WILEY-INDIA EDITION

UNIX Programming

THE FIRST DRIVE

Kumar Saurabh

Foreword by :

A S Murty

Satyam Computer Services Ltd

Unix Programming

The First Drive

Unix Programming

The First Drive

Kumar Saurabh

Satyam Computer Services Ltd., Hyderabad.

Wiley India Pvt. Ltd.

Unix Programming
The First Drive

First Indian Edition: 2008

Printed at: *Nice Printing Press, Delhi*

ISBN: 978-81-265-1613-1

ISBN: 978-81-265-8005-7 (ebk)

To my loving wife Anju
and adorable daughter Shivika

Foreword

Unix operating systems and Unix Internals are irrefutably at the core of the important elements of maintaining a small, medium or large network of Unix systems.

Many attempts have been made to bridge the gaps between basic understanding of Unix and the Internals of Unix. Switching from Unix Commands level to programming is one of the hard aspects of enterprise computing today.

This book helps as an easy reference guide for the first time users; at the same time, it excites the specialists in the Unix World. While Unix online documentation has always been user-friendly, books like this serve the purpose of a refresher even in an offline mode.

The scope of Unix operating systems and Unix Internals continues to expand at a rapid rate. As core Unix grows ever more sophisticated, it is increasingly important for developers to have access to a book like this, which simplifies the complicated features and constructs (with examples). This book also helps non-Unix developers to transition to a Unix environment smoothly.

The smooth narration in the entire book reflects in-depth experience of the author on various aspects like Commands, Shell Scripts, Internals, Sockets and Device Drivers.

It facilitates a tutorial approach from Unix commands to Device Drivers comprehensively. The author provides extensive insights into the underlying issues of Unix systems programming, system calls and other design choices to Unix environment. This book would serve the needs of students–faculty community as well as experienced IT professionals

Unix Programming, The First Drive is a definitive and state-of-the-art approach through which the reader would gain a deeper understanding of the core features of Unix.

A S Murty
Head – Leadership Development Group
Satyam Computer Services Ltd., Hyderabad

Acknowledgement

I take this opportunity to express my deep sense of gratitude to my research guide Dr V. B. Gupta, Devi Ahilya University, Indore, India for his enormous help, dynamic guidance and enthusiastic encouragement.

I would like to thank Satyam Computer Services Ltd. especially Mr. B. Ramalinga Raju, Chairman of Satyam Computer Services Ltd., and Mr. A. S. Murthy, Sr. Vice President, Satyam Computer Services Ltd., for motivating and encouraging me to work. I take immense pleasure in offering my acknowledgements to Mr. Rajul Asthana, Sr. Vice President, Satyam Learning Center (SLC), Satyam Computer Services Ltd. Hyderabad; Brig. Satish Bahuguna, Director; and Cmd. Shekhar Murthy, Head, SLC-ELTP team for their valuable guidance that not only acted as a source of inspiration, but also encouraged me from time to time to discuss the problems critically and provided concrete suggestions without which it would have been impossible to bring this work in the present form.

I am very much thankful to Dr. S. Ramesh Babu, Associate Vice President, Infosys Technologies, Bangalore, India for evincing keen interest in my work and continuous encouragement.

I wish to place on records my sincere thanks to my wife, Anju Saxena, for her unconditional support and constant encouragement throughout this book.

No form of acknowledgement can encompass the multitude of contribution that my family members have made. Their support, motivation, affection, and patience made me to work hard and complete this work successfully. Special thanks to my mother and father for their continued love and patience. I cannot forget the continuous support, affection and love provided by my younger brother Anuj Saxena.

I am very much grateful to my lovely daughter Shivika Saxena who in her own way provided me support and comfort.

Last but not the least I thank all the colleagues at Satyam Computers Services Ltd. and many others who directly or indirectly helped me in completing my book.

Hyderabad
March, 2008.

Kumar Saurabh

Preface

Why This Book?

You might be a developer already experienced with programming for the system, and you want to learn about some of its advanced features and capabilities. You might be interested in UNIX Basics, Commands, UNIX shell scripting, writing more sophisticated programs with features such as File and Directory I/O, Process, Signals, Inter-process communication and interaction with hardware devices. You might want to improve your programs by making them run faster, more reliably, and more securely, or by designing them to interact better with the rest of the UNIX system.

You might be a developer making the transition from a non-UNIX environment, such as Microsoft's Win32 platform. You might already be familiar with the general principles of writing good software, but you need to know the specific techniques that UNIX programs use to interact with the system and with each other. And you want to make sure your programs fit naturally into the UNIX system and behave as users expect them to.

Because this is book from UNIX basics to advanced topics, I'll assume that you are already familiar with the C programming language and that you know how to use the standard C library functions in your programs. The C language is the most widely used language for developing UNIX system; most of the commands and libraries that we discuss in this book, and most of the UNIX kernel itself, are written in C.

The information in this book is equally applicable to C++ programs because it is roughly a superset of C. Even if you program in another language, you'll find this information useful because of C language APIs.

Approach

This book addresses the fundamentals of UNIX system, UNIX commands, UNIX shell scripts, internals, sockets and device drivers focuses on the important elements of maintaining a small, medium or large network of UNIX systems. It tells you everything you need to know to be a successful system programmer. Enterprise networks (or clusters) of UNIX systems have become the critical link and key component of the information landscape for corporate world.

UNIX systems have been deployed within every corporate function and within a broad section of businesses and markets. This widespread acceptance and deployment means that UNIX systems are now more on the critical path than ever before. In the financial community as well as other market segments, even a relatively small system failure or outage can result in significant financial impact or have other far-reaching implications.

This book will give you the knowledge of important sections from the scratch, the step-by-step procedures and the skills necessary to effectively UNIX systems programmers. It is meant to be very practical in nature, and focuses on only the more important elements to system programming, not esoteric subjects that have little relevance to the important issues faced by today's UNIX system programmers.

Audience

This book should be an interesting source of information both for people who want to experiment with their computer and for technical programmers who face the need to deal with the inner levels of a UNIX from scratch. We hope this book is useful as a starting point for people who want to become kernel hackers but don't know where to start.

On the technical side, this text should offer a hands-on approach to understanding the UNIX Commands, UNIX shell scripts, UNIX internals and device drivers for some of the design choices made by the UNIX developers from scratch.

Organization of the Book

The first section of the book deals with shell commands. The shell commands are the basic utilities provided by the UNIX shell, which can be viewed as a command interpreter. The book will talk about the default and most commonly used shell is tcsh (`/bin/tcsh`). (Others possible are listed in the file `/etc/shells`.) This part visualizes that shell not only provide the user environment but are also powerful programming languages.

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

Third section works with the UNIX internals that includes file and directory maintenance, process and signals in efficient fashion. File and directory I/O will emphasize on all operations that can be done to files. The process image as viewed by the kernel runs in its own user address space a protected space which can not be distributed by other users that's why it gives the close look to all aspects of process. Signals are software interrupts. This section will put importance how to handle asynchronous events.

Fourth section will depict the picture of inter-process communication. Multiple co-operating processes needs to communicate to perform a related operation. It will depict that processes must communicate in order to share information and resources. Since processes themselves cannot access the address space of other processes the kernel provides IPC (inter-process communication) facility. The kernel is responsible for the data transfer between processes. This part will provide facilities such as PIPE, FIFO, Message Queue, Shared memory, Semaphore.

The fifth section of the book provides the concept of sockets that is the communication API. It is developed for C language to design systems for communication. This part will depict protocol, local address, local process, foreign address and foreign process.

The sixth section of the book will highlight device drivers with the context to kernel programming. It will distinguish character as well as block drivers. It introduces UNIX device drivers and gives trivial explanation of I/O operation from a user's perspective.

The UNIX enthusiast should find in this book enough food for their mind to start playing with the UNIX commands, shell scripts and with the code base from file and directories, process, signals, IPC, device drivers and should be able to join the group of developers that is continuously working on new capabilities and performance enhancements.

Therefore, it makes a good introduction to UNIX commands to device drivers in general. UNIX is still a work in progress, and there's always a place for new programmers to jump into the game. This book is not intended to be a comprehensive guide or reference to all aspects of programming. Instead, I'll take a tutorial approach, introducing the most important concepts and techniques and giving examples of how to use them.

Kumar Saurabh

Contents

1 First Drive

Before our first drive on the UNIX road, let us refresh our memory on some key computing concepts.

Computer system: As you know, a computer system is a combination of hardware and software joined to perform desired tasks. The system's software makes the hardware function. A typical computer system has memory and set of states that define the relationship between the system's inputs and outputs.

Operating system: An operating system (OS) is a program that interacts between the user and the hardware. It may also be termed as a middleman between the user and the hardware. An OS is defined as a set of computer programs that manage the hardware and software resources of a computer. It performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking and managing file systems. Before the evolution of operating systems, one had to write programs to keep track of all the hardware devices, which was a very complex task. Some popular operating systems are UNIX, LINUX (a free version of UNIX), Windows 2000, Windows XP, Windows Vista, etc.

Kernel: Kernel is the central component of most computer operating systems. Its responsibilities include managing the system's resources and processes, and the communication between hardware and software components. The kernel is responsible for creating, scheduling and deleting processes and often for inter-process communication.

UNIX: UNIX is a computer operating system that was designed to be portable, multi-tasking and multi-user in a time-sharing configuration. UNIX has been evolving over the last 30 years. Today many different varieties of UNIX exist. However all UNIX systems have some common characteristics, like they use plain text for storing data, have a hierarchical file system, treat devices and certain types of inter-process communication (IPC) as files, and use a large number of small programs that can be strung together through a command line interpreter using pipes. The UNIX operating system also comprises a master control program, the kernel. More about UNIX kernel in Section 1.4.1.

1.1 Functions of Operating System

1.1.1 Process Management

Every program running on a computer, be it background services or applications, is a **process**. Modern operating systems enable concurrent execution of many processes at once via multi-tasking even with one CPU. Process management is an operating system's way of dealing with running multiple processes. It involves computing and distributing CPU time as well as other resources. Most operating systems allow a process to be assigned a priority which affects its allocation of CPU time.

UNIX is a **time-sharing** system, which means that the processes take turns running. Each turn is a called a **time-slice**; on most systems this is set at much less than one second. The reason this turns-taking approach is used is fairness: We do not want a two-second job to have to wait for a five-hour job to finish, which is what would happen if a long-duration job had the CPU to itself until it completed. UNIX identifies every process by a Process Identification Number (PID) which is assigned when the process is initiated. When we want to perform an operation on a process, we usually refer to it by its pid.

A major advantage of multi-user, multi-process operating systems, like UNIX, is their ability to run many programs simultaneously as well as handling various users, and keep track of both at the same time. Under these circumstances, it is necessary to be able to monitor the system for things such as runaway programs and memory hogs, so that these problems can be dealt with effectively.

A process goes through a series of discrete process states described below:

1. **New state:** The process is being created.
2. **Terminated state:** The process has finished execution.
3. **Blocked (waiting) state:** When a process blocks, it does so because logically it cannot continue, primarily because it is waiting for an input that is not yet available. Formally, a process is said to be blocked if it is waiting for some event to happen (such as an input/output (I/O) completion) before it can proceed. In this state a process is unable to run until some external event happens.
4. **Running state:** A process is said to be running if it currently has the CPU, that is, actually using the CPU at that particular instant.
5. **Ready state:** A process is said to be ready if it uses a CPU if one were available. It is runable but temporarily stopped to let another process run.

Logically, the running and ready states are similar. In both cases the process is willing to run; only in the case of the ready state, there is temporarily no CPU available for it. The blocked state is different from the running and ready states in that the process cannot run even if the CPU is available.

1.1.2 Resource Allocation

The primary task of an operating system is to keep track of who is using which resource, to grant a resource request, to account for usage and to meditate conflicting requests from different programs and users.

When a computer has multiple users, the need for managing and protecting the memory, I/O devices and other resources is even more apparent. This need arises because it is usually necessary to share expensive resources such as tapes drives and phototypesetters.

Modern operating systems often provide users and applications with a virtual machine, that is, an interface to the underlying hardware that makes it appear as though only the user is using the machine and its hardware.

1.1.3 CPU Utilization

Whether the computer has one CPU or several CPUs does not matter, it is usually the case that there are more processes than CPUs. Thus, the operating system is responsible for scheduling the processes on the CPU(s). It performs basic tasks such as controlling and allocating memory, prioritizing system requests, controlling input and output devices, facilitating networking and managing file systems.

1.1.4 Memory Management

Today, applications are getting bigger and bigger, requiring a bigger system memory so that the system could hold the application data, instruction and threads to load it. The system needs to copy the application data from the hard disk (HDD) into the system memory to process and execute the data. Once the memory gets filled up with data, the system will stop loading the program, creating problems for the applications with large data. To run such applications, the users need to add more memory onto their system. However, adding more system memory costs money, and a normal user may need large memory to run intense applications only for one or two days. Therefore, virtual memory is introduced to fulfill this type of temporary memory need. Let us now understand the difference between system memory and virtual memory.

System memory is a memory that is used to store the application data and instructions for the system to process and execute that application. When you install memory sticks to increase the system RAM, you are adding to your system memory. The system memory is also called physical memory or main memory.

Virtual memory is a memory that uses a portion of HDD space as the memory to store the application data and instructions that the system deemed it doesn't need to process for now. Virtual memory, also known as logical memory, is controlled by the operating system, which is UNIX. The virtual memory can be added by changing the system configuration.

In multi-tasking UNIX systems, total memory isolation, otherwise referred to as a discrete address space, can be provided to every task except the lowest level operating system. This greatly increases reliability by isolating program problems within a specific task and allowing unrelated tasks to continue the process.

1.1.5 I/O Devices Handling

Managing input and output is largely a matter of managing **queues** and **buffers** which are special storage facilities. These facilities take a stream of bits from a device, perhaps a keyboard or a serial port, hold those bits, and release them to the CPU at a rate slow enough for the CPU to cope with. This function is especially important when a number of processes are running and taking up processor time. The operating system will instruct a buffer to continue taking input from a device, but stop sending data to the CPU while the process using the input is suspended. And, when the process needing input is made active once again, the operating system will command the buffer to send the input data. This process allows a keyboard or modem to deal with external users or computers at a high speed even though there are times when the CPU can't use input from those sources.

Managing all the resources of the computer system is a large part of the operating system's function and, in the case of real-time operating systems, it may virtually be the functionality required. However, for other operating systems, providing a relatively simple, consistent way for applications and humans to use the power of the hardware is a crucial part of their job.

Several processes may be trying to access a single I/O device, and the operating system must manage these accesses. However, this is a different issue than process scheduling because often I/O are being performed for processes that are not currently executing.

Some devices (e.g. disks) have resources that can be shared among users and/or user processes. The operating system is responsible for managing and protecting these resources.

1.2 Operating System Design Principles

Designing an operating system is a complex task. In fact, the complexity of OS design emerged as a major driving force behind software engineering development.

1.2.1 OS Design Goals

An OS design should fulfill the following goals:

1. **User interface**

   The user interface is a combination of menus, screen design, keyboard commands, command language and online help, which creates the way a user interacts with a computer. If input devices other than a keyboard and mouse are required, this is also included. In the future, natural language recognition and voice recognition will become standard components of the user interface.

   A program interface, which takes advantage of the computer's graphics capabilities, is easier to use. Well-designed graphical user interfaces can free the user from learning complex command codes. On the other hand, many users find that they work more effectively with a command-driven interface, especially if they already know the command language.

2. **Efficient system resource management**

   The OS must provide two main functions: First, it must manage the resources available to the computer system. Second, it must provide a reliable, stable, secure and consistent interface for applications to access the computer's resources.

   The first function is critical to the OS because it defines how applications access the system's resources. By controlling the various aspects of how hardware and software are used, the OS ensures that every application gets a chance to use the processor. The second, related function defines the methods by which an application can access these resources. Because the OS often acts as a buffer between an executing program and the hardware, it needs to provide some means of allowing applications to access resources without needing to know the details of each and every unique computer system.

3. **Security**

   While operating systems vary on many levels, the most common operating systems provide much more than a simple interface between the user and the machine. Usually included in the OS are programs that provide the user with numerous extras, from simple screen savers to complex file-encryption schemes. However, it is important to understand that these programs are extras that are added to the OS and are not necessary for the computer to operate.

   Many users become intimately familiar with the operating system's accessories, but forget about the security features that are included to help the user maintain a safe and reliable operating environment. As a result, many information systems exist in an insecure state that leaves the system at risk to a virus infection or a complete compromise by an attacker.

4. **Flexibility**

   Most operating systems come preconditioned to support many devices. Part of the process of setting up a particular machine is to construct an OS version that is tuned for the local installation. This tuning often involves setting certain limits such as the maximum number of processes for the concurrent situation. It also involves specifying the attached hardware so that only the necessary drivers will be loaded. Some operating systems can load and unload drivers automatically at run-time. This gives ease to the user not to think about the inner details of system programming regarding how the drivers are actually working internally.

5. **Portability**

   As the system is written in high-level language, it is easier to read, understand, change and move on to other machines. To port the OS some part of the kernel has to be rewritten in assembly language of the new machine, and the rest is obtained by just compiling the high-level code. This makes porting from one platform to another in UNIX easy.

1.2.2 Layered Design

An OS consists of multiple layers (Figure 1.1). Each layer depends on the layer(s) beneath it. UNIX is a layered operating system. The innermost layer is the hardware that provides the services of device definitions for the OS. For example, the operating system, referred to in UNIX as the kernel, interacts directly with the hardware and provides the services (scheduling and synchronizations) to the user programs. These user programs don't need to know anything about the hardware. They just need to know how to interact with the kernel which will provide the desired service. A big appeal of UNIX to programmers comes from the fact that most of its well-written user programs are independent of the underlying hardware, making them readily portable to new systems.

The user programs interact with the kernel through a set of standard system calls. These system calls request for services to be provided by the kernel. Such services would include accessing a file: open, close, read, write, link or execute a file; starting or updating accounting records; changing ownership of a file or directory; changing to a new directory; creating, suspending or killing a process; enabling access to hardware devices; and setting limits on system resources. As UNIX is a multi-user, multi-tasking operating system, we can have many users logged into a system simultaneously, each running many programs.

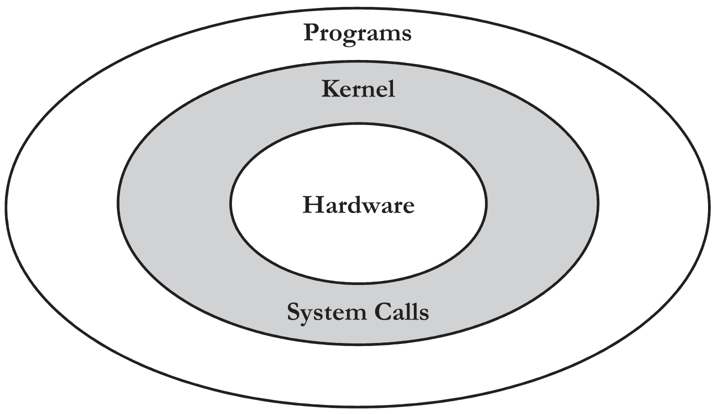


Figure 1.1 Layered design of operating system.

1.2.3 Virtual Machines

The concept of virtual machines is closely related to layering. In a typical multi-user system, a user is expected to know that the machine is shared by other users who also share resources such as devices.

In virtual machine operating systems, an additional layer of abstraction is placed between the users and the system so that it appears to the users that they are using a machine dedicated to them.

Usually, in this case a more powerful machine is used to host several virtual machines. For example, the 80386 and later Intel CPUs supported virtual 8086 machines. Thus, an OS designed for the 80386 CPU could actually run several copies of MS-DOS and appear to the user as different PCs at the same time.

Another example of a virtual machine system is the IBM 370 running the VM operating system. This system allowed users to work with a dedicated (but smaller, less powerful) 370 completely at their disposal.

1.3 History of UNIX

As already stated, UNIX is a multi-user operating system, that is an operating system that can handle more than one user at a time. It was developed by Ken Thompson at Bell Labs in 1969. UNIX was formed from a time-sharing system called Multiplexed Information and Computing Service (MULTICS), and was first loaded onto PDP-7 which was a minicomputer from Digital Equipment Corporation (DEC) computers with less processing power. Among the early users of the system was Dennis Ritchie who helped to move the system to the PDP-11. The main breakthrough came in 1973 when Ritchie and Thompson rewrote the UNIX kernel in C language thus breaking the tradition of writing the system software in a low-level language.

1.4 Structure of UNIX

The components of a UNIX system are

1. Kernel.
2. Shell.
3. Commands and utilities.
4. Application software.

1.4.1 Kernel

The kernel is the hub of the UNIX operating system. It is the master control program. The kernel provides services to start and stop programs, allocates time and memory to programs, handles the file system and other common high-level tasks that most programs share. It schedules access to hardware to avoid conflicts if two programs try to access the same resource or device simultaneously. The kernel controls computer's resources. When one user logs into the system, the kernel checks whether he is an authorized user. It also handles the shell scripts, the storage process and the information transfer between the user and the resources. There is only one kernel per system.

1.4.1.1 Time Sharing and Processes

One important duty of the kernel is to run the time-sharing system. As UNIX is a multi-user, multi-task system, more than one user can use the computer at the same time and more than one task can be given to the system by the user. Each task or program in UNIX is termed as process. UNIX uses time-sharing to fulfill multiple demands. The kernel maintains a list of current tasks (processes) and allots a bit of time to one process and to the next, and so on. But the kernel switches so fast between processes that the user thinks only his process is being executed by the system.

1.4.2 Shell

When a user logs in, the login program checks the username and password, and then starts another program called shell. Shell is the command line interpreter (CLI). It interprets the commands the user types in and arranges their implementation. In other words, the shell shuttles the command or request keyed in by the user to the kernel and then the result from the kernel to the user. The shell begins to operate as soon as the user logs in to the system. It is the shell, which hides the kernel from the user by covering it. The shell acts as an interface between the user and the kernel. The commands are themselves programs: When they terminate, the shell gives the user another prompt (% on our systems).

1.4.2.1 Major Features of Shell

1. Interactive processing.
2. Background processing.
3. I/O redirection.
4. Pipes.
5. Shell scripts.
6. Shell variables.

1.5 Features of UNIX

UNIX is an operating system with lot of features. Some major features are described below:

1. Multi-user, time-sharing OS.
2. File system.
3. Shell.
4. Pipes and filters.
5. Programming facility.
6. Background processing.
7. Communication.
8. Security.

1. Multi-user, time-sharing OS

As already stated, in UNIX more than one user can use the computer at the same time and more than one task can be given to the system by a single user. The users are signed on to the system through terminals attached to a host computer. For providing the multi-user atmosphere, UNIX uses the time-sharing method. Here the user's programs are lined up and are executed for a fixed amount of time and the user's status is saved, and then another user's request is undertaken, and so on.

2. File system

Everything in a UNIX system is a file. UNIX has a hierarchical file structure that allows the files to grow dynamically. Files are assigned into various directories for easy access. The UNIX file structure also allows implementation of the file security system.

3. Shell

As discussed earlier, the shell is the command interpreter, which interacts with the kernel. The shell also has the programming facility known as shell programming, which has all the ingredients for programming like control structures, loops, variables, etc.

4. Pipes and filters

In UNIX, the output of one command can be the input of another command. Pipes are used to join the two commands. The pipe feature helps in avoiding the complex programming. Filters are programs that perform simple transformation on data as it flows through. Examples of filters are *`sort, grep,`* etc. which will be discussed in detail later.

5. Programming facility

UNIX was actually designed for programming. A programmer can exploit the full potential of UNIX. The UNIX shell has all the facilities that help in programming, such as variables (user-defined, global), control structures, loops, etc. you can find in a programming language.

6. Background processing

The background processing features allow UNIX to carry on a process in the foreground while keeping another waiting in the background. UNIX gives the background process an identification number. This feature helps the user to implement two processes without wasting much time.

7. Communication

UNIX provides the electronic mail facility for communication among its users. This feature is a must for a multi-user system like UNIX, because the users, logged into the system, may be some distance apart and may need a communication link to pass some messages. The commands that help for communication are `write` and `mail`.

8. Security

UNIX can be used by only those who maintain an account with the computer system. To work on the system the user must have a password (optional) and a user name. When the user switches on the system, he will be asked to enter his username/login name. The user will be allowed to use the system only if both are correct.

Login names are always entered using lower case letters. The UNIX system will never indicate what went wrong with a login name or password when an error occurs, but will give an option for retry. The security feature helps in keeping the unauthorized users away from the system.

1.6 UNIX File System

The UNIX file system (files and directories) is organized in a hierarchical fashion that is similar to the way many corporations are organized (technically designated as a tree structure as it looks like an upside down tree). Similar files are clustered into a directory. Directories can have sub-directories, etc. This file system is similar to the concept of folders and sub-folders that is well understood by the users of Windows and Macintosh systems.

A key characteristic of UNIX file systems is that virtually everything is defined as being a file – regular text files are of course files, but so are executable programs, and directories; even hardware devices are mapped to file names. This provides tremendous flexibility to programmers and users, even if there is a bit of a learning curve at first.

UNIX file systems are also probably the most different from the other file systems used on PC, both internally and externally (referring to how the user accesses the file system). While most Windows users are accustomed to the explorer-type interface for managing files and folders, UNIX files are usually managed with discrete text commands – similar to how DOS works. (In fact, many principles of the FAT file system are based on UNIX.) There are graphical UNIX shells as well.

The more important differences, however, are internal. UNIX file systems are designed not for easy use, but for robustness, security and flexibility. These systems offer the following features:

1. Excellent expandability, and support for large storage devices.
2. Directory-level and file-level security and access controls, including the ability to control which users or groups of users can read, write or execute a file.
3. Very good performance and efficient operation.
4. The ability to create flexible file systems containing many different devices, to combine devices and present them as a single file system, or to remotely mount other storage devices for local use.
5. Facilities for effectively dealing with many users and programs in a multi-tasking environment, while requiring a minimum of administration.
6. Ways to create special constructs such as logically linked files.

1.6.1. File Protection

The UNIX file system allows you to control which users have access to your files and directories and what access modes are permitted. Although the file protection mechanism is fairly simple, it serves quite well under most circumstances. The protection mechanism is the same for files as it is for directories, therefore for this discussion the term 'object' refers to either a file or a directory.

Every UNIX file was created by an owner. An owner may belong to a certain group of people the file might be associated with (people working on one experiment, for example). And there is also the general world-wide web audience. Those are the three classifications UNIX gives to its users. Based on those classifications, UNIX protects its files.

The basis of UNIX protection is the permissions one may attach to a file. Once a file is created, there are three things an owner can give a permission to do with his files:

1. Permission to read.
2. Permission to write (change).
3. Permission to execute (to run the file).

Each of these three permissions is applied to each one of the groups described above: owner, group and www users.

1.6.2 File Categories

Figure 1.2 depicts the different categories of file available in UNIX.

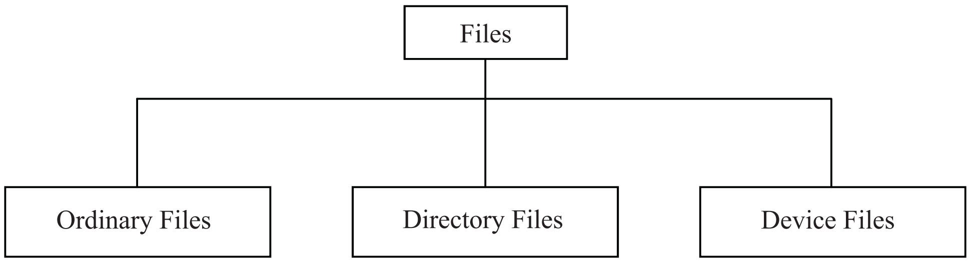


Figure 1.2 File categories.

1.6.1.1 Basic Files Distinction

There are three types of file in UNIX – Ordinary files, Directory files and Device files. Figure 1.2 shows the UNIX file categorization.

Ordinary files: These files represent a stream of data resident on some magnetic media. These include all data, source programs, object codes or executable codes and all UNIX programs as well as files created by the user. All text files also fall in this category.

Directory files: Every file is assigned to a directory. A directory is a specialized form of file that maintains the file system based on the specific use of that directory. An important thing about a directory file is that it can be a directory itself. This quality makes it possible to develop a tree-like structure for *f* directories and files. For example, one can create one directory for C programs, one for correspondence, etc. These directories can be further subdivided. We will discuss the commands with which you can create, change or delete directories in detail in a later chapter. A directory contains two fields: name of the file and a pointer to separate disk area where details about the file are stored.

Device files: These file include printers, tapes, floppy disks, hard disks and terminals. Any output redirected to the device file will be reflected onto the respective I/O device associated with the file name.

1.6.3 Structure of UNIX File System

As stated earlier UNIX has a hierarchical structure, and files can be stored under directories. All files in UNIX are related to each other, that is, the file system is a collection of related files. UNIX makes use of an inverted branching-tree file structure. The tree trunk consists of a single large file, '/' (called root), which contains all the files on the system. (In this structure, all files which contain other files are called 'directories'). Any number of subdirectories may branch from this main trunk. However, the hierarchical structure does not end there, for each major branch any number of further branches (sub-subdirectories) may spawn. These subdirectories may, in their turn, send forth yet more levels of files, and so on.

In most UNIX-based systems, a directory is created under the root directory. This directory is called 'usr'. Each user is given a directory to store private files and create subdirectories. To access files, pathnames are used. Pathnames are a sequence of directory names separated by a /. Pathnames are of two types: absolute pathnames and relative pathnames.

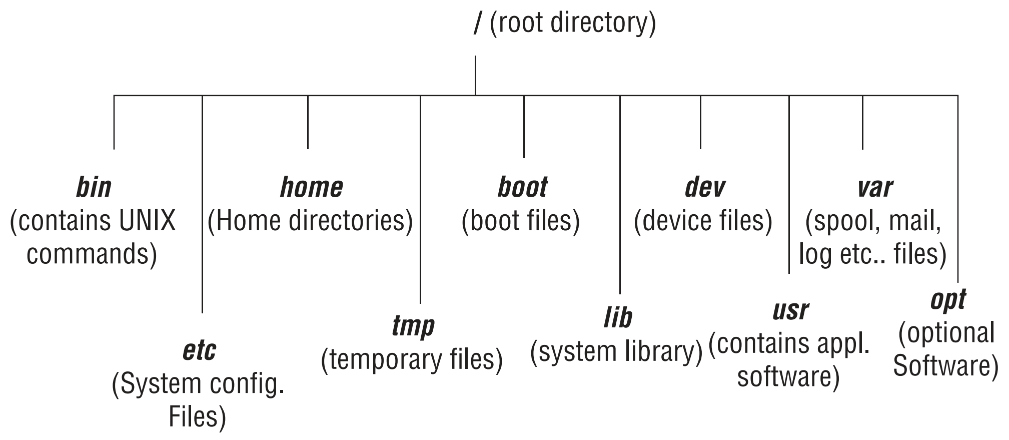


Figure 1.3 UNIX file system.

Absolute pathname: It is the complete path that UNIX must follow to reach a particular file, and in this name file location is determined with respect to the root directory.

Relative pathname: If the file location is determined with respect to the current directory, then it is a relative pathname.

Figure 1.3 shows the file system for UNIX operating system. It is clear that all the directories and files are maintained under the root (/). Functionality-wise use of all the directory files is explained below:

1. **dev:** This directory contains all the information about hardware.
2. **etc:** This directory contains the information related to the machine. The system and password files related to a machine are stored here.
3. **tmp:** This directory contains the files that are in open or in use, but it never stores any file.
4. **usr:** This directory stores the directories and files created by the user. The information in this directory varies from machine to machine.
5. **bin:** It contains the binary files, that is executables.
6. **lib:** It contains library function. The programmers working in UNIX will use these libraries.

1.7 Internal Structure of File System

The internal structure of the UNIX file system is very complex. Every file has a table associated with it, which is stored in a special area of a disk (memory). This special area is called the identification node (i-node). Every file system consists of a sequence of blocks. Each block consists of 512 bytes. Some blocks are reserved for the use of the kernel. The file system breaks the disk into the following four segments:

1. Boot block.
2. Super block.
3. I-node blocks.
4. Data blocks.

1.7.1 Boot Block

This is the first block, numbered zero (0). It is normally not used by the file system and is set aside for the booting procedure.

1.7.2 Super Block

This is the second block, numbered one (1). This block controls the allocation of disk blocks. It contains details of the active file system, the size and status of the file system, and details of free blocks and i-nodes. The super block needs to be updated in a periodic manner. Unlike MS DOS, the UNIX system must be powered down through a formal routine because the file system information needs to be written to the disk before power is turned off.

1.7.3 I-node Blocks

This is third block, numbered two (2) and goes up to a number determined during the creation of the file system. An i-node block contains nearly all the essential information about a file. This information includes the location of the disk where the file is stored, size of the file when the file was last modified, etc. The complete list of i-nodes is known as i-list. Every i-node is identified by a unique number, known as the i-number, which indicates the position of the i-node in the i-list.

An i-node is 64 bytes long, and one physical block contains eight i-nodes. Each i-node contains following attributes of a file:

1. File type (regular, directory, etc.).
2. Number of links (the number of aliases the file has).
3. Owner (the user-id number of the owner).
4. Group (group-id number).
5. File mode (the triad of the three permissions).
6. Number of bytes in the file.
7. Data and time of last modification.
8. Date and time of last access.
9. Data and time of the last change of the i-node.
10. An array of 13 pointers to the file.

It is important to know that the name of the file is not stored in the i-node. When we change the contents of the file, its i-node also gets updated. However, it is possible to alter the i-node contents without changing the file contents.

1.7.4 Data Blocks

The final disk segment contains a long chain of blocks for storing the contents of files. The UNIX file system stores data in physical blocks of 512 bytes. These data blocks commence at the point where i-node blocks terminate. A UNIX file is a sequentially organized set of blocks scattered throughout the disk. It is an array of 13 disk block addresses, which keep track of all disk blocks containing file segments. The first 10 pointers are direct addresses of a file. If a file has only five blocks, then the first five pointers will contain the list of five disk block addresses and the remaining eight pointers will contain zeros.

1.8 Summary

This chapter gives the first drive to learn the basics of the UNIX operating system. It is visualized that each UNIX-like system has a kernel. This program controls the computer hardware. All other programs on the system are part of 'userland', which means they lie outside the kernel. The kernel shares the system between all running 'userland' programs. It is also highlighted that UNIX is hierarchical file system with the root at the level zero. The kernel allows multiple programs to share the hardware safely. It also switches programs in and out of the processors; thus it is a 'multi-tasking' kernel. The shell is a program unique to UNIX-like systems. It lets you type commands to launch other programs.

2
UNIX Commands

When a user logs into a UNIX system, the user is automatically placed in his home directory. In normal case, the directory name and the login name will be the same. The user can navigate into other directories from his home directory.

2.1 Internal and External Commands

As you know, the UNIX shell is the command line interpreter (CLI). It interprets the commands the user types in and arranges their implementation. The UNIX commands are of two types – internal commands and external commands. The internal commands, like *`cd`* and *`pwd`*, are built into the shell. An internal command is executed by the shell itself rather than launching a separate process to implement the command. That is, the shell interprets an internal command and changes your current directory for you. On the other hand, an external command like `ls` is stored in the file `/bin/ls.` For any external command, there will be an entry in `/bin` or `/usr/bin` on most UNIX systems.

The earliest UNIX systems had few internal commands, and this resulted in a very small shell but very slow execution of commands.

Let us understand in detail the command execution by the shell. As already stated, the shell doesn't start a separate process to run internal commands. But external commands require the shell to `fork` and `exec` a new sub-process; this takes some time, especially on a busy system.

When you type a command name, the shell first checks whether it is a built-in command, and executes a built-in command on its own. If the command name is an absolute pathname beginning with /, like `/bin/ls`, the command is executed accordingly without any problem. If the given command is neither built-in nor specified with an absolute pathname, the shell looks in its search path for an executable program or script with the given name.

Just to refresh your memory, the search path is exactly what its name implies: a list of directories that the shell should look through for an executable program whose name matches what is typed.

Traditionaly, UNIX system programs are kept in directories called `/bin` and `/usr/bin`, with additional programs, usually used only by system administrators, lying in `/etc` and `/usr/etc`. Many UNIX versions also have programs stored in `/usr/ucb` (named after the University of California at Berkeley, where many UNIX programs were written). There may be other directories containing programs.

Command structure: In UNIX commands, the first word that the user enters after the $ prompt is the command and subsequent words are called arguments. For example, in `$ls -l note`, `ls` is the command and `-l` is the option and `note` is called the argument.

Points to remember:

1. Commands and arguments have to be separated by spaces or tabs. There can be any number of spaces or tabs.
2. Arguments can have options, expressions, instructions, filenames, etc.
3. Options have to be preceded by a minus sign (—).
4. Options are used to change the default behaviour of a command.
5. Options can be combined. For example, combining the options

```
$ls -l -a is equivalent to $ls –la
```

Command line: The UNIX commands are entered with their arguments (if any). A command usually ends with a new line, that is pressing the RETURN key is necessary for terminating a command. A semicolon is also a command terminator, but even after using the semicolon as command terminator nothing will happen until the user presses the RETURN key. Also take care of the following two tips while working with UNIX commands:

1. Backspacing doesn't work if we want to remove some mistake in the command line. Here Ctrl-h or Del (Delete) should be used.
2. If after entering a command there is no action, it means the command is waiting for you to enter something more. In such cases, enter Ctrl-d bring back the prompt.

In UNIX commands some special characters are used. These have been described below:

`	Backquote or backtick
~	Tilde
#	Pound or number
^	Caret or hat
_	Underscore
\	Backslash
|	Pipe
<	Left chevron
>	Right chevron

You may be eager to work with the UNIX commands. To enable you handle these commands with ease, given below are some key shortcuts. Use them quite often to add speed to your work.

Ctrl-j $ Ctrl-m	Works as alternative to Enter
Ctrl-h	Used to erase text if backspace key doesn't work
Ctrl-c	Used to interrupt a command
Ctrl-u	Used to kill command line without executing it
Ctrl-\	Used to kill a running command. However, a core file containing the memory image of the program is created
Ctrl-s	Used to stop scrolling of screen output
Ctrl-q	Used to resume scrolling of screen output
Ctrl-d	Used to terminate login session or a program that expects its input from the keyboard
exit	Used to terminate login session

logout	Used to terminate login session in the C shell
Ctrl-z	Used to suspend process and return shell prompt
Sty sane	Used to restore terminal to normal status

Before discussing in detail various UNIX commands, let us have a look at Table 2.1 that groups these commands in suitable categories for a quick reference.

Table 2.1 Categorization of various UNIX commands

Command group	*Individual command*	*Command function*
A. Directory commands	1. `mkdir`	Making a directory
	2. `pwd`	Identifying current working directory
	3. `rmdir`	Removing a directory
	4. `cd`	Changing current working directory
	5. `ls`	Listing directory contents
	6. `mv`	Renaming files
	7. `find`	Searching the file System
B. File commands	1. `cp`	Copying a file
	2. `rm`	Deleting files
	3. `sort`	Sorting a file
	4. `cat`	Displaying the contents of a file
	5. `pg`	Displaying the contents of a file page by page
	6. `lp`	Printing a file
	7. `file`	Displaying the file type
	8. `more`	Displaying the contents of a file page by page
	9. `tr`	Manipulating individual characters in a character stream
	10. `cpio`	Copying the file to and from archive
	11. `tar`	Saving and restoring file to and from the archive
C. Protection and security commands	1. `chmod`	Changing the file access mode
	2. `chown`	Changing the file owner
	3. `chgrp`	Changing the user group
	4. `passwd`	Changing the password
D. Communication	1. `write`	Sending message to commands the other logged-in user's terminal
	2. `mail`	Sending message to a user who is not logged-in

(*Cont.*)

Table 2.1 (*Cont.*)

Command group	*Individual command*	*Command function*
	3. `mesg`	Receiving or avoiding other user's messages
	4. `wall`	Messaging all logged-in users in a system
E. Information commands	1. `man`	Getting online help
	2. `who`	Displaying the names of all the logged-in users
	3. `who am i`	Displaying the login details of the users who gives this command
	4. `date`	Displaying the date, month, time and year
	5. `cal`	Displaying the calendar
	6. `tty`	Displaying information about the terminal type
	7. `stty`	Used to configure the current terminal
F. Process management commands	1. `at`	Scheduling, checking and cancelling a job
	2. `ps`	Knowing the process status
	3. `kill`	Aborting a process
	4. `wait`	Keeping a process on hold
	5. `sleep`	Suspending a process execution for a specified interval-based time
	6. `nice`	Altering the scheduled priority of processes
	7. `batch`	Deciding on the execution of a scheduled job
G. Pipes and filters commands	1. `tail`	Displaying the last few lines of a file
	2. `head`	Displaying the first few lines of a file
	3. `uniq`	Eliminating or counting duplicate lines in a presorted file
	4. `grep`	Searching a pattern in a list of files
	5. `cut`	Displaying specified columns to the standard output file
	6. `paste`	Combining two files horizontally
	7. `join`	Merging lines of two sorted text files on the basis of a common field
	8. `comm`	Comparing two files for similarities and differences line by line
	9. `cmp`	Comparing two files
	10. `diff`	Finding differences only between two files
	11. `tee`	Redirecting output to multiple files

2.2 UNIX Access Through Login and Password

To access a UNIX system, the users are given login and password, which are created by the system administrator. A password is not visible while typing. After three successive failures of login, that particular login will be disabled. If the login or password are not entered correctly, the system will give 'login incorrect' message. It won't tell which is incorrect – login or password. This is a security measure.

2.3 Directory Commands

Explained below are some directory commands:

2.3.1 `mkdir` (Making a Directory)

With the help of the `mkdir` (Make Directory) command, the user can create a new directory. The command is followed by the name of the directories to be created. It takes multiple arguments. The user should have write permissions on the current directory to create the new subdirectory. The following make directory command creates the directory dir1:

```
$mkdir dir1
```

2.3.2 `rmdir` (Removing a Directory)

With the help of the `rmdir` (Remove Directory) command, the user can remove a directory. This command can delete more than one directory at a time. This command deletes only empty directory. A subdirectory cannot be removed unless it is placed in a directory that is hierarchically above the one that is chosen to remove. The following command will remove the directory named dir1 in the current directory.

```
$rmdir dir1
```

2.3.3 `pwd` (Identifying Current Working Directory)

The `pwd` command is used to identify what is our current working directory. For instance, the following `pwd` command shows that the user's current working directory is `/usr/dir1`.

```
$pwd <enter>
/usr/dir1
$
```

2.3.4 `cd` (Changing Current Working Directory)

With the help of the `cd` (Change Directory) command, the user can move around the directories. The `cd` command can take relative and absolute pathname as arguments as shown below:

```
$cd /usr/dir1/progs (Using absolute pathname)
$cd progs (Using relative pathname)
```

Both the above commands will take the user to the directory progs, which is under directory dir1.

Let us understand the use of the `cd ..` command. As shown below, we have issued the `pwd` command. It reflects that our current working directory is progs under directory dir1.

```
$pwd <enter>
/usr/dir1/progs
$cd ..
```

`cd` with .. will take the user to the parent directory of the current working directory. Whenever there is the requirement to move from the current directory to the parent directory, we will use the same `cd ..` command with one space. If we issue the `pwd` command as shown below:

```
$pwd <enter>
/user/kumar
```

it reflects that our current working directory is `/user/kumar.`

2.3.5 `ls` (Listing Directory Contents)

The `ls` command is used to list the names of files and sub-directories in the current directory. The command

```
$ls <enter>
```

will display the names of the files and sub-directories in the current working directory.

Table 2.2 lists various options available with the `ls` command. The options follow the command name on the command line, and are usually made up of an initial minus sign (−) and a single letter meant to suggest the meaning. For example,

```
$ls -l
```

will give the directory listing with all the information. Note that there is a space between the command and the minus sign.

2.3.6 `mv` (Renaming Files)

The `mv` (move) command is used to change the name of a file or directory. With the help of this command, we can also move a file from one directory to another. The following command will rename the file chap 01 to chap 02.

```
$mv chap01 chap02
```

Table 2.2 The options of the `ls` command

Option	*Description*
−l	This option is used for a long list.
−a	This option lists hidden files also.
−d	This option is used to list only directories.
−r	This option is used to get output in the reverse order.
−R	This option is used for recursive listing of all files in subdirectories.

2.3.7 `find` (Searching the File System)

The `find` command is very powerful. It can search the entire file system for one or more files that you specify to look for. This is very helpful in locating a 'lost' file. You can also use the `find` command to locate a desired file and then perform some action on it using any UNIX command. For example, the command

```
$find /-name Chapter1 -type f -print
```

searches through the root file system ('/') for the file named 'Chapter 1'. If it finds the file, it prints the location to the screen.

Another command

```
$find /usr -name Chapter1 -type f -print
```

searches through the `/usr` directory for the file named 'Chapter 1'.

Here is another command

```
$find /usr -name "Chapter*" -type f -print
```

This command searches through the `/usr` directory for all the files that begin with the word 'Chapter' but may end with any other combination of characters. This search will match filenames such as 'Chapter', 'Chapter1', 'Chapter1.bad', 'Chapter_in_life'.

2.4 File Commands

2.4.1 `cp` (Copying a File)

The `cp` command is used to copy a file. For example, the command

```
$cp chap01 unit1
```

will copy the contents of file chap01 into unit1. If unit1 already has some contents then these will be over-written by the contents of chap01.

2.4.2 `rm` (Deleting Files)

The `rm` command is used to delete files. For example, the command

```
$rm unit1
```

will remove unit1 from the current working directory. If the file to be deleted is not present in the current working directory then its full path has to be specified. For example, the command

```
$rm /usr/dir1/progs/unit1
```

indicates that unit1 is in progs under directory dir1.

To remove all files from the current working directory, the following command has to be given:

```
$rm*
```

2.4.3 `sort` (Sorting a File)

The UNIX `sort` command sorts the input of a file in alphabetical order line by line. If no arguments are given then the `sort` command waits for the user to enter the data from the standard input device. By default, the sorting order puts blanks first, then upper case letters, and then lower case letters. The following `sort` command

```
$sort shortlist
```

will arrange records in the following order – soft space, numerals, uppercase letters and lowercase letters. Table 2.3 lists the options available with the sort command applied on the shortlist file.

2.4.4 `cat` (Displaying the Contents of a File)

The `cat` (*concatenate*) command in UNIX is used to display the contents of a file. For example, the command

```
$cat dept.lst
```

will display the contents of the file `dept.lst.` In this command, it is assumed that the file `dept.lst` is in the current working directory.

To display the contents of a file in another directory, pathname can be used. The `cat` command can also be used to display the information of more than one file. For example, the command

```
$cat chap01 chap02
```

Table 2.3 The options of the **`sort`** command applied on the shortlist files

Option	*Description*
$sort -b shortlist	This option ignores leading blanks in a field of the file *shortlist*.
$sort -f shortlist	This option treats uppercase and lowercase with equal priority.
$sort -n shortlist	This option sorts numeric values of the file *shortlist*.
$sort -d shortlist	This option ignores all characters except letters, digits and blanks in comparison.
$sort -r shortlist	This option sorts the file *shortlist* in reverse order.
$sort -u shortlist	This option ignores duplicate fields while sorting the *shortlist* file.
$sort -c shortlist	This option checks whether the file *shortlist* has actually been sorted.
$sort +1 shortlist	This option starts sorting after skipping the first field.
$sort -m shortlist1 shortlist2	This option merges the files *shortlist1* and *shortlist2* but both the files should be in the sorted order with the same set of options.

will display the contents of the files `chap01` and `chap02`

The `cat` command can also create files. Let us understand how files are created using `cat`. Have a look at the following code that is used to create the file abc:

```
$cat> abc

This is the first UNIX Session

<press ctrl-d>
```

In the above code, first we write `cat > abc` at the command prompt. This creates the file with the name abc. After this, we write the text *This is the first UNIX Session* in the file abc. To save abc in the memory, we press Ctrl-d. This creates the file abc in the current directory.

Now let us check the contents of the abc file using the `cat` command. When we write the following command, it will show the same content that we entered in the abc file previously.

```
$cat abc <enter>

This is the first UNIX Session.
```

2.4.5 pg (Displaying the Contents of File Page by Page)

The navigation through the file is very easy using the `pg` command. With the help of this command, the contents of the file on the screen can be seen, one page at a time. To quit from the page at a point, we will have to press q.

The following is the format of using `pg` command with the filename:

```
$pg filename
```

The `pg` command has various options or arguments that have been described in Table 2.4. Table 2.5 shows the navigation and other file operations of the `pg` command.

Now let us use the pg command to display the contents of a file. The following command displays the contents of the `myfile.txt` file.

```
$pg myfile.txt
```

2.4.6 `lp` (Printing a File)

The `lp` command sends the user's print job to the print queue. For example, the following command will print the two files – file1 and file2 – one by one:

```
$lp file1 file2
```

The `lp` command can take the input from the terminal.

Many options available for different ways of printing related to the `lp` command have been described in Table 2.6.

Table 2.4 The options of the **pg** command

Option	*Description*
-number	The number option is an integer specifying the size (in lines) of the window that a particular pg is to use instead of the default size. (On a terminal containing 24 lines, the default window size is 23.)
-p string	`pg` uses string as the prompt. If the prompt string contains a %d, the first occurrence of %d in the prompt will be replaced by the current page number when the prompt is issued. The default prompt string is ':'.
-c	The -c option brings the cursor home and clears the screen before displaying each page. This option is ignored if clear_screen is not defined.
-e	The -e option prevents the pg from pausing at the end of each file.
-f	Normally, `pg` splits lines longer than the screen width, but some sequences of characters in the text being displayed (for instance, escape sequences for underlining) generate undesirable results. The -f option inhibits `pg` from splitting lines.
-n	Normally, commands must be terminated by a <new line> character. This option causes an automatic end of a command as soon as a command letter is entered.
-r	Restricted mode. The shell escape is disallowed using the -r option. `pg` prints an error message but does not exit.
-s	The -s option allows `pg` to print all messages and prompts in the standard output mode (usually inverse video).
+linenumber	This option is used to start up at line number.
|/pattern/	This option is used to start up at the first line containing the regular expression pattern.
filename	This option gives a path name of a text file to be displayed. If no filename is given, or if it is -, the standard input is read.

Now we will use the `lp` command to print a file. The following command prints the file `myfile.txt`:

```
$lp myfile.txt
```

2.4.7 `file` (Displaying the File Type)

This command is used to know the type of a particular file. For example, the command

```
$file star.lst
```

will show what type of the file, `star.lst`, is placed in the current working directory. It will indicate that type of `star.lst` file is English text.

You can also find out the type of all the files in a directory. For example, to know the type of all the files in directory `progs` use the following command:

```
$file progs/*
```

Table 2.5 The navigation and other file operations executed by the **pg** command

Option	*Description*
h	Help
q or Q	Quit
<blank> or <newline>	Next page
l	Next line
d or <^D>	Display half a page more
. or <^L>	Redisplay current page
f	Skip the next page forward
n	Next file
p	Previous file
$	Last page
w or z	Set window size and display next page
s savefile	Save current file in savefile
/pattern/	Search forward for pattern
?pattern? or ^pattern^	Search backward for pattern
!command	Execute command

This command will list the names of all the files and their types that are in the directory `progs`. Also this command identifies the files by context. It identifies the text files, C and FORTRAN program files.

2.4.8 `more` (Displaying the Content of a File Page by Page)

This command is used to view one output screen at a time. If the content of the file is more than one output screen, issuing the `cat` or `man` command will display the last screen. Therefore, if you want to navigate through your file in a customized way, `more` is the answer. When we give the command,

```
$more chap01
```

the message ..`more`.. along with the percentage of the file that has been viewed is displayed at the bottom of the screen. To go to the next page (next screen) use <spacebar>. Press *b* to go to the previous page and *q* to escape from the `more` command.

The `more` command can work with more than two arguments. For example, the command

```
$more chap01 chap02
```

will first show the output of chap01 and then that of chap02.

Table 2.6 The printing options available in the `lp` command

Option	*Description*	
-c	This option makes a copy of the file that is being printed.	
-m	This option sends e-mail when the printing is done.	
-p	This option enables notification on completion of the print request. Delivery of the notification is dependent on additional software.	
-s	This option suppresses the display of messages sent from lp.	
-w	This option displays message on screen when printing is done.	
-d destination	This option gives the name of the printer that you want your file to print on.	
-f form-name	This option prints file on form-name. The lp print service ensures that the form is mounted on the printer. The print request is rejected if the printer does not support form-name, if form-name is not defined for the system, or if the user is not allowed to use form-name	
-H special-handling	This option prints the print request according to the value of special-handling. The following special-handling values are acceptable:	
	hold	This option restricts not to print the print request until notified. If priting has already begun, stop it. In this case other print requests will go ahead of the request that has been put on hold until the held print request is resumed.
	resume	This option resumes a held print request. If the print request had begun to print when held, it will be the next print request to be printed unless it is superseded by another immediate print request.
	immediate	This option prints the print request next. If more than one print request are assigned, the most recent print request is printed next. If a print request is currently printing on the desired printer, a hold request must be issued to allow the immediate request to print. The immediate request is only available to LP.
-n number	This option gives the number of files to be printed.	
-o options	Different print options are:	
	Nobanner	This option does not print the banner with username and filename information.
	Nofilebreak	This option does start each new file on a new line.
	cpi=pitch	This option prints characters per inch.
	lpi=lines	This option prints lines per inch (default = 6).
	length=inchesi	This option prints the number of x printed lines per inch (the i must be at the end.)

(*Cont.*)

Table 2.6 (*Cont.*)

Option	*Description*	
	lenth=lines1	This option prints pages that are x lines long (must have 1 at the end.)
	width=chars	This option prints pages the max of chars per line.
	width=inchesi	This option prints pages the max of x inches wide.
-p pagenumbers	This option gives the number of pages that you want to print in ascending order. Specify page numbers as a range of numbers, single page number, or a combination of both.	
-q priority level	The option assigns the print request a priority in the print queue. Specify priority-level as an integer between from 0 and 39. Use 0 to indicate the highest priority; 39 to indicate the lowest priority. If no priority is specified, the default priority for a print service is assigned by the LP administrator. The LP administrator may also assign a default priority to individual users.	
-S character-set | print-wheel	This option prints the request using the character set or print wheel. If a form was requested and requires a character set or print wheel other than the one specified with the -S option, the request is rejected. Printers using mountable print wheels or font cartridges use the print wheel or font cartridge mounted at the time of the print request, unless the -S option is specified. *Printers using print wheels:* If print wheel is not the one listed by the LP administrator as acceptable for the printer, the request is rejected unless the print wheel is already mounted on the printer. *Printers using selectable or programmable character sets*: If the -S option is not specified, LP uses the standard character set. If character set is not defined in the terminfo database for the printer, or is not an alias defined by the LP administrator, the request is rejected.	
-t title	This option prints a title on the banner page of the output. Enclose title in quotes if it contains blanks. If title is not specified, the name of the file is printed on the banner page.	
-T content-type [-r]	This option prints the request on a printer that can support the specified content-type. If no printer accepts this type directly, a filter will be used to convert the content into an acceptable type. If the -r option is specified, a filter will not be used. If -r is specified and no printer accepts the content-type directly, the request is rejected. If the content-type is not acceptable to any printer, either directly or with a filter, the request is rejected.	
-y mode-list	This option prints the request according to the printing modes listed in mode-list. The allowed values for mode-list are locally defined. This option may be used only if there is a filter available to handle it; otherwise, the print request will be rejected.	
filename	This option gives the name of the file that you want to print.	

In this case, the sample output of the given command will look like the following code:

```
………..
………
chap01
…………..
…………….
.
.
-more-(98%)
-more-(next file :chap02)
…………
………….
Chap02
…………
………….
..
..
..
```

Here is another `more` command

```
$more +r "background" chap01
```

This command will give the output of the lines containing the pattern *background*. Yet another `more` command

```
$more +200 chap01
```

will show the contents of the file chap01 from line number 200 onwards

Let us have a look at the options available with the `more` command in Table 2.7.

Table 2.7 The options of the **`more`** command

Option	*Description*
f	This option displays the current line number.
spacebar	This option displays the next screen.
kf	This option skips k screens forward.
kb	This option skips k screens backward.
/pattern	This option searches for a pattern forward.
n	This option repeats last search forward.
v	This option starts up vi editor.
! command	This option executes UNIX command 'command'
=	This option displays current line number.

2.5 Protection and Security Commands

2.5.1 File Permissions

In UNIX every file has permissions associated with it, which determine who can work with the files. Actually in UNIX there are three types of users – file owner, group owner and other users. The user who creates the file is called the file owner. Consider a group of five people working on a project. Each one is working on a specific part of the project. Here each user has his own file of which he is the file owner, and the other four are group owners of his file. The third type of users, the other users, include data entry operators and those who are not involved in the project.

In UNIX, the three types of users are identified by the following symbols:

-u – Symbol for the file owner.
-g – Symbol for the group user.
-o – Symbol for the other users.

2.5.2 `chmod` (Changing the File Access Mode)

A user may have three types of access permissions – read, write and execute represented by `r`, `w` and `x`, respectively. The file owner can define the following kinds of access with the help of `chmod` (change mode) command.

1. **`$chmod u+x file1`:** This command will give the execute permission to the user (file owner) for the file *file1*.
2. **`$chmod u-x file1`:** This command will deprive the user of the execute permission for the file *file1*.
3. **`$chmod ugo +x file1`:** This command will give the execute permission to all three types of users – the file owner, the group user and other users – for the file *file1*.

To find out the permission for a particular file, use `ls -l` command with filename as argument as shown below:

```
$ls -l file1
```

Let us understand the following command and check the permissions bits (rwx) of the file *file1*. The following command shows the file owner is having read, write and execute permissions, while group as well as others are having the read permission:

```
-rwxr--r-- 1 file1 group 0 may 10 20:30 file1
```

In the above command, the very first symbol (-) shows that *file1* is an ordinary file. The next three letters (`rwx`) depict that the file owner has read, write and execute permission on this particular file. The next notation `-r` says that the group has read permission only and the others also have read permission only.

The permission weights can also be used to change permission using the `chmod` command. The permission weights for the file-related permissions are:

1. 4 – for read.
2. 2 – for write.
3. 1 – for execute.

Table 2.8 The options used with the **chown** command

Option	*Description*
-R	This option changes the permission on files that are in the subdirectories of the directory that you are currently in.
Newowner	This option changes the alias/username of the new (authorized) owner of the file.
Filenames	This option gives the file that you are changing the rights to.

Here is another file permission-related command:

```
$chmod 777 file1
```

This command will give read, write and execute permissions to all users.

2.5.3 `chown` (Changing the File Owner)

Having created any directory or file, the owner of the file/directory can give the ownership to any other person who has an authorized access to the system. This is possible through the `chown` command. If for some reason, the owner desires to give permissions to any other person or a single person alone, this is also possible. The syntax of the command is given below and the command options are defined in Table 2.8:

```
chown [-R] newowner filenames
```

Here is an example of the `chown` command. The following command

```
$chown newown file.txt
```

will give permissions as owner to the newown user for the file.txt file.

2.5.4 `chgrp` (Changing the User Group)

This command changes the group that has access to a file or directory. Every user in UNIX belongs to at least one group; and it is possible that the super user may also authorize a user to be more than one group. But anytime user can belong to only one group. This can be accomplished by the command `chgrp` in the following way.

```
chgrp newgroup filenames [-f] [-h]
```

The arguments/options available in the `chgrp` command have been described in Table 2.9.

In the following example of the `chgrp` command, the group for the file file.txt is changed to newown if this group is valids

```
$chgrp newown file.txt
```

2.5.5 `newgrp` (Logging into a New Group)

The users can change the group if they wish under the following two distinct situations:

1. While dealing with files and directories to be created.
2. While working with files already created (old files).

Table 2.9 The arguments/options used with the **`chgrp`** command

Argument/Option	*Description*
newgroup	This option specifies the new group that you wish to allow access to.
filenames	This option specifies the files that the new group has access to.
-f	This option compels not to report errors.
-h	If the file is a symbolic link, this option changes the group of the symbolic link. Without this option, the group of the file referenced by the symbolic link is changed.
-R	Recursive. `chgrp` descends through the directory, and any subdirectories, setting the specified group ID as it proceeds. When a symbolic link is encountered, the group of the target file is changed (unless the -h option is specified), but no recursion takes place.

The `newgrp` can be used in the following way:

```
newgrp [ -| -l ] [ group ] [argument]
```

The arguments/options used in the `newgrp` command have been described in Table 2.10.

2.5.6 `passwd` (Changing the Password)

The user can change the password with the help of the `passwd` command. On giving this command, the user is asked to enter the old password. If it is entered correctly then the user is asked to enter the new password twice – second time for confirmation. If both do not match then the user is given another chance to enter the new one.

The command needed to change the password is:

```
$passwd<enter>
```

Old password:***** <enter>
New password:*****<enter>
Re-enter new password:***** <enter>
$
Prompt returns command complete.

Table 2.10 The arguments/options used in the **`newgrp`** command

Option/Argument	*Description*
- | -l	This option changes the environment to what would be expected if the user actually logged in again as a member of the new group.
group	This option is a group name from the group database or a non-negative numeric group ID. It specifies the group ID to which the real and effective group IDs will be set. If the group is a non-negative numeric string and exists in the group database as a group name, the numeric group ID associated with that group name will be used as the group ID.
argument	`sh` and `ksh` only. Options and/or operand of the `newgrp` command.

2.6 Communication Commands

2.6.1 `write` (Sending Message to the Other Logged-in User)

The external `write` command sends messages you type on your terminal in real time to another user's terminal. In other words, the `write` command works when the intended receiver of the message to be given is logged in. The `write` command reads the standard input and writes to the specified user's terminal type (tty). The `write message` will appear on the receiver's terminal as he logs in. If there are two users with the same login name then the terminal name has also to be used with the login name to send the `write message` to the intended user.

Here is a `write` command example in which we send a message to the user kumar's terminal. The message – *What about going out for a dinner?* – will appear on kumar's terminal when he logs in.

```
$write kumar
```

What about going out for a dinner?

```
^d
$
```

2.6.2 `mail` (Sending Message to the User who is not Logged-in)

This command enables sending mail to a user even if he is not logged in. It is mainly used for non-interactive communication. It uses the standard input. Unlike the `write` command, the `mail` command doesn't need the user to be logged in. It saves the message in a mailbox which is placed in the directory */usr/spool/mail* (`/var/mail` in release 4).

In the following example, a mail is sent to kumar with the message: *The new system will start functioning from the next month.*

```
$mail kumar
subject: New system
The new system will start functioning from the next month
^d
$
```

This command will put a message exclusively for the user kumar. When the user kumar logs in, he will see the message *you have a mail* on his terminal. To see the received message, he will give the following command

```
$mail
```

On execution of the above command, a listing of all incoming mails, received by kumar, will be produced. If there is only one message, the user presses <*enter*> to see that message. If there are more messages, to see a particular message the user needs to enter its number at the & prompt. For example, the following command

```
&3
```

will show the message numbered 3.

Table 2.11 The options used with the `mail` command

Option	*Description*
+	This option prints the next message.
–	This option prints the previous message.
N	This option prints the message numbered N.
h	This option prints headers of all messages.
d	This option deletes the current message.
dN	This option deletes the message numbered N.
u	This option undeletes the currently deleted message.
uN	This option undeletes the message numbered N.
s filename	This option saves the current message in the file filename. If the file name is not specified, save the message in the file m box.
q	This option quits the mail program.
p	This option prints the mail.

The mail message can also be stored in a desired file. For example, the command

```
&w mfile
```

will store the current message into the file mfile instead of storing it into mail box. The options used with the `mail` command have been described in Table 2.11. You can also send the `mail` command to yourself. Here your login name will be the argument. Assuming your username is *kumar,* you can use the `mail` command to send a mail message to yourself in the following way:

```
$mail kumar
```

2.6.3 `mesg` (Receiving or Avoiding Other Users' Messages)

In UNIX, one is allowed, as stated earlier, to get real-time message via the `write` command. This can be avoided by issuing the `mesg` command. This command is used to send the messages to the specific recipient. It will also prevent other people from writing to one's terminal. In the `mesg` command syntax, `n` stands for NO and `y` stands for YES. To prevent other people from writing to one's terminal, we will give the command

```
$mesg n
```

To enable the response to other users, we will give the command

```
$mesg y
```

2.6.4 `wall` (Messaging All Logged-in Users in a System)

The system administrators normally use this command to send message to all users who are currently logged in. This command is available only in the directory `/etc`. However, a user can invoke this command provided he has the authority. There are no arguments and standard input is used.

