Chapter 2

UNIX: Operating System and Shells

2.1 INTRODUCTION

This chapter introduces you to the UNIX Operating System and gives a brief introduction to UNIX shells so that you can easily migrate to the practicing sessions discussed in next chapter**.** Computer software can be roughly divided into two kinds: the system programs and the application programs. The system programs manage the operation of the computer itself, while the application programs solve the problems of the users. The most fundamental of all system programs is the operating system whose purpose is to control a computer's activities.

There are two main functions that an operating system performs: first, it presents the user with an equivalent of a virtual machine that is easier to program than the underlying hardware. Secondly, the operating system acts as a resource manager, by keeping track of who is using which resource, to grant resource requests, to account for usage and to mediate conflicting requests from different programs and users.

Operating systems have evolved through the years. Various operating systems differ in how they do their job and what additional features they offer. The UNIX operating system is one such operating system developed by Bell laboratories.

2.1.1 History of UNIX

UNIX is the happy outcome of the proverbial rags-to-riches story. What is now heralded as the most powerful and popular multiuser operating system had a very humble beginning in the premises of AT & T's Bell laboratories. The origin of UNIX can be traced back to 1965, when a joint venture was undertaken by Bell Telephone Laboratories, the General Electric Company and Massachusetts Institute of Technology.

The aim was to develop an operating system that could serve a large community of users and allow them to share data, if need be. This enterprise was called Multics, for Multiplexed Information and Computing Service. Even after much time, resources and efforts had been devoted to the project, the convenient, interactive computing service failed to materialise. This led Dennis Ritchie and Ken

Thompson, both of AT & T, to start afresh on what their mind's eye had so illustriously envisioned. Thus, in 1969, the two along with a few others evolved what was to be the first version of the multiuser system UNIX. Though this was not tapped to the fullest, it had all the trappings of a truly potent multiuser operating system. This system was christened "UNIX" by Brian Kernighan, as a very reminder of the ill-fated Multics. The UNIX Operating System was re-written in a high level language, C, designed and implemented by Ritchie.

2.1.2 Features of UNIX

The UNIX system has many useful features, the most important of which are its

- multitasking capability
- multiuser capability
- transportability
- large selection of powerful UNIX supplied programs
- communications and electronic mail
- library of applications software

Multitasking Capability

Multitasking means performing more than one task at a time. When you print a file and while it is printing you start editing another document, you are performing multitasking operations. Multitasking lets you simultaneously perform tasks formerly performed sequentially. This is managed by dividing the CPU time intelligently between all processes or tasks.

Multiuser Capability

A multiuser system permits several users to use the same computer simultaneously. In a multiuser system, the same computer resources-hard disk, memory, etc.-are accessible to many users. A terminal, in turn, is a monitor and a keyboard, which are the input and output devices for the user.

A user at any terminal can not only use the resources of the computer but also the peripherals that may be attached, for example: a printer. One can easily appreciate how economical such a setup is than having as many computers as there are users, and also how much more convenient when the same data is to be shared by all. The heart of a UNIX installation is the host machine, often known as a server or a console.

Transportability

The UNIX system itself is very portable, the reason being that UNIX is written in C. That is, it is easier to modify the UNIX System code for installation on a new computer than to rewrite another operating system from the scratch for the new computer.

The ability to transport the UNIX system from one brand of computer to another has been the major reason for the acceptance of this system. There are two types of portability to be considered: the portability of the UNIX operating system itself and of the application programs. More than 90% of the kernel is written in C and hence can be easily transported to another machine. Also, the application programs written in higher level languages are also portable. Portable applications decrease programming costs.

UNIX System—Supplied Tools

The UNIX system comes with several supplied programs. These programs are divided into two classes; namely, integral utilities and tools, which are discussed below.

Integral utilities Integral utilities are part of the UNIX system that provide such assistance to the operating system that they are absolutely necessary for the practical operation of a computer with UNIX. One example is the UNIX system command interpreter or the shell program. Without this you could not request your UNIX system to perform any work for you.

