UNIT-1

Introduction to Unix-Brief History-What is Unix-Unix Components-Using Unix-Commands in Unix-Some Basic Commands-Command Substitution-Giving Multiple Commands.

INTRODUCTION TO UNIX

1.1 BRIEF HISTORY

Unix has a longer history than any other popular operating system. Though many schools have contributed to its development, the initial contributions by The Bell Laboratory of AT&T and the University of California, Berkley (UCB) are notable.

Bell Laboratory's contribution In 1965, Massachusetts Institute of Technology (MIT), General Electric, and The Bell Laboratories of AT&T worked on a joint venture project called Multics (Multiplexed Information & Computing System), which intended to develop a multi-user operating system. As the progress was not satisfactory, AT&T withdrew itself from the Multics project in early 1969.

On the basis of the ideas acquired while working on Multics, Ken Thompson, a researcher started working on a different project. He and Dennis Ritchie developed an operating system (OS), called UNICS (Uniplexed Information and Computing Systemuring the latter part of 1969. UNICS was developed completely in the assembly language of PDP-7 and so it was not portable.

To achieve portability, Thompson considered implementing the system in a higher level language. Ritchie developed a higher level language called C in 1973. Ritchie completely rewrote the entire UNIX system during the same year using C. Actually around 95% of this Unix system was written in C and the remaining was written in the assembly language. At the same time many researchers in AT&T showed interest in the Unix project (around 1970 UNICS became Unix). During those days many text-processing utilities along with a text editor called the *ed* editor and a simple command interpreter called the *shell* were developed. The ed editor was a *line* editor and the then developed shell became the Bourne shell (sh), the grandfather of almost all the currently available *shells.*

A system called Unix System V was announced in 1983. System V has since then undergone many revisions and releases. The most important of the releases is the System V release 4 (SVR4) in 1991. SVR4 brought all the important features of various operating systems like BSD, XENIX and SUN operating systems together that were available by then. During the early days of the development of UNIX. However, AT&T made the Unix system available to universities, commercial firms and defence laboratories either free of cost or at a nominal price.

UCB's contribution University of California at Berkeley (UCB) was one of the early universities that was interested in the Unix operating system and its development. The team at Berkeley was responsible for many important technical contributions as well as the development of useful utilities. For example, an editor called the ex editor and a Pascal compiler were developed during 1974 by Bill Joy and Chuck Haley, then graduate students at UCB. Later the ex editor, which was also a line editor, was provided with the screen-editing facilities and was called the vi editor. Another important contribution of Bill Joy was the C-Shell (csh). In general, researchers at Berkeley filled the gaps that existed in AT&T's Unix at that time with their contributions and released their own version of Unix, called BSD-Unix (Berkeley Software Distribution), during the spring of 1978. Since then UCB has had several of BSD releases. These BSD releases are referred to as 4.0BSD(1980), 4.1BSD(1981), 4.2BSD(1983), 4.3BSD(1986) and 4.4BSD(1993). In fact, UCB made many important technical contributions like Virtual Memory System (VMS), Fast File Systems (FFS), socket facility,

Other's contribution During the same period, many computer vendors had developed their own Unix systems. For example, Sun Microsystems (a company that was promoted by Bill Joy) developed Sun operating system, which was revised and renamed Solaries. Solaries 7 is one of the widely used OS even today. Digital Equipment Corporation (DEC) developed a system called Ultrix, which was revised and renamed Digital Unix. Microsoft developed a system called XENIX, the first Unix variant to be run on a PC. This OS was based on both AT&T and BSD systems. XENIX was finally sold to SCO (Santa Cruz Operations). Later, SCO developed its own version of these systems—named SCO Unixware-7 and the SCO open server. Other important systems developed are AIX (by IBM), HP–UX (by HP) and IRIX (by Silicon Graphics).

From the mid-1970s there have been many variants of the Unix system. One of the reasons for this is that being a telephone company, AT&T was not permitted to sell computer-based products. However, it could do so free of cost or for a nominal fee. Because BSD was also giving its products free of cost, many obtained the copies of Unix and worked on them. This resulted in a number of Unix variants. One of the important points that worked against the popularity of any Unix variant for a long time was its user-unfriendliness.

Attempts were made to standardize the Unix system. The first attempt was made by the IEEE standards board. This group came out with a set of rules that should be complied with for an OS to be called standard Unix. These set of rules are widely known as POSIX (Portable Operating System Unix). Now POSIX has also undergone many revisions. The latest one is IEEE 1003.10. In fact, AT&T also has its own standard called Unix international (UI). IBM, HP and DEC also formed a consortium called Open Software Foundation for the same purpose.

In August 1991, a system called Linux was announced by Linus Torvalds (who was only 21 years then) in Finland. Actually it was based on a system called Minix (chiefly developed by Andrew S Tanenbaum) which again was based on Unix. It brought in the speed, efficiency and flexibility of Unix to a PC environment, thereby using the advantages of all the capabilities of Unix. In March 1994, Torvalds released the 1.0 kernel of the Linux. Actually Linux is an open source program—its source code is freely available. Anyone can work on it and make enhancements to it. As a result, it is under constant development. Like other Unix variants it was also initially popular only among the researchers and programmers at universities and research environments. However, at present, Linux has become widely popular among commercial and industrial circles along with the universities and research organizations around the world. Today, Linux has many flavors and can be found on computers ranging from desktops to corporate servers. Red Hat Linux is one of the most popular flavors of Linux. All versions of Linux may be downloaded free of cost from the Web.

1.2 WHAT IS UNIX?

Unix is an operating system. An operating system is a software that acts as an interface between the user and the computer hardware. An operating system acts as a resources manager. Here resources mean hardware resources like the processor, the main memory, the hard disk, I/O devices and other peripherals. In addition to being a multi-user operating system, Unix gives its users, the feeling of working on an independent computer system. Unix also provides communication facility with other users who are connected to the system either directly or indirectly, using networking. It is highly portable and has a large number of utilities and can work both on desktops as well as network environments with equal ease.

1.2.1 Salient Features of Unix: Unix is a multi-tasking operating system—has the ability to support concurrent execution of two or more active processes. Here it may be noted that an instance of a program in execution is known as a process.

Unix is a multi-user operating system—has the ability to support more than one user to login into the system simultaneously and execute programs. For this, the Unix presents a virtual computer to every user by creating simulated processors, multiple address spaces and the like.

The difference between multi-tasking and multi-user system is subtle. In multitasking, different tasks like processes running concurrently belong to one user whereas in a multi-user environment, different tasks belong to different users. However, from the system point of view, the concurrently running tasks are just different processes—them belonging to the same user or to different users is immaterial.

Unix operating system is highly portable. Compared to other OS, it is very easy to port Unix on to different hardware platforms with minimal or no modifications at all. This is because a larger chunk of Unix is built on the language C, which itself is highly portable.

As already mentioned, Unix operating system supports multi-users. These users might be directly connected to the same machine through different terminals or may be connected to different machines that are interconnected. Though initially Unix had no interconnection networking with different computers, the development of communication protocols like TCP/IP have made this possible. This has enabled different users connected to the computer networks to exchange information in the form of e-mail and shared data.

As Unix is a multi-user system, there is every chance that a user may intrude into another user's area either intentionally or unintentionally. Because the security of every user as well as the system is very important, Unix offers solid security at various levels, beginning from the system startup level to accessing files as well as saving data in an encrypted form.

One of the very important key features of any Unix system is that it treates everything, including memory and I/O devices. as files. Thus, there are a large number of files under any Unix environment. Unix has a very well-organized file and directory system that allows users to organize and maintain these files/directories easily and efficiently.

1.3 UNIX COMPONENTS

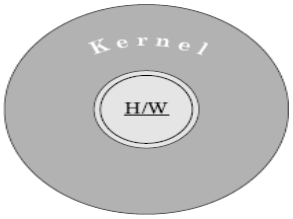
The unix system is said to consist of the following three major components.

1. The kernel
2. The shell
3. The file system

In addition to the above components all commercial Unix systems also include other general utility programs.

1.3.1 The Kernel

The kernel is the heart of any Unix operating system. This kernel is relatively a small piece of code that is embedded on the hardware. Actually, it is a collection of programs that are mostly written in C. Every Unix system has a kernel (just one) that gets automatically loaded on to the memory as soon as the system is booted. As the kernel sits on the hardware it can directly communicate with the hardware.



The kernel

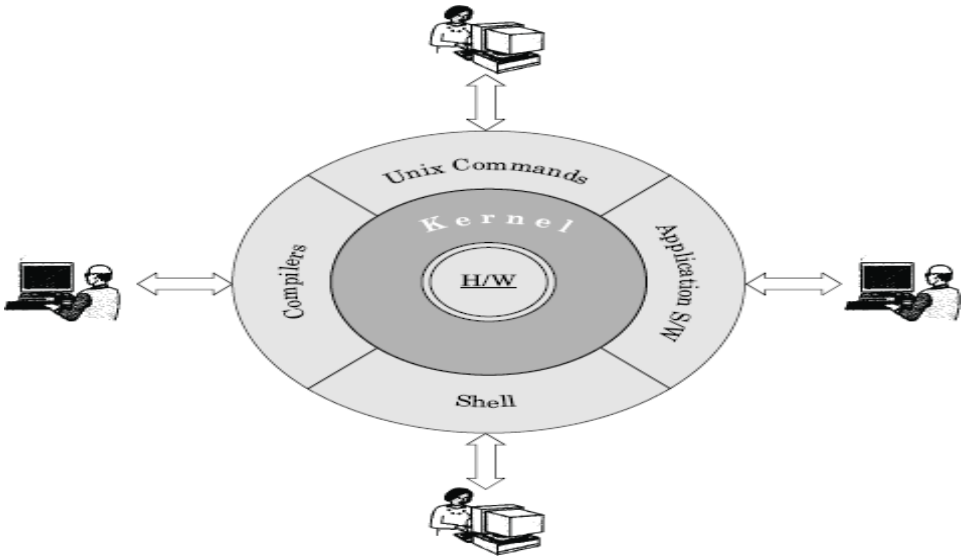
In fact, the kernel is the only component that can communicate with the hardware directly. It is the kernel that manages all the system resources like memory and I/O devices, allocates time between

users and processes in the case of multi-user environment, decides process priorities, manages interprocess communication (IPC) and performs many other such tasks.

Earlier, all the programs that were a part of a kernel, were integrated together and moved onto the memory during booting. Such integrated kernels are referred to as monolithic kernels. However, only the just-necessary module is moved onto the memory during booting, is called a microkernel. Other modules are moved in and out of the memory depending on the requirement.

1.3.2 The Shell

Every Unix system has, at least, one shell. A shell is a program that sits on the kernel and acts as an agent or interface between the users and the kernel and hence the hardware. It is similar to the command prompt in the MS-DOS environment. The shell has certain programming capability of its own. Using this capability, programs called shell programs can be written. Generally shell programs are called shell scripts.



Unix system components

Types of shells There are different types of shells available. Some of them are discussed here.

The Bourne shell (sh) This is the most common shell available on Unix systems and the first major shell to be developed. This shell is widely used. It has been named after its author, Stephen Bourne at AT&T Bell Labs. This shell is distributed as the standard shell on almost all Unix systems.

The C shell (csh) Bill Joy developed this shell at UCB as a part of the BSD release. It is called the C shell because its syntax and usage is very similar to the C programming language. Unfortunately this shell is not available on all machines. Shell scripts written in the C shell are not compatible with the Bourne shell. One of the major advantages of the C shell over the Bourne shell is its capability to execute processes in the background. A version of this shell called tcsh is available free of cost under Linux.

The Korn shell (ksh) This shell was developed by David Korn at AT&T Bell labs. Basically it is built on the Bourne shell. It also incorporates certain features of the C shell. At present it is one of the widely used shells. It can run Bourne shell scripts without any modifications. One of its versions, the public-domain Korn shell (pdksh), comes with Linux free of cost.

The Bourne-Again shell (bash) This shell was developed by B Fox and C Ramey at Free Software Foundation. Certain Linux operating system variants come with this shell as its default shell. This is clearly a free ware shell.

Shell as a Command Processor As already discussed, the shell acts both as a command processor and a small programming language. Given here is a brief account of the behaviour of the shell as a

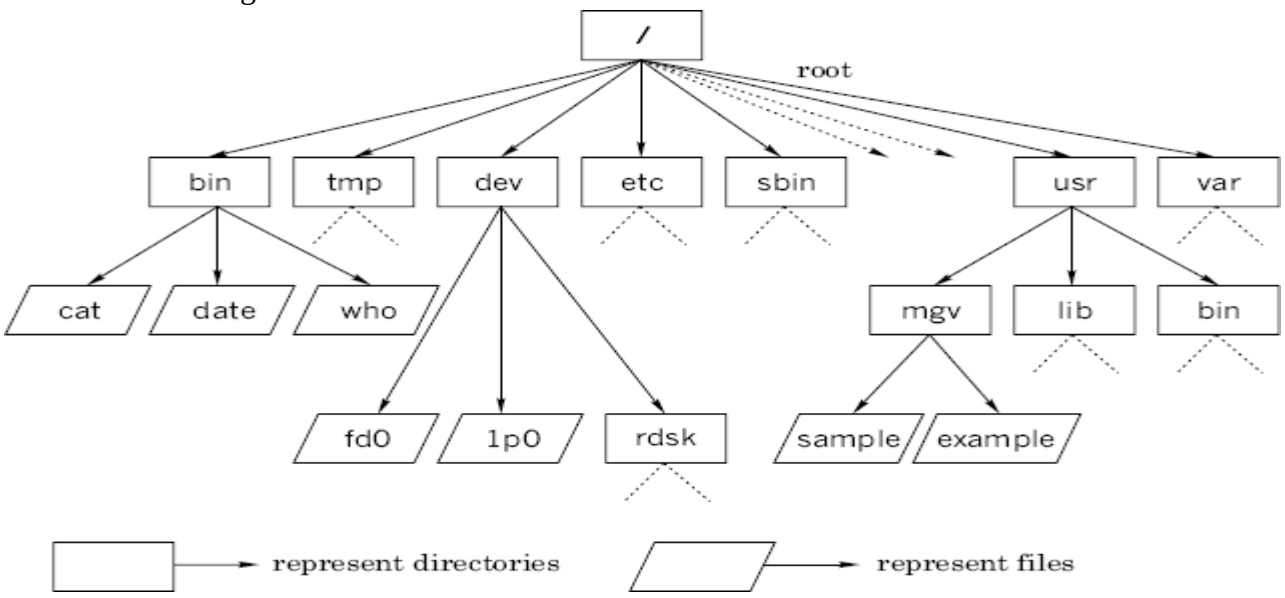
command processor. When interpreting a command line given at its prompt, the shell follows one or more or all of the following steps, depending on the contents of the command line given to it.

1. It parses the command line and identifies each and every word in it and removes additional spaces or tabs present, if any.
2. Evaluates all the variables present that might be prefixed with a $.
3. If commands are present within back quotes, they are executed and their output is substituted into the command line. In other words, command substitution takes place.
4. It then checks for any redirection of the input and/or output and establishes the connectivity between the concerned files accordingly.
5. It then checks for the presence of wildcard characters like *, ? and [,]. If any of these characters are present, file name generation and substitution take place.

It then looks out for the required commands as well as files, retrieves them and hands them to the kernel for execution. The route or the path taken for looking out for the required commands will be in the PATH shell variable. Also the semicolon that allows multiple commands, and logical operators are taken care of by the shell.

1.3.3 The File System

A file system is another major component of a Unix system. Unix treats everything—including hardware devices—as a file. All the files in a Unix system are organized in an inverted tree-like hierarchical structure. This structured arrangement in which all the files are stored is referred to as a file system. A file system could be local to a system or it could be distributed. Local file systems store and manage their data on devices directly connected to the system. Distributed file systems allow a user to access files residing on remote machines.



1.4 USING UNIX

To use Unix, one has to get into the Unix environment. The process of getting into the Unix environment is known as *logging in* into the system. As soon as the system is booted a daemon called init gets started. This init daemon spawns a process called getty for every terminal. Each one of these gettys print the login prompt on the respective terminal. When a user attempts to enter into the Unix environment, that is, tries to login, the login program is executed in order to verify the user name and the password. A file called password file under the /etc directory contains a line for every user, containing the user's login name, numerical user id, encrypted password, home directory, and other such information. When the user logs in, the login program encrypts the password just read from the

terminal and compares it with the password in the password file. If they agree, the login is permitted; if not, it is disallowed. Every user has a user id as well as a password allocated to them by the system administrator. This is true even in the case of single-user systems. However, it may be noted that the user will be the system administrator in the case of single-user systems. The sequence of events in a complete login process can be listed as follows.

- The user enters a login name at the getty's login prompt on the terminal.
- getty executes the login program with the login name as the argument.
- login requests for a password and validates it against /etc/passwd.
- login sets up the TERM environment variable and runs a shell.
- The shell executes the appropriate startup files like .profile.
- The shell then prints a prompt, usually a $ or a % symbol and waits for further input. This indicates the successful entry made into a Unix environment with a proper shell.

The above-listed sequence of events that take place during a login process is schematically shown

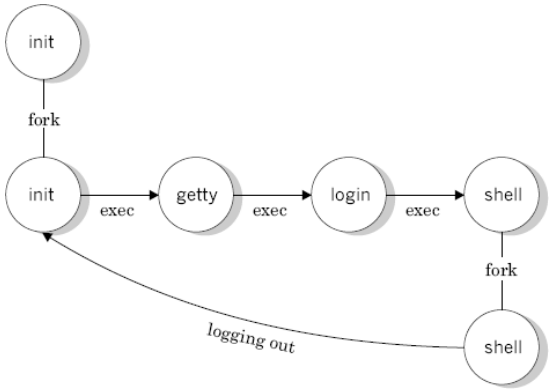


Fig: The log process

When the user completes the session with the system he comes out of the Unix environment. The process of coming out of the Unix environment is known as *logging out.* As soon as the user logs out, the control returns to the init daemon, which in turn spawns a new getty on the corresponding terminal. This facilitates a new user to login to the system.

1.4.1 The Shell Prompt

Successful login into a Unix system is indicated by the appearance of a prompt called the shell prompt or system prompt on the terminal. The character that appears as a prompt depends on the shell used.

List of Default Prompts

Prompt	***Shell***
$ (dollar)	Bourne and Korn shells (sh, bash and ksh)
% (percent)	C shells (csh and tcsh)
# (hash)	Any shell as root

1.5 COMMANDS IN UNIX

Unix has a large number of commands. A list of some general features of a Unix command is given below.

1. A Unix command is a program written to perform certain specific action. All such programs have a name. For example, the program that is used to print today's date in a specific manner has the name date and the program that is used to create a small file or display the contents of a file has the name cat and so on.

2. All Unix commands are written using lower case letters. For example, cat, ls, who, date, and so on.
3. Almost all the Unix commands are cryptic. For example, cat stands for concatenation, ls stands for listing and so on. Unix commands were developed to be cryptic because it was developed by researchers for researchers and the early computer systems were very slow which demanded more time for typing, editing and executing long commands.
4. Unix commands can have zero, one or more number of arguments associated with them.
5. Unix commands can also have format specifiers as well as options associated with them. Format specifiers, whenever present, are indicated by the + character. Options, whenever present, are indicated by hyphen (–). There could be many number of options associated with a command. It is interesting to note that the listing command (ls) has nearly two dozens options that could be used with it.
6. In certain situations, a Unix command with its arguments or a series of commands may not fit in a single line (80 characters). In such cases it may overflow. This is permitted in Unix. Whenever there is an overflow, it is indicated by the appearance of a special prompt in the form of a > symbol in the beginning of the next line. Such a special prompt is known as the secondary prompt.
7. A current Unix command can be killed by using either <delete> or <ctrl-u> command.
8. Commands can be given to the system even when a command given earlier is being executed in the background. This is not possible with the Bourne shell, sh.

1.5.1 Types of Unix Commands

Basically there are two types of Unix commands. They are—external commands and internal commands.

External Commands A command with an independent existence in the form of a separate file is called an external command. For example, programs for the commands such as cat and ls, exist independently in a directory called the /bin directory. When such commands are given, the shell reaches these command files with the help of a system variable called the PATH variable and executes them. Most of the Unix commands are external commands.

Internal Commands A command that does not have an independent existence is called an internal command. Actually the routines for internal commands will be a part of another program or routine. For example, the echo command is an internal command as its routine will be a part of the shell's routine, sh. In other words the echo command is built into the shell. As such, internal commands are also called the built-in commands. cd and mkdir, are two examples of internal commands.

1.6 SOME BASIC COMMANDS

Unix has several hundreds of commands within it. Most of them are simple and are powerful. Some of the commands are general in nature from the user's point of view. A few of such commands are introduced in the following sections.

1.6.1 The echo Command

The echo command is used to display messages. It is quite useful in developing interactive shell programs. It takes zero, one or more number of arguments. Arguments may be given either as a series of individual symbols or as a string within a pair of double quotes (" "). Some examples are given below.

```
$ echo
                    #A Blank line is displayed
$
```

1. $ echo I am studying computer science.
   I am studying computer science.
   $
2. $ echo I am studying computer science.
   I am studying computer science.
   $
3. $ echo "I am studying computer science."
   I am studying computer science.
   $
4. $ echo The home directory is $HOME
   The home directory is /usr/mgv
   $

1.6.2 The tput Command

This command is used to control the movement of the cursor on the screen as well as to add certain features like blinking, boldface and underlining to the displayed messages on the screen. Such facilities might be used to add aesthetic value to the shell programs. For example, this command along with the clear argument clears the screen and puts the cursor at the left– top of the screen. However, it may be noted that clear itself is a command which alone could be used to clear the screen.

```
$tput clear
```

This command along with the cup argument and certain co-ordinate values is used to position the cursor at any required position on the screen. An example is given below.

```
$tput cup 10 20
```

When the above command line is executed, the cursor will be placed at the tenth row and the twentieth column on the screen. Now, if an echo command is given, the message will be displayed starting from the new position.

