Rs = 110/-

Working with

UNIX

Table of Contents

- tar
- cpio
- df, du, mount, unmount

Redirection operators

- Redirecting Output
- Redirecting Input
- Standard error

Pipes and Filters

- tr, comm., tee, sed, nl
- pg, more, less, head, tail, paste, cut, sort

Exporting Variables

Sed (Stream Editor) & GREP

- grep, grep with options
- fgrep, egrep

Processes and Job control

- ps, kill, Pkill
- Foreground jobs
- Background jobs
- Killing jobs, Nohup
- At, Crontab, Batch

Editing Files

- Command mode
- Insert mode
- Ex command mode

Shell Scripting

- What is shell scripting?
- Importance of shell scripting
- Different types of shells
- Responsibilities of Shell
- What is a variable?
- System defined variables
- Environment Variables

Operators

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operators

Conditional Statements

- If, if....else
- If... elif
- Case

Looping

- While, until , for , break , continue
- More on Loops
- Example Scripts

UNIX/LINUX

Introduction:

What is an Operating System?

The 1960s definition of an operating system is "the software that controls the hardware". However, today, due to microcode we need a better definition. We see an operating system as the programs that make the hardware useable. In brief, an operating system is the set of programs that controls a computer. Some examples of operating systems are UNIX, Mach, MS-DOS, MS-Windows, Windows/NT, Chicago, OS/2, MacOS, VMS, MVS, and VM. Controlling the computer involves software at several levels. We will differentiate kernel services, library services, and application-level services, all of which are part of the operating system. Processes run Applications, which are linked together with libraries that perform standard services. The kernel supports the processes by providing a path to the peripheral devices. The kernel responds to service calls from the processes and interrupts from the devices. The core of the operating system is the kernel, a control program that functions in privileged state (an execution context that allows all hardware instructions to be executed), reacting to interrupts from external devices and to service requests and traps from processes. Generally, the kernel is a permanent resident of the computer. It creates and terminates processes and responds to their request for service. Operating Systems are resource managers. The main resource is computer hardware in the form of processors, storage, input/output devices, communication devices, and data. Some of the operating system functions are: implementing the user interface, sharing hardware among users, allowing users to share data among themselves, preventing users from interfering with one another, scheduling resources among users, facilitating input/output, recovering from errors, accounting for resource usage, facilitating parallel operations, organizing data for secure and rapid access, and handling network communications.

Objectives of Operating Systems

Modern Operating systems generally have following three major goals. Operating systems generally accomplish these goals by running processes in low privilege and providing service calls that invoke the operating system kernel in high-privilege state.

→ To hide details of hardware by creating abstraction an abstraction is software that hides lower level details and provides a set of higher-level functions. An operating system transforms the physical world of devices, instructions, memory, and time into virtual world that is the result of abstractions built by the operating system. There are several reasons for abstraction. First, the code needed to control peripheral devices is not standardized. Operating systems provide subroutines called device drivers that perform operations on behalf of programs for example, input/output operations. Second, the operating system introduces new functions as it abstracts the hardware. For instance, operating system introduces the file abstraction so that programs do not have to deal with disks. Third, the operating system transforms the computer hardware into multiple virtual computers, each belonging to a different program. Each program that is running is called a process. Each process views the hardware through the lens of abstraction. Fourth, the operating system can enforce security through abstraction.

→To allocate resources to processes (Manage resources) an operating system controls how processes (the active agents) may access resources (passive entities).

→ Provide a pleasant and effective user interface the user interacts with the operating systems through the user interface and usually interested in the "look and feel" of the operating system. The most important components of the user interface are the command interpreter, the file system, on-line help, and application integration. The recent trend has been toward increasingly integrated graphical user interfaces that encompass the activities of multiple processes on networks of computers. One can view Operating Systems from two points of views: Resource manager and extended machines. Form Resource manager point of view Operating Systems manage the

different parts of the system efficiently and from extended machines point of view Operating Systems provide a virtual machine to users that is more convenient to use. The structurally Operating Systems can be design as a monolithic system, a hierarchy of layers, a virtual machine system, an exocrine, or using the client-server model. The basic concepts of Operating Systems are processes, memory management, I/O management, the file systems, and security.

History of Operating Systems:

Historically operating systems have been tightly related to the computer architecture, it is good idea to study the history of operating systems from the architecture of the computers on which they run.

Operating systems have evolved through a number of distinct phases or generations which corresponds roughly to the decades.

The 1940's - First Generations

The earliest electronic digital computers had no operating systems. Machines of the time were so primitive that programs were often entered one bit at time on rows of mechanical switches (plug boards). Programming languages were unknown (not even assembly languages). Operating systems were unheard of .

The 1950's - Second Generation

By the early 1950's, the routine had improved somewhat with the introduction of punch cards. The General Motors Research Laboratories implemented the first operating systems in early 1950's for their IBM 701. The system of the 50's generally ran one job at a time. These were called single-stream batch processing systems because programs and data were submitted in groups or batches.

The 1960's - Third Generation

The systems of the 1960's were also batch processing systems, but they were able to take better advantage of the computer's resources by running several jobs at once. So operating systems designers developed the concept of multiprogramming in which several jobs are in main memory at once; a processor is switched from job to job as

needed to keep several jobs advancing while keeping the peripheral devices in use. For example, on the system with no multiprogramming, when the current job paused to wait for other I/O operation to complete, the CPU simply sat idle until the I/O finished. The solution for this problem that evolved was to partition memory into several pieces, with a different job in each partition. While one job was waiting for I/O to complete, another job could be using the CPU. Another major feature in third-generation operating system was the technique called spooling (simultaneous peripheral operations on line). In spooling, a high-speed device like a disk interposed between a running program and a low-speed device involved with the program in input/output. Instead of writing directly to a printer, for example, outputs are written to the disk. Programs can run to completion faster, and other programs can be initiated sooner when the printer becomes available, the outputs may be printed. Note that spooling technique is much like thread being spun to a spool so that it may be later be unwound as needed. Another feature present in this generation was time-sharing technique, a variant of multiprogramming technique, in which each user has an on-line (i.e., directly connected) terminal. Because the user is present and interacting with the computer, the computer system must respond quickly to user requests, otherwise user productivity could suffer. Timesharing systems were developed to multiprogramming large number of simultaneous interactive users.

Fourth Generation

With the development of LSI (Large Scale Integration) circuits, chips, operating system entered in the system entered in the personal computer and the workstation age. Microprocessor technology evolved to the point that it becomes possible to build desktop computers as powerful as the mainframes of the 1970s. Two operating systems have dominated the personal computer scene: MS-DOS, written by Microsoft, Inc. for the IBM PC and other machines using the Intel 8088 CPU and its successors, and UNIX, which is dominant on the large personal computers using the Motorola 6899 CPU family.

Operating Systems Structure

System Components:

Even though, not all systems have the same structure many modern operating systems share the same goal of supporting the following types of system components.

1. Process Management

The operating system manages many kinds of activities ranging from user programs to system programs like printer spooler, name servers, file server etc. Each of these activities is encapsulated in a process. A process includes the complete execution context (code, data, PC, resisters, OS resources in use etc.). It is important to note that a process is not a program. A process is only ONE instant of a program in execution. There are many processes can be running the same program. The five major activities of an operating system in regard to process management are

→Creation and deletion of user and system processes.

→ Suspension and resumption of processes.

→A mechanism for process synchronization.

→ A mechanism for process communication.

→A mechanism for deadlock handling.

Main-Memory Management

Primary-Memory or Main-Memory is a large array of words or bytes. Each word or byte has its own address. Main-memory provides storage that can be access directly by the CPU. That is to say for a program to be executed, it must in the main memory. The major activities of an operating in regard to memory-management are:

→ Keep track of which part of memory are currently being used and by whom.

→Decide which process is loaded into memory when memory space becomes available.

→Allocate and de-allocate memory space as needed.

File Management

A file is a collected of related information defined by its creator. Computer can store files on the disk (secondary storage), which provide long term storage. Some examples of storage media are magnetic tape, magnetic disk and optical disk. Each of these media has its own properties like speed, capacity, and data transfer rate and access methods. File systems normally organized into directories to ease their use. These directories may contain files and other directions. The five main major activities of an operating system in regard to file management are

1. The creation and deletion of files.
2. The creation and deletion of directions.
3. The support of primitives for manipulating files and directions.
4. The mapping of files onto secondary storage.
5. The back up of files on stable storage media.

I/O System Management

I/O subsystem hides the peculiarities of specific hardware devices from the user. Only the device driver knows the peculiarities of the specific device to which it is assigned.

Secondary-Storage Management

Generally speaking, systems have several levels of storage, including primary storage, secondary storage and cache storage. Instructions and data must be placed in primary storage or cache to be referenced by a running program. Because main memory is too small to accommodate all data and programs, and its data are lost when power is lost, the computer system must provide secondary storage to back up main memory. Secondary storage consists of tapes, disks, and other media designed to hold information that will eventually be accessed in primary storage (primary, secondary, Cache) is ordinarily divided into bytes or words consisting of a fixed number of bytes. Each location in storage has an address; the set of all addresses available to a program

is called an address space. The three major activities of an operating system in regard to secondary storage management are:

1. Managing the free space available on the secondary-storage device.
2. Allocation of storage space when new files have to be written.
3. Scheduling the requests for memory access.

Networking

A distributed system is a collection of processors that do not share memory, peripheral devices, or a clock. The processors communicate with one another through communication lines called network. The communication-network design must consider routing and connection strategies, and the problems of contention and security.

Protection System

If a computer system has multiple users and allows the concurrent execution of multiple processes, then the various processes must be protected from one another's activities. Protection refers to mechanism for controlling the access of programs, processes, or users to the resources defined by computer systems.

Command Interpreter System

A command interpreter is an interface of the operating system with the user. The user gives commands with are executed by operating system (usually by turning them into system calls). The main function of a command interpreter is to get and execute the next user specified command. Command-Interpreter is usually not part of the kernel, since multiple command interpreters (shell, in UNIX terminology) may be support by an operating system, and they do not really need to run in kernel mode. There are two main advantages to separating the command interpreter from the kernel.

1. If we want to change the way the command interpreter looks, i.e., I want to change the interface of command interpreter, I am able to do that if the command interpreter is separate from the kernel. I cannot change the code of the kernel so I cannot modify the interface.

2. If the command interpreter is a part of the kernel it is possible for a malicious process to gain access to certain part of the kernel that it showed not have to avoid this ugly scenario it is advantageous to have the command interpreter separate from kernel.

Operating Systems Services following are the five services provided by operating systems to the convenience of the users.

Program Execution

The purpose of computer systems is to allow the user to execute programs. So the operating systems provide an environment where the user can conveniently run programs. The user does not have to worry about the memory allocation or multitasking or anything. These things are taken care of by the operating systems. Running a program involves the allocating and deal locating memory, CPU scheduling in case of multiprocessing. These functions cannot be given to the user-level programs. So user-level programs cannot help the user to run programs independently without the help from operating systems.

I/O Operations

Each program requires an input and produces output. This involves the use of I/O. The operating systems hides the user the details of underlying hardware for the I/O. All the user sees is that the I/O has been performed without any details. So the operating systems by providing I/O make it convenient for the users to run programs. For efficiently and protection users cannot control I/O so this service cannot be provided by user-level programs.

File System Manipulation

The output of a program may need to be written into new files or input taken from some files. The operating systems provide this service. The user does not have to worry about secondary storage management. User gives a command for reading or writing to a file and sees his task accomplished. Thus operating systems make it easier for user programs to accomplish their task. This service involves secondary storage

management. The speed of I/O that depends on secondary storage management is critical to the speed of many programs and hence I think it is best relegated to the operating systems to manage it than giving individual users the control of it. It is not difficult for the user-level programs to provide these services but for above mentioned reasons it is best if this service s left with operating system.

Communications

There are instances where processes need to communicate with each other to exchange information. It may be between processes running on the same computer or running on the different computers. By providing this service the operating system relieves the user of the worry of passing messages between processes. In case where the messages need to be passed to processes on the other computers through a network it can be done by the user programs. The user program may be customized to the specifics of the hardware through which the message transits and provides the service interface to the operating system.

Error Detection

An error is one part of the system may cause malfunctioning of the complete system. To avoid such a situation the operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning. This service cannot allow to be handled by user programs because it involves monitoring and in cases altering area of memory or deal location of memory for a faulty process. Or may be relinquishing the CPU of a process that goes into an infinite loop. These tasks are too critical to be handed over to the user programs. A user program if given these privileges can interfere with the correct (normal) operation of the operating systems.

History of UNIX Operating System:

Since it began to escape from AT&T's Bell Laboratories in the early 1970's, the success of the UNIX operating system has led to many different versions: recipients of the (at that time free) UNIX system code all began developing their own different versions in

their own, different, ways for use and sale. Universities, research institutes, government bodies and Computer Company all began using the powerful UNIX system to develop many of the technologies which today are part of a UNIX system.

Computer aided design, manufacturing control systems, laboratory simulations, even the Internet it; all began life with and because of UNIX systems. Today, without UNIX systems, the Internet would come to a screeching halt. Most telephone calls could not be made, electronic commerce would grind to a halt and there would have never been "Jurassic Park"!

By the late 1970's, a ripple effect had come into play. By now the under- and post-graduate students whose lab work had pioneered these new applications of technology were attaining management and decision-making positions inside the computer system suppliers and among its customers. And they wanted to continue using UNIX systems.

Soon all the large vendors, and many smaller ones, were marketing their own, diverging, versions of the UNIX system optimized for their own computer architectures and boasting many different strengths and features. Customers found that, although UNIX systems were available everywhere, they seldom were able to interwork or co-exist without significant investment of time and effort to make them work effectively. The trade mark UNIX was ubiquitous, but it was applied to a multitude of different, incompatible products.

In the early 1980's, the market for UNIX systems had grown enough to be noticed by industry analysts and researchers. Now the question was no longer "What is a UNIX system?" but "Is a UNIX system suitable for business and commerce?"

Throughout the early and mid-1980's, the debate about the strengths and weaknesses of UNIX systems raged, often fuelled by the utterances of the vendors themselves who sought to protect their profitable proprietary system sales by talking UNIX systems down. And, in an effort to further differentiate their competing UNIX system products, they kept developing and adding features of their own.

In 1984, another factor brought added attention to UNIX systems. A group of vendors concerned about the continuing encroachment into their markets and control of system interfaces by the larger companies, developed the concept of "open systems."

Open systems were those that would meet agreed specifications or standards. This resulted in the formation of X/Open Company Ltd whose remit was, and today in the guise of The Open Group remains, to define a comprehensive open systems environment. Open systems, they declared, would save on costs; attract a wider portfolio of applications and competition on equal terms. X/Open chose the UNIX system as the platform for the basis of open systems.

Although UNIX was still owned by AT&T, the company did little commercially with it until the mid-1980. Then the spotlight of X/Open showed clearly that a single, standard version of the UNIX system would be in the wider interests of the industry and its customers. The question now was, "which version?"

In a move intended to unify the market in 1987, AT&T announced a pact with Sun Microsystems, the leading proponent of the Berkeley derived strain of UNIX. However, the rest of the industry viewed the development with considerable concern. Believing that their own markets were under threat they clubbed together to develop their own "new" open systems operating system. Their new organization was called

the Open Software Foundation (OSF). In response to this, the AT&T/Sun faction formed UNIX International.

The ensuing "UNIX wars" divided the system vendors between these two camps clustered around the two dominant UNIX system technologies: AT&T's System V and the OSF system called OSF/1. In the meantime, X/Open Company held the center ground. It continued the process of standardizing the APIs necessary for an open operating system specification.

In addition, it looked at areas of the system beyond the operating system level where a standard approach would add value for supplier and customer alike, developing or adopting specifications for languages, database connectivity, networking and mainframe inter-working. The results of this work were published in successive X/Open Portability Guides.

XPG 4 was released in October 1992. During this time, X/Open had put in place a brand program based on vendor guarantees and supported by testing. Since the publication of XPG4, X/Open has continued to broaden the scope of open systems specifications in line with market requirements. As the benefits of the X/Open brand became known and understood, many large organizations began using X/Open as the basis for system design and procurement. By 1993, over $7 billion had been spent on X/Open branded systems. By the start of 1997 that figure has risen to over $23 billion. To date, procurements referencing the Single UNIX Specification amount to over $5.2 billion.

In early 1993, AT&T sold it UNIX System Laboratories to Novell which was looking for a heavyweight operating system to link to its NetWare product range. At the same time, the company recognized that vesting control of the definition (specification) and

trademark with a vendor-neutral organization would further facilitate the value of UNIX as a foundation of open systems. So the constituent parts of the UNIX System, previously owned by a single entity are now quite separate

In 1995 SCO bought the UNIX Systems business from Novell, and UNIX system source code and technology continues to be developed by SCO.

In 1995 X/Open introduced the UNIX 95 brand for computer systems guaranteed to meet the Single UNIX Specification. The Single UNIX Specification brand program has now achieved critical mass: vendors whose products have met the demanding criteria now account for the majority of UNIX systems by value.

For over ten years, since the inception of X/Open, UNIX had been closely linked with open systems. X/Open, now part of The Open Group, continues to develop and evolve the Single UNIX Specification and associated brand program on behalf of the IT community. The freeing of the specification of the interfaces from the technology is allowing many systems to support the UNIX philosophy of small, often simple tools that can be combined in many ways to perform often complex tasks. The stability of the core interfaces preserves existing investment, and is allowing development of a rich set of software tools. The Open Source movement is building on this stable foundation and is creating a resurgence of enthusiasm for the UNIX philosophy. In many ways Open Source can be seen as the true delivery of Open Systems that will ensure it continues to go from strength to strength.

1969 The Beginning The history of UNIX starts back in 1969, when Ken Thompson, Dennis Ritchie and others started working on the "little-used PDP-7 in a corner" at Bell Labs and what was to become UNIX.

1971 First Edition It had a assembler for a PDP-11/20, file system, fork(), roff and ed. It was used for text processing of patent documents.

1973 Fourth Edition It was rewritten in C. This made it portable and changed the history of OS's.

1975 Sixth Edition UNIX leaves home. Also widely known as Version 6, this is the first to be widely available out side of Bell Labs. The first BSD version (1.x) was derived from V6.

1979 Seventh Edition It was a "improvement over all preceding and following Unices" [Bourne]. It had C, UUCP and the Bourne shell. It was ported to the VAX and the kernel was more than 40 Kilobytes (K).

1980 XENIX Microsoft introduces XENIX. 32V and 4BSD introduced.

1982 System III AT&T's UNIX System Group (USG) release System III, the first public release outside Bell Laboratories. SunOS 1.0 ships. HP-UX introduced. Ultrix-11 Introduced.

1983 System V Computer Research Group (CRG), UNIX System Group (USG) and a third group merge to become UNIX System Development Lab. AT&T announces UNIX System V, the first supported release. Installed base 45,000.

1984 4.2BSD University of California at Berkeley releases 4.2 BSD, includes TCP/IP, new signals and much more. X/Open formed.

1984 NIT2 System V Release 2 introduced. At this time there are 100,000 UNIX installations around the world.

1986 4.3BSD 4.3 BSD released, including internet name server. SVID introduced. NFS shipped. AIX announced. Installed base 250,000.

1987 NIT3 System V Release 3 including STREAMS, TLI, RFS. At this time there are 750,000 UNIX installations around the world. IRIX introduced.

1988 POSIX.1 published. Open Software Foundation (OSF) and UNIX International (UI) formed. Ultrix 4.2 ships.

1989 AT&T UNIX Software Operation formed in preparation for spin-off of USL. Motif 1.0 ships.

1989 NIT UNIX System V Release 4 ships, unifying System V, BSD and XENIX. Installed base 1.2 million.

1990 XPG3 X/Open launches XPG3 Brand. OSF/1 debuts. Plan 9 from Bell Labs ships.

1991 UNIX System Laboratories (USL) becomes a company - majority-owned by AT&T. Linus Torvalds commences Linux development. Solaris 1.0 debuts.

1992 NIT.2 USL releases UNIX System V Release 4.2 (Destiny). October - XPG4 Brand launched by X/Open. December 22nd Novell announces intent to acquire USL. Solaris 2.0 ships.

1993 4.4 BSD 4.4 BSD the final release from Berkeley. June 16 Novell acquires USL

Late 1993 NIT.2MP Novell transfers rights to the "UNIX" trademark and the Single UNIX Specification to X/Open. COSE initiative delivers "Spec 1170" to X/Open for fasttrack. In December Novell ships NIT.2MP , the final USL OEM release of System V

1994 Single UNIX Specification BSD 4.4-Lite eliminated all code claimed to infringe on USL/Novell. As the new owner of the UNIX trademark, X/Open introduces the Single UNIX Specification (formerly Spec 1170), separating the UNIX trademark from any actual code stream.

1995 UNIX 95 X/Open introduces the UNIX 95 branding programmed for implementations of the Single UNIX Specification. Novell sells UNIXWare business line to SCO. Digital UNIX introduced. UNIXWare 2.0 ships. OpenServer 5.0 debuts.

1996 The Open Group forms as a merger of OSF and X/Open.

1997 Single UNIX Specification, Version 2 The Open Group introduces Version 2 of the Single UNIX Specification, including support for real-time, threads and 64-bit and larger processors. The specification is made freely available on the web. IRIX 6.4, AIX 4.3 and HP-UX 11 ship.

1998 UNIX 98 The Open Group introduces the UNIX 98 family of brands, including Base, Workstation and Server. First UNIX 98 registered products shipped by Sun, IBM and NCR. The Open Source movement starts to take off with announcements from Netscape and IBM. UNIXWare 7 and IRIX 6.5 ship.

1999 UNIX at 30 The UNIX system reaches its 30th anniversary. Linux 2.2 kernel released. The Open Group and the IEEE commence joint development of a revision to

POSIX and the Single UNIX Specification. First LinuxWorld conferences. Dot com fever on the stock markets. Tru64 UNIX ships.

2001 Single UNIX Specification, Version 3 Version 3 of the Single UNIX Specification unites IEEE POSIX, The Open Group and the industry efforts. Linux 2.4 kernel released. IT stocks face a hard time at the markets. The value of procurements for the UNIX brand exceeds $25 billion. AIX 5L ships.

2003-- ISO/IEC 9945--2003 The core volumes of Version 3 of the Single UNIX Specification are approved as an international standard. The "Westwood" test suite ship for the UNIX 03 brand. Solaris 9.0 E ships. Linux 2.6 kernel released.

2007-- Apple Mac OS X certified to UNIX 03.

2008-- ISO/IEC 9945--2008 Latest revision of the UNIX API set formally standardized at ISO/IEC, IEEE and The Open Group. Adds further APIs

2009-- UNIX at 40-- IDC on UNIX market -- says UNIX $69 billion in 2008, predicts UNIX $74 billion in 2013

What is Unix, Exactly

Unix is a computer operating system first developed at Bell Labs (and, to get the legal language out of the way, a trademark of AT&T Bell Laboratories). An "operating system" is a master program which coordinates other programs' activities and manages files.

One of the most popular and widespread operating systems in the world, Unix runs on more brands of computers than probably any other operating system in existence. This is partly because Unix is "portable": it is written in C, a high-level, machine-

independent language. Programs written on one Unix machine can be easily adapted to other Unix machines (C is particularly well-integrated with the operating system itself).

In addition, UNIX is based on a collection of small, easily understood utilities which allow you to connect them in many different ways (and in ways that the authors did not predict), building procedures and sophisticated tasks to suit your own needs. This "Unix philosophy" is often contrasted with monolithic programming environments (IBM mainframes or the Macintosh *** are sometimes mentioned) in which you can only perform tasks the system designers could predict; such systems, while becoming increasingly complex, often have bells and whistles you may not use, and lack those you want.

The power and flexibility of UNIX do *** not come without a cost; many people find UNIX systems extremely unfriendly. Commands and responses are not only terse -- *** some are positively opaque. Still, if you are willing to put up with a certain amount of difficulty, you may find that you are more than rewarded by the power of UNIX.

So why do you, want to use UNIX? Here are the most commonly given reasons:

* Unix allows a number of people to work on the same machine at once. It also allows multitasking ***, so each person can work on several things at once.

* It's a good environment for programmers: UNIX is portable, has excellent software tools, and has unified concepts and a flexible file system in which everything is text.

* Unix is particularly well-suited to networking; electronic mail, network news, file-transfer programs, remote logins, and terminal-to-terminal communication are very easy on UNIX systems.

* Unix systems are all over campus, and if you learn UNIX on one machine, you have an advantage when you move to any of the others.

* Unix systems are everywhere. (This is especially true in the academic world.)

INTRODUCTION:

UNIX is a **CUI** operating system. Operating System is an interface between hardware and applications software's. It serves as the operating system for all types of computers, including single user personal computers and engineering workstations, multi-ser microcomputers mini computers, mainframes and super computers as well as special purpose devices.

Features of UNIX:

Multi-user: More than one user can use the machine at a time supported via terminals (serial or network connection)

Multi-tasking: more than one program can be run at a time hierarchical directory structure to support the organization and maintenance of files.

Portability: only the kernel (<10%) written in assembler tools for program development a wide range of support tools (debuggers, compilers).

Portability: The system is written in high-level language making it easier to read, understand, change and, therefore move to other machines. The code can be changed and complied on a new machine. Customers can then choose from a wide variety of hardware vendors without being locked in with a particular vendor.

Machine-independence: The System hides the machine architecture from the user, making it easier to write applications that can run on micros, mini and mainframes

Hierarchical File System: UNIX uses a hierarchal file structure to store information. This structure has the maximum flexibility in grouping information in a way that reflects its natural state. It allows for easy maintenance and efficient implementation.

Multiprogramming -Support more than one program, in memory, at a time. Amounts to multiple user processes on the system.

Operating System is a collection of **"System Software Programs"** which is coordinating the actions of the computer. That is allocating the resources of the system. Operating System is a software component. There exist two types of Soft wares. They are

1. **Application Software**
2. **System Software**

1) The software used by the users of the computer, are known as **Application Software.** E.g.: word, excel, power point, commands, database packages etc...

2) The software used by the computer is known as **System Software**. E.g.: Compilers, Loaders, OS, and Interpreters etc... UNIX and Linux are two different Operating Systems. There exist different types of OS. For this we have to see the history of Operating Systems.

Operating System and its Types:

1. Single User and Single tasking OS: In late 1979's DOS was invented by Microsoft Corporation. It is a "Single User and **Single Tasking OS".** That means "At a moment of time only one user is allowed to work with the computer and he is able to do only one task". The important feature of **DOS** is **CUI** (Character User Interface).

2. Single User and Multi tasking OS: In late 1983's another OS, **WINDOWS** was developed and released by **Microsoft corp.** It is a **"Single User and Multi Tasking OS"**. That means "At a moment of time "only one user is allowed to work with the system and he can be able to do more than one work". The important feature of **WINDOS** is **GUI** (Graphical User Interface).

3. Multi user and Multi tasking OS: In late 1960's **UNIX** was invented by **Dennis Ritchie** [founder of C Language] at Bell Laboratories. It is a **"Multi-user and Multi tasking OS"**. That means "At a moment of time More than one user is allowed and he is able to do more than one task". The important feature of **UNIX** is **CUI** (Character User Interface). Later, the software component that supports the graphical objects was developed by Massachusetts University, which is named as **XWindows**. While installing the UNIX os if you can check the Option **XWindows, the UNIX** will be installed in GUI Mode.

Types of UNIX: There are many different versions of UNIX, although they share common similarities. The most popular varieties of UNIX are Sun Solaris, GNU/Linux, and MacOS X. Here in the School, we use Solaris on our servers and workstations, and Fedora Linux on the servers and desktop PCs.

The UNIX operating system is made up of three parts; the kernel, the shell and the programs.

The kernel

The kernel of UNIX is the hub of the operating system: it allocates time and memory to programs and handles the file store and communications in response to system calls. As an illustration of the way that the shell and the kernel work together, suppose a user types rm myfile (which has the effect of removing the file myfile). The shell searches the filestore for the file containing the program rm, and then requests the kernel, through system calls, to execute the program rm on myfile. When the process rm myfile has finished running, the shell then returns the UNIX prompt % to the user, indicating that it is waiting for further commands.

The shell

The shell acts as an interface between the user and the kernel. When a user logs in, the login program checks the username and password, and then starts another program called the shell. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for them to be carried out. The commands

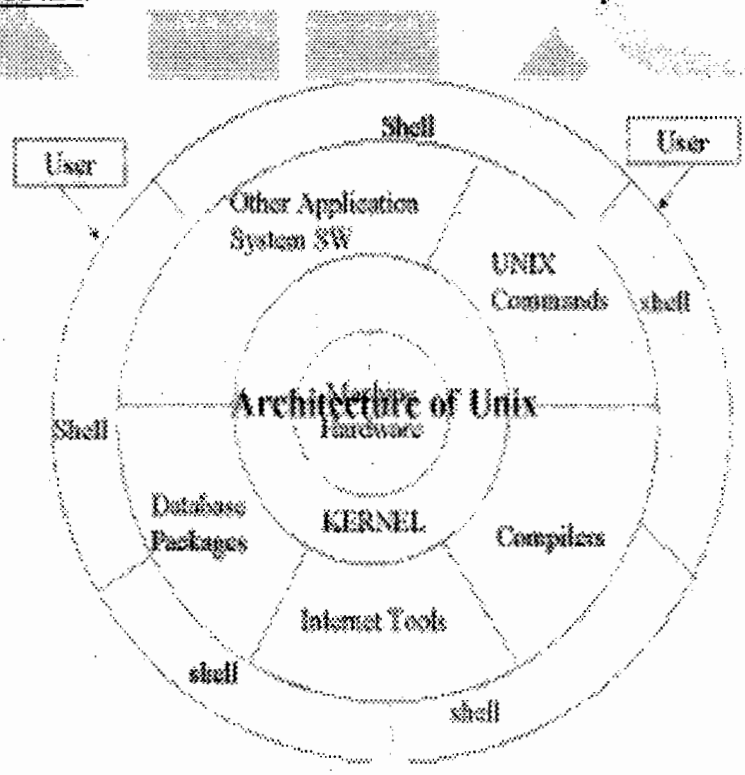
are themselves programs: when they terminate, the shell gives the user another prompt (% on our systems).