Tools Tools, on the other hand, are programs that are not necessary to the computer's basic operation but provide significant additional value to UNIX. These tools include many application programs. Integral utilities are utilities that let you set up sophisticated automatic procedures to perform a series of tasks that would otherwise have to be performed as many separate actions. UNIX system tools include an extensive "electronic filing cabinet", word processing, typesetting capabilities, and many other programs.

Communications and electronic mail

UNIX communications include the following options:

a. Communicating between different terminals hooked into the same computer.
b. Communicating between the users of one computer with users of another computer(the second computer being of a different brand in the same location).
c. Communicating between computers of different sizes and types in different locations.

Third party application programs

In addition to application programs supplied by Bell, a library of over 500 UNIX application programs have been developed and sold by various computer manufacturers and software companies. UNIX application programs take advantage of the UNIX's multiuser and multitasking capabilities and can be used by more than one person at a time.

2.1.3 The Structure of the UNIX System

The UNIX system may be functionally viewed as consisting of the kernel, the shell and utilities, as shown in Fig. 2.1. The role of each of these is given briefly in the next section.

The kernel

1. schedules programs,
2. manages data/file access and storage,
3. enforces security mechanisms, and
4. performs all hardware access.

The shell

1. presents each user by a prompt,
2. interprets the commands typed by a user,
3. executes user commands, and
4. supports a custom environment for each user.

Utilities

1. File management (rm, cat, ls, rmdir, mkdir)
2. User management (passwd, chmod, chgrp)
3. Process management (kill, ps)
4. Printing (lp, troff, pr)
5. Program development tools

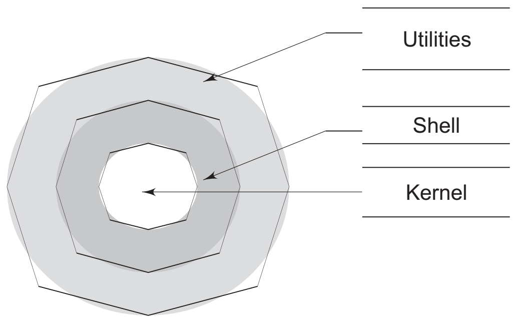


Fig. 2.1 *Schematic of basic UNIX system*

2.1.4 UNIX Architecture

The UNIX system can be regarded as a kind of pyramid as shown in Fig. 2.2.

At the bottom is the hardware consisting of the CPU, memory, disks, terminals and other devices. Running on the bare hardware is the UNIX operating system. Its function is to control the hardware and provide a system call interface to all the programs. These system calls allow the users to create and manage processes, files and other resources. Programs also make system calls by issuing instructions to switch from the user mode to kernel mode to start up UNIX. So a library is provided with one procedure (these procedures are written in assembly language, but can be called from C) per system call.

In addition to the operating system and system call library, there are a large number of standard programs (which may differ from version to version) such as command processors (shell), editors, compilers etc. Thus, there are three different interfaces to UNIX: true system call interface, the library interface, and the interface formed by a set of standard utility programs.