The number of rows and columns on the current terminal is known by using the lines and cols as arguments to the tput command as shown in the following examples.

```
$tput    lines
48
$
$tput    cols
142
$
```

From the above examples, it is seen that there are 142 columns and 48 lines on the current terminal.

1.6.3 The tty Command

In Unix, every terminal is associated with a special file, called the device file. All the device files will be present in the /dev directory. A user can know the name of his device file on which he is working by using the tty command, as shown in the example below.

```
$tty
/dev/tty01
$
```

Here, tty01 is the device file name and will be available in the directory /dev. Under Linux, the output of this command will be as shown below.

```
$tty
/dev/pts/0
$
```

1.6.4 The who Command

Unix maintains an account for all the current users of the system. Because it is a multi-user system it is prudent for the user to be aware of other current users so that s/he can communicate with them, if required.

The user can know login details of all the current users by using the who command. The use of who command provides a list of all the current users in the three-column format by default, as shown follows.

```
$who
root        console        Nov  19   09:35
mgv         tty01          Nov  19   09:40
dvm         tty02          Nov  19   09:41
$
```

The first column shows the name of the users, the second column shows the device names and the third column shows the login time.

Some options like –H, –u and –T can be used with this command. The –H option provides headers for the columns and the –u option provides more details like idle time, PID and comments as shown in the example below.

```
$who –Hu
NAME        LINE        TIME                IDLE      PID      COMMENTS
root        console     Nov 19  09:35       .         32
mgv         tty01       Nov 19  09:40       0:20      33
dvm         tty02       Nov 19  09:41       0:40      34
$
```

If any terminal is idle (not active) for the last 1 minute, the information, that is, for how long that terminal is idle will be indicated on the IDLE column. Thus, 0:20 indicates that mgv's terminal is IDLE for the last 20 minutes. This information will be useful to the system administrator. The PID indicates the process identification number.

The self-login details of a user can be obtained as a single line output using am and i arguments along with the who command as follows.

```
$who am i
mgv         tty01          Nov  19   09:40
$
```

Generally the who command is used by the system administrator for monitoring terminals.

1.6.5 The uname Command

Using this command one can know the details of one's Unix system. We already know that there are many Unix variants. When this command is used, it gives the name of the Unix system being used by the user. Certain options like r, v, m and a, can be used with this command. Following are given some examples.

1.
```
$uname
Linux
$
```
2.
```
$uname –r
2.4.18 – 3              #release details
$
```
3.
```
$uname –m
i686                    #machine details
$
```

The use of the option –v gives the version of the system being used. The use of the –a option gives all the details of the system.

1.6.6 The date Command

Using this command the user can display the current date along with the time nearest to the second. This is done with the help of the Unix system's internal clock that runs with battery back up during shutdowns.

```
$date
Sat  Jan 10  11:58:00  IST  2004
$
```

This is one of the very few commands that allows the use of format specifiers as arguments. Format specifiers are single characters using which, one can print the date in a specific manner. Each format specifier is preceded by a + symbol followed by the % operator. For example, by using the format specifier m one can display only the month in the numeric form as follows.

```
$date +%m
09
$
```

Instead of the numeric form, the name of the month can be displayed using the h format specifier as shown below.

```
$date +%h
Sep
$
```

More than one *format specifier* can be specified at a time. In such cases either double quotes (" ") or single quotes (' ') are used.?

```
$date +"%h %m"
Sep 09
$
```

Some of the other codes that could be used as format specifiers with the date command are

1. D and d for the day of the month. (D gives the day in the format mm/dd/yy, where as d gives the day in the format dd).
2. Y and y for the year (Y gives all the four digits of the year, whereas y gives only the *last* two digits).
3. H, M and S stand for hour, minute and second, respectively.

Many number of options like u, r, R, f, can also be used with this command. For example, the use of the u option displays the universal time (Greenwich Mean Time) as shown in the example below, where UTC is Coordinated Universal Time.

```
$date –u
Sat  Sep 25  05:58:20  UTC  2004
$
$date "Today's date is +%D"
Today's date is 03/16/04
$
```

The System Date The date command is also used by the system administrator to change or reset the system date. This usage has a different syntax. For changing the date, that is, to set the date, a numeric argument is given. This argument is usually an eight characters long string having the form MMDDhhmm (month, day, hour in 24-hour format and minutes) followed by an optional two-digit year. A typical example of this is given below.

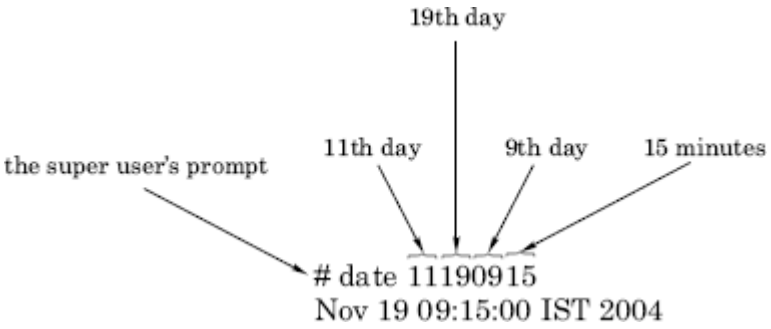


Fig. Setting system date

Under Unix, lot of importance is given to the system time. For example, the time at which a file is created, modified or accessed is recorded depending upon the system time. Also, login and logout times are recorded based on the system's time. There are certain commands (like, cron and at) whose action depends upon the system's time. For these and more reasons the system should sufficiently indicate the correct time. However, excessive manipulation of the system time should be avoided.

1.6.7 The cal Command

This command is used to print the calendar of a specific month or a specific year. It prints the Gregorian or New Style calendar for any month or year between the years 1 and 9999. When this command is used without any arguments, the calendar of the current month of the current year will be printed, as shown below.

```
$cal
                Dec 2004
Su      Mo      Tu      We      Th      Fr      Sa
                        1       2       3
4       5       6       7       8       9       10
11      12      13      14      15      16      17
18      19      20      21      22      23      24
25      26      27      28      29      30      31
$
```

When two numeric arguments, are given the first argument will be considered as the month, the second argument will be considered the year and the calendar for that month of that year will be printed as shown in the following example.

```
$cal 09 1949
             September 1949
Su      Mo      Tu      We      Th      Fr      Sa
                                1       2       3
4       5       6       7       8       9       10
11      12      13      14      15      16      17
18      19      20      21      22      23      24
25      26      27      28      29      30
$
```

When given with a single numeric argument, the complete calendar for the entire year represented by the numeric argument will be printed as follows.

```
$cal 2002
                                2002

                Jan                     Feb                     Mar
Su Mo Tu We Th Fr Sa    Su Mo Tu We Th Fr Sa    Su Mo Tu We Th Fr Sa
          1  2  3  4  5                    1  2    31                 1  2
 6  7  8  9 10 11 12     3  4  5  6  7  8  9     3  4  5  6  7  8  9
13 14 15 16 17 18 19    10 11 12 13 14 15 16    10 11 12 13 14 15 16
20 21 22 23 24 25 26    17 18 19 20 21 22 23    17 18 19 20 21 22 23
27 28 29 30 31          24 25 26 27 28          24 25 26 27 28 29 30

                April                   May                     Jun
Su Mo Tu We Th Fr Sa    Su Mo Tu We Th Fr Sa    Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6                 1  2  3  4    30                    1
 7  8  9 10 11 12 13     5  6  7  8  9 10 11     2  3  4  5  6  7  8
14 15 16 17 18 19 20    12 13 14 15 15 17 18     9 10 11 12 13 14 15
21 22 23 24 25 26 27    19 20 21 22 23 24 25    16 17 18 19 20 21 22
28 29 30                26 27 28 29 30 31       23 24 25 26 27 28 29

                Jul                     Aug                     Sep
Su Mo Tu We Th Fr Sa    Su Mo Tu We Th Fr Sa    Su Mo Tu We Th Fr Sa
    1  2  3  4  5  6                    1  2  3     1  2  3  4  5  6  7
 7  8  9 10 11 12 13     4  5  6  7  8  9 10     8  9 10 11 12 13 14
14 15 16 17 18 19 20    11 12 13 14 15 16 17    15 16 17 18 19 20 21
21 22 23 24 25 26 27    18 19 20 21 22 23 24    22 23 24 25 26 27 28
28 29 30 31             25 26 27 28 29 30 31    29 30

                Oct                     Nov                     Dec
Su Mo Tu We Th Fr Sa    Su Mo Tu We Th Fr Sa    Su Mo Tu We Th Fr Sa
          1  2  3  4  5                    1  2     1  2  3  4  5  6  7
 6  7  8  9 10 11 12     3  4  5  6  7  8  9     8  9 10 11 12 13 14
13 14 15 16 17 18 19    10 11 12 13 14 15 16    15 16 17 18 19 20 21
20 21 22 23 24 25 26    17 18 19 20 21 22 23    22 23 24 25 26 27 28
27 28 29 30 31          24 25 26 27 28 29 30    29 30 31
   $
```

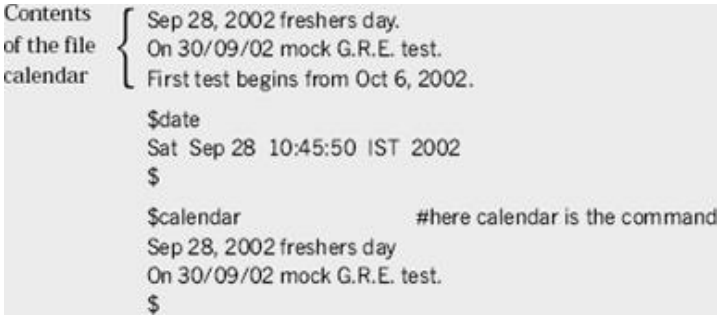
On some systems the month can be given in short—as Dec, Sep, and so on. Care should be taken to give the year in proper numeric format. For example, the year 1949 should be given as 1949. If, by chance, it is given as only 49, the calendar of AD 49 will be printed.

If, for any reason, the calendar-display on the monitor scrolls up (that is, it cannot fit into a single screen and moves up), the scrolling can be paused using the <ctrl-s> command and continued using the <ctrl-q> command. However, the use of the more command is recommended in such cases. The use of the more command displays the output one page at a time.

1.6.8 The calendar Command

It is like an engagement dairy that contains text information and offers a reminder service based on a file called the calendar. This file must be in the present working directory/home directory. This file is created and managed by the user with the help of an editor (like vi editor) on the system. This command works on today and tomorrow dates concept. The present working day's date (picked up from the machine) is taken as today and the days upto and including the next working day are treated as tomorrow. For example, if it is a five-day working week and today is a Friday, then all the days upto and including the next Monday will be considered as tomorrow.

The success of this facility depends upon the date formats permitted to write the calendar text file and the way in which the calendar command searches the file calendar. Typically a calendar file may contain the information as shown below.



1.6.9 The passwd Command

As already mentioned, Unix is a multi-user system due to which there is always a security threat. Many levels of security measures have been included into Unix systems. The simplest and most widely used by all individual users is the use of passwords. During the addition of new users, the system administrator permits or authorizes the new user by assigning a unique password to him or her. A user can change his or her password using the passwd command. In fact, users are advised to change their passwords quite often. The following illustration shows how a user can change the password using the passwd command.

```
$passwd
Old Password: ********
New Password: ********
New Password: ********
$
```

As soon as the passwd command is executed, the system first asks for the old password. When the old password is keyed in correctly the system asks for the new password twice. First time for entering the new password and second time for the confirmation. When everything is keyed in properly, the $ prompt reappears. It may be noted that neither the old password nor the new password is displayed.

1.6.10 The lock Command

For security reasons it is necessary that one should not leave any terminal session unattended as someone else might sneak onto the system and cause problems intentionally or unintentionally. It is advisable either to logout or lock the terminal session before leaving it temporarily. The lock command is used for locking a session for any required amount of time.

Generally this command is used without any argument as shown in the following illustration. By default, the user can lock it for 30 minutes. This locking period can be changed by assigning a different value for the systemvariable DEFLOGOUT. When the lock command is given, the terminal asks for a password twice as shown in the example below.

```
password:********
re-enter password:********
terminal locked by mgv 0 min ago
```

The password used here need not be the actual password that is used to log into the system. It could be any temporary password. One can lock a terminal for a maximum period of 60 minutes. A numeric option may be used to lock the terminal for any period ranging between 1 and 60 minutes as shown in the example below.

```
$lock–45          #locks for 45 minutes
```

The locked terminal can be unlocked by re-entering the password with which the terminal was locked earlier. If the terminal is not activated before the lock period expires, the system automatically gets unlocked at the end of the specified time period.

Many Linux distributions include a locking command called vlock. The lock command is used to lock sessions individually, whereas vlock may be used to lock all individual sessions simultaneously. Also a utility called lock screen is available, with many modern OS, using which a session on a terminal can be locked.

1.6.11 The banner Command

This command is available on SCO Unix (but not on Linux). It is used to display banners or posters. This command simply produces a blown-up version of the characters that are supplied with the command. An example is given below.

```
$banner Larry Wall
#
#          ##    ######  ###### #    #
#         # #    #    # #    #  #  #
#        #  #    #    # #    #   #
#        #####   #####   #####    #
#       #    #   #    # #    #    #
######  #    #   #    # #    #    #

#      #
#      #   ##    #      #
#  #   #  #  #   #      #
#  #   #  #    #  #      #
#  #   #  ####   #      #
#  #   #  #    #  #      #
 ##  ##   #    #  ###### ######
$
```

It may be observed that there are two arguments and each argument has been printed on a separate line. It prints a maximum of 10 characters per line. In case an argument consists of more than 10 characters, only first 10 characters will be printed and the remaining will be truncated. As seen from the example above, the output will be made up of the # (hash). A series of arguments may be given as a single argument in the form of a string as shown in the following example.

```
$banner "Larry Wall"
#                                    #      #
#          ##    ######  ###### #   #  #      #   ##    #      #
#         # #    #    # #    #  #  #   #   #  #  #  #  #      #
#        #  #    #    # #    #   #     #   #  #  #   # #      #
#        #####   #####   #####     #    #   #  #  #### #      #
#       #    #   #    # #    #    #    #   #  #  #   # #      #
######  #    #   #    # #    #    #     ##  ##  #    #  ###### ######
$
```

Further, it may be noted that Larry Wall is the creator of the Perl language.

1.6.12 The cat Command—Creating Small Files

This is one of the most useful and quite frequently used Unix commands. One of the basic purposes of this command is to create small Unix files. A file can be created by writing a command line as shown in the following example where review is the name of the file being created.

$cat > review
A > symbol following the command
means that the output
goes to the file name following it.
<ctrl-d>
$

In the above example after executing the $cat > review command, the $ prompt vanishes and the system is ready to accept the input from the standard input—the keyboard. At this point the user can type in whatever—s/he wants. The input operation is terminated by using <ctrl-d> on a new line. The input termination command <ctrl-d> does not get into the file. If the file being created already exists, it will be overwritten. One of the drawbacks of this method of creating files is that it lacks editing capabilities. Therefore, it is seldom used for creating files of any considerable size. For creating files of considerable size, editors like vi and emacs are used.

Name:cat

Purpose: It is used

i. To create the files
ii. To display the contents of a file
iii. To Append the files

Syntax:

```
cat    [option(s)]    file(s)
cat    file1  file2>newfile
cat    file1 >> file2
```

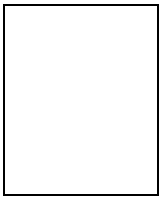
Options:

Option	Description
-e	Print $ at the end of line
-n	Numbering lines
-s	Silent(no error messages)
-t	Print tabs and form feeds
-v	Print control characters
-b	Ignore blank lines used with –n option.

Examples & Description:

- Simple *cat* command accepts the data from the standard input (key board) and immediately prints the data to standard output(monitor.)
- It accepts and prints the data until it encounters the **eof** character**(ctrl-d)**
- **EX:**

```
$cat
 1234
1234
Abcd
Abcd
Xyzl
Xyzl
[ctrl-d]
$
```



(i)File Creation

- The second use for *cat* is file creation.
- It is accomplished by typing *cat* followed by the output redirection operator and the name of the file to be created, then pressing ENTER and enter data into the file finally simultaneously pressing the CONTROL and *d* keys. So cat command stops to accepts data from key board .For example, a new file named *file1* can be created by typing **cat > file1**
- If a file named *file1* already exists, it will be *overwritten* by the new.

```
$cat>x
Unix is a multitasking os
Unix is a multi user os
In unix the command prompt is $
We can set our own command prompt
[ctrl-d]
$
```

With this a new file **x** is created.

- Using output redirection operator(>) in cat command we can copy the contents of more files into specified file.
- **cat f1 f2 f3 >f4**
  It copies all three files f1,f2,f3 contents into f4.
  Ex:

```
$cat   f1
This is pen
$cat   f2
This is rose
$cat  f3
This is jasmine
```

```
$cat f1 f2 f3>f4
$cat f4
This is pen
This is rose
This is jasmine
```

(ii) File displaying:

Cat command displays the contents of a file on monitor.

```
$cat  x
Unix is a multitasking os
Unix is a multi user os
In unix the command prompt is $
We can set our own command prompt
$
```

```
$cat   y
This is pen
$cat f
This is jasmine
$cat   z
 This is rose
```

- The cat command displays the contents of 2 or more files at the same time.
- the following command will concatenate copies of the contents of the four files *x*,y,z, and *f* and displays all 4 file contents.

```
$cat  x  y  z  f
Unix is a multitasking os
Unix is a multi user os
In unix the command prompt is $
We can set our own command prompt
This is pen
This is rose
This is jasmine
$
```

- While displaying more number of files, if any one file is not existed in the working directory ,then the cat command displays the contents of all available files and generates an error message about the non-existent file.
- **Ex:**

```
$cat   y  z  f   p
This is pen
This is rose
This is jasmine
Cannot open: p no such file or directory
$
```

Here **x,y,z,f** files are existed so the contents of all these files are displayed. But **p** is not available so error message was displayed regarding **p**

Options:

-s(suppressing errors): It suppresses the error messages. While displaying the files if the file is not available then no error message is displayed.

```
$cat  -s  y  z  f   p
This is pen
This is rose
This is jasmine
$
```

- No error message regarding *file p* even though it is not existed. Because of –s option.

-n(numbering to lines):

a. This option gives the number to each line while displaying the output.

```
$cat  x
1:Unix is a multitasking os
2:Unix is a multi user os
3:In unix the command prompt is $
4:We can set our own command prompt
$
```

b. cat command gives line numbers to blank lines also

```
$cat m
Abc

Def

lmn
```

```
$cat  –n  m
1:  Abc
2:
3: Def
4:
5:  lmn
```

-b(excluding Blank lines): This option is used with –n option.

Because of this option blank lines are ignored . so –n option does not give the number to blank lines

```
$cat  –bn  m
1: Abc
2: Def
3: lmn
```

-e (print $ at end of each line): this option prints $ at the end of each line in a file.

```
$cat  x
Unix is a multitasking os$
Unix is a multi user osS
In unix the command prompt is $$
We can set our own command prompt$
$
```

-t(prints tabs and form feeds): It prints tabs and form feeds.

```
$cat>k
this    is      an      example
[ctrl-d]
```

```
$cat  k
this  ^It  is  ^It  an  ^It example
$
```

(iii) File appending:

While creating a new file if the file is not existed *cat* creates new file, if already a file is existed then it is overwritten. to avoid this cat command uses append operator.

It is indicated with >> symbol. It adds one file content to the end of the other file.

Ex:

```
$cat   y
This is pen
```

```
$cat   z
This is rose
```

```
$cat   y>>z
$cat   z
This is rose
This is pen
```

1.6.13 The bc Command

The bc command is both a calculator and a small language for writing numerical programs. Using this command one can perform all the usual arithmetic operations as well as change of bases in the range of base 2–16.

Arithmetic operations are performed using the built-in library functions. The special functions that are available in the library are sin( ), cos( ), arctan( ), ln( ), exp( ), bessel( ). The arguments to the trigonometric functions must be given in radians. Math functions are used by invoking bc with the option –l. Also these math functions are used with the acronyms that follow.

Function	***Acronym***
Cosine	c(n)
Sine	s(n)
Tan	t(n)
Arctan	a(n)
natural log	l(n)
exponential function	e(n)
square root	sqrt(n)
exponent	^

The bc can be used by either entering expressions to be evaluated from the keyboard or running programs stored in files. The user can define one's own functions and use them.

The syntax used to write numeric programs and to define user-defined functions is similar to the ones used in the C language.

This command uses the infix method of entering data and specifying operations. The required degree of precision is obtained using the scale function available in the library. By default, the scale factor will be 0. It is possible to set a maximum value of 100 to the scale.

This calculator is invoked by just typing in the bc command at the system prompt. When this command is given, a cursor appears on the monitor at which the expression to be evaluated is given. It should be noted that one command line is executed at a time. In other words it works in the interpreter mode. Below are given some examples of such a mode.

```
1.$bc
  sqrt55
  7
  quit
  $
```

```
2.$bc
  scale=4
  sqrt55
  7.4161
  quit
  $
```

```
3.$bc
  ibase=5
  obase=16
  2341
  424
  ibase=16
  obase=5
  424
  2341
  quit
  $
```

From the above examples one can understand the following.

1. The default value of the function scale is 0 (Zero).
2. Precision is set to 4 or any required value using the scale function above.
3. The result is displayed immediately in the next line after the execution of every line.
4. A session with bc is terminated by using the quit command.
5. Base conversion is carried out using ibase and obase functions.
6. ibase stands for the input base and obase stands for the output base. Default values for both ibase and obase is 10.