The adept user can customize his/her own shell, and users can use different shells on the same machine. Staff and students in the school have the tcsh shell by default. The tcsh shell has certain features to help the user inputting commands.

Filename Completion -

By typing part of the name of a command, filename or directory and pressing the [Tab] key, the tcsh shell will complete the rest of the name automatically. If the shell finds more than one name bExinning with those letters you have typed, it will beep, prompting you to type a few more letters before pressing the tab key again.

Architecture of UNIX:



Kernel: Kernel is a part of the UNIX/Linux OS and is loaded in to the memory when the system is booted. It manages the system resources, allocates time between users and processes, decides process priorities, and performs all other tasks. Kernel is an interface between shell and hardware.

Shell: Shell is an interface between user and kernel. And it is the "Command Interpreter". In UNIX/Linux there exist different types of shells. They are In **UNIX**

1. Bourne Shell developed by Stephen Bourne
2. Korn Shell developed by David Korn
3. C Shell Developed by Ian Bell

In **Linux:**

3. Bash Shell is nothing but Bourne again
4. tcsh shell is nothing but turbo c shell

/bin: This directory is the home of binary executables. These include the common commands we have already learned like ls, cat, gzip and tar.

Different Flavors of UNIX

- BSD (Berkeley Software Distribution)
- Sun Solaris (Sun Micro Systems)
- Novell Netware (NOVELL)
- IBM SIX (International Business Machines)
- HP-UX (Hewllet Packard)
- TRU 64 (Hewllet Packard)
- LINUX

UNIX File System Architecture

In UNIX Operating System, a hard disk partition is considered as any array of disk block logically; where as disk block can be a physical sector or multiples of physical sectors on the disk. It contains four important areas, they are

BOOT BLOCK

SUPER BLOCK

INODE BLOCK

DATA BLOCK

Boot Block: these stores bootable objects that are necessary for booting the system. It has completely bootable information.

Super Block: This Stores all the information about the file system like

Size and status of the file system

Size of file system logical block

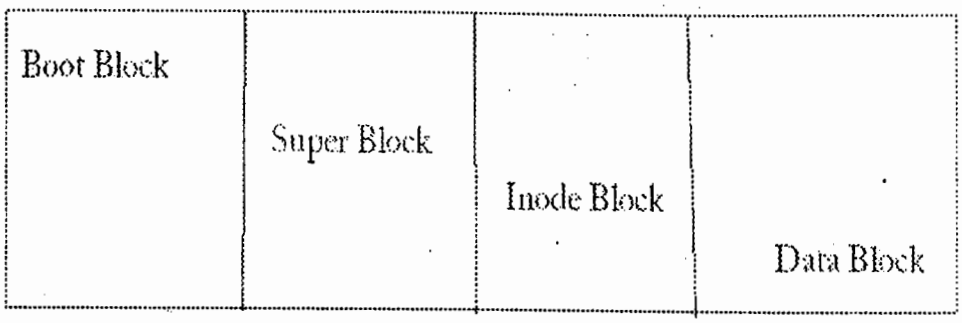
Date and time of the last update

Number blocks allocated

Number blocks unallocated

File System State

UNIX FILE SYSTEM ARCHITECTURE



Inode Block: An inode contains all the information about a file except its name, which is kept in directory. It contains the following

The type of the file and the mode of the file

The number of hard links to the file

Owner of the file and the number of bytes of the file

The data and time last accessed

The date and last modified and the date and time last created

Data Block: Data blocks are storage blocks contains the rest of the space that is allocated to the file system.

UNIX Command Syntax:

- All commands have a similar format
- Commands are generally two to five characters long
- Commands are case sensitive
- Options always preceded filenames
- Options are prefixed using a - symbol
- The man command can be used to display the correct syntax
- If you make typing mistake press backspace to erase characters.
- To cancel entire the command before you press enter, press the delete key.
- Don't turn off the computer if you have made mistake press ctrl+d.

UNIX Command Line Structure:

A command is a program that tells the UNIX system to do something. It has the form:

Command [options] [arguments]

Where an argument indicates on what the command is to perform its action, usually a file or series of files. An option modifies the command, changing the way it performs.

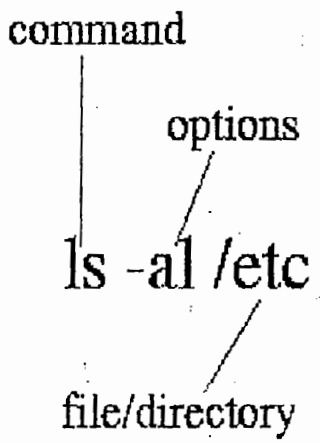
Commands are case sensitive. Command and Command are not the same.

Options are generally preceded by a hyphen (-), and for most commands, more than one option can be strung together, in the form:

Command - [option][option][option]

These are the standard conventions for commands. However, not all UNIX commands will follow the standard. Some don't require the hyphen before options and some won't let you group options together, i.e. they may require that each option be preceded by a hyphen and separated by white space from other options and arguments.

Options and syntax for a command are listed in the man page for the command.



Control Keys

Control keys are used to perform special functions on the command line or within an editor. You type these by holding down the Control key and some other key simultaneously. This is usually represented as ^Key. Control-S would be written as

^S. With control keys upper and lower case are the same, so ^S is the same as ^s. This particular example is a stop signal and tells the terminal to stop accepting input. It will remain that way until you type a start signal, ^Q.

Control-U is normally the "line-kill" signal for your terminal. When typed it erases the entire input line.

In the VI editor you can type a control key into your text file by first typing ^V followed by the control character desired, so to type ^H into a document type ^V^H.

$stty - terminal control

stty reports or sets terminal control options. The "tty" is an abbreviation that harks back to the days of teletypewriters, which were associated with transmission of telExraph messages, and which were models for early computer terminals.

For new users, the most important use of the stty command is setting the erase function to the appropriate key on their terminal. For systems programmers or shell script writers, the stty command provides an invaluable tool for configuring many aspects of I/O control for a given device, including the following:

- erase and line-kill characters
- data transmission speed
- parity checking on data transmission
- hardware flow control
- newline (NL) versus carriage return plus linefeed (CR-LF)
- interpreting tab characters
- edited versus raw input
- mapping of upper case to lower case

This command is very system specific, so consult the man pages for the details of the stty command on your system.

Syntax

stty [options]

Options

(none) report the terminal settings

all (or -a) report on all options

echoe echo ERASE as BS-space-BS

dec set modes suitable for Digital Equipment Corporation operating systems (which distinguishes between ERASE and BACKSPACE) (Not available on all systems)

kill set the LINE-KILL character

erase set the ERASE character

intr set the INTERRUPT character

Examples

You can display and change your terminal control settings with the stty command. To display all (-a) of the current line settings:

% stty -a

You can change settings using stty, e.g., to change the erase character from ^? (the delete key) to ^H:

% stty erase ^H

This will set the terminal options for the current session only. To have this done for you automatically each time you login, it can be inserted into the **.login or .profile** file that we'll look at later.

Getting Help

The UNIX manual, usually called **man pages,** is available on-line to explain the usage of the UNIX system and commands. To use a man page, type the command "**man**" at the system prompt followed by the command for which you need information.

Syntax: man [options] command_name

Common Options

-k keyword list command synopsis line for all keyword matches

-M path path to man pages

-a show all matching man pages (NIT)

Examples

You can use man to provide a one line synopsis of any commands that contain the keyword that you want to search on with the "-k" option, e.g. to search on the keyword password, type:

$man -k password

passwd (5) - password file

passwd (1) - change password information

The number in parentheses indicates the section of the man pages where these references were found. You can then access the man page (by default it will give you the lower numbered entry, but you can use a command line option to specify a different one) with:

$man passwd

Login procedure:-

When switch on your system it will ask your Login name and Password, that is as follows

Login : NareshTech

Passwd : NareshTech

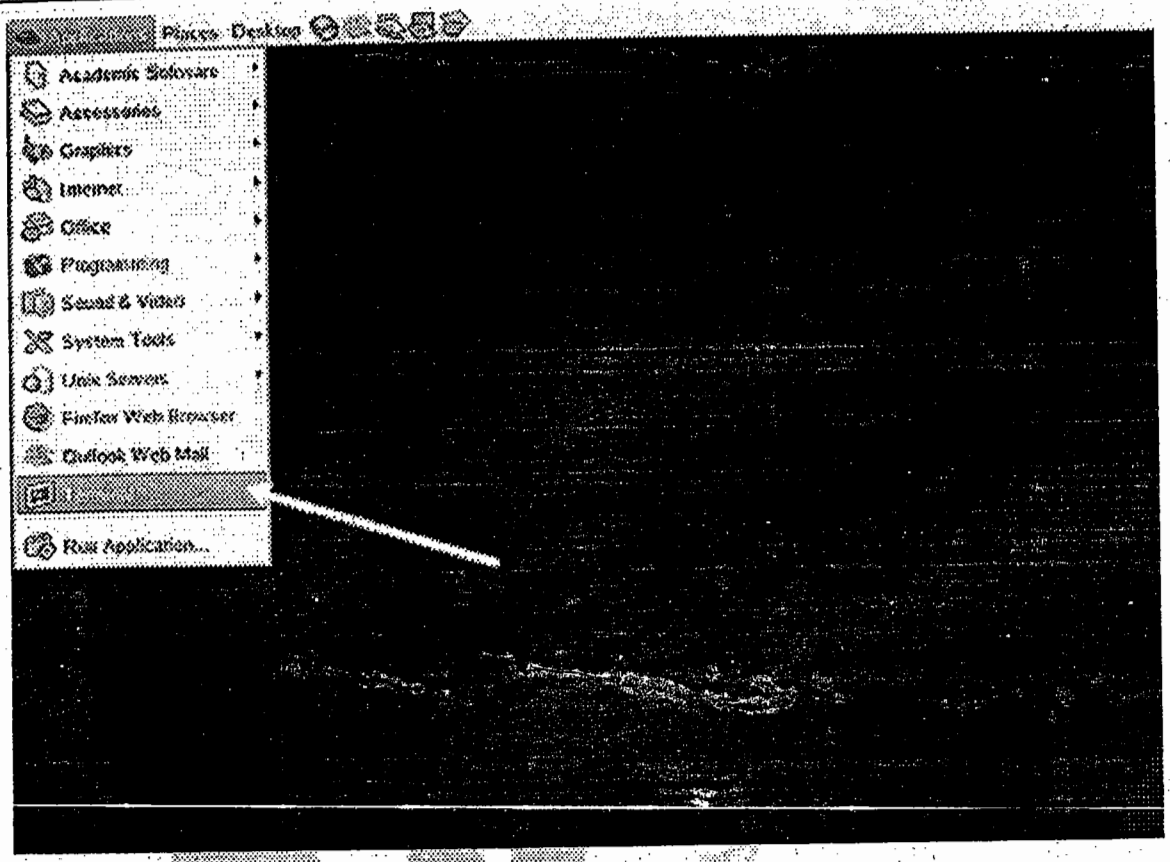
After you had entered correct Login name and password, then it will show

 # → System Administrator Prompt

$ → User Working Prompt

Starting an UNIX terminal:

To open an UNIX terminal window, click on the "Terminal" icon from Applications/Accessories menus.



How to create a new user:

#useradd username

Example: useradd NareshTech

How to create a passwd

#passwd username

Example: passwd NareshTech

Different UNIX Commands:

$date: It is used to display current system date and time.

Syntax: date [options] [+format]

Example: $date <enter>

Mon Jan 1 04:30 IST 2006

Common Options:

$date +%d → It displays the current system date

$date +%m → It displays the current system month

$date +%y → It displays the current system year in short form

$date +%Y → It displays the current system year in full form

$date +%H → It displays the current system time in number of hours

$date +%M → It displays the current system time in number of minutes

$date +%S → It displays the current system time in number of seconds

$date +%a → It displays the current system day in short form

$date +%A → It displays the current system day in full form

$date +%b → It displays the current system month in short form

$date +%B → It displays the current system month in full form

$cal: It displays previous month, current month and next month calendar.

Syntax: $cal [options] [month] [year]

Example: $cal 8 2002

Cal command will print the calendar on current month by default. If you want to print calendar of August of 2002.

August 2002

S M Tu W Th F S

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

$cal year: It will display the specified year calendar.

Ex1: $cal 2000 (it takes year from 1 to 9999)

$who: This command displays a list of the people using the system.

Syntax: $who [options]

Example:

NareshTech	tty01	Jan 1 04:00
Neo	tty02	Jan 2 04:50
NareshTech123	tty03	Jan 1 06:00
Neo123	tty04	Jan 6 04:00

NareshTech → Login Name

Tty01 → Terminal Number

Jan 1 → Mon and Date

04:00 → Time

$whoami: It will display the name of the current user, terminal number date and time at which you logged into the system.

Syntax: $whoami [options]

Example:

NareshTech tty01 Jan 1 04.00

$clear: It is used to clear the screen.

$pwd: It will display current working directory name.

Syntax: $pwd [options]

Example: /home/NareshTech/Neo123

$su: Stands for Switch User, you want switch, when you are switching user to admin, we should enter password, if we are switching Admin to user , should not required password.

Syntax: **$su [options] [username/admin name]**

Example: $su NEO

Enter Password: ******

$passwd: It is used to change the password of the current user. After executing this command it will first ask the old password of the current user and then it will wait for the new password. Then enter new password and again enter new password. After doing like this the system will set the new password for the current user.

Syntax: $passwd [options]

Example:

$Enter current Passwd

$Enter New Passwd

$Confirm New Passwd

$banner: This will display the character or word followed this command in the banner like format, banner prints characters in a sort of ASCII art poster, for example to print wait in big letters. I will type banner wait at UNIX command line or in my script. This is how it will look.

Ex: **$banner** WAIT

```
          #       #     ######
#    #   #  #      #       #
#    #  #    #     #       #
# ##.#  ######     #       #
##  ##  #    #     #       #
#    #  #    #     #       #
```

$exit: To logout from current user

Administrator can connect any user without password. But a user wants the password to go for any other user.

^D - indicates end of data stream; can log a user off. The latter is disabled on many systems

^C - interrupt

logout - leave the system

exit - leave the shell

$tty: Tty command will display your terminal.

Syntax: $tty [options]

Options

-l will print the synchronous line number.

-s will return only the codes: 0 (a terminal), 1 (not a terminal), 2 (invalid options) (good for scripts)

Changing Run Levels: (#init)

Admin always work in the /root

1). #init 0 → to shutdown the system

2). #init 1 → to bring the user into single user mode

3). #init 2 → to bring system into multi-user mode

Note: Here system resources are not possible to share

4). #init 3 → to bring the system into multi-user mode

Note: Here system resources are possible to share

5). #init 6 → to restart the system

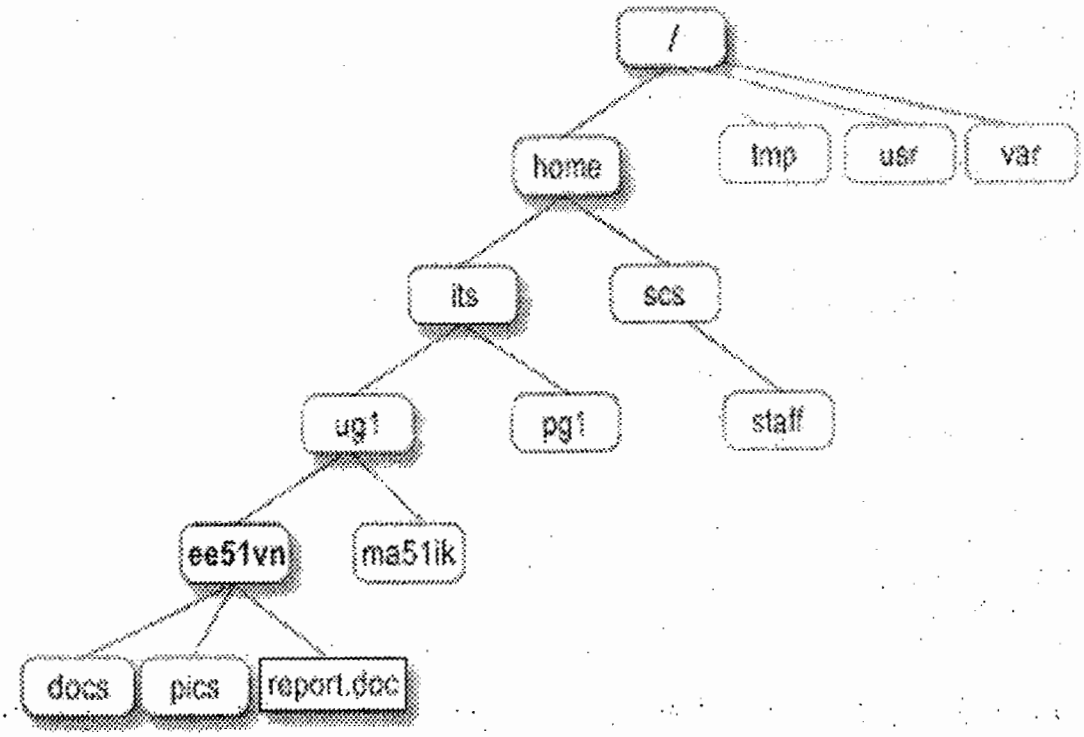
How to inter shift from GUI to CUI

GUI to CUI	**CUI to GUI**
Ctrl+alt+F1	
Ctrl+alt+F2	Ctrl+alt+F7
Ctrl+alt+F3	
Ctrl+alt+F4	
Ctrl+alt+F5	
Ctrl+alt+F6	

The Directory Structure

All the files are grouped together in the directory structure. The file-system is arranged in a hierarchical structure, like an inverted tree. The top of the hierarchy is traditionally called root (written as a slash /). In UNIX/Linux there exists some standard directories; they can vary among the versions of these OS. Here the structure of the directories is Hierarchical structure. This structure is also called as UNIX File System Structure, because directories are also called as Directory Files. The structure is as follows.

In the diagram below, we see that the home directory of the undergraduate student "ee51vn" contains two sub-directories (docs and pics) and a file called report.doc.
The full path to the file report.doc is "/home/its/ug1/ee51vn/report.doc"



Directory Related Commands:

$mkdir (make directory):

This command is used to create a directory with the specified name.

Syntax: $mkdir [options] [dirname (s)]

Example: $ mkdir NEO

This command creates a directory with name NEO

Change Directory:

$cd: Stands for change directory this command is used to change a directory from current directory to the specified directory.

Syntax: $ cd [options] [target-dirname]

Ex:$ cd NEO

$pwd

/usr/NareshTech/NEO

The above command moves the user from the current directory to the directory **NEO**

Note: Whenever a user using the above command, he has to notice the current working directory. That is why because you can't delete a directory in two cases.

a) If the user is working with that directory.

b) If that directory contains any files.

.. Represents home directory of the current directory.

. Represents current directory.

Examples

cd (also chdir in some shells) change directory

cd changes to user's home directory

cd / changes directory to the system's root

cd .. goes up one directory level

cd ../.. goes up two directory levels

cd /full/path/name/from/root changes directory to absolute path named (note the leading slash)

cd path/from/current/location changes directory to path relative to current location (no leading slash)

cd ~username/directory changes directory to the named username's indicated directory.

$rmdir : This command is used to remove the specified directory, when it is empty.

Syntax:$rmdir [options] [dirname]

Ex: **$ rmdir NEO**

This command deletes the directory **NEO**

Disk Related Commands:

The system Administrator has to regularly monitor the integrity of the file system and the amount of disk space available.

Checking Disk Free Space:

If we want to see how much of the disk is being used and what part of it lies free. UNIX has for us a command called **df** (for disk free). This command reports the free as well as the used disk space for all the file systems installed on your machine. We have on our machine only one file system installed, the root file system or simply **/dev/root**. **df** reports the number of free disk blocks and free inodes for this file system.

Syntax: df

Example: $df

If we want more detailed information about disk usage we should say,

$df -ivt

Dfspace makes more sense:

The above command reports the free disk in terms of megabytes and percentage of total disk space.

$dfspace

dfspace not found

It came because the **dfspace** command is present in **/etc** directory.

Example: $/etc/dfspace

Disk Usage- The du command:

du sounds similar to **df** but is different in its working. **df** and **dfspace** report the disk space available in the file system as a whole whereas **du** reports the disk space used by specified files and directories.

Example: $du

Exampe: $du /hdev

The file Command:

If we want to see all the files from current directory then we have type the following command.

$file *

file - file type

This command displays several types of files. This program, files, examines the selected file and tries to determine what type of file it is. It does this by reading the first few bytes of the file and comparing them with the table in /etc/magic. It can determine ASCII text files, tar formatted files, compressed files, etc.

Syntax: $file [options] [-m magic_file] [-f file_list] file

Common Options

-c check the magic file for errors in format

-f file_list file_list contains a list of files to examine

-h don't follow symbolic links (SVR4 only)

-L follow symbolic links (BSD only)

-m magic_file use magic_file as the magic file instead of /etc/magic

Files and Users

FILES: A file is nothing but a collection of data. That means it contains some data. In UNIX/Linux, we are having three types of files. They are

1. Directory Files.
2. Ordinary Files.
3. Special Files or Device Files.

A file that contains at least one file with in that is called as a Directory File.

A file that is not containing any file with in it is called as an Ordinary File.

A file that contains some files that are related to devices of the system devices, they are known as Device Files.

File Related commands

$cat (Concatenate) this command is used for three purposes as follows.

a) To create a file.

b) To display file data.

c) To append data at the end of the file.

Common Options

-n precede each line with a line number

-v display non-printing characters, except tabs, new-lines, and form-feeds

-e display $ at the end of each line (prior to new-line) (when used with -v option)

a) To create a file, user has to do as follows.

Syntax: **cat > filename**

I) Ex: $ cat > NiT

II) After typing the above command the shell waits for entering data, and then enters your required data.

III) To quit from the cat command press <ctrl + d >, means saving and quit from the cat or file.

b) To display the file contents

Example: $cat < NiT

Example: $cat NiT NiT1 NiT2 NiT3: →The above command shows the data of the specified files.

c) To append data at the end of the file.

Syntax: $ cat >> filename

Ex: $ cat >> NiT

By doing this, the shell waits for entering data. This data will be added at the end of the file **managers**. After typing your required data press **ctrl+d**.

$touch - create a file

The touch command can be used to create a new (empty) file or to update the last access date/time on an existing file. The command is used primarily when a script requires the pre-existence of a file (for example, to which to append information) or when the script is checking for last date or time a function was performed.

Syntax

touch [options] [date_time] file

touch [options] [-t time] file

Common Options

-a change the access time of the file (SVR4 only)

-c don't create the file if it doesn't already exist

-f force the touch, regardless of read/write permissions

-m change the modification time of the file (SVR4 only)

-t time use the time specified, not the current time (SVR4 only)

When setting the "-t time" option it should be in the form:

[[CC]YY]MMDDhhmm[.SS]

$touch <File Name> → to create an empty file.

$ touch NiT NeO file3 → To create multiple empty files

Syntax: $touch [options] [filename (s)]

Example: $touch Neo Neo1 Neo2 Neo3

Later we can insert data by using **$cat** command

Creating Hidden files:

Which file is creating starting with dot (.) is called hidden file.

Example: $cat > .<filename>

The above example creates hidden file with a specified name

Hide Existing file

$mv NiT .NiT

The above command hide the existing file.

Unhide the files

$mv .NiT NiT

The above command unhide the file.

Removing files:

$rm: To remove the given files

Syntax: $rm [options] [filename (s)]

Example $rm NiT → It removes NiT file

Example $rm –i NiT → It asks the conformation before deleting file.

Do u want remove NiT? Y→ It removes

? n→ It won't removes

$rm NiT NiT1 NiT2 NiT3→ It removes all these files

$rm * → It removes all files from current directory

$rm –r [Dir Name] → It removes all files, directories and sub directories including specified directory.

Common Options

-i interactive (prompt and wait for confirmation before proceeding)

-r recursively remove a directory, first removing the files and subdirectories beneath it

-f don't prompt for confirmation (overrides -i)

The shift Command

The shift command allows you to effectively left shift your positional parameters. If you execute the command

shift

whatever was previously stored inside $2 will be assigned to $1, whatever was previously stored in $3 will be assigned to $2, and so on. The old value of $1 will be irretrievably lost.

When this command is executed, $# (the number of arguments variable) is also automatically decremented by one:

```
$ cat tshift          Program to test the shift
echo $# $*
```

```
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
shift
echo $# $*
$ tshift a b c d e
5 a b c d e
4 b c d e
3 c d e
2 d e
1 e
0
$
```

If you try to shift when there are no variables to shift (that is, when $# already equals zero), you'll get an error message from the shell (the error will vary from one shell to the next):

```
prog: shift: bad number
```

where prog is the name of the program that executed the offending shift.

You can shift more than one "place" at once by writing a count immediately after shift, as in

```
shift 3
```

This command has the same effect as performing three separate shifts:

```
shift
shift
shift
```

The shift command is useful when processing a variable number of arguments

Listing Files and Directories:

ls (stands for listing): This command is used to display all files names under current directory.

Options:

$ls -l → It displays long listing of files. It displays all files and directories with following attributes.

i). Number of blocks allocated

ii). Type of the file

iii). File permissions

iv). Number of links

v). Owner of the file

vi). Group of the file

vii). Number of bytes allocated

viii). File created time and date

ix). File Name

$ls -r → It displays all files and directories in reverse order

$ls -i → It displays inode numbers and respective names.

$ls -x → It displays all files and directories width wise

$ ls -a → As you can see, ls -a lists files that are normally hidden

ls does not, in fact, cause all the files in your home directory to be listed, but only those ones whose name does not begin with a dot (.) Files beginning with a dot (.) are known as hidden files and usually contain important program configuration information. They are hidden because you should not change them unless you are very familiar with UNIX. To list all files in your home directory including those whose names begin with a dot.

$ls -F: → It displays all files, directories, executable file and symbolic files.

NiT\ → Directory file

NEO * → Executable file

UNIX @ → Linked file etc.,

CHARACTER IF ENTRY IS A

d directory

- plain file

b block-type special file

c character-type special file

l symbolic link

s socket

Copying Files:

$cp NiT NeO is the command which makes a copy of NiT in the current working directory and calls it NeO. If NeO already existed it overwrites.

Syntax: $cp [options] [source file name] [target filename]

Example: $cp NiT NeO

Example: $cp –i NiT NeO → if NeO already existed then it asks the conformation

Does u want to overwrite NeO?

Example: cp /home/neo123/NiT neo/ (Target filename is not mandatory)

Rename a file (Moving files)

$mv(move): This command rename the given file name with the given new name. We can also move files physically from one directory to another directory.

Syntax: $mv [options] [source] [target]

Example: $mv NiT NeO

To move a file from one place to another, use the **mv** command. This has the effect of moving rather than copying the file, so you end up with only one file rather than two

Different Types of files

- \- Ordinary File
- d Directory File
- b Special block File
- c Special Character file
- L Symbolic File

Different file symbols

Ended with / → Directory, Ended with @→ Link file, Ended with *-→ Exe file

Ended with any special character is ordinary file.

Linux Shows Colors

Sky blue → Link file, Dark blue → Directory, Green → EXE file, Red → Zip file

Black or white → ordinary file

$wc (word count): It is used to count and display the number of lines, words and characters of the specified file and it display the file name also.

Syntax:-$wc [options] [file-name (s)]

Example: $wc NiT → It displays the following output

15	**30**	**78**	**NiT**

1. 15→ Number of lines
2. 30 → Number of words
3. 78 → Number characters
4. NiT → Name of the file

Example: $wc NiT NeO BiT

10	20	67	NiT
62	98	200	NeO
87	78	98	BiT
159	196	361	Total

Examples:

i). $wc -l NiT→ It displays 10 lines

ii). $wc -w NiT→ It displays 20 words

iii). $wc -c NiT→ It displays 67 characters

iv). $wc -wc NiT→ It displays 20 words and 67 characters

vi). $wc -wl NiT→ It displays 20 words 10 lines

Vii). $wc -lc NiT→ It displays 10 lines and 67 characters.

Links

A link is not a kind of file but instead is second name for a file. When a file had two links, it is not physically present at two places, but can be referred to by either of the names. This command links the specified files, in affect if the user made any changes to any one of these file, then those changes will affect the both files.

