UNIX Programming

UNIX Processes, Memory Management, Process Communication, Networking, and Shell Scripting

DR. VINEETA KHEMCHANDANI
DR. DARPAN ANAND
DR. K.K. MISHRA
DR. SANDEEP HARIT

bpb

UNIX Programming

UNIX Processes, Memory Management, Process Communication, Networking, and Shell Scripting

DR. VINEETA KHEMCHANDANI

DR. DARPAN ANAND

DR. K.K. MISHRA

DR. SANDEEP HARIT

UNIX Programming

UNIX Processes, Memory Management, Process Communication, Networking, and Shell Scripting

Dr. Vineeta Khemchandani

Dr. Darpan Anand

Dr. K.K. Mishra

Dr. Sandeep Harit

www.bpbonline.com

Group Product Manager: Marianne Conor

Publishing Product Manager: Eva Brawn

Senior Editor: Connell

Content Development Editor: Melissa Monroe

Technical Editor: Anne Stokes

Copy Editor: Joe Austin

Language Support Editor: Justin Baldwin

Project Coordinator: Tyler Horan

Proofreader: Khloe Styles

Indexer: V. Krishnamurthy

Production Designer: Malcolm D'Souza

Marketing Coordinator: Kristen Kramer

First published: June 2022

Published by BPB Online

WeWork, 119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55510-402

www.bpbonline.com

Dedicated to

Parampita - Paramatama

The lord Shiva and Goddess of Knowledge Maa Swarswati

The inspiration and source of all the possible knowledge

About the Authors

Prof. (Dr.) Vineeta Khemchandani is a Doctorate (Ph.D) in Computer Science, Post Graduate in Computer Applications and ‘Diploma in Banking Technology” from Indian Institute of Banking and Finance, Mumbai. Overall experience of around 23 years in the field of IT in various organizations. The experience spans from software development, Corporate Training , System Administration, Academics, Administration and Research, in various capacities. She significantly contributed with guiding several B.Tech/ MCA projects, M.Tech dissertations and PhDs. Publications of 30 research papers in journals of repute, 6 book chapters and authored 3 books. She filed three patents. She is part of review committee for Journal of King Saud University, INDERSCIENCE Journal of Electronic Government and Science Publishing Group, USA.

She contributed towards education innovation as members of “board of studies” of State level universities in Uttar Pradesh. Currently also she member of board of studies of AKTU, Lucknow for CS, It and MCA programs. She delivered several talks during International conferences and TEQIP programmes. She contributed to National and International professional bodies like IEEE and Computer Society of India as Branch counselor and management committee member. She was part of jury to judge “SMART INDIA HACKTHON”, an initiative of AICTE and Ministry of Human Resource Development (Govt of India) for four consecutive years. Her research work over the past several years has been focused on different aspect of information security and algorithms. Brief areas of research include cryptography, Digital watermarking, visual cryptography, stenography, multi- dimensional data structures, and e-Governance projects.

Prof. (Dr.) Darpan Anand is presently working as Professor in the Department of Computer Science Engineering Department at Chandigarh University, India with more than 18 years of experience in teaching, industry, and research. He is currently a member of the Board of Studies, a Member of the research Degree Committee, and Outcome Based Education

Coordinator, ABET Accreditation Coordinator, Research Coordinator, and the Coordinator of Projects in the Department of Computer Science, Chandigarh University. He has guided several Ph.D. and PG Dissertations. He is an author/co-author of more than 50+research papers (indexed in SCI, ESCI, Scopus, etc.), 1 Textbook, 6 book chapters (IET, Springer, and Elsevier), 4 patents, SWAYAM MOOC courses, etc. In addition to it, he has reviewed many publications for SCI and Scopus indexed journal. He is also a member of various esteemed research associations as IEEE, ACM, IAENG, TAEI, CSI, AIS, CSTA, etc.

Dr. K. K. Mishra is presently working as Assistant Professor in Department of Computer Science and Engineering, MNNIT Allahabad, Prayagraj. He has successfully organized around 6 IEEE conferences in India (ICCCT Series) as a conference secretary and worked as a program chair for many other conferences. He has worked as PC members for many conferences in India and abroad and has successfully organized some special issues in highly index journals. He is a regular reviewer of Journal of Supercomputing (Springer), Applied Intelligence, Applied Soft Computing, IEEE Transaction on Cybernetics, IEEE System Journal, Neural computing and application and IETE journals. In addition to it, he has reviewed many publications for SCI and Scopus indexed journal.

Dr. Sandeep Harit started his formal education from BIET Jhansi with B.Tech program completed, Postgraduate and PhD from MNNIT Allahabad. He is presently working as faculty at department of Computer Science and engineering at Punjab Engineering College Chandigarh, He supervised more than thirty M.Tech Thesis and two PhD thesis. Since last 20 years in profession, he has been involved in various Academic, administrative and research activities. He has published 40+ quality research article in leading journals and conferences of international standards including SCI and Scopus, and IEEE Indexed, with good number of citations as well. He possess great acumen for research and have inquiring spirit with his desire to keep expanding the boundaries of his knowledge. Apart from cognitive abilities, he is also proficient in implementing theoretical concepts into practices.

About the Reviewer

Dr. Akash Punhani is currently working as Associate Professor at SRM Institute of Science and Technology, Delhi NCR campus, Modinagar, Ghaziabad, India. He is the part of academics since 2007. He has done his Ph.D. on Interconnection Networks titled "On Improving throughput and latency of mesh interconnection networks" in year 2018 from Jaypee University of Information technology, Solan Himachal pradesh, India. During his Ph.D. he proposed the routing algorithms and topologies that can be used on chips and help in improving communication on System on Chips. He has published the articles in the various reputed journals and conferences. His area of research includes Network on chips, Machine learning and Optimization Algorithms. He is also an active reviewer of various journals of high impact factors. Apart from the research, he has good command over the various programming languages C, Python, Matlab. He also has worked simulators like NS2, OMNET++ and NOXIM. He has guided 2 M. Tech. Thesis and over 20 B. Tech. Projects. Akash Punhani has on going mission for the developing and implementing ideas for exploring, improving and optimizing the utilization of the resources in the society and education.

Akash Punhani,

Ph.D. Computer Science & Engineering

Acknowledgement

From the depth of our heart, we dedicate this work to the supreme teacher and supreme guide, The God, who has given us the strength and paved the path to carry this task. We'd also like to thank many people who have helped us to learn and practice fundamentals and advanced concepts of the UNIX operating system. We would like to thank our superiors from industry who have shown confidence in us and given the challenging task of UNIX server administration.

We would like to thank our colleagues, reviewers, editors and publishers for adding value to the contents. We'd like to thank our students and scholars because their enthusiasm always motivates us to give more.

Last but not the least we are very grateful to our family members who are always supportive in all the work we do.

Preface

Having rich industrial experience of authors to work in UNIX operating system. We feel very passionate about the subject and enjoy working and teaching. While working, We consulted various books, journals and articles. Hence, we are trying to consolidate all material and bring out a book, which, we are sure will help the readers.

The book is useful for Engineering students and Master's degree students for their academic preparations as well as competitive exams. In each chapter a problem-solving approach has been followed. More emphasis has been given to develop self-learning and greater thinking skills to understand the concepts rather than following textual content.

The book fulfills the beginner's requirements to clear Multi-user OS concepts with gradual lead to the high-level concepts and programming, so that it can be used as text book at the university level as well it can be used to apply the high-level concepts further. The book contains numerous debugged programming examples to increase understanding of the subject.

Chapter 1 discusses the evolution of the UNIX operating system since its beginning and its continuing significance as technology and operating system. The chapter discusses UNIX system architecture, programming environment and it gives a structured overview of all system resources and their management. The chapter is organized in such a way that all major entities and their function should be clear at a glance to enhance further understanding of the topic when it is covered in details in other sections of the book.

Chapter 2 discusses the most important function of the UNIX operating system responsible for all activities in the file, the chapter covers UNIX file system, Kernel Data structures to perform File I/O, Basic File permissions, Library functions and UNIX system calls for File input/output.

Chapter 3 will cover more comprehensive and detailed functions required for process management like process status, process control block, process control and further also discusses about the accessing of user information.

Chapter 4 will cover Inter process communication which is a method through which processes share data and information with each other. Inter-process communication is required to synchronize action between cooperating processes for uninterrupted sharing of system resources. This chapter entails the IPC options available in UNIX operating system. The chapter also discusses the methods of processes synchronization in details.

Chapter 5 Socket programming is a technique which allows two network nodes to communicate with each other. Socket is used as an interface to send. Receive messages between networking processes. The chapter discusses socket data structure, system call for socket communication and different types of I/O models. Naming and address conversion system calls are also discussed in this chapter.

Chapter 6 Memory Management is considered as one of the important resources i.e. memory of the system. This chapter discusses briefly about the types of memory. Different types of memory management techniques implemented in the UNIX operating system and functions used to allocate and de-allocate memory to the program for its efficient use.

Chapter 7 explains, Shell a command-line interface to the Unix system. Shell provides an environment to execute commands and programs. The chapter provides comprehensive concepts of shell including its type, modes of execution, functions of shells and command execution.

Chapter 8 is to explain about the working of the shell which is both an interactive command language and a scripting language, and is used by the operating system to control the execution of the system using shell scripts. Shell scripts are computer programs to execute a series of shell commands to perform a system task. The chapter all constructs to write a shell script along with examples.

Table of Contents

CHAPTER 1
Fundamental Concepts of UNIX Operating System

UNIX is a powerful operating system, first developed in Bell Labs and has been under development ever since. It is very popular among scientific, engineering, and academic communities because of its stability, multi-user, and multi-tasking environment for servers, desktops, and laptops. The system is divided into two parts consisting of programs and services that have made the UNIX system environment so popular. The second part consists of the operating system, which supports these programs and services. The two entities files and processes are the main control concepts in the UNIX system model. This chapter discusses the architecture and overview of these control entities of the operating system.

Structure

We will cover the following topics in this chapter:

- History of UNIX
- Salient features of UNIX
- Architecture of UNIX operating system
- Unix programming environment
- Unix process
- Handling files and process control entities.
- Unix file system

Objectives

After going through this chapter, you will be able to:

- Understand salient features of the UNIX operating system to make it a popular OS.
- Get to know about the architecture and functioning of each layer.
- Learn about UNIX programming environments.
- Understand fundamental concepts of files and processes.

History of UNIX

Bell Laboratories, jointly with MIT and General Electric, developed a new operating system Multics (Multiplexed Instruction Computing Service), in 1965. The main features included in the operating system were multi-user, multi-processor, and multi-level (hierarchical) file systems, among its many forward-looking features.

In the year 1969, a few programmers, Ken Thompson, Dennis Ritchie, and others working in Multics, designed and implemented the first version of the UNIX file system on a PDP-7. After including a few more utilities, it was given the name UNIX by Brian Kernighan. UNIX was written mainly in assembly language.

In the year 1971, the system was implemented on PDP-11 with 16K bytes of memory, including 8K bytes for user programs and a 512K byte disk. It was designed with the following features due to which it caught the interest of researchers:

- Programmers environment.
- Simple user interface.
- Simple utilities that can be combined to perform powerful functions.
- Hierarchical file system.
- Simple interface to devices consistent with file format.
- Multi-user and multi-process system.
- Architecture is independent and transparent to the user.

In the year 1973, it was decided to re-write the UNIX in the C language developed by Dennis Ritchie. This decision made it easy to port the UNIX to new machines.

In the year 1974, it finally got recognition from the academic community when Thompson and Ritchie published a paper in the Communications of the ACM describing the new UNIX OS.

In the year 1980, Berkeley released the BSD 4.1 (Berkeley Software Development) version of UNIX.

In the year 1988, AT&T and Sun Microsystems jointly developed System V Release 4 (SVR4).

In the year 1993, Novell bought UNIX from AT&T.

Salient features of UNIX

UNIX is a very popular operating system due to its special and useful features, and these are discussed as follows.

Portability

UNIX is considered a portable operating system as a large portion of it is written in the C language, and only a very small portion of it is written in assembly language. Because the C program can be easily moved from one hardware environment to another, provided a standard Compiler is available, this makes the UNIX code run on different hardware. All this only requires a standard C compiler.

The application program interface allows many different types of applications to be easily implemented under UNIX without writing assembly language. These applications are relatively portable across multiple vendor hardware platforms.

Multi-tasking

This is managed by dividing the CPU time intelligently between all jobs. Each job gets a time slot depending on its priority. A user can print a file and, at the same time, execute a C program. Users do not have to wait for an application to end before starting another one. A huge file can be sorted as a background job while working on a foreground job.

Multi-user

The UNIX environment design allows multiple users to work concurrently, and hence, they can share hardware and software resources of hard disk, CPU, memory, printer, and file by working on a separate terminal. A terminal is a keyboard and monitor, which is connected to the main computer called the host machine, server, or console. In UNIX, every user gets an equal chance to share resources while preventing the user from locking other's resources.

Device independence

In UNIX, input/output devices are treated like ordinary files. Input and output from the files and devices are handled using a design feature called indirection (<, >. <<, >>) without going into detail specifications of devices. Input to a program can come from any file or device, and output from a program can go to any file or device.

Modularity

UNIX kernel consists of modules, and system administrators can customize it to include only those modules that are normally required. If a new feature is required later, the corresponding module can be added to the Kernel, and it can be reconfigured or rebuilt.

Networking

Originally networking was not incorporated into the UNIX system. Networking was added after the separation of UNIX between BSD UNIX and At&T UNIX.

Networking allows a user to log on to the remote system. Once access is gained to a remote system, users can use system resources using UNIX commands according to permission granted to him/her. A standard communication protocol known as TCP/IP is used to access other system resources. UNIX also supports a network file system, which allows users to access files on another network too.

Tools and utilities

Productivity of the system is increased with available software utilities and tools. UNIX is very rich in tools and comes with hundreds of programs or tools. These tools are either integral utilities that are absolutely necessary for the operation of the computer, such as command interpreter, and tools that are not necessary for the operation of the UNIX but provide the user with additional capabilities such as email, typesetting capabilities such as troff, nroff, awk, sed, and so on.

Security

UNIX has several levels of security:

- Assigning login name and password to individual users provides the first level of security.
- The second level is at a file level, where each file has read, write and execute permission associated with the owner of the file, group members, and others, which decide who can read, write, or execute a file.
- The third type of security is that files can be encrypted so that the file is in an unreadable format, and only you can decrypt it to read it.

The UNIX system architecture

The UNIX system has become quite popular since its inception in 1969, running on machines of varying processing capacities from microprocess to the mainframe and providing a common execution environment across them. The operating system interacts directly with the hardware, providing common services to the programs and insulating them from the hardware.

UNIX operating system is made up of layers around the hardware layer, as shown in *figure 1.1*:

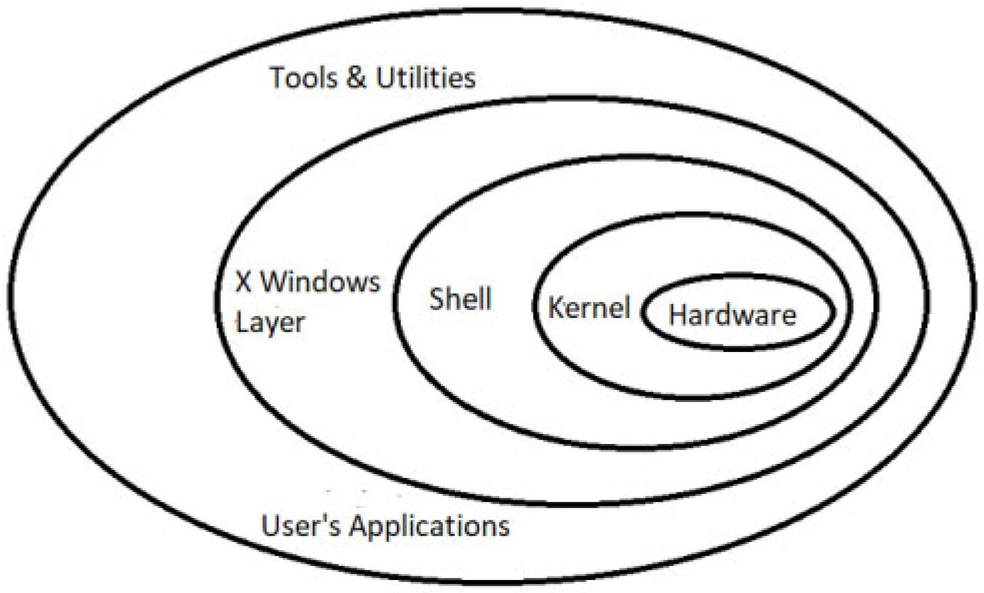

Figure 1.1: *UNIX system architecture*

It contains:

- Hardware layer.
- Kernel layer.
- Shell layer.
- Windowing layer as X—interacts with the shell but can interact with applications and commands.
- Utilities and application programs or tools layer.

Viewing the system as set of layers, the operating system is commonly called the system kernel or just the Kernel. It emphasizes its isolation from user programs because programs are independent of the underlying hardware, and it is easy to move them between UNIX systems running on different hardware. If the programs do not make an assumption about the underlying hardware, programs such as the shell and editors shown in the next layer interact with the Kernel by invoking a well-defined set of system calls. The system calls to interact with the Kernel to do various operations for calling the program and exchange data between the Kernel and the program. Several programs known as commands are above this layer. Other application programs can build on top of lower-level programs. Let us discuss them in detail:

- Hardware layer: Innermost layer that provides the services for the OS is called the hardware layer. This includes the terminals, terminal controllers, disks, tapes, memory, and various device controllers.
- Kernel: This is the central part of the operating system, which interacts directly with the hardware and provides the services to the user programs. The Kernel is stored in a file called "UNIX". There is only one Kernel running on the system. User programs interact with the Kernel through a set of standard system calls that are a part of the Kernel and provide the following basic functions of the Kernel:
    - File system management: This includes management of the UNIX file system that comprises files and directories. This includes creating, opening, reading, writing, closing files, and other file system-related operations and data transfer between the file system and the hardware.
    - Memory management: This includes allocation of memory to programs, sharing of memory spaces, or freeing memory. If the system is low on free memory, the Kernel frees memory by writing a program temporarily to secondary memory called a swap device.
    - Process management: In UNIX, a process is a program in execution. Process managing includes scheduling of various processes to the CPU and execution of processes, including creating, terminating, or suspension of a process. It also includes inter-process communication.
    - Storage management: Allocation of secondary storage disks for user data and management of free space.
    - Device management: This includes allowing controlled access to peripheral devices such as terminals, tape drives, disk drives, and network devices.
- Shell: This is the part of the UNIX that is most visible to the user. It is the command interpreter that interprets the commands entered by the user and conveys them to the Kernel, which executes them. The shell is a program running separately for each user working with UNIX. The number of shells running in the system is equal to the number of users working in the system at a time. There are different types of

shells, and the commonly used ones are the Bourne shell, C Shell, Korn Shell, and so on:

  - Bourne shell created by Steve Bourne is most popular and bundled with every UNIX system.
  - The C shell was popular among UNIX programmers and was created by Bill Joy at Berkeley. The advantages of the C shell over Bourne are as follows:
    - It allows the aliasing of commands; that is, we can rename command with our alias so that we can use a short alias name for a long command.
    - C shell has a command history feature in which previously typed commands can be recalled.
  - The Korn Shell is a superset of the Bourne shell and is more powerful. David Korn of AT and T Bell labs designed it.
  - Bourne Again Shell (Bash) from the Free Software Foundations GNU project, based on shell.

- Utilities and application programs: There are hundreds of utilities in UNIX. A utility is a standard UNIX program to provide support to the user. Most of the UNIX utilities are available as application programs:
  - File management (**`rm`**, **`cat`**, **`ls`**, **`rmdir`**, **`mkdir`**).
  - User management (**`passwd`**, **`chmod`**, **`chgrp`**).
  - Process management (**`kill`**, **`ps`**).
  - Printing (**`lp`**, **`troff`**, **`pr`**).
  - Program development tools such as sccs, make editors (vi and ed), and compiler.
  - UNIX email utility.
- Windowing layer: The GUI on UNIX is a separate layer that sits on top of the core operating system. X window system is an underlying graphical layer for all modern UNIX operating systems. The X Window System just facilitates the system to accept input from devices such as keyboard and mouse and to draw the graphical objects on display. Drawing ad taking input from devices is the job of

software called a Window manager. There are many Window managers available for X. Generally, the Window manager is proprietary software and will only run on a particular type of UNIX. Some of these are open and will run on any system that has X installed.

The X Window system has some very unique and useful features. These things are provided by the X itself and, as such, are common to all X window managers and all UNIX/X11 systems.

UNIX programming environment

UNIX is a portable operating system basically designed for multi-user and multi-tasking environments. It contains hundreds of simple, single-purpose functions that can be combined virtually to do every processing task imaginable. UNIX system is used in three different programming environments.

Personal environment

Although UNIX was originally designed for multi-user environments, but some users install it on their personal computers. This trend is accelerated with the invention of LINUX, a free UNIX system. In 2001, Apple systems incorporated UNIX as its Kernel.

Time-sharing environment

UNIX is a multi-user and multi-tasking system that runs in a time-sharing environment to handle multiple tasks by a single processing unit. The Kernel maintains a list of current tasks and allots a slot of time to each task, then to the next, and so on. Normally, Kernel switches from one process to the next so rapidly that each user has an impression that he has individual attention.

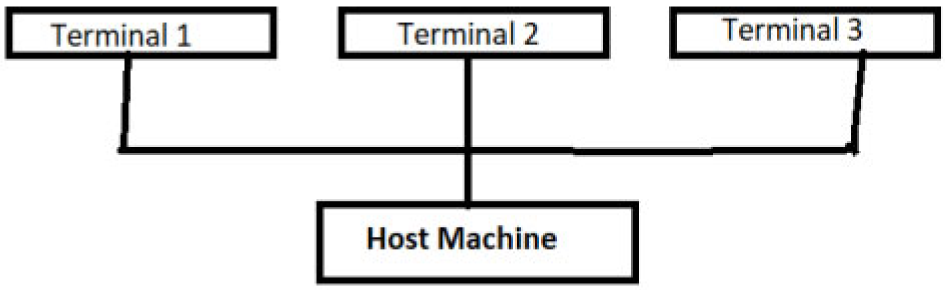

Figure 1.2: *Multi-user environment*

Client-server environment

A client/server computing environment divides processing functions between a server computer and a client computer. In the client/server model, data processing load is shared between the client computer and more powerful server computers. The server computer is dedicated to storing and managing data and fulfills client requests. The server receives structured requests from the client, processes the requested data, and sends the data back to the client. His client computers take the user's input from an interface, process some information using its CPU, and present the result to the user.

Note: In the single-user environment, a program may have multiple instances because the process may be split into two processes by default, sharing the same code.

UNIX process

A program in execution is a process. In UNIX, a process is a unit of work. If n users are executing the same program, then there are *n* processes executing in the system. A process executes by following a strict sequence of instructions that is self-contained and does not jump to that of another process. All the processes execute concurrently with the CPU switching between the processes.

All the UNIX commands are programmed, and thus, contribute to the process in the system when they are under execution.

Process attributes

Every process in UNIX is attributed to some properties (as shown in *table 1.1*), which affect the execution of the process.

Attribute	Meaning
PID	Process identification is a unique number
PPID	Process ID of the Parent
UID	User ID of the owner of the process
GID	ID of the group to which the owner belongs to
EUID	Effective User ID
EGID	Effective Group ID
Priority	The priority the process run at

Table 1.1: *Process attributes*

Process states

The lifetime of a process can be modeled by a set of states, and each process can be in any of the states at one point in time. In UNIX, the process may undergo a total of nine stages during its lifecycle, as shown in *figure 1.3*:

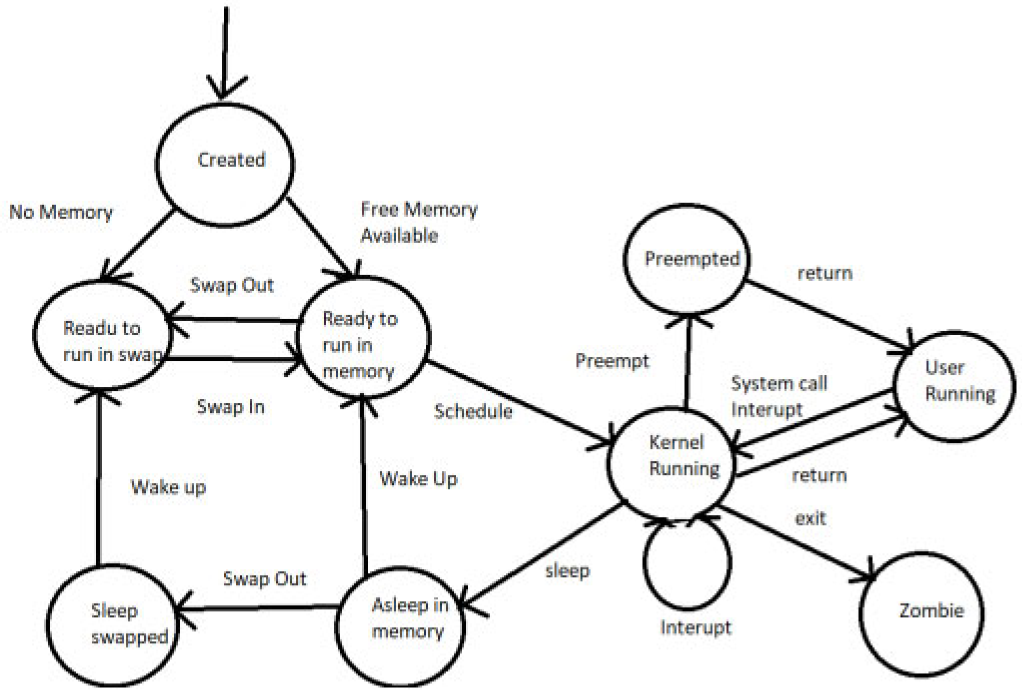

Figure 1.3: *UNIX process sates*

The description of each process state is shown in table 1.2:

Process states	Description
Executing (user mode)	Running in user mode.
Executing (Kernel-mode)	Running in kernel mode.
Ready in memory	Ready to run as soon as kernel schedules it
Ready in swap device	Process is ready to run but must be swapped in memory so that Kernel can schedule it.
Sleeping in memory	Unable to execute, waiting for an event to occur.
Sleeping in swap device	Process is blocked and awaiting an event in swap device.
Preempted	Process is returning from Kernel to user mode, but Kernel preempts to schedule another process.
Created	The newly created process is not yet ready to execute.
Zombie	Process just completed but not yet left its resources.

Table 1.2: *Process states*

In UNIX, processes are executed in two different modes, user mode and kernel mode (explained in the next section).

Ready to run in memory and preempted are essentially the two same states, but the distinction is made to just emphasize how the process enters in the preempted state. When the process executes system calls and interrupts the handler (clock or I/), it is in kernel mode. When Kernel finishes its job, the process moves from kernel mode to the user mode. However, Kernel can preempt such jobs to execute other ready-to-run jobs that have higher priorities.

Modes of execution of a process

There are two modes of execution of the system as seen by the hardware:

- User mode: When the user process executes, it does so in the user mode. In the user mode, processes can access their own instructions and data but not kernel instructions and data or those of other processes.
- Kernel mode: When a user process executes a system call, the execution mode of the process changes from user mode to kernel mode; the Kernel executes the users' request. Processes in kernel mode can access both Kernel and user addresses.

A typical UNIX system consists of a collection of processes: operating system processes executing system code and user processes executing user code. System processes run in kernel mode to perform administrative and housekeeping activities such as allocation of memory, system accounting, and process scheduling. User processes run in user mode to execute programs and utilities. User processes move into kernel mode if it invokes a system call or when an interrupt or exception has occurred.

Note: UNIX is not suitable for real-time processing because preemption can only occur when the process is about to move from kernel mode to user mode. When the process is in Kernel mode, it cannot be preempted.