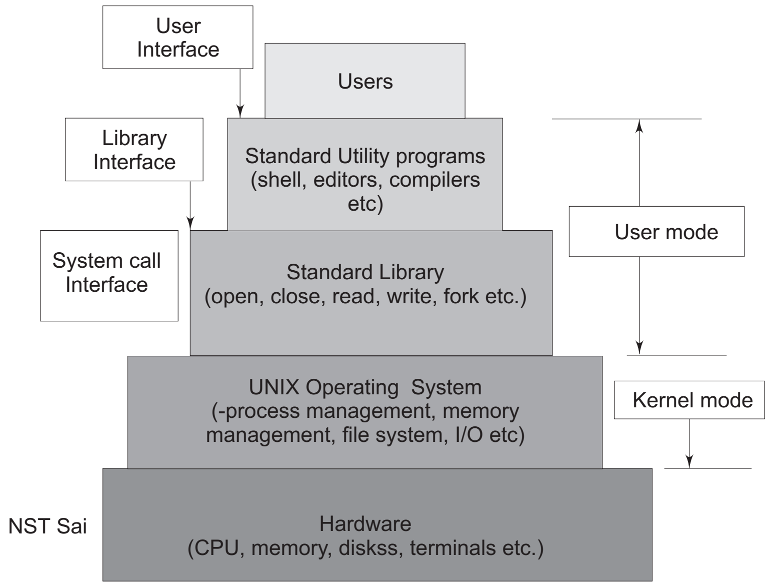

User Interface
Users
Library Interface
Standard Utility programs (shell, editors, compilers etc)
System call Interface
Standard Library (open, close, read, write, fork etc.)
User mode
UNIX Operating System (-process management, memory management, file system, I/O etc)
Kernel mode
NST Sai
Hardware (CPU, memory, diskss, terminals etc.)

Fig. 2.2 *The layered architecture of a UNIX system*

2.2 UNIX FILE SYSTEM

2.2.1 Introduction

All computer applications need to store and retrieve information. For many applications, the information must be retained for weeks, months or forever. The solution to these problems is to store information on disk or other media in units called files. Processes can then read them and write new ones if need be. A file is a unit of data that is stored on a magnetic disk (or tape). A filename identifies the data uniquely. A collection of files on the disk is called a file system. A directory is a special type of file that contains a list of filenames.

Some operating systems use a flat or one-dimensional file system, where all files reside in a single directory. The UNIX file system allows a hierarchical structure. The programs and data can be organized conveniently since files can be grouped according to usage. Files are managed by the operating system. The operating system design deals with how these files are structured, named, accessed, used, protected and implemented. That part of the operating system that deals with the files is known as the file system.

The UNIX file system is a structure for organizing information. When a process wants to read or write a file, it must first open the file. A file is opened with a OPEN call, whose first argument gives the path name of the file to be opened and the second one specifies whether a file is to be read, written, or both. The system checks to see if the file exists and, if it exists, it checks to see if the caller has the permission to access it in the desired way as if that is also permitted. Than the system returns a small positive integer called a file descriptor, to the caller. If the access is prohibited then it returns -1 to indicate an error.

The calls for reading and writing the file use the file descriptor to identify the file. When a process starts up, it has three descriptors available: 0 for standard input, 1 for standard output and 2 for standard error. The first file opened is given the file descriptor 3 and next one 4 and so on. When a file is closed, its file descriptor is freed and can be allocated on a subsequent open. There are two ways to specify file names in UNIX. The first is using an absolute path starting from the root, and the second is a relative path, which is specified relative to the working directory.

2.2.2 Files

Introduction

A file is a collection of information in the form of data, an application, or documents. When a file is created, UNIX assigns the file a unique internal number called **inode**. The majority of the UNIX versions allow the file name to be of 14 characters in length, though some versions also allow longer file names. All versions of UNIX recognize at least three types of files:

Ordinary files This type of file is used to store data. Users can add data to ordinary files using an editor. Executable programs are also stored as ordinary files.

Directory files A directory file contains a list of files. Each entry in the file consists of two parts: the name of the file and a pointer to the actual file on the disk. Directories behave just like ordinary files except that some of the operations that you would use for manipulating ordinary files do not work for the directories and vice versa.

The directory is a structure for organizing files. In the list of filenames that a directory contains, the names may refer to ordinary files, special files or to other directory files. Since directories may be listed inside or other directories, complex hierarchical structures of ordinary, special and directory files result.

Special files These files are used to reference physical devices such as terminal printers, disks etc. They are read from and written to just like ordinary files but such requests cause activation of the associated physical device.