File links are divided into two types:

1. Hard links
2. Soft links

Hard links: A file can have as many as many names as you want to give it, but the only thing that is common to all of them they all have the same inode number.

Syntax: $ ln [source file] [target file]

Ex: $ ln NiT NeO

Here -I options shows that they have the sane inode number, that means they are actually on and the same file.

$ls -i

1536	-rw-r--r--	2	NiT	NeO	81	August	22	10:20	NiT
1536	-rw-r--r--	2	NiT	NeO	81	August	22	10:20	NeO

The number of links which is normally one for unlinked files, is shown to be two. All these linked files have equal status, its not that one file contains the original data and the others don't, changes made in one file will be affected automatically in the other file for the simple reason that there is only single copy of the file in the disk.

These links have two limitations

1. You can't have two linked files in two file system
2. You can't link a directory even within the same file system

Soft links: These are links are also known as soft links. These soft links were first introduced by Berkeley with BSD UNIX, but it is now available in all flavors of UNIX. Unlike hard links, a symbolic link points to the file which actually have contents. In-

fact, Windows shortcuts are more like symbolic links. You can create a symbolic link with the -s option of ln, but this time the listing tells you the different story.

Syntax: $ln -s [source file] [target file]

Example: $ln -s NiT NeO

$ls -i

2536	-rw-r--r--	2	NiT NeO	81	August	22	10:25	NiT
2538	-lrwxrwxrwx	2	NiT NeO	81	August	22	10:20	NeO

You can identify symbolic links by the character 1 seen in the permission field. It supports directory linking also.

Unlink: The name itself indicates that it is used for removing the links from the files and the directories. This command removes the links of the files and directories either it could be a soft link or hard link.

Syntax: $unlink [option] [filename]

Example: $unlink NeO

Here, it removes the links which are associated with the given file **NeO**

File comparison commands

$cmp: This command compares the given two files, byte by byte and displays the differed lines with their filenames. The **cmp** command compares two files, and (without options) reports the location of the first difference between them. It can deal with both binary and ASCII file comparisons. It does a byte-by-byte comparison.

Syntax: $cmp [options] file1 file2 [skip1] [skip2]

The skip numbers are the number of bytes to skip in each file before starting the comparison.

Common Options

-l report on each difference

-s report exit status only, not byte differences **Syntax: $ cmp [option] [NiT] [NeO]**

Ex: $ cmp NiT NeO

Note: If, it doesn't display any output, that means the files are same.

$comm: This command requires two sorted files, and it compares the data of these files, line by line and display.

Syntax: $comm [options] [NiT] [NeO]

Example: $comm NiT NeO

$Diff: Display which lines in one file have to be changed to make two files identical. The diff command compares two files, directories, etc, and reports all differences between the two. It deals only with ASCII files. Its output format is designed to report the changes necessary to convert the first file into the second.

Syntax

diff [options] file1 file2

Common Options

-b ignore trailing blanks

-i ignore the case of letters

-w ignore <space> and <tab> characters

-e produce an output formatted for use with the editor, ed

-r apply diff recursively through common sub-directories

Syntax: $diff [option] [NiT] [NeO]

Example: $diff NiT NeO

$Uniq: This command compares the lines with in a single file. This reads a line and the reads the second line, if the second is similar to the first, then it discards the display of second line.

Syntax: $uniq [options] [+|-n] file [file.new]

Common Options

-d one copy of only the repeated lines

-u select only the lines not repeated

+n ignore the first n characters

-s n same as above (SVR4 only)

-n skip the first n fields, including any blanks (<space> & <tab>)

-f fields same as above (SVR4 only)

Syntax: $uniq filename

Ex: $uniq NiT

Options:

-u out put only the unique lines

-c out put only the single copy of each line with a number to its left indicating the number of times that line is in the input.

-d out put only the single copy of the duplicated lines

Wild card characters or Meta characters

Regular Expression Metacharacters

Metacharacter	Function	Example	What It Matches
^	Beginning-of-line anchor	/^love/	Matches all lines beginning with love
$	End-of-line anchor	/love$/	Matches all lines ending with love
.	Matches one character	/l..e/	Matches lines containing an l, followed by two characters, followed by an e
*	Matches zero or more of the preceding characters	/ *love/	Matches lines with zero or more spaces, followed by the pattern love
[ ]	Matches one in the set	/[LI]ove/	Matches lines containing love or Love
[x-y]	Matches one character within a range in the set	/[A-Z]ove/	Matches letters from A through Z followed by ove
[^]	Matches one character not in the set	/[^A-Z]/	Matches any character not in the range between A and Z
\	Used to escape a metacharacter	/love\./	Matches lines containing love, followed by a literal period; Normally the period matches one of any character

In UNIX/Linux some characters in our key board are having special meaning according to the shell. They are

? → Represent zero or single occurrence of any character.

$ls ? → It displays all single character files

$ls ?? → It displays all double character files

$ls N?T → It displays all files which are starting with N and Ending With T but file length should be three characters.

* →Represent zero or single or more than one Occurrences of any character.

$ls * → It displays all files

$ls N*T → It displays all files which are starting with N and Ending With T

List: [ch1ch2] Represent either ch1 or ch2.

$ls [abcdef]* → It displays all files which are starting with specified list

$ls [!abcdef]* → It displays all files which are not starting with specified list

Range: [ch1-ch2] Represent a single character with in the specified range.

[!ch1-ch2]Represent neither of the characters with in the range.

$ls [a-g] * → It displays all files between specified range

Printing Statements

$echo It is used to display the text after the echo. But it can be differed when we use the quoting characters and shell variables.

The single quote character '

The double quote character "

The backslash character \

The back quote character `

The Single Quot:

There are several reasons that you might need to use quotes in the shell. One of these is to keep characters otherwise separated by whitespace characters together. Let's look at an example. Here's a file called phonebook that contains names and phone numbers:

$ cat phonebook

Alice Chebba 973-555-2015

Barbara Swingle 201-555-9257

Liz Stachiw 212-555-2298

Susan Goldberg 201-555-7776

Susan Topple 212-555-4932

```
Tony Iannino   973-555-1295

$
```

To look up someone in our phonebook file—which has been kept small here for the sake of example—you use grep:

```
$ grep Alice phonebook

Alice Chebba   973-555-2015

$
```

Look what happens when you look up Susan:

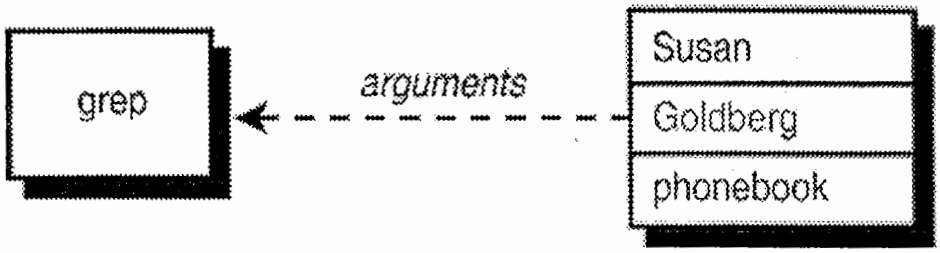
```
$ grep Susan phonebook

Susan Goldberg  201-555-7776

Susan Topple   212-555-4932

$
```

There are two lines that contain Susan, thus explaining the two lines of output. One way to overcome this problem would be to further qualify the name. For example, you could specify the last name as well:

```
$ grep Susan Goldberg phonebook

grep: can't open Goldberg

Susan Goldberg  201-555-7776

Susan Topple   212-555-4932

$
```

Recalling that the shell uses one or more whitespace characters to separate the arguments on the line, the preceding command line results in grep being passed three arguments: Susan, Goldberg, and phonebook

grep Susan Goldberg phonebook.



When grep is executed, it takes the first argument as the pattern and the remaining arguments as the names of the files to search for the pattern. In this case, grep thinks it's supposed to look for Susan in the files Goldberg and phonebook. So it tries to open the file Goldberg, can't find it, and issues the error message:

grep: can't open Goldberg

Then it goes to the next file, phonebook, opens it, searches for the pattern Susan, and prints the two matching lines. The problem boils down to trying to pass whitespace characters as arguments to programs. This can be done by enclosing the entire argument inside a pair of single quotes, as in

grep 'Susan Goldberg' phonebook

When the shell sees the first single quote, it ignores any otherwise special characters that follow until it sees the closing quote.

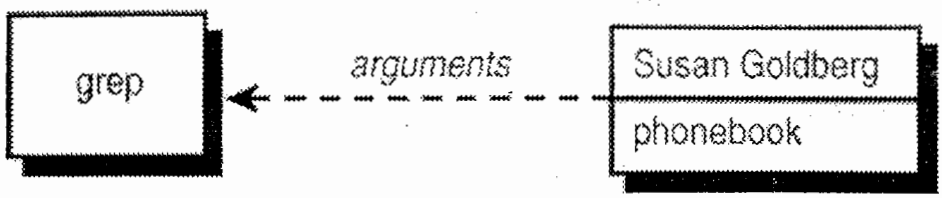
$ grep 'Susan Goldberg' phonebook

Susan Goldberg 201-555-7776

$

In this case, the shell encountered the first ', and ignored any special characters until it found the closing '. So the space between Susan and Goldberg, which would have normally delimited the two arguments, was ignored by the shell. The shell therefore divided the command line into two arguments, the first Susan Goldberg (which includes the space character) and the second phonebook. It then executed grep, passing it these two arguments.

grep 'Susan Goldberg' phonebook.



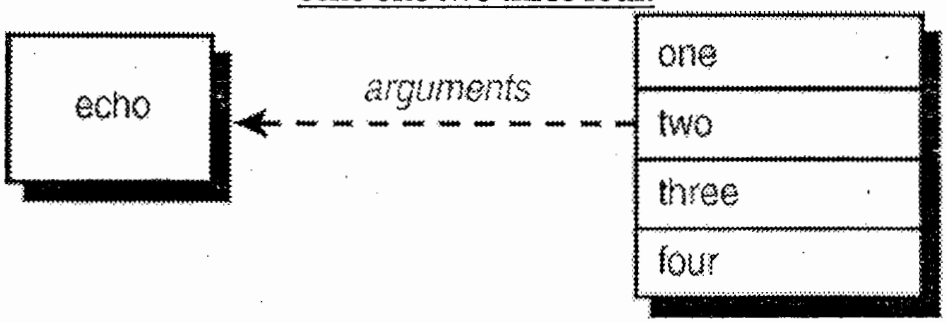
grep then took the first argument, Susan Goldberg, and looked for it in the file specified by the second argument, phonebook. Note that the shell removes the quotes from the command line and does not pass them to the program.

No matter how many space characters are enclosed between quotes, they are preserved by the shell.

```
$ echo one    two    three   four

one two three four

$ echo 'one    two    three   four'

one    two    three   four

$
```

In the first case, the shell removes the extra whitespace characters from the line and passes echo the four arguments one, two, three, and four

echo one two three four.



As we mentioned, all special characters are ignored by the shell if they appear inside single quotes. That explains the output from the following:

```
$ file=/users/steve/bin/prog1
```

```
$ echo $file
/users/steve/bin/progl
$ echo '$file'        $ not interpreted
$file
$ echo *
addresses intro lotsaspaces names nu numbers phonebook stat
$ echo '*'
*
$ echo '< > | ; ( ) { } >> " ` &'
< > | ; ( ) { } >> " ` &
$
```

Even the Enter key will be ignored by the shell if it's enclosed in quotes:

```
$ echo 'How are you today,
> John'
How are you today,
John
$
```

After typing the first line, the shell sees that the quote isn't matched, so it waits for you to type in the closing quote. As an indication that the shell is waiting for you to finish typing in a command, it changes your prompt character from $ to >. This is known as your secondary prompt character and is displayed by the shell whenever it's waiting for you to finish typing a command.

Quotes are also needed when assigning values containing whitespace or special characters to shell variables:

```
$ message='I must say, this sure is fun'

$ echo $message

I must say, this sure is fun

$ text='* means all files in the directory'

$ echo $text

names nu numbers phonebook stat means all files in the directory

$
```

The quotes are needed in the assignments made to the variables message and text because of the embedded spaces. In the preceding example, you are reminded that the shell still does filename substitution after variable name substitution, meaning that the * is replaced by the names of all the files in the current directory before the echo is executed. There is a way to overcome this annoyance, and it's through the use of double quotes.

The Double Quote

Double quotes work similarly to single quotes, except that they're not as restrictive. Whereas the single quotes tell the shell to ignore all enclosed characters, double quotes say to ignore most. In particular, the following three characters are not ignored inside double quotes:

Dollar signs
Back quotes
Backslashes

The fact that dollar signs are not ignored means that variable name substitution is done by the shell inside double quotes.

```
$ x=*

$ echo $x
```

addresses intro lotsaspaces names nu numbers phonebook stat

$ echo '$x'

$x

$ echo "$x"

*

$ Here you see the major differences between no quotes, single quotes, and double quotes. In the first case, the shell sees the asterisk and substitutes all the filenames from the current directory. In the second case, the shell leaves the characters enclosed within the single quotes alone, which results in the display of $x. In the final case, the double quotes indicate to the shell that variable name substitution is still to be performed inside the quotes. So the shell substitutes * for $x. Because filename substitution is not done inside double quotes, * is then passed to echo as the value to be displayed.

So if you want to have the value of a variable substituted, but don't want the shell to treat the substituted characters specially, you must enclose the variable inside double quotes.
Here's another example illustrating the difference between double quotes and no quotes:

```
$ address="39 East 12th Street
> New York, N. Y. 10003"
$ echo $address
39 East 12th Street New York, N. Y. 10003
$ echo "$address"
39 East 12th Street
New York, N. Y. 10003

$
```

It makes no difference whether the value assigned to address is enclosed in single quotes or double quotes. The shell displays the secondary command prompt in either case to tell you it's waiting for the corresponding closed quote.

After assigning the two-line address to address, the value of the variable is displayed by echo. Notice that the address is displayed on a single line. The reason is the same as what caused.
echo one two three four

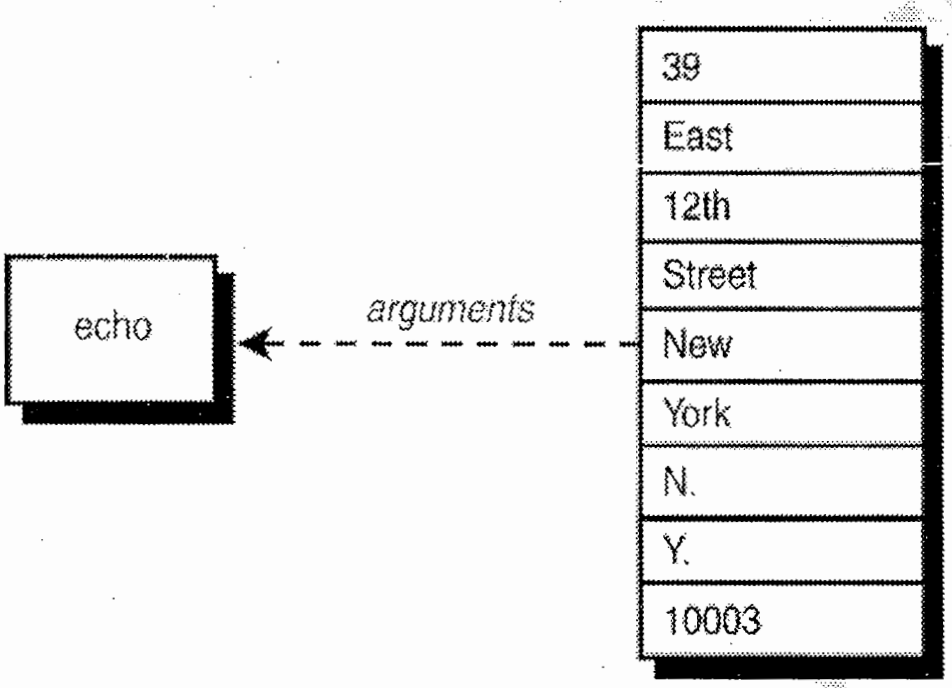
to be displayed as
one two three four

Recalling that the shell removes spaces, tabs, and newlines (that is, whitespace characters) from the command line and then cuts it up into arguments, in the case of

echo $address

the shell simply removes the embedded newline character, treating it as it would a space or tab: as an argument delimiter. Then it passes the nine arguments to echo to be displayed. echo never gets a chance to see that newline; the shell gets to it first.

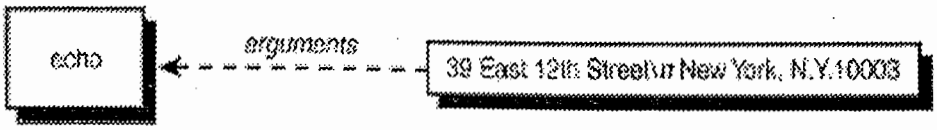
echo $address.



When the command
echo "$address"

is used instead, the shell substitutes the value of address as before, except that the double quotes tell it to leave any embedded whitespace characters alone. So in this case, the shell passes a single argument to echo—an argument that contains an embedded newline. echo simply displays its single argument at the terminal. The newline character is depicted by the characters \n.

echo "$address".



Double quotes can be used to hide single quotes from the shell, and vice versa:

```
$ x="' Hello,' he said"

$ echo $x

'Hello,' he said

$ article=' "Keeping the Logins from Lagging," Bell Labs Record'

$ echo $article

"Keeping the Logins from Lagging," Bell Labs Record
```

The Backslash

Basically, the backslash is equivalent to placing single quotes around a single character, with a few minor exceptions. The backslash quotes the single character that immediately follows it. The general format is

\c

where c is the character you want to quote. Any special meaning normally attached to that character is removed. Here is an example:

```
$ echo >
syntax error: 'newline or ;' unexpected
$ echo \>
>
$
```

In the first case, the shell sees the > and thinks that you want to redirect echo's output to a file. So it expects a filename to follow. Because it doesn't, the shell issues the error message. In the next case, the backslash removes the special meaning of the >, so it is passed along to echo to be displayed.

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

```
$x
$
```

In this case, the shell ignores the $ that follows the backslash, and as a result, variable substitution is not performed.

Because a backslash removes the special meaning of the character that follows, can you guess what happens if that character is another backslash? Right, it removes the special meaning of the backslash:

```
$ echo \\
\
$
```

Naturally, you could have also written

```
$ echo '\'
\
$
```

Using the Backslash for Continuing Lines

As mentioned at the start of this section, \c is basically equivalent to 'c'. One exception to this rule is when the backslash is used as the very last character on the line:

```
$ lines=one'
> 'two        Single quotes tell shell to ignore newline
$ echo "$lines"
one
two
$ lines=one\        Try it with a \ instead
> two
$ echo "$lines"
onetwo
$
```

The shell treats a backslash at the end of the line as a line continuation. It removes the newline character that follows and also does not treat the newline as an argument delimiter (it's as if it wasn't even typed). This construct is most often used for typing long commands over multiple lines.

The Backslash Inside Double Quotes

We noted earlier that the backslash is one of the three characters interpreted by the shell inside double quotes. This means that you can use the backslash inside these quotes to remove the meaning of characters that otherwise would be interpreted inside double quotes (that is, other backslashes, dollar signs, back quotes, newlines,

and other double quotes). If the backslash precedes any other character inside double quotes, the backslash is ignored by the shell and passed on to the program:

```
$ echo "\$x"
$x
$ echo "\ is the backslash character"
\ is the backslash character
$ x=5
$ echo "The value of x is \"$x\""
The value of x is "5"
$
```

In the first example, the backslash precedes the dollar sign, interpreted by the shell inside double quotes. So the shell ignores the dollar sign, removes the backslash, and executes echo. In the second example, the backslash precedes a space, not interpreted by the shell inside double quotes. So the shell ignores the backslash and passes it on to the echo command. The last example shows the backslash used to enclose double quotes inside a double-quoted string.

As an exercise in the use of quotes, let's say that you want to display the following line at the terminal:

<<< echo $x >>> displays the value of x, which is $x

The intention here is to substitute the value of x in the second instance of $x, but not in the first. Let's first assign a value to x:

```
$ x=1
$
```

Now try displaying the line without using any quotes:

```
$ echo <<< echo $x >>> displays the value of x, which is $x
syntax error: '<' unexpected
$
```

The < signals input redirection to the shell; this is the reason for the error message.
If you put the entire message inside single quotes, the value of x won't be substituted at the end. If you enclose the entire string in double quotes, both occurrences of $x will be substituted. Here are two different ways to do the quoting properly (realize that there are usually several different ways to quote a string of characters to get the results you want):

```
$ echo "<<< echo \$x >>> displays the value of x, which is $x"
<<< echo $x >>> displays the value of x, which is 1
$ echo '<<< echo $x >>> displays the value of x, which is' $x
<<< echo $x >>> displays the value of x, which is 1
$
```

In the first case, everything is enclosed in double quotes, and the backslash is used to prevent the shell from performing variable substitution in the first instance of $x. In the second case, everything up to the last $x is enclosed in single quotes. If the variable x might have contained some filename substitution or whitespace characters, a safer way of writing the echo would have been
echo '<<< echo $x >>> displays the value of x, which is' "$x"

Command Substitution:

Command substitution refers to the shell's capability to insert the standard output of a command at any point in a command line. There are two ways in the shell to perform command substitution: by enclosing a shell command with back quotes and with the $(...) construct.

The Back Quote

The back quote is unlike any of the previously encountered types of quotes. Its purpose is not to protect characters from the shell but to tell the shell to execute the enclosed command and to insert the standard output from the command at that point on the command line. The general format for using back quotes is

`command`

where command is the name of the command to be executed and whose output is to be inserted at that point.[1]

[1] Note that using the back quote for command substitution is no longer the preferred method; however, we cover it here because of the large number of older, canned shell programs that still use this construct. Also, you should know about back quotes in case you ever need to write shell programs that are portable to older Unix systems with shells that don't support the newer $(...) construct.

Here is an example:

```
$ echo The date and time is: `date`

The date and time is: Wed Aug 28 14:28:43 EDT 2002
$
```

When the shell does its initial scan of the command line, it notices the back quote and expects the name of a command to follow. In this case, the shell finds that the date command is to be executed. So it executes date and replaces the `date` on the command line with the output from the date. After that, it divides the command line into arguments in the normal manner and then initiates execution of the echo command.

```
$ echo Your current working directory is `pwd`

Your current working directory is /users/steve/shell/ch6

$
```

Here the shell executes pwd, inserts its output on the command line, and then executes the echo. Note that in the following section, back quotes can be used in all the places where the $(...) construct is used.

The $(...) Construct

The POSIX standard shell supports the newer $(...) construct for command substitution. The general format is

```
$(command)
```

where, as in the back quoting method, command is the name of the command whose standard output is to be substituted on the command line. For example:

```
$ echo The date and time is: $(date)

The date and time is: Wed Aug 28 14:28:43 EDT 2002

$
```

This construct is better than back quotes for a couple of reasons. First, complex commands that use combinations of forward and back quotes can be difficult to read, particularly if the typeface you're using doesn't have visually different single quotes and back quotes; second, $(...) constructs can be easily nested, allowing command substitution within command substitution. Although nesting can also be performed with back quotes, it's a little trickier. You'll see an example of nested command substitution later in this section.

You are not restricted to executing a single command between the parentheses: Several commands can be executed if separated by semicolons. Also, pipelines can be used. Here's a modified version of the nu program that displays the number of logged-in users:

```
$ cat nu
```

```
echo There are $(who | wc -l) users logged in
```

$ nu Execute it

```
There are 13 users logged in
$
```

Because single quotes protect everything, the following output should be clear:

```
$ echo '$(who | wc -l) tells how many users are logged in'
$(who | wc -l) tells how many users are logged in
$
```

But command substitution is interpreted inside double quotes:

```
$ echo "You have $(ls | wc -l) files in your directory"
You have       7 files in your directory
$
```

(What causes those leading spaces before the 7?) Remember that the shell is responsible for executing the command enclosed between the parentheses. The only thing the echo command sees is the output that has been inserted by the shell.
Suppose that you're writing a shell program and want to assign the current date and time to a variable called now, perhaps to display it later at the top of a report, or log it into a file. The problem here is that you somehow want to take the output from date and assign it to the variable. Command substitution can be used for this:

$ now=$(date) Execute date and store the output in now

$ echo $now See what got assigned

```
Wed Aug 28 14:47:26 EDT 2002
$
```

When you write

now=$(date)

the shell realizes that the entire output from date is to be assigned to now. Therefore, you don't need to enclose $(date) inside double quotes.

Even commands that produce more than a single line of output can be stored inside a variable:

$ filelist=$(ls)

$ echo $filelist

addresses intro lotsaspaces names nu numbers phonebook stat

$

What happened here? You end up with a horizontal listing of the files even though the newlines from ls were stored inside the filelist variable (take our word for it). The newlines got eaten up when the value of filelist was substituted by the shell in processing the echo command line. Double quotes around the variable will preserve the newlines:

$ echo "$filelist"

addresses

intro

lotsaspaces

names

nu

numbers

phonebook

stat

$

To store the contents of a file into a variable, you can use cat:
$ namelist=$(cat names)

$ echo "$names"

Charlie

Emanuel

Fred

Lucy

Ralph

Tony

Tony

$

If you want to mail the contents of the file memo to all the people listed in the names file (who we'll assume here are users on your system), you can do the following:

$ mail $(cat names) < memo

$

Here the shell executes the cat and inserts the output on the command line so it looks like this:

mail Charlie Emanuel Fred Lucy Ralph Tony Tony < memo

Then it executes mail, redirecting its standard input from the file memo and passing it the names of seven users who are to receive the mail.

Notice that Tony receives the same mail twice because he's listed twice in the names file. You can remove any duplicate entries from the file by using sort with the -u

option (remove duplicate lines) rather than cat to ensure that each person only receives mail once:

```
$ mail $(sort -u names) < memo

$
```

It's worth noting that the shell does filename substitution after it substitutes the output from commands. Enclosing the commands inside double quotes prevents the shell from doing the filename substitution on this output if desired.

Command substitution is often used to change the value stored in a shell variable. For example, if the shell variable name contains someone's name, and you want to convert every character in that variable to uppercase, you could use echo to get the variable to tr's input, perform the translation, and then assign the result back to the variable:

```
$ name="Ralph Kramden"

$ name=$(echo $name | tr '[a-z]' '[A-Z]')  Translate to uppercase

$ echo $name

RALPH KRAMDEN

$
```

The technique of using echo in a pipeline to write data to the standard input of the following command is a simple yet powerful technique; it's used often in shell programs.

The next example shows how cut is used to extract the first character from the value stored in a variable called filename:

```
$ filename=/users/steve/memos

$ firstchar=$(echo $filename | cut -c1)

$ echo $firstchar

/
```

```
$
```

sed is also often used to "edit" the value stored in a variable. Here it is used to extract the last character from the variable file:

```
$ file=exec.o

$ lastchar=$(echo $file | sed 's/.*\(.\)$/\1/')

$ echo $lastchar

o

$
```

The sed command says to replace all the characters on the line with the last one. The result of the sed is stored in the variable lastchar. The single quotes around the sed command are important because they prevent the shell from messing around with the backslashes (would double quotes also have worked?)

Finally, command substitutions can be nested. Suppose that you want to change every occurrence of the first character in a variable to something else. In a previous example, firstchar=$(echo $filename | cut -c1) gets the first character from filename, but how do we use this character to change every occurrence in filename? A two-step process is one way:

```
$ filename=/users/steve/memos

$ firstchar=$(echo $filename | cut -c1)

$ filename=$(echo $filename | tr "$firstchar" "^")   translate / to ^

$ echo $filename

^users^steve^memos

$
```

Or a single, nested command substitution can perform the same operation:

```
$ filename=/users/steve/memos
```

```
$ filename=$(echo $filename | tr "$(echo $filename | cut -c1)" "^")

$ echo $filename

^users^steve^memos

$
```

If you have trouble understanding this example, compare it to the previous one: Note how the firstchar variable in the earlier example is replaced by the nested command substitution.

Syntax: - $ echo "Any normal Text or $ variable name"

Ex: $echo "Naresh i Technologies"

Ex: $ echo "Welcomes you"

Common Options

-n don't print <new-line> (BSD, shell built-in)

\c don't print <new-line> (NIT)

\0n where n is the 8-bit ASCII character code (NIT)

\t tab (NIT)

\f form-feed (NIT)

\n new-line (NIT)

\v vertical tab (NIT)

The $# Variable

Whenever you execute a shell program, the special shell variable $# gets set to the number of arguments that were typed on the command line. As you'll see in the next chapter, this variable can be tested by the program to determine whether the correct number of arguments was typed by the user.