Process context

The process context of the process represents the current operating state of the process. The context of a process includes its address space, stack space, virtual address space, register set image, accounting information, associated kernel data structures, and state information. In general, process context has the following three components:

1. User-level context: User-level context contains the basic user's program generated after compiling the source file. The basic elements of the program are categorized in text and data. The text holds a read-only set of instructions for the program. The data area is the shared memory area accessible by many processes. There is only one shared memory area, but with the concept of virtual memory, each sharing process may have a shared memory region in its address space. The user stack area is used to store parameters that return values and pointer to function calls.
2. Register context: Register level context stores process status information in the processor's registers when the process is not executing.
3. System-level context: System-level context consists of information required by the operating system to manage processes. System-level context has static and dynamic parts. The static part contains an entry for each process in the process table to store process control information to be available to the Kernel at all times. The user (U) area is another part of the static context, which contains additional control information required by the Kernel when it is executing within the context of the process. The third part of the static context is the per-process region table to be used by a memory management system to swap in and out processes from/to main memory. Finally dynamic part of the system-level context is the Kernel stack.

Kernel stack is used for processes executing in kernel mode. Kernel stack stores information required when an interrupt, or procedure call occurs.

Kernel stack is used by a process executing in Kernel mode. Kernel stack stores information required when an interrupt, or procedure call occurs. *Table 1.3* shows the set of information available under different contexts of the process currently executing in the system.

User-level context	
Process text	Executable machine instructions associated with the program.
Process data	Data accessible by the program.

User stack	Contains local variable, arguments, and pointers to functions executing in user code.
Register context	
Program counter	Address of next instruction to be executed.
Stack pointer	Point to the top of the kernel or user stack depending upon the operation mode of process.
General-purpose registers	Value of these registers depends on implementing hardware.
Processor status register	Status of processor at the time of preemption.
System-level context	
Process table entry	Contains the state of the process.
Per process region table	Mapping from virtual to physical addresses and access permission on the process.
User (U) area	Process control information is required for a context switch.
Kernel stack	Stack frames for kernel procedures.

Table 1.3: *Process contexts*

Process relationship

Processes in UNIX have a parent–child relationship, shown in *figure 1.4*. Each process has one parent and one or more children. The process ID 0 is the scheduler process and is called swapper. Swapper is part of the Kernel and is called the system process. The init process (Process id 1) is the process dispatcher, which gives birth to the shell, and all processes initiated by us are children of the shell, and so descendants of the init process. The init is a normal user process and never dies. UNIX keeps track of all processes in an internal data structure called a process table.

When the UNIX system is started for normal multi-user operation, then Kernel invokes the program. etc/init. The init process has Process ID 1 and runs under super-user privileges (User-ID 0). The operation of init depends on different UNIX systems.

init process in 4.3 BSD

In 4.3 BSD, to startup the system init executes the shell script `/etc/rc`. The file does some bookkeeping and then calls daemon processes. The init then reads the **/etc/ttys** process to determine which terminal is activated in a multi-user environment.

init process in UNIX system V

In system V, to startup the system init read /etc/initab file to put the system in a different mode (single-user or multi-user) depending on the value of the run level. For normal multi-user operation **/etc.rc** file is executed, which starts various daemon processes.

Regardless of the UNIX system, init process invokes fork copies of itself to activate terminals. Each child process executes the **/etc/getty** process to set the terminal's speed, display a greeting message, and waits for the user to enter the login name. The Getty process executes **/bin/login** program to check whether login exists in /etc/passwd file or not. If it finds the login name in ./etc/passwd file, it prompts the user to enter a password. If rejected, Getty gives three chances to enter your login and password.

The init, Getty, and login processes are executed with the user ID and effective user ID of the super-user. All these processes run with the same process ID because the process ID does not change during the exec system call.

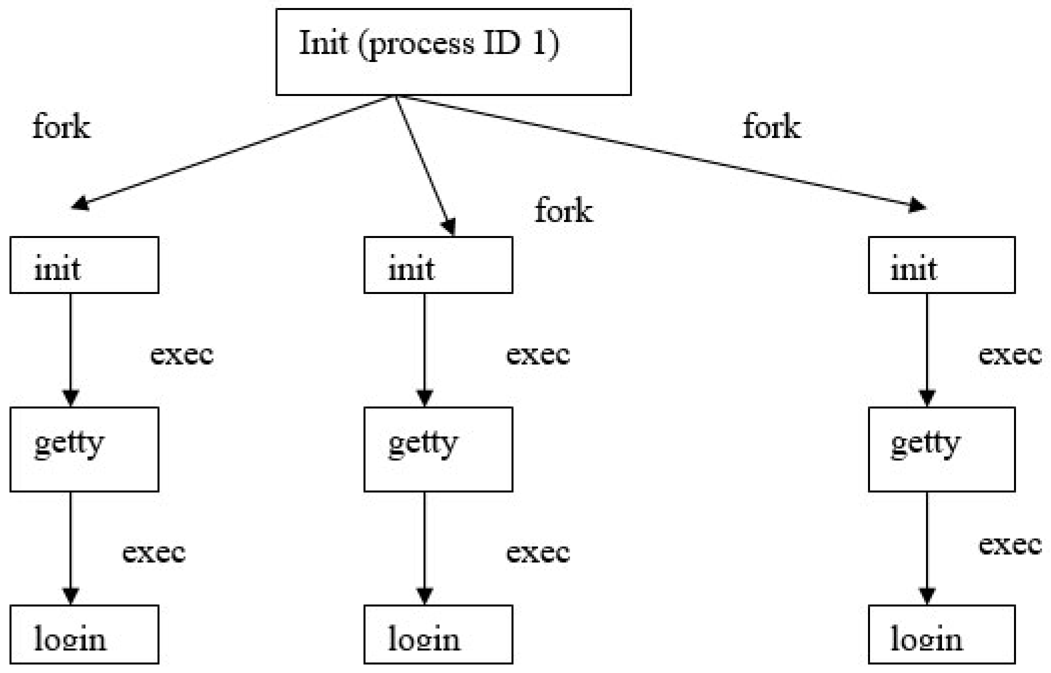

***Figure 1.4:** Process relationship*

Login program

While logging in, the system login program sets the following things:

- The current working directory as a login directory using **`chdir`** system call.
- The effective group ID and the effective user ID of the login process in the order specified in the password file entry using **`setgid`** and **`setuid`** system calls.
- Executes the shell program as specified in the **`/etc/passwd`** file, or **`/bin/sh`** if the shell is not specified in the passwd file.

The shell that is invoked by the login program is called login-shell and is treated as the parent of all processes.

Shell process

The shell process normally waits for the user to enter the command. When the user enters the command on the command line, the shell forks make a copy of itself and wait for the child process to terminate. The child shell process checks the PATH environment variable to search the command file. Commands in UNIX are generally available in /bin directory.

When the command finishes, it calls an exit system call and exits with its exit status, which terminates the child process and allows the wait to return to the parent process.

Network login

In 4.3 BSD, to startup the system, init executes the shell script **`/etc/rc`**. The file does some bookkeeping and then calls daemon processes. One of the daemons that are started by the shell script is netd. Once the shell script terminates, the parent process of inetd becomes init. This daemon can be used by a server that uses either TCP or UDP. This daemon creates sockets on behalf of a number of services and listens to all of them simultaneously.

Files and directories

The organization of the UNIX files is shown in the following figure:

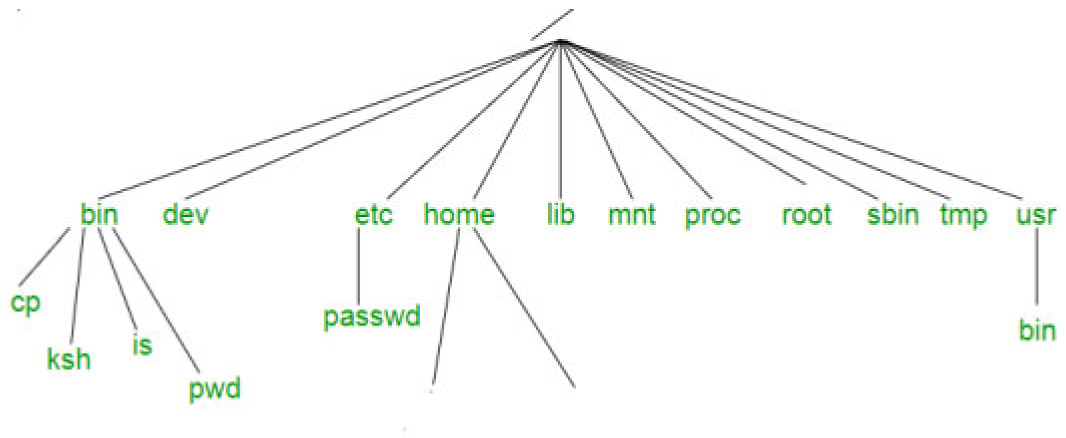

Figure 1.5: *UNIX directory structure*

UNIX file system is a hierarchical tree structure of directories and files. The tree grows or decays dynamically.

The file system tree starts with a single node root `/`; every non-leaf node is a directory of files, and leaf nodes are directories, regular files, or special device files.

File

To UNIX, everything is a file; this means in UNIX data, program files, directories, devices, and links are considered as files. Each file has a name, an owner, and access rights. Each file is assigned an inode number that is unique. All attributes of files are stored in the filesystem in an inode entry. An inode entry stores everything except the filename. The names are stored in directories and associated with inode through pointers.

Each inode generally contains the following:

- The location of the file's contents on the disk.
- Type of file.
- Size of the file in bytes.
- The time when file's inode was last modified.
- The time when file's contents were last modified.
- The time when the file was last accessed for reading, writing, and for other operations.

- The reference count, different number of names the file has.
- The file's owner's ID.
- The file's group ID.
- File's access permission bits.

In UNIX, every file has a nonnegative number associated with it called the file descriptor. The Kernel uses a file descriptor to identify a file when any program accesses it. Every file or directory has an associated filename in the directory entry. Slash and null characters cannot appear in the filename. Some UNIX systems restrict the filename to 14 characters, and most other systems allow 256 characters in the filename.

Every directory and file has a corresponding entry in its parent directory. Directory entry specifies an attribute of the file, such as type of file, size of the file, owner of the file, and permission on the file. The root directory is the first directory of the directory tree, and it contains its own information. Every directory also contains information on two special directory entries, the current directory and the parent of the current directory.

`Note: [Reserved Inode Number]:`

`0: Deleted files / directories`

`1: File system Dependent, ( file system creation time / bad block count etc )`

`2: Refers to the root directory of the file system.`

Pathname

Every file or directory is identified by its pathname; the pathname of the file or directory is either absolute or relative to a location.

The absolute pathname starts with the root (`/`) and follows the branches of the directory tree, each separated by `/`, until the required file is found, for example, **`/usr/bin`**.

A relative pathname specifies the path relative to another, usually the current working directory. So if the current working directory is **`/usr/bin`** then the path of the file in this current directory is specified as: **`../date`**. This indicates that is control should move one directory level up, then come down through the date.

Types of files

UNIX considers everything as a file. There are seven types of files in UNIX, as shown in the following figure:

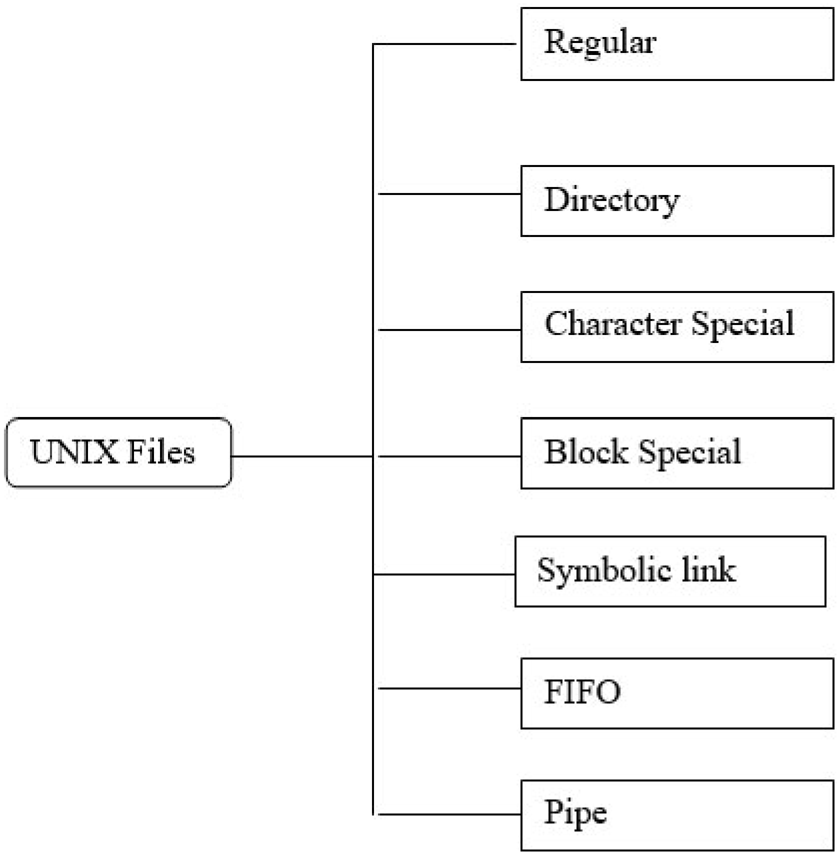

Figure 1.6: *Types of UNIX files*

The UNIX files can be described as follows:

- **Regular files:** These are a sequence of unformatted bytes. A regular file may contain ASCII or binary characters or a combination of both. A text file is also considered as a regular file. ALL shell scripts and programs are written in high-level languages, and even object codes are also considered as regular files in UNIX. There's no header, trailer, label information, or EOF character as part of the file.
- **Directory files:** These are a sequence of formatted bytes and contain information about other files and subdirectories. A directory contains

information on the name of the file or directory in it and its inode (index node) number.

For directory files, the output of the `ls -l` command shows the first character as d. To change into a directory or list files in a directory, we need read permission for that directory. To create a file or subdirectory in that directory, we need write permission.

- **Special files:** This represents physical devices such as keyboards, tapes, disks, and printers. They are characters, block files, sockets, and so on. The operating system uses these files to communicate with the hardware.

We can use the same commands for these special files that we use for regular files, but the operating system takes care of handling the devices. The physical devices are represented /dev directory.

- **PIPES:** A type of file used for inter-process communication between processes. Their implementation allows processes to communicate even though they do not know what processes are on the other end of the pipe. Pipes are basically of two types: named pipes and unnamed pipes.
- **Sockets:** UNIX sockets are used to establish communication between two processes residing same or different machines. Just like other types of files, sockets are also associated with a file descriptor.
- **Symbolic link:** A symbolic link is a special type of file that points to an existing file. It is also call soft link. The file contents can be either accessed through the file name or through the link name.
- **FIFO:** FIFO that is First In First Out is a file, which that has a directory entry and is accessed by pathname. It works as a pipe between two processes.

UNIX file name convention

Every file or directory has an associated filename in the directory entry. Some UNIX systems restrict the filename to 14 characters, and most other systems allow 256 characters in the filename. In general, UNIX follows some naming conventions for files:

- File names are case-sensitive.
- Almost all characters on the keyboard are allowed in the filename except.

```
| ; , ! @ # $ () <> / \ " ' `{} [] + = & ^ <space> <tab>
```

- Character delimiters are used to make file name easy to read.
- In UNIX file name does not specify its functionality.

File names and meta characters

The shell expands the filename if it contains *, ?, and [characters. The * in the file name can be replaced with 0 or more permissible characters, the ? can be replaced with any single character in the file name, and square brackets specify a range of particular combinations of characters. The range of characters is specified using—in the square brackets. If the first character inside the square bracket in ! sign, the complement of the range of other characters is used. *Table 1.4* shows different patterns and meanings to refer to a file or set of files.

Pattern	Expansion
*	All files are in the current directory.
?	All files with one character in the file name.
[a-h]	One character file name consisting a to d.
[abcdefgh]	Same as above.
[a-ef-h]	Same as above.
*[0-9]	File where the name ends with a number.
?[0-9]	Two character file name that ends with a number.
[a-zA-Z0-9]	File names that contain a single letter or number.

Table 1.4: *Pattern used for file naming*

Note: In UNIX, the dot is not a special character. The file name may or may not contain a dot. Normally file names where the name starts with a dot are not listed using the ls command.

UNIX file system

Each disk drive is divided into various partitions. Each partition may contain a file system. A file system consists of a sequence of logical blocks, each consisting of 512, 1,024, or any other multiple of 512 bytes. The size of the logical block within the filesystem is the same, but it can vary across the different file systems.

The file system has the following structure, which is described as follows:

Boot block	Super block	Inode list	Data blocks

The boot block is the first block and contains the bootstrapping code required to boot the system. Although only one boot block is needed to boot the system, every file system has an empty boot block.

The superblock describes the state of the system and contains information including the following:

- Size of the file system
- Number of free blocks
- List of available free blocks
- Index of next free block
- Size of inode list
- Number of free inodes
- List of free inode
- Flag to check whether the superblock has been modified

The inode list is a list of the inode. It contains entries for every index node of a file. Inode contains:

- File type
- Permissions
- Number of hard links
- Owner user-id
- Group id
- Size of the file
- Time of last access

- Time of last modification and
- location of data blocks

Finally, data blocks start at the end of the inode list and contain file data and administrative data. An allocated data block belongs to only one file system.

Important UNIX directories

Table 1.5 shows important UNIX directories related to administrating the system resources, including users, files, processes, and devices.

Directory	Description
`/`	Root-kernel.
`/etc`	Files are required to boot the system, communicate, and scripts to control the boot process.
`/etc/config`	System configuration option files.
`/etc/cron.d`	Cron access files and FIFO.
`/etc/rc#.d`	Operations performed when entering run level # (S,0,1,2,3).
`/usr`	Directories of system files.
`/usr/bin`	System binary files.
`/usr/etc`	Further system communication and administration programs.
`/usr/lib`	Libraries of object files, send mail.
`/var`	Directories for administrative programs and logs.
`/var/adm`	System log and account files.
`/var/log`	System log files.
`/var/spool/mail`	Mail spool directory.
`/var/mail`	Mail spool directory.
`/var/spool`	Directories for cron, logs, and so on.
`/dev`	Devices directory.
`/home` `/usr/users`	User directories.
`/usr/local`	Locally installed files.

`/tmp`	Contains temporary files created in the system.

Table 1.5: *Important UNIX directories*

Conclusion

The operating system is a special type of system software that manages all operating resources to facilitate users to perform their work. UNIX is a multi-user, multi-tasking portable operating system to facilitate text processing, programming communication, and many other tasks. UNIX operating system works in various different environments. The major components of the UNIX system include the Kernel, the shell, and the Utilities and application program. Among the functionality of the Kernel, the most important is process and file management. A process is an instance of a program that can open many files. Each process in the system is identified by different identifiers. Each process undergoes nine states during its life cycle. The file management and various file related functions will be discussed in next chapter. It includes the I/O functions, various system calls and library functions for handling file operations and many more related topics.

Key terms

- **Kernel:** Part of the operating system that resides in memory all time and performs most essential tasks.
- **Multiprocessing:** A technique that allows a single processor to process multiple programs residing simultaneously in the memory.
- **Parent process:** A job that controls one or many child processes which it created.
- **Child process:** A process that is created and controlled by the parent process.
- **File:** Collection of data. File in UNIX typically includes ordinary file, directory, and device files.
- **File attributes:** Characteristics that describe the file.
- **init:** First non-kernel process, thus, parent of all processes.
- **Process:** Program under execution.

- **Process ID:** Numeric identifier of process.
- **Directory:** Type of file that serves as a container for other files and directories.

Test your skills

1. **Which of the following information is stored in the inode structure?**
   a. The file size
   b. The name of the owner of the file
   c. The access permission for the file
   d. All dates since the file was last modified or accessed
   e. The number of symbolic links
2. **Absolute path begins with the path from the root.**
   a. TRUE
   b. FALSE
3. **Which of the following files in the current directory are identified by the regular expression a?b*.**
   a. afile
   b. aab
   c. abb
   d. abc
   e. axbb
   f. abxy
4. **Works as a command interpreter.**
   a. Hardware
   b. Kernel
   c. Shell
   d. CPU

5. **The process which terminates before the parent process exits is called as.**
    a. Zombie
    b. Orphan
    c. Child
    d. None of the other options listed for this question
6. **Context switch means**
    a. Kernel switches from executing one process to another
    b. Process switches from kernel mode to user mode
    c. Process switches from user mode to kernel mode
    d. None of the other options listed for this question

Answers

1. **e**
2. **a**
3. **b-c-e**
4. **c**
5. **b**
6. **a**

Review exercise

1. What happens if the file mode creation mask is set to 777?
2. Does UNIX have a fundamental limitation on the depth of a directory tree?
3. Explain absolute and relative pathnames with examples.
4. What do multiprogramming, multi-user, and multi-tasking means?
5. With a neat diagram explain the relationship between the Kernel and the shell of Unix.
6. Explain the wild cards * and ?

7. Where is the password stored ?
8. Explain standard UNIX file hierarchy
9. What do UID and GID signify?

CHAPTER 2
File Management

A file is a collection of related information that is recorded on secondary storage. Or file is a collection of logically related entities. From user's perspective, a file is the smallest unit of secondary storage to provide input and receive output from the computer program. In storage, a media file is just a sequence of bits, bytes, or records whose logical meaning is defined by the file owner.

The structure and logical rules to manage files in the data storage are called a file system. To understand the internal operation of a filesystem, it is required to know how files and directories are presented externally to the user and how they are represented internally in the system. A storage medium may have many file systems existing in different partitions. This chapter discusses the structure of the UNIX file system and file system-related system calls to manipulate files.

Structure

We will cover the following topics in this chapter:

- System calls and library functions for I/O
- Kernel data structures used to handle all opened files
- File access permission
- Standard files for maintain user passwords and aging information
- Structure and operations on the UNIX file system

Objectives

After going through this chapter, you will be able to:

- Know how the UNIX operating system organizes files in secondary storage

- Understand the file-related activities such as naming, retrieval, and sharing
- Learn how the file protection policies are implemented in UNIX operating system

File input/output

File input/output (I/O) is an important function of any operating system, and file input is performed to provide information to the executing process to manipulate existing information. File output is performed to permanently store either first-time created information or updated information through a UNIX process. UNIX maintains some Kernel data structures to manage FILE I/O. These data structures store Metadata required to access and update all files opened in the system. UNIX FILE I/O is performed using either C standard library functions or UNIX system call. In this section, we are discussing various system calls to perform UNIX FILE I/O.

Kernel data structures for file input/output

The following three data structures are used to manage opened files in the UNIX file system:

- **The Kernel file table:** The file table contains an entry for each file opened by any process. It is a global data structure that keeps track of bytes offset during read and write operations; each entry in this file contains flags to indicate read/write access, blocking/no blocking, and so on, and an entry contains a pointer to the v-node table entry for the file. The Kernel file table contains a reference count, which shows how many times the file is opened.
- **Per-process file descriptor table:** It monitors all opened files for a process. Each entry contains a pointer to the Kernel file table. A file descriptor used in read, write, and other system calls is an index in this table. The file descriptor is an unsigned positive number with negative values being reserved to indicate “no value”, or error conditions. Values 0, 1, and 2 are reserved for “standard input”, “standard output”, and standard error, respectively.

- **The in-core table:** The in-core table holds an in-memory copy of the i-node of each open file.

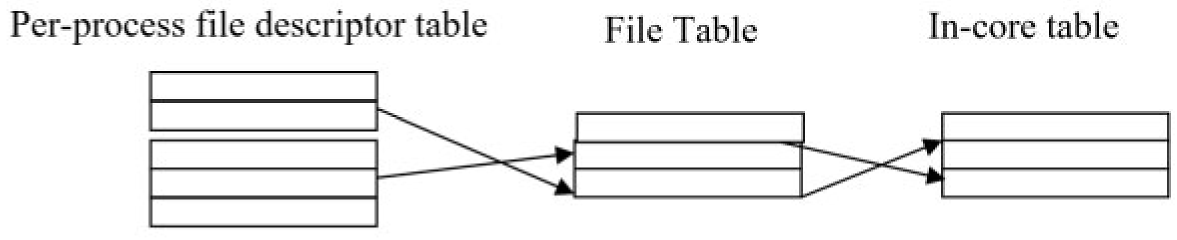

Figure 2.1: *Kernel data structures for files*

Whenever a file is opened in any process, a corresponding entry is made in the file table, the file's I-node table entry is copied to v-node table, and an entry is made in the per-process file descriptor table. When a process closes a file, the Kernel deletes the entry, which refers to the file from the per-process file table and decrements the reference count in the Kernel file table. If the reference count becomes zero, the Kernel file table entry is deleted, and the file is closed.

When a process is forked, a copy of its parent's per-process file table is created. The parent and child share the same file pointer.

On the other hand, the same file can be opened more than once by the same process or different processes. In that case, each open results in a different entry in the global file table. Each of these has its own current position pointer. Closing one has no effect on the others.

System call for UNIX file I/O

UNIX provides system calls for basic file I/O, such as for opening, creating, closing, and reading writing operations.

Kernel refers to all the opened files by file descriptors. The file descriptor is a unique integer number assigned to all opened files in the operating system. When an existing file is opened, or a new file is created, Kernel returns a file descriptor to the process. This file descriptor is further used for other file operations like read and write.

The open function

The **`open()`** is used to open an existing file for reading and writing. If the file does not already exist, it also creates a new file for further file operations.

Prototype:

The syntax for the open function is as follows:

```
int open (char*pathname, int oflag, int mode);
```

If the file is opened successfully, the **`open`** function returns a file descriptor for the file. (If not, the return value is –1.) All of the other system calls use a file descriptor (rather than its name) to refer to a file.

The first argument is the name of the file to be opened. **`oflag`** contains the following 1-bit flags to indicate how the file is to be opened:

- O_RDONLY /* open for reading only. */
- O_WRONLY /* open for writing only. */
- O_RDWR /* open for reading and writing. */
- O_APPEND /* open in append mode (initializes the position pointer to the end of the file instead of the beginning). */
- O_CREAT . /* create the file if it does not exist. */
- O_TRUNC /* if the file exists and is opened for either writing or read/write, truncate its length to 0. */

The third argument is used only if a file is created. It is used to set the permission bits for the new file.

Example

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
Static char msg = "greeting message";
int main()
{
 int fd;
 char buffer[80];
fd = open("dfile.dat", O_RDWR | O_CREAT | O_EXCL |
S_IREAD                 S_IWRITE);
if (fd != -1)
```

```
 {
  printf("datafile opened\n");
  write(fd,msg, sizeof(msg))
  close(fd);
 }
 else
  printf("data file already exists");
 exit(0);
 }
```

In the given example, a file is created to write a greeting message. If the file already exists, a message or data file that already exists will be printed.

The close function

The process closes a file when it is no longer required. Closeting file releases all locks on the file that the process may have. Kernel decreases the reference count of the file in the file table. If the reference count reaches to 0, then Kernel frees the file table entry and releases the in-core memory allocated at the time of opening the file.

When a process terminates, the Kernel automatically terminates all the opened files in that process.

Prototype:

The syntax for **`close`** function is as follows:

```
 int close(int fd);
```

Where **`fd`** is the file descriptor for the file. It returns **`0`** on success and **`-1`** on error.

The creat function

The **`creat()`** system call creates a new empty file in the file system. It creates a file table entry. If a file with the same name already exists, Kernel releases all the data blocks and sets file size equal to 0 subject to the suitable access permission. Kernel assigns a new I-node to a newly created file and creates an entry in the parent directory.

Prototype

```
 int creat(char*pathname, int mode);
```

Equivalent to:

```
open(pathname, O_RDWR | O_CREAT | O_TRUNC, mode);
```

The **`creat`** function opens the newly created file in write mode only. It sets and returns the first unused file descriptor. On failure, it returns **`-1`**.

Example:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
  int main()
    {
     int fd;
     fd = creat("datafile.dat", S_IREAD | S_IWRITE);
if(fd != -1)
      {
      printf("datafile.dat opened for read/write access\n");
      }
else
      printf("Error in opening datafile.dat\n");
  close(fd);
  exit(0);
  }
```

In the given example, a file is created for reading and writing access both. The newly created file is always opened in WRITE mode. If the error occurs, the program prints the message **`Error in opening datafile.dat`**.