Suppose our system administrator wants to send the message that every body should logout as he is going to shut down the system. He issues the following command. After execution of this command, every logged-in user will receive the message: *Please, logout. Thank you.* on their terminal.

```
$/etc/wall
Please, logout. Thank you.
^d
$
```

2.7 Information Commands

2.7.1 `man` (Getting On-line Help)

This command is used to get help on a particular command. For example, the command

```
$man ls
```

will show you the help topics related to the `ls` command.

2.7.2 `who` (Displaying the Names of All Logged-in Users)

The `who` command displays the names of all the users who are logged onto the system. It produces a simple three-columnar output. The first column is the user's name, the second column represents the terminal type, and the third and fourth columns give date and time.

The following `who` command indicates that currently two users – root (console itself) and kumar – are logged in.

```
$who
root console Jan 30 10.3
kumar tty01 Jan 30 14.09
```

To have the extended output in `who`*,* we should use the `who` command with `-H` option that will give the column heading, user, line and login time.

`who am i` command: This command also functions like the `who` command. With the help of the `who am i` command, the login details of the user who invoked the command will be displayed. Its syntax is

```
$who am I
```

Like the `who` command, the `who am i` command produces a simple three-columnar output. The first column gives the username, the second column represents the terminal type, and the third and fourth columns give date and time. In the following `who am i` command, the user is *kumar* who issued the command:

```
kumar tty01 jan30 14.09
```

2.7.3 `date` (Displaying the Date, Month, Time and Year)

With the help of the `date` command, the user can display the system-specified date, month, time and year. The `date` command can also be used for setting the date and time. In specific cases, if only date and month or date and time are required then we will have to work with special combination of options.

Table 2.12 The options of the `date` command

Option	*Description*
%a	Abbreviated weekday name
%A	Full weekday name
%b	Abbreviated month name
%B	Full month name
%c	Day and time
%d	Day of the month: (0, 31)
%H	Hour of the 24-hour day : (00, 23)
%I	Hour of the 24-hour day : (00, 12)
%j	Day of the year
%m	Month: (0, 12)
%M	Minute: (00, 59)
%p	AM/PM
%S	Second
%T	Time as HH:MM:SS
%x	Date
%X	Time
%y	Year without century: (00, 99)
%Y	Year with century
%Z	Time zone name

The following example shows how to use the `date` command.

```
$date
Fri Mar 24 12:30:05 IST 1994
```

The `date` command has the following options discussed in Table 2.12.

2.7.4 `cal` (Displaying the Calendar)

UNIX will display a calendar of one or more months with the help of the appropriate `cal` command. The general format of the `cal` command is

```
$cal [month] year
```

The following `cal` command

```
$cal 3 1998
```

will display the calendar for the month of March for the year 1998.

2.7.5 `tty` (Displaying Information about the Terminal Type)

The `tty` command tells you which terminal you are using. It only gives you the information about the terminal type. The following `tty` command shows the terminal type as `tty01`.

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

`stty` command: The `stty` command is a variant of the `tty` command. This command is used to configure the current terminal. We can set tabs using the `stty` command. The command `stty -tabs` will set tab characters if the terminal doesn't have them. When the user issues this command, the system will convert tabs into the right number of spaces. The command `stty  -a` will display all the current settings.

2.8 Process Management Commands

Process: In UNIX, each task or program that the computer undertakes or runs is called a process. When you run a command then it is a process. If you run the same command again, then it is another process. A UNIX system can handle several processes simultaneously. But the systems that use UNIX can do only one thing at a time because they have only one processing unit.

UNIX uses time-sharing to solve the problem of multiple demands on its single-track mind. Here the kernel maintains a list of processes and issues each process a bit of time. Before finishing the process the kernel is currently handling, it switches over to another process. So all the processes have to experience several cycles of time-sharing before their completion. But this time-sharing process is so fast that the user never comes to know that his particular command waits in a queue before execution.

The process that is actually running is given a run status. The processes that are waiting for their turns for execution are issued a sleeping status. Each time a process is initiated, the kernel assigns the process a unique Process Identification Number (PID). The shell also has its own PID, which is stored in a variable called $$. Each process in UNIX has a parent and child. The shell is the parent, and all user commands are children to it. After the completion of each process, the child dies.

For example, in the command

```
$cat emp.lst
```

the shell is the parent and the *cat* command is the child.

2.8.1 `at` (Scheduling a Job)

The `at` command allows a script to be executed once at a specified time and/or date. Once the user submits a job for execution at a specified time, the scheduler takes care of the job. The process is called cron which stands for chronograph. Neither the user nor the super user can start cron. UNIX starts cron during booting and activates for every minute. It checks the jobs for execution and goes back to sleep. `at` uses the cron service. This is an alternative to running one-off jobs in the *cron*. While using the `at` command, you need not remove the cron line even if it is no longer required.

Let us learn about different uses of the *at* command.

1. **Scheduling a Job**

   The `at` command, given in the following syntax at the UNIX command prompt, allows a job to be scheduled:

```
at {time-to-run} {date-to-run}
```

   The following *at* command

```
at 0800 Feb 23
```

will schedule a job to be run at 0800 hours on February 23.

   Note that if in the `at` command:

- No date is specified, today is assumed.
- No date is specified and the time is less than now, tomorrow is assumed.
- No month is specified, the current month is assumed.
- A month earlier than the current month is given, next year is assumed.

   Using Ctrl-d will exit the prompt ($) and offer you a job reference number. When an at job has run, a message will be written to the UNIX mailbox about the account name running the job.

2. **Checking and Cancelling Jobs**

   You can view your queued up jobs using the command *at* the UNIX prompt. This will reveal the job reference number, the owners (who initiated the job), and the date and time that jobs are due to run.

   To cancel a job, use the following command

```
atrm {job reference}
```

at the UNIX prompt. You must have appropriate permissions to do this.

   To cancel all jobs in your current UNIX account, use the following command:

```
atrm –a
```

2.8.2 ps (Knowing the Process Status)

To get details about a particular process running on a UNIX system, you can use the following `ps` (process status) command.

```
$ps
```

The following table gives the outcome of a `ps` command. Primarily it shows four columns that actually depict the status of the process. The column headings have been explained after the table.

PID	TTY	TIME	COMMAND
2330	01	12.09	sh
2340	01	00.00	ps

PID – Process Identification Number
TTY – Terminal type or input device that the user is using
TIME – The time in which the process is being executed
COMMAND – Name of the command

The `ps` command has the following option

`-f` – In addition to normal `ps` command, the -f option gives the login name, parent PlD, amount of CPU time consumed by the process and the command with arguments.

The `ps` command with the `-f` option is given as

```
$ps -f
```

Given below is the extended information on our process which we get when we use the `ps` command with the `-f` option. This information is given as an eight-column output. Each field is explained after the output.

```
UID     PID   PPID   C    STIME      TTY    TIME    COMMAND
Kumar   30    1      0    12:09:11   02     2:34    sh
Kumar   89    30     22   12:22:12   02     0.19    ps-f
```

UID – User_id
PID – Process_id
PPID – Parent PID
C – Amount of CPU time consumed by the process.
STIME – Time that has been elapsed ever since the birth of the process
COMMAND – Full command with options. So if you are running the program and
forget the exact options, you could use the –f option to see
the command and argument typed.
-u – Lets you know the activities of the user. This option has to be
followed with a user-id

Here is another `ps` command.

```
$ps -u kumar
```

This command will display what kind of processes the user (kumar) is using.

2.8.3 Background Processing

Background processes are generally used for a longer task such as sorting a file or compiling a huge program. To make a process run in the background, terminate the process with &. The shell waits until the child process produced by the command line dies. You can't issue a command until the preceding one finishes. In the command example given below we have a huge file emp.lst. Imagine if it is going to take one hour to sort, our system will be busy in executing it, affecting other priority-based processes. Therefore if we make the sorting of emp.lst run in the background, this will give the user the option to use the system for another process. Assigning & after our command example will run this job of sorting in the background. This will return the process ID which will help the user know the status of the process.

```
$sort emp.lst &
4463
$
```

The number 4463 received is PID.

Here since no redirection has been applied to the output, the user may come across the output of this command when working with other commands.

The UNIX system will give a completion report when the background process is over:

```
[1] + Done sort emp.lst
```

2.8.4 `kill` (Aborting a Process)

The `kill` command is used to abort a process. For example, the following `kill` command

```
$kill 120
```

terminates the process with ID 120.

If you are running more than one process in the background, they all can be killed with a single `kill` statement. For example, the following command will kill the processes with IDs 123, 133, and 345.

```
$kill 123 133 345
$
```

If a background process gives birth to child processes, you need to kill only the parent process to kill the child processes.

Another useful `kill` command is

```
$kill 0
```

This command terminates all background processes without requiring to mention their PIDs. But this command will not kill the login shell. Also if some process doesn't oblige the `kill` command, to kill that process, the -9 option with the `kill` command is used, like

```
$kill -9
$
```

The -9 option is known as sure kill.

2.8.5 `wait` (Keeping a Process on Hold)

The `wait` command is used for keeping a process on hold till other processes get completed. It can be given in the following format

```
wait [n] [jobnumber]
```

where n is the PID or job number of a currently executing background process (job). If n is not given, the command waits until all jobs known to the invoking shell have terminated. The `wait` command normally returns the exit code of the last terminated job. It can also be defined in the following way

```
wait [pid] [jobid]
```

Table 2.13 describes the options used in the `wait` command. Here is an example of the `wait` command. The following command

```
$wait 2017
```

Table 2.13 The options used in the **`wait`** command

Option	*Description*
Pid	This option is the unsigned decimal integer process ID of a command, for which the utility is to wait for the termination.
Jobid	This option is a job control job ID that identifies a background process group to be waited for. The job control job ID notation is applicable only for invocations of `wait` in the current shell execution environment, and only on systems supporting the job control option.

instructs the system to wait on PID 2017 until termination of other processes. Here 2017 is the PID number of the process that will wait for all termination of all other background processes.

2.8.6 `sleep` (Sending a Process to Sleep for Specified Time)

This process command will make a process to wait an *x* amount of seconds. This command is very useful in a multi-tasking environment where two or more processes are executed. In a multi-tasking environment, there are some process(es) that require(s) more attention than other processes. The `sleep` command will make a specified process to sleep for a specific period of time. Only after the completion of the first process, the second process will be executed. The syntax of the `sleep` command is

```
$sleep [time]
```

where time represents the amount of seconds to wait.

The following `sleep` command

```
$sleep 10
```

will make the respective process to sleep for 10s.

2.8.7 `nice` (Altering the Scheduled Priority of Processes)

This command is used to alter the scheduled priority of processes. UNIX processes have an associated system nice value which is used by the kernel to determine when these processes should be scheduled to run. This value can be increased to facilitate processes executing quickly or decreased so that the processes execute slowly and thus do not interfere with other system activities.

The process scheduler, which is part of the UNIX kernel, keeps the CPU busy by allocating it to the highest priority process. The nice value of a process is used to calculate the scheduling priority of the process. Other factors that are taken into account when calculating the scheduling priority of a process include the recent CPU usage and its process state, for example 'waiting for I/O' or 'ready to run'.

Normally, processes inherit the system nice value of their parent process. At system initialization time, the system executes the init process with a system nice value of 20. This is the system default priority. All processes will inherit this priority unless this value is modified with the command `nice.` The nice value of 0 establishes an extremely high priority, whereas a value of 39 indicates a very low priority on SVR4 derived

Table 2.14 The options used with the ***nice*** command

Option	*Description*
-increment | -n increment	This option is the increment that must be in the range 1 – 19; if not specified, an increment of 10 is assumed. An increment greater than 19 is equivalent to 19. The super-user may run commands with priority higher than normal by using a negative increment such as −10. A negative increment assigned by an unprivileged user is ignored.
Command	This is the option for the name of a command that is to be invoked. If command names any of the special built-in utilities, the results are undefined.
Argument	This is the option for any string to be supplied as an argument when invoking command.

systems. On BSD derived systems scheduling priorities range from 0 to 127. The higher the value, the lower the priority, and the lower the value, the higher the priority.

Table 2.14 describes the options used with the `nice` command.

On systems derived from BSD, the `nice` command uses the numbers −20 to 20 to indicate the priorities, where 20 is the lowest and −20 is the highest. Any user can lower the priority of his/her processes; however, only the super-user can increase the priority of a job. To decrease the priority, the following command is given:

```
$nice -6 emp.lst
```

To increase the priority, the following command is given:

```
$nice --6 emp.lst
```

2.8.8 batch

Once the user submits a job for execution at a specified time, the scheduler takes care of the job. This can be accomplished by using the `batch` command. Here, the system decides when a job is to be executed using `batch`. The syntax for the `batch` command is

```
$batch
```

Let us look at the following example of the batch command:

```
sort < file1 > file2
^d (Ctrl+d)
Job 12345. b...
```

The above command creates a file with extension b. Here we have scheduled that first file1 will be sorted and after that file2. Here one file is created with the job number and extension of *b* (means batch).

2.9 I/O Redirection

Many UNIX commands take text-like input and/or produce text-like output. It is sometimes useful to be able to control where the input comes from and where the output goes (via redirection), or even pass the output from one command to another's input (via pipes).

Three ways to redirect output/input and their corresponding symbols are:

>	–	Redirect the output from a command to a file on disk. If the destination file already exists, it will be erased and overwritten without warning; so be careful.
>>	–	Append the output from a command to an existing file on disk.
<	–	Read a command's input from a disk file, rather than the user. Be careful not to type '>' by mistake, or you will erase the contents of the file you are trying to read from.

2.9.1 Standard Files

Every UNIX program, including certain commands, is connected to three files called standard input, standard output and standard error. If a program doesn't explicitly request other files, it by default gets the input from the standard input device, and sends the output and error messages to the standard output and standard error files, respectively. By default, the keyboard is the standard input file, and the VDU is the standard output and error file.

Each standard file is identified by a file descriptor assigned to it as described below

Standard file	**File descriptor**
Standard input	0
Standard output	1
Standard error	2

2.9.2 Standard Input

As stated in the above topic, the keyboard is the standard input file. When the user types `cat` at the $ without giving any arguments, and then presses RETURN, the `cat` command waits for the user to input arguments through the keyboard. Let us work with the following example of the `cat` command:

```
$cat

This is a trial

^d

This is a trial

$
```

In the above example, when the user types *This is a trial* and presses Ctrl-d (to tell that no more input is there) and then presses the RETURN key, the same message *This is a trial* appears on the screen. Here, the `cat` command waits for the user to input something, and the output of the command is returned to the standard output file (screen).

2.9.3 Input Redirection

Instead of entering the input values (through the standard input file), the user can modify the `cat` command to extract the input values from a disk file. For example, the following *cat* command

```
$cat < chap01
```

will take input from the disk file *chap01* and then display the taken input on the screen.
The above command can also be written as

```
$cat 0 <chap01
```

2.9.4 Standard Output

Generally, the output of a `cat` command is displayed on the VDU screen because the `cat` command by default assigns the VDU as the standard output file. Let us now learn how to redirect the `cat` command output.

Output redirection: To redirect the `cat` command output, symbol '>' is used. For example, the execution of the following command

```
$cat chap01 > newfile
```

will result in writing the output of chap01 on newfile instead of displaying it on the screen. If newfile is already present, its existing contents will be overwritten by the contents of chap01. And, if newfile does not exist, it will be created by the shell.

The above command can also be written as

```
$cat chap01 1>newfile
```

Preventing Overwriting: To prevent overwriting while redirecting the output to an existing file, use '>>' instead of '>'. When we give our chap01 related command with the symbol '>>' as shown below

```
$cat chap01 >> newfile
```

the contents of chap01 will be appended to the newfile. This command can also be written as

```
$cat chap01 1>> newfile
```

2.9.5 Standard Error File

Standard error is another output stream typically used by programs to output error messages or diagnostics. It is a stream independent of standard output and can be redirected separately. The usual destination is the text terminal which started the program to provide the best chance of being seen even if the standard output is redirected (so not readily observed). For example, the output of a program in a pipeline is redirected to the input of the next program, but errors from each program still go directly to the text terminal.

It is acceptable – and normal – for the standard output and standard error to be directed to the same destination, such as the text terminal. Messages appear in the same order as the program writes them unless buffering is involved.

When the user types an invalid command, an error message appears on the screen because the VDU is assigned as standard error file.

2.9.6 Redirecting the Error Messages

The following example shows how an error message is redirected:

```
$cat file1 2> error_msg
```

Here if file1 doesn't exist, the related error message will be stored in the file called error_msg.

2.9.7 Pipes

UNIX commands alone are powerful, but by combining them you can accomplish complex tasks with ease. The way you combine UNIX commands is through using pipes and filters.

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

Simply put, pipes are the means to connect one program with the other. A pipe connects the output of one program with the input of another. For example, in the following command

```
$command1 | command2 | command3
```

the output of command1 is sent as input to command2, and the output of command2 becomes the input for command3.

With the help of piping facility we can create new commands. Here is an example of a pipe command

```
$who am i | wc -c
```

In the above command, the output of `who am I` command acts as input to the `wc` command, and the result that will be displayed on the screen will be the total number of characters in the output of `who am I` command.

Here is another example of the pipe command:

```
$cat emp.txt
sita
gita rita
ramesh
$cat apple.txt | wc
3 4 21
$
```

In this example, at the first shell prompt, the contents of the file emp.txt are shown. In the next shell prompt, the `cat` command displays the contents of emp.txt. This display is not sent to the screen but to the `wc` (word count) command through a pipe. The `wc` command then does its job and counts the lines, words and characters of what it got as input.

2.9.8 Filters

Any command which takes input from a standard input device and sends output to the standard output device is a filter. In other words, a filter is a program that reads the input from a standard input device, filters (transforms) it and writes it on the standard output device. Commands like `cat`, `sort`, `grep`, etc. are examples of filters.

`tee` command: The external `tee` command is used to write to the standard output and to a file simultaneously. You place the `tee` command anywhere in a pipe command to divert a copy of the standard input (of `tee`) to disk and another copy to your terminal or the next stage of the pipe. The name of this command is from the world of plumbing as it allows one input and two outputs. In any complex pipes, the `tee` command serves as a glue with to join one stage with the subsequent stage.

The `tee` command reads from the standard input file and then sends the output to the standard output file. While doing this, it also sends a copy of the output to a file of the user's choice. Here is an example of the `tee` command.

```
$ls | tee user.lst
```

After the execution of the above command, the output of the `ls` command is displayed on the screen as well as stored on the file user.lst. Any number of files can be specified to hold the output. If the output file already exists then it will be overwritten. To prevent overwriting a file's contents, the -3 option is used.

The following example shows the use of the `tee` command. In this example the *tee* filter command is used for the `ps` command output that is redirected to emp.txt file and finally printing is done page by page.

```
$ps -ax | tee emp.txt | more
```

You can use the `ps` command to get a list of processes running on the system, store it in the file emp.txt, and also pass it to `more` to display it one screen at a time.

2.10 Piping and Filter Commands

2.10.1 `tail` (Displaying Last Few Lines of a File)

The `tail` command displays the last few lines of a file. If the number of lines option is not specified, it prints the last 10 lines of a file by default. For example, the following `tail` command

```
$tail -1 emp.lst
```

will print the last line of the emp.lst file.

The `tail` command can also be used to print a file starting at a specified line. For example, the command

```
$tail +3 emp.lst
```

will print the contents from the third line.

2.10.2 `head` (Displaying First Few Lines of a File)

The `head` command displays the first few lines of a file. If the number of lines option is not specified, the `head` command by default will show the first 10 lines of the specified file. For example, the following `head` command

```
$head emp.lst
```

will show first 10 lines of the emp.lst file.

Another `head` command

```
$head -3 emp.lst
```

will display the first three lines of the emp.lst file.

2.10.3 `nl` (Line Numbering Filter)

The `nl` filter reads lines from a specified file or from the standard input if no file is named, and reproduces the lines on the standard output. Lines are numbered on the left in accordance with the command options in effect.

The `nl` filter views the text it reads in terms of logical pages. Line numbering is reset at the start of each logical page. A logical page consists of a header, a body and a footer section. Empty sections are valid. Different line numbering options are independently available for header, body and footer. For example, the default numbering option `bt` numbers non-blank lines in the body section and does not number any lines in the header and footer sections.

`nl` copies each specified file to the standard output, with line numbers added to the lines. The line number is reset to 1 at the top of each logical page. `nl` treats all of the input files as a single document and does not reset line numbers or logical pages between files. A logical page consists of header, body and footer.

The beginnings of the sections of logical pages are indicated in the input file by a line containing nothing except one of the following delimiter strings:

```
\:\:\: start of header
\:\: start of body
\: start of footer
```

The section delimiter strings are replaced by an empty line on output. Any text that comes before the first section delimiter string in the input file is considered to be part of a body section, so a file that does not contain any section delimiter strings is considered to consist of a single body section.

The following `nl` filter command numbers the lines in the file myfile and puts the output to the file file2.

```
$nl file1 > file2
```

2.10.4 `uniq` (Eliminating or Counting Duplicate Lines in a Presorted File)

The `uniq` command can eliminate or count duplicate lines in a presorted file. It reads in lines and compares the previous line to the current line. Depending on the options specified on the command line, it may display only unique lines or one occurrence of repeated lines or both types of lines. The `uniq` syntax is

```
sort | uniq
```

and

```
sort | uniq -c
```

For example,

$uniq -u dept.lst – Prints only those lines of the file `dept.lst` that are unique (i.e., the lines are not duplicated).

$uniq -c dept.lst – Counts the frequency of occurrence of all lines in the file dept.lst.

$sort dept| uniq –u – First sorts the file `dept` and the output is piped to `uniq` command which will print only those lines which are not duplicated.

2.10.5 `grep` (Searching a Pattern in a List of Files)

The abbreviation `grep` stands for 'global regular expression printer'. The `grep` command allows you to search for a pattern in a list of files. Such patterns are specified as 'regular expressions' which in their simplest form are strings such as words or sentence fragments.

The way we search for a string with `grep` is to put the words we are searching for together in single quotes. The `grep` *`syntax`* is

```
$grep pattern file-name-1 file-name-2 …file-name-n
```

While executing the above command, `grep` searches the named file filename for the pattern and prints the line containing the pattern. If no filename is given, `grep` takes input from the standard input device.

A `grep` command example is

```
$grep 'ramesh publishers' emp office
```

On the execution of the above command, the operating system will print all the lines in the file emp and the file office that contain the string 'ramesh publishers'. By default the line will be printed on the computer screen. Common options available with `grep` are discussed in Table 2.15.
The following are the two `grep` commands with -c and -s options:

```
$grep -c abc file1 – It will give the count of lines that contain the
                     pattern abc.
$grep -s xyz file1 – It will return 1 if no match is found, else it
                     will return 0.
```

`grep` regular expressions: Regular expressions are the facility to specify a generalized expression to match group-similar patterns. These are a string of ordinary and meta characters, which can be used to match more than one pattern. The `grep` search pattern is the first argument following the command name and possible options. If the pattern contains space or a tab character, it should be enclosed within single quotes (' ') or double quotes (" "). For example, the command

```
$grep new delhi places.lst
```

Table 2.15 Options available in the **grep** command

Option	*Description*
-v	This option prints those lines that do not match the pattern.
-c	This option prints only a count of number of matching lines.
A	This option prints only the names of files with matching lines in a group of files.
-n	This option prints the line number with each output line.
-s	This option prints no output, just returns the status.
-i	This option ignores case for matching.
-y	This option gives lowercase letters in the pattern will also match the uppercase letters.

will cause `grep` to seek the pattern new in the delhi and places.lst files. To avoid this give new delhi in single or double quotes as follows:

```
$grep 'new delhi' places.lst
```

More complex regular expressions can be specified by using characters summarized in Table 2.16.

egrep syntax vs grep syntax: Ironically, despite the origin of its name, `egrep` (i.e. `extended grep`) actually has less functionality as it is designed for compatibility with the traditional `grep`. A better way to do an extended `grep` is to use `grep -E` which uses the extended regular expression syntax without loss of functionality.

Table 2.16 Characters used to specify regular expressions

Character	*Matches*	*Example*	*Description*
[pqr]	Matches a single character p, q or r.	`grep 'appl [liye]' file1`	Specifies search pattern as appli, apply or apple.
[c1-c2]	Matches a single character within the ASCII range of c1 and c2.	`grep 'new [a-c]' file1`	Specifies search pattern as newa, newb or newc.
[^pqr}	Matches a single character which is not a p, q or r.	`grep 'new[^pqr]' file1`	Matches search patterns as new with any character other than p, 1 or r.
^pat	Matches a pattern pat which must occur at the beginning of a line.	`grep '^a[pat]' file`	Specifies the search pattern as ap, aa or at but this must be at the beginning of line.
*	Matches zero or more occurrences of the previous character.	`grep 'a*' file1`	Searches pattern with a and followed by any character or none.
Pat$	Matches pat at the end of line.	`grep 'pat$' file1`	Searches for pattern pat but this must be at the end of the line.
.(dot)	Matches a single character.	`grep 'pat' file1`	Searches for the pattern pat followed by any character.

Table 2.17 The options of the **`grep`** and **`fgrep`** commands

Option	*Description*
-i	This option ignores case so that uppercase and lowercase characters match each other.
-n	This option displays the line number with each line containing a match.
-l	This option displays only the names of files that contain matching lines, but not the lines themselves. This is useful if you are searching through a set of files to see which of them contain a particular pattern.
-v	This option displays lines that don't match a given pattern.

`fgrep:` The `fgrep` command doesn't do any expansion. However, with a single `fgrep` command, you can search more than one string of characters. To do this, place the possible strings, one per line, in a file. Then use the option `-f` file instead of the string option when you call `fgrep`, like this:

```
$fgrep -f list intro.draft
```

If the file list contains

manual
manuals
manually

The `fgrep` command would find all lines in `intro.draft` that contain 'manual' or 'manuals' or 'manually'.
Some options used with both `grep` and `fgrep` have been described in Table 2.17.

2.10.6 `cut` (Displaying Specified Columns to the Standard Output File)

The `cut` filter is used to display specified columns to the standard output file. The `cut` filter can take filenames as command lines, and arguments of input from the standard input.

Cut options

1. **–f option:** This option is used to display specified fields. The filename whose fields are to be extracted is given at the end of the command. Given below are some -f option related commands:
   - `$cut –f1,2 file1`
     This command will cut or display the first and second columns of the disk file file1.
   - `$cut –f13-9 file1`
     This command will display fields three to nine.
   - `$cut -f16 file1`
     This command will display all the fields from line 6.
2. **-d option**: This option is valid only with -f option. By default tab character is the delimiter until and unless the user explicitly specifies it. The following -d option command

   ```
   $cut –d " " -f1-6 file1
   ```

will print the files 1 through 6 where the column delimiter is a space.

3. **-c option:** This option extracts the specified character(s). For example, the command

   ```
   $cut -c6-22 shortlist
   ```

   will display the data that has been occupied from column 6 to 22 in the file shortlist.

2.10.7 `paste` (Combining Two Files Horizontally)

The `paste` command combines files horizontally or this command is used to display two files vertically. The `paste` command is primarily used to paste files together horizontally. That is, each file is placed in a column on the output. It may also be used to take multiple files for input and generate one long serial line of output. A serial line of output has no new-line characters. Thus it is one long line. Another use is to take a stream of input and generate multiple columns of output.

If you have two files, the `paste` command would display the first file in the left (first) column and the second file in the right (second) column. The `paste` command has various functions which include:

- λ Combining files horizontally (side-by-side columns).
- λ Changing single-column input to multicolumn output. It is useful for generation of files like `/etc/passwd`
- λ Removing all new-lines from all files creating one long line (serial output).

Let us see how does the `paste` command work. For example, we have two files file 1 and file 2 shown below:

```
$cat file1
a.k.shukla
jai sharma
n.k. gupta
```

```
$cat file2
g.m.
director
d.g.m.
```

Using the `paste` command, file 1 and file 2 can be displayed vertically as

```
$paste file1 file2
a.k.shukla g.m
jai sharma director
n.k.gupta d.g.m.
```

Some options used with the `paste` command have been described as follows:

1. The -d option can be used to specify the delimiter.
2. The `paste` command uses tab as the default delimiter.
3. The data for one file can be supplied from the standard input.

2.10.8 `join` (Merging Lines of Two Sorted Text File on the Basis of a Common Field)

The `join` command merges the lines of two sorted text files based on the presence of a common field. It is a sort of implementation of the join operator used in relational databases but operating on text files.

The `join` command takes as input two text files and a number of options. If no command-line argument is given, this command looks for a pair of lines from the two files having the same first field (a sequence of characters that are different from space), and outputs a line composed of the first field followed by the rest of the two lines.

The program arguments specify which character to be used in place of space to separate the fields of the line, which field to use when looking for matching lines, and whether to output lines that do not match. The output can be stored to another file rather than printing using redirection. The `join` command forms on the standard output a join of the two relations specified by the lines of two files. Following is the syntax of the `join` command:

```
join [-a filenumber| -v filenumber] [-1 fieldnumber] [-2 fieldnumber]
[-o list] [-e string] [-t char] file1 file2
```

Table 2.18 describes the options used in the `join` command.

Now we will see the use of the `join` command with the help of the following example. Here, the two files list the known fathers and mothers of some people:

```
gopal jogi
mary jagat
arun mamta
gopal sita
```

The `join` of these two files (with no argument) would produce:

```
gopal jogi sita
```

Indeed, only 'gopal' is common as a first word of both files

2.10.9 `comm` (Comparing Two Files for Similarities and Differences)

This command is a file comparison program. To use the `comm` command, two sorted files have to be given. This command prints three columns of output. The first column contains lines that occur only in the first file, the second column contains lines that occur only in second file, and the third column contains lines that are common to both the files. The `comm` command syntax is:

```
$comm [options] file1 file2
```

Let us consider two files file1 and file2.

Contents of file1	Contents of file2
-------------------	-------------------
India	Australia
Pakistan	England

Table 2.18 The options used in the `join` command

Option	*Description*
-a filenumber	In addition to the normal output, this option produces a line for each unpairable line in file filenumber, where filenumber is 1 or 2. If both -a 1 and -a 2 are specified, all unpairable lines will be output.
-v filenumber	Instead of the default output, this option produces a line only for each unpairable line in filenumber, where filenumber is 1 or 2. If both -v 1 and -v 2 are specified, all unpairable lines will be output.
-l fieldnumber	This option is a join on the fieldnumber field of file 1. Fields are decimal integers starting with 1.
-2 fieldnumber	This option is a join on the fieldnumber field of file 2. Fields are decimal integers starting with 1.
-j fieldnumber	This option is equivalent to -1fieldnumber -2fieldnumber.
-j1 fieldnumber	This option is equivalent to -1fieldnumber.
-j2 fieldnumber	This option is equivalent to -2fieldnumber. Fields are numbered starting with 1.
-o list	This option instructs that each output line includes the fields specified in list. Fields selected by list but not appearing in the input will be treated as empty output fields (see the -e option). Each element in the empty output fields has either the form filenumber, fieldnumber, or 0, which represents the join field. The common field is not printed unless specifically requested.
-e string	This option replaces empty output fields with string.
-t char	This option uses character char as a separator. Every appearance of char in a line is significant. The character char is used as the field separator for both input and output. With this option specified, the collating term should be the same as `sort` without the -b option.
file1 file2	This option facilitates to file name and/or directory and file name of the files being joined.

Applying the following `comm` command on these files

```
$comm file1 file2
```

we get the output

```
 India          Australia
 Pakistan       England
---------------- --------------- ------------------
column 1        column 2     column 3 (Blank)
```

Here column 1 contains the lines unique to file1, column 2 contains the lines unique to file2 and column 3 is blank because there is no common line in both the files. The user can choose which column(s) should or should not be displayed with the help of field number(s).

Table 2.19 The options used in the *join* command

Option	*Description*
comm.-1	This option suppresses the display of column 1 (displays columns 2 and 3).
comm.-2	This option displays columns 1 and 3.
comm.-3	This option displays columns 1 and 2.
comm.-4	This option displays only column 1.

Here is another example of the `comm` command:

```
$comm -13 file1 file2
```

The above command will display only the second column. Table 2.19 describes the options used with the `comm` command.

2.10.10 `cmp` (Comparing Two Files for Differences Only)

With the help of the `cmp` command, the user can compare two files. If the files are different then UNIX will show the character and line numbers at which the two files differ. Let us consider the following output of a `cmp` command:

```
$cmp file1 file2
```

```
file 1 file2 differ: char 34, line 1
$
```

The above output indicates that the two files – file1 and file2 – differ at character 34 of line1. If the two files are identical, the `cmp` command won't give the output.

2.10.11 `diff`

As you know, the `cmp` command is used to compare those two files which, we are sure, really have same contents. The `cmp` command is fast and will work with any file. But the `diff` command works only on the text files, and it is used when the files are expected to be somewhat different. The output of the `diff` command will be the actions that should be done in order to make the two files. `diff` will work like this:

```
$diff <from-file> <to-file>
```

`diff` compares the contents of the from-file and to-file. A file name of - (dash) stands for the text read from the standard input.

If from-file is a directory and to-file is not, `diff` compares the file in from-file whose file name is that of to-file, and vice versa. If both from-file and to-file are directories, `diff` compares corresponding files in both directories in alphabetical order. `diff` options begin with -, so normally from-file and to-file may not begin with -. However, - as an argument by itself treats the remaining arguments as file names even if they begin with -.

The following example shows the difference between two files oldfile and newfile:

```
$cat oldfile
This is a file
$cat newfile
This is a file
Which is created to check the diff command.
$diff oldfile newfile
2c2
< Which is created to check diff command
> Which is created to check the diff command
```

This says that line 2 of the old file has to be changed to line 2 of the new file.

2.10.12 tr

The `tr` command copies the standard input to the standard output with substitution or deletion of selected characters. Input characters in `set1` are mapped to corresponding characters in `set2`. The `Utility` command performs classic *alphabet1* to *alphabet2* type of translation sometimes called 1:1 transliteration and as such suitable for implementation of caesar cipher. The `tr` command helps us to replace the specified characters with other characters in short transliterate characters that is why the name `tr`.

The `tr` command syntax is:

```
tr<options> <expression 1> <expression 2> <standard input>
```

When invoked without options, the `tr` command translates each character in *expression* 1 to its mapped: counterpart in *expression* 2.

Let us have a look at some examples of the *tr* command

1. `$tr A-z. a-z`

   This command converts uppercase to lowercase.

   The `tr` command does not accept a filename as argument, but takes its input through redirection from a pipe (i.e. standard input only).

2. `$tr 'r '-'< emp.lst`

   This replaces `all` **with**

   The lengths of the two expressions must be equal.

   Some options of the *tr* command are

   -d – Deletes specified range of characters.
   -s – Squeezes multiple occurrences of a character into a single word.

2.10.13 tar

The `tar` command is used to create tape archives and add or extract files. The name of the `tar` command is short for tape archiving, the storing of entire file systems onto magnetic tape, which is one use for the command. However, a more common use of `tar` is to simply combine a few files into a single file for easy storage and distribution.

To combine multiple files and/or directories into a single file, use the following command:

```
$tar -cvf file.tar inputfile1 inputfile2
```

Replace `inputfile1` and `inputfile2` with the files and/or directories you want to combine. You can use any name in place of `file.tar`, though you should keep the `.tar` extension. If you don't use the `-f` option, `tar` assumes you really **do** want to create a tape archive instead of joining up a number of files. The `-v` option tells `tar` to be verbose, which reports all files as they are added.

A `tar` file is not a compressed file; it is actually a collection of files within a single file that are not compressed. If the file is a `.tar.gz` (tarball) or `.tgz` file, it is a collection of files that are compressed.

2.10.14 cpio

This command is used to copy files into and out of archive storage and directories.

If you redirect the output from the `cpio` command to a special file (device), you should redirect it to the raw device and not the block device. Because writing to a block device is done asynchronously, and there is no way to know whether the end of the device is reached. Given below are two features of the *`cpio`* command:

1. The `cpio` command is not enabled for files greater than 2 giga in size due to limitations imposed by XPG/4 and POSIX.2 standards.
2. The `cpio` command does not preserve the sparse nature of any file that is sparsely allocated. Any file that was originally sparse before the restoration will have all space allocated within the file system for the size of the file.

The `cpio -o` command reads file pathnames from the standard input and copies these files to the standard output along with pathnames and status information. Avoid giving the `cpio` command pathnames made up of many uniquely linked files as it `(cpio)` may not have enough memory to keep track of them and would lose linking information.

2.11 vi Editor

UNIX has a number of editors that can process the contents of text files, whether those files contain data, source code, or sentences. There are line editors, such as `ed` and `ex`, which display a line of the file on the screen; and there are screen editors, such as `vi` and `emacs`, which display a part of the file on your terminal screen.

`vi` is the most useful standard text editor on your system. (`vi` is short for *vi*sual editor and is pronounced 'vee-eye.') Unlike `emacs`, it is available in nearly identical form on almost every UNIX system. The same might be said of `ed` and `ex`, but screen editors are generally much easier to use. With a screen editor you can scroll the page, move the cursor, delete lines, insert characters, and more, while seeing the results of your edits as you make them. Screen editors are very popular because they allow you to make changes as you read through a file, much as you would edit a printed copy, only faster.

To many beginners, `vi` looks unintuitive and cumbersome – instead of using special control keys for word processing functions and just letting you type normally, it uses all of the regular keyboard keys for issuing commands. When the keyboard keys are issuing commands, `vi` is said to be in command mode.

You must be in a special insert mode before you can type actual text on the screen. In addition, there seem to be so many commands.

While handling the `vi` editor, you have to keep many important things in mind. The vi editor makes use of terminal characteristics that are defined in a separate control file `/etc/termcap`. The terminal type that is used is made available to the `vi` editor by the environment variable TERM. Using this name, it searches either the file `/etc/termcap` or respective terminal file in `/usr/lib/terminfo` to select the terminal characteristics.

Getting started with `vi` editor: The vi editor is invoked by pressing `vi` at the shell prompt as shown below:

```
$vi <filename>
```

When you type this command, a screen will appear on which each line begins with a ~ (tilde) symbol, and at the bottom of the screen the following a message will appear

```
"filename"[New file]
```

When you open a file in the `vi` editor, the cursor is placed at the left most corner of the first line. Now you can't type anything until you press [*Esc*] and then *i*. That is, you have to change the mode to input mode. Remember, when we open the `vi` editor, we are in the command mode, and when we press [Esc] and then *i* we are in input mode.

2.11.1 Adding and Replacing Text in Input Mode

When you open the `vi` editor you have to press [Esc] to invoke input mode and then press i to insert any text at the current cursor position

You can use `vi` to edit any text file. *`vi`* copies the file to be edited into a buffer (an area temporarily set aside in memory), displays the buffer (though you can see only one screenful at a time), and lets you add, delete and change text. When you save your edits, `vi` copies the edited buffer back into a permanent file, replacing the old file of the same name. Remember that you are always working on a copy of your file in the buffer, and that your edits will not affect your original file until you save the buffer. Saving your edits is also called 'writing the buffer', or more commonly 'writing your file'.

General startup of `vi` editor: Let us see how to start work in `vi` editor. Given below are the steps for navigating through `vi` editor.

To use `vi`: `vi` <filename>
To exit `vi` and save changes: ZZ or :wq
To exit `vi` without saving changes: :q!
To enter `vi` command mode: [*esc*]

To move in the `vi` editor we will use the following keys:

Cursor movement

h	–	Moves left (backspace).
j	–	Moves down.
k	–	Moves up.

l	– Moves right (spacebar).
[return]	– Moves to the beginning of the next line.
$	– Goes to last column on the current line.
0	– Moves cursor to the first column on the current line.
^	– Moves to first nonblank column on the current line.
w	– Moves to the beginning of the next word or punctuation mark.
W	– Moves past the next space.
b	– Moves to the beginning of the previous word or punctuation mark.
B	– Moves to the beginning of the previous word, ignores punctuation.
e	– Moves to the end of next word or punctuation mark.
E	– Moves to the end of next word, ignoring punctuation.
H	– Moves to the top of the screen.
M	– Moves to the middle of the screen.
L	– Moves to the bottom of the screen.

There may be the possibility that you may be requiring to freeze some lines and want to move to a specific area of the screen. For movement in the `vi` editor screen we use the following keys:

Screen movement

G	– Moves to the last line in the file.
xG	– Moves to line x.
z+	– Moves current line to top of screen.
z	– Moves current line to the middle of screen.
z-	– Moves current line to the bottom of screen.
^F	– Moves forward one screen.
^B	– Moves backward one line.
^D	– Moves forward one half screen.
^U	– Moves backward one half screen.
^R	– Redraws screen (does not work with VT100 type terminals).
^L	– Redraws screen (does not work with Televideo terminals).

The following keys are used to insert/delete/copy the character at a particular location of the vi screen.

Inserting

r	– Replaces character under cursor with next character typed.
R	– Keeps replacing character until [*esc*] is hit.
i	– Inserts before the cursor.
a	– Appends after the cursor.
A	– Appends at the end of line.
O	– Opens line above the cursor and enters the append mode.

Deleting

x	– Deletes the character under the cursor.
dd	– Deletes the line under the cursor.

dw	– Deletes the word under the cursor.
db	– Deletes the word before the cursor.

Copying code

yy (Yank)	– Copies line which may then be put by the `p` (put) command. Precedes with a count for multiple lines.

Put command

This command brings back previous deletion or yank of lines, words, or characters. Its options are:

P	– Brings back before the cursor.
p	– Brings back after the cursor.

Find command

?	– Finds a word going backward.
/	– Finds a word going forward.
f	– Finds a character on the line under the cursor going forward.
F	– Finds a character on the line under the cursor going backward.
t	– Finds a character on the current line going forward and stops one character before it.
T	– Finds a character on the current line going backward and stops one character before it.
;	– Repeats last f, F, t, T.

Miscellaneous commands

.	– Repeats the last command.
u	– Undoes the last command issued.
U	– Undoes all the commands on one line.
xp	– Deletes the first character and inserts after the second (swap).
J	– Joins the current line with the next line.
^G	– Displays current line number.
%	– If at one parenthesis, will jump to its mate.
mx	– Marks the current line with character x.
‘x	– Finds the line marked with character x.

Note: Marks are internal and not written to the file.

Line editor mode

Any commands from the line editor `ex` can be issued upon entering line mode.

To enter: type ‘:’
To exit: press [*return*] or [*esc*]

`ex` Commands

Reading files

The use of `:r filename` gives us the option to read the file being edited after its editing. For this we will have to type on the vi editor screen `:r filename` at a particular location.

Write file

:w	–	Saves the current file without quitting.

Moving

:#	–	Moves to line #.
:$	–	Moves to the last line of file.

2.12 System Administration

The System Administrator is responsible for the smooth operation of a system. His task ranges from maintaining security, managing floppies and disks, performing backups, starting up and shutting down the system, installing and maintaining software packages, maintaining security by keeping log system etc. Let us go into some detail of these functions.

1. **Monitoring disk free space**
   Disk monitoring is a function performed by the system administrator. If the free disk space is less, then the system slows down. So the system administrator should identify and remove unnecessary files, and allot much more space to those users who work more on the system. The `df` (disk free) command reports the amount of free space available on the system. Here is an example of the `df` command:

   ```
   #df
   /(/dev/dsk/Os 1) : 60541 11225
   ```

   # is the prompt used by the system administrator when he logs in as root.

   /dev/dsk/Os1 is the logical filename of the hard disk.

   The disk has 60541 free disk spaces each consisting of 512 bytes.

   The use of the `-t` option with `df` will give the above output as well as the total amount of disk space in the file system.

2. **Monitoring disk usage (du)**
   This command without any arguments reports the space used by the current directory and all its sub-directories.
   In the following `du` command example, first we are checking the present working directory and after that we are looking for the disk usage with the help of the `du` command:

   ```
   #pwd

   /usr/kumar

   #du
   1433
   14
   503
   3788
   #
   ```

This command tells us the space consumed by HOME directory (`/usr/kumar`) and its three subdirectories (`HOME`, `usr`, `kumar`)

To get the disk usage of a specific directory, for example `/usr` the command is

```
#du/usr
```

2.13 Summary

This chapter gives the reader an insight into various UNIX commands. It explains the difference between external and internal commands. All the commands are classified according to their functionality and usage. The chapter also contains tips for the efficient use of the `vi` editor. The next chapter will explain the concept of shell programming in the UNIX environment.

3
Shell Scripts

The shell is a command programming language that provides an interface to the UNIX operating system. Its features include control-flow primitives, parameter passing, variables and string substitution. Constructs such as while, if-then-else, case and for are available. Two-way communication is possible between the shell and the commands. Stringvalued parameters, typically filenames or flags, may be passed to a command. A return code is set by commands that may be used to determine control-flow, and the standard output from a command may be used as the shell input.

3.1 A Peek into Shell Activities and Shell Programming

As you know, UNIX shell is the interface between the operating system and the user. The shell is a command line interpreter (CLI). It interprets the commands the user types in and arranges for their execution. However, it doesn't execute the commands itself. When a command is issued, the shell interprets the command and passes it to the kernel and waits for the command execution. The commands are themselves programs: when they terminate, the shell gives the user another prompt.

We can also say that 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 the shell. The adept users can customize their own shell and can use different shells on the same machine.

Shell as command processor: Let us understand the working of a shell. The shell issues the $ prompt and goes into sleep mode until you wake it up by issuing a command at the $ prompt. The shell scans your command for metacharacters like * and creates a simplified command line by expanding metacharacter abbreviations. Then it passes the simplified command to the kernel for execution. After the command execution, the prompt reappears and the shell again goes into sleep mode waiting for the next command to be entered by you.

There are several shells available in UNIX. These are described below:

1. **Bourne shell (sh):** This is the original UNIX shell written by Steve Bourne of Bell Labs. It is available on all UNIX systems. The Bourne shell does provide an easy-to-use language with which you can write shell scripts. This shell does not have the interactive facilities provided by modern shells such as the C shell and the Korn shell. You are advised to use another shell which has these features.
2. **C shell (csh):** This shell was written at the University of California, Berkeley. It provides a C-like language for writing shell scripts, hence its name.
3. **TC shell (tcsh):** This shell is available in the public domain. It provides all the features of the C shell together with emacs style editing of the command line.
4. **Korn shell (ksh):** This shell was written by David Korn of Bell Labs. It is now provided as the standard shell on UNIX systems. The Korn shell has all the features of the C and TC shells together with a shell programming language similar to that of the original Bourne shell. It is the most efficient shell. Consider using this as your standard interactive shell.

5. **Bourne again shell (bash):** This is a public domain shell written by the Free Software Foundation under their GNU initiative. Ultimately it is intended to be a full implementation of the IEEE Posix Shell and Tools specification. This shell is widely used within the academic community. It provides all the interactive features of the C shell (csh) and the Korn shell (ksh). Its programming language is compatible with the Bourne shell (sh). If you use the Bourne shell (sh) for shell programming, consider using bash as your complete shell environment.

3.1.1 Shell Programming

In addition to command interpretation, the ability to support programming is another main feature of the UNIX shell. The scope of shell programming often goes beyond the limits of conventional languages. With shells, programs can be developed as in any high-level language (HLL). Inside a shell program, you can use the entire set of shell's internal commands apart from some external commands like `tr, grep,` etc. The shell programming is so powerful that it can even handle important system procedures like system shutdown. The only drawback of shell programs is that they run slower than those developed in HLL.

A shell program is a series of UNIX commands. Instead of typing the commands one after the other, the commands may be written in a file, and the file can be executed to get individual command results. The shell programming offers much more versatility than individual UNIX commands. Using shell script with UNIX commands, one can develop an entire system. A shell script can be executed by giving execute permissions (`$chmod +x shell`) or by shell command `($sh shell).`

Shell scripts are used to accomplish a variety of tasks like:

1. Customizing the working environment.
2. Automating the daily tasks like backups.
3. Automating repetitive tasks like compilation.
4. Executing important system procedures like shutdown.
5. When the user logins into the system, UNIX starts a shell (sh) on that terminal which accepts commands from '$' prompt.
6. When a shell script is given for execution, the shell executes the commands from the shell script.

3.2 Shell Working

Your login shell reads its standard input from your terminal, and sends its standard output and standard error back to your terminal unless you tell it to send them elsewhere. The shell is line oriented; it does not process your commands until you press <Enter> to indicate the end of a line. You can correct your typing as you go. Different shells provide different facilities for editing your commands, but they generally recognize <Bksp> or <Del> as the keystroke to delete the previous character.

When you press <Enter>, the shell interprets the line you have entered before it executes the commands on that line. The steps it runs through are as follows:

1. The shell splits the line into tokens. A token is a command, variable or other symbol recognized by the shell. It continues to build up a sequence of tokens until it comes to a reserved word (a shell internal command that governs the flow of control of a shell script), function name or operator (a symbol

denoting a pipe, a logical condition, a command separator or some other operation that cannot be carried out until the preceding command is evaluated).

2. The shell organizes the tokens into three categories:
   - **I/O redirection:** In this category those types of commands come into picture that determine where the input or the output of a program is directed. For example, in the following command line, the text '>firstfile' is interpreted as an output redirection, which is later applied to the preceding command:

     ```
     $ls -al >firstfile
     ```

   - **Variable assignment:** This category constitutes commands that assign a value to a variable.
   - **Miscellaneous commands:** These are tokens that are checked to see if they are aliases. The first word is checked. If it is an alias, it is replaced by the original meaning of the alias; if it is not an alias, or if it is followed by a whitespace character before the next word, the process of alias checking is repeated until no more words remain (or until an alias has been detected that is not followed by a space).
3. After the commands' categorization, they may be executed either as internal shell commands (that cause the shell itself to take some action) or as external programs (if the shell can locate an executable file of that name).

3.2.1 Wildcard Characters

To work efficiently in the shell one should know the importance of wildcard characters because the shell works with several files and text in a file. To manipulate and search files using different commands, UNIX offers the facility of listing a set of files that have something common in their names without specifying their individual names. To list the files, characters like *, ?, [ ] can be used. These characters are known as wildcard characters. As stated above, these characters are used in place of filenames or directory names. Following are the search functions discharged by the wildcard characters:

*	An asterisk matches any number of characters in a filename, including none.
?	The question mark matches any single character.
[ ]	Brackets enclose a set of characters, any one of which may match a single character at that position.
-	A hyphen used within [ ] denotes a range of characters.
~	A tilde at the beginning of a word expands to the name of your home directory. If you append another user's login name to the character, it refers to that user's home directory.
!	This symbol is called bang. When it is placed at the beginning of a class, it matches all the characters except the ones in the class. This is very useful wild card character whenever we are using `grep` based commands where we have to match pattern based on except class.

Explained below are some wildcard character-related commands:

1. *** :** As you know, this wildcard character matches any number of characters including none. The following is a command using * (asterisk):

```
$cat c*
```

The above command will display the contents of the files that begin with letter *c* (rest be anything).

Similarly, the command

```
$cat c*.c
```

will display the contents of the files that begin with *c* and have an extension *.c* (rest be anything or none).

2. **? :** As this wildcard character matches a single character, therefore the command

   ```
   $ls ab??
   ```

   will display the names of the files that start with *ab* as the third character and the extension may be any thing but only a single character.
3. **[ ]:** Given below is the command using wildcard character [ ] that restricts the characters to be matched:

   ```
   $cat a[ijk]
   ```

   The above command will display the contents of the files whose names start with *a* and they have letters *i* or *j* or *k* after *a*.

3.3 echo Command

The echo command echos a string to the terminal. One use of this command is in determining the contents of environment variables. Environment variables are variables that UNIX keeps track of at the shell level. In the following example, we are using the echo command to display the message good morning. The echo command displays its argument on the screen.

```
$echo good morning
good morning
```

Cursor management in echo: It is essential to understand how to manage the cursor while using the echo command. It will help the user to get the echo command response/output at a particular place. The following cursor management options work in echo:

\c	–	To position the cursor immediately after the argument.
\t	–	To set tabs.
\f	–	To form feeds.
\n	–	To start new line.

In the following example, the echo command will set the tab space and start in the new line, and the cursor will be placed after the message Enter the name.

```
$ echo' \t Enter name \n on this line\c'
Enter the name
{cursor waits}.
```

3.4 Special Files

Before going to the basics of shell scripts, it is important to know the different categories of shell-based special files. Special files are associated with input/output devices. Special files are also known as *device files*. In UNIX all physical devices are accessed via device files; these are used by programs to communicate with hardware. These files hold information on location, type and access mode for a specific device. Explained below are some special files used in UNIX for special purposes:

/dev/null: This incinerates the output directed towards it and is useful in redirecting error messages away from the terminal. No physical device is associated with this file. Size of this special file is always zero.

/dev/tty: This file indicates one's own terminal. Suppose kumar is working on the terminal/`dev/tty01`. Suppose kumar issues the command

```
$who>/dev/tty
```

Following the execution of the above command, the special file/dev/tty sends the list of current users to kumar's terminal.

3.5 Command Substitution

The shell enables the argument of a command to be obtained from the output of another command. The standard output from a command can be substituted in a similar way as parameters. The `pwd` command prints on its standard output the name of the current directory. For example, if the current directory is `/usr/kumar/bin` then the command

```
d = "pwd"
```

is equivalent to

```
d = /usr/kumar/bin
```

The entire string between grave accents ('...') is taken as the command to be executed and is replaced with the output from the command. The command is written using the usual quoting conventions except that a' must be escaped using a \ . For example,

```
$ls 'echo "$1"'
```

is equivalent to

```
$ls $1
```

Command substitution occurs in all contexts where parameter substitution occurs and the treatment of the resulting text is the same in both the cases. This mechanism allows string-processing commands to be used within shell procedures. An example of such a command is `basename` which removes a specified suffix from a string. For example, the command

```
basename main.c .c
```

will print the string `main`.

When a command is enclosed in single quotes the shell executes the command first, and the enclosed command text is replaced by the command output. Let us understand this execution through the following example:

```
$ echo The date today is 'date'
```

The output would be:

```
The date today is Tue Sep 19 11:39:25 IST 2006
```

In the above command, the displayed message is: `The date today is 'date'`. During execution of this command, the word 'date' in quotes will be treated as the `date` command. Hence the output of the `date` command (i.e. Tue Sep 19 11:39:25 IST 2006) replaces 'date'.

3.6 Shell Variables

Standard UNIX variables are split into two categories, environment or system variables and shell variables. In broad terms, shell variables apply only to the current instance of the shell and are used in set short-term working conditions; environment variables have a far-reaching significance, and those set at login are valid for the duration of the session. By convention, environment variables have uppercase names and shell variables have lowercase names.

Programs look 'in the environment' for particular variables, and if found their stored values are used by the programs. Some values are set by the system, others by you, yet others by the shell, or any program that loads another program. There are two types of shell variables: one, those created and maintained by UNIX, and, two, those created by the user.

The shell provides string-valued variables. Variable names begin with a letter and consist of letters, digits and underscores. Variables may be given values by writing; for example, variables `user, name` and `acno` may be given values as shown below:

```
user=kumar

name=saurabh

acno=a001
```

The value of a variable is substituted by preceding its name with $; for example,

```
echo $user
```

will echo `kumar`.

```
$x=1
```

creates a variable *x* having been assigned a value 1.

```
$echo $x
```

will print the value of variable *x*.

Actually values of variables are stored in ASCII form. When the shell encounters a variable preceded by $, the variable is converted to its corresponding value.

System or environment variables

As already stated, apart from the variables created by the user, there are some other variables that are set by the system. Some of them are set during the booting process and some after the user logs in. These are called system variables. To have a look at the system variables use the `set` command. The output of the `set` command is as follows:

```
$set

HOME =/usr/kumar
IFS=
MAIL =/usr/spool/mail/kumar
PATH=/bin :/usr/bin
PS1=$
PS2= > PWD=/usr/kumar/progs
SHELL= /bin/sh
TERM=ansi
```

Here all the system variables can be seen in uppercase letters. To avoid confusion between the user-defined and system variables, one should write the user-defined (shell) variables in lowercase.

Standard shell variable – PATH

This variable sets the search path for any executable command. In other words, PATH instructs the shell about the route it should follow to locate executable commands. The search is made in current working directory and system directories devoted to store files. In the PATH command, the delimiter between the list of directories is a semicolon (;). The directories, usually searched for this purpose, are `/bin` and `/usr/bin`. The PATH command first searches for the command given by the user at the current working directory. If not found there, it searches in the `/bin` directory, and if not found there also, it searches in the `/usr/bin` directory. If unable to find the user's command in one of these directories, the shell reports back that the command was not found.

Like user-defined variables PATH variable can also be changed if the user wants to add some more directories in which the shell should search for the executable command. For example, by changing the PATH variable in the following way, directory `/usr/kumar/progs` will also be searched after the current directory:

```
PATH =/bin:/usr/bin:./usr/kumar/progs
```

or

```
PATH =$PATH:/usr/kumar/progs
```

HOME variable is a shell environment variable. The user will be placed in HOME directory as soon as he logs in. This directory is controlled by the HOME variable. The `cd` command used without arguments resets the current directory to the directory assigned to HOME. The `cd` with `$HOME` as argument also takes you to the HOME directory.

Internal field separator (IFS) variable

Internal field seperator (IFS) variable is a shell variable used to give separation in a command line. The IFS list includes space characters, tab character and a new line character. All these characters are invisible, so they can't be seen in the output. You can change the space character with any character you like. By default the IFS between two or more command-based arguments is a blank space. We can change it to # by the following command:

```
$IFS = #
```

Now the separation (or delimiter) between two arguments will be the symbol # instead of a blank space.

The following example indicates the use of delimiter in UNIX. In this example, the delimiter between cat and temp.lst is not blank space, but it is set to #:

```
$cat#temp.1st
```

MAIL command

This command determines in which directory the MAIL addressed to the user is to be stored. This directory is searched every time the user logs in. If any information is found, the shell informs the user with a message *You have a mail.*

PS1 and PS2 variables

PS1 (Prompt string 1) is the symbol used as your prompt. Normally it is set to be $. But the user can redefine the prompt by assigning a new value. For example,

```
$PS1= #
```

sets the prompt as # and

```
$PS1 = #
#
```

resets the prompt back to $

PS2 (Prompt string 2) is used as prompt when UNIX thinks the user has started a new line without finishing a command.

Shell terminal

Shell variable determines the type of shell that the user will log in. The user can choose any of the shells he wants to work with. If you type `csh` you can switch over to Cshell and `ksh` to Kornshell..

TERM variable

This variable identifies the kind of terminal you use. Some utilities, like `vi` editor, require to know the type of terminal. If `term` is not defined properly, `vi` editor won't work. When variables are used they are referred to with the $ symbol in front of them.

Special character variables: There are several useful variables available in the shell program. Some of them are described below:

1. $$: The PID number of the process executing the shell.
2. $?: Exit status variable.
3. $0: The name of the command you used to call a program.
4. $1: The first argument on the command line.
5. $2: The second argument on the command line.

6. $n: The nth argument on the command line.
7. $*: All the arguments on the command line.
8. $#: The number of command line arguments.

3.7 `read` Statement

This is shell's internal tool for taking input from the standard input. It is similar to `scanf()` in C and `cin>>` function in C++. The `read` statement makes the script more user-friendly. It can be used with one or more variables to make shell scripts interactive. In the `read` command when the user enters a value through keyboard that value is stored in the corresponding variable.

Two basic words in shell script are `read` and `echo`

1. `read` accepts input.
2. `echo` writes the output.

In the following example, using `echo` and `read` we are displaying a message and then reading the name in the variable 'name'.

```
echo Enter your name \?
read name
echo Good morning $name
```

```
$chmod 755 sh01
$sh01
```

After execution we will get

```
Enter your name ?
Vijay
Good morning Vijay
```

3.8 Quoting

Quoting is used to accomplish the following two goals:

1. To control (i.e. limit) substitutions of commands.
2. To perform grouping of words.

Single and double quotes: The shell recognizes both single and double quote characters. The following two examples with single and double quote characters are equivalent:

```
var='this is first UNIX session'
var="this is first UNIX session"
```

However, there is an important difference between single and double quotes. Single quotes limit substitution. You can place variables in double quoted text and the shell still performs substitution. We can see this with the echo command:

```
$echo "My host name is $HOSTNAME."
```

The output is:

```
My host name is localhost.
```

If we change to single quotes, the behaviour changes:

```
$echo 'My host name is $HOSTNAME.'
```

The output is:

```
My host name is $HOSTNAME.
```

Quoting a single character: There is another quoting character you will encounter. It is the backslash. The backslash tells the shell to 'ignore the next character' Here is an example:

```
$echo "My host name is \$HOSTNAME."
```

The output is:

```
My host name is $HOSTNAME.
```

3.9 Positional Parameters and Command Line Arguments

The shell script can also accept arguments from the command line, that is, when you execute the program itself. To handle options on the command line, we use a facility in the shell called positional parameters. Positional parameters are a series of special variables (`$0` through `$9`) that contain the contents of the command line. The total number of parameters are stored in `$#`. The first argument is read by the shell into a special variable `$1`, the second argument into `$2`, and so on. Let us consider some cases where we will require command line arguments:

1. When you would like to specify the name of the output file on the command line, as well as set a default output filename if no name is specified.
2. When you would like to offer an interactive mode that will prompt for a filename and warn the user if the file exists and prompt the user to overwrite it.
3. When you would like to have a help option that will display a usage message.

Let us consider the following command line:

```
$some_program word1 word2 word3
```

If some_program were a bash shell script, we could read each item on the command line because the positional parameters contain the following:

1. `$0` would contain "some_program".
2. `$1` would contain "word1".
3. `$2` would contain "word2".
4. `$3` would contain "word3".

Here is a script you can use to try this out:

```
echo "Positional Parameters"
echo '$0 = '$0
echo '$1 = '$1
echo '$2 = '$2
echo '$3 = '$3
```

The above script will print the values of positional parameters for `$0, $1, $2 and $3`.

3.10 Arithmetic in Shell

The arithmetic command takes an arithmetic expression, such as sum or product, evaluates it and gives the result. It can also manipulate strings to a limited extent.

We cannot directly perform arithmetic operations in shell scripting. In the following examples of arithmetic command, we are performing basic arithmetic operations. `expr` is the keyword for doing arithmetic in shell scripts. We can use all mathematical symbols except multiplication symbol (*) as it is used as wild card character. Therefore a multiplication symbol should be preceded by '\' to give this symbol the actual meaning of multiplication as symbol * is metacharacter. To give comments we use '#' sign. Anything after # will be treated as comment. Remember, the terms in `expr` should be separated by space. Parenthesis may be used for clarity of expression. We should use a pair of parenthesis. `expr` can handle only integers. Use `bc` to handle real numbers.

Write the following script in a file and execute:

```
a=10 b=4
echo 'expr $a + $b'
echo 'expr $a - $b'
echo 'expr $a \* $b'
echo 'expr $a / $b'
echo 'expr $a % $b'    #modulus
```

On execution of the above script, the output is:

```
14
6
40
2
2
```

In the following example, the values of `a` and `b` are real numbers. As shell scripts support only integers, we are using `bc` to compute the result:

```
a=1.5 b=2.5
'echo $a + $b | bc'
```

3.11 Exit Status of a Command

A properly written UNIX application will tell the operating system if it was successful or not. It does this by means of an exit status (`$?`). Every command in UNIX returns a value after execution. This value is called the exit status or return value. The exit status is a numeric value in the range of 0 to 255. If the command is executed correctly, the exit status value is 0. If something goes wrong, the exit status is nonzero. The actual value will depend on the type of the problem occurred. The exit status is the status of the last command.

The exit status provides two important features. First, it can be used to detect and handle errors, and, second, it can be used to perform true/false tests. The exit status is used by the programmers to devise program logic which branches into different paths, depending on the success or the failure of a command.

3.12 Decision Control Structure

Now let us learn how to add intelligence to our scripts. So far, our script has only consisted of a sequence of commands that starts at the first line and continues line by line until it reaches the end. Most programs do more than this. They make decisions and perform different actions depending on 'if then else' conditions. These are called decision control structures or conditional commands. Let us understand some of these conditional commands described below:

```
if then fi
if then else fi
if then elif else fi
case - esac
```

3.12.1 The if Conditional

The `if` statement in UNIX is quite similar to the one found in high-level programming languages. The only difference between the two is that in the UNIX shell, the `if` statements are usually concerned with the status of commands and not directly with the logical and numerical conditions. The `if` statement syntax is

```
if control command
 then
<commands>
fi
```

First, the command following `if` is executed. If the command is successful (i.e. if exit status = 0), then the commands between `then` and `fi` are executed.