Special files are divided into two categories: block and character. A block special file is one consisting of a sequence of numbered blocks. The key property of the block special file is that each block can be individually addressed and accessed. A program can open a block special file and directly read the block 124, for instance, without reading blocks 0 to 123. Block special files are used for disks. Character special files are normally used for devices that input or output a character stream.

Some basic commands for file manipulation

• **ls : List files in a Directory** **ls** is used to display file names of the directory in an alphabetical order where you type this command. This command has many options that allow us to see the detailed and variety of information about the file.

• **cat : Concatenate and Print a File** **cat** is a short form for concatenate, which means to join together. This utility is often used to display the contents of a single file, although you can use it to display multiple files in succession. The command is of the form

```
$cat filename
```

and it displays the contents of the ***filename*** on the monitor, which is the standard output device.

• **cp : Copy an Ordinary File** **cp** command allows you to create a duplicate copy of an ordinary file. The command line format for using the cp command is

```
$cp srcfile destfile
```

Here the ***srcfile*** is the original or the source file and ***destfile*** is the name of the copy to be created. The important use of this command is to create a backup.

- **rm : Remove an Ordinary File** This command removes one or more ordinary files from the directory, effectively erasing the file. The rm utility removes a file by deleting its pointer in the appropriate directory. In this way, the link between the filename and the physical file is destroyed, so the file can no longer be accessed. The command format is

```
$rm filename...
```

where multiple filenames can be typed separated by spaces.

- **mv : Move (Rename) a file** **mv** command changes the name associated with a file by associating a new filename with a pointer to the physical file in the directory and deleting the link to the old filename. The command is as follows

```
$mv oldfile newfile
```

This changes the name of the file from ***oldfile*** to ***newfile***

- **ln : Creating Filename Aliases** The UNIX file system allows more than one filename for the same physical file, which means that it is possible to have aliases for any given file. The command line format is as follows

```
$ln oldfile newfile
```

After the **ln** linking the ***oldfile*** and the ***newfile,*** both refer to the same file. *S*o to remove a file with more than one links you have to destroy all the links.

2.2.3 File Access Permissions

Introduction

The data in the UNIX system is contained in the form of files. One may want to restrict or permit access to this data by restricting or permitting access to the files containing the data. The UNIX system allows an easy means of controlling the file access that the system users may have to all three types of files (ordinary, directory, special files).

Even for a sole user, this is a necessity so that you don't accidentally damage the contents of your files. So in a multiuser system, restricting access becomes all the more important, so that you can protect your file from being read, written, or erased by other users in the system.

Types of File Access Permissions

UNIX provides three different types of class users and three different modes of file access. These three classes of users and modes of access give rise to nine different kinds of access permissions allowed within the UNIX file system.

There are three classes of system users:

Owner (denoted by *u* for users) Every file has an owner. The owner is the system user who created the file. The superuser can change the individual ownership of a file if necessary. The owner has full control over restricting or permitting access to the file at any time.

Group (denoted by *g*) It is possible to have one or more system users own the file collectively in a kind of group ownership. The group is more than one user who may access the file. A system user who is not the file owner may access the file if he is a group owner, but he cannot restrict or permit access to those files unless he himself is the owner.

Other (denoted by *o*) The other category refers to any other user of the system. These are users who are neither individuals nor group owners.

In addition to classes of file users, there are three ways of accessing a file. The meaning of these access modes is somewhat different for ordinary files than it is for directories. These modes and their meanings are summarized in Table 2.1.