The read function

Read system call is used to read data from an opened file. If the read is successful, the number of bytes read is written. If the end of the file is encountered, 0 is written. To begin reading from a file, Kernel sets various I/O parameters such as I/O mode, a flag to indicate that the I/O will go to user address space, a count field to indicate the number of bytes to read, the target address of the user data buffer, and offset field to indicate the byte offset into the file where the I/O should begin.

Prototype

```
ssize_t read(int fd, void *buff, size_t nbytes);
```

Example

```
#include <cntl.h>                /* defines options flags */
#include <;sys/types.h>      /* defines types used by
sys/stat.h */
#include <sys/stat.h>          /* defines S_IREAD &amp;
S_IWRITE */

  static char message[] = "File programming in C";
  int main()
  {
   int fd;
   char buffer[80];

  fd = open("datafile.dat",O_RDWR | O_CREAT | O_EXCL, S_IREAD
  | S_IWRITE);
   if (fd != -1)
    {
    printf("datafile.dat opened for read/write access\n");
    write(fd, message, sizeof(message));
    lseek(fd, 0L, 0);     /* go back to the beginning of the
    file */
    if (read(fd, buffer, sizeof(message)) == sizeof(message))
     printf(" Contents of the datafile are %s\n", buffer);
    else
     printf("Error in reading datafile\n");
    close (fd);
    }
   else
    printf("\n File already exists\n");
 exit(0);
}
```

In the given example, it first creates a file both for reading and writing. If the file is already existing with the same name, it prints a message **`File already exists`**. Once the file is created, the message **`File programming in C`** is written into it. In the next step, the file pointer is set to the beginning of the file, and the read operation is performed. It checks if the contents read are the same as the message; it prints the message; otherwise, an error message is printed.

The write function

The **`write`** function writes data from the buffer to the file. The number of bytes to be written is given as the third argument to the system call. On success number of bytes written is returned. On error `-1` is written.

If the value of the third argument is zero and the file descriptor refers to a regular file, 0 may be returned, or an error could be detected. For a special file, the results are not portable.

Prototype:

```
int write(int fd, void*buffer, int nbytes);
```

Where:

- First argument: **`fd`** represents file descriptor.
- Second argument: Temporary buffer contains data to write on the file.
- Third argument: Size of data in bytes.

The lseek function

The **`lseek()`** function sets the current position of the file pointer. If successful, **`lseek`** returns the new position; else, it returns `-1`.

Prototype

```
int lseek(int fd, int offset, int whence);
```

Where **`fd`** is the file descriptor of the file in which the file pointer will be set; offset is the number of bytes by which the pointer will be adjusted; hence, maybe one of the following:

- **`SEEK_SET`** offset is from the beginning of the file.
- **`SEEK_CUR`** offset is from the current position.
- **`SEEK_END`** offset if from the end of the file.
- Whence maybe **`0`**, **`1`**, or **`2`**, respectively, for **`SEEK_SET`**, **`SEEK_CUR`**, and **`SEEK_END`**.

Example

```
#include <stdio.h>
#include <fcntl.h>
   int main()
   {
     int fd;
     long position;

     fd = open("datafile.dat", O_RDONLY);
     if ( fd != -1)
      {
      position = lseek(fd, 0L, 2);  /* seek 0 bytes from end-
      of-file */
      if (position != -1)
       printf("The length of datafile.dat is %ld bytes.\n",
       position);
      else
       perror("lseek error");
      }
     else
   printf("can't open datafile \n");
   close(fd);
}
```

In the given example, a file is opened in read-only mode. On error, a message **`can't open datafile`** will be printed. Once the file is opened successfully, the file pointer is set to the End of the File.

The link() function

The **`link()`** system call links an existing file to a new one. Linking a file in the UNIX system creates a new directory entry (hard link) that contains the linked filename and the I-node number of the existing file. All linked files refer to the same data. Changing content in one file is reflected in all links. In a simple way, we can say that it makes a new name for a file.

Prototype

```
int link(char*original_name, char * new_name)
```

On success, it returns **`0`**, and on error **`-1`** is returned. The **`link()`** system call fails if any of the following conditions exist:

- **`Original_name`** does not exist.
- **`New_name`** does exist.
- **`Original_name`** is a directory, and the user is not a superuser.
- Try to create a link across the file system.

Example

```
 #include <stdio.h>
   int main()
   {
    if ((link("file.old", "file.new")) == -1)
     {
     perror("ERROR");
  exit(1);
    }
 exit(0);
 }
```

In the given example, a new name file. New is created for a file. If an error occurs, it prints **`ERROR`**.

The unlink system call

The unlink system call removes a hard link and corresponding directory entry and decreases the link count of the file. If the link count reaches 0, the contents of the file are also deleted. If the file is opened by any process, the contents of the file are not deleted till it is closed. When the process closes the file, the Kernel removes the contents of the file.

Prototype

```
 int unlink(const char *pathname)
```

Where **`pathname`** is the name of the linked file which is to be removed. For **`unlink()`** to be successful, the directory containing the corresponding directory entry must have write and execute permission. On success, **`0`** is returned, and on error, **`-1`** is returned.

Example

```
  #include <stdio.h>
   int main()
   {
     if ((unlink("file.old")) == -1)
       {
       perror("ERROR ");
  exit(1);
}
exit(0);
}
```

In the given example, on successful unlinking, the directory entry of the file will be removed, and on error, an **`ERROR`** message will be printed.

The dup() system call

The **`dup()`** system call duplicates the file descriptor for a file, which points to the same file table entry as the old file descriptor does. The **`dup()`** system call generates the lowest available file descriptor. Both the file descriptors share the same file offset and status flags (read, write, and append).

Prototype

```
int dup(int fd)
```

This system call returns a new file descriptor for the file. Both file descriptors can be interchangeably used to edit the file.

Example

```
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
   int main()
   {
     int fd;
     fd = open("file.dat",O_WRONLY | O_CREAT, S_IREAD | S_IWRITE
     );
     if (fd == -1)
       {
       perror("FILE already opened");
       exit (1);
       }
     close(1);         /* close standard output  */
     dup(fd);       /* fd will be duplicated  */
     close(fd);        /* close the extra slot */
     printf("Hello\n");
   exit(0); }
```

In the given example, the file is created in write-only mode. If it already exists, an error occurs, and **`FILE already opened`** message is printed. The file is duplicated, and both the standard output and original file are closed afterward. The only opened file is the second one, and **`Hello`** message is printed on this.

The stat and fstat system calls

The **`stat()`** and **`fstat()`** system calls give the status of files. These system calls return information such as file owner, i-node number, file type, file size, access permission, and file access time.

Prototype

```
int stat(const char *pathname, struct stat *buf);
int fstat(int fd, struct stat *buf);
int lstat(const char *oathname, struct stat *buf);
```

Where:

`buf` is the address of the data structure in the user process that contains the status information of the file. These system calls simply write fields of the i-node table into the **`buf`**.

The **`fstat`** system call to obtain information about the file that is already open on the descriptor **`fd`**. A Stat system call gives information about a file specified in the pathname; **`lstat`** system call is the same as a stat system call.

Stat structure contains following information:

```
dev_t     st_dev         /* Device ID of device containing
file. */
ino_t     st_ino          /*  File serial number.  */
mode_t    st_mode    /* Mode of file (see below). */
```

```
nlink_t   st_nlink       /* Number of hard links to the file.
*/
uid_t     st_uid          /* User ID of file.  */
gid_t     st_gid          /* Group ID of file.  */
     dev_t     st_rdev         /* Device ID (if file is
     character or block special). */
off_t     st_size         /* For regular files, the file size
in bytes.
        For symbolic links, the length in bytes of the
        pathname contained in the symbolic link.
        For a shared memory object, the length in bytes.
        For a typed memory object, the length in bytes.
        For other file types, the use of this field is
        unspecified. */
time_t    st_atime     /* Time of last access.  */
time_t    st_mtime   /* Time of last data modification. */
time_t    st_ctime    /* Time of last status change. */
blksize_t st_blksize /*A file system-specific preferred I/O
block size for
        this object. In some file system types, this may
        vary from file to file. */
blkcnt_t  st_blocks  /* Number of blocks allocated for this
object. */
```

BASIC file permission

In UNIX, each file can be accessed by different users as per the permission assigned to them. Each user is identified by a unique number, known as **`User_Id`**. The user who has created the file is the owner of the file. Users can also belong to a group. All members of the group share the same file permissions. Users who are neither owners nor belong to the group are known as others. The file owner can provide the file access permissions to the group members and other users. The file access permissions may read (r), write (w), or execute (x). the ls –l command may be used to see the assigned access permission given to the various files in the directory.

Example

```
>ls -l /usr/bin/new1
```

```
-rwxr-xr-x    1 user1   use1    8632 Sep 19  2021 /usr/bin/new1
```

In the given example, the file **`/usr/bin/new1`** file is owned by user1. The file belongs to the group user. **`-rwxr-xr-x`** shoes the file access permission and type of file. The first letter shows that the file type is regular. The next three letters show permission assigned to the owner. The owner can read, write, and execute the file. The next three-letter show permission given to the group members, and the last three letters show the permission given to the others. The group members and others can only read and execute the file.

The file access permission is represented as a 9-bit vector, where the first 3 bits are allocated to the owner, the next three bits are allocated to group members, and the last three bits are allocated to other users. Thus, **`rw-`** corresponds to 110, that is, 4+2+0 = 6 in decimal. Where r-bit corresponds to decimal 4, the w-bit to decimal 2, and the *x*-bit to decimal 1.

`rwxr-xr-x` means 755 in decimal for **`/usr/bin/new`**.

Each user in UNIX has different access permission to their files. Every user has a user-Id, a unique number that identifies her/him. Users also belong to one or more groups. Groups can be used to restrict access to a number of people. Access permissions can be set per file for the owner, group, and others on the basis of read (r), write (w), and execute permissions (x). The command **`ls -l`** is used to see these permissions.

```
>ls -l /usr/bin/new -rwxr-xr-x    1 root   root  8632 Sep 9
2008 /usr/bin/new
```

The file **`/usr/bin/new`** is owned by the user root and belongs to a group called root. The -rwxr-xr-x shows the file access permissions. This file is readable(r), writable (w), and executable (x) for the owner. For the group and all others, it is readable(r) and executable (x).

The permissions are represented as a bit vector with 3 bits each for the owner, group, and others. Thus *r-x* corresponds to 101 as a bit pattern or 4+1=5 in decimal. The r-bit corresponds to decimal 4, the w-bit to decimal 2, and the x-bit to decimal 1.

rwx 421 user (owner)	rwx 421 group	rwx 421 others

Real and effective user-IDs and group-IDs

Each user name and a group name are mapped to a unique unsigned number, called user and group ID. This mapping is done via the /etc/passwd and /etc/group files, respectively. The user and group ID 0 are commonly called root, but that is really just a convention.

Each UNIX process has a user ID and a group ID associated with it, and when trying to open a file for writing, for instance, these IDs are used to determine whether the process should be granted access or not. These IDs constitute the effective privilege of the process because they determine what a process can do and what it cannot. Most of the time, these IDs will be referred to as the effective `uid` and `gid`.

Set user-ID

Normally, a program executes under the privileges of the normal user. But there are situations where normal users require special privileges in order to perform some special tasks such as changing passwords. A hashed representation of each user's password is stored in a /etc/passwd file, which has write permission to superusers only. When a normal user wants to change his or her password, this file must be updated. But a normal user does not have write and save permissions for this file. So the program that changes the password must be running on behalf of the normal user but with the privileges of the superuser. For such situations, a special bit (set-user-ID or SUID) is set on the file permissions. When this bit is set, the program it is applied to does run with the privileges of the file owner. Doing a long file listing, this bit will show up as "s" instead of an "x" in the owner permissions. A lowercase s means both the SUID bit and the execute bit are set. An uppercase *S* represents a SUID bit without the execute bit.

The SUID bit can also be set on directories. When set on a directory, all the files and directories created within this directory will have the same owner as of the SUID directory itself, no matter who has created these files.

Set-group ID

If the user executes a program with this bit set. He/she inherits the file access permission assigned to the group of the owner of the program. The SGID bit is represented with a lowercase or uppercase "s" in place of "x" for the group execute permission bit. This bit is generally set for utilities, such as mail utility, print utility, and so on. If the SGID bit is set for a directory, then all subdirectories and files created in it also inherit the group permission.

When the program is executing with this bit set, the user will automatically inherit the privileges of the group owning the program. Just like the SUID bit, the SGID bit is represented with a lowercase or uppercase “s” in place of “x” for the group execute bit. This usually is used for various utilities of some subsystems, such as the mail subsystem or the printing subsystem, and so on. An SGID bit on a directory means that all files and subdirectories created in the directory inherit the group of the directory.

Sticky-bit permission

The sticky bit is primarily used on public directories. It is useful for shared directories such as **`/var/tmp`** and **`/tmp`**. All the users should be able to create files, read, and execute files of other users in these temporary directories. If the sticky bit is applied to a directory, a file in that directory can only be deleted or renamed if the user has to write permission to the directory itself, and in addition to this, he is either the owner of the file, the owner of the directory itself or the superuser (root). Users are not allowed to remove files owned by other users. So the sticky bit makes files stick to the owner and prevents users to delete and rename other users’ files in publicly writable directories.

User and group IDs of new files

The user ID of a new file is set to the effective user ID of the process of creating the file. The group ID of a new file is either:

- The effective group ID of the process of creating the file.
- The group ID of the directory in which the file is created.

Effective user and group IDs

For most processes, Effective User ID (EUID) is the same as User ID (UID), and Effective Group ID (EGID) is the same as Group ID (GID). When the executable file that contains the image that is executed by the process has the Set-UID or Set-GID bit set, then the process’ EUID/EGID is set to the UID/GID of the executable file, not of the user executing the process! This is UNIX’s mechanism to grant users well-defined and limited access to a resource that would otherwise be inaccessible. For example, changing the password requires writing to **`/etc/passwd`**. Ordinary users, of course, do not have write access to this file. But the Set-UID root program /bin/passwd provides a well-defined gateway.

File Access Permission Mask

The **`st_mode`** member of stat structure contains file permissions:

```
st_mode mask - owner, group, other
#define S_IRWXU 00700 /* read, write, execute: owner */
#define S_IRUSR 00400 /* read permission: owner */
#define S_IWUSR 00200 /* write permission: owner */
#define S_IXUSR 00100 /* execute permission: owner */
#define S_IRWXG 00070 /* read, write, execute: group */
#define S_IRGRP 00040 /* read permission: group */
#define S_IWGRP 00020 /* write permission: group */
#define S_IXGRP 00010 /* execute permission: group */
#define S_IRWXO 00007 /* read, write, execute: other */
#define S_IROTH 00004 /* read permission: other */
#define S_IWOTH 00002 /* write permission: other */
#define S_IXOTH 00001 /* execute permission: other */
```

User file creation mode—umask() call

The UNIX uses a four-digit octal number **`umask`** to determine the file permission for newly created files. Every process has its own **`umask`**, inherited from its parent process.

The **`umask()`** sets the process's file creation mask and returns the previous value of the mask. The low-order 9 bits of mask are used whenever a file is created. To set default access permission for regular file

UNIX Kernel sets the default permission 666 for a regular file at the time of the creation and 777 for a directory. When the file or a directory is created using **`open()`**, **`mkdir()`**, and other system calls, the UNIX kernel subtracts the set umask value from the default access permission to set the resulting access permission for a file or directory.

Example

Let the set umask value is 044. To assign access permission to a regular file that has default access permission 666, the Kernel subtracts 044 from 666 (666-044) and sets 622 as the resulting access permission.

Clearing corresponding bits in the file access permissions restricts the default access to a file.

Prototype

```
Mode_t umask(mode_t cmask)
```

The **`cmask`** argument is formed as bitwise OR of any of the nine constants from **`st_mode`** mask.

Example

```
 #include <sys/types.h>
 #include <sys/stat.h>
main()
  {
   int fd;
   int old_mask;
   old_mask = umask(0);
   fd = open("f1", O_WRONLY | O_CREAT | O_TRUNC, 0666);
   close(i);
   printf("created f1: 0666\n");
   fd= open("f2", O_WRONLY | O_CREAT | O_TRUNC, 0200);
   close(i);
   printf("created f2: 0200\n");
   umask(022);
   fd = open("f4", O_WRONLY | O_CREAT | O_TRUNC, 0777);
   close(i);
   printf("created f4: %o\n", 0777 & ~022 & 0777);
   fd = open("f5", O_WRONLY | O_CREAT | O_TRUNC, 0200);
   close(i);
   printf("created f5: %o\n", 0200 & ~022 & 0777);
  }
```

In the given example, umask is set to 0, so all files created will take access permission given in **`open()`** function. The umask is set to 022. To set final access permission, the umask will be subtracted from the access permission in the open() function.

The access () function

The **`access()`** system call checks the accessibility of the file named by the pathname argument for the access permissions indicated by the mode argument. The value of mode is either the bitwise- inclusive OR of the access permissions to be checked (**`R_OK`** for read permission, **`W_OK`** for write permission, and **`X_OK`** for execute/search permission) or the existence test (**`F_OK`**).

Prototype

```
int access (const char *pathname, int mode);
```

Upon successful completion, the **`access()`** returns value `0`; otherwise, it returns value `-1`, and the global variable **`errno`** is set to indicate the error.

Changing owner and changing mode

Changing the owner and mode of a file are operations on the i-node. The process owner executing system calls to change owner or mode must be either superuser or owner of the file.

Prototype

```
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
int chown(const char *pathname, uid_t owner, gid_t group);
```

After the change of ownership, the old owner losses owner access rights to the file. The mode is bitwise OR of constants in **`st_mode`** member of the stat structure.

Mounting and unmounting a file system

The mount system call connects the file system in a specified partition of a disk to the existing file system hierarchy. The mount system call thus allows users to access data in a disk section as a file system instead of a sequence of disk blocks.

Prototype

```
int mount (const char * source , const char * target , const
char * filesystemtype , unsigned long mountflags , const void
```

```
 * data );
```

The **`mount()`** attaches the filesystem specified by the source (which is often a device name but can also be a directory name or a dummy) to the directory specified by the target.

The umount system call disconnects a file system and frees the hierarchy.

Prototype

```
 int umount(const char *target);
```

The **`umount2()`** removes the attachment of the (topmost) filesystem mounted on the target.

Directory related system calls

Like file-related system calls, the UNIX Kernel also defines a set of system calls for performing directory-related operations as in UNIX; everything is considered as a file. These system calls are used to create, remove, and manipulate directory contents.

The mkdir system call

The **`mkdir()`** function is used to create a new directory named by the pathname pointed by the path.

Prototype

```
 int mkdir(const  char *path, mode_t mode);
```

The mode of the new directory is initialized from mode. The protection part of the mode argument is modified by the process's file creation mask.

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID, or if the **`S_ISGID`** bit is set in the parent directory, then the group ID of the directory is inherited from the parent. The **`S_ISGID`** bit of the new directory is inherited from the parent directory.

If path names a symbolic link, **`mkdir()`** fails and sets **`errno`** to **`EEXIST`**.

The newly created directory is empty with the exception of entries for itself (.) and its parent directory (..).

Upon successful completion, the system call returns **`0`**. Otherwise, it returns **`-1`**, no directory is created, and **`errno`** is set to indicate the error.

The rmdir system call

The **`rmdir()`** system call to remove a directory file whose name is given by path. The directory must not have any entries other than "." and "..".

Prototype

```
int rmdir(const char *path);
```

The **`rmdir()`** function returns the value **`0`** if successful; otherwise, it returns a value **`-1,`** and the global variable **`errno`** is set to indicate the error.

If the link count of the directory becomes 0 with this call and no other process has the directory opened, then the space occupied by this directory is freed.

The chdir() system call

The **`chdir()`** system call is used to change the current directory pointed by the argument path. The process executing **`chdir()`** system call must have to execute access permission on the directory.

Prototype

```
int chdir(const char *path);
```

Upon successful completion, **`chdir()`** returns a value of **`0`**. Otherwise, it returns a value of **`-1`**, and **`errno`** is set to indicate the error.

Standard I/O library in UNIX

In UNIX standard I/O library is used to provide a stream I/O interface to the program. The stream I/O interface is the simple and efficient way to provide input to the program and write the produced output.

Three streams, standard input, standard output, and standard error, are by default opened and provided to the program whenever it starts executing.

Stream and FILE object

Streams are a portable way of reading and writing data.

A stream can be a file or a physical device (for example, a printer or monitor). All streams are defined by an internal C data structure, FILE. The FILE data structure is declared in `stdio.h`, which is manipulated with a pointer to the stream. To perform I/O with stream, it is simply required to refer to the FILE structure in C programs.

A stream must be opened before doing any I/O to access it, and it must be closed after use.

I/O buffering

Buffering is used to minimize the number of read and write calls. I/O buffering can be of basically three types, as discussed in the following sections.

Fully/blocked buffered

When a stream is a block buffered, many characters are saved up and written as a block. Normally, all files are block buffered. Actual I/O occurs when the buffer is filled up. When the first I/O operation occurs on a file, malloc(3) is called, and an optimally sized buffer is obtained.

The term flush describes the writing of a standard I/O buffer. A buffer can be flushed automatically by the standard I/O routines (such as when a buyer fills) or by using the function `fflush` explicitly.

Standard input and standard output are fully buffered streams unless they are not referred to as interactive devices.

Line buffered I/O

When streams are line-buffered, characters are saved up until a newline (`\n`) is output or input is read from any stream attached to a terminal device (typically stdin). This allows us to output a single character at a time (for example, with `fputc`), knowing that actual I/O will take place only when we finish writing each line. Line buffering is typically used on a stream when it refers to a terminal (for example, standard input and standard output).

Unbuffered I/O

The standard I/O library does not buffer the characters. When an output stream is unbuffered, information appears on the destination file or terminal as soon as written. For instance, by using the discussed write(2) call, the standard error stream is normally unbuffered. Any error messages are displayed as quickly as possible (regardless of whether they contain a newline or not).

Stream buffering operations

UNIX provides buffer operation to buffer information to avoid frequent access to external storage devices. The buffering makes data reading and writing less time-consuming.

setbuf(3), setvbuf(3)

Functions must be called after the stream has been opened and before any other operation is performed on the stream. The **`setbuf(3)`** is used to turn to buffer on and off.

Prototype

```
void setbuf ( FILE * stream , char * buf );
```

To enable buffering, **`buf`** must point to a buffer of length **`BUFSIZ`**. To disable buffering, **`buf`** is set to NULL.

`setvbuf()` may be used to alter the buffering behavior of a stream.

Prototype

```
int setvbuf ( FILE * stream , char *buf , int mode , size_t
size );
```

The **`mode`** parameter must be one of the three macros following. The **`size`** parameter may be given as zero to obtain deferred:

```
# define _IOFBF 0 /* setvbuf should set fully buffered */
# define _IOLBF 1 /* setvbuf should set line buffered */
# define _IONBF 2 /* setvbuf should set unbuffered */
```

Flushing a stream

When **fflusg()** function is called, all data buffered for given output is forcefully written to the **FILE** object pointed out by stream argument. The steam is remained open. Stream argument is set to NULL, and then all opened output streams are fflushed. The second function **fpurge()** deletes all input or output buffered in the given stream

The **fflush()** forces a write of all buffered data for the given output or update stream via the stream's underlying write function. The open status of the stream is unaffected. If the stream argument is NULL, **fflush()** flushes all open output streams. **fpurge()** erases any input or output buffered in the given stream.

```
int fflush ( FILE * stream );
int fpurge ( FILE * stream );
```

Opening a stream

The **fopen()** function is used to open the file whose name is given in the string pointed to by pathname and associates a fully buffered stream with it. It also clears the error and end-of-file indicators for the stream.

Prototype

```
FILE * fopen ( const char *path , const char * mode );
```

The **freopen()** function first attempts to flush the stream and close any file descriptor associated with a stream, but the success or failure to flush or close the file descriptor is ignored. It clears the error and end-of-file indicators for the stream.

The **freopen()** function opens the file whose name is in the string pointed to by pathname and associates the stream pointed to by stream with it. The mode argument shall be used just as in **fopen**.

Prototype

```
FILE * freopen ( const char *path , const char *mode , FILE *
stream );
```

The **fdopen()** function associates a stream with a file descriptor.

Prototype

```
FILE * fdopen ( int fildes , const char * mode );
```

Following are the different modes to open a stream:

Mode	Description
R	Opens file for reading
r+	Opens for reading and writing
W	Truncates file to zero length or creates text file for writing
w+	Opens for reading and writing. The file is created if it does not exist; otherwise, it is truncated.
a	Opens for writing. The file is created if it does not exist.
a+	Opens for reading and writing. The file is created if it does not exist.

Table 2.1: *Stream opening modes*

Following are the restrictions on opening a stream:

Restriction	r	w	a	r+	w+	a+
File must already exist	x			x		
Previous contents of file are discarded		x			x	
Stream can be read	x			x	x	x
Stream can be written		x	x	x	x	x
Stream can be written only at end			x			x

Table 2.2: *Opening restrictions on a stream*

Closing a stream

The **`fclose()`** function closes a file and disassociates the named stream from it. If the stream was being used for output, any buffered data is written first, using **`fflush(3)`**. If the standard library had automatically allocated a buffer, that buffer is released.

Prototype

```
int fclose (FILE * stream);
```

Reading and writing a stream

Once a stream is opened, three different types of unformatted I/O can be performed:

- Character-at-a-time: Read and write one character at a time, with the standard I/O functions handling all the buffering (if the stream is buffered).
- Line-at-a-time: To read or write a line at a time using **`fgets(3)`** and **`fputs(3)`** functions. Each line is terminated with a newline character.
- Direct: The **`fread(3)`** and **`fwrite(3)`** functions read or write some number of objects, respectively, where each object is of specified size.

Reading one character at a time

The following are the prototypes for reading a character:

```
int fgetc ( FILE * stream );
int getc ( FILE * stream );

int getchar ( void );
int getw ( FILE * stream );
```

Here, **`fgetc()`** function takes the next input character (if present) from the stream pointed at by the stream argument or the next character pushed back on the stream via **`ungetc(3)`**. **`getw()`** takes the next int (if present) from the stream pointed at by stream.

Writing one character at a time

The following are the prototypes for writing a character:

```
int fputc ( int c, FILE * stream );
int putc ( int c, FILE * stream );

int putchar ( int c);
int putw ( int w, FILE * stream );
```

`fputc()` writes one character (converted to an unsigned char) to the output stream pointed to by a stream such as a monitor. It may evaluate the stream more than once, so arguments given to **`putc()`** should not be expressions with potential side effects. **`putw()`** writes the specified integer to the named output stream.

Reading one line-at-a-time

The following are the prototypes for reading a line:

```
char * fgets ( char *str , int size , FILE * stream );
char * gets ( char * str );
```

The **`fgets()`** and gets() functions are used to read line from the stdin or specified stream, respectively, into the buffer.

Writing one line at a time

The following are the prototypes for writing a line:

```
int fputs ( const char *str , FILE * stream );
int puts ( const char * str );
```

The **`fputs()`** and **`puts()`** are used to write the string pointed to by **`str`** to the stream and **`stdout`**, respectively.

Direct I/O

Direct I/O functions are used to read and write objects from/on the **`FILE *stem`** object. The size of the object is given by the **`size`** parameter.

Prototypes

```
size_t fread ( void *ptr , size_t size , size_t objs , FILE *
stream );
size_t fwrite ( const void *ptr , size_t size , size_t objs ,
FILE * stream );
```

Here, the **`fread()`** reads a number of objects from the stream pointed to by stream and stores them at the location given by **`ptr`**. The size of each object read is given by **`size_t`** in bytes.

The **`fwrite()`** writes the objects stored at the location pointed by **`ptr`** to the stream pointed to by **`stream`** argument.

Standard data files

UNIX maintains some standard data files to store data to manage users' file access permissions. These files can only be accessed with root privileges. These files contain information corresponding to each user, group, and life span of the password. All these files are stored `in/etc` directory.

/etc/passwd file

The **/etc/passwd** file is an ASCII file that contains an entry for each user. Each entry defines the basic attributes applied to a user. The **/etc/passwd** file is updated when a new user is added to the system using mkuser command.