In the following `if` statement example, the script is looking for two arguments – `$1` and `$2` – to be entered by the users as filenames. If `$1` is exiting, it will copy the contents of file1 (`$1`) to file2 (`$2`) and will display the message `File copied successfully:`

```
if cp $1 $2
then
  echo File copied successfully
fi
```

3.12.2 The if then else Conditional

This conditional command takes either of the two possible decisions, depending on the fulfilment of a certain condition. Its syntax is

```
if      <condition is true>
```

```
then
   <execute commands>
else
   <execute commands>
fi
```

In the following example of `if then else` statement, the script is looking for two arguments – `$1` and `$2` – to be entered by the users as filenames. If `$1` is exiting, it will copy the contents of file1 (`$1`) to file2 (`$2`). It will display the message File copied successfully in case if then section is successful. Otherwise, it will display File copy failed in case `if else` section is successful.

```
if cp $1 $2
then
  echo File copied successfully
else
   echo File copy failed
fi
```

3.12.3 The if then elif else fi Conditional

This conditional command is used for multi-level decision making. Its syntax is

```
if <condition> then
   statements
elif <condition>
   statements
else
  statements
fi
```

3.12.4 The Test and [ ] Commands

The `if` construct depends on whether or not a command has a successful status. But the status of a command is not the only matter of interest to us or to a shell script. Other factors that are to be checked by the shell script are whether the command argument relates to a file or a directory? Did the user provide enough arguments? Did the user provide correct code word? And, so on.

All these questions are investigated by the `test` command. The values returned by the `test` statement are used by the `if` command for taking decisions.

Although the `test` command exists in every version of Linux and UNIX, there is an alias for it that is much more commonly used, the left bracket: `[`**.** Both test and its alias are usually found in `/usr/bin` or `/bin`, depending on the operating system version and vendor.

When you are using the left bracket instead of test, it must always be followed by a space, the condition to evaluate, a space and the right bracket. The right bracket is not an alias for anything but signifies the end of the parameters that need evaluation. The spaces around the condition are needed to signify that test is being summoned to distinguish this from a character/pattern-matching operation for which the brackets are also commonly used.

The syntax for `test` and `[` is as follows:

```
test expression
[ expression ]
```

In both cases, `test` evaluates an expression and returns true or false. If it is used in conjunction with the `if, while` or `until` command, you can have extensive control over the program flow. However, you do not need to use the `test` command with any other construct, and you can run it directly from the command line to check the status of virtually anything.

There are three tests, namely

1. String test.
2. Numerical test.
3. File test.

String test

As the title implies, this set of functions compares the value of strings. You can check to see if they exist, are the same or are different.

Described below are the string test-related flags:

`string`	Tests to see if the string is not null.
`-n string`	Tests to see if the string is not null; the string must be seen by test.
`-z string`	Tests to see if the string is null; the string must be seen by test.
`string1 = string2`	Tests to see if string1 is identical to string2.
`string1 != string2`	Tests to see if string1 is not identical to string2.

In the following example of string test, we are checking whether both the strings entered as argument values are same or not:

```
if [ $1 = $2 ] then
   echo Both strings are same
else
  echo $?
fi
```

Numerical test

Just as the string comparison operators verify whether strings are equal or different, the integer comparison operators perform the same functions for numbers. Expressions test as true if the variables match in value, and false if they do not. Integer comparison operators do not work with strings, just as the string operators do not work with numbers. Following are the numerical test syntaxes and their special meaning:

`int1 -eq int2`	True if `int1` is equal to `int2`.
`int1 -ge int2`	True if `int1` is greater than or equal to `int2`.
`int1 -gt int2`	True if `int1` is greater than `int2`.
`int1 -le int2`	True if `int1` is less than or equal to `int2`.
`int1 -lt int2`	True if `int1` is less than `int2`.
`int1 -ne int2`	True if `int1` is not equal to `int2`.

Given below are the expansions of the short terms used in numerical test syntaxes:

`-gt:`	Greater than.
`-lt:`	Less than.
`-ge:`	Greater than or equal to.
`-le:`	Less than or equal to.
`-ne:`	Not equal.
`-eq:`	Equal.

The following example shows a snippet of the code in which the value given on the command line must equal 7:

```
if [ $1 -eq 7 ]
then
  echo "You've entered the magic number."
else

  echo "You've entered the wrong number."

fi
```

In operation:

```
$script 6
You've entered the wrong number.
$
$script 7
You've entered the magic number.
$
```

As with strings, the values being compared can be variables assigned values outside of the script and don't always have to be provided on the command line.

We should take the following precautions before writing the test condition:

1. Use square braces to avoid writing word test in the command.
2. Provide a space after '['.
3. Provide a space before ']'.

File test

Like string and numerical tests, file test is also a special class of test. The following are the file `test` related flags

1. `-s` returns true if the file exists and size > 0.
2. `-f` returns true if the file exists and not directory.
3. `-d` return true if the file exists and is a directory.

The following examples show the use of the file `test` related flags:

`[ -f "somefile" ]`	:	Tests if `somefile` is a file.
`[ -x "/bin/ls" ]`	:	Tests if `/bin/ls` exists and is executable.
`[ -n "$var" ]`	:	Tests if the variable `$var` contains something.
`[ "$a" = "$b" ]`	:	Tests if the variables `"$a"` and `"$b"` are equal.

In the following example, we are checking that the file enter argument value is a file, not a directory and that it exists:

```
if [ -f $1 ] then
   echo File exists
fi
```

3.12.5 Logical Operators && and ||

The && operator (AND) is used by the shell in the same sense as it is used in C. The second command will be executed if and only if the first one is true. On the other hand, if the `||` (OR) operator is used, the command to the left of `||` is executed first. If its exit status is zero (success), the command to the right of `||` is ignored. If the exit status of the left command is non-zero (failure), the right command is executed, and the value of the entire expression represents the exit status of the right command.

We can use logical operators in the following compound comparison way also:

`-a`: logical AND

`exp1 -a exp2` returns true if both `exp1` and `exp2` are true.

`-c`: logical OR

`exp1 -o exp2` returns true if either `exp1` or `exp2` are true.

Here `-a` stands for AND condition `-o` stands for OR condition.

In the following example we are using the AND condition through `-a` option. If positional parameter `$1` is greater than 60 AND `$2` is less than 50, then other statements will work in this example:

```
if [ $1 -gt  60  -a $2 -lt 50 ] then
  statements ...
```

3.12.6 case Statement

The `case` statement lets a shell script choose from a list of alternatives. It is used for menu-based operations. The syntax of the `case` statement is as follows:

```
case value in
choice 1) statements;;
choice 2) statements;;
*) statements;;
esac
```

In the `case` statement, `choice1` and `choice2` are called labels that identify potential choices of action. When the `value` gets an argument `choice1`, the commands following `choice1` label are executed. In the `case` syntax, *) is the default choice. `esac` is the delimiter of `case`. In other words, it marks the end of the `case` statement. It is important to note that all choice statements should be terminated by double semicolons (;;).

The following example indicates the use of the `case` statement:

```
echo "enter the choice"
echo "1 date"
echo “2 who”
echo “3 ls”
echo “4 pwd”
read choice
case $choice in
1) date;; .
2) who;;
3) ls;;
4) pwd;;
*) echo “Not one of the choices”
esac
```

In the above code, when the user enters his choice, the command with respect to that choice (`date or who or ls or pwd`) will be executed. The double semicolons separate one choice from the next. In this shell script we are not coming out of the script, therefore we will modify the same script by introducing the `exit` command option in the default choice. The new script with a slight modification of the `exit` command is given below.

```
echo “enter the choice”
echo “1 Display date”
echo “2 who”
echo “3 ls”
echo “4 pwd”
echo “5 exit”
read choice
case $choice in
1) date;;
2) who;;
3) ls;;
4) pwd;;
5) exit;;
*) echo “Not one of the choices”
esac
```

In the above example, if the users do not want to try any commands, they could just type 5 and exit from the program.

3.13 Loop Controls Structures

Computer can repeat a particular instruction again and again until a particular condition is satisfied. A group of instructions that are executed repeatedly is called a loop. In other words, looping is the facility to repeat a set of statements till a certain condition is met. Most languages have the concept of loops. If we want to repeat a task 20 times but don't want to type in the code 20 times, with maybe a slight change each time, then we can use `for` and `while` loops in the Bourne shell.

The loop controls are

1. `while` loop.
2. `for` loop.
3. `until` loop.

3.13.1 `while` Statement

In the `while` statement the control flow depends on the exit status of the command.

A `while` loop has the following syntax:

```
while <control command-list1>
do
   <command-list2>
done
```

The value tested by the `while` command is the exit status of the last simple command following `while`. Each time round the loop `command-list1` is executed; if a zero exit status is returned then `command-list2` is executed; otherwise, the loop terminates. During each cycle, the shell checks the condition (command list 1) and if its exit status is zero (success), the commands between `do` and `done` are executed. This process continues till the exit status becomes non-zero (failure). The loop will continue so long as the condition is true. When the condition is false, the next command after `done` will be executed. `Done` statement is the delimiter of `do` statement.

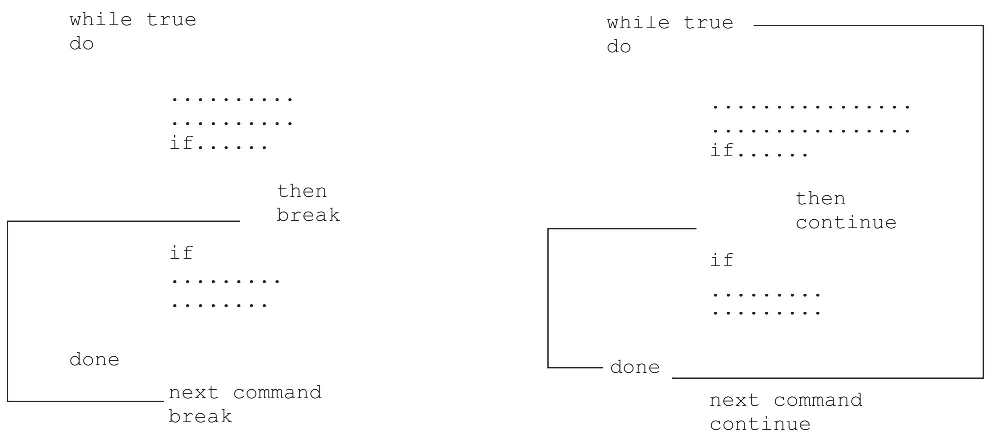
It is important to note that the condition in the `while` loop can be any valid UNIX command. The `while` condition can be simple or complex. The condition should have an exit status. Otherwise, it may go into infinite loop. The test statement can be used here also, with its associated expressions, numeric and string comparisons and file tests.

Let us understand the following example of the `while` loop.

```
count = 1
while [ $count -le 3 ]
do
  echo Loop value $count
  count = 'expr $count + 1'
done
```

In the above example, variable count is initialized to 1 and it is incremented by 1 till count value is less than or equal to 3.

break and **continue:** The `break` command is a specialized version of the `goto` command of many languages. It is used to break the current loop and come out of it. It is usually associated with the `if` statement. The `continue` statement causes the flow to skip to the next cycle of the loop.



```
while true
do

        ..........
        ..........
        if......

                then
                break
        if
        .........
        ........

done
        next command
        break
```

```
while true
do

        ................
        ................
        if......

                then
                continue
        if

        .........
        .........

done
        next command
        continue
```

Example

```
if [ $1 -eq 5 ]
then
  I= 2
  break
fi
```

To take the control to the beginning of the loop by passing the statements

```
I = 1
while [ $I -le 5 ] then
do
  I = 'expr $I 1 1'
  continue
done
```

3.13.2 `until` Statement

The `while` loop runs till the command fails and the `until` statement runs till the command succeeds. The `until` loop waits until some external event occurs and then runs some commands. In an `until` loop the termination condition is just reverse of the one used in the `while` loop. The `until` syntax is as follows:

```
until <control command-list1>
do
   <command-list2>
done
```

Here the `control` command is executed. If it fails, the commands between `do` and `done` are executed and the `control` command is attempted again. This process is repeated till the `control` command succeeds.

In the following example, the `until` loop will be executed until `file` exists. Each time round the loop waits for 5 minutes before trying again.

```
until test -f file
do sleep 300
done
commands
```

3.13.3 `for` Statement

The `while` and `until` loops are sometimes termed as indefinite loops. This is because we don't know how many times the iteration will take place. This disadvantage can be eliminated by using the `for` loop. In this loop we can specify the number of iterations. A frequent use of shell procedures is to loop through the arguments ($1, $2, ...) executing commands once for each argument can be better handled by for statement.

The `for` loop notation is recognized by the shell and its syntax is

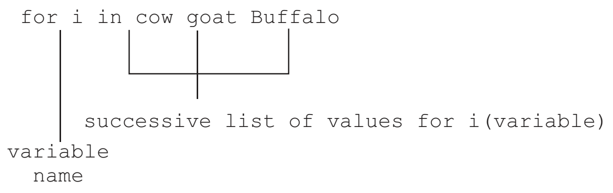
```
for name in w1 w2 ...
do
      <command-list>
done
```

A command-list in for loop is a sequence of one or more simple commands separated or terminated by a newline or semicolon. Furthermore, reserved words like `do` and `done` are only recognized following a newline or semicolon. `name` is a shell variable that is set to the words `w1 w2`, ... In turn, each time the command-list following do is executed. If `'in w1 w2 ...'` is omitted then the loop is executed once for each positional parameter; that is, in `$*` is assumed.

***for* loop – points to remember**

1. The list in `w1 w2` consists of a series of character strings, each one separated with white space.
2. Each item is assigned to the variable in turn, and the loop of the body is executed.
3. The loop is performed as many times as there are words in the list.

Now we will see the use of `for` loop through the following example. In this example, *i* is variable name and 'cow, goat and buffalo' are successive list of values for *i*.


for i in cow goat Buffalo
successive list of values for i(variable)
variable name

Sample code 1

```
for i in cow goat Buffalo
      do
```

```
        echo "The $i is grazing the field. "
        done
```

After the execution of the above code, the output will look like this

```
The cow is grazing the field.
The goat is grazing the field.
The Buffalo is grazing the field.
```

When the first time loop is executed, the variable gets the value 'cow', for the next iteration the variable receives the value 'goat', and for the next 'Buffalo'.

Sample code 2

Accepting values from the command line

```
for i $*
   do
   echo "The $i is grazing the field."
   done
```

After passing and execution of arguments like `name sam1.sh` in the above script in the following way

```
$sam1.sh cow goat buffalo
```

the output will be

```
The cow is grazing the field.
The goat is grazing the field.
The buffalo is grazing the field.
```

Here the variable `i` takes, one by one, each of the three arguments as its value.

Sample code 3

In the following example of the `for` loop, all the files having .c extension will be displayed one by one.

```
for file in *.c
do
cat $file
done
```

3.14 `set` Statement

The value of positional parameters and the parameters associated with $# and $? can't be set by making a direct assignment with the = operator. The `set` commands can be used for these assignments. This makes it possible to convert scripts arguments into positional parameters.

In the following example of the `set` command, the value 9876 will be assigned to positional parameter `$1`, 2345 to `$2` and 6213 to `$3`.

```
$ set 9876 2345 6213
```

Hence, if we are printing the values of positional parameters (`$1`, `$2` and `$3`), as shown in the following example, it will display the values that we have set previously. $# is printing no of parameters, that is, 3. `$*` will display all the positional parameters value at a time.

```
$echo $1
9876
$echo $2
2345
$echo $3
6213
$echo $#
 3
$echo $*
9876 2345 6213
```

3.15 `shift` Statement

The `shift` statement transfers the contents of the positional parameters to its immediate lower numbered one. In other words, the `shift` command can be used to shift command line arguments to the left, that is, `$1` becomes the value of `$2`, `$3` shifts into `$2`, etc. The command `shift 2` will shift two places meaning the new value of `$1` will be the old value of `$3`, and so forth.

3.16 Command Grouping

The shell uses two sets of operators for grouping commands. These are: () and {}.

Use of (): Suppose we have to send the current date and time along with the contents of a file called `results` to a second file, `result1`. Then the command would be

```
$(date cat results»result1)
```

Here the output of `date` followed by the output of `cat` are redirected to a file called `result1`. Had we used the command

```
$date; cat results >result1
```

the output of the `date` command would have been sent to the screen, and the output of the `cat` command redirected to the file `result1`.

Use of {}: Unlike (), the commands between {} are executed in the current shell itself. In the following example, since no separate environment was created, the change of directory will become permanent even after the execution of the command.

```
$pwd
/usr/kumar
$ {cd progs ; pwd}
$pwd
/usr/kumar/progs
```

3.17 dot (.) Command

This command is used to execute any program without creating a sub-shell. When a script name is preceded by a dot (.) the script is executed within the current shell.

The following example depicts the use of the `dot` command. It shows that a file whose name starts with a dot (.) doesn't require execution permission and it is also a hidden file.

```
$ .profile
```

However, dot is not a command as such. If a file is spelled with a dot at the beginning, UNIX treats it as a hidden file. Such files don't show up in a regular `ls` command, and you don't get rid of them with a `rm *` command. It is common for UNIX tools to create 'dot' files for temporary files and to maintain control and history information. Configuration files are often preceded by a dot.

3.18 Summary

This chapter highlights the use of shell scripts and the shell usage. It digs into the topic of shell variables. It talks about the decision control, loop control and checking various conditions in shell scripting. It discusses command line arguments and their use in shell scripts.

4

File and Directories Maintenance

The basic system object used to manipulate file is called file descriptor (fd). This is a non-negative integer that is used by various I/O system calls to access a memory area containing data about an open file. This memory area has a role similar to that of the file structure in the standard C library I/O functions, and thus the pointer returned from `fopen()` has a role similar to that of a file descriptor.

Each process has its own table of file descriptors, with each entry pointing to an entry in the system file descriptor table. File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file.

The UNIX kernel uses a three-table data structure to manage open file. This apparently complicated structure is actually very flexible and elegant, and allows open file to be shared among processes in a simple way. We can see in Figure 4.1 that each process entry in the process table contains a table of open file descriptors. The file descriptors index the entries in this table, and each entry contains a pointer to an entry in a table that the kernel maintains for all open file. Each entry in this open file' table contains the file status flags (read, write, append, etc.), the current file offset, and a pointer to the entry for this file in the so-called v-node table. This table (or part of it) is stored on the physical device.

This allows several processes to share file descriptors by having an entry of their table pointing to the same entry in the system file descriptors' table. The file descriptors index the entries in this table, and each entry contains a pointer to an entry in a table that the kernel maintains for all open file.

As already stated, the value of the file descriptor is a non-negative integer. Usually, three file descriptors – 0, 1, 2 – are automatically opened by the shell that started the process. File descriptor 0 is used for the standard input of the process. File descriptor 1 is used for the standard output of the process, and file descriptor 2 is used for the standard error of the process. Normally, the standard input gets input from the keyboard, while the standard output and the standard error write data to the terminal from which the process was started.

4.1 File Access Functions

In this section we will discuss some file functions that are used to perform various operations on the file. Let us begin with the `open()` function.

4.1.1 open Function

The `open()` function establishes the connection between a file and a file descriptor. It creates an open file description that refers to a file, and a file descriptor that refers to that open file description. The file descriptor is used by other I/O functions to refer to that file. The `path` argument points to a pathname naming

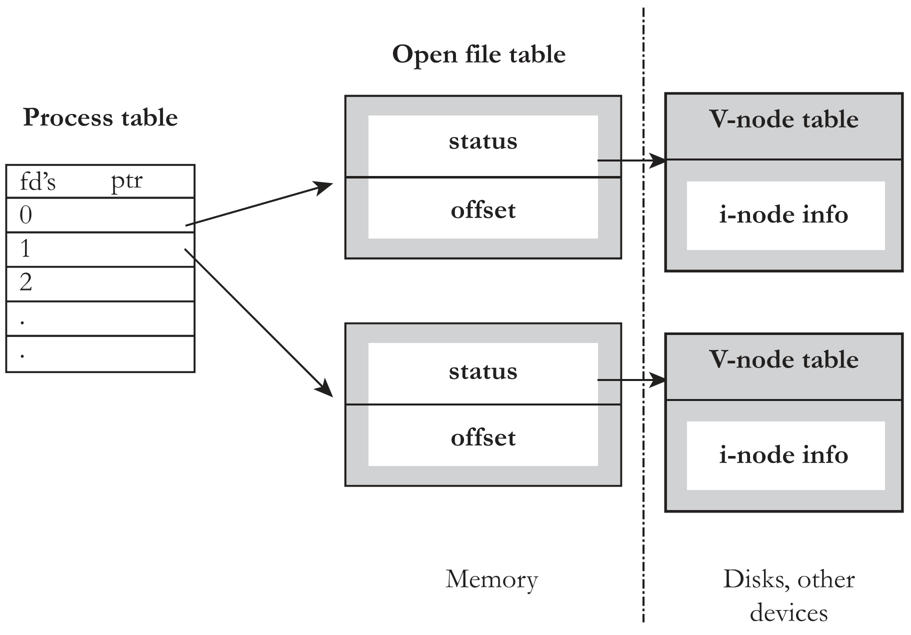


Figure 4. 1 Structure of file descriptors (fds) in kernel.

the file. The `open()` function returns a file descriptor for the named file that is the lowest file descriptor not currently open for that process. The open file description is new, and therefore the file descriptor does not share it with any other process in the system. The FD_CLOEXEC file descriptor flag associated with the new file descriptor is cleared.

The function action, header file required and syntax of the `open()` function are as follows:

Function action

`open` – opens a file.

Header file required

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

Syntax

```
int open(const char *path, int oflag, /* mode_t mode */...);
```

The file offset used to mark the current position within a file is set to the beginning of the file. The file status flags and file access modes of the open file description are set according to the value of `oflag`, that is actually a file offset. The `mode` argument is used only when O_CREAT is specified.

Values for `oflag` are constructed by a bitwise-inclusive-OR of flags from the following list, defined in <fcntl.h> header file. Applications must specify exactly one of the first three values (file access modes, i.e. O_RDONLY, O_WRONLY, O_RDWR) in the value of `oflag`:

Following are the flags used in the `open()` function. Flags other than the first three are optional in the `open()` function.

O_RDONLY

This flag is used to open a file for reading only.

O_WRONLY

This flag is used to open a file for writing only.

O_RDWR

This flag is used to open a file for reading and writing. The result is undefined if this flag is applied to a first in first out (FIFO). Any combination of the following first three flags may be used.

O_APPEND

If this flag is set, the file offset is set to the end of the file prior to each write.

O_CREAT

This flag is used to create the file if it does not exist. This flag requires that the `mode` argument be specified. If the file desired to be created already exists, this flag has no effect except under the O_EXCL flag as noted below. Otherwise, the file is created with the user ID of the file set to the effective user ID of the process. The group ID of the file is set to the effective group IDs of the process. Or, if the S_ISGID bit is set in the directory in which the file is being created, the file's group ID is set to the group ID of its parent directory. If the group ID of the new file does not match the effective group ID or the ID of one of the supplementary groups, the S_ISGID bit is cleared.

O_DSYNC

This flag is used to write I/O operations on the file descriptor to get completed as defined by synchronized I/O data integrity completion.

O_EXCL

If O_CREAT and O_EXCL flags are set, the `open()` function fails if the file exists. The check for the existence of the file and the creation of the file if it does not exist is atomic with respect to other processes executing `open()` naming the same filename in the same directory with O_EXCL and O_CREAT flags set. If the O_CREAT flag is not set, the effect is undefined.

O_LARGEFILE

If the O_LARGEFILE flag is set, the offset maximum in the open file description is the largest value that can be represented correctly in an object of type `off64_t`.

O_NOCTTY

If the O_NOCTTY flag is set and `path` identifies a terminal device, `open()` does not cause the terminal device to become the controlling terminal for the process.

O_NONBLOCK or O_NDELAY

These flags may affect subsequent reads and writes. If both O_NDELAY and O_NONBLOCK flags are set, the O_NONBLOCK flag takes precedence.

Here are some special situations of the use of O_NONBLOCK or O_NDELAY flags.

1. When opening an FIFO with O_RDONLY or O_WRONLY set.
   - If O_NONBLOCK or O_NDELAY is set:

     (a) An `open()` for reading only returns without delay.
     (b) An `open()` for writing returns an error only when no process currently has the file open for reading.
   - If O_NONBLOCK and O_NDELAY are clear:

     (a) An `open()` for reading only blocks until a process opens the file for writing.
     (b) An `open()` for writing only blocks until a process opens the file for reading.

     After both ends of an FIFO have been opened, there is no guarantee that further calls to `open()` O_RDONLY (O_WRONLY) will synchronize with later calls to `open()` O_WRONLY (O_RDONLY) until both ends of the FIFO have been closed by all readers and writers. Any data written into an FIFO will be lost if both ends of the FIFO are closed before the data is read.
2. When opening a block-special or character-special file that supports non-blocking.
   - If O_NONBLOCK or O_NDELAY is set:

     (a) The `open()` function returns without blocking for the device to be ready or available. Subsequent behaviour of the device is device-specific.
   - If O_NONBLOCK and O_NDELAY are clear:

     (a) The `open()` function blocks until the device is ready or available before returning. Otherwise, the behaviour of O_NONBLOCK and O_NDELAY is unspecified.

O_RSYNC

This flag is used to read I/O operations on the file descriptor to complete the synchronous nature at the same level of integrity as specified by the O_DSYNC and O_SYNC flags. If both O_DSYNC and O_RSYNC flags are set in `oflag`, all I/O operations on the file descriptor get completed as defined by synchronized I/O data integrity completion. If both O_SYNC and O_RSYNC are set in `oflag`, all I/O operations on the file descriptor get completed as defined by synchronized I/O file integrity completion.

O_SYNC

This flag is used to write I/O operations on the file descriptor to complete as defined by synchronized I/O file integrity completion.

O_TRUNC

If the file exists and is a regular file, and is successfully opened through O_RDWR or O_WRONLY, its length is truncated to 0, and the mode and owner are unchanged. The O_TRUNC has no effect on FIFO special file or terminal device file. Its effect on other file types is implementation-dependent. The result of using O_TRUNC with O_RDONLY is undefined. If the file exists and if the file is successfully opened for either write or read/write, truncate its length to 0.

Return values: Upon successful completion, the `open()` function opens the file and returns a non-negative integer representing the lowest numbered unused file descriptor. Otherwise, −1 is returned, `errno` is set to indicate the error, and no file are created or modified.

Errors: The `open()` function will fail in case of the following errors:

EACCES

Search permission is denied on a component of the `path` prefix, or the file exists and the permissions specified by `oflag` are denied, or the file does not exist and the write permission is denied for the parent directory of the file to be created, or O_TRUNC is specified and the write permission is denied.

EBADF

The file descriptor provided to `openat()` is invalid.

EDQUOT

The file does not exist, O_CREAT is specified, and either the directory where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted, or the user's quota of i-nodes on the file system where the file is being created has been exhausted.

EEXIST

The O_CREAT and O_EXCL flags are set, and the named file exists.

EINTR

A signal was caught during the `open()` function.

EFAULT

The `path` argument points to an illegal address.

EINVAL

The system does not support synchronized I/O for this file, or the O_XATTR flag was supplied and the underlying file system does not support extended file attributes.

EIO

The `path` argument names a STREAMS file and a hang-up or error occurs during the `open()` function.

EISDIR

The named file is a directory, and `oflag` includes O_WRONLY or O_RDWR.

ELOOP

Too many symbolic links were encountered in resolving `path`.

EMFILE

OPEN_MAX file descriptors are currently open in the calling process.

EMULTIHOP

Components of the `path` function require hopping to multiple remote machines, but the file system does not allow it.

ENAMETOOLONG

The length of the `path` argument exceeds PATH_MAX or a pathname component is longer than NAME_MAX.

ENFILE

The maximum allowable number of file is currently open in the system.

ENOENT

The O_CREAT flag is not set, and the named file does not exist. Or, the O_CREAT flag is set but either the `path` prefix does not exist or the `path` argument points to an empty string.

ENOLINK

The `path` argument points to a remote machine, and the link to that machine is no longer active.

ENOSR

The `path` argument names a STREAMS-based file, and the system is unable to allocate a STREAM.

ENOSPC

The directory or the file system that would contain the new file cannot be expanded, the file does not exist and the O_CREAT flag is specified.

ENOSYS

The device specified by `path` does not support the `open` operation.

ENXIO

The O_NONBLOCK flag is set, the named file is an FIFO, the O_WRONLY flag is set and no process has the file open for reading. Or, the named file is a character-special or block-special file, and the device associated with this special file does not exist.

EOPNOTSUPP

An attempt was made to open a path that corresponds to AF_UNIX socket.

EOVERFLOW

The named file is a regular file, and either the _LARGEFILE flag is not set and the size of the file cannot be represented correctly in an object of type `off_t`. Or, the _LARGEFILE flag is set, and the size of the file cannot be represented correctly in an object of type `off64_t`.

EROFS

The named file resides on a read-only file system, and either O_WRONLY, O_RDWR, O_CREAT (if file does not exist), or O_TRUNC is set in the `oflag` argument.

4.1.2 `creat` Function

The `creat()` function creates a new ordinary file or prepares to rewrite an existing file named by the pathname pointer to by `path.` If the file exists, the length is truncated to 0 and the mode and owner are unchanged.

If the file does not exist, the file's owner ID is set to the effective user ID of the process. The group ID of the file is set to the effective group ID of the process. Or, if the S_ISGID bit is set in the parent directory, the group ID of the file is inherited from the parent directory.

Upon successful completion of the `creat()` function, a write-only file descriptor is returned, and the file is open for writing even if the mode does not permit writing. The file pointer is set to the beginning of the file. The file descriptor is set to remain open across `exec` functions. A new file may also be created with a mode that forbids writing.

The function action, header file required and syntax for the `creat()` function are as follows:

Function action

`creat` – creates a new file or rewrites an existing one.

Header file required

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

Syntax

```
int creat(const char *path, mode_t mode);
```

`The call creat(path, mode) is equivalent to:`

`open(path, O_WRONLY | O_CREAT | O_TRUNC, mode)`

Return values: Upon successful completion of the `creat()` function, a non-negative integer representing the lowest numbered unused file descriptor is returned. Otherwise, −1 is returned; no file are created or modified, and `errno` is set to indicate the error.

Errors: The `creat()` function will fail in case of the following errors:

EACCES

Search permission is denied on a component of the `path` prefix; the file does not exist and the directory in which the file is to be created does not permit writing or, the file exists and the write permission is denied.

EAGAIN

The file exists, the mandatory file/record locking is set, and there are outstanding record locks on the file.

EDQUOT

The directory where the new file entry is being placed cannot be extended because the user's quota of disk blocks on that file system has been exhausted; or the user's quota of i-nodes on the file system where the file is being created has been exhausted.

EFAULT

The `path` argument points to an illegal address.

EINTR

A signal was caught during the execution of the `creat()` function.

EISDIR

The named file is an existing directory.

ELOOP

Too many symbolic links were encountered in translating `path`.

EMFILE

The process has too many open file.

ENFILE

The system file table is full.

ENOENT

A component of the `path` prefix does not exist, or the pathname is null.

ENOLINK

The `path` argument points to a remote machine, and the link to that machine is no longer active.

ENOSPC

The file system is out of i-nodes.

ENOTDIR

A component of the `path` prefix is not a directory.

EOVERFLOW

The file is a large file at the time of execution of the `creat()` function.

EROFS

The named file resides or would reside on a read-only file system.

4.1.3 `write` Function

Data is written to an open file with the `write()` function. The `write()` function attempts to write `nbyte` bytes from the buffer pointed by `buf` to the file associated with the open file descriptor, `fildes`.

The function action, header file required and syntax of the `write()` function are as follows:

Function action

`write` – writes on a file.

Header file required

```
#include <unistd.h>
```

Syntax

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

If `nbyte` is 0, the `write()` function will return 0, and it will have no other results if the file is a regular file; otherwise, the results are unspecified.

On a regular file or any other file capable of seeking, the actual writing of data proceeds from the position in the file indicated by the file `offset` associated with the open file descriptor `fildes`. Before successful return from `write()`, the file offset is incremented by the number of bytes actually written. On a regular file, if this incremented file `offset` is greater than the length of the file, the length of the file will be set to this file `offset`.

If the O_SYNC bit has been set, write I/O operations on the file descriptor get completed as defined by synchronized I/O file integrity completion.

If `fildes` refers to a socket, the `write()` function is equivalent to with no flags set.

On a file not capable of seeking, writing always takes place starting at the current position. The value of a file offset associated with such a device is undefined.

If the O_APPEND flag of the file status flags is set, the file offset will be set to the end of the file prior to each write, and no intervening file modification operation will occur between changing the file offset and the write operation. A common cause for a write error is either filling up a disk or exceeding the file size limit for a given process.

A `write()` to a regular file is blocked in the following two conditions when mandatory file/record locking is set, and there is a record lock owned by another process on the segment of the file to be written:

1. If O_NDELAY or O_NONBLOCK is set, `write()` returns −1 and sets `errno` to EAGAIN.
2. If O_NDELAY and O_NONBLOCK are clear, `write()` sleeps until all blocking locks are removed or `write()` is terminated by a signal.

If `write()` is interrupted by a signal before it writes any data, it will return −1 with `errno` set to EINTR.

If `write()` is interrupted by a signal after it successfully writes some data, it will return the number of bytes written.

After a `write()` to a regular file has successfully returned:

1. Any successful completion operation of the `write()` function from each byte position in the file that was modified by the `write()` function will return the data specified by the `write()` function for that position until such byte positions are again modified.
2. Any subsequent successful `write()` to the same position in the file will overwrite that file data.
3. The write requests to a pipe or FIFO are handled in the same way as a regular file with the following exceptions:

    - There is no file offset associated with a pipe, hence each write request appends to the end of the pipe.
    - If O_NONBLOCK and O_NDELAY are clear, a write request may cause the process to block, but on normal completion it returns `nbyte` (number of bytes).
    - If O_NONBLOCK and O_NDELAY are set, `write()` does not block the process. If a `write()` request for PIPE_BUF or fewer bytes succeeds completely, `write()` returns `nbyte`. Otherwise, if O_NONBLOCK is set, it returns −1 and sets `errno` to EAGAIN; or, if O_NDELAY is set, it returns 0. A `write()` request for greater than {PIPE_BUF} bytes transfers what it can and returns the number of bytes written or it transfers no data. And, if O_NONBLOCK is set, the write returns −1 with `errno` set to EAGAIN; or if O_NDELAY is set, it returns 0. Finally, if a request is greater than PIPE_BUF bytes and all data previously written to the pipe has been read, `write()` transfers at least PIPE_BUF bytes.

When attempting to write to a file descriptor (other than a pipe, an FIFO, a socket or a STREAM) that supports non-blocking writes and cannot accept the data immediately, the following results materialize:

1. If O_NONBLOCK and O_NDELAY are clear, `write()` blocks until the data can be accepted.
2. If O_NONBLOCK or O_NDELAY is set, `write()` does not block the process. If some data can be written without blocking the process, `write()` writes what it can and returns the number of bytes

written. Otherwise, if O_NONBLOCK is set, it returns −1 and sets `errno` to EAGAIN; or if O_NDELAY is set, it returns 0.

Upon successful completion of `write()`, where `nbyte` is greater than 0, `write()` will mark for update the `st_ctime` and `st_mtime` fields of the file, and if the file is a regular file, the S_ISUID and S_ISGID bits of the file mode may be cleared.

Return values: Upon successful completion, `write()` returns the number of bytes actually written to the file associated with `fildes`. This number is never greater than `nbyte`. Otherwise, −1 is returned, the file-pointer remains unchanged, and `errno` is set to indicate the error.

Errors: The `write()` function will fail in case of the following errors:

EAGAIN

Mandatory file/record locking is set, O_NDELAY or O_NONBLOCK is set, and there is a blocking record lock; an attempt is made to write to a STREAM that cannot accept data with the O_NDELAY or O_NONBLOCK flags set; or a write to a pipe or FIFO of PIPE_BUF bytes or less is requested and less than `nbytes` of free space is available.

EBADF

The `fildes` argument is not a valid file descriptor open for writing.

EDEADLK

`write()` was going to sleep and causes a deadlock situation.

EDQUOT

The user's quota of disk blocks on the file system containing the file has been exhausted.

EFBIG

An attempt is made to write a file that exceeds the process's file size limit or the maximum file size.

EFBIG

The file is a regular file, `nbyte` is greater than 0, and the starting position is greater than or equal to the offset maximum established in the file description associated with `fildes`.

EINTR

A signal was caught during the write operation and no data was transferred.

EIO

The process is in the background and is attempting to write to its controlling terminal whose TOSTOP flag to set; or the process is neither ignoring nor blocking SIGTTOU signals, and the process group of the process is orphaned.

ENOLCK

Enforced record locking is enabled and {LOCK_MAX} regions are already locked in the system, or the system record lock table is full and `write()` cannot go to sleep until the blocking record lock is removed.

ENOLINK

The `fildes` argument is on a remote machine and the link to that machine is no longer active.

ENOSPC

During a write to an ordinary file, there is no free space left on the device.

4.1.4 `read` Function

The `read()` function attempts to read `nbyte` bytes from the file associated with the open file descriptor `fildes` into the buffer pointed to by `buf.` The `read()` function reads data previously written to a file. If any portion of a regular file prior to the end-of-file has not been written, `read()` returns bytes with value 0.

For regular file, no data transfer will occur past the offset maximum established in the open file description associated with `fildes`.

The function action, header file required and syntax of the `read()` function are as follows:

Function action

`read` – reads from a file.

Header file required

```
#include <unistd.h>
```

Syntax

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

If `nbyte` is 0, the `read()` function returns 0 and has no other result.

On file that support seeking (e.g. a regular file), the `read()` function starts at a position in the file given by the file offset associated with `fildes`. The file offset is incremented by the number of bytes actually read.

On file that do not support seeking (e.g. terminals), the `read()` function always starts from the current position. The value of the file offset associated with such a file is undefined. If the file refers to a device-special file, the result of subsequent `read()` function requests is implementation-dependent.

When the `read()` function is attempting to read from a regular file with mandatory file/record locking set, and there is a write lock owned by another process on the segment of the file to be read, the following different situations arise:

1. If O_NDELAY or O_NONBLOCK is set, the `read()` function returns −1 and sets `errno` to EAGAIN.
2. If O_NDELAY and O_NONBLOCK are clear, the `read()` function sleeps until the blocking record lock is removed.

When the `read()` function is attempting to read from an empty pipe (or FIFO), the following different situations arise:

1. If no process has the pipe open for writing, the `read()` function returns 0 to indicate end-of-file.
2. If some process has the pipe open for writing and O_NDELAY is set, the `read()` function returns 0.
3. If some process has the pipe open for writing and O_NONBLOCK is set, the `read()` function returns −1 and sets `errno` to EAGAIN.
4. If O_NDELAY and O_NONBLOCK are clear, the `read()` function blocks until data is written to the pipe, or the pipe is closed by all processes that had opened the pipe for writing.

When the `read()` function is attempting to read a file associated with a terminal that has no data currently available, the following different situations arise:

1. If O_NDELAY is set, the `read()` function returns 0.
2. If O_NONBLOCK is set, the `read()` function returns −1 and sets `errno` to EAGAIN.
3. If O_NDELAY and O_NONBLOCK are clear, the `read()` function blocks until data becomes available.

When the `read()` function is attempting to read a file associated with a socket or a stream that is not a pipe, a FIFO, or a terminal, and the file has no data currently available, the following different situations arise:

1. If O_NDELAY or O_NONBLOCK is set, the `read()` function returns −1 and sets `errno` to EAGAIN.
2. If O_NDELAY and O_NONBLOCK are clear, the `read()` function blocks until data becomes available.

Upon successful completion of the `read()` function, where `nbyte` is greater than 0, `read()` will mark for update the st_atime field of the file, and return the number of bytes read. This number will never be greater than `nbyte`. The value returned may be less than `nbyte` if the number of bytes left in the file is less than `nbyte`, if the `read()` request was interrupted by a signal, or if the file is a pipe or FIFO or special file and has fewer than `nbyte` bytes immediately available for reading. For example, a `read()` from a file associated with a terminal may return one typed line of data.

Return values: Upon successful completion of the `read()` function, a non-negative integer is returned indicating the number of bytes actually read. Otherwise, the `read()` function returns −1 and sets `errno` to indicate the error.

Errors: The `read()` function will fail if the following errors occur:

EAGAIN

Mandatory file/record locking is set, O_NDELAY or O_NONBLOCK is set and there is a blocking record lock; total amount of system memory available when reading using raw I/O is temporarily insufficient; no data is waiting to be read on a file associated with `tty` device and O_NONBLOCK is set; or no message is waiting to be read on a stream and O_NDELAY or O_NONBLOCK is set.

EBADF

The `fildes` argument is not a valid file descriptor open for reading.

EBADMSG

The message waiting to be read on a stream is not a data message.

EDEADLK

The `read()` function is going to sleep and causes a deadlock.

EINTR

A signal is caught during the `read()` function operation and no data is transferred.

EINVAL

An attempt is made to read from a stream linked to a multiplexor.

EIO

A physical I/O error has occurred, or the process is in a background process group and is attempting to read from its controlling terminal, and either the process is ignoring or blocking the SIGTTIN signal or the process group of the process is orphaned.

EISDIR

The `fildes` argument refers to a directory on a file system type that does not support read operations on file.

ENOLCK

The system record lock table is full, so the `read()` function cannot go to sleep until the blocking record lock was removed.

ENOLINK

The `fildes` argument is on a remote machine and the link to that machine is no longer active.

ENXIO

The device associated with `fildes` is a block-special or character-special file and the value of the file pointer is out of range.

4.1.5 `close` Function

The `close()` function will de-allocate the file descriptor indicated by `fildes`. To de-allocate means to make the file descriptor available for return by subsequent calls to `open()` or other functions that allocate file descriptors. All outstanding record locks owned by the process on the file associated with the file descriptor will be removed (i.e. unlocked).

The function action, header file required and syntax of the `close()` function are as follows:

Function action

`close` – closes a file descriptor.

Header file required

```
#include <unistd.h>
```

Syntax

```
int close(int fildes);
```

If `close()` is interrupted by a signal that is to be caught, it will return −1 with `errno` set to EINTR and the state of `fildes` is unspecified.

When all file descriptors associated with a pipe or FIFO special file are closed, any data remaining in the pipe or FIFO will be discarded.

When all file descriptors associated with an open file description have been closed, the open file description will be freed.

If the link count of the file is 0 and all file descriptors associated with the file are closed, the space occupied by the file will be freed and the file will no longer be accessible.

If `fildes` refers to the master side of a pseudo-terminal, and this is the last close, a SIGHUP signal is sent to the process group, if any, for which the slave side of the pseudo-terminal is the controlling terminal. It is unspecified whether closing the master side of the pseudo-terminal flushes all queued input and output.

If the asynchronous input and output option is supported, the following situations occur with respect to the `close` function:

1. When there is an outstanding cancelable asynchronous I/O operation against `fildes` and the `close()` function is called, the asynchronous I/O operation may be cancelled.
2. An I/O operation that is not cancelled completes as if the `close()` operation had not yet occurred.
3. All operations that are not cancelled get completed as if the `close()` function is blocked until the operations are completed.
4. The `close()` function operation itself need not block awaiting such I/O completion. Whether any I/O operation is cancelled, and which I/O operation may be cancelled upon `close()`, is implementation-dependent.

Return values: Upon successful completion of the `close()` function, 0 is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors: The `close()` function will fail if the following errors occur:

EBADF

The `fildes` argument is not a valid file descriptor.

EINTR

The `close()` function is interrupted by a signal.

EIO

An I/O error occurs while reading from or writing to the file system.

Prog. 1 How to open and read a file?

In the following example, we open and read a file using the `open()` and `read()` functions. First, we open a file whose filename is passed as argument by the user, then we read the contents of the file and finally print the file using the `printf()` function. At last we close the file using the `close()` function.

```
#include <stdio.h>
# include <errno.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# define DATASIZE 100
int main(int argc,char *argv[])
{    int fd,numbytes;
     char buf[DATASIZE];
     if ((fd=open(argv[1],O_RDONLY))==-1)
     {    perror("open error");
         exit(1);
     }
     if ((numbytes=read(fd,buf,DATASIZE))==-1)
     {    perror("read error");
           exit(2);
     }
     buf[0umbytes]='\0';
      printf("buf contains %s \n",buf);
      close(fd);
      exit(0);
}
```

Prog. 2 How to open and read a file, and write to the buffer?

In the following example, we open a file using the `open()` function and then write it to the buffer using the `write()` function. First, we open a file whose filename is passed as argument by the user, then we write the contents to the file typed by the user as a standard input using the `fgets()` function and finally print it using the `printf()` function. At last we close the file using the `close()` function.

```
#include <stdio.h>
# include <errno.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <string.h>
# define DATASIZE 100
int main(int argc,char *argv[])
{     int fd,numbytes;
```

```
      char buf[DATASIZE];
      if ((fd=open(argv[1],O_WRONLY ))==-1)
      {   perror("open error");
          exit(1);
      }
      printf("enter a string");
      fgets(buf,100,stdin);
      if ((numbytes=write(fd,buf,strlen(buf)))==-1)
      {   perror("write error");
          exit(2);
      }
      printf("%d bytes written in file \n",numbytes);
      close(fd);
      exit(0);
}
```

Prog. 3 How to open a file with O_CREAT macro and read the file?

The difference between the following example and Prog. 2 is that here we ensure that if the filename passed by the user does not exist, the system will create a new file with the same name using O_CREAT macro. 0666 reflects that file is going to be opened in read and write mode for all.

```
# include <stdio.h>
# include <errno.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <string.h>
# define DATASIZE 100
int main(int argc,char *argv[])
  {   int fd,numbytes;
      char buf[DATASIZE];
      char *ch;
      if ((fd=open(argv[1],O_WRONLY |O_CREAT,0666))==-1)
      {   perror("open error");
          exit(1);
      }
      printf("enter a string");
      fgets(buf,100,stdin);
      if ((numbytes=write(fd,buf,strlen(buf)))==-1)
      {   perror("read error");
          exit(2);
      }
      printf("%d bytes written in file \n",numbytes);
```

```
        close(fd);
        exit(0);
    }
```

4.2 File Control

In this section we will talk about activities performed in file control and I/O control on file descriptors. It is an interface to the `fcntl(), lseek(), stat(), fstat(), lstat(), dup(), dup2()` and `ioctl()` functions.

All functions in this module take a file descriptor `fd` as their first argument. This can be an integer file descriptor, such as returned by `sys.stdin.fileno()`, or a file object, like `sys.stdin` itself, which provides a `fileno()` that returns a genuine file descriptor.

4.2.1 `fcntl` Function

The `fcntl()` function performs the requested operation on file descriptor `fd`. The operation is defined by `op` and is operating system dependent. The codes for file control module are also found in the `fcntl` module. The argument `arg` is optional, and defaults to the integer value 0. When present, it can either be an integer value or a string. With the argument missing or being an integer value, the return value of this function is the integer return value of the C `fcntl()` function call. When the argument is a string, it represents a binary structure. The binary data is copied to a buffer whose address is passed to the C `fcntl()` function call. The return value after a successful call is the contents of the buffer.

The function action, header file required and syntax of the `fcntl()` function are as follows:

Function action

`fcntl` - file control.

Header file required

```
#include <unistd.h>
```

Syntax

```
#include<fcntl.h>
int fcntl(int fildes, int cmd, op[,arg ].....);
```

The file control function `fcntl()` performs the requested operations on open file. The `fildes` argument is a file descriptor. The available values for `cmd` are defined in `<fcntl.h>` and are as follows

F_DUPFD

Returns a new file descriptor which shall be the lowest numbered available (i.e. not already open) file descriptor greater than or equal to the third argument, `arg`, taken as an integer of type int. The new file descriptor shall refer to the same open file description as the original file descriptor, and shall share any locks. The FD_CLOEXEC flag associated with the new file descriptor shall be cleared to keep the file open across calls to one of the `exec` functions.

F_GETFD

Gets the file descriptor flags defined in `<fcntl.h>` that are associated with the file descriptor `fildes.` File descriptor flags are associated with a single file descriptor and do not affect other file descriptors that refer to the same file.

F_SETFD

Sets the file descriptor flags defined in `<fcntl.h>`, which are associated with `fildes,` to the third argument, `arg,` taken as type int. If the FD_CLOEXEC flag in the third argument is 0, the file shall remain open across all `exec` functions; otherwise, the file shall be closed upon successful execution of one of the `exec` functions.

F_GETFL

Gets the file status flags and file access modes defined in `<fcntl.h>` for the file description associated with `fildes.` The file access modes can be extracted from the return value using the mask O_ACCMODE, which is defined in `<fcntl.h>`. File status flags and file access modes are associated with the file description and do not affect other file descriptors that refer to the same file with different open file descriptions.

F_SETFL

Sets the file status flags, defined in `<fcntl.h>`, for the file description associated with `fildes` from the corresponding bits in the third argument, `arg,` taken as type int. Bits corresponding to the file access mode and the file creation flags, as defined in `<fcntl.h>,` that are set in `arg` shall be ignored. If any bits in `arg` other than those mentioned here are changed by the application, the result is unspecified.

F_GETOWN

If `fildes` refers to a socket, gets the process or process group ID specified to receive SIGURG signals when out-of-band data is available. Positive values indicate a process ID, and negative values, other than −1, indicate a process group ID. If `fildes` does not refer to a socket, the results are unspecified.

F_SETOWN

If `fildes` refers to a socket, sets the process or process group ID specified to receive SIGURG signals when out-of-band data is available, using the value of the third argument, `arg`, taken as type int. Positive values indicate a process ID, and negative values, other than −1, indicate a process group ID. If `fildes` does not refer to a socket, the results are unspecified.

The following values for `cmd` are available for advisory record locking. Record locking is supported for regular file, and may be supported for other file.

F_GETLK

Gets the first lock which blocks the lock description pointed to by the third argument, `arg`, taken as a pointer to type struct flock, defined in `<fcntl.h>`. The retrieved information shall overwrite the information passed to `fcntl()` in the structure flock. If no lock is found that would prevent this lock from being created, then the structure shall be left unchanged except for the lock type which shall be set to F_UNLCK.

F_SETLK

Sets or clears a file segment lock according to the lock description pointed to by the third argument, `arg`, taken as a pointer to type struct flock, defined in `<fcntl.h>`. F_SETLK can establish shared (or read) locks (F_RDLCK) or exclusive (or write) locks (F_WRLCK) as well as remove either type of lock (F_UNLCK). F_RDLCK, F_WRLCK and F_UNLCK are defined in `<fcntl.h>`. If a shared or exclusive lock cannot be set, `fcntl()` shall return immediately with a return value of −1.

F_SETLKW

This command shall be equivalent to F_SETLK except that if a shared or exclusive lock is blocked by other locks, the thread shall wait until the request can be satisfied. If a signal that is to be caught is received while `fcntl()` is waiting for a region, `fcntl()` shall be interrupted. Upon return from the signal handler, `fcntl()` shall return −1 with `errno` set to EINTR, and the lock operation shall not be done.

Additional implementation-defined values for `cmd` may be defined in `<fcntl.h>`. Their names shall start with F_.

Setting shared locks on file segments using `fcntl()` function

When a shared lock is set on a segment of a file, other processes shall be able to set shared locks on that segment or a portion of it. A shared lock prevents any other process from setting an exclusive lock on any portion of the protected area. A request for a shared lock shall fail if the file descriptor was not opened with read access.

An exclusive lock shall prevent any other process from setting a shared lock or an exclusive lock on any portion of the protected area. A request for an exclusive lock shall fail if the file descriptor was not opened with write access.

The structure `flock` describes the type `(l_type)`, starting offset `(l_whence)`, relative offset `(l_start)`, size `(l_len)` and process ID `(l_pid)` of the segment of the file to be affected.

The value of `l_whence` is SEEK_SET, SEEK_CUR or SEEK_END, to indicate that the relative offset `l_start` bytes shall be measured from the start of the file, current position or end of the file, respectively. The value of `l_len` is the number of consecutive bytes to be locked. The value of `l_len` may be negative (where the definition of `off_t` permits negative values of `l_len`). The `l_pid` field is only used with F_GETLK to return the process ID of the process holding a blocking lock. After a successful F_GETLK request, when a blocking lock is found, the values returned in the `flock` structure shall be as follows:

`l_type`

Type of blocking lock found.

`l_whence`

SEEK_SET.

`l_start`

Start of the blocking lock.

`l_len`

Length of the blocking lock.

`l_pid`

Process ID of the process that holds the blocking lock.

If the command is F_SETLKW and the process must wait for another process to release a lock, then the range of bytes to be locked shall be determined before the `fcntl()` function blocks. If the file size or file descriptor seeks offset change while `fcntl()` is blocked, this shall not affect the range of bytes locked.

If `l_len` is positive, the area affected shall start at `l_start` and end at `l_start + l_len-1.` If `l_len` is negative, the area affected shall start at `l_start + l_len` and end at `l_start-1.` Locks may start and extend beyond the current end of a file, but shall not extend before the beginning of the file. A lock shall be set to extend to the largest possible value of the file offset for that file by setting `l_len` to 0. If such a lock also has `l_start` set to 0 and `l_whence` is set to SEEK_SET, the whole file shall be locked.

There shall be at most one type of lock set for each byte in the file. Before a successful return from an F_SETLK or an F_SETLKW request when the calling process has previously existing locks on bytes in the region specified by the request, the previous lock type for each byte in the specified region shall be replaced by the new lock type. As specified above under the descriptions of shared locks and exclusive locks, an F_SETLK or an F_SETLKW request (respectively) shall fail or block when another process has existing locks on bytes in the specified region, and the type of any of those locks conflicts with the type specified in the request.

All locks associated with a file for a given process shall be removed when a file descriptor for that file is closed by that process or the process holding that file descriptor terminates. Locks are not inherited by a child process.

A potential for deadlock occurs if a process controlling a locked region is put to sleep by attempting to lock another process' locked region. If the system detects that sleeping, unless a locked region is unlocked, would cause a deadlock, `fcntl()` shall fail with an EDEADLK error.

An unlock (F_UNLCK) request in which `l_len` is non-zero and the offset of the last byte of the requested segment is the maximum value for an object of type `off_t` the process shall be treated as a request to unlock. When the process has an existing lock in which `l_len` is 0 and which includes the last byte of the requested segment, the process shall be treated as a request to unlock from the start of the requested segment with an `l_len` equal to 0. Otherwise, an unlock (F_UNLCK) request shall attempt to unlock only the requested segment.

When the file descriptor `fildes` refers to a shared memory object, the behaviour of `fcntl()` shall be the same as for a regular file except that the effect of the following values for the argument `cmd` shall be unspecified: F_SETFL, F_GETLK, F_SETLK and F_SETLKW.

If `fildes` refers to a typed memory object, the result of the `fcntl()` function is unspecified.

Return values: Upon successful completion of the `fcntl()` function, the values returned shall be as follows:

F_DUPFD

A new file descriptor.

F_GETFD

Value of flags defined in `<fcntl.h>`. The return value shall not be negative.

F_SETFD

Value other than −1.

F_GETFL

Value of file status flags and access modes. The return value is not negative.

F_SETFL

Value other than −1.

F_GETLK

Value other than −1.

F_SETLK

Value other than −1.

F_SETLKW

Value other than −1.

F_GETOWN

Value of the socket owner process or process group; this will not be −1.

F_SETOWN

Value other than −1.

Otherwise, −1 shall be returned and `errno` set to indicate the error.

Errors: The `fcntl()` function shall fail in case of the following errors.

EACCES or EAGAIN

The `cmd` argument is F_SETLK, the type of lock `(l_type)` is a shared (F_RDLCK) or exclusive (F_WRLCK) lock, and the segment of a file to be locked is already exclusive – locked by another process, or the type is an exclusive lock and some portion of the segment of a file to be locked is already shared-locked or exclusive-locked by another process.

EBADF

The `fildes` argument is not a valid open file descriptor, or the argument `cmd` is F_SETLK or F_SETLKW; the type of lock, `l_type,` is a shared lock (F_RDLCK), and `fildes` is not a valid file descriptor open for reading; or the type of lock, `l_type,` is an exclusive lock (F_WRLCK), and `fildes` is not a valid file descriptor open for writing.

EINTR

The `cmd` argument is F_SETLKW, and the function was interrupted by a signal.

EINVAL

The `cmd` argument is invalid; or the `cmd` argument is F_DUPFD and `arg` is negative or greater than or equal to {OPEN_MAX}; or the `cmd` argument is F_GETLK, F_SETLK or F_SETLKW and the data pointed to by `arg` is not valid; or `fildes` refers to a file that does not support locking.

EMFILE

The argument `cmd` is F_DUPFD and {OPEN_MAX} file descriptors are currently open in the calling process, or no file descriptors greater than or equal to `arg` are available.

ENOLCK

The argument `cmd` is F_SETLK or F_SETLKW and satisfying the lock or unlock request would result in the number of locked regions in the system exceeding a system-imposed limit.

EOVERFLOW

One of the values to be returned cannot be represented correctly.

EOVERFLOW

The `cmd` argument is F_GETLK, F_SETLK or F_SETLKW and the smallest or, if `l_len` is non-zero, the largest offset of any byte in the requested segment cannot be represented correctly in an object of type `off_t`.

Prog. 4 How to lock and unlock a file using `fcntl`?

The following example demonstrates how to place a lock on bytes 200–209 of a file and then remove the lock. F_SETLK is used to perform a non-blocking lock request so that the process does not have to wait if an incompatible lock is held by another process; instead the process can take some other action.

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <errno.h>

intmain(int argc, char *argv[])
{
  int fd;
  struct flock fl;

  fd = open("testfile", O_RDWR);
  if (fd == -1)
/* Handle error */;

/* Make a non-blocking request to place a write lock
on bytes 200-209 of testfile */

fl.l_type = F_WRLCK;
fl.l_whence = SEEK_SET;
fl.l_start = 100;
fl.l_len = 10;

if (fcntl(fd, F_SETLK, &fl) == -1) {
  if (errno == EACCES || errno == EAGAIN) {
  printf("Already locked by another process\n");
   /* We can't get the lock at the moment */
} else {
  /* Handle unexpected error */;
}
} else { /* Lock was granted... */

  /* Perform I/O on bytes 200 to 209 of file */
  /* Unlock the locked bytes */
fl.l_type = F_UNLCK;
fl.l_whence = SEEK_SET;
fl.l_start = 200;
fl.l_len = 10;
if (fcntl(fd, F_SETLK, &fl) == -1)
  /* Handle error */;

}  exit(EXIT_SUCCESS);

} /* main */
```

Prog. 5 How to set the close-on-exec flag?

The following program sets the close-on-exec flag for the file descriptor fd.

```
#include <unistd.h>
#include <fcntl.h>
…
  int flags;

flags = fcntl(fd, F_GETFD);
if (flags == -1)
  /* Handle error */;
flags |= FD_CLOEXEC;
if (fcntl(fd, F_SETFD, flags) == -1)
  /* Handle error */;
```

4.2.2 `lseek` Function

The `lseek()` function is used for positioning a file descriptor and move the read/write file offset. The behaviour of `lseek()` on devices that are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

The `lseek()` function allows the file offset to be set beyond the end of the existing data in the file. If data is later written at this point, subsequent reads of data in the gap return bytes with the value 0 until data is actually written into the gap. The `lseek()` function does not, by itself, extend the size of a file.

If `fildes` refers to a shared memory object, the result of the `lseek()` function is unspecified. If `fildes` refers to a typed memory object, the result of the `lseek()` function is unspecified.

The function action, header file required and syntax of the `lseek()` function are as follows.

Function action

`lseek` – moves the read/write file offset.

Header file required

```
#include<unistd.h>
```

Syntax

```
off_t lseek(int fildes, off_t offset, int whence);
```

The `lseek()` function sets the file offset for the open file description associated with the file descriptor `fildes` as follows:

1. If `whence` is SEEK_SET, the file offset is set to offset bytes.
2. If `whence` is SEEK_CUR, the file offset is set to its current location plus offset.
3. If `whence` is SEEK_END, the file offset is set to the size of the file plus offset.

The symbolic constants SEEK_SET, SEEK_CUR and SEEK_END are defined in `<unistd.h>`

The behaviour of `lseek()` on devices which are incapable of seeking is implementation-defined. The value of the file offset associated with such a device is undefined.

Return values: Upon successful completion of the `lseek()` function, the resulting offset, as measured in bytes from the beginning of the file, is returned. Otherwise, (`off_t`) −1 is returned, `errno` is set to indicate the error and the file offset remains unchanged.

Errors: The `lseek()` function will fail if the following errors occur:

EBADF

The `fildes` argument is not an open file descriptor.

EINVAL

The `whence` argument is not a proper value, or the resulting file offset would be negative for a regular file, block-special file or directory.

EOVERFLOW

The resulting file offset is a value which cannot be represented correctly in an object of type `off_t`.

ESPIPE

The `fildes` argument is associated with a pipe, FIFO or socket.

Prog. 6 How to open a file with `lseek()` and read it?

In the following program, we open a file and we pass filename as argument. Then we read the file contents from a particular location and finally print it using the `printf()` function. At the end, we are close the file using the `close()` function.

