structures for condition-testing and iteration. What differentiated the C shell, especially in the 1980s, were its interactive features and overall style. Its new features made it easier and faster to use. The overall style of the language looked more like C and was seen as more readable.

**Korn Shell**

The Korn shell (ksh) is a Unix shell which was developed by David Korn (AT&T Bell Laboratories) in the early 1980s and announced at Toronto USENIX on July 14 1983. ksh is backwards-compatible with the Bourne shell and includes many features of the C shell as well, such as a command history, which was inspired by the requests of Bell Labs users.

The main advantage of ksh over the traditional Unix shell is in its use as a programming language. Since its conception, several features were gradually added, while maintaining strong backwards compatibility with the Bourne shell.

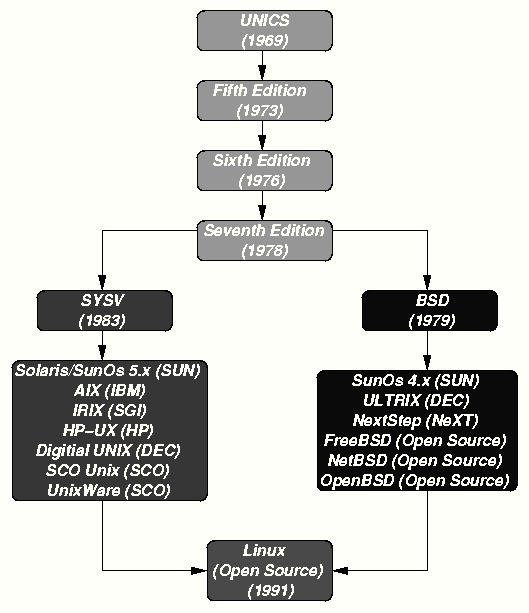
**Bash Shell**

Bash is a Unix shell. it stands for Bourne-again shell, Bash is a POSIX shell with a number of extensions. It is the shell for the GNU operating system from the GNU Project. It can be run on most Unix-like operating systems. It is the default shell on most systems built on top of the Linux kernel as well as on Mac OS X and Darwin.

**3.3. Tools and Applications**

There are hundreds of tools available to UNIX users, although some have been written by third party vendors for specific applications. Typically, tools are grouped into categories for certain functions, such as word processing, business applications, or programming.

**4. Type of Unix OS’s**



**Chapter 2**

**File Systems and Logging in**

**1. File System**

The Unix file system includes directories containing files and directories, each directory of which can contain yet more files and directories. Our own home directory, however, probably doesn’t contain any directories (except . and .. ofcourse), which prevents us from exploiting what we call the virtual file cabinet of the file system.

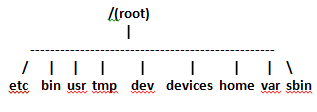
Managing filesystems is one of the important tasks of Systems Administrator. Unix filesystems when originally designed to favor its users. Unix filesystems look like an inverted tree, so that a root is at top and then branches underneath it. Major tasks for Systems administrators are:

**Tasks**

* Making files available to users.
* Managing and monitoring the system's disk resources.
* Protecting against file corruption, hardware failures, user errors through backup.
* Security of these filesystems, what users need and have access to which files.
* Adding more disks, tape drives, etc when needed.

When Unix operating systems is intalled, some directories depending upon the Unix being installed are creaed under / (or root) , such as /usr /bin /etc /tmp /home /var. As we can attach many disks to Unix operating systems, it is recommended that when working in professional environment / and /usr /home are on separate disks.

Same time the files are organized into a tree structure with a root named by the character '/'. The first few levels of the tree look like this:



We will have a look on each and every file or directory in this hierarchy:

|  |  |
| --- | --- |
| etc | Contains all system configuration files and the files which maintain information about users and groups. |
| bin | Contains all binary executable files (command that can be used by normal user also) |
| usr | Default directory provided by Unix OS to create users home directories and contains manual pages |
| tmp | System or users create temporary files which will be removed when the server reboots. |
| dev | Contains all device files i.e. logical file names to physical devices. |
| devices | Contains all device files i.e. physical names of physical devices |
| home | Default directory allocated for the home directories of normal users when the administrator don’t specify any other directory. |
| var | Contains all system log files and message files. |
| sbin | Contains all system administrator executable files (command which generally normal users don’t have |

**2. How do we login to Unix system?**

We need to have an account created by unix administrator for that particular system. For example if We want to login to unix systems named **will-1** , open up a new session to **will-1** and at prompt enter the user name and password as supplied by Administrator.

**How to connect**

When We connect to a UNIX computer and when We log in locally using a text-only terminal, We will see the prompt:

**Login:**

At this prompt, type in the username and press the enter/return/ key. Remember that UNIX is case sensitive (i.e. Will, WILL and will are all different logins). We should then be prompted for the password:

**Login : will-1**

**password :**

Type the password in at the prompt and press the enter/return/ key. Note that the password will not be displayed on the screen as We type it in.

If We mistype the username or password we will get an appropriate message from the computer and We will be presented with the **login:** prompt again. Otherwise We should be presented with a shell prompt which looks something like this:

**$**

To log out of a text-based UNIX shell, type "exit" at the shell prompt (or if that doesn't work try "logout "; if that doesn't work press **ctrl-d**).

**3. Changing the password**

The UNIX command to create/change the password is ***Passwd***:

**$ passwd**

The system will prompt us for our old password, then for our new password. To eliminate any possible typing errors we have made in our new password, it will ask We to reconfirm our new password.

Remember the following points when choosing our password:

* Avoid characters which might not appear on all keyboards, e.g. '£'.
* Make it at least 7 or 8 characters long and try to use a mix of letters, numbers and punctuation.

Note: One of the things we should do when we log in for the first time is to change our password.

**4. File Types**

There are four types of files in the Unix file system.

**4.1. Ordinary Files**

An ordinary file may contain text, a program, or other data. It can be either an ASCII file, with each of its bytes being in the numerical range 0 to 127, i.e. in the 7-bit range, or a binary file, whose bytes can be of all possible values 0 to 255, in the 8-bit range.

**4.2. Directory Files**

Suppose that in the directory x we have a, b and c, and that b is a directory, containing files u and v. Then b can be viewed not only as a directory, containing further files, but also as a file itself. The file b consists of information about the directory b; i.e. the file b has information stating that the directory b has files u and v, how large they are, when they were last modified, etc.

**4.3. Device Files**

In Unix, physical devices (printers, terminals etc.) are represented as ``files.'' This seems odd at first, but it really makes sense: This way, the same read() and write() functions used to read and write real files can also be used to read from and write to these devices.

**4.4. Link Files**

Suppose we have a file X, and type

**$ ln X Y**

If we then run **ls** , it will appear that a new file, Y, has been created, as a copy of X, as if we had typed

**$ cp X Y**

However, the difference is the **cp** does create a new file, while **ln** merely gives an alternate name to an old file. If we make Y using **ln**, then Y is merely a new name for the same physical file X.

**5. Creating Directories and Files**

**5.1. Create a File**

**$ cat > file Enter text and end with ctrl-D**

**$ vi file Edit file using the vi editor**

**5.2. Display File Contents**

**$ cat file display contents of file**

**5.3. Make a Directory**

**$ mkdir directory-name**

**5.4. Making a sub Directory**

To create a directory (dir1) which is having child sub directory(dir2) also in a single command.

**$ mkdir –p dir1/dir2**

**5.5 Denoting Paths**

We can mention the SOURCE path of DESTINATION path in any format as specified below

|  |  |
| --- | --- |
| **cd /** | go to the root directory |
| **cd** | go to your login (home) directory |
| **cd ~username** | go to username's login (home) directory not allowed in the Bourne shell |
| **cd ~username/directory** | go to username's indicated directory |
| **cd ..** | go up one directory level from here |
| **cd ../..** | go up two directory levels from here |
| **cd /full/path/name/from/root** | change directory to absolute path named note the leading slash |
| **cd path/from/current/directory** | change directory to path relative to here. note there is no leading slash |

**Chapter 3**

**Additional Commands in UNIX**

**1. uname**

By default, the uname utility will write the operating system characteristics to standard output. When options are specified, symbols representing one or more system characteristics shall be written to the standard output. The format and contents of the symbols are implementation-defined.

**Syntax**

**$ uname [-a] [-i] [-n] [-p] [-r] [-v]**

|  |  |
| --- | --- |
| -a | Print basic information currently available from the system. |
| -i | Print the name of the hardware implementation (platform). |
| -n | Print the nodename (the nodename is the name by which the system is known to a communications network). |
| -r | Print the operating system release level. |
| -v | Print the operating system version. |
| -p | Print the current host's ISA or processor type. |

**Example**

**$ uname –n**

**<hostname>**

**2. mv**

mv (short for move) is a Unix command that moves a file from one place to another. The original file is deleted, and the new file may have the same or a different name.

**Syntax**

**$ mv [-f] [-i] oldname newname**

|  |  |
| --- | --- |
| -f | mv will move the file(s) without prompting even if it is writing over an existing target. Note that this is the default if the standard input is not a terminal. |
| -i | Prompts before overwriting another file. |
| oldname | The oldname of the file renaming. |
| newname | The newname of the file renaming. |
| filename | The name of the file we want to move directory - The directory of were we want the file to go. |

**Examples**

**$ mv myfile.txt newdirectory/**

moves the file myfile.txt to the directory newdirectory.

**$ mv myfile.txt ../**

moves the file myfile.txt back one directory (if available).

**3. date**

The date command can be used to display or set the date. If a user has superuser privileges, he or she can set the date by supplying a numeric string with the following command:

Fortunately there are options to manipulate the format. The format option is preceded by a + followed by any number of field descriptors indicated by a % followed by a character to indicate which field is desired. The allowed field descriptors are:

|  |  |
| --- | --- |
| Option | Meaning |
| %n | A newline |
| %t | A tab |
| %m | Month of year (01-12) |
| %d | Day of month (01-31) |
| %y | Last two digits of year (00-99) |
| %D | Date as mm/dd/yy |
| %H | Hour (00-23) |
| %M | Minute (00-59) |
| %S | Second (00-59) |
| %T | Time as HH:MM:SS |
| %j | Day of Year (001-366) |
| %w | Day of week (0-6) Sunday is 0 |
| %a | Abbreviated weekday (Sun-Sat) |
| %h | Abbreviated month (Jan-Dec) |
| %r | 12-hour time w/ AM/PM |

**Examples**

**$ date**

**Mon Jan 6 16:07:23 PST 1997**

**$ date '+%a %h %d %T %y'**

**Mon Jan 06 16:07:23 97**

**$ date '+%a %h %d %n %T %y'**

**Mon Jan 06**

**16:07:23 97**

**3.1. Set Date and Time**

**date [-s datestr]**

|  |  |
| --- | --- |
| **-s datestr** | Sets the time and date to the value specified in the datestr. The datestr may contain the month names, timezones, 'am', 'pm', etc. See examples for an example of how the date and time can be set. |

**Examples**

$ date -s "11/20/2003 12:48:00"

Set the date to the date and time shown.

**4. Removing Files and Directories**

**4.1. rm**

Deletes a file without confirmation (by default).

The options must start with ‘-‘. One or more filenames can be specified, and wildcards are permitted (because the shell, not rm, expands them).

If we are not familiar with wildcards (\* and ? to name the most dangerous), read up on them. Placing a wildcard character in the file name list by accident can make we a very unhappy camper

**Syntax**

**rm [-f] [-i] [-R] [-r] [filenames | directory]**

|  |  |
| --- | --- |
| -f | Remove all files (whether write-protected or not) in a directory without prompting the user. In a write-protected directory, however, files are never removed (whatever their permissions are), but no messages are displayed. If the removal of a write-protected directory is attempted, this option will notsuppress an error message. |
| -i | Interactive. With this option, rm prompts for confirmation before removing any files. It over- rides the –f option and remains in effect even if the standard input is not a terminal. |
| filenames | A path of a filename to be removed. |

**Examples**

To remove the file myfile.txt without prompting the user.

**$ rm myfile.txt**

A number of files can all be removed at the same time, Here we remove all the files with the lis extension.

**$ rm \*.lis**

**ultimateanswer.lis : ?y**

**lessultimateanswer.lis : ?n**

**moreultimateanswer.lis : …**

Remove can also be used without asking for a confirmation. To do this uses the -force option. Here we remove all the files with the lis extension, without asking for a confirmation.

**$ rm -f \*.lis**

Beware when using the -f option we can easily remove all files from a directory by accident! To remove all files without asking for a confirmation.

**$ rm -f \***

**4.2. rmdir**

Used to remove the directories form the system

**Syntax**

**rmdir [-p] [-s][-r] directoryname**

|  |  |
| --- | --- |
| -p | Allow users to remove the directory dirname and its parent directories which become empty. A message is printed to standard error if all or part of the path could not be removed. |
| -s | Suppress the message printed on the standard error when -p is in effect. |
| -r | Delete nonempty directory |
| directory | The name of the directory that we wish to delete. |

**Examples**

removes the directory mydir

**$ rmdir mydir**

To deletes all directories in the current directory whose directory names begins with the characters "index".

**$ rmdir index\***

To remove a directory, even if files existed in that directory.

**$ rm -r directory**

To delete the directory named "new-novel". This directory, and all of it’s contents, are erased from the disk, including any sub-directories and files.

**$ rm -r new-novel**

**5. ls**

The ls command lists the files in your current working directory. When we log onto your account on UNIX, your current working directory is your home or personal directory. This is the directory in which we have personal disk space to put files on or to create sub-directories under. The ls command also has options available. Options follow the hyphen ( - ) sign. Two of the most useful options are a (return all files, even "hidden") and we (give long or full file information). The ls command also accepts strings with the asterisk \* used as a "wildcard" to tell UNIX to search for all files that contain the specified sub-string.

**Syntax**

**ls [-a] [-A] [-b] [-c] [-C] [-d] [-f] [-F] [-g] [-i] [-l] [-L] [-m] [-o] [-p] [-q] [-r] [-R] [-s] [-t] [-u] [-x] [pathnames]**

|  |  |
| --- | --- |
| -a | Shows us all files, even files that are hidden (these files begin with a dot.) |
| -A | List all files including the hidden files. However, does not display the working directory (.) or the parent  directory (..). |
| -b | Force printing of non-printable characters to be in octal \ ddd notation. |
| -c | Use time of last modification of the i-node (file created, mode changed, and so forth) for sorting (-t) or printing (-l or -n). |
| -C | Multi-column output with entries sorted down the columns. Generally this is the default option. |
| -d | If an argument is a directory it only lists its name not its contents. |
| -f | Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off -l, -t, -s, and -r, and turns on -a; the order is the order in which entries appear in the directory. |
| -F | Mark directories with a trailing slash (/), doors with a trailing greater-than sign (>), executable files with a trailing asterisk (\*), FIFOs with a trailing vertical bar (|), symbolic links with a trailing at-sign (@), and AF\_UNIX address family sockets with a trailing equals sign (=). |
| -g | Same as -l except the owner is not printed. |
| -i | For each file, print the i-node number in the first column of the report. |
| -l | Shows us huge amounts of information (permissions, owners, size, and when last modified.) |
| -L | If an argument is a symbolic link, list the file or directory the link references rather than the link itself. |
| -m | Stream output format; files are listed across the page, separated by commas. |
| -n | The same as -l, except that the owner's UID and group's GID numbers are printed, rather than the associated character strings. |
| -o | The same as -l, except that the group is not printed. |
| -p | Displays a slash ( / ) in front of all directories. |
| -q | Force printing of non-printable characters in file names as the character question mark (?). |
| -r | Reverses the order of how the files are displayed. |
| -R | Includes the contents of subdirectories. |
| -s | Give size in blocks, including indirect blocks, for each entry. |
| -t | Shows us the files in modification time. |
| -u | Use time of last access instead of last modification for sorting (with the -t option) or printing (with the –l option). |
| -x | Displays files in columns. |
| -1 | Print one entry per line of output. |
| pathnames | File or directory to list. |

**Examples**

**$ ls -al \*test\***

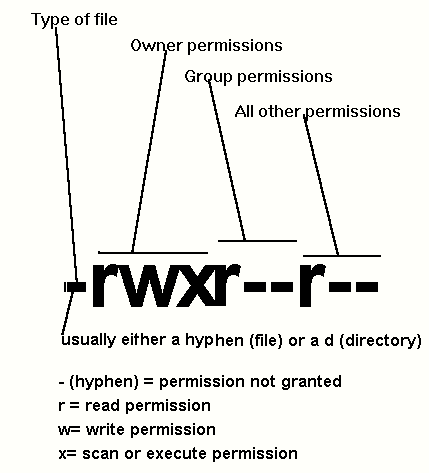
**-rw-r--r-- 1 hcsdar usg 592 Sep 1 1993 .test**

**drwx------ 2 hcsdar usg 512 Nov 11 16:21 dirtest**

**-rw-r--r-- 2 hcsdar usg 1097 Nov 2 1993 test**

**-rw------- 1 hcsdar usg 1097 Oct 19 15:54 test.bin**

**-rw------- 1 hcsdar usg 1216 Jul 15 1993 test.fil**

**What Does Coloumn1 tells us?**

**Column 1-** tells us the type of file, what privileges it has and to whom these privileges are granted. There are three types of privileges. Read and write privileges are easy to understand. The exec privilege is a little more difficult. We can make a file "executable" by giving it exec privileges. This means that commands in the file will be executed when we type the file name in at the UNIX prompt. It also means that when a directory which, to UNIX is a file like any other file, can be "scanned" to see what files and sub-directories are in it. Privileges are granted to three levels of users:

1. The owner of the file. The owner is usually, but not always, the userid that created the file.
2. The group to which the owner belongs. At GSU, the group is usually, but not always designated as the first three letters of the userid of the owner.
3. Everybody else who has an account on the UNIX machine where the file resides.