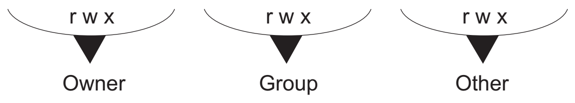
Table 2.1 File Access Modes and Their Meanings

Access Mode	Ordinary File	Directory File
Read	Allows examination of the file contents	Allows listing of files within the directory
Write	Allows changing the contents of the File	Allows creating new files and removing old ones.
Execute	Allows executing file as Command	Allows searching directory

- System users with read permission for files may read (examine) the contents of an ordinary file(for example, by using **cat**), and with read permission for directory can read the contents of a directory (by using the **ls** command).
- System users with write permission for files may write to a file and change its contents (for example, by using an editor); those with write permission for directory can use certain privileged programs to be written on a directory (allows creation and removal of files itself). Write permission is also required to delete a file.
- And finally, the system users with execute permission may execute the file as a command. Executing makes sense only if the file is an executable program or a shell script. The execute permission for the directory file is called search permission, since directories may be searched but not executed like a command. A user must have the execute permission for a directory in order to access the files named in that directory. Thus, even if you had read and write permissions for an ordinary file that was listed in a directory, unless you had an execute permission for the directory containing the ordinary file, the UNIX system would not allow you to read or write the contents of the ordinary file.

Determining File Access Permissions

The three classes of file users (owners, group and others) may be combined with the three types of access(read, write and execute) to give nine possible sets of permissions as shown here:



The presence of permission is indicated by the appropriate letter being in its correct location. The absence of a permission is indicated by a dash **(-)** in the same place.
For example:
r - - r - - r - - means this file can be read by all three user classes but cannot be written or executed by anyone and
r w x - - - - - - means this file can be read, written and executed by the owner but is not accessible to groups and others.

Changing File Access Permissions

The **chmod** command (meaning change mode) allows you to change permission modes of one or more files or directories.

```
$chmod [who] op-code permission… file…
```

The *who* argument tells **chmod** the user class and may be any of the following:

- **u** User (owner)
- **g** Group file owner
- **o** Other users
- **a** All users (owner, group and others)

The *op-code* argument represents the operation to be performed by **chmod**:

- **+** Add the specified permission to the existing permission
- **–** Remove the indicated permission from the existing permissions
- **=** Assign the indicated permissions.

The permission argument uses the following abbreviations:

- **r** read
- **w** write
- **x** execute

Now, for example, a command of the type,

```
$chmod go-rw letter
```

means to remove the read and write permissions from the file **letter** for the group and the other users of the file.

2.2.4 The File System Hierarchy

Introduction

The UNIX file system has quite a few significant features. It has a hierarchical structure providing support for directories. These files are expandable, enabling of growing as required. Files are treated as byte streams so that they can contain any characters. Security rights are associated with files and directories enabling read, write, execute privileges for owner, group and others. Files may be shared enabling concurrent access. Hardware devices are treated just like files. Up until now, we have seen only a directory containing entries of ordinary files. But, a directory can contain other directories. This arrangement gives rise to the tree-like branching structure of the file system.

Figure 2.3 shows the structure diagrammatically.

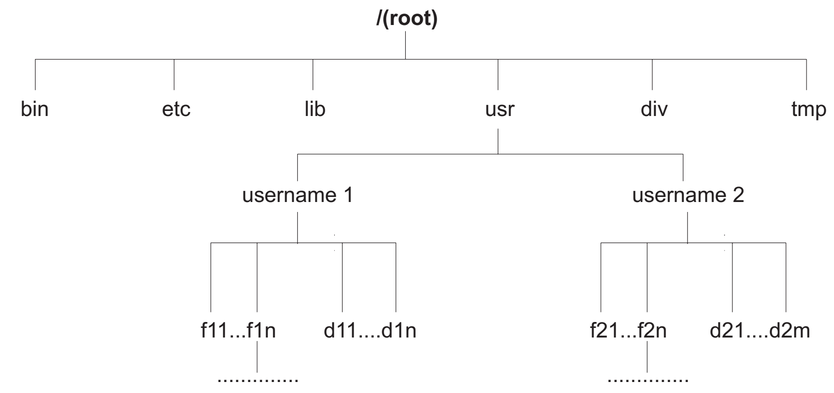


Fig. 2.3 *A sample file system*

The UNIX file system begins with a directory called root. The root directory is designated by a slash(/). Branching from the root are several other directories (named bin, dev, usr, tmp, lib, etc.). These directories are considered as subdirectories of the root directory and conversely the root is considered to be the parent of these directories. Each subdirectory can point to other subdirectories and to other ordinary files, which is the upside down branching as shown in the Fig. 2.3.