```
# include <stdio.h>
# include <errno.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <string.h>
# define DATASIZE 100

int main(int argc,char *argv[])
{    int fd,numbytes,pos;
     char buf[DATASIZE];
     char *ch;
     if ((fd=open(argv[1],O_RDWR | O_APPEND))==-1)
     {    perror("open error");
          exit(1);
     }
     if ((pos=lseek(fd,10,SEEK_SET))==-1)
     {    perror("lseek error");
          exit(2);
     }
     printf("current file pointer poinintg to %d byte \n",pos);
     if ((numbytes=read(fd,buf,100))==-1)
     {    perror("read eorror");
          exit(2);
     }
     buf[numbytes]='\0';
     if ((pos=lseek(fd,0,SEEK_CUR))==-1)
     {    perror("seek error");
          exit(3);
     }
     printf("current file offset at : %d \n",pos);
     printf("data read is \t %s \n",buf);
     printf("enter a string");
     fgets(buf,100,stdin);
     if ((numbytes=write(fd,buf,strlen(buf)))==-1)
```

```
        {   perror("read error");
            exit(2);
        }
        printf("%d bytes written in file \n",numbytes);
        if ((pos=lseek(fd,0,SEEK_CUR))==-1)
        {   perror("seek error");
            exit(3);
        }
        printf("current file offset at : %d \n",pos);
        close(fd);
        exit(0);
    }
```

4.2.3 `stat`, `lstat`, `fstat` Functions

The `stat()` function obtains information about the file pointed to by `path`. Read, write or execute permission of the named file is not required, but all directories listed in the pathname leading to the file must be searchable.

The `lstat()` function obtains file attributes similar to `stat()` except when the named file is a symbolic link; in that case, `lstat()` returns information about the link, while `stat()` returns information about the file the link references.

The `fstat()` function obtains information about an open file known by the file descriptor fildes, obtained from a successful `open()`, `creat()`, `dup()`, `fcntl()` or `pipe()` function.

The function action, header file required and syntax for the `stat()`, `lstat()` and `fstat()` functions are as follows.

Function action

`stat, lstat, fstat` – get file status.

Header file required

```
#include <sys/types.h>
#include <sys/stat.h>
```

Syntax

```
int stat(const char *path, struct stat *buf);
int lstat(const char *path, struct stat *buf);
int fstat(int fildes, struct stat *buf);
```

The first argument is self explanatory, that is `path` or `fildes` (file descriptor). The `buf` argument is a pointer to a `stat` structure into which information is placed concerning the file. A `stat` structure includes the following members:

```
mode_t  st_mode;      /* File mode */
ino_t  st_ino;        /* I-node number */
dev_t  st_dev;        /* ID of device containing */
                      /* a directory entry for this file */
dev_t  st_rdev;       /* ID of device */
                      /* This entry is defined only for */
                      /* character-special or block-special file */
nlink_t st_nlink;     /* Number of links */
uid_t  st_uid;        /* User ID of the file's owner */
gid_t  st_gid;        /* Group ID of the file's group */
off_t  st_size;       /* File size in bytes */
time_t  st_atime;     /* Time of last access */
time_t  st_mtime;     /* Time of last data modification */
time_t  st_ctime;     /* Time of last file status change */
                      /* Times measured in seconds since */
                      /* 00:00:00 UTC, Jan. 1, 1970 */
long  st_blksize;     /* Preferred I/O block size */
blkcnt_t st_blocks;   /* Number of 512 byte blocks allocated*/
```

The description of stat structure members used above are as follows:

`st_mode`

It is the mode of the file as described in `mknod()`. In addition to the modes described in `mknod()`, the mode of a file can also be S_IFLNK if the file is a symbolic link. S_IFLNK can be returned by either `lstat()` or `fstat()` when the AT_SYMLNK_NOFOLLOW flag is set.

`st_ino`

This field uniquely identifies the file in a given file system. The pair `st_ino` and `st_dev` uniquely identifies regular file.

`st_dev`

This field uniquely identifies the file system that contains the file. Its value may be used as input to the `ustat()` function to determine more information about this file system. No other meaning is associated with this value.

`st_rdev`

This field should be used only by administrative commands. It is valid only for block-special or character-special file, and it only has meaning on the system where the file was configured.

`st_nlink`

This field should be used only by administrative commands.

`st_uid`

This field represents the user ID of the file's owner.

`st_gid`

This field represents the group ID of the file's group.

`st_size`

For regular file, this is the address of the end of the file. For block-special or character-special file, this is not defined.

`st_atime`

It represents the time when file data was last accessed. It can be changed by the following functions: `creat()`, `mknod()`, `pipe()`, `utime()` and `read()`.

`st_mtime`

It represents the time when data was last modified. It can be changed by the following functions: `creat()`, `mknod()`, `pipe()`, `utime()` and `write()`.

`st_ctime`

It represents the time when the file status was last changed. It can be changed by the following functions: `chmod()`, `chown()`, `creat()`, `link()`, `mknod()`, `pipe()`, `unlink()`, `utime()` and `write()`.

`st_blksize`

It gives a hint as to the 'best' unit size for I/O operations. This field is not defined for block-special or character-special file.

`st_blocks`

It gives the total number of physical blocks of size 512 bytes actually allocated on the disk. This field is not defined for block-special or character-special file.

Return values: Upon successful completion of the `stat()`, `fstat()`, and `lstat()` functions, 0 is returned. Otherwise, −1 is returned, and `errno` is set to indicate the error.

Errors: The `stat()`, `fstat()`, `lstat()` functions will fail in case of the following errors:

EOVERFLOW

The file size in bytes or the number of blocks allocated to the file or the file serial number cannot be represented correctly in the structure pointed to by `buf`.

EACCES

Search permission is denied for a component of the `path` prefix.

EFAULT

The `buf` or `path` argument points to an illegal address.

EINTR

A signal was caught during the execution of the `stat()` or `lstat()` function.

ELOOP

Too many symbolic links were encountered in translating `path`.

ENAMETOOLONG

The length of the `path` argument exceeds PATH_MAX, or the length of a `path` component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect.

ENOENT

The named file does not exist or is the null pathname.

ENOLINK

The `path` argument points to a remote machine, and the link to that machine is no longer active.

ENOTDIR

A component of the `path` prefix is not a directory, or the `fildes` argument does not refer to a valid directory when given a non-null relative path.

EOVERFLOW

A component is too large to store in the structure pointed to by `buf`.

EBADF

The `fildes` argument is not a valid open file descriptor. Note that in `fstat()` the `fildes` argument may also have the valid value of AT_FDCWD.

EFAULT

The `buf` argument points to an illegal address.

EINTR

A signal was caught during the execution of the `fstat()` function.

ENOLINK

The `fildes` argument points to a remote machine and the link to that machine is no longer active.

EOVERFLOW

A component is too large to store in the structure pointed to by `buf`.

Prog. 7 Demonstration of the working of the `stat()` function.

In the following program, we using the `stat()` function to obtain information about a file pointed to by `path`. First, we open the file and pass the filename as argument, then print the information of the file using the members of the `stat` structure. At last we close the file using the `close()` function.

```
# include <stdio.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <unistd.h>
int main (int argc,char *argv[])
    {   int n;
        struct stat buf;
        if ((n=stat(argv[1],&buf))==-1)
        {   perror("stat error");
            exit(1);
        }
        printf("i-node no: = %d \n",buf.st_ino);
        printf("uid no: = %d \n",buf.st_uid);
        printf("gid no: = %d \n",buf.st_gid);
        printf("no: links = %d \n",buf.st_nlink);
        printf("block size = %d \n",buf.st_blksize);
        printf("no: of blocks = %d \n",buf.st_blocks);
        printf("file size = %d \n",buf.st_size);
        if (S_ISREG(buf.st_mode))
          printf("regular file \n");
        else if (S_ISDIR(buf.st_mode))
          printf("directory file \n");
        else if (S_ISCHR(buf.st_mode))
          printf("character file \n");
        else if (S_ISBLK(buf.st_mode))
          printf("block file \n");
        else if (S_ISLNK(buf.st_mode))
          printf("symbolic file \n");
        else if (S_ISFIFO(buf.st_mode))
          printf("fifo file \n");
        else if (S_ISSOCK(buf.st_mode))
          printf("socket file \n");
        exit(0);
}
```

Prog. 8 Demonstration of the working of the `lstat()` function.

The following program demonstrates the same function as was done with the `stat()` function in Prog. 7 except that here the named file is a symbolic link, and `lstat()` returns information about the link. First, we open a file and pass the filename as argument, then we print the information of the file using the members of the `stat` structure. At last we close the file using `close()` function.

```
# include <stdio.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <unistd.h>
int main (int argc,char *argv[])
   {    int n;
        struct stat buf;
        char *mstr,*astr,*cstr;
        if ((n=lstat(argv[1],&buf))==-1)
        {    perror("stat error");
             exit(1);
        }
        printf("i-node no: = %d \n",buf.st_ino);
        printf("uid no: = %d \n",buf.st_uid);
        printf("gid no: = %d \n",buf.st_gid);
        printf("no: links = %d \n",buf.st_nlink);
        printf("block size = %d \n",buf.st_blksize);
        printf("no: of blocks = %d \n",buf.st_blocks);
        printf("file size = %d \n",buf.st_size);
        if (S_ISREG(buf.st_mode))
             printf("regular file \n");
        else if (S_ISDIR(buf.st_mode))
              printf("directory file \n");
        else if (S_ISCHR(buf.st_mode))
              printf("character file \n");
        else if (S_ISBLK(buf.st_mode))
              printf("block file \n");
        else if (S_ISLNK(buf.st_mode))
              printf("symbolic file \n");
        else if (S_ISFIFO(buf.st_mode))
              printf("fifo file \n");
        else if (S_ISSOCK(buf.st_mode))
              printf("socket file \n");
        mstr=ctime(&buf.st_mtime);
              printf("Modification time = %s \n",mstr);
        astr=ctime(&buf.st_atime);
              printf("Access time = %s \n",astr);
        cstr=ctime(&buf.st_ctime);
              printf("Change of ststus time = %s \n",cstr);
        exit(0);
}
```

Prog. 9 Demonstration of the working of the `fstat()` function.
The following program demonstrates how the `fstat()` function obtains information about an open file known by the file descriptor `fildes`. First, we open a file and pass the filename as argument, then we print the information of the file using the members of the `stat` structure. At last, we close the file using the `close()` function.

```
# include <stdio.h>
# include <sys/types.h>
# include <sys/stat.h>
# include <fcntl.h>
# include <unistd.h>
int main (int argc,char *argv[])
    {   int n,fd;
        struct stat buf;
        if ((fd=open(argv[1],O_RDONLY))==-1)
        {   perror("open error");
            exit(1);
        }
        if ((n=fstat(fd,&buf))==-1)
        {   perror("fstat error");
            exit(2);
        }
        printf("i-node no: = %d \n",buf.st_ino);
        printf("uid no: = %d \n",buf.st_uid);
        printf("gid no: = %d \n",buf.st_gid);
        printf("no: links = %d \n",buf.st_nlink);
        printf("block size = %d \n",buf.st_blksize);
        printf("no: of blocks = %d \n",buf.st_blocks);
        printf("file size = %d \n",buf.st_size);
        if (S_ISREG(buf.st_mode))
          printf("regular file \n");
        else if (S_ISDIR(buf.st_mode))
          printf("directory file \n");
        else if (S_ISCHR(buf.st_mode))
          printf("character file \n");
        else if (S_ISBLK(buf.st_mode))
          printf("block file \n");
        else if (S_ISLNK(buf.st_mode))
          printf("symbolic file \n");
        else if (S_ISFIFO(buf.st_mode))
          printf("fifo file \n");
        else if (S_ISSOCK(buf.st_mode))
          printf("socket file \n");
        exit(0);
}
```

4.2.4 dup Function

The `dup()` function returns a new file descriptor having the following features in common with the original open file descriptor `fildes`:

1. Same open file (or pipe).
2. Same file pointer (i.e. both file descriptors share one file pointer).
3. Same access mode (read, write or read/write).
4. The new file descriptor is set to remain open across `exec` functions.

The function action, header file required and syntax of the `dup()` function are as follows:

Function action

`dup` – duplicates an open file descriptor.

Header file required

```
#include <unistd.h>
```

Syntax

```
int dup(int fildes);
```

It is important to note that the file descriptor returned is the lowest one available. The `dup(fildes)` function call is equivalent to:

```
fcntl(fildes, F_DUPFD, 0)
```

Return values: Upon successful completion of the `dup()` function, a non-negative integer representing the file descriptor is returned. Otherwise, −1 is returned, and `errno` is set to indicate the error.

Errors: The `dup()` function will fail in case of the following errors:

EBADF

The `fildes` argument is not a valid open file descriptor.

EINTR

A signal was caught during the execution of the `dup()` function.

EMFILE

The process has too many open file.

ENOLINK

The `fildes` argument is on a remote machine, and the link to that machine is no longer active.

dup2 function: The `dup2()` function causes the file descriptor `fildes2` to refer to the same file as `fildes.` The `fildes` argument is a file descriptor referring to an open file, and `fildes2` is a non-negative integer less than the current value for the maximum number of open file descriptors allowed in the calling process. If `fildes2` already refers to an open file, not `fildes`, it is closed first. If `fildes2` refers to `fildes`, or if `fildes` is not a valid open file descriptor, `fildes2` will not be closed first.

The function action, header file required and syntax of the `dup2()` function are as follows:

Function action

`dup2` – duplicates an open file descriptor.

Header file required

```
#include <unistd.h>
```

Syntax

```
int dup2(int fildes, int fildes2);
```

The `dup2()` function is equivalent to `fcntl(fildes, F_DUP2FD, fildes2)`.

Return values: Upon successful completion of the `dup2()` function, a non-negative integer representing the file descriptor is returned. Otherwise, −1 is returned, and `errno` is set to indicate the error.

Errors: The `dup2()` function will fail in case of the following errors:

EBADF

The `fildes` argument is not a valid open file descriptor.

EBADF

The `fildes2` argument is negative or is not less than the current resource limit returned by `getrlimit(RLIMIT_NOFILE, ...)`.

EINTR

A signal was caught during the `dup2()` call.

EMFILE

The process has too many open file.

4.2.5 `ioctl` Function

The `ioctl()` function performs a variety of control functions on devices and STREAMS. For non-STREAMS file, the functions performed by this call are device-specific control functions.

The request argument and an optional third argument with varying types are passed to the file designated by `fildes` and are interpreted by the device driver.

The function action, header file required and syntax of the `ioctl()` function are as follows:

Function action

`ioctl` – controls device.

Header file required

```
#include <unistd.h>
#include <stropts.h>
```

Syntax

```
int ioctl(int fildes, int request, /* arg */ ...);
```

In the `ioctl()` function, the `fildes` argument is an open file descriptor that refers to a device. The `request` argument selects the control function to be performed, and it depends on the device being addressed. The `arg` argument represents a third argument that has additional information needed by this specific device to perform the requested function. The data type of `arg` depends on the particular control request, but it is either an `int` or a pointer to a device-specific data structure.

Return values: Upon successful completion of the `ioctl()` function, the value returned depends on the device control function, but it must be a non-negative integer. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors: The `ioctl()` function will fail in case of the following errors:

EBADF

The `fildes` argument is not a valid open file descriptor.

EINTR

A signal was caught during the execution of the `ioctl()` function.

4.3 Directory Maintenance

This module performs file control and I/O control on directories. It is an interface to the `chmod()`, `chown()`, `unlink()`, `link()`, `symlink()`, `mkdir ()`, `rmdir()`, `chdir()` and `getcwd()` functions.

4.3.1 chmod Function

The `chmod()` function sets the access permission portion of the mode of the file whose name is given by `path` or referenced by the open file descriptor `fildes` to the bit pattern contained in mode.

The function action, header file required and syntax of the `chmod()` function are as follows:

Function action

`chmod` – changes access permission mode of file.

Header file required

```
#include <sys/types.h>
#include <sys/stat.h>
```

Syntax

```
int chmod(const char *path, mode_t mode);
```

Access permission bits are interpreted as follows:

`S_ISUID`	04000	Set user ID on execution.
`S_ISGID`	020#0	Set group ID on execution if # is 7, 5, 3 or 1.
`S_ISVTX`	01000	Save text image after execution.
`S_IRWXU`	00700	Read, write, execute by owner.
`S_IRUSR`	00400	Read by owner.
`S_IWUSR`	00200	Write by owner.
`S_IXUSR`	00100	Execute (search if a directory) by owner.
`S_IRWXG`	00070	Read, write, execute by group.
`S_IRGRP`	00040	Read by group.
`S_IWGRP`	00020	Write by group.
`S_IXGRP`	00010	Execute by group.
`S_IRWXO`	00007	Read, write, execute (search) by others.
`S_IROTH`	00004	Read by others.
`S_IWOTH`	00002	Write by others.
`S_IXOTH`	00001	Execute by others.

Modes are constructed by the bitwise-OR operation of the access permission bits.

The effective user ID of the process must match the owner of the file, or the process must have the appropriate privilege to change the mode of a file. If the process is not a privileged process and the file is not a directory, mode bit 01000 (save text image on execution) is cleared.

If neither the process is privileged nor the file's group is a member of the process's supplementary group list, and the effective group ID of the process does not match the group ID of the file, mode bit 02000 (set group ID on execution) is cleared.

If a directory is writable and has S_ISVTX (the sticky bit set), file within that directory can be removed or renamed only if one or more of the following is true:

1. The user owns the file.
2. The user owns the directory.
3. The file is writable by the user.
4. The user is a privileged user.

If a directory has the set group ID bit set, a given file created within that directory will have the same group ID as the directory if the directory group ID is part of the group ID set of the process that created the file. Otherwise, the newly created file's group ID will be set to the effective group ID of the creating process.

If the mode bit 02000 (set group ID on execution) is set and the mode bit 00010 (execute or search by group) is not set, mandatory file/record locking will exist on a regular file. This may affect future calls to `open()`, `creat()`, `read()` and `write()` on this file.

Upon successful completion, `chmod()` marks for update the `st_ctime` field of the file.

Return values: Upon successful completion of the `chmod()` function, 0 is returned. Otherwise, −1 is returned, the file mode is unchanged, and `errno` is set to indicate the error.

Errors: The `chmod()` function will fail in case of the following errors:

EACCES

Search permission is denied on a component of the `path` prefix of `path`.

EFAULT

The `path` argument points to an illegal address.

EINTR

A signal was caught during execution of the function.

EIO

An I/O error occurred while reading from or writing to the file system.

ELOOP

Too many symbolic links were encountered in translating `path`.

ENAMETOOLONG

The length of the `path` argument exceeds PATH_MAX, or the length of a `path` component exceeds NAME_MAX while _POSIX_NO_TRUNC is in effect.

ENOENT

Either a component of the `path` prefix or the file referred to by `path` does not exist or is a null pathname.

ENOLINK

The `fildes` argument points to a remote machine and the link to that machine is no longer active.

ENOTDIR

A component of the prefix of `path` is not a directory.

EPERM

The effective user ID does not match the owner of the file and is not super-user.

EROFS

The file referred to by `path` resides on a read-only file system.

4.3.2 chown Function

The `chown()` function changes the user and group ownership of a file. The `path` argument in `chown()` function points to a pathname naming a file. The user ID and group ID of the named file are set to the numeric values contained in owner and group, respectively.

The function action, header file required and syntax of the `chown()` function are as follows:

Function action

`chown` – changes owner and group of a file.

Header file required

```
#include<unistd.h>
```

Syntax

```
int chown(const char *path, uid_t owner, gid_t group);
```

In `chown()` function, only processes with an effective user ID equal to the user ID of the file or with appropriate privileges may change the ownership of a file. If _POSIX_CHOWN_RESTRICTED is in effect for `path`:

1. Changing the user ID is restricted to processes with appropriate privileges.
2. Changing the group ID is permitted to a process with an effective user ID equal to the user ID of the file, but without appropriate privileges, if and only if the owner is equal to the file's user ID or `(uid_t)`−1 and the group is equal either to the calling process's effective group ID or to one of its supplementary group IDs.

If the specified file is a regular file, one or more of the S_IXUSR, S_IXGRP or S_IXOTH bits of the file mode are set, and the process does not have appropriate privileges, the set-user-ID (S_ISUID) and set-group-ID (S_ISGID) bits of the file mode shall be cleared upon successful return from `chown()`. If the specified file is a regular file, one or more of the S_IXUSR, S_IXGRP or S_IXOTH bits of the file mode are set, and the process has appropriate privileges, it is implementation-defined whether the set-user-ID and set-group-ID bits are altered. If the `chown()` function is successfully invoked on a file that is not a regular file and one or more of the S_IXUSR, S_IXGRP or S_IXOTH bits of the file mode are set, the set-user-ID and set-group-ID bits may be cleared.

If owner or group is specified as (**`uid_t`**)−1 or (**`gid_t`**)−1, respectively, the corresponding ID of the file shall not be changed. If both owner and group are −1, the times need not be updated.

Upon successful completion, `chown()` shall mark for update the `st_ctime` field of the file.

Return values: Upon successful completion of `chown()` function, 0 shall be returned. Otherwise −1 shall be returned, and `errno` is set to indicate error. If −1 is returned, no changes are made in the user ID and group ID of the file.

Errors: The `chown()` function shall fail in case of the following errors:

EACCES

Search permission is denied on a component of the `path` prefix.

ELOOP

A loop exists in symbolic links encountered during resolution of the `path` argument.

ENAMETOOLONG

The length of the `path` argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

ENOTDIR

A component of the `path` prefix is not a directory.

ENOENT

A component of `path` does not name an existing file or `path` is an empty string.

EPERM

The effective user ID does not match the owner of the file, or the calling process does not have appropriate privileges and _POSIX_CHOWN_RESTRICTED indicates that such privilege is required.

EROFS

The named file resides on a read-only file system.

\EIO

An I/O error occurred while reading or writing to the file system.

EINTR

The `chown()` function was interrupted by a signal which was caught.

EINVAL

The owner or group ID supplied is not a value supported by the implementation.

ELOOP

More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the `path` argument.

ENAMETOOLONG

As a result of encountering a symbolic link in resolution of the `path` argument, the length of the substituted pathname string exceeded {PATH_MAX}.

4.3.3 `link` Function

The `link()` function creates a new link (directory entry) for the existing file, `path1`.

The `path1` argument points to a pathname naming an existing file. The `path2` argument points to a pathname naming the new directory entry to be created. The `link()` function shall automatically create a new link for the existing file, and the link count of the file shall be incremented by one.

If `path1` names a directory, `link()` shall fail unless the process has appropriate privileges, and the implementation supports using `link()` on directories.

The function action, header file required and syntax of the `link()` function are as follows:

Function action

`link` – links to a file.

Header file required

```
#include<unistd.h>
```

Syntax

```
int link(const char *path1, const char *path2);
```

Upon successful completion, `link()` shall mark for update the `st_ctime` field of the file. Also, the `st_ctime` and `st_mtime` fields of the directory that contain the new entry shall be marked for update.

If `link()` fails, no link shall be created and the link count of the file shall remain unchanged. The implementation may require that the calling process has permission to access the existing file.

Return values: Upon successful completion of the `link()` function, 0 is returned. Otherwise, −1 is returned, and `errno` is set to indicate the error.

Errors: The `link()` function shall fail in case of the following errors:

EACCES

A component of either `path` prefix denies search permission, or the requested link requires writing in a directory that denies write permission, or the calling process does not have permission to access the existing file which is required by the implementation.

EEXIST

The `path2` argument resolves to an existing file or refers to a symbolic link.

ELOOP

A loop exists in symbolic links encountered during resolution of the `path1` or `path2` argument.

EMLINK

The number of links to the file named by `path1` would exceed {LINK_MAX}.

ENAMETOOLONG

The length of the `path1` or `path2` argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

ENOENT

A component of either `path` prefix does not exist, the file named by `path1` does not exist or `path1` or `path2` point to an empty string.

ENOSPC

The directory to contain the link cannot be extended.

ENOTDIR

A component of either `path` prefix is not a directory.

EPERM

The file named by `path1` is a directory and either the calling process does not have appropriate privileges or the implementation prohibits using `link()` on directories.

EROFS

The requested link requires writing in a directory on a read-only file system.

EXDEV

The link named by `path2` and the file named by `path1` are on different file systems and the implementation does not support links between file systems.

EXDEV

`path1` refers to a named STREAM.

ELOOP

More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the `path1` or `path2` argument.

ENAMETOOLONG

As a result of encountering a symbolic link in resolution of the `path1` or `path2` argument, the length of the substituted pathname string exceeded {PATH_MAX}.

Prog. 10 How to create a link to a file?

The following program creates a link to a file named `/home/alp/file1` by creating a new directory entry named `/alps/dir2`.

```
#include <unistd.h>
char *path1 = "/home/alp/dir1";
char *path2 = "/alps/dir2";
int  status;
...
status = link (path1, path2);
```

4.3.4 `unlink` Function

The `unlink()` function removes a link to a file. If `path` names a symbolic link, `unlink()` removes that symbolic link but does not affect any file or directory named by the contents of the symbolic link. Otherwise, `unlink()` removes the link named by the pathname pointed to by `path` and decrements the link count of the file referenced by the link.

The function action, header file required and syntax of the `unlink()` function are as follows:

Function action

`unlink` – removes a directory entry.

Header file required

```
#include <unistd.h>
```

Syntax

```
int unlink(const char *path);
```

When the file's link count becomes 0 and no process has the file open, the space occupied by the file is freed, and the file is no longer accessible. If one or more processes have the file open when the last link is removed, the link is removed before `unlink()` returns, but the removal of the file contents shall be postponed until all references to the file are closed.

The `path` argument does not name a directory unless the process has appropriate privileges, and the implementation supports using `unlink()` on directories.

Upon successful completion, `unlink()` shall mark for update the `st_ctime` and `st_mtime` fields of the parent directory. Also, if the file's link count is not 0, the `st_ctime` field of the file shall be marked for update.

Return values: Upon successful completion of the `unlink()` funtion, 0 is returned. Otherwise, −1 is returned, and `errno` is set to indicate the error. If −1 is returned, the named file cannot be changed.

Errors: The `unlink()` function will fail in case of the following errors:

EACCES

Search permission is denied for a component of the `path` prefix, or write permission is denied on the directory containing the directory entry to be removed.

EBUSY

The file named by the `path` argument cannot be unlinked because it is being used by the system or another process, and the implementation considers this an error.

ELOOP

A loop exists in symbolic links encountered during resolution of the `path` argument.

ENAMETOOLONG

The length of the `path` argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

ENOENT

A component of `path` does not name an existing file, or `path` is an empty string.

ENOTDIR

A component of the `path` prefix is not a directory.

EPERM

The file named by `path` is a directory, and either the calling process does not have appropriate privileges, or the implementation prohibits using `unlink()` on directories.

EPERM or EACCES

The S_ISVTX flag is set on the directory containing the file referred to by the `path` argument and the caller is not the file owner, nor is the caller the directory owner, nor does the caller have appropriate privileges.

EROFS

The directory entry to be unlinked is part of a read-only file system.

EBUSY

The file named by `path` is a named STREAM.

ELOOP

More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the `path` argument.

ENAMETOOLONG

As a result of encountering a symbolic link in resolution of the `path` argument, the length of the substituted pathname string exceeded {PATH_MAX}.

ETXTBSY

The entry to be unlinked is the last directory entry to a pure procedure (shared text) file that is being executed.

Prog. 11 How to remove a link to a file?

The following program removes a link to a file named `/home/alp/file1` by removing the entry named `/alps/dir2`.

```
#include <unistd.h>

char *path = "/alps/dir2";
int  status;
...
status = unlink(path);
```

4.3.5 `symlink` Function

The `symlink()` function creates a symbolic link called `path2` that contains the string pointed to by `path1`. (`path2` is the name of the symbolic link created, and `path1` is the string contained in the symbolic link.) The string pointed to by `path1` is treated only as a character string and is not validated as a pathname. If the `symlink()` function fails for any reason other than EIO error, any file named by `path2` remains unaffected.

The function action, header file required and syntax of the `symlink()` function are as follows:

Function action

`symlink` – makes a symbolic link to a file.

Header file required

```
#include<unistd.h>
```

Syntax

```
int symlink(const char *path1, const char *path2);
```

Return values: Upon successful completion, `symlink()` returns 0. Otherwise, it will return −1 and set `errno` to indicate the error.

Errors: The `symlink()` function will fail if the following errors occur:

EACCES

Write permission is denied in the directory where the symbolic link is being created, or search permission is denied for a component of the `path` prefix of `path2`.

EEXIST

The `path2` argument names an existing file or symbolic link.

EIO

An I/O error occurs while reading from or writing to the file system.

ELOOP

A loop exists in symbolic links encountered during resolution of the `path2` argument.

ENAMETOOLONG

The length of the `path2` argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX} or the length of the `path1` argument is longer than {SYMLINK_MAX}.

ENOENT

A component of `path2` does not name an existing file, or `path2` is an empty string.

ENOSPC

The directory in which the entry for the new symbolic link is being placed cannot be extended because no space is left on the file system containing the directory, or the new symbolic link cannot be created because no space is left on the file system which shall contain the link, or the file system is out of file-allocation resources.

ENOTDIR

A component of the `path` prefix of `path2` is not a directory.

EROFS

The new symbolic link would reside on a read-only file system.

ELOOP

More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the `path2` argument.

ENAMETOOLONG

As a result of encountering a symbolic link in resolution of the `path2` argument, the length of the substituted pathname string exceeded {PATH_MAX} bytes (including the terminating null byte), or the length of the string pointed to by `path1` exceeded {SYMLINK_MAX}.

4.3.6 `mkdir` Function

The `mkdir()` function creates a new directory with name path. The file permission bits of the new directory are initialized from `mode`. These file permission bits of the `mode` argument shall be modified by the process's file creation mask. When bits in `mode` other than the file permission bits are set, the meaning of these additional bits is implementation-defined.

The function action, header file required and syntax of the `mkdir()` function are as follows:

Function action

`mkdir` – creates a new directory with name path.

Header file required

```
#include <sys/stat.h>
```

Syntax

```
int mkdir(const char *path, mode_t mode);
```

The directory's user ID is set to the process's effective user ID. The directory's group ID is set to the group ID of the parent directory or to the effective group ID of the process. Implementations provide a way to initialize the directory's group ID to the group ID of the parent directory. Implementations may, but need not, provide an implementation-defined way to initialize the directory's group ID to the effective group ID of the calling process.

The newly created directory is an empty directory.

If `path` names a symbolic link, the `mkdir()` function fails and `errno` is set to EEXIST.

Upon successful completion, the `mkdir()` function marks for update the `st_atime, st_ctime,` and `st_mtime` fields of the directory. Also, the `st_ctime` and `st_mtime` fields of the directory that contains the new entry are marked for update.

Return values: Upon successful completion, the `mkdir()` function returns 0. Otherwise, −1 is returned, and no directory is created, and `errno` is set to indicate the error.

Errors: The `mkdir()` function will fail if the following errors occur:

EACCES

Search permission is denied on a component of the `path` prefix, or write permission is denied on the parent directory of the directory to be created.

EEXIST

The named file exists.

ELOOP

A loop exists in symbolic links encountered during resolution of the `path` argument.

EMLINK

The link count of the parent directory would exceed {LINK_MAX}.

ENAMETOOLONG

The length of the `path` argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

ENOENT

A component of the `path` prefix specified by `path` does not name an existing directory, or `path` is an empty string.

ENOSPC

The file system does not contain enough space to hold the contents of the new directory or to extend the parent directory of the new directory.

ENOTDIR

A component of the `path` prefix is not a directory.

EROFS

The parent directory resides on a read-only file system.

ELOOP

More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the `path` argument.

ENAMETOOLONG

As a result of encountering a symbolic link in resolution of the `path` argument, the length of the substituted pathname string exceeded {PATH_MAX}.

Prog. 12 How to create a directory?

The following program creates a directory named `/home/alp/dir1`, with read/write/search permissions for owner and group, and with read/search permissions for others.

```
#include <sys/types.h>
#include <sys/stat.h>

int status;
...
status = mkdir("/home/alp/dir1", S_IRWXU | S_IRWXG | S_IROTH
| S_ IXOTH);
```

4.3.7 `rmdir` Function

The `rmdir()` function removes a directory whose name is given by `path`. The directory is removed only if it is an empty directory.

The function action, header file required and syntax of the `rmdir()` function are as follows:

Function action

`rmdir` – removes a directory.

Header file required

`#include <unistd.h>`

Syntax

```
int rmdir(const char *path);
```

If the directory is the root directory or the current working directory of any process, it is unspecified whether the function succeeds, or whether it fails and sets `errno` to EBUSY. If `path` names a symbolic link, then `rmdir()` function fails and `errno` is set to ENOTDIR. If the `path` argument refers to a path whose final component is either dot or dot-dot, the `rmdir()` function shall fail.

If the directory's link count becomes 0 and no process has the directory open, the space occupied by the directory is freed, and the directory is no longer accessible. If one or more processes have the directory open when the last link is removed, the dot and dot-dot entries, if present, are removed before the `rmdir()` function returns and no new entries may be created in the directory, but the directory shall not be removed until all references to the directory are closed.

If the directory is not an empty directory, the `rmdir()` function fails and `errno` is set to EEXIST or ENOTEMPTY.

Upon successful completion, the `rmdir()` function shall mark for update the `st_ctime` and `st_mtime` fields of the parent directory.

Return values: Upon successful completion, the `rmdir()` function shall return 0. Otherwise, −1 is returned, and `errno` is set to indicate the error. If −1 is returned, the named directory is not be changed.

Errors: The `rmdir()` function will fail if the following errors occur:

EACCES

Search permission is denied on a component of the `path` prefix, or write permission is denied on the parent directory of the directory to be removed.

EBUSY

The directory to be removed is currently in use by the system or some process, and the implementation considers this to be an error.

EEXIST or ENOTEMPTY

The `path` argument names a directory that is not an empty directory, or there are hard links to the directory other than dot or a single entry in dot-dot.

EINVAL

The `path` argument contains a last component that is dot.

EIO

A physical I/O error has occurred.

ELOOP

A loop exists in symbolic links encountered during resolution of the `path` argument.

ENAMETOOLONG

The length of the `path` argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

ENOENT

A component of `path` does not name an existing file, or the `path` argument names a non-existent directory or points to an empty string.

ENOTDIR

A component of `path` is not a directory.

EPERM or EACCES

S_ISVTX flag is set on the parent directory of the directory to be removed, and the caller is not the owner of the directory to be removed, nor is the caller the owner of the parent directory, nor does the caller have the appropriate privileges.

EROFS

The directory entry to be removed resides on a read-only file system.

ELOOP

More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the `path` argument.

ENAMETOOLONG

As a result of encountering a symbolic link in resolution of the `path` argument, the length of the substituted pathname string exceeded {PATH_MAX}.

Prog. 13 How to remove a directory?

The following program removes a directory named `/home/alp/dir1`.

```
#include <unistd.h>
int status;
...
status = rmdir("/home/alp/dir1");
```

4.3.8 `chdir` Function

The `chdir()` function causes the directory named by the pathname, pointed to by the `path` argument, to become the current working directory; that is, the starting point for path searches for pathnames not beginning with '/'.

The function action, header file required and syntax of the `chdir()` function are as follows:

Function action

`chdir` – changes working directory.

Header file required

`#include<unistd.h>`

Syntax

```
int chdir(const char *path);
```

Return values: Upon successful completion of the `chdir()` function, 0 is returned. Otherwise, −1 is returned, the current working directory remains unchanged, and `errno` is set to indicate the error.

Errors: The `chdir()` function will fail if the following errors occur:

EACCES

Search permission is denied for any component of the pathname.

ELOOP

A loop exists in symbolic links encountered during resolution of the `path` argument.

ENAMETOOLONG

The length of the `path` argument exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

ENOENT

A component of `path` does not name an existing directory or `path` is an empty string.

ENOTDIR

A component of the pathname is not a directory.

ELOOP

More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the `path` argument.

ENAMETOOLONG

As a result of encountering a symbolic link in resolution of the `path` argument, the length of the substituted pathname string exceeded {PATH_MAX}.

Prog. 14 How to change the current working directory?

The following program makes the value pointed to by directory, `/tmp`, the current working directory.

```
#include <unistd.h>
...
char *directory = "/tmp";
int ret;

ret = chdir (directory);
```

4.3.9 `getcwd` Function

The `getcwd()` function places an absolute pathname of the current working directory in the array pointed to by `buf,` and returns `buf.` The pathname copied to the array contains no components that are symbolic links. The `size` argument is the size in bytes of the character array pointed to by the `buf` argument. If `buf` is a null pointer, the behaviour of the `getcwd()` function is unspecified.

The function action, header file required and syntax of the `getcwd()` function are as follows:

Function action

`getcwd` – gets the pathname of the current working directory.

Header file required

```
#include <unistd.h>
```

Syntax

```
char *getcwd(char *buf, size_t size);
```

Return values: Upon successful completion, the `getcwd()` function returns the `buf` argument. Otherwise, the `getcwd()` function returns a null pointer and `errno` is set to indicate the error. The contents of the array pointed to by `buf` are then undefined.

Errors: The `getcwd()` function will fail if the following errors occur:

EINVAL

The `size` argument is 0.

ERANGE

The `size` argument is greater than 0, but is smaller than the length of the pathname +1.
The `getcwd()` function may fail if:

EACCES

Read or search permission was denied for a component of the pathname.

ENOMEM

Insufficient storage space is available.

Prog. 15 How to determine the absolute pathname of the current working directory?

The following program returns a pointer to an array that holds the absolute pathname of the current working directory. The pointer is returned in the `ptr` variable, which points to the `buf` array where the pathname is stored.

```
#include <stdlib.h>
#include <unistd.h>
...
long size;
char *buf;
char *ptr;
size = pathconf(".", _PC_PATH_MAX);
if ((buf = (char *)malloc((size_t)size)) != NULL)
ptr = getcwd(buf, (size_t)size);
...
```

4.3.10 access Function

When accessing a file with the `open()` function, the kernel performs its access tests based on the effective user ID and effective group ID to test the access permissions of a file using `access()` function:

Function action

`access` – to test the access permissions

Header file required

```
#include <unistd.h>
```

Syntax

```
int access(char *cfile, int mode);
```

where `cfile` points to a cfile name naming a file and `mode` is the bitwise OR of any of the constants, that is `R_OK` – test for read permission, `W_OK` – test for write permission , `X_OK` – test for execute or search permission , `F_OK` – test whether the directories leading to the file can be searched and the file exists

Return values: Upon completion, it returns 0 if OK , −1 on error and sets errno to indicate the error.

4.4 Summary

This chapter is the first step to enter the great and exciting field of UNIX internals. It begins with explaining the concept of file descriptor. It specifies the file and directory controls in three classified fields: file access functions, file control and directory maintenance. This chapter discusses in detail different functions, outlining their arguments, return values and error conditions.

5 Process

Before going into UNIX processes let us brush up on some fundamentals. You may be knowing that a process is an invocation or activation of a program. A program is a list of instructions for execution by a computer. A program may be stored in any convenient form including handwritten notes on pieces of paper but is most commonly stored in binary form in a soft file. To run the program, it needs to be copied (or loaded) into the main computer memory, and the central processing unit (CPU) told to start reading (and obeying) instructions taken from that area of memory. The activity of executing the program's instructions is called running the process. In other words, a process is an instance of running a program. It is also a sequence of bytes interpreted as instructions to be run by the CPU.

5.1 Process Structure

UNIX is a time-sharing system, which means that the processes take turns in running, and each turn is a called a time-slice. On most systems, the time-slice is set at much less than one second. The operating system must interleave the execution of several processes to maximize processor utilization while providing reasonable response time to each process. The operating system must allocate resources to processes according to a specific policy to avoid deadlock in process execution while taking care of those applications that need to be run on priority. The operating system may be required to support inter-process communication and user creation of processes, both of which may aid in the structuring of applications.

A process in an operating system is represented by a data structure known as a process control block (PCB) or process descriptor shown in Figure 5.1. The process image as viewed by the kernel runs in its own user address space that is a protected space, and it cannot be shared by other users. This address space has three main segments that provide information about process identification, processor state and process control. We can group the PCB information into three general categories:

1. Process identification.
2. Process state information.
3. Process control information.

5.1.1 Process Identification

UNIX identifies every process by a process identification number (PID) which is assigned when the process is initiated. When we want to perform an operation on a process, we usually refer to it by its PID. Numeric identifiers that may be stored in the process control block include the identifier of a particular process, the identifier of the process (parent process) that created this process, and the user identifier. There will inevitably be a significant amount of information about a process that is relevant

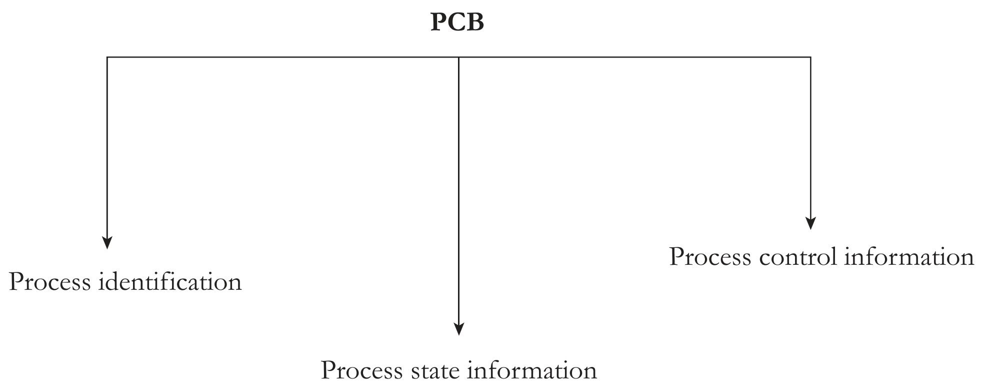


Figure 5.1 Process control block.

only to the kernel. This will include data such as the process user-id (for access rights checks), pointers to open file information blocks, saved register states for processes not currently running, process environment, etc. Some of this information is stored in an entry in the process table which has an entry for every current process, and some information is stored in a per-process memory area called the `u_area` on UNIX systems.

Virtually in all operating systems, each process is assigned a unique numeric identifier that may simply be an index into the primary process table. When processes communicate with one another, the process identifier informs the operating system of the destination of a particular communication. When processes are allowed to create other processes, identifiers indicate the parent and descendants of each process. Given below are identifiers of some processes:

1. Special processes have IDs 0, 1 and 2.
2. Swapper process has an ID of 0.
3. Init process has an ID of 1.
4. Pagedaemon process has an ID of 2.

As listed above, the swapper process has a PID of 0. Also known as the scheduler process, the swapper process is a system process (part of the kernel), and it is created by the kernel at the system booting time.

The process with PID 1 (initialization process) is invoked by the kernel at system start-up. The program is stored on the disk at `/etc/init` or `/sbin/init` file. This process never terminates and is a normal user process that adopts all the orphan processes.The process with PID 2 (pagedaemon process) is found on virtual memory systems. It is responsible for supporting demand paging. It is a kernel process and never terminates.

Apart from process IDs, there are also other identifiers for every process. The following functions return these identifiers.

`pid_t getpid(void)`	Returns the process ID of the calling process.
`pid_t getppid(void)`	Returns the parent process ID of the calling process.
`uid_t getuid(void)`	Returns the real user ID of the calling process.
`uid_t geteuid(void)`	Returns the effective user ID of the calling process.

`gid_t getgid(void)`	Returns the real group ID of the calling process.
`gid_t getegid(void)`	Returns the effective group ID of the calling process.

`getpid` function

The `getpid()` function returns the process ID of the calling process, when completed. This call always returns successfully. The function action, header files required and syntax of the `getpid()` function are as follows:

Function action

`getpid` – returns the process ID of the calling process.

Header file required

```
#include<sys/types.h>
#include<unistd.h>
```

Syntax

```
pid_t getpid(void);
```

5.1.2 Process State Information

The process state information can be taken by these parameters: user-visible registers, control and status registers and stack pointers. A user-visible register is one that may be referenced by means of the machine language that the processor executes. Control and status registers are a variety of processor registers that are employed to control the operation of the processor. A control register comprises a program counter (that contains the address of the next instruction to be fetched) and condition codes (that are the result of the most recent arithmetic or logical operations (e.g. sign, zero, carry, equal and overflow). The status information includes interrupt enabled/disabled flags, and execution mode. Each process has one or more last in first out (LIFO) system stacks associated with it. A stack is used to store parameters and the calling address for procedures and system calls. The stack pointer points to the top of the stack.

A process can be in any of the following nine distinct states:

1. **Created:** This is the state of a freshly created process. Whether freshly created processes entirely reside in the memory depends on the details of the memory management system. The created state may also include the processes that have not yet been fully created.
2. **Ready to run, in memory:** This is the state in which a process is in the ready state for execution. It is important to note that there is no reason, apart from the fact that some other process is currently running, why the process should not run.
3. **Running in kernel mode:** In this mode, a process may be handling a system call or an interrupt. Or, some other process (also running in the kernel mode) may have scheduled the process to run. The process may determine that it has finished (either normally via an `exit()` or via some kernel-detected abnormal condition) or that it is blocked awaiting some event such as a time signal or peripheral activity.
4. **Running in user mode:** This is the normal state of a process.
5. **Pre-empted:** In this state, the process has been interrupted and is about to resume normal user mode operation. The kernel scheduler may move a process into this state.

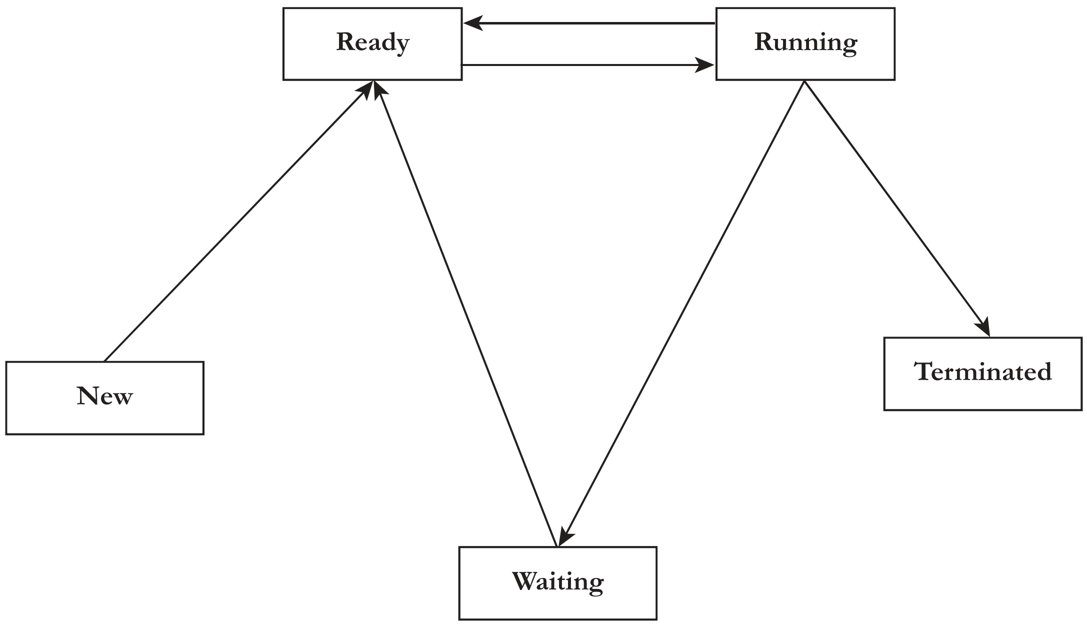


Figure 5.2 Different process states.

6. **Zombie (or defunct):** In this state, the process will not run again, but information, like the exit code, has not been collected by the parent process, so a process table slot is still required to store this information.
7. **Sleeping in memory:** This is the state, when the process is blocked awaiting an event. All that can happen is that the process can be woken up (by changing its status to 'ready to run') or swapped out.
8. **Sleeping, swapped out:** In this state, the process is waiting for an event (details in the process table entry for the process) and has been swapped out.
9. **Ready to run, swapped out:** This is the state in which the process is in ready mode for execution, and it is transferred from memory to the CPU for further execution.

5.1.3 Process Control Information

The process control information is needed by the operating system to perform its scheduling function. This information is generally about (a) process state, (b) process priority and (c) identity of the event the process is awaiting before it can be resumed. The process state (Figure 5.2) defines the readiness of the process to be scheduled for execution (e.g. running, ready, waiting and halted). To describe the scheduling priority of a process, one or more fields may be used.

5.2 `exit()` and `_exit()` Functions

`exit()` function

The `exit()` function causes the normal termination of a program. If a program executes more than one call to the `exit()` function, the behaviour is undefined.

The function action, header file required and syntax of the `exit()` function are as follows:

Function action

`exit` – causes normal termination.

Header file required

```
#include<stdlib.h>
```

Syntax

```
void exit(int status);
```

The `exit()` function works in the following way. First, `exit()` calls all functions registered by that `exit()` function in the reverse order of their registration. Next, `exit()` flushes all open streams with unwritten buffered data, and closes all open streams. Last, `exit()` returns control to the host environment by calling `_exit(status)`. If the value of status is zero or EXIT_SUCCESS, `exit()` returns an implementation-defined form of the status successful termination. If the value of status is EXIT_FAILURE, it returns an implementation-defined form of the status unsuccessful termination. The `exit()` function has the effect of `fclose()` on every open stream, with the properties of `fclose()`. For maximum portability, use only the EXIT_SUCCESS and EXIT_FAILURE macros for `status`. Here `status` indicates the value to be returned to the parent process. The `exit()` function cannot return to its caller.

`_exit()` function

The `_exit()` function terminates a process. If a process terminates for any reason, this function performs the following actions: `_exit()` closes any open file descriptors or handles. If the parent process of the calling process is executing a `wait()` or `waitpid()`, `_exit()` notifies the parent that the calling process is to be terminated, and the low-order eight bits of status are made available to it.

The function action, header file required and syntax of the `_exit()` function are as follows:

Function action

`_exit` – terminates a process.

Header file required

```
#include<unistd.h>
```

Syntax

```
void _exit(int status);
```

If the parent process of the calling process is not executing a `wait()` or `waitpid()` function, `_exit()` saves the exit status code for return to the parent process whenever the parent process executes an appropriate subsequent `wait()` or `waitpid()`.

Terminating a process does not directly terminate its children. Sending a SIGHUP signal indirectly terminates children in some circumstances. `_exit()` assigns a new parent process ID (corresponding to an implementation-defined system process) to the children of a terminated process. Its `status` represents the termination status. It never returns.

5.3 `fork()` Function

The `fork()` function creates a new process. The new process (child process) shall be an exact copy of the calling process (parent process).

The function action, header file required and syntax of the `fork()` function are as follows:

Function action

`fork` – creates a new process.

Header file required

```
#include<unistd.h>
```

Syntax

```
pid_t fork(void);
```

We can differentiate the newly created process and calling process on the basis of the following unique qualities:

1. The child process shall have a unique process ID.
2. The child process ID shall not match any active process group ID.
3. The child process shall have a diferent parent process ID, which shall be the process ID of the calling process.
4. The child process shall have its own copy of the parent's file descriptors. Each of the child's file descriptors shall refer to the same open file description with the corresponding file descriptor of the parent.
5. The child process shall have its own copy of the parent's open directory streams. Each open directory stream in the child process may share directory stream positioning with the corresponding directory stream of the parent.
6. The child process shall have its own copy of the parent's message catalog descriptors.
7. After `fork()` execution, as shown in Figure 5.3, both the parent and the child processes shall be capable of executing independently before either one terminates.

Return values: Upon successful completion, `fork()` shall return 0 to the child process and the process ID of the child process to the parent process. Both processes shall continue to execute from the `fork()` function. Otherwise, −1 shall be returned to the parent process, no child process shall be created, and `errno` shall be set to indicate the error.

Errors: The `fork()` function shall fail if the following errors occur:

EAGAIN

The system lacked the necessary resources to create another process, or the system-imposed limit on the total number of processes under execution system-wide or by a single user {CHILD_MAX} would be exceeded.

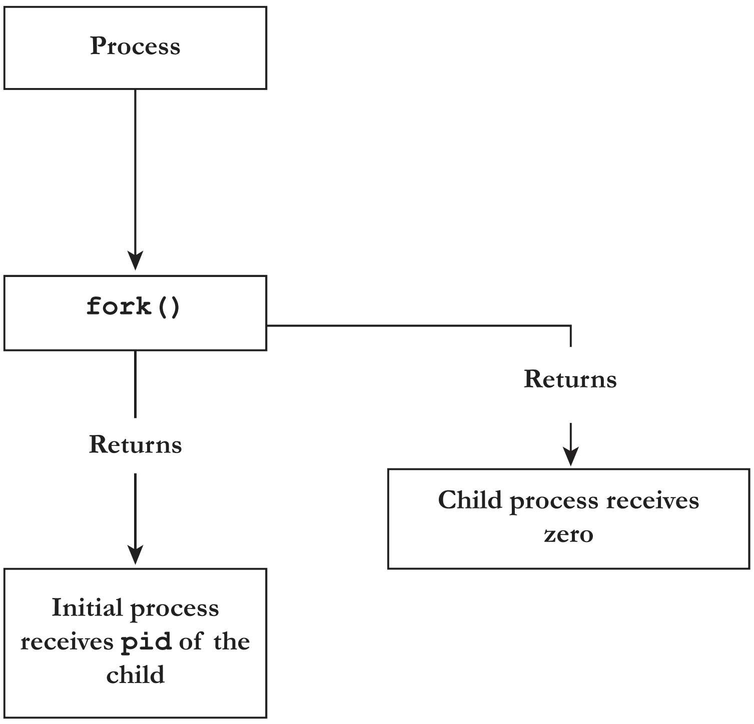

Process
fork()
Returns
Returns
Child process receives zero
Initial process receives pid of the child

Figure 5.3 The `fork()` function execution.

ENOMEM

Insufficient storage space is available.

Let us now do some programming exercises.

Prog. 1

This program demonstrates the working of the `fork()` function by displaying the statement 'Join The Unique UNIX Caravan' twice on screen. In this program, the `fork()` function creates a child that is a duplicate of the parent process. As there are two identical processes in memory, the 'Join The Unique UNIX Caravan' statement is printed twice. The child process begins from the `fork()` function. All the statements after the call to `fork()` will be executed twice – once by the parent process and once by the child process. But had there been any statements before the `fork()` function, they would be executed only by the parent process.

```
#include<sys/type.h>
#include<stdio.h>
  main()
  {
    fork();
    printf(" Join The Unique UNIX Caravan\n"):
}
```

Prog. 2

In the following program, the 'Join The UNIX Caravan' statement is printed twice. The statement above the `fork()` function will be displayed only once since it comes before the creation of the child process. This `fork()` function is called once but returns twice. The only difference in the two returns is that the return value in the child is 0 while that in the parent is the process ID of the new child. The reason the child's process ID is returned to the parent is that a process can have more than one child, so no function allows a process to obtain the process IDs of its children. The reason `fork()` returns 0 to the child is that a process can have only a single parent, so the child can always call `getppid()` to obtain the process ID of its parent.

```
main()
{
    printf("Demo the fork()\n"):
    fork();
    printf("Join The UNIX Caravan\n");
}
```

Prog. 3

If we have two calls of `fork()` instead of one call in the above program then consider the following example:

```
main()
{
    fork();
    fork();
    printf("'Join The Unique UNIX Caravan'\n");
}
```

In the above program, instead of two processes, three processes will be created. This gives a total of four processes – in memory, the parent, two children and one grandchild. The first call to `fork()` creates one child process. Now there are two processes. Both processes begin executing from the second call to `fork()` thus giving us a total of four processes.

Sometimes, `fork()` may not create the child process due to some memory problem. In that case, variable `pid` of the parent process will reflect a −1, indicating an error.

Prog. 4

In this program we are checking whether `fork()` function has been executed successfully or not. It is not only the parent process that can register its child's PID. The child process too can figure out who its parent is. The `getppid()` function will tell us that the child is copy of parent.

```
main()
{
    int pid;
    pid=fork();
if (pid < 0)
   printf(" failure of fork()\n "):
else
   printf(" success of fork()\n"):
}
```

Prog. 5

In this program the `fork()` function will result in two processes being in memory: a parent and its child. The `if-else` statements will be executed in both the processes. In the child process, the variable `pid` will reflect a 0. Therefore we will get its (the child's) as well as its parent's PID number. In the parent process, the else part will be executed. This will give us the parent's PID number. In this case, the parent of the parent process is the shell.

```
main()
  {
    int pid;
    pid=fork();
    if(pid==0)
  {
    printf ("I am the child and my process ID is %d\n", getpid());
    printf ("the child's parent process ID is %d\n", getppid());
  }
  else
    {
    printf ("I am the parent, my process ID is %d\n, getpid());
    printf ("the parents parent process ID is %d\n", getppid());
    }
  }
```

Prog. 6

Let us consider the following program that uses `fork()` system call to create a child process that prints 'Join the unique Unix Caravan' to the screen, and exits.

```
#include <unistd.h>    /* defines fork(), and pid_t.      */
#include <sys/wait.h>  /* defines the wait() system call. */

/* storage place for the pid of the child process, and its exit
status. */
pid_t child_pid;
int child_status;

/* lets fork off a child process... */
child_pid = fork();

/* check what the fork() call actually did */
switch (child_pid) {
 case -1:   /* fork() failed */
```

```
      perror("fork"); /* print a system-defined error message */
      exit(1);
   case 0: /* fork() succeeded, we're inside the child process */
      printf("Join the unique Unix Caravan \n");
      exit(0); /* here the CHILD process exits, not the parent. */
   default: /* fork() succeeded, we're inside the parent process */
      wait(&child_status);  /* wait till the child process exits */
  }
  /* parent's process code may continue here... */
```

In Prog. 6 it is important to note that the `perror()` function prints an error message, based on the value of the `errno` variable, to `stderr`. The `wait()` system call waits until any child process exits, and stores its exit status in the variable supplied.

5.4 `wait()` Function

The `wait()` function lets the calling process obtain status information about one of its child processes. If the status information is available for two or more child processes, the order in which their status is reported is unspecified. The `wait()` function blocks the calling process until one of its child processes exits or a signal is received. A primary function of `wait()` is to wait for the completion of child processes. The `wait()` function takes the address of an integer variable and returns the process ID of the completed process. Some flags that indicate the completion status of the child process are passed back with the integer pointer.

The function action, header files required and syntax of the `wait()` function are as follows:

Function action

`wait` – lets the calling process obtain status information about one of its child processes.

Header file required

```
#include <sys/types.h>
#include <sys/wait.h>
```

Syntax

```
pid_t wait(int *stat_loc);
```

The execution of `wait()` can have any of the two following possible situations:

1. If there is at least one child process running when the call to `wait()` is made, the caller will be blocked until one of its child processes exits. The moment the child process exits, the caller resumes its execution.
2. If there is no child process running when the call to `wait()` is made, then this `wait()` has no effect at all. That is, it is as if no `wait()` is present.

The `stat_loc` can be the parameter for `wait()`. It specifies the location where the child process's exit status is stored. If NULL is passed, no exit status is returned. Otherwise, the following macros defined in `<sys/wait.h>` can be used to evaluate the returned status:

WIFEXITED

Evaluates to a non-zero value if status was returned for a child process that exited normally.

WEXITSTATUS

If the value of WIFEXITED is non-zero, this macro evaluates to the low-order bits of the status argument that the child process passed to `exit()` or `_exit()`, or the value the child process returned from `main()`.

WIFSIGNALED

Evaluates to a non-zero value if status was returned for a child process that terminated due to receipt of a signal that was not caught.

WTERMSIG

If the value of WIFSIGNALED is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFCORED

Evaluates to a non-zero value if status was returned for a child process that terminated due to receipt of a signal that was not caught, and whose default action is to dump core.

WCOREDUMP

Evaluates to a non-zero value if status was returned for a child process that terminated due to receipt of a signal that was not caught, and whose default action is to dump core.

WCORESIG

If the value of WIFCORED(s) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED(s)

Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG(s)

If the value of WIFSTOPPED(s) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

Return values: On successful completion, `wait()` returns the process ID of a child when the status of that child is available. Otherwise, it returns −1 and `errno` is set to indicate error.

Errors: The `wait()` function will fail in case of the following errors:

ECHILD

The process or process group specified by PID does not exist or is not a child of the calling process.

EFAULT

`stat_loc` is not a writable address.

EINTR

The function was interrupted by a signal. The value of the location pointed to by `stat_loc` is undefined.

Let us now understand the use of `wait()` through the following programming exercise.

Prog. 7

In this programming exercise, the main program creates two child processes to execute the same printing loop and display a message before exit. The parent process (i.e. the main program), after creating two child processes, enters the wait state by executing the system call `wait()`. Once a child exits, the parent starts execution and the ID of the terminated child process is returned in PID so that it can be printed. There are two child processes and thus two `wait()`s, one for each child process.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>

#define MAX_COUNT 200
#define BUF_SIZE  100

void ChildProcess(char [], char []); /*child process prototype */

void main(void)
{
  pid_t  pid1, pid2, pid;
  int  status;
  int  i;
  char buf[BUF_SIZE];

  printf("*** Parent is about to fork process 1 ***\n");
  if ((pid1 = fork()) < 0) {
    printf("Failed to fork process 1\n");
    exit(1);
  }
  else if (pid1 == 0)
    ChildProcess("First", " ");
  printf("*** Parent is about to fork process 2 ***\n");
  if ((pid2 = fork()) < 0) {
    printf("Failed to fork process 2\n");
    exit(1);
  }
  else if (pid2 == 0)
    ChildProcess("Second", " ");
  printf(buf, "*** Parent enters waiting status .....\n");
  write(1, buf, strlen(buf));
  pid = wait(&status);
  printf(buf, "*** Parent detects process %d was done ***\n",
  pid);
  write(1, buf, strlen(buf));
  pid = wait(&status);
  printf("*** Parent detects process %d is done ***\n", pid);
  printf("*** Parent exits ***\n");
  exit(0);
}
void ChildProcess(char *number, char *space)
{
  pid_t pid;
  int  i;
  char  buf[BUF_SIZE];
  pid = getpid();
  printf(buf, "%s%s child process starts (pid = %d)\n",
    space, number, pid);
  write(1, buf, strlen(buf));
  for (i = 1; i <= MAX_COUNT; i++) {
    printf(buf, "%s%s child's output, value = %d\n", space,
    number, i);
    write(1, buf, strlen(buf));
  }
  printf(buf, "%s%s child (pid = %d) is about to exit\n",
    space, number, pid);
  write(1, buf, strlen(buf));
  exit(0);
}
```

5.5 Orphan Process

Like the real life, UNIX too supports the concept of orphans. In UNIX, after execution of the `fork()` function, if the parent terminates first, the child is helpless. This creates the situation of orphans. Normally after the execution of the `fork()` function, a time-slice is given to the child process in which the child process should be executed first. But if the parent process were to terminate before the child process, then unlike the real life where some orphans remain parentless in their entire life, the UNIX creators arranged for a policy of adoption. Let us see in the following example program a situation that simulates an orphan process.

Prog. 8

```
 main()
{
 int pid;
 pid = fork();
 if(pid = =0)
 {
 printf("I am the child, my process ID is %d\n", getpid());
 printf("the child's parent process ID is %d\n", getppid());
 sleep(30);
 printf("I am the child, my process ID is %d\n", getpid());
 printf("the child's parent process ID is %d\n", getppid());
 }
 else
 {
 printf("I am the parent, my process ID is %d\n", getpid());
 printf("the parents parent process ID is %d\n", getppid());
 }
}
```

In Prog. 8, a child process will be created when the `fork()` statement is encountered. If the IF condition evaluates to true, the statements in it will be executed. This means that the child process is active. The two calls to the `printf()` function will print the child processes as well as its parent's PID. At this point, the child process will be put to `sleep()`. This means that for 30 seconds, the child process will remain inactive. It will not get the time-slice in this period. In the meanwhile, the parent process will be executed. This results in the parent process's and its parent's (the shell's) PID being displayed on screen. After which the parent process will terminate. Now we have a parentless child process in memory. Once its sleep wears off, the next two `printf()`s are executed. The first displays the same PID of the child as was shown earlier. But the next `printf()` will not print the PID of the process dispatcher(1) because this process dispatcher immediately adopts an orphan.

5.6 Zombies

The UNIX concept of Zombies applies to processes that are dead but have not been removed from the process table. In other words, a *zombie process* is a process that has terminated but has not been cleaned up yet. Assume that a parent process creates a child. Both have an entry in the process table. Assume further that the child process is terminated well before the parent does. Because the parent process is yet in action, the child process cannot be removed from the process table. It, therefore, exists in the twilight zone as a zombie process.

It is the responsibility of the parent process to clean up its zombie children. The wait functions work for cleaning the zombie children, so it's not necessary to track whether your child process is still executing before waiting for it. Suppose a program forks a child process, performs some other computations and then calls `wait()`. If the child process has not terminated at that point, the parent process will block in the wait call until the child process finishes. If the child process finishes before the parent process calls `wait()`, the child process becomes a zombie. When the parent process calls `wait()`, the zombie child's termination status is extracted, the child process is deleted and the `wait()` call returns immediately. Let us work with the following program that will simulate the situation of zombie process.

Prog. 9

```
main()
    {
        if (fork()>0)
        {
      printf("parent\n"):
      sleep(100);
      }
      }
```

Run Prog. 9 as a background process (`a.out &`). Once you are at the UNIX prompt, type `ps -el`. You will see in the process table an entry with a Z for zombie in the second column, and a `<defunct>` in the last column. The child in the last column is nothing but a duplicate of the parent process. Execution of both processes starts from the line after the `fork()` function. In the above program, that line is an IF condition which checks whether the value returned by `fork()` is greater than 0. Since this condition is not valid for a child process, the child process terminates. But, as the parent process is yet in memory because of the `sleep()` function, the process table remains unrectified.

5.7 `vfork()` Function

The `vfork()` function creates a new process as does `fork()` except that the child process in the `vfork()` function shares the same address space as the calling process. The `vfork()` function has the same effect as the `fork()` function except that the behaviour is undefined if the process created by the `vfork()` function either modifies any data other than a variable of type `pid_t` used to store the return value from the `vfork()` function, or returns from the function in which the `vfork()` function was called, or calls any other function before successfully calling the `_exit()` function or any `exec()` family function. Execution of the calling process is blocked until the child process calls an `exec()` family function or calls the`_exit()` functions. Because the parent and child share the address space, you must not return from the function that called the `vfork()` function; doing so can corrupt the parent's stack. You should use the `vfork()` function when your child process simply modifies the process state and then calls an `exec()` function. Because of the shared address space, you must avoid doing anything in the child that impacts the parent when it resumes execution. For example, if your `exec()` function call fails, you must call the `_exit()` function, not the `exit()` function, because calling the `exit()` function would close standard I/O stream buffers for the parent as well as the child.

The function action, header file required and syntax of the `vfork()` function are as follows:

Function action

`vfork` – creates a new process that shares the same address space.

Header file required

```
#include <unistd.h>
```

Syntax

```
pid_t vfork(void);
```

The `vfork()` function differs from `fork()` only in that the child process can share code and data with the calling process (parent process). This speeds cloning activity significantly at a risk to the integrity of the parent process if `vfork()` is misused.

The use of `vfork()` for any purpose except as a prelude to an immediate call to a function from the `exec()` family, or to `_exit()`, is not advised.

The `vfork()` function can be used to create new processes without fully copying the address space of the old process. If a forked process is simply going to call `exec()`, the data space copied from the parent to the child by `fork()` is not used. This is particularly inefficient in a paged environment where `vfork()`proves highly useful. Depending on the size of the parent's data space, `vfork()` can give a significant performance improvement over `fork()`.