**Column 2 -**Number of links

**Column 3 -** Owner of the file. Normally the owner of the file is the user account that originally created it.

**Column 4 -** Group under which the file belongs. This is by default the group to which the account belongs or first three letters of the userid. The group can be changed by the chgrp command.

**Column 5 -** Size of file (bytes).

**Column 6 -** Date of last update

**Column 7 -** Name of file

**Examples**

Rather than list the files contained in the /usr directory, this command lists information about the /usr directory itself (without generating a listing of the contents of /usr). This is very useful when we want to check the permissions of the

**$ ls -ld /usr**

List the contents of your home directory by adding a tilde after the ls command.

**$ ls ~**

List the contents of your root directory.

**$ ls /ls ../**

List the contents of the parent directory.

**$ ls \*/**

List the contents of all sub directories.

Only list the directories in the current directory.

**$ ls -d \*/**

**6. finger**

In Unix, finger is a program we can use to find information about computer users. It usually lists the login name, the full name, and possibly other details about the user we are fingering. These details may include the office location and phone number (if known), login time, idle time, time mail was last read, and the user's plan and project files. The information listed varies, and we may not be able to get any information from some sites.

**Syntax**

**finger [-b] [-f] [-h] [-i] [-l] [-m] [-p] [-q] [-s] [-w] [username]**

|  |  |
| --- | --- |
| -b | Suppress printing the user's home directory and shell in a long format printout. |
| -f | Suppress printing the header that is normally printed in a non-long format printout. |
| -h | Suppress printing of the .project file in a long format printout. |
| -i | Force "idle" output format, which is similar to short format except that only the login name, terminal, login time, and idle time are printed. |
| -l | Force long output format. |
| -m | Match arguments only on user name (not first or last name). |
| -p | Suppress printing of the .plan file in a long format printout. |
| -q | Force quick output format, which is similar to short format except that only the login name, terminal, and login time are printed. |
| -s | Force short output format. |
| -w | Suppress printing the full name in a short format printout. |

**Examples**

We can find out someone's username and whether they are logged in with the finger command.

**finger name\_of\_person[@remote\_host]**

"name" can be a first or last name, or a username.

**$ finger wil-1**

**$ Login name: wil-1 In real life: Faculty1**

**how to change Finger**

Most Unix systems have a chfn (change finger) command. It allows us to change the standard information that is displayed when someone fingers your account.

To change your finger information, on most systems, at the Unix shell prompt, enter chfn. We will be prompted to enter values for each of the following fields:

Changing finger information for username

**$ chfin**

**Name [your name]:**

**Location [XY 0436]:**

**Office Phone [555-1212]:**

**Home Phone [555-7352]:**

**7. who**

The who command displays a list of users currently logged in to the local system. It displays each users login name, the login device (TTY port), the login date and time. The command reads the binary file /var/admn/utmpx to obtain this information and information about where the users logged in from

If a user logged in remotely the who command displays the remote host name or internet Protocol (IP) address in the last column of the output.

**Syntax**

$ who [-a] [-b] [-d] [-H] [-l] [-m] [-nx] [-p] [-q] [-r] [-s] [-t] [-T] [-u] [am i] [ file ]

|  |  |
| --- | --- |
| -a | Process /var/adm/utmp or the named file with -b, -d, -l, -p, -r, -t, -T, and -u options turned on. |
| -b | Indicate the time and date of the last reboot. |
| -d | Display all processes that have expired and not been respawned by init . The exit field appears for dead processes and contains the termination and exit values (as returned by wait), of the dead process. This canbe useful in determining why a process terminated. |
| -H | Output column headings above the regular output. |
| -l | List only those lines on which the system is waiting for someone to login. The name field is LOGIN in such cases. Other fields are the same as for user entries except that the state field does not exist. |
| -m | Output only information about the current terminal. |
| -n x | Take a numeric argument, x, which specifies the number of users to display per line. x must be at least 1. The -n option may only be used with -q. |
| -p | List any other process which is currently active and has been previously spawned by init . The name field is the name of the program executed by init as found in /sbin/inittab. The state, line , and idle fields have no meaning. The comment field shows the id field of the line from /sbin/inittab that spawned this process. |
| -q | (quick who ) display only the names and the number of users currently logged on. When this option is used, all other options are ignored. |
| -r | Indicate the current run-level of the init process. |
| -s | (default) List only the name, line, and time fields. |
| -t | Indicate the last change to the system clock (using the date utility) by root. See su and date. |
| -T | Same as the -s option, except that the state field is also written. state is one of the characters listed under the /usr/bin/who version of this option. If the -u option is used with -T, the idle time is added to the end of the previous format. |
| -u | List only those users who are currently logged in. The name is the user's login name. The line is the name of the line as found in the directory /dev. The time is the time that the user logged in. The idle column contains the number of hours and minutes since activity last occurred on that particular line. A dot (.) indicates that the terminal has seen activity in the last minute and is therefore ``current''. If more than twenty-four hours have elapsed or the line has not been used since boot time, the entry is marked old. This field is useful when trying to determine whether a person is working at the terminal or not. The pid is the process-ID of the user's shell. The comment is the comment field associated with this line as found in /sbin/inittab. This can contain information about where the terminal is located, the telephone number of the dataset, type of terminal if hard- wired, and so forth. |
| am i | In the "C" locale, limit the output to describing the invoking user, equivalent to the -m option. The am and i or I must be separate arguments. |
| file | Specify a path name of a file to substitute for the database of logged-on users that who uses by default. |

**Examples**

**$ who**

The general format for output is:

**name [state] line time [idle] [pid] [comment] [exit]**

**where:**

name user's login name.

state capability of writing to the terminal.

line name of the line found in /dev.

time time since user's login.

idle time elapsed since the user's last activity.

pid user's process id.

comment comment line in inittab(4).

**8. cal**

Print a 12-month calendar (beginning with January) for the given year, or a one-month calendar of the given month and year. month ranges from 1 to 12. year ranges from 1 to 9999. With no arguments, print a calendar for the current month.

Before we can do the calendar program we must have a file named calendar at the root of your profile. Within that file we may have something similar to:

**Syntax**

**$ cal [options] [[month] year]**

|  |  |
| --- | --- |
| -j | Display Julian dates (days numbered 1 to 365, starting from January 1). |
| -m | Display Monday as the first day of the week. |
| -y | Display entire year. |
| -V | Display the source of the calendar file. |
| month | Specifies the month for us want the calendar to be displayed. Must be the numeric representation of the month. For example: January is 1 and December is 12. |
| year | Specifies the year that we want to be displayed. |

**EXAMPLES**

$ cal

$ cal -j

$ cal –m

$ cal –y

$ cal –y 1980

$ cal 12 2006

$ cal 2006 > year\_file

**Chapter 4**

**Wildcards in UNIX**

**1. How to use UNIX Wildcards**

Many computer operating systems provide ways to select certain files without typing complete filenames. For example, we may wish to remove all files whose names end with "old". Unix allows us to use wildcards (more formally known as metacharacters ) to stand for one or more characters in a filename.

The two basic wildcard characters are ? and \*. The wildcard ? matches any one character. The wildcard \* matches any grouping of zero or more characters. Some examples may help to clarify this. (Remember that Unix is case-sensitive). Assume that your directory contains the following files:

**Chap bite bin**

**bit Chap6 it**

**test.new abc**

**Lit site test.old**

**Big snit bin.old**

**1.1 The ? wildcard**

The command ls will list all the files. The command

**$ ls ?bit**

**Lit bit**

lists only the files Lit and bit. The file snit was not listed because it has two characters before "it". The file it was not listed because it has no characters before "it".

The ? wildcard may be used more than once in a command. For example,

**$ ls ?i?**

**Lit big bin bit**

Finds any files with "i" in the middle, one character before and one character after.

**1.2 The \* wildcard**

The \* wildcard is more general. It matches zero or any number of characters, except that it will not match a period that is the first character of a name.

**$ ls \*t**

**Lit bit it snit**

Using this wildcard finds all the files with "it" as the last two characters of the name (although it would not have found a file called .bit).

We could use this wildcard to remove all files in the directory whose names begin with "test". The command to do this is

**$rm test\***

Be careful when using the \* wildcard, especially with the rm command. If we had mistyped this command by adding a space between test and \*, Unix would look first for a file called test, remove it if found, and then proceed to remove all the files in the directory!

**1.3 Matching a range of characters with [ ]**

The ? wildcard matches any one character. To restrict the matching to a particular character or range of characters, use square brackets [ ] to include a list. For example, to list files ending in "ite", and beginning with only "a", "b", "c", or "d" we would use the command:

**$ ls [abcd]ite**

This would list the file bite, but not the file site. Note that the sequence [ ] matches only one character. If we had a file called delite, the above command would not have matched it.

We can also specify a range of characters using [ ]. For instance, [1-3] will match the digits 1, 2 and 3, while[A-Z] matches all capital letters.

**ls [A-Z]it**

Will find any file ending in "it" and beginning with a capital letter (in this case, the file Lit).

Wildcards can also be combined with [ ] sequences. To list any file beginning with a capital letter, we would use:

**$ ls [A-Z]\***

**Chap1 Chap6 Lit**

**1.4 Matching a string of characters with { }**

The method described in the previous section matches a single character or range of characters. It is also possible to match a particular string by enclosing the string in { } (braces). For example, to list only the files ending in the string "old", we would use

**$ ls \*{old}**

**bin.old test.old**

To list all files ending in either "old" or "new", use

**$ ls \*{old,new}**

**bin.old test.new test.old**

**2. I/O Redirection**

**2.1 Standard File Descriptors**

The Unix environment allows for each process to have access to three standard file descriptors by default. They are

* 0 standard input
* 1 standard output
* 2 standard error

It is the responsibility of the shell when executing a command to provide appropriate file descriptors to the process for each of these standard files. Most Unix tools are developed to take their input from the standard input file and write their output to the standard output file. Error messages that do not make up part of the expected output are usually written to the standard error file.

Unless otherwise specified, the shell will usually pass it's own standard file descriptors down to the process that it executes, allowing the output from any called tools to be included with the output of the script.

Through using I/O redirection, the developer can modify how the shell handles the file descriptors and usually either replace one of the standard interactive file descriptors with a file on disk, or create a pipe to connect the output file descriptor of one process to the input file descriptor of another process.

Redirection can also be used to perform redirection on file descriptors for a group of commands.

**2.2 Basic File Redirection**

* Disk file redirection is done using the < and > characters.
* > redirects the standard output of the command to write to a file
* >> redirects the standard output of the command to append to a file
* < redirects the standard input of the command to read from a file

**Example:**

**$ ls -al>dirlist.txt**

**$ ls -al>> longlist.txt**

**$ cat a>f1**

For the redirections, the standard file description redirection can be modified by placing the file descriptor identifier in front of the redirection symbol.

For example 2> redirects standard error instead of the default standard input. 1> redirects standard output, which is the default setting.

**Example:**

**$ command > output.log 2>error.log**

**2.3 Advanced File Redirection**

>& is used to redirect one file descriptor to another.

**Example:**

**$ command > common.log 2>&1**

This redirects standard output to common.log, and then redirects standard error to the same place as standard output. The order of these redirections is important, if the 2>&1 is placed before the >common.log, then standard error will be redirected to the standard output (console) then standard output will be redirected to common.log.

<< redirects the standard input of the command to read from what is called a "here document". Here documents are convenient ways of placing several lines of text within the script itself, and using them as input to the command. The

<< Characters are followed by a single word that is used to indicate the end of file word for the Here Document. Any word can be used, however there is a common convention of using EOF (unless we need to include that word within your here document).

**Example:**

**$ sort << EOF**

**bravo**

**delta**

**alpha**

**chrlie**

**EOF**

This would be the same as having a text file with the lines "bravo", "delta", "alpha" and "charlie" in it and redirecting it using sort <input.txt but is simpler and cleaner (no problems with accidentally forgetting to include the input.txt file when we distribute your script)

**Chapter 5**

**Path Names**

**1. Relative a nd Absolute Pathnames**

**1.1 Relative Pathnames**

The use of the ".." notation allows us to navigate the directory tree structure. The ".." symbol means "parent directory." Names with ".." in them are relative names because their meaning depends on where they are issued (the present working directory). we can string together several ".." symbols, separated by the / symbol and other directory names, to change directories. For example, if we are in portfolio and want to change to mary, we can do this with a cd command followed by the relative pathname between portfolio and mary like this (first using pwd to show where we are):

**$ pwd**

**/users/john/portfolio**

**$cd ../../mary**

**$pwd**

**/users/mary**

**$**

Directory or file references starting with .. are relative pathnames.

Directory or file references starting with a directory name are also relative pathnames. For example, if we are in the users directory, the directory reference john/portfolio is a relative pathname:

**$ pwd**

**/users**

**$cd john/portfolio**

**$pwd**

**/users/john/portfolio**

**$**

**1.2 Absolute Pathnames**

If we string together the unique name of all the intervening subdirectories in the file system to a particular subdirectory, we have created the absolute pathname for that directory. The absolute pathname allows us to switch to a directory no matter what my present working directory is. Absolute pathnames always start with a "/". we can navigate the file system by using absolute pathnames. So we could do something like this:

**$ pwd**

**/users/john**

**$ cd /users/mary**

**$ pwd**

**/users/mary**

**$ cd /tmp**

**$ pwd**

**/tmp**

**$ cd /users/john/portfolio**

**$ pwd**

**/users/john/portfolio**

**$**

Every directory or file on the file system has a unique absolute pathname. Although john may create a file called "test.txt" in his home directory and mary may create a file called test.txt in her home directory, the absolute pathnames of these files are different. John's is called /users/john/test.txt, and Mary's is /users/mary/test.txt.

**2. Directory Abbreviations**

|  |  |
| --- | --- |
| **Keyword** | **Description** |
| ~ | Your home (login) directory |
| ~username | Another user's home directory |
| . | Working (current) directory |
| .. | Parent of working directory |
| ../.. | Parent of parent directory |

**Examples**

**cp foo bar**

Copy a file named "foo" (in the current directory); name the copy "bar"

**cp foo ~/Documents**

Copy a file named "foo" (in the current directory) into your Documents directory

**cp foo ~/Documents/bar**

Copy a file named "a" (in the current directory) into your Documents directory and name the copy "bar"

**cp \*.jpg ~/Documents**

Copy all files with names ending in ".jpg" into your Documents directory

**cp -R Documents "Documents backup"**

Copy an entire directory named "Documents"; name the copy "Documents backup". The quotes are needed because of the space in the directory name.