The main reason for having different directories is to keep some files at once together and separated from other files. For example, files that are used by the system might be kept in certain directories such as bin, lib, etc., while files created by the system users may be kept in other directories; for example, in tmp and the subdirectories of usr.

Figure 2.4 shows the representation of the entries for the directory file username 1.

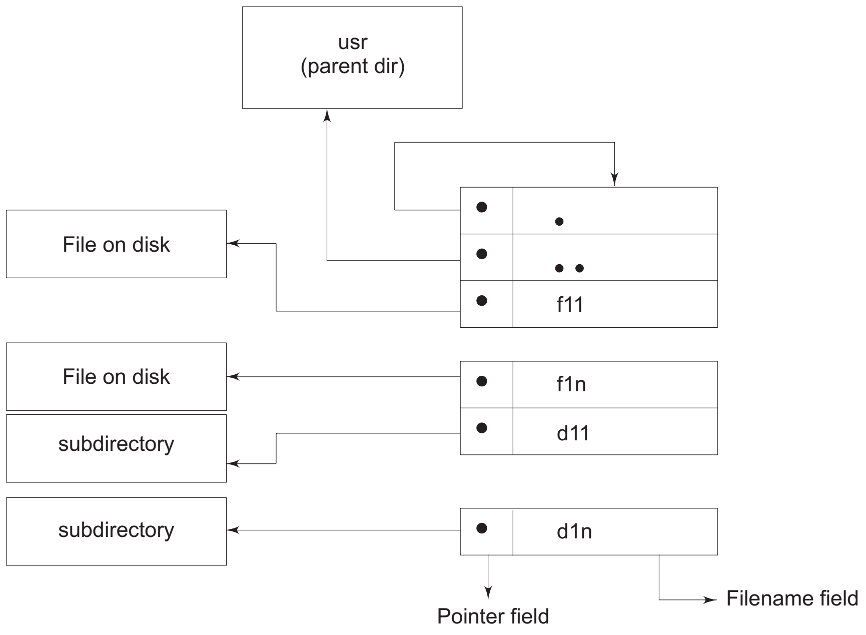


Fig. 2. 4 *Detail of directory structure for username1*

Working Directories and Pathnames

When you first sign on to your UNIX system you begin to work in a particular directory known as your home directory. Your home directory is a unique fixed directory that is assigned when your system administrator establishes your account.

The directory in which you are working at any point of time is known as your working directory or current directory and it need not be fixed; it can be changed. Every file in UNIX has one unique absolute pathname and it begins with the root directory.

Basic Commands for Working with Directories

The following are the basic commands with which you can work when you are dealing with directories

- **pwd : Print Working Directory** The **pwd** command displays the absolute pathname of your working directory and the command line format is

```
$pwd
```

- **mkdir : Create a Directory** You may wish to store files in a particular directory of your own creation. **mkdir** command allows you to create one or more directories. The mkdir has the following command line format

```
$ mdir dirname…
```

The argument ***dirname*** may be either an absolute or relative path name and you may specify more than one directory name on a single command line. If a ***dirname*** already exists, the **mkdir** command aborts and does not overwrite the existing directory.

- **cd : Change Directory** To change to any directory in the file system use the **cd** command (for change directory). The command format is simply

```
$ cd pathname
```

where **pathname** is either an absolute or relative path name to the desired target directory. Unless you do use a full **pathname,** any filename you specify will be taken to mean a file relative to your current working directory.

- **rmdir : Remove a Directory** If you decide you no longer need a directory, you can use the command **rmdir** to remove it. To remove one or more directories use the command line format

```
$ rmdir pathname…
```

The **rmdir** cannot remove a directory unless it is empty. This prevents you from accidentally removing files you wish to keep.