As already mentioned, numeric programs can be written and used with the bc command. While writing numeric programs only single-character variables are used and only lower case English letters are allowed as variables. There can be a maximum of only 26 variables in a program. However, with Linux, variable names having more than one alphanumeric character with the first character being an alphabet can also be used.

An example that uses the control construct is shown here.

```
$bc
for(i=1;i<=4;i=i+1)          i^2
1
4
9
16
quit
$
```

1.6.14 The spell and ispell Commands: Spell-Check Programs

The spell command is the first program that was developed to check for words that are wrongly spelt in a document. This command displays a list of misspelled words in the document used as argument, as shown below.

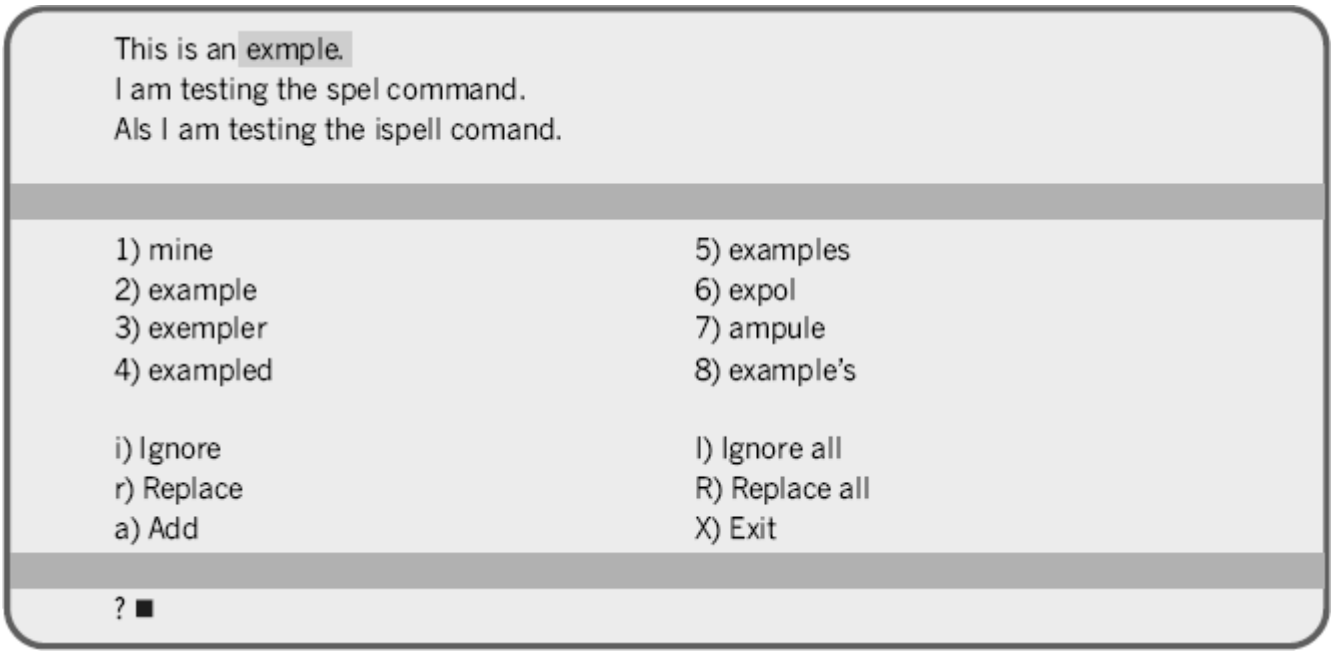
```
$cat spell.ux
```

```
This is an exmple.
I am testing the spel command.
Als I am testing the ispell comand.
$
$spell spell.ux
Als
comand
exmple
spel
$
```

As seen from this example all the misspelled words are displayed in alphabetical order. These words are noted down (or saved in a separate file) and necessary corrections are made using an editor. By default, the spell command checks for the spelling based on American usage. Spell checking may be made according to British usage using the –b option. Actually spell check is made by comparing the words in the text with the words on an inbuilt dictionary. A user can provide one's own dictionary also.

The ispell command is an interactive spell-check program available in Linux. When used, this command displays a screen full of information in three sections as shown below.

```
$ispell spell.ux
```



As seen from the above display, the misspelled words are highlighted. Alternate suggestions (a maximum of 10) for the misspelled words will be given in the middle section. Also the information about alternate actions that one could take will be displayed in this section. A question mark (?) appears in the last section with a blinking cursor along with it. A misspelled word can be substituted with a correct one by using the serial number of the correct word. Suggestions can be ignored. In case none of the suggestions are suitable, an external word can be input and substituted by using the replace command. This new word can be added to the dictionary by using the add action.

1.7 COMMAND SUBSTITUTION

In Unix, it is possible to run a command within a command. For example, the date command can be run within the echo command by writing a command line as follows.

```
$echo Today the date is 'date'
```

Today the date is Fri Oct 3 16:25:00 IST 2002

As shown in the above example, the command to be executed (that is, date, in this example) within another command (that is, echo in this example) has to be written within a pair of backquotes ('). The shell while parsing the parameters list of the echo command treats the words that are backquoted as a command, executes it and substitutes the result of this execution at the corresponding position in the parameters list. This process is known as command substitution.

In Korn shell the command substitution is accomplished by using a $ sign followed by the command within a pair of parenthesis as shown below.

```
$echo Today the date is $(date)
Today the date is Fri Oct 3 16:25:00 IST 2002
```

1.8 GIVING MULTIPLE COMMANDS

Normally a single command is given to the shell at its prompt. However, there are many situations when more than one command is given in a single command line. One of the ways of giving multiple commands is to use a semicolon (;) between successive commands as shown below.

```
$echo "Giving multiple commands"; date; who
```

Commands given in this way does not mutually interact with each other in any manner. They are executed independently one after the other, from left to right as they appear in the command line. Giving multiple commands in a single command line has a definite advantage as the entire command line could be executed as a background job and something else could be done in the foreground. Of course, the Bourne shell (sh) does not permit processing of jobs in the background where as the Korn shell (ksh) does.

1.9 ALIASES—GIVING ALTERNATE NAMES TO COMMANDS

With some of the recent and popular shells like korn and bash it is possible to assign short and meaningful names to a long command or a combination of commands and use these short names as alternate names. Such short names are called aliases of the original command names or combination of commands. This is accomplished using a command called the alias command.

Assume that one needs to know the date, time and users information quite frequently. For this one has to use the who and date commands at all the times. In such circumstances one can use an alias command as follows.

```
[/home/mgv]alias whoda=`who;date`
[/home/mgv]_
```

Here one should observe that there should not be any spaces either before or after the equal to (=) operator. After aliasing a command or a sequence of commands one can just use the alias name for the required purpose. In other words, henceforth the word whoda can be used as a command to get the date, time and users information.

A list of all the aliases can be obtained by using the alias command without arguments. An alias is removed by using the unalias command, as shown in the following example. Too much usage of aliases may lead to confusion and is highly error prone. So, one should be very selective while using this facility.

```
[/home/mgv]unalias whoda
[/home/mgv]_
```

THE FILE SYSTEM & FILE ATTRIBUTES AND PERMISSIONS

The File system –The Basics of Files-What's in a File-Directories and File Names-Permissions-I Nodes-The Directory Hierarchy, File Attributes and Permissions-The File Command knowing the File Type-The Chmod Command Changing File Permissions-The Chown Command Changing the Owner of a File-The Chgrp Command Changing the Group of a File.

2.1The basics of flies

A file is a sequence of bytes. No structure is imposed on a file by the system, and no meaning is attached to its contents — the meaning of the bytes depends solely on the programs that interpret the file.

create a small file

```
$ cat> j
now is the time
for all good people
$
```

```
$ ls -1 j
-rw-r—r— 1 cse501 36 Sep 27 06:11 j
$
```

j is a file with 36 bytes — the 36 characters you typed while appending. To see the file,

```
$ cat j
now is the time
for all good people
$
```

cat shows what the file looks like.

The command **od (octal dump)** prints a visible representation of all the bytes of a file:

```
$ od -c j
0000000   n  o     w     i     s     t     h     e     t     i     m     e    \n
0000020   f  o     r     a     l     l     g     o     o     d     p     e     o
0000040   p  1     e    \n
0000044
$
```

The -c option means "interpret bytes as characters."

The –b option will show the bytes as octal (base 8) numbers.

The -x option tells od to print in hex.

```
$ od -cb j
0000000   n  o     w           i  s           t     h     e           t     i     m     e          \n
          156  157  167  040 151 163 040 164  150   145 040 164  151   155   145       012
0000020   f  o     r           a     l     l           g     o     o     d           p     e     o
          146 157 162 040 141 154 154 040 147 157 157 144 040 160  145 157
0000040   p  1     e          \n
          160  154   145     012
0000044
$
```

The 7-digit numbers down the left side are positions in the file, that is, ordinal number of the next character shown, in octal.

Notice that there is a character after each line, with octal value 012. This is the ASCII newline character. Other characters associated with some terminal control operation include backspace (octal value 010, printed as \b), tab (011, \t), and carriage return (015, \r).

$ stty -tabs

causes tabs to be replaced by spaces when printed on your terminal.

cat normally saves up or buffers its output to write in large chunks for efficiency, but cat -u "unbuffers" the output, so it is printed immediately as it is read:

```
$ cat
123
456
ctl-d
123
456
$ cat -u
123
123
456
456
ctl-d
$
```

cat receives each line when you press RETURN; without buffering, it prints the data as it is received.

Type some characters and then a ctl-d rather than a RETURN:

$ cat -u

123ctl-d123

Now type a second ctl-d, with no other characters:

$ cat -u

123ctl- d 123ctl-d$

The shell responds with a prompt, because cat read no characters, decided that meant end of file, and stopped, ctl-d sends whatever you have typed to the program that is reading from the terminal.

2.2 What's in a file?

The format of a file is determined by the programs that use it; there is a wide variety of file types. But file types are not determined by the file system, the kernel can't tell you the type of a file: it doesn't know it. The **file command** makes an educated guess:

```
$ file /bin   /bin/ed    /usr/src/cmd/ed . c    /usr/man/manl/ed . 1
/bin: directory
/bin/ed: pure executable
/usr/src/cmd/ed . c : c program text
/usr/man/manl/ed . 1 : roff, nroff, or eqn input text
$
```

These are four fairly typical files, all related to the editor: the directory in which it resides (/bin), the "binary" or runnable program itself (/bin/ed), the "source" or C statements that define the program (/usr/src/cmd/ed . c) and the manual page (/usr/man/manl/ed. 1).

To determine the types, file didn't pay attention to the names (although it could have), because naming conventions are just conventions, and thus not perfectly reliable. For example, files suffixed .c are almost always C source, but there is nothing to prevent you from creating a . c file with arbitrary contents.

Instead, file reads the first few hundred bytes of a file and looks for clues to the file type. Sometimes the clues are obvious. A runnable program is marked by a binary "magic number" at its beginning, od with no options dumps the file in 16-bit, or 2-byte, words and makes the magic number visible:

```
$ od /bin/ed
0000000    000410    025000    000462    011444    000000    000000    000000
000001
0000020    170011    016600    000002    005060    177776    010600    162706
000004
0000040    016616    000004    005720    010066    000002    005720    001376
020076
$
```

The octal value 410 marks a pure executable program, one for which the executing code may be shared by several processes. (Specific magic numbers are system dependent.) The bit pattern represented by 410 is not ASCII text, so this value could not be created inadvertently by a program like an editor. But you could certainly create such a file by running a program of your own, and the system understands the convention that such files are program binaries.

For text files, the clues may be deeper in the file, so file looks for words like #include to identify C source, or lines beginning with a period to identify nroff or troff input.

2.3 Directories and filenames

Each running program, that is, each process, has a current directory , and all filenames are implicitly assumed to start with the name of that directory, unless they begin directly with a slash. Your login shell, and **ls**, therefore have a current directory.

The command pwd (print working directory) identifies the current directory:

```
$ pwd
/usr/you
$
```

1. Directory related commands:

1. *Pwd (Print Working Directory)*

Name: pwd

Purpose: It displays the name of current working directory(where the user is working)

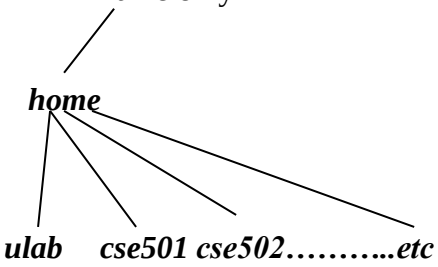
in a directory structure. *Pwd* always reports the full path of your current directory.

Examples & Description:

```
$pwd
/home/cse501
```

- The pwd command displayed a path name */home/cse501 (i.e)* the user is working with the directory *cse501* which is a subdirectory of *home.* Again *home* is a subdirectory of *root.*
- Generally all the users of UNIX will be placed under either *home* or *usr* directory

root By default the system places each user under their login name only



2. mkdir(Make directory)

Name: mkdir

Purpose: This command is used to create a new directory.

Syntax: *mkdir [option]* directory

Options:

Option	Description
-m mode	Set permission mode
-p	No error if existing, make parent directories as needed.
-v	Print a message for each created directory

Examples & Description:

$mkdir d1 **(**here *mkdir* creates a directory *d1* under the directory cse501)

$mkdir d2 d3 d4 **(** With a single *mkdir* command we can create more no.of directories.)

$mkdir d5 d5/d6 d5/d7 **(**We can create a directory tree with one *mkdir* command.)

The above command creates directory *d5* under *d5* it creates directories *d6, d7*

While creating the directory trees the order should be followed. i.e first parent has to be created then children

Examples & Description:

-p(creates Parent): this option creates parent directory as well as sub directories.

```
$mkdir –p x/y/z
```

creates first the parent directory ***x*** **,** inside ***x*** it creates ***y*** inside ***y*** it creates ***z***

-m mode: Each directory has permission modes(rwxrwxrwx). When a directory is created the default permissions are assigned as rwxrwxr_x by the kernel.

- The user can set his own permissions while creating the directory using –m mode option.
  Ex:

```
$mkdir –m 700 di
$ls –l
d rwx------ 0 cse501 cse501 512 Nov 19:14:17: di
```

The directory *di* has all permissions(read,write,execute) for owner. No permissions for group and other users.

3. Name: rmdir(Remove directory)

Name: rmdir

Purpose: Deletes a directory.

Syntax: rmdir [OPTION(s)] DIRECTORIE(S)

Option	Description
-p	--parents Remove DIRECTORY and its ancestors. E.g., `rmdir -p a/b/c' is similar to `rmdir a/b/c a/b a'.

Examples & Description:

$rmdir d1 (It removes directory *d1*.)
$rmdir d3,d2 (It removes directory *d2, d3* at a time.)
$rmdir d5/d6 d5/d7 d5 (It removes *d6 d7* which are subdirectories of *d5* and also removes *d5* finally).

Note1: We can't delete a directory unless it is empty. So executing the above command in the following way displays error message.

```
$rmdir d5 d5/d6 d5/d7
rmdir:d5: directory not empty
```

4.Name: cd (change directory)

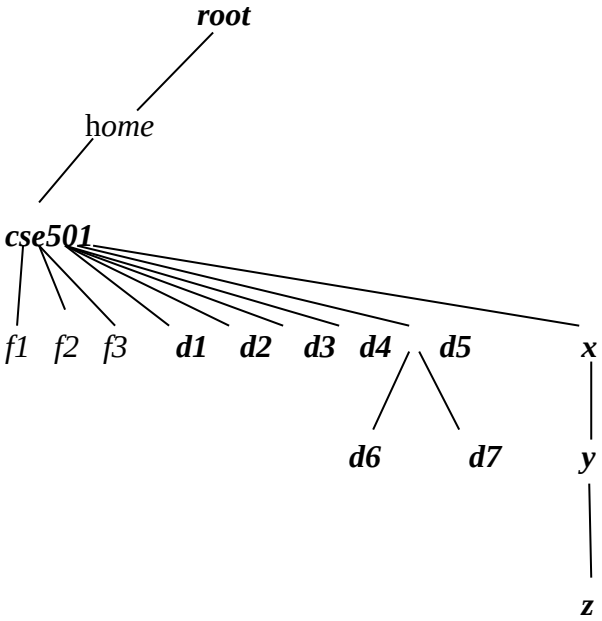
Name: cd

Purpose: cd is a command commonly used to change the directory

Syntax: cd pathname

There are no options for cd command.

Examples: consider the below directory structure for the following examples.



```
$pwd
/home/cse501
```

Here the user is working in *cse501* directory. Suppose the user wants to move to directory *d1* then cd command is used as

```
$cd d1
$pwd
/home/cse501/d1
```

From ***d1*** if the user wants to move to ***x*** then first the user has move to its parent(***cse501***) from there move to ***x***.

```
$cd ../x
$pwd
/home/cse501/x
```

cd Without arguments:

previously with whatever directory may be we are working, a simple ***cd*** command without arguments moves directly to the directory where the user originally logged onto.

```
$pwd
/home/cse501/x/y/z
$cd
$pwd
/home/cse501
```

2.du(Disk Usage):-

The command **du (disc usage)** was written to tell how much disc space is consumed by the files in a directory, including all its subdirectories.

Syntax:- du [options] [file(s)]

Examples:

ex: The 'du' command without any option (or) file name gives the output as:

```
$du
4    ./d1
2    ./d1/d2
7    ./d2
2    ./d2/d3
3    ./d2/d4
```

Here the output is for every sub-directory of current working directory the usage of disk in terms of BLOCKS is displayed. It also gives summary report for the usage of disk space for all directories under any one directory.

ex:- We can get the disk usage for a specified directory as follows

```
$du /home/sales/tml
10      /home/sales/tml/form
12      /home/sales/tml/data
40      /home/sales/tml
```

here it gives the usage of each subdirectory under a specified directory & it also gives the summary report for the disk usage of a specified directory.

Options:-

- -s :For showing only the summary report of the required directory(if we mentioned).

otherwise it gives the usage of a whole root directory.

```
$du  -s  /home/sales/tml
 40       /home/sales/tml
```

- -a : It gives the disk usage for files also.(x1,x2,x3 are files)

```
$du  -a
2      ./ d1/x1
3      ./d1/x2
5      ./d1
3      ./d2/x3
3      ./d2
16      /
```

- -b : It shows the disk usage in bytes rather than blocks.

```
$du  -b  /home/tm
11514      /home/tm/d1
12820      /home/tm/d2
75190      /home/tm
```

Here we get the disk usage in terms of bytes for sub directories d1&d2 and so on & finally for

Despite their fundamental properties inside the kernel, directories sit in the file system as ordinary files. They can be read as ordinary files. But they can't be created or written as ordinary files — to preserve its sanity and the users' files, the kernel reserves to itself all control over the contents of directories.

The time has come to look at the bytes in a directory:

```
$ od -cb .
0000000    4    ;    .   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0
           064  073  056 000  000  000  000  000  000  000  000  000  000  000 000  000
0000020    273  (    .   .   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0
           273 050 056 056 000  000  000  000  000  000  000  000  000  000  000  000
0000040    252  d    1   d   2    \0   \0   \0   \0   \0   \0  \0
           252 144  162 144 143  000  000 000 000 000 000 000
0000060    230  =    j   \0   \0   \0   \0   \0   \0   \0   \0   \0   \0
           230  075 152  000 000 000 000 000 000 000 000 000 000
0000100
$
```

See the filenames buried in there? The directory format is a combination of binary and textual data. A directory consists of 16-byte chunks, the last 14 bytes of which hold the filename, padded with ASCII NUL's (which have value 0) and the first two of which tell the system where the administrative information for the file resides — we'll come back to that. Every directory begins with the two entries ("dot") and .' ("dot-dot").

2.4 Permissions

There is a special user on every UNIX system, called the super-user , who can read or modify any file on the system. The special login name root carries super-user privileges; it is used by system administrators when they do system maintenance. There is also a command called **su** that grants super-user status if you know the root password. Thus anyone who knows the super-user password can read your files,.

If you need more privacy, you can change the data in a file so that even the super-user cannot read (or at least understand) it, using the **crypt** command (crypt(l)). Of course, even crypt isn't perfectly secure. A super-user can change the **crypt** command itself, and there are cryptographic attacks on the crypt algorithm.

The system actually recognizes you by a number, called your user-id, or uid. In fact different login-id's may have the same uid, making them indistinguishable to the system, although that is relatively rare and perhaps undesirable for security reasons. Besides a *uid,* you are assigned a group identification, or *groupid,*which places you in a class of users.

On many systems, all ordinary users are placed in a single group called *other*, but your system may be different. The file system, and therefore the UNIX system in general, determines what you can do by the permissions granted to your uid and group-id.

Owners, Groups, and Permissions:

File ownership is an important component of UNIX that provides a secure method for storing files. Every file in UNIX has the following attributes:

***Owner permissions*:** The owner's permissions determine what actions the owner of the file can perform on the file.

Group permissions: The group's permissions determine what actions a user, who is a member of the group that a file belongs to, can perform on the file.

Other permissions *:*The permissions for others indicate what action all other users can perform on the file.

The file **/etc/passwd** is the password file ; it contains all the login information about each user. You can discover your uid and group-id, as does the system, by looking up your name in **/etc/passwd**:

$ grep user1/etc/passwd

user1 : gkmbCTrJ 0 4C0M : 604 : 1 : Sonia,RBA,S/W: /usr/you:/bin/bash

$

The fields in the password file are separated by colons and are laid out like this

login-id : encrypted-password : uid : group-id : miscellany : login-directory : shell

login name: Unique name assigned to each user, consists up to 8 characters and is case sensitive

Password: The password is present in the encrypted form, consists up to 13 characters,

UID: The unique user ID number, that represents the user in OS.