**sudo cp -Rp /Users "/Users backup"**

Copy the entire /Users directory (including all of the user home folders inside it), preserving as much as possible of the files' information (ownership, permissions, etc, but not resource forks) as cp knows how to; name the copy "Users backup". Root access is required to use -p, so the example uses sudo to get root access temporarily.

**Chapter 6**

**File Permissions**

**1. Understanding file permissions on Unix: a brief tutorial**

Every user on a Unix system has a unique username, and is a member of at least one group (the primary group for that user). This group information is held in the password file (/etc/passwd). A user can also be a member of one or more other groups. The auxiliary group information is held in the file /etc/group. Only the administrator can create new groups or add/delete group members (one of the shortcomings of the system).

Every directory and file on the system has an owner, and also an associated group. It also has a set of permission flags which specify separate read, write and execute permissions for the 'user' (owner), 'group', and 'other' (everyone else with an account on the computer) The 'ls' command shows the permissions and group associated with files when used with the ‘-l’ option. On some systems (e.g. Coos), the '-g' option is also needed to see the group information.

An example of the output produced by 'ls -l' is shown below.

**drwx------ 2 richard staff 2048 Jan 2 1997 private**

**drwxrws--- 2 richard staff 2048 Jan 2 1997 admin**

**-rw-rw---- 2 richard staff 12040 Aug 20 1996 admin/userinfo**

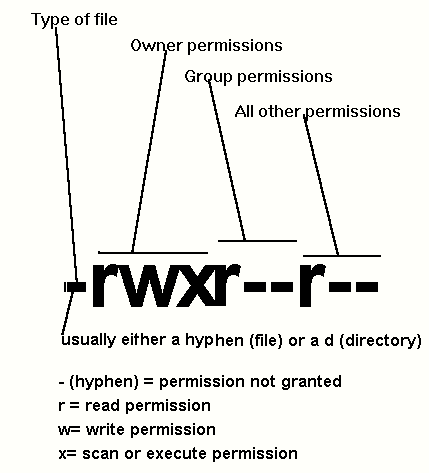
**drwxr-xr-x 3 richard user 2048 May 13 09:27 public**

Understanding how to read this output is useful to all unix users, but especially people using group access permissions.

* Field 1: a set of ten permission flags.
* Field 2: link count (don't worry about this)
* Field 3: owner of the file
* Field 4: associated group for the file
* Field 5: size in bytes
* Field 6-8: date of last modification (format varies, but always 3 fields)
* Field 9: name of file (possibly with path, depending on how ls was called)

The permission flags are read as follows (left to right)

|  |  |
| --- | --- |
| **position** | **Meaning** |
| **1** | Directory flag, 'd' if a directory, '-' if a normal file, something else occasionally may appear here for special devices. |
| **2,3,4** | read, write, execute permission for User (Owner) of file |
| **5,6,7** | read, write, execute permission for Group |
| **8,9,10** | read, write, execute permission for Other |
| **value** | **Meaning** |
| **-** | In any position means that flag is not set |
| **R** | file is readable by owner, group or other |
| **W** | File is writeable. On a directory, write access means we can add or delete files |
| **X** | File is executable (only for programs and shell scripts - not useful for data files). Execute permission on a directory means we can list the files in that directory |
| **S** | In the place where 'x' would normally go is called the set-UID or set-groupID flag. |



Graphical representation of file permission field

**2. Difference in access permissions for files and folders**

Access permissions for files and folders mean different things from the user standpoint. The table below shows the difference.

|  |  |  |
| --- | --- | --- |
| Access type | File | Directory |
| Read (r) | the file can be read by the corresponding userid or group to which this set of symbols applies | If the directory listing can be obtained |
| Write (w) | The file can be changed or deleted by the corresponding user or group to which this set of symbols applies | If user or process can change directory contents somehow: create new or delete existing files in the directory or rename files. |
| Execute (x) | The file is considered executable, and may be executed by the user or group to which this set of symbols applies | If user or process can access the directory, that is, go to it (make it to be the current working directory) |

On an executable program with set-UID or set-groupID, that program runs with the effective permissions of its owner or group.

For a directory, the set-groupID flag means that all files created inside that directory will inherit the group of the directory. Without this flag, a file takes on the primary group of the user creating the file. This property is important to people trying to maintain a directory as group accessible. The subdirectories also inherit the set-groupID property.

**3. The default file permissions (umask):**

Each user has a default set of permissions which apply to all files created by that user, unless the software explicitly sets something else. This is often called the 'umask', after the command used to change it. It is either inherited from the login process, or set in the .cshrc or .login file which configures an individual account, or it can be run manually.

Typically the default configuration is equivalent to typing 'umask 22' which produces permissions of:

**-rw-r--r-- for regular files, or**

**drwxr-xr-x for directories.**

In other words, user has full access, everyone else (group and other) has read access to files, lookup access to directories.

When working with group-access files and directories, it is common to use 'umask 2' which produces permissions of:

**-rw-rw-r-- for regular files, or**

**drwxrwxr-x for directories.**

For private work, use 'umask 77' which produces permissions:

**-rw------- for regular files, or**

**drwx------ for directories.**

The logic behind the number given to umask is not intuitive.

The command to change the permission flags is "chmod". Only the owner of a file can change its permissions.

The command to change the group of a file is "chgrp". Only the owner of a file can change its group, and can only change it to a group of which he is a member.

See the online manual pages for details of these commands on any particular system (e.g. "man chmod").

The basic form of the chmod command is:

**chmod who add-or-remove what\_permissions filename**

**Note:**

there should not be any spaces between the "who", "add-or-remove", and "what\_permissions" portions of the command, in a real chmod command. The spaces were included in the above diagram to make it more readable. See the following examples for samples of proper syntax.)

We'll break that diagram down a little further, and then give some examples.

**Command "Breakdown": chmod**

**chmod**

This is the name of the command.

**who**

Any combination of u (for "user"), g (for "group"), or o (for "others"), or a (for "all"--that is, user, group, and others).

**add-or-remove**

Use + to add the attribute (set the flag), or - to remove the attribute (clear the flag).

**what\_permissions**

Any combination of r (for Read), w (for Write), or x (for Execute).

**filename**

A file or directory name (or wildcard pattern) to which we wish to apply the listed permission changes.

Examples of typical useage are given below:

1. Files in the current directory readable by anyone.

**$chmod a+r \***

NOTE: -rw-r--r-- - - - myfile (myfile is having these permissions)

2. Give group write permission to "myfile", leaving all other permission flags alone

**$ chmod g+w myfile (symbolic mode)**

**$ chmod 664 myfie (numeric mode)**

3. Remove read and write access to "myfile", leaving all other permission flags alone

**$ chmod g-rw myfile (symbolic mode)**

**$ chmod 604 myfile (numeric mode)**

4. Give group read write access to this directory, and everything inside of it (-R = recursive)

**$ chmod -R g+rw (symbolic mode)**

**$ chmod –R 664 (numeric mode)**

5. To give read, write to user(owner) , to give read permission and remove write permission to Group to remove all permission for others to the file named ‘bigfile’ $ chmod u+rw, g+r, g-w, o-a bigfile (symbolic mode)

**$ chmod 6 4 0 bigfile (numeric mode)**

6. Change the ownership of this directory to group 'medi' and everything inside of it (-R = recursive). The person issuing this command must own all the files or it will fail.

**$ chgrp -R medi . (symbolic mode)**

**$ There is no numeric mode equality command for this**

**Warnings:**

Putting 'umask 2' into a startup file (.login or .cshrc) will make these settings apply to everything we do unless manually changed. This can lead to giving group access to files such as saved email in your home directory, which is generally not desirable.

Making a file group read/write without checking what its group is can lead to accidentally giving access to almost everyone on the system. Normally all users are members of some default group such as "users", as well as being members of specific project-oriented groups. Don't give group access to "users" when we intended some other group.

Remember that to read a file, we need execute access to the directory it is in AND read access to the file itself. To write a file, your need execute access to the directory AND write access to the file. To create new files or delete files, we need write access to the directory. We also need execute access to all parent directories back to the root. Group access will break if a parent directory is made completely private.

**Chapter 7**

**Pipes and Filters**

**1. Pipes and Filters**

The purpose of this lesson is to introduce us to the way that we can construct powerful Unix command lines by combining Unix commands.

**1.1 Concepts**

Unix commands alone are powerful, but when we combine them together, we can accomplish complex tasks with ease. The way we combine Unix commands is through using pipes and filters.

**1.2 using a Pipe**

The symbol | is the Unix pipe symbol that is used on the command line. What it means is that the standard output of the command to the left of the pipe gets sent as standard input of the command to the right of the pipe. Note that this functions a lot like the > symbol used to redirect the standard output of a command to a file. However, the pipe is different because it is used to pass the output of a command to another command, not a file.

Here is an example:

**$ cat apple.txt**

**core**

**worm seed**

**jewel**

**$ cat apple.txt | wc**

**3 4 21**

**$**

In this example, at the first shell prompt, we show the contents of the file apple.txt to us. In the next shell prompt, we use the cat command to display the contents of the applex.txt file, but we sent the display not to the screen, but through a pipe to the wc (word count) command. The wc command then does its job and counts the lines, words, and characters of what it got as input.

We can combine many commands with pipes on a single command line. Here's an example where we count the characters, words, and lines of the apple.txt file, then mail the results to nobody@december.com with the subject line "The count."

**$ cat apple.txt | wc | mail -s "The count" nobody@december.com**

**1.3 using filter**

In UNIX and UNIX-like operating systems, a filter is program that gets most of its data from standard input (the main input stream) and writes its main results to standard output (the main output stream). UNIX filters are often used as elements of pipelines. The pipe operator ("|") on a command line signifies that the main output of the command to the left is passed as main input to the command on the right.

List of UNIX filter programs

* awk
* cut
* grep
* head
* sed
* sort
* tail
* tac
* tee
* tr
* uniq
* wc

**1.3.1 head**

head is a program on Unix and Unix-like systems used to display the first few lines of a text file or piped data. The command-syntax is:

**$ head [options] <file\_name>**

By default, head will print the first 10 lines of its input to the standard output. The number of lines printed may be changed with a command line option. The following example shows the first 20 lines of filename:

**$ head -20 filename**

This displays the first 5 lines of all files starting with foo:

**$ head -5 foo\***

**1.3.2 tail**

tail is a program on Unix and Unix-like systems used to display the last few lines of a text file or piped data. The command-syntax is:

**$ tail [options] <file\_name>**

By default, tail will print the last 10 lines of its input to the standard output. With command line options the number of lines printed and the printing units (lines, blocks or bytes) may be changed. The following example shows the last 20 lines of filename:

**$ tail -20 filename**

This example show all lines of filename after the first 2 lines:

**$ tail +2 filename**

**File monitoring**

tail has a special command line option -f (follow) that allows a file to be monitored. Instead of displaying the last few lines and exiting, tail displays the lines and then monitors the file. As new lines are added to the file by another process, tail updates the display. This is particularly useful for monitoring log files. The following command will display the last 10 lines of messages and append new lines to the display as new lines are added to messages:

**$ tail -f /var/adm/messages**

To interrupt tail while it is monitoring, break-in with CTRL-C

**1.3.3 more**

more is better, isn't it? Better than what? Better than the cat command. cat dumps its arguments to stdout, which is the terminal (unless we redirect it with > or >>). But what if we're working on your dissertation, and we'd like to read it page by page, we'd use a command like:

**$ more dissertation.txt**

This will generate a nice page-by-page display of your masterpiece. Type

**$ man more**

at a command prompt, and check out the man page to get more details. Here we are only going to tell us the most important features of more ( i.e. the features that we use). There are three important things we should know:

7. Typing q while examining a file quits more

8. Typing /SEARCHSTRING while examining a file searches for SEARCHSTRING

9. more is a great example of a filter

**1.3.4 less**

Opposite of the more command.

Both less and more display the contents of a file one screen at a time, waiting for us to press the Spacebar between screens. This lets us read text without it scrolling quickly off your screen. The less utility is generally more flexible and powerful than more, but more is available on all Unix systems while less may not be.

The less command is a pager that allows us to move forward and backward (instead of only forward, as the more pager behaves on some systems) when output is displayed one screen at a time. :

To read the contents of a file named textfile in the current directory, enter:

**$ less textfile**

The less utility is often used for reading the output of other commands. For example, to read the output of the ls command one screen at a time, enter:

**$ ls -la | less**

In both examples, we could substitute more for less with similar results. To exit either less or more, press q . To exit less after viewing the file, press q .

**1.3.5 wc**

In Unix, to get the line, word, or character count of a document, use the wc command. At the Unix shell prompt, enter: wc filename Replace filename with the file or files for which we want information. For each file, wc will output three numbers. The first is the line count, the second the word count, and the third is the character count. For example, if we entered wc .login, the output would be something similar to the following: 38 135 847 .login To narrow thefocus of your query, we may use one or more of the following wc options:

|  |  |
| --- | --- |
| Option | Entities counted |
| -c | bytes |
| -l | lines |
| -m | characters |
| -w | words |

Note: In some versions of wc, the -m option will not be available or -c will report characters. However, in most cases, the values for -c and -m are equal.

**Syntax:**

To count the characters in a file. Here it counts the no of characters in the file abc.txt

**$ Wc –c / abc.txt**

For example, to find out how many bytes are in the .login file, we could enter:

**$ wc -c .login**

We may also pipe standard output into wc to determine the size of a stream. For example, to find out how many files are in a directory, enter:

**/bin/ls -l | wc -l**

**1.3.6 sort**

sort is a standard Unix command line program that prints the lines of its input or concatenation of all files listed in it's argument list in sorted order. The -r flag will reverse the sort order.

Example of sort in action:

**$ cat phonebook**

**Smith, Brett 555-4321**

**Doe, John 555-1234**

**Doe, Jane 555-3214**

**Avery, Cory 555-4321**

**Fogarty, Suzie 555-2314**

**$ cat phonebook | sort**

**Avery, Cory 555-4321**

**Doe, Jane 555-3214**

**Doe, John 555-1234**

**Fogarty, Suzie 555-2314**

**Smith, Brett 555-4321**

The -n option makes the program to sort according to numerical value:

**$ du /bin/\* | sort -n**

**4 /bin/domainname**

**4 /bin/echo**

**4 /bin/hostname**

**4 /bin/pwd**

**...**

**24 /bin/ls**

**30 /bin/ps**

**44 /bin/ed**

**54 /bin/rmail**

**80 /bin/pax**

**102 /bin/sh**

**304 /bin/csh**

The -n +1 option makes the program to sort according to numerical value, using the second column of data:

**$ cat zipcode**

**Adam 12345**

**Bob 34567**

**Joe 56789**

**Sam 45678**

**Wendy 23456**

**$ cat zipcode | sort -n +1**

**Adam 12345**

**Wendy 23456**

**Bob 34567**

**Sam 45678**

**Joe 56789**

The -r option just reverses the order of the sort:

**$ cat zipcode | sort -nr +1**

**Joe 56789**

**Sam 45678**

**Bob 34567**

**Wendy 23456**

**Adam 12345**

**1.3.7 tr**

tr (abbreviated from translate or transliterate) is a command in Unix-like operating systems.

When executed, the program reads from the standard input and writes to the standard output. It takes as parameters two sets of characters, and replaces occurrences of the characters in the first set with the corresponding elements from the other set. The following inputs, for instance, shift the input letters of the alphabet back by one character.

**$ echo "ibm 9000" >computer.txt**

**$ tr a-z za-y <computer.txt**

**hal 9000**

Note: when ever we are using the “tr” operator we have to use inpur rediction operator

In some older versions of tr (not POSIX-compliant), the character ranges must be enclosed in brackets, which must then be quoted against interpretation by the shell:

**$ tr "[a-z]" "z[a-y]" <computer.txt**

If it's not known in advance which variant of tr is being invoked, then in this example one would have to write the ranges in unabbreviated form (tr abcdefghijklmnopqrstuvwxyz zabcdefghijklmnopqrstuvwxy). For some applications,a single invocation is portable despite containing ranges: ROT13 can be portably implemented as tr "[A-M][N-Z][a-m][n-z]" "[N-Z][A-M][n-z][a-m]". (This works because the brackets, which are not part of the range syntax in POSIX tr, align properly in the two strings, and hence will be safely interpreted as a request to map the bracket character to itself.)