An entry in the **/etc/passwd** file has the following form: Attributes in an entry are separated by a: (colon).

```
Name:Password: UserID:PrincipleGroup:Gecos:
HomeDirectory:Shell
```

The are defined as follows:

Name	Specifies the user's login name. The login name must be a unique string of 8 bytes or less containing numerals. Alphabet and special characters.
Password	An **x** character indicates that the encrypted password is stored in **/etc/shadow** file.
UserID	Specifies the user's unique numeric ID. The User ID is a decimal integer.
PrincipleGroup	Specifies the user's principal group ID. This must be the numeric ID of a group in the user database or a group defined by a network information service. The value is a unique decimal integer.
Gecos	Specifies general information about the user that is not needed by the system.
HomeDirectory	Specifies the full pathname of the user's home directory. If the user does not have a defined home directory, the home directory of the guest user is used. The value is a character string.
Shell	Specifies the initial program or shell that is executed after a user invokes the **login** command or **su** command. If a user does not have a defined shell, **/usr/bin/sh**, the system shell, is used. The value is a character string that may contain arguments to pass to the initial program.

Table 2.3: *File attributes with description*

/etc/group file

The **/etc/group** file is an ASCII file that contains records for system groups. Each entry contains basic group attributes. Each entry has the following format:

```
Name:Password:ID :User-List
```

Attributes are separated by a colon. Records are separated by newline characters. The attributes in a record have the following values:

`Name`	Specifies a group name that is unique on the system. The name is a string of 8 bytes or less.
`Password`	Not used.
`ID`	Specifies the group-ID. The value is a unique decimal integer string.
`User-List`	Identifies a list of one or more users. Each user is separated by commas. Each user must already be defined in the local database configuration files.

Table 2.4: *Group's Attributes with description*

/etc/shadow file

The **`/etc/shadow`** file stores the actual password in encrypted format for the user's account with additional properties related to the user password. It contains one entry per line for each user listed in **`/etc/passwd`** file, generally. Each entry has the following format:

```
smithj:Ep6mckrOLChF.:10063:0:99999:7:::
```

Each field is separated by a colon (:) symbol:

User name	Specifies user's login name.
Password	Specifies encrypted password. The password must be minimum 6–8 characters long, including special characters/digits.
Last password change (**`lastchanged`**):	Days since Jan 1, 1970, the password was last changed.
Minimum	The minimum number of days required between password changes, that is, the number of days left before the user is allowed to change his/her password.
Maximum	The maximum number of days the password is valid (after that user is forced to change his/her password).
Warn	The number of days before the password is to expire the user is warned that his/her password must be changed.
Inactive	The number of days after the password expires, that account is disabled.

Expire	Days since Jan 1, 1970, that account is disabled, that is, an absolute date specifying when the login may no longer be used.

Table 2.5: *Fields in shadow record with description*

The last six fields provide password aging and account lockout features (you need to use the change command to set up password aging). According to the man page of shadow—the password field must be filled. The encrypted password consists of 13–24 characters from the 64 character alphabet a through z, A through Z, 0 through 9, \. and /. Optionally it can start with a $ character. This means the encrypted password was generated using another (not DES) algorithm. For example, if it starts with 1 it means the MD5-based algorithm was used.

Conclusion

UNIX provides a number of methods to perform input/output operations. The two most dominating methods are I/O system calls and library functions. The library function, when executed at the Kernel level, they are also converted into a corresponding system call. UNIX Kernel maintains various data structures to manage all open files. How to access the file by user's process is controlled by file access permission provided to the user by the administrator. To manage file access, users are divided into three categories, namely, owner, group members, and others. The owner can provide access to all other types of users. All group members share the same access permissions. UNIX also provides library functions to buffer data for both reading and writing. I/O buffers are maintained to avoid frequent access to external devices and to speed up input and output operation. After completing the two major components, process and file-management, in Chapters 1 and 2, respectively, the upcoming chapter will discuss the UNIX shell environment and provide the user interface for functioning with processes and files.

Review Exercise

1. Implement setbuf-using setvbuf.
2. What happens if the file mode creation mask is set to 777?
3. How can you set only one of the two times?

4. Does UNIX have a fundamental limitation on the depth of a directory tree? To find out, write a program that
5. What exactly is stored in data block of the UNIX File system?
6. Can two processes open a file at a time? Justify your answer.
7. What is password aging?
8. Write a program that gives the user the opportunity to remove any or all of the files in a current working directory. The name of the file should appear, followed by a prompt as to whether it should be removed.

CHAPTER 3
Process Management

Introduction

The Kernel performs various primitive operations on behalf of user processes. These operations include controlling the execution of processes by allowing their creation, termination, or suspension and communication. Scheduling processes fairly for execution on the CPU. Allocating main and secondary memory for executing processes. Protecting processes' address spaces. The Kernel also controls access to peripheral devices such as terminals and other devices. UNIX maintains a parent–child relationship among different processes executed in the system. The UNIX Kernel maintains this process tree through different process IDs to keep control of process creation, execution, and termination. This chapter discusses all these primitive operations to perform effective process management.

Structure

We will cover the following topics in this chapter:

- Processes and their relationships
- Processes related operations
- Process control and execution
- Memory allocation to the executing process.
- Process communication

Objectives

After going through this chapter, you will be able to:

- Understand fundamental concepts of process management
- Learn how processes are identified in the UNIX system

- Get to know the process related operations such as creation, execution, termination, and suspension
- Understand how processes communicate with each other
- Understand how UNIX controls process execution and switching

UNIX process

A program in execution is a process. In UNIX, a process is a unit of work. The system consists of a collection of processes: operating system processes executing system code and user processes executing user code. All the processes execute concurrently with the CPU switching between the processes.

A process executes by following a strict sequence of instructions that is self-contained and does not jump to that of another process.

Process IDs

Each process has an identification number called process id that identifies it uniquely. The **`getpid()`** function is used to get the process id of a process. Processes in UNIX have a parent–child relationship. Each process has one parent and one or more children.

The process ID 0 is the scheduler process and is called swapper. Swapper is part of the Kernel and is called a system process.

The init process (process ID 1) is the process dispatcher, which gives birth to the shell, and all processes initiated by us are children of the shell and so descendants of init process. Init is a normal user process and never dies.

UNIX keeps track of all processes in an internal data structure called a process table. The process ID of a parent can be obtained by the function **`getppid()`** as follows:

```
main()
{
 printf("In child process pid is %d \n",getpid());
 printf("In child process parents pid is %d \n",getppid());
}
```

Executing process in UNIX environment

Process in UNIX is a C program. Whenever a C program executes, it calls the main function.

Prototype:

```
 int main(int argc, char * argv);
```

Here, **`argc`** is the number of command-line arguments, and **`agrv`** is the pointer to command line arguments.

The Kernel starts executing a C program through the exec function, which calls a special start-up routine specified as starting address by the executable program file. The starting routine takes command-line arguments from the Kernel and calls the main function.

Modes of execution of a process

There are the following two modes of execution of the system as seen by the hardware:

1. User mode: When the user process executes, it does so in the user mode. In the user mode, processes can access their own instructions and data but not Kernel instructions and data or those of other processes.
2. Kernel mode: When a user process executes a system, call the execution mode of the process changes from user mode to Kernel mode; the Kernel executes the users' request. Processes in Kernel mode can access Kernel and user addresses.

Process termination

When a process terminates, the operating system deallocates all resources allocated to the process, updates statistics related to the process, and intimate all processes about the process termination. UNIX process terminates either normally or abnormally. When a process terminates, it goes into a zombie state, that is, it does not release the process ID; rather, it waits for the parent process to acknowledge that child is terminated and PID is released.

Normal termination

Normal termination of the process is done by returning from the main function or calling **`exit()`** or **`_exit()`** functions.

The **`exit()`** function performs a cleanup function before exiting from the process. It closes all open files and flushes all buffers.

Prototype:

```
void exit(int status);
```

The **`_exit()`** function does not perform any cleanup operation before exiting.

Abnormal process termination

A process terminates abnormally if any or more of the following conditions occur:

- Resource usage of the process exceeds
- The task no longer needed
- If the parent process is exiting

Another reason for abnormal process termination is calling **`abort()`** function in some other process, especially by the parent. This function sends the SIGABRT signal to the caller.

For abnormal termination, the Kernel generates the termination status. The termination status is obtained by the parent process using the **`wait()`** system call.

Command-line arguments and environment variables

The **`main()`** function in C has two standard arguments **`argc`** and **`argv`** called command-line arguments. **`argc`** specifies the number of arguments counting the command name as an argument itself.

argv is an array of pointers to strings. The first member of argv, that is, **`argv[0]`**, points to the command name, and the second one **`argv[1]`**, points to the first argument of the command.

Because argv exists, the program can react to command line parameters entered by the user fairly easily. For example, you might have your program to detect the word help as the first parameter following the program name and dump a help file to stdout. File names can also be passed in and used in your open statements. Any input too can be passed as command-line arguments, and avoid using I/O statements to read input.

The **`envp`** gives the program's environment variables represented as an array of strings as **`VAR=value`**. For example, **`PATH=/bin;/usr/bin`**. The last element of the array is a null pointer. This gives C programmers access to the shell's global environment.

In addition to the **`envp`** vector, it is possible to access the environment variables through the call **`getenv()`**. This is used as follows; suppose we want to access the shell environment variable **`$HOME`**.

```
char *string;
string = getenv("HOME");
```

The **`getenv()`** gets the value of **`$HOME`** variable and stores it to the string.

Example:

```
/* myprog.c*/
#include <stdio.h>
int main(int argc, char *argv[], char*  envp[])
{
int I;
 printf("The program %s has %d arguments\n", argv[0], argc);
 printf("The first two arguments are %s and %s \n", argv[1],
 argv[2]);
 for (I=0; envp[I]; I++)
 printf("%s\n",envp[I]);
}
```

In the given example, the program is executed as follows:

```
$myprog this is good
```

The output is:

```
The program myprog has 4 arguments
The first two arguments are this and is
```

The **`argc = 4`** while **`argv[0]`** points to **`myprog`**, **`argv[1]`** points to **`this`**, **`argv[2]`** to **`is`** and **`argv[3]`** to **`good`**. After this, it displays the environment variables.

Memory layout of a UNIX process

The memory area allocated to a program will usually be split into several sub-areas for particular:

- **The code area:** This is known as the text area in UNIX and simply contains the executable code of the program. If there are several processes running the same program, there will still only be one code area as it is identical for all processes. The text area is usually read-only to prevent a program from modifying its instructions.
- **The data area:** This holds the data being processed by the program. This section contains both initialized and non-initialized data variables.
- **The stack area:** This is automatically created, and its size is adjusted at run time. It consists of logical stack frames that are pushed when calling a function and popped when returning. It contains parameters to a function, local variables, and data. A process has one stack for user mode and another for Kernel mode.
- **Heap:** This part of the virtual address space contains the dynamically allocated variables to be used by the process.

Setting branch into another function

C program does allow to jump to the label set in another function. To branch the control to another function **`setjump`** and **`longjump`** C library macros are used. Generally, these kinds of jumps should be avoided because it is not considered as a good programming practice.

Prototype

```
int setjmp(jmp_buf env); void longjmp(jmp_buf env, int val);
```

The **`setjmp()`** macro saves the current environment in the variable **`env`**. This macro returns more than once. The first time it always returns **`0`**; the second time, when it returns from **`longjmp()`**, it returns the value set for the second argument of **`longjmp()`**. Here it is represented by **`val`**.

Let us see an example:

```
include <setjmp.h>
#include <stdio.h>

jmp_buf ebuf;
void f(void);
int main(void)
{
 int i;
 printf("1 ");
 i = setjmp(ebuf);
 if(i == 0) {
  f();
 }
 printf("%d", i);
 return 0;
}
void f(void)
{
 printf("2 ");
 longjmp( ebuf, 3);
}
```

In the given example, the output is as follows:

```
1
2
3
```

The first time when **`setjmp()_`** is called, it returns **`0`** to the variable **`I`**, and the control jumps to the function **`f()`**. In function **`f()`** **`2`** will be printed, and the return value of **`setjmp`** is set to **`3`** as it is passed as the second argument of the **`longjmp()`** macro.

Process states

The lifetime of a process can be modeled by a set of states, and each process can be in any of the states at one point in time. These states can be shown by a state transition diagram shown in *figure 3.1*, which is a directed graph whose nodes represent the states a process can enter and whose edges represent the events that cause the process to move from one state to another.

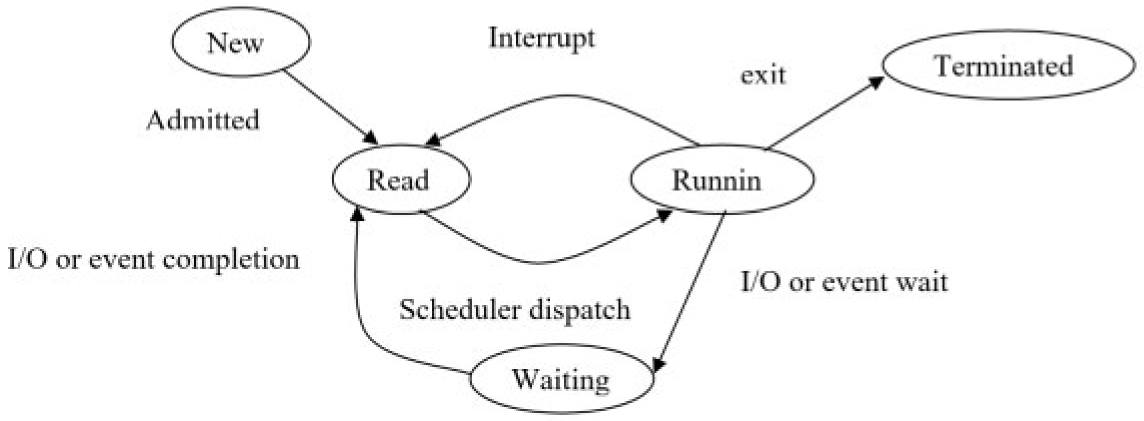

Figure 3.1: *Process transition diagram*

UNIX process undergoes into following different states throughout its lifetime:

- **Executing:** The process executes in two different modes user and Kernel mode.
- **Ready:** Process is ready to execute as soon as Kernel schedules it, or process is in a ready state in secondary memory.
- **Sleeping:** Process is in sleeping state in main memory, or swapper has swapped the process in the secondary memory to swap in other processes in the main memory.
- **Created:** Process is just created. It is neither in a ready state nor in a sleeping state.
- **Zombie:** Process has just executed the `exit()` system call. The process does not exist but leaves the exit code.
- **Returning to Kernel mode:** Process is preempted by Kernel until it is scheduling another process.

Process control block

Each process is represented in the operating system by a Process Control Block (PCB). This contains information associated with a process such as:

- **Process state:** New, running, waiting, and so on.

- **Program counter:** Indicates the address of the next instruction to be executed for this process.
- **CPU registers:** Includes accumulators, index registers, stack pointers, and general-purpose registers.
- **CPU scheduling information:** Like process priority, pointers to process scheduling queues, and other scheduling parameters.
- **Memory management information:** Like the value of base and limit registers, page tables.
- **Accounting information:** Like the amount of time for which the CPU is used.
- **I/O status information:** Like a list of I/O devices allocated to this process, a list of open files.

The context of a process includes the value of CPU registers, process state, and memory management information. The context of the process contains (1) User-Level context, (2) CPU registers, and (3) Kernel level context.

The user-level context contains the memory layout of the virtual address space of the process, such as text, data, user stack, and shared memory.

The CPU registers consist of the following information:

- The Program Counter contains the address of the next instruction to be executed.
- The Program Status Register contains various flags to indicate the result of the recent computation, the subfield that shows the current execution mode to determine whether the process can run in privileged instructions, and the subfields that show the current processor execution level.
- The Stack Pointer points to the next free entry or last used in the Kernel or user stack.
- The General-Purpose Registers contain intermediate data generated during the execution of the process.

The Kernel-level context of a process contains one static part and a variable number of dynamic parts throughout its lifetime. The components of system-level context are as follows:

- The Process Table entry contains state of process and control information.
- The u area contains process control information that needs to be accessed only once.
- Pregion entries, region table, and page table provide text, data, stack, and other regions of the process.
- Kernel Stack contains a stack of Kernel functions when the process executes in the Kernel mode. The Kernel stack is empty when the process executes in user mode.
- The Dynamic part contains several system-level context layers where each layer contains information about the previous layer.

Process control

Process control handles all operations related to the process, including the creation of the child process, changing context from one process to another while executing cooperating or independent processes, and terminating the process in a normal and normal manner.

Process creation

Processes are initiated in UNIX using the **`fork()`** function. The **`fork()`** function creates one child process identical to the parent, and both parent and child processes are running. Both parent and child share the same real and effective user and group ID, current working directory, root directory, file creation mask, and so on.

The **`fork()`** returns the value of the PID of the child to the parent, and the return value in a child is `0`. Kernel increments file and inode table counters for files associated with the process. If the child process cannot be created due to (**`A`**) there are too many processes in the system.

(B) limit on the number of processes for real-user ID of the parent process exceeds, then return value of parent process is **`-1`**.

Let us see an example:

```
main()
{
 int pid;
```

```
  pid = fork();
  if (pid < 0)
  printf("Fork failed \n");
  else
  {
  if (pid == 0)
  {
   printf("In child process pid is %d \n",getpid());
   printf("In child process parents pid is %d \n",getppid());
  }
  else
  {
printf("In parent process pid is %d \n",getpid());
   printf("In parent process parents parent pid is %d
   \n",getppid());
  }
 }
 }
```

File locks set in the parent process are not inherited in the child process. In the given example, if **`fork()`** is successful, it returns either **`0`** to the newly created child or the process ID of the newly created child to the parent process in which **`fork()`** is called. Both the processes execute one after another as scheduled by the CPU. If **`fork()`** is unsuccessful, it returns a negative number.

Awaiting process termination

The simple way of a process to acknowledge the death of a child process is by using the **`wait()`** system call. When **`wait()`** is called, the process is suspended until one of its child processes exits, and then the call returns with the exit status of the child process. If it has a zombie child process, the call returns immediately, with the exit status of that process.

A call to **`wait()`** function does a number of things. A check is made to verify if the parent process has any children. If it does not, a –1 is returned by **`wait()`**. If a parent process has a child that is terminated (zombie), the child's PID is returned, and it is removed from the process table.

However, if a parent process has a child or children that have not terminated, the parent process is suspended till it receives a signal. The signal is received as soon as the child dies.

The **`wait()`** can also tell us in what manner the child process was terminated. For this, we need to pass an integer variable to **`wait()`**. If the process was terminated normally, the high order 8 bits of the integer variable passed to the **`wait()`** will be updated, whereas the lower order 8 bits will be initialized to **`0`**.

On the other hand, if it has been terminated abnormally, the lower-order 8 bits are updated, and higher-order 8 bits are initialized to **`0`**.

In case of a core dump error, the wait returns an integer whose 7th bit is put on.

Let us see an example:

```
/* example of  wait() */
main()
{
 int i, pid, exitstat, status;
 pid = fork();
 if (pid == 0)
 {
 printf("Enter exit stat:");
 scanf("%d", &i);
 exit(i);
 }
 else
 {
 wait(&status);
 if ((status & 0xff) != 0) /* abnormal exit */
 {
  printf("Signal interrupted\n");
 }
 else /* normal exit */
 {
  exitstat = (int) status/256 ;
  printf("Exit status from %d was %d\n", pid, exitstat);
 }
```

```
   }
 }
```

Here in the child process, the user enters an exit code, and the process exits with this exit code. The parent process receives this exit code in the status variable. We can check the status variable to know the type of exit.

`fork()` creates a duplicate of all variables in the child process, so even global variables are duplicated. If we change the value in one process, it is not reflected in the other. Even the value of location pointed to by pointer variable is different for different processes.

Executing another program

An existing process can call another process by executing exec system call. The process ID of the running process does not change because **`exec()`** function does not create new process rather it calls an existing process. The various function prototypes of **`exec()`** function are as follows:

- `int execl(const char *path, const char *arg, …);`
- `int execlp(const char *file, const char *arg, …);`
- `int execle(const char *path, const char *arg,…, char * const envp[]);`
- `int execv(const char *path, char *const argv[]);`
- `int execvp(const char *file, char *const argv[]);`

`path` or **`file`** contains the filename being invoked, **`arg`** is a pointer to character string passed as an argument to invoked file. The first argument, by convention, should point to the filename associated with the file being executed. The list of arguments must be terminated by a NULL pointer.

The functions **`execlp()`** and **`execvp()`** will duplicate the actions of the shell in searching for an executable file if the specified filename does not contain a slash (/) character. The search path is the path specified in the environment by the PATH variable. If this variable is not specified, the default path **`:/bin:/usr/bin`**' is used. In addition, certain errors are treated specially.

The new process inherits the following properties from the old process such as process ID and parent process ID, real user ID and real group ID, session

ID, current working directory, root directory, file mode creation mark, and so on.

Effective IDs of the invoked process may change depending on the status of the set-user-ID and set-group-Id bits. If the set-user-ID bit of the new program is set, then the effective user-ID will become the owner ID of the program.

Accessing user information

There are a number of functions and system calls that let us access information about UNIX users.

User details

The passwd structure in the **`pwd.h`** header file contains the user information similar to **`/etc/passwd`** directory:

```
#include <pwd.h>
struct passwd {
  char *pw_name;  /* name of the user */
  char *pw_passwd;  /* password */
  uid_t pw_uid;  /* user id*/
  gid_t pw_gid;  /* group id */
  time_t pw_change; /* validity and
  char *pw_class;  password class */
  char *pw_gecos;  /* full user name */
  char *pw_dir;  /* login directory */
  char *pw_shell;  /* user login shell */
  time_t pw_expire; /* password expiry date*/
};
```

The **`passwd`** structure is returned by the **`getpwnam()`** and **`getpwuid()`** functions. It provides information about a user account. The **`getpwnam()`** and **`getpwuid()`** function searches the user database for an entry with a matching name or UID.

The **`getuid()`** gets the user id of the logged-in user.

Example of a program that prints details about the user is as follows:

```
#include <pwd.h>
```

```
main()
{
  struct passwd *pass;
  int uid;
uid = getuid();
  pass = getpwuid(uid);
  printf("Login name: %s\n", pass->pw_name);
  printf("Encrypted Password: %s\n", pass->pw_password);
  printf("User ID: %d\n", pass->pw_uid);
  printf("Group ID: %d\n", pass->pw_gid);
  printf(" Password Age: %s\n", pass->pw_age);
  printf("Comment: %s\n", pass->pw_comment);
printf("Login Dir  %s\n", pass->pw_pw_dir);
printf("Shell: %s\n", pass->pw_shell);
}
```

Group details

The group structure in **`grp.h`** header file contains group information as in the following:

`/etc/grpup`:

```
#include <grp.h>
struct group {
  char *gr_name;   /* group name */
  char *gr_passwd; /* group password */
  gid_t gr_gid;   /* group id*/
  char **gr_mem; /* pointer to group member names */
};
```

The **`getgid()`** gets the group id of the user, whereas **`getgrgid()`** populates the group structure.

The following is the example to print group information:

```
#include  <grp.h>
main()
{
  int I;
  struct group *grp;
  grp = getgrgid(getgid());
  printf("Group Name %s\n". grp->gr_name);
  printf("Group Password %s\n". grp->gr_passwd);
  printf("Group ID %s\n". grp->gr_gid);
  printf("Group Members :");
  for (I = 0; grp->gr_mem[I];I++)
   printf("\n: %s", grp->gr_mem[I];
}
```

Show information of all users logged in

Struct **`utmp`** in **`utmp.h`** header file has information of logged in users as **`/etc/utmp`**, as shown following. We can populate this structure by reading logged in user information from **`/etc/utmp`** file:

```
struct utmp {
  char ut_user[8];  /* User login name */
  char ut_id[4];    /* /etc/inittab id(usually line 2  char
  ut_line[12];  /* device name (console, lnxx) */
  short ut_pid;   /* short for compat. - process id     short
  ut_type;   /* type of entry */
  struct exit_status ut_exit; /* The exit status of a process
  */
  time_t ut_time;   /* time entry was made */
};
```

The following is a program to display information of logged in users:

```
#include <sys/types.h>
#include <stdio.h>
#include <utmp.h>
#include <pwd.h>
#define UTMP "/etc/utmp"
#define NAMELEN 8
main()
{
  FILE *fp;
  struct utmp u;
  struct passwd *p;
 char temp[NAMELEN+1];
  fp = fopen(UTMP,"r");
while (!feof(fp))
{
 fread(&u,sizeof(u),1,fp);
 if (u.ut_name == NULL) continue;
 strncpy(temp, u.ut_name, NAMELEN);
 p = getpwnam(temp);
 if (p == NULL) continue;
 printf("%-10.8s %-10.8s %-30.30s %s\n",u.ut_name, u.ut_line,
 p->pw_gecos, ctime(&u.ut_time));
}  /*The ctime() converts time to ascii format.*/
fclose(fp);
exit(0);
}
```

Process groups

Processes (under UNIX, at least) are organized into process groups, generally corresponding to an entire job. When a single shell command consists of a series of filter commands that pipe data from one to the other, those processes (and their child processes) all belong to the same process group. Each process group has a process group header and a process group number corresponding to the process number of the process group leader.

Process group header can create a process group and processes in the group. The process group exists as long as there is at least one process in the group, regardless of whether the process group header terminates. The time when the process group is created to the time when the last remaining process in the group leaves the group is called the process group lifetime. The last process in the group can terminate or can enter into some other group.

A process joins an existing group or creates a new process group using setpgid() function.

Prototype:

```
int setpgid(pid_t pid, pid_t pgid);
```

A process can set the process group ID of only itself or one of its children.

Sessions

A session is a collection of one or more process groups. A new session can be established by a process using the **`setsid()`** function.

Prototype:

```
Pid_t setsid(void);
```

It returns an error if the process is already a process group header. If the process is not a process group header, then the process becomes both session and group header, and the process group ID is set to the process ID of the process.

Let us see an example:

```
#include <sys/types.h>
  #include <unistd.h>
  #include <stdio.h>
void main()
{
 pid_t pid;
 printf("The process group ID is %d\n", (int) getpgrp());
setsid();
   printf("The new process group ID is %d\n", (int) getpgrp());
}
```

Signals

Signals are asynchronous software interrupts that indicates that an event has occurred. Signals are identified by their names and specified in the header file **`<signal.h>`**. Signals are sent by the following:

- One process to another (or itself)
- The Kernel to a process

Table 3.1 shows the free signal available in the BSD system:

Signal name	Signal number	Signal description
SIGHUP	1	Terminal line hangup
SIGINT	2	Interrupt program

SIGQUIT	3	Quit program
SIGILL	4	Illegal instruction
SIGTRAP	5	Trace trap
SIGABRT	6	Abort
SIGEMT	7	Emulate instruction executed
SIGFPE	8	Floating-point exception
SIGKILL	9	Kill program
SIGBUS	10	Bus error
SIGSEGV	11	Segmentation violation
SIGSYS	12	Bad argument to the system call
SIGPIPE	13	Write on a pipe with no one to read it
SIGALRM	14	Real-time timer expired
SIGTERM	15	Software termination signal
SIGURG	16	Urgent condition on I/O channel
SIGSTOP	17	Stop signal, not from terminal
SIGTSTP	18	Stop signal from terminal
SIGCONT	19	A stopped process is being continued
SIGCHLD	20	Notification to parent on child stop or exit
SIGTTIN	21	Read on a terminal by a background process
SIGTTOU	22	Write to terminal by a background process
SIGIO	23	I/O possible on a descriptor
SIGXCPU	24	CPU time limit exceeded
SIGXFSZ	25	File-size limit exceeded
SIGVTALRM	26	Virtual timer expired
SIGPROF	27	Profiling timer expired
SIGWINCH	28	Window size changed
SIGINFO	29	Information request
SIGUSR1	30	User-defined Signal 1
SIGUSR2	31	User-defined Signal 2

SIGTHR	32	Thread interrupt

Table 3.1: *BSD signals*

Sending a signal to processes

The kill system call is used to send a signal from one process to another process or to itself. To send a signal, the sending process and receiving process must have the same effective user ID, or sending process must be a superuser.