The `vfork()` function can normally be used just like `fork()`. However, it does not work to return while running in the child's context from the caller of `vfork()` because the eventual return from `vfork()` would then return to a no longer existent stack frame. Be careful also to call `_exit()` rather than `exit()` if you cannot `exec()` because `exit()` flushes and closes standard I/O channels, thereby damaging the parent process's standard I/O data structures. (Even with `fork()`, it is wrong to call `exit()` because buffered data would then be flushed twice.)

Return values: Upon successful completion, `vfork()` returns 0 to the child process and returns the process ID of the child process to the parent process. Otherwise, −1 is returned to the parent, no child process is created, and `errno` is set to indicate the error.

Errors: The `vfork()` function will fail in case of the following errors:

EAGAIN

The system-wide limit on the total number of processes under execution would be exceeded, or the system-imposed limit on the total number of processes under execution by a single user would be exceeded.

ENOMEM

There is insufficient swap space for the new process.

Let us work with the following program to understand the use of the `vfork()` function.

Prog. 10

In this program, we use the `vfork()` function for ensuring that every time the child process should run first and will share the same address space as the calling process.

```
# include<sys/types.h>
# include<unistd.h>
# include<stdio.h>
# include<stdlib.h>
int uni=8;
int main()
{  int x,n;
   pid_t pid;
   x=97;
   printf("before fork\n");
   if ((pid=vfork())<0)
  {  perror("fork error ");
     exit(1);
  }
  else if (pid==0)
  {  uni++;
     x++;
     _exit(0);
  }
  printf("pid = %d, uni = %d, x = %d\n",getpid(),uni,x);
  write(STDIN_FILENO, "unix",8);
  exit(0);
}
```

5.8 exec() Function

When a process calls an exec function, the new program starts executing at its main function. The process ID does not change across an `exec()` function because a new process is not created. The `exec()` function merely replaces the current process with a brand new program from disk. There are six different `exec()` functions, which means we could use any of the six different functions, that is, `execl()`, `execv()`, `execle()`, `execve()`, `execlp()` and `execvp()`.

The function action, header file required and syntax for the `exec()` functions are as follows:

Function action

`exec` – replaces the current process with a new program.

Header file required

```
#include <unistd.h>
```

Syntax

```
extern char **environ;
int execl(const char *path, const char *arg0, ... /*,
   (char *)0 */);
int execv(const char *path, char *const argv[ ]);
int execle(const char *path, const char *arg0, ... /*,
   (char *)0, char *const envp[ ]*/);
int execve(const char *path, char *const argv[ ],
   char *const envp[ ]);
int execlp(const char *file, const char *arg0, ... /*,
   (char *)0    */);
int execvp(const char *file, char *const argv[ ]);
```

The `exec()` family of functions shall replace the current process image with a new process image. The new image shall be constructed from a regular, executable file called the new process image file. There shall be no return from a successful `exec()` because the calling process image is overlaid by the new process image.

Let us consider all the `exec()` functions one by one:

```
int execl(const char *path, const char org(),… /(char *) 0 */);
```
Takes arguments in a list format.

```
int execv (const char *pathname, char *const argv[ ]);
```
Takes arguments as a pointer to an array of args.

`execle()` and `execve()`
```
int execle (const char *pathname, const char *arg(),….
                 /* (char*) 0, char *const envp[ ]*/);
```
Takes arguments in the list format.

```
int execve (const char *pathnamem char*const argv[], char *const
envp[]);
```
Takes arguments as a pointer to an array of args.

It is important to note that the `execle()` function and the `execve()` function require programmer to construct environment variables.

`execlp()` and `execvp()`

```
    int execlp(const char*filename, const char arg(), …..
    /* char  (*) 0/);
```

Takes arguments in a list format.

```
    int execvp (constant char *filename, const char arg(),…..
    /* char(*) 0/);
```

Takes arguments as pointer to an array of args.

The `execlp` and `execvp` functions use a PATH string to find the executable file.

Return values: All six `exec` functions return −1 on error, and no return on success.

Note: When a C-language program is executed as a result of this call, it shall be entered as a C-language function call as follows:

```
int main (int argc, char *argv[ ]);
```

where `argc` is the argument count, and `argv` is an array of character pointers to the arguments themselves. In addition, the following variable

```
extern char **environ;
```

is initialized as a pointer to an array of character pointers to the environment strings.

The `argv` and `environ` arrays are each terminated by a null pointer. The null pointer terminating the `argv` array is not counted in `argc`.

The arguments specified by a program with an `exec()` function are passed on to the new process image in the corresponding `main()` arguments.

The argument `path` points to a pathname that identifies the new process image file.

The argument `file` is used to construct a pathname that identifies the new process image file. If the `file` argument contains a slash character, the `file` argument shall be used as the pathname for this file.

Return values: If any `exec()` function returns to the calling process image, an error has occurred; the return value shall be −1, and `errno` shall be set to indicate the error.

Errors: The `exec()` functions shall fail in case of the following errors:

E2BIG

The number of bytes used by the new process image's argument list and environment list is greater than the system-imposed limit of {ARG_MAX} bytes.

EACCES

Search permission is denied for a directory listed in the new process image file's `path` prefix, or the new process image file denies execution permission, or the new process image file is not a regular file and the implementation does not support execution of files of its type.

EINVAL

The new process image file has the appropriate permission and has a recognized executable binary format, but the system does not support execution of a file with this format.

ELOOP

A loop exists in symbolic links encountered during resolution of the `path` or `file` argument.

ENAMETOOLONG

The length of the `path` or `file` arguments exceeds {PATH_MAX} or a pathname component is longer than {NAME_MAX}.

ENOENT

A component of `path` or `file` does not name an existing file, or `path` or `file` is an empty string.

ENOTDIR

A component of the new process image file's `path` prefix is not a directory.
The `exec()` functions, except for `execlp()` and `execvp()`, will also fail in case of the following errors:

ENOEXEC

The new process image file has the appropriate access permission but has an unrecognized format.

ELOOP

More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the `path` or `file` argument.

ENAMETOOLONG

As a result of encountering a symbolic link in resolution of the `path` argument, the length of the substituted pathname string exceeded {PATH_MAX}.

ENOMEM

The new process image requires more memory than is allowed by the hardware or system-imposed memory management constraints.

ETXTBSY

The new process image file is a pure procedure (shared text) file that is currently open for writing by some process.

Let us go through some program examples of `exec()` functions.

1. `environ`

Prog. 11

This program demonstrates to set the environment variables for the `exec()` functions.

```
# include<stdio.h>
int main(int argc,char *argv[])
{   int i;
    extern char **environ;
    char **ptr;
    for (i=0;i<argc;i++)
      printf("argv[%d]=%s\n",i,argv[i]);
      for (ptr=environ;*ptr!=NULL;ptr++)
      printf("%s\n",*ptr);
    exit(0);
}
```

2. `execl()`

Prog. 12

In this program using the `execl()` function we take arguments in a list format.

```
# include<stdio.h>
# include<unistd.h>
int main()
{
if (execl("./cmd_env","cmdarg","using","execl","function",NULL)<0)
{  perror("exec error");
   exit(1);
}
}
```

3. `execle()`

Prog. 13

In this program using the `execle()` function we take arguments in a list format.

```
# include <stdio.h>
# include <unistd.h>
int main()
{  char
*my_env[]={"HM=/home/anuj/unit3","LN=saurabh","UNAME=saurabh_
kumar","abc=xyz",NULL};
 if (execle("./cmd_env","env","using","execle","funtion",
 (char *)0,my_env)<0)
{  perror("exec error ");
   exit(1);
}
}
```

4. `execv()`

Prog. 14

In this program using the `execv()` function we take arguments as a pointer to an array of args.

```
# include<stdio.h>
# include<unistd.h>
int main()
{
char *carg[ ]={"cmdarg","hello","using","execv",NULL};
if (execv("./cmd_env",carg)<0)
{  perror("exec error ");
   exit(1);
}
}
```

5. `execve()`

Prog. 15

In this program using the `execve()` function we take arguments as a pointer to an array of args.

```
# include<stdio.h>
# include<unistd.h>
int main()
{
char *my_env[ ]={"HOME=/home/saurabh/unit3","LOGNAME=saurabh",
"USERNAME=saurabh_kumar",NULL};
       char *cmdarg[ ]={"env","using","execve","function",NULL};
if (execve("./cmd_env",cmdarg,my_env)<0)
{  perror("exec error ");
   exit(1);
}
}
```

5.9 `waitpid()` Function

The `waitpid()` function lets the calling process obtain status information about one of its child processes. If the status information is available for two or more child processes, the order in which their status is reported is unspecified. If more than one thread is suspended in `waitpid()` awaiting termination of the same process, exactly one thread returns the process status at the time of the target child process termination. The other threads return −1, with `errno` set to ECHILD.

If the calling process sets SIGCHLD to SIG_IGN, and the process has no unwaited-for children that were transformed into zombie processes, the calling thread blocks until all the children of the process terminate, at which time `waitpid()` returns −1 with `errno` set to ECHILD.

If the parent process terminates without waiting for all of its child processes to terminate, the remaining child processes are assigned a new parent process ID corresponding to a system-level process.

The function action, header files required and syntax of the `waitpid()` function are as follows:

Function action

`waitpid` – lets the calling process obtain status information about one of its child processes.

Header file required

```
#include<sys/types.h>
#include<sys/wait.h>
```

Syntax

```
pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

The arguments for `waitpid()` are
`pid`

It specifies a set of child processes for which the status is requested in the following cases:

1. If `pid` is equal to −1, status is requested for any child process. In this respect, `waitpid()` is equivalent to `wait()`.
2. If `pid` is greater than 0, it specifies the process ID of a single child process for which status is requested.
3. If `pid` is 0, status is requested for any child process whose process group ID is equal to that of the calling process. This setting is not currently supported.
4. If `pid` is less than −1, status is requested for any child process whose process group ID is equal to the absolute value of `pid`. This setting is not currently supported.

`stat_loc`

It specifies the location in which the child process's exit status is stored. If NULL is passed, no exit status is returned. Otherwise, the following macros defined in `<sys/wait.h>` can be used to evaluate the returned status:

WIFEXITED

Evaluates to a non-zero value if status was returned for a child process that exited normally.

WEXITSTATUS

If the value of WIFEXITED(s) is non-zero, this macro evaluates to the low-order eight bits of the status argument that the child process passed to `exit()` or `_exit()`, or to the value that the child process returned from `main()`.

WIFSIGNALED

Evaluates to a non-zero value if status was returned for a child process that terminated due to receipt of a signal that was not caught.

WTERMSIG

If the value of WIFSIGNALED(s) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFCORED

Evaluates to a non-zero value if status was returned for a child process that terminated due to receipt of a signal that was not caught, and whose default action is to dump core.

WCOREDUMP

Evaluates to a non-zero value if status was returned for a child process that terminated due to receipt of a signal that was not caught, and whose default action is to dump core.

WCORESIG

If the value of WIFCORED(s) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED

Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG

If the value of WIFSTOPPED(s) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

`options`

It is the bitwise inclusive-OR of zero or more of the following flags, defined in `<sys/wait.h>`:

WNOHANG

The `waitpid()` function does not suspend execution of the calling thread if status is not immediately available for one of the child processes specified by `pid`.

WUNTRACED

The status of any child processes specified by `pid` that are stopped, and whose status has not yet been reported since they stopped, is also reported to the requesting thread. This value is currently not supported, and is ignored.

Return values: The return values for `waitpid()` are as follows:

If `waitpid()` was invoked with WNOHANG set in `options,` and there are children specified by `pid` for which status is not available, `waitpid()` returns 0. If WNOHANG was not set, `waitpid()`

returns the process ID of a child when the status of that child is available. Otherwise, it returns −1 and sets `errno` to one of values given under the following error conditions.

Errors: The `waitpid()` function will fail in case of the following errors:

ECHILD

The process or process group specified by `pid` does not exist or is not a child of the calling process.

EFAULT

`stat_loc` is not a writable address.

EINTR

The function was interrupted by a signal. The value of the location pointed to by `stat_loc` is undefined.

EINVAL

The `options` argument is not valid.

ENOSYS

`pid` specifies a process group (0 or less than −1), which is not currently supported.

Let us work with the following program that uses the `waitpid()` function.

Prog. 16

In this program we use the `waitpid()` function that lets the calling process obtain status information about one of its child processes.

```
# include<stdio.h>
# include<sys/types.h>
# include<sys/wait.h>
int main()
{  pid_t pid;
   int status,n;
 if ((pid=fork())<0)
   {  perror("fork error");
      exit(1);
   }
   else if (pid ==0 )
   {  printf("child pid = %d\n",getpid());
      sleep(5);
      exit(2);
   }
   else
```

```
    if ((n=waitpid(pid,NULL,0))<0)
    {  perror("waitpid error");
       exit(3);
    }
    printf("n= %d\n",n);
    exit(0);
}
```

5.10 wait3() and wait4()

The `wait3()` function delays its caller until a signal is received or one of its child processes terminates or stops due to tracing. If any child process has died or stopped due to tracing, and this has not already been reported, return is immediate; therefore, the process ID and status of such a child process is returned. If that child process has died, it is discarded. If there are no children, −1 is returned immediately. If there are only running or stopped but reported children, the calling process is blocked.

The `wait4()` function is an extended interface. With a `pid` argument of 0, it is equivalent to `wait3()`. If `pid` has a nonzero value, then `wait4()` returns the status only for the indicated process ID, but not for any other child processes.

The function action, header files required and syntax of the `wait3()` and `wait4()` functions are as follows:

Function action

`wait3` and `wait4` – delays its caller until a signal is received.

Header file required

```
#include <sys/wait.h>
#include <sys/time.h>
#include <sys/resource.h>
```

Syntax

```
pid_t wait3(int *stat_loc, int options, struct rusage  *rusage);
pid_t wait4(pid_t pid, int *stat_loc, int options, struct rusage
*rusage);
```

The arguments for the `wait3()` and `wait4()` functions are:

`pid`

It specifies a set of child processes for which the status is requested:

1. If `pid` is equal to −1, status is requested for any child process. In this respect, `wait4()` is equivalent to `wait()`.
2. If `pid` is greater than 0, it specifies the process ID of a single child process for which status is requested.

3. If `pid` is 0, status is requested for any child process whose process group ID is equal to that of the calling process. This setting is not currently supported.
4. If `pid` is less than −1, status is requested for any child process whose process group ID is equal to the absolute value of `pid`. This setting is not currently supported.

`options`

It is the bitwise inclusive-OR of zero or more of the following flags, defined in `<sys/wait.h>`:

WNOHANG

Execution of the calling process is not suspended if status is not immediately available for any child process.

WUNTRACED

The status of any child processes that are stopped, and whose status has not yet been reported since they stopped, is also reported to the requesting process.
When the WNOHANG option is specified and no processes have status to report, `wait3()` returns 0. The WNOHANG and WUNTRACED options may be combined by the bitwise-OR operation of the two values.

`stat_loc`

It specifies the location where the child process's exit status is stored. If NULL is passed, no exit status is returned. Otherwise, the following macros defined in `<sys/wait.h>` can be used to evaluate the returned status:

WIFEXITED

Evaluates to a non-zero value if status was returned for a child process that exited normally.

WEXITSTATUS

If the value of WIFEXITED(s) is non-zero, this macro evaluates to the low-order eight bits of the status argument that the child process passed to `exit()` or `_exit()`, or the value the child process returned from `main()`.

WIFSIGNALED

Evaluates to a non-zero value if status was returned for a child process that terminated due to receipt of a signal that was not caught.

WTERMSIG

If the value of WIFSIGNALED(s) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFCORED

Evaluates to a non-zero value if status was returned for a child process that terminated due to receipt of a signal that was not caught, and whose default action is to dump core.

WCOREDUMP

Evaluates to a non-zero value if status was returned for a child process that terminated due to receipt of a signal that was not caught, and whose default action is to dump core.

WCORESIG

If the value of WIFCORED(s) is non-zero, this macro evaluates to the number of the signal that caused the termination of the child process.

WIFSTOPPED

Evaluates to a non-zero value if status was returned for a child process that is currently stopped.

WSTOPSIG

If the value of WIFSTOPPED(s) is non-zero, this macro evaluates to the number of the signal that caused the child process to stop.

`rusage`

It specifies the location where a summary of the resources used by the terminated process and all of its children is returned. Only the user time used and the system time used is currently available. They are returned in the `ru_utime` and `ru_stime` members of the `rusage` structure respectively. If NULL is passed, then no resource usage is returned.

Return values: The return values for `wait3()` or `wait4()` are as follows:

If `wait3()` or `wait4()` returns due to a stopped or terminated child process, the process ID of the child is returned to the calling process. Otherwise, −1 is returned and `errno` is set to indicate the error.

If `wait3()` or `wait4()` returns due to the delivery of a signal to the calling process, −1 is returned and `errno` is set to EINTR. If WNOHANG was set in `options`, it has at least one child process specified by `pid` for which status is not available, and if status is not available for any process specified by `pid`, 0 is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

The `wait3()` and `wait4()` functions return 0 if WNOHANG is specified, and there are no stopped or exited children. These functions return the process ID of the child process if they return due to a stopped or terminated child process. Otherwise, they return −1 and set `errno` to indicate the error.

Errors: The `wait3()` and `wait4()` functions will fail and return immediately in case of the following errors:

ECHILD

The calling process has no existing unwaited-for child processes.

EFAULT

The `stat_loc` or `rusage` arguments point to an illegal address.

EINTR

The function was interrupted by a signal. The value of the location pointed to by `stat_loc` is undefined.

EINVAL

The value of `options` is not valid.

The `wait4()` function may fail if:

ECHILD

The process specified by `pid` does not exist or is not a child of the calling process.

The `wait3()` and `wait4()` functions will terminate prematurely, return −1, and set `errno` to EINTR upon the arrival of a signal whose SA_RESTART bit in its flags field does not exist.

5.11 `system()` Function

The `system()` function is used to issue a UNIX shell command that we cannot use in UNIX programming directly. It is important to note that in `system()` function execution of your program will not continue until the command has completed. It is convenient to execute a command string from within a program. The `system()` function is passed as a pointer to a string.

The function action, header file required and syntax of the `system()` function are as follows:

Function action

`system` – issues a UNIX command.

Header file required

```
#include <stdlib.h>
```

Syntax

```
int system(const char *cmd);
```

Since the `system()` function is implemented by calling `fork()`, `exec()` and `waitpid()` functions, there are three different types of return values with it.

1. If either fork fails or `waitpid()` function returns error other than EINTR, the `system()` function returns −1 with `errno` set to indicate the error.
2. If the `exec()` function fails, the return value is as if the shell had executed the exit.
3. If the first two conditions fail then `fork()`, `exec()` and `waitpid()` functions succeed and the return value from the system is the terminator status of the shell, in the format specified for `waitpid()`.

Let us work with the following a program that executes a `system` command and waits for it to end.

Prog. 17

In this program we execute the `ls-l` command using the `system()` function.

```
# include <stdio.h>
# include <stdlib.h>
int main()
  { int x;
    x=system("ls -l");
    printf("x= %d\n",x);
    exit(0);
  }
```

5.12 Summary

The operating system control is based on process control, therefore this chapter gives an insight into the OS control process. If we are looking for the next level of control mechanism, we have to be well-versed with `fork(), exec(), wait(), waitpid(), wait3()` and `wait4()` functions. It throws light on the relationship between child and parent processes. The condition of zombie and orphan processes is also studied. It explains the concept of the `vfork()` function which ensures that the child process should run first. The entire concept of the process control will be explained in more detail in Chapter 6 on signals.

6 Signals

Signals are mechanisms for communicating with and manipulating processes in UNIX. Simply put, a signal is a special message sent to a process. When a process receives a signal, it processes the signal immediately without finishing the current function or even the current line of code. The topic of signals has a large sweep, but here we discuss some important signals and techniques to handle the uncoordinated events that are used for controlling processes. Signals are asynchronous. In other words, signals represent those events that are not synchronized, or coordinated, in time. There are several dozen different signals, each with a different meaning. Each signal type is specified by its signal number; but in programs, we usually refer to a signal by its name.

When a process receives a signal, it may do one of several things, depending on the signal's disposition. For each signal, there is a default disposition that determines what happens to the process if the program does not specify some other behaviour. For most signal types, a program may specify some other behaviour – either to ignore the signal or to call a special signal handler function to respond to the signal. If a signal handler is used, the current executing program is paused, the signal handler is executed, and the program resumes only after the signal handler returns.

The following points outline various signal characteristics:

1. Signals are software interrupts.
2. Signals provide a way of handling asynchronous events. They occur at what appear to be random times to process.
3. Every signal has a name which begins with three characters – SIG. For example, SIGABRT is the abort signal that is generated when a process calls the abort function. Similarly, SIGALRM represents the alarm signal that is generated when the timer, set by the alarm function, goes off.
4. A signal is generated for a process when the event that causes the signal occurs. The event could be a hardware exception (divide by 0), a software condition (e.g. alarm), a terminal generated signal, or a call to the kill function. In other words, a signal is delivered to a process when the action for that signal is taken.
5. When the signal is generated, the kernel usually sets a flag in the process table. During the time between generation of a signal and its delivery, the signal is said to be pending.
6. A process has the option of blocking the signal delivery. If a signal that is blocked for a process is generated, and if the action for that signal is either the default action or catching the signal, then the signal remains pending for the process until the process either (a) unblocks the signal, or (b) changes the action to ignore the signal.
7. The system determines what to do with a blocked signal when it is delivered, not when it is generated.

Here are some important points regarding signals:

1. Signals names are defined by positive integer constants in the header `<signal.h>`.
2. No signal has number 0.

3. Suppose we run a process that takes an inordinately long time, leading us to believe that something is wrong. To terminate this process, we will press DEL key.

The UNIX system also sends signals to processes in response to specific conditions. For instance, signals like SIGBUS (bus error), SIGSEGV (segmentation violation) and SIGFPE (floating point exception) may be sent to a process that attempts to perform an illegal operation. The default disposition of these signals is it to terminate the process and produce a core file.

A process may also send a signal to another process. One common use of this mechanism is to end another process by sending it a SIGTERM or SIGKILL signal. Another common use is to send a command to a running program. Two user-defined signals reserved for this purpose are: SIGUSR1 and SIGUSR2. The SIGHUP signal is also sometimes used for this purpose – commonly to wake up an idling program or cause a program to re-read its configuration files.

Described below are the situations in which some signals are generated and their default dispositions.

1. **SIGABRT**
    - Generated by calling abort system call.
    - Default to abort.
2. **SIGALRM**
    - Time or interval expires.
    - Default to exit.
3. **SIGBUS**
    - Implementation-defined H/W fault.
    - Default to abort.
4. **SIGCHLD**
    - When child terminates or stops.
    - Default ignored.
5. **SIGCONT**
    - Job-control signal.
    - Continues the stopped process.
    - Default continue / ignored.
6. **SIGFPE**
    - Arithmetic exception such as divide by zero, overflow, etc.
    - Default to abort.
    - Do not ignore it, behaviour unspecified.
7. **SIGILL**
    - Process executed illegal instruction.
    - Default to abort.

8. **SIGINT**
   - Generated by the terminal driver.
   - Ctrl+ \ or DEL key.
   - Only foreground process affected.
   - Default to exit.

9. **SIGKILL**
   - A sure way to kill a process.
   - Cannot be caught or ignored.

10. **SIGUSR1 and SIGUSR2.**
    - User-defined signal.
    - Used by applications.
    - Default to exit.

11. **SIGQUIT**
    - Generated by the terminal driver.
    - Ctrl+C or DEL key.
    - Generates a core file.
    - Default to abort.

12. **SIGSEGV**
    - Segmentation violation.
    - Terminates the process.
    - Default to abort.
    - Do not ignore this signal, behaviour unspecified.

13. **SIGSTOP**
    - Job-control signal.
    - Stops the process.
    - Cannot be caught or ignored.
    - Default to stop.

14. **SIGSYS**
    - Signals invalid system call.
    - Default to exit.

15. **SIGURG**
    - Notifies a process that an urgent condition has occurred.
    - Default ignored.

6.1 `signal()` Function

The `signal()` function chooses one of the three ways in which the receipt of the signal number `sig` is to be subsequently handled. If the value of `func` for a given signal is SIG_DFL, the default handling for that signal occurs. If the value of `func` for a given signal is SIG_IGN, the signal is ignored. Otherwise, the application ensures that `func` points to a function that is to be called when that signal occurs. An invocation of such a function because of a signal, or (recursively) of any further functions called by that invocation (other than the functions in the standard library), is called a signal handler.

The function action, header file required and syntax of the `signal()` function are as follows:

Function action

`signal` – receipt of the signal number signal is to be subsequently handled.

Header file required

```
#include <signal.h>
```

Syntax

```
void (*signal(int sig, void (*func)(int)))(int);
```

When a signal occurs, and `func` points to a function, it is implementation-defined whether the equivalent of a previous definition:

```
signal(sig, SIG_DFL);
```

is executed or the implementation prevents some implementation-defined set of signals (at least including `sig`) from occurring until the current signal handling has completed. (If the value of `sig` is SIGILL, the implementation may alternatively define that no action is taken.)

Next, the equivalent of:

```
(*func)(sig);
```

is executed if and when the function returns. If the value of `sig` was SIGFPE, SIGILL or SIGSEGV or any other implementation-defined value corresponding to a computational exception, the behaviour is undefined. Otherwise, the program shall resume execution at the point it was interrupted.

The `func` function may terminate by executing a `return` statement or by calling `abort()`, `exit()` or `longjmp()`. If `func` executes a `return` statement and the value of `sig` was SIGFPE or any other implementation-dependent value corresponding to a computational exception, the behaviour is undefined. Otherwise, the program will resume execution at the point it was interrupted.

If the signal occurs not as a result of calling `abort()`, `kill()` or `raise()`, the behaviour is undefined. The signal handler calls any function in the standard library other than any function listed on the `sigaction()` page or refers to any object with static storage duration other than by assigning a value to a static storage duration variable of type volatile sig_atomic_t. Furthermore, if such a call fails, the value of `errno` is indeterminate.

At program start-up, the equivalent of:

```
signal(sig, SIG_IGN);
```

is executed for some signals, and can be written as

```
signal(sig, SIG_DFL);
```

Return values: If the request can be honoured, `signal()` returns the value of `func` for the most recent call to `signal()` for the specified signal `sig`. Otherwise, `SIG_ERR` is returned and a positive value is stored in `errno`.

Upon successful completion, `sigset()` returns SIG_HOLD if the signal has been blocked, or the signal's previous disposition if it has not been blocked. Otherwise, SIG_ERR is returned and `errno` is set to indicate the error.

The `sigpause()` function suspends execution of the thread until a signal is received. After the receipt of the signal, `sigpause()` returns −1 and sets `errno` to EINTR.

For all other functions, upon successful completion, 0 is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors: The `signal()` function will fail in case of the following errors:

EINVAL

The `sig` argument is not a valid signal number or an attempt is made to catch a signal that cannot be caught or ignore a signal that cannot be ignored.

EINVAL

An attempt was made to set the action to SIG_DFL for a signal that cannot be caught or ignored (or both).

The `sigset()`, `sighold()`, `sigrelse()`, `sigignore()` and `sigpause()` functions will fail if the following error occurs:

EINVAL

The sig argument is an illegal signal number.

The `sigset()` and `sigignore()` functions will fail if the following error occurs:

EINVAL

An attempt is made to catch a signal that cannot be caught or to ignore a signal that cannot be ignored.

Let us work with the following program that uses the `signal()` function.

Prog. 1

In this program, we use the `signal()` function to simulate an asynchronous event by executing the `for()` loop without any arguments, which forms the infinite loop.

```
# include <stdio.h>
# include <unistd.h>
# include <signal.h>
```

```
void usrfunction(int);
int main()
{
    if (signal(SIGUSR1,usrfunction)==SIG_ERR)
    {   perror("signal error");
        exit(1);
    }
    if (signal(SIGUSR2,usrfunction)==SIG_ERR)
    {   perror("signal error");
        exit(1);
    }
    for (;;)
        pause();
    exit(0);
}
void usrfunction(int sno)
{
    if (sno==SIGUSR1)
        printf("SIGUSR1 received \n");
    if (sno==SIGUSR2)
        printf("SIGUSR2 received \n");
}
```

6.2 `kill()` Function

The `kill()` function sends a signal to a process or a group of processes specified by PID. The signal to be sent is specified by `sig` and it is either one from the list given in `<signal.h>` or 0. If `sig` is 0 (the null signal), the error checking is performed, but no signal is actually sent. The null signal can be used to check the validity of PID.

The function action, header file required used and syntax for the `kill()` function are as follows:

Function action

`kill` – sends a signal to processes specified by PID.

Header file required

```
#include<signal.h>
```

Syntax

```
int kill(pid_t pid, int sig);
```

For a process to have permission to send a signal to a process designated by PID, unless the sending process has appropriate privileges, it is necessary that the real or effective user ID of the sending process matches the real or saved set-user-ID of the receiving process.

If PID is greater than 0, `sig` is sent to the process whose process ID is equal to PID.

If PID is 0, sig is sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the process group ID of the sender, and for which the process has permission to send a signal.

If PID is −1, sig is sent to all processes (excluding an unspecified set of system processes) for which the process has permission to send that signal.

If PID is negative, but not −1, sig is sent to all processes (excluding an unspecified set of system processes) whose process group ID is equal to the absolute value of PID, and for which the process has permission to send a signal.

If the value of PID causes sig to be generated for the sending process, and if sig is not blocked for the calling thread, and if no other thread has sig unblocked or is waiting in a sigwait() function for sig, either sig or at least one pending unblocked signal is delivered to the sending thread before kill() returns.

The user ID tests described above are not to be applied when sending SIGCONT to a process that is a member of the same session as the sending process.

An implementation that provides extended security controls may impose further implementation-defined restrictions on the sending of signals including the null signal. In particular, the system may deny the existence of some or all of the processes specified by PID.

The kill() function is successful if the process has permission to send sig to any of the processes specified by PID. If kill() fails, no signal is sent.

Return values: Upon successful completion of the kill() function, 0 is returned. Otherwise, −1 is returned and errno is set to indicate the error.

Errors: The kill() function fails if the following errors occur:

EINVAL

The value of the sig argument is an invalid or unsupported signal number.

EPERM

The process does not have permission to send the signal to any receiving process.

ESRCH

No process or process group can be found corresponding to that specified by PID.

6.3 raise() Function

The raise() function sends the signal sig to the executing process.

The function action, header file required used and syntax of the raise() function are as follows:

Function action

raise – sends the signal sig to the executing process.

Header file required

```
# include <signal.h>
```

Syntax

```
int raise(int sig);
```

Return values: Upon successful completion, 0 is returned. Otherwise, a non-zero value is returned and `errno` is set to indicate the error.

Errors: The `raise()` function will fail in case of the following error:

EINVAL

The value of the `sig` argument is an invalid signal number.

6.4 `alarm()` Function

The `alarm()` function causes the system to generate a SIGALRM signal for the process after the number of real-time seconds specified by seconds have elapsed.

The function action, header file required used and syntax of the `alarm()` function are as follows:

Function action

`alarm` – generates a SIGALRM signal for the process.

Header file required

```
#include <unistd.h>
```

Syntax

```
unsigned alarm(unsigned seconds);
```

Processor scheduling delays may prevent the process from handling the signal as soon as it is generated, therefore `alarm()` is useful such cases.

If the number of specified seconds is 0, a pending alarm request, if any, is cancelled. Alarm requests are not stacked; only one SIGALRM generation can be scheduled in this manner. If the SIGALRM signal has not yet been generated, the call results in rescheduling the time at which the SIGALRM signal is generated.

What happens if an alarm call is called twice in a program? The answer is provided in the following points:

1. If the first call to alarm has not expired and if the second alarm call is called with `seconds` = 0 the alarm is cancelled.
2. If `seconds` > 0 the new value is set.
3. The return value in both the above conditions is the number of seconds remaining from the previous call.

Let us work a program that uses the `alarm()` function.

Prog. 2

The following program demonstrates the use of two alarms in the same program and shows how the second alarm affects the remaining time of the first alarm.

```
# include <std io.h>
# include <unistd.h>
# include <signal.h>
void usr_fun(int);
int main()
{    int n,m;
     if (signal(SIGALRM,usr_fun)==SIG_ERR)
     {    perror("signal int error");
        exit(1);
     }
     n=alarm(13);
     printf("alarm started, n= %d \n",n);
     sleep(8);
     m=alarm(10);
     printf("second alarm started, m= %d \n",m);
     pause();
     exit(0);
}
void usr_fun(int sno)
{
    if (sno==SIGALRM)
           printf("SIGALRM received \n");
}
```

Return values: If there is a previous `alarm()` request with time remaining, `alarm()` returns a non-zero value, that is, the number of seconds until the previous request would have generated a SIGALRM signal. Otherwise, `alarm()` returns 0.

Errors: The `alarm()` function is always successful, and no return value is reserved to indicate an error.

6.5 pause() Function

The `pause()` function suspends the calling process until a signal is caught.

The function action, header file required used and syntax of the `pause()` function are as follows:

Function action

`pause` – suspends the calling process.

Header file required

```
#include <signal.h>
```

Syntax

```
int pause(void);
```

Return values: Since `pause()` suspends thread execution indefinitely unless interrupted by a signal, there is no return value for successful completion. If interrupted, it returns −1 and sets `errno` to indicate the error.

Errors: The `pause()` function will fail if:

EINTR
A signal is caught by the calling process and control is returned from the signal-catching function.

6.6 `sleep()` Function

The `sleep()` function causes the calling thread to be suspended from execution until either the number of real-time seconds specified by the argument `seconds` has elapsed or a signal is delivered to the calling thread, and its action is to invoke a signal-catching function or terminate the process. The suspension time may be longer than requested due to the scheduling of other activity by the system.

The function action, header file required used and syntax of the `sleep()` function are as follows:

Function action

`sleep` – suspends execution for an interval of time.

Header file required

```
#include <unistd.h>
```

Syntax

```
unsigned sleep(unsigned seconds);
```

If a SIGALRM signal is generated for the calling process during execution of `sleep()` and if the SIGALRM signal is being ignored or blocked from delivery, it is unspecified whether `sleep()` returns when the SIGALRM signal is scheduled. If the signal is being blocked, it is also unspecified whether it remains pending after `sleep()` returns or it is discarded.

If a SIGALRM signal is generated for the calling process during execution of `sleep()`, except as a result of a prior call to `alarm()`, and if the SIGALRM signal is not being ignored or blocked from delivery, it is unspecified whether that signal has any effect other than causing `sleep()` to return.

If a signal-catching function interrupts `sleep()` and examines or changes either the time a SIGALRM is scheduled to be generated or the action associated with the SIGALRM signal, or whether the SIGALRM signal is blocked from delivery, the results are unspecified.

Return values: If `sleep()` returns because the requested time has elapsed, the value returned shall be 0. If `sleep()` returns due to delivery of a signal, the return value shall be the 'unslept' amount (the requested time minus the time actually slept) in seconds.

Errors: No errors are defined for the `sleep()` function.

Asynchronous child death notification: The problem with calling `wait()` directly is that usually you want the parent process to do other things while its child process executes its code. Otherwise, you are not really enjoying multi-processes. Are you? That problem has a solution by using signals. When a child process dies, a signal, SIGCHLD (or SIGCLD), is sent to its parent process. Thus, using a proper signal handler, the parent will get an asynchronous notification, and then when it will call `wait()`, the system assures that the call will return immediately, since there is already a zombie child.

Let us now work with a program that uses a signal handler.

Prog. 3

The following program is an example of our *Join the Unique UNIX Caravan* program. Here we use a signal handler with this program.

```
# include <stdio.h>         /* basic I/O routines. */
# include <unistd.h>        /* define fork(), etc. */
# include <sys/types.h>     /* define pid_t, etc. */
# include <sys/wait.h>      /* define wait(), etc. */
# include <signal.h>        /* define signal(), etc. */

/* first, here is the code for the signal handler */
void catch_child(int sig_num)
{
/* when we get here, we know there’s a zombie child waiting */
int child_status;

wait(&child_status);
printf(“child exited.\n”);
}

.
.
/* and somewhere in the main() function ... */
.
.

/* define the signal handler for the CHLD signal */
signal(SIGCHLD, catch_child);

/* and the child process forking code... */
{
int child_pid;
int i;

child_pid = fork();
switch (child_pid) {
 case −1:    /* fork() failed */
  perror(“fork”);
  exit(1);
case 0:    /* inside child process */
```

```
 printf("hello world\n");
 sleep(5);  /* sleep a little, so we'll have */
    /* time to see what is going on */
 exit(0);
default:    /* inside parent process */
 break;
}
/* parent process goes on, minding its own business... */
/* for example, some output...        */
For (i=0; i<10; i++) {
 printf("%d\n", i);
 sleep(1);  /* sleep for a second, so we'll have time to see
the mix */
 }
}
```

Let us examine the flow of the above program:

1. A signal handler is defined, so whenever we receive a SIGCHLD, `catch_child` will be called.
2. We call `fork()` to spawn a child process.
3. The parent process continues its control flow, while the child process is doing its own chores.
4. When the child calls `exit()`, a CHLD signal is sent by the system to the parent. The parent process's execution is interrupted, and its CHLD signal handler `catch_child` is invoked.
5. The `wait()` call in the parent causes the child to be completely removed from the system.
6. Finally, the signal handler returns, and the parent process continues execution at the same place where it was interrupted.

6.7 `abort()` Function

The `abort()` function causes abnormal process termination to occur unless the signal SIGABRT is being caught and the signal handler does not return. The abnormal termination processing includes the default actions defined for SIGABRT and may include an attempt to effect `fclose()` on all open streams. The SIGABRT signal is sent to the calling process as if by means of `raise()` with the argument SIGABRT.

The function action, header file required used and syntax of the `abort()` function are as follows:

Function action

`abort` – causes abnormal process termination.

Header file required

```
# include <stdlib.h>
```

Syntax

```
void abort(void);
```

Return values: The status made available to `wait()` or `waitpid()` by `abort()` is that of a process terminated by the SIGABRT signal. The `abort()` function overrides blocking or ignoring the SIGABRT signal. The `abort()` function does not return.

Errors: No errors are defined for `abort()`.

6.8 Signal Sets

The `sigset()`, `sighold()`, `sigignore()`, `sigpause()` and `sigrelse()` functions provide simplified signal management.

The `sigset()` function is used to modify signal dispositions. The `sig` argument specifies the signal that may be any signal except SIGKILL and SIGSTOP. The `disp` argument specifies the signal's disposition that may be SIG_DFL, SIG_IGN or the address of a signal handler. If `sigset()` is used, and `disp` is the address of a signal handler, the system will add `sig` to the calling process's signal mask before executing the signal handler. And, when the signal handler returns, the system restores the calling process's signal mask to its pre-signal delivery state. In addition, if `sigset()` is used, and `disp` is equal to SIG_HOLD, `sig` will be added to the calling process's signal mask, and `sig`'s disposition will remain unchanged. If `sigset()` is used, and `disp` is not equal to SIG_HOLD, `sig` will be removed from the calling process's signal mask.

Now we will discuss the functionalities of signal sets in brief:

1. The `sighold()` function adds `sig` to the calling process's signal mask.
2. The `sigrelse()` function removes `sig` from the calling process's signal mask.
3. The `sigignore()` function sets the disposition of `sig` to SIG_IGN.
4. The `sigpause()` function removes `sig` from the calling process's signal mask and suspends the calling process until a signal is received. The `sigpause()` function restores the process's signal mask to its original state before returning.

If the action for the SIGCHLD signal is set to SIG_IGN, child processes of the calling processes are not transformed into zombie processes when they terminate. If the calling process subsequently waits for its children, and the process has no unwaited-for children that were transformed into zombie processes, it blocks until all of its children terminate, and `wait()`, `wait3()`, `waitid()` and `waitpid()` fail with `errno` set to ECHILD.

Return values: Upon successful completion, `sigset()` returns SIG_HOLD if the signal was blocked; if it was not blocked, the signal's previous disposition is returned. Otherwise, SIG_ERR is returned and `errno` is set to indicate the error. For all other functions, upon successful completion, 0 is returned. Otherwise, −1 is returned and `errno` is set to indicate the error.

Errors: The `sigset()`, `sighold()`, `sigrelse()`, `sigignore()` and `sigpause()` functions will fail if the following error occurs:

EINVAL

The `sig` argument is an illegal signal number.

The `sigset()` and `sigignore()` functions will fail in case of the following errors:

EINVAL

An attempt is made to catch a signal that cannot be caught, or to ignore a signal that cannot be ignored.

`sigset()`, `sigaddset()`, `sigdelset()`, `sigemptyset()`, `sigfillset()` and `sigismember()` functions: These system calls that handle signals, like `sigaction()` and `sigprocmask()`, use sets of signals to keep a process from being interrupted by those signals while executing a signal handler or a critical code segment.

The function action, header file required used and syntax of the `signal` set functions are as follows:

Function action

`signal set` – keeps a process from being interrupted.

Header file required

```
# include <signal.h>
```

Syntax

```
int sigaddset(sigset_t *set, int sig)
int sigdelset(sigset_t *set, int sig)
int sigemptyset(sigset_t *set)
int sigfillset(sigset_t *set)
int sigismember(const sigset_t *set, int sig)
```

Now let us understand the actions taken by some `signal` set functions given below.

`int sigaddset(sigset_t *set, int sig)`
Adds signal sig to the signal set referenced by `set`.

`int sigdelset(sigset_t *set, int sig)`
Removes signal sig from the signal set referenced by `set`.

`int sigemptyset(sigset_t *set)`
Initializes the signal set referenced by `set` to an empty set.

`int sigfillset(sigset_t *set)`
Initializes the signal set referenced by `set` to a full set, that is, all signals are in the set.

`int sigismember(const sigset_t *set, int sig)`
Returns 1 if signal `sig` is present in the set referenced by `set`, otherwise 0.

6.9 `sigaction()` Function

The `sigaction()` function is used to examine, set or modify the attributes of a signal. The argument `sig` is the signal in question. The `act` argument points to a structure containing the new attributes of the signal, and the structure pointed to by `oact` will receive the old attributes that were in effect before the call.

The function action, header file required used and syntax of the `sigaction()` function are as follows:

Function action

`sigaction` – examines, sets or modifies the attributes of a signal.

Header file required

```
#include <signal.h>
```

Syntax

```
int sigaction(int sig, const struct sigaction *act, struct sigac-
tion *oact)
```

The `act` and `oact` arguments may be NULL, indicating that either no new attributes are to be set or the old attributes are not of interest.

The structure containing the signal attributes is defined in `<signal.h>` and looks like this:

```
struct sigaction {
void (*sa_handler)(int sig);
sigset_t sa_mask;
int sa_flags;
};
```

Let us understand different parts of the above code.

The `sa_handler` field contains the address of a signal handler, a function that is called when the process is signalled, or one of the following special constants:

SIG_DFL

Default signal handling is to be performed. This usually means that the process is killed, but some signals may be ignored by default.

SIG_IGN

Ignore the signal.

The `sa_mask` field indicates a set of signals that must be blocked when the signal is being handled. Whether the signal `sig` itself is blocked when being handled is not controlled by this mask. The mask is of a 'signal set' type that is to be manipulated by the `sigset()` functions.

How the signal is handled is specified precisely by bits in `sa_flags`. If none of the flags is set then the handler is called when the signal arrives. The signal is blocked during the call to the handler, and unblocked when the handler returns.

A system call that is interrupted returns −1 with `errno` set to EINTR. The following bit flags can be set to modify this behaviour:

SA_RESETHAND	Reset the signal handler to SIG_DFL when the signal is caught.
SA_NODEFER	Do not block the signal on entry to the handler.
SA_COMPAT	Handle the signal in a way that is compatible with the old `signal()` call.

Signal handlers are reset to SIG_DFL on an `execve()`. Signals that are ignored stay ignored.

Signal numbers and description

The following table lists the names, numbers and function descriptions of various signals.

Signal	*Num*	*Notes*	*Description*
SIGHUP	1	k	Hangup
SIGINT	2	k	Interrupt (usually DEL or CTRL +C)
SIGQUIT	3	kc	Quit (usually CTRL+\)
SIGILL	4	kc	Illegal instruction
SIGTRAP	5	xkc	Trace trap
SIGABRT	6	kc	Abort program
SIGFPE	8	k	Floating point exception
SIGKILL	9	k	Kill
SIGUSR1	10	k	User-defined signal #1
SIGSEGV	11	kc	Segmentation fault
SIGUSR2	12	k	User-defined signal #2
SIGPIPE	13	k	Write to a pipe with no reader
SIGALRM	14	k	Alarm clock
SIGTERM	15	k	Terminate (default for kill)
SIGCHLD	17	pvi	Child process terminated
SIGCONT	18	p	Continue if stopped
SIGSTOP	19	ps	Stop signal
SIGTSTP	20	ps	Interactive stop signal
SIGTTIN	21	ps	Background read
SIGTTOU	22	ps	Background write
SIGWINCH	23	xvi	Window size change

What the letters in the notes column indicate has been described below:

k	The process is killed if the signal is not caught.
c	The signal causes a core dump.
i	The signal is ignored if not caught.
v	Only Minix-vmd implements this signal.
x	Minix extension, not defined by POSIX.
p	These signals are not implemented, but POSIX requires that they are defined.
s	The process should be stopped, but is killed instead.

The SIGKILL signal cannot be caught or ignored. The SIGILL and SIGTRAP signals cannot be automatically reset. The system silently enforces these restrictions. This may or may not be reflected by the attributes of these signals and the signal masks.

6.10 `sigsuspend()` Function

The `sigsuspend()` function installs the signal mask referenced by `set` and suspends the process until signalled. The signal is handled, the signal mask is restored to the value it had before the `sigsuspend()` call and the call returns.

The function action, header file required used and syntax of the `sigsuspend()` function are as follows:

Function action

`sigsuspend` – installs the signal mask referenced by `set`

Header file required

```
# include <signal.h>
```

Syntax

```
int sigsuspend(const sigset_t *set)
```

6.11 Signal Masks

UNIX processes contain a signal mask that defines which signals can be delivered and which have to be blocked from delivery at any given time. When a signal arrives, the UNIX kernel checks the signal mask. If the signal is in the process mask, it is delivered; otherwise, it is noted as undeliverable and nothing further is done until the signal mask changes. Sets of signals are represented within Interactive Data Language (IDL) with the opaque type `IDL_SignalSet_t`.

UNIX IDL provides several functions that manipulate signal sets to change the process mask and allow/disallow delivery of signals. Let us discuss the functionalities of some functions.

1. **`IDL_SignalSetInit()`**

   `IDL_SignalSetInit()` initializes a signal set to be empty, and optionally sets it to contain one signal.

   Following is the syntax of the `IDL_SignalSetInit()` function:

   ```
   void IDL_SignalSetInit(IDL_SignalSet_t *set, int signo)
   ```

   where:

   `set` – The signal set to be emptied/initialized.

   `signo` – If non-zero, a signal is to be added to the new set. This is provided as a convenience for the large number of cases where a set contains only one signal. Use `IDL_SignalSetAdd()` to add additional signals to a set.

2. **`IDL_SignalSetAdd()`**

   `IDL_SignalSetAdd()` adds the specified signal to the specified signal set:

   Following is the syntax of the `IDL_SignalSetAdd()` function:

   ```
   void IDL_SignalSetAdd(IDL_SignalSet_t *set, int signo)
   ```

where:

`set` – The signal set to be added to. The signal set must have been initialized by `IDL_SignalSetInit()`.
`signo` – The signal to be added to the signal set.

3. `IDL_SignalSetDel()`

`IDL_SignalSetDel()` deletes the specified signal from a signal set.

Following is the syntax of the `IDL_SignalSetdel()` function:

```
void IDL_SignalSetDel(IDL_SignalSet_t *set, int signo)
```

where:

`set` – The signal set to delete from. The signal set must have been initialized by

`IDL_SignalSetInit()`.

`signo` – The signal to be removed from the signal set.

4. `IDL_SignalSetIsMember()`

`IDL_SignalSetIsMember()` tests a signal set for the presence of a specified signal, returning TRUE if the signal is present, otherwise FALSE.

Following is the syntax of the `IDL_SignalSetIsMember()` function:

```
int IDL_SignalSetIsMember(IDL_SignalSet_t *set, int signo)
```

where:

`set` – The signal set to test. The signal set must have been initialized by `IDL_SignalSetInit()`.

`signo` – The signal to be removed from the signal set.

5. `IDL_SignalMaskGet()`

`IDL_SignalMaskGet()` sets a signal set to contain the signals from the current process signal mask.

Following is the syntax of the `IDL_SignalMaskGet()` function:

```
void IDL_SignalMaskGet(IDL_SignalSet_t *set)
```

where:

`set` – The signal set in which the current process signal mask will be stored.

6. `IDL_SignalMaskSet()`

`IDL_SignalMaskSet()` sets the current process signal mask to contain the signals specified in a signal mask.

Following is the syntax of the `IDL_SignalMaskSet()` function:

```
void IDL_SignalMaskSet(IDL_SignalSet_t *set, IDL_SignalSet_t *omask)
```

where:

`set` – The signal set from which the current process signal mask will be set.
`omask` – If `omask` is non-NULL, the unmodified process signal mask is stored in it. This is useful for restoring the mask later using `IDL_SignalMaskSet()`.

There are some IDL signals that cannot be blocked. This limitation is silently enforced by the operating system.

7. **`IDL_SignalMaskBlock()`**

`IDL_SignalMaskBlock()` adds signals to the current process signal mask.

Following is the syntax of the `IDL_SignalMaskBlock()` function:

```
void IDL_SignalMaskBlock(IDL_SignalSet_t *set, IDL_SignalSet_t *oset)
```

where:

`set` – The signal set containing the signals that will be added to the current process signal mask.

`oset` – If `oset` is non-NULL, the unmodified process signal mask is stored in it. This is useful for restoring the mask later using `IDL_SignalMaskSet()`.

8. `IDL_SignalBlock()`

`IDL_SignalBlock()` does the same thing as `IDL_SignalMaskBlock()` except that it accepts a single signal number instead of requiring a mask to be built.

Following is the syntax of the `IDL_SignalBlock()` function:

```
void IDL_SignalBlock(int signo, IDL_SignalSet_t *oset)
```

where:

`signo` – The signal to be blocked.

9. **`IDL_SignalSuspend()`**

`IDL_SignalSuspend()` replaces the process signal mask with the ones in `set` and then suspends the process until a signal is delivered. On return, the original process signal mask is restored.

Following is the syntax of the `IDL_SignalSuspend()` function:

```
void IDL_SignalSuspend(IDL_SignalSet_t *set)
```

where:

`set` – The signal set containing the signals that will be added to the current process signal mask.

6.12 setjmp(), sigsetjmp(), longjmp(), siglongjmp() Functions

These functions are useful for dealing with errors and interrupts encountered in a low-level subroutine of a program.

The function action, header file required used and syntax of the signal jump functions are as follows:

Function action

`signal jump` – deals with errors and interrupts at low level.

Header file required

```
#include <setjmp.h>
```

Syntax

```
int setjmp(jmp_buf env);
int sigsetjmp(sigjmp_buf env, int savemask);
void longjmp(jmp_buf env, int val);
void siglongjmp(sigjmp_buf env, int val);
```

Described below are the tasks accomplished by various signal jump functions.

The `setjmp()` function saves its stack environment in `env` for later use by `longjmp()`.

The `sigsetjmp()` function saves the calling process's registers and stack environment in `env` for later use by `siglongjmp()`. If `savemask` is non-zero, the calling process's signal masks and schedules parameters.

The `longjmp()` function restores the environment saved by the last call of `setjmp()` with the corresponding `env` argument. After `longjmp()` completes, the program execution continues as if the corresponding call to `setjmp()` has just returned the value `val`. The caller of `setjmp()` must not have returned in the interim. The `longjmp()` function cannot cause `setjmp()` to return the value 0. If `longjmp()` is invoked with the second argument of 0, `setjmp()` will return 1. At the time of the second return from `setjmp()`, all external and static variables have values as of the time when `longjmp()` was called.

The `siglongjmp()` function restores the environment saved by the last call of `sigsetjmp()` with the corresponding `env` argument. After `siglongjmp()` completes, the program execution continues as if the corresponding call to `sigsetjmp()` has just returned the value `val`. The `siglongjmp()` function cannot cause `sigsetjmp()` to return the value 0. If `siglongjmp()` is invoked with a second argument of 0, `sigsetjmp()` will return 1. At the time of the second return from `sigsetjmp()`, all external and static variables have values as of the time when `siglongjmp()` was called.

If a signal-catching function interrupts `sleep()` and calls `siglongjmp()` to restore an environment saved prior to the `sleep()` call, the action associated with SIGALRM and the time it is scheduled to be generated are unspecified. It is also unspecified whether the SIGALRM signal is blocked, unless the process's signal mask is restored as part of the environment.

The `siglongjmp()` function restores the saved signal mask if and only if the `env` argument was initialized by a call to the `sigsetjmp()` function with a non-zero `savemask` argument.

The values of register and automatic variables are undefined. Register or automatic variables whose value must be relied upon must be declared as volatile.

If the return is from a direct invocation, `setjmp()` and `sigsetjmp()` return 0. If the return is from a call to `longjmp()`, `setjmp()` returns a non-zero value. If the return is from a call to `siglongjmp()`, `sigsetjmp()` returns a non-zero value.

After `longjmp()` is completed, the program execution continues as if the corresponding invocation of `setjmp()` has just returned the value specified by `val`. The `longjmp()` function cannot cause `setjmp()` to return 0; if `val` is 0, `setjmp()` returns 1.

After `siglongjmp()` is completed, the program execution continues as if the corresponding invocation of `sigsetjmp()` has just returned the value specified by `val`. The `longjmp()` function cannot cause `setjmp()` to return 0; if `val` is 0, `setjmp()` returns 1.

Let us now work with a program based on `setjmp()` and `longjmp()` functions.

Prog. 4

The following program uses both `setjmp()` and `longjmp()` to return the flow of control to the appropriate instruction block:

```
#include <stdio.h>
#include <setjmp.h>
#include <signal.h>
#include <unistd.h>
jmp_buf env; static void signal_handler();

  main() {
     int returned_from_longjump, processing = 1;
     unsigned int time_interval = 4;
     if ((returned_from_longjump = setjmp(env)) != 0)
      switch (returned_from_longjump)  {
        case SIGINT:
        printf("longjumped from interrupt %d\n",SIGINT);
        break;
        case SIGALRM:
        printf("longjumped from alarm %d\n",SIGALRM);
        break;
      }
     (void) signal(SIGINT, signal_handler);
     (void) signal(SIGALRM, signal_handler);
     alarm(time_interval);
     while (processing)   {
      printf(" waiting for you to INTERRUPT (cntrl-C) ...\n");
      sleep(1);
     }   /* end while forever loop */
  }
```

```
static void signal_handler(sig)
int sig; {
   switch (sig)  {
    case SIGINT:  ...  /* process for interrupt */
       longjmp(env,sig);
          /* break never reached */
    case SIGALRM: ...  /* process for alarm */
       longjmp(env,sig);
         /* break never reached */
    default:  exit(sig);
   }
}
```

When this program is compiled and executed, and the user sends an interrupt signal, the output will be:

```
longjumped from interrupt
```

Additionally, every four seconds the alarm will expire, send the signal to this process, and the output will be:

```
longjumped from alarm
```

6.13 Summary

The UNIX systems programming requires working with the concept of signals that deal with asynchronous events. This chapter takes an in-depth look into various facets of signals. Starting with an explanation of the concept of signals, it focuses on their practical use, giving an insight into different signal functions. The chapter tells the reader how to use these signals to handle interrupt situations. It explains the use of `abort, sleep, alarm, kill, raise,` etc. functions in different situations. It also explains the concept of signal masking, signal sets and signal jump.

7 Memory and Data Management

Unlike most other PC operating systems, modern UNIX systems (evolved since around 1989) apply sophisticated memory management algorithms to make efficient use of memory resources. When it comes to efficient memory management, one may ask: *How much memory do I have? How much memory is being used?* To answer these questions, we must understand the basics of memory, like there are three kinds of memory, three different ways in which these can be used by an operating system, and three different ways in which these can be used by processes.

7.1 Kinds of Memory

Described below are three types of memory that reside on an OS memory.

1. **Main:** It is the physical random access memory (RAM) located on the CPU motherboard. Also called real memory, RAM does not include processor caches, video memory or other peripheral memory.
2. **File system:** It is the disk memory accessible via pathnames. It includes all network file systems. However, it does not include raw devices, tape drives, swap space or other storage not addressable via normal pathnames.
3. **Swap space:** This is also disk memory but is used to hold the data that is not in real or file system memory.

Described below are three types of memory that are especially used by operating system to facilitate other requests.

1. **Kernel:** The operating system's own (semi-)private memory space. This is always in the main memory.
2. **Cache:** Main memory that is used to hold elements of the file system and other I/O operations.
3. **Virtual:** The total addressable memory space of all processes running on the given machine. The physical location of such data may be spread among any of the three kinds of memory.

Virtual memory uses:

1. **Data:** Memory allocated and used by the program (via `malloc` or directly through `brk` and `sbrk`).
2. **Stack:** The program's execution stack (managed by the OS).
3. **Mapped:** File contents addressable within the process memory space.

The amount of memory available for processes is at least the size of swap (process of moving some pages out of the main memory and moving others in), minus kernel. On more modern systems, the main memory constitutes at least main plus swap minus kernel. It may also include any files via mapping.

7.2 Swapping

The virtual memory is divided into pages, chunks that are usually either 4kb or 8kb in size. The memory manager considers pages to be the atomic unit of memory. For the best performance, we want each page to be accessible in the main memory as it is needed by the CPU. When a page is not needed, it does not matter where it is located.

The collection of pages that a process is expected to use in the near future is called its resident set. (Some operating systems consider all the pages currently located in the main memory to be the resident set, even if they aren't being used.) The process of moving some pages out of the main memory and moving others in is called swapping. (For the purpose of our discussion, disk caching activity is also included in this notion of swapping even though it is generally considered a separate activity.)

A page fault occurs when the CPU tries to access a page that is not in the main memory, thus forcing the CPU to wait for the page to be swapped in. Since moving data to and from the disk takes a significant amount of time, the goal of the memory manager is to minimize the number of page faults.

Where a page will go when it is swapped out depends on how it is being used. In general, pages are swapped out as follows:

Kernel

Never swapped out.

Cache

Page is discarded.

Data

Moved to swap space.

Stack

Moved to swap space.

Mapped

Moved to originating file.

It is important to note that swapping itself does not necessarily slow down the computer. Performance is only impeded when a page fault occurs. At that time, if memory is scarce, a page of the main memory, that is not being used, must be freed for every page that is needed. If a page that is being swapped out has changed since it was last written to the disk, it can't be freed from the main memory until the changes have been recorded (either in swap space or a mapped file).

Writing a page to disk need not wait until a page fault occurs. Most modern UNIX systems implement pre-emptive swapping in which the contents of changed pages are copied to the disk during times when the disk is otherwise idle. The page is also kept in the main memory so that it could be accessed when needed. But, if a page fault occurs, the system can instantly reclaim the pre-emptively swapped pages in the same time that is needed to read the new page. This saves tremendous amount of time since writing to the disk usually takes two to four times longer than reading. Thus pre-emptive swapping may also occur even if the main memory is plentiful as a hedge against future shortages.

As it is extremely rare for all (or even most) of the processes on a UNIX system to be in use simultaneously, most of virtual memory may be swapped out at any given time without significantly impeding the system's performance. If the activation of one process occurs at a time when another is idle, they simply trade places with minimum impact as the virtual memory concept makes work for both the processes easier. The performance is significantly affected only when more memory is needed to take care of some simultaneous processes than is available. This point will become more clear after the following discussion.

7.3 Mapped Files

The subject of file mapping deserves special attention because most people, even experienced programmers, never have direct experience of it. However, file mapping is integral to the functioning of modern operating systems. When a process maps a file, a segment of its virtual memory is designated as an address containing the contents of the given file. Retrieving data from such a memory address actually means retrieving data from the file. Thanks to file mapping, the OS handles the data retrieval transparently, and it is much faster and more efficient than the standard file access methods.

In case, if multiple processes map and access the same file, the same real memory and swap pages will be shared among all these processes. This allows multiple programs to share data without having to maintain multiple copies in memory.

The primary use of file mapping is in loading the executable code. When a program is executed, one of the first actions is to map the executable program and all of its shared libraries into the newly created virtual memory space.

As the program begins execution, it page faults, forcing the machine instructions to be loaded into memory as they are needed. Multiple invocations of the same executable program, or the programs that use the same code libraries, will share the same pages of real memory.

A program is a collection of segments. A fault arising with any of the following segments in a logical unit – such as main program, procedure, function, method, object, local variables, global variables, common block, stack, symbol table and arrays – is segmentation fault. What happens when a process attempts to change a mapped page depends on the particular OS and the mapping parameters. Basically, pages are mapped in three ways – 'read-only', 'shared' and 'private'. Executable pages are usually mapped 'read-only' so that a segmentation fault occurs if they are written to. Pages mapped as 'shared' will have changes marked in the shared real memory pages and eventually will be written back to the file. Those marked as 'private' will have private copies created in swap space as they are changed and will not be written back to the originating file.

Some operating systems always copy the contents of a mapped file into the swap space. This limits the quantity of mapped files to the size of swap space, but it means that if a mapped file is deleted or changed (by means other than mapping), the mapping processes will still have a clean copy. Other operating systems only copy privately changed pages into swap space. This allows to map files in large numbers, but deleting or changing such a file may cause a bus in error in the processes using it.

7.4 Estimation of Memory

The total real memory is calculated by subtracting the kernel memory from the amount of RAM. Some of the real memory may be used for caching, but the process needs usually take priority over cached data as far as memory usage is concerned.

The total virtual memory depends on the degree to which processes use mapped files. For data and stack space, the amount of real and swap memories sets the virtual memory limits. On some systems the limiting factor is simply the amount of swap space, while on others it is the sum of the real and swap memories. If the mapped files are automatically copied into swap space, then they must also fit into swap memory making that amount the limiting factor. But if the mapped files act as their own swap area, then there is no practical limit to the amount of virtual memory that could be mapped onto them.

In practice, it is easy and cheap to add arbitrary amounts of swap space and thus virtual memory. The real limiting factor on the system performance is the amount of real memory.

7.5 Actual Memory Used

If no programs were sharing memory or mapping files, you could just add up their resident sets to get the amount of real memory in use and their virtual memories to get the amount of swap space in use. But the shared memory (both through mapped files and inter-process communication system calls) means that the resident sets of multiple processes may be counting the same real memory pages more than once. Likewise, the mapped files (on OSs that use them for swapping) will count toward a process's virtual memory use but won't consume swap space.

The only practical measure of the memory in use is relative to the amount available and its effect on performance. If there is less memory available than the total resident sets of all running processes (after accounting for sharing), then the computer will need to swap continuously. This non-stop disk activity is called thrashing and it indicates that there are too many active processes or that more memory is needed. If there is just barely enough memory for the resident sets but not enough for all virtual memory, then swapping will occur only when new programs are run or the patterns of user interaction change. If there is more real memory than virtual memory, then there will be plenty of extra memory for disk caching.

7.6 Memory Allocation in C Programs

The C language supports two kinds of memory allocation – static and automatic – through the variables in C programs:

1. Static allocation takes place when you declare a static or global variable. Each static or global variable defines one block of space of a fixed size. This space is allocated once when your program is started (as part of the `exec` operation), and is never freed.
2. Automatic allocation happens when you declare an automatic variable such as a function argument or a local variable. The space for an automatic variable is allocated when the compound statement containing the declaration is entered. This space is freed when the compound statement is exited.

In GNU C, the size of the automatic storage varies, but in other C implementations, it must be a constant.

A third important kind of memory allocation, dynamic allocation, is not supported by C variables but is available via GNU C library functions.

7.7 Dynamic Memory Allocation

Dynamic memory allocation is a technique in which running programs determine where to store some information. You need dynamic allocation when the amount of memory required, or duration for which it is needed, depends on factors that are not known before the program runs.

For example, you may need a block to store a line read from an input file but you don't know how long this line could be. In this case, you must allocate the memory dynamically and make it dynamically larger as you read more of the line.

You may also need a block for each record or each definition in the input data, but you don't know in advance the number of records, and you must allocate a new block for each record or definition as you read it. In this case also, dynamic allocation of memory is required.

When you use dynamic allocation, the allocation of a block of memory is an action that the program requests explicitly. You call a function or macro when you want to allocate space, and specify the size with an argument. If you want to free the space, you do so by calling another function or macro. You can do these things as often as you want.

Dynamic allocation is not supported by C variables, therefore there is no storage class 'dynamic', and there can never be a C variable whose value is stored in dynamically allocated space. The only way to get dynamically allocated memory is via a system call (which is generally via a GNU C library function call), and the only way to refer to dynamically allocated space is through a pointer. This method being less convenient and because the actual process of dynamic allocation requires more computation time, programmers generally use dynamic allocation only when neither static nor automatic allocation serves the purpose.

For example, if you want to dynamically allocate some space to hold a `struct foobar`, you cannot declare a variable of type `struct foobar` whose contents are the dynamically allocated space. But you can declare a variable of pointer type `struct foobar *` and assign it the address of the dynamically allocated space. Then you can use operators * and -> on this pointer variable to refer to the contents of the space.

The following program snippet demonstrates how to allocate space dynamically to hold a `struct foobar`.