GID : The unique group ID number ,assigned to a group, for sharing accounts of users belonging to same group.

comment : A miscellaneous field also referred as GECOS/GCOS field used for specifying additional user information. It usually contains full name of user, office location, telephone number or any other number**.**

Home directory: Directory into which a user is placed by default when he is logged in**.(*ex/home/cse501)***

login program : This field contains path name of the program (of the shell) that will run for that user after login.(***bin/bash.)***

The file is ordinary text file.

Note that your password appears here in the second field, but only in an encrypted form. Anybody can read the password file (you just did), so if your password itself were there, anyone would be able to use it to masquerade as you. When you give your password to login,

it encrypts it and compares the result against the encrypted password in /etc/passwd. If they agree, it lets you log in.

For example, if your password is *ka-boom*, it might be encrypted as *gkmbCTrJ04COM*, but given the latter, there's no easy way to get back to the original.

The kernel decided that you should be allowed to read /etc/passwd by looking at the permissions associated with the file. There are three kinds of permissions for each file: read (i.e., examine its contents), write (i.e., change its contents), and execute (i.e., run it as a program).

Furthermore, different permissions can apply to different people. As file owner, you have one set of read, write and execute permissions. Your "group" has a separate set. Everyone else has a third set.

The -l option of ls prints the permissions information, among other things:

$ ls -1 /etc/passwd
-rw-r—r-- 1 root 5115 Aug 30 10:40 /etc/passwd
$ ls -lg /etc/passwd
-rw-r—r-- 1 adm 5115 Aug 30 10:40 /etc/passwd
$

These two lines may be collectively interpreted as: /etc/passwd is owned by login-id *root*, group *adm*, is 5115 bytes long, was last modified on August 30 at 10:40 AM, and has one link .

The string -rw-r--r-- is how **ls** represents the permissions on the file.

The first - indicates that it is an ordinary file. If it were a directory, there would be a **d** there.

The next three characters encode the file owner's (based on uid) read, write and execute permissions, rw- means that root (the owner) may read or write, but not execute the file. An executable file would have an x instead of a dash.

The next three characters (r--) encode group permissions, in this case that people in group adm, presumably the system administrators, can read the file but not write or execute it. The next three (also r--) define the permissions for everyone else — the rest of the users on the system.

The file **/etc/group** encodes group names and group-id's, and defines which users are in which groups, **/etc/passwd** identifies only your login group; the **newgrp** command changes your group permissions to another group.

The program to change passwords is called passwd; you will probably find it in /bin:

$ ls -1 /bin/passwd
-rwsr-xr-x 1 root 8454 Jan 4 1983 /bin/passwd
$

(Note that /etc/passwd is the text file containing the login information, while /bin/passwd, in a different directory, is a file containing an executable program that lets you change the password information.)

Directory permissions operate a little differently, but the basic idea is the
same.

$ ls –ld .
drwxrwxr-x 3 you 80 Sep 27 06:11 .
$

The **-d** option of **ls** asks it to tell you about the directory itself, rather than its contents, and the leading d in the output signifies that '.' is indeed a directory.

File Permissions

Read: If a user has *read* permissions, that person can view the contents of a file ex: **cat**

Write: A user with *write* permissions can change the contents of a file ex: open with **vi** editor to modify data.

Execute: A user with *execute* permissions can run a file as a program.

Directory Permissions:

Execute(x): The **x** bit on a directory grants access to the directory. The ***cd d1***command is executed successfully if the directory ***d1*** has execute permission. Otherwise the user can't access the directory. The read and write permissions have no effect if the access bit is not set.

Read(r): The read(**r)** permission on a directory enables users to view file names and directory names in that i.e use the ***ls*** command to view files and their attributes that are located in the directory.

Write(w): The write(**w**) permission on a directory is the permission to watch out for because it lets a user to add and also remove files from the directory. $***cp f1 f2 d1*** it copies **f1, f2** files to directory **d1** when **d1** has write permission otherwise copying or removing files from d1 is not possible.

A directory that grants a user only execute permission will not enable the user to view the contents of the directory or add or delete any files from the directory, but it will let the user run executable files located in the directory.

If a directory is writable, however, people can remove files in it regardless of the permissions on the files themselves.

$ cd
$ date >temp
$ chmod -w .
$ ls -ld .
dr-xr-xr-x 3 you
$ rm temp
rm: temp not removed
$ chmod 775 .
$ ls -ld .
drwxrwxr-x 3 you
$ rm temp *file is removed*
$

2.5 Inodes

A file has several components: a name, contents, and administrative information such as permissions and modification times. The administrative information is stored in the *inode* along with essential system data such as how long it is, where on the disc the contents of the file are stored, and so on.

There are three times in the inode: the time that the contents of the file were last modified (written); the time that the file was last used (read or executed); and the time that the *inode* itself was last changed, for example to set the permissions.

```
$ date
Tue Sep 27 12:07:24 EDT 1983
$ date >jiink
$ ls -l junk
-rw-rw-rw- 1 you 29 Sep 27 12:07 junk
```

```
$ ls -lu junk
-rw-rw-rw- 1 you 29 Sep 27 06:11 junk

$ ls -1c junk
-rw-rw-rw- 1 you 29 Sep 27 12:07 junk
$
```

Changing the contents of a file does not affect its usage time, as reported by **ls -lu**, and changing the permissions affects only the inode change time, as reported by **ls -1c**.

```
$ chmod 444 junk
$ ls -lu junk
-r--r--r-- 1 you 29 Sep 27 06:11 junk
$ ls -1c junk
-r--r--r-- 1 you 29 Sep 27 12:11 junk
$ chmod 666 junk
$
```

The **-t** option to **ls**, which sorts the files according to time, by default that of last modification, can be combined with -c or -u to report the order in which inodes were changed or files were read:

```
$ ls d1
d2
d3
$ ls -lut
total 2
drwxrwxrwx 4 you 64 Sep 27 12:11 d1
-rw-rw-rw- 1 you 29 Sep 27 06:11 junk
$
```

All the directory hierarchy does is provide convenient names for files. The system's internal name for a file is its i-number : the number of the *inode* holding the file's information. **ls -i** reports the i-number in decimal:

```
$ date >x
$ ls -i
15768 junk
15274 d1
15852 x
$
```

It is the *i-number* that is stored in the first two bytes of a directory, before the name, **od -d** will dump the data in decimal by byte pairs rather than octal by bytes and thus make the i-number visible.

```
$ od -d .
0000000 15156 00046 00000 00000 00000 00000 00000 00000
0000020 10427 11822 00000 00000 00000 00000 00000 00000
0000040 15274 25970 26979 25968 00115 00000 00000 00000
0000060 15768 30058 27502 00000 00000 00000 00000 00000
0000100 15852 00120 00000 00000 00000 00000 00000 00000
0000120
$
```

The first two bytes in each directory entry are the only connection between the name of a file and its contents. A filename in a directory is therefore called a link , because it links a name in the directory hierarchy to the inode, and hence to the data. The same i-number can appear in more than one directory. The **rm** command does not actually remove inodes; it removes directory entries or links. Only when the last link to a file disappears does the system remove the
inode, and hence the file itself. If the i-number in a directory entry is zero, it means that the link has been removed, but not necessarily the contents of the file — there may still be a link somewhere else. You can verify that the i-number goes to zero by removing the file:

The ***ln*** command makes a link to an existing file, with the syntax

```
$ ln old-file new-file
```

The purpose of a link is to give two names to the same file, often so it can appear in two different directories.

```
$ ln junk lnk
```

```
$ ls -li
total 3
15768 -rw-rw-rw- 2 you 29 Sep 27 12:07 junk
15768 -rw-rw-rw- 2 you 29 Sep 27 12:07 lnk
15274 drwxrwxrwx 4 you 64 Sep 27 09:34 d1
$
```

```
$ echo
one>junk
$cat junk
  one
$cat lnk
   one
```

The integer printed between the permissions and the owner is the number of links to the file.
When you change a file, access to the file by any of its names will reveal the changes, since all the links point to the same file.

```
$ ls -1
total 3
-rw-rw-rw- 2 you 2 Sep 27 12:37 junk
-rw-rw-rw- 2 you 2 Sep 27 12:37 linktojunk
drwxrwxrwx 4 you  64 Sep 27 09:34 d1
$ rm $ is -1
total 2
-rw-rw-rw- 1 you 2 Sep 27 12:37 junk
drwxrwxrwx 4 you 64 Sep 27 09:34 d1
```

After *lnk* is removed the link count goes back to one. Removing a file just breaks a link; the file remains until the last link isremoved. once the last link to a file is gone, the data is irretrievable

cp makes copies of files:

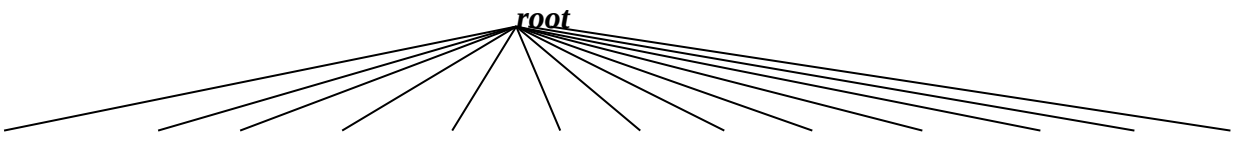
```
$ cp junk cnk
$ ls -li
total 3
15850 -rw-rw-rw- 1 you 2 Sep 27 13:13 cnk
15768 -rw-rw-rw- 1 you 2 Sep 27 12:37 junk
15274 drwxrwxrwx 4 you 64 Sep 27 09:34 d1
$
```

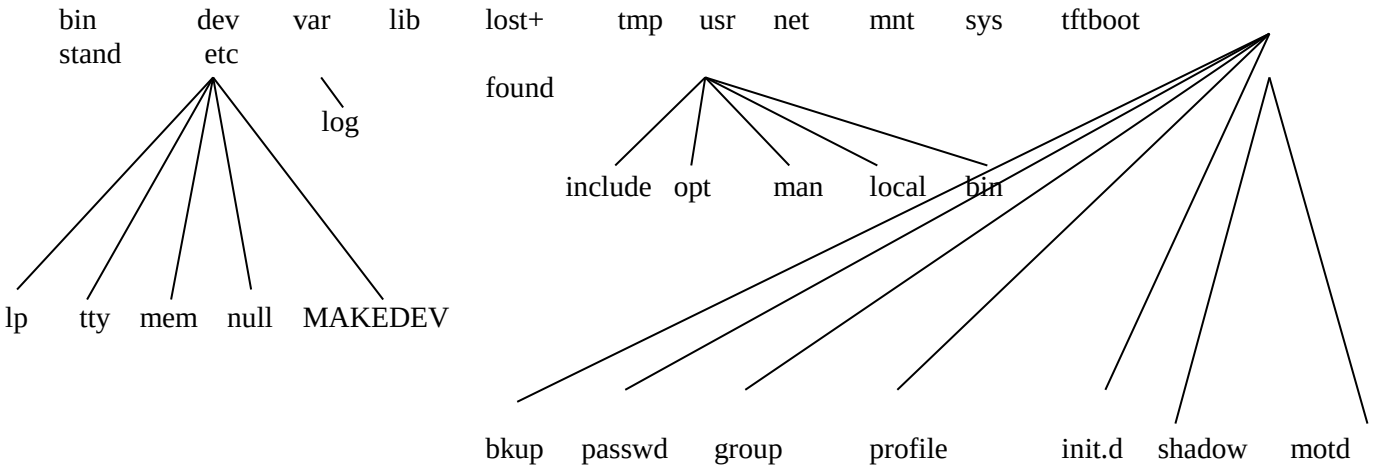
The i-numbers of junk and cnk are different, because they are different files, even though they currently have the same contents. Changing the copy of a file doesn't change the original, and removing the copy has no effect on the original.
There is one more common command for manipulating files: ***mv*** moves or renames files, simply by rearranging the links. Its syntax is the same as ***cp*** and ***ln***:

```
$ mv junk snk
$ ls -li
total 2
15274 drwxrwxrwx 4 you 64 Sep 27 09:34 d1
15768 -rw-rw-rw- 1 you 29 Sep 27 13:16 snk
```

There are 8 attributes of a file in inode table:
1.File Type : Type of a file(regular,directory, device, symbolic link, named pipe, socket etc..)
2. File Access Permissions: read(r),write(w),execute(x) permission for 3 users like owner, group, others
3. No.of links: Total no.of links of a file(created with *ln* command)
4. Owner of file
5.Group of file
6.File access time
7.file modification time
8. Size of file.

2.6 The directory hierarchy





The UNIX OS uses a hierarchically organized directories to keep track of the files, referred to as the directory hierarchy or directory tree. In this directory tree, there is a single **root** directory also referred as the parent directory and all other directories are subdirectories of the root directory. Every directory includes the files and some other directories.

/unix is the program for the UNIX kernel itself: when the system starts, /unix is read from disc into memory and started. Actually, the process occurs in two steps: first the file /boot is read; it then reads in /unix.

root Directory:-The root directory and all other sub directories are collectively called as **the file system**. The name of the root directory is the name of the file system.
The files presented in **root** directory are :
bin dev etc lib lost+found usr tmp net mnt sys var etc…

/bin Directory:- UNIX uses the bin directory to store the executable files. It also consists of most commonly used UNIX commands and tools like compilers, assemblers, editors etc.,

/dev Directory :-UNIX treats devices as if they are files. The files present in device directories are referred to as device files.
The files presented in the dev directory are as follows
lp file **tty** file **null** file **mem** file **MAKE DEV** file
/dev/lp: This file represents the printer in the terminal action or request mode on the printer must go through the /dev/lp file.
/dev/tty:-tty file represents terminal device files. It monitors the process of user's login shell.
/dev/null:- When you do not want to display the output on the screen, then the output is sent to this null file. This file also referred to as bit-bucket.
/dev/mem:- This file represents memory.
/dev/MAKE DEV:- It is used to create a device file for any device you specify.

3.The etc directory:-The programs used for system maintenance or system administration are stored in the ***etc*** directory.
bkup file passwd file group file profile file init.d file shadow file motd file
/etc/bkup:- this /etc/bkup file is used to backup and restore the system files.

/etc/group file:-The group file is used to store all the groups in the system and the list of users in each group and their passwords.
The following fields are existed in the **/etc/group** file.

group name :-unique name assigned to each group, consist up to 8 characters and is case sensitive*password* :-passwords present in an encrypted form for the group. This field is usually empty.***(ex:X123ERFD)***

GID :-the unique group ID number, that represents the group in the OS.***(ex:100)***

User-list :-An optional list of users separated by commas who are authorized to access files in that group.***(ex:cse501..)***

A sample line of the **/etc/group file** is shown below.
Student : X123ERFD : 100 : cse501, cse502,cse503,cse504

/etc/ init.d :- The */etc/init.id* file is used by the system when it changes the system state(i.e from single user mode to multiuser mode or shutdown state mode or shutdown and reboot mode or system administration mode or vice versa) also called as run levels.
/etc/**motd (**method of day):- This file usually contains information sent to the user by the system administrator(regarding shutdown of the system ,repair &maintenance of the system or any other information of the system he feels important for users to know).This file is displayed every time the users logs into the system
/etc/passwd:-The *password* file in the ***etc*** directory is the first file to grant access to users of UNIX OS.
This file is the systems user db, which lists login information for each user on the system.
A simple line of **/etc/passwd** file is.
Cse501 : AIEW23J : 201 : 100: Sonia,RBA,S/W : /user/cse501 :/bin/bash.

login name: Unique name assigned to each user, consists up to 8 characters and is case sensitive.**(ex: *Cse501)***
Password: The password is present in the encrypted form, consists up to 13 characters, which includes .,/, 0to9, A to Z. ***(*ex: AIEW23J)**
UID: The unique user ID number, that represents the user in OS.**(*ex:201)***
GID : The unique group ID number ,assigned to a group, for sharing accounts of users belonging to same group.**(*ex:100)***
comment : A miscellaneous field also referred as GECOS/GCOS field used for specifying additional user information. It usually contains full name of user, office location, telephone number or any other number**.**
(*Sonia,RBA,S/W)*
Home directory: Directory into which a user is placed by default when he is logged in**.(*ex/home/cse501)***
login program : This field contains path name of the program (of the shell) that will run for that user after login.(***bin/bash.)***
/etc/Profile:-This file contains shell configuration files of Bourne and korn shells.
This file is used by the system administrator for setting up any environment for a user who is different from the system default, and to make sure that all user have a common environment.
/etc/shadow:- This file contains the secured password information. The read access of this file is restricted only to the super user and all other users doesn't have capability to access this file.

- Each line contains entries of each user in fields delimited by colons.
- Fields present in the file are listed below.
  - *Login name*

- *Encrypted password.*
- *Last change of the password in days.*
- *Minimum number of days required b/n password changes.*
- *Maximum no. of days the password is valid.*
- *No. of days to issue a warning message about the expiry of password.etc..*

.

4.The Lib directory :-

The UNIX OS uses the *lib* directory to store functional and procedural librariesThe functional and procedural libraries in the '*lib'* directory includes library functions for c,c++ languages(such as stdio.h, iostream.h,math.h etc..),system calls(such as open(),chdir(),mount(),etc..) and I/O routines users can also add their own custom routines to the '*lib'* directory.

5.usr directory: Most UNIX systems place the users directory in their home directory, which is a subdirectory for the *usr* directory. It is used to store all user accounts and mail commands. The user directory was intended to be the central storage place for all user related commands.

The files present in the user directory are : *opt directory ,bin directory, local directory, man directory, include directory*

/usr/opt:- As the name implies it is an optional directory, the optional S/W packages are usually stored here.

/usr/bin :-This directory contains the various executable binary files and additional UNIX commands not included in the */bin* directory.

/usr/local:- The */user/local* directory is used to store local programs man pages(help) and libraries necessary for the UNIX OS.

/usr/man:-The source text for the various commands are stored in a special directory called as */usr/man* ..

/usr/ include *:-*This directory contains special files referred as header files (with an extension .h)

/usr/adm system administration: accounting info., etc.
/usr/dict dictionary (words) and support for spell(l)
/usr/games game programs
/usr/lib libraries for C, FORTRAN, etc.
/usr/mdec hardware diagnostics, bootstrap programs, etc.
/usr/src source code for utilities and libraries
/usr/src/cmd source for commands in /bin and /usr/bin
/usr/src/lib source code for subroutine libraries
/usr/spool working directories for communications programs
/**usr/spool/lpd** line printer temporary directory
/usr/spool /mail mail in-boxes
/usr/you your login directory

6. /lost +found:-. there are chances of problems or failures due to non-synchronization of the computer with the file system. These files can be recovered by the UNIX kernel from problems and are placed in a special directory named lost+found directory.

7*. / tmp:-* The *'tmp'* directory is a special directory that is available for storing temporary files.

The files stored in this directory are deleted as soon as the system is shutdown.

Some other directories present in the UNIX system of less interest are **: *net*** *directory,* ***tft boot*** *directory* **,** ***var*** *directory* **,** ***sys*** *directory,* ***mnt*** *directory,* ***stand*** *directory*

8./net directory :-.The word *'net'* stands for network is a directory in UNIX that contains files for accessing other computers on your N/W.

9./tft boot directory :- The *tft* (trivial file transfer protocol boot) directory is relatively new feature of UNIX, which contains versions of the kernel suitable for X windows system.

***10./var directory* :-**The *var* directory is used to hold files with constantly varying contents as the system runs. The *var* directory contains a directory named log, which is used to store messages generated by the system.

11./sys directory :-This directory contains the files indicating the system configuration.

***12.mnt directory* :-**.The *mnt* directory is said to be a common place to mount external media like hard disk, removable cartridges, driver, floppy disks, CD-ROMS and so on.

***13.stand directory* :-**It contains programs and configuration files used when booting the system.

2.7 THE file COMMAND—KNOWING THE FILE TYPE

Sometimes, apart from classifying Unix files as regular files, directories, device files and other files they are also classified as text files, executable files, and directories. This classification is based on the contents of the file. The file command is used to identify the type of the files on the basis of their contents. When this command is used, it reads either the header or first few hundreds of bytes of the file (given as an argument) and an educated guess is made on the type of the file. More often this guess is correct. One might argue that a filetype could be guessed or even concluded looking at the extension names. But Unix has nothing to do with extension names. This is because Unix puts no restriction on extensions in filenames. Certain category of files such as executables are recognized by the information stored on their headers—the information stored in the first-byte. This first byte information is known as the magic number. This magic number is consistent for similar file types between files and systems. The correlation between magic numbers and file types is contained in the file /etc/magic. For example the octal 410 is the magic number of executable files. These magic numbers can be verified by taking the octal dump of the relevant file.

For text files, the clues may not be available directly with the magic numbers. Rather, such clues will be available deeper in the file. For example, the clue for identifying text files could be, the use of a new line character at the end of every line. The presence of words such as #include indicate a C source file, lines beginning with a period may indicate nroff or troff input and so on. The study of following examples will give a better understanding of the usefulness of this command.

```
$file mgv
mgv: ASCII text
$file /bin
/bin: directory
$file mac.c
mac.c: ASCII C program text
$touch liju
$file liju
liju: empty
$cd /bin
```

$file csh
csh: symbolic link tcsh
$
In all the examples shown above, filenames have been given in the form of relative pathnames. Filenames can be given in the form of absolute pathnames also. Here it may be recalled that the listing command ls with the flag option F also gives an idea about the filetypes but in a limited way.

2.8 THE chmod COMMAND—CHANGING FILE PERMISSIONS

The chmod command is used to change the permissions of a file after its creation. Only the owner or the super user can change file permissions. The general syntax of this command is

$chmod assigenment_expression filename

The assignment expression holds the following information.

1. The information about the category of users {user –u, group –g, others –o, all –a}.
2. The information about granting or denial of the permission {the operators +, – and =}.
3. The information about the type of permission {read –r, write –w, execute –x}.