Prototype:

```
 int kill(int pid, int sig);
```

If **`pid`** argument is `0`, signal is sent to all processes in the group. If **`pid`** argument is **`-1`** and the user is not a superuser, the signal is sent to all processes whose real user ID is the same as the effective user ID of the sending process. If **`pid`** argument is **`-1`** and the user is superuser, the signal is sent to all processes except system processes. If **`pid`** argument is negative but not **`-1`**, then the signal is sent to all processes whose process group ID equals the absolute value of **`pid`**. If **`pid`** argument is `0`, error checking is done, but no signal is sent.

Another way of sending signals to processes is to use a certain keypress that is interpreted as a request to send a signal to the running process. A few examples are as follows:

- **Ctrl + C:** Pressing *C* on the keyboard with the *Ctrl* key causes the system to send an INT signal (SIGINT) to the running process. By default, this signal causes the process to terminate immediately.
- **Ctrl + Z:** Pressing *Z* with *Ctrl* key causes the system to send a TSTP signal (SIGTSTP) to the running process. By default, this signal causes the process to suspend execution.
- **Ctrl + \\:** Pressing \\ with *Ctrl* key causes the system to send an ABRT signal (SIGABRT) to the running process. By default, this signal causes the process to terminate immediately. Note that this redundancy (that is, *Ctrl* + \\ doing the same as *Ctrl* + *C*) gives us some better flexibility. We will explain that later on.

The **kill** command is also used for sending signals to processes. The **kill** command accepts two parameters: a signal name (or number) and a process ID.

```
kill -<signal> <PID>
```

For example, in order to send the **INT** signal to process with PID 5342, type:

```
kill -INT 5342
```

This has the same effect as pressing *Ctrl-C* in the shell that runs that process. If no signal name or number is specified, the default is to send a TERM signal to the process, which normally causes its termination.

Some hardware conditions also cause signals. For example, floating-point error, referencing an address outside a process, the specific hardware condition, and corresponding signals that they generate are different in different UNIX implementations.

Signal handling

The Kernel has a default action corresponding to each signal. The process can handle most of the signals in a customized way by writing code in the program. This custom piece of code is called signal handlers.

Two signals are unable to be redefined by a signal handler. SIGKILL always stops a process, and SIGSTOP always moves a process from the foreground to the background. These two signals cannot be caught by a signal handler.

The **signal()** system call is used to set a signal handler for a single signal type. **signal()** accepts a signal number and a pointer to a signal handler function and sets that handler to accept the given signal.

Prototype:

```
void (*signal(int sig, void (*func)(int)))(int);
```

The first argument sig is the name of the signal. The value of the second argument is either **SIG_DFL** or **SIG_ING** or the address of the signal handler. SIG_DFL is used to specify the default handling of the signal, and **SIG_ING** is used to ignore the signal.

Example: In the given example, after printing the message `press del key`, an endless for loop will be executed, but as soon as *del* key is pressed, an interrupt signal SIGINT will be sent to the handler function **sig_handle**.

The function will print the message `DEL key is pressed`, and the program will be terminated.

```
#include <signal.h>
void sig_handle();
void main()
{
 printf("press del key\n");
 signal(SIGINTSIDINT, sig_handle);
for(;;);
}
void sig_handle()
{
 printf(" DEL key is pressed\n");
}
```

Table 3.2 shows all possible values of signals for the first parameter of the **`signal()`** system call. These signals are handled either by the signal handler mentioned as the second parameter of the **`signal()`** system call or by the system if the handler is not mentioned.

Signal	Description
SIGALRM	A process can set an alarm clock by calling the alarm call. Unsigned int alarm (unsigned int sec). The sec argument specifies the number of seconds to elapse before the Kernel is to send the process a SIGALRM signal. If the argument is zero, any previous alarm clock for the process is canceled. The sleep function usually sets a SIGALRM signal, which it catches.
SIGINT	This interrupt signal is usually generated when the interrupt key is pressed on the terminal.
SIGILL	This signal is generated by an implementation-dependent hardware condition.
SIGUSR1	There are two user-defined signals that can be used to communicate between processes.
SIGUSR2	Same as SIGUSR1.
SIGXFSI	This signal is generated if the process exceeds its file size limit.
SIGXCPU	If the process exceeds its soft CPU time limit, this signal is generated.
SIGTERM	This is the termination signal sent by the kill command.

SIGPWR	This signal is generated at the time of power failure. The system continues running if battery power is available.
SIGPIPE	This signal is generated if the writer process continues writing on the pipeline and the reader process is terminated.
SIGQUIT	This signal is generated by the terminal driver when `Ctrl \` key is pressed. This signal is sent to all foreground processes.
SIGTSTP	The terminal driver generates this interactive stop signal when `Ctrl+D` is pressed. This signal is sent to all foreground processes.
SIGSTOP	The terminal driver generates this signal to stop all foreground processes. This signal cannot be caught and ignored.
SIGORG	This signal specifies the process that an urgent condition has occurred.
SIGCHLD	This signal is sent to the parent process when the process stops. Usually, this signal is ignored, so the parent must catch the signal if it wants to be notified whenever a child's status changes.
SIGIO	This signal occurred at the time of asynchronous IO.
SIGSYS	This signal indicates an invalid system call.

Table 3.2: *Signal and their descriptions*

Thread

A thread is a lightweight process. The threads have different execution states, and multiple threads share their states. Multiple threads can write and write the same memory area, whereas different processes cannot access the same memory area. Each thread still has its own registers and has its own stack, but other threads can read and write the stack memory.

Typically a process can be viewed as follows:

Process = Thread + Code, Data and Kernel Context

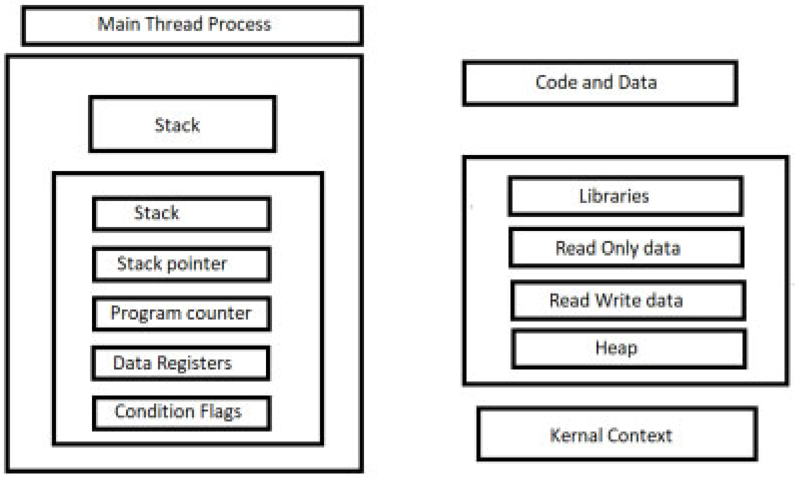

Figure 3.2: *Single thread process*

A process can have separate threads for different activities. Having a separate thread for each activity allows the programmer to define the actions associated with that activity as a single sequence of actions and events.

Each thread associated with the process has its own logical control flow; each thread shares the code, data, and Kernel context, and each thread has its own thread ID.

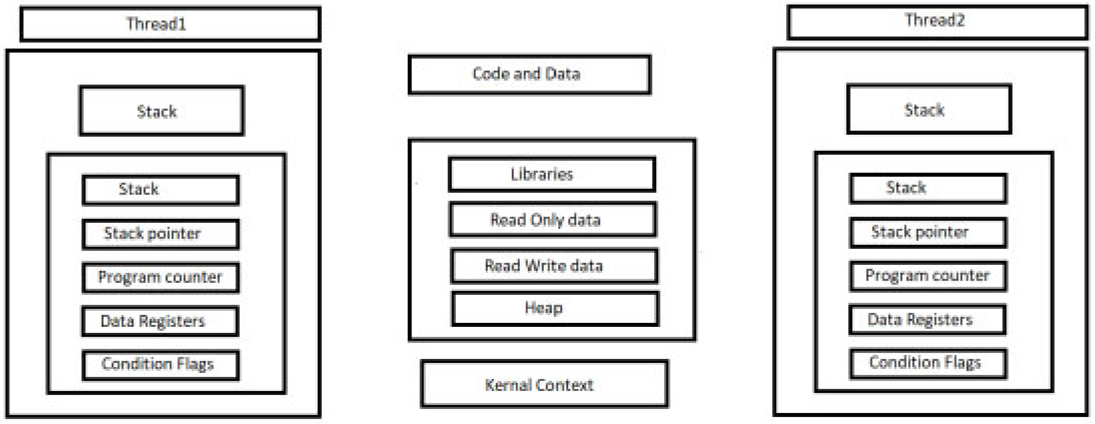

Figure 3.3: *Process with multiple threads*

Why allow threads to access the same memory? Because inside OS, threads must coordinate their activities very closely. Following are the situations when shared memory and disjoint memory are required by the threads:

- If two processes issue read file system calls at close to the same time must make sure that the OS serializes the disk requests appropriately.
- When one process allocates memory, its thread must find some free memory and give it to the process. It must ensure that multiple threads allocate disjoint pieces of memory.

Having threads share the same address space makes it much easier to coordinate activities—can build data structures that represent system state and have threads read and write data structures to figure out what to do when they need to process a request.

One complication that threads must deal with asynchrony. Asynchronous events happen arbitrarily as the thread is executing and may interfere with the thread's activities unless the programmer does something to limit the asynchrony. Examples are as follows:

- An interrupt occurs, transferring control away from one thread to an interrupt handler.
- A time-slice switch occurs, transferring control from one thread to another.
- Two threads running on different processors read and write the same memory.

Asynchronous events, if not properly controlled, can lead to incorrect behavior. Examples:

- Two threads need to issue disk requests. The first thread starts to program the disk controller (assume it is memory-mapped and must issue multiple writes to specify a disk operation). In the meantime, the second thread runs on a different processor and also issues the memory-mapped writes to program the disk controller. The disk controller gets horribly confused and reads the wrong disk block.
- Two threads need to write to the display. The first thread starts to build its request, but before it finishes, a time-slice switch occurs, and the second thread starts its request. The combination of the two threads issues a forbidden request sequence, and smoke starts pouring out of the display.

- For accounting reasons, the operating system keeps track of how much time is spent in each user program. It also keeps a running sum of the total amount of time spent in all user programs. Two threads increment their local counters for their processes, then concurrently increment the global counter. Their increments interfere, and the recorded total time spent in all user processes is less than the sum of the local times.

So, programmers need to coordinate the activities of the multiple threads so that these bad things do not happen. The key mechanism and synchronization can be adopted to coordinate the activities of the multiple threads. These operations allow threads to control the timing of their events relative to events in other threads. Appropriate use allows programmers to avoid problems like the ones outlined preceding.

Conclusion

This chapter describes the process subsystem of the UNIX operating system. The process subsystem manages memory when required by various processes, schedules processes for execution, and controls other aspects related to the processes. This chapter described the memory layout of the process and functions of managing memory. This chapter also described the context of the UNIX process and how context switching is performed when a process is either interrupted or the CPU is switched to another process. This chapter also described how processes inform each other about an asynchronous event by sending signals with the `signal()` system call. The upcoming chapter discusses all methods of inter-process communication in more detail.

Review Exercise

1. Write a program to ask the user to enter the total number of bytes he or she wants to allocate. Then, initialize the allocated memory with consecutive integers, starting from 1. Add all the integers contained by the memory block and print out the final result on the screen.
2. Write a program that allocates a block of memory space to hold 100 items of the float data type by calling the `calloc()` function. Then,

reallocate the block of memory in order to hold 50 more items of the float data type.

3. Write a program to ask the user to enter the total number of float data. Then use the **`calloc()`** and **`malloc()`** functions to allocate two memory blocks with the same size specified by the number and print out the initial values of the two memory blocks.
4. Write a program to display information about all the users logged in currently in the system.

CHAPTER 4
Inter-Process Communication

Introduction

Most modern operating systems allow for concurrent execution of multiple processes in a computer system. The processes can be independent of cooperating, that is, sharing data with each other or sometimes require to share event information with each other. An operating system provides inter-process communication mechanisms to execute cooperating processes either on the single system or in the networking environment. The IPC can be either synchronous or asynchronous. This chapter describes IPC and corresponding techniques used to perform IPC among different processes.

Structure

We will cover the following topics in this chapter:

- Interprocess communication and its different methods
- Pipes as means of IPC
- FIFO as means of IPC
- Shared memory and its usage
- Sending and receiving messages using the message queue
- Process synchronization using semaphores

Objective

After completing this chapter, you will be able to:

- Understand the concept of inter-process communication.
- Understand how named and unnamed pipes are used to share data between two cooperating processes.
- Understand how to share data directly between cooperating processes.

- Understand the functioning of semaphores for process synchronization.

Introduction to IPC

Two processes can communicate with each other using interposes communication techniques provided by an operating system. The processes may need to communicate to share data or to inform about some event. In a traditional single processing environment, modules within the single process can communicate with each other using global variables and function calls. In a multi-processing environment where processes run under different address spaces, an operating system must provide some facility for inter-process communication.

The use of IPC is not restricted to multiple processes running on a single system. *Figure 4.1* shows three processes residing in a single system that can communicate using different modes of IPC.

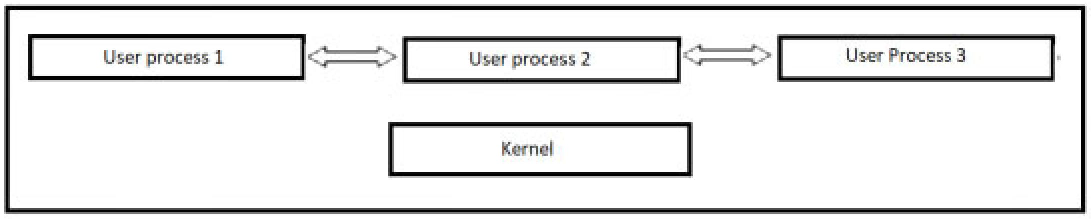

Figure 4.1: *Process communications on a single system*

Figure 4.2 shows a network environment where processes are running on different systems, and they can communicate with each other using means of interprocess communication.

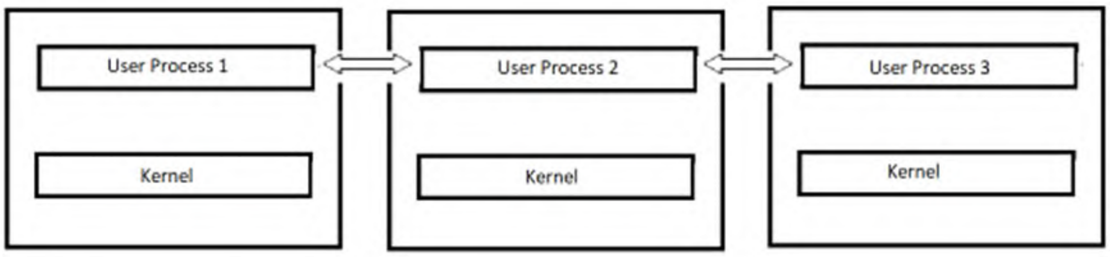

Figure 4.2: *Process communications on multiple systems*

Means of interprocess communication

The main methods of sharing data or inter-process communication are using the following:

- Files
- Message queues
- Pipes
- Shared memory
- Sockets

The communication between processes is synchronized using signals and semaphores. The C library functions and system calls are available for interprocess communication.

In a UNIX environment, the shared information can reside at different levels. The method of IPC used to share information is dependent on where information is available. The IPC methods corresponding to the availability of information are listed following:

- If the information to be shared is residing in a file in the file system, then Kernel system calls such as **`read()`**, **`write()`**, and so on are used to share the information between the processes. Some synchronization is needed when the file is being updated.
- If information is residing in the pipes, message queue, and semaphores, Kernel system calls corresponding to these data structures are used to share information between cooperative processes.
- Two processes can directly access information if it is available in shared memory. Once shared memory is set up by the processes, then for communication, Kernel involvement is not required.
- Two processes can share information by residing in a file in the file system. The Kernel system calls for files that are used to access information in files. Some synchronization is needed when the file is being updated.
- Two processes can share information that resides within the Kernel. Pipes, message queues, and semaphores are used to share the information.

- Two processes can refer to a region of shared memory. Once the shared memory is set up by the processes, two processes can access information without involving the Kernel at all.

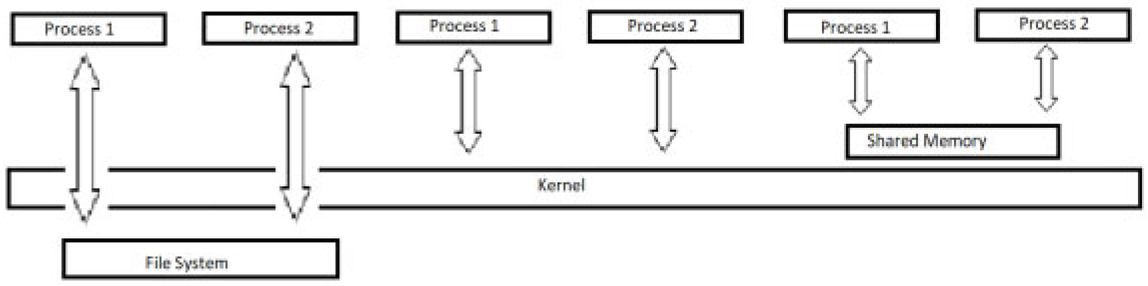

Figure 4.3: *Information sharing among processes*

Pipes and FiFOs

Through pipes, the output of one process is made the input of another process, as shown in *figure 4.4*. One process writes data on the pipe, and at the same time, the other process reads data from the pipe. The pipe must be open before use.

The **`pipe()`** system call creates a pipe and returns two file descriptors, **`fd[0]`**, **`fd[1]`**, **`fd[0]`** is opened for reading, and **`fd[1]`** is opened for writing. **`pipe()`** returns **`0`** on success, **`-1`** on failure and sets errno accordingly.

In the standard programming model, once the pipe has been set up, two (or more) cooperative processes will be created by a fork, and data will be passed using **`read()`** and **`write()`**.

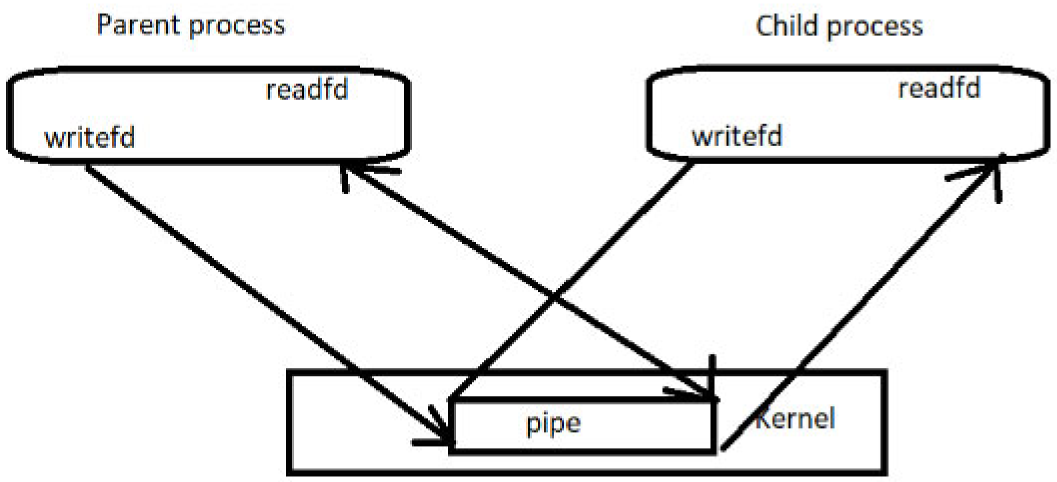

Figure 4.4: Reading and writing pipe

After use, pipes should be closed with **`close(int fd)`**.

Example:

In the given example, a pipe is created using **`pipe()`** System call. The parent process writes the message at one end of the pipe when it is executed, and the child process reads the same message from the other end of the pipe:

```
void main()
{
int fd[2];
pipe(fd);
if ( fork() == 0 )
{ /* child  process*/
  close(fd[1]);    /* fd[1] is not  used in this process*/
      printf("\n read data from parent );
  read( fd[0]); /* read from parent */
  }
else
  {   close(fd[0]);   /* not used in this process */
  printf("\n Write data to child process\n")_;
  write( fd[1]);
  }
exit(0)
}
```

One end closed pipe

If a read operation is performed from the pipe whose write end is closed, the read returns 0 to indicate the end of the file.

If the write operation is done whose read end is closed, then the SIGPIPE is generated. If we either ignore or catch it and return it from the signal handler, write returns an error with the errno set to EPIPE.

The popen() I/O library function

The **popen()** is a standard I/O library function that creates a pipe and forks another process to read from or write in the pipe.

Prototype:

```
FILE *popen(char *cmd, char *type);
```

The **cmd** is the shell command. The function **popen** forks and executes the **cmd** command and returns a FILE pointer that is used for either input or output depending upon the character string type.

If the value of type is **r**, then the calling process reads the standard output of the command. If the value of type is **w**, then the calling process writes to the standard input of the command. In case of failure, **popen()** returns NULL.

The pclose() I/O function

The **pclose()** function closes the I/O stream created by the **popen()** function. It returns an exit status of the command executed or –1 if the stream is not created.

Prototype:

```
int pclose(FILE *stream);
```

Example:

In the given example, **popen()** function executes the **who** command, and the output is written in the file pointed by **fp** pointer. The **fgets()** function reads the file and stores the data in the character string user. The contents of the user string are printed on the standard output using **printf()** function, and then the pipe is closed using **pclose()** function.

```
include <stdio.h>
main()
{
  FILE *fp;
  char user[130];   /* line of data from unix command*/
  fp = popen(“who”, “r”);  /* Issue the command.  */
    /* Read a line file pointed by fp*/
  while ( fgets( user, sizeof (user), fp))
  {
    printf(“%s”, user);
  }
```

```
 pclose(fp);
}
```

FIFOs

A FIFO i.e. First In First Out is named pipe. A UNIX FIFO allows one-way data flow between processes; however, with FIFO, a name is also associated. A FIFO allows unrelated processes to access a single FIFO.

FIFO is a file that has a directory entry and is accessed by pathname. It permanently exists in the file system hierarchy.

Creating FIFO

A FIFO is created using **`mknode()`** system call, which is generally reserved for a superuser to create a directory entry, but any user can create FIFO using **`mkfifo`** system call.

Prototype:

```
int mkfifo(const char *pathname, node_t node);
```

Where **`pathname`** is the UNIX pathname name of the fifo. The **`node`** contains open permission on the file specified in the pathname.

The **`mkfifo()`** returns an error EEXIST if FIFO already exists. Once FIFO is created, it must be opened for reading or writing using either **`open()`** system call or standard I/O library function **`fopen()`**.

Example:

In the given example, a FIFO named fifo is created with read and write permission from the owner. An abnormal termination occurs if FIFO already exists; otherwise, the program is terminated normally.

```
#include <stdlib.h>
main()
{
   int rc;
   int fifo_fds;
   char data[100];
   rc = mkfifo("fifo", S_IRUSR | S_IWUSR);
   if (rc != 0) {
     perror("mkfifo failure");
     exit(1);
    }
    exit(0);                                /* EXIT_SUCCESS */
}
```

Interprocess communication using FIFO

A FIFO is named pipe, and it is used for inter-process communication between two cooperating processes compared to the pipe, which is used to communicate between parent and child. The two processes communicating with each other are treated as server and client processes, as shown in *figure 4.5*.

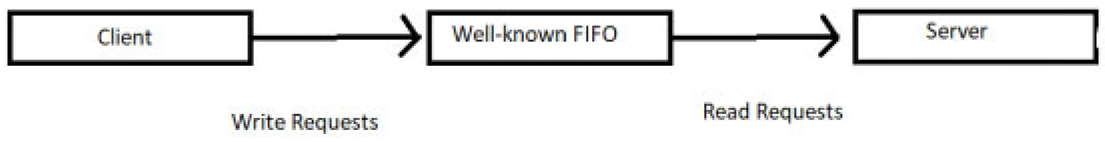

Figure 4.5: *Data transfer using FIFOs*

Server process

A FIFO is created and opened in the server process to accept client requests. The client request is read using the read function.

In the server process, FIFO should be opened for both reading and writing because if the client terminates, server's read will return 0 to show end-of-file, the server has to close FIFO and open it again in O_RDONLY mode, and the server will block until the next client request arrives. In the case of FIFO, which is always open for reading and writing, server read will never return 0.

Instead, the server will just block in a call to read, waiting for the next client request.

Example:

In the given example, a FIFO with the name fifo is created both for reading and writing. If fifo already exists, it returns an error, and an error message **`FIFO already exists`** will be printed. The server process waits to read a request from the client. If the client terminates in between, an error message `read failed` will be printed; otherwise, a message from fifo will be printed on the screen.

```
#include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#define MSGSIZ  100
char *fifo = "fifo";
int main(int argc, char **argv){
 int fd;
 char msg_buf[MSGSIZ+1];
if (mkfifo(fifo, 0666) == -1){
 if (errno != EEXIST)
 printf("\n FIFO already exists \n");
 }
 if ((fd = open(fifo, O_RDWR)) < 0)
 printf ("\nfifo open failed\n");
 for(;;)
 {
     if (read(fd, msg_buf, MSGSIZ+1) <0)
        printf ("\n message read failed \n");
 printf ("message received:%s\n", msgbuf);
 }
}
exit(0);
}
```

Client process

FIFO created in server process is opened in client process in **`O_WRONLY`** mode. The server's reply is read from the FIFO and written to standard output. The client's FIFO is then closed and deleted.

Example:

In the given example, the client process opens the FIFO in write-only mode. If FIFO open operation is successful, it checks if messages passed for printing on FIFO are having length than the buffer created to temporally hold the messages, then all the messages are copied to the buffer **`msg_buf`**, and one by one, they are written to the fifo.

```
include <fcntl.h>
#include <stdio.h>
#include <errno.h>
#define MSGSIZE 100
char *fifo = "fifo";
int main (int argc, char **argv){
  int fd, j, nwrite;
  char msg_buf[MSGSIZ+1];
  if (argc < 2){
printf ( "No message passed. \n");
  exit(1);
  }
  if ((fd = open(fifo, O_WRONLY | O_NONBLOCK)) < 0)
  printf("\n fifo open failed\n");
  for ( j = 1; j < argc; j++)
  {
  if (strlen(argv[j]) > MSGSIZ)
  {
printf ("message is long er than buffer %s\n", argv[j]);
       continue;
  }
  strcpy (msg_buf, argv[j]);
  if ((nwrite = write(fd, msg_buf, MSGSIZ+1)) == -1)
       printf("\n message write failed \n");
  }
  exit (0);
 }
```

Message queues

The message queue is the Kernel resident means of the IPC. Message queues are stored within the Kernel in the form of a linked list. An identifier identifies each message queue. Processes read and write messages to arbitrary queues. It is possible for a process to write some messages on the queue and exit and have the messages read by another process at a later time.

Each message on a queue has the following attributes:

- Long integer type message queue identifier.
- Length of the data portion of the message.
- Data.

Kernel maintains a data structure **`msqid_ds`** for every message queue, as follows:

```
/* one msqid structure for each queue on the system */
struct msqid_ds {
   struct ipc_perm msg_perm;
   struct msg *msg_first;  /* first message on queue */
   struct msg *msg_last;   /* last message in queue */
   time_t msg_stime;       /* last msgsnd time */
   time_t msg_rtime;       /* last msgrcv time */
   time_t msg_ctime;       /* last change time */
   struct wait_queue *wwait;
   struct wait_queue *rwait;
   ushort msg_cbytes;
   ushort msg_qnum;
   ushort msg_qbytes;      /* max number of bytes on queue */
   ushort msg_lspid;       /* pid of last msgsnd */
   ushort msg_lrpid;       /* last receive pid */
}
```

Let us discuss some of them as follows:

- **`msg_perm`**: Object of the **`ipc_perm`** structure defined in **`linux/ipc.h`**. This holds the permission information for the message queue, including the access permissions and information about the user who created the message queue (**`uid`**).
- **`msg_first`**: Contains a pointer to the first message in the message queue (the head of the list).
- **`msg_last`**: Contains a pointer to the last message in the message queue (the tail of the list).
- **`msg_stime`**: Timestamp (**`time_t`**) of the last message sent to the queue.