```
{
struct foobar *ptr
  = (struct foobar *) malloc (sizeof (struct foobar));
ptr->name = x;
ptr->next = current_foobar;
current_foobar = ptr;
}
```

7.8 Basic Memory Allocation

`malloc()` function

The `malloc()` function returns a pointer to a newly allocated block *size* bytes long, or a null pointer if the block could not be allocated. The contents of the block are undefined; you must initialize it yourself. Normally you would cast the value as a pointer to the kind of object that you want to store in the block.

The function action, header files required and syntax of the `malloc()` function are as follows:

Function action

`malloc` – allocates a block of memory.

Header file required

`#include<stdlib.h>`

Syntax

`void * malloc (size_t size)`

Here is an example of `malloc()` function where we initialize the space with zeros using the library function `memset()`.

```
struct foo *ptr;

...
ptr = (struct foo *) malloc (sizeof (struct foo));
if (ptr == 0) abort ();

memset (ptr, 0, sizeof (struct foo));
```

You can store the result of `malloc()` into any pointer variable without a cast, because ISO C automatically converts the type `void *` to another type of pointer when necessary. But the cast is necessary in contexts other than assignment operators or if you want your code to run in traditional C.

Remember that when allocating space for a string, the argument to `malloc()` must be one plus the length of the string. This is because a string is terminated with a null character that doesn't count in the '`length`' of the string but needs space. For example,

```
char *ptr;

...
ptr = (char *) malloc (length + 1);
```

Examples of `malloc()`

It is important to know where to use the `malloc()` function, therefore here are some peculiar cases for the use of this function.

1. If no more space is available, `malloc()` returns a null pointer. You should check the value of every call to `malloc()`. It is useful to write a subroutine that calls `malloc()` and reports an error if

the value is a null pointer, returning only if the value is nonzero. This function is conventionally called `xmalloc`. It is as follows:

```
void *
xmalloc (size_t size)
{
register void *value = malloc (size);
if (value == 0)
  fatal ("virtual memory exhausted");
return value;
}
```

2. Here is a real example of using `malloc()` (by way of `xmalloc`). The `savestring` function will copy a sequence of characters into a newly allocated null-terminated string:

```
char *
savestring (const char *ptr, size_t len)
{
register char *value = (char *) xmalloc (len + 1);
value[len] = '\0';
return (char *) memcpy (value, ptr, len);
}
```

The block that `malloc()` gives you is guaranteed to be aligned so that it can hold any type of data. In the GNU system, the address is always a multiple of eight on most systems, and a multiple of 16 on 64-bit systems. Only rarely is any higher boundary (like a page boundary) necessary; for those cases, use `memalign(), posix_memalign()` or `valloc()` functions.

7.9 Freeing Memory Allocated with `malloc()`

In this section we will discuss some of the memory freeing functions that will be implemented using the `malloc()` function.

1. `free()` function

When you no longer need a block that you got with `malloc()`, use the `free()` function to make the block available for reallocation. The prototype for this function is in `<stdlib.h>`.

The function action, header file required and syntax of the `free()` function are as follows:

Function action

`free` – makes the block available.

Header file required

```
#include<stdlib.h>
```

Syntax

```
void free (void *ptr)
```

The `free()` function de-allocates the block of memory pointed at by `ptr`.

2. `cfree()` function

This function does the same thing as `free()`. It's provided for backward compatibility with the OS; but you should use `free()` instead.

The function action, header files required and syntax for the `cfree()` function are as follows:

Function action

`cfree` – provides backward compatibility.

Header file required

```
#include <stdlib.h>
```

Syntax

```
void cfree (void *ptr)
```

Freeing a block alters the contents of the block. Do not expect to find any data (like a pointer to the next block in a chain of blocks) in the block after freeing it. Copy whatever you need out of the block before freeing it! Here is an example of the proper way of freeing all the blocks in a chain, and the strings that they point to:

```
struct chain
  {
  struct chain *next;
  char *name;
  };

void
free_chain (struct chain *chain)
{
while (chain != 0)
  {
  struct chain *next = chain->next;
  free (chain->name);
  free (chain);
  chain = next;
  }
}
```

Occasionally, `free()` can actually return memory to the operating system and make the process smaller. Usually, all it can do is allow a later call to `malloc()` to reuse the space. In the meantime, the space remains in your program as part of a free-list used internally by `malloc()`.

There is no point in freeing blocks at the end of a program because all of the program's space is given back to the system when the process terminates.

7.10 Changing the Size of a Block

Often you do not know for certain how big a block you will actually need when it's the time to use it. For example, the block might be a buffer that you use to hold a line being read from a file; initially no matter for how long you use the buffer, you may encounter a line that is longer than the block. To cope with such situations, the `realloc` function proves handy.

`realloc()` function

We can make the block longer by calling `realloc()`. This function is declared in `<stdlib.h>`.

The function action, header file required and syntax of the `realloc()` function are as follows:

Function action

`realloc` – makes the block longer.

Header file required

`#include<stdlib.h>`

Syntax

```
void * realloc (void *ptr, size_t newsize)
```

The `realloc()` function changes the size of the block whose address is `ptr` to `newsize`.

Since the space after the end of the block may be in use, `realloc()` may find it necessary to copy the block to a new address where more free space is available. The value of `realloc() is` the new address of the block. If the block needs to be moved, `realloc()` copies the old contents.

If you pass a null pointer for `ptr`, `realloc()` behaves just like `malloc()` (`newsize`). This can be convenient; but beware that older implementations (before ISO C) may not support this behaviour, and will probably crash when `realloc()` is passed a null pointer.

Like `malloc(), realloc()` may return a null pointer if no memory space is available to make the block longer. When this happens, the original block is untouched; it has not been modified nor reallocated.

In most cases it makes no difference what happens to the original block when `realloc()` fails because the application program cannot continue when it is out of memory; the only thing it does is give a fatal error message. Often it is convenient to write and use a subroutine, conventionally called `xrealloc()`.

The following program snippet displays that section of code that takes care of the error message as `xmalloc()` does for `malloc()`:

```
void *
xrealloc (void *ptr, size_t size)
{
  register void *value = realloc (ptr, size);
  if (value == 0)
```

```
      fatal ("Virtual memory exhausted");
    return value;
  }
```

You can also use `realloc()` to make a block shorter. The reason you would do this is to avoid tying up a lot of memory space when only a little space is needed. In several allocation implementations, making a block shorter sometimes necessitates copying it; so it can fail if no other space is available.

If the new size you specify is the same as the old size, `realloc()` is guaranteed to change nothing and return the same address that you gave.

7.11 `calloc()` Function

The `calloc()` function allocates memory and clears it to zero. It is declared in `<stdlib.h>.`

The function action, header file required and syntax of the `calloc()` function are as follows:

Function action

`calloc` – allocates cleared space.

Header file required

`#include<stdlib.h>`

Syntax

`void * calloc (size_t count, size_t eltsize)`

This function allocates a block long enough to contain a vector of `count` elements, each of size `elt-size`. The block's contents are cleared to zero before `calloc()` returns.

You can define `calloc()` as follows:

```
void *
calloc (size_t count, size_t eltsize)
{
  size_t size = count * eltsize;
  void *value = malloc (size);
  if (value != 0)
    memset (value, 0, size);
  return value;
}
```

In general, it is not guaranteed that `calloc()` calls `malloc()` internally. Therefore, if an application provides its own `malloc()/realloc()/free()` outside the C library, it should always define `calloc()` also.

7.12 Efficiency Considerations for `malloc()`

As opposed to other versions, `malloc()` in the GNU C library does not round up block sizes to powers of two, neither for large nor for small sizes. Neighbouring chunks can be coalesced on a free section, no matter what their size is. This makes the implementation suitable for all kinds of allocation patterns without generally incurring high memory waste through fragmentation.

Very large blocks (much larger than a page) are allocated with `mmap()` (anonymous or via `/dev/zero`) by this implementation. A great advantage of this implementation is that these chunks are returned to the system immediately when they are freed. Therefore, it cannot happen that a large chunk becomes 'locked' in between smaller ones and even after calling the `free()` function wastes memory. The size threshold for `mmap()` can be adjusted with `mallopt()`. The use of `mmap()` can also be disabled completely.

7.13 Allocating Aligned Memory Blocks

The address of a block returned by `malloc()` or `realloc()` in the GNU system is always a multiple of eight (or 16 on 64-bit systems). If you need a block whose address is a multiple of a higher power of two, use `memalign()`, `posix_memalign()` or `valloc()`. `memalign()` is declared in `<malloc.h>` and `posix_memalign()` is declared in `<stdlib.h>`.

With the GNU library, you can use `free()` to free the blocks that `memalign()`, `posix_memalign()` and `valloc()` return. That does not work in BSD; however, BSD does not provide any way to free such blocks.

1. `memalign()` function

The `memalign()` function allocates a block of `size` bytes whose address is a multiple of `boundary`. The `boundary` must be a power of two. The `memalign()` function works by allocating a somewhat larger block, and then returning an address within the block that is on the specified boundary.

The function action, header file required and syntax of the `memalign()` function are as follows:

Function action

`memalign` – allocates a block of `size` bytes whose address is a multiple of `boundary`.

Header file required

```
#include <stdlib.h>
```

Syntax

```
void * memalign (size_t boundary, size_t size)
```

2. `posix_memalign()` function

The `posix_memalign()` function is similar to the `memalign()` function in that it returns a buffer of `size` bytes aligned to a multiple of `alignment`. But it adds one requirement to the parameter `alignment` – that is, the value must be a power of two, multiple of size of (`void *`).

The function action, header file required and syntax of the `posix_memalign()` function are as follows:

Function action

`posix_memalign` – returns a buffer of `size` bytes aligned to a multiple of `alignment` same as the `memalign()` function.

Header file required

```
#include<stdlib.h>
```

Syntax

```
int posix_memalign (void **memptr, size_t alignment, size_t size)
```

If the function succeeds in allocating memory, a pointer to the allocated memory is returned in `*memptr` and the return value is zero. Otherwise, the function returns an error value indicating the problem.

This function was introduced in POSIX 1003.1d.

3. `valloc()` function

Using the `valloc()` function is like using the `memalign()` function and passing the page size as the value of the second argument.

The function action, header file required and syntax of the `valloc()` function are as follows:

Function action

`valloc` – allocates a block of `size` bytes by passing the page size

Header file required

```
#include <stdlib.h>
```

Syntax

```
void * valloc (size_t size)
```

The `valloc()` function is implemented in the following way:

```
void *
valloc (size_t size)
{
  return memalign (getpagesize (), size);
}
```

7.14 Malloc Tunable Parameters

`mallopt()` function

We can adjust some parameters for dynamic memory allocation with the `mallopt()` function. This function is the general SVID/XPG interface, defined in <`malloc.h`>.

The function action, header file required and syntax of the `mallopt()` function are as follows:

Function action

`mallopt` – adjusts some parameters for dynamic memory allocation.

Header file required

```
#include<malloc.h>
```

Syntax

```
int mallopt (int param, int value)
```

When calling `mallopt()`, the `param` argument specifies the parameter to be set, and `value` gives the new value to be set. Possible choices for `param`, as defined in `<malloc.h>`, are:

M_TRIM_THRESHOLD

This is the minimum size (in bytes) of the top-most, releasable chunk that will cause `sbrk` to be called with a negative argument in order to return memory to the system.

M_TOP_PAD

This parameter determines the amount of extra memory to be obtained from the system when a call to `sbrk` is required. It also specifies the number of bytes to be retained when shrinking the heap by calling `sbrk` with a negative argument. This provides the necessary hysteresis in heap size such that excessive amounts of system calls can be avoided.

M_MMAP_THRESHOLD

All chunks larger than this value are allocated outside the normal heap, using the `mmap()` system call. This way it is guaranteed that the memory for these chunks can be returned to the system on `free()`. Note that requests smaller than this threshold might still be allocated via `mmap()`.

M_MMAP_MAX

This parameter determines the maximum number of chunks to be allocated. Setting this to zero disables all use of `mmap()`.

7.15 Freeing Memory

The memory allocated with `malloc()` lasts as long as you want it to. It does not automatically disappear when a function returns as automatic-duration variables do, but it does not have to remain for the entire duration of your program. Just as you can use `malloc()` to control exactly when and how much memory you allocate, you can also control exactly when you de-allocate it.

In fact, many programs use memory on a transient basis. They allocate some memory, use it for a while, but then reach a point where they don't need that particular piece any more. Because memory is not inexhaustible, it's a good idea to de-allocate (i.e. release or free) the memory you're no longer using.

Dynamically allocated memory is de-allocated with the `free()` function. If `p` contains a pointer previously returned by `malloc()`, you can call

```
free(p);
```

which will give the memory back to the stock of memory (sometimes called the 'arena' or 'pool') from which `malloc()` requests are satisfied. Calling `free()` is sort of the ultimate in recycling – it costs you almost nothing, and the memory you give back is immediately usable by other parts of your program. (Theoretically, it may even be usable by other programs.)

(Freeing unused memory is a good idea, but it's not mandatory. When your program exits, any memory which it has allocated but not freed should be automatically released. If your computer were to somehow 'lose' memory just because your program forgot to free it, it would indicate a problem or deficiency in your operating system.)

Naturally, once you've freed some memory, you must remember not to use it any more. After calling

```
free(p);
```

it is probably the case that `p` still points at the same memory. However, since we have given it back, it's now 'available', and a later call to `malloc()` might give that memory to some other part of your program. If variable `p` is a global variable or will otherwise stick around for a while, one good way to record the condition that it's not to be used any more would be to set it to a null pointer as follows:

```
free(p);
p = NULL;
```

Now we don't even have the pointer to the freed memory any more, and (as long as we check to see that `p` is non-NULL before using it) we won't misuse any memory via the pointer `p`.

When thinking about `malloc()`, `free()`, and dynamically allocated memory in general, remember again the distinction between a pointer and what it points to. If you call `malloc()` to allocate some memory, and store the pointer which `malloc()` gives you in a local pointer variable, what happens when the function containing the local pointer variable returns? If the local pointer variable has automatic duration (which is the default unless the variable is declared static), it will disappear when the function returns. That memory still exists and, as far as `malloc()` and `free()` are concerned, is still allocated. The only thing that has disappeared is the pointer variable you had and which pointed at the allocated memory. (Furthermore, if the memory that is the only copy of the pointer we had, once it disappears we will have no way of freeing the memory, and no way of using it either. Using memory and freeing memory both require that we have at least one pointer to the memory.)

7.16 File Locking

File locking provides a very simple yet incredibly useful mechanism for coordinating file accesses. There are two types of locking mechanism – mandatory and advisory. Mandatory systems will actually prevent `read()`s and `write()`s to file. Several UNIX systems support them. But, here we are going to ignore them, preferring to talk solely about advisory locks in this chapter. With an advisory lock system, processes can still read and write from a file while it is locked. Is this locking useless? Not quite, since there is a way for a process to check for the existence of a lock before a read or write. It is a kind of cooperative locking system that is easily sufficient for almost all cases where file locking is necessary.

Now, let us know more about the advisory locks. There are two types of advisory locks – read locks and write locks (also referred to as shared locks and exclusive locks, respectively). The read locks work in a way that they don't interfere with other read locks. For instance, multiple processes can have a file locked for reading at the same time. However, when a process has a write lock on a file, no other process can activate either a read or write lock on this file until its present write lock is relinquished. In other words, we can say that there could be multiple users reading a file simultaneously, but there can be only one writer at a time.

There are many ways to lock files in UNIX systems. System V likes `lockf()`. Better systems support `flock()` that offers better control over the lock, but still lacks in certain ways. For portability and for completeness, we will be talking about how to lock files using `fcntl()`. It is better to use one of the higher level `flock()` style functions if it suits your needs.

Another thing to notice is that we can't get a write lock if there are any read locks on the same region of the file. A process that is waiting to get the write lock will wait until all the read locks are cleared. One upshot of this prohibition is that we can keep piling on read locks (because a read lock doesn't stop other processes from getting read locks) and any processes waiting for a write lock will sit there and starve. No rule keeps us from adding more read locks if there is a process waiting for a write lock. We must be careful.

Practically, though, we will mostly be using write locks to guarantee exclusive access to a file for a short amount of time while it is being updated; that is, the most common use of locking.

7.16.1 Setting a Lock

The `fcntl()` function does just about everything, but we will use it only for file locking. Setting the lock consists of filling out a struct flock (declared in `<fcntl.h>`) that describes the type of the lock needed, opening the file with the matching mode, and calling the `fcntl()` function with the proper arguments.

The following example program demonstrates the use the `fcntl()` function for setting the locks.

```
struct flock fl;
int fd;

fl.l_type   = F_WRLCK;    /* F_RDLCK, F_WRLCK, F_UNLCK     */
fl.l_whence = SEEK_SET;   /* SEEK_SET, SEEK_CUR, SEEK_END */
fl.l_start  = 0;          /* Offset from l_whence         */
fl.l_len    = 0;          /* length, 0 = to EOF           */
fl.l_pid    = getpid();   /* our PID                      */

fd = open("filename", O_WRONLY);

fcntl(fd, F_SETLKW, &fl); /* F_GETLK, F_SETLK, F_SETLKW   */
```

To understand the above program, let's start with the struct flock since the fields in it are used to describe the locking action taking place. Here are some field definitions:

`l_type`

This is where you signify the type of lock you want to set. It's either F_RDLCK, F_WRLCK or F_UNLCK if you want to set a read lock write lock or clear the lock, respectively.

`l_whence`

This field determines where the `l_start` field starts from (it's like an offset for the offset). It can be either SEEK_SET, SEEK_CUR or SEEK_END, for beginning of file, current file position or end of file, respectively.

`l_start`

This is the starting offset in bytes of the lock, relative to `l_whence`.

Table 7.1 Lock types and corresponding `open()` modes

1_type	*Mode*
F_RDLCK	O_RDONLY or O_RDWR
F_WRLCK	O_WRONLY or O_RDWR

`l_len`

This is the length of the lock region in bytes (which starts from `l_start`) which is relative to `l_whence`.

`l_pid`

This contains the process ID of the process dealing with the lock. Use the `getpid()` function to get this.

In our example, we explained how to make a lock of type `F_WRLCK` (a write lock), starting relative to `SEEK_SET` (the beginning of the file), offset 0, length 0 (a zero value means 'lock to end-of-file'), with the PID set to the `getpid()` function.

The next step is to use the `open()` function to open the file since the `flock()` function needs a file descriptor of the file that's being locked. Note that when you open the file, you need to open it in the same mode as you have specified in the lock, as shown in Table 7.1. If you open the file in the wrong mode for a given lock type, the `open()` function will return EBADF.

Finally, the call to the `fcntl()` function actually sets, clears or gets the lock. Here the second argument (the `cmd`) to `fcntl()` tells it what to do with the data passed to it in the struct flock. The following list summarizes what each of the `fcntl()` function `cmd` does.

F_SETLKW

This argument tells the `fcntl()` function to attempt to obtain the lock requested in the struct flock structure. If the lock cannot be obtained (since someone else has it locked already), the `fcntl()` function will wait (block) until the lock has been cleared, then will set it itself. This is a very useful command.

F_SETLK

This function is almost identical to F_SETLKW. The only difference is that it will not wait if it cannot obtain a lock. It will return immediately with −1. This function can be used to clear a lock by setting the `l_type` field in the struct flock to `F_UNLCK`.

F_GETLK

If you want to only check whether there is a lock, but don't want to set one, you can use this command. It looks through all the file locks until it finds one that conflicts with the lock you specified in the struct flock. It then copies the conflicting lock's information into the struct and returns it to you. If it can't find a conflicting lock, the `fcntl()` function returns the struct as you passed it, except that it sets the `l_type` field to `F_UNLCK`.

In our above example, we call the `fcntl()` function with `F_SETLKW` as the argument, so it blocks until it can set the lock, then sets it and continues.

7.16.2 Clearing a Lock

After understanding the locking operation, let us learn about a somewhat easy operation, that is unlocking or clearing the lock. In the following example, we will use the same section of code that we utilized for setting the lock, and unlock the newly set lock in the end.

```
struct flock fl;
int fd;

fl.l_type   = F_WRLCK;           /* F_RDLCK, F_WRLCK, F_UNLCK    */
fl.l_whence = SEEK_SET;          /* SEEK_SET, SEEK_CUR, SEEK_END */
fl.l_start  = 0;                 /* Offset from l_whence         */
fl.l_len    = 0;                 /* length, 0 = to EOF           */
fl.l_pid    = getpid();          /* our PID                      */

fd = open("filename", O_WRONLY); /* get the file descriptor      */
fcntl(fd, F_SETLKW, &fl);        /* set the lock, waiting if necessary*/
.
.
.
fl.l_type   = F_UNLCK;           /* tell it to unlock the region */
fcntl(fd, F_SETLK, &fl);         /* set the region to unlocked   */
```

In the above code, we have just changed the `l_type` field to `F_UNLCK` (leaving the others completely unchanged!) and called the `fcntl()` function with `F_SETLK` as the command.

7.16.3 Lock Program

In the following program, `lock.c` waits for the user to hit RETURN, then locks its own source, waits for another RETURN, then unlocks it. By running this program in two (or more) screens, you can see how programs interact while waiting for locks.

It is notable that if we run lock with no command line arguments, it tries to grab a write lock (`F_WRLCK`) on its source (`lock.c`). If we start it with any command line arguments, it tries to get a read lock (`F_RDLCK`) on it.

```
#include<stdio.h>
#include<stdlib.h>
#include<errno.h>
#include<fcntl.h>
#include<unistd.h>

int main(int argc, char *argv[])
{
       /* l_type   l_whence  l_start  l_len  l_pid   */
   struct flock fl = { F_WRLCK, SEEK_SET, 0,    0,    0 };
   int fd;

  fl.l_pid = getpid();
```

```
      if (argc > 1)
          fl.l_type = F_RDLCK;
      if ((fd = open("lockdemo.c", O_RDWR)) == -1) {
          perror("open");
          exit(1);
    }
    printf("Press <RETURN> to try to get lock: ");
      getchar();
    printf("Trying to get lock...");
    if (fcntl(fd, F_SETLKW, &fl) == −1) {
          perror("fcntl");
          exit(1);
    }
    printf("got lock\n");
    printf("Press <RETURN> to release lock: ");
    getchar();
    fl.l_type = F_UNLCK;  /* set to unlock same region */
    if (fcntl(fd, F_SETLK, &fl) == −1) {
          perror("fcntl");
          exit(1);
    }
    printf("Unlocked.\n");
    close(fd);
  }
```

First we will compile this code and look for two screens. It is to be observed that when one `lock.c` program has a read lock, other instances of the program can get their own read locks with no problem. It's only when a write lock is obtained that other processes can't get a lock of any kind.

7.17 Deadlocks

We know one definition describes an operating system as a resource allocator. There are many resources that the operating system can allocate to only one process at a time, and we have seen several operating system features that allow this, such as mutexes, semaphores or file locks.

Sometimes a process has to reserve more than one resource. For example, a process which copies files from one tape to another generally requires two tape drives. A process which deals with databases may need to lock multiple records in a database.

Generally, resources allocated to a process cannot be pre-empted; this means that once a resource has been allocated to a process, there is no simple mechanism by which the system can take the resource back from the process unless the process voluntarily gives it up or the system administrator kills the process. This can lead to a situation called deadlock. A set of processes or threads is deadlocked when each process or thread is waiting for a resource to be freed that is controlled by another process. Here is an example of a situation where a deadlock can occur.

```
Mutex M1, M2;

/* Thread 1 */
while (1) {
  NonCriticalSection()
  Mutex_lock(&M1);
  Mutex_lock(&M2);
  CriticalSection();
  Mutex_unlock(&M2);
  Mutex_unlock(&M1);
}

/* Thread 2 */
while (1) {
  NonCriticalSection()
  Mutex_lock(&M2);
  Mutex_lock(&M1);
  CriticalSection();
  Mutex_unlock(&M1);
  Mutex_unlock(&M2);
}
```

Suppose `Thread 1` is running and locks `M1`, but before it can lock `M2`, it is interrupted. `Thread 2` starts running; it locks `M2`; when it tries to obtain and lock `M1`, it is blocked because `M1` is already locked (by `Thread 1`). Eventually `Thread 1` starts running again, and it tries to obtain and lock `M2`, but it is blocked because `M2` is already locked by `Thread 2`. Both threads are blocked; each is waiting for an event which will never occur.

For a deadlock to occur, four conditions, described below, must be true.

1. **Mutual exclusion:** Each resource is either currently allocated to exactly one process or it is available. (Two processes cannot simultaneously control the same resource or be in their critical section.)
2. **Hold and wait:** Processes currently holding resources can request new resources.
3. **No Pre-emption:** Once a process holds a resource, it cannot be taken away by another process or the kernel.
4. **Circular wait:** Each process is waiting to obtain a resource which is held by another process.

Deadlock can be modelled with a directed graph. In a deadlock graph, vertices represent either processes (circles) or resources (squares). A process which has acquired a resource is shown with an arrow (edge) from the resource to the process. A process which has requested a resource but has not yet been given the same is modelled with an arrow from the process to the resource. If these create a cycle, there is deadlock.

The deadlock situation of our above example code can be modeled like Figure 7.1.

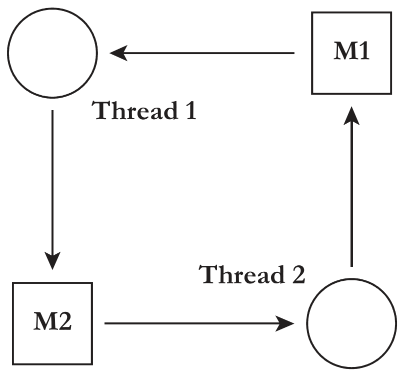


Figure 7.1 Deadlock involving two processes and two resources.

Figure 7.1 shows an extremely simple deadlock situation, but a more complex situation can also create a deadlock. Figure 7.2 shows an example of a complex deadlock involving four processes and four resources.

A deadlock can occur in an operating system in various ways. We have seen some examples, here are two more.

1. Two processes need to lock two files, the first process locks one file and the second process locks the other, and each waits for the other to free up the locked file.
2. Two processes want to write a file to a print spool area at the same time and both start writing. However, the print spool area is of fixed size, and it fills up before either process finishes writing its file, so both wait for more space to become available.

7.17.1 Solutions to Deadlock

There are several ways to address the problem of deadlock in an operating system. Some of them are described below:

1. Just ignore it and hope it doesn't happen.
2. Detection and recovery: If it happens, take action.
3. Dynamic avoidance by careful resource allocation. Check whether a resource can be granted, and if on granting there is a possibility of it causing a deadlock, don't grant it.
4. Prevention: Change the rules for assigning the resources and processes.

7.17.2 Ignore Deadlock

Deadlock is unlikely to occur very often; a system can run for years without a deadlock. If the operating system has a deadlock prevention or detection system in place, it will have a negative impact (slow-down) on the system performance. The reason is whenever a process or thread requests a resource, the system will have to check whether granting this request could cause a potential deadlock situation. If the deadlock occurs, it may be necessary to bring the system down, or at least manually kill a number of processes, but even that is not an extreme solution in most situations.

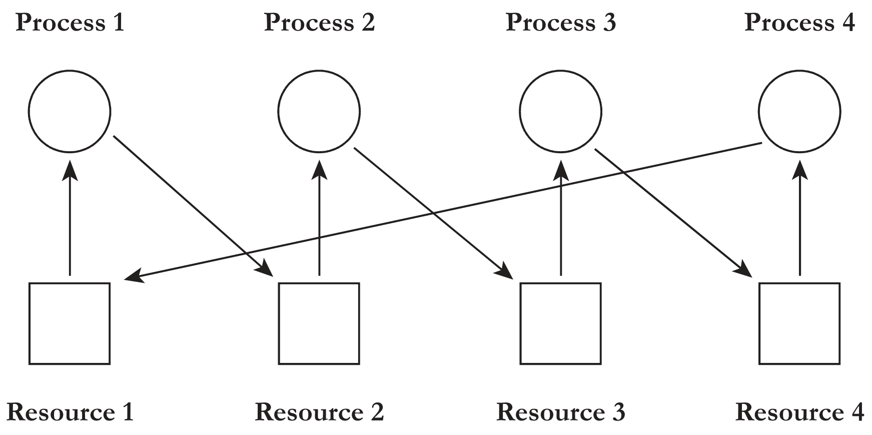


Figure 7.2 Deadlock involving four processes and four resources.

7.17.3 Deadlock Detection and Recovery

As we saw above, if there is only one instance of each resource, it is possible to detect deadlock by constructing a resource allocation/request graph and checking for cycles. Graph theorists have developed a number of algorithms to detect cycles in a graph. The following section discusses one of these. It uses only one data structure 'L' that is a list of nodes that is defined in the algorithm given below.

7.17.4 A Cycle Detection Algorithm

For each node N in the resource/allocation graph:

1. Initialize L to the empty list and designate all edges as unmarked.
2. Add the current node to L and check whether it appears twice. If it does, there is a cycle in the graph.
3. From the given node, check whether there are any unmarked outgoing edges. If yes, go to the next step; if no, skip the next step.
4. Pick an unmarked edge, mark it, then follow it to the new current node and go to Step 3.
5. We have reached a dead end. Go back to the previous node and make it the current node. If the current node is the starting node and there are no unmarked edges, there are no cycles in the graph. Otherwise, go to Step 3.

Let's work through an example with five processes and five resources. The resource request/allocation graph of this example is shown in Figure 7.3.

The algorithm needs to search each node. Let's start at node P1. We add P1 to L and follow the only edge to R1, marking that edge. R1 is now the current node, so we add that to L, checking to confirm that it is not already in L. We then follow the unmarked edge to P2, marking the edge, and making P2 the current node. We add P2 to L, checking to make sure that it is not already in L, and follow the edge to R2. This makes R2 the current node, so we add it to L, checking to make sure that it is not already

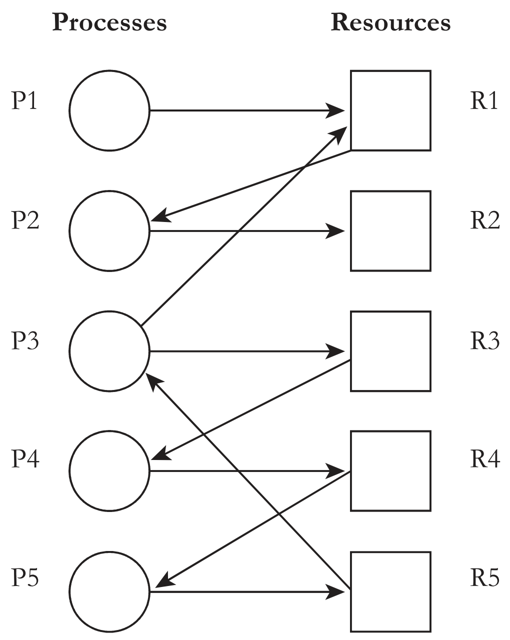


Figure 7.3 Resource request/allocation graph with five processes and five resources.

there. We are now at a dead end so we back up, making P2 the current node again. There are no more unmarked edges from P2, so we back up yet again, making R1 the current node. There are no more unmarked edges from R1, so we back up yet again, making P1 the current node. Because there are no more unmarked edges from P1 and this was our starting point, we are through with this node (and all of the nodes visited so far).

We move to the next unvisited node P3, and initialize L to empty. We first follow the unmarked edge to R1, putting R1 on L. Continuing, we make P2 the current node and then R2. Because we are at a dead end, we repeatedly back up until P3 becomes the current node again.

L now contains P3, R1, P2 and R2. P3 is the current node, and it has another unmarked edge to R3. We make R3 the current node, add it to L, follow its edge to P4. We repeat this process, visiting R4, then P5, then R5, then P3. When we visit P3 again, we note that it is already on L, so we have detected a cycle, meaning that there is a deadlock situation.

Once a deadlock has been detected, it is not clear what the system should do to correct the situation. There are three strategies, described below, to cope with a deadlock.

1. **Pre-emption:** We can take an already allocated resource away from a process and give it to another process. This can present problems. Suppose the resource is a printer and a print job is half completed. It is often difficult to restart such a job without completely starting over.
2. **Rollback:** In situations where deadlock is a real possibility, the system can periodically make a record of the state of each process; and when the deadlock occurs, it can roll everything back to the last checkpoint and restart. This time the system allocates resources differently so that the

deadlock does not occur. This means that all work done after the checkpoint is lost and will have to be redone.

3. **Kill one or more processes:** This is the simplest and crudest, but it works.

7.17.5 Deadlock Avoidance

We have just discussed how to detect a deadlock that has already occurred and try to fix the problem. Another solution is to avoid deadlock by granting only those resources whose grant cannot result in a deadlock situation later. However, this works only if the system knows what requests for resources a process will be making in the future, and this is an unrealistic assumption. The text describes the bankers algorithm but then points out that it is essentially impossible to implement because of this assumption.

7.17.6 Deadlock Prevention

There is a subtle difference between deadlock avoidance and deadlock prevention. Deadlock avoidance refers to a strategy where whenever a resource is requested, it is only granted if it cannot result in a deadlock. On the other hand, deadlock prevention strategies involve changing the rules so that processes will not make requests that could result in a deadlock.

Here is a simple example of one such strategy. Suppose every possible resource is numbered (easy enough in theory, but often hard in practice), and processes make their requests in order; that is, they cannot request a resource with a number lower than any of the resources that they have been granted so far. Deadlock cannot occur in this situation.

7.17.7 Livelock

A variant of deadlock, livelock is a situation in which two or more processes continuously change their state in response to changes in the other process(es) without doing any useful work. Livelock is similar to deadlock in that no progress is made but differs in that neither process is blocked nor waiting for anything.

A human example of livelock would be two people who meet face-to-face in a corridor and each moves aside to let the other pass, but they end up swaying from side to side without making any progress because they always move the same way at the same time.

7.17.8 Addressing Deadlock in Real Systems

None of the deadlock solutions discussed above can be implemented in a real-world, general-purpose operating system. It would be difficult to require a user program to make requests for resources in a certain way or in a certain order. As a result, most operating systems use the ostrich algorithm.

Some specialized systems have deadlock avoidance/prevention mechanisms. For example, many database operations involve locking several records, and this can result in deadlock, so database software often has a deadlock prevention algorithm.

The UNIX file locking system `lockf` has a deadlock detection mechanism built into it. Whenever a process attempts to lock a file or a record of a file, the operating system checks whether that process has locked other files or records. If the process has other files or records, the operating system uses a graph algorithm similar to the one discussed above to see whether granting that request will cause a deadlock; and, if it does, the request for the lock will fail, and the `lockf` system call will return and `errno` will be set to EDEADLK.

7.18 Summary

This chapter goes into the efficient use of memory. After discussing different types of memory, it focuses on the importance of estimating the size of actual memory used. The discussion also centres on the two types of memory allocation – static and automatic – supported by C language through the variables in C programs. The chapter also points out that the file locking provides a very simple yet incredibly useful mechanism for coordinating file accesses. Lastly, we discussed the locked lock without key (i.e deadlock), seen different consequences of deadlocks and found ways how to handle it.

8

Inter-Process Communication

A very useful feature of UNIX-like operating systems is their ability to run several processes simultaneously, and let them all share the CPU(s), memory and other resources. Any non-trivial or complex system developed on a UNIX system will sooner or later resort to splitting its tasks into more than one process as functionally we will have *n* number of processes to accomplish any task. True, many times threads will be preferred, but the methods used in both of them tend to be rather similar – how to start and stop processes, how to communicate with other processes, how to synchronize processes.

Facilities for inter-process communication (IPC) and networking were a major addition to UNIX. These facilities required major additions and some changes to the system interface. The basic idea of this interface is to make IPC similar to file I/O. In UNIX a process has a set of I/O descriptors, from which one reads and to which one writes. Descriptors may refer to normal files, to devices (including terminals) or to communication channels. The use of a descriptor has three phases: its creation, its use for reading and writing and its destruction. By using descriptors to write files, rather than simply naming the target file in the write call, one gains a surprising amount of flexibility. Often, the program that creates a descriptor will be different from the program that uses the descriptor. UNIX offers several choices for IPC. To aid the programmer in developing programs that are composed of cooperating processes, the different choices are discussed and a series of example programs are presented.

Before we talk about processes, we need to understand exactly what a process is. A process is an entity that executes a given piece of code and has its own execution stack, its own set of memory pages, its own file descriptors table and a unique process ID.

As is clear from the above definition, a process is not a program because several processes may be executing the same computer program at the same time, for the same user or for several different users. For example, there is normally one copy of the `tcsh` shell on the system, but there may be many `tcsh` processes running – one for each interactive connection of a user to the system. It is possible that many different processes will try to execute the same piece of code at the same time, perhaps try to utilize the same resources, and we should be ready to accommodate such situations. This leads us to the concept of re-entrancy.

Re-entrancy: It is the ability to have the same function (or part of a code) being in some phase of execution more than once at the same time. The re-entrancy implies that two or more processes try to execute a similar piece of code at the same time. It may also mean that a single process tries to execute the same function several times simultaneously. How is it possible? A simple example of re-entrancy is a recursive function. A process starts executing a recursive function, and somewhere in the middle (before exiting the function), it calls the same function again. This means that the function should only use local variables to save its state information. However, in a multi-process code, we don't have conflicts of variables because normally the data section of each process is

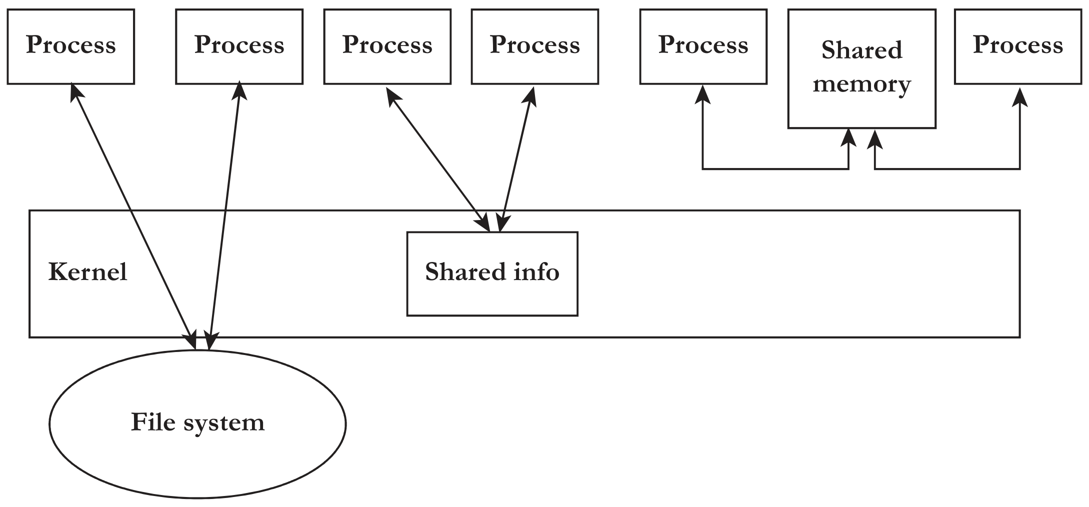


Figure 8.1 Three ways to share information between UNIX processes.

different from that of other processes. For example, process A that runs program P and process B that also runs program P have distinct copies of global variable *i* of program P. However, there might be other resources that would cause a piece of code to be non re-entrant. For example, if a program opens a file and writes some data to it, and two processes try to run the program simultaneously, the contents of the open file may get ruined in an unpredictable way. This is why a program must protect itself by using some locking mechanism that will only allow one process at a time to open the file and write data into it.

8.1 Communication among UNIX Processes

The inter-process communication (IPC) describes different ways of passing messages among different processes that are running on the same operating system. The IPC also represents numerous forms of synchronization because newer forms of communication, like shared memory, require some form of synchronization to operate.

In the traditional UNIX programming model, we have multiple processes running on a system, each process having its own address space. Information can be shared between UNIX processes in various ways. The three ways of this information sharing, depicted in Figure 8.1, have been described below.

1. The two processes on the left (shown in Figure 8.1) share some information that resides in a file in the file system. To access this data, each process must go through the kernel (e.g. use `read()`, `write()`, `lseek()` and similar functions). In this sharing some form of synchronization is required when the file is being updated, both to protect multiple writers from each other and to protect one or more readers from writer.
2. In another way of sharing depicted in Figure 8.1, the two processes in the middle are sharing some information that resides within the kernel. Here to access the shared information, each operation

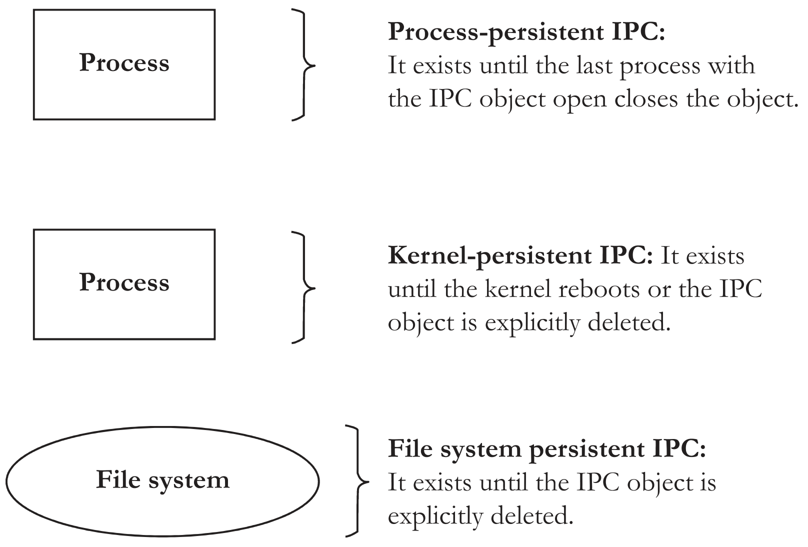

Process
Process-persistent IPC:
It exists until the last process with the IPC object open closes the object.
Process
Kernel-persistent IPC: It exists until the kernel reboots or the IPC object is explicitly deleted.
File system
File system persistent IPC:
It exists until the IPC object is explicitly deleted.

Figure 8.2 Three types of IPC persistence.

needs to make a system call to the kernel. Examples of this type of sharing are pipes, message queues and semaphores.

3. In the third way of sharing, the two processes on the right (Figure 8.1) have a region of shared memory that each process can reference. Once the shared memory is set up by each process, they can access the data in the shared memory without involving the kernel at all.

8.1.1 IPC Objects

An IPC continues or persists as long as the objects involved in it remain in existence. Three types of objects giving rise to three types of IPC persistence (Figure 8.2) have been described below.

1. **Process-persistent IPC object:** It remains in existence until the last process, which holds the object open, closes the object. Examples are pipes and FIFOs.
2. **Kernel-persistent IPC object:** It remains in existence until the kernel reboots or until the object is explicitly deleted. Message queues, semaphores and shared memory should, at least, be kernel-persistent, but they may also be file system persistent, depending on the implementation.
3. **File system persistent IPC:** This object remains in existence until it is explicitly deleted. This type of object retains its value even if the kernel reboots. Message queues, semaphores and shared memory have this property when they are implemented using mapped files.

We must be careful when defining the persistence of an IPC object because it is not always as it seems. For example, the data in a pipe is maintained within the kernel, but pipes are process-persistent, not kernel-persistent. It means when the last process that has the pipe open for reading closes the pipe, the kernel discards all the data and removes the pipe. Similarly, even though FIFOs have names within the file system, they also have process persistence because all the data in an FIFO is discarded after the last process that has the FIFO open, closes the FIFO.

No type of IPC has file system persistence. Writing data to a file provides file system persistence, but this is normally not used as a form of IPC. Most IPC forms are not intended to survive a system reboot, because processes do not survive the reboot. Requiring file system persistence would probably degrade the performance for a given form of IPC, while a common design objective of an IPC is higher performance.

8.1.2 IPC_PERM Structure

There is a predefined structure, `IPC_PERM`, in the `sys/ipc.h` header file that holds member information of `IPC_PERM` structure. The structure has the following members:

```
struct ipc_perm
{
   ushort uid;   /* Owner used id              */
   ushort gid;   /* Owners Group ID            */
   ushort cuid;  /*Creators user ID            */
   ushort cgid;  /*Creators group ID           */
   ushort mode;  /*Access modes                */
    ushort seq;  /*slot usage sequence number  */
   key_t key;    /*key                         */
};
```

Whenever a new message queue is created, the `uid, gid, cuid` and `cgid` are all set for the effective user and group. The `uid` and `gid` are called the owner IDs while the `cuid` and `cgid` are called creator IDs. The creator ID can never be changed, but an owner ID can be changed through certain system calls.

8.1.2.1 Creating and Opening IPC Channels

The three types of IPC are: shared memory, semaphore and message queue. They share many similarities in the functions that access them, and in the information that the kernel maintains on them. Table 8.1 represents the creating and opening of IPC channels based functions with respective header files used in different IPC operations.

All the IPC types `getXXX` functions that create or open an IPC object, take an IPC key value, whose type is `key_t` and return an integer identifier.

Table 8.1 Creating and opening of IPC channels

Function	*Message queues*	*Semaphores*	*Shared memory*
Header	`<sys/msg.h>`	`<sys/sem.h>`	`<sys/shm.h>`
Function to create or open	`msgget`	`semget`	`shmget`
Function for control operation	`msgctl`	`semctl`	`shmctl`
Function for IPC operation	`msgsend`	`semop`	`shmat`
	`msgrcv`		`shmdt`

All three `getXXX` functions also take a flag argument that specifies the read/write permission bits (the mode member of IPC_PERM structure) for the IPC object, and whether a new IPC object is being created or an existing one is being referenced. The rules for whether a new IPC object is created or whether an existing one is referenced are as follows:

1. Specifying a key of IPC_PRIVATE guarantees that unique IPC object is created.
2. Setting IPC_CREAT bit of the flag arguments creates a new entry for the specified key if it does not already exist. If an existing entry is found then that entry is returned.
3. Setting both the IPC_CREAT and IPC_EXCL bits of the flag argument creates a new entry for the specified key, only if the entry does not already exist. If an existing entry is found, an error of EEXIST is returned because the IPC object already exists. Setting the IPC_EXCL bit, without setting the IPC_CREAT bit, has no meaning.

8.1.3 IPC Permissions

Whenever a new IPC object is created using one of the `getXXX` functions with the IPC_CREAT flag, the following information is saved in the IPC_PERM structure.

1. Some of the bits in the flag argument initialize the mode member of the IPC_PERM structure.
2. The two members – `cuid` and `cgid` – are set to effective user ID and effective group ID of the calling process, respectively. These two members are called the creator IDs.
3. The two members – `uid` and `gid` – in IPC_PERM structure are also set to the effective user ID and effective group ID of the calling process. These two members are called the owner IDs.

Table 8.2 represents the mode values for different IPC-based read/write permissions.

The creator IDs never change although a process can change the owner IDs by calling the `ctlXX` function for the IPC mechanism with a command of IPC-SET. The three `ctlXX` functions also allow the process to change the permission bits of the `mode` member for the IPC object.

Following are the two levels of checking which are done whenever any process accesses an IPC object – once when the IPC object is opened, and then each time the IPC object is used.

1. Whenever a process establishes access to an existing IPC object with one of the `getXXX` functions, an initial check is made that the caller's flag argument does not specify any access bits that are not

Table 8.2 Mode values for IPC read/write permissions

	Symbolic	*Values*		
Numeric (octal)	*message queue*	*semaphore*	*Shared memory*	*Description*
0400	MSG_R	SEM_R	SHM_R	Read by user
0200	MSG_W	SEM_A	SHM_A	Write by user
0040	MSG_R>>3	SEM_R>>3	SHM_R>>3	Read by group
0020	MSG_W>>3	SEM_A>>3	SHM_A>>3	Write by group
0004	MSG_R>>6	SEM_R>>6	SHM_R>>6	Read by others
0002	MSG_W>>6	SEM_A>>6	SHM_A>>6	Write by other

in the `mode` member of the IPC_PERM structure. For example, a server process can set the mode member for its input message queue so that the `group_read` and the `other_read` permission bits are off. Any process that tries to specify a flag argument that includes these bits gets an error return from the `msgget` function. But this test, done by the `getXXX` functions, is of little use. It implies that the caller knows which permission category falls into user, group and other. If the creator specifically turns off certain permission bits, and if the caller specifies these bits, the error is detected by the `getXXX` function. Any process, however, can totally bypass this check by just specifying a flag argument of zero if it knows that the IPC object already exists.

2. Every IPC operation does a permission test for the process using the operation. For example, every time a process tries to put a message to a message queue with the `msgsnd` function, the following tests are performed in the order listed. As soon as a test grants access, no further tests are performed.

- The super user is always granted access.
- If the effective user id equals either the `uid` value or the `cuid` value of the IPC object, and if the appropriate access bit is on in the mode member for the IPC object, permission is granted. By appropriate access bit, we mean the read-bit must be set if the caller wants to do a read operation on the IPC object. Or, the write-bit must be set for a write operation.
- If the effective group ID equals either the `gid` value or the `cgid` value for the IPC object, and if the appropriate access bit is on in the mode member for the IPC object, permission is granted.

If none of the above tests are true, the appropriate 'other' access bit must be on in the mode member for the IPC object for the grant of permission.

8.2 Pipe

Among the mechanisms that allow related processes to communicate is the pipe or the anonymous pipe. A pipe is a one-way mechanism that allows two related processes (i.e. one is an ancestor of the other) to send a byte stream from one of them to the other. Naturally, to use such a channel properly, one needs to form some kind of protocol in which data is sent over the pipe. Also, if we want a two-way communication, we will need two pipes, and a lot of caution.

The system assures us of one thing: The order, in which data is written to the pipe, is the same as that in which data is read from the pipe. The system also assures that data won't get lost in the middle unless one of the processes (the sender or the receiver) exits prematurely.

8.2.1 `pipe()` System Call

This system call is used to create a read/write pipe that may later be used to communicate with a process we will fork off. The call takes as an argument an array of two integers that will be used to save the two file descriptors used to access the pipe.

The function action, header file required and syntax of the `pipe()` function are as follows:

Function action

`pipe` – creates an inter-process channel.

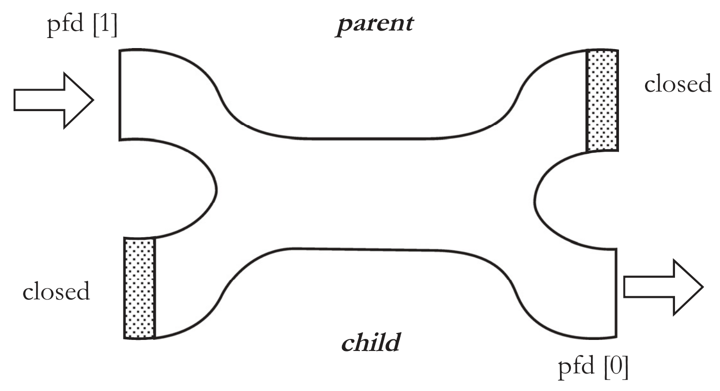


Figure 8.3 The `pipe()` system call.

Header file required

```
#include <unistd.h>
```

Syntax

```
int pipe(int fildes[2]);
```

Return values: Upon successful completion, 0 shall be returned. Otherwise, −1 shall be returned and `errno` set to indicate the error.

Errors: The `pipe()` function shall fail in case of the following errors:

EMFILE

More than {OPEN_MAX} minus two file descriptors are already in use by this process.

ENFILE

The number of simultaneously open files in the system would exceed a system-imposed limit.

Figure 8.3 depicts the structure using two file descriptors which represent parent and child. As seen in the figure, the first operation is to read from the pipe, and then write to the pipe. Prog. 1 shows the use of this function.

Prog. 1

The following program demonstrates the creation of two file descriptors using the `pipe()` system call. This program also gives information about the write and read file descriptors.

```
/* first, define an array to store the two file descriptors */
int pipes[2];

/* now, create the pipe */
int rc = pipe(pipes);
```

```
if (rc == −1) { /* pipe() failed */
  perror("pipe");
  exit(1);
}
```

As shown in Prog. 1 if the call to `pipe()` succeeds, a pipe will be created. `pipe[0]` will contain the number of its read file descriptor, and `pipe[1]` will contain the number of its write file descriptor.

Prog. 2

This program demonstrates the creation of two file descriptors using the `pipe()` system call. These file descriptors are used to write to, and read from a pipe.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <unistd.h>

int main()
{
  int pfds[2];
  char buf[30];

  if (pipe(pfds) == −1) {
    perror("pipe");
    exit(1);
  }
  printf("writing to file descriptor #%d\n", pfds[1]);
  write(pfds[1], "test", 5);
  printf("reading from file descriptor #%d\n", pfds[0]);
  read(pfds[0], buf, 5);
  printf("read \"%s\"\n", buf);
}
```

As you can see in the above program, `pipe()` takes an array of two ints as an argument. Assuming no errors, it connects two file descriptors and returns them in the array. The first element of the array is the reading end of the pipe, the second is the writing end.

Now that a pipe has been created, it should be put to some real use. To do this, we first call `fork()` to create a child process. As the memory image of the child process is identical to the memory image of the parent process, the `pipes[]` array is still defined the same way for both of them, therefore both have the file descriptors of the pipe. We will provide a solution to this situation in the subsequent section where we will use the `fork()` function to communicate between child and parent processes. Further, as the file descriptor table is also copied during `fork()`, the file descriptors are still valid inside the child process.

8.2.2 fork() and pipe()

From Prog. 2, it is pretty hard to see how communication would even be useful as we are having only one process to communicate. Well, since this is an IPC document, let us put a `fork()` in the mix and see what happens.

First, we will have the parent make a pipe. Second, we will `fork()`. The child will receive a copy of all the parent's file descriptors, and this includes a copy of the pipe's file descriptors. The child will be able to send stuff to the write-end of the pipe, and the parent will get it off the read-end.

Prog. 3

This program demonstrates how we create two file descriptors using the `pipe()` system call and then use `fork()` function to create a child process that writes to the pipe, while the parent reads from it.

```
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/types.h>
    #include <unistd.h>

    int main()
    {
      int pfds[2];
      char buf[30];

      pipe(pfds);

      if (!fork()) {
        printf("CHILD: writing to the pipe\n");
        write(pfds[1], "test", 5);
        printf("CHILD: exiting\n");
        exit(0);
      } else {
      printf("PARENT: reading from pipe\n");
      read(pfds[0], buf, 5);
      printf("PARENT: read \"%s\"\n", buf);
      wait(NULL);
      }
  }
```

Remember, your programs should have a lot of error checking.

Prog. 3 is just like Prog. 2 except now we `fork()` a new process and write it to the pipe, while the parent reads from it. The resultant output will be something similar to the following:

```
PARENT: reading from pipe
 CHILD: writing to pipe
 CHILD: exiting
PARENT: read "test"
```

In this case, the parent tries to read from the pipe before the child writes to it. When this happens, the parent is said to block, or sleep, until data arrives to be read. It seems that the parent tried to read and went to sleep, the child wrote and exited and the parent woke up and read the data.

Prog. 4

This program is an example of a two-process system in which one (the parent process) reads input from the user, and sends it to the other (the child), which then prints the data on the screen. The sending of the data is done using the pipe, and the protocol simply states that every byte passed via the pipe represents a single character typed by the user.

```
*
* one-way-pipe.c - example of using a pipe to communicate data
*       between a process and its child process. The parent reads
*       input from the user, and sends it to the child via a pipe.
*       The child prints the received data to the screen.
*/

#include <stdio.h>    /* standard I/O routines.
*/
#include <unistd.h>   /* defines pipe(), amongst other things.
*/

/* this routine handles the work of the child process. */
void do_child(int data_pipe[]) {
  int c;     /* data received from the parent. */
  int rc;    /* return status of read().       */

  /* first, close the un-needed write-part of the pipe. */
  close(data_pipe[1]);

  /* now enter a loop of reading data from the pipe, and print-
        ing it */
  while ((rc = read(data_pipe[0], &c, 1)) > 0) {
    putchar(c);

  }

  /* probably pipe was broken, or got EOF via the pipe. */
  exit(0);
}

/* this routine handles the work of the parent process. */
void do_parent(int data_pipe[])
```

```
{
  int c;     /* data received from the user. */
  int rc;    /* return status of getchar().  */

  /* first, close the un-needed read-part of the pipe. */
  close(data_pipe[0]);

  /* now enter a loop of read user input, and writing it to the
        pipe. */
  while ((c = getchar()) > 0) {
      /* write the character to the pipe. */
      rc = write(data_pipe[1], &c, 1);
        if (rc == −1) { /* write failed - notify the user and
                             exit */
          perror("Parent: write");
          close(data_pipe[1]);
          exit(1);
    }
  }

  /* probably got EOF from the user. */
  close(data_pipe[1]); /* close the pipe, to let the child know
                            we're done. */
  exit(0);
}

/* and the main function. */
int main(int argc, char* argv[])
{

  int data_pipe[2]; /* an array to store the file descriptors of
                       the pipe. */
  int pid;          /* pid of child process, or 0, as returned
                       via fork.    */
  int rc;           /* stores return values of various
                       routines */

  /* first, create a pipe. */
  rc = pipe(data_pipe);
  if (rc == −1) {
       perror("pipe");
       exit(1);
  }
  /* now fork off a child process, and set their handling rou-
tines.  */
  pid = fork();

  switch (pid) {
       case −1:     /* fork failed. */
            perror("fork");
            exit(1);
       case 0:     /* inside child process.  */
            do_child(data_pipe);
      /* NOT REACHED */
    default:   /* inside parent process. */
      do_parent(data_pipe);
      /* NOT REACHED */
  }

  return 0;  /* NOT REACHED */
}
```

As we can see in Prog. 4, the child process closed the write-end of the pipe (because it only needs to read from the pipe), while the parent process closed the read-end of the pipe (because it only needs to write to the pipe). This closing of the un-needed file descriptor was done to free up a file descriptor entry from the file descriptors table of the process. It isn't necessary in a small program such as this but because the file descriptors table is limited in size, we should not waste unnecessary entries.

8.2.3 Two-way Communication with Pipes

In more complex systems, one-way communication is too limiting. In cases where we need to communicate in both directions – from parent to child, and from child to parent, we need to open two pipes – one for each direction. However, here the bad news is that using two pipes might cause us to get into a situation known as deadlock. As we have already discussed in Chapter 7, deadlock is a situation in which a group of two or more processes are all waiting for a set of resources that are currently taken by other processes in the same group, or waiting for events that are supposed to be sent from other processes in the group.

Such a situation might occur when two processes communicate via two pipes. Here are two scenarios that could lead to a deadlock:

1. Both pipes are empty, and both processes are trying to read from their input pipes. Each one is blocked on the read (because the pipe is empty), and thus they will remain stuck like this forever.
2. This one is more complicated. Each pipe has a buffer of limited size associated with it. When a process writes to a pipe, the data is placed on the buffer of that pipe until it is read by the reading process. If the buffer is full, the `write()` system call gets blocked until the buffer has some free space. The only way to free space on the buffer is by reading data from the pipe. Thus, if both

processes write data, each to its 'writing' pipe, until the buffers are filled up, both processes will get blocked on the `write()` system call. Since no other process is reading from any of the pipes, our two processes have just entered a deadlock.

Prog. 5

This program is an example of a (hopefully) deadlock-free program in which one process reads the input from the user, writes it to the other process via a pipe. The second process translates each uppercase letter to a lowercase letter and sends the data back to the first process. Finally, the first process writes the data to standard output.

```
/*
 * two-way-pipe.c  - two processes communicating both ways by
using two
 *         pipes. One process reads input from the user and
                  handles
 *         it. The other process makes some translation of the
 *         input (translates upper-case letters to lower-case),
 *         and hands it back to the first process for printing.
 */

#include <stdio.h>    /* standard I/O routines. */
#include <unistd.h>  /* defines pipe(), amongst other things.*/
#include <ctype.h>   /* defines isascii(), toupper(), and other*/
        /* character manipulation routines. */

/* function executed by the user-interacting process. */
void user_handler(int input_pipe[], int output_pipe[])
{
  int c;  /* user input - must be 'int', to recognize EOF (=
             −1). */
  char ch;  /* the same - as a char. */
  int rc;  /* return values of functions. */

  /* first, close unnecessary file descriptors */
  close(input_pipe[1]); /* we don't need to write to this pipe.*/
  close(output_pipe[0]); /* we don't need to read from this
                           pipe. */

  /* loop: read input from user, send via one pipe to the trans-
lator, */
  /* read via other pipe what the translator returned, and write
     to */
  /* stdout. exit on EOF from user. */
  while ((c = getchar()) > 0) {
       /* note-when we ‘read’ and ‘write’, we must deal with a
          char, */
       /* rather then an int, because an int is longer then a
          char, */
       /* and writing only one byte from it, will lead to unex-
          pected */
       /* results, depending on how an int is stored on the sys-
          tem. */
       ch = (char)c;
       /* write to translator */
  rc = write(output_pipe[1], &ch, 1);
      if (rc == −1) {/* write failed-notify the user and exit.*/
                  perror(“user_handler: write”);
                  close(input_pipe[0]);
                  close(output_pipe[1]);
                  exit(1);
  }

       /* read back from translator */
   rc = read(input_pipe[0], &ch, 1);
   c = (int)ch;
   if (rc <= 0) { /* read failed-notify user and exit. */
     perror(“user_handler: read”);
     close(input_pipe[0]);
     close(output_pipe[1]);
     exit(1);

  }
     /* print translated character to stdout. */
     putchar(c);
  }

  /* close pipes and exit. */
  close(input_pipe[0]);
  close(output_pipe[1]);
  exit(0);
}


/* now comes the function executed by the translator process.*/
void translator(int input_pipe[], int output_pipe[])
{

int c;    /* user input - must be ‘int’, to recognize EOF
             (= -1). */
```

```
char ch;  /* the same - as a char. */
int rc;   /* return values of functions. */

/* first, close unnecessary file descriptors */
close(input_pipe[1]); /* we don’t need to write to this pipe.*/
close(output_pipe[0]); /* we don’t need to read from this pipe.*/

/* enter a loop of reading from the user_handler’s pipe, trans-
   lating */
/* the character, and writing back to the user handler.  */
while (read(input_pipe[0], &ch, 1) > 0) {
     c = ch;

     /* translate any upper-case letter to lowr-case. */
     if (isascii(c) && isupper(c))
   c = tolower(c);

    ch = c;
/* write translated character back to user_handler. */
rc = write(output_pipe[1], &ch, 1);
if (rc == −1) { /* write failed-notify user and exit. */
     perror(“translator: write”);
     close(input_pipe[0]);
     close(output_pipe[1]);
     exit(1);
  }
}

/* close pipes and exit. */
close(input_pipe[0]);
close(output_pipe[1]);
exit(0);
}

/* and finally, the main function: spawn off two processes, */
/* and let each of them execute its function.       */
int main(int argc, char* argv[])
{
  /* 2 arrays to contain file descriptors, for two pipes. */
  int user_to_translator[2];
  int translator_to_user[2];
  int pid;    /* pid of child process, or 0, as returned via
                 fork.  */
  int rc;    /* stores return values of various routines. */
```

```
    /* first, create one pipe. */
    rc = pipe(user_to_translator);
    if (rc == −1) {
          perror("main: pipe user_to_translator");
          exit(1);
    }
    /* then, create another pipe. */
    rc = pipe(translator_to_user);
    if (rc == −1) {
          perror("main: pipe translator_to_user");
          exit(1);
    }

  /* now fork off a child process, and set their handling rou-
  tines. */
  pid = fork();

  switch (pid) {
       case −1:    /* fork failed. */
       perror("main: fork");
       exit(1);
     case 0:        /* inside child process.  */
        translator(user_to_translator, translator_to_user); /*
                  line 'A' */
        /* NOT REACHED */
     default:  /* inside parent process. */
        user_handler(translator_to_user, user_to_translator); /*
                    line 'B' */
        /* NOT REACHED */
    }

    return 0;  /* NOT REACHED */
  }
```

Now we will understand the basic execution of Prog. 5.