Although we generally consider only three types of users such as the owner, the group and others, a fourth category called all {a} that refers to all the three conventional categories is also considered. Further the + (plus) operator is used for granting the permission, the – (minus) operator is used for removing the permission and the = (equal to) operator is used for assigning absolute permission. Obviously the different permissions that are either granted or denied are the read permission (r), the write permission (w) and the execute permission (x).

Some examples that illustrate the use of the chmod command with reference to a file named sample with initial permissions of –rw–r– –r– – are given here.

$chmod u+x sample
$ls –l sample
–rwxr—r— 1 mgv csd 5180 Jan 07 12:06 sample

In this example u+x is the argument expression. The user has been granted the execution permission. As already mentioned above u stands here for user, x for execution and + for granting.

$chmod ugo+x sample;ls –l sample
–rwxr–xr–x 1 mgv csd 5180 Jan 07 12:06 sample
$

Execution permission has been granted (because of +) to all categories of users, that is, owner (because of u), group (because of g) and others (because of o). The expression ugo+x can also be written as a+x where a means all. Whenever the category is all, the category of user need not be mentioned explicitly. It is used by implication when the argument expression is just as +x. Thus the previous command can be rewritten in either of the following two ways.

$chmod a+x sample; ls –l sample

or

$chmod +x sample; ls –l sample

The chmod command can work on more than one file at a time as shown in the following example.

```
$chmod u+x sample1 sample2 sample3
$ls –l sample1 sample2 sample3
–rwxr––r––      1       mgv     csd     5180    Jan 07 12:06  sample1
–rwxr––r––      1       mgv     csd     6191    Jan 07 01:16  sample2
–rwxr––r––      1       mgv     csd     7101    Jan 07 02:26  sample3
$
```

More than one permission can be set using multiple argument expressions like u-x, go+x.

$chmod u-x, go+x sample

$ls –l sample

–rw-r-xr-x 1 mgv csd 5180 Jan 07 12:06 sample

$

Relative and Absolute Permissions Assignment

In all the examples discussed hitherto changes made were relative to the present settings. In other words, an expression like u+x sets the execute permissions to the user. It will not disturb other settings of either this or any other category. This type of permission assignment is called *relative permission* assignment.

The use of the = operator in the chmod expression assigns or grants only specified permissions and removes all other permissions. This type of granting permissions is called *absolute permission* assignment. Below is given an example where absolute assignment is made.

$chmod a=r sample; ls –l sample

–r––r––r–– 1 mgv csd 5180 Jan 07 12:06 sample

$

From the output of the above example, one may observe that all have been given read permissions after removing the permissions associated with the file earlier.

Permissions with Octal Numbers

File permissions can also be assigned using octal numbers. In this representation

1. 4_8 (as it is equivalent to 100_2) assigns read permission, 2_8 (as it is equivalent to 010_2) assigns write permission and 1_8 (as it is equivalent to 001_2) assigns the execute permission and so on.
2. Permission assignments made using octal numbers are always absolute assignments. In other words, octal numbers cannot be used for relative permissions assignment.

For example, a 6_8 $(110)_2$ assigns both read and write permissions and denies the execute permission 5_8 $(101)_2$ assigns read and execute permissions and denies write permission.

Because there are three categories of users, one has to use *three octal digits* in the expression field, as shown in the following example.

$chmod 644 sample; ls –l sample

–rw–r––r–– 1 mgv csd 5180 Jan 07 12:06 sample

$

The $chmod 761 sample is the octal notation equivalent of the following command.

$chmod u=rwx, g=rw, o=x sample

$

Permissions can be granted to all the files and sub-directories in a directory by using the recursive option (–R) with the chmod command. For this the argument must be the directory

name. For example, the execute permission to all category of users with respect to all files and directories under the current directory can be granted using the command given below.

```
$chmod –R a+x
$
```

2.9 THE chown COMMAND—CHANGING THE OWNER OF A FILE

As already mentioned, every file has a owner. When a file is created, the creator becomes the owner of the file. Only the owner can change the major attributes of a file (of course, the system administrator also can do it).

Sometimes it is necessary to change the ownership of a file. There are two ways in which the ownership can be changed—by copying the file in to the target user's directory, and by using the chown command.

For example, the file sample from the directory of hmk is *copied* to the home directory of someone else, say mgv. Then mgv becomes the new owner of the file sample. If, now, the oldfile and newfile are listed using the ls –l command, one sees that every detail will be same except the owner.

The copying method of changing the ownership has the following disadvantages:

- it creates an additional file and thus uses extra space.
- the new owner should have the knowledge about the permissions of the file.

Changing the owner of a file using the chown command is more simpler and direct method of changing the ownership. This command takes two arguments, login name of the new user and the name of the file. An example is given below.

```
$ls –l sample
–rwxr– –r–x 1   rajcsd    425  May 10 20:30   sample
$chown kumar sample ; ls –l sample
–rwxr– –r–x        1        kumar     csd     425       May 10 20:30 sample
$
```

It should be noted that the ownership once surrendered can not be reinstated. Also moving a file does not change the ownership. Further this command can use the –R option—the recursive option. When this option is used the ownership of all the files in the current directory are changed.

2.10 THE chgrp COMMAND—CHANGING THE group OF A FILE

In Unix, all files not only belong to an owner but also to a group. One may need to change the group of a file under certain circumstances such as when new groups are set up on a system or when files are copied to a new system. This is done by using the chgrp command. Only the owner of a file can change the group (of course, the system administrator also can do the same). Changing the group using the chgrp command is also straightforward.

This command also takes two arguments; the name of the new group and the name of the file. For example, $chgrp planning sample $

```
$chgrp                    planning                    sample
$
```

As shown above, the name of the new group must appear as the first argument and the name of the file has to appear as the second argument. The recursive option –R can also be used with this command. When used with the –R option, the group of all the files under the current directory is changed.

File Commands cp(Copying Files)

Name: cp

Purpose: The ***cp*** command copies files or directories from one place to another.

Syntax1 : cp [options] source dest

Here The *source* is the name of the file you want to copy, *dest* is the name of the new file.

Syntax2: cp [options] source(s) destdir

If you specify a *dest* that is a directory, **cp** will put a copy of the *source(s)* in the directory. The filename will be the same as the filename of the source file

Option	Descritpion
-i	Copies interactively
-p	Preserves modification time and access permissions.
-r	Recursive copying through subdirectories

Description and Examples:

1. The *cp* command copies a file or group of files into a directory or directory structure.
2. It creates an exact image of the source file on the disk with different name.

Examples: 1.

```
$cp  x.txt y.txt
```

This command copies the content of *x.txt* file into *y.txt*. Before copying if *y.txt* is not existed then cp command creates the new file *y.txt* and copies. If *y.txt* is already existed before copying and it has some data that data will be overwritten with the new data of *x.txt.*

2. With cp command we can copy more number of source files into a directory. If more than two inputs are given, ***cp*** treats the last argument as the destination and the other files as sources. This works only if the sources are files and the destination is a directory, as in the following

```
$cp  x.txt a b f1 f2 d1
```

This command places all 5 files *x.txt,a,b,f1,f2* files into a directory ***d1.***all these files are copied with the same names into the directory .

Options:

-i (Interactive Mode):

-i option of the cp command warns the user before overwriting the destination file.

```
$ cp -i a.txt c.txt
overwrite c.txt? (y/n)
```

If you choose y (yes), the file will be overwritten. If you choose n (no), the file c.txt isn't changed.

-p(preserves modification time):. With –p option the modification time of destination file is assigned to the modification time of source file.

```
$cp x.txt y.txt
$ls –l x.txt y.txt
_rw--r-- r--   1   cse501  cse501   40   May 25 15:46   x.txt
_rw--r-- r--   1   cse501  cse501   40   Nov 15  14:02  y.txt
```

By using –p option we can preserve the modification time.

```
$cp –p x.txt y.txt
$ls –l x.txt y.txt
_rw--r-- r--   1   cse501  cse501   40   May 25 15:46   x.txt
_rw--r-- r--   1   cse501  cse501   40   May 25 15:46   y.txt
```

-r(recursive copying): To copy all files in directrories as well as files in subdirectories and files in sub-sub directories use –r opiton.

$cp -r d1/* d2/d3

3. rm(Deleting Files)

Name: rm

Purpose: To delete files

Syntax: rm option(s) file(s)

Options:

-f	Forcily remove files that donot have write permission
-i	Interactively removes the files by confirming with the user before removing each file
-r	Recursively deletes the entire contents of the directory as well as subdirectories.

Description & Examples:

1. ***rm f1*** this command deleted file f1.

2.With single ***rm*** command we can delete more than one file.

$rm f2 f3 f4 deletes f2 ,f3,f4 files at a time**.**

1. **$rm d1/d2/f1** deletes a file **f1** which is in directory **d2**
2. **$ rm *** deletes all files in the current working directory.
3. **$rm f*** deletes all files ,for which the file name started with **f.**

-i(interactive): this option deletes files interactively. It confirms from user before deleting each file.

```
$rm –i a.txt b.txt c.doc z y
a.txt:? y
b.txt:? n
c.doc :? n
z:?y
a:?y
```

In the above while removing each file the command display **file:?** If user press **'y'** then the file is removed. Otherwise the file is not removed.

-f(forcibly):

-f option removes a file forcibly ,even though a file doesn't have write permission. **–f** removes the file without user confirmation.

-r(recursive): with –r option ***rm*** deletes all files, subdirectories, files under subdirectories, sub-sub directories, files under sub-sub directories.

Generally ***rm*** can't remove directories ,but with **–r** option it will delete subdirectories & files under it.

$rm –r * deletes current directory tree completely. All files in current directory, as well as all subdirectories and their files will be removed.

4. mv(Renaming Files)

Name: mv

Purpose: To rename or move the files

Syntax: mv option(s) file1 file2

Options:

-i	Interactively moves the files
-r	Recursively moves the files, subdirectories and files in sub

	directories.
-p	Preserves modification time.

Description & Examples:

1. **mv** command doesn't create a copy of the source file, but merely renames it to destination file. So **mv** renames files and directories
2. After moving the source file name is deleted, only the destination file name is remained.

Example:

1. **$mv f1 f2** it renames the file **f1** with **f2**.(deletes the f1 name from directory and places name f2).
2. **$mv f1 f1** it gives error message.
3. **$mv d1/f1 d2/f2** the command moves the file **f1** from directory **d1** to directory **d2** and names the file as **f2** after moving **f1** will be removed from **d1.**
4. **$mv d1/f1 d2/** it moves the file **f1** from directory **d1** to directory **d2**.and maintains the same name (**f1**). After moving **f1** will be removed from **d1**.
5. **$mv d1/f1** when the destination path is not specified the file is moved to current working directory.

-i(interactive): mv with –i option moves files interactively.

$mv –i f1 f2 If **f2** is already existed in current working directory the **–i** option displays the message as

mv:Overwrites f2? If user press '**y**' then **f2** data is erased and **f1** data is placed. If user press '**n'** then the **mv** command failed.

-p(preserves modification time):

When mv command is executed successfully, the modification time of a destination file is assigned to the new time. (i.e the time when mv is executed). But to preserve the modification time of original file we can use –p option.

```
$ls –l x.txt
rw--r-- r--    1    cse501  cse501    40    May 25 15:46   x.txt
$mv x.txt y.txt
$ls –l  y.txt
_rw--r-- r--   1    cse501  cse501    40    Nov 15  14:02  y.txt
```

```
$ls –l x.txt
rw--r-- r--    1    cse501  cse501    40    May 25 15:46   x.txt
$mv -p x.txt y.txt
$ls –l  y.txt
_rw--r-- r--   1    cse501  cse501    40    May 25 15:46   y.txt
```

-r(recursive):

To move all the files under a directory, and to move all the subdirectories, files under subdirectories, sub-sub directories and files under sub-sub directories use **–r** option.

$mv d3/* d2/ it moves all files of directory **d3** to a directory **d2** but subdirectories of **d3** are not moved.

$mv -r d3/* d2/ it moves all files ,subdirectories, sub-subdirectories, and files of d2 to d3.

UNIT-III
USING THE SHELL

Using the Shell-Command Line Structure-Met characters-Creating New Commands-Command Arguments and Parameters-Program Output as Arguments-Shell Variables- -More on I/O Redirection-Looping in Shell Programs.

3.1 Command line structure

The simplest command is a single word , usually naming a file for execution

```
$ who                                                    Execute the file /bin/who
you tty2 Sep 28 07:51
jpl tty4 Sep 28 08:32
$
```

A command usually ends with a newline, but a semicolon **;** is also a command terminator :

```
$ date;
Wed Sep 28 09:07:15 EDT 1983
$ date; who
Wed Sep 28 09:07:23 EDT 1983
you tty2 Sep 28 07:51
jpl tty4 Sep 28 08:32
$
```

Although semicolons can be used to terminate commands, as usual nothinghappens until you type RETURN. Notice that the shell only prints one prompt after multiple commands, but except for the prompt,

```
$ date; who
```

is identical to typing the two commands on different lines. In particular, **who**doesn't run until **date** has finished.Try sending the output of "**date ; who**" through a pipe:

```
$ date; who|wc
Wed Sep 28 09:08:48 EDT 1983
2 10 60
$
```

Connecting **who** and **wc** with a pipe forms a single command, called a pipeline , that runs after **date**. The precedence of | is higher than that of **;** as the shell parses your command line.

Parentheses can be used to group commands:

```
$ (date; who)
Wed Sep 28 09:11:09 EDT 1983
you tty2 Sep 28 07:51
jpl tty4 Sep 28 08:32

$ (date; who)|wc
3 16 89
$
```

The outputs of **date** and **who** are concatenated into a single stream that can besent down a pipe.

Data flowing through a pipe can be tapped and placed in a file (but not another pipe) with the t**ee** command,. One use is to save intermediate outputin a file:

```
$ (date; who)|tee save | wc
3 16 89                                                  Output from wc
$ cat save
Wed Sep 28 09:13:22 EDT 1983
you tty2 Sep 28 07:51
jpl tty4 Sep 28 08:32
$ wc<save
3 16 89
$
```

tee copies its input to the named file or files, as well as to its output, so **wc** receives the same data as if **tee** weren't in the pipeline.

Another command terminator is the ampersand **&**. It's exactly like thesemicolon or newline, except that it tells the shell not to wait for the command to complete. Typically, **&**. is used to run a long-running command "in thebackground" while you continue to type interactive commands:

$ long-running-command&.
5273 *Process-id of long-running-command*
$ *Prompt appears immediately*

Given the ability to group commands, there are some more interesting uses ofbackground processes. The command sleep waits the specified number ofseconds before exiting:

```
$ sleep 5
$                                          Five seconds pass before prompt
$ (sleep 5;date) & date
5278
Wed Sep 28 09:18:20 EDT 1983               Output from second date
$ Wed Sep 28 09:18:25 EDT 1983             Prompt appears, then date 5 sec. later
```

The background process starts but immediately sleeps; meanwhile, the second **date** command prints the current time and the shell prompts for a new command. Five seconds later, the `sleep` exits and the first `date` prints the newtime.

```
$ (sleep 300; echo Tea is ready) &          Tea is ready in 5 minutes
5291
$
```

since the precedence of **&** is higher than that of **;**

The &. terminator applies to commands, and since pipelines are commands you don't need parentheses to run pipelines in the background:

```
$ pr file | lpr&
```

arranges to print the file on the line printer without making you wait for thecommand to finish. Parenthesizing the pipeline has the same effect

```
$ (pr file |lpr) &                          Same as last example
```

Most programs accept arguments on the command line, such as file (anargument to pr) in the above example. Arguments are words, separated byblanks and tabs, that typically name files to be processed by the command, but they are strings that may be interpreted any way the program sees fit. Forexample, pr accepts names of files to print, echo echoes its arguments without interpretation, and grep's first argument specifies a text pattern to search for.

The various special characters interpreted by the shell, such as <, >, !, ;

and&, are not arguments to the programs the shell runs. They instead controlhow the shell runs them. For example,

```
$ echo Hello >junk
```

tells the shell to run **echo** with the single argument **Hello**, and place the outputin the file **junk**. The string **> junk** is not an argument to **echo**; it isinterpreted by the shell and never seen by echo. In fact, it need not be thelast string in the command:

```
$ > junk echo Hello
```

3.2 Metacharacters

The shell has a big family of meta characters to which special significance has been given.

All meta characters can be classified as shown ;

a. File name substitution: **? * […] [!...]**
b. I/O redirection : **> < >> << m> m>&n**
c. Process execution : **; () & && ||**
d. Quoting characters : **' ` " " **
e. Positional parameters: **$1 $2 $3 …$9**
f. Special characters: **$0 $* $@ $# #! $$ $_**

a.**Pattern matching or wild cards for File name substitution**: **? * […] [!...]**

wild card	significance
*	Matches any number of characters including none.
?	Matches a single character
[characters]	Matches one occurrence of any of the given characters
[abc]	Matches a single character either a,b,or c
[!characters]	Matches all characters except the set in class.
[!abc]	Matches a single character i.e not a ,b,c

Examples:

1. * Matches any number of characters including none.

$echo * displays all files in your current working directory

$ls exp* It displays all files, that are starting with **exp(exp,exp1, expab,exp.c …)**

$mv * ../d1 moves all files from current working directory to **d1** directory

1. ? Matches a single character.

$ls exp? Lists all files whose name is started with **exp** and after has one haracter.(expa,exp2..)

$ls ch?? Lists all files whose name is started with **ch** and after it has any two characters(ch1a, ch1b..)

$ls ?? Lists all files whose file name has only two characters.(ex,a1,po,…)

3.The character class **[ ]:** can have multiple characters inside this enclosure,but matching takes place for a single character in the class.

$ls exp[34] lists the file **exp3** or **exp 4 but** not **exp34.**

$ls exp[abc] lists the file **expa or expb or expc** but not **expab..**

4.Negating the character class[!]: placing the ! as 1st character of [] reverses the matching criteria. It matches all other characters except the one's in the class.

$[!a-zA-Z]* Matches all filenames where the first character is not alphabetic

$*.[!z] Matches all files having extensions except .z

$ls [!d-m]* Matches all files whose first character is anything other than an alphabet in range **d to m**

b.**I/O redirection meta characters**: **> < >> << 2> 2>&1**

> Input redirection

< output redirection

>> output redirection with append

2> Error redirection

2>&1 both error and output redirection

<< (Input inline redirection or **here document)**: If the character **<<** follows a command in the format **Command** << **word** the shell uses the lines that follows as the standard input for command, until a line that contains just **'word'** is found.

EXAMPLES:

```
$wc -l << END
>here's a line
>and another
>And yet another
>Now you can end
>END
4
```

Here the shell fed every line typed into the standard input of **wc** until it encounters the line containing just **END**. And displayed the no.of lines as output

```
$cat<<foo
>$HOME
>*****
>`date`
>foo
/home/cse501
*****
Sun jan 21 15:23:15 IST 2013
$
```

The shell performs parameter substitution for the redirected input data, executes back-quoted commands and recognizes the back slash character. However any other special characters such as *, " are ignored.

c. Process execution **; () & && ||**

; To run more than one command at same command line, we can run several commands each separated by **;**

```
$date;ls;pwd
Sun jan 21 15:23:15 IST 2013
f1 f2  m  x  d1
/home/cse501
```

1st **date** is executed and displayed its o/p on monitor.
2nd **ls,** next **pwd** is executed.

```
$date;ls;pwd > a.txt
Sun jan 21 15:23:15 IST 2013
f1 f2  m  x  d1
$cat a.txt
/home/cse501
```

date,ls command o/p is dipalyed on terminal. p**wd** command o/p is redirected to a **file a.txt.**

( ) Command can be grouped by placing them between parentheses, which causes them to be executed by a child shell(sub shell). The commands given in a group share the same standard i/p, standard o/p, standard error channels, and the group may be redirected and piped as if it is a single command. The first form ( ) ,causes the commands to be executed by a sub shell,.

```
$(date;ls;pwd) >p.txt
$cat p.txt
Sun jan 21 15:23:15 IST 2013
f1 f2  m  x  d1
/home/cse501
```

The 3 commands o/p is redirected to *p.txt* file

```
$(cd d1; pwd)
/home/cse501/d1
```

& this character causes a process execution in background. If a process takes more time to complete, then we can make that process as background process, and we can proceed with another job in the foreground.
A back ground job is indicated with &
$ sort f1>f2 & /* sort command is executed in back ground*/
&&,|| These characters can be used optionally execute a command depending on the success or failure of the previous command.

&& : **Command1 && Command2** , here if *command1* returns an exit status as zero(true), then only *command2* is executed. if *command1* returns an exit status as nonzero(false), then *command2* gets skipped.

```
$grep rose f1>f2 && echo rose is found
```

in this example if **grep** command find the pattern **rose** in file **f1**, then **echo** command is executed. If the **rose** is not found in **f1,** then the **grep** returns 1(false),**so echo** is not executed.

|| : Command1 || Command2 here if *command1* returns an exit status as zero(true), then *command2* is not executed. if *command1* returns an exit status as nonzero(false), then *command2* is executed.

```
$grep rose f1>f2 || echo rose is not found
```

in this example if **grep** command find the pattern **rose** in file **f1**, then **echo** command is not executed. If the **rose** is not found in **f1,** then the **grep** returns 1(false),**so echo** is executed.

d .Quote characters: The shell recognizes 4 different types of quote characters.
The single quote ‘
The double quote “
The Back slash \
The single quote ‘
1.The meaning of any meta character is lost if they enclosed in single quotes
2. Spaces are preserved in single quotes.

```
$echo one          two          three        four
one two three four
$echo 'one         two          three        four'
one          two          three        four
```

In first **echo** statement the shell removes extra white spaces from line and passes 4 arguments to **echo.**
Ins 2nd **echo** statement because the argument s are enclosed in single quotes all 4 words are considered as single argument and passed to **echo**. So the spaces are preserved when they are in single quote.

```
$a=5
$echo $a
   5
$echo '$a'
   $a
$echo '*'
   *
$echo  '> |; ( ) { } >> " &'
> |; ( ) { } >> " &
```

Generally **echo $a** prints the **a** value. But it is enclosed in single quote it printed **$a** in 2nd **echo** statement.
$echo * prints all files in pwd. But ***** is enclosed in single quote so it lost its meaning and considered as a normal character.
In last echo statement also different special characters are there they all lost their meaning because they were enclosed in single quotes.

```
 $echo 'how are you
 >today John '
 how are you
 today John
```

Even the enter key will be ignored by the shell if it is enclosed in quotes.