- **msg_rtime**: Timestamp of the last message retrieved from the queue.
- **msg_ctime**: Timestamp of the last “change” made to the queue.
- **wwait** and **rwait**: Pointers into the Kernel’s wait queue. They are used when an operation on a message queue deems the process to go into a sleep state (that is, the queue is full, and the process is waiting for an opening).
- **msg_cbytes**: Total number of bytes residing on the queue (sum of the sizes of all messages).
- **msg_qnum**: Total number of messages currently in the message queue.
- **msg_qbytes**: Maximum number of bytes that can reside on the message queue.
- **msg_lspid**: The process ID of the process sent the last message.
- **msg_lrpid**: The PID of the process retrieved the last message.

Creating and opening a message queue

A process can create a new message queue, or it can connect to an existing message queue. The **msgget()** function is used for both creation and connection.

Prototype

```
 int msgget(key_t key, int msgflg);
```

The key is a system-wide unique queue identifier describing the queue. The process of creating the message queue generates a key identifier, and other processes use this key to connect to the queue. The **msgflg** argument is a flag that contains read-write permission, which can be bitwise OR with **IPC_CREAT** or **IPC_EXCT** to create a new message queue or connect to an existing message queue.

Following members of **msgid_ds** structure are initialized when a new message queue is created:

- **msg_qbytes**: Set to the system limit.
- **msg_ctime**: Set to the current time.
- **msg_perm.mode**: Contain read-write permission from **msgflg** argument.

- **msg_qnum**: Set to **0**.
- **msg_lspid**: Set to **0**.
- **msg_lrpid**: Set to **0**.
- **msg_stime**: Set to **0**.
- **msg_rtime**: Set to **0**.
- **msg_perm.uid** and **msg_perm.cuid**: Set to the effective user ID of the process.
- **msg_perm.gid** and **msg_perm.cgid**: Set to the effective group ID of the process.

msgget() returns the message queue ID on success or −1 on failure

Sending a message to the queue

The **msgsnd()** function is used to send a message to the message queue.

Prototype:

```
int msgsnd(int msqid, const void *msgp, size_t msgsz, int
msgflg);
```

msqid is the message queue identifier returned by **msgget()**. The pointer **msgp** is a pointer to the message to put on the queue. Message is defined in **sys/msg.h** file in the form of structure **msgbuf**, as follows:

```
struct msgbuf {
   long mtype;
char mtext[[1];
}
```

There is no limit on the length of the **mtext** at compile time; rather, the limit is set by the system administrator. **msgsz** is the size in bytes of the message. **msgflg** is set to either **0** or **IPC_NOWAIT**.

The function returns as follows:

- If too many messages are available.
- If the number of bytes on the message queue exceeds the **msg_qbyte** limit.

If **msgflg** is set to **IPC_NOWAIT**, then the function returns the error **EAGAIN**, and if **msgflg** is not set to **IPC_NOWAIT**, then the thread is put to sleep.

Example:

In this given example, the process creates a message queue for writing a message. If message queue is not created an error message will be printed and program exits abnormally. Once message queue is created successfully, a message is read from keyboard and sent to the queue. In case of error message queue will be destroyed:

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

struct msgbuf {
 long mtype;
 char mtext[200];
};
int main(void)
{
 struct msgbuf buf;
 int msqid;
 key_t key= 100;
 if ((msqid = msgget(key, 0644 | IPC_CREAT)) == -1) {
  perror("msgget");
  exit(1);
 }
 printf("Get message from keyboard\\\\n");
 buf.mtype = 1; /* we don't really care in this case */
 while(gets(buf.msg_text), !feof(stdin)) {
  if (msgsnd(msqid, (struct msgbuf *)&buf, sizeof(buf), 0)
  == -1)
   perror("message not sent properly");
 }
 if (msgctl(msqid, IPC_RMID, NULL) == -1) {
  perror("message queue destroyed");
  exit(1);
 }
```

```
    return 0;}
```

Receiving a message from the queue

The **`msgrcv()`** function is used to read a message from the message queue.

Prototype

```
 int msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp,
 int msgflg);
```

The **`msqid`** is the message identifier, **`msgp`** is the pointer to the buffer to store the read message, **`msgsz`** is the length of the data part of the buffer storing the message, and messages are received from the queue depending on the **`msgtyp`** parameter. The following table shows which message will be read based on the value of **`msgtyp`** parameter:

msgtyp	Effect on msgrcv()
Zero	Retrieves the next message on the queue, regardless of its `mtype`.
Positive	Retrieves the next message with `mtype` equal to the specified `msgtyp`.
Negative	Retrieves the first message on the queue whose `mtype` field is less than or equal to the absolute value of the `msgtyp` argument.

Table 4.1: *Working of msgrcv()*

If the message of the specified type is not there in the message queue, then the action will be decided based on the value of **`msgflg`** parameter.

If **`msgflg`** is set to **`IPC_NOWAIT`**, the function returns immediately with the error **`ENOMSG`**, and the caller process is blocked until:

- The message queue is destroyed.
- A message of specified type ids arrived at the message queue.
- The calling thread is interrupted; in this case, the function returns with the error **`EINTR`**.

If **`msgflg`** is set with an additional bit **`MSG_NOERROR`**, the function does not return an error if the length of the actual message is greater than the **`msgsz`** parameter; instead, it truncates an extra portion of the message.

Example

In the given example, the receiver process connects to the message queue; if the message queue does not exist, an error will be generated, and the process terminates abnormally. Otherwise, all messages are read and printed to the screen till the end of the message queue.

```
   #include <stdlib.h>
   #include <errno.h>
   #include <sys/types.h>
   #include <sys/ipc.h>
   #include <sys/msg.h>

   struct msgbuf {
      long mtype;
      char mtext[200];
   };
   int main(void)
   {
      struct msgbuf buf;
      int msqid;
      key_t key = 41;

      if ((msqid = msgget(key, 0644)) == -1) { /* connect to
      the queue */
      perror("message queue does not exist ");
      exit(1);
      }
      printf("Receiving messages from message queue \n");
      for(;;)
{             if (msgrcv(msqid, (struct msgbuf *)&buf,
sizeof(buf), 0, 0) == -1) {
       perror("msgrcv");
       exit(1);
      }
      printf("%s"\n", buf.mtext);
      }
      return 0;
   }
```

Destroying a message queue

The message queue is destroyed using **`msgctl()`** function.

Prototype

```
 int msgctl(int cmd, struct msqid, int msqid_ds *buf);
```

Here, **`msqid`** is the queue identifier obtained from **`msgget()`**. The **`cmd`** argument is set to **`IPC_RMID`** to destroy a queue.

This **`cmd`** can only be executed by a process that has an effective user ID equal to either that of a superuser or to the value of either **`msg_perm.uid`** or **`msg_perm.cuid`** in the data structure associated with **`msqid`**.

Controlling message queue

The **`msgctl()`** system call is used to perform control operations on a message queue.

Prototype

```
 int msgctl ( int msgqid, int cmd, struct msqid_ds *buf );
```

Where **`msqid`** is a unique positive integer identifying a message queue, and **`cmd`** specifies a message control operation. **`cmd`** can take the following values:

- **`IPC_STAT`**: Returns the current message queue structure **`msqid_ds`** for the specified message queue ID.
- **`IPC_SET`**: Sets **`msg_perm.uid`**, **`msg_perm.gid`**, **`msg_perm.node`**, **`msg_qbytes`** members of the **`msqid_ds`** structure from the **`buf`** argument.
- **`IPC_RMID`**: Removes the message queue identifier specified by **`msqid`** from the system and destroys the message queue and data structure associated with it.

The **`buf`** points to a structure used for message queue data manipulation operations. On success, the function returns `0`. Otherwise, it returns `-1`, and an errno is set.

Shared memory

The problem with pipes, message queues, and FIF is that for two processes to exchange information, and the information has to go through the Kernel.

Shared memory is the form of IPC that allows processes to communicate by writing onto a memory area that is shared among them. Processes need to coordinate the use of shared memory segments to exchange information. If one process is reading a shared segment, the other processes have to wait.

Creating shared memory

The **`shmget()`** is used to obtain access to a shared memory segment.

Prototype

```
int shmget(key_t key, size_t size, int shmflg);
```

The **`key`** argument is an access value associated with the shared memory. The **`size`** argument is the size in bytes of the requested shared memory. The **`shmflg`** argument specifies the initial access permissions and creation control flags.

When the call succeeds, it returns the shared memory segment ID. This call is also used to get the ID of an existing shared segment (from a process requesting sharing of some existing memory portion).

Controlling a shared memory segment

The **`shmctl()`** function is used to alter the permissions and other characteristics of a shared memory segment.

Prototype

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

The process must have an effective **`shmid`** of the owner, creator, or superuser to perform this command. The **`cmd`** argument is one of the following control commands:

- **`SHM_LOCK`**: Locks the specified shared memory segment in memory. The process must have the effective ID of the superuser to perform this command.
- **`SHM_UNLOCK`**: Unlocks the shared memory segment. The process must have the effective ID of the superuser to perform this command.
- **`IPC_STAT`**: Returns the status information contained in the control structure and places it in the buffer pointed to by **`buf`**. The process

must have read permission on the segment to perform this command.

- **IPC_SET**: Sets the effective user and group identification and access permissions. The process must have an effective ID of owner, creator, or superuser to perform this command.
- **IPC_RMID**: Removes the shared memory segment.

The **buf** is the structure of type struct **shmid_ds**, which is defined in **<sys/shm.h>**.

Attaching and detaching a shared memory segment

The **shmat()** and **shmdt()** are used to attach and detach shared memory segments to the address space of the calling process.

Prototypes

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

The **shmat()** function attaches the shared memory segment associated with the shared memory identifier, **shmid**; it reruns a pointer, **shmaddr**, to the head of the shared segment associated with a **shmid**.

The **shmdt()** detaches the shared memory segment located at the address indicated by **shmaddr**.

Example

In the given example, a shared memory segment is created for reading and writing. The created segment is attached to the current program, and string **str** is written onto it. After writing, the process goes into a sleep state until some other process reads the string and puts **'*'** onto the memory segment.

Server program

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSIZE     100
```

```
#define  SHM_KEY   1267
main()
{
   char *str;
   int shmid;
   key_t key;
   char *shm, *s;
   key = SHM_KEY;
  /*     * Create the segment.     */
   if ((shmid = shmget(key, SHMSZ, IPC_CREAT | 0666)) < 0) {
      printf("Segment not created successfully");
      exit(1);
   }
   /* Now we attach the segment to our data space     */
   if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
    perror("shmat");
      exit(1);
   }
   s = shm;
scanf("%s",&str);
strcpy(s,str);
   *s = NULL;
  /* Process goes in sleep state until some other process read
  shared segment and put *   */
   while (*shm != '*')
      sleep(1);
   exit(0);
}
```

In the client program, the memory segment is accessed and attached to the client process, If successfully attached, the string from memory segment is read and printed to the screen until NULL character found. Once memory segment is empty, the client process puts a `'*'` onto the segment. Server process terminates as soon as it find `'*'`.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
```

```
   #define SHMSIZE     100
#define  SHM_KEY   1267
main()
{
   int shmid;
   key_t key;
   char *shm, *s;
   if ((shmid = shmget(SHM_KEY, SHMSIZE, 0666)) < 0) {
     printf('\n Memory segment not created successfully \n");
     exit(1);
   }
   if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
     perror("shmat");
     exit(1);
   }
   /* Now read what the server put in the memory     */
   for (s = shm; *s != NULL; s++)
     putchar(*s);
   putchar('\n');
  /*After reading characters from the shared memory. put * at
  the first location     */
   *shm = '*';
   exit(0);
}
```

Process synchronization

Two concurrent processes must be synchronized while sharing resources with each other to avoid system deadlock. The UNIX uses semaphores to control access to the shared resource between two processes.

Semaphores

A semaphore is not exactly a form of IPC; it is a variable to control access to the shared resource among multiple processes.

A semaphore is a resource that contains an integer value and 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) is guaranteed no other process will interfere with the operation in the middle.

A process needs to perform testing before getting access to the shared resource. The access to the shared resource is controlled by a semaphore.

The following steps are performed to check the value of the semaphore before using the resource as follows:

1. Process tests the semaphore that controls access to the resource.
2. If the value of the semaphore is positive, then only the process can use the resource. As soon as the process gets access to the resource, it decrements the value of the semaphore by 1 to indicate that it has used one unit of resource.
3. If the value of the semaphore is 0, the process goes into a sleep state; when the process wakes up, it again checks the value of the semaphore.
4. After finishing the use of the resource, the value of the semaphore is incremented by 1 to give access to any other process that is in the sleep state.

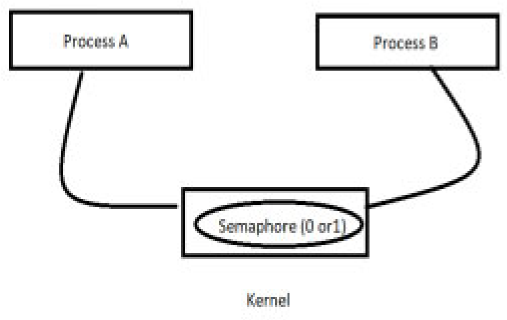

Figure 4.6: *Semaphore*

Semaphore operations

Two types of operations are carried on a semaphore: wait and signal. The wait operation first checks if 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 for the semaphore.

Semaphore set

A semaphore set is a structure that stores a group of semaphores together and possibly allows the process to commit a transaction on part or all of the semaphores in the set together. A transaction means that we are guaranteed that either all operations are done successfully or none is done at all.

Kernel maintains a structure for every semaphore set:

```
 struct semid_ds {
      struct ipc_perm sem_perm;       /* permissions .. see
      ipc.h */
      time_t          sem_otime;      /* last semop time */
      time_t          sem_ctime;      /* last change time */
      struct sem      *sem_base;      /* ptr to first semaphore
      in array */
      struct wait_queue *eventn;
      struct wait_queue *eventz;
      struct sem_undo  *undo;        /* undo requests on this
      array */
      ushort          sem_nsems;      /* no. of semaphores in
      array */
      };
```

Let us discuss them in detail:

- **`ipc_perm`**: **`ipc_perm`** structure contains access permissions for a particular semaphore.
- **`sem_otime`**: Time of the last **`semop()`** operation (more on this in a moment).
- **`sem_ctime`**: Time of the last change to this structure (mode change).
- **`sem`**: The **`sem`** structure is an internal structure used by the Kernel to maintain the set of values for the semaphore.
- **`sem_undo`**: Number of undo requests in this array.
- **`sem_nsems`**: Number of semaphores in the semaphore set.

- **wait_queue**: A **wait_queue** is a circular list of pointers to task structures.

Creating semaphore

A semaphore is created, or an existing semaphore can be accessed using **semget()** system call, which returns the semaphore ID.

Prototype

```
int semget(key_t key, int nsems, int semflg);
```

Key is a unique identifier that is used by different processes to identify this semaphore set. The next argument, **nsems**, is the number of semaphores that should be created in the new semaphore set; the **semflg** argument specifies what permissions should be on the new semaphore set.

If the value of **senflg** is **IPC_CREAT** alone, it returns an identifier for a newly created semaphore set or an identifier for an already existing semaphore set with which the key argument matches. If **IPC_CREAT** is used along with **IPC_EXCL**, then it creates the new set or returns –1 if the semaphore set already exists.

Controlling semaphore

After the semaphore set is created, the whole set or a particular semaphore is controlled using **semctl()** system call.

Prototype

```
int semctl ( int semid, int semnum, int cmd, union semun arg
);
```

The **semid** is the key identifier of the semaphore set, and the **semnum** argument is an index of semaphore in the semaphore set, which is targeted for control operation. The **cmd** argument controls the semaphore operation, and it has the following values:

- **IPC_STAT**: Retrieves the **semid_ds** structure for a set and stores it in the address of the **buf** argument in the **semun** union.
- **IPC_SET**: Sets the value of the **ipc_perm** member of the **semid_ds** structure for a set. Takes the values from the **buf** argument of the

semun union.

- **IPC_RMID**: Destroys the set from the Kernel.
- **GETALL**: Gives values of all semaphores in a set. The integer values are stored in an array of unsigned short integers pointed to by the array member of the union.
- **GETNCNT**: Returns the number of processes currently waiting for resources.
- **GETPID**: Returns the PID of the process which performed the last **semop** call.
- **GETVAL**: Returns the value of a single semaphore within the set.

Semaphore operations semop()

The **semop()** system call is used to perform atomically a user-defined set of semaphore operations on the semaphores set.

Prototype

```
 int semop(int semid, struct sembuf *sops, unsigned int nsops);
```

The argument **semid** is the number obtained from the call to **semget()** to identify the semaphore set, and the next argument sops is a pointer to a user-defined array of the struct **sembuf** that is filled with semaphore commands; the third argument **nsops** specifies the number of semaphore operation structures.

```
struct sembuf {
 ushort sem_num;
 short sem_op;
 short sem_flg;
 };
```

The **sem_num** is the number of the semaphore in the set that is targeted for manipulation, and the **sem_op** defines the operation on the semaphore. This takes on different meanings, depending on whether **sem_op** is positive, negative, or zero. Let us have a look at them:

sem_op	What happens
Positive	The value of `sem_op` is added to the semaphore's value. This is how a program uses a semaphore to mark a resource as allocated.

Negative	If the absolute value of `sem_op` is greater than the value of the semaphore, the calling process will block until the value of the semaphore reaches that of the absolute value of `sem_op`. Finally, the absolute value of `sem_op` will be subtracted from the semaphore's value. This is how a process releases a resource guarded by the semaphore.
Zero	This process will wait until the semaphore in question reaches 0.

Table 4.2: *Values of sem_op variable*

The **`sem_flg`** is set to **`IPC_NOWAIT`** and **`SEM_UNDO`**. If an operation asserts **`SEM_UNDO`**, it will be undone when the process exits.

Destroying a semaphore

The semaphore set is destroyed using **`semctl()`** system call. The **`cmd`** argument is set to **`IPC_RMID`**, which tells **`semctl()`** to remove this semaphore set. The two parameters **`semnum`** and **`arg`** have no meaning in the **`IPC_RMID`** context and can be set to anything.

Example:

Assume the semaphore in our set whose id is **`semset_id`** was initialized to **`1`** initially:

```
/* 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 code needs some explanations, especially regarding the semantics of the **`semop()`** calls, which are as follows:

- **Comment 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 (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.
- **Comment 2:** The semantics of **`semop()`** assures us that when we return from this function, the value of the semaphore is 0. Why? It could not be less, or else **`semop()`** will not return. It could not be more due to the way we, later on, signal the semaphore.
- **Comment 3:** After we are done 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 wakened and continues its execution.

Now, let us assume that any process that tries to access the 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:

- 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.
- 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()`**.

Conclusion

Interprocess communication is an important mechanism that allows two processes to communicate with each other. IPC is either used to share data among cooperating processes or through this one process can control activities in other processes; IPC is used to communicate between two threads in a single process, two processes residing in a single system, or processes residing in a distributed environment. We discussed the concepts of IPC, including all methods used for IPC. This also discussed the concept of semaphores for process synchronization to avoid deadlock situations while sharing the shared resource by two concurrent processes. The upcoming chapter discusses another inter-process communication method based on socket between client and server residing in or across the network.

Review Exercise

1. How does a semaphore provide access to a shared object to multiple processes?
2. Define function calls in UNIX for implementing semaphores.
3. Write a program to create a message queue and send some data to it.

4. Write a program to destroy a message queue after receiving data available on it.
5. A shared memory segment is a memory area that can be attached to multiple processes. Write a client and server program that passes data back and forth.
6. Write a program to show race conditions between client and server programs.
7. What happens if the argument to open is a non-existent command?
8. Write a program to send and receive data between parent and child processes through pipes.

CHAPTER 5
Socket Programming

Introduction

Socket programming is the method through which a server can communicate to the client. A client creates a connection with the server on a particular port. The server listens to the client's request for connection. The server can communicate with multiple clients at a time and serves these clients in a synchronous manner. This chapter discusses the socket as a communication interface between client and server to send and receive information.

Structure

We will cover the following topics in this chapter:

- Socket
- Types of sockets
- Socket data structure
- System calls for socket communication
- I/O models
- Name and address conversion

Objective

After reading this chapter, you will be able to:

- Understand concepts and types of sockets for communication between server and client.
- Understand execution of various system calls involved in socket communication.
- Understand I/O models to facilitate socket communication.

- Understand system calls used for the name to address calculation.

Socket

The socket is an interface between an application process and the transport layer. The socket is a medium through which an application process can send/receive messages to/from another local or remote application process. This type of communication is commonly used between server processes and client processes.

In UNIX, sockets are called UNIX domain sockets because sockets can be used to communicate between processes residing on a single host only. In UNIX, a socket is defined by a file descriptor—an integer associated with an open file. In contrast, Internet domain sockets support various communication protocols. Sockets can be supported by the Kernel or as a library that translates calls to the native network API. The following figure illustrates the socket diagram:

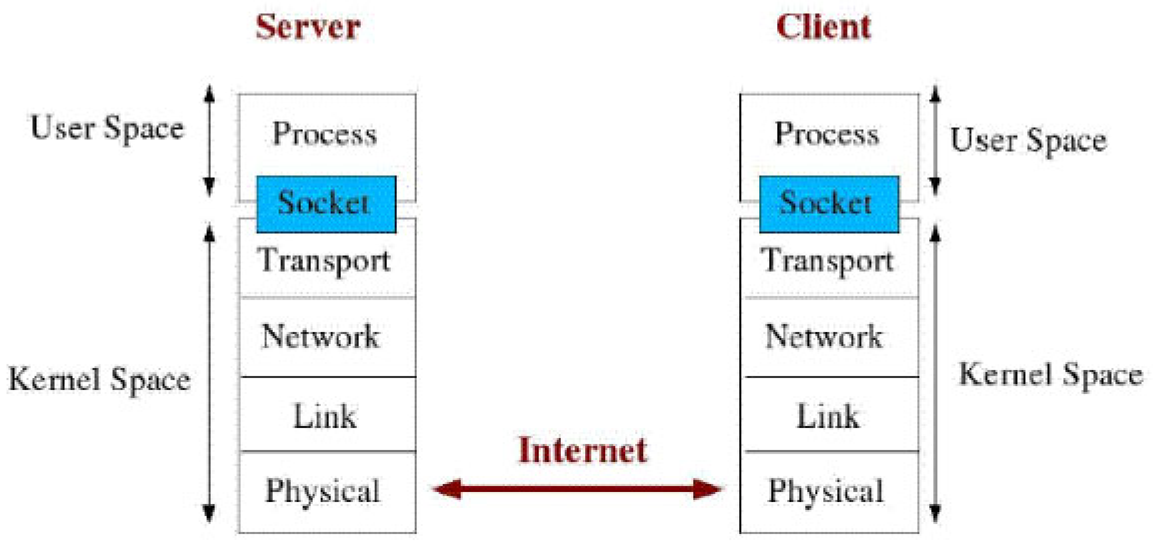

Figure 5.1: *Socket*

Types of sockets

While creating the socket, the program needs to specify the address domain and the type of socket to be created, and two processes are allowed to communicate with each other only if the socket through which they are communicating is in the same domain and has the same socket type.

The two widely used socket address domains are the UNIX domain and the Internet domain. In the UNIX domain, the processes sharing a common file system can communicate with each other, while in the Internet domain, the processes are not necessarily in the same file system; rather, they can reside on any hosts on the Internet to communicate with each other. Both the domain types have their own address formats. In the UNIX domain, the address has an entry in the file system and is represented by a character string. In the Internet domain, the socket address is represented by the 32-bit IP address of a host machine on the Internet. In the Internet domain socket also need a port number to represent a 16-bit unsigned integer.

Socket data structure

Information about the socket is stored in a data structure named as **`sockaddr`**. The **`sockaddr`** structure contains two elements **`sa_family`** and **`sa_data`**. The **`sa_family`** element stores the protocol addresses.

```
struct sockaddr {
   unsigned short sa_family; //address family AF_xxx
   unsigned short sa_data[14]; //14 bytes of protocol addr
}
```

System calls for socket communication

Socket communication is established on the client-side by the following:

- Creating a socket with the **`socket()`** system call.
- Connecting the newly created socket to the address of the server using the **`connect()`** system call.
- Sending and receiving data with **`read()`** and **`write()`** system calls.

Socket communication is established on the server-side by the following:

- Creating a socket with the **`socket()`** system call
- Binding the newly created 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.
- Listening for connections with the **`listen()`** system call.

- Accepting a connection request from a client with the **`accept()`** system call. This call typically blocks until a client connects with the server.
- Sending and receiving data with the **`read()`** and **`write()`** calls.

Table 5.1 shows the set of functions used for communication between server and client through sockets in both TCP and UDP protocols.

Protocol type	Client	Server
Connection-oriented TCP	`socket()`	`socket()`
	`connect()`	`bind()`
		Listen() TCP
		`accept()`
	`send()`	`recv()`
`recv()`	`send()`	
connectionless UDP	`socket()`	`socket()`
		`bind()`
	`sendto()`	`recvfrom()`
	`recvfrom()`	`sendto()`

Table 5.1: *Different socket related system calls*

Creating a socket (server and client)

The **`socket()`** system call creates the socket and returns a unique file descriptor for the socket. The socket can either be a stream (TCP) socket or a datagram (UDP) socket depending on the input arguments.

Prototype:

```
 int socket(int domain, int type, int protocol);
```

The domain argument is set to AF INET or PF INET, the type argument specifies the kind of socket and is set to SOCK STREAM for TCP streams (telnet, HTTP, and so on) or to SOCK DGRAM for UDP datagrams; the protocol takes value 0 to automatically select the correct protocol based on type or IPPROTO TCP to select TCP protocol or IPPROTO UDP to select UDP protocol.

Binding socket to an address (server)

The **`bind()`** system call is used to associate socket **`sockfd`** with a port on the local machine. This function only needs to be called for incoming connections on the server. It returns -1 if there is an error.

Prototype:

```
int bind(int sockfd, struct sockaddr *addr, int addrlen);
```

The first argument **`sockfd`** is the value of the descriptor returned by the **`socket()`** system call, and the **`addr`** is the **`sockaddr`** in the data structure containing the IP address and port number on the local machine, **`addrlen`** is the size of the **`sockaddr`** in the data structure.

Listening incoming connection

The **`listen()`** system call listens for incoming connections. It only needs to be called by the server for connection-oriented (TCP) sockets. The function returns −1 on an error.

Prototype:

```
int listen(int sockfd, int backlog);
```

The **`sockfd`** is the value returned by the **`socket()`** function call, and the backlog is the maximum number of connections allowed to wait in the incoming queue. Incoming connections remain in the queue until it is accepted by the server.

Initiating connection

The client calls **`connect()`** system call to initiate a connection to a server. It is used for connection-oriented (TCP) sockets. The system call returns −1 on an error.

Prototype:

```
int connect(int sockfd, struct sockaddr *addr, int addrlen);
```

The **sockfd** is the value returned by the **socket()** function call, **addr** is the **sockaddr** containing the IP address and port number of the server, and **addrlen** is the size of the **sockaddr**.

Accepting the connection from the incoming queue

The server uses **accept()** system call to accept a connection from the incoming queue associated with the socket **sockfd**. It is used for connection-oriented (TCP) sockets. The system call returns a new socket file descriptor, which can be used to send and receive information on the connection. The system call returns **-1** on an error.

Prototype:

```
int accept(int sockfd, void *addr, int *addrlen);
```

The **sockfd** is the socket descriptor for the socket that is listening for connections. **addr** is a pointer to a local **sockaddr** containing the IP address and port number of the incoming connecting client. This data structure is different from the one that contains the IP address and port number of the server; **addrlen** is the size of the preceding **sockaddr** in the data structure.