The next program called args was written just to get you more familiar with the way arguments are passed to shell programs. Study the output from each example and make sure that you understand it:

$ cat args Look at the program

```
echo $# arguments passed
echo arg 1 = :$1: arg 2 = :$2: arg 3 = :$3:
$ args a b c                    Execute it
3 arguments passed
arg 1 = :a: arg 2 = :b: arg 3 = :c:

$ args a b                      Try it with two arguments

2 arguments passed

arg 1 = :a: arg 2 = :b: arg 3 = ::      Unassigned args are null

$ args                          Try it with no arguments

0 arguments passed

arg 1 =:: arg 2 =:: arg 3 = ::

$ args "a b c"                  Try quotes

1 arguments passed

arg 1 = :a b c: arg 2 = :: arg 3 = ::

$ ls x*                         See what files start with x

xact

xtra

$ args x*                       Try file name substitution

2 arguments passed

arg 1 = :xact: arg 2 = :xtra: arg 3 = ::

$ my_bin=/users/steve/bin

$ args $my_bin                  And variable substitution

1 arguments passed
```

```
arg 1 = :/users/steve/bin: arg 2 = :: arg 3 = ::

$ args $(cat names)              Pass the contents of names

7 arguments passed

arg 1 = :Charlie: arg 2 = :Emanuel: arg3 = :Fred:

$
```

As you can see, the shell does its normal command-line processing even when it's executing your shell programs. This means that you can take advantage of the normal niceties such as filename substitution and variable substitution when specifying arguments to your programs.

The $* Variable

The special variable $* references all the arguments passed to the program. This is often useful in programs that take an indeterminate or variable number of arguments. You'll see some more practical examples later. Here's a program that illustrates its use:

```
$ cat args2

echo $# arguments passed

echo they are :$*:

$ args2 a b c

3 arguments passed

they are :a b c:

$ args2 one              two

2 arguments passed

they are :one two:
```

$ args2

0 arguments passed

they are ::

$ args2 *

8 arguments passed

they are :args args2 names nu phonebook stat xact xtra:
$

Exit Status

Whenever any program completes execution under the Unix system, it returns an exit status back to the system. This status is a number that usually indicates whether the program successfully ran. By convention, an exit status of zero indicates that a program succeeded, and nonzero indicates that it failed. Failures can be caused by invalid arguments passed to the program, or by an error condition detected by the program. For example, the cp command returns a nonzero exit status if the copy fails for some reason (for example, if it can't create the destination file), or if the arguments aren't correctly specified (for example, wrong number of arguments, or more than two arguments and the last one isn't a directory). In the case of grep, an exit status of zero (success) is returned if it finds the specified pattern in at least one of the files; a nonzero value is returned if it can't find the pattern or if an error occurs (the arguments aren't correctly specified, or it can't open one of the files).

In a pipeline, the exit status is that of the last command in the pipe. So in

who | grep fred

the exit status of the grep is used by the shell as the exit status for the pipeline. In this case, an exit status of zero means that fred was found in who's output (that is, fred was logged on at the time that this command was executed).

The $? Variable

The shell variable $? is automatically set by the shell to the exit status of the last command executed. Naturally, you can use echo to display its value at the terminal.

```
$ cp phonebook phone2
$ echo $?
0                Copy "succeeded"
$ cp nosuch backup
cp: cannot access nosuch
$ echo $?
2                Copy "failed"
$ who            See who's logged on
root   console Jul 8 10:06
wilma  tty03   Jul 8 12:36
barney tty04   Jul 8 14:57
betty  tty15   Jul 8 15:03
$ who | grep barney
barney tty04   Jul 8 14:57
$ echo $?        Print exit status of last command (grep)
```

```
0               grep "succeeded"
$ who | grep fred
$ echo $?
1               grep "failed"
$ echo $?
0               Exit status of last echo
$
```

Note that the numeric result of a "failure" for some commands can vary from one Unix version to the next, but success is always signified by a zero exit status.

Let's now write a shell program called on that tells us whether a specified user is logged on to the system. The name of the user to check will be passed to the program on the command line. If the user is logged on, we'll print a message to that effect; otherwise we'll say nothing. Here is the program:

```
$ cat on
#
# determine if someone is logged on
#
user="$1"
if who | grep "$user"
then
  echo "$user is logged on"
```

```
fi

$
```

This first argument typed on the command line is stored in the shell variable user. Then the if command executes the pipeline

```
who | grep "$user"
```

and tests the exit status returned by grep. If the exit status is zero, grep found user in who's output. In that case, the echo command that follows is executed. If the exit status is nonzero, the specified user is not logged on, and the echo command is skipped. The echo command is indented from the left margin for aesthetic reasons only (tab characters are usually used for such purposes because it's easier to type a tab character than an equivalent number of spaces). In this case, just a single command is enclosed between the then and fi. When more commands are included, and when the nesting gets deeper, indentation can have a dramatic effect on the program's readability. Later examples will help illustrate this point.

Here are some sample uses of on:

```
$ who
root     console Jul 8 10:37
barney   tty03   Jul 8 12:38
fred     tty04   Jul 8 13:40
joanne   tty07   Jul 8 09:35
tony     tty19   Jul 8 08:30
lulu     tty23   Jul 8 09:55
$ on tony                    We know he's on
```

```
tony    tty19   Jul 8 08:30   Where did this come from?

tom     is logged on

$ on steve              We know he's not on

$ on ann                Try this one

joanne  tty07   Jul 8 09:35

ann is logged on

$
```

We seem to have uncovered a couple of problems with the program. When the specified user is logged on, the corresponding line from who's output is also displayed. This may not be such a bad thing, but the program requirements called for only a message to be displayed and nothing else.

This line is displayed because not only does grep return an exit status in the pipeline

```
who | grep "$user"
```

but it also goes about its normal function of writing any matching lines to standard output, even though we're really not interested in that. We can dispose of grep's output by redirecting it to the system's "garbage can," /dev/null. This is a special file on the system that anyone can read from (and get an immediate end of file) or write to. When you write to it, the bits go to that great bit bucket in the sky!

```
who | grep "$user" > /dev/null
```

The second problem with on appears when the program is executed with the argument ann. Even though ann is not logged on, grep matches the characters ann for the user joanne. What you need here is a more restrictive pattern specification, which you learned how to do in Chapter 4, "Tools of the Trade," where we talked about regular expressions. Because who lists each username in column one of each output

line, we can anchor the pattern to match the beginning of the line by preceding the pattern with the character ^:

who | grep "^$user" > /dev/null
But that's not enough. grep still matches a line like

bobby tty07 Jul 8 09:35

if you ask it to search for the pattern bob. What you need to do is also anchor the pattern on the right. Realizing that who ends each username with one or more spaces, the pattern

"^$user "

now only matches lines for the specified user.

Let's try the new and improved version of on:

```
$ cat on
#
# determine if someone is logged on -- version 2
#
user="$1"
if who | grep "^$user " > /dev/null
then
    echo "$user is logged on"
fi
$ who                Who's on now?
root    console Jul 8 10:37
```

```
barney   tty03   Jul 8 12:38

fred     tty04   Jul 8 13:40

joanne   tty07   Jul 8 09:35

tony     tty19   Jul 8 08:30

lulu     tty23   Jul 8 09:55

$ on lulu

lulu is logged on

$ on ann            Try this again

$ on                What happens if we don't give any arguments?

$
```

If no arguments are specified, user will be null. grep will then look through who's output for lines that start with a blank (why?). It won't find any, and so just a command prompt will be returned. In the next section, you'll see how to test whether the correct number of arguments has been supplied to a program and, if not, take some action.

File compression

There are three commands to zip the files

1. gzip
2. compress
3. pack

$gzip

On gzip the original file is replaced by another which has the same name with .gz extension.

Syntax: $gzip [options] [filename (s)]

Example: $gzip NiT

$gunzip:

On gunzip, the zip file is replaced by normal file.

Syntax: $gunzip [options] [filename (s)]

Example: $gunzip NiT.gz

$zcat:

I display the contents of zip formatted file.

Syntax: $zcat [options] [filename(s)]

Example: $zcat NiT.gz

$compress

On gzip the original file is replaced by another which has the same name with .z extension.

Syntax: $compress [options] [filename (s)]

Example: $compress NiT

$uncompress:

On uncompress, the zip file is replaced by normal file.

Syntax: $uncompress [options] [filename (s)]

Example: $uncompressp NiT.z

$zcat:

I display the contents of zip formatted file.

Syntax: $zcat [options] [filename(s)]

Example: $zcat NiT.z

$pack

On gzip the original file is replaced by another which has the same name with .gz extension.

Syntax: $pack [options] [filename (s)]

Example: $pack NiT

$unpackp:

On gunzip, the zip file is replaced by normal file.

Syntax: $unpack [options] [filename (s)]

Example: $unpack NiT.z

$pcat:

I display the contents of zip formatted file.

Syntax: $pcat [options] [filename(s)]

Example: $pcat NiT.z

File Permissions

UNIX is unusual in that, under default protection, anyone on the system may read any file (except for mail files), and whether or not it is in their home directory, though only the actual owner of the file may alter it. Directories are also, as a rule, open: others may list the files in or connect to (though not alter) most directories.

(Don't forget that the best way to ensure the security of your files is to change your password regularly, using the 'passwd' command.)

Access privileges in Unix are divided into three kinds: "read," "write," and "execute." There's also three kinds of people: the owner (almost always yourself), people in one of your groups (groups are rarely used on the NSIT Server Cluster), and everyone else on the machine. So each file has nine areas to consider: the read, write, and execute privileges for you, the group, and everyone else.

If you use the "ls -l" command, these file and directory privileges are listed, in somewhat cryptic fashion. ("ls -la" also lists this for files whose names begin with "." -- usually preferences files.)

In UNIX each and every file may have 3 permissions. They are **read [r], write [w],** and **execute [x]**. We can modify or change these permissions for a directory or file. Here changing means adding or removing a particular or a set of permissions.

The column of ten letters or -'s on the far left are the privileges information. A "d" in the far left indicates a directory. Files would have a "-" there instead. The next nine characters are the read (r), write (w), and execute (x) privileges for owner or "user" (first three), group (next three), and everyone else or "others" (last three):

Read

means that a file can be looked at and copied or, for a directory, the contents can be listed. (Usually, you use 'more filename' to view a text file one screen at a time; this does not work for compiled files.)

Write

Means that a file can be edited, overwritten, deleted, moved, etc. For a directory, this means that files can be placed there, the directory can be moved, and so on.

Execute

Means that the file can be run (like a program). For directories, "execute" permission means that the directory can be opened. (Standard UNIX commands, like 'ls', are just executable files somewhere, as are more complicated programs, like 'ftp'.)

So, for example, the file "filename" has -rwx--x--x privileges, which means the owner can read, write, and execute the file. Anyone else on the system can only execute (run) the file.

You own your own home directory, and, by default, this is set so that you can read, write, and execute it and everyone else can read and execute it. The default for files inside your home directory is for you to read and write them and everyone else to read them. If you want, you can use **chmod** to change the protection on your home directory so that "others" cannot read or execute your home directory, but this is not encouraged. It is better to change the protection on individual files. If you do change the protection on your home directory, you should maintain "execute" privilege for "others". This will not allow people to see the contents of your directory, but it will allow some utilities like finger to work properly.

So these permissions can be modified using two methods. They are

a) Symbolic mode

b) Absolute mode

a) In **Symbolic mode** we can represent file permissions and modifying symbols and types of users (users will be discusses later) are as follows.

Permissions		users		symbols	
read	r	owner	u	adding	+
write	w	group	g	removing	-
execute	x	others	o		
		all	a		

Types of users: In the above, **owner/user [u], group [g] and others [o]** are known as different types of users.

$chmod (change mode): It is used to change the file permissions either in symbolic or absolute mode.

Example: $chmod [option] [arg].. [filename]

In Symbolic mode

$chmod g+w

This command will provide write permission to the group

$ chmod u+x, g-w, o+rwx NiT

This command will provide execute permission to owner, remove write from group and provide read, write and execute permissions on the file NiT.

In Absolute mode

In **Absolute mode** the representation is a combination of 3 digit code. In this code we should use the digits in the range 0---7, and 1 represents the presence and 0 represents the absence of a particular permission. The permissions can be changed by using one command.

$ chmod 427 NiT

This command provides read for owner, write for group, all permissions to the file NiT. That is how, see below.

4 is for owner	2 is for group	7 is for others
100	010	111 → (binary form of 427)
r w x	r w x	r w x
r - -	- w -	r w x (result)

Permission	Weight
Read	4
Write	2
Execute	1
Read and Write	6
Write and Execute	3
Read and Execute	5
Read write and Execute	7

In the above form, **owner** is having only **read** permission, **group** is having only **write** permission and **others** are having **all** permissions.

Using umask:

Changing privileges

The way to change the privileges setting on a file or directory is the "chmod" (change mode) command. There's two ways to do this, and the online manual pages talk about both of them in great detail.

In brief, the first way is numeric and the second is symbolic. In the numeric system, read privileges are given the value 4, write privileges are given 2, and execute privileges are given 1. These numbers are added together for the owner (user), your group, and others. So that, for example, "chmod 644 myfile" gives you read and write privileges for "myfile" (4+2) and your group and others only read privileges (4). To make the file executable, you can type "chmod 755 myfile"; so that now you can read, write and execute it (4+2+1) and everyone else can read and execute it (4+1).

The other way to use chmod is with the abbreviations "u" for user (you), "g" for group, and "o" for others; read, write, and execute are r, w, and x, just like in the "ls -l" listing. Thus, to add write privileges for members of your group to myfile, type "chmod g+w myfile"; to remove read privileges for others, type "chmod o-r myfile".

So if you want others to be able to read and copy or use a file but not alter it, give them read privileges (and execute, if it's an executable file). To allow others to list a directory, give them read privileges (and to connect to the directory, execute privileges).

You own your own home directory, and, by default, this is set so that you can read, write, and execute it and everyone else can read and execute it. The default for files inside your home directory is for you to read and write them and everyone else to read them. You might want to change the privileges of your home directory for others, or you might want to just protect certain files in your home directory.

To reset the level of privacy by changing the privileges on a file or directory, you'll need the 'chmod' (change mode) command, which you can use in two completely different ways. The easier way is using the abbreviations "u" for user (you), "g" for group, and "o" for others; read, write, and execute are "r", "w", and "x". Thus, to add write privileges for members of your group to myfile, type "chmod g+w myfile"; to remove read privileges for others, type "chmod o-r myfile".

The chmod command allows you to alter permissions on a file –by-file basis. The **umask** command allows you to do this automatically when you create any file or directory.
The **umask** allows you to specify the permissions of all files created after issue the **umask** command. Instead of dealing with individual file permissions, you can determine permissions for all future files with a single command.
1. Whenever file is created UNIX assumes that the permissions for this file should be 666.
2. Whenever directory is created UNIX assumes that the permissions for this file should be 777.
Example:
$umask 242

This would see to it here onwards any new file that you create would have the

permissions **424 (666-242)**

That means: for a **file**

The **user** has **Read**

The **group** has **write**

The **others** has **Read** permissions

For **Directory**

Any directory that you create would have the permissions **535 (777-242)**

The **user** has **Read** and **Execute**

The **group** has **Write** and **Execute**

The **others** has **Read** and **Execute** permissions

#chown - change ownership

Ownership of a file can be changed with the **chown** command. On most versions of UNIX this can only be done by the super-user, i.e. a normal user can't give away ownership of their files. **chown** is used as below, where # represents the shell prompt for the super-user:

Syntax

chown [options] user[:group] file (KSR4)

chown [options] user[.group] file (NiT)

Common Options

-R recursively descend through the directory structure

-f force, and don't report any errors

Examples

chown [new_owner] [file name]

Creating New Group

#groupadd: It creates a new group under existing user.

Syntax:

#groupadd [options] [new group] [filename]

Example

#groupadd NiT NeO

#chgrp - change group

Anyone can change the group of files they own, to another group they belong to, with the **chgrp** command.

Syntax

chgrp [options] group file

Common Options

-R recursively descend through the directory structure

-f force, and don't report any errors

Examples

#chgrp [new_group] {filename]

The test Command

A built-in shell command called test is most often used for testing one or more conditions in an if command. Its general format is

test expression

where expression represents the condition you're testing. test evaluates expression, and if the result is true, it returns an exit status of zero; otherwise, the result is false, and it returns a nonzero exit status.

String Operators

As an example of the use of test, the following command returns a zero exit status if the shell variable name contains the characters julio:

test "$name" = julio

The = operator is used to test whether two values are identical. In this case, we're testing to see whether the contents of the shell variable name are identical to the characters julio. If it is, test returns an exit status of zero; nonzero otherwise.

Note that test must see all operands ($name and julio) and operators (=) as separate arguments, meaning that they must be delimited by one or more whitespace characters.

Getting back to the if command, to echo the message "Would you like to play a game?" if name contains the characters julio, you would write your if command like this:

```
if test "$name" = julio
then
    echo "Would you like to play a game?"
fi
```

(Why is it better to play it safe and enclose the message that is displayed by echo inside quotes?) When the if command gets executed, the command that follows the if is executed, and its exit status is tested. The test command is passed the three arguments $name (with its value substituted, of course), =, and julio. test then tests to see whether the first argument is identical to the third argument and returns a zero exit status if it is and a nonzero exit status if it is not.

The exit status returned by test is then tested. If it's zero, the commands between then and fi are executed; in this case, the single echo command is executed. If the exit status is nonzero, the echo command is skipped.

It's good programming practice to enclose shell variables that are arguments to test inside a pair of double quotes (to allow variable substitution). This ensures that test sees the argument in the case where its value is null. For example, consider the following example:

```
$ name=                 Set name null
$ test $name = julio
sh: test: argument expected
$
```

Because name was null, only two arguments were passed to test: = and julio because the shell substituted the value of name before parsing the command line into arguments. In fact, after $name was substituted by the shell, it was as if you typed the following:
test = julio
When test executed, it saw only two arguments.
By placing double quotes around the variable, you ensure that test sees the argument because quotes act as a "placeholder" when the argument is null.

```
$ test "$name" = julio

$ echo $?               Print the exit status
1
$
```

String Operators

Operator	Returns TRUE (exit status of 0) if
$string_1$ = $string_2$	$string_1$ is identical to $string_2$.
$string_1$!=	$string_1$ is not identical to $string_2$.

String Operators

Operator	Returns TRUE (exit status of 0) if
string$_2$	
string	string is not null.
-n string	string is not null (and string must be seen by test).
-z string	string is null (and string must be seen by test).

You've seen how the = operator is used. The != operator is similar, only it tests two strings for inequality. That is, the exit status from test is zero if the two strings are not equal, and nonzero if they are.

Let's look at three similar examples.

```
$ day="monday"
$ test "$day" = monday
$ echo $?
0                    True
$
```

The test command returns an exit status of 0 because the value of day is equal to the characters monday. Now look at the following:

```
$ day="monday "
$ test "$day" = monday
$ echo $?
1                    False
$
```

Here we assigned the characters monday—including the space character that immediately followed—to day. Therefore, when the previous test was made, test returned false because the characters "monday " were not identical to the characters "monday".

If you wanted these two values to be considered equal, omitting the double quotes would have caused the shell to "eat up" the trailing space character, and test would have never seen it:

```
$ day="monday "
$ test $day = monday
$ echo $?
0
$                    True
```

Although this seems to violate our rule about always quoting shell variables that are arguments to test, it's okay to omit the quotes if you're sure that the variable is not null (and not composed entirely of whitespace characters).
You can test to see whether a shell variable has a null value with the third operator listed in Table 8.1:

```
test "$day"
```

This returns true if day is not null and false if it is. Quotes are not necessary here because test doesn't care whether it sees an argument in this case. Nevertheless, you are better off using them here as well because if the variable consists entirely of whitespace characters, the shell will get rid of the argument if not enclosed in quotes.

```
$ blanks="    "
$ test $blanks              Is it not null?
$ echo $?
1                     False—it's null
$ test "$blanks"            And now?
$ echo $?
0                      True—it's not null
$
```

In the first case, test was not passed any arguments because the shell ate up the four spaces in blanks. In the second case, test got one argument consisting of four space characters; obviously not null.

In case we seem to be belaboring the point about blanks and quotes, realize that this is a sticky area that is a frequent source of shell programming errors. It's good to really understand the principles here to save yourself a lot of programming headaches in the future.

There is another way to test whether a string is null, and that's with either of the last two operators listed previously in Table 8.1. The -n operator returns an exit status of zero if the argument that follows is not null. Think of this operator as testing for nonzero length.

The -z operator tests the argument that follows to see whether it is null and returns an exit status of zero if it is. Think of this operator as testing to see whether the following argument has zero length.

So the command

```
test -n "$day"
```

returns an exit status of 0 if day contains at least one character. The command

test -z "$dataflag"
returns an exit status of 0 if dataflag doesn't contain any characters.
Be forewarned that both of the preceding operators expect an argument to follow; therefore, get into the habit of enclosing that argument inside double quotes.

```
$ nullvar=

$ nonnullvar=abc

$ test -n "$nullvar"       Does nullvar have nonzero length?

$ echo $?
1                  No
$ test -n "$nonnullvar"    And what about nonnullvar?
$ echo $?
0                  Yes
$ test -z "$nullvar"       Does nullvar have zero length?
$ echo $?
0                  Yes
$ test -z "$nonnullvar"    And nonnullvar?
$ echo $?
1                  No
$
```

Note that test can be picky about its arguments. For example, if the shell variable symbol contains an equals sign, look at what happens if you try to test it for zero length:

```
$ echo $symbol
=
$ test -z "$symbol"
sh: test: argument expected
$
```

The = operator has higher precedence than the -z operator, so test expects an argument to follow. To avoid this sort of problem, you can write your command as
test X"$symbol" = X
which will be true if symbol is null, and false if it's not. The X in front of symbol prevents test from interpreting the characters stored in symbol as an operator.

An Alternative Format for test

The test command is used so often by shell programmers that an alternative format of the command is recognized. This format improves the readability of the command, especially when used in if commands.

You'll recall that the general format of the test command is
test expression
This can also be expressed in the alternative format as
[expression]
The [is actually the name of the command (who said anything about command names having to be alphanumeric characters?). It still initiates execution of the same test command, only in this format, test expects to see a closing] at the end of the expression. Naturally, spaces must appear after the [and before the].
You can rewrite the test command shown in a previous example with this alternative format as shown:

```
$ [ -z "$nonnullvar" ]
$ echo $?
1
$
```

When used in an if command, this alternative format looks like this:

```
if [ "$name" = julio ]
then
        echo "Would you like to play a game?"
fi
```

Which format of the if command you use is up to you; we prefer the [...] format, so that's what we'll use throughout the remainder of the book.

Communication Commands

The following commands are known as "communication commands". They are write, mesg, mail and etc. Here we are having two types of communications.

They are:

1. On-line communication or two-way communication
2. Off-line communication or single-way communication

1. On-line communication:

On-line communication can be performed by using write and commands.

write: The write command lets you have a two-way communication with any person who is currently logged in. write uses login name of the recipient as argument, and text of the message from the standard input.

This is how Raju wants to send a message to Stanley:

$write Stanley

have u completed your work?

I have completed mine – Raju

Then the typed message will send to Stanley if he is logged in. If he is not, the system responds with an error message: Stanley is not logged in.

If Stanley is logged in, then he will see the message, and he has the option of replying and if he wants to reply then he has to invoke the write command.

$write Raju

another half an hour After the above if any person wants to close his communication he has to press <ctrl+d>.

Note: when a user is using the above command, he has to remember one thing. If there exists more than one person with same user name then the user must includes the terminal no. also, that is as follows.

$write Stanley tty03

$mesg: This command decides the willingness of a person to receive or not receive the messages send by other persons. This can be applied in two-way and single-way communication.

a) $mesg n: By executing the above command means the person is not ready to receive messages.

b) $mesg y: This means, the person is ready to receive the messages.

2) Off-line communication:

mail: Command is used for this type of communication. At this it is not necessary that the receiver must be logged in once you received the mail you can do the following. View it on your terminal save it in a mail box save it in a file Delete it reply to it forward it to others.

$mail Stanley

Hai we are going to Thailand to celebrate the launching of our company's new branch, are you ready to come .or press <ctrl+d>to terminate mail.

The sent message doesn't directly appear on Stanley's terminal, if he is running another program at this moment. When he is free then system display the message like

you have a new mail

To view the mail messages you have to execute the mail command without any options.

$mail: Then mail list the all messages with the names of respective sender names and prompts you with the symbol & Here Stanley has to enter his choice, whether he may press enter key or a number to view that particular message and so on.

mail command options:

&+	to view next message.
&-	to view previous message.
&N	to the Nth numbered message.
&d N	deletes Nth message.

&u N	undeletes the Nth message.
&s filename	saves the current message in the specified file.
&m user-name	forwards the message to the specified user.
&r N	replies to the sender of message N.
&q	quits from mail.
& $	Displays the last message
& =	It prints current message number
& .	Displays the current message
& u	Undelete last deleted message
& u	1 2 3 Undelete specified messages
& v	permits editing of current message with vi
& f	log name Forwards the current mail message to the specified user
& l	1 2 3 Prints specified mail on line printer

wall: It is used for the send message to all user's who are logged in.

$wall

Good Morning to u All

ctrl+d

$mail: It is used to send mails

$mail <User Name>

Subject:

Type the mail

- - - - - - - -

- - - - - - - -

- - - - - - - -

- - - - - - - -

ctrl+d

$mail <User1> <User2> <User3> <User4>

Subject:

Type the mail

- - - - - - - -

- - - - - - - -

- - - - - - - -

- - - - - - - -

ctrl+d

$mail Rama < Dept

Sending the department file to user Rama

mail <Enter>

mail Prompt: &

& 2 → to see second mail.

& 3 → to see third mail

& q → Quit

& **mail -f** : It is used to open secondary mail box

& **S file name :** It is used to save the contents of a file.

& p → It is used to print out

& d → It is used to delete the current opened mail

& d 5 → It is used to delete the 5th mail

& d 1-5 → It is used to delete from mail 1-5 mails

Networking Commands:

TELNET and FTP are Application Level Internet protocols. The TELNET and FTP protocol specifications have been implemented by many different sources, including The National Center for Supercomputer Applications (NCSA), and many other public domain and shareware sources.

The programs implementing the TELNET protocol are usually called telnet, but not always. Some notable exceptions are tn3270, WinQVT, and QWS3270, which are also TELNET protocol implementations. TELNET is used for remote login to other computers on the Internet.

The programs implementing the FTP protocol are usually called ftp, but there are exceptions to that too. A program called Fetch, distributed by Dartmouth CollExe, WS_FTP, written and distributed by John Junod, and Ftptool, written by a Mike Sullivan, are FTP protocol implementations with graphic user interfaces. There's an enhanced FTP version, ncftp, that allows additional features, written by Mike Gleason. Also, FTP protocol implementations are often included in TELNET implementation programs, such as the ones distributed by NCSA. FTP is used for transferring files between computers on the Internet.

rlogin is a remote login service that was at one time exclusive to Berkeley 4.3 BSD UNIX. Essentially, it offers the same functionality as telnet, except that it passes to the remote computer information about the user's login environment. Machines can be configured to allow connections from trusted hosts without prompting for the users' passwords. A more secure version of this protocol is the Secure SHell, SSH

From a UNIX prompt, these programs are invoked by typing the command (program name) and the (Internet) name of the remote machine to which to connect. You can also specify various options, as allowed, for these commands.

Syntax

telnet [options] [remote_host [port_number]]

tn3270 [options] [remote_host [port_number]]

ftp [options] [remote_host]

Common Options

ftp telnet Action

-d set debugging mode on

-d same as above (SVR4 only)

-i turn off interactive prompting

FTP → File Transfer Protocol

1). **FTP >** → It is used to copy data from one server to another

2) **FTP** > ls → It displays server side files

3) **FTP > !ls** → It displays client side files

4) **FTP> pwd** → It displays server password

5). **FTP > !pwd** → It displays client password

Downloading/Uploading Files

1). **FTP > get** <filename> → It is used to download a file

2). **FTP> m get** NiT NeO file3 → It is used to download multiple files

3). **FTP > put <File Name>** → To upload file.

4). **FTP > mput** NiT NeO file3 → To upload multiple files

5). **FTP >?** It display all FTP commands

6) **FTP** > bye → It quit from FTP

7). **FTP > get** file * → to download all files

$finger - get information about users

finger displays the .plan file of a specific user, or reports who is logged into a specific machine. The user must allow general read permission on the .plan file.

Syntax: finger [options] [user[@hostname]]

Common Options

-l force long output format

-m match username only, not first or last names

-s force short output format

Example: $finger

Find - find files

The find command will recursively search the indicated directory tree to find files matching a type or pattern you specify. find can then list the files or execute arbitrary commands based on the results.

Syntax: $find directory [search options] [actions]

Common Options

For the time search options the notation in days, n is:

+n more than n days

n exactly n days

-n less than n days

Some file characteristics that find can search for are:

time that the file was last accessed or changed

-atime n access time, true if accessed n days ago

-ctime n change time, true if the files status was changed n days ago

-mtime n modified time, true if the files data was modified n days ago

-newer filename true if newer than filename

-type type of file, where type can be:

b block special file

c character special file

d directory

l symbolic link

p named pipe (FIFO)

f regular file

Example: $find .-NiT NeO_file

The first argument is the name of the directory in which the search starts. In the case of current directory (represented by the.). The second part of the command specifies the file or files to search for.

Running find in the background:

We can run find command to search file in the background also. To run a command in the background, you end it with an ampersand **(&).**

Example: $find / -name NiT _file-print >found_NeO&

The advantage of the running command in the background is that you can go on to run other commands without waiting for the background job to finish.

Terminal Type

Most systems are set up so the user is by default prompted for a terminal type, which should be set to match the terminal in use before proceeding. Most computers work if you choose "vt100". Users connecting using a Sun workstation may want to use "sun"; those using an X-Terminal may want to use "xterms" or "xterm".