The double quote " : double quotes work similar to single quotes, except that they are not as restrictive. The single quote tell the shell to ignore all enclosed characters. Double quote tell the shell to ignore all enclosed characters except *Dollar sign ($), back quote(`), back slash (\)*

```
$a=5
$echo '$a'
   $a
$echo "$a"
    5
```

In first **echo** statement **$a** is in single quote so it is ignored and **$a** was displayed.
In 2nd **echo** statement **$a** is in double quotes,t he **$** is preserved in double quotes so **the** value of **a** is displayed

```
$ x=*
$echo '$x'
    $x
$echo "$x"
  *
$echo $x
 f1 f2  m  x  d1
```

In first **echo** statement **$x** is in single quote so it is ignored and **$x** was displayed.
In 2nd **echo** statement **$x** is in double quotes, the **$** is preserved in double quotes so **the** value of **x** is displayed. i.e ***,** but ***** is not evaluated in double quotes.
In 3rd **echo** statement **$x** is replaced with ***** at **echo, echo *** displays all files in pwd.

```
$echo "one         two          three        four"
 one         two          three        four
```

Spaces are preserved in double quotes.

```
 $echo "how are you
 >today John "
 how are you
 today John
```

Even the enter key will be ignored by the shell if it is enclosed in quotes.

The back slash: The back slash is equivalent to placing a single quote around a single character, with a few minor exceptions. Format is **\c** where **c** is a character that is to be quote. Any special meaning of character **c** is ignored.

```
$echo >
Syntax error: 'newline or:' expected
$echo \>
>
$
```

In 1st **echo >** expects a file name because **>** is a o/p redirection operator.
In **2nd echo** statement the special meaning of **> is** lost because of **** and is considered as normal character.

```
$echo \*
   *
$x=*
$echo \$x
      $x
$
```

Positional parameters: In shell programming, the data can be passed to the shell script at the command line. These are known as command line arguments or positional parameters. All arguments that are passed at command line are stored in a special shell variables. There are 9 such variables that capture and hold values given in command line. They are : **$1 ,$2,$3,…$9**

$1 holds the **1st argument, $2** holds **the 2nd** argument and so on

set command is used to assign the positional parameters.

```
$set Jan  Feb Mar  Apr
$echo $1 $2 $3 $4
Jan  Feb Mar  Apr
```

1st argument is assigned **to $1(jan), 2nd** argument t**o $2(feb)** and so on.

```
$cat l
 Give luck a little time
$set `cat l`
$echo $1 $2 $3 $4 $5
 Give luck a little time
```

Here positional parameters were assigned to the data of **file l.**

If more than 9 arguments are passed at command line, then enclose all 2 digit number with in { } to assign positional parameters.

```
$set a b c d e f g h I j k l
$echo $1 $2 $3 $4 $5 $6 $7 $8 $9 {$10} {$11} {$12}
a b c d e f g h I j k l
```

shift command : shift command shifts the arguments to its left,by default it shifts one position left.

```
$set a b c d e
$shift
$echo $1 $2 $3 $4
b c d e
```

The shift command shifts $2 value to left side, so the $1 lost its value, and $1=$2, $2=$3 and so on.

```
$set a b c d e f g
$shift  3
$echo $1 $2 $3
d e f
```

Special characters:

$1 $2	positional parameters.
$#	number of positional parameters supplied at command line
$*	List of positional parameters.
$@	Same as $*, except when the arguments are enclose in double quotes.
$$	Process ID of the current shell
$!	Process ID of the last background job
$0	Name of the last command being executed
$?	Exit status of the last execute command.

Examples

```
$set do u want credit or results
$echo $#
6
$echo $*
do u want credit or results
```

>

Shell Metacharacters

```
>     prog >file direct standard output to file
>>    prog >>file append standard output to file
<     prog <file take standard input from file
|     P1|p2 connect standard output of p1 to standard input of p2
<<str  here document: standard input follows, up to next str on a line by itself
*      match any string of zero or more characters in filenames
?      match any single character in filenames
[ccc]  match any single character from ccc in filenames;
            ranges like 0-9 or a-z are legal
;     command terminator: p1;p2 does p1,then p 2
&      like ; but doesn't wait for p1 to finish
'...'  run command(s) in ...; output replaces '...'
(...)  run command(s) in ... in a sub-shell
{...}  run command(s) in ... in current shell (rarely used)
$1, $2 etc .    $0...$9 replaced by arguments to shell file
$var      value of shell variable var
${var}     value of var; avoids confusion when concatenated with text;
\c       take character c literally, \newline discarded
'...'     take ... literally
'..."              take ... literally after $, and \ interpreted
#     if # starts word, rest of line is a comment
var-value assign to variable var
P1 && P2 run p1; if successful, run p2
P1||P2   run p1; if unsuccessful, run p2
```

3.3 Creating new commands

Given a sequence of commands that is to be repeated more than a few times, it would be convenient to make it into a "new" command with its own name, so you can use it like a regular command. To be specific, suppose you intend to count users frequently with the pipeline

```
$ who |wc -1
```

you want to make a new program `nu` to do that .

The first step is to create an ordinary file that contains `'who |wc -1'`.

```
 $ echo 'who | wc -1 ' >nu
```

Run the shell with its input coming from the file `nu` instead of the terminal:

```
$ who
you tty2 Sep 28 07:51
rhh tty4 Sep 28 10:02
moh tty5 Sep 28 09:38
ava tty6 Sep 28 10 : 17
$ cat nu
who |wc -l
$ sh<nu
4
$
```

The output is the same as it would have been if you had typed `who | wc -1` at the terminal.

Again like most other programs, the shell takes its input from a file if one is named as an argument;

```
$ sh nu
```

If a file is executable and if it contains text, then the shell assumes it to be a file of shell commands. Such a file is called a **shell file**. All you have to do is to make `nu` executable, once:

```
$ chmod +x nu
```

and thereafter you can invoke it with

```
$ nu
```

From now on, users of `nu` cannot tell, just by running it, that you implemented it in this easy way.

The way the shell actually runs `nu` is to create a new shell process exactly as if you had typed

```
$ sh nu
```

This child shell is called a sub-shell — a shell process invoked by your currentshell, **`sh nu`** is not the same as **`sh<nu`**, because its standard input is still connected to the terminal.

As it stands, nu works only if it's in your current directory (provided, ofcourse, that the current directory is in your PATH, which we will assume from now on). To make **`nu`** part of your repertoire regardless of what directory you're in, move it to your private bin directory, and add **`/usr/you/bin`** toyour search path:

```
$ pwd
/usr/you
$ mkdir bin                                   Make a bin if you haven't already
$ echo $PATH                                  Check PATH for sure
: /usr/you/bin : /bin : /usr/bin              Should look like this
$ mv nu bin                                   Install nu
$ nu
4                                             But it's found by the shell
$
```

3.4 Command arguments and parameters

Suppose we want to make a program called cx to change the mode of a fileto executable, so

```
$ cx nu
```

is a shorthand for

```
$ chmod +x nu
```

We already know almost enough to do this. We need a file called cx whose contents are

```
chmod +x filename
```

The only new thing we need to know is how to tell cx what the name of the file is, since it will be different each time cx is run. When the shell executes a file of commands, each occurrence of $1 isreplaced by the first argument, each $2 is replaced by the second argument, and so on through $9, So if the file **`cx`** contains

```
chmod +x $1
```

when the command

```
$ cx nu
```

is run, the sub-shell replaces "$1" by its first argument, "nu."Let's look at the whole sequence of operations:

```
$ echo 'chmod +x $1' >cx                      Create cx originally
$ sh cx cx                                    Make cx itself executable
$ echo `who|cut -f1 -d" "'>na                 Make a test program
$ na                                          Try it
$ mv cx  /usr/you/bin                         Install cx
$ mv na /usr/you/na                           Install cx
$ cx na
$ na
you
rhh
moh
ava
$
```

Notice that we said

```
$ sh cx cx
```

exactly as the shell would have automatically done if **`cx`** were already executable and we typed

```
$ cx cx
```

If you want to handle more than one argument, for example to makea program like **`cx`** handle several files at once then we can use command line arguments or positional parameters.

```
chmod +x $1 $2 $3 $4 $5 $6 $7 $8 $9
```

the effect is that only the arguments that were actually provided are passed to chmod by the sub-shell. The proper way to define cx, then, is

```
chmod +x $*
```

which works regardless of how many arguments are provided.With $* added to your repertoire, you can make some convenient shellfiles, such as 1c or m:
Create a command lc that counts the no.of lines of specified files.$ cd /usr/you/bin

```
$ cat 1c
wc -1 $*
$chmod +x lc
$mv lc /usr/you/bin
$lc f1 f2 f3 a b c
4   f1
5   f2
3   f3
10  a
1   b
4   c
$
```

For example, consider searching a personal telephone directory. If you have a file named *phone-book* that contains lines like

```
dial-a- joke 212-976-3838
dial -a -prayer 212-246-4200
dialsanta 212-976-3636
dow jones report 212-976-4141
```

then the grep command can be used to search it. (Your own lib directory is a good place to store such personal data bases.). Let's make a directory assistance program, which we'll call 411 in honor of the telephone directory assistance number where welive:

```
$ echo ' grep $* phone-book ' >411
$ cx 411
$ 411 joke
dial-a- joke 212-976-3838
$411 dial
dial-a- joke 212-976-3838
dial-a-prayer 212-246-4200
dialsanta 212-976-3636
$
```

3.5 Program output as arguments

The output of any program can be placed in a command line by enclosing the invocation in backquotes` `

```
$ echo At the tone the time will he 'date'.
At the tone the time will be Thu Sep 29 00:02: 15 EDT 1983.
$
```

A small change illustrates that **`...`** is interpreted inside double quotes **"…"**

```
$ echo "At the tone
> the time will be `date`."
At the tone
the time will be Thu Sep 29 00:03:07 EDT 1983.
$
```

As another example, suppose you want to send mail to a list of people whose login names are in the file **mailinglist.** A clumsy way to handle thisis to edit **mailinglist** into a suitable mail command and present it to the shell, but it's far easier to say

```
$ mail `cat mailinglist` <letter
```

This runs cat to produce the list of user names, and those become the arguments to mail. (When interpreting output in backquotes as arguments, theshell treats newlines as word separators, not command-line terminators; Backquotes are easy enough to usethat there's really no need for a separate mailing-list option to the mail command.
A slightly different approach is to convert the file mailinglist from just a list of names into a program that prints the list of names:

```
$ cat mailinglist New version
echo don whr ejs mb
$ cx mailinglist
$ mailinglist
```

```
don whr ejs mb
$
```
Now mailing the letter to the people on the list becomes
```
$ mail `mailinglist ` <letter
```
With the addition of one more program, it's even possible to modify the user list interactively. The program is called **pick:**
```
$ pick arguments ...
```
presents the arguments one at a time and waits after each for a response. The output of pick is those arguments selected by y (for "yes") responses; any other response causes the argument to be discarded. For example,
```
$ pr `pick *.c`|lpr
```
presents each filename that ends in .c; those selected are printed with **pr** and **lpr**.
```
 $ mail  `pick \ `mailinglist\ `` <letter
don? y
whr?
ejs?
mb? y
$
```
sends the letter to **don and mb**. Notice that there are nested backquotes; the back slashes prevent the interpretation of the inner during the parsing ofthe outer one.

3.6 Shell variables

SHELL VARIABLES: Variables are "words" that hold a *value*.

Shell variables are of two types: 1. User define variables 2. Environment or System variables.

1. Environment variables: system variables or environment variables are defined by the system itself. System variables are set either during the boot sequence or intermediately after login.

- The working environment, under which a user works, depends entirely upon the values of these variables. When a new shell is started by user, some of these variables are inherited by the sub shell from its parent.
- Environment variables are different in scope from simple shell variables. These are similar to global variables . they visible in the users total environment.
- Environment variables are: **HOME, PATH, USER or LOGNAME, MAIL, MAILCHECK, HISTSIZE, HISTFILE, HISTFILESIZE, TERM, PWD, CDPATH, PS1, PS2, SHELL, IFS, TZ.**
- The set statement displays a complete list of all variables. The *env* command statement shows only the environment variables.

```
$env
CDPATH=.:..:$HOME
HOME=/home/cse501
LOGNAME=cse501
MAIL=/var/mail/cse501
MAILCHECK=60
PAGER=/usr/bin/more
PATH=/bin:/usr/bin:/usr/dt/bin:/home/cse501/d1
PWD= home/cse501
PS1=$
SHELL=/bin/bash
TERM=ansi
…..
```

1.HOME: Indicates the home directory of the current user. When a user is login , UNIX places him in a directory called as home directory. A user's home directory is specified in the line pertaining to that user in */etc/passwd* file. This file contains a line for every user with 7 fields' per line.

cse501:xxx:208:50::/home/cse501/bin/sh the homedirectory is specified in the 6th field.

A user can change the value of home as **HOME=/home/John.**

2.LOGNAME: stores login name of user.

```
 $echo $LOGNAME
 /home/cse501
```

3.MAIL: the mail variable determines the path of the file where all incoming mail addressed to the user is stored.

Normally the mail variable holds **/usr/spool/mail/cse501 or /usr/spool/mail or /var/mail or /var/spool/mail.** The login shell searches this path every time when a user logs into the system. If the file contains any mail, the shell informs the user with a message **"you have a mail".**

```
$echo $MAIL
 /var/spool/mail
```

4. MAILCHECK: This variable stores the positive number in seconds that specifies how often the shell should check for the arrival of the mail in the path specified by the **MAIL** variable. The default value of **MAILCHECK v**ariable is **600** Seconds. If the **MAILCHECK v**ariable is **0 ,**the checks for mail every time a user logs into the system.

```
$echo $MAILCHECK
600
```

5.PATH: Indicates search path for commands. It is a colon-separated list of directories in which the shell looks for commands. It is a list of directories, that are to be searched to execute a command.

A common value is

```
$echo $PATH
/bin:/sbin:/usr/bin:/usr/sbin:/usr/local/bin:/usr/ucb
```

The user can add or remove the search path to the existing contents of the PATH variable.

```
PATH=$PATH:/home/cse501
```

After this statement, to execute a command, the shell searches your own directory also.

6.PS1: It contains the login shell prompt. The default value of the **PS1** variable is **'$'.**

```
$echo PS1
   $
```

User can change the shell prompt by changing the value of PS1

```
$PS1="UNIX"
UNIX mkdir d1
UNIX rmdir d1  /* here the shell prompt is changed from $ to UNIX*/
```

7. PS2: It contains the sub shell prompt. The default value of the **PS2** variable is **'>'.**

```
  $echo PS2
   >
```

User can change the shell prompt by changing the value of PS2

```
$PS2="OS"
OS for a in 1 2
OS do
OS echo $a
OS done
o/p:
1
2
```

8.IFS: Indicates the Internal Field Separator that is used by the parser for word splitting after expansion. **$IFS** is also used to split lines into words with the read built-in command. The default value is the string, **" \t\n ",** where **" "** is the space character, **\t** is the tab character, and **\n** is the newline character.

User can assign a character as a field separator value. **$IFS=:** after this statement the field separator is not space or tab but it is **:**

9.HZ: the **HZ(hertz)** variable stores the no.of clock interrupts per second. The default value of **HZ** is **100.**the value of **HZ** is stored in the /etc/default/login

```
$echo $HZ
100
```

10.SHELL: The **SHELL** variable stores the pathname of the shell in which the user is currently working. The default value of **SHELL** is **/bin/bash.** The system administrator sets the value of the **SHELL** variable when he creates an account for the user.

```
$echo $SHELL
/bin/bash
```

11.TERM : the **TERM** variable indicates the terminal type being used. Several programs like *vi* are terminal dependent and they require to know the type of the terminal you are using. If the **TERM** variable is not set correctly *vi* won't work properly and display will be faulty.

12. PWD: Indicates the current working directory as set by the *cd* command.

13.TZ(Time zone): It stores the time zone specification, used by commands like `date, at, batch.` To set the time appropriate to your time zone. The value stored in the TZ variable will be in the format XXXYZZZ where XXX stores the standard local time zone abbreviation like IST,EST,PST etc.. , Y is the difference b/n the local time zone and Green wich Mean Time(GMT) and ZZZ is the abbreviation for the summer daylight savings time zone.

```
$echo $TZ
IST5TDT
```

2.User defined variables: Variables are "words" that hold a value. The shell enables you to create, assign, and delete variables. Although the shell manages some variables, it is mostly up to the programmer to manage variables in shell scripts. By using variables, you are able to make your scripts flexible and maintainable.

The value of a variable is associated with the shell that creates it, and is not automatically passed to the shell's children.

```
$ x=Hello                          Create x
$ sh                               New shell
$ echo $x                          Newline only: x undefined in the sub-shell
$ ctl-d                            Leave this shell
$                                  Back in original shell
$ echo $x                          x still defined
Hello
$
```

This means that a shell file cannot change the value of a variable, because the shell file is run by a sub-shell:

```
$ echo 'x= "Good Bye"              Make a two-line shell file ...
>echo $x' >setx                    ...to set and print x
$ cat setx
x="Good Bye"
echo $x
$ echo $x
Hello                                  x is Hello in original shell
$ sh setx
Good Bye                           x is Good Bye in sub-shell...
$ echo $x
Hello                              ...but still Hello in this shell
$
```

The other way to set the value of a variable in a sub-shell is to assign to it explicitly on the command line before the command itself:

```
$ echo 'echo $x ' >ec
$ cx ec
$ echo $x
Hello                                  As before
$ ec
                                       x not set in sub-shell
$ x=Hi ec
Hi                                     Value of x passed to sub-shell
$
```

When you want to make the value of a variable accessible in sub-shells, thes hell's export command should be used.

```
$ x=Hello
$ export x
$ sh                                   New shell
$ echo $x
Hello                                   x known in sub-shell
$x="Good Bye "                         Change its value
$ echo $x
Good Bye
$ ctl-d                                Leave this shell
$                                      Back in original shell
$ echo $x
Hello                                   x still Hello
$
```

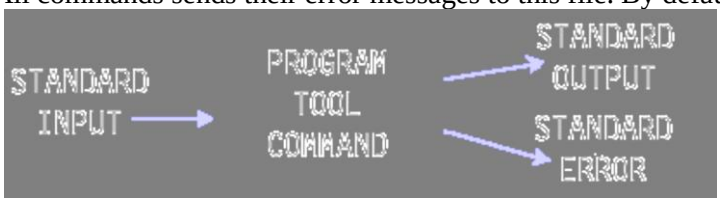
3.7 More on I/O redirection

In UNIX the files are divided into 3 types: standard input file, standard output file, standard error file

standard input file(stdin): From this file all commands accepts the input, by default commands are entered at the keyboard, so keyboard is a standard input file:

***standard output file(stdout*)**: all commands sends their output to this standard output file. By default it is monitor.

***standard error file(stderr*)**: All commands sends their error messages to this file. By default it is monitor.



Every file in Unix is associated with an unique number called as file descriptor value.

The file descriptor for standard input is **0 (zero)**

The file descriptor for standard output is **1**

The file descriptor for standard error is **2.**

***Input devices*:** The input devices in Unix are Keyboard ,file, pipe. i.e a command can accepts its i/p from Keyboard or, file or , pipe

***Out put devices*:** The output devices in Unix are monitor ,file, pipe. i.e a command can send its O/P to monitor or, file or , pipe.

Redirection: this is a process of reading input from a file instead of keyboard, pipe and writing output to file instead of monitor or pipe.

I/P redirection: If a command accepts its input from a file it is called as input redirection. and indicated with the symbol < It can be write as: **command < file**

```
$cat<f1  [Enter]
   foo
   bar
   fred
   barney
   dino
```

Here **cat<f1** or **cat f1** both are same. In these 2 cases **cat a**ccepts its input from file **f1.** And displays that on monitor

The **wc (word count)** command counts the number of bytes, word and lines in a file.

```
$ wc<f1 [Enter]
6       7       39    f1
```

wc<f1 here **wc** command accepts input from file **f1** and displays the no. of lines 6, words 7, characters 39 followed by the name of the file **wc** opened.

wc<f1 or wc f1 both are same**. wc f1** assumes that there is an i/p redirection operator between **wc and f1**

O/P redirection: If a command writes its output to a file it is called output redirection. And indicated with the symbol >

It can be write as:**command > file**

```
$ echo "Hello World!" > s [Enter]
$ cat s [Enter]
  Hello World!
```

Here the echo command written it's o/p to a file *s* ,because of o/p redirection.

2.$ ls > m [Enter] /* send the out put of **ls** command to a file called **m */**

3.

```
$ wc<f1>x[Enter]
$cat x
6       7       39
```

/* here the **wc f1** command writes its output to file **x. w**e can view that by **cat x*/**

The same command can be written as : **wc<f1>x or wc>x<f1 or >x<f1 wc** in these all commands the **wc** accepts its i/p from **f1** and writes its o/p to **x.**

When **<, >** appears in the same line **<(I**/P redirection) has the higher priority than **>(**O/P redirection).

Note: If the file mentioned at output redirection is already exists, it is overwritten.