Input/Output in UNIX (Device files)

Like all computers, those running UNIX have I/O devices such as terminals, disks, printers and networks connected to them. Some way is needed to allow programs to access these devices. Although many solutions are available, UNIX integrates them into the file system as special files.

When you log in to the UNIX system, you are assigned a particular terminal from which you control all your processes. This terminal is called your Control Terminal. The UNIX system knows each device by a unique file name; using this file name you can refer to your control terminal or another terminal device.

Each I/O device is assigned a path name usually in ***/dev***. For example, the printer may be **/dev/lp**, terminal 1 may be ***/dev/tty1*** and network may be ***/dev/net***. When you need to refer to your terminal, you may use the **tty** command to find out the file name for your control terminal. These special files can be accessed the same way as any other files. No special commands or system calls are needed. The usual READ and WRITE system calls will do just fine. For example, the command

```
cp file /dev/lp
```

copies the file to printer causing the file to be printed.(assuming this is permitted). An additional advantage is that the usual file-protection rules apply automatically to I/O devices.

There are three major parts to every file in the UNIX file system. They are:

- the inode
- the data blocks
- the directory

The inode

Each file in the file system is described by a structure called an **inode** . To keep track of which blocks belong to which file, one can associate with each file a little table called an **inode (**short form for **index-node),** which lists the attributes and disk addresses of the file's blocks.

inodes are located in special data blocks (not used for file data) and each 512 byte block can contain as many as eight 64 byte inodes. The **inode** contains all data about the file except the file name and the actual data contained in the file. The disk addresses for the file's data blocks are contained in the **inode** area. The inodes are numbered from 2 (reserved for the root directory,/) through 65,535. i-node 1 is reserved for bad block handling. This identifying number is known as the **inode** number or the i-number.

The first few disk addresses are stored in the **inode** itself so that for small files, all the necessary information is right in the i-node. Hence, whenever a small file is opened the information is directly fetched from disk to the main memory. But for somewhat larger files, one of the addresses in the i-node is the address of a disk block called a **single indirect block.** This block contains additional disk addresses. Similarly if this still is not enough, another address in the **inode**, called a **double indirect block,** contains the address of a block that contains a list of single indirect blocks. Each of these single indirect blocks points to a few hundred data blocks. Similarly, one can also have a **triple indirect block,** and UNIX uses this scheme.

The data blocks

The data blocks are located on the disk and contain the actual data of a file. Each block can typically hold 512 characters. Some UNIX implementation use larger block sizes ranging from 1024 characters and up. Even if the file contains only one character, an entire data block must be allocated to hold this single character.

The directory

A directory contains one or more filenames. Each entry in a directory contains one filename and the **inode** number that points to the **inode** for the file. Directories also have an **inode.**

When a file is opened, UNIX uses the path name given by the user to locate the directory entry. The directory entry provides the information needed to find the disk blocks. The directory system's job is to map the name of the file onto the information needed to locate the data.

Figure 2.5(a) shows a UNIX directory entry, 2.5(b) shows the relationship between the data blocks, the **inode** and a directory that references the file and 2.5(c) shows an **inode**.