The terminal type indicates to the UNIX system how to interact with the session just opened.

Should you need to reset the terminal type, enter the command:

setenv TERM <term type> - if using the C-shell (see Chapter 4.)

(On some systems, e.g. MAGNUS, it's also necessary to type "unsetenv TERMCAP".)

-or-

TERM=<term type>; export TERM - if using the Bourne shell (see Chapter 4.)

where <term type> is the terminal type, such as vt100, that you would like set.

$whereis - report program locations

whereis reports the filenames of source, binary, and manual page files associated with command(s).

Syntax: $whereis [options] command(s)

Common Options

-b report binary files only

-m report manual sections only

-s report source files only

$which - report the command found

$which will report the name of the file that is be executed when the command is invoked. This will be the full path name or the alias that's found first in your path.

Syntax $which command(s)

Example: $ which mail

$script - record your screen I/O

$script creates a script of your session input and output. Using the script command, you can capture all the data transmission from and to your terminal screen until you exit the script program. This can be useful during the programming-and-debugging process, to document the combination of things you have tried, or to get a printed copy of it all for later perusal.

Syntax: script [-a] [file] <. . .> exit

Common Options

-a append the output to file

Typescript is the name of the default file used by script.

You must remember to type exit to end your script session and close your typescript file.

Backup Utilities:

Every file, every database, every byte of information stores on the system. If some hardware failure and any system crash occurred then all the data which is present on the system will be lost. To overcome this by the following backup utilities.

1. $tar: Stands for tape archieve, this command is used to create archieve onto the tapes and the floppies. This command will create an archieve in the media.

The common options are:-

c Copy

-x extract

-t table of contents

Syntax: $tar [options] <device>

Example: $tar -cvf /tape device/author.sql*

Example: $tar -xvf /tape device

Here first example tar command will take the backup of all author.sql files. In the second example it will restore the backup from the tape device.

$cpio: Stands for copy input output. This utility copies files to and from backup device. It uses standard input to take the list of file names. The cpio command can't use without the help of pipes and redirection filters.

Syntax: $command | cpio [options] <device>

Example: $ls | cpio -oc > /dev

-oc --> Run in copy outmode.

History

The C shell, the KORN shell and some other more advanced shells, retain information about the former commands you've executed in the shell. How history is done will depend on the shell used. Here we'll describe the C shell history features. You can use the **history** and **savehist** variables to set the number of previously executed commands to keep track of in this shell and how many to retain between logins,

respectively. You could put a line such as the following in **.cshrc** to save the last 100 commands in this shell and the last 50 through the next login.

set history=100 savehist=50

The shell keeps track of the history list and saves it in **~/.history** between logins.

You can use the built-in **history** command to recall previous commands, e.g. to print the last 10:

$history 10

You can repeat the last command by typing!!:

$!!

You can repeat any numbered command by prefacing the number with a !, e.g.:

$!da

date

Tue Apr 9 09:55:31 EDT 1996

When the shell evaluates the command line it first checks for history substitution before it interprets anything else. Should you want to use one of these special characters in a shell command you will need to escape, or quote it first, with a \ before the character, i.e. \!. The history substitution characters are summarized in the following table.

C Shell History Substitution

Command	Substitution Function
!!	repeat last command
!n	repeat command number **n**
!-n	repeat command n from last
!str	repeat command that started with string **str**
!?str?	repeat command with **str** anywhere on the line
!?str?%	select the first argument that had **str** in it
!:	repeat the last command, generally used with a modifier
!:n	select the **n**th argument from the last command (n=0 is the command name)
!:n-m	select the **n**th through **m**th arguments from the last command
!^	select the first argument from the last command (same as !:1)

!$	select the last argument from the last command
!*	select all arguments to the previous command
!:n*	select the nth through last arguments from the previous command
!:n-	select the nth through next to last arguments from the previous command
^str1^str2^	replace str1 with str2 in its first occurrence in the previous command
!n:s/str1/str2/	substitute str1 with str2 in its first occurrence in the nth command, ending with a g substitute globally

Identity

The system identifies you by the user and group numbers (userid and groupid, respectively) assigned to you by your system administrator. You don't normally need to know your userid or groupid as the system translates username « userid, and group name « groupid automatically. You probably already know your username; it's the name you logon with. The group name is not as obvious, and indeed, you may belong to more than one group. Your primary group is the one associated with your username in the password database file, as set up by your system administrator. Similarly, there is a group database file where the system administrator can assign you rights to additional groups on the system.

In the examples below % is your shell prompt; you don't type this in.

You can determine your userid and the list of groups you belong to with the id and groups commands. On some systems id displays your user and primary group information, e.g.:

```
$id
uid=1101(frank) gid=10(staff)
```

on other systems it also displays information for any additional groups you belong to:

```
$id
uid=1101(frank) gid=10(staff) groups=10(staff),5(operator),14(sysadmin),110(uts)
```

The groups command displays the group information for all the groups you belong to, e.g.:

$groups

staff sysadmin uts operator

The ulimit Command:

It stands for **user limit,** though most files in UNIX occupy tens of blocks, in some odd case a program my go away add create files which occupy huge amount of disk space. Some times things might take such a bad turn that the file might occupy several megabytes of disk space and ultimately harm for the file system. To avoid creation of such files UNIX uses a variable called **ulimit.**

If type **ulimit**

$ulimit

2097152

This implies that the user cannot create a file. Whose size is bigger than 2097152 bytes, or 2048KB. As a user can reduce this value by saying

$ulimit 1

Hence onwards no file can be created whose size is bigger than 512 bytes. Thus this change will be effectively only for the current session and the system will return to its default value when you log out.

An ordinary user can only reduce the **ulimit** value and is never permitted to increase it. A super user is an exception to the rule and can increase or decrease this value.

Redirection operators:

One of the most important contributions UNIX has made to Operating Systems is the provision of many utilities for doing common tasks or obtaining desired information. Another is the standard way in which data is stored and transmitted in UNIX systems. This allows data to be transmitted to a file, the terminal screen, or a program, or from a file, the keyboard, or a program; always in a uniform manner.

The standardized handling of data supports two important features of UNIX utilities: I/O redirection and piping. With output redirection, the output of a command is redirected to a file rather than to the terminal screen. With input redirection, the input to a command is given via a file rather than the keyboard. Other tricks are possible with input and output redirection as well, as you will see. With piping, the output of a command can be used as input (piped) to a subsequent command. In this chapter we discuss many of the features and utilities available to UNIX users.

File Descriptors

File Redirection

Other Special Command Symbols

1. If there exists < in between two set of commands, then that total command execution is from right to left. The output of extreme right command is becomes input to its left side command.

2. If there exists > or | then the command execution is from left to right. Here the output of left command is an input to its right command.

File Descriptors

There are 3 standard file descriptors:

- stdin 0 Standard input to the program
- stdout 1 Standard output from the program
- stderr 2 Standard error output from the program

Stream	Device	Value

Standard Input	Keyboard	0
Standard Output	Terminal Screen	1
Standard Error	Terminal Screen	2

Normally input is from the keyboard or a file. Output, both stdout and stderr, normally go to the terminal, but you can redirect one or both of these to one or more files. You can also specify additional file descriptors, designating them by a number 3 through 9, and redirect I/O through them.

File Redirection:

Output redirection takes the output of a command and places it into a named file. Input redirection reads the file as input to the command. The following table summarizes the redirection options.

File Redirection

Symbol	Redirection
>	output redirect
>!	same as above, but overrides noclobber option of csh
>>	append output
>>!	same as above, but overrides noclobber option on csh and creates the file if it doesn't already exist.
|	pipe output to another command
<	input redirection
<<String	read from standard input until "String" is encountered as the only thing on the line. Also known as a "here document" (see Chapter 8).
<<\String	same as above, but don't allow shell substitutions

An example of output redirection is:

$cat NiT NeO > file3

The above command concatenates NiT then NeO and redirects (sends) the output to file3. If file3 doesn't already exist it is created. If it does exist it will either be truncated to zero length before the new contents are inserted, or the command will be rejected, if the noclobber option of the csh is set. (See the csh in Chapter 4). The original files, NiT and NeO, remain intact as separate entities.

Pipes and Filters

Pipes: If an operator is useful to redirect the output of a command to a file or to any command is known as a **pipe**. There exist 2 **pipe** operators.

1. **> Ex: $ who > userlist**

In this example, who command has to display the list of users, but here the output of who command is stored in the file **userlist**. In this command some disadvantages are there.

2. **| (pipe) Ex: $ who | sort >** sortwho **this** command display the number of logged in users. Here the output of who command (standard output) is given to wc –l command (standard input).

Filters If a command, which accepts some data as input, performs some manipulation on it, and produces the output, then it is known as a Filter. That means the command is performing some filtering action on the input data.

$head: This command is used to display the output from the beginning line of the file. By default it will display 10 lines from the BOF.

Syntax: $ head [options] file

Common Options

-n number of lines to display, counting from the top of the file

-number same as above

Ex: $ head NiT Display first 10 lines from the file NiT.

Ex: $ head -3 NiT Display first 3 lines from NiT.

Ex: $ head -15 NiT Display 1 to 15 lines from the BOF of the file NiT.

$tail: It will display the output from the EOF. By default it will display the last 10 lines.

Syntax: $ tail [options] file

Common Options

-number number of lines to display, counting from the bottom of the file

Ex: $tail NiT Display last 10 lines from the file NiT.

Ex: $tail -5 NiT Display last 5 lines of NiT.

Ex: $tail -25 NiT Display last 25 lines of NiT.

Ex: $tail +20 NiT Displays all line from 21 to all lines

more, less, and pg - page through a file

UNIX provides three commands which offer more flexibility in viewing files. These are the above.

more, less, and pg let you page through the contents of a file one screen at a time. These may not all be available on your UNIX system. They allow you to back up through the previous pages and search for words, etc.

Syntax

more [options] [+/pattern] [filename]

Example: $more NiT

less [options] [+/pattern] [filename]

Example: $less NiT

pg [options] [+/pattern] [filename]

Example: $pg -c NiT

Options

more less pg Action

-c -c -c clear display before displaying

-i ignore case

-w default don't exit at end of input, but prompt and wait

-lines -lines # of lines/screen

+/pattern +/pattern +/pattern search for the pattern

1). It sets the number of lines to be displayed per page.

2). Ability to move either forwards or backwards in a file just at the touch of a keys like f and b.

3). Skip pages while viewing the file page by page.

4). Search the file for a pattern in forward or backward direction.

5). If you hit the key spacebar it displays page wise and enter line by line.

6). To quit pg or more or less, hit the keys **q** or **Q** or **BREAK** or **DELETE**

Taking printouts:

UNIX permits sharing of one printer amongst all users or several printers amongst several users. The **lp** command is used to send the user's print job the print queue.

Syntax: $lp [options] [filename (s)]

Examples:

$lp NiT

$lp NiT NeO

$lp -w NiT

Common Options

1. -w → Sends a message to the user when the file is printed
2. -n num → print number is copies
3. -p list → prints the page numbers specified by list
4. -d printer → Specifies a printer other than default printer
5. -q level → Sets a priority level for the print job

Once the print request has been made using the **lp** command we can watch its progress using the **lpstat** command which shows the status of our print job. There are various options that can be used with the **lpstat** command. Out of these most comprehensive information is obtained using the **-t** option.

$lpstat -t

Scheduler is running

System default destination: lp

Device for lp: dev/lp0, lp is accepting requests since Wednesdav August 6 20:20:20 2009

lp-13 NiT 198 Apr23 12:10

lp-12 root 188 Mar03 12:28

lp-14 NiT 291 Feb23 18:11

lp-15 NiT 98 Apr23 16:13

$cancel: Some times submitted the job for printing we decide to cancel it, we can do so using **cancel** command.

$cancel [options] [jobid]

Example: $cancel lp -13

Where **lp-13** is the request id of our print job. The cancel command responds with, **request "lp-13,190 canceled**

Cut: It is also filter. This command displays the specified positional characters. It must be associated with -c option and we can use, and - operators to specify position and range respectively.

Common Options

-c character_list character positions to select (first character is 1)

-d delimiter field delimiter (defaults to <TAB>)

-f field_list fields to select (first field is 1)

$cat > NiT

StdNo	StdName	Course	Marks	Grade
1001	Ram Kumar	.Net	78	D
1002	Ajit Panday	UNIX	67	D
1003	Kavleen	PHP	89	B
1004	Subin	JAVA	92	A
1005	Raju	Oracle	76	D
1006	Priya	QA	98	A

Ex: $cut -c2, 5 NiT

It will display second and fifth characters from each line of the file NiT.

Ex: $cut -c2-5 NiT

It will display second, third and fifth characters from the file NiT

Ex: $cut -f 1,4 NeO

It displays 1 and 4 fields from NeO file

Ex: $cut -f 1-4 NeO

It displays all fields between 1 to 4 from NeO file

Sort: This command is used to sort the file contents according to alphabetical order or numerical order and display them. The sort command is used to order the lines of a file. Various options can be used to choose the order as well as the field on which a file is sorted. Without any options, the sort compares entire lines in the file and outputs them in ASCII order (numbers first, upper case letters, then lower case letters).

Syntax: $sort [options] [+pos1 [-pos2]] file

Common Options

-b ignore leading blanks (<space> & <tab>) when determining starting and ending characters for the sort key

-d dictionary order, only letters, digits, <space> and <tab> are significant

-f ignores case

-i ignore non-printable characters

-n numeric sort

-m Merge files that have been already sorted

-o outfile output file

-r reverse the sort

-t char use char as the field separator character

-u unique; omit multiple copies of the same line (after the sort)

Ex: $ sort Raju

Ex: $ sort NiT NeO NiT1

Ex: $ sort -r Raju

Ex: $sort -oresult NiT NeO NiT1

Ex: $sort -u -o result NiT NeO NiT1

Ex: $sort -m NiT NeO

Ex: $sort -n KiT

Ex: $sort -nr KiT

This command can reverse sort the contents and display.

Paste: This command display the combined data of the specified two files. The paste command allows two files to be combined side-by-side. The default delimiter between the columns in a paste is a tab, but options allow other delimiters to be used.

Syntax: $paste [options] file1 file2

Common Options

-d list list of delimiting characters

-s concatenate lines

The list of delimiters may include a single character such as a comma; a quoted string, such as a space; or any of the following escape sequences:

\n <newline> character

\t <tab> character

\\ backslash character

\0 empty string (non-null character)

It may be necessary to quote delimiters with special meaning to the shell.

A hyphen (-) in place of a file name is used to indicate that field should come from standard input.

Ex: $ paste NiT NeO

Debugging

Debugging is the process of removing bugs. In Unix Operating System type the following statement:

$set -vx

Here, v ensures that each line in the shell script is displayed before it gets executed, and x ensures that the command along with the argument values that it may have is also displayed before execution.

note that by setting the -v option the line from the script which is about to get executed is displayed, where the lines which are preceded by the '+' sign, have come courtesy the -x option

We can unset the debugging options that may have been set by the following:

$set +vx

Built-in Integer Arithmetic

The POSIX standard shell provides a mechanism for performing integer arithmetic on shell variables called arithmetic expansion. Note that some older shells do not support this feature.

The format for arithmetic expansion is

$((expression))

where expression is an arithmetic expression using shell variables and operators. Valid shell variables are those that contain numeric values (leading and trailing whitespace is allowed). Valid operators are taken from the C programming language and are listed in Appendix A, "Shell Summary."

The result of computing expression is substituted on the command line. For example,
echo $((i+1))

adds one to the value in the shell variable i and prints the result. Notice that the variable i doesn't have to be preceded by a dollar sign. That's because the shell knows that the only valid items that can appear in arithmetic expansions are operators, numbers, and variables. If the variable is not defined or contains a NULL string, its value is assumed to be zero. So if we have not assigned any value yet to the variable a, we can still use it in an integer expression:

$ echo $a Variable a not set

$

$ echo $((a = a + 1)) Equivalent to a = 0 + 1

1

$ echo $a

```
1                    Now a contains 1
$
```

Note that assignment is a valid operator, and the value of the assignment is substituted in the second echo command in the preceding example. Parentheses may be used freely inside expressions to force grouping, as in

```
echo $((i = (i + 10) * j))
```

If you want to perform an assignment without echo or some other command, you can move the assignment before the arithmetic expansion. So to multiply the variable i by 5 and assign the result back to i you can write

```
i=$(( i * 5 ))
```

Note that spaces are optional inside the double parentheses, but are not allowed when the assignment is outside them.
Finally, to test to see whether i is greater than or equal to 0 and less than or equal to 100, you can write

```
result=$(( i >= 0  &&  i <= 100 ))
```

which assigns result 1 if the expression is true and 0 if it's false:

```
$ i=$(( 100 * 200 / 10 ))
$ j=$(( i < 1000 ))      If i is < 1000, set j = 0; otherwise 1
$ echo $i $j
2000 0                   i                    is 2000, so j was set to 0
$
```

That concludes our introduction to writing commands and using variables.

$tr: This **tr** reads input and either deletes target characters or translates each target character into a specified replacement character. The output is a translated version of the input. The tr command translates characters from stdin to stdout.

Syntax: tr [options] string1 [string2]

With no options the characters in string1 are translated into the characters in string2, character by character in the string arrays. The first character in string1 is translated into the first character in string2, etc.

A range of characters in a string is specified with a hyphen between the upper and lower characters of the range, e.g. to specify all lower case alphabetic characters use '[a-z]'.

Repeated characters in string2 can be represented with the '[x*n]' notation, where character x is repeated n times. If n is 0 or absent it is assumed to be as large as needed to match string1.

Characters can include \octal (BSD and SVR4) and \character (SVR4 only) notation. Here "octal" is replaced by the one, two, or three octal integer sequences encoding the ASCII character and "character" can be one of:

b back space

f form feed

n new line

r carriage return

t tab

v vertical tab

The SVR4 version of **tr** allows the operand ":class:" in the string field where class can take on character classification values, including:

alpha alphabetic characters

lower lower case alphabetic characters

upper upper case alphabetic characters

Common Options

-c complement the character set in string1

-d delete the characters in string1

-s squeeze a string of repeated characters in string1 to a single character

a) **Ex: $ tr a T < NiT** This command display the translated version of the file NiT. That is each instance of **a** replaces with **T**.

b) **Ex: $ tr 'abc' 'xyz' < NiT** This command replaces all instances of **a** with **x**, **b** with **y**, and **c** with **z** respectively and then display the resultant data.

c) Ex: $ tr -d 'abc' < NiT This command deletes all instances of **a**, **b** and **c** in the file NiT and display the result.

$nl: If any of the file you want to see with the line numbers then we can use **ln** command to accomplish this task. Some files are very large files if we want to see particular line in that file then it will become very difficult for us to see that line.

Syntax: $nl [options] [filename]

Example: $nl NiT

It displays all lines with line numbers in the file NiT file.

$tee: tee utility sends what it reads from its input in two directions: one to a file named as an argument and to the standard output(monitor). tee sends standard in to specified files and also to standard out. It's often used in command pipelines.

Syntax: tee [options] [file[s]]

Common Options

-a append the output to the files

-i ignore interrupts

Ex: $ cal 9 2009 | tee sept2009

This command will display **September** month of **2009** and save it in the file **sept2009**.

Sed (Stream Editor) & GREP

Stream editor for filtering and transforming text from standard input to standard output. **Sed is** a streamlined, noninteractive editor. It allows you to perform the same kind of editing tasks used in the vi and ex editors. Instead of working interactively with the editor, the **sed** program lets you type your editing commands at the command line, name the file, and then see the output of the editing command on the screen.

Options:

Name	Purpose
q	Quits after the specified line
p	Prints the specified line
-n	Suppress duplicate printing
d	deletes the specified lines.
a	To append at the EOL
i	To insert data into the file, after each line.
w	Writes data on to a new file
-e	Search for more than one pattern
/pattern/	Search for the specified pattern
^	Represents Beginning of the lines
s/str1/str2/	Substitutes str2 at str1

Here are some examples for **sed** command

1. $ sed '5q' NiT Display first 5 lines.

2. $ sed '2p' NiT

Display all lines, but 2nd line will be displayed twice.

3. $ sed -n '2p' NiT

Display only 2nd line.

4. $ sed '3d' NiT

Display all lines except 3rd line.

5. $ sed –n –e '2p' –e '9p' NiT

Display 2nd and 3rd lines.

6. $ sed –n '/Raju/p' NiT

Display all lines that contain the pattern Raju.

7. $ sed –n '/Raju/w NiT1' NiT

This command searches the specified pattern containing lines and save those lines into the file NiT1.

8. $ sed –n '/^Raju/p' NiT

It will display the lines that are beginning with the pattern Raju.

9. $ sed 's/Raju/Gupta/' NiT

Display the lines that are containing Raju, but it will be replaced with gupta.

10. $ sed '1,5s/Raju/gupta/' NiT

This is similar to the above command, but it is applied to the first 5 no. of lines only.

11. $ sed 'i\' NiT This command inserts a blank line after each line.

Note: sed editor can also supports the special characters like *, [and].

GREP Family:

$grep (Globally search for Regular Expression) is one of the most popular and useful UNIX filters. It scans a file for the occurrence of a pattern, and can display the selected pattern, the line numbers in which they were found, or the filenames where the pattern occurs. **grep** can also select lines not containing the pattern.

Syntax: $ grep options pattern filename(s)

1. searching for a single pattern

Ex: $ grep sales NiT: This command displays all lines which contain the pattern **sales** from the file **NiT**.

a) If the pattern was not found, then it will display nothing.

2. If you want to search more than one pattern, then include the patterns in single quotes. **Ex: $ grep 'Net' NiT**

Options: -c counting occurrences

-n displaying line numbers

-v deleting lines

-l displaying file names

-i ignoring case sensitivity

-n displays line numbers of text

$grep command can also allowed the wild card characters, but pattern must be included with in double quotes. Below see the Examples.

1. **$grep hello NiT→** It finds hello string in Nit File.
2. **$grep hello * →** It finds hello string in all file from current directory
3. **$grep Unix NiT NiT1 NiT2 NiT3 →** It finds Unix string at specified all files
4. **$grep –i Unix NiT →** It displays all strings which are starting with capital or small u/U .
5. **$grep –c Unix NiT →** It counts all UNIX strings, how many times repeated
6. **$grep –n Unix NiT →** It prints the line numbers of Unix string where existed
7. **$grep –l Unix * →** It prints all file names where Unix string found.
8. **$grep "c[ae]ll" NiT →** It searches all lines which are matching call cell etc..
9. **$grep "R..u" NiT →** It should match R and u with this two character should be there.
10. **$grep "\<Unix\>" NiT →** It will search only the word Unix

Anchors:

^→ Start of the line

$ → End of the line

11. **$grep "^U" NiT →** It prints all lines which, are starting with 'U'
12. **$grep "^Unix" NiT →** It prints all lines, which are starting with Unix string
13. **$grep "x$" NiT →** It prints all lines which are ending with 'x'
14. **$grep "^\<Unix\>" NiT →** It prints all lines which are starting which UNIX string.

15. $grep "[0-9]$" NiT → It prints all lines which are ending with digits
16. $grep "^[abc]" NiT → It prints all lines which are starting with a b and c
17. $grep "^[^abc]" NiT → It prints all lines which are not starting with a b and c
18. $grep "^$" NiT → It displays all blank lines from NiT file
19. $grep -c "^$" NiT → It displays all blank lines numbers
20. $grep "^...$" NiT → It displays the line only having three characters

$fgrep command:

The **fgrep** command is similar to **grep**, but with three main differences; you can use it to search for several targets at once, it doesn't allow you to use regular expressions to search for patterns, and is faster than **grep**

Example: $fgrep "Shell

- Java
- QTP " NeO NiT

It prints all lines which are having Shell, Java and QTP from NeO and NiT files

$egrep command:

The **egrep** command is the most powerful member of the **grep** command family. You can use it like **fgrep** to search for multiple targets. Like, **grep**, it allows you to use regular expressions to specify targets, but it provides, more powerful set regular expressions than **grep**.

The **egrep** command accepts all of the basic regular expressions recognized by **grep**.

Example:

$egrep "Unix|Java|QTP" NiT

Exporting Variables

By default, any variable is a available only in the shell in which it is defined.

Example:

$a=30

$sh

$echo

$a$exit

$echo $a

30

After defining a, we invoked a sub shell. This is achieved by saying **sh** at the prompt. Since a was defined in the parent shell, it no longer holds any identify in the sub shell. If we want variables to be available to all sub shells we must export them from current shell.

$export a

$sh

$echo $a

30

Example:

$export a b c

a=100 b=200 c=300

is same as

a=100 b=200 c=300

$export a b c

Processes in UNIX:

UNIX is we know is a multi-user, multi -tasking operating system. If we want to see which processes are running at any instant type the following

Common Options

-a -e all processes, all users

-e environment/everything

-g process group leaders as well

-l -l long format

-u -u user user oriented report

-x -e even processes not executed from terminals

-f full listing

-w report first 132 characters per line

$ps

UNIX assigns a unique number to every process in memory. This number is called process ID or simply PID. The PID starts with 0 and run upto a maximum of 32767. When the maximum number is reached it starts counting all over again from 0 onwards.

If we want to find out which processes are running for the other users who have logged in execute the ps command with the -a option. -a standing for processes of all the users.

$ps -a

If we want to see what a particular user is doing just say the following

$ps -u user1 where u stands for user and user1 for his login name.

Another useful option available with **ps is -t**. It lets you find out the processes that have been launched from a particular terminal.

$ ps -t tty3d

Additional information that UNIX stores about each running processes. This information can be obtained by using the option -f stands for full listing.

$ps -f

Still More Processes:

So far we have encountered only the processes associated with individual users. But there is several more running in memory which are necessary for the system to work.

$ps -e here -e stands for every process running at that instant.

Background Processes:

Most system processes run in the background, while the users execute their processes in the foreground. To run a process in the background, UNIX provides the ampersand (&) symbol. While executing a command, if the symbol is placed at the end of the command then the command will be executed in the background. When you run a process in the background a number is displayed on the screen. This is PID of the processes that you have just executed in the background.

Ex:$ sort hello.dat > NiT.out & 176 53

The task of sorting the hello.dat and storing the output in NiT.out has now been assigned to the background, the user free to carry other task in the foreground.

Limitations:

1). On termination of a background processes no success or failure is reported on the screen.

2). The output of a background process should always be redirected to a file.

3). With too many processes running in the background the overall system performance is likely to dExrade.

4). If you logout while some of your processes are running in the background all these processes would be abandoned halfway through.

$nohup:

The **nohup** command:

If we are to ensure that the processes that we have executed should not die even when we log out, the **nohup** command is the answer. Using this command we can submit

the time consuming commands in the background, log out and leave the terminal and come next day to see our output ready. **nohup** stands for no hang up.

$nohup sort hello.dat>output.pl

17695

now we can safely log out without our processes getting terminated on logging out.

Note: If we do not redirect the output of our background process the command acts intelligently and stores output in the file 'nohup.out'.

$nohup sort hello.dat

16779

sending output to nohup.out

the nohup.out file is always created in the current directory.

Killing a process:

When a terminal has hanged, infinite loop, system performance has gone below acceptable limits, in these situations you would like to 'kill' the process. To carry out this killing we must first note the PID of the process to be killed using ps command. Kill command to terminate the process.

$ ps

$kill 6173

6173 terminated how the kill command works

at such times we can employ signal number 9. the 'sure kill' signal to forcibly terminate a process as follow.

$kill -9 6173

Scheduling of processes:

cron stands for **chronograph**. During booting UNIX executes this file and displays the message 'cron is started' on the host terminal. Once UNIX launches this process there onwards cron is activated once every minute. When cron wakes up it checks whether any scheduled job is available for it to execute.

Job scheduling

$at

$batch

The at command:

This command is capable of executing UNIX commands at a future date and time.

$ at 17:00

echo "It's 5 PM ! Backup your files and logout"

ctrl+d

then it gives job-id and the date and time when it will be executed.

job id always terminates with .a indicating that this job was submitted using the at command.

There are two options available with the **at** command which permit to view the list of jobs submitted using at and to remove any unwanted jobs from this queue. The options are -l for listing jobs and -r for removing jobs. While removing a submitted job its job-id should be mentioned.

$at -r 853158864.a

$at -l

Some Examples:

$at 0915 am Mar 24

$at 9:15 am Mar 24

$at now+10minutes

$at now+1day

The Batch Command:

Instead of we specifying that our commands be executed at a precise moment in time sometimes we may let the system decide the best time for executing our commands. The way to achieve this is through a command called batch. When we submit our jobs using this command, UNIX executes our job when it is relatively free and the system load is light.

$batch sort hello.dat>adress.out

ctrl+d

Once again note that the '.b' extension given to our job-id signifies that has been submitted using the batch command.

The Null Command :

This seems about as good a time as any to talk about the shell's built-in null command. The format of this command is simply

:

and the purpose of it is—you guessed it—to do nothing. So what good is it? Well, in most cases it's used to satisfy the requirement that a command appear, particularly in if commands. Suppose that you want to make sure that the value stored in the variable system exists in the file /users/steve/mail/systems, and if it doesn't, you want to issue an error message and exit from the program. So you start by writing something like

```
if grep "^$system" /users/steve/mail/systems > /dev/null
then
```

but you don't know what to write after the then because you want to test for the nonexistence of the system in the file and don't want to do anything special if the grep succeeds. Unfortunately, the shell requires that you write a command after the then. Here's where the null command comes to the rescue:

```
if grep "^$system" /users/steve/mail/systems > /dev/null
then
    :
else

    echo "$system is not a valid system"
     exit 1
fi
```

So if the system is valid, nothing is done. If it's not valid, the error message is issued and the program exited.