Perl also has a tr operator, which operates analogously

**1.3.8 cut**

cut is a Unix command which is typically used to extract a certain range of characters from a line, usually from a file.

**Syntax**

**cut [-c] [-f list] [-d delim] [-s] [file]**

Flags which may be used include

* -c Characters; a list following -c specifies a range of characters which will be returned, e.g. cut -c1-66 would return the first 66 characters of a line
* -f Specifies a field list, separated by a delimiter
* list A comma separated or blank separated list of integer denoted fields, incrementally ordered. The - indicator may be supplied as shorthand to allow inclusion of ranges of fields e.g. 4-6 for ranges 4 - 6 or 5 – as shorthand for field 5 to the end, etc.
* -d Delimiter; the character immediately following the -d option is the field delimiter for use in conjunction with the -f option; the default delimiter is tab. Space and other characters with special meanings within the context of the shell in use must be enquoted or escaped as necessary.
* -s Bypasses lines which contain no field delimiters when -f is specified, unless otherwise indicated.
* file The file (and accompanying path if necessary) to process as input. If no file is specified then standard input will be used

**Example**

Extract columns of data

**$ cut -f -3,5,7-9 -d ' ' infile1 > outfile1**

**-f 2,4-6 field**

**-c 35-44 character**

**-d ':' delimiter (default is a tab)**

**1.3.9 paste**

Paste is a Unix utility tool which is used to join files horizontally (parallel merging), e.g. to join two similar length files which are comma delimited. It is effectively the horizontal equivalent to the utility cat command which operates on the vertical plane of two (or more) files, i.e. by adding one file to another in order.

**Example**

To paste several columns of data together, enter:

**$ paste who where when > www**

This creates a file named www that contains the data from the names file in one column, the places file in another, and the dates file in a third. If the names, places, and dates file look like:

**Who where when**

**Sam Detroit January 3**

**Dave Edgewood February 4**

**Sue Tampa March 19**

then the www file will contain:

**Sam Detroit January 3**

**Dave Edgewood February 4**

**Sue Tampa March 19**

**1.3.10 uniq**

uniq is a Unix utility which, when fed a text file, outputs the file with adjacent identical lines collapsed to one. It is a kind of filter program. Typically it is used after sort. It can also output only the duplicate lines (with the -d option), or add the number of occurrences of each line (with the -c option).

An example: To see the list of lines in a file, sorted by the number of times each occurs:

**$ sort file|uniq -c|sort -n**

Using uniq like this is common when building pipelines in shell scripts.

**Switches**

* -u Print only lines which are not repeated in the original file
* -d Print one copy only of each repeated line in the input file.
* -c Generate an output report in default style except that each line is preceded by a count of the number of times it occurred. If this option is specified, the -u and -d options are ignored if either or both are also present.
* -i Ignore case differences when comparing lines
* -s Skips a number of characters in a line
* -w Specifies the number of characters to compare in lines, after any characters and fields have been skipped
* --help Displays a help message
* --version Displays version number on stdout and exits.

**Chapter 8**

**vi Editor**

**1. General Introduction**

The vi editor (short for visual editor) is a screen editor which is available on almost all Unix systems. Once we have learned vi, we will find that it is a fast and powerful editor. vi has no menus but instead uses combinations of keystrokes in order to accomplish commands. If we are just beginning to learn Unix, we might find the Pico editor easier to use (most command options are displayed at the bottom of the screen). If we use the Pine email application and have composed or replied to a message we have probably already used Pico as it is used for text entry. For more information please refer to the Pine/Pico page.

**1.1 Starting vi**

To start using vi, at the Unix prompt type vi followed by a file name. If we wish to edit an existing file, type in its name; if we are creating a new file, type in the name we wish to give to the new file.

**$ vi filename**

Then hit Return. We will see a screen similar to one below which shows blank lines with tildes and the name and status of the file.

**~**

**"myfile" [New file]**

**1.2 vi's Modes**

vi has two modes: the command mode and the insert mode. It is essential that we know which mode we are in at any given point in time. When we are in command mode, letters of the keyboard will be interpreted as commands. When we are in insert mode the same letters of the keyboard will type or edit text. vi always starts out in command mode. When we wish to move between the two modes, keep these things in mind. We can type i to enter the insert mode. If we wish to leave insert mode and return to the command mode, hit the ESC key. If we're not sure where we are, hit ESC a couple of times and that should put us back in command mode.

**2. General Command Informati on**

As mentioned previously, vi uses letters as commands. It is important to note that in general vi commands:

* are case sensitive - lowercase and uppercase command letters do different things
* are not displayed on the screen when we type them
* Generally do not require a Return after we type the command.

We will see some commands which start with a colon (:). These commands are ex commands which are used by the ex editor. ex is the true editor which lies underneath vi -- in other words, vi is the interface for the ex editor.

**2.1 Entering Text**

To begin entering text in an empty file, we must first change from the command mode to the insert mode. To do this, type the letter i. When we start typing, anything we type will be entered into the file. Type a few short lines and hit Return at the end of each of line. Unlike word processors, vi does not use word wrap. It will break a line at the edge of the screen. If we make a mistake, we can use the Backspace key to remove your errors. If the Backspace key doesn't work properly on your system, try using the Ctrl h key combination.

**2.2 Cursor Movement**

We must be in command mode if we wish to move the cursor to another position in your file. If we've just finished typing text, we're still in insert mode and will need to press ESC to return to the command mode.

**2.2.1 Moving One Character at a Time**

Try using your direction keys to move up, down, left and right in your file. Sometimes, we may find that the direction keys don't work. If that is the case, to move the cursor one character at the time, we may use the h, j, k, and l keys. These keys move we in the following directions:

|  |  |
| --- | --- |
| h | left one space |
| l | right one space |
| j | down one space |
| k | up one space |

If we move the cursor as far as we can in any direction, we may see a screen flash or hear a beep.

**2.2.2 Moving among Words and Lines**

While these four keys (or your direction keys) can move us just about anywhere we want to go in your file, there are some shortcut keys that we can use to move a little more quickly through a document. To move more quickly among words, we might use the following:

|  |  |
| --- | --- |
| w | Moves the cursor forward one word |
| b | Moves cursor backward one word (if in middle of a word, b will move us to beginning of the current word). |
| e | Moves to the end of a word. |

To build on this further, we can precede these commands with a number for greater movement. For example, 5w would move us forward five words; 12b would move us backwards twelve words. [We can also use numbers with the commands mentioned earlier. For example, 5j would move us down 5 characters.]

**2.3 Command Keys and Case**

We will find when using vi that lower case and upper case command keys are interpreted differently. For example, when using the lower case w, b, and e commands, words will be defined by a space or a punctuation mark. On the other hand, W, B, and E commands may be used to move between words also, but these commands ignore punctuation.

**2.4 Shortcuts**

Two short cuts for moving quickly on a line include the $ and the 0 (zero) keys. The $ key will move us to the end of a line, while the 0 will move us quickly to the beginning of a line.

**2.5 Screen Movement**

To move the cursor to a line within your current screen use the following keys:

|  |  |
| --- | --- |
| h | Moves the cursor to the top line of the screen. |
| m | Moves the cursor to the middle line of the screen. |
| l | Moves the cursor to the last line of the screen. |

To scroll through the file and see other screens use:

|  |  |
| --- | --- |
| ctrl-f | scrolls down one screen |
| ctrl-b | scrolls up one screen |
| ctrl-u | scrolls up a half a screen |
| ctrl-d | scrolls down a half a screen |

Two other useful commands for moving quickly from one end to the other of a document are G to move to the end of the file and 1G to move to the beginning of the file. If we precede G with a number, we can move to a specific line in the document (e.g. 15G would move us to line 15).

**2.6 Moving by Searching**

One method for moving quickly to a particular spot in your file is to search for specific text. When we are in command mode, type a / followed the text we wish to search for. When we press Return, the cursor will move to the first incidence of that string of text. We can repeat the search by typing n or search in a backwards direction by using N.

**Basic Editing**

To issue editing commands, we must be in command mode. As mentioned before, commands will be interpreted differently depending upon whether they are issued in lower or upper case. Also, many of the editing commands can be preceded by a number to indicate a repetition of the command.

**3. Deleting (or Cutting) Characters, Words, and Lines**

To delete a character, first place your cursor on that character. Then, we may use any of the following commands:

|  |  |
| --- | --- |
| x | Deletes the character under the cursor. |
| X | Deletes the character to the left of your cursor. |
| dw | Deletes from the character selected to the end of the word. |
| dd | Deletes all the current line. |
| D | Deletes from the current character to the end of the line. |

Preceding the command with a number will delete multiple characters. For example, 10x will delete the character selected and the next 9 characters; 10X will delete the 10 characters to the left of the currently selected character. The command 5dw will delete 5 words, while 4dd deletes four lines.

**4. Pasting Text using Put**

When we delete or cut text, we may need to reinsert it in another location of the document. The Put command will paste in the last portion of text that was deleted since deleted text is stored in a buffer. To use this command, place the cursor where we need the deleted text to appear. Then use p to reinsert. If we are inserting a line or paragraph use the lower case p to insert on the line below the cursor or upper case P to place in on the line above the cursor.

**5. Copying Text with Yank**

If we wish to make a duplicate copy of existing text, we may use the yank and put commands to accomplish this function. Yank copies the selected text into a buffer and holds it until another yank or deletion occurs. Yank is usually used in combination with a word or line object such as the ones shown below:

|  |  |
| --- | --- |
| yw | copies a word into a buffer (7yw copies 7 words) |
| yy | copies a line into a buffer (3yy will copy 3 lines) |

Once the desired text is yanked, place the cursor in the spot in which we wish to insert the text and then use the put command (p for line below or P for line above) to insert the contents of the buffer.

**6. Replacing or Chang ing Characters, Words, and Li nes**

When we are using the following commands to replace text, we will be put temporarily into insert mode so that we can change a character, word, line, or paragraph of text.

|  |  |
| --- | --- |
| r | Replaces the current character with the next character we enter/type. Once we enter the character we are returned to command mode. |
| R | Puts us in overtype mode until we hit ESC which will then return us to command mode. |
| cw | Changes and replaces the current word with text that we type. A dollar sign marks the end of the text we're changing. Pressing ESC when we finish will return us to command mode. |

**6.1 Inserting Text**

If we wish to insert new text in a line, first position the cursor to the right of where we wish the inserted text to appear. Type i to get into insert mode and then type in the desired text (note that the text is inserted before the cursor). Press ESC to return to command mode.

**6.2 Inserting a Blank Line**

To insert a blank line below the line your cursor is currently located on, use the o key and then hit ESC to return to the command mode . Use O to insert a line above the line the cursor is located on.

**6.3 Appending Text**

We can use the append command to add text at any place in your file. Append (a) works very much like Insert (i) except that it insert text after the cursor rather than before it. Append is probably used most often for adding text to the end of a line. Simply place your cursor where we wish to append text and press a. Once we've finished appending, press ESC to go back to command mode.

**6.4 Joining Lines**

Since vi does not use automatic word wrap, it is not unusual in editing lines to end up with lines that are too short and that might be improved if joined together. To do this, place your cursor on the first line to be joined and type J. As with other commands, we can precede J with a number to join multiple lines (4J joins 4 lines).

**6.5 Undoing**

Be sure to remember this command. When we make a mistake we can undo it. DO NOT move the cursor from the line where we made the change. Then try using one of the following two commands:

|  |  |
| --- | --- |
| u | undoes the last change we made anywhere in the file. Using **u** again will "undo the undo". |
| U | undoes all recent changes to current line. We can not have moved from the line to recover the original line. |

**7. Closing a nd Saving Files**

When we edit a file in vi, we are actually editing a copy of the file rather than the original. The following sections describe methods we might use when closing a file, quitting vi, or both.

**7.1 Quitting and Saving a File**

The command ZZ (notice that it is in uppercase) will allow us to quit vi and save the edits made to a file. We will then return to a Unix prompt. Note that we can also use the following commands:

|  |  |
| --- | --- |
| :w | to save your file but not quit vi (this is good to do periodically in case of machine crash!). |
| :q | to quit if we haven't made any edits. |
| :wq | to quit and save edits (basically the same as ZZ). |

**7.2 Quitting without Saving Edits**

Sometimes, when we create a mess (when we first start using vi this is easy to do!) we may wish to erase all edits made to the file and either start over or quit. To do this, we can choose from the following two commands:

|  |  |
| --- | --- |
| :e! | Reads the original file back in so that we can start over. |
| :q! | Wipes out all edits and allows us to exit from vi. |

**More about Combining Commands, Objects, and Numbers**

Now that we've learned some basic vi commands we might wish to expand your skills by trying some fancy combination steps. Some commands are generally used in combination with a text object. We've already seen some examples of this. For example, when we use the command dw to delete a word, that combines the delete (d) command with the word (w) text object. When we wish to delete multiple words, we might add a number to this combination. If we wished to delete 2 words we might use 2dw or d2w. Either of these combinations would work. So, as we can see, the general format for a command can be

**(number) (command) (text object) or (command) (number) (text object)**

We might wish to try out some of the following combinations of commands and objects:

|  |  |
| --- | --- |
| **Command** | **Description** |
| **d** | (delete) |
| **w** | (word to the left) |
| **y** | (yank/copy) |
| **b** | (word to the right or backward) |
| **c** | (change) |
| **e** | (end of word) |
| **H** | (top of the screen) |
| **L** | (bottom of the screen) |
| **M** | (middle of the screen) |
| **0** | (zero - first character on a line) |
| **$** | (end of a line) |
| **(** | (previous sentence) |
| **)** | (next sentence) |
| **[** | (previous section) |
| **]** | (next section) |

**8. Repeating a Command**

If we are doing repetitive editing, we may wish to use the same command over and over. vi will allow we to use the dot (.) to repeat the last basic command we issued. If for example, we wished to deleted several lines, we could use dd and then . (dot) in quick succession to delete a few lines.

**A Quick Word about Customizing Your vi Environment**

There are several options that we can set from within vi that can affect how we use vi. For example, one option allows us to set a right margin that will then force vi to automatically wrap your lines as we type. To do this, we would use a variation of the :set command. The :set command can be used to change various options in vi. In the example just described, we could, while still in vi, type :set wrapmargin=10 to specify that we wish to have a right margin of 10. Another useful option is :set number. This command causes vi to display line numbers in the file we are working on.

**8.1 Other Options**

To view a listing of other options, we could type :set all. To view only those options which are currently in effect, we can type set: by itself. Options that we set while in a vi session will apply during that session only. To make permanent changes to your vi environment, we could edit your .exrc file. However, we should not edit this file unless we know what we are doing!