In case you want to append to an existing file, then instead of the '**> '** operator you should use the **' >> '** operator. This would append to the file if it already exists, else it would create a new file by that name and then add the output to that newly created file.

```
$ echo "Hello World!" > s [Enter]
$ cat s [Enter]
   Hello World!
$wc  f1>s [Enter]
$cat s [Enter]
  6        7       39
```

In this example first the output of **echo** is redirected to a file called s.
Second time when **wc** command o/p is redirected to the same file **s** ,then the previous contents of s are overwritten. So we can find only **wc** o/p in file **s**. we can avoid this with **>>** symbol.

```
$ echo "Hello World!" > s [Enter]
$ cat s [Enter]
   Hello World!
$wc  f1>>s [Enter]
$cat s [Enter]
   Hello World!
  6        7       39
```

In this example first the output of **echo** is redirected to a file called s.
Second time when **wc** command o/p is redirected to the same file **s** using the operator **>>** , we can find both outputs **(echo, wc) in s** file.

Standard error devices are indicated with the FD 2. This is a device where all error messages are redirected. It may a terminal or a file.

```
$cat> y
This is pen
[ctrl-d]
$cat   z
This is rose
[ctrl-d]
```

```
$ls [Enter]
f1    m    s    x    y    z
$cat  a  [Enter]
cat: cannot open a: No such file or directory
```

In this example file **a** is not existed so it displayed error message on terminal.

```
$ls [Enter]
f1    m    s   x  y    z
$cat  y z a >f5  [Enter]
cat: cannot open a: No such file or directory
$ cat  f5  [Enter]
This is pen
This is rose
```

By using o/p redirection operator also the error is not redirected to a file **f5** . instead it was displayed on terminal. Only **x y** files o/p was redirected to **f5** file.

To redirect error messages to a file use O/P redirection operator with the file descriptor value of standard error device i.e 2. So error redirection is possible with (2>).

```
$ls [Enter]
f1    m    s   x  y    z
$cat  a  2> f6
```

In this example the error message regarding the file **a** is redirected to a file **f6**

```
$cat  y  z  a  >p   2>q [Enter]
$cat p [Enter]
This is pen
This is rose
$cat q [Enter]
cat: cannot open a: No such file or directory
```

In this example the output is redirected to a file **p** and the error message is redirected to a file **q**.

To send both standard output and standard error to the same file use the following format:

command > file 2>> file or **command > file 2>&1**

```
$cat  y  z  a  >k   2>&1 [Enter]
$cat k [Enter]
This is pen
This is rose
cat: cannot open a: No such file or directory
$
```

In this example the output , error messages two are redirected to a same **file k.** the following command also produces the same output:

Shell I/O Redirections

>file direct standard output to file
>>file append standard output to file
<file take standard input from file
P1|P2 connect standard output of program p1 to input of p2
^ obsolete synonym for !
n>file direct output from file descriptor n to file
n>>file append output from file descriptor n to file
n>&m merge output from file descriptor n with file descriptor m
n<&m merge input from file descriptor n with file descriptor m
<<s here document: take standard input until next *s* at beginning of a line;
<<\s here document with no substitution
<< 's' here document with no substitution

Pipes: A Pipe is a general mechanism by using which the output of one command is connected or redirected as the input to another command directly without using any temporary files.

To send the output of one command as input for another, the two commands must be joined using a **|** operator.

We can pipe or connect any no.of commands on the pipeline. The generalize form is

Command1|command2|command3|…..|commandn

In the above format

a. Command1 should support the o/p redirection. i.e it should produce some output ex: **ls,who,wc ..**
b. Command 2,command3,…command n-1 all these commands should support both i/p,o/p redirection. Ex: **cmp,cut,head, wc**…
c. Command n should support i/p redirection. **Ex:bc,wc,cut**…

The pipes, redirection operators several commands can be combined together to perform complex tasks

2. Write a command to display the no. of users that are logged in.

To get the no.of users that are currently logged in we don't have a direct command. But we can solve this using either redirection or piping.

```
$who
 root     console    Feb12:00
cse501    tty01      Feb 14:09
cse502    tty03      Feb 14:13
cse503    tty07      Feb 15:02
```

Assume this is the o/p of **who** command

```
$who|wc  -l
4
```

Here first **who** command writes its output to a pipe (|). Next **wc** command reads the input from pipe. (Which is the o/p **of who**) and counts the no. of lines from it.

2.**Write a command to display only login names of users who are currently logged n.**

```
$who|cut –d " "  -f 1
root
cse501
cse502
cse503
```

Here first **who** command writes its output to a pipe (|). Next **cut** command reads the input from pipe. (Which is the o/p **of who**) and cuts the first field(which is login name of users.)

3. **.Write a command to display login names, login time of users who are currently logged n.**

```
$who|cut –d " "  -f 1,3
root        Feb12:00
cse501      Feb 14:09
cse502      Feb 14:13
cse503      Feb 15:02
```

Here first **who** command writes its output to a pipe (|). Next **cut** command reads the input from pipe. (Which is the o/p **of who**) and cuts the first ,third fields(which is login name, login time of users.)

5. **Display no.of files and directories in current working directory .**

```
$ls|wc  -l
12
```

ls command displays all files in pwd to pipe. ***wc*** reads from pipe & counts the no.of lines from it.

7.**Display the files which have write permission for the group.**

```
$ls -l|grep "-----w" | cut -f9
```

8.Display Both modification and access time of files

```
$ls -l| tr -s ' '|cut -d" " f 6-9  >temp
$ls -lu| tr -s ' '|cut -d" " f 6-9|paste temp
```

`ls -l` displays long listing of files with modification time. **`ls -lu`** displays long listing of files with access time. **`tr`** command squeezes multiple spaces to single space. **`cut`** command cuts the field number 6 to 9 which is the time, file name ,**`paste`** command pastes both timings in single row.

3.8 Looping in shell programs

The shell is actually a programming language: it has variables, loops,decision-making, and so on. the shell's for statement is the only shell control-flow statement that you might commonly type at the terminal rather than potting in a file for later execution. The syntax is:

```
for var in list of words
do
commands
done
```

For example, a for statement to echo filenames one per line is just

```
$ for i in *
> do
>echo $i
> done
```

The "i" can be any shell variable, although i is traditional. Note that the variable's value is accessed by $i, but that the for loop refers to the variable as i. We used * to pick up all the files in the current directory, but any other list of arguments can be used. Normally you want to do something moreinteresting than merely printing filenames

```
$ ls ch2 .* | 5
ch2 . 1 ch2 . 2 ch2 . 3 ch2 . 4 ch2 .5
ch2 . 6 ch2 .7
$ for i in ch2 .*
> do
>echo $i:
>diff -b old/$i $i
>echo                                          Add a blank line for readability
> done | pr -h "diff `pwd`/old `pwd `" lpr&
3712                                           Process-id
$
```

We piped the output into pr and lpr just to illustrate that it's possible: the standard output of the programs within a for goes to the standard output of the for itself.

`for i in *` which loops over all filenames in the current directory, and

`for i in $*` which loops over all arguments to the shell file.)

The argument list for a for most often comes from pattern matching on filenames, but it can come from anything. It could be

```
$ for i in 'cat . . .'
```

or arguments could just be typed.

1. SHELL:

- The shell is a program (usually stored in the file 'sh' in the 'bin' directory) that acts as an interface b/w the user and the OS (kernel).
- As the name suggests the shell envelops the kernel.
- The shell is a program that acts as a command interpreter reads lines (i.e., commands) typed by the user and translates the requests into actions passing the commands directly to the kernel.

- All communications b/w the kernel and the user pass only through the shell. Shell also works as a programming language. The shell programs are known as shell scripts.
- The shell is not a part of the kernel but an application program that you see and use.
- Whenever the user logs into the system, a UNIX shell is automatically invoked and starts running as a separate, personal copy for each user i.e., at a particular instance there may be several copies of shell running on the system simultaneously with a minimum of one shell per user.
- Once authorized by the kernel to enter the login session, the shell displays the shell prompt ($ for ordinary users and # for super users).
- When the user types a command, the shell decodes the command line and searches for a program with the same name as the command in the list of directories containing program files as specified by the environmental variable PATH.
- In UNIX, there are only error messages and no success messages. If you type a command, and press the 'enter' key wait for the execution of the command. If there are no error messages, then, the meaning of that is the command has been successfully executed.
- Some of the most popular shells used in UNIX systems are.
  a) Bourne shell
  b) C shell
  c) Korn shell

a) Bourne Shell: The oldest UNIX shell, and most used one, was originally developed by Steve Bourne of AT&T Bell labs in the late 1970's.

- The Bourne shell is stored in the program file *sh.*

Features*:*

- A built in command set for writing shell scripts
- A set of shell variables for configuring your environment.
- Background execution of commands.
- Supports both I/O redirection.
- Supports use of wild card characters.

b) C shell: The C shell also referred as the Berkeley C shell was designed and developed by Bill Joy at the university of California at Berkeley for BSD UNIX distributions is now a part of system V.

- The C shell is stored in the program file *csh*

Features*:*

- A built in command set for writing shell scripts
- A set of shell variables for configuring your environment.
- Supports both I/O redirection .
- Supports command aliasing of frequently used commands.
- Supports rerunning of previously used commands
- Supports command substitution & command history.
- Supports job control

c)Korn Shell: The Korn shell was developed by David Korn of AT&T Bell labs in the late 1980's. The Korn shell is the superset of the Bourne shell and hence everything that works with the Bourne shell also works in the Korn shell.

- The Korn shell is stored in the program file *Ksh.*

Features*:*

- Supports command line editing with editors like vi, ed or emacs.
- Supports command aliasing and command history.
- Supports job control.
- Supports extensive pattern matching.

All the Shells are most similar, their syntax, standard features, and notations are extremely close to one another.

The default prompts of shells are:

(i) Bourne Shell(**$**) (ii) C Shell(**%**) (iii) Korn Shell(**$**)

Other Shells used in UNIX system includes.

a)Bash shell **b)** pd-Ksh shell **c)** dt-Ksh shell **d)** Tc Shell **e)** POSIX shell **f)**z shell **g)**Restricted Shell

a) Bash Shell(Bourne Again Shell): is an expanded version of the Bourne shell was developed by the free s/w foundation the bash shell has an extended shell scripting language that includes advanced loop constructs and other functions. The bash shell is stored in the program file */bin/bash.*

b)pd- Ksh Shell(public domain Korn Shell): is an expanded version of the Korn Shell developed at AT & T Bells labs.

The pd-Ksh is available in both binary and source form.

For a shell programmer there is no difference b/n the korn shell and pd-ksh shell.

All scripts that run in one version will also run in other version.

c)dt-Ksh shell(desktop Korn Shell) : is also an expanded version of the Korn shell, which provides the capability to create and display GUI's using the Korn shell syntax. It is stored in /*bin/dt-ksh*

***d*)TC Shell(Turbo C Shell):** is an expanded version of the C shell developed at the BSD UNIX distribution systems.

It has many advanced features like the command line editing, spelling correction, and easy retrieval of previously executed commands, immediate documentation access as the user types a command, ability to schedule for periodic execution.

e**)POSIX Shell(Portable Operating System Interface Shell)** is a superset of the Bourne shell. But it is much like the

Korn shell .The POSIX shell also offers aliases, path searches, command history, filename completion and job control.

f) Z- Shell: Developed by Paul Falstand in the early 90's as expanded version of the Bourne Shell, C shell and Korn shell with its own features. It is stored in a file */bin/zsh*

h) Restricted Shell: the restricted shell is almost same as the regular shell. But it is designed to restrict user's capabilities by restricting some of the standard actions that an ordinary shell allows.

A user working in the restricted shell

- Cannot change from his HOME directory to any system directories.
- Cannot create new files or append to existing files by redirecting the O/P using > or >> operators.
- Cannot change the PATH shell variable.
- Cannot use a command containing the symbol /

UNIT-8

The Process-The Meaning-Parent and Child Processes-Types of Processes-More about Foreground and Background processes-Internal and External Commands-Process Creation-The Trap Command-The Stty Command-The Kill Command-Job Control.

Because Unix is a multi-user as well as a multi-tasking system, number of programs can run simultaneously. All the programs that have been loaded into the memory for execution are referred to as processes.

1 THE MEANING

A process is defined as a program in execution. Unix being a multiuser and a multi-tasking system, there could be several programs belonging to different users or the same user running at the same time. All these programs share the same CPU. The kernel generates or spawns processes for every program under execution and allocates definite and equal CPU time slots to these various programs. Each of these processes have a unique identification number allocated to it by the kernel. Individual processes are identified by using these unique numbers, and are called process identification numbers or PIDs.

Mathematically, a process is represented by the tuple—

(*process id, code, data, register values, pc value*),

where process id (PID) is the unique identification number that is used to identify the process uniquely from other processes, code is the program code that is under execution, data is the data used during execution, register values are the values in CPU registers and PC value is the address in the program counter from where the execution of the program starts or continues. At present the maximum value of PID is 32767.

As soon as the system is booted, the kernel gets loaded into the memory and then gets executed. Immediately, a system process called the swapper is created. The PID of this process will be 0 (zero). This process 0 creates another process called init, meaning initialiser. This init is one of the first programs that is loaded which starts running immediately after the bootstrapping. The PID of init process is 1. This init process is responsible for setting up or initialising all subsequent processes on the system. init sets the user mode in either the single-user or the multi-user mode. Also init is responsible for generating processes on log-ins. It (process 1) exists as long as the system is running and it is the ancestor of all other processes on the system.

2 PARENT AND CHILD PROCESSES

In Unix, a process is responsible for generating another process. A process that generates another process is called the parent of the newly generated process, called the child. For example, when a command like $cat sample.lst is given, the shell creates a process for running the cat command. Thus, the shell sh (ksh or bash) being a process, generates another process (cat). Here the shell process is the parent process and the cat process is the child process. When a parent process

creates or generates a child process, a process is said to have born. As long as a process is active, it is said to be alive. Once the job of a process is over it becomes inactive and is said to be dead.

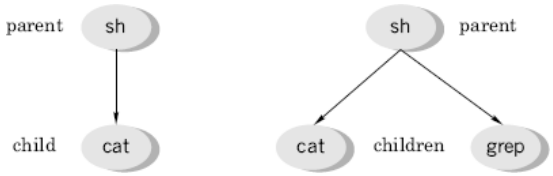


Fig 1 Parent and child processes

When a command like $cat sample.lst | grep lecturer is given to the shell, two processes, one for running the cat program and another for running the grep program are created simultaneously. Here, once again the shell process is the parent process and the cat and grep processes are its child processes. The cat and grep processes, which are the children of the same parent, will have different PIDs. All the child processes will inherit almost all the environmental parameters of their parent processes.

In general, a parent process waits for the complete execution of its child process—a parent waits for its child to die. However, sometimes a parent may die before its child. In such cases the child is said to be orphan. Generally these orphan processes are attached to the init process—the process with PID 1.

It should be noted that all the commands do not create processes. For example, running of the commands like cd, mkdir, pwd and others do not create processes.

2.1 Program and a Process

A careful observation of the parent and child processes and the relation between them (as discussed in the previous section) reveals that all processes get themselves arranged in the form of a hierarchical, inverted tree like structure. This is similar to file organization. The only difference between these two organizations is that file organization is *locational* whereas process organization is *temporal*.

A program exists in a single location in space and exists for any length of time. Thus a program is a static object that exists in a file. It contains just the entire set of instructions. But a process is a program in execution. Thus, it is a *dynamic object* and can never be in a file. It is a sequence of instructions under execution. Thus process has a definite life cycle.

3 TYPES OF PROCESSES

Processes within Unix are classified into three general categories—as *interactive processes, non-interactive processes* and *daemons*.

3.1 Interactive Processes—Foreground Processes

All the user processes, which are created by users with the shell, act upon the directions of the users and are normally attached to the terminal are called interactive processes. These types of processes are also called *foreground processes.*

3.2 Non-interactive Processes—Background Processes

Certain processes can be made to run independent of terminals. Such processes that run without any attachment to a terminal are called noninteractive processes. These types of processes are also called *background processes.*

3.3 Daemons

All processes that keep running always without holding up any terminals and keep waiting for certain instructions either from the system or the user and then immediately get into action are called daemons. swapper, init, cron, bdflush, vhandle are some examples of daemons. These daemons come into existence as soon as the system is booted and will be alive till the system is shut down. One cannot kill these processes prematurely.

4 MORE ABOUT FOREGROUND AND BACKGROUND PROCESSES

When a command is given, the shell parses, rebuilds and then hands it over to the kernel for execution. The shell then keeps on waiting for the kernel to complete the execution. During this shell-waiting period the user cannot issue any other command because the terminal is held up with the command under execution. As already mentioned, commands that hold up the terminal during their execution are called foreground processes. The chief disadvantage of foreground processing is that, no further commands can be given from the terminal as long as the older one is running. This disadvantage becomes significant when a currently running process is big and takes a lot of time for processing.

It is possible to make processes to run without using the terminal. Such processes take their input from some file and process it without holding up the terminal (non-interactively), and write their output on to another file are called background processes. Typical jobs that could be run in background are sorting of a large database file or locating a file in a big file system by using the find command and so on.

4.1 Running a Command in the BackgroundA command is made to run in the background (as a background process) by terminating the command line with an ampersand (&) character as shown in the following example.

```
 $sort –o student.lst student.lst &
567
$
```

The shell immediately returns the process identification (PID) number as well as the shell prompt $. In the above example, 567 is the PID of the just-submitted background job. As the shell prompt ($) re-appears immedietly, one can now readily work at the terminal. One should be careful in running background processes as the user may get into problems under certain situations. Some of these problems could be due to any one of the following.

1. The success or failure of the background processes are not reported. The user has to find it out. For this purpose the identification number is used.
2. The output has to be redirected to a file as otherwise the display on the monitor gets mixed up.
3. Too many processes running in the background degrades the overall efficiency of the system.

4. There is a danger of the user logging out when some processes are still running in the background.

5 INTERNAL AND EXTERNAL COMMANDS

The classification of commands depending on whether they generate separate processes or not upon their running is discussed here. Most of the commands such as cat, who and others generate separate processes as soon as they are used. Commands that generate separate processes upon their running are called external commands. Some commands such as mkdir, rm, cd and others do not generate new processes when they are used; such commands are called internal commands.

6 THE ps COMMAND—KNOWING PROCESS ATTRIBUTES

The ps command is used to display the attributes of processes that are running currently. This is one of the commands that varies too much from one system to another. This comes with a number of options like –a (all users), –f (full list), –u (user), –t (terminal) and –e (every).

When used with *no option*, the ps command lists out certain attributes associated with the terminal as shown below.

```
$ps
PID        TTY        TIME        CMD
476        tty03      00:00:01    login
659        tty03      00:00:01    sh
684        tty03      00:00:00    ps
$
```

where PID = process identification number
TTY = terminal type
TIME = cumulative time
CMD = command

A *full listing* of the processes can be obtained by using the –f option with the ps command, as shown below. As seen from the example on the next page, using this option, one can trace the ancestry of different processes also.

```
$ps –f
UID     PID     PPID    C   STIME      TTY      TIME       CMD
mgv     16118   3211    0   15:58:00   tty03    00:00:01   /usr/bin/vi/notes
mgv     3211    1       0   15:16:15   tty03    00:00:00   /usr/lbin/ksh
mgv     2187    16118   0   16:35:00   tty03    00:00:00   sh
mgv     4700    2187    27  16:43:50   tty03    00:00:00   ps – f
$
```

where UID = user ID

PPID = parent process ID
STIME = starting time
C = CPU time consumed

Here the user mgv uses vi to edit a file named notes. The ksh is the users' login shell since its parent process id is 1. ksh is the parent of vi/notes, and so on.

All the process of a particular user only can be listed by using the –u option along with the user-ID as an argument to the ps command, as shown below.

```
$ps –f –u mgv
UID   PID    PPID   C   STIME      TTY    TIME      CMD
mgv   16118  3211   0   15:58:00   tty03  00:00:01  /usr/bin/vi/notes
mgv   3211   1      0   15:16:15   tty03  00:00:00  /usr/lbin/ksh
mgv   2187   16118  0   16:35:00   tty03  00:00:00  sh
mgv   4700   2187   27  16:43:50   tty03  00:00:00  ps – f –u mgv
$
```

The process of all the users only (not the system processes) can be listed by using the –a option as shown in the following example.

```
$ps –a
PID     TTY     TIME      CMD
625     tty03   00:00:00  sh
337     tty01   00:00:00  ksh
680     tty02   00:04:00  vi
749     tty04   00:04:00  ksh
$
```

All the processes including the system processes are listed using the ps command along with the –e (every process) option as shown in the following illustration.

```
$ps –e
PID     TTY     TIME      CMD
0       ?       00:00:00  sched
1       ?       00:01:00  init
2       ?       00:00:00  vhand
3       ?       00:00:00  bdflush
487     tty01   00:01:00  sh
289     tty03   00:00:00  getty
125     ?       00:00:00  lpsched
531     ?       00:00:00  nfsd
311     ?       00:00:00  cron
622     ?       00:00:00  inetd
$
```

The appearance of a question mark (?) in the TTY column indicates that these are system processes. In the above listing, bdflush is the buffer to disk flushing activity support routine, nfsd is the network file system daemon, inetd is the internet daemon without which the TCP/IP does not work, vhand is the system routine that handles virtual memory management implementations and so on.

It should be noted that system processes support activities of the system and keep on doing their task, independent of what users are doing, as long as the system is on.

7 PROCESS CREATION

There are three distinct phases in the creation of a process. They are (1) forking, (2) overlaying and execution and (3) waiting. These three phases are taken care of by making calls to the system routines fork(), exec() and wait(), respectively.

Forking is the first phase in the creation of a process by a process. The calling process (parent) makes a call to the system routine fork() (the call here is referred to as a system call) which then makes an exact copy of itself. The copy will be of the memory of the calling process at the time of the fork() system call and not of the complete program the calling process was started with. Right after the fork() there will be two processes with identical memory images. Each one of these two processes has to return from the fork() system call. Thus there will be two return values. The fork of the parent process returns the PID of the new process, that is the child process just created, whereas the fork of the child returns a 0 (zero). Incase a new child process is not created a –1 is returned.