Remember this simple command when these types of situations arise.

The $$ Variable and Temporary Files

If two or more people on your system use the rolo program at the same time, a potential problem may occur. Look at the rem program and see whether you can spot it. The problem occurs with the temporary file /tmp/phonebook that is used to create a new version of the phone book file.

```
grep -v "$name" phonebook > /tmp/phonebook
```

```
mv /tmp/phonebook phonebook
```

If more than one person uses rolo to remove an entry at the same time, there's a chance that the phone book file can get messed up because the same temporary file will be used by all rolo users.[1] Naturally, the chances of this happening (that is, the preceding two commands being executed at the same time by more than one user) are rather small, but, nevertheless there still is that chance. Anyway, it brings up an important point when dealing with temporary files in general.

[1] Actually, it depends on the users' default file creation mask (known as umask). If one person has created /tmp/phonebook and it's not writable by anyone else, the next person who comes along and tries to create it will get an error message from the shell. The net result is that the first user's file will get properly updated, and the second user's won't; neither file will get corrupted.

When writing shell programs to be run by more than one person, make your temporary files unique. One way is to create the temporary file in the user's home directory, for example. Another way is to choose a temporary filename that will be unique for that particular process. To do this, you can use the special $$ shell variable, which contains the process id number (PID) of the current process:

```
$ echo $$

4668

$ ps

 PID  TTY TIME COMMAND

 4668 co  0:09 sh

 6470 co  0:03 ps

$
```

As you can see, $$ is equal to the process id number of your login shell. Because each process on the Unix system is given a unique process id number, using the value of $$ in the name of a file minimizes the possibility of another process using the same file. So you can replace the two lines from rem with these

```
grep -v "$name" phonebook > /tmp/phonebook$$
```

```
mv /tmp/phonebook$$ phonebook
```

to circumvent any potential problems. Each person running rolo will run it as a different process, so the temporary file used in each case will be different.

The && and || Constructs

The shell has two special constructs that enable you to execute a command based on whether the preceding command succeeds or fails. In case you think this sounds similar to the if command, well it is. It's sort of a shorthand form of the if.

If you write
command1 && command2
anywhere where the shell expects to see a command, command1 will be executed, and if it returns an exit status of zero, command2 will be executed. If command1 returns an exit status of nonzero, command2 gets skipped.

For example, if you write

```
sort bigdata > /tmp/sortout && mv /tmp/sortout bigdata
```

then the mv command will be executed only if the sort is successful. Note that this is equivalent to writing

```
if sort bigdata > /tmp/sortout
then
    mv /tmp/sortout bigdata
fi
```

The command

```
[ -z "$EDITOR" ] && EDITOR=/bin/ed
```

tests the value of the variable EDITOR. If it's null, /bin/ed is assigned to it.
The || construct works similarly, except that the second command gets executed only if the exit status of the first is nonzero. So if you write

```
grep "$name" phonebook || echo "Couldn't find $name"
```

the echo command will get executed only if the grep fails (that is, if it can't find $name in phonebook, or if it can't open the file phonebook). In this case, the equivalent if command would look like

```
if grep "$name" phonebook
then
  :
```

```
else
    echo "Couldn't find $name"
fi
```

You can write a pipeline on either the left- or right-hand sides of these constructs. On the left, the exit status tested is that of the last command in the pipeline; thus

```
who | grep "^$name " > /dev/null || echo "$name's not logged on"
```

causes execution of the echo if the grep fails.
The && and || can also be combined on the same command line:

```
who | grep "^$name " > /dev/null && echo "$name's not logged on" \
  || echo "$name is logged on"
```

(Recall that when \ is used at the end of the line, it signals line continuation to the shell.) The first echo gets executed if the grep succeeds; the second if it fails.
These constructs are also often used in if commands:

```
if validsys "$sys" && timeok
then
    sendmail "$user@$sys" < $message
fi
```

If validsys returns an exit status of zero, timeok is executed. The exit status from this program is then tested for the if. If it's zero, then the sendmail program is executed. If validsys returns a nonzero exit status, timeok is not executed, and this is used as the exit status that is tested by the if. In that case, sendmail won't be executed.

The use of the && operator in the preceding case is like a "logical AND"; both programs must return an exit status of zero for the sendmail program to be executed. In fact, you could have even written the preceding if as

```
validsys "$sys" && timeok && sendmail "$user@$sys" < $message
```

When the || is used in an if, the effect is like a "logical OR":

```
if endofmonth || specialrequest
then
    sendreports
```

```
fi
```

If endofmonth returns a zero exit status, sendreports is executed; otherwise, specialrequest is executed and if its exit status is zero, sendreports is executed. The net effect is that sendreports is executed if endofmonth or specialrequest return an exit status of zero.

The printf Command

Although echo is adequate for displaying simple messages, sometimes you'll want to print formatted output: for example, lining up columns of data. Unix systems provide the printf command. Those of you familiar with the C programming language will notice many similarities.

The general format of the printf command is

```
printf "format" arg1 arg2 ...
```

where format is a string that describes how the remaining arguments are to be displayed. (Note that the format string is a single argument, so it's a good idea to get into the habit of enclosing it in quotes because it often contains whitespace.) Characters in the format string that are not preceded by a percent sign (%) are written to standard output. One or more characters preceded by a percent sign are called conversion specifications and tell printf how the corresponding argument should be displayed. So, for each percent sign in the format string there should be a corresponding argument, except for the special conversion specification %%, which causes a single percent sign to be displayed.

Here's a simple example of printf:

```
$ printf "This is a number: %d\n" 10

This is a number: 10
$
```

printf doesn't add a newline character to its output like echo; however, printf understands the same escape characters that echo does , so adding \n to the end of the format string causes the prompt to appear on the next line.

Although this is a simple case that could easily be handled by echo, it helps to illustrate how the conversion specification (%d) is interpreted by printf: When the format string is scanned by printf, it outputs each character in the string without modification until it sees the percent sign; then it reads the d and recognizes that the %d should be replaced by the next argument, which must be an integer number. After that argument (10) is sent to standard output, printf sees the \n and outputs a newline.

The different conversion specification characters.

printf Conversion Specification Characters

Character	Use for Printing
d	Integers
u	Unsigned integers
o	Octal integers
x	Hexadecimal integers, using a-f
X	Hexadecimal integers, using A-F
c	Single characters
s	Literal strings
b	Strings containing backslash escape characters
%	Percent signs

The first five conversion specification characters are all used for displaying integers. %d displays signed integers, and %u displays unsigned integers; %u can also be used to display the positive representation of a negative number (note that the result is machine dependent). By default, integers displayed as octal or hexadecimal numbers do not have a leading 0 or 0x, but we'll show you how to enable this later in this section.

Strings are printed using %s or %b. %s is used to print strings literally, without any processing of backslash escape characters; %b is used to force interpretation of the backslash escape characters in the string argument.

Here are a few printf examples:

```
$ printf "The octal value for %d is %o\n" 20 20
```

The octal value for 20 is 24

$ printf "The hexadecimal value for %d is %x\n" 30 30

The hexadecimal value for 30 is 1e

$ printf "The unsigned value for %d is %u\n" -1000 -1000

The unsigned value for -1000 is 4294966296

$ printf "This string contains a backslash escape: %s\n" "test\nstring"

This string contains a backslash escape: test\nstring

$ printf "This string contains an interpreted escape: %b\n" "test\nstring"

This string contains an interpreted escape: test string

$ printf "A string: %s and a character: %c\n" hello A

A string: hello and a character: A

$

In the last printf, %c is used to display a single character. If the corresponding argument is longer than one character, only the first is displayed:

$ printf "Just the first character: %c\n" abc

a

$

The general format of a conversion specification is

%[flags][width][.precision]type

The type is the conversion specification character, you can see, only the percent sign and type are required; the other parameters are called modifiers and are optional. Valid flags are -, +, #, and the space character. - left justifies the value being printed; this will make more sense when we discuss the width modifier. + causes printf to precede integers with a + or - sign (by default, only negative integers are printed with a sign). # causes printf to precede octal integers with 0 and hexadecimal integers with 0x or 0X for %#x or %#X, respectively. The space character causes printf to precede positive integers with a space and negative integers with a -.

```
$ printf "%+d\n%+d\n%+d\n" 10 -10 20

+10

-10

+20

$ printf "% d\n% d\n% d\n" 10 -10 20

 10

-10

 20

$ printf "%#o %#x\n" 100 200

0144 0xc8

$
```

As you can see, using + or space as the flag lines up columns of positive and negative numbers nicely.

The width modifier is a positive number that specifies the minimum field width for printing an argument. The argument is right justified within this field unless the - flag is used:

```
$ printf "%20s%20s\n" string1 string2
```

```
        string1         string2

$ printf "%-20s%-20s\n" string1 string2

string1         string2

$ printf "%5d%5d%5d\n" 1 10 100

   1   10  100

$ printf "%5d%5d%5d\n" -1 -10 -100

  -1  -10 -100

$ printf "%-5d%-5d%-5d\n" 1 10 100

1    10   100

$
```

The width modifier can be useful for lining up columns of text or numbers (note that signs for numbers and leading 0, 0x, and 0X characters are counted as part of the argument's width). The width specifies a minimum size for the field; if the width of an argument exceeds width, it is not truncated.

The .precision modifier is a positive number that specifies a minimum number of digits to be displayed for %d, %u, %o, %x, and %X. This results in zero padding on the left of the value:

```
$ printf "%.5d %.4X\n" 10 27

00010 001B

$
```

For strings, the .precision modifier specifies the maximum number of characters to be printed from the string; if the string is longer than precision characters, it is truncated on the right:

```
$ printf "%.5s\n" abcdefg

abcde
```

```
$
```

A width can be combined with .precision to specify both a field width and zero padding (for numbers) or truncation (for strings):

```
$ printf ":%#10.5x:%5.4x:%5.4d\n" 1 10 100

:   0x00001: 000a: 0100

$ printf ":%9.5s:\n" abcdefg
:    abcde:

$ printf ":%-9.5s:\n" abcdefg

:abcde    :

$
```

Finally, if a * is used in place of a number for width or precision, the argument preceding the value to be printed must be a number and will be used as the width or precision, respectively. If a * is used in place of both, two integer arguments must precede the value being printed and are used for the width and precision:

```
$ printf "%*s%*.*s\n" 12 "test one" 10 2 "test two"

   test one      te

$ printf "%12s%10.2s\n" "test one" "test two"

   test one      te

$
```

As you can see, the two printfs in this example produce the same results. In the first printf, 12 is used as the width for the first string, 10 as the width for the second string, and 2 as the precision for the second string. In the second printf, these numbers are specified as part of the conversion specification.

The various conversion specification modifiers.

printf Conversion Specification Modifiers

Modifier	Meaning
flags	
-	Left justify value.
+	Precede integer with + or -.
(space)	Precede positive integer with space character.
#	Precede octal integer with 0, hexadecimal integer with 0x or 0X.
width	Minimum width of field; * means use next argument as width.
precision	Minimum number of digits to display for integers; maximum number of characters to display for strings; * means use next argument as precision.

Here's a simple example that uses printf to align two columns of numbers from a file:

```
$ cat align
#
# Align two columns of numbers
# (works for numbers up to 12 digits long, including sign)
cat $* |
while read number1 number2
do
    printf "%12d %12d\n" $number1 $number2
done
$ cat data
1234 7960
593 -595
395 304
```

```
3234 999

-394 -493

$ align data

    1234    7960

     593    -595

     395     304

    3234     999

    -394    -493

$
```

Editing Files

If you spend any time on a Unix system, you'll want to become comfortable with a text editor. You may want to edit your configuration files, make quick changes to a web page, edit a script you've created, or simply jot yourself some notes: all these activities are easily done with a command line text editor.

In the beginning, Unix had only one editor, the line-by-line editor "ed". A medieval version of Unix contained a new program called "ex"; its most notable feature was allowing people to work with a full screen of text by giving the command "vi". The new display editor proved so popular that AT&T's Unix System V included vi as a separate program.

Since then, vi has remained as a standard command line editor, but two others have emerged: emacs, a full-featured text-editor, and the easy to use pico. This document provides a brief introduction to each and references for how to learn more.

EDITORS

1. ed 2. ex 3. vi

Ed: editor

This is known as line editor means a user can make any changes at line level only. We can't move the cursor to a character position. So this is having some limited features only. When we open this editor by default it is in command mode. **ed** is a line editor for the Unix operating system. It was one of the first end-user programs hosted on the system and has been standard in Unix-based systems ever since. **ed** was originally written by **Ken Thompson** and contains one of the first implementations of regular expressions. Prior to that implementation, the concept of regular expressions was only formalized in a mathematical paper, which **Ken Thompson** had read. **ed** was influenced by an earlier editor known as QED from University of California at Berkeley, Ken Thompson's alma mater. ed went on to influence ex.

Ex editor

ex, short for Extended, is a line editor for Unix systems. The original **ex** was an advanced version of the standard UNIX editor **ed**, included in the Berkeley Software

Distribution. **ex** is similar to ed, with the exception that some switches and options are modified so that they are more user-friendly.

Switches

ex recognizes the following switches:

- (obsolete) suppresses user-interactive feedback

-s (XPG4 only) suppresses user-interactive feedback

-l sets lisp editor option

-r recover specified files after a system crash

-R sets read-only

-t tag Edit the file containing the specified tag

-v invoke visual mode (vi)

-w set window size n

-x set encryption mode

-C encryption option

file specifies file to be edited

vi editor

This editor is known as visual editor. In some of the Linux versions it can also be called as vim (visual Improved) editor. The other names of this are screen editor or character editor. This editor is most user-friendly editor than all others. In this editor we are having 3 modes. They are

Command mode,

Insert mode

Ex command mode

Here Command mode is the default mode when we open the **vi** editor. In this mode we can execute **vi** commands. Append or Insert mode is used to append or insert the data. Escape mode is used to come to the command mode from the insert or append mode, for this press **Esc key**.

Command mode commands

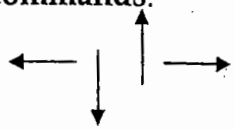
1. Entering in to vi.

a) vi without opening a file.

b) vi filename Opening a file using vi.

2. Basic cursor movement commands.

In UNIXh j k l



Moving the Cursor

the symbol ^ before a letter means that the <Ctrl> key should be held down while the letter key is pressed.

* j or <Return>
  [or down-arrow] move cursor down one line
* k [or up-arrow] move cursor up one line
* h or <Backspace>
  [or left-arrow] move cursor left one character
* l or <Space>
  [or right-arrow] move cursor right one character
* 0 (zero) move cursor to start of current line (the one with the cursor)
* $ move cursor to end of current line
  w move cursor to beginning of next word
  b move cursor back to beginning of preceding word
  :0<Return> or 1G move cursor to first line in file
  :n<Return> or nG move cursor to line n
  :$<Return> or G move cursor to last line in file

(Respectively)

0(zero) go to beginning of the line.

$ go to end of the line.

vi filename edit filename starting at line 1

vi -r filename recover filename that was being edited when system crashed.

Screen Manipulation

The following commands allow the vi editor screen (or window) to move up or down several lines and to be refreshed.

^f move forward one screen

^b move backward one screen

^d move down (forward) one half screen

^u move up (back) one half screen

^l redraws the screen

^r redraws the screen, removing deleted lines

Adding, Changing, and Deleting Text

Unlike PC editors, you cannot replace or delete text by highlighting it with the mouse. Instead use the commands in the following tables.

Perhaps the most important command is the one that allows you to back up and undo your last action. Unfortunately, this command acts like a toggle, undoing and redoing your most recent action. You cannot go back more than one step.

* u UNDO WHATEVER YOU JUST DID, a simple toggle

Inserting or Adding Text

The following commands allow you to insert and add text. Each of these commands puts the vi editor into insert mode; thus, the <Esc> key must be pressed to terminate the entry of text and to put the vi editor back into command mode.

* i insert text before cursor, until <Esc> hit

I insert text at beginning of current line, until <Esc> hit

* a append text after cursor, until <Esc> hit

A append text to end of current line, until <Esc> hit

* o open and put text in a new line below current line, until <Esc> hit

* O open and put text in a new line above current line, until <Esc> hit

Changing Text

The following commands allow you to modify text.

* r replace single character under cursor (no <Esc> needed)

R replace characters, starting with current cursor position, until <Esc> hit

cw change the current word with new text,

starting with the character under cursor, until <Esc> hit

cNw change N words beginning with character under cursor, until <Esc> hit;

e.g., c5w changes 5 words

C change (replace) the characters in the current line, until <Esc> hit

cc change (replace) the entire current line, stopping when <Esc> is hit

Ncc or cNc change (replace) the next N lines, starting with the current line,

stopping when <Esc> is hit

Deleting Text

The following commands allow you to delete text.

* x delete single character under cursor

Nx delete N characters, starting with character under cursor

dw delete the single word beginning with character under cursor

dNw delete N words beginning with character under cursor;

e.g., d5w deletes 5 words

D delete the remainder of the line, starting with current cursor position

* dd delete entire current line

Ndd or dNd delete N lines, beginning with the current line;

e.g., 5dd deletes 5 lines

Cutting and Pasting Text

The following commands allow you to copy and paste text.

yy copy (yank, cut) the current line into the buffer

Nyy or yNy copy (yank, cut) the next N lines, including the current line, into the buffer

p put (paste) the line(s) in the buffer into the text after the current line

Saving and Reading Files

These commands permit you to input and output files other than the named file with which you are currently working.

:r filename<Return> read file named filename and insert after current line (the line with cursor)

:w<Return> write current contents to file named in original vi call

:w newfile<Return> write current contents to a new file named newfile

:12,35w smallfile<Return> write the contents of the lines numbered 12 through 35 to a new file named smallfile

:w! prevfile<Return> write current contents over a pre-existing file named prevfile

To Exit vi

* :x<Return> quit vi, writing out modified file to file named in original invocation

:wq<Return> quit vi, writing out modified file to file named in original invocation

:q<Return> quit (or exit) vi

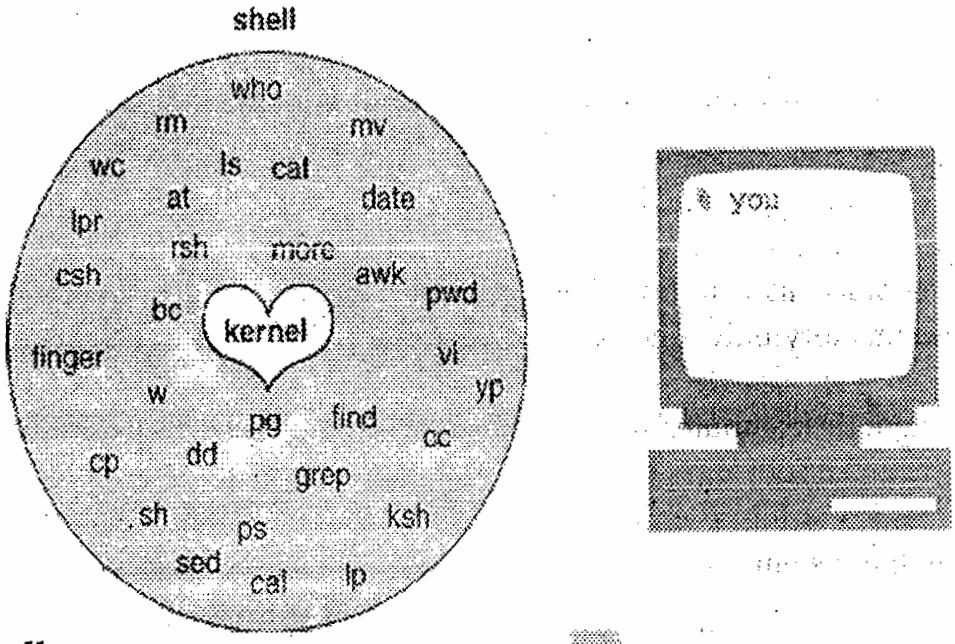
* :q!<Return> quit vi even though latest changes have not been saved for this vi call

8. Control commands.

:w	Save changes in same file
:wq	Save and quit.
:w filename	Save changes in given file
:q	quit without saving.
.num	Go to line num

Shell Scripting

Shell Script file means, a file, which contains a set of commands with in it. If any file contains commands, then that file can be an executable file. Shell Scripts can be useful to execute the set of commands at a single moment of time, we will get our required outputs and those can also be saved under a file. Here executing the commands separately will consume much more time. Using shell scripts we can reduce this time at a greater extent. Shell Scripts can also take arguments is known as Command Line Arguments.

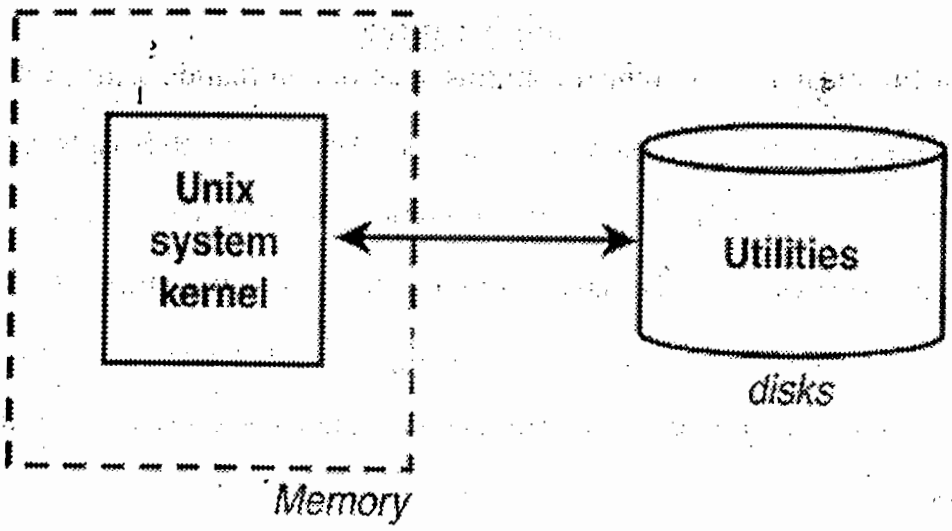


Uses of shells

1. Customizing your work environments.
2. Automating your daily tasks.
3. Automating repetitive tasks
4. Executing important system procedure like shutting down the system
5. performing same operating on many files.

The Kernel and the Utilities

The UNIX system is itself logically divided into two pieces



The kernel is the heart of the UNIX system and resides in the computer's memory from the time the computer is turned on and booted until the time it is shut down.

The utilities, on the other hand, reside on the computer's disk and are only brought into memory as requested. Virtually every command you know under the UNIX system is classified as a utility; therefore, the program resides on the disk and is brought into memory only when you request that the command be executed. So, for example, when you execute the date command, the UNIX system loads the program called date from the computer's disk into memory and initiates its execution.

The shell, too, is a utility program. It is loaded into memory for execution whenever you log in to the system.

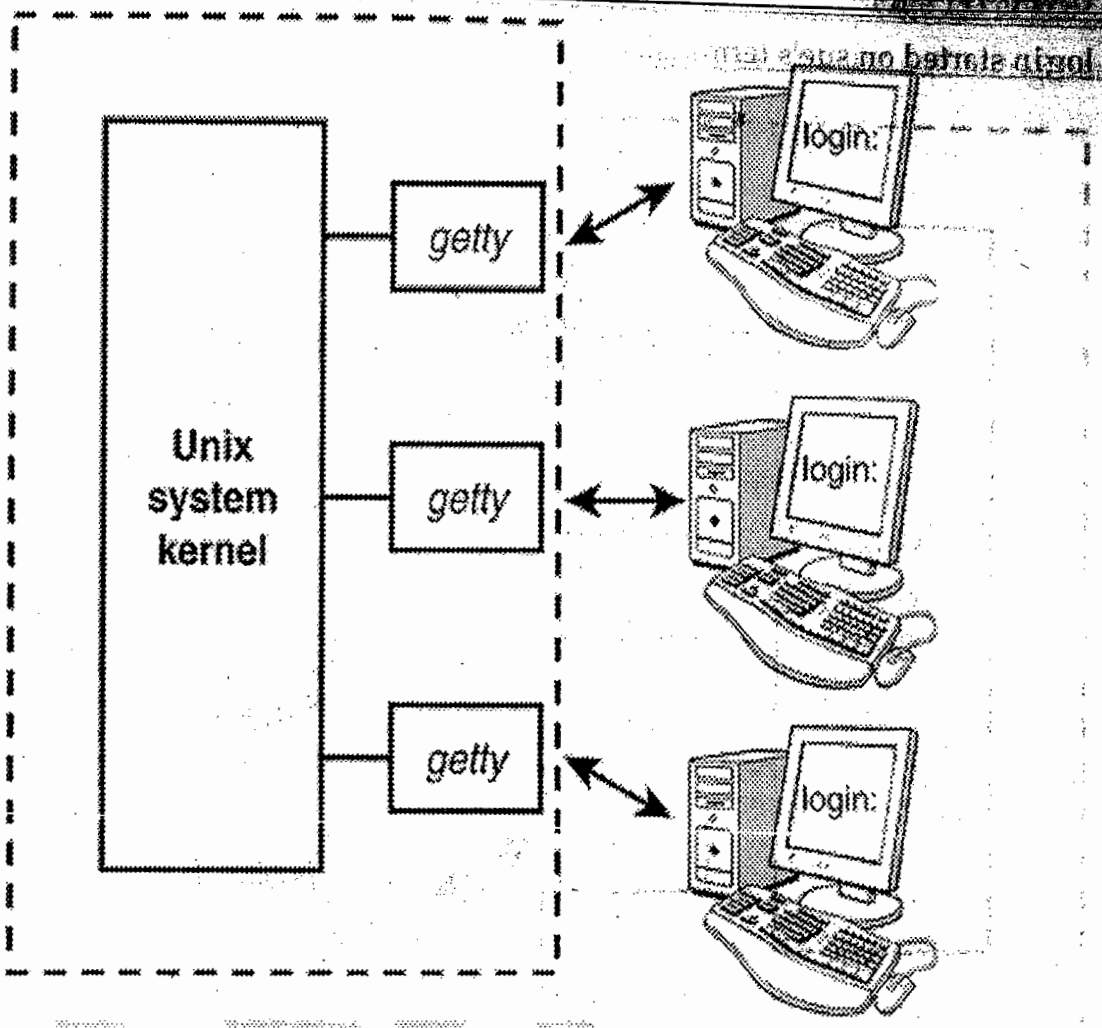
In fact, it's worth learning the precise sequence of events that occurs when the first shell on a terminal or window starts up.

The Login Shell

A terminal is connected to a UNIX system through a direct wire, modem, or network. In the first case, as soon as you turn on the terminal (and press the Enter key a couple of times if necessary), you should get a login: message on your screen. In the second case, you must first dial the computer's number and get connected before the login: message appears. In the last case, you may connect over the network via a program such as ssh, telnet, or rlogin, or you may use some kind of networked windowing system (for example, X Window System) to start up a terminal emulation program (for example, xterm).

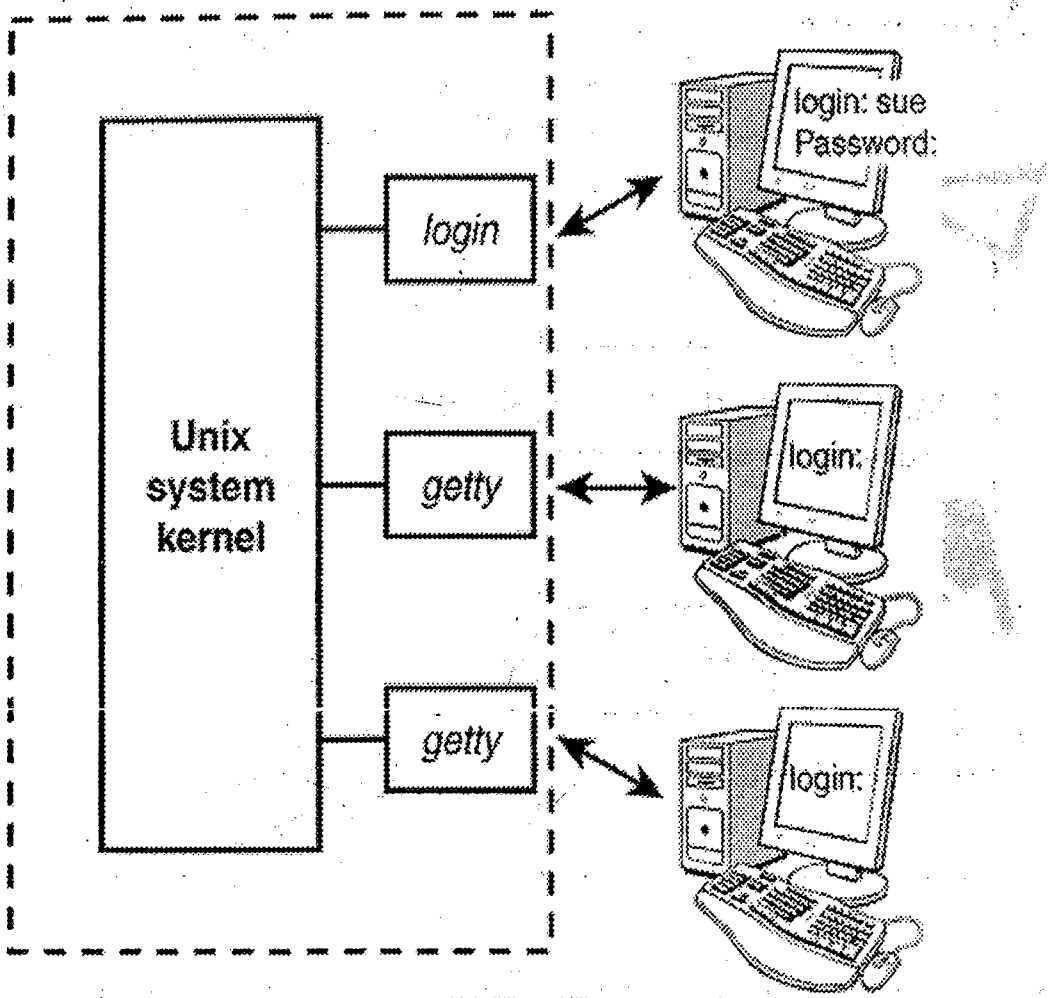
For each physical terminal port on a system, a program called getty will be active.