**9. Useful vi Commands**

**9.1 Cut/Paste Commands**

|  |  |
| --- | --- |
| x | delete one character (destructive backspace) |
| dw | delete the current word (Note: ndw deletes n numbered words) |
| dd | delete the current line (Note: ndd deletes n numbered lines) |
| D | delete all content to the right of the cursor |
| d$ | same as above |
| :u | undo last command |
| p,P | paste line starting one line below/above current cursor location |
| J | combine the contents of two lines |
| "[a-z]nyy | yank next n lines into named buffer [a-z] |
| "[a-z]p/P | place the contents of selected buffer below/above the current line |

**9.2 Extensions to the Above Commands:**

|  |  |
| --- | --- |
| :3,18d | delete lines 3 through 18 |
| 16,25m30 | move lines 16 through 25 to after line 30 |
| 23,29co62 | copy specified lines and place after line 62 |

**9.3 Cursor Relocation commands:**

|  |  |
| --- | --- |
| 0 | goto line [n] |
| :[n] | place cursor on last line of text |
| shift g | move cursor left, right, down and up |
| h/l/j/k | move forward, backward in text, one page |
| ^f/^b | move up, down one half page |
| ^u/^d | move to end of line |
| $ | move to beginning of line |

**9.4 Extensions to the Above:**

|  |  |
| --- | --- |
| b | move backwards one word (Note: nb moves back n number of words) |
| e | move to end of current word |
| ( | move to beginning of curent block |
| ) | move to the end of current block |

**Searching and Substitution commands**

|  |  |
| --- | --- |
| / [string] | search forward for string |
| ? [string] | search backwards for string |
| n | repeat last search |
| N | repeat search in opposite direction |
| cw | change the contents of the current word, (use ESC to stop replacement mode) |
| c$ | Replace all content to the right of cursor (exit replacement mode with ESC) |
| c0 | Replace all content to the left of cursor (exit with ESC) |
| :1,$s/s1/s2/g | Yow!) global replacement of string1 with string2 |
| r | replace current character with next character typed |

**9.5 Entering the Insert Mode:**

|  |  |
| --- | --- |
| i | Begin inserting text at current cursor location |
| I | Begin inserting text at the beginning of the current line |
| a | Begin appending text, one character to the right of current cursor location |
| A | Begin appending text at the end of the current line |
| o/O | Begin entering text one line below\above current line |
| ESC | Exit insertion mode and return to command mode |

**9.6 Exiting and Entering VI**

|  |  |
| --- | --- |
| ZZ | save file and exit VI |
| :wq | same as above |
| :e! | return to last saved version of current file |
| :q | quit without save, (Note :q! is required if changes have been made) |
| :w | write without exit (:w! to force write) |

**9.7 Fancy Stuff:**

|  |  |
| --- | --- |
| :1,10w file | write lines 1 through 10 to file newfile |
| :340,$w >> file | write lines 340 through the end of the file and append to file newfile |
| :sh | escape temporarily to a shell |
| ^d | return from shell to VI |
| :![command] | execute UNIX command without leaving VI |
| :r![command] | read output of command into VI |
| :r[filename] | read filename into VI |
| :$r newfile | read in newfile and attach at the end of current document |
| :r !sort file | read in contents of file after it has been passed through the UNIX sort |
| :n | open next file (works with wildcard filenames,ex: vi file\*) |
| :^g | list current line number |
| :set number | show line numbers |
| :set showinsert | show flag ("I") at bottom of screen when in insert mode |
| :set all | display current values of VI variables |
| :set ai | set autoindent; after this enter the insert mode and tab, from this point on VI will indent each line to this location. Use ESC to stop the indentations. |
| ^T | set the autoindent tab one tab stop to the right |
| ^D | set the autoindent tab one stop to the left |
| :set tabstop=n | sets default tab space to number n |
| >> | shift contents of line one tab stop to the right |
| << | shift contents of line one tab stop to the left |

**10. COMMAND SUMMARY TEST**

Answer the following questions before continuing with the remainder of the exercises.

10. What is the command to move up one line?

11. What is the command to move down one line?

12. What is the command to delete a line?

13. What is the command to abandon the editor, and return to the shell?

14. What is the command to undo the last command?

15. What is the command to position the cursor at the end of the current line?

16. What is the command to write the buffer to the file and remain in vi?

17. What is the command to erase the character at the cursor position?

18. What is the command to position the cursor at the beginning of the previous word?

**Chapter 9**

**$Find and other useful Commands**

**1. find**

The "find" command is very powerful. It can search the entire filesystem for one or more files that we specify to look for. This is very helpful when a file has been "lost".

We can also use the find command to locate files, and then perform some type of action on the files after they've been located. With this capability, we can locate files using powerful search criteria, and then run any Unix command we want on the files we locate. (See the examples below.):

The find command allows the Unix user to process a set of files and/or directories in a file subtree.

We can specify the following:

* where to search (pathname)
* what type of file to search for (-type: directories, data files, links)
* how to process the files (-exec: run a process against a selected file)
* the name of the file(s) (-name)
* perform logical operations on selections (-o and -a)

**Syntax**

$ find <path> options action

Major options of the find command include

* -name : Finds files with certain naming conventions in the directory structure
* -ctime ***time interval***Locates files that that were created during the specified time interval
* -mtime ***time interval*** Finds files that have been modified during the specified time interval
* -atime **time interval** Locates files that have been accessed during the specified time interval
* -perm permissions Locates files with certain permission settings
* -user Locates files that have specified ownership
* -group Locates files that are owned by specified group
* -size Locates files with specified size
* -type Locates a certain type of file

Time interval in options -ctime, -mtime and -atime is an integer with optional sign.

* n: If the integer n does not have sign this means exactly n days ago, 0 means today.
* +n: if it has plus sing, then it means "more then n days ago", or older then n ,
* -n: if it has the minus sign, then it means less than n days ago (-n), or younger then n.
* It's evident that -1 and 0 are the same and means "today".

It is possible to locate files and directories that match or do not match multiple conditions, for example:

* a to have multiple conditions ANDed
* o to have multiple conditions ORed
* ! to negate a condition
* Expression to satisfy any complex condition

It is possible to specify the action to be taken on the files or directories that are found:

* print prints the names of the files on standard output (usually enabled by default)
* exec command executes the specified command.

The most common reason for using the find command is to utilize its capability to recursively process the subdirectories. For example, if we want to obtain a list of all files accessed in the last 24 hours, execute the following command:

**$ find . -atime 0 -print**

If the system administrator wants a list of .profile used by all users, the following command should be executed:

**$ find / -name .profile -print**

We can also execute the find command with multiple conditions. If we wanted to find a list of files that have been modified in the last 24 hours and which has a permission of 777, we would execute the following command:

**$ find . -perm 777 -a -mtime 0 –print**

**Total Options:**

|  |  |
| --- | --- |
| -atime n | True if the file was accessed n days ago. find updates the access times of the directories in the *pathname\_list.* |
| -ctime n | True if the file's status was changed n days ago. A change to a file means the inode of the file was modified. |
| -depth | Always true if specified on the command line. Causes find to descend the directory structures and perform the specified options on the subdirectories and files before processing the top directory. We may find this useful when using find to pipe pathnames to cpio. If we do not have write permission in directories this will allow we to transfer the files to the archive file. |
| -exec cmd | True if the command cmd executes successfully and returns a zero exit status. We can use a set of braces ({}) to signify the presence of the current pathname. The cmd command must end with an escaped semicolon (\;). For example,  **$ find . -name '\*.o' -exec rm {} \;**  Would remove all files ending with a .o from the current directory structure. |
| -group name | True if the file belongs to the group *name* .We may specify a group ID instead of a group name. |
| -inum n | True if the file has inode number n. |
| -links n | True if the file has n links. |
| -local | True if the specified file physically resides on the local UNIX system. This is useful when searching directories on systems that are networked together and share disk space. |
| -mtime n | True if the file's contents were modified n days ago. |
| -name file | True if the filename matches file. The shell's filename generation characters may be used ifenclosed in quotes. For example,  **$ find . -name '\*.c' –print**  Prints all files ending with a .c in the current directory or any of its subdirectories. |
| -newer file | True if the current file is newer than the specified file . The modification dates are compared to decide which is newer. |
| -nogroup | True if the file belongs to a group (groupid) not in /etc/passwd. |
| -nouser | True if the file belongs to a user (userid) not in /etc/passwd. |
| -ok cmd | Same as the -exec option except the command is displayed followed by a question mark. The command cmd is executed only if we respond with a y. |
| -perm on | True if the permissions of the file match the octal number on. Refer to Module 17 on chmod for a full description of the octal number representation used for on . We can prefix on with a minus sign to force the comparison of only the bits that are set in on to the file permission. |
| -print | Always true if specified on the command line. Causes the current pathname to be displayed. |
| -size n[c] | True if the file contains n blocks. If the n is followed by c then n is counted in characters instead of blocks. |
| -type c | True if the file type is type c , where c is one of the following. |

The find command checks the specified options, going from left to right, once for each file or directory encountered. The simplest invocation of find can be used to create a list of all files and directories below the current directory:

**$ find . –print**

We can use regular expressions to select files, for example those that have a .html suffix):

**$ find . -name "\*.html: -print**

We can search for files more recent than, older than, or exactly the same age as a specified date,

* -n - more recent then n days old
* +n - older then n days old
* n exactly of age n

Here are some useful examples. To find html files that have been modified in the last seven days, we can use –mtime with the argument -7 (include the hyphen):

**$ find . -mtime -7 -name "\*.html" -print**

If we just use the number 7 (without a hyphen), we will match only html files that were modified exactly seven days ago:

**$ find . -mtime 7 -name "\*.html" -print**

19. We can specify more than one directory as a starting point for the search. To look across the /bin and /usr directory trees for filenames that contain the pattern \.htm, we can use the following command:

**$ find /usr /bin -name "\*\.htm\*" -print**

20. To find a list of the directories use the -type specifier. Here's one example:

**find . -type d -print**

The most typical options for -type are as following:

* d –Directory
* f – File
* l – Link

**1.1 Using -exec option with find**

Find is able to execute one or more commands for each file it has found with the -exec option. Unfortunately, one cannot simply enter the command. We need to remember two tricks:

The command that we want to execute need to contain a special (obscure) argument {}, which will be replaced by the matched filename, and \; (or ';' ) at the end of the command. (If the \ is left out, the shell will interpret the; as the end of the find command.) If {} id is the last item in the command then it should be a space between the {} and the \;,

for example:

**$ find . -type d -exec ls -ld {} \;**

Here are several "global" chmod tricks based on fine -exec capabilities:

**$ find . -type f -exec chmod 500 {} ';'**

**$ find . -name "rc.conf" -exec chmod o+r '{}' ';'**

This command will search in the current directory and all sub directories for a file named rc.conf.

Note: The -print option will print out the path of any file that is found with that name. In general -print will print out the path of any file that meets the find criteria.

**$ find . -name core -ctime +4 -exec /bin/rm -f {} \;**

There's no output from this command because we didn't use the - print at the end of the command. What it does is find all files called "core" that have a creation time that's more than 4 days ago and remove them.

The find command is a powerful command in UNIX. It helps us find files by owner, type, filename, and other attributes. The most awkward part of the command is the required elements of the -exec option, and that's where the xargs command helps immensely.

**2. Mail**

The mail command is a quick and easy way to send an email to someone. Just type mail and the address of the person we want to mail. W e will then be prompted for a subject and any cc's. Then just type our message and control-d on a line by itself to send the mail

**2.1 Mail Headers**

Mail headers have the following construction

**$mail <username>**

**Subject: Title describing the message (optional)**

**Cc: List of people to receive a carbon copy (optional)**

**Bcc: List of people to receive blind carbon copy (they do not see user names in the received message. Optional)**

**2.2 Mailboxes**

UNIX uses two mailboxes to hold mail messages

**system mailbox (/vsr/spool/mail/)**

**user mail box (..../.../mbox)**

Mail arrives in the system mailbox, and is saved in our user mail box after we have read it. The user mail box is normally located in their $HOME directory.

To list all mails which he got say command only mail without any arguments Type

**$mail**

The mail program displays a title message and lists all available mail headers,

**SCO System V Mail (version 3.2) Type ? for help.**

**"/usr/spool/mail/brianb": 3 messages 3 new**

**N 3 brianb Mon May 31 15:02 10/299 My message3**

**N 2 brianb Mon May 31 15:01 9/278**

**>N 1 brianb Mon May 31 15:00 12/415 My first message**

**&**

This initial screen displays the subject fields of messages which have arrived. The format of the display is,

**Type Message\_number From\_User Date/Time Subject**

**N denotes a new message**

**> denotes the current message**

**& mail prompt symbol**

Note how message number 2 does not have a subject heading. This is because the mail message was sent from a file, and the -s option was not specified on the command line when the mail program was invoked. To read message type number associated to that mail. To quit from this mail prompt say q.

**2.3 Sending Mail to People at Other Host Machines or Sites**

We send mail to other people at remote sites by specifying their specific mail address.

**Examples**

**$ mail your\_user\_name@server.com**

**3. write**

The write command is used to send on-line messages to another user on the same machine. The format of the write command is as follows:

**$ write username**

**text of message**

**^D**

After typing the command, we enter our message, starting on the next line, terminating with the end-of-file character. The recipient will then hear a bleep, then receive our message on screen, with a short header attached. The following is a typical exchange. User UserRavi types:

**$ write UserRavi**

**Hi there - want to go to lunch?**

**^D**

**$**

User lnp8zz will hear a beep and the following will appear on his/her screen:

**Message from UserRavi on sun050 at 12:42**

**Hi there - want to go to lunch?**

**EOF**

If UserAjay wasn't logged on, the sender would see the following:

**$ write UserAjay**

**UserAjay not logged in.**

**4. talk**

The command talk creates a two-way, screen-oriented communication program. It allows users to type simultaneously, with their output displayed in separate regions of the screen. To send a talk request, type talk and then <user name>@<their machine>. So, for example, to send a talk request to dmb, who is logged on to cslab0a, type talk raju@company. The recipient of the request is then prompted for a response.

**Example:**

**$ talk <username>**

**$ talk username@<their machinename> [to send a message to remote user]**

Note: The mesg command may, of course, be used to disable write,talk access to a terminal.

We can stop messages being flashed up on our screen if we wish. To turn off direct communications type:

**% mesg n**

It will remain off for the remainder of our session, unless we type:

**% mesg y**

To turn the facility back on. Typing just mesg lets we know whether it is on or off.

**5. walls**

**wall** is a Unix command line utility. That would only invoked by Administrator, it displays the contents of a file or standard input to all logged-in users.

**Invocation**

wall is invoked as follows:

**$ raju@company:# wall <filename>**

To take standard input as the file, omit the filename.

**Output:**

When invoked, wall displays the following output:

**Broadcast Message from raju@wilshiresoft**

**(/dev/pts/0) at 01:23 ...**

**6. Pine (e-mail client)**

Pine is a powerful freeware e-mail client: the University of Washington's " Program for Internet News & Email." Many people believe that Pine stood for "Pine is not Elm." However, its original author, Laurence Lundblade

**$pine <Enter>**

**Exercises:**

1. Send a message to another user on our Unix system, and get them to reply.

2. Create a small text file and send it to another user.

3. When we receive a message, save it to a file other than our mailbox. (Remember we can always send

yourself a message if we don't have one.)

4. Send a message to a user on a different computer system.

5. Send a note to our course tutor telling him that we can use mail now.

**Chapter 10**

**Advanced Filters**

**1. grep**

The grep program searches a file or files for lines that have a certain pattern. The syntax is:

**$ grep "pattern" file(s)**

The name "grep" derives from the ed (a Unix line editor) command g/re/p, which means "globally search for a regular expression and print all lines containing it." A regular expression is either some plain text (a word, for example) and/or special characters used for pattern matching. When we learn more about regular expressions, we can use them to specify complex patterns of text.

The simplest use of grep is to look for a pattern consisting of a single word. It can be used in a pipe so that only those lines of the input files containing a given string are sent to the standard output. But let's start with an example reading from files: searching all files in the working directory for a word--say, Unix. We'll use the wildcard \* to quickly give grep all filenames in the directory.

**$ grep "Unix" \***

**ch01:Unix is a flexible and powerful operating system**

**ch01:When the Unix designers started work, little did**

**ch05:What can we do with Unix?**

**$**

When grep searches multiple files, it shows the filename where it finds each matching line of text. Alternatively, if we don't give grep a filename to read, it reads its standard input; that's the way all filter programs work:

**$ ls -l | grep "Aug"**

**-rw-rw-rw- 1 john doc 11008 Aug 6 14:10 ch02**

**-rw-rw-rw- 1 john doc 8515 Aug 6 15:30 ch07**

**-rw-rw-r-- 1 john doc 2488 Aug 15 10:51 intro**

**-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros**

**$**

First, the example runs ls -l to list our directory. The standard output of ls -l is piped to grep, which only outputs lines that contain the string Aug (that is, files that were last modified in August). Because the standard output of grep isn't redirected, those lines go to the terminal screen.

grep options let we modify the search. Given table lists some of the options.

|  |  |
| --- | --- |
| **Option** | **Description** |
| -v | Print all lines that do not match pattern. |
| -n | Print the matched line and its line number. |
| -l | Print only the names of files with matching lines (lowercase letter "L"). |
| -c | Print only the count of matching lines. |
| -i | Match either upper- or lowercase. |

**Some grep options**

Next, let's use a regular expression that tells grep to find lines with carol, followed by zero or more other characters (abbreviated in a regular expression as ".\*"), then followed by Aug:

[15] Note that the regular expression for "zero or more characters," ".\*", is different than the corresponding filename wildcard "\*". W e can't cover regular expressions in enough depth here to explain the difference--though more-detailed books do. As a rule of thumb, remember that the first argument to grep is a regular expression; other arguments, if any, are filenames that can use wildcards.