Immediately after forking, the parent makes a system call to one of the wait() functions. By doing so, the parent keeps waiting for the child process to complete its task. It awakens only when it receives a complete signal from the child, after which it will be free to continue with its other functions.

The child process inherits almost the entire environment of the calling process. In other words, the child process will have the same priority, same signal handling settings, same group and user ids, same current directory and so on. However, children will not inherit the local variables and will have different PID’s.

In the second phase, the parent makes a system call to one of the exec() functions. This system call simply overwrites the text and data area of the child process by the text and data of the new program and then starts executing this new program. At the end of the overlaying and execution, a call is made to the exit() function that terminates the child and sends a signal back to the parent after which, the parent becomes free to continue with its other functions. The entire mechanism of process creation is pictorially shown in Fig. 7.2.

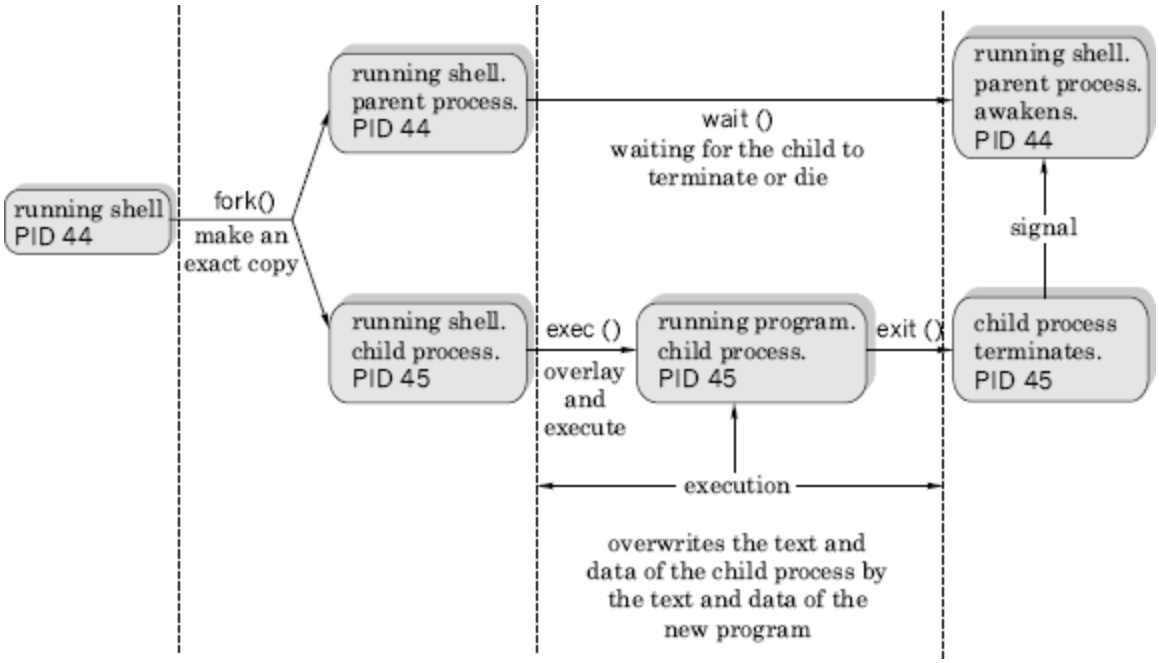


Fig..2 Mechanism of process creation

8 SIGNALS

A signal is a message sent to a program under execution, that is a process, on one of the following two occasions.

1. Under some error conditions or the user interruption, the kernel generates signals.
2. During interprocess communication between two or more processes. The participating processes generate these signals. For example, a child process sends a signal to its parent process upon its termination.

In Unix, signals are identified by integers. They have names too. These names are in uppercase and start with SIG. There are about 30 such signals, numbered from 1. Some commercial implementations like AIX have more signals. The table below gives a list of exit or interrupt signals.

Signal Number	***Name***	***Function***
1	SIGHUP	Hangup; closes process communication links.
2	SIGINT	Interrupt; tells process to exit.(Ctrl–c)
3	SIGQUIT	Quit; forces the process to quit. (Ctrl–\)
9	SIGKILL	Sure kill; cannot be trapped or ignored.
15	SIGTERM	Software termination; default signal for the kill command.
24	SIGSTOP	Stop; (Ctrl–Z)

9 THE trap COMMAND

Normally signals are used to prematurely terminate the execution of a process either intentionally or unintentionally. The trap command is used to trap one or more signals and then decide about the further course of action. If no action is mentioned, then the signal or signals are just trapped and the execution of the program resumes from the point from where it had been left off. The general format of this command is given below.

```
$trap [commands] signal_numbers
```

The commands part is optional. When it is present, all the commands present in this part are executed one by one as soon as the process receives one of the signals specified in the signal_numbers list. The commands used, must be enclosed using either single or double quotation marks. Multiple commands in the commands part are separated by the; (semicolon) character. Following are some examples.

```
(i) $trap "echo killed by signal 15; exit" 15
```

When the process receives a kill command, causing signal 15, the above command first gives the message killed by signal 15 and then terminates the current process because of the execution of the exit command.

```
(ii) $trap "ls –l" 1 2 3
```

When the process generates anyone of the signals 1, 2 or 3, a long listing of the current working directory is generated and then execution of the process resumes from the point where it had been left off.

```
(iii) $trap " " 1 2 3 15
```

This command just traps the signal numbers 1, 2, 3 and 15.

Though majority of the signals can be trapped, certain signals like signal number 9 (the sure kill) cannot be trapped. Given below is a simple script that keeps on running till the user interrupts it by using the interrupt key.

```
$cat –n sample.trap
1 trap "echo PROGRAM INTERRUPTED; exit 1" 2
2 while true
3 do
4  echo "program running."
5 done$
```

Resetting Traps Normally a trap command changes the default actions of the signals. Using the trap command, without the commands' part, changes the specified signals to their default

actions. This behaviour of the trap command is useful under certain situations. For example, one might need to trap a certain signal in one part of a script and need the same signals not to be trapped in some other part. The command to trap the signal will be as shown below.

```
$trap "exit" 2 3 15
```

The effect of the signals 2, 3 and 15 are restored by using the trap command without the command part in it as shown in the following example.

```
 $trap 2 3 15
```

10THE stty COMMAND

One of the most widely used methods to communicate with a system is to use terminals, that is via keyboards. There are certain combination of keys, on these terminals, which control the behavior of any program in execution. For example, we have been using

1. <Ctrl–m>(^m), that is the <RETURN> key to end a command line and execute the command.
2. <Ctrl–c>(^c) to interrupt a current process and to come back to the shell.
3. <Ctrl–s>(^s) to pause display on the monitor.
4. <Ctrl–d>(^d) to indicate end of file and so on.

The stty command is used to see or verify the settings of different keys on the keyboard. The user can have a short listing of the settings by using this command without any arguments. In order to see all the settings, it has to be used with the –a (all) option, as shown in the following example.

```
 $stty –a
speed 9600 baud; ispeed 9600 baud; line = 0(tty);
erase = ^?; kill = ^U; eof = ^D; intr = ^C ; stop = ^S;
echo echoe – – – – – – – – – – – – – – – – – – –
$
```

The output shown above is just illustrative. From the output one can see that the terminal speed is 9600 bauds, ^U is used for killing a line, ^D is used to indicate end of file, because of echo everything typed at the keyboard gets echoed on the display terminal, backspacing over a character retains its display, and so on.

This command can also be used to change the key settings as shown in the following examples.

```
 $stty –echo
$stty eof \^a
```

Execution of the former command, stops the display of characters that are typed at the keyboard. It may be noted that this is the setting used to handle passwords. After the execution of the latter command, the use of <Ctrl–a> terminates all standard input.

It is recommended not to play around with the terminal settings. This may lead to improper working of the terminal. However, if the user finds that the terminal is not working properly, he

or she may restore the sanity into terminal settings by using the word sane as a single argument to stty, as shown below. **$stty sane**

The execution of the above command sets the terminal settings with reasonable values.

11 THE kill COMMAND There are certain situations when one likes to terminate a process prematurely. Some of these situations are as follows.

- When the machine has hung.
- When a running program has gone into an endless loop.
- When a program is doing unintended things.
- When the system performance goes below acceptable levels because of too many background processes.

Terminating a process prematurely is called killing. Killing a foreground process is straightforward. This is done by using either the DEL key or the BREAK key. However, to kill a background process the kill command is used. This command is given with the PID of the process to be killed as its argument. If the PID is not known the ps command is used to know the same.

For example, a process having an identification number 555 can be killed using the kill command as shown in the following example.

```
$kill 555
```

More than one process can be terminated using a single kill command as shown in the following example.

```
$kill 330 333 375          # here 330,    333,    375 are process id's.
```

A kill command, when invoked, sends a termination signal to the process being killed. When used without any option, it sends 15 as its default signal number. This signal number 15 is known as the software termination signal and is ignored by many processes. For example, the shell process sh, ignores signal 15. In other words, signal 15 does not guarantee the killing of all processes. At such times, one can use signal number 9, the sure kill signal, to terminate a process forcibly as shown in the following example.

```
$kill –9 666              # 666 is the id number of the process
```

All the processes of a user (except his login shell) can be terminated by using a 0 (zero) as the argument of the kill command as shown in the following example.

```
$kill 0           # kill all the processes except the login shell
```

However using 9 as option and 0 (zero) as the argument, all processes including the shell can be killed as shown in the following example.

```
$kill –9 0          # kills all processes including the login shell
```

$! and $$ System Variables The special variable $! holds the PID value of the last background job, and the special variable $$ holds the PID value of the current shell. The last background job can be killed using the command $kill $!. The current shell can be killed using the sure kill command $kill –9 $$.

THE wait COMMAND

With some shells like the korn and bash, jobs can be run in the background as background processes. Sometimes it is necessary to wait for either all the background jobs or a specific job to be executed completely before any further action is initiated. Under such circumstances, the wait command is used for waiting background process(s) to be completely executed. Some examples are given here.

```
$wait #waits till all the background processes are completely executed
$wait 227 #waits for the completion of the process with PID 227
```

12 JOB CONTROL Unix is a multi-tasking system and there will be many number of jobs or processes running simultaneously in a Unix environment. Quite often, it will be required to know

1. how many as well as which processes are running currently,
2. terminate either a misbehaving or a unwanted process,
3. modify the priority of a process,
4. to push a process into the background,
5. to bring up a required process to the foreground, and so on.

The above-listed type of activities is generally referred to as job-control activities. In Unix, there exist many commands by using which, one can perform any of the job-control activities. For example, the ps command is used to know details of currently running processes. The kill command is used to prematurely kill (or terminate) a process, the wait command is used to make a process to wait till its child is terminated, and so on. All these commands are available with all the shells including the Bourne Shell and have already been discussed. Other job-control commands such as jobs, fg and bg, which are available in Korn and some other recent shells (not with the Bourne Shell), have been discussed in the following sections.

.1 Job Control Commands—The jobs, fg and bgA command or a command line with a number of commands put together or a script is generally referred to as a job. In Unix, as one can run commands in the background, there could be a number of commands, that is, processes, running in the background. Also there could be a command—a process—running in the foreground.

The jobs **Command** A list of all the current jobs is obtained using the jobs commands as shown below.

```
[ksh]jobs       #The{ksh}prompt has been used intentionally
[1] + Running sort emp.data|grep `Bangalore`>address.lst &
[2] – Runningsleep 1000 &
```

[ksh]

In the above output, a + (plus) and – (minus) that appear after the job number mark the current and previous jobs, respectively. The word running indicates that the job is currently being executed. The alternate information that could appear in this position are stopped, suspended, terminated, done and exits. The output also displays the command name. After knowing the status of the jobs running in the background one may take any required action like bringing a job to the foreground, killing a job and so on.

The fg **Command** This command is used to bring a job that is being executed in the background currently to the foreground. This command can be either used without any argument or with a job number as its argument. Some simple illustrations are given here.

```
[ksh]fg            # Brings the most recent background process
                   # to the foreground
[ksh]fg %2         # Brings job number 2 to the foreground
[ksh]fg %sort      # Brings the job the name of which begins
                   # with sort to the foreground
```

As seen from the above examples, whenever a job number is used as an argument with a job-control command (not necessarily with fg only) it must be preceded by a percent sign (%). Here it may be noted that the current job may be referred to by using any one of the representations—%1 or %+ or %%. Also, it may be noted that first few characters of a command sequence can be used to refer to job as shown in the last example in the previous set of illustrations.

The bg **Command** A new job can be made to run in the background by using the & (ampersand) at the end of a command line as discussed in Section 7.4.1. The question here is how to make a currently running foreground process to run in the background? The answer is very simple. The currently running foreground process is first suspended, by using the <ctrl–z> keys, and then making it to run in the background by using the bg command. By assuming that the currently running process has been suspended right now, the following command line puts it in the background.

```
[ksh]bg %1     # resumes job number 1 in background
```

SCHEDULING JOBS' EXECUTION

Normally, commands or programs are executed by using a suitable command line as and when required by typing them at the system prompt. In Unix, it is possible to get commands executed at any required time, whenever the system is relatively free and repeatedly according to certain requirement.

Commands such as at, batch and cron are used for scheduling execution of commands according to requirements.

The at Command—Running a Command at a Future Date and Time

This command is capable of executing Unix commands at a future date and time. The input to this command has to come from the standard input. In other words, commands may be typed in through the keyboard or may be provided through a file.

```
$at 17:00
clear > /dev/tty03
echo "It is 5 P.M. Back up your files and logout" > /dev/tty03
<ctrl–d>
job 801346789.a at Fri Jan 11 17:00:00 IST 2002
$
```

Once a job is submitted using the at command, details regarding the job id number, the date and time at which commands are to be executed are displayed. The job id number is based on number of seconds elapsed since the beginning of 1970. Note that neither the PID nor the filename of the process are displayed. One has to be extra careful in monitoring the jobs that are scheduled when using this command. It should be observed that the job id terminates with a .a. If the output of the at command is not redirected as shown in the above example, the output will arrive at the terminal as a mail at the scheduled time.

Once the command is submitted in the above-mentioned manner, the message will be displayed on the terminal at 5 pm sharp.

The time can use am and pm suffixes. If these suffixes are not given, the time will be taken in the 24-h format. Keywords like *now, noon, midnight, today, tomorrow, hours, days, weeks, months* and *years* can be used with this command. A list of some typical examples are given below.

```
 $at 1 pm today
$at noon
$at 15
$at 10 am tomorrow
$at now + 1 year
```

A file can be given as an argument to an at command using the –f option as shown in the following example. `$at –f scriptfile 7 am Monday`

The information regarding jobs that are scheduled using at will be available on a queue called the at queue. The details of this can be obtained using the –l option as shown below.

```
$at –l
889673410.a Wed Dec 31   15:08:00 2003
.. ... ..... .... ....

.. ... ..... .... ....
```

A job scheduled with at command can be removed prematurely by using the –r option. For this, one has to remember and use the job id as shown in the following example.

```
$at –r 889673410.a
```

The batch Command

Jobs submitted by using this command are executed when the system is relatively free and the system load is light. Since the time at which the commands are executed is decided by the system, there is no need to specify the time. An example is given here.

```
$batch
sort emp.dat   |   grep `Bangalore` > address.lst
<ctrl–d>
job 6423 22445.b at Fri Jan 16 17:00:00 IST 2004
$
```

The extension .b attached to the job identification number indicates that it has been submitted by using the batch command. Jobs scheduled using this command also sit in the at queue.

The cron Daemon and the crontab Command

The term cron is derived from the word chronograph. Using this facility one can schedule required jobs to run periodically. Cron is a system daemon that keeps sleeping most of the time. It typically wakes up once every minute and checks its crontab file for any jobs to be executed during this minute. All users have a crontab file of his or her own. The name of this crontab file will be the user's login name. Scheduled jobs will be present in the crontab file. crontab files will be present in the /var/spool/cron/crontabs directory.

A crontab file may contain one or more lines, each corresponding to a command that is to be executed periodically at a specified day, date and time. Figure 7.3 gives the basic syntax of a line on a crontab file. Every such line will be made up of six fields with each field separated by a blank.

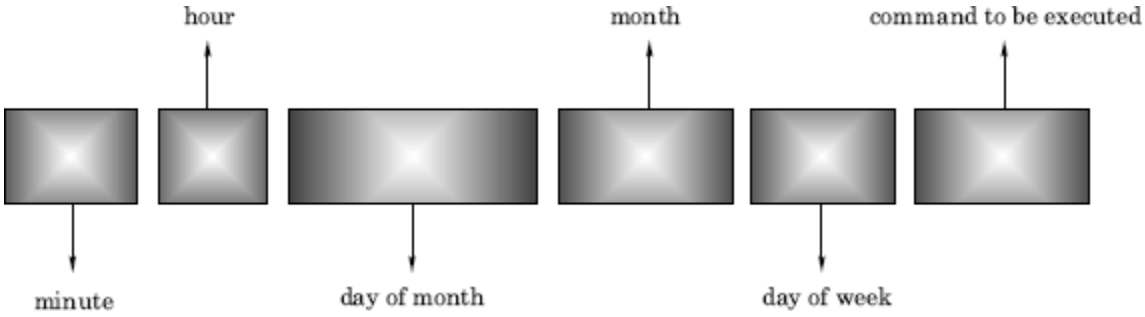


Fig. 7.3 Syntax of a crontab line

As seen from the figure above, the first field specifies the minute (0–59), the second field specifies the hour (24-h format), the third field specifies the day of the month (1–31), the fourth field specifies the month (1–12), the fifth field specifies the day of the week (0–6), 0 being Sunday, and the sixth field contains the command line to be executed. In a crontab line, an asterisk (*) represents all possible values. For example, if a * character appears in the fifth field, then the command mentioned in the line will be executed on all the days of the week at the specified time. If necessary groups of numbers can be specified within a single field by separating them with commas. No spaces are allowed within a field. Below are given two typical crontab lines.

1. 0 0 * * * backup.sh
   When executed, the above line runs the backup.sh script at midnight everyday.
2. 00,30 09–17 * * 1–5 mail.sh
   When executed, the above line runs the mail.sh script on all weekdays—Monday to Friday every half hour between 9 and 17 hours.

From the above examples, one can see that a crontab line not only contains commands to be repeatedly executed but also the details of date and time in a specific format.

When more than one command has to be periodically executed, every command has to be written in a separate line having the above format in a separate file. Then this file is submitted to the crontab command, as shown in the example below, where cmdfile is the name of the file that contains the command lines that are to be executed periodically.

$crontab cmdfile

When a file is submitted by using the crontab command, its contents are automatically transferred to the /var/spool/cron/crontabs directory. The crontab command when used without any argument accepts the input from the standard input—the keyboard. As usual, the input operation from the keyboard has to be terminated using <ctrl–d> keys. A careless use of this method removes all the entries on the existing crontab file. One has to be extra careful while entering crontab lines via the keyboard.

The contents of the crontab file can be seen using the $crontab –l command. A submitted file can be removed using the –r option as in $crontab –r command. It may be noted that here job name or job id is not required, as every user will have just one crontab file of his or her own.

THE nohup COMMAND

One of the dangers of running processes in the background is that, if a user logs out with one or more processes still running in the background, the running processes will have an unnatural death. Further, processes that need large processing time are run in the background. In such cases, rather than waiting, one can logout deliberately keeping the process running, come back and get the results. In either of the two cases, the user can keep running the process in the background till they are completed even when s/he logs out (not that the system is shut down). In other words, the user does not want the system to hang up the background process. This is

accomplished by using a command called the nohup command, as shown in the following example.

```
$nohup sort –o students.lst students.lst &
79
$
```

Once a command is submitted with nohup one can logout without the process getting terminated on logging out. As shown in the above example, the output filename has to be mentioned. If it is not mentioned, the output will be stored in a file called nohup.out by default. Further, whenever commands are piped, each command should be qualified by the nohup command, as shown in the following example. This is because, every command in the pipeline spawns a process of its own.

```
 $nohup cat students.lst|nohup grep `murthy`|nohup sort > names.lst &
86
$
```

THE nice COMMAND Processes in Unix have equal priority. In Unix, the priority of a process is measured by using integer numbers ranging from 0 to 39 (in Linux this ranges is –19 to +20). A 0 (zero) indicates the highest priority whereas 39 indicates the lowest priority. The default value of the priority assigned to a process upon its creation is 20 in the Bourne shell (0 [zero] in the case of Linux). The priority of a process can be reduced to a lower-than-normal value by using the nice command as shown in the following example. Default value of reduction is 10 units.

```
 $nice big_program
```

This command runs big_program with a priority value reduced by 10 units, that is, with a priority value of 30. When the priority of a command is reduced it uses less CPU time and runs slower. It is possible to reduce the priority of a job by using a number option along with the nice command, as shown in the following example.

```
$nice –19 big_program
```

The above-mentioned command runs the file big_program with a priority value of 39. The following command lowers the priority and runs the command in the background.

```
$nice –19 big_program &
```

Users cannot increase the priority of a job. If such a facility is given, everyone likes to run his or her job with the highest priority. However, the system administrator or the supervisor can raise the priority of a process by using the nice command, with double minus option (– –), as shown in the following example.

```
#nice – –12 big_program
```

The above command runs the file big_program with a priority value of 8. The # character in this command line indicates that this command is issued by the supervisor.

THE time COMMAND This command is used to know the resource usage. It runs a program or command with given arguments, generates a timing statistics about the program run and directs this statistics report to the standard output. This statistics consists of the elapsed time between invocation and termination, the user CPU time and the system CPU time. By analyzing this, one can assess the efficiency of a program or command

```
 $find  /  –name  makefilex –print
real    0m14.509s
user    0m0.150s
```

```
sys     0m0.390s
$
```