The getty process:



The Unix system—more precisely a program called init—automatically starts up a **getty** program on each terminal port whenever the system is allowing users to log in. **getty** determines the baud rate, displays the message login: at its assigned terminal, and then just waits for someone to type in something. As soon as someone types in some characters followed by Enter, the **getty** program disappears; but before it goes away, it starts up a program called login to finish the process of logging in. It also gives login the characters you typed in at the terminal—characters that presumably represent your login name.

login started on sue's terminal.

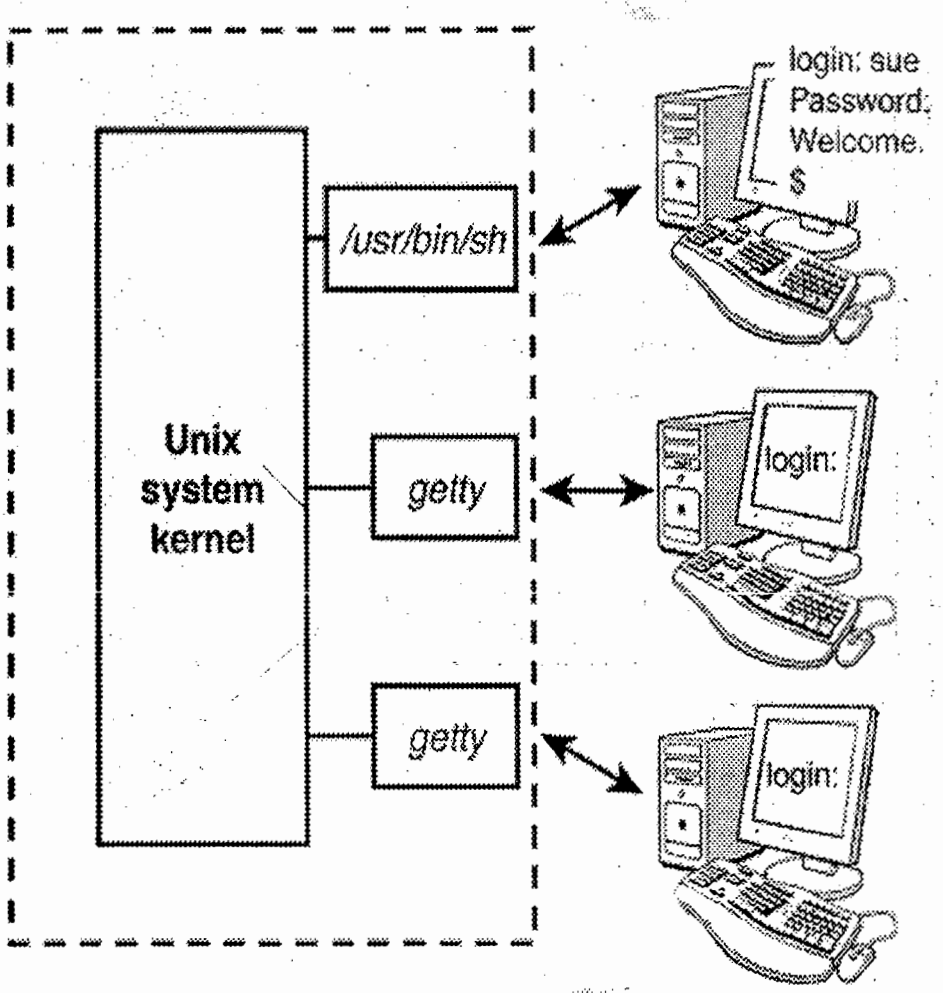


When login begins execution, it displays the string Password: at the terminal and then waits for you to type your password. After you have typed it, login then proceeds to verify your login name and password against the corresponding entry in the file /etc/passwd. This file contains one line for each user of the system. That line specifies, among other things, the login name, home directory, and program to start up when that user logs in.[1] The last bit of information (the program to start up) is stored after the last colon of each line. If nothing follows the last colon, the standard shell /usr/bin/sh is assumed by default. The following three lines show typical lines from /etc/passwd for three users of the system: sue, pat, and bob:

After login checks the password you typed in against the one stored in /etc/shadow, it then checks for the name of a program to execute. In most cases, this will be /usr/bin/sh, /usr/bin/ksh, or /bin/bash. In other cases, it may be a special custom-designed program. The main point here is that you can set up a login account to automatically run any program whatsoever whenever someone logs in to it. The shell just happens to be the program most often selected.

So login initiates execution of the standard shell on sue's terminal after validating her password.

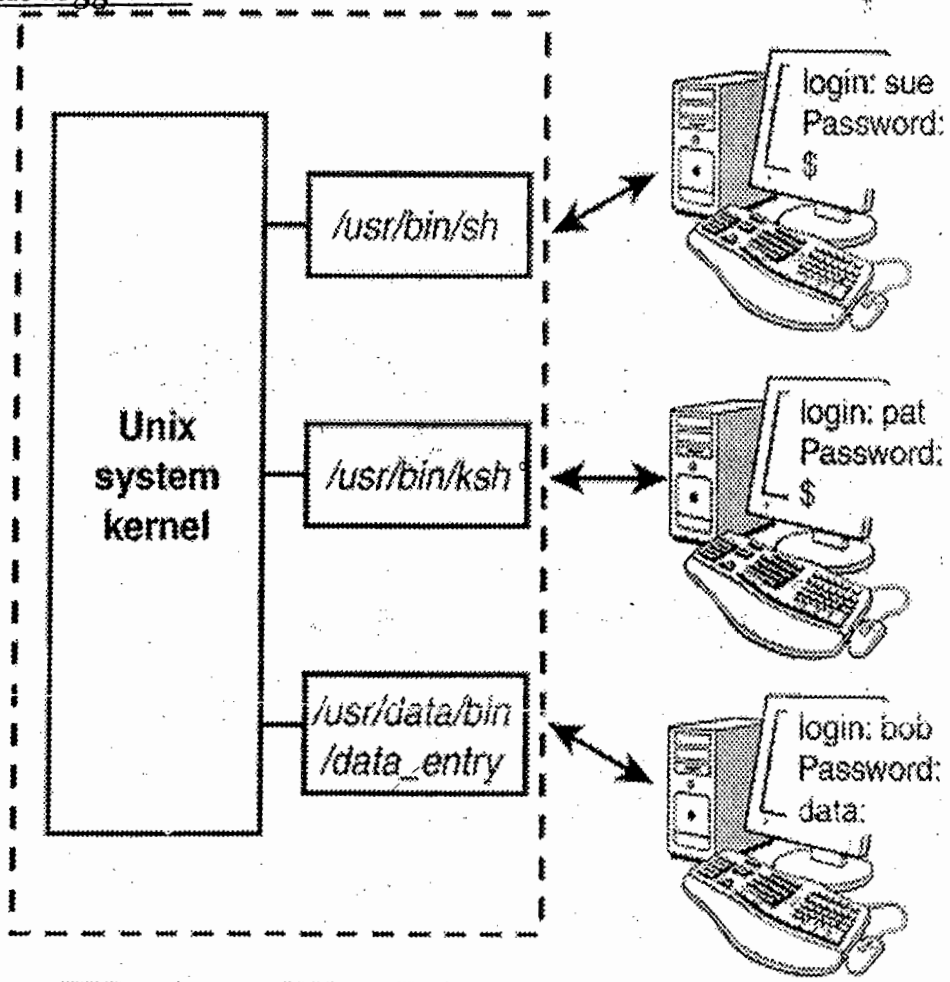
login executes /usr/bin/sh.



According to the other entries from /etc/passwd shown previously, pat gets the program ksh stored in /usr/bin (this is the Korn shell), and bob gets the program data_entry.

The init program starts up other programs similar to getty for networked connections. For example, sshd, telnetd, and rlogind are started to service logins via ssh, telnet, and rlogin, respectively. Instead of being tied directly to a specific, physical terminal or modem line, these programs connect users' shells to pseudo ttys. These are devices that emulate terminals over network connections. You can see this whether you're logged in to your system over a network or on an X Windows screen:

Three users logged in.



Types of Shells

They are different types of shells:

1. The Bourne Shell
2. The C shell
3. The Korn shell
4. Bash, Bourne Again Shells
5. Tcsh, The T C Shell

The Bourne Shell

The original UNIX shell is known as sh, short for Shell or the Bourne Shell, named for Steven Bourne, the creator of sh. As shell go, sh remains fairly primitive, but it was quite advanced for the 1970s. Bourne Shell has been considered a standard part of UNIX for decades. The shell prompt is $, Execution command **sh.**

The C Shell

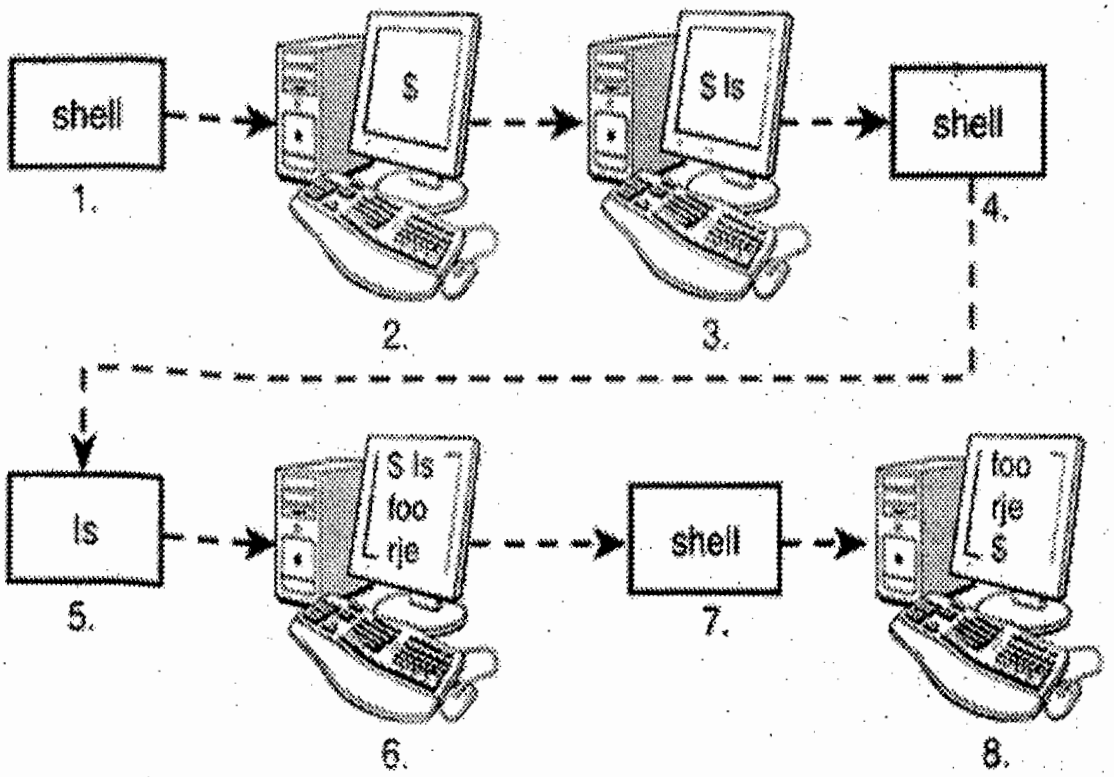
Designed by Bill Joy at the University of California at Berkeley, the C shell was so named because much of its syntax parallels that of the C programming language. The shell prompt is % and execution command **Csh.**

The KORN Shell

The Korn shell became one of the main salvos in AT & T's response to the growing popularity of BSD Unix. Created by David Korn at AT & T Bell Laboratories, the KORN shell or ksh, the prompt is **$.**

Typing Commands to the Shell

When the shell starts up, it displays a command prompt – typically a dollar sign $ – at your terminal and then waits for you to type in a command. Each time you type in a command and press the Enter key (Step 3), the shell analyzes the line you typed and then proceeds to carry out your request (Step 4). If you ask it to execute a particular program, the shell searches the disk until it finds the named program. When found, the shell asks the kernel to initiate the program's execution and then the shell "goes to sleep" until the program has finished (Step 5). The kernel copies the specified program into memory and begins its execution. This copied program is called a process; in this way, the distinction is made between a program that is kept in a file on the disk and a process that is in memory doing things.



If the program writes output to standard output, it will appear at your terminal unless redirected or piped into another command. Similarly, if the program reads input from standard input, it will wait for you to type in input unless redirected from a file or piped from another command (Step 6).

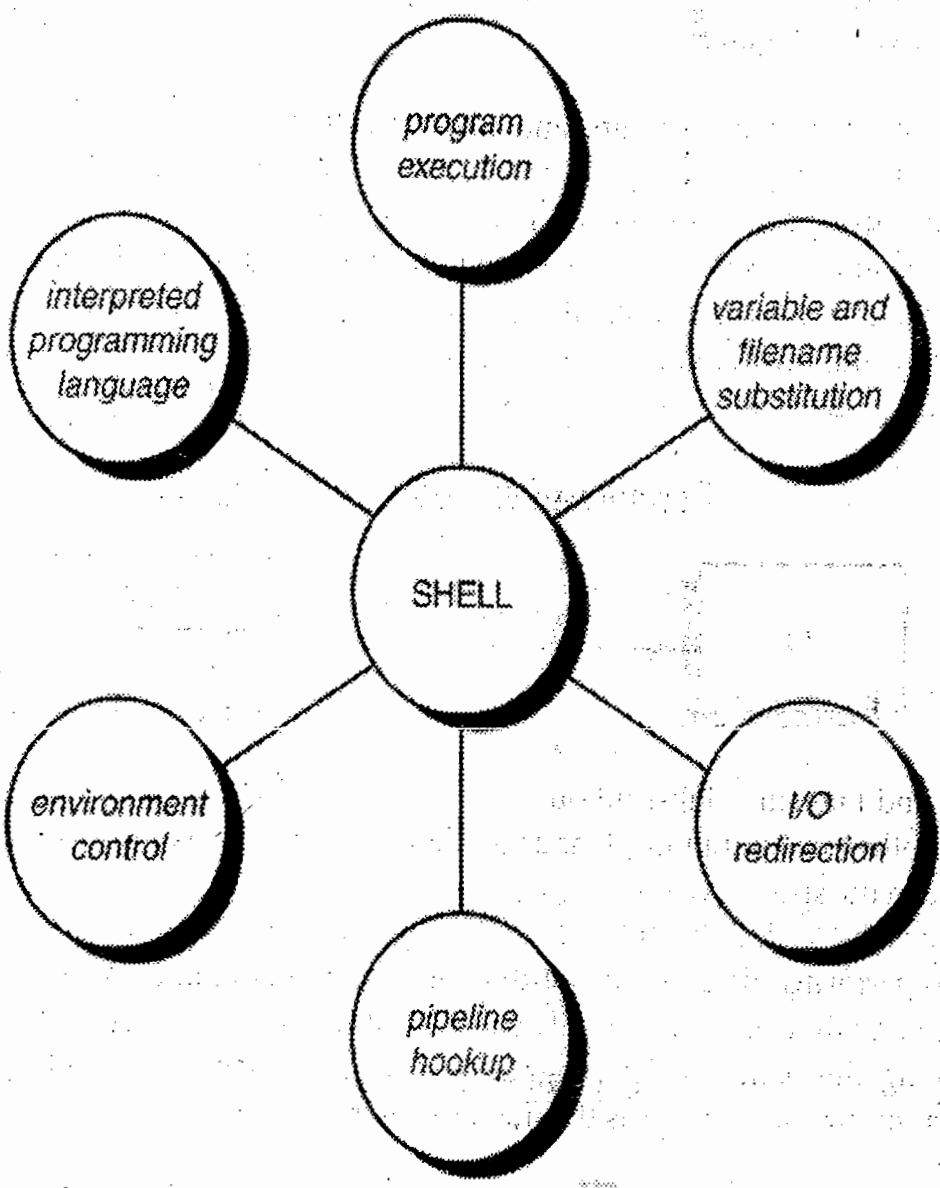
When the command finishes execution, control once again returns to the shell, which awaits your next command (Steps 7 and 8).

Responsibilities of the SHELL

The shell is ultimately responsible for making sure that any commands typed at the prompt get properly executed. The following are the responsibilities of a shell.

1. Program Execution
2. Variable and file Substitution
3. I/O redirection
4. Pipeline hookup
5. Environment Control
6. Interpreted Programming Language

The shell's responsibilities



Program Execution

The shell is responsible for the execution of all programs that you request from your terminal. Each time you type in a line to the shell, the shell analyzes the line and then determines what to do. As far as the shell is concerned, each line follows the same basic format:

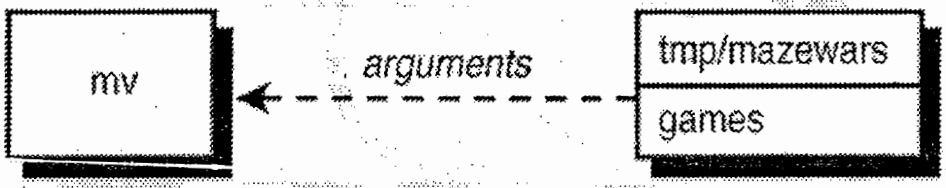
Program-name arguments

The line that is typed to the shell is known more formally as the command line. The shell scans this command line and determines the name of the program to be executed and what arguments to pass to the program. The shell uses special characters to determine where the program name starts and ends, and where each argument starts and ends. These characters are collectively called whitespace characters, and are the space character, the horizontal tab character, and the end-of-line character, known

more formally as the newline character. Multiple occurrences of whitespace characters are simply ignored by the shell. When you type the command.
mv tmp/mazewars games

The shell scans the command line and takes everything from the start of the line to the first whitespace character as the name of the program to execute: mv. The set of characters up to the next whitespace character is the first argument to mv: tmp/mazewars. The set of characters up to the next whitespace character (known as a word to the shell)—in this case, the newline—is the second argument to mv: games. After analyzing the command line, the shell then proceeds to execute the mv command, giving it the two arguments tmp/mazewars and games.

Execution of mv with two arguments.



Variable and Filename Substitution
Like any other programming language, the shell lets you assign values to variables. Whenever you specify one of these variables on the command line, preceded by a dollar sign, the shell substitutes the value assigned to the variable at that point. The shell also performs filename substitution on the command line. In fact, the shell scans the command line looking for filename substitution characters *, ?, or [...] before determining the name of the program to execute and its arguments. Suppose that your current directory contains the files as shown:

```
$ ls
mrs.todd
prog1
shortcut
sweeney
$
```

Now let's use filename substitution for the echo command:

```
$ echo *          List all files
mrs.todd prog1 shortcut sweeney
$
```

How many arguments do you think were passed to the echo program, one or four? Because we said that the shell is the one that performs the filename substitution, the answer is four. When the shell analyzes the line

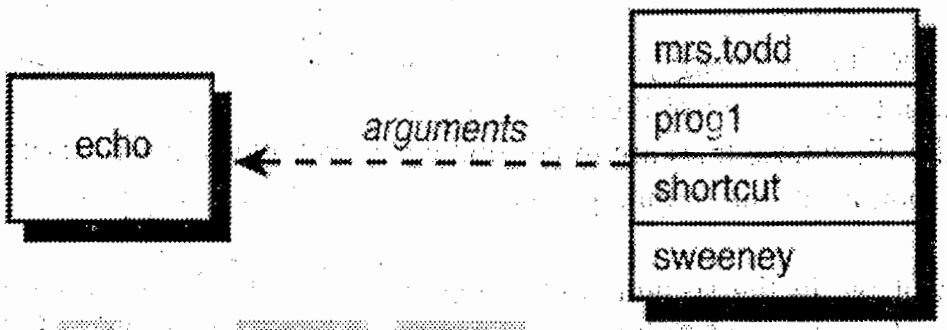
```
echo *
```

it recognizes the special character * and substitutes on the command line the names of all files in the current directory (it even alphabetizes them for you):

echo mrs.todd prog1 shortcut sweeney

Then the shell determines the arguments to be passed to the command. So echo never sees the asterisk. As far as it's concerned, four arguments were typed on the command line

Execution of echo.



I/O Redirection

It is the shell's responsibility to take care of input and output redirection on the command line. It scans the command line for the occurrence of the special redirection characters <, >, or >>.

When you type the command

echo Remember to tape Law and Order > reminder

the shell recognizes the special output redirection character > and takes the next word on the command line as the name of the file that the output is to be redirected to. In this case, the file is reminder. If reminder already exists and you have write access to it, the previous contents are lost (if you don't have write access to it, the shell gives you an error message).

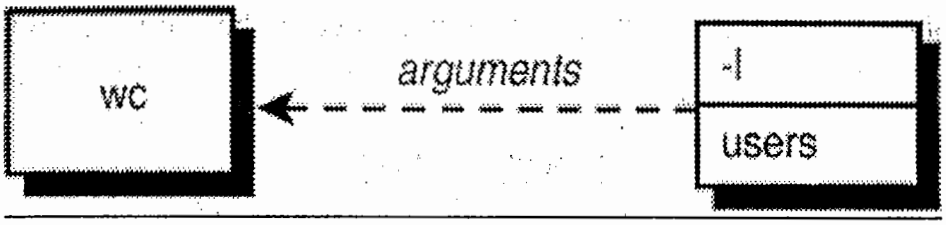
Before the shell starts execution of the desired program, it redirects the standard output of the program to the indicated file. As far as the program is concerned, it never knows that its output is being redirected. It just goes about its merry way writing to standard output (which is normally your terminal, you'll recall), unaware that the shell has redirected it to a file.

Let's take another look at two nearly identical commands:

```
$ wc -l users
    5 users
$ wc -l < users
    5
$
```

In the first case, the shell analyzes the command line and determines that the name of the program to execute is wc and it is to be passed two arguments: -l and user

Execution of wc -l users.



When wc begins execution, it sees that it was passed two arguments. The first argument, -l, tells it to count the number of lines. The second argument specifies the name of the file whose lines are to be counted. So wc opens the file users, counts its lines, and then prints the count together with the filename at the terminal.

Operation of wc in the second case is slightly different. The shell spots the input redirection character < when it scans the command line. The word that follows on the command line is the name of the file input is to be redirected from. Having "gobbled up" the < users from the command line, the shell then starts execution of the wc program, redirecting its standard input from the file users and passing it the single argument -l

Pipeline Hookup

Just as the shell scans the command line looking for redirection characters, it also looks for the pipe character |. For each such character that it finds, it connects the standard output from the command preceding the | to the standard input of the one following the |. It then initiates execution of both programs.

So when you type

who | wc -l

the shell finds the pipe symbol separating the commands who and wc. It connects the standard output of the former command to the standard input of the latter, and then initiates execution of both commands. When the who command executes, it makes a list of who's logged in and writes the results to standard output, unaware that this is not going to the terminal but to another command instead.

When the wc command executes, it recognizes that no filename was specified and counts the lines on standard input, unaware that standard input is not coming from the terminal but from the output of the who command.

Environment Control

The shell provides certain commands that let you customize your environment. Your environment includes your home directory, the characters that the shell displays to

prompt you to type in a command, and a list of the directories to be searched whenever you request that a program be executed.

Interpreted Programming Language

The shell has its own built-in programming language. This language is interpreted, meaning that the shell analyzes each statement in the language one line at a time and then executes it. This differs from programming languages such as C and FORTRAN, in which the programming statements are typically compiled into a machine-executable form before they are executed.

Programs developed in interpreted programming languages are typically easier to debug and modify than compiled ones. However, they usually take much longer to execute than their compiled equivalents.

The shell programming language provides features you'd find in most other programming languages. It has looping constructs, decision-making statements, variables, and functions, and is procedure-oriented. Modern shells based on the IEEE POSIX standard have many other features including arrays, data typing, and built-in arithmetic operations.

Shell Variables

Variable is data name and it is used to store value. Variable value can change during execution of the program.

Variables are two types:

1. System defined variables
2. User defined variables

System defined variables

Environmental variables are used to provide information to the programs you use. You can have both global environment and local shell variables. Global environment variables are set by your login shell and new programs and shells inherit the environment of their parent shell. Local shell variables are used only by that shell and are not passed on to other processes. A child process cannot pass a variable back to its parent process.

The current environment variables are displayed with the "env" or "printenv" commands. Some common ones are:

- DISPLAY The graphical display to use, e.g. nyssa:0.0
- EDITOR The path to your default editor, e.g. /usr/bin/vi
- GROUP Your login group, e.g. staff
- HOME Path to your home directory, e.g. /home/frank
- HOST The hostname of your system, e.g. nyssa
- IFS Internal field separators, usually any white space (defaults to tab, space and <newline>)
- LOGNAME The name you login with, e.g. frank
- PATH Paths to be searched for commands, e.g. /usr/bin:/usr/ucb:/usr/local/bin
- PS1 The primary prompt string, Bourne shell only (defaults to $)
- PS2 The secondary prompt string, Bourne shell only (defaults to >)
- SHELL The login shell you're using, e.g. /usr/bin/csh
- TERM Your terminal type, e.g. xterm
- USER Your username, e.g. frank

Many environment variables will be set automatically when you login. You can modify them or define others with entries in your startup files or at anytime within the shell. Some variables you might want to change are PATH and DISPLAY. The PATH variable specifies the directories to be automatically searched for the command you specify. Examples of this are in the shell startup scripts below.

You set a global environment variable with a command similar to the following for the C shell:

$setenv NAME value

and for Bourne shell:

$ NAME=value; export NAME

User defined variables

These are defined by user and are used most extensively in shell programming.

Creating user defined variables

1. The first character of a variable name should be alphabet or underscore
2. No commas or blanks are allowed within a variable name
3. variables names should be of any reasonable length
4. variable names are case sensitive
5. It shouldn't be reserved word.

Shell keywords

Echo	esac
If	eval
Read	break
Else	exec
Set	continue
If	read-only
Unset	while
Until	do
Trap	ulimit
Case	shift
Wait	Exit
Done	Umask
Export	return
For	

The readonly Command

The readonly command is used to specify variables whose values cannot be subsequently changed. For example,

readonly PATH HOME

makes the PATH and HOME variables read-only. Subsequently attempting to assign a value to these variables causes the shell to issue an error message:

$ PATH=/bin:/usr/bin:.:

$ readonly PATH

$ PATH=$PATH:/users/steve/bin

sh: PATH: is read-only

$

Here you see that after the variable PATH was made read-only, the shell printed an error message when an attempt was made to assign a value to it.

To get a list of your read-only variables, type readonly -p without any arguments:[2]

[2] By default, Bash produces output of the form declare -r variable. To get POSIX-compliant output, you must run Bash with the -posix command-line option or run the set command with the -o posix option.

$ readonly -p
readonly PATH=/bin:/usr/bin:.:
$
unset removes both exported and local shell variables.
You should be aware of the fact that the read-only variable attribute is not passed down to subshells. Also, after a variable has been made read-only in a shell, there is no way to "undo" it.

The unset Command

Sometimes you may want to remove the definition of a variable from your environment. To do so, you type unset followed by the names of the variables:

$ x=100

$ echo $x

100

$ unset x Remove x from the environment

$ echo $x

$

The eval Command

This section describes another of the more unusual commands in the shell: eval. Its format is as follows:

eval command-line

where command-line is a normal command line that you would type at the terminal. When you put eval in front of it, however, the net effect is that the shell scans the command line twice before executing it.[1] For the simple case, this really has no effect:

[1] Actually, what happens is that eval simply executes the command passed to it as arguments; so the shell processes the command line when passing the arguments to eval, and then once again when eval executes the command. The net result is that the command line is scanned twice by the shell.

```
$ eval echo hello
hello
$
```

But consider the following example without the use of eval:

```
$ pipe="|"
$ ls $pipe wc -l
|: No such file or directory
wc: No such file or directory
-l: No such file or directory
$
```

Those errors come from ls. The shell takes care of pipes and I/O redirection before variable substitution, so it never recognizes the pipe symbol inside pipe. The result is that the three arguments |, wc, and -l are passed to ls as arguments.