**$ ls -l | grep "carol.\*Aug"**

**-rw-rw-r-- 1 carol doc 1605 Aug 23 07:35 macros**

**$**

**2. sed**

sed (which stands for Stream EDitor) is a simple but powerful computer program used to apply various pre-specified textual transformations to a sequential stream of text data. It reads input files line by line, edits each line according to rules specified in its simple language (the sed script), and then outputs the line. While originally created as a Unix utility by Lee E. McMahon of Bell Labs from 1973 to 1974, sed is now available for virtually every operating system that supports a command line.

**2.1 Functions**

sed is often thought of as a non-interactive text editor. It differs from conventional text editors in that the processing of the two inputs is inverted. Instead of iterating once through a list of edit commands applying each one to the whole text file in memory, sed iterates once through the text file applying the whole list of edit commands to each line. Because only one line at a time is in memory, sed can process text files with an arbitrarily-large number of lines. Some implementations of sed can only process lines of limited lengths.

sed's command set is modeled after the ed editor, and most commands work similarly in this inverted paradigm. For example, the command 25d means if this is line 25, then delete (don't output) it, rather than go to line 25 and delete it as it does in ed. The notable exceptions are the copy and move commands, which span a range of lines and thus don't have straightforward equivalents in sed. Instead, sed introduces an extra buffer called the hold space, and additional commands to manipulate it. The ed command to copy line 25 to line 76 (25t76) for example would be coded as two separate commands in sed (25h; 76g), to store the line in the hold space until the point at which it should be retrieved.

**Usage:**

The following example shows a typical usage of sed, where the -e option indicates that the sed expression follows:

**$ sed -e 's/oldstuff/newstuff/g' inputFileName > outputFileName**

The s stands for substitute; the g stands for global, which means that all matching occurrences in the line would be replaced. After the first slash is the regular expression to search for and after the second slash is the expression to replace it with. The substitute command (s///) is by far the most powerful and most commonly used sed command.

Under Unix, sed is often used as a filter in a pipeline:

**$ generate\_data | sed -e 's/x/y/'**

That is, generate the data, but make the small change of replacing x with y.

Several substitutions or other commands can be put together in a file called, for example, subst. sed and then be applied using the -f option to read the commands from the file:

**$ sed -f subst.sed inputFileName > outputFileName**

Besides substitution, other forms of simple processing are possible. For example, the following deletes empty lines or lines that only contain spaces:

**$ sed -e '/^ \*$/d' inputFileName**

This example used some of the following regular expression meta characters:

* The caret (^) matches the beginning of the line.
* The dollar sign ($) matches the end of the line.
* A period (.) matches any single character.
* The asterisk (\*) matches zero or more occurrences of the previous character.
* A bracketed expression delimited by [ and ] matches any of the characters inside the brackets.

Complex sed constructs are possible, to the extent that it can be conceived of as a highly specialised, albeit simple, programming language. Flow of control, for example, can be managed by use of a label (a colon followed by a string which is to be the label name) and the branch instruction b; an instruction b followed by a valid label name will move processing to the block following the label; if the label does not exist then the branch will end the script.

A third one should be added to remove all blanks and tabs immediately before the end of line:

**$ sed -e 's/#.\*//' -e 's/[ ^I]\*$//' -e '/^$/ d'**

The character "^I" is a CRTL-I or tab character. We would have to explicitly type in the tab. Note the order of operations above, which is in that order for a very good reason. Comments might start in the middle of a line, with white space characters before them. Therefore comments are first removed from a line, potentially leaving white space characters that were before the comment. The second command removes all trailing blanks, so that lines that are now blank are converted to empty lines. The last command deletes empty lines. Together, the three commands remove all lines containing only comments, tabs or spaces.

This demonstrates the pattern space sed uses to operate on a line. The actual operation sed uses is:

* Copy the input line into the pattern space.
* Apply the first sed command on the pattern space, if the address restriction is true.
* Repeat with the next sed expression, again

operating on the pattern space.

* When the last operation is performed, write out the pattern space and read in the next line from the input file.

|  |  |
| --- | --- |
| n | replace nth instance of pattern with replacement |
| g | replace all instances of pattern with replacement |
| p | write pattern space to STDOUT if a successful substitution takes place |
| w file | Write the pattern space to file if a successful substitution takes place |

**2.2 Printing with p**

Another useful command is the print command: "p." If sed wasn't started with an "-n" option, the "p" command will duplicate the input. The command

**$ sed 'p'**

will duplicate every line. If we wanted to double every empty line, use:

**$ sed '/^$/ p'**

Adding the "-n" option turns off printing unless we request it. Another way of duplicating head's functionality is to print only the lines we want. This example prints the first 10 lines:

**$ sed -n '1,10 p' <file**

**2.3 Deleting with D**

**$ sed ‘1,10 d’ < file**

**2.4 Substituting with S**

Sed has several commands, but most people only learn the substitute command: s. The substitute command changes all occurrences of the regular expression into a new value. A simple example is changing "day" to"night:"

**$ sed s/day/night/ <old >new**

I didn't put quotes around the argument because this example didn't need them. If we read my earlier tutorial, we would understand why it doesn't need quotes. If we have meta-characters in the command, quotes are necessary. In any case, quoting is a good habit, and I will henceforth quote future examples. That is:

**$ sed 's/day/night/' <filename>**

There are four parts to this substitute command:

|  |  |
| --- | --- |
| s | Substitute command |
| /../../ | Delimiter |
| day | Regular Expression Pattern String |
| night | Replacement string |

We've covered quoting and regular expression. That's 90% of the effort needed to learn the substitute command. To put it another way, we already know how to handle 90% of the most frequent uses of sed. There are a few fine points that must be covered.

**3. awk**

The name awk comes from the initials of its designers: Alfred V. Aho, Peter J. Weinberger, and Brian W. Kernighan. The original version of awk was written in 1977 at AT&T Bell Laboratories. In 1985 a new version made the programming language more powerful, introducing user-defined functions, multiple input streams, and computed regular expressions. This new version became generally available with Unix System V Release 3.1.

.awk is a programming language designed to search for, match patterns, and perform actions on files. awk programs are generally quite small, and are interpreted. This makes it a good language for prototyping.

**input lines to awk:**

When awk scans an input line, it breaks it down into a number of fields. Fields are separated by a space or tab character. Fields are numbered beginning at one, and the dollar symbol ($) is used to represent a field.

For instance, the following line in a file

**I like money.**

Has three fields. They are

**$1 I**

**$2 like**

**$3 money.**

Field zero ($0) refers to the entire line. awk scans lines from a file(s) or standard input.

Note: The most frustrating thing about trying to learn awk is getting our program past the shell's parser. The proper way is to use single quotes around the program, like so:

**$ awk '{print $0}' filename**

The single quotes protect almost everything from the shell. In csh or tcsh, we still have to watch out for exclamation marks, but other than that, we're safe.

**$ awk '{print $2,$1}' filename**

Will print the second field, then the first. All other fields are ignored

* Variables
  + Need not be declared
  + May contain any type of data, their data type may change over the life of the program
  + As in C, case matters; since all the built-in variables are all uppercase, avoid this form.
  + Some of the commonly used built-in variables are:
    - NR-- The current line's sequential number
    - NF-- The number of fields in the current line
    - FS-- The input field separator; defaults to whitespace and is reset by the -F command line parameter
    - OFS-- Output Field Separator default ' '
  + RS-- Input Record Separator default \n
  + FILENAME-- The name of the file currently being processed
* Fields
  + Each record is separated into fields named $1, $2 , etc
  + $0 is the entire record
  + NF contains the number of fields in the current line
* Print
  + print prints each of the values of $1 through $NF separated by OFS then prints a \n onto stdout; the default value of OFS is a blank

**Example:**

See a file(sample) contains these contents.

ramu:8:17:d

RAMU:8:17:D

king:89:37:j

smith:8:17:c

scott:19:4:d

allen:73:99:f

**$ awk –F “:” ‘NR==1{print $1,$4}’ sample**

**ramu d**

**$ awk –F “:” ‘NR==1 || NR==2 {print $0}’ sample**

**ramu 8 17 d**

**RAMU 8 17 D**

BEGIN and END are special patterns. They are not used to match input records. Rather, they are used for supplying start-up or clean-up information to our awk script. A BEGIN rule is executed, once, before the first input record has been read. An END rule is executed, once, after all the input has been read. For example:

**Example1:**

Program to count number of records and Number of fields

**awk 'BEGIN{FS=":";OFS="#";print "Student info"}**

**{print $1,$3}**

**END{print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}**

**END{print "The Number Of records are" NR}**

**END{print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}**

**END{print "The Number Of Fields are" NF}**

**END{print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}' sample**

**Result:**

**Student info**

**ramu#17**

**RAMU#17**

**king#37**

**smith#17**

**scott#4**

**allen#99**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**The Number Of records are6**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**The Number Of Fields are4**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Example2:**

Program to find out idle users

**who -i|tr -s " "|awk -F " " '$6 !~/\./{print $1,$2,$6}' >smith**

**clear**

**awk 'BEGIN{FS=" ";OFS="\t";print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}**

**BEGIN{print "IDLE USERS ARE";print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";print "username**

**TTy IDLETIME"}**

**{print $1,$2,$3}**

**END{print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}' smith**

**Result:**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**IDLE USERS ARE**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**username TTy IDLETIME**

**uf027 pts/2 01:48**

**uf088 ttyW1 00:02**

**srujana ttyW1 00:02**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Example3:**

Program to count characters in a given file

**clear**

**echo "Enter the file name"**

**read name**

**awk 'BEGIN{print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*";print "The length of a given**

**file is";print "\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*"}**

**{len=length($0);x=len+x}END{print "the length of is"x}' $name**

**Note:**

Write these programs in a file and then execute them with sh <file name> to get the desired results.

**String functions:**

These are additional functions we can use with awk

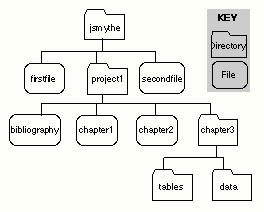
|  |  |
| --- | --- |
| sub(regexp,sub) | Substitute sub for regexp in $0 |
| sub(regexp,sub,var) | Substitute sub for regexp in var |
| gsub(regexp,sub) | Globally substitute sub for regexp in $0 |
| gsub(regexp,sub,var) | Globally substitute sub for regexp in var |
| split(var,arr) | Split var on white space into arr |
| split(var,arr,sep) | Split var on white space into arr on sep as separator |
| index(bigvar,smallvar) | Find index of smallvar in bigvar |
| match(bigvar,expr) | Find index for regexp in bigvar |
| length(var) | Number of characters in var |
| substr(var,num) | Extract chars from position num to end |
| substr(var,num1,num2) | Extract chars from num1 through num2 |
| sprintf(format,vars) | Format vars to a string |
| getline | reads in a line each time it is called |

**Chapter 11**

**Linking Files and Background/Foreground Processes**

**1. Linking files**

The ln command (short for link) lets we give multiple names to a single file. This is useful when we want to get at a file quickly from within different directories. Assume that our directory structure looks like this:



We may need to refer frequently to a file called table1 in the tables subdirectory when we are in our home directory (jsmythe). Rather than typing the whole pathname

**project1/chapter3/tables/table1**

we could link the file to our home directory. If we are in the home directory, the command to do this is:

**$ ln project1/chapter3/tables/table1 mytable1**

To create the link when we are in the tables directory, the command would have been:

**$ ln table1 ~/mytable1**

After issuing either of these commands, an ls command in our home directory would show an entry for mytable1. The long format of the same command would show 2 links for the file mytable1:

**$ ls -l**

**-rw------- 2 jsmythe 6 Jul 4 14:23 mytable1**

A long format listing of the file table1 would also show 2 links. What if a file called mytable1 had already existed in our home directory? Unix would let we know that a file by that name exists, and would not make the link.

The effect of linking is that the file now has two names. We may call up the file by either name. Creating a link does not change the ownership, group, or permissions for a file. The inode number of two files are same.

**1.1 Removing links**

Links are removed using the rm command. To continue the example above, the command

**$ rm mytable1**

Removes one link to the file table1 by removing the file mytable1. The file table1 itself, and its contents, still exists. Only when all the links to the file have been removed will the file itself be erased.

**1.2 Symbolic links**

The links described in the sections above are "hard" links. In effect, making a hard link creates a standard directory entry just like the one made when the file was created. Hard links have certain limitations. Hard links cannot be made to a directory, only to files, and hard links cannot be made across file systems and disk partitions.

There is another kind of link, called a symbolic link. Symbolic links can span file systems, and can be made for directories.

In the figure above, assume that we want to make a symbolic link from our home directory (jsmythe) to the directory called chapter3. To create the symbolic link, we would move to our home directory and give the command:

**$ ln -s project1/chapter3 linkdir**

The -s option means that a symbolic link is created (instead of the default hard link). This command creates a symbolic link called linkdir which points to the directory called chapter3.

When we list the contents of linkdir

**$ ls linkdir**

**$ data tables**

we see a listing of the contents (data and tables)

We can learn more about the new directory by using the long format and directory options with the ls command:

**$ ls -ld linkdir**

**l--------- 1 staff 7 Jun 11 13:27 linkdir -> project1/chapter3**

Symbolic links are shown with an arrow (->) in the name column at the right.

**1.3 Removing symbolic links**

Use rm to remove a symbolic link, for example

**$ rm linkdir**

This removes the link only, not the file or directory it was pointing to. If the file or directory is removed but the link remains, the link will no longer work.

**2. Unix Processes**

A process is an instance of running a program. If, for example, three people are running the same program simultaneously, there are three processes there, not just one. In fact, we might have more than one process running even with only person executing the program, because (we will see later) the program can ``split into two,'' making two processes out of one.

Keep in mind that all Unix commands, e.g. cc and mail, are programs, and thus contribute processes to the system when they are running. If 10 users are running mail right now, that will be 10 processes. At any given time, a typical Unix system will have many active processes, some of which were set up when the machine was first powered up.

Every time we issue a command, Unix starts a new process, and suspends the current process (the C-shell) until the new process completes (except in the case of background processes, to be discussed later).

Unix identifies every process by a Process Identification Number (pid) which is assigned when the process is initiated. When we want to perform an operation on a process, we usually refer to it by its pid.

Unix is a timesharing system, which means that the processes take turns running. Each turn is a called a timeslice; on most systems this is set at much less than one second. The reason this turns-taking approach is used is fairness: We don't want a 2-second job to have to wait for a 5-hour job to finish, which is what would happen if a job had the CPU to itself until it completed.1

**2.1 Determining Information about Current Processes**

The `ps -x' command will list all our currently-running jobs. An example is:

**$ PID TT STAT TIME COMMAND**

**6799 co IW 0:01 -csh[rich] (csh)**

**6823 co IW 0:00 /bin/sh /usr/bin/X11/startx**

**6829 co IW 0:00 xinit /usr/lib/X11/xinit/xinitrc --**

**6830 co S 0:12 X :0**

**6836 co I 0:01 twm**

**6837 co I 0:01 xclock -geometry 50x50-1+1**

**6841 p0 I 0:01 -sh[rich on xterm] (csh)**

**6840 p1 I 0:01 -sh[rich on xterm] (csh)**

**6847 p2 R 0:00 ps -x**

The meaning of the column titles is as follows:

|  |  |
| --- | --- |
| PID | process identification number |
| TT | controlling terminal of the process |
| STAT | state of the job |
| TIME | amount of CPU time the process has acquired so far |
| COMMAND | name of the command that issued the process |

The TT information gives terminal names, which we can see by typing the who command. E.g. we see p2 in the TT column above, which is the terminal listed as ttyp2 in the who command.

The state of the job is given by a sequence of four letters, for example, `RWNA'. The first of these four is typically one of the following:

first letter runnability of the process R runnable process T stopped process S process sleeping for less than about 20 seconds we processes that are idle (sleeping longer than about 20 seconds) A state-R process is runnable, i.e. it is be able to make use of a turn given to it, and is waiting for one. We can put a process in state T, i.e. stop the process, by typing control-z. Suppose, for example, that we are using ftp to get some files from some archive site, and we notice a file there called README. we can use the ftp `get' command to get the README file, and then type C-z. This will stop (suspend) the ftp process, and get me back to the C-shell. At that point we can read the README file, say using more, and then reactivate the ftp process, by typing `fg' to the shell.