1. **Character handling**: `isascii()` is a function that checks whether the given character code is a valid ASCII code. `isupper()` is a function that checks if a given character is an uppercase letter. `tolower()` is a function that translates an uppercase letter to its equivalent lowercase letter.
2. Note that both functions get an `input_pipe` and an `output_pipe` array. However, when calling the functions, we must make sure that we give one array as its input pipe, and the other as its output pipe, and vice versa. Failing to do that, the `user_handler` function will write a character to one pipe, and then both functions will try to read from the other pipe, thus causing both of them to block, as this other pipe is still empty.

8.3 Named Pipes

One limitation of anonymous pipes is that only processes 'related' to the process that created the pipe (i.e. siblings of that process) may communicate using them. If we want two unrelated processes to communicate via pipes, we need to use the named pipes.

Named pipes – also called named FIFOs (first in, first out) or just FIFOs – are identified by their access point which is basically in a file kept on the file system. Because named pipes have the pathname of a file associated with them, it is possible for unrelated processes to communicate with each other; in other words, two unrelated processes can open the file associated with a named pipe and begin communication. By opening the file for reading, the process has access to the reading end of the pipe, and by opening the file for writing, the process has access to the writing end of the pipe.

A named pipe must be opened either read-only or write-only. It must not be opened for read/write because it is half-duplex, that is, a one-way channel. Unlike anonymous pipes, which are process-persistent objects, named pipes are file system persistent objects – that is they exist beyond the life of the process. They have to be explicitly deleted by one of the processes by calling 'unlink' or else deleted from the file system via the command line.

A named pipe supports blocked read and write operations by default: if a process opens a file for reading, it is blocked until another process opens the file for writing, and vice versa. However, it is possible to make named pipes support non-blocking operations by specifying the O_NONBLOCK flag while opening them.

Shells make extensive use of pipes; for example, we use pipes to send the output of one command as the input of the other command. In real-life UNIX applications, named pipes are used for communication when the two processes need a simple method for synchronous communication.

8.3.1 Creating a Named Pipe

A named pipe can be created in two ways – via the command line or from within a program.

Command line method: To create a named pipe from the shell command line, one can use either `mknod` command or `mkfifo` command. For example, to create a named pipe with the file named `npipe` you can use one of the following commands:

```
$mknod npipe p
```

or

```
$mkfifo npipe
```

You can also provide an absolute path of the named pipe to be created.

We have seen that one file type is FIFO type, that is, pipe. Now if we look at the file using `'ls -l'`, we will see the following output:

```
prw-rw-r-- 1 secf other   0  Jun 6 17:35  npipe
```

The `'p'` on the first column denotes that this is a named pipe. Just like any file in the system, a named pipe has access permissions that define which users may open it, and whether for reading or writing, or both.

Creating named pipe within a program:

The `mkfifo()` function can be used to create a named pipe from within a program.

The function action, header files required and syntax of the `mkfifo()` function are as follows:

Function action

`mkfifo` – creates a named pipe.

Header file required

```
#include <sys/types.h>
#include <sys/stat.h>
```

Syntax

```
int mkfifo(const char *path, mode_t mode)
```

The `mkfifo` function takes the path of the file and the mode (permissions) with which the file should be created. It creates the new named pipe file as specified by the path.

The function call assumes the O_CREAT |O_EXCL flags, that is, it creates a new named pipe or returns the error EEXIST if the named pipe already exists. The named pipe's owner ID is set to the process's effective user ID, and its group ID is set to the process's effective group ID. If the S_ISGID bit is set in the parent directory, the group ID of the named pipe is inherited from the parent directory.

8.3.2 Opening a named pipe

A named pipe can be opened for reading or writing, and it is handled just like any other normal file in the system. For example, a named pipe can be opened by using the `open()` system call, or by using the `fopen()` standard C library function.

As with normal files, if the call succeeds, you will get a file descriptor in the case of `open()`, or a 'FILE' structure pointer in the case of `fopen()`, which you may use for either reading or writing, depending on the parameters passed to `open()` or to `fopen()`.

Therefore, from a user's point of view, once you have created a named pipe, you can treat it as a file as far as the operations for opening, reading, writing and deleting are concerned.

8.3.3 Reading from and Writing to a Named Pipe

Reading from and writing to a named pipe are very similar to reading from and writing to a normal file. The standard C library function calls `read()` and `write()` can be used for reading from and writing to a named pipe. These operations are blocking by default.

The following points need to be kept in mind while doing read/write to a named pipe:

1. A named pipe cannot be opened for both reading and writing. The process opening it must choose either read mode or write mode. The pipe opened in one mode will remain in that mode until it is closed.
2. Read and write operations to a named pipe are blocking by default. Therefore if a process reads from a named pipe and if that pipe does not have data in it, the reading process will be blocked. Similarly if a process tries to write to a named pipe that has no reader, the writing process gets blocked until another process opens the named pipe for reading. This, of course, can be overridden by specifying the O_NONBLOCK flag while opening the named pipe.
3. Seek operations (via the standard C library function lseek()) cannot be performed on named pipes.

8.3.4 Full-Duplex Communication Using Named Pipes

Although named pipes give a half-duplex (one-way) flow of data, you can establish full-duplex communication by using two different named pipes, with each named pipe providing the flow of data in one direction. However, you have to be very careful about the order in which these pipes are opened in the client and server, otherwise a deadlock may occur. Let us understand this through the following example.

Suppose you create two named pipes – NP1 and NP2 to establish a full-duplex channel. Here is how the server and the client should treat these two named pipes:

Let us assume that the server opens named pipe NP1 for reading and the second pipe NP2 for writing. Then to ensure that this works correctly, the client must open the first named pipe NP1 for writing and the second named pipe NP2 for reading. This way a full-duplex channel can be established between the two processes. Failure to observe the just mentioned sequence may result in a deadlock situation.

8.3.5 Benefits of Named Pipes

1. Named pipes are very simple to use.
2. `mkfifo` is a thread-safe function.
3. No synchronization mechanism is needed when using named pipes.
4. Write (using write function call) to a named pipe is guaranteed to be automatic. It is automatic even if the named pipe is opened in non-blocking mode.
5. Named pipes have permissions (read and write) associated with them unlike anonymous pipes. These permissions can be used to enforce secure communication.

8.3.6 Limitations of Named Pipes

1. Named pipes can only be used for communication among processes on the same host machine.
2. Named pipes can be created only in the local file system of the host, that is, you cannot create a named pipe on the NFS file system.
3. Due to the basic blocking nature of pipes, careful programming is required for the client and server to avoid deadlocks.
4. Named pipe data is a byte stream, and no record identification exists.

Now we will see the client/server implementation of named pipe. That is also visualized in Figure 8.4. The following code samples illustrate half-duplex and full-duplex communication between two unrelated processes by using named pipes.

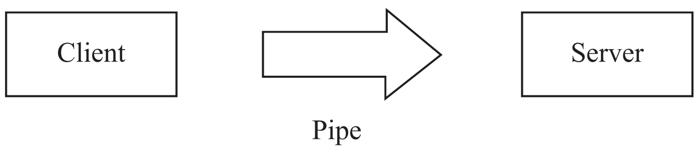


Figure 8.4 Client-server architecture through pipe.

Prog. 6

In this example, first we will develop two programs – `server.c` and `client.c` – with the help of named pipe. Then we will establish communication between the client and the server.

Server code

```
#define HALF_DUPLEX "/tmp/halfduplex"
#define MAX_BUF_SIZE 255

#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
   int fd, ret_val, count, numread;
   char buf[MAX_BUF_SIZE];

   /* Create the named - pipe */
   ret_val = mkfifo(HALF_DUPLEX, 0666);

   if ((ret_val == −1) && (errno != EEXIST)) {
     perror("Error creating the named pipe");
     exit (1);
}

/* Open the pipe for reading */
fd = open(HALF_DUPLEX, O_RDONLY);

/* Read from the pipe */
numread = read(fd, buf, MAX_BUF_SIZE);
buf[numread] = '0';
printf("Half Duplex Server : Read From the
pipe : %sn", buf);

/* Convert to the string to upper case */
count = 0;
while (count < numread) {
   buf[count] = toupper(buf[count]);
   count++;
   }
   printf("Half Duplex Server : Converted String : %sn", buf);
}
```

Prog. 7

This program deals with the client code of example Prog. 6.

Client code

The following section shows the contents of file client.c.

```
#include <stdio.h>
#include <errno.h>
#include <ctype.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])

{
    int fd;

    /* Check if an argument was specified. */

  if (argc != 2) {
    printf("Usage : %s <string to be sent to the server>n",
argv[0]);
    exit (1);
  }

  /* Open the pipe for writing */
  fd = open(HALF_DUPLEX, O_WRONLY);


  /* Write to the pipe */
  write(fd, argv[1], strlen(argv[1]));
}
```

In Progs. 6 and 7, a client and server use named pipes for one-way communication. The server creates a named pipe, opens it for reading and waits for input on the read-end of the pipe. As named pipe reads are blocking by default, the server waits for the client to send some request on the pipe. Once data becomes available, it converts the string to uppercase and prints via STDOUT.

The client opens the same named pipe in write mode and writes a user-specified string to the pipe.

Running the client and the server: When you run the server, it will create two named pipes and will block on the read call. It will wait until the client writes something to the named pipe. After that it will convert the string to uppercase and then write it to the other pipe, which will be read by the client and displayed on STDOUT. When you run the client, you will need to give a string as an argument.

Make sure you run the server first so that the named pipe gets created.

Expected output of client/server code:

1. Run the server:
   ```
   $fd_server &
   ```
   The server program will block here, and the shell will return control to the command line.
2. Run the client:
   ```
   $fd_client hello
   ```
   The client program will send the string to the server and block on the read to await the server's response.
3. The server prints the following:
   ```
   Full Duplex Server : Read From the pipe : hello
   ```
4. The client prints the following:
   ```
   Full Duplex Client : Read From the pipe : HELLO
   ```

8.4 Message Queues

One problem with pipes is that it is up to you, as a programmer, to establish the protocol. Now, usually this protocol is based on sending separate messages. With a stream taken from a pipe, it means you have to somehow parse the bytes, and separate them into packets. Another problem is that data sent via pipes always arrives in an FIFO order. This means that before you can read any part of the stream, you have to consume all the bytes sent before the piece you are looking for, and thus you need to construct your own queuing mechanism on which you place the data you just skipped to be read later. If that is what you're interested at, this is a good time to get acquainted with message queues.

A message queue is a queue onto which messages can be placed. A message is composed of a message type (which is a number) and message data. A message queue can be either private or public. If it is private, it can be accessed only by its creating process or child processes of that creator. If it is public, it can be accessed by any process that knows the queue's key. Several processes may write messages onto a message queue, or read messages from the queue. Messages may be read by type, and thus not have to be read in an FIFO order as is the case with pipes.

In the subsequent section, we will discuss message queue based functions that will be used to create message queue, receive and send messages.

8.4.1 Creating a Message Queue – `msgget()`

To use a message queue, it has to be created first. The `msgget()` system call is used to create a message queue.

The function action, header file required and syntax of the `msgget()` system call are as follows:

Function action

`msgget` – gets message queue.

Header file required

```
#include <sys/msg.h>
```

Syntax

```
int msgget(key_t key, int msgflg);
```

This system call accepts two parameters – a queue key and flags. The key may be one of the following parameters:

1. IPC_PRIVATE – used to create a private message queue.
2. A positive integer – used to create (or access) a publicly accessible message queue.

The second parameter contains flags that control how the system call is to be processed. It may contain flags like IPC_CREAT or IPC_EXCL, which behave in a similar way as O_CREAT and O_EXCL in the `open()` system call. (These will be explained later.) The second parameter also contains access permission bits. The lowest nine bits of the flags are used to define access permission for the queue, much like similar nine bits used to control access to files. The bits are separated into three groups – user, group and others. In each set, the first bit refers to read permission, the second bit to write permission, and the third bit is ignored (no execute permission is relevant to message queues).

Prog. 8

Let us work with the following example program that creates a private message queue.

```
#include <stdio.h>      /* standard I/O routines.  */
#include <sys/types.h> /* standard system data types.  */
#include <sys/ipc.h>   /* common system V IPC structures.  */
#include <sys/msg.h>   /* message-queue specific functions.  */

/*create a private message queue, with access only to the owner.*/
int queue_id = msgget(IPC_PRIVATE, 0600); /* <-- this is an
octal number.  */
if (queue_id == −1) {
  perror("msgget");
  exit(1);
}
```

In Prog. 8, the system call `msgget()` returns an integer identifying the created queue. Later on we can use this key to access the queue for reading and writing messages. The created queue belongs to the user whose process created the queue. Thus as the permission bits are '0600', only processes run on behalf of this user will have access to the queue.

Message structure – `struct msgbuf`

Before we start writing messages to a queue or reading messages from it, we need to see how a message looks. For this purpose, the system defines a structure named `'msgbuf'` described below:

```
struct msgbuf {
  long mtype;     /* message type, a positive number (cannot be
                     zero). */
  char mtext[1];  /* message body array. usually larger than one
                     byte. */
};
```

In the above structure, the message type part is rather obvious. But how do we deal with a message text that is only one byte long? Well, we actually may place a much larger text inside a message. For this, we allocate more memory for a `msgbuf` structure than `sizeof(struct msgbuf).` Let us learn to create a ‘hello world’ message by using the message structure in the following way:

```
/* first, define the message string */
char* msg_text = “hello world”;
/* allocate a message with enough space for length of string
and */
/* one extra byte for the terminating null character.  */
struct msgbuf* msg =
   (struct msgbuf*)malloc(sizeof(struct msgbuf) + strlen(msg_
          text));
/* set the message type. for example - set it to ‘1’. */
msg->mtype = 1;
/* finally, place the “hello world” string inside the message.
*/
strcpy(msg->mtext, msg_text);
```

When allocating a space for a string, one always needs to allocate one extra byte for the null character terminating the string. In our case, we allocated `strlen(msg_text)` more than the size of ‘`struct msgbuf`’, and did not need to allocate an extra place for the null character because that is already contained in the `msgbuf` structure (the one byte of mtext there).

We don’t need to place only text messages in a message. We may also place binary data. In that case, we could allocate space as large as the `msgbuf` struct plus the size of our binary data minus one byte. Of course then to copy the data to the message, we will use a function such as `memset()`, but not `strcpy()`.

8.4.2 Writing Messages onto a Queue – `msgsnd()`

Once we created a message queue and a message structure, we can place a message on the message queue, using the `msgsnd()` system call.

The function action, header file required and syntax for the `msgsnd()` system call are as follows:

Function action

`msgsnd` – sends message to the message queue.

Header file required

```
#include <sys/msg.h>
```

Syntax

```
int msgsnd(int  msqid,  const  void  *msgp,  size_t  msgsz,  int
            msgflg);
```

The `msgsnd()` system call copies our message structure and places it as the last message on the queue. It takes the following parameters:

1. `int msqid` – ID of message queue, as returned from the `msgget()` call.
2. `struct msgbuf* msg` – A pointer to a properly initialized message structure, such as the one we prepared in the previous section.
3. `int msgsz` – The size of the data part `(mtext)` of the message in bytes.
4. `int msgflg` – Flags specifying how to send the message. It may be a logical 'or' of the following flag:

   - IPC_NOWAIT – If the message cannot be sent immediately, without blocking the process, '−1' is returned and `errno` is set to EAGAIN.

To set no flags, use the value '0'.

So to send our message on the queue, we will use `msgsnd()` in the following way:

```
int rc = msgsnd(queue_id, msg, strlen(msg_text)+1, 0);
if (rc == −1) {
  perror("msgsnd");
  exit(1);
}
```

Note that we used a message size one larger than the length of the string because we are also sending the null character. `msgsnd()` assumes the data in the message to be an arbitrary sequence of bytes, so it cannot know we have got the null character there too unless we state it explicitly.

8.4.3 Reading a Message from the Queue – `msgrcv()`

We may use the system call `msgrcv()` to read a message from a message queue.

The function action, header file required and syntax of the `msgrcv()` system call are as follows:

Function action

`msgrcv` – reads a message from a message queue.

Header file required

```
#include <sys/msg.h>
```

Syntax

```
ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long int
msgtyp, int msgflg);
```

This system call accepts the following list of parameters:

1. `int msqid` – ID of the queue, as returned from `msgget()`.
2. `struct msgbuf* msg` – A pointer to a pre-allocated `msgbuf` structure. It should generally be large enough to contain a message with some arbitrary data (more explanation in the following point).

3. `int msgsz` – Size of the largest message text we wish to receive. Must not be larger than the amount of space we allocated for the message text in msg.
4. `int msgtyp` – Type of message we wish to read. It may be one of the following types:
   - 0 – The first message on the queue will be returned.
   - A positive integer – The first message on the queue whose type (`mtype`) equals this integer (unless a certain flag is set in `msgflg`). Also see the following point.
   - A negative integer – The first message on the queue whose type is less than or equal to the absolute value of this integer.
5. `int msgflg` – A logical 'or' combination of any of the following flags:
   - IPC_NOWAIT – If there is no message on the queue matching what we want to read, −1 is returned, and `errno` is set to ENOMSG.
   - MSG_EXCEPT – If the message type parameter is a positive integer, then return the first message whose type is not equal to the given integer.
   - MSG_NOERROR – If a message with a text part larger than 'msgsz' matches what we want to read, then truncate the text when copying the message to our `msgbuf` structure. If this flag is not set and the message text is too large, the system call returns −1, and `errno` is set to E2BIG.

8.4.4 Controlling Messages Queue Operations – msgctl()

The following function is used for message control operations:

Function action

`msgctl` – message control operations

Header file required

```
#include <sys/msg.h>
```

Syntax

```
int msgctl(int msqid, int cmd, struct msqid_ds *buf);
```

The `msgctl()` function provides a variety of message control operations as specified by `cmd`. The following `cmds` are available: `IPC_STAT` (places the current value of each member of the data structure associated with `msqid`), `IPC_SET` (sets the value of the members of the data structure associated with msqid), `IPC_RMID` (removes the message queue identifier specified by `msqid` from the system)

Return values: Upon successful completion, msgctl() returns 0. Otherwise, it returns −1 and sets errno to indicate the error.

Prog. 9

Now we will implement `private-queue-hello-world.c` – a 'hello world' example in which a single process creates a message queue and sends itself a 'hello world' message via queue.

```
*private-queue-hello-world.c - a "hello world" example in which
 a single
*           process creates a message queue and sends
*           itself a "hello world" message via this queue.
*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int main(int argc, char* argv[])
{
  /* create a private message queue, with access only to the
owner. */
  int queue_id = msgget(IPC_PRIVATE, 0600);
  struct msgbuf* msg;
  struct msgbuf* recv_msg;
  int rc;

  if (queue_id == -1) {
       perror("main: msgget");
       exit(1);
}
printf("message queue created, queue id '%d'.\n", queue_id);
msg = (struct msgbuf*)malloc(sizeof(struct msgbuf)+strlen("hello
world"));
msg->mtype = 1;
strcpy(msg->mtext, "hello world");
rc = msgsnd(queue_id, msg, strlen(msg->mtext)+1, 0);
if (rc == -1) {
     perror("main: msgsnd");
     exit(1);
}
free(msg);
printf("message placed on the queue successfully.\n");
recv_msg = (struct msgbuf*)malloc(sizeof(struct
msgbuf)+strlen("hello world"));
rc = msgrcv(queue_id, recv_msg, strlen("hello world")+1, 0, 0);
if (rc == -1) {
        perror("main: msgrcv");
        exit(1);
  }
  printf("msgrcv: received message: mtype '%d'; mtext '%s'\n",
        recv_msg->mtype, recv_msg->mtext);
  return 0;
  }
```

Now let us work with a sample program of complete message queue.

Prog. 10

For the sake of completeness, we will include a brace of programs that will communicate using message queues. The first program `A.c` adds messages to the message queue, and `B.c` retrieves them.

Here is the source for `A.c`:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct my_msgbuf {
    long mtype;
    char mtext[200];
};
int main(void)
{
  struct my_msgbuf buf;
  int msqid;
  key_t key;
if ((key = ftok("A.c", 'B')) == -1) {
  perror("ftok");
  exit(1);
}
if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
  perror("msgget");
  exit(1);
}
printf("Enter lines of text, ^D to quit:\n");
buf.mtype = 1; /* we don't really care in this case */
while(gets(buf.mtext), !feof(stdin)) {
  if (msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0) == −1)
    perror("msgsnd");
}
if (msgctl(msqid, IPC_RMID, NULL) == −1) {
  perror("msgctl");
  exit(1);
  }

  return 0;
}
```

In Prog. 10, the way 'A' works is that it allows you to enter lines of text. Each line is bundled into a message and added to the message queue. The message queue is then read by B.

Here is the source for `B.c`.

Prog. 11

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
struct my_msgbuf {
  long mtype;
  char mtext[200];
};
int main(void)
{
  struct my_msgbuf buf;
  int msqid;
  key_t key;

  if ((key = ftok("A.c", 'B')) == −1) {  /* same key as A.c */
    perror("ftok");
    exit(1);
  }

  if ((msqid = msgget(key, 0644)) == −1) { /* connect to the
queue */
   perror("msgget");
     exit(1);
  }
```

```
    printf("B: ready to receive messages, captain.\n");

    for(;;) { /* B never quits! */
      if (msgrcv(msqid, (struct msgbuf *)&buf, sizeof(buf), 0, 0)
         == −1) {
        perror("msgrcv");
        exit(1);
    }

      printf("B: \"%s\"\n", buf.mtext);
    }

      return 0;
}
```

In Prog. 11, notice that B, in the call to `msgget()`, does not include the IPC_CREAT option. We have left it up to A to create the message queue, and B will return an error if A hasn't done so.

Notice what happens when you are running both A and B in separate windows and you kill one or the other. Also try running two copies of A or two copies of B to get an idea of what happens when you have two readers or two writers. Another interesting demonstration is to run A, enter a bunch of messages, then run B and see it retrieve all the messages in one swoop. Just messing around with these dummy programs will help you gain an understanding of what is really going on.

8.5 Shared Memory

As we have seen, many methods were created to let processes communicate. The objective of all this communication is to share data. A major problem with all these methods is that they are sequential in nature. What can we do to allow processes to share data in a random-access manner? Here the concept of shared memory proves a great help. As you know, on a UNIX system, each process has its own virtual address space, and the system makes sure no process would access the memory area of another process. This means that if one process corrupts its memory's contents, it will not directly affect any other process in the system.

With shared memory, we declare a given section in the memory as one that will be used simultaneously by several processes. This means that the data found in this memory section (or memory segment) will be seen by several processes. This also means that several processes might try to alter this memory area simultaneously, thus some method should be used to synchronize their access to this memory area. To understand the concept of shared memory, we should first check how virtual memory is managed on UNIX systems.

Virtual memory management under Unix

To achieve virtual memory, the system divides its memory into small pages, each of the same size. For each process, a table mapping virtual memory pages into physical memory pages is kept. When a process is scheduled for running, its memory table is loaded by the operating system, and each memory access causes a mapping (by the CPU) to a physical memory page. If the virtual memory page is not found in the memory, it is looked up in the swap space, and loaded from there. This operation is called 'page in'.

When the process is started, it is being allocated a memory segment to hold the runtime stack, a memory segment to hold the programs code (the code segment), and a memory area for data (the data segment). Each segment might be composed of many memory pages. Whenever the process needs to allocate more memory, new pages are being allocated for it to enlarge its data segment.

When a process is being forked off from another process, the memory page table of the parent process is being copied to the child process, but not the pages themselves. If the child process will try to update any of these pages, only the desired page will be copied, and only the copy of the child process will be modified. This behaviour is very efficient for processes that call `fork()` and immediately use the `exec()` system call to replace the program it runs.

From the above discussion it is clear that all we need to support shared memory is some memory pages as shared, and to allow a way to identify them. This way, one process will create a shared memory segment, other processes will attach to it (by placing their physical address in the process's memory pages table). Thereafter all these processes will access the same physical memory when accessing these pages, thus sharing this memory area.

8.5.1 Allocating a Shared Memory Segment

A shared memory segment first needs to be allocated (created) using the `shmget()` system call.

The function action, header files required and syntax of the `shmget()` system call are as follows:

Function action

`shmget` – gets shared memory segment identifier.

Header file required

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

Syntax

```
int shmget(key_t key, size_t size, int shmflg);
```

This call gets a key for the segment [like the keys used in `msgget()` and `semget()`], the desired segment size and flags to denote access permissions and whether to create this page if it does not exist yet. `shmget()` returns an identifier that can be later used to access the memory segment. Here is how to use this call:

```
/* this variable is used to hold the returned segment identi-
fier. */
int shm_id;

/* allocate a shared memory segment with size of 2048 bytes,  */
/* accessible only to the current user.                  */
```

```
shm_id = shmget(100, 2048, IPC_CREAT | IPC_EXCL | 0600);
if (shm_id == -1) {
  perror("shmget:");
  exit(1);
}
```

If several processes try to allocate a segment using the same ID, they will all get an identifier for the same page unless they defined IPC_EXCL in the flags to `shmget()`. In that case, the call will succeed only if the page did not exist before.

8.5.2 Attaching and Detaching a Shared Memory Segment

After we allocated a memory page, we need to add it to the memory page table of the process. This is done using the `shmat()` (shared-memory attach) system call.

The function action, header files required and syntax of the `shmat()` system call are as follows:

Function action

`shmat` – attaches the shared memory segment

Header file required

```
#include <sys/types.h>
#include <sys/shm.h>
```

Syntax

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

Assuming `shm_id` contains an identifier returned by a call to `shmget()`, here is how to add a memory page to the memory page table:

```
/* these variables are used to specify where the page is
attached.  */
char* shm_addr;
char* shm_addr_ro;

/* attach the given shared memory segment, at some free posi-
tion */
/* that will be allocated by the system.                    */
shm_addr = shmat(shm_id, NULL, 0);
if (!shm_addr) { /* operation failed. */
  perror("shmat:");
  exit(1);
}

/* attach the same shared memory segment again, this time in */
/* read-only mode. Any write operation to this page using this
*/
```

```
/* address will cause a segmentation violation (SIGSEGV) sig-
nal. */
shm_addr_ro = shmat(shm_id, NULL, SHM_RDONLY);
if (!shm_addr_ro) { /* operation failed. */
  perror("shmat:");
  exit(1);
}
```

As you can see in the above example, a page may be attached in read-only mode, or in read/write mode. The same page may be attached several times by the same process, and then all the given addresses will refer to the same data. In our example, we can use `shm_addr` to access the segment both for reading and writing, while `shm_addr_ro` can be used for read-only access to this page. Attaching a segment in read-only mode makes sense if our process is not supposed to alter this memory page, and it is recommended in such cases. The reason is that if a bug in our process corrupts its memory image, it might corrupt the contents of the shared segment, thus causing all other processes using this segment to possibly crush. By using a read-only attachment, we protect the rest of the processes from a bug in our process.

8.5.3 Placing Data in Shared Memory

Placing data in a shared memory segment is done by using the pointer returned by the `shmat()` system call. Any kind of data may be placed in a shared segment except for pointers. The reason for this is simple: pointers contain virtual addresses. As the same segment may be attached in a different virtual address in each process, a pointer referring to one memory area in one process might refer to a different memory area in another process. We can try to overcome this problem by attaching the shared segment in the same virtual address in all processes (by supplying an address as the second parameter to `shmat()`, and adding the SHM_RND flag to its third parameter), but this may fail if the given virtual address is already in use by the process.

Here is an example of placing data in a shared memory segment, and later on reading this data. We assume that `shm_addr` is a character pointer, containing an address returned by a call to `shmat()`.

```
/* define a structure to be used in the given shared memory
segment. */
  struct country {
  char name[30];
  char capital_city[30];
  char currency[30];
  int population;
};

/* define a countries array variable. */
int* countries_num;
struct country* countries;

/* create a countries index on the shared memory segment. */
countries_num = (int*) shm_addr;
```

```
*countries_num = 0;
countries = (struct country*) ((void*)shm_addr+sizeof(int));

strcpy(countries[0].name, "INDIA");
strcpy(countries[0].capital_city, "LUCKNOW");
strcpy(countries[0].currency, "Rupees");
countries[0].population = 250000000;
(*countries_num)++;

strcpy(countries[1].name, "Bangaladesh");
strcpy(countries[1].capital_city, "dhaka");
strcpy(countries[1].currency, "taka");
countries[1].population = 6000000;
(*countries_num)++;

strcpy(countries[1].name, "Pakistan");
strcpy(countries[1].capital_city, "Islamabad");
strcpy(countries[1].currency, "Rupees");
countries[1].population = 60000000;
(*countries_num)++;

/* now, print out the countries data. */
for (i=0; i < (*countries_num); i++) {
  printf("Country %d:\n", i+1);
  printf("  name: %s:\n", countries[i].name);
  printf("  capital city: %s:\n", countries[i].capital_city);
  printf("  currency: %s:\n", countries[i].currency);
  printf("  population: %d:\n", countries[i].population);
}
```

Regarding the above example, it is important to note

1. **No usage of `malloc()`**: As the memory page was already allocated when we called `shmget()`, there is no need to use `malloc()` when placing data in that segment. Instead, we do all memory management ourselves by simple pointer arithmetic operations. We also need to make sure the shared segment was allocated enough memory to accommodate future growth of our data because there are no means of enlarging the size of the segment once allocated. This situation is unlike normal memory management where we can always move data to a new memory location using the `realloc()` function.
2. In the above example, we assumed that the page's address is aligned properly for an integer to be placed in it. If it was not, any attempt to try to alter the contents of `countries_num` would trigger a bus error (SIGBUS) signal. Further, we assumed the alignment of our structure is the same as that needed for an integer (when we placed the structures array right after the integer variable).
3. **Completeness of the data model:** By placing all the data relating to our data model in the shared memory segment, we make sure all processes attaching to this segment can use the full data kept in it.

A naive mistake would be to place the countries counter in a local variable while placing the countries array in the shared memory segment. If we did that, other processes trying to access this segment would have no means of knowing how many countries are there.

8.5.4 Destroying a Shared Memory Segment

After we finished using a shared memory segment, we should destroy it using `shmctl()` system call.

The function action, header files required and syntax of the `shmctl()` system call are as follows:

Function action

`shmctl` – provides various shared memory control operations.

Header file required

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
```

Syntax

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

It is safe to destroy a shared memory segment even if it is still in use (i.e. attached by some process). In such a case, the segment will be destroyed only after all processes detach it. Here is how to destroy a segment:

```
/* this structure is used by the shmctl() system call. */
struct shmid_ds shm_desc;

/* destroy the shared memory segment. */
if (shmctl(shm_id, IPC_RMID, &shm_desc) == −1) {
    perror("main: shmctl: ");
}
```

Remember, not only the process that created the memory segment but any process that has write permission to this segment may destroy it.

Prog. 12

This program shows how a single process uses shared memory. Naturally, when two (or more) processes use a single shared memory segment, there may be race conditions if one process tries to update this segment, while another is reading from it.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

#define SHM_SIZE 1024  /* make it a 1K shared memory segment */
```

```
int main(int argc, char *argv[])
{
  key_t key;
  int shmid;
  char *data;
  int mode;

  if (argc > 2) {
    fprintf(stderr, "usage: shmdemo [data_to_write]\n");
    exit(1);
  }

  /* make the key: */
  if ((key = ftok("shmdemo.c", 'R')) == -1) {
     perror("ftok");
    exit(1);
  }

  /* connect to (and possibly create) the segment: */
  if ((shmid = shmget(key, SHM_SIZE, 0644 | IPC_CREAT)) == −1)
{
    perror("shmget");
    exit(1);
  }

  /* attach to the segment to get a pointer to it: */
  data = shmat(shmid, (void *)0, 0);
  if (data == (char *)(21)) {
    perror("shmat");
    exit(1);
  }

  /* read or modify the segment, based on the command line: */
  if (argc == 2) {
    printf("writing to segment: \"%s\"\n", argv[1]);
    strncpy(data, argv[1], SHM_SIZE);
  } else

    printf("segment contains: \"%s\"\n", data);

  /* detach from the segment: */
  if (shmdt(data) == −1) {
    perror("shmdt");
    exit(1);
  }

  return 0;
}
```

As shown in Prog. 12, commonly a process will attach to the segment and run for a bit while other programs are changing and reading the shared segment. It is neat to watch one process update the segment and see the changes appear to other processes. Again, for simplicity, the sample code doesn't do that, but you can see how the data is shared between independent processes.

8.6 Semaphores

One major problem when writing multi-process applications is the need to synchronize various operations between the processes. Communicating requests using pipes, sockets and message queues is one way to do it. However, sometimes we need to synchronize operations amongst more than two processes, or to synchronize access to data resources that might be accessed by several processes in parallel. Semaphores are a means supplied with SysV IPC that allow us to synchronize such operations.

A semaphore is a resource that contains an integer value, and it allows processes to synchronize by testing and setting this value in a single atomic operation. This means that the process that tests the value of a semaphore and sets it to a different value (based on the test) gives guarantee that no other process will interfere with the operation in the middle.

Two types of operations – wait and signal – can be carried out on a semaphore. A set operation first checks whether the semaphore's value equals some number. If it does, it decreases its value and returns. If it does not, the operation blocks the calling process until the semaphore's value reaches the desired value. A signal operation increments the value of the semaphore, possibly awakening one or more processes that are waiting on the semaphore.

A semaphore set is a structure that stores a group of semaphores together, and possibly allows a process to commit a transaction on some or all semaphores in the set together. Here, a transaction means that we are guaranteed that either all operations are done successfully, or none is done at all. Note that a semaphore set is not a general parallel programming concept, it is just an extra mechanism supplied by SysV IPC.

8.6.1 Creating a Semaphore Set – `semget()`

Creation of a semaphore set is done using the `semget()` system call. Similar to the creation of message queues, for creating a semaphore set we supply some ID for the set, and some flags (used to define access permission mode and a few options). We also supply the number of semaphores we want to have in the given set. This number is limited to SEMMSL, as defined in the file `/usr/include/sys/sem.h`.

The function action, header files required and syntax of the `semget()` system call are as follows:

Function action

`semget` – gets set of semaphores.

Header file required

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

Syntax

```
int semget(key_t key, int nsems, int semflg);
```

The function `semget()` initializes or gains access to a semaphore. It is prototyped by:

```
int semget(key_t key, int nsems, int semflg);
```

When the call succeeds, it returns the semaphore ID (`semid`).

The key argument is an access value associated with the semaphore ID.

The `nsems` argument specifies the number of elements in a semaphore array. The call fails when `nsems` is greater than the number of elements in an existing array; when the correct count is not known, supplying 0 for this argument ensures that it will succeed.

The `semflg` argument specifies the initial access permissions and creation of control flags.

Let us work with the following example that creates the semaphore set:

```
/* ID of the semaphore set.*/
int sem_set_id_1;
int sem_set_id_2;

/* create a private semaphore set with one semaphore in it, */
/* with access only to the owner. */
sem_set_id_1 = semget(IPC_PRIVATE, 1, IPC_CREAT | 0600);
if (sem_set_id_1 == −1) {
   perror("main: semget");
   exit(1);
}

/* create a semaphore set with ID 250, three semaphores */
/* in the set, with access only to the owner.           */
sem_set_id_2 = semget(250, 3, IPC_CREAT | 0600);
if (sem_set_id_2 == −1) {
  perror("main: semget");
  exit(1);
}
```

Note that in the second case of the above example, if a semaphore set with ID 250 already existed, we would get access to the existing set rather than a new set be created. This works just like it worked with message queues.

8.6.2 Setting and Getting Semaphore Values with `semctl()`

After the semaphore set is created, we need to initialize the value of the semaphores in the set. We do this using the `semctl()` system call. This system call also has other uses, but they are not relevant to our needs right now. Suppose we want to set the values of the three semaphores in our second set to 3, 6 and 0, respectively. The ID of the first semaphore in the set is 0, the ID of the second semaphore is 1 and so on.

The function action, header files required and syntax of the `semctl()` system call are as follows:

Function action

`semctl` – discharges semaphore control operations.

Header file required

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

Syntax

```
int semctl(int semid, int semnum, int cmd, union semun arg);
```

`semctl()` must be called with a valid semaphore ID, `semid.` The `semnum` value selects a semaphore within an array by its index. The `cmd` argument is one of the following control flags:

GETVAL

Returns the value of a single semaphore.

SETVAL

Sets the value of a single semaphore. In this case, `arg` is taken as `arg.val,` an int.

GETPID

Returns the PID of the process that performed the last operation on the semaphore or array.

GETNCNT

Returns the number of processes waiting for the value of a semaphore to increase.

GETZCNT

Returns the number of processes waiting for the value of a particular semaphore to reach zero.

GETALL

Returns the values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of unsigned shorts (also see the following flag).

SETALL

Sets values for all semaphores in a set. In this case, `arg` is taken as `arg.array`, a pointer to an array of unsigned shorts.

IPC_STAT

Returns the status information from the control structure for the semaphore set and place it in the data structure pointed to by `arg.buf`, a pointer to a buffer of type `semid_ds`.

IPC_SET

Sets the effective user and group identification and permissions. In this case, `arg` is taken as `arg.buf`.

IPC_RMID

Removes the specified semaphore set.

A process must have an effective user identification of owner, creator or superuser to perform an IPC_SET or IPC_RMID command. Read and write permission is required as for other control commands.

The fourth argument `union semun arg` is optional, depending on the operation requested. If required, it is of type `union semun`, which must be explicitly declared by the application program as:

```
union semun {
  int val;
  struct semid_ds *buf;
  ushort *array;
} arg;
```

The following code illustrates `semctl()`.

```
/* use this to store return values of system calls.   */
int rc;

/* initialize the first semaphore in our set to '3'.  */
rc = semctl(sem_set_id_2, 0, SETVAL, 3);
if (rc == −1) {
  perror("main: semctl");
  exit(1);
}

/* initialize the second semaphore in our set to '6'. */
rc = semctl(sem_set_id_2, 1, SETVAL, 6);
if (rc == −1) {
  perror("main: semctl");
  exit(1);
}

/* initialize the third semaphore in our set to '0'.  */
rc = semctl(sem_set_id_2, 2, SETVAL, 0);
if (rc == −1) {
  perror("main: semctl");
  exit(1);
}
```

There is one comment to be made about the way we used `semctl()` in the above sample code. According to the UNIX manual, the last parameter for this system call should be a union of semun type union. However, since the SETVAL (set value) operation only uses the int val part of the union, we simply passed an integer to the function. The proper way to use this system call was to define a variable of this union type, and set its value appropriately, like this:

```
/* use this variable to pass the value to the semctl() call */
union semun sem_val;
/* initialize the first semaphore in our set to '3'. */
sem_val.val = 0;
rc = semctl(sem_set_id_2, 2, SETVAL, sem_val);
if (rc == -1) {
  perror("main: semctl");
  exit(1);
}
```

We used the first form just for simplicity. From now on, we will only use the second form.

8.6.3 Using Semaphores for Mutual Exclusion with `semop()`

`semop()` performs operations on a semaphore set.

The function action, header files required and syntax of the `semop()` system call are as follows:

Function action

`semop` – performs operations on a semaphore set.

Header file required

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
```

Syntax

```
int semop(int semid, struct sembuf *sops, size_t nsops);
```

The `semid` argument is the semaphore ID returned by a previous `semget()` call. The sops argument is a pointer to an array of structures, each containing the following information about a semaphore operation:

1. The semaphore number.
2. The operation to be performed.
3. Control flags, if any.

The `sembuf` structure specifies a semaphore operation, as defined in `<sys/sem.h>`.

```
struct sembuf {
   ushort_t  sem_num;   /* semaphore number */
   shor      sem_op;    /* semaphore operation */
   short     sem_flg;   /* operation flags */
};
```

The `nsops` argument specifies the length of the array, the maximum size of which is determined by the SEMOPM configuration option; this is the maximum number of operations allowed by a single `semop()` call, and is set to 10 by default. The operation to be performed is determined as follows:

1. A positive integer increments the semaphore value by that amount.
2. A negative integer decrements the semaphore value by that amount. An attempt to set a semaphore to a value less than zero fails or blocks, depending on whether IPC_NOWAIT is in effect.
3. A value of zero means to wait for the semaphore value to reach zero.

The two control flags that can be used with `semop()` are:

IPC_NOWAIT

Can be set for any operations in the array. Makes the function return without changing any semaphore value if any operation, for which IPC_NOWAIT is set, cannot be performed. The function fails if it tries to decrement a semaphore more than its current value, or tests a nonzero semaphore to be equal to zero.

SEM_UNDO

Allows individual operations in the array to be undone when the process exits.

This function takes a pointer, `sops`, to an array of semaphore operation structures. Each structure in the array contains data about an operation to perform on a semaphore. Any process with read permission can test whether a semaphore has a zero value. To increment or decrement a semaphore requires write permission. When an operation fails, none of the semaphores is altered.

The process blocks (unless the IPC_NOWAIT flag is set), and remains blocked until:

1. the semaphore operations can all finish, so the call succeeds,
2. the process receives a signal or
3. the semaphore set is removed.

Only one process at a time can update a semaphore. Simultaneous requests by different processes are performed in an arbitrary order. When an array of operations is given by a `semop()` call, no updates are done until all operations on the array can finish successfully.

If a process with exclusive use of a semaphore terminates abnormally and fails to undo the operation or free the semaphore, the semaphore stays locked in memory in the state the process left it. To prevent this, the SEM_UNDO control flag makes `semop()` allocate an undo structure for each semaphore operation, which contains the operation that returns the semaphore to its previous state. If the process dies, the system applies the operations in the undo structures. This prevents an aborted process from leaving a semaphore set in an inconsistent state.

If processes share access to a resource controlled by a semaphore, operations on the semaphore should not be made with SEM_UNDO in effect. If the process that currently has control of the resource terminates abnormally, the resource is presumed to be inconsistent. Another process must be able to recognize this to restore the resource to a consistent state. When performing a semaphore operation with SEM_UNDO in effect, you must also have it in effect for the call that will perform the reversing operation. When the process runs normally, the reversing operation updates the undo structure with a complementary value. This ensures that, unless the process is aborted, the values applied to the undo structure are cancelled to zero. When the undo structure reaches zero, it is removed.

Sometimes we have a resource that we want to allow only one process at a time to manipulate. For example, we have a file that we want written into only by one process at a time to avoid corrupting its contents. Of course, we could use various file locking mechanisms to protect the file, but we will demonstrate the usage of semaphores for this purpose as an example. Later on we will see the real usage of semaphores to protect access to shared memory segments. Anyway, here is a code snippet. It assumes the semaphore in our set whose id is '`sem_set_id`' was initialized to 1:

```
/* this function updates the contents of the file with the
given path name. */
void update_file(char* file_path, int number)
{
  /* structure for semaphore operations. */
  struct sembuf sem_op;
  FILE* file;

  /* wait on the semaphore, unless it's value is non-negative */
  sem_op.sem_num = 0;
  sem_op.sem_op = −1;   /* <-- Comment 1 */
  sem_op.sem_flg = 0;
  semop(sem_set_id, &sem_op, 1);

  /* Comment 2 */
  /* we "locked" the semaphore, and are assured exclusive access
to file. */
  /* manipulate the file in some way. for example, write a
number into it. */
  file = fopen(file_path, "w");
  if (file) {
    fprintf(file, "%d\n", number);
    fclose(file);
  }

  /* finally, signal the semaphore - increase its value by one.
*/
  sem_op.sem_num = 0;
  sem_op.sem_op = 1;   /* <-- Comment 3 */
  sem_op.sem_flg = 0;
  semop(sem_set_id, &sem_op, 1);
}
```

This above code needs some explanations, especially regarding the semantics of the `semop()` calls.

1. Before we access the file, we use `semop()` to wait on the semaphore. Supplying '−1' in sem_op.sem_op means: If the value of the semaphore is greater than or equal to '1', decrease this value by one, and return to the caller. Otherwise (if the value is 1 or less), block the calling process until the value of the semaphore becomes '1', at which point we return to the caller.

2. The semantics of `semop()` assure us that when we return from this function, the value of the semaphore is 0. Why? It couldn't be less, or else `semop()` won't return. It couldn't be more due to the way we later on signal the semaphore. And, why it cannot be more than '0'? Read on to find out.
3. After we are done with manipulating the file, we increase the value of the semaphore by 1, possibly waking up a process waiting on the semaphore. If several processes are waiting on the semaphore, the first that got blocked on it is awakened and continues its execution.

Now, let us assume that any process that tries to access a file does it only via a call to our `update_file` function. As you can see, when it goes through the function, it always decrements the value of the semaphore by 1, and then increases it by 1. Thus the semaphore's value can never go above its initial value which is '1'. Now let us check two scenarios:

1. **No other process is executing the update_file concurrently:** In this case, when we enter the function, the semaphore's value is '1'. After the first `semop()` call, the value of the semaphore is decremented to '0', and thus our process is not blocked. We continue to execute the file update, and with the second `semop()` call, we raise the value of the semaphore back to '1'.
2. **Another process is in the middle of the update_file function:** If it already managed to pass the first call to `semop()`, the value of the semaphore is 0, and when we call `semop()`, our process is blocked. When the other process signals the semaphore with the second `semop()` call, it increases the value of the semaphore back to 0, and it wakes up the process blocked on the semaphore, which is our process. We now get into executing the file handling code, and finally we raise the semaphore's value back to 1 with our second call to `semop()`.

We have the source code for a program (Prog. 13) demonstrating the mutex concept. The program launches several processes (five, as defined by the NUM_PROCS macro), each of which is executing the `update_file` function several times in a row, and then exits. Try running this program, and scan its output. Each process prints out its PID as it updates the file, so you can see what happens when. Try to play with the DELAY macro (specifying how long a process waits between two calls to `update_file`) and see how it affects the order of the operations. Check what happens if you replace the delay loop in the `do_child_loop` function with a call to `sleep()`.

Prog. 13

```
*
* sem-mutex.c  - demonstrates the usage of a semaphore as a
mutex that
*       that synchronizes accesses of multiple processes
*       to a file.
*/

#include <stdio.h>     /* standard I/O routines.  */
#include <stdlib.h>    /* rand() and srand() functions  */
#include <unistd.h>    /* fork(), etc.  */
#include <time.h>      /* nanosleep(), etc.  */
#include <sys/types.h> /* various type definitions.  */
```

```
#include <sys/ipc.h>   /* general SysV IPC structures  */
#include <sys/sem.h>   /* semaphore functions and structs.  */
#include <sys/wait.h>  /* wait(), etc.  */

#define NUM_PROCS  5    /* number of processes to launch. */
#define SEM_ID    250          /* ID for the semaphore.  */
#define FILE_NAME “sem_mutex” /* name of file to manipulate  */
#define DELAY     400000       /* delay between file updates by
one process.  */

/* this function updates the contents of the file with the
given path name.  */
void update_file(int sem_set_id, char* file_path, int number)
{

  /* structure for semaphore operations. */
  struct sembuf sem_op;
  FILE* file;

  /* wait on the semaphore, unless it’s value is non-negative.*/
  sem_op.sem_num = 0;
  sem_op.sem_op = —1;   /* <-- Comment 1 */
  sem_op.sem_flg = 0;
  semop(sem_set_id, &sem_op, 1);

  /* Comment 2 */
  /* we “locked” the semaphore, and are assured exclusive access
to file.  */
  /* manipulate the file in some way. for example, write a
number into it. */
  file = fopen(file_path, “w”);
  if (file) {
    fprintf(file, “%d\n”, number);
    printf(“%d\n”, number);
    fclose(file);
  }

  /* finally, signal the semaphore - increase its value by one.
*/
  sem_op.sem_num = 0;
  sem_op.sem_op = 1;   /* <-- Comment 3 */
  sem_op.sem_flg = 0;
  semop(sem_set_id, &sem_op, 1);
}

/* this function calls “file_update” several times in a row, */
/* and waiting a little time between each two calls, in order*/
/* to allow other processes time to operate.                  */
void do_child_loop(int sem_set_id, char* file_name)
{
  pid_t pid = getpid();
  int i, j;
  for (i=0; i<3; i++) {
      update_file(sem_set_id, file_name, pid);
      for (j=0; j<400000; j++)
        ;
  }
}

/* finally, the main() function. */
void main()
{
  int sem_set_id;     /* ID of the semaphore set.       */
  union semun sem_val;  /* semaphore value, for semctl(). */
  int child_pid;      /* PID of our child process.      */
  int i;              /* counter for loop operation.    */
  int rc;             /* return value of system calls.  */

  /* create a semaphore set with ID 250, with one semaphore   */
  /* in it, with access only to the owner.        */
  sem_set_id = semget(SEM_ID, 1, IPC_CREAT | 0600);
  if (sem_set_id == −1) {
      perror("main: semget");
      exit(1);
  }

  /* intialize the first (and single) semaphore in our set to
'1'. */
  sem_val.val = 1;
  rc = semctl(sem_set_id, 0, SETVAL, sem_val);
  if (rc == −1) {
       perror(«main: semctl»);
       exit(1);
  }

  /* create a set of child processes that will compete on the
semaphore */
  for (i=0; i<NUM_PROCS; i++) {
      child_pid = fork();
      switch(child_pid) {
        case 21:
             perror("fork");
             exit(1);
        case 0:  /* we're at child process. */
             do_child_loop(sem_set_id, FILE_NAME);
             exit(0);
        default: /* we're at parent process. */
             break;
      }
}

/* wait for all children to finish running */
  for (i=0; i<NUM_PROCS; i++) {
       int child_status;

       wait(&child_status);
  }

  printf("main: we're done\n");
  fflush(stdout);
}
```

8.6.4 Using Semaphores for Producer/Consumer Operations with `semop()`

Using a semaphore as a mutex is not utilizing the full power of the semaphore. As we have seen, a semaphore contains a counter that may be used for more complex operations. Those operations often use a programming model called 'producer/consumer'. In this model, we have one or more processes that produce something, and one or more processes that consume that something. For example, one set of processes accepts printing requests from clients and places them in a spool directory, and another set of processes takes the files from the spool directory and actually prints them using the printer.

To control such a printing system, we need the producers to maintain a count of the number of files waiting in the spool directory and incrementing it for every new file placed there. The consumers check this counter, and whenever it gets above zero, one of them grabs a file from the spool, and sends it to the printer. If there are no files in the spool (i.e. the counter value is zero), all consumer processes get blocked. The behaviour of this counter sounds very familiar – exactly this is the behaviour of a counting semaphore.

Let us see how we can use a semaphore as a counter. We still use the same two operations on the semaphore, namely signal and wait.

Prog. 14

This program demonstrates the basic producer/consumer implementation.

```
/*
* sem-producer-consumer.c  - demonstrates a basic producer-con-
  sumer
```

```
 *            implementation.
 */
#include <stdio.h>                /* standard I/O rou-
tines.       */
#include <stdlib.h>               /* rand() and srand() func-
tions   */
#include <unistd.h>               /* fork(), etc.            */
#include <time.h>                /* nanosleep(), etc.     */
#include <sys/types.h>            /* various type defini-
tions.     */
#include <sys/ipc.h>              /* general SysV IPC struc-
tures   */
#include <sys/sem.h>             /* semaphore functions and
structs.   */
#define NUM_LOOPS    20      /* number of loops to perform.  */
int main(int argc, char* argv[])
{
  int sem_set_id;          /* ID of the semaphore set.   */
  union semun sem_val;     /* semaphore value, for semctl().  */
  int child_pid;           /* PID of our child process.   */
  int i;                   /* counter for loop operation. */
  struct sembuf sem_op;    /* structure for semaphore ops.   */
  int rc;                  /* return value of system calls.  */
  struct timespec delay;     /* used for wasting time.   */
  /* create a private semaphore set with one semaphore in it, */
  /* with access only to the owner. */
  sem_set_id = semget(IPC_PRIVATE, 1, 0600);
  if (sem_set_id == −1) {
      perror("main: semget");
      exit(1);
  }
  printf("semaphore set created, semaphore set id '%d'.\n",
sem_set_id);

  /* intialize the first (and single) semaphore in our set to
'0'. */
  sem_val.val = 0;
  rc = semctl(sem_set_id, 0, SETVAL, sem_val);
  /* fork-off a child process, and start a producer/consumer
job. */
  child_pid = fork();
```

```
    switch (child_pid) {
      case -1:   /* fork() failed */
           perror("fork");
           exit(1);
      case 0:   /* child process here */
           for (i=0; i<NUM_LOOPS; i++) {
           /* block on the semaphore, unless it's value is non-
              negative. */
           sem_op.sem_num = 0;
           sem_op.sem_op = −1;
           sem_op.sem_flg = 0;
           semop(sem_set_id, &sem_op, 1);
           printf("consumer: '%d'\n", i);
           fflush(stdout);
      }
      break;
     default:  /* parent process here */
      for (i=0; i<NUM_LOOPS; i++) {
            printf("producer: '%d'\n", i);
            fflush(stdout);
            /* increase the value of the semaphore by 1. */
            sem_op.sem_num = 0;
            sem_op.sem_op = 1;
            sem_op.sem_flg = 0;
            semop(sem_set_id, &sem_op, 1);
            /* pause execution for a little bit, to allow the */
            /* child process to run and handle some requests. */
            /* this is done about 25% of the time. */
          if (rand() > 3*(RAND_MAX/4)) {
           delay.tv_sec = 0;
           delay.tv_nsec = 10;
           nanosleep(&delay, NULL);
      }
      }
      break;
    }
    return 0;
  }
```

In Prog. 14, our wait and signal operations are just like what we performed while using the semaphore as a mutex. The only difference is in who is doing the wait and who is giving the signal. With a mutex, the same process did both wait and signal (in that order) operations. In the producer/consumer example, one process is doing the signal operation, while the other is doing the wait operation.

8.6.5 Some Examples of Semaphore Programs

The following suite of programs can be used to investigate interactively a variety of semaphore scenarios.

The semaphore must be initialized with the semget.c program. The effects of controlling the semaphore queue and sending and receiving semaphore can be investigated with `semctl.c` and `semop.c`, respectively.

Let us now work with an example program using the `semget()` function.

Prog. 15

This program is a simple exercise of the `semget()` function. It prompts for the arguments, makes the call and reports the results.

```
/*
 * semget.c: Illustrate the semget() function.
 *
 * This is a simple exerciser of the semget() function. It
prompts
 * for the arguments, makes the call, and reports the results.
*/

#include   <stdio.h>
#include   <sys/types.h>
#include   <sys/ipc.h>
#include   <sys/sem.h>

extern void    exit();
extern void    perror();

main()
{
key_t  key;    /* key to pass to semget() */
int  semflg;    /* semflg to pass to semget() */
int  nsems;    /* nsems to pass to semget() */
int  semid;    /* return value from semget() */

(void) fprintf(stderr,
"All numeric input must follow C conventions:\n");
(void) fprintf(stderr,
"\t0x... is interpreted as hexadecimal,\n");
(void) fprintf(stderr, "\t0... is interpreted as octal,\n");
(void) fprintf(stderr, "\totherwise, decimal.\n");
(void) fprintf(stderr, "IPC_PRIVATE == %#lx\n", IPC_PRIVATE);
(void) fprintf(stderr, "Enter key:");
(void) scanf("%li", &key);
(void) fprintf(stderr, "Enter nsems value:");
(void) scanf("%i", &nsems);
(void) fprintf(stderr, "\nExpected flags for semflg are:\n");
(void) fprintf(stderr, "\tIPC_EXCL = \t%#8.8o\n", IPC_EXCL);
(void) fprintf(stderr, "\tIPC_CREAT = \t%#8.8o\n", IPC_CREAT);
(void) fprintf(stderr, "\towner read = \t%#8.8o\n", 0400);
(void) fprintf(stderr, "\towner alter = \t%#8.8o\n", 0200);
(void) fprintf(stderr, "\tgroup read = \t%#8.8o\n", 040);
(void) fprintf(stderr, "\tgroup alter = \t%#8.8o\n", 020);
(void) fprintf(stderr, "\tother read = \t%#8.8o\n", 04);
(void) fprintf(stderr, "\tother alter = \t%#8.8o\n", 02);
(void) fprintf(stderr, "Enter semflg value:");
(void) scanf("%i", &semflg);
(void) fprintf(stderr, "\nsemget: Calling semget(%#lx, %
  %#o)\n",key, nsems, semflg);
 if ((semid = semget(key, nsems, semflg)) == -1) {
  perror("semget: semget failed");
  exit(1);
 } else {
  (void) fprintf(stderr, "semget: semget succeeded: semid =
%d\n",
   semid);
  exit(0);
 }
}
```

Let us now work with an example program using the `semctl()` function.

Prog. 16

This program is a simple exercise of the `semctl()` function. It lets you perform one control operation on one semaphore set. It gives up immediately if any control operation fails, so be careful for not setting permissions to preclude read permission; you won't be able reset the permissions once given with this code.

```
/*
 * semctl.c:  Illustrate the semctl() function.
 *
 * This is a simple exerciser of the semctl() function. It lets you
 * perform one control operation on one semaphore set. It gives up
 * immediately if any control operation fails, so be careful not to
 * set permissions to preclude read permission; you won't be able to
 * reset the permissions with this code if you do.
 */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>
#include    <time.h>
```

```
struct semid_ds semid_ds;

static void   do_semctl();
static void   do_stat();
extern char   *malloc();
extern void   exit();
extern void   perror();

char warning_message[] = "If you remove read permission\
  for yourself, this program will fail frequently!";


main()
{
 union semun arg; /* union to pass to semctl() */
 int cmd, /* command to give to semctl() */
  i, /* work area */
  semid, /* semid to pass to semctl() */
  semnum; /* semnum to pass to semctl() */

(void) fprintf(stderr,
  "All numeric input must follow C conventions:\n");
(void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
(void) fprintf(stderr, "\t0... is interpreted as octal,\n");
(void) fprintf(stderr, "\totherwise, decimal.\n");
(void) fprintf(stderr, "Enter semid value:");
(void) scanf("%i", &semid);
(void) fprintf(stderr, "Valid semctl cmd values are:\n");
(void) fprintf(stderr, "\tGETALL = %d\n", GETALL);
(void) fprintf(stderr, "\tGETNCNT = %d\n", GETNCNT);
(void) fprintf(stderr, "\tGETPID = %d\n", GETPID);
(void) fprintf(stderr, "\tGETVAL = %d\n", GETVAL);
(void) fprintf(stderr, "\tGETZCNT = %d\n", GETZCNT);
(void) fprintf(stderr, "\tIPC_RMID = %d\n", IPC_RMID);
(void) fprintf(stderr, "\tIPC_SET = %d\n", IPC_SET);
(void) fprintf(stderr, "\tIPC_STAT = %d\n", IPC_STAT);
(void) fprintf(stderr, "\tSETALL = %d\n", SETALL);
(void) fprintf(stderr, "\tSETVAL = %d\n", SETVAL);
(void) fprintf(stderr, "\nEnter cmd:");
(void) scanf("%i", &cmd);

/* Do some setup operations needed by multiple commands. */
switch (cmd) {
 case GETVAL:
 case SETVAL:
```

```
case GETNCNT:
case GETZCNT:
 /* Get the semaphore number for these commands. */
 (void) fprintf(stderr, "\nEnter semnum value:");
 (void) scanf("%i", &semnum);
 break;
case GETALL:
case SETALL:
 /* Allocate a buffer for the semaphore values. */
(void) fprintf(stderr,
 "Get number of semaphores in the set.\n");
arg.buf = &semid_ds;
do_semctl(semid, 0, IPC_STAT, arg);
if (arg.array =
 (ushort *)malloc((unsigned)
 (semid_ds.sem_nsems * sizeof(ushort)))) {
 /* Break out if you got what you needed. */
 break;
}
(void) fprintf(stderr,
"semctl: unable to allocate space for %d values\n",
semid_ds.sem_nsems);
exit(2);
}
/* Get the rest of the arguments needed for the specified
  command. */
switch (cmd) {
 case SETVAL:
   /* Set value of one semaphore. */
   (void) fprintf(stderr, "\nEnter semaphore value:");
   (void) scanf("%i", &arg.val);
   do_semctl(semid, semnum, SETVAL, arg);
   /* Fall through to verify the result. */
   (void) fprintf(stderr,
   "Do semctl GETVAL command to verify results.\n");
 case GETVAL:
   /* Get value of one semaphore. */
   arg.val = 0;
   do_semctl(semid, semnum, GETVAL, arg);
   break;
 case GETPID:
 /* Get PID of last process to successfully complete a
 semctl(SETVAL), semctl(SETALL), or semop() on the semaphore. */
 arg.val = 0;
 do_semctl(semid, 0, GETPID, arg);
 break;
 case GETNCNT:
 /* Get number of processes waiting for semaphore value to
    increase. */
  arg.val = 0;
  do_semctl(semid, semnum, GETNCNT, arg);
  break;
 case GETZCNT:
 /* Get number of processes waiting for semaphore value to
  become zero. */
 arg.val = 0;
 do_semctl(semid, semnum, GETZCNT, arg);
 break;
case SETALL:
/* Set the values of all semaphores in the set. */
(void) fprintf(stderr,
   "There are %d semaphores in the set.\n",
   semid_ds.sem_nsems);
(void) fprintf(stderr, "Enter semaphore values:\n");
for (i = 0; i < semid_ds.sem_nsems; i++) {
(void) fprintf(stderr, "Semaphore %d:", i);
(void) scanf("%hi", &arg.array[i]);
}
do_semctl(semid, 0, SETALL, arg);
/* Fall through to verify the results. */
(void) fprintf(stderr,
 "Do semctl GETALL command to verify results.\n");
case GETALL:
/* Get and print the values of all semaphores in the
  set.*/
do_semctl(semid, 0, GETALL, arg);
(void) fprintf(stderr,
  "The values of the %d semaphores are:\n",
  semid_ds.sem_nsems);
for (i = 0; i < semid_ds.sem_nsems; i++)
(void) fprintf(stderr, "%d", arg.array[i]);
(void) fprintf(stderr, "\n");
break;
case IPC_SET:
 /* Modify mode and/or ownership. */
 arg.buf = &semid_ds;
 do_semctl(semid, 0, IPC_STAT, arg);
 (void) fprintf(stderr, "Status before IPC_SET:\n");
 do_stat();
 (void) fprintf(stderr, "Enter sem_perm.uid value:");
 (void) scanf("%hi", &semid_ds.sem_perm.uid);
 (void) fprintf(stderr, "Enter sem_perm.gid value:");
 (void) scanf("%hi", &semid_ds.sem_perm.gid);
 (void) fprintf(stderr, "%s\n", warning_message);
 (void) fprintf(stderr, "Enter sem_perm.mode value:");
 (void) scanf("%hi", &semid_ds.sem_perm.mode);
 do_semctl(semid, 0, IPC_SET, arg);
 /* Fall through to verify changes. */
 (void) fprintf(stderr, "Status after IPC_SET:\n");
case IPC_STAT:
 /* Get and print current status. */
 arg.buf = &semid_ds;
 do_semctl(semid, 0, IPC_STAT, arg);
 do_stat();
 break;
case IPC_RMID:
/* Remove the semaphore set. */
arg.val = 0;
 do_semctl(semid, 0, IPC_RMID, arg);
 break;
 default:
  /* Pass unknown command to semctl. */
  arg.val = 0;
  do_semctl(semid, 0, cmd, arg);
  break;
  }
 exit(0);
}
```

Prog. 17

This programs deals with the print indication of arguments being passed to `semctl()`, calls `semctl()` and reports the results. If `semctl()` fails, it doesn't return; this example doesn't deal with errors, it just reports them.