Putting eval in front of the command sequence gives the desired results:

```
$ eval ls $pipe wc -l
   16
$
```

The first time the shell scans the command line, it substitutes | as the value of pipe. Then eval causes it to rescan the line, at which point the | is recognized by the shell as the pipe symbol.

The eval command is frequently used in shell programs that build up command lines inside one or more variables. If the variables contain any characters that must be seen by the shell directly on the command line (that is, not as the result of substitution), eval can be useful. Command terminator (;, |, &), I/O redirection (<, >), and quote characters are among the characters that must appear directly on the command line to have any special meaning to the shell.

For the next example, consider writing a program last whose sole purpose is to display the last argument passed to it. You needed to get at the last argument in the mycp program in Chapter 10, "Reading and Printing Data." There you did so by shifting all the arguments until the last one was left. You can also use eval to get at it as shown:

```
$ cat last
eval echo \$$#
$ last one two three four
four
$ last *                Get the last file
zoo_report
$
```

The first time the shell scans
echo \$$#

the backslash tells it to ignore the $ that immediately follows. After that, it encounters the special parameter $#, so it substitutes its value on the command line. The command now looks like this:

echo $4
(the backslash is removed by the shell after the first scan). When the shell rescans this line, it substitutes the value of $4 and then executes echo.

This same technique could be used if you had a variable called arg that contained a digit, for example, and you wanted to display the positional parameter referenced by arg. You could simply write

eval echo \$$arg

The only problem is that just the first nine positional parameters can be accessed this way; to access positional parameters 10 and greater, you must use the ${n} construct:
eval echo \${$arg}
Here's how the eval command can be used to effectively create "pointers" to variables:

```
$ x=100
$ ptrx=x
$ eval echo \$$ptrx          Dereference ptrx
100
$ eval $ptrx=50          Store 50 in var that ptrx points to
$ echo $x              See what happened
50
$
```

The wait Command

If you submit a command line to the background for execution, that command line runs in a subshell independent of your current shell (the job is said to run asynchronously). At times, you may want to wait for the background process (also known as a child process because it's spawned from your current shell—the parent) to finish execution before proceeding. For example, you may have sent a large sort into the background and now want to wait for the sort to finish because you need to use the sorted data.

The wait command is for such a purpose. Its general format is

wait process-id

where process-id is the process id number of the process you want to wait for. If omitted, the shell waits for all child processes to complete execution. Execution of your current shell will be suspended until the process or processes finish execution. You can try the wait command at your terminal:

```
$ sort big-data > sorted_data &        Send it to the background

[1] 3423                    Job number & process id from the shell

$ date                      Do some other work

Wed Oct  2 15:05:42 EDT 2002

$ wait 3423                 Now wait for the sort to finish
```

$ When sort finishes, prompt is returned

The $! Variable

If you have only one process running in the background, then wait with no argument suffices. However, if you're running more than one command in the background and you want to wait on a particular one, you can take advantage of the fact that the shell stores the process id of the last command executed in the background inside the special variable $!. So the command

wait $!

waits for the last process sent to the background to complete execution. As mentioned, if you send several commands to the background, you can save the value of this variable for later use with wait:

```
prog1 &
pid1=$!
...
prog2 &
pid2=$!
...
wait $pid1      # wait for prog1 to finish
...
wait $pid2      # wait for prog2 to finish
```

We are having two types of shell scripts. They are

1) Non-Interactive Shell Scripts.
2) Interactive Shell Scripts.

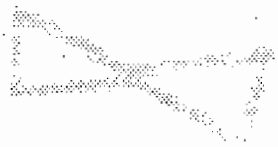
1) Non-Interactive Shell Scripts:-

If we are not interacting with the shell script while it is executing then it is known as Non-Interactive shell scripts.

Examples:

```
# This is a sample script, to display message
echo "Welcome to the world of UNIX Shell Scripting"
echo "good morning"

# Shell Script @1
# Finding the biggest of two numbers.
a=123
b=167
if [ $a -gt  $b ]
then
      echo " a is big"
fi

# Shell Script @2
# Finding the biggest of two numbers
a=123
b=234
if [ $a -gt $b ]
then
      echo " a is big "
else
      echo " b is big "
# Shell Script @3
# Script for display the list of files, the current working user's list and present working directory.
ls -x
who
```

```
pwd
# Shell script @4
# Script for count number of users
Echo "There are `who | wc -l` users
```



2) Interactive Shell Scripts:-

If we are interacting with the shell script while it is executing then it is known as Interactive shell scripts.

$read:- This command is used to take a value from the console or an end user and save that value in a variable.

```
# This is a sample script ro read input, interactive
echo -n "enter a value :"
read x
echo "given value = $x"

# sample1 script to get two values and calculate sum
echo -n "enter first value :"
read x
echo -n "enter second value :"
read y
((z = x + y))
echo "result = $z"

# sample script to get input through command line arguments,
# or also called positional parameters
echo "script name = $0"
echo "no.of args = $#"
```

```
echo "first arg = $1"
echo "second arg = $2"
echo "all args = $@"

# input through files, store the data in a file separated by spaces
# this can read one line
# first field assigned to x and rest of the line assigned to y
echo -n "enter file name :"
read f
read x y < $f
echo "x = $x   y = $y"

# example to calculate sqrt
# since there is no function in shell we can call a cprogram
#!/bin/bash
echo -n "enter a num :"
read x
r=`./sqrt $x`
echo "square root = $r"

# example on reading strings and calculating length
echo -n "enter a string :"
read x
echo "given string = $x"
echo "length = ${#x}"

# reading passwords, display is disabled
echo -n "enter a password :"
```

```
stty -echo -icanon
read x
stty echo icanon
echo
echo "given password = $x"

echo -n "enter user name :"
read x
who | grep $x > /dev/null && echo "user logged in"
who | grep $x > /dev/null || echo "user not logged in"
```

Arithmetic Operators

+	Addition
-	Subtract
*	Multiplication
/	Division
%	Mod
^	Power

Relational Operators

On Strings

=	two strings are equal
!=	two strings are not equal
-n	a string has non-zero length
-z	a string has zero length

On Integers

-eq	two intExers are equal
-ne	two intExers are not equal

-lt less than

-le less than or equal to

-gt greater than

-ge greater than or equal to

On Files

-f a file is an ordinary file

-d a file is a directory file

-s a file has non-zero length

Logical Operators to combine Expressions

-a And -o Or ! Not

The Logical Negation Operator !

The unary logical negation operator ! can be placed in front of any other test expression to negate the result of the evaluation of that expression. For example,

```
[ ! -r /users/steve/phonebook ]
```

returns a zero exit status (true) if /users/steve/phonebook is not readable; and

```
[ ! -f "$mailfile" ]
```

returns true if the file specified by $mailfile does not exist or is not an ordinary file. Finally,

```
[ ! "$x1" = "$x2" ]
```

returns true if $x1 is not identical to $x2 and is obviously equivalent to

```
[ "$x1" != "$x2" ]
```

The Logical AND Operator -a

The operator -a performs a logical AND of two expressions and returns true only if the two joined expressions are both true. So

```
[ -f "$mailfile"  -a  -r "$mailfile" ]
```

returns true if the file specified by $mailfile is an ordinary file and is readable by you. An extra space was placed around the -a operator to aid in the expression's readability and obviously has no effect on its execution.

The command

```
[ "$count" -ge 0  -a  "$count" -lt 10 ]
```

will be true if the variable count contains an integer value greater than or equal to zero but less than 10. The -a operator has lower precedence than the integer comparison operators (and the string and file operators, for that matter), meaning that the preceding expression gets evaluated as

```
("$count" -ge 0) -a ("$count" -lt 10)
```

as you would expect.

Parentheses

Incidentally, you can use parentheses in a test expression to alter the order of evaluation; just make sure that the parentheses are quoted because they have a special meaning to the shell. So to translate the preceding example into a test command, you would write

```
[ \( "$count" -ge 0 \) -a \( "$count" -lt 10 \) ]
```

As is typical, spaces must surround the parentheses because test expects to see them as separate arguments.

The Logical OR Operator -o

The -o operator is similar to the -a operator, only it forms a logical OR of two expressions. That is, evaluation of the expression will be true if either the first expression is true or the second expression is true.

```
[ -n "$mailopt" -o -r $HOME/mailfile ]
```

This command will be true if the variable mailopt is not null or if the file $HOME/mailfile is readable by you.

The -o operator has lower precedence than the -a operator, meaning that the expression

```
"$a" -eq 0  -o  "$b" -eq 2 -a  "$c" -eq 10
```

gets evaluated by test as

```
"$a" -eq 0  -o  ("$b" -eq 2 -a  "$c" -eq 10)
```

Naturally, you can use parentheses to change this order if necessary:

```
\( "$a" -eq 0  -o  "$b" -eq 2 \) -a "$c" -eq 10
```

Conditional Statements

Every programming language provides at least one statement that lets you conditionally perform some actions based on the results of a question. In the shell, the **if** statement performs this function.

Syntax of if is as follows

if [condition or test]

then

statements

statements

.....

fi

Examples

Write a script to change a directory

```
# Script @ 5
echo Enter directory name
read fname
if cd $fname
then
echo "dir changed"
pwd
fi
```

If the condition or test is true then the commands in between the **then** and the **fi** will be executed; otherwise they will be skipped.

Syntax: if.... then...else...fi

if [condition or test]

then

statements

statements

```
.....
else
    statements
    statements
    .....
fi
```

Examples:

Simple if and if ..... then ..... else Statements:-

```
# Finding the Biggest of two Strings
echo " enter one string"
read first
echo "enter second string"
read sec
        if [ $first = $sec ]
        then
            echo " First string is big"
         else
           echo "second string is big"

# example on if, to check string is null or not
echo -n "enter a string :"
read x
if [ -z "$x" ]
then
echo "no input ,script will be terminated"
exit
else
```

```
echo "given string = $x"
echo "length = ${#x}"
fi
# this is a sample script on time based input
echo -n "enter a string :"
read -t 10 x
if [ $? -eq 0 ]
then
echo "given string = $x"
else
echo "time out"
exit
fi

# second syntax of if
#!/bin/bash
echo -n "enter an user name : "
read n
if grep $n /etc/passwd > /dev/null
then
echo "login name $n is present"
else
echo "login name $n not present"
fi

# example on using a menu and if
echo "add sub mul"
echo -n "enter your choice :"
```

```
read op
op=`echo "$op" | tr [:upper:] [:lower:]`
if [ "$op" = "add" -o "$op" = "sub" -o "$op" = "mul" ]
then
echo -n "enter two values :"
read x y
fi
if [ $op = "add" ]
then
echo "res = $((x+y))"
elif [ $op = "sub" ]
then
echo "res = $((x-y))"
elif [ $op = "mul" ]
then
echo "res = $((x*y))"
else
echo "invalid option"
fi

#Script 10
echo -n "enter first:"
read x
echo -n "enter second :"
read y
if [ "$x" \> "$y" ]
then
echo "$y"
```

```
echo "$x"
else
echo "$x"
echo "$y"
fi
```

LOOPS:

All programming languages provide mechanisms that enable you to execute repeatedly a group of statements. The shell is no exception, as it provides two such language statements: the for and the while. This section discusses then for, the more widely used of the two. The general format of the for statement in the shell is:

- **for statement**

for variablename **in** list of values

do command

command

.....

done: variable is any shell variable that you choose. It is listed without a leading dollar sign. The number of items specified in list determines the number of times the commands enclosed between the **do** and **done** will be executed. Each time the loop is executed, the next value in list is assigned to variable.

 # example on using for

```
for((i=1;i<=5;i++))
do
echo "hello"
done
```

 # calculate sum of squares

```
s=0
for((i=1;i<=5;i++))
do
((s = s + i*i))
done
echo "sum of square = $s"
```

```
# displaying multiplication table
echo -n "enter a value :"
read x
for((i=1;i<=10;i++))
do
((p = x * i))
echo "$x x $i = $p"
done

# example on sum of n values using for
echo -n "enter how many values :"
read x
for((i=1;i<=x;i++))
do
echo -n "enter value $i : "
read n
((s = s+ n))
done
echo "sum = $s"

# example on nested for
for((i=1;i<=5;i++))
do
for((j=4;j>=i;j--))
do
echo -n " "
done
```

```
for((j=1;j<=i;j++))
do
echo -n "* "
done
echo
done

# example on nested for
for((i=1;i<=5;i++))
do
for((j=1;j<=i;j++))
do
echo -n "* "
done
echo
done

# example to display file and dir
for i in *
do
if [ -f "$i" ]
then
echo "file -> $i"
elif [ -d "$i" ]
then
echo "dir -> $i"
fi
done
```

```
#script for wait  5 sec.
echo -n "enter a name (5 seconds) : "
old=`stty -g`
stty -icanon min 0 time 50
read x
stty "$old"
if [ -z "$x" ]
then
echo "timed out"
else
echo "given name $x"
fi

# display the list of values
for val  in  10 20 30 40 50
do
  echo "$val"
done
```

The above example will display 10, 20, 30, 40, and 50.

The while Statement

The **while** is another looping statement provided by the shell. It enables you to repeatedly execute a set of commands while a specified condition is true. The format of the while is:

```
while [ condition ]
do
    command
    command
```

.....

done

```
# sample on using while
c=1
while [ $c -le 10 ]
do
echo "hello"
((c++))
done

# reading input till eof
# calculating no.of lines & no.of chars in the given input
c=0
while read l
do
((c++))
((n = n+ ${#l} + 1))
done
echo "no.of lines = $c"
echo "no.of chars = $n"

# reading from a file
# calculating how many times a word 'unix' repeated
echo -n "enter file :"
read f
c=0
while read l
```

```
do
for i in $l
do
if [ "$i" = "unix" ]
then
((c++))
fi
done
done < $f
echo "no.of times unix = $c"
# script can be executed in the background
# which will intimate when a particular user log in
u=$1
if [ $# -eq 1 ]
then
while [ 1 ]
do
w=`who | cut -d " " -f1`
for i in $w
do
if [ "$u" = "$i" ]
then
echo
echo "$u logged in"
exit
fi
done
sleep 30
```

```
done
else
echo "usage : s85 username"
fi

#user logged in or not
echo -n "enter user name :"
read x
if who | grep $x > /dev/null
then
 echo "user logged in"
else
 echo "user not logged in"
fi

#generating password
x="abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789"
c=1
while [ $c -le 8 ]
do
((p = $RANDOM % 62))
w=${x:$p:1}
pw=`echo "${pw}$w"`
let "c++"
done
echo "password = ${pw}"
```

In the above syntax, until the condition is true the commands with in the **do** and **done** will be repeatedly executed.

Here we are introducing one new command, i.e.

More on Loops

Breaking Out of a Loop

Sometimes you may want to make an immediate exit from a loop. To just exit from the loop (and not from the program), you can use the break command, whose format is simply

break

When the break is executed, control is sent immediately out of the loop, where execution then continues as normal with the command that follows the done.

The Unix command true serves no purpose but to return an exit status of zero. The command false also does nothing but return a nonzero exit status. If you write

```
while true
do
    ...
done
```

the while loop will theoretically be executed forever because true always returns a zero exit status. By the way, the : command also does nothing but return a zero exit status, so an "infinite" loop can also be set up with

```
while :
do
    ...
done
```

Because false always returns a nonzero exit status, the loop

```
until false
do
    ...
done
```

will theoretically execute forever.

The break command is often used to exit from these sorts of infinite loops, usually when some error condition or the end of processing is detected:

```
while true
do
    cmd=$(getcmd)

    if [ "$cmd" = quit ]
    then
        break
    else
        processcmd "$cmd"
    fi
done
```

Here the while loop will continue to execute the getcmd and processcmd programs until cmd is equal to quit. At that point, the break command will be executed, thus causing the loop to be exited.

If the break command is used in the form

break n
the n innermost loops are immediately exited, so in

```
for file
do
    ...
    while [ "$count" -lt 10 ]
    do
        ...
        if [ -n "$error" ]
        then
            break 2
        fi
        ...
    done
    ...
done
```

both the while and the for loops will be exited if error is nonnull.

Skipping the Remaining Commands in a Loop

The continue command is similar to break, only it doesn't cause the loop to be exited, merely the remaining commands in the loop to be skipped. Execution of the loop then continues as normal. Like the break, an optional number can follow the continue, so

```
continue n
```

causes the commands in the innermost n loops to be skipped; but execution of the loops then continues as normal.

```
for file
do
    if [ ! -e "$file" ]
    then
        echo "$file not found!"
        continue
    fi

    #
    # Process the file
    #

    ...
done
```

Each value of file is checked to make sure that the file exists. If it doesn't, a message is printed, and further processing of the file is skipped. Execution of the loop then continues with the next value in the list. Note that the preceding example is equivalent to writing

```
for file
do
    if [ ! -e "$file" ]
    then
        echo "$file not found!"
    else
        #
        # Process the file
        #
        ...
    fi
```

done

Executing a Loop in the Background

An entire loop can be sent to the background for execution simply by placing an ampersand after the done:

```
$ for file in memo[1-4]
> do
>       run $file
> done &              Send it to the background
[1] 9932
$
request id is laser1-85 (standard input)
request id is laser1-87 (standard input)
request id is laser1-88 (standard input)
request id is laser1-92 (standard input)
```

I/O Redirection on a Loop

You can also perform I/O redirection on the entire loop. Input redirected into the loop applies to all commands in the loop that read their data from standard input. Output redirected from the loop to a file applies to all commands in the loop that write to standard output:

```
$ for i in 1 2 3 4
> do
>       echo $i
> done > loopout          Redirect loop's output to loopout
$ cat loopout
1
2
3
4
$
```

You can override redirection of the entire loop's input or output by explicitly redirecting the input and/or output of commands inside the loop. To force input or output of a command to come from or go to the terminal, use the fact that /dev/tty always refers to your terminal. In the following loop, the echo command's output is explicitly redirected to the terminal to override the global output redirection applied to the loop:

```
for file
do
    echo "Processing file $file" > /dev/tty
    ...
done > output
```

echo's output is redirected to the terminal while the rest goes to the file output. Naturally, you can also redirect the standard error output from a loop, simply by tacking on a 2> file after the done:

```
while [ "$endofdata" -ne TRUE ]
do
    ...
done 2> errors
```

Here output from all commands in the loop writing to standard error will be redirected to the file errors.

Piping Data Into and Out of a Loop

A command's output can be piped into a loop, and the entire output from a loop can be piped into another command in the expected manner. Here's a highly manufactured example of the output from a for command piped into wc:

```
$ for i in 1 2 3 4
> do
>     echo $i
> done | wc -l
    4
$
```

Typing a Loop on One Line

If you find yourself frequently executing loops directly at the terminal, you'll want to use the following shorthand notation to type the entire loop on a single line: Put a semicolon after the last item in the list and one after each command in the loop. Don't put a semicolon after the do.

Following these rules, the loop

```
for i in 1 2 3 4
do
    echo $i
done
```

becomes
for i in 1 2 3 4; do echo $i; done
And you can type it in directly this way:
$ for i in 1 2 3 4; do echo $i; done
1
2
3
4
$
The same rules apply to while and until loops.
if commands can also be typed on the same line using a similar format:

$ if [1 = 1]; then echo yes; fi
yes
$ if [1 = 2]; then echo yes; else echo no; fi
no
$

Note that no semicolons appear after the then and the else.

The Until Statement

The **Until** is another looping statement provided by the shell. It enables you to repeatedly execute a set of commands while a specified condition is false. The format of the until is:

```
until [ condition ]
do
    command
    command
    ……
done
```

```
# example on using until
c=1
until [ $c -gt 10 ]
```

```
do
echo "hello"
((c++))
Done
```

expr (expression)

This command is used to perform a single arithmetic calculation and display the result. Although the POSIX standard shell supports built-in integer arithmetic operations, older shells don't. It's likely that you may see command substitution with a Unix program called expr, which evaluates an expression given to it on the command line:

```
$ expr 1 + 2
3
$
```

Each operator and operand given to expr must be a separate argument, thus explaining the output from the following:

```
$ expr 1+2
1+2
$
```

The usual arithmetic operators are recognized by expr: + for addition, - for subtraction, / for division, * for multiplication, and % for modulus (remainder).

```
$ expr 10 + 20 / 2

20
$
```

Multiplication, division, and modulus have higher precedence than addition and subtraction. Thus, in the preceding example the division was performed before the addition.

```
$ expr 17 * 6
expr: syntax error
$
```

What happened here? The answer: The shell saw the * and substituted the names of all the files in your directory! It has to be quoted to keep it from the shell:

```
$ expr "17 * 6"
17 * 6
$
```

That's not the way to do it. Remember that expr must see each operator and operand as a separate argument; the preceding example sends the whole expression in as a single argument.

```
$ expr 17 \* 6
102
$
```

Naturally, one or more of the arguments to expr can be the value stored inside a shell variable because the shell takes care of the substitution first anyway:

```
$ i=1
$ expr $i + 1
2
$
```

This is the older method for performing arithmetic on shell variables. Do the same type of thing as shown previously only use the command substitution mechanism to assign the output from expr back to the variable:

```
$ i=1
$ i=$(expr $i + 1)      Add 1 to i
$ echo $i
2
$
```

In legacy shell programs, you're more likely to see expr used with back quotes:

```
$ i=`expr $i + 1`      Add 1 to i
$ echo $i
3
$
```

Note that like the shell's built-in integer arithmetic, expr only evaluates integer arithmetic expressions. You can use awk or bc if you need to do floating point calculations. Also note that expr has other operators. One of the most frequently used ones is the : operator, which is used to match characters in the first operand against a regular expression given as the second operand. By default, it returns the number of characters matched.

The expr command

expr "$file" : ".*"

returns the number of characters stored in the variable file, because the regular expression .* matches all the characters in the string.

Examples:
```
$ expr 1 + 2
3
$ expr 5 \* 50
250
$ expr 200 % 50
0
$ count=10
$ expr $count + 10
20
$ expr 234 - 197
37
$ expr 200 / 50
4
```

The case--in--esac Statement:

The shell **case** statement is useful when you want to compare a value against a whole series of values. You know that this can be done with an **if - elif** statement chain, but the **case** statement is more concise and also easier to write and to read. The general format of this statement is as follows.

```
case value
in
        pattern1) command
                  command
                  ..........;;
        pattern2) command
                  command
                  ..........;;
        .......
        .......
        pattern n)      command
                  command
                  ..........;;

          *)      command
                  command
                  ..........;;
```

esac:

Operation of the **case** is as follows: value is successively compared against pattern1, pattern2, ... pattern n. As soon as a match is found, the commands listed after the matching pattern are executed, until a **double semicolon** is reached. At that point, the

case statement is terminated. If value does not match any of the specified patterns, then no action is taken, and the special pattern * commands will be executed.

Here one sample example:

 # Shell Script @5

```
#!/bin/bash
# example on case
echo -n "enter a single digit : "
read x
case $x in
0) echo "zero";;
1) echo "one";;
2) echo "two";;
3) echo "three";;
4) echo "four";;
*) echo "enter 0 to 4";;
esac
#!/bin/bash
# finding given char is alpha or digit or other
echo -n "enter a char : "
read x
case $x in
[[:alpha:]]) echo "an alphabet";;
[0-9]) echo "digit";;
*) echo "other char";;
Esac
#!/bin/bash
# the following is a sample program
# to use , s52 val1 operator val2
```

```
# example s52 10 + 20
x=$1
y=$3
case $2 in
+) echo "sum = $((x+y))";;
-) echo "diff = $((x-y))";;
\*) echo "prod = $((x*y))";;
*) echo "use +/-/*";;
esac
# example on using menu and case
echo "add sub mul"
echo -n "enter your choice :"
read op
op=`echo "$op" | tr [:upper:] [:lower:]`
if [ "$op" = "add" -o "$op" = "sub" -o "$op" = "mul" ]
then
echo -n "enter two values :"
read x y
fi
case $op in
add)      echo "sum = $((x+y))";;
sub)      echo "res = $((x-y))";;
mul)   echo "res = $((x*y))";;
*) echo "invalid selection"
esac
```

Example Scripts

1). Write a program to copy a file

```
vi copy.sh
echo Enter source filename and target filename
read src trg
if cp $src $trg
then
echo file copied successfully
else
echo file fail to copy
fi
```

2). Write a program to check given no is even or odd

```
vi even.sh
echo "Enter any number"
read n
if['expr $n%2' -eq 0]
then
echo "$n is even number"
else
echo "$n is odd number"
fi
```

3). Write a program given string is empty or not

```
vi string.sh
echo "Enter a string"
read str
if[-z $str]
then
echo "String is NiTty"
```

```
else
echo "String is not NiTty"
fi
```

4). Write a program to display all sub-directories in the current directory.

```
Vi hello.sh
For I in *
do
if[-d $i]
then
echo $i
fi
done
```

5). Write a program print 1 to 10 numbers

```
vi number.sh
echo "Enter the number from 1 to 10 "
i=1
while[$i-le 10]
do
echo $i
I='expr $i+1'
done
```

6). Write a program to delete given file

```
vi rama.sh
echo "Enter any file which is existed"
read fname
if rm $fname
then
echo "File deleted successfully"
```

```
else
echo "File not deleted"
fi
:wq
```

7. Count files in directory (not hidden one's)

```
#!/bin/ksh
echo * | wc -w
```

8.To change uppercase filenames to lowercase

```
#!/bin/sh
if [ $# -eq 0 ] ; then
echo Usage: $0 Files
exit 0
fi
for f in $* ; do
g=`echo $f | tr "[A-Z]" "[a-z]"`
echo mv -i $f $g
mv -i $f $g
done
```

9.Find ten day old files

```
#/bin/ksh
echo "This script will put the names of ten day old files in /tmp in /tmp/checkold.txt
file"
find /tmp -size 0 -atime +10 -exec ls -l {} \; > /tmp/checkold.txt
find /tmp -size 0 -atime +10 -exec rm -f {} \;
```

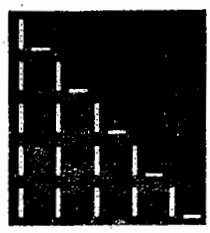
10. swap two files (if they exist):

```
if test -f $1
then
 if test -f $2
 then
  mv $1 tempfile.temp
  mv $2 $1
  mv tempfile.temp $2
 else
  echo "$2 is not a file. Quitting"
 fi
else
 echo "$1 isn't a file"
fi
```

11. Dispaying small or big

```
for a in 4 8 3 7 13 4
do
 echo -n $a" "
 if test $a -ge 5
 then
  echo "big"
 else
  echo "small"
 fi
done
```

12. Display the steps of success



```
echo "Climb the steps of success"

for (( i=1; i<=5; i++ ))
do
   for (( j=1; j<=i; j++ ))
   do
    echo -n " |"
   done
   echo "_"
done
```

13. Write script to print given number in reverse order, for eg. If no is 123 it must print as 321.

```
if [ $# -ne 1 ]
then
   echo "Usage: $0   number"
   echo "       I will find reverse of given number"
   echo "       For eg. $0 123, I will print 321"
   exit 1
fi
n=$1
```

```
rev=0
sd=0
while [ $n -gt 0 ]
do
   sd=`expr $n % 10`
   rev=`expr $rev \* 10 + $sd`
   n=`expr $n / 10`
done
   echo "Reverse number is $rev"
```

14. Write script to print given numbers sum of all digit, For eg. If no is 123 it's sum of all digit will be 1+2+3 = 6.

```
if [ $# -ne 1 ]
then
   echo "Usage: $0   number"
   echo "       I will find sum of all digit for given number"
   echo "       For eg. $0 123, I will print 6 as sum of all digit (1+2+3)"
   exit 1
fi

n=$1
sum=0
sd=0
while [ $n -gt 0 ]
do
   sd=`expr $n % 10`
   sum=`expr $sum + $sd`
```

```
  n=`expr $n / 10`
done
  echo  "Sum of digit for numner is $sum"
```

15. Display Good Morning, Good Afternoon, Good Evening , according to system time.

```
temph=`date | cut -c12-13`
dat=`date +"%A %d in %B of %Y (%r)"`

if [ $temph -lt 12 ]
then
  mess="Good Morning $LOGNAME, Have nice day!"
fi

if [ $temph -gt 12 -a $temph -le 16 ]
then
  mess="Good Afternoon $LOGNAME"
fi

if [ $temph -gt 16 -a $temph -le 18 ]
then
  mess="Good Evening $LOGNAME"
fi
```

16. Display the following pattern

1

22

```
333
4444
55555
echo "Can you see the following:"

for (( i=1; i<=5; i++ ))
do
   for (( j=1; j<=i;  j++ ))
   do
    echo -n "$i"
   done
   echo ""
done
```

17. Display the following pattern

```
1
12
123
1234
12345

echo "Can you see the following:"

for (( i=1; i<=5; i++ ))
do
   for (( j=1; j<=i;  j++ ))
   do
    echo -n "$j"
```

```
  done
  echo ""
done
```

18. Displays the stars pattern

```
echo "Stars"
for (( i=1; i<=5; i++ ))
do
  for (( j=1; j<=i; j++ ))
  do
   echo -n " *"
  done
  echo ""
done

for (( i=5; i>=1; i-- ))
do
  for (( j=1; j<=i; j++ ))
  do
   echo -n " *"
  done
  echo ""
done
```