Sending data through socket (connection-oriented)

Once a connection has been established, the client and server use the **send()** system call to send information from client to server or from server to client. If the client is using **send()**, the server should be using **recv()** or vice-versa. This function either returns the number of bytes sent out or returns **-1** if there is an error.

Prototype:

```
int send(int sockfd, const void *msg, int len, int flags);
```

The **sockfd** is the socket file descriptor for the socket being used to send the data. On the client-side, **sockfd** is the same socket used when calling **connect()**. On the server-side, **sockfd** is the socket returned from **accept()**, **msg** is a pointer to the data that is being sent, **len** is the length of the data in bytes, flags are set to **0**.

Receiving message through socket (connection-oriented)

The **`recv()`** system call is used to receive data that has been sent over the socket. The function returns the number of bytes actually received; in case of an error, it returns **`-1`** or returns **`0`** if the other end has closed the connection.

Prototype

```
int recv(int sockfd, void *buf, int len, unsigned int flags);
```

The **`sockfd`** is the socket descriptor for the socket from which the data is being read. On the client-side, **`sockfd`** is the same socket used when calling **`connect()`**. On the server-side, **`sockfd`** is the socket returned from **`accept()`**, **`buf`** is a pointer to the buffer that will hold data after reading, **`len`** is the maximum length of the buffer in bytes, flags are set to **`0`**.

Receiving message through socket (connectionless)

This function is similar to **`send()`**; however, it is used for connectionless datagrams rather than connection-oriented communications.

Prototype

```
int sendto(int sockfd, const void *msg, int len, unsigned int
flags, const struct sockaddr
*toaddr, int addrlen);
```

The **`sockfd`** is the socket descriptor for the socket being used to send the data, **`msg`** is a pointer to the data that is being sent, **`len`** is the length of the data in bytes, flags are set to **`0`**, **`toaddr`** is the **`sockaddr`** containing the IP address and port number to which the data is being sent, **`addrlen`** is the size of the preceding **`sockaddr`** in the data structure.

Closing socket

The **`close()`** system call is used to close the socket.

Prototype

```
void close(sockfd);
```

Let us see an example of the client and server programs.

Client program

The following is the code for the client side:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#define PORT        5000
#define HOST        “uptu.ac.in”
#define DIRSIZE     8000
main(argc, argv)
int argc; char **argv;
{
     char hostname[100];
  char dir[DIRSIZE];
  int  sd;
  struct sockaddr_in sin;
  struct sockaddr_in pin;
  struct hostent *hp;
    strcpy(hostname,HOST);
    if (argc>2)
     { strcpy(hostname,argv[2]); }
  if ((hp = gethostbyname(hostname)) == 0) {
  perror(“gethostbyname”);
  exit(1);
  }
  /* fill in the socket structure with host information */
  memset(&pin, 0, sizeof(pin));
  pin.sin_family = AF_INET;
  pin.sin_addr.s_addr = ((struct in_addr *)(hp->h_addr))-
  >s_addr;
  pin.sin_port = htons(PORT);
  /* grab an Internet domain socket */
  if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
```

```
 perror(“socket”);
 exit(1);
 }
 /* connect to PORT on HOST */
 if (connect(sd,(struct sockaddr *)  &pin, sizeof(pin)) == -1)
 {
 perror(“connect”);
 exit(1);
 }
 /* send a message to the server PORT on machine HOST */
 if (send(sd, argv[1], strlen(argv[1]), 0) == -1) {
 perror(“send”);
 exit(1);
 }
     /* wait for a message to come back from the server */
     if (recv(sd, dir, DIRSIZE, 0) == -1) {
      perror(“recv”);
      exit(1);
     }
     printf(“%s\n”, dir);
 close(sd);
}
```

Server program

Following is the code for the server side:

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <string.h>
#define PORT  5000
#define DIRSIZE  8000
main()
{
     char     dir[DIRSIZE];  /* used for incoming dir name,
     and
```

```
  outgoing data */
 int   sd, sd_current, cc, fromlen, tolen;
 int   addrlen;
 struct   sockaddr_in sin;
 struct   sockaddr_in pin;
 /* get an internet domain socket */
 if ((sd = socket(AF_INET, SOCK_STREAM, 0)) == -1) {
 perror("socket");
 exit(1);
 }
 /* complete the socket structure */
 memset(&sin, 0, sizeof(sin));
 sin.sin_family = AF_INET;
 sin.sin_addr.s_addr = INADDR_ANY;
 sin.sin_port = htons(PORT);
 /* bind the socket to the port number */
 if (bind(sd, (struct sockaddr *) &sin, sizeof(sin)) == -1) {
 perror("bind");
 exit(1);
 }
 /* show that we are willing to listen */
 if (listen(sd, 5) == -1) {
 perror("listen");
 exit(1);
 }
 /* wait for a client to talk to us */
    addrlen = sizeof(pin);
 if ((sd_current = accept(sd, (struct sockaddr *)  &pin,
 &addrlen)) == -1) {
 perror("accept");
 exit(1);
 }
/* if you want to see the ip address and port of the client,
uncomment      the next two lines */
   /*
printf("Hi there, from  %s#\n",inet_ntoa(pin.sin_addr));
printf("Coming from port %d\n",ntohs(pin.sin_port));
```

```
      */
  /* get a message from the client */
  if (recv(sd_current, dir, sizeof(dir), 0) == -1) {
  perror("recv");
  exit(1);
  }
      /* get the directory contents */
      read_dir(dir);
     /* strcat (dir," DUDE");
     */
  /* acknowledge the message, reply w/ the file names */
  if (send(sd_current, dir, strlen(dir), 0) == -1) {
  perror("send");
  exit(1);
  }
      /* close up both sockets */
  close(sd_current); close(sd);
      /* give client a chance to properly shutdown */
      sleep(1);
}
```

I/O models

There are normally two distinct phases for an input operation:

- Waiting for the data to be ready.
- Copying the data from the Kernel's buffer to the process.

Blocking I/O model

When the socket is in blocking mode, Kernel puts the process in the wait state until there is data to read or data is fully written. If data is not available, wait till data is ready and block the process. As shown in *figure 5.2*, the application is asking for data from Kernel in blocking mode. In this case, Kernel puts the application in a wait state till data is available. Once data is available and written to the application, it resumes execution.

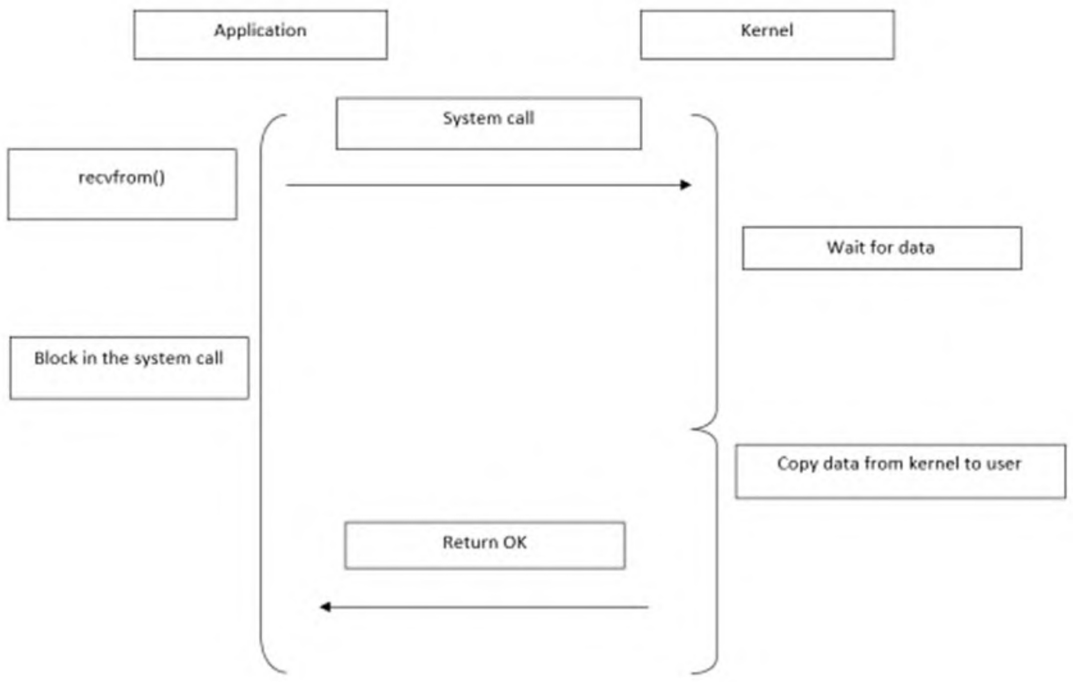

Figure 5.2: *Block diagram of I/O model (blocking)*

Non-blocking I/O model

When the socket is in non-blocking mode, the Kernel does not put the process in a sleep state but returns with an error **`EWOULDBLOCK`**.

The non-blocking I/O model looks as shown in *figure 5.3*:

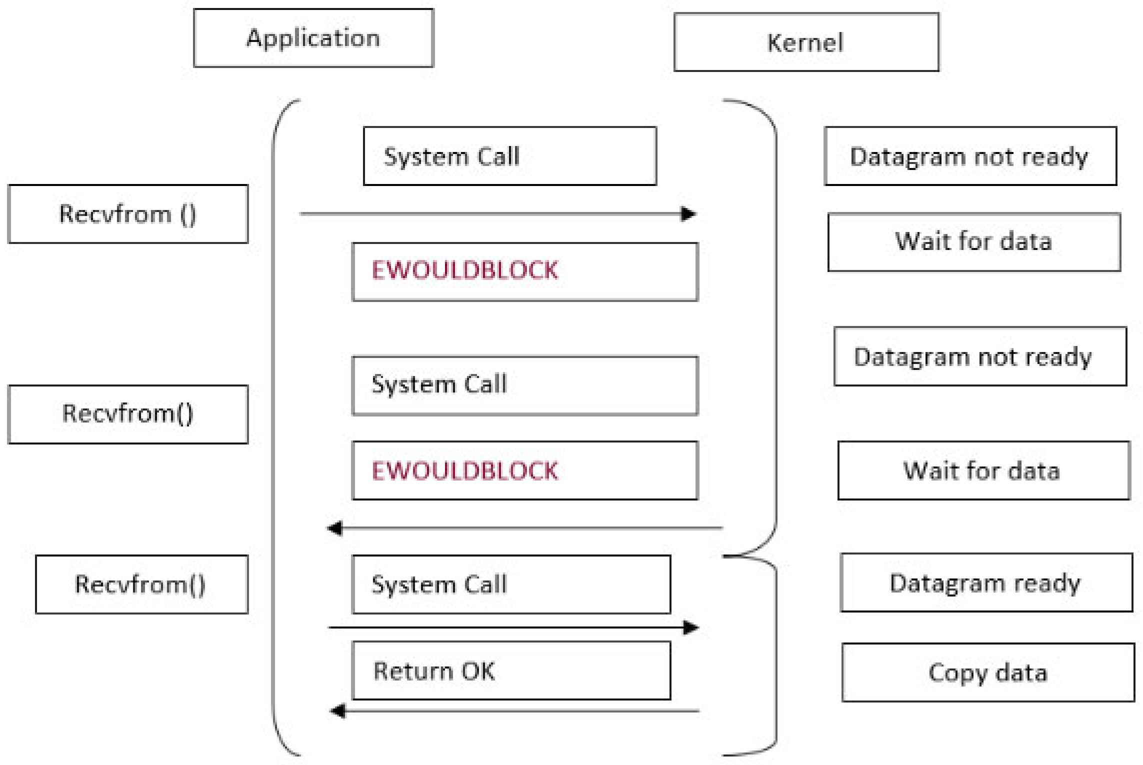

Figure 5.3: *Block diagram of I/O model (non-blocking)*

To receive data, the application repeatedly calls **`recvfrom`**. During a call, if data is not available, then a Kernel returns an error of **`EWOULDBLOCK`**. If data is available during **`recvfrom`**, it is copied to the application's buffer, and the function returns successfully. This process is called polling, where the application is continuously polling the Kernel to see if some data is ready. In this process, there is a wastage of CPU time, but this model is occasionally encountered, normally on systems dedicated to one function.

I/O multiplexing model

The process is blocked in **`select()`** or **`poll()`** if one of these systems calls instead of blocking on the actual I/O itself. On return, the socket is readable; that is, data is ready on any of the descriptors, and process calls **`recvfrom()`** to get the actual data.

The advantage of this model is that process can wait for more than one file descriptor to be ready. As shown in *figure 5.4*, the application executes **`select()`** system call; if data is not ready at the server end, the application blocks the system call, but the application itself will be in running mode. Once data is available, the application executes **`recefrom()`** function to receive data from the Kernel and immediately returns.

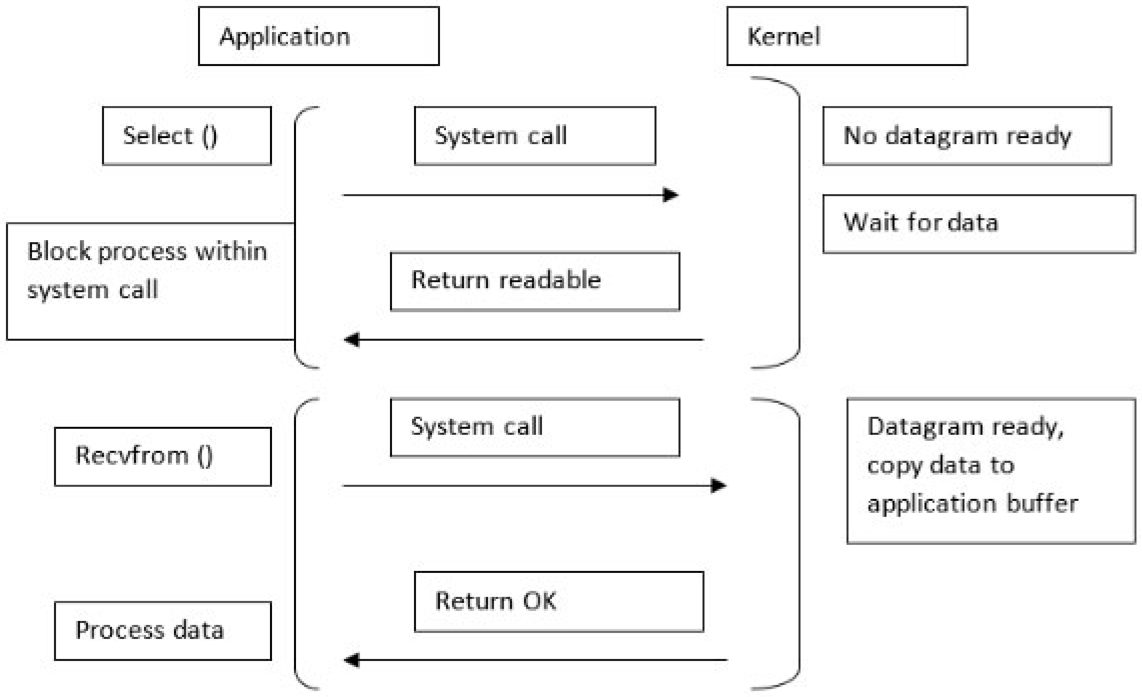

Figure 5.4: *Block diagram of multiplexing I/O model*

A disadvantage of this model is that it requires two system calls instead of one.

Signal-driven I/O model

In this model, Kernel notifies the process that data is available by sending the SIGIO signal. As shown in *figure 5.5*, the application enables signal handler using **`sigaction()`** function. After enabling, the application keeps on executing. Once data is available at the Kernel end, Kernel sends a SIGIO signal to the application. SIGIO signal is handled by the signal handler at the application end. Now, the application call **`receivfrom()`** function to receive data from Kernel, and when data is completely written, it starts executing.

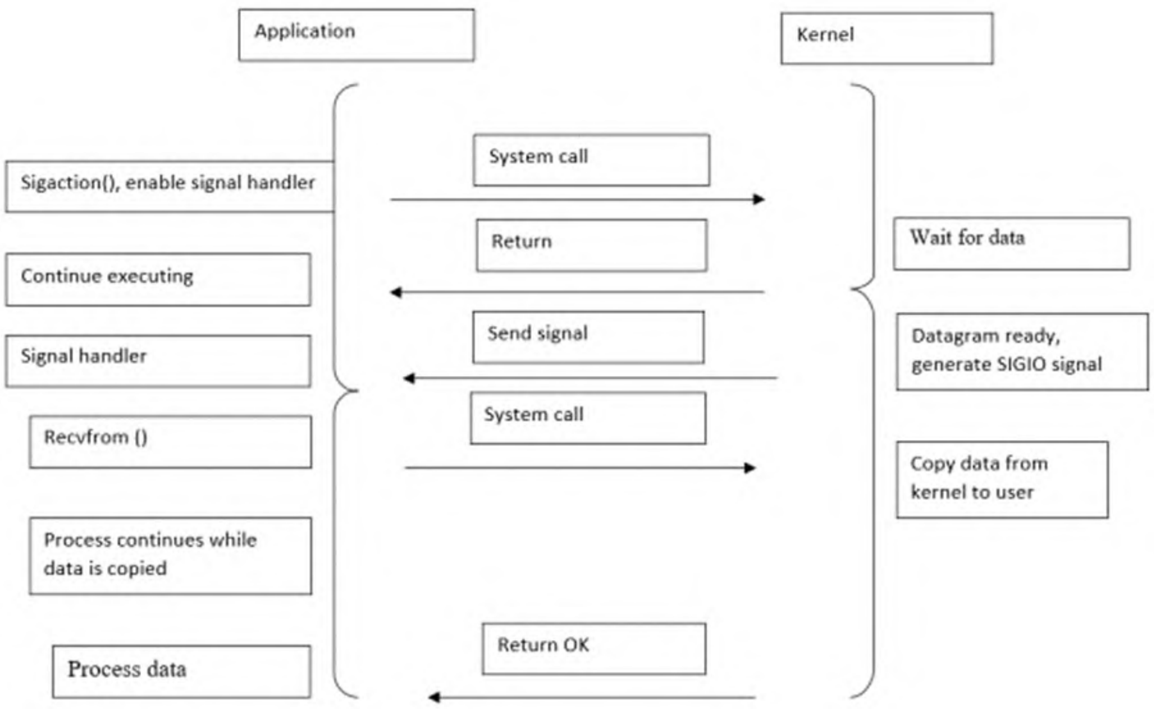
Application
Kernel
Sigaction(), enable signal handler
System call
Return
Wait for data
Continue executing
Signal handler
Send signal
Datagram ready, generate SIGIO signal
Recvfrom ()
System call
Copy data from kernel to user
Process continues while data is copied
Return OK
Process data

Figure 5.5: *Block diagram of signal-driven I/O model*

Initially, the socket is made enabled for signal-driven I/O. The **`sigaction()`** system call is executed to install a signal handler for handling I/O. Once a handler is installed, the system calls to return and does not block the process. Once the datagram is ready, the Kernel sends a SIGIO signal to the process. The process then reads the datagram either from the signal handler by calling **`recvfrom()`** system call, or the main function of the process can also read the datagram.

In this mode, the process is not blocked while waiting for the datagram to arrive; rather, it continues executing. Once the datagram is ready to arrive, the process is notified by the signal handler.

Asynchronous I/O model

In this mode of I/O, the **`io_read()`** function is called defining the descriptor, buffer pointer, buffer size (same as in **`read()`** function) to the Kernel, file offset, and the method of notification of data arrival. When the entire operation is complete, the system calls immediately. This does not require blocking the process while waiting for the I/O to complete. The system model is illustrated in *figure 5.6*.

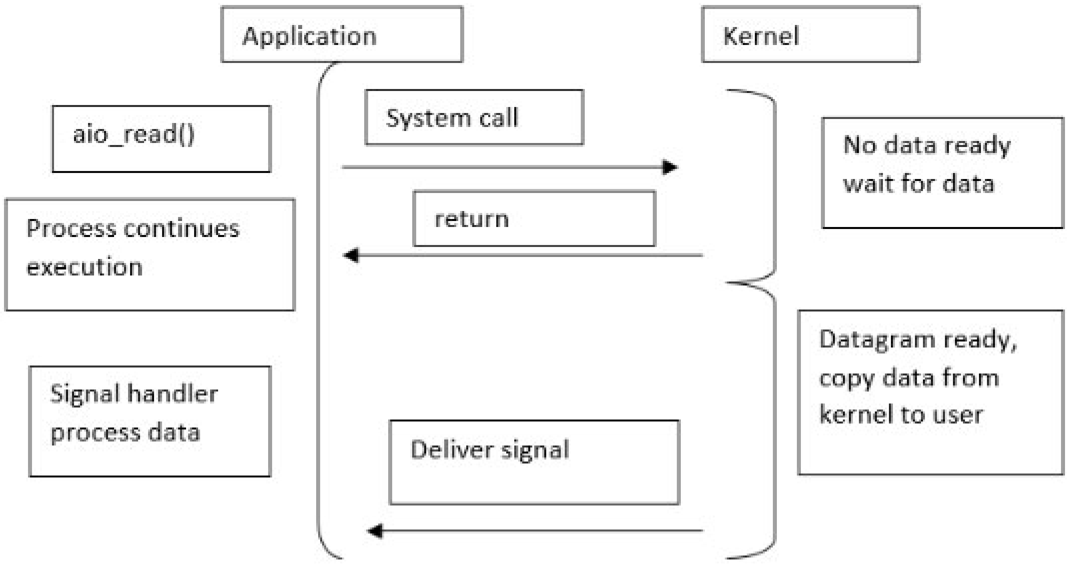
Application
Kernel
aio_read()
System call
No data ready wait for data
return
Process continues execution
Datagram ready, copy data from kernel to user
Signal handler process data
Deliver signal

Figure 5.6: *Block diagram of asynchronous I/O model*

This model is quite different from signal-driven I/O as the signal is generated until data has been copied into the application's buffer.

Name and address conversion

DNS is used to map between hostnames and an IP address. The hostname can be either a simple name (solaris) or a Fully Qualified Domain Name (FQDN) (solaris.kohala.com).

Resource records

DNS entries are called Resource Records (RRs). It defines all the information about the domain name, including the type of resource, protocol family, time in seconds during which a server can cache the RR, length of the data field, and resource data. The following table describes different types of resource records.

Resource record type	Mapping
A	Maps hostname into 32-bit IPv4 address

AAAA	Maps hostname into a 128-bit IPv6 address
PTR	Map IP address into a hostname. For IPv4, the 4 bytes are reversed, and each byte is converted into a decimal ASCII value (0-255). Then in-`addr.arpa` is appended. For an IPv6 address, the 32 4-bit nibbles of the 128-bit address are reversed. Each nibble is converted to its corresponding hexadecimal ASCII value (0-9-a-f), and `ip6.int` is appended.
MX	Mail exchange for the specified host.
CNAME	Canonical name, for example, assigns CNAME to common services such as FTP, WWW, and so on.

Table 5.2: *Resource types with their mapping*

The gethostbyname() function

The **`gethostbyname()`** function takes the hostname of the current host and returns the corresponding IP address to pass as a parameter to the **`connect()`** function. It performs a query for an A and AAAA records and returns IPv4 and IPv6 addresses.

Prototype

```
#include <netdb.h> /* Berkeley */
struct hostent *gethostbyname (const char *hostname);
```

The argument to the **`gethostbyname()`** is a string like **`www.yahoo.com`**, and it returns a struct hostent, which contains a huge amount of information, including the IP address. Other information is the official hostname, a list of aliases, the address type, the length of the addresses, and the list of addresses.

The hostent structure

The hostent structure contains information about host in hosts database. The elements of the hostent structure are: host name, alternative name of the host, type of address type IPv4 or IPv6, length of address in bytes, address vector of hosts.

```
/* <netdb.h> */
struct hostent { /* structure returned by network */
char *h_name;  /* official name of host */
char **h_aliases;  /* alias list */
int h_addrtype;  /* host address type */
int h_length;   /* length of address */
char **h_addr_list;  /* list of addresses from name server */
};
/* h_addrtype */ /* always is AF_INET */
/* h_length */ /* always is 4 */
/* h_addr_list[] */ /* for multinode host */
void main()
{
int i;
struct hostent *he;
struct in_addr addr;
// get the addresses of www.yahoo.com:
he = gethostbyname("www.yahoo.com");
if (he == NULL) {
  herror("gethostbyname"); // herror(), NOT perror()
  exit(1);
}
// print information about this host:
printf("Official name is: %s\n", he->h_name);
printf("IP address: %s\n", inet_ntoa(*(struct in_addr*)he-
>h_addr));
printf("\n");
exit(0);
}
```

The output is the official name and IP Address of the requested URL.

The gethostbyaddress() function

The **`gethostbyaddr()`** returns a hostent structure containing the host's name and other information.

Prototype

```
#include <netdb.h>
struct hostent *gethostbyaddr(const char *addr, int len, int
type);
```

For TCP/IP, the first argument **`addr`** should point to struct **`in_addr`** or an unsigned long integer in network byte order, **`len`** is normally the **`sizeof(struct in_addr)`**, and type should be **`AF_INET`**.

Host information is found either through the resolver or in your system's equivalent of the **`/etc/hosts`** file.

```
main()
{
  const char *ipstr = "66.94.230.32";
  struct in_addr ip;
  struct hostent *hp;
    if (!inet_aton("66.94.230.32", &addr);
      errx(1, "can't parse IP address %s", ipstr);
    if ((hp = ethostbyaddr(&addr, sizeof addr, AF_INET); ==NULL)
      errx(1, "no name associated with %s", ipstr);
  printf("Host name: %s\n", hp->h_name);
exit(0);
```

Resolver

The **`gethostbyname()`** and **`gethostbyaddress()`** functions use **`resolver()`** library function to get host information. The **`resolver()`** function reads the file **`/etc/resolv.conf`** to get the IP address of the DNS server to start from.

The getservbyname() and getservbyport() functions

The **`getservent()`** functions reads the service database to identify server to open the connection.

The **`getservbyname()`** searches the service database to identify the service that matches with the name specified and service protocol and opens a connection if necessary.

The **`getservbyport()`** searches the service database to identify the service with given port and specified protocol and opens the connection if necessary.

All these three functions search the database in a sequential manner until a match is found or until EOF occurs.

Prototype

```
#include <netdb.h>
struct servent *getservbyname (const char *servname, const
char *protname);
struct servent *getservbyport (int port, const char
*protname);
```

The **servent** structure is as follows:

```
struct  servent {
   char    *s_name;    /
   char    **s_aliases;
   int         s_port;
   char    *s_proto;
   };
```

The members of this structure are:

- **s_name**: The official name of the service.
- **s_aliases**: A zero-terminated list of alternate names for the service.
- **s_port**: The port number at which the service resides. Port numbers are returned in network byte order.
- **s_proto**: The name of the protocol to use when contacting the service.

These functions use static data storage; if the data is needed for future use.

Conclusion

The socket is an interface between client and server either residing in a single system, on a network, or across different networks to send and receive information. The client program sends a request of data to the server through connections oriented or a connection less interface. In this chapter, all concepts related to socket programming have been discussed. These include different types of sockets, functions to establish a connection, and sending and receiving messages between client and server. In this chapter, we discussed about socket programming for communication in a UNIX environment. There various other environments are also available for the

UNIX platform user may customize these environments. We will discuss these environments in the upcoming chapter.

Review Exercise

1. From the command line, read (1) the URL from which you can extract the name of the remote WWW server and the file to retrieve and (2) the server port number. Create a socket that is connected to the server machine at the specified port (e.g., HTTP port 80) [getservbyname, gethostbyname, socket, connect].
2. What information does a process running on one host to identify a process running on another host use?
3. Does a server program request and receive services from a client program?
4. Create the socket using socket(), convert the hostname to an IP address using gethostbyname(), and then issue the connect() call, passing the relevant structures containing the IP address and port to connect to.

CHAPTER 6

Memory Management

Memory is considered as an important resource in computers, and memory management is the process of managing the computer memory, which consists of primary memory and secondary memory. The purpose of memory management is to efficiently allocate the different types of available memories among processes that are running or residing in the system. It keeps track of which parts of memory are in use and which parts are not in use to allocate memory to processes when they need it. The memory allocated to the running process is deallocated once the process terminates so that it can be given to another process requesting the memory resource. UNIX operating system implements swapping and demand paging techniques to manage the system's memory. This chapter discusses all concepts and techniques used for the proper management of memory resources.