```
* Print indication of arguments being passed to semctl(), call
* semctl(), and report the results. If semctl() fails, do not
* return; this example doesn't deal with errors, it just reports
 * them.
 */
```

```
static void
do_semctl(semid, semnum, cmd, arg)
union semun  arg;
int  cmd,
 semid,
 semnum;

{
 register int i; /* work area */
 void) fprintf(stderr, "\nsemctl: Calling semctl(%d, %d, %d, ",
  semid, semnum, cmd);
switch (cmd) {
 case GETALL:
  (void) fprintf(stderr, "arg.array = %#x)\n",
   arg.array);
  break;
 case IPC_STAT:
 case IPC_SET:
  (void) fprintf(stderr, "arg.buf = %#x)\n", arg.buf);
  break;
 case SETALL:
  (void) fprintf(stderr, "arg.array = [", arg.buf);
  for (i = 0;i < semid_ds.sem_nsems;) {
  (void) fprintf(stderr, "%d", arg.array[i++]);
  if (i < semid_ds.sem_nsems)
   (void) fprintf(stderr, ", ");
 }

 (void) fprintf(stderr, "])\n");
 break;

case SETVAL:
default:
 (void) fprintf(stderr, "arg.val = %d)\n", arg.val);
 break;
 }
 i = semctl(semid, semnum, cmd, arg);
 if (i == −1) {
  perror("semctl: semctl failed");
  exit(1);
 }
 (void) fprintf(stderr, "semctl: semctl returned %d\n", i);
 return;
}
```

```
/*
 * Display contents of commonly used pieces of the status
structure.
 */
static void
do_stat()
{
 (void) fprintf(stderr, "sem_perm.uid = %d\n",
  semid_ds.sem_perm.uid);
 (void) fprintf(stderr, "sem_perm.gid = %d\n",
  semid_ds.sem_perm.gid);
 (void) fprintf(stderr, "sem_perm.cuid = %d\n",
   semid_ds.sem_perm.cuid);
 (void) fprintf(stderr, "sem_perm.cgid = %d\n",
   semid_ds.sem_perm.cgid);
 (void) fprintf(stderr, "sem_perm.mode = %#o,",
   semid_ds.sem_perm.mode);
 (void) fprintf(stderr, "access permissions = %#o\n",
   semid_ds.sem_perm.mode & 0777);
 (void) fprintf(stderr, "sem_nsems = %d\n",
semid_ds.sem_nsems);
 (void) fprintf(stderr, "sem_otime = %s", semid_ds.sem_otime ?
   ctime(&semid_ds.sem_otime) : "Not Set\n");
 (void) fprintf(stderr, "sem_ctime = %s",
   ctime(&semid_ds.sem_ctime));
}
```

Prog. 18

This program is a simple exercise of the `semop ( )` function. It lets you set up arguments for `semop ( )` and make the call. It then reports the results repeatedly on one semaphore set. You must have read permission on the semaphore set, otherwise this exercise will fail. It needs the read permission to get the number of semaphores in the set.

```
/*
 * semop.c: Illustrate the semop() function.
 *
 * This is a simple exerciser of the semop() function. It lets you
 * to set up arguments for semop() and make the call. It then
   reports
 * the results repeatedly on one semaphore set. You must have
   read
 * permission on the semaphore set or this exerciser will fail.
```

```
(It
  * needs read permission to get the number of semaphores in the
set
  * and to report the values before and after calls to semop().)
  */

#include    <stdio.h>
#include    <sys/types.h>
#include    <sys/ipc.h>
#include    <sys/sem.h>

static int      ask();
extern void     exit();
extern void     free();
extern char     *malloc();
extern void     perror();

static struct semid_ds  semid_ds;            /* status of sema-
phore set */

static char    error_mesg1[] = "semop: Can't allocate space for
%d\
   semaphore values. Giving up.\n";
static char   error_mesg2[] = "semop: Can't allocate space for
%d\
   sembuf structures. Giving up.\n";

main()

{
 register int i; /* work area */
 int    nsops; /* number of operations to do */
 int    semid; /* semid of semaphore set */
 struct sembuf *sops;   /* ptr to operations to perform */

 (void) fprintf(stderr,
  "All numeric input must follow C conventions:\n");
 (void) fprintf(stderr,
  "\t0x... is interpreted as hexadecimal,\n");
 (void) fprintf(stderr, "\t0... is interpreted as octal,\n");
 (void) fprintf(stderr, "\totherwise, decimal.\n");
 /* Loop until the invoker doesn't want to do anymore. */
 while (nsops = ask(&semid, &sops)) {
/* Initialize the array of operations to be performed.*/
for (i = 0; i < nsops; i++) {
 (void) fprintf(stderr,
  "\nEnter values for operation %d of %d.\n",
    i + 1, nsops);
 (void) fprintf(stderr,
  "sem_num(valid values are 0 <= sem_num < %d):",
  semid_ds.sem_nsems);
(void) scanf("%hi", &sops[i].sem_num);
(void) fprintf(stderr, "sem_op: ");
(void) scanf("%hi", &sops[i].sem_op);
(void) fprintf(stderr,
  "Expected flags in sem_flg are:\n");
(void) fprintf(stderr, "\tIPC_NOWAIT =\t%#6.6o\n",
  IPC_NOWAIT);
(void) fprintf(stderr, "\tSEM_UNDO =\t%#6.6o\n",
  SEM_UNDO);
(void) fprintf(stderr, "sem_flg:");
(void) scanf("%hi", &sops[i].sem_flg);
}

/* Recap the call to be made. */
(void) fprintf(stderr,
  "\nsemop: Calling semop(%d, &sops, %d) with:",
  semid, nsops);
for (i = 0; i < nsops; i++)
{
(void) fprintf(stderr, "\nsops[%d].sem_num = %d,", i,
  sops[i].sem_num);
(void) fprintf(stderr, "sem_op = %d,", sops[i].sem_op);
(void) fprintf(stderr, "sem_flg = %#o\n",
  sops[i].sem_flg);
   }

/* Make the semop() call and report the results. */
 if ((i = semop(semid, sops, nsops)) == −1) {
  perror("semop: semop failed");
} else {
  (void) fprintf(stderr, "semop: semop returned %d\n", i);
  }
 }
}

/*
 * Ask if user wants to continue.
 *
 * On the first call:
```

```
 * Get the semid to be processed and supply it to the caller.
 * On each call:
 *  1. Print current semaphore values.
 *  2. Ask user how many operations are to be performed on the
        next
 *  call to semop. Allocate an array of sembuf structures
 *  sufficient for the job and set caller-supplied pointer to that
 *  array. (The array is reused on subsequent calls if it is big
 *  enough. If it isn't, it is freed and a larger array is
 *  allocated.)
 */
static
ask(semidp, sopsp)
int   *semidp;  /* pointer to semid (used only the first time)
*/
struct sembuf   **sopsp;

{
static union semun arg; /* argument to semctl */
int i; /* work area */
static int nsops = 0; /* size of currently allocated
    sembuf array */
static int semid = -1;   /* semid supplied by user */
static struct sembuf *sops; /* pointer to allocated array */
if (semid < 0) {
 /* First call; get semid from user and the current state of
  the semaphore set. */
 (void) fprintf(stderr,
  "Enter semid of the semaphore set you want to use:");
 (void) scanf("%i", &semid);
 *semidp = semid;
 arg.buf = &semid_ds;
 if (semctl(semid, 0, IPC_STAT, arg) == -1) {
 perror("semop: semctl(IPC_STAT) failed");
 /* Note that if semctl fails, semid_ds remains filled
   with zeros, so later test for number of semaphores will
   be zero. */
 (void) fprintf(stderr,
  "Before and after values are not printed.\n");
 } else {
 if ((arg.array = (ushort *)malloc(
  (unsigned)(sizeof(ushort) * semid_ds.sem_nsems)))
  == NULL) {
 (void) fprintf(stderr, error_mesg1,
   semid_ds.sem_nsems);
  exit(1);
  }
 }
 }
/* Print current semaphore values. */
 if (semid_ds.sem_nsems) {
 (void) fprintf(stderr,
  "There are %d semaphores in the set.\n",
  semid_ds.sem_nsems);
if (semctl(semid, 0, GETALL, arg) == −1) {
  perror("semop: semctl(GETALL) failed");
 } else {
  (void) fprintf(stderr, "Current semaphore values are:");
  for (i = 0; i < semid_ds.sem_nsems;
  (void) fprintf(stderr, " %d", arg.array[i++]));
  (void) fprintf(stderr, "\n");
  }
 }
 /* Find out how many operations are going to be done in the
    next
  call and allocate enough space to do it. */
(void) fprintf(stderr,
  "How many semaphore operations do you want %s\n",
  "on the next call to semop()?");
 (void) fprintf(stderr, "Enter 0 or control-D to quit: ");
 i = 0;
 if (scanf("%i", &i) == EOF || i == 0)
  exit(0);
 if (i > nsops) {
  if (nsops)
  free((char *)sops);
 nsops = i;
if ((sops = (struct sembuf *)malloc((unsigned)(nsops *
  sizeof(struct sembuf)))) == NULL) {
  (void) fprintf(stderr, error_mesg2, nsops);
  exit(2);
 }
}
*sopsp = sops;
return (i);
}
```

8.7 Summary

In this chapter, we introduced the concept of inter-process communication and its implementation using pipes, named pipes, shared memory, message queues and semaphores. We have seen the benefits of named pipes over pipes. We experienced the communication using pipes for two processes to handle the situation of concurrency. Using shared memory, we attached and detached the memory segment. Ultimately semaphore is discussed. Semaphores facilitate the access to shared resources through synchronization mechanism.

9 Sockets

In inter-process communication, mostly the client/server model is used. The term client/server model refers to two processes, which will be communicating with each other. One process is the client that connects to the other process, the server, to make a request for information. A good analogy of the client is a person who makes a phone call to another person, that is the server.

A socket is a piece of code which we write to enable communication between two separate entities, meaning computers in this context, or within the entity itself. For example, we use a telephone and dial the telephone number of the person whom we wish to talk. In this process we create an end point of communication between two telephones. The same principle is applied in sockets.

The early Berkeley versions of UNIX have introduced a communication tool, called socket interface, which was an extension of the pipes concept (a pipe is a connection of a data flow from one process to another). Sockets can be used in much the same way as pipes, but they are generalized to include communication across a network of computers.

To make a connection with a server, the client needs to know about the existence of and address of the server. But the server does not need to know the address of (or even the existence of) the client prior to the establishment of connection. Once a connection is established, both sides can send and receive information.

The system calls for establishing a connection are somewhat different for the client and the server, but both involve the basic construct of a socket. A socket is one end of an inter-process communication channel. Two processes, that is the client and the server, each establish their own socket. Two processes can communicate with each other only if their sockets are of the same type and in the same domain.

9.1 Socket Structure

Socket is a two-way communications pipe which can be used to communicate in a wide variety of domains. One of the most common domains sockets communicate over is the UNIX domain; that is, sockets can be used between processes on the same UNIX system.

UNIX sockets are just like two-way FIFOs. The only difference is that all data communication will be taking place through the sockets interface instead of the file interface. However, UNIX sockets are special files in the file system (just like FIFOs).

When programming with sockets, you will usually create server and client programs. The server will sit listening for incoming connections from clients and handle them. This is very similar to the situation that exists with Internet sockets, but with some fine differences.

The steps involved in establishing a socket on the client side are as follows:

1. Create a socket with the `socket()` system call.
2. Connect the socket to the address of the server using the `connect()` system call.

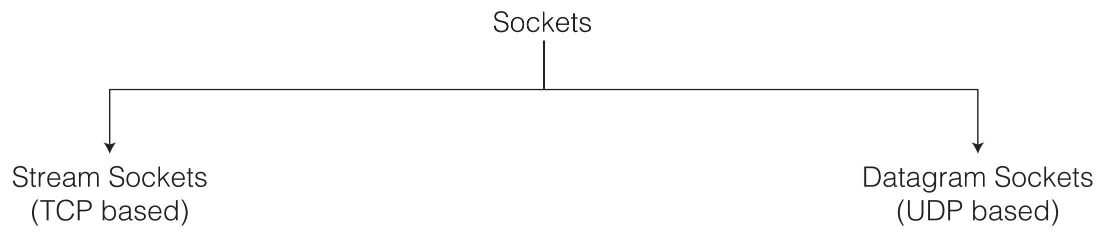


Figure 9.1 Socket types.

3. Send and receive data. There are a number of ways to do this, but the simplest is to use the `read()` and `write()` system calls.

The steps involved in establishing a socket on the server side are as follows:

1. Create a socket with the `socket()` system call.
2. Bind the socket to an address using the `bind()` system call. For a server socket on the Internet, an address consists of a port number on the host machine.
3. Listen for connections with the `listen()` system call.
4. Accept a connection with the `accept()` system call. This call typically blocks until a client connects with the server.
5. Send and receive data.

9.1.1 Types of Socket

When a socket is created, the program has to specify the address domain and the socket type. As you know, two processes can communicate with each other only if their sockets are of the same type and in the same domain. There are two widely used address domains, the UNIX domain and the Internet domain. In the UNIX domain, two processes that share a common file system communicate, while in the Internet domain, two processes running on any two hosts on the Internet communicate.

The UNIX and Internet domains have their own address formats. The address of a socket in the UNIX domain is a character string that is basically an entry in the file system. The address of a socket in the Internet domain consists of the Internet address of the host machine (every computer on the Internet has a unique 32-bit address, often referred to as its IP address). In addition, each socket needs a port number on that host. Port numbers are 16-bit unsigned integers. The lower numbers are reserved in UNIX for standard services. For example, the port number for the FTP server is 21. It is important that standard services be at the same port on all computers so that clients will know their addresses. However, port numbers above 2000 are generally available.

There are two widely used socket types – stream and datagram, as shown in Figure 9.1. Stream sockets treat communications as a continuous stream of characters, while datagram sockets have to read entire messages at once. Each socket type uses its own communications protocol. Stream sockets use transmission control protocol (TCP), which is a reliable, stream-oriented protocol, and datagram sockets use UNIX datagram protocol (UDP), which is unreliable and message-oriented.

When describing which UNIX socket you want to use (i.e. the path to the special file that is the socket), you use a `struct sockaddr_un,` which has the following fields:

```
struct sockaddr_un {
 unsigned short sun_family; /* AF_UNIX */
 char sun_path[108];
};
```

This is the structure you will be passing to the `bind()` function, which associates a socket descriptor (a file descriptor) with a certain file (the name for which is in the `sun_path` field).

9.2 `socket()` Function

The `socket()` function creates an unbound socket in a communications domain, and returns a file descriptor that can be used in later function calls that operate on sockets.

The function action, header file required and syntax of the `socket()` function are as follows:

Function action

`socket` – creates an endpoint for communication.

Header file required

```
#include <sys/socket.h>
```

Syntax

```
int socket(int domain, int type, int protocol);
```

The `socket()` function takes the following arguments:

`domain`

Specifies the communications domain in which a socket is to be created.

`type`

Specifies the type of the socket to be created.

`protocol`

Specifies a particular protocol to be used with the socket. Specifying a protocol of 0 causes `socket()` to use an unspecified default protocol appropriate for the requested socket type.

The `domain` argument specifies the address family used in the communications domain. The address families supported by the system are implementation-defined.

Symbolic constants that can be used for the domain argument are defined in the `<sys/socket.h`> header.

The `type` argument specifies the socket type, which determines the semantics of communication over the socket. The following socket types are defined; implementations may specify additional socket types:

SOCK_STREAM

Provides sequenced, reliable, bi-directional, connection-mode byte streams, and may provide a transmission mechanism for out-of-band data.

SOCK_DGRAM

Provides datagrams that are connectionless-mode, unreliable messages of fixed maximum length.

SOCK_SEQPACKET

Provides sequenced, reliable, bi-directional, connection-mode transmission paths for records. A record can be sent using one or more output operations and can be received using one or more input operations, but a single operation never transfers part of more than one record. Record boundaries are visible to the receiver via the MSG_EOR flag.

If the `protocol` argument is non-zero, it shall specify a protocol that is supported by the address family. If the `protocol` argument is zero, the default protocol for this address family and type shall be used. The protocols supported by the system are implementation-defined.

A process may need to have appropriate privileges to use the `socket()` function or to create some sockets.

Return values: Upon successful completion, `socket()` returns a non-negative integer as the socket file descriptor. Otherwise, a value of −1 is returned and `errno` is set to indicate the error.

Error: The `socket()` function fails in case of the following error:

EAFNOSUPPORT

The implementation does not support the specified address family.

9.3 `bind()` Function

The `bind()` function assigns a local socket address to a socket identified by the descriptor socket that has no local socket address assigned. Sockets created with the `socket()` function are initially unnamed; they are identified only by their address family.

The function action, header file required and syntax for the `bind()` function are as follows:

Function action

`bind` – assigns a local socket address to a socket identified.

Header file required

```
#include <sys/socket.h>
```

Syntax

```
   int bind(int socket, const struct sockaddr *address, socklen_t
address_len);
```

The `bind()` function takes the following arguments:

`socket`

Specifies the file descriptor of the socket to be bound.

`address`

Points to a `sockaddr` structure containing the address to be bound to the socket. The length and format of the address depend on the address family of the socket.

`address_len`

Specifies the length of the `sockaddr` structure pointed by the `address` argument.

The socket specified by `socket()` may require the process to have appropriate privileges to use the `bind()` function.

Return values: Upon successful completion, `bind()` shall return 0. Otherwise, −1 shall be returned and `errno` set to indicate the error.

Errors: The `bind()` function shall fail in case of the following errors:

EADDRINUSE

The specified address is already in use.

EADDRNOTAVAIL

The specified address is not available from the local machine.

EAFNOSUPPORT

The specified address is not a valid address for the address family of the specified socket.

EBADF

The `socket` argument is not a valid file descriptor.

EINVAL

The socket is already bound to an address, and the protocol does not support binding to a new address; or the socket has been shut down.

ENOTSOCK

The `socket` argument does not refer to a socket.

EOPNOTSUPP

The socket type of the specified socket does not support binding to an address.

If the address family of the socket is AF_UNIX, then `bind()` shall fail in case of the following errors:

EACCES

A component of the `path` prefix denies search permission, or the requested name requires writing in a directory with a mode that denies write permission.

EDESTADDRREQ or EISDIR

The `address` argument is a null pointer.

EIO

An I/O error occurred.

ELOOP

A loop exists in symbolic links encountered during resolution of the pathname in `address`.

ENAMETOOLONG

A component of a pathname exceeded `{NAME_MAX}` characters, or an entire pathname exceeded `{PATH_MAX}` characters.

ENOENT

A component of the pathname does not name an existing file or the pathname is an empty string.

ENOTDIR

A component of the `path` prefix of the pathname in `address` is not a directory.

EROFS

The name would reside on a read-only file system.

EACCES

The specified address is protected and the current user does not have permission to bind to it.

EINVAL

The `address_len` argument is not a valid length for the address family.

EISCONN

The socket is already connected.

ELOOP

More than `{SYMLOOP_MAX}` symbolic links were encountered during resolution of the pathname in `address`.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds `{PATH_MAX}`.

ENOBUFS

Insufficient resources were available to complete the call.

9.4 `listen()` Function

The socket in use may require the process to have appropriate privileges to use the `listen()` function. The `listen()` function marks a connection-mode socket, specified by the `socket` argument, as accepting connections.

The function action, header file required and syntax of the `listen()` function are as follows:

Function action

`listen` – marks a connection-mode socket.

Header file required

```
#include <sys/socket.h>
```

Syntax

```
int listen(int socket, int backlog);
```

The `backlog` argument provides a hint to the implementation that the implementation shall use to limit the number of outstanding connections in the socket's listen queue. Implementations may impose a limit on `backlog` and silently reduce the specified value. Normally, a larger `backlog` argument value shall result in a larger or equal length of the listen queue. Implementations shall support values of `backlog` up to SOMAXCONN, defined in `<sys/socket.h>`.

The implementation may include incomplete connections in the listen queue. The limits on the number of incomplete connections and completed connections queued may be different.

The implementation may have an upper limit on the length of the listen queue – either global or per accepting socket. If `backlog` exceeds this limit, the length of the listen queue is set to the limit.

If `listen()` is called with a `backlog` argument value that is less than 0, the function behaves as if it had been called with a `backlog` argument value of 0.

A `backlog` argument of 0 may allow the socket to accept connections, in which case the length of the listen queue may be set to an implementation-defined minimum value.

Return values: Upon successful completion, `listen()` shall return 0. Otherwise, −1 shall be returned and `errno` set to indicate the error.

Errors: The `listen()` function shall fail in case of the following errors:

EBADF

The `socket` argument is not a valid file descriptor. The socket is not bound to a local address, and the protocol does not support listening on an unbound socket.

EINVAL

The `socket` is already connected.

ENOTSOCK

The `socket` argument does not refer to a socket.

EOPNOTSUPP

The socket protocol does not support `listen()`.

EACCES

The calling process does not have appropriate privileges.

EINVAL

The `socket` has been shut down.

ENOBUFS

Insufficient resources are available in the system to complete the call.

9.5 `accept()` Function

The `accept()` function shall extract the first connection on the queue of pending connections, create a new socket with the same socket type protocol and address family as the specified socket, and allocate a new file descriptor for that socket.

The function action, header file required and syntax for the `accept()` function are as follows:

Function action

`accept` – extracts the first connection on the queue.

Header file required

```
#include <sys/socket.h>
```

Syntax

```
int accept(int socket, struct sockaddr *restrict address, sock-
len_t *restrict address_len);
```

The `accept()` function takes the following arguments:

`socket`

Specifies a socket that was created with `socket()`. The newly created socket has been bound to an address with `bind()`, and has issued a successful call to `listen()`.

`address`

Points to either a null pointer or a pointer to a `sockaddr` structure where the address of the connecting socket shall be returned.

`address_len`

Points to a `socklen_t` structure that on input specifies the length of the supplied `sockaddr` structure, and on output specifies the length of the stored address.

If `address` is not a null pointer, the address of the peer for the accepted connection shall be stored in the `sockaddr` structure pointed to by `address,` and the length of this address shall be stored in the object pointed to by `address_len.`

If the actual length of the address is greater than the length of the supplied `sockaddr` structure, the stored address shall be truncated.

If the protocol permits connections by unbound clients, and the peer is not bound, then the value stored in the object pointed to by `address` is unspecified.

If the listen queue is empty of connection requests, and O_NONBLOCK is not set on the file descriptor for the socket, `accept()` shall block until a connection is present. If the `listen()` queue is empty of connection requests and O_NONBLOCK is set on the file descriptor for the socket, `accept()` shall fail, setting `errno` to EAGAIN or EWOULDBLOCK.

The accepted socket cannot itself accept more connections. The original socket remains open and can accept more connections.

Returns values: Upon successful completion, `accept()` shall return the non-negative file descriptor of the accepted socket. Otherwise, −1 shall be returned and `errno` set to indicate the error.

Errors: The `accept()` function shall fail in case of the following errors:

EAGAIN or EWOULDBLOCK

O_NONBLOCK is set for the socket file descriptor and no connections are present to be accepted.

EBADF

The `socket` argument is not a valid file descriptor.

ECONNABORTED

A connection has been aborted.

EINTR

The `accept()` function was interrupted by a signal that was caught before a valid connection arrived.

EINVAL

The `socket` is not accepting connections.

EMFILE

{OPEN_MAX} file descriptors are currently open in the calling process.

ENFILE

The maximum number of file descriptors in the system is already open.

ENOTSOCK

The `socket` argument does not refer to a socket.

EOPNOTSUPP

The socket type of the specified socket does not support accepting connections.

ENOBUFS

No buffer space is available.

ENOMEM

There was insufficient memory available to complete the operation.

EPROTO

A protocol error has occurred; for example, the STREAMS protocol stack has not been initialized.

9.6 `connect()` Function

The `connect()` function shall attempt to make a connection on a socket.

The function action, header file required and syntax of the `connect()` function are as follows:

Function action

`connect` – makes a connection.

Header file required

```
#include <sys/socket.h>
```

Syntax

```
int connect(int socket, const struct sockaddr *address,
   socklen_t address_len);
```

The function takes the following arguments:

`socket`

Specifies the file descriptor associated with the socket.

`address`

Points to a `sockaddr` structure containing the peer address. The length and format of the address depend on the address family of the socket.

`address_len`

Specifies the length of the `sockaddr` structure pointed to by the `address` argument.

If the socket has not already been bound to a local address, `connect()` shall bind it to an address that, unless the socket's address family is AF_UNIX, is an unused local address.

If the initiating socket is not connection-mode, then `connect()` shall set the socket's peer address, and no connection is made. For SOCK_DGRAM sockets, the peer address identifies where all datagrams are sent on subsequent `send()` functions, and limits the remote sender for subsequent `recv()` functions. If `address` is a null address for the protocol, the socket's peer address shall be reset.

If the initiating socket is connection-mode, then `connect()` shall attempt to establish a connection to the address specified by the `address` argument. If the connection cannot be established immediately and O_NONBLOCK is not set for the file descriptor for the socket, `connect()` shall block for up to an unspecified time-out interval until the connection is established. If the time-out interval expires before the connection is established, `connect()` shall fail and the connection attempt shall be aborted. If `connect()` is interrupted by a signal that is caught while blocked waiting to establish a connection, `connect()` shall fail and set `errno` to EINTR, but the connection request shall not be aborted, and the connection shall be established asynchronously.

If the connection cannot be established immediately and O_NONBLOCK is set for the file descriptor for the socket, `connect()` shall fail and set `errno` to EINPROGRESS; but the connection request shall not be aborted, and the connection shall be established asynchronously. Subsequent calls to `connect()` for the same socket, before the connection is established, shall fail and set `errno` to EALREADY.

Return values: Upon successful completion, `connect()` shall return 0. Otherwise, −1 shall be returned and `errno` set to indicate the error.

Errors: The `connect()` function shall fail in case of the following errors:

EADDRNOTAVAIL

The specified address is not available from the local machine.

EAFNOSUPPORT

The specified address is not a valid address for the address family of the specified socket.

EALREADY

A connection request is already in progress for the specified socket.

EBADF

The `socket` argument is not a valid file descriptor.

ECONNREFUSED

The target address was not listening for connections or refused the connection request.

EINPROGRESS

O_NONBLOCK is set for the file descriptor for the socket and the connection cannot be immediately established; the connection shall be established asynchronously.

EINTR

The attempt to establish a connection was interrupted by delivery of a signal that was caught; the connection shall be established asynchronously.

EISCONN

The specified socket is connection-mode and is already connected.

ENETUNREACH

No route to the network is present.

ENOTSOCK

The `socket` argument does not refer to a socket.

EPROTOTYPE

The specified address has a different type than the socket bound to the specified peer address.

ETIMEDOUT

The attempt to connect timed out before a connection was made.

If the address family of the socket is AF_UNIX, then `connect()` shall fail in case of the following errors:

EIO

An I/O error occurred while reading from or writing to the file system.

ELOOP

A loop exists in symbolic links encountered during resolution of the pathname in `address`.

ENAMETOOLONG

A component of a pathname exceeded {NAME_MAX} characters, or an entire pathname exceeded {PATH_MAX} characters.

ENOENT

A component of the pathname does not name an existing file or the pathname is an empty string.

ENOTDIR

A component of the `path` prefix of the pathname in `address` is not a directory.

The `connect()` function may also fail in case of the following errors:

EACCES

Search permission is denied for a component of the `path` prefix; or write access to the named socket is denied.

EADDRINUSE

Attempt to establish a connection that uses addresses that are already in use.

ECONNRESET

Remote host resets the connection request.

EHOSTUNREACH

The destination host cannot be reached (probably because the host is down or a remote router cannot reach it).

EINVAL

The `address_len` argument is not a valid length for the address family; or invalid address family in the `sockaddr` structure.

ELOOP

More than {SYMLOOP_MAX} symbolic links were encountered during resolution of the pathname in `address`.

ENAMETOOLONG

Pathname resolution of a symbolic link produced an intermediate result whose length exceeds {PATH_MAX}.

ENETDOWN

The local network interface used to reach the destination is down.

ENOBUFS

No buffer space is available.

EOPNOTSUPP

The socket is listening and cannot be connected.

9.7 Local Server Example

Now we will work with a sample program of the client/server communication using sockets. We will develop the local server that will communicate through the client.

Prog. 1

This program demonstrates the implementation of the local server that runs in the background.

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCK_PATH "echo_socket"

int main(void)
```

```
{
  int s, s2, t, len;
  struct sockaddr_un local, remote;
  char str[100];

  if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == −1) {
    perror("socket");
    exit(1);
  }

  local.sun_family = AF_UNIX;
  strcpy(local.sun_path, SOCK_PATH);
  unlink(local.sun_path);
  len = strlen(local.sun_path) + sizeof(local.sun_family);
  if (bind(s, (struct sockaddr *)&local, len) == −1) {
    perror("bind");
    exit(1);
  }

  if (listen(s, 5) == −1) {
    perror("listen");
    exit(1);
  }

  for(;;) {
    int done, n;
    printf("Waiting for a connection...\n");
    t = sizeof(remote);
    if ((s2 = accept(s, (struct sockaddr *)&remote, &t)) == −1) {
      perror("accept");
      exit(1);
    }

    printf("Connected.\n");

    done = 0;
    do {
      n = recv(s2, str, 100, 0);
      if (n <= 0) {
        if (n < 0) perror("recv");
        done = 1;
      }

      if (!done)
        if (send(s2, str, n, 0) < 0) {
          perror("send");
          done = 1;
```

```
            }
        } while (!done);

        close(s2);
    }

    return 0;
}
```

In Prog. 1, the following steps have been undertaken:

1. **Call `socket()`:** A call to `socket()` with the proper arguments creates the UNIX socket:
   - `unsigned int s, s2;`
   - `struct sockaddr_un local, remote;`
   - `int len;`
   - initialize the structure.
   - `s = socket(AF_UNIX, SOCK_STREAM, 0);`

   The second argument, `SOCK_STREAM`, tells `socket()` to create a stream socket.

   Important note: All these calls return −1 on error and set the global variable `errno` to reflect whatever went wrong. Be sure to do your error checking.

2. **Call `bind()`:** You got a socket descriptor from the call to `socket()`, now you want to bind that to an address in the UNIX domain. (That address is a special file on disk.)
   - `local.sun_family = AF_UNIX; /*local is declared before socket() ^ */`
   - `local.sun_path = "/home/kumar/mysocket";`
   - `unlink(local.sun_path);`
   - `len = strlen(local.sun_path) + sizeof(local.sun_family);`
   - `bind(s, (struct sockaddr *)&local, len);`

3. **Call `listen()`:** This instructs the socket to listen for incoming connections from client programs:
   - `listen(s, 5);`

   The second argument, 5, is the number of incoming connections that can be queued before you call `accept()` described below. If there are many connections waiting to be accepted, additional clients will generate error `ECONNREFUSED`.

4. **Call `accept()`:** This will accept a connection from a client. This function returns another socket descriptor. The old descriptor is still listening for new connections, but this new one is connected to the client:
   - `len = sizeof(struct sockaddr_un);`
   - `s2 = accept(s, &remote, &len);`

5. **Handle the connection and loop back to `accept()`:** Usually you will want to communicate to the client here (we will just echo back everything it sends us), close the connection, then `accept()` a new one.
   - `while (len = recv(s2, &buf, 100, 0), len > 0)`
   - `send(s2, &buf, len, 0);`
   - `/* loop back to accept() from here */`

6. **Close the connection.**
   All it does is wait for a connection on a UNIX socket (named, in this case, 'echo_socket').

There needs to be a program, that is a client, to talk to the above server. Here is the program:

Prog. 2

This program demonstrates the implementation of the client section of Prog. 1. This implementation takes place when the server code is running in the background.

Client code

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>

#define SOCK_PATH "echo_socket"

int main(void)
{
  int s, t, len;
  struct sockaddr_un remote;
  char str[100];

  if ((s = socket(AF_UNIX, SOCK_STREAM, 0)) == −1) {
    perror("socket");
    exit(1);
  }

  printf("Trying to connect...\n");

  remote.sun_family = AF_UNIX;
  strcpy(remote.sun_path, SOCK_PATH);
  len = strlen(remote.sun_path) + sizeof(remote.sun_family);
  if (connect(s, (struct sockaddr *)&remote, len) == −1) {
        perror("connect");
        exit(1);
    }

    printf("Connected.\n");

    while(printf(">"), fgets(str, 100, stdin), !feof(stdin)){
        if (send(s, str, strlen(str), 0) == -1) {
            perror("send");
            exit(1);
        }

        if ((t=recv(s, str, 100, 0)) > 0) {
            str[t] = '\0';
            printf("echo> %s", str);
        } else {
            if (t < 0) perror("recv");
            else printf("Server closed connection\n");
            exit(1);
        }
    }

    close(s);

    return 0;
}
```

In Prog. 2, the following steps have been undertaken:

1. Call `socket()` to get a UNIX domain socket to communicate through.
2. Set up a `struct sockaddr_un` with the remote address (where the server is listening), and call `connect()` with that as an argument.

9.8 Summary

This chapter describes use of sockets in different ways of inter-process communication. It discusses various functions required to implement IPC using sockets. It also discusses the feasibility of developing client/server architecture across networks. We also developed a client/server program through two program examples that shows how to handle multiple clients.

10

Introduction to Device Drivers

The kernel is the heart of the UNIX operating system, managing all system threads, processes, resources and resource allocation. Unlike most other operating systems, UNIX enables users to reconfigure the kernel, which is usually done to reduce its size and add or deactivate support for specific devices or subsystems. Reconfiguring the kernel to remove support for unused devices and subsystems is quite common when developing embedded systems because a smaller kernel requires less memory, increasing the resources available to your applications.

The kernel manages the system resources, including file systems, processes and physical devices. The kernel provides applications with system services such as I/O management, virtual memory and scheduling. The kernel coordinates interactions of all user processes and system resources. The kernel assigns priorities, services resource requests and services hardware interrupts and exceptions. The kernel schedules and switches threads, pages memory and swaps processes.

Kernel-device driver relationship

Device drivers are the software interface between your hardware and the UNIX kernel. Device drivers are low-level, hardware-specific software components that enable devices to interact with more generic, high-level application programming interfaces (APIs). Providing support for a specific subsystem or hardware interface, such as SCSI, USB or PCMCIA, is quite different than providing support for every SCSI, USB or PCMCIA device. Testing every possible device that could be used over a specific subsystem is not possible because new devices are being made available every day. The kernel provides support for specific subsystems, and device drivers provide support for specific devices that use those subsystems. Maintaining the separation of high-level APIs and low-level device functionality makes the kernel relatively fast. It also facilitates adding support for new devices to existing systems by writing the appropriate driver for a new device and making it available to the kernel.

Device drivers can be integrated into the kernel in two ways – either by compiling them into the kernel so that they are always available, or by compiling them into an object format that the kernel can load whenever access to a specific device is required. The kernel code that can be automatically loaded into the kernel is referred to as a loadable kernel module. When configuring the kernel, each kernel configuration editor displays a description of available kernel configuration variables and enables you to specify whether each should be deactivated, compiled into the kernel or compiled as a loadable kernel module.

Compiling device drivers into the kernel has the advantage that they are always instantly available, but each device driver increases the size of the kernel that you are running. Compiling device drivers as loadable kernel modules implies some slight overhead when you search for and initially load the module to access the associated device, plus some small runtime overhead. But these are negligible compared to the savings in size and associated memory requirements. Writing device drivers as loadable kernel modules also provides significant advantages during development. As you develop and

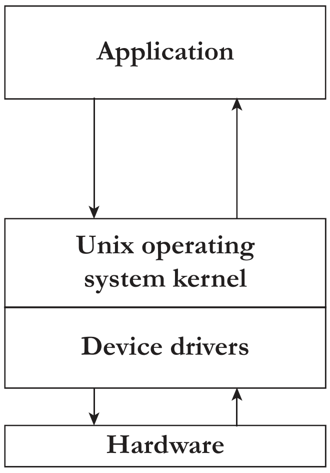


Figure 10.1 UNIX device drivers.

debug your device driver, you can dynamically unload the previous version and load the new version each time you want to test the new version. If your device driver is compiled into the kernel, you have to recompile the kernel and reboot each time you want to test a set of iterative changes. Similarly, developing and deploying device drivers as loadable kernel modules simplifies maintaining them in the field because a device driver can be updated as a separate system component without requiring a kernel update.

An important feature of the UNIX device driver is that its routines are linked into the kernel and form part of the operating system. This is depicted in Figure 10.1.

The UNIX device driver is a special file. If we were to examine the `/dev` files using `ls -l` command, we would find: `crw------ 2 bin 6 0 Oct 3 1999 /dev/drv1`

Notice that in the above example code, the first character on the line is `c`. This indicates that this special file represents a character device driver. Also note that the numbers `3` and `0` appear in the place where we would normally expect to find the file size. These are the major (3) and minor (0) device numbers for this special file. The major number specifies the device driver, while the minor device number is used by the device driver to distinguish between different devices under the control of a single driver.

How does the UNIX kernel tell the device driver what it wants it to do? These requests are performed in the two steps described below:

1. The operating system calls an entry point of the device driver. This causes the control to pass to the device driver.
2. The device driver examines the parameters passed and the kernel data structure used for information to determine exactly what to do.

Each device driver has its own:

1. Set of entry points that the Unix operating system expects to find.
2. Conversions for the exchange of data and commands.

A device driver may have any or all of the following basic entry points:

1. `init()`: The `init()` entry point is called by the kernel immediately after the system is booted. It provides the device driver with an opportunity to initialize the device driver and hardware. It is important not to use system services, such as sleep delay or wakeup. Do not attempt to reference any members of the user structure. Also, keep in mind that interrupts are not available during `init` time.
2. `open()`: The `open()` entry point is called whenever a user process performs an `open` system c all on a special file that is related to the device driver.
3. `close()`: The `close()` entry point is called by the kernel when the last user process, that has the device driver open, performs a close s ystem call.
4. `read()`: The `read()` entry point is called whenever a user process performs a read system call on a special file that is related to the device driver.
5. `write()`: The `write()` entry point is called whenever a user process performs a write system call on a special file that is related to the device driver.
6. `halt()`: The `halt()` entry point is called just before the system is shut down. It provides an opportunity for the device driver to flush data that may still be resident in the device driver.
7. `intr()`: The `intr()` entry point is called whenever an interrupt is received from the hardware. Interrupts are signals from the hardware that indicate a significant event has occurred, which requires the attention of the driver.
8. `ioctl()`: The `ioctl()` entry point is called whenever a user process performs an `ioctl` system call on a special file that is related tox the device driver. These calls are used to pass special requests to the device driver or to obtain information on the configuration or status of the device and device driver.

10.1 Execution Differences between Kernel Modules and User Programs

The following characteristics of kernel modules highlight important differences between the execution of kernel modules and user programs:

1. **Kernel modules have separate address space:** A kernel module runs in kernel space. An application runs in user space. The system software is protected from user programs. Kernel space and user space have their own memory address spaces.
2. **Kernel modules have higher execution privilege:** The code that runs in the kernel space has greater privilege than the code that runs in the user space. Driver modules potentially have a much greater impact on the system than the user programs. Test and debug your driver modules carefully and thoroughly to avoid any adverse impact on your system.
3. **Kernel modules do not execute sequentially:** A user program executes in a typical sequential way and performs a single task from beginning to end. However, a kernel module does not execute sequentially. A kernel module registers itself to serve future requests.
4. **Kernel modules can be interrupted:** More than one process can request your driver at the same time. An interrupt handler can request your driver at a time when your driver is serving a system call. In a symmetric multi-processor (SMP) system, your driver could be executing concurrently on more than one CPU.
5. **Kernel modules must be pre-emptable:** You cannot assume that your driver code is safe just because your driver code does not block. Design your driver assuming your driver might be pre-empted.

6. **Kernel modules can share data:** Different threads of an application program usually do not share data. By contrast, the data structures and routines that constitute a driver are shared by all threads that use the driver. Your driver must be able to handle contention issues that result from multiple requests. Design your driver data structures carefully to keep multiple threads of execution separate. Driver code must access shared data without corrupting the data.

10.2 Structural Differences between Kernel Modules and User Programs

The following characteristics of kernel modules highlight important differences between the structures of kernel modules and user programs:

1. **Kernel modules do not define a main program:** Kernel modules, including device drivers, have no `main()` routine. Instead, a kernel module is a collection of subroutines and data. A device driver is a kernel module that forms a software interface to an I/O device. The subroutines in a device driver provide entry points to the device. The kernel uses a device number attribute to locate the `open()` routine and other routines of the correct device driver.
2. **Kernel modules are linked only to the kernel:** Kernel modules do not link in the same libraries that user programs link in. The only functions a kernel module can call are the functions that are exported by the kernel.
3. **Kernel modules should avoid global variables:** Avoiding global variables in kernel modules is even more important than avoiding global variables in user programs. As far as possible, declare symbols as static. When you must use global symbols, give them a prefix that is unique within the kernel. Using this prefix for private symbols within the module is also a good practice.
4. **Kernel modules can be customized for hardware:** Kernel modules can dedicate process registers to specific roles. Kernel code can be optimized for a specific processor.
5. **Kernel modules can be dynamically loaded:** The collection of subroutines and data that constitute a device driver can be compiled into a single loadable module of object code. This loadable module can then be statically or dynamically linked into and unlinked from the kernel. You can add functionality to the kernel while the system is up and running. You can test new versions of your driver without rebooting your system.

10.3 Data Transfer Differences between Kernel Modules and User Programs

Data transfer between a device and the system typically is slower than data transfer within the CPU. Therefore, a driver typically suspends execution of the calling thread until the data transfer is complete. While the thread that called the driver is suspended, the CPU is free to execute other threads. When the data transfer is complete, the device sends an interrupt. The driver handles the interrupt that it receives from the device. The driver then tells the CPU to resume execution of the calling thread. Drivers must work with user process (virtual) addresses, system (kernel) addresses and I/O bus addresses. Drivers sometimes copy data from one address space to another address space, and sometimes just manipulate address-mapping tables.

10.4 Character Devices vs. Block Devices

Devices are divided into two types: character devices and block devices. The difference is that block devices have a buffer for requests, so they can choose by which order to respond to them. This is important in the case of storage devices where it is faster to read or write those sectors that are close to each other rather than those that are further apart.

Another difference is that block devices can only accept input and return output in blocks (whose size can vary according to the device), whereas character devices are allowed to use as many or as few bytes as they like. Most devices in the world are character-type because they do not need this type of buffering, and they do not operate with a fixed block size. You can tell whether a device file is for a block device or a character device by looking at the first character in the output of `ls -l.` If it's 'b' then it's a block device, and if it's 'c' then it's a character device.

In general, we can say a block device is something (e.g. a disk) that can host a file system. A block device can only be accessed as multiples of a block, and a block is usually 1KB of data. A character device is one that can be accessed like a file, and a character driver is in charge of implementing this behaviour. This driver implements the `open, close, read` and `write` system calls. The console and parallel ports are examples of character devices.

The main difference between character-mode and block-mode devices in the UNIX kernel lies in the way they make requests for data transfer. As you can see, character devices have read and write functions that are called directly whenever I/O is required. Block devices generally don't have read or write functions. Instead, they implement a 'strategy routine' or 'request function', which is called not directly by system calls, but indirectly by the buffer cache.

If a program requests data from a file, the particular file system that holds that data determines what block the data is on and requests that block from the buffer cache. That block might be cached, in which case the request is satisfied by the buffer cache. If the required block is not cached, the buffer cache creates a request for the device driver to fetch the block from disk and store the data in a buffer.

Of course, while finding the required data block, the file system might have to read other blocks containing directory entries and i-nodes. When it needs to read those blocks, it requests them from the buffer cache in the same way it requests any other data block. The buffer cache does not need to know what the blocks are used for.

This indirect approach speeds up disk access considerably, and ends up simplifying some things. The block device driver has very little interaction with user programs; only the calls to `ioctl()` are likely to be the most common direct interaction. This means that block device drivers don't have to be as suspicious of their input as character device drivers. By the time a request for a data block has made it to the strategy routine, it has been pretty well checked to make sure it is valid. There is no question of writing to user-space memory, and there is no chance that a buggy user-level program passed in bad arguments that need to be checked.

10.5 User Space vs. Kernel Space

A kernel is all about access to resources, whether the resource in question happens to be a video card, a hard drive or even memory. Programs often compete for the same resource. The kernel needs to keep things orderly, and not give users access to resources whenever they feel like it. To this end, a CPU can run in different modes. Each mode gives a different level of freedom to do what you want to do on the system. UNIX uses only two rings – the highest ring (ring 0, also known as 'supervisor mode' where everything is allowed to happen) and the lowest ring (which is called 'user mode').

10.6 Name Space

When you write a small C program, you use variables which are convenient and make sense to the reader. If, on the other hand, you are writing routines, which will be part of a bigger problem; some of the variable names can clash. For example, any global variables you have are part of a community of other peoples' global variables. When a program has lots of global variables, which are not meaningful enough to be distinguished, you get name space pollution. In large projects, effort must be made to remember reserved names, and to find ways to develop a scheme for using unique variable names and symbols.

When writing kernel code, even the smallest module will be linked against the entire kernel, so this is definitely an issue. The best way to deal with this is to declare all your variables as static and to use a well-defined prefix for your symbols. By convention, all kernel prefixes are lowercase. If you don't want to declare everything as static, another option is to declare a `symbol table` and register it with a kernel. The file `/proc/ksyms` holds all the symbols that the kernel knows about and which are therefore accessible to your modules because they share the kernel's codespace.

10.7 Inserting a User Module in Kernel

A kernel module has to have at least two functions: `init_module`, which is called when the module is inserted into the kernel, and `cleanup_module`, which is called just before it is removed. Typically, `init_module` either registers a handler for something with the kernel, or it replaces one of the kernel functions with its own code (usually code to do something and then call the original function). The `cleanup_module` function is supposed to undo whatever `init_module` did, so the module can be unloaded safely.

The following sample code deals with the insertion and working of the user module at the kernel level. Once the user module is inserted, it becomes the kernel module. First we have defined the framework and then the sample code is given.

Kernel module framework

```
# define MODULE
# include <linux/module.h>
# include <linux/config.h>
# include <linux/init.h>
static int __init name_of_initialization_routine(void) {
  / *
    * code here      */
}
static void __exit name_of_cleanup_routine(void) {
  / *
    * code here
    */
}
module_init(name_of_initialization_routine);
module_exit(name_of_cleanup_routine);
```

Sample kernel module code

```
# define MODULE
# include <linux/module.h>
int init_module (void) /* Loads a module in the kernel */
{
printk("Hi kernel n");
return 0;
}
void cleanup_module(void) /* Removes module from kernel */
{
printk("Bye,Kernel\n");
}
```

10.8 Makefiles for Kernel Modules

A kernel module is not an independent executable, but an object file which will be linked into the kernel in runtime. As a result, kernel modules should be compiled with the `-c` flag. Also, all kernel modules have to be compiled with certain symbols defined.

Now we will discuss the compile parameters of kernel modules one by one:

KERNEL__ – This tells the header files that this code will be run in the kernel mode, not as part of a user process.

MODULE – This tells the header files to give appropriate definitions for a kernel module.

LINUX – Technically speaking, this is not necessary. However, if you ever want to write a serious kernel module, which will compile on more than one operating system, you will be happy you did. This will allow you to do conditional compilation on the parts that are OS dependent.

There are other symbols which have to be included, or not, depending on the flags the kernel was compiled with. If you are not sure how the kernel was compiled, look it up in `/usr/include/linux/config.h`

__SMP__ – Symmetrical multi-processing. This has to be defined if the kernel was compiled to support SMP (even if it is running just on one CPU). If you use SMP there are other things you need to do.
CONFIG_MODVERSIONS – If CONFIG_MODVERSIONS was enabled, you need to have it defined when compiling the kernel module and to include `/usr/include/linux/modversions.h.` This can also be done by the code itself.

10.9 File Operations

If you know about file operations, it should not be surprising that devices 'export' their functionality to the VFS by registering a `file_operations` structure with the VFS. We will see exactly how this is done later. Here is the `file_operations` structure:

```
struct file_operations {
      int (*lseek) (struct inode *, struct file *, off_t, int);
      (*read) (struct inode *, struct file *, char *, int);
      (*write) (struct inode *, struct file *, char *, int);
      int (*readdir) (struct inode *, struct file *, struct
           dirent *, int);
      int (*select) (struct inode *, struct file *, int,
           select_table *);
      int (*ioctl) (struct inode *, struct file *, unsigned
           int, unsigned long);
      int (*mmap) (struct inode *, struct file *, struct vm_
           area_struct *);
      int (*open) (struct inode *, struct file *);
      void (*release) (struct inode *, struct file *);
      int (*fsync) (struct inode *, struct file *);
      int (*fasync) (struct inode *, struct file *, int);
      int (*check_media_change) (dev_t dev);
      int (*revalidate) (dev_t dev);
};
```

Some of the names of these function pointers should look suspiciously like system calls with which you are familiar. `lseek()`, `read()`, `write()`, `readdir()`, `select()`, `ioctl()`, `mmap()`, `open()` and `fsync()` all are called directly or indirectly by the system calls of the same name. `release()` is called on `close()` and when a file is closed by a process exiting or calling `exec()` (when `close-on-exec`) is set on the file. `check_media_change()` and `revali-date()` are not really file operations, they are device operations, as can be seen by their arguments. `fasync()` is a bit unusual; it is called when `fcntl(fd, F_SETFL, FASYNC) (or ~FASYNC)` is called; devices implementing this need to be aware when this change is made. The functions are provided with sensible defaults; most of the time more than half of the functions are set to NULL because the VFS does the right thing without having to call the driver.

10.10 Kernel Module Registrations

Adding a driver to your system means registering it with the kernel. This is synonymous with assigning it a major number during the module's initialization.

Kernel module is divided into two separate parts: the module part which registers the device, and the device driver part. The `init_module` function calls `module_register_chrdev` to add the device driver to the kernel's character device driver table. It also returns the major number to be used for the driver. The `cleanup_module` function de-registers the device.

This (registering something and unregistering it) is the general specification of those two functions. Things in the kernel don't run on their own initiative, like processes, but are called by processes via system calls, by hardware devices via interrupts or by other parts of the kernel (simply by calling specific functions). As a result, when you add code to the kernel, you're supposed to register it as the handler for a certain type of event; and when you remove it, you're supposed to unregister it.

The device driver proper is composed of the four device_<action> functions, which are called when somebody tries to do something with a device file that has our major number. The way the kernel knows to call them is via the `file_operations` structure, `Fops`, which was given when the device was registered, includes pointers to those four functions.

Another point we need to remember here is that we can't allow the kernel module to be rmmoded whenever root feels like it. The reason is that if the device file is opened by a process and then we remove the kernel module, using the device file would cause a call to the memory location where the appropriate function (read/write) used to be. If we are lucky, no other code was loaded there, and we will get an ugly error message. If we're unlucky, another kernel module was loaded into the same location, which means a jump into the middle of another function within the kernel. The results of this would be impossible to predict, but they can't be positive.

Normally, when you don't want to allow something, you return an error code (a negative number) from the function, which is supposed to do it. With `cleanup_module` that is impossible because it is a void function. Once `cleanup_module` is called, the module is dead. However, there is a use counter which counts how many other kernel modules are using this kernel module, called the reference count (that's the last number of the line in `/proc/modules)`. If this number isn't zero, rmmod will fail. The module's reference count is available in the variable `mod_use_count_.` Since there are macros defined for handling this variable (MOD_INC_USE_COUNT and MOD_DEC_USE_COUNT), we prefer to use them, rather than `mod_use_count_` directly, so we will be safe if the implementation changes in the future.

Following is the function to insert the character device driver in the kernel space:

```
int register_chrdev(unsigned int major, const char *name,
    struct file_operations *fops);
```

where `unsigned int major` is the major number you want to request, `const char *name` the name of the device as it will appear in `/proc/devices` and `struct file_operations *fops` a pointer to the `file_operations` table for your driver. A negative return value means the registration failed. Note that we didn't pass the minor number to `register_chrdev`. That is because the kernel doesn't care about the minor number; only our driver uses it.

10.11 Unregistering a Device

If the device file is opened by a process and then we remove the kernel module, using the file would cause a call to the memory location where the appropriate function (read/write) used to be.

10.12 Summary

This chapter introduces the concept of device drivers. It distinguishes character drivers from block drivers. It also clarifies basic differences between kernel space and user space. It discusses writing the sample kernel module and lists all the steps needed to insert the module in the kernel space.

Appendix 1

Time Functions

In this section, we look at the ways to access the clock time with UNIX system calls. Accessing the clock is done through `times()` function. This function is basically used for

1. telling the time.
2. timing programs and functions.
3. setting number seeds.

`times()` ***function***

The `times()` function fills the `tms` structure pointed by `buffer` with time-accounting information. The `tms` structure, defined in `<sys/times.h>`, contains the following members:

```
clock_t  tms_utime;
clock_t  tms_stime;
clock_t  tms_cutime;
clock_t  tms_cstime;
```

All times are reported in clock ticks. The specific value for a clock tick is defined by variable `CLK_TCK` found in the header `<limits.h>`.
Function action, header file required and syntax of `times()` function are as follows:

Function action

`times` – get process and child process times.

Header file required

```
# include <sys/times.h>
# include <limits.h>
```

Syntax

```
clock_t times(struct tms *buffer);
```

The times of a terminated child process are included in the `tms_cutime` and `tms_cstime` members of the parent when `wait(2)` or `waitpid(2)` returns the process ID of this terminated child process. If a child process has not waited for its children, their times will not be included in its times.

The `tms_utime` member is the CPU time used while executing instructions in the user space of the calling process. The `tms_stime` member is the CPU time used by the system on behalf of the calling process. The `tms_cutime` member is the sum of the `tms_utime` and the `tms_cutime` of the child processes. The `tms_cstime` member is the sum of the `tms_stime` and the `tms_cstime` of the child processes.

Return values: Upon successful completion, `times()` returns the elapsed real time, in clock ticks since an arbitrary point in the past (e.g. system start-up time). This point does not change from one invocation of `times()` within the process to another. The return value may overflow the possible range of type clock_t. If `times()` fails, `(clock_t)-1` is returned and `errno` is set to indicate the error.

Errors: The `times()` function will fail in case of the following error:

`EFAULT`
The `buffer` argument points to an illegal address.

Some basic time functions prototypes are as follows:
`time_t time(time_t *tloc)` – returns the time since 00:00:00 GMT, Jan. 1, 1970, measured in seconds.

If `tloc` is not NULL, the return value is also stored in the location to which tloc points.

`time()` returns the value of time on success.

On failure, it returns `(time_t) −1. time_t` is typedefined to a long (int) in `<sys/types.h>` and `<sys/time.h>` header files.

`int ftime(struct timeb *tp)` – fills in a structure pointed by `tp` as defined in `<sys/timeb.h>`:

struct timeb
```
        {
             time_t time;
             unsigned short millitm;
             short timezone;
             short dstflag;
        };
```

The structure given above contains the time since the epoch in seconds, up to 1000 milliseconds of more precise interval, the local time zone (measured in minutes of time westward from Greenwich), and a flag that, if non-zero, indicates that day light saving time applies locally during the appropriate part of the year.

On success, `ftime()` returns no useful value. On failure, it returns −1.

Two other functions defined in #include <time.h>

```
char *ctime(time_t *clock),
char *asctime(struct tm *tm)
```

`ctime()` converts a long integer, pointed by clock, to a 26-character string of the form produced by `asctime()`. It first breaks down clock to a `tm` structure by calling `localtime()`, and then calls `asctime()` to convert that tm structure to a string.

`asctime()` converts a time value contained in a `tm` structure to a 26-character string of the form:

```
Sun Sep 16 01:03:52 1973
```
`asctime()` returns a pointer to the string.

Now we will discuss the program that computes time (in seconds) to perform some action.

Prog. 1

This is a simple program that illustrates that calling the time function at distinct moments and noting the different times is a simple method of timing fragments of code.

```
/* timer.c */

#include <stdio.h>
#include <sys/types.h>
#include <time.h>

main()
 { int i;
   time_t t1, t2;

   (void) time(&t1);
   for (i=1; i<=300; ++i)
   printf("%d %d %d n", i, i*i, i*i*i);
   (void) time(&t2);
   printf("n Time to do 300 squares and
   cubes= %d seconds n", (int) t22t1);
}
```

Appendix 2

Threads

Technically, a thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system. To the software developer, the concept of a ‘procedure’ that runs independently from its main program may best describe a thread. To go one step further, imagine a main program (a.out) that contains a number of procedures. Then imagine all of these procedures being able to be scheduled to run simultaneously and/or independently by the operating system. That would describe a ‘multi-threaded’ program.

Before understanding a thread, one needs to understand a UNIX process. A process is created by the operating system, and requires a fair amount of ‘overhead’. Processes contain information about program resources and program execution state, including:

1. Process ID, process group ID, user ID and group ID.
2. Environment.
3. Working directory.
4. Program instructions.
5. Registers.
6. Stack.
7. Heap.
8. File descriptors.
9. Signal actions.
10. Shared libraries.
11. Inter-process communication tools (such as message queues, pipes, semaphores or shared memory).

Threads use and exist within these process resources, yet are able to be scheduled by the operating system and run as independent entities largely because they duplicate only the bare essential resources that enable them to exist as executable code. This independent flow of control is accomplished because a thread maintains its own:

1. Stack pointer.
2. Registers.
3. Scheduling properties (such as policy or priority).
4. Set of pending and blocked signals.
5. Thread-specific data.

So, in sum, in the UNIX environment a thread exists within a process and uses the process resources. It has its own independent flow of control as long as its parent process exists, and the OS supports it. Thread duplicates only the essential resources it needs to be independently schedulable. It may share the process resources with other threads that act equally independently (and dependently). It is ‘lightweight’ because most of the overhead has already been accomplished through the creation of its process.

Threads vs. Processes

If implemented correctly, threads have some advantages of (multi) processes. They take

1. less time to create a new thread than a process because the newly created thread uses the current process address space.
2. less time to terminate a thread than a process.
3. less time to switch between two threads within the same process, partly because the newly created thread uses the current process address space.
4. less communication overheads. Communicating between the threads of one process is simple because the threads share everything, particularly the address space. So, data produced by one thread is immediately available to all the other threads.

Now we will see some thread creation and termination based system functions.

`pthread_create()` function

Initially, your `main()` program comprises a single, default thread. All other threads must be explicitly created by the programmer. `pthread_create()` function creates a new thread and makes it executable. Typically, threads are first created from within `main()` inside a single process. Once created, threads are peers, and may create other threads.

Function action, header file required and syntax for `pthread_create()` function is as follows:

Function action

`pthread_create` – thread creation.

Header file required

```
#include <pthread.h>
```

Syntax

```
int  pthread_create(pthread_t  *thread,  const  pthread_attr_t
*attr,
   void *(*start_routine)(void*), void *arg);
```

1. Arguments:
2. `thread` – Returns the thread id. (unsigned long int defined in `bits/pthreadtypes.h`)
3. `attr` – Set to NULL if default thread attributes are used. (else define members of the `struct pthread_attr_t` defined in `bits/pthreadtypes.h`) Attributes include:
   - detached state (joinable? default: PTHREAD_CREATE_JOINABLE. Other option: PTHREAD_CREATE_DETACHED).
   - scheduling policy (real-time? PTHREAD_INHERIT_SCHED, PTHREAD_EXPLICIT_SCHED, SCHED_OTHER).
   - scheduling parameter.
   - inheritsched attribute (default: PTHREAD_EXPLICIT_SCHED Inherit from parent thread: PTHREAD_INHERIT_SCHED).

- scope (kernel threads: PTHREAD_SCOPE_SYSTEM User threads: PTHREAD_SCOPE_PROCESS Pick one or the other not both.)
- guard size.
- stack address (see `unistd.h` and `bits/posix_opt.h` _POSIX_THREAD_ATTR_STACKADDR).
- stack size (default minimum PTHREAD_STACK_SIZE set in `pthread.h`).

  (a) `void * (*start_routine)` – Pointer to the function to be threaded. Function has a single argument: pointer to void.

  (b) `*arg` – pointer to argument of function. To pass multiple arguments, send a pointer to a structure.

Return value: If successful, the `pthread_create()` function returns zero. Otherwise, an error number is returned to indicate the error.

Errors: The `pthread_create()` function will fail in case of the following errors:

EAGAIN
    The system lacked the necessary resources to create another thread, or the system-imposed limit on the total number of threads in a process PTHREAD_THREADS_MAX would be exceeded.
EINVAL
    The value specified by `attr` is invalid.
EPERM
    The caller does not have appropriate permission to set the required scheduling parameters or scheduling policy.

The `pthread_create()` function will not return an error code of EINTR.

`pthread_exit()`

`pthread_exit()` function is used to explicitly exit a thread. Typically, the `pthread_exit()` routine is called after a thread has completed its work and is no longer required to exist.

Function action, header file required and syntax for `pthread_exit()` function is as follows:

Function action

`pthread_exit` – thread termination.

Header file required

```
#include <pthread.h>
```

Syntax

```
void pthread_exit(void *value_ptr);
```

If `main()` function finishes before the threads it has created, and exits with `pthread_exit()`, the other threads will continue to execute. Otherwise, they will be automatically terminated when `main()`

function finishes. The programmer may optionally specify a termination `status` that is stored as a void pointer for any thread that may join the calling thread.

Return value: The `pthread_exit()` function cannot return to its caller.

Errors: No errors are defined.

The `pthread_exit()` function will not return an error code of EINTR.

pthread_join

The `pthread_join()` function blocks the calling thread until the specified thread terminates. The specified thread must be in the current process and must not be detached. When status is not NULL, it points to a location that is set to the exit status of the terminated thread when `pthread_join()` function returns successfully. Multiple threads cannot wait for the same thread to terminate. If they try to, one thread returns successfully and the others fail with an error of ESRCH. After `pthread_join()` function returns, any stack storage associated with the thread can be reclaimed by the application.

Function action, header file required and syntax for `pthread_join()` function are as follows:

Function action

`pthread_join -` wait for thread termination

Header file required

```
#include <pthread.h>
```

Syntax

```
int pthread_join(pthread_t thread, void **value_ptr);
```

The `pthread_join()` routine takes two arguments, giving you some flexibility in its use. When you want the caller to wait until a specific thread terminates, supply that thread's ID as the first argument. If you are interested in the exit code of the defunct thread, supply the address of an area to receive it. Remember that `pthread_join()` works only for target threads that are non-detached. When there is no reason to synchronize with the termination of a particular thread, then that thread should be detached. Think of a detached thread as being the thread you use in most instances and reserve non-detached threads for only those situations that require them.

Return value: If successful, the `pthread_join()` function returns zero.
Otherwise, an error number is returned to indicate the error.

Errors: The `pthread_join()` function will fail in case of the following errors:

EINVAL
: The implementation has detected that the value specified by `thread` does not refer to a joinable thread.

ESRCH
: No thread could be found corresponding to that specified by the given thread ID.

EDEADLK
: A deadlock was detected or the value of `thread` specifies the calling thread.

The pthread_join() function will not return an error code of EINTR.

Prog. 1.
In this example the same function is used in each thread. The arguments are different. The functions need not be the same. Threads terminate by explicitly calling the pthread_exit() function, by letting the function return, or by a call to the function exit which will terminate the process including any threads.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

void *print_message_function( void *ptr );

main()
{
     pthread_t thread1, thread2;
     char *message1 = "Thread 1";
     char *message2 = "Thread 2";
     int  iret1, iret2;

     /* Create independent threads each of which will execute
     function */

     ret1 = pthread_create( &thread1, NULL, print_message_func-
     tion, (void*) message1);
     iret2 = pthread_create( &thread2, NULL, print_message_
     function, (void*) message2);

     /* Wait till threads are complete before main continues.
     Unless we  */
     /* wait we run the risk of executing an exit which will
     terminate   */
     /* the process and all threads before the threads have
     completed.   */

     pthread_join( thread1, NULL);
     pthread_join( thread2, NULL);

     printf("Thread 1 returns: %d\n",iret1);
     printf("Thread 2 returns: %d\n",iret2);
     exit(0);
}

void *print_message_function( void *ptr )
{
     char *message;
     message = (char *) ptr;
     printf("%s \n", message);
}
```

Index

"The author's approach is very objective and lucid….
Adequate coverage of key topics within the chapters….
The book has been well developed to help a variety of users."

Dr S Ramesh Babu

Salient Features of the Book

Unix Programming, The First Drive helps as an easy reference guide for the first time users. At the same time, it excites the specialists in the Unix World. It:

- simplifies the complicated features and constructs (with examples) and hence it is increasingly important for developers;
- helps non-Unix developers to transition to a Unix environment smoothly;
- facilitates a tutorial approach from Unix commands to Device Drivers comprehensively;
- offers a tutorial approach to understand the UNIX commands, Shell Scripts, Internals and Sockets;
- enables the readers to write sophisticated UNIX programs with features such as File and Directory I/O, Process, Signals, Inter-Process Communication and interaction with hardware devices;

The information provided in the book is equally applicable to C++ programs because C++ is roughly a superset of C. The smooth narration in the entire book reflects in-depth experience of the author on various aspects like Commands, Shell Scripts, Internals, Sockets and Device Drivers.

Visit us at www.wiley.com

READER LEVEL
Undergraduate

SHELVING CATEGORY
Engineering

Price: Rs. 299/-

Wiley India Pvt. Ltd.
4435/7, Ansari Road, Daryaganj,
New Delhi-110 002.
Tel: 91-11-43630000.
Fax: 91-11-23275895.
E-mail: csupport@wileyindia.com
Website: www.wileyindia.com

ISBN: 978-81-265-1613-1

9 788126 516131