A typical example of an S/I process is one that is waiting for user input. If we are using the emacs editor, for example, the process will go to state S when it is waiting for me to type something; if we take more than 20 seconds to decide what to type, the process will be in state I.

**2.2 Foreground/Background Processes**

Suppose we want to execute a command but do not want to wait for its completion, i.e. we want to be able to issue other commands in the mean time. We can do this by specifying that the command be executed in the background.

There are two ways to do this. The first is to specify that it be a background process when we submit it, which we can do by appending an ampersand (`&') to the end of the command. For example, suppose we have a very large program, which will take a long time to compile. We could give the command

**$ cc bigprog.c &**

Which will execute the compilation while allowing me to submit other commands for execution while the compile is running. The C-shell will let me know what the pid is for this background process (so that we can later track it using ps, or kill it), but will also give me my regular prompt, inviting me to submit new commands while the other one is running.

But what about the compiler error messages? We hope we don't have any :-) but if we do have some, we don't want them to be interspersed with the output of other commands we are running while the compile is executing. To avoid this, we redirect the error messages:

**$ cc bigprog.c >& errorlist &**

All error messages will now be sent to the file `errorlist', which we can view later.

Another good example is when we start a window during a X session. We would like to start the window from an existing window, but we still want to be able to use the original window. W e execute the command

**$ xterm &**

This will start a new window, and allow us to keep using the current window.

The other way to put a job in the background is to stop it, using C-z as described earlier, and then use another command, bg, to move the process to the background.

For example, suppose we started our long-running compile,

**$ cc bigprog.c**

but we forget to append the ampersand. We can type control-z to suspend/stop the job, and then type `bg' to resume the job in the background, allowing us to submit other commands while the compilation takes place. Unix will tell us when the background job has completed, with a statement like

**[1] Done cc bigprog.c**

By the way, if we log out, whatever background processes we have running at the time will not be killed; they will continue to run.

**2.3 Terminating a Process**

We can terminate a process by using the kill command. We simply find its pid (say by using ps), and then type

**$ kill -9 pid**

If the process is in the foreground, though, the easiest way to kill it is to simply type control-C.

**Notes:**

* The basic mechanism for setting up the turns is as follows. The machine will have a piece of hardware which sends electrical signals to the CPU at periodic intervals. These signals force the CPU to stop the program it is running, and jump to another program, which will be the operating system program (OS). The OS can then determine whether the current program's timeslice is finished, and if so, then give a turn to another program, by jumping to that program. Note the interaction of hardware (the electrical signals, and the CPU's reaction to them) and software (the OS) here.
* Keep in mind, though, that there is ``no free lunch'' here. The more processes on the machine, the longer it is between turns for each process, so overall response time goes down.
* Though a much better solution to the problem is to use emacs, since the error messages will automatically be placed into a special buffer.

**2.4 Processes and Jobs**

A process is an executing program identified by a unique PID (process identifier). To see information about wer processes, with their associated PID and status, type

**$ ps**

A process may be in the foreground, in the background, or be suspended. In general the shell does not return the UNIX prompt until the current process has finished executing. Some processes take a long time to run and hold up the terminal. Backgrounding a long process has the effect that the UNIX prompt is returned immediately, and other tasks can be carried out while the original process continues executing.

**2.4.1 Running background processes**

To background a process, type an & at the end of the command line. For example, the command sleep waits a given number of seconds before continuing. Type

**$ sleep 10**

This will wait 10 seconds before returning the command prompt %. Until the command prompt is returned, we can do nothing except wait.

To run sleep in the background, type

**$ sleep 10 &**

**[1] 6259**

The & runs the job in the background and returns the prompt straight away, allowing we do run other programs while waiting for that one to finish. The first line in the above example is typed in by the user; the next line, indicating job number and PID, is returned by the machine. The user is be notified of a job number (numbered from 1) enclosed in square brackets, together with a PID and is notified when a background process is finished. Backgrounding is useful for jobs which will take a long time to complete.

**2.4.2 Backgrounding a current foreground process**

At the prompt, type

**$ sleep 100**

We can suspend the process running in the foreground by holding down the [control] key and typing [z] (written as ^Z) Then to put it in the background, type

**$ bg**

Note: do not background programs that require user interaction e.g. pine

**2.4.3 Listing suspended and background processes**

When a process is running, backgrounded or suspended, it will be entered onto a list along with a job number. To examine this list, type

**$ jobs**

An example of a job list could be

**[1] Suspended sleep 100**

**[2] Running netscape**

**[3] Running nedit**

To restart (foreground) a suspended processes, type

**$ fg %jobnumber**

For example, to restart sleep 100, type

**$ fg %1**

Typing fg with no job number foregrounds the last suspended process.

**3. Killing a process**

kill (terminate or signal a process), It is sometimes necessary to kill a process (for example, when an executing program is in an infinite loop) To kill a job running in the foreground, type ^C (control c). For example, run

**$ sleep 100**

**^C**

To kill a suspended or background process, type

**$ kill %jobnumber**

For example, run

**$ sleep 100 &**

**$ jobs**

If it is job number 4, type

**$ kill %4**

To check whether this has worked, examine the job list again to see if the process has been removed.

**3.1 ps (process status)**

Alternatively, processes can be killed by finding their process numbers (PIDs) and using kill PID\_number

**$ sleep 100 &**

**$ ps**

**PID TT S TIME COMMAND**

**20077 pts/5 S 0:05 sleep 100**

**21563 pts/5 T 0:00 netscape**

To kill off the process sleep 100, type

**$ kill 20077**

and then type ps again to see if it has been removed from the list.

If a process refuses to be killed, uses the -9 option, i.e. type

**$ kill -9 20077**

Note: It is not possible to kill off other users' processes !!!

**Chapter 12**

**Shell Programming**

**1. Why shell programming?**

Even though there are various graphical interfaces available for Linux the shell still is a very neat tool. The shell is not just a collection of commands but a really good programming language. We can automate a lot of tasks with it, the shell is very good for system administration tasks, we can very quickly try out if our ideas work which makes it very useful for simple prototyping and it is very useful for small utilities that perform some relatively simple tasks where efficiency is less important than ease of configuration, maintenance and portability.

So let's see now how it works

**1.1 Creating a script**

There are a lot of different shells available for Linux but usually the bash (bourne again shell) is used for shell programming as it is available for free and is easy to use. So all the scripts we will write in this article use the bash (but will most of the time also run with its older sister, the bourne shell).

For writing our shell programs we use any kind of text editor, e.g. nedit, kedit, emacs, vi...as with other programming languages. The program must start with the following line (it must be the first line in the file)

**$ !/bin/sh**

The #! Characters tell the system that the first argument that follows on the line is the program to be used to execute this file. In this case /bin/sh is shell we use.

When we have written our script and saved it we have to make it executable to be able to use it. To make a script executable type

**$ chmod +x filename**

Then we can start our script by typing: ./filename

**1.2 Comments**

Comments in shell programming start with # and go until the end of the line. We really recommend to use comments. If we have comments and we don't use a certain script for some time we will still know immediately what it is doing and how it works.

**1.3 Variables**

As in other programming languages we can't live without variables. In shell programming all variables have the datatype string and we do not need to declare them. To assign a value to a variable we write:

**varname=value**

To get the value back we just put a dollar sign in front of the variable:

**$!/bin/sh**

Assign a value:

**$ a="hello world"**

Now print the content of "a":

**echo "A is:"**

**echo $a**

Type this lines into our text editor and save it e.g. as first. Then make the script executable by typing **chmod +x first** in the shell and then start it by typing ./first

The script will just print:

**A is:**

**hello world**

Sometimes it is possible to confuse variable names with the rest of the text:

**$num=2**

**echo "this is the $numnd"**

This will not print "this is the 2nd" but "this is the " because the shell searches for a variable called numnd which has no value. To tell the shell that we mean the variable num we have to use curly braces:

**$ num=2**

**echo "this is the ${num}nd"**

This prints what we want: this is the 2nd

There are a number of variables that are always automatically set. We will discuss them further down when we use them the first time.

If we need to handle mathematical expressions then we need to use programs such as expr. Besides the normal shell variables that are only valid within the shell program there are also environment variables. A variable preceded by the keyword export is an environment variable. We will not talk about them here any further since they are normally only used in login scripts.

**1.4. Exporting Variables**

By default, variables defined within a shell are local to it. Any process/command executed by that shell will not get access to these variables, unless they are exported to the environment.

**$ export VARNAME**

This export includes changes to existing variables that are in the path. for example, if we wish to modify the PATH environment variable we would:

**$ PATH=$PATH: additional\_directory**

**$ export PATH**

**1.5. Positional (Parameter) Variables**

When the shell script is run, any command line parameters to the shell are provided by a set of shell variables $0 $1 $2 $3 $4 $5 $6 $7 $8 and $9.

$0 (or ${0}) contains the name of the shell script itself, and depending on how the shell was called, may contain the full path to the command. If there are more than 9 parameters/arguments to the script, then they are accessed using the shift command which shifts all the parameters ($1=$2, $2=$3 etc) with $9 getting the next parameter in the line ($0 is unaffected).

$# is a variable that contains the number of command line parameters.

$\* and $@ can be used to get all the parameters in one go. Very useful for passing the complete command line parameter set to a child process

**Example**

**!/bin/sh**

A wrapper for the GNU C compiler that has the debug flag set

**$ gcc -g $\***

Note:

|  |  |
| --- | --- |
| $\* and $@ | are the same except when placing quotes around them |
| "$\*" | is the same as "$1 $2 $3..." |
| "$@" | is the same as "$1" "$2" "$3"... |
| $? | The exit status (error code) of the last command |
| $$ | The PID of the current command - Useful for creating uniquely named temp files (eg: tmp.$$) |
| $! | The process id of the last command started in the background (with &) |

Example:

Write these lines in a file

**Echo “First argument is $1”**

**Echo “Second argument is $2”**

**Echo “The Total number of arguments are $#”**

**Echo “The list of arguments are $\*”**

**Echo “The pid of the current process is $$”**

**Echo “The name of the file where we written this program is $0”**

**Echo “The exit status of last command $?”**

**Result:**

Execute the above program using some command line arguments like

**$ sh <filename> 1 2 3 4 5**

**1.6 Quoting**

Before passing any arguments to a program the shell tries to expand wildcards and variables. To expand means that the wildcard (e.g. \*) is replaced by the appropriate file names or that a variable is replaced by its value. To change this behaviour we can use quotes: Let's say we have a number of files in the current directory. Two of them are jpg-files, mail.jpg and tux.jpg.

**$ !/bin/sh**

**echo \*.jpg**

This will print "mail.jpg tux.jpg".

Quotes (single and double) will prevent this wildcard expansion:

**$ !/bin/sh**

**echo "\*.jpg"**

**echo '\*.jpg'**

This will print "\*.jpg" twice.

Single quotes are most strict. They prevent even variable expansion. Double quotes prevent wildcard expansion but allow variable expansion:

**$ !/bin/sh**

**echo $SHELL**

**echo "$SHELL"**

**echo '$SHELL'**

This will print:

**/bin/bash**

**/bin/bash**

**$SHELL**

Finally there is the possibility to take the special meaning of any single character away by preceeding it with a backslash:

**echo \\*.jpg**

**echo \$SHELL**

This will print:

**\*.jpg**

**$SHELL**

**2. shift: Shifts Parameters**

When a large number of parameters (;SPMgt; 9) are passed to the shell, shift can be used to read those parameters. If the number of parameters to be read is known, say three, a program similar to the following could be written:

**$!/bin/sh**

**echo The first parameter is $1.**

**shift**

**echo The second parameter is $1.**

**shift**

**echo The third parameter is $1.**

**exit 0**

**3. read: Reading Input from User**

The following short example shows how read can be used to get input from the user:

**#!/bin/sh**

**echo -e "Please enter your name: \c"**

**read NAME**

**echo "Your name is $NAME."**

**exit 0**

The \c means that the line feed will be suppressed, so that the prompt sits at the end of the line, not at the beginning of the following line.

Two more common controls available to the ***echo*** command are to use \n to add a line feed, and \t to add a tab. Multiple values may be read on a single line by using:

**$!/bin/sh**

**echo -e "Please enter two numbers: \c"**

**read NUM1 NUM2**

**echo The numbers entered are $NUM1 and $NUM2**

**exit 0**

This ensures that if two numbers are entered on a single line, they will be read within two variables. If three numbers were entered, the second variable (NUM2) would contain the last two numbers.

Assuming three numbers were the input of the above example, the first two numbers could be assigned to the first variable by entering them as

**num1 \num2 num3**

The backslash ( \) allows the blank space between num1 and num2 to be part of the variable (ordinarily, spaces are used as **field seperators.**

**4. Conditional Statements**

**4.1. if Statements**

The core condi3tional concept of wrapping a block of statements that is only to be processed ***if*** some condition is met.

Shells also support ***else*** and the combined ***elif*** else-if condition, 3 basic layouts for if statements are shown below. Note the use of the then and keyword to separate the condition commands from internal command block, and the ***fi*** keyword to mark the end of the block.

**First Model:**

**if condition-command**

**then**

**command1**

**command2**

**...**

**fi**

**Second Model:**

**if condition-command**

**then**

**commandA1**

**commandA2**

**...**

**else**

**commandB1**

**commandB2**

**...**

**fi**

**Third model:**

**if condition-command-A**

**then**

**commandA1**

**commandA2**

**...**

**elif condition-command-B**

**then**

**commandB1**

**commandB2**

**...**

**else**

**commandC1**

**commandC2**

**...**

**fi**

The commands inside the blocks are used the same as any other command within the system, and it is possible to nest other conditions statements inside those blocks. For conciseness, many people will use the semicolon (;) command separating character to allow the then keyword to be placed on the same line as the if and the condition-command. (Note that no semicolon is needed after the then keyword)

The condition-command is a any command, and the if statement is evaluated based on the success (exit status) of that command.

**Example:**

**if ls -al ; then**

**echo "Directory was successfully listed"**

**else**

**echo "Failed to list directory"**

**fi**

**4.2. The test or [ ] Command**

There are other types of tests beyond running a simple command and checking its exit status. The shells support a number for "test" commands that can be used to perform more useful tests.

The test command has two formats:

**test expression**

**or**

**[ expression ]**

Note: for the [ ] version there must be spaces between the brackets and the expression.

The test command/statement will evaluate the expression and return a status of success if the expression is true. The following tests can be performed:

**4.2.1 File Tests**

|  |  |
| --- | --- |
| -r file | true if file is readable (by current user) |
| -w file | true if file is writable (by current user) |
| -c file | true if file is executable (by current user) |
| -f file | true if file is an ordinary file |
| -d file | true if file is a directory |
| -s file | true if the length of file is > 0 |
| -t | true if standard output is associated with a terminal |
| -t fd | true if file descriptor 'fd' is associated with a terminal |

An example would be where a program needs to output something to a file, but first checks that the file exists:

**#!/bin/sh**

**echo “Enter the filename”**

**read file1**

**if [ ! -s file1 ] or if test ! –s file1**

**then**

**echo "file1 is empty or does not exist."**

**ls -l > file1**

**exit**

**else**

**echo "File file1 already exists."**

**fi**

**exit 0**

**4.2.2 String Tests**

|  |  |
| --- | --- |
| -z str | true if the length of str is zero |
| -n str | true if the length of str is non zero |
| str | true if the str is not empty |
| str1 = str2 | true if str1 and str2 are the same |
| str1 != str2 | true if str1 and str2 are different |

An example would checks that the given strings are same or not:

**#!/bin/sh**

**echo “Enter the strings(string1,string2)”**

**read str1**

**read str2**

**if test str1 = str2**

**then**

**echo "Both Strings are equal"**

**exit**

**else**

**echo "Given strings are not equal"**

**fi**

**exit 0**

**4.2.3 Numeric Tests**

|  |  |
| --- | --- |
| int1 -eq int2 | true if int1 = int2 |
| int1 -ne int2 | true if int1 != int2 |
| int1 -lt int2 | true if int1 < int2 |
| int1 -le int2 | true if int1 <= int2 |
| int1 -gt int2 | true if int1 > int2 |
| int1 -ge int2 | true if int1 >= int2 |

**Example:**

**$!/bin/sh**

**if test $# -le 5**

**then**

**echo Less than or equal to five parameters.**

**else**

**echo More than 5 parameters.**

**fi**

**exit 0**

**4.2.4 Combining Operators**

* ! NOT - Invert the result of the next expression
* -a AND - Only true if both prev and next expr are true
* -o OR - True if either prev or next expr is true
* ( ) parentheses for grouping expressions

All parameters to test must be separated by spaces, including parentheses, operators and strings/integers. Note, that test operates on strings (or strings containing integer numbers) It cannot process commands directly, we must use the back-quotes to perform command substitution.