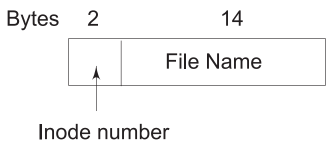


Fig. 2.5(a) *A UNIX directory structure*

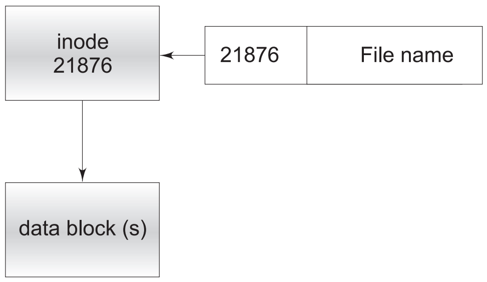


Fig. 2.5(b) *The data blocks, inode and the directory*

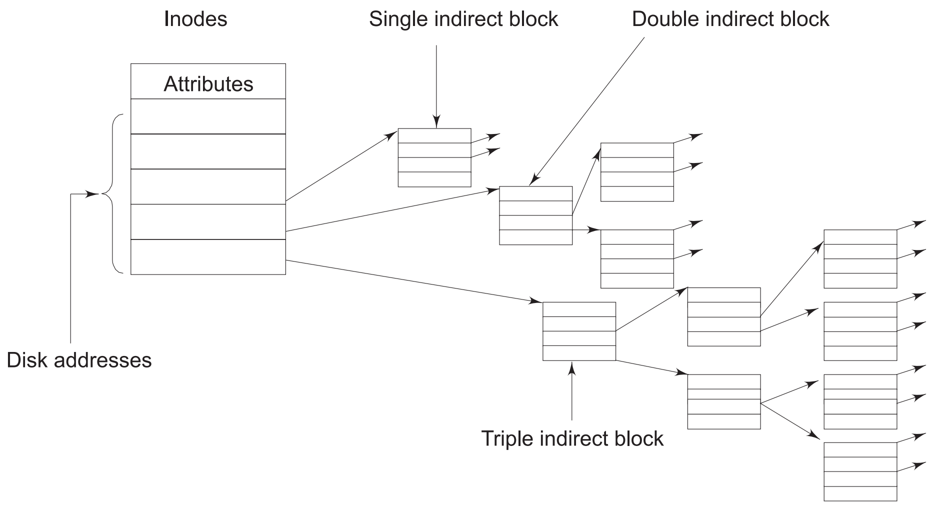


Fig. 2.5(c) *An inode*

2.3 UNIX SHELLS

To use UNIX, you must first log in by typing your name and password which the login program reads and checks. This identification is necessary to provide security as, UNIX keeps track of who owns each file. A file may be used only by authorized users. The UNIX password file contains one line for each user, containing the user's login name, numerical user id, encrypted password, home directory and other information. When a user logs in, the login program encrypts the password just read from the terminal and compares it to the one in the password file. If they agree the login is permitted, if not it is disallowed.

After a successful login the *login* program starts up the command line interpreter specified by the user's *password* file entry and then exits. Most of the command line interpreters is the shell. The **shell** initialises itself and types a prompt character, often a % or $ sign on the screen and waits for the use to type a command line.

When the user types a command line, the **shell** extracts the first word from it, assumes it is the name of the program to be run, searches for this program and, if it finds it, runs the program. The **shell** then suspends itself until the program terminates, at which time, it tries to read the next command. The **shell** is an ordinary user program. All it needs is the ability to read from and write to the terminal, and the power to execute other programs.

Commands may take arguments which are passed to the called program as character strings. For example, the command line

```
$ cp src dest
```

invokes the cp program with two arguments ***src*** and ***dest***

This program interprets the first one to be the name of an existing file. It makes a copy of this file and calls the copy ***dest*.**

A program like the **shell** does not have to open the terminal in order to read from it or write to it. Instead, when it starts up it automatically has access to a file called **standard input**, a file called **standard output** and a file called **standard error**. Normally all three default to the terminal so that reads from the **standard input** come from the keyboard and writes to the **standard output** or **standard error** go to the screen.

2.4 SUMMARY

You have now been introduced to the basic features and issues in the **UNIX** operating system. You may now feel comfortable to try your hands on the UNIX system as a user. Please contact your system administrator and get yourself registered as an authorized user, with a specific login name and password. You will then be better able to appreciate the discussion in Chapter 3.

Review Questions

2.1 Give an account of how UNIX versions came chronologically into existence?
2.2 Explain the main features of UNIX.
2.3 Explain the important constituents of UNIX structure.
2.4 What is a file in UNIX? And what is a file link?
2.5 What are the security rights associated with files and directories?