**if [ "$HOME" = `pwd` ] ; then echo "I am home" ; fi**

Also double quoting of variable is important within a test statement, as an undefined variable will resolve to nothing which will not be correctly processed by the shell.

For example, if $HOME is undefined

**if [ $HOME = `pwd` ] ; then...**

will expand to be:

**if [ = /export/home/me ] ; then...**

Which is invalid.

**4.3. case Statements**

When there are several different sets of operations to be performed based on different values of a single string, it can get quite messy using a long string of if, elif, elif, elif, else statements to perform the operations. The case command allows a convenient structured method of performing flow control through one of multiple blocks using pattern matching of strings.

**$ case string in**

**pattern1 ) commands;;**

**pattern2 ) commands ;;**

**pattern3 ) commands ;;**

**...**

**\*) commands ;; # default case**

**esac**

In addition, multiple pattern options can be specified for a single case using pattern1a | [I pattern1b] style blocks.

When specifying patterns,

* \* matches any string of zero or more characters
* ? matches any single character
* [ ] matches any single character within the set
* [! ] matches any single character not in the set

These can be combined together to form more advanced patterns:

[Yy]\* Matches any string starting with an upper or lower case y.

Use quotes or escape/back-slash the special characters if we wish to pattern match them specifically.

**$!/bin/sh**

# an example with the case statement, which reads a command from the user and processes it

**echo "Enter your command (who, list, or cal)"**

**read command**

**case "$command" in**

**who)**

**echo "Running who..."**

**who**

**;;**

**list)**

**echo "Running ls..."**

**ls**

**;;**

**cal)**

**echo "Running cal..."**

**cal**

**;;**

**\*)**

**echo "Bad command, your choices are: who, list, or cal"**

**;;**

**esac**

**exit 0**

**4.4. while Statements .**

Similar to the basic if statement, except the block of commands is repeatedly executed as long as the condition is met.

**while condition-command**

**do**

**command1**

**command2**

**...**

**done**

As with if statements, a semicolon (;) can be used to remove include the do keyword on the same line as the while condition-command statement.

The example below loops over two statements as long as the variable i is less than or equal to ten. Store the following in a file named while1.sh and execute it

**$!/bin/sh**

Illustrates implementing a counter with a while loop

Notice how we increment the counter with ***expr*** in backquotes

**i="1"**

**while [ $i -le 10 ]**

**do**

**echo "i is $i"**

**i=`expr $i + 1`**

**done**

**4.5 until loop**

Execute statements as long as a condition is false

**until grep "sort" dbase\_log > /dev/null**

**do**

**sleep 10**

**done**

**echo "Database has been sorted"**

Example executes until grep is unsuccessful

**4.6 for Statement**

The ***for*** statement is used to iterate/loop through a predefined list of values, setting a shell variable to the current value each time through the loop. Unlike many other languages shell for loops operate on word lists, not integers with start and stop values.

**for VAR in wordlist**

**do**

**commands**

**...**

**done**

**Example:**

**#!/bin/sh**

**for i in Claude Paul Wayne Roger Tom**

**do**

**echo The name is $i.**

**done**

**exit 0**

Within the shell, parameters are read as $1 for the first parameter, $2 for the second parameter, $3 for the third parameter, and so on. $\* is the entire list of parameters. If the ``in list " is omitted, the list taken is the list of parameters passed to the shell on the command line.

Note: To excite above said example programs we have to compile them in a file and excite that file. Like

**$ sh <filename>**

**5. Debugging**

The simplest debugging help is of course the command echo. We can use it to print specific variables around the place where we suspect the mistake. This is probably what most shell programmers use 80% of the time to track down a mistake. The advantage of a shell script is that it does not require any re-compilation and inserting an "echo" statement is done very quickly.

The shell has a real debug mode as well. If there is a mistake in our script "strangescript" then we can debug it like this:

**$ sh -x strangescript**

This will execute the script and show all the statements that get executed with the variables and wildcards already expanded.

The shell also has a mode to check for syntax errors without actually executing the program. To use this run:

**$ sh -n our\_script**

If this returns nothing then our program is free of syntax errors.

**Chapter 13**

**Administration File System and General System Administration**

**Device Files-**

The definition of a file has been broadened by unix by declaring all the physical devices as a file. It includes tapes, printer, floppy drive, hard disks and terminals.

Any output directed to it will be reflected onto the respective physical associative with the filename. Whenever you are issuing a command to print a file, actually you are directing the file’s output to the file associated with the printer. The kernel takes care of this mapping, these special files to respective devices.

$ls –l /dev

This listing will reveal two vital points:

1. Device files can be grouped into mainly two categories block and character devices depending on the first character of the permission field (b or c).
2. The fifth field–normally representing the size for other files–consist of a pair of numbers (major and minor numbers). A device file contains no data.

**Block and Character devices:**

Floppy drive, cdrom, hard disk comes into this section because to whenever a user performs a write operation firstly it is written into chunks or blocks. And when it performs the read operation, the buffer cache accessed the recently used data. So these devices are using buffer and writing there data into blocks for such devices prefix ‘b’ is used.

The terminal, tape drive and printer are character special devices because the read/write operation ignores the buffer cache and access the device directly.

**Major and Minor No.:**

The set of routines is requires to operate specific devices is known as device drivers. When a device is called the kernel passes some parameters for it to act properly. So the kernel must know not only the type of device but also certain details of it (like density, partition no of it).

The fifth column shows the two no. which are major and minor numbers, major no. shows actually the type of device all hard disk or same kind of hardware will have the same major numbers but the minor will be different.

Minor no. tells the kernel about the parameters to operate the particular device.

**Significance of the Device Names:**

Unix device files is that the same device can often be accessed with several different filename, this is sometimes required to provide backward compatibility and for associating a separate device with specific function.

**The UNIX File System-**

* etc : Contains all system configuration files and the files which maintain information about the user and group.
* Bin : Contains all binary executables
* usr : default directory provided by UNIX OS to create users home directories and contains manual pages
* tmp: System or users temporary files which will be removed when the system reboots.
* dev: Contains all device files i.e logical names to physical devices.
* devices : Contains all the device files. i.e. physical names to physical devices.
* home : default directory allocated for the home directories of normal users when the administrator don’t specify any other directory.
* var : contains all the system log files and message files.
* sbin : Contains all the system administrator executables files.

**Partitions:**

Partition divides hard disk into many logically independent disk and accessed by its own device files. And to use a partition, it is requires to create a file system on it. File system defines a directory structure. Some rules and the advantages of the partitions are:

1. Data is growing day by day but a partition should restrict it spill it over to other partition.
2. If one area is corrupted then other should be shielded for that harmful things which corrupt that area.
3. If the system have adequate number of partitions, each partition can be backed up separately into a single volume of tape. The administrator can have different backup schedules for the different partitions.

**File System:**

In unix is configured with multiple file system where each has its own directory headed by root . but usually we see one root that’s because all have the same file system and becomes single one at a time of use.

Each file system is organized in a sequence of blocks of 1024 bytes and generally has three four components.

1. The boot block – contains a small boot program.
2. The super block – it contains global information about the file system and it also maintains a free list of inodes and data blocks that can be immediately allocated be the kernel when creating a file.
3. The inode block – contains a table for every file system. All attributes of a file or directory stored in this area except the name of the file.
4. The data block – contains the files or programs of the user.

**File System Components**

**Inode Blocks**

every file has an inode – a 128 byte table that contains virtually everything you could possibly need to know about a file. All inodes are stored into a contiguous inode blocks. Each 128 byte inode contains –

1. File type (regular, directory, device etc…)
2. Number of links
3. Numeric user id of the user
4. Numeric group user id of the user.
5. File mode
6. Number of bytes in a file
7. Date and time of the last modification of the file data.
8. Date and time of last access of the file data.
9. Date and time of last change of the inode.
10. The array of 15 pointers to the file.

Whenever a file is opened its inode is copied to the memory inode table and then kept synchronizing periodically with the disk copy.

And 15 pointers are required to get the different-2 information about a file using ls command.

**The Superblock**

It contains global file information about disk usage and availability of data blocks and inodes. So its information should therefore be correct for healthy operation of the system. It mainly contains –

1. The size of the file system.
2. The length of the file systems logical block.
3. Last time of updating.
4. The number of free data blocks available and free partial list of immediately allocated free data blocks.
5. Number of free inodes available and free partial list of immediately usable inodes.
6. The state of the file system.(clean or dirty)

Kernel also maintains a copy of the superblock in memory. It reads and write this copy when controlling allocation of inodes and data block.

So the copy cannot be newer in disk as in memory. And till the superblock is corrupt unix will not boot for this kind of problem many systems have multiple superblocks written in different area of disk.

**Data Blocks**

The smallest data block that can be read or written by the disk controller is 512 bytes. The kernel reads and writes the data using different block size and this block is often referred to as a logical block.

Every block is identifiable by an address- a number that refers to the position of block in data block area. The block containing data is known as direct blocks.

When a file expands the kernel may not find the adjacent block free. So it has to locate the expanded part in some other part of disk. This slows down read/write operation and leads to the disk fragmentation.

Inode keeps track of all the direct block addresses, but inode can store only 12 of them. So the file system also contains indirect blocks. And the inodes keeps track of all the indirect blocks.

**Boot Block**

This is the master boot record (MBR). The boot block contains the partition table and the a small “bootstrapping” program.

When the system is booted the system BIOS looks for the first hard disk and loads the entire segment of the boot block into memory. It then hands over the control to the bootstrapping program. This in turn loads the kernel into memory.

**Mounting and Unmounting File System:**

Before mounting the file system to the we need to check the logical name of it by issuing command

**$ sudo fdisk –l**

**How to mount/unmount Windows partitions (FAT) manually, and allow all users to read/write -**

e.g. Assumed that /dev/hda1 is the location of Windows partition (FAT)

Local mount folder: /media/windows

**\* To mount Windows partition**

**$ sudo mkdir /media/windows**

**$ sudo mount /dev/hda1 /media/windows/ -t vfat -o iocharset=utf8,umask=000**

**\* To unmount Windows partition**

**$ sudo umount /media/windows/**

**How to mount/unmount Windows partitions (FAT) manually, and allow all users to read/write -**

e.g. Assumed that /dev/hda1 is the location of Windows partition (FAT)

Local mount folder: /media/windows

**\* To mount Windows partition**

**$ sudo mkdir /media/windows**

**$ sudo mount /dev/hda1 /media/windows/ -t vfat -o iocharset=utf8,umask=000**

**\* To unmount Windows partition**

**$ sudo umount /media/windows/**

**How to mount/unmount network folders manually, and allow all users to read/write -**

e.g. Assumed that network connections have been configured properly

Network computer's IP: 192.168.0.1

Network computer's Username: myusername

Network computer's Password: mypassword

Shared folder's name: linux

Local mount folder: /media/sharename

**\* To mount network folder**

**$ sudo mkdir /media/sharename**

**$sudo mount //192.168.0.1/linux /media/sharename/ -o username=myusername,password=mypassword,dmask=777,fmask=777.**

**\* To unmount network folder**

**$ sudo umount /media/sharename/**

**To find out the free disk space (df) and disk usage (du) :**

**Disk-free-**

Unix file system as one single tree structure with a single root directory. But there can be multiple disks and each will have its own root and file system.

So to find out the free space in all the disk ‘df’ is the command.

**$ df**

**$ df partition\_name**

**Disk-Usage-**

It will show you all the usages of disk and also can find out the disk usage of specific folder and all the files of the folder.

**$ du – will show the details for all files and directories.**

**$ du /home/oracle/ - will show all the details for all the files and directories in home/oracle.**

**$ du –s /home/oracle – will show summary.**

**Compression of Files using - Compress, gzip and zip:**

To save the space compression is used.

Compress and gzip work with multiple files and remove the original files after it but the zip compress the files and retains the original file. But can also group multiple files into a single file.

**$ compress filename**

**$ uncompress filename**

**$ gzip filename**

**$ gunzip filename**

**$ zip filename**

**$ unzip filename**

**Printing a file ls and cancel–**

**$ lp filename**

**$ lp –dlaser filename – providing the printer name also.**

**$ lp –t “First” filename – will show First in the first page title.**

**$ cancel laser- cancel current job on printer laser.**

**$lpq - to know the number of jobs.**

**$ lprm id– if you want to remove any job of the printer.**

**$ lprm - - removes all the jobs owned by the user.**

**File System Checking fsck-**

Delaying updating superblock and inode block from the memory copy of both, leaves lot of scope for the file system inconsistency. If the power goes off or system shutdown due to some failure in that case superblock are not updated may cause serious error or corruption. The common are –

1. Two or more inodes claiming the same disk block.
2. A block marked as free but not listed in superblock.
3. An inode neither marked free nor in use or having a bad block number that is out of range.
4. Mismatch between the file size specified in inode and the number of data blocks specified in the address array.

Fsck commands are used to check and repair a damaged file system. On many systems, file systems are marked as “dirty” and “clean”. Fsck then checks only the dirty file systems during the next startup.

**$ fsck**

**$ fsck /dev/sda5/**

It does following things –

1. Phase 1- Check Blocks and Sizes
2. Phase 2 – Check path names
3. Phase 3 – Check connectivity
4. Phase 4 – Check Reference counts
5. Phase 5 - Check free list.

**Useradd, usermod and userdel –**

**User-add:**

**#useradd –u 210 –g dba –d /home/oracle –s /bin/ksh –m oracle**

Username –oracle

User ID – 210

Group – dba

Home Path - /home/oracle

Shell - /bin/ksh (optional)

-m option- ensures that the home directory is created added.

Useradd command creates the user entry in /etc/passwd

**#passwd oracle**

Will set the new password for the user oracle.

**/etc/passwd**: all the information of user stored in here except the encryption technique of password. The encryption information is stored in **/etc/shadow**.

**oracle : x : 210 : 241: /home/oracle : /bin/ksh**

**user-mod:**

**# usermod –s /bin/bash oracle**

used to modify some parameters of user. Like sometimes need to change the login shell.

**user-del:**

**# userdel oracle**

this commands removes the user from the system. This removes all entries but the users home directory does not get deleted.

**Role of init in startup and shutdown:**

The startup and shutdown process are controlled by automated shell scripts which are changed quite infrequently.

The kernel (/stand/unix, kernel/genunix or vmliuz) is loaded into the memory and it starts spawning further process. The most important of these is /sbin/init with PID 1, which is responsible for the birth of all subsequent processes. Role of init –

1. It controls the run levels (the system states) and decides which process to run (and kill) for each run level.
2. It spawns a getty process at every terminal or modem port so that users can log in.
3. It also ensures that all the system daemons are running.

**init Run levels-**

0 – system shutdown

1 – System administration mode (local file system mounted)

2 – Multiuser Mode

3 – Full Multiuser Mode

6 – Shutdown and reboot mode

S or s – single user more (file system mounted)

**# init 0**

To check the current run level you are using ‘**who –r**’.

**/etc/inittab –**

The behavior of init controls by /etc/inittab. Init reads all the process from inittab. Its fields determine the process that should be spawned for each of the **init** run levels.

**Backing Up Files (tar, dump, cpio, dd)-**

**dd:** it was a very powerful copying command and mainly for administrator. Previously it was used to copy file system but now its use is restricted to only the floppy diskettes.

Insert first floppy drive and copied data temporarily.

**#dd if=/dev/sda4/ of=$$ bs=147456**

Insert second floppy drive and temporary data is copied

**#dd if=$$ of=/dev/sda4/ bs=147456**

if =input filename

of = output filename

bs =block size