Structure

In this chapter, we will cover the following topics:

- Memory management
    - Use of operating system memory
    - Memory contents of the running process in UNIX
    - Swapping
    - Implementation of memory management functions

Memory management functions

- Setting branch to another function

Objective

After reading this chapter, the reader will be able to know how UNIX is written in C and how UNIX operating system is used extensively for programming in C. In this section chapter, we will study some UNIX and C interface concepts such as:

- To understand the use of different types of memory resources in the UNIX operating system.
- To understand different memory management techniques such as swapping and demand paging.
- To understand system calls and memory management functions.
- To understand context switching during the function call.

System calls and library functions:

- Command-line arguments and environment variables.
- multitasking in C in UNIX using `fork()`, `exec()`.
- Process synchronization using `wait()`.
- Accessing user information using system calls.
- Sharing of data and inter-process communication mechanisms.
- Socket programming fundamentals.

Memory management

The memory management component of the UNIX Kernel manages the contents of processor memory. UNIX-like operating systems use sophisticated memory management algorithms for the efficient use of memory resources. In UNIX, there are three different types of memory resources, including main and secondary memory:

- **Main:** Physical memory available on the CPU motherboard is called main memory. This does not include processor caches, video memory, or other peripheral memory.
- **File system:** Physical memory, available on the local disk or on the network and accessible through pathname. This does not include raw devices, tape drives, swap space, or other storage not addressable via normal pathnames.

- **Swap device:** A swap device is a physical memory available on the disk to hold data that is not in RAM or in the file system. A swap device can be used efficiently when it is on a separate disk or partition.

Use of operating system memory

For efficient management of memory resources, it is kept reserved for both the operating system’s own functions and for the processes being executed. Some part of memory is always reserved and cannot be used for users’ processes.

- **Kernel memory: I**t is the operating system’s private memory always available on the main memory.
- **Cache memory:** It is part of the main memory to hold elements of the file system and I/O operations. Cache memory does not include CPU cache disk drive.
- **Virtual memory:** Virtual memory includes the total addressable memory space of all processes running on the machine. The total address space of the process can be spread over all three types of memories.

Memory contents of a running process in UNIX

A process on UNIX uses memory to hold its data, stack, and mapped files:

- **Data segment:** This includes allocated memory to hold process data.
- **Stack segment:** This segment of the process holds the execution stack and is fully managed by the operating system.
- **Mapped files:** This section of the process includes files contents accessed within the process memory space.

Swapping

Swapping occurs when a running process needs a virtual page in the physical memory, and there is no free space available in the physical memory. In this situation, the operating system moves out some of the pages from the main memory and brings in the required pages from the swap device. The decision regarding which pages to be swapped in or swapped out is done by the scheduler (swapper). The scheduler wakes up at least once every 4 seconds to check which processes can be swapped out. A process that has been idle in the main for a long time is most likely to be swapped out. If no such process is there in the main memory, then a large process is selected for swapping out. A process that has been swapped out for a long time and small is most likely to be swapped in. To prevent thrashing, processes that have not been in the main memory for a certain amount of time are not selected for swapping out. Process pages selected for swapping out to the location depending on their type. Pages from the data segment are moved to swap space, pages from the stack segment are moved to swap space, and pages from the mapped file are moved to the originating file if changed and shared. The pages from the mapped file are moved to swap space if changed and private. If pages from mapped files are unchanged, they are discarded.

Demand paging

Demand paging is a paging policy in which a page is not read into the memory until it is requested. Demand paging transfers only required memory pages instead of a process to and from a secondary device.

The Kernel implements a page table to achieve demand paging. Page table converts logical page address to physical page address. The page table uses a binary flag to check the page's validity. A page is valid if it resides in the main memory and invalid if it currently resides in secondary memory. When a process needs a page, and the page is not there, a page fault interrupts to the Kernel occurs. Kernel checks if page reference is valid in the secondary memory, it schedules disk read operation to swap in the demanded page. Once the page is in the main memory, it restarts the interrupted instructions.

Note: Berkeley introduced demand paging to UNIX with BSD (Berkeley System).

Advantages of the demand paging policy are that it permits greater flexibility in mapping the virtual address of a process into the physical memory of a machine, usually allowing the size of a process to be greater than the amount of availability of physical memory and allowing a greater number of processes to fit in the main memory simultaneously.

Disadvantages of the demand paging policy are that a single process may face extra latency when a process requires a page first time. More page faults may cause thrashing, and the process may face the security risk of a timing attack.

The mkdir system call

The **`mkdir()`** function creates a new directory named by the pathname pointed to by path.

Prototype:

```
int mkdir(const char *path, mode_t mode);
```

The mode of the new directory is initialized from mode. The protection part of the mode argument is modified by the process's file creation mask.

The directory's owner ID is set to the process's effective user ID. The directory's group ID is set to the process's effective group ID, or if the **`S_ISGID`** bit is set in the parent directory, then the group ID of the directory is inherited from the parent. The **`S_ISGID`** bit of the new directory is inherited from the parent directory.

If path names a symbolic link, **`mkdir()`** fails and sets **`errno`** to **`EEXIST`**.

The newly created directory is empty with the exception of entries for itself (.) and its parent directory (..).

Upon successful completion, **`0`** is returned. Otherwise, **`-1`** is returned, no directory is created, and **`errno`** is set to indicate the error.

The rmdir system call

The **`rmdir()`** system call removes a directory file whose name is given by path. The directory must not have any entries other than "." and "..".

Prototype:

```
int rmdir(const char *path);
```

The **`rmdir()`** function returns the value **`0`** if successful; otherwise, the value **`-1`** is returned, and the global variable **`errno`** is set to indicate the error.

Before deleting a directory entry, the system checks the link count entry (**`ls -l`** command) to identify whether the directory to be deleted is opened by some other process or not. The directory is deleted only if the link count of the directory is 0 with this call and no other process has the directory open, then the space occupied by this directory is freed.

Memory management functions

Memory to the variables can be allocated and freed dynamically.

The malloc() function

`malloc` requires one argument - the number of bytes you want to allocate dynamically.

If the memory allocation is successful, **`malloc`** will return a void pointer. You can assign this to a pointer variable, which will store the address of the allocated memory.

If memory allocation fails (for example, if you are out of memory), malloc will return a NULL pointer.

The free() function

Passing the pointer into **`free`** will releases the allocated memory. It is good practice to free memory when you have finished with it.

```
void free(void *ptr);
```

Where **`ptr`** is the pointer.

Consider the following Example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  int number;
  int *ptr;
  int i;
  printf("How many ints would you like store? ");
  scanf("%d", &number);
  ptr = malloc(number*sizeof(int)); /* allocate memory */
  if(ptr!=NULL) {
    for(i=0 ; i<number ; i++) {
      *(ptr+i) = i;
    }
    for(i=number ; i>0 ; i--) {
     printf("%d\n", *(ptr+(i-1))); /* print out in reverse order
     */
    }
    free(ptr); /* free allocated memory */
    return 0;
  }
  else {
    printf("\nMemory allocation failed - not enough memory.\n");
    return 1;
  }
}
```

The calloc() function

`calloc` is similar to **`malloc`**, but the main difference is that the values stored in the allocated memory space are zero by default. With malloc, the allocated memory could have any value.

`ptr = (cast_type *) calloc (n, size)`; **`calloc`** requires two arguments. The first is the number of n variables you had like to allocate memory for. The second is the size of each variable represented as size. Like **`malloc`**, **`calloc`** returns a void pointer **`ptr`** if the memory allocation was successful; else, it returns a NULL pointer.

Consider the following example:

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  float *ptr1;
  int i;
  ptr1 = calloc(3, sizeof(float));
if(ptr!=NULL ) {
    for(i=0 ; i<3 ; i++) {
     printf("ptr1[%d] holds %05.5f, ", i, ptr1[i]);
    }
    free(ptr1);
       return 0;
  }
  else {
   printf("memory not successfully allocated\n");
   return 1;
  }
}
```

The realloc() function

The **`realloc()`** function is used to change the size of the already allocated block of memory without losing data in that memory location.

```
ptr = realloc (ptr,newsize);
```

`realloc` takes two arguments. The first is the pointer referencing the memory, that is, **`ptr`** whose size is needs to be changed. The second is the total number of bytes that needs to be reallocated, that is, **`newsize`**.

Passing zero as the second argument is the equivalent of calling free, **`realloc`** returns a void pointer if successful, else a NULL pointer is returned.

```
#include<stdio.h>
#include <stdlib.h>
int main() {
  int *ptr;
  int i;
  int number;
  printf("How many ints would you like store? ");
  scanf("%d", &number);
  ptr = calloc(number, sizeof(int));
  if(ptr!=NULL) {
for (int I =0; I<number ;I++)
     *(ptr + I) = i;
     ptr = realloc(ptr, (number 2 )*sizeof(int));
     if(ptr!=NULL) {
      printf("Now allocating more memory... \n");
        ptr[number] = 32; /* now it's legal! */
      ptr[number ++1] = 64;
        for(i=0 ; i<(number +2 ); i++) {
      printf("ptr[%d] holds %d\n", i, ptr[i]);
     }
      realloc(ptr,0); /* same as free(ptr); - just fancier! */
        return 0;
     }
     else {
      printf("Not enough memory - realloc failed.\n");
      return 1;
     }
   }
   else {
     printf("Not enough memory - calloc failed.\n");
     return 1;
   }
}
```

The alloca() function

The **alloca()** function allocates space in the stack frame of the caller function. This temporary space is automatically freed when the function that is called **alloca()** returns to its caller. The **alloca()** function returns a pointer to the beginning of the allocated space. If the allocation causes stack overflow, program behavior is undefined.

The **alloca()** function is machine and compiler-dependent. On many systems, its implementation is buggy.

Prototype:

```
#include <alloca.h >
void *alloca(size_t size);
```

Setting branch to another function

C program does allow to jump to the label set in another function. To branch the control to another function **setjump** and **longjump** functions are used. Generally, these kinds of jumps should be avoided because it is not considered as a good programming practice.

Prototype:

```
int setjmp(jmp_buf env); void longjmp(jmp_buf env, int val);
```

Consider the following example:

```
#include <setjmp.h>
#include <stdio.h>
jmp_buf ebuf;
void f(void);
int main(void)
{
 int i;
 printf("1 ");
 i = setjmp(ebuf);
 if(i == 0) {
  f();
  printf("This will not be printed.");
 }
 printf("%d", i);
 return 0;
}
void f(void)
{
 printf("2 ");
 longjmp( ebuf, 3);
}
```

Conclusion

Memory is considered as an important resource of the computer system. If memory is not managed properly, it can degrade the overall performance of the operating system. The memory management function is used to keep track that which part of memory is used and how much and which part of memory is free to assign to the processes requiring memory space. The memory management function controls both primary and secondary memory available in the system. The UNIX operating system implements swapping and demand paging techniques for efficient use of system memory. The memory can be dynamically allocated and deallocated as and when required.

Multiple choice questions

1. **Which technique allows the execution of processes that may not be completely in memory?**
    a. Virtual memory
    b. Memory management
    c. Physical memory
    d. None of the above
2. **Which of the following are all UNIX system calls?**
    a. time(), chdir(), createdir(), execve()
    b. chmod(), main(), waitpid(), exit()
    c. (c) open(), fork(), delete(), read()
    d. lseek(), stat(), link(), kill()
3. **The mechanism that brings a page into memory when there is a page fault.**
    a. Demand paging
    b. Segmentation
    c. Fragmentation
    d. Page replacement
4. **The virtual memory manages the logical address space of a process.**
    a. True
    b. False
5. **Which header file is required to be included to access dynamic memory allocation functions?**
    a. stdlib.h
    b. stdio.h
    c. memory.h
    d. alloc.h
6. **Which of the following functions is used to modify the size of dynamically allocated memory?**

   a. realloc()
   b. malloc()
   c. free()
   d. alloca()

7. **Indicate which amongst the following statements are true for virtual memory.**
   a. It allows for multiple users to use the system
   b. It enhances the scope for multi-programming
   c. It extends the address space
   d. It reduces external fragmentation as well as internal fragmentation.

Answers

1. **(a)**
2. **(c)**
3. **(b)**
4. **(b)**
5. **(a)**
6. **(a)**
7. **(b)**

Questions

1. What is the main goal of memory management?
2. What is the difference between swapping and paging?
3. What are the processes that are not bothered by the swapper? Give reason.
4. Why UNIX does not allow the paging-out of Kernel memory?

CHAPTER 7

UNIX Shell and Custom Environment

As we know, the most visible part of the UNIX operating system is the shell, and this makes it the most important component to be known to the users. To perform any task, we must give a command to the shell. If the command needs a utility or an application program, the shell requests Kernel to execute it. A shell has two parts; the first is an interpreter, which takes the user's command and converts it to the system's understandable form. The second component is the shell's programming capability that allows us to the right shell script. Another important function performed by the shell is customizing the environment to work in. In this chapter, we will closely look at the types, features, and functions of the shell as an interpreter and environment customizer.

Structure

We will cover the following topics in this chapter:

- Functionality and execution modes of UNIX's shell.
- Shell execution cycle
- Features provided by an available standard shell in the UNIX
- Significance and uses of shell metacharacters
- Avoiding interpretation of meta characters using escaping and quoting
- Command input/output using three standard files
- Manipulating default input and output streams using redirections
- Use of noclobber option
- Use /dev/null files
- Grouping commands together
- Significance of pipelining different commands together
- Customizing environment using shell environment variables

- Understanding and customizing user’s profiles

Objectives

After going through this chapter, you will be able to:

- Understand the fundamentals and functionality of shell
- Understand different functional modes of the shell
- Understand concepts involved in executing shell commands
- Understand command grouping and pipelining
- Understand user’s profiling

Introduction to shell

A shell is simply a program that acts as an interface UNIX. The shell is often called as a command interpreter because it allows the system to understand the user’s commands. There are several shells provided by the system, each having different features, but all of them affect how commands are interpreted. The shell provides tools to create the UNIX environment.

The shell is a utility in UNIX. It is loaded into the memory when the user logs in to the system. The shell to be executed during the login process is listed in **`/etc/passwd`** file corresponding to the respective entry of the user. If no shell is specified in the **`/etc/passwd`** file, the standard shell **`/usr/bin/sh`** is executed by default.

Execution modes of UNIX shell

UNIX shell is basically executed in the following three mode:

1. Interactive mode
2. Shell programming mode
3. UNIX session customization mode

Interactive mode

In the interactive mode, the shell waits for the user to enter the command, which is then interpreted by the shell. The command gets executed after removing all errors. The command may include special symbols to abbreviate filenames and to redirect input and output.

Programming mode

In the programming mode, the system executes a set of UNIX commands together in the form of a shell script. Many built-in commands are grouped together in the form of a shell script. In contrast with the interactive mode, the script can be preserved and can be executed layer also.

UNIX session customization mode

UNIX shell controls the session behavior through predefined shell variables. Shell variables are used to set the user's home directory, prompt symbol, setting mails, and so on. Some shell variables are automatically set by the system, whereas the user can set other shell variables in startup files.

Shell interpretation cycle

Shell command processing is generally performed in five steps:

1. The shell displays a prompt symbol on the terminal as soon as it starts.
2. The shell simply waits for the user to enter the command to be executed.
3. The shell analyzes the command entered by the user.
4. The shell proceeds to carry out the user's request. If a particular program is required to be executed, the shell searches the disk to find the specified program.
5. Once the program is found shell asks the Kernel to initiate the program execution and goes to sleep until the program execution is completed. In case the program reads input from the standard input, it waits for the user to type input on the command prompt during execution.

Once the program execution is over control again returns to the shell to execute the next program/command.

Functions of UNIX shell

Figure 7.1 shows the basic functions performed by the shell. Further details of the functions are given in subsequent sections.

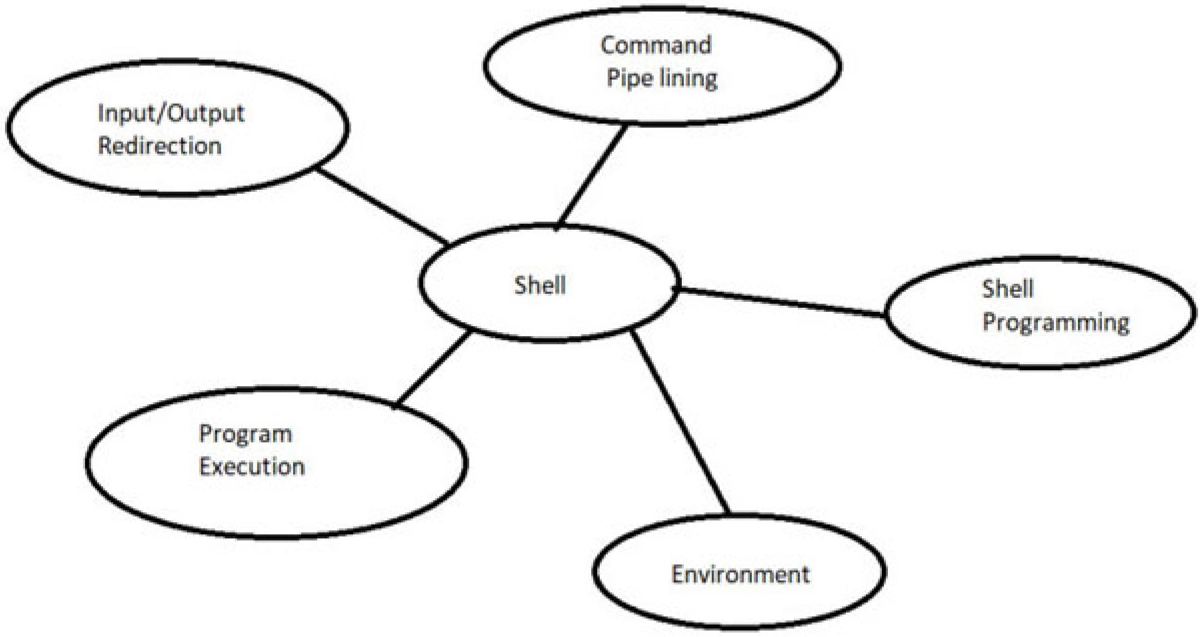

Figure 7.1: *Uses of shell*

Program execution

Shell initiates the execution of all programs requested from the user terminals. The program to be executed is given as a command line on the terminal as follows:

Program-name list-of-arguments

Shell scans the command line and identifies the name of the program to be executed and the values of parameters passed to execute the program.

Till the first white space, the shell takes everything as the program name; each argument is also separated by white space. If multiple white spaces are given between program name and argument, the shell ignores extra white spaces.

Example:

```
prog.sh  first_arg second_arg
```

- `prog.sh`: Program name
- `first_arg second_arg`: Argument list

Input/output redirection

Shell takes care of input/output redirection on the command line. During the execution of the program, the shell scans the input command for special redirection operators <, >, and >>.

Shell has the ability to redirect input and/or output from and/or to files.

During the execution of the program shell takes care of input/output redirection on the command using special redirection operators <, >, and >>. The < operator takes the input to the command from the file given on the right-hand side. The > redirects the output of the command to a file. The >> append s output of the command at the end of an existing file.

```
ls  -l >  file1
```

In the preceding example output of `ls -l` command is redirected to file1. If file1 already exists, then its contents are replaced by the current output. The command being executed is not concerned with where the output is redirected.

Let us consider two exactly same commands:

```
wc -l  file1
output : 10
wc -l < file1
output : 10 file1
```

In the first case shell determines the command `wc` to be executed for counting the number of lines in the file `file1`; it opens the file file1 and counts the number of lines.

In the second case, `wc` executes slightly different. The shell determines the input redirection operator < when it scans the command line, the word following < symbol is the name of the file from which input is to be redirected. The shell executes `wc` command, redirecting its standard input from the file `file1`.

Command pipelining

Like redirection operators shell determines the | symbol while scanning the command line. For each such symbol, the shell transfers the standard output of the command preceding | to the command following | symbol.

Let us consider the command:

```
ls -l | wc -l
```

The preceding command outputs the total number of files in the current directories. The command `ls -l` lists files and directories in the current directory and writes the result to the standard output, unaware of the | symbol; the output of the command is then transferred as standard input to the command `wc -l`, which, when then executes and determines that there is the no of files provided as input and calculates a total number of lines on the standard input.

Environment customization

Shell environment contains a set of predefined variables. These variables are called environment variables. Shell keeps these variables to control and configure behavior of utility programs. The value of shell variables is generally set outside the program by the operating system or through microservices. In UNIX, there is a large set of environment variables that includes:

- Path to the user's home directory: (HOME)
- Search for shell command (PATH)
- Default prompt (PS1)
- The terminal type (TERM)
- UNIX hostname (HOSTNAME)

Shell provides commands to customize the environment.

Shell programming language

Shell has its own programming language to write shell scripts. In contrast with other functional programming languages such as C and C++, where the whole program is compiled and converted into execution code, the shell programming language is interpreted, which means each statement written in a shell programming language is interpreted and checked for error once and executed.

Shell programming involves using the shell commands and shell programming language to write shell scripts. A shell script is a series of commands written in plain text commands. The commands may be internal or external shell commands and shell programming constructs such as if-then-else construct, loops, case construct, variables, expressions, and so on.

Types of shells

Like most of the operating systems, the UNIX shell does not built-in rather shell in UNIX is just another program. UNIX shell protects the Kernel as well as the user from each other. There are several shells provided by the system, each having different features. Different shells can be considered in two shell families—the Bourne shell family and the C shell family.

The Bourne shell family

The Bourne shell family consists of three shells the Bourne shell (/bin/ sh) and the two derivatives of the Bourne shell—Korn shell (/bin/ksh) and Bourne again shell /bin/bash (BASH)

The Bourne shell

The Bourne shell was developed by Steve Bourne AT&T labs is the oldest UNIX shell. This comes bundled with almost all UNIX versions. The unenhanced version of the Bourne shell is BASH, which is used in LINUX. The Bourne shell commands are algorithm-style programming language (ALOGOL) style.

The Korn shell

The Korn shell was developed by David G Korn at AT&T labs. It is a superset of the Bourne shell and has many more capabilities. The Korn shell incorporates all interactive features of the C shell in the Bourne shell's syntax. Korn shell adds the following new features to the Bourne shell:

- Arrays

- Filename completion
- Command aliases
- Functions
- Command history and substitution

Another variant of the Korn shell is POSIX, available with HP-UX 11.0. In HP-UXPOSIX is installed in **`/bin/sh`** and Bourne shell is installed in **`/usr/old/bin/sh`**.

The BASH (Bourne again) shell

The bash was developed by Brian Fox of the free software foundation. It is currently maintained by Chester Ramey. The unenhanced version of the Bourne shell is BASH, which is used in LINUX.

Bash provides the following few additional features to the Korn shell:

- An array of unlimited sizes.
- Correction of the pathname in cd command.
- Integer arithmetic in any base between 2 and 64.
- Completion of filename, hostname, and variable name

The C shell family

The C shell family consists of a C shell (/bin/csh) and its derivative tcsh (/bin/tcsh).

The C shell was developed by BILL joy in Berkeley and contained the command, which looks like C statements. The compatible version of the C shell is tcsh, which is used in LINUX. The C-like commands in the C shell were intended to improve programming because programmers were familiar with the C language at Berkeley. TENEX/TOP is the new version of the C shell, which allows scrolling through the command history, and command listing is possible through arrow keys.

The C shell has become popular for interactive use because of its improved features:

- **Command history:** Previously executed commands can be recalled for further execution. These commands can be re-edited before execution.

- **Command aliases:** Short mnemonic names can be given to the commands.
- **Filename completion:** C shell automatically completes the filename by just writing a few characters of the filename.
- **Job control:** The C shell executes multiple processes. The concurrent processes are controlled using jobs commands.

The C shell is used for writing scripts because of the following reasons:

- Lazy interpreter
- Lack of functions
- Weak input and output

Summary of features

The following table shows the summary of features provided by the different shells:

	Command history	Command alias	Shell scripts	Filename completion	Command-line editing	Job control
Bourne	N	N	Y	N	N	N
C	Y	Y	Y	Y*	N	Y
TC	Y	Y	Y	Y	Y	Y
Korn	Y	Y	Y	Y*	Y*	Y
BASH	Y	Y	Y	Y	Y	Y

***Table 7.1:** Shell wise features*

Meta character and wild cards

Shell metacharacters are special symbols that the shell interprets rather than passing them to the command.

The list of metacharacters is as follows:

Meta character symbol	Meaning

?	Filename substitution wild card single character
*	Filename substitution wild card multiple characters
[]	Filename substitution wild card any character between brackets
<	Input redirection
<<	Input redirection (here, document)
>	Output redirection
>>	Output redirection (append)
#	Comment
()	Command grouping
“cmd”	Command substitution
$(Cmd)	Command substitution
|	Pipe
;	Command sequence
&	Run command in the background
||	OR logical operator
&&	AND logical operator
$	Expand the value of the variable
\	Avoid interpretation of the next character

***Table 7.2:** Metacharacters for file name substitution*

Sometimes it is required not to interpret these symbols by the shell. Shell provides three different options to prevent the interpretation of these metacharacters (discussed in the next section).

Command standard input/output

In the UNIX system, a program including many UNIX commands is aromatically connected to the terminals. Terminals are represented by generic files. A command takes input from a generic file when no filename/pathname is specified and sends the output to the generic file when no filename/pathname is specified. By default generic file (terminal) through which input is provided is called standard input; the generic file (terminal) to which the command's output is displayed is called standard output and sends any error message to standard error. When the command is reading input from standard input and writing output to the standard output, it means that it is reading from the keyboard and writing to the screens, as shown in *figure 7.2*.

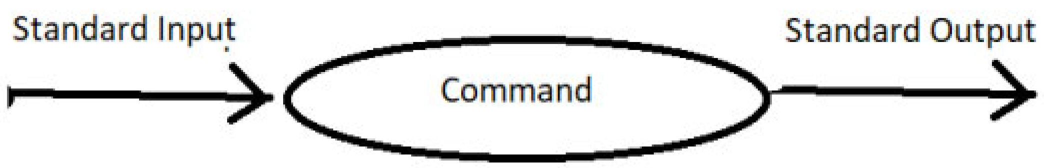

Figure 7.2: *Standard I/O*

In UNIX, standard input is accessed by a generic file named stdin, standard output is accessed by a generic file named **`stdout`**, and standard error is accessed by a generic file named stderr. In UNIX, every file has a number associated with it. All operations performed on the file are actually performed on the associated file descriptor. The generic file for standard output, standard input, and standard error is always open and access to the command or program that runs. These generic files also have associated file descriptors to be accessed and modified. The file descriptors reserved for standard I/O are as follows:

Filename	**File descriptor**
`stdin`	0 (Keyboard)
`stdout`	1 (Screen)
`stderr`	2 (Screen)

Table 7.3: *File descriptors for standard files*

Note: Some commands do not take input from standard input.

Example: cat file1

Redirection

By default, each command or program is connected to a standard file for input and output. This default assignment can be changed temporarily using redirection. Redirection is the process of specifying which file will be used in place of any of the standard generic files to provide input or to take the output from the command.

Output redirection

Execution of the program always generates some output on the shell prompt; however, there are some situations when output is required to be directed elsewhere, such as:

- If output of the program is required to be transferred directly to the printer.
- If the output is required to be used later.
- If a lot of output is generated from the program, it is difficult to see all the output because the screen scrolls very rapidly.

In all these situations, the output can be redirected elsewhere, either to the file or to the printer.

There are two basic redirection operators for standard output > and >>. Think > symbol as an arrow pointing output of the command to the file that is to receive the output. The choice of operator used will be dependent on how the user wants to handle the output. If the user wants that file should contain the output of the current command only, the > operator will be used. In this case, if a file already contains something, that will be replaced by the output of the current command.

Example:

```
$ls -l > file1
```

Symbol > is recognized as the special output redirection, and the word next to this symbol is treated as the file name.

In the preceding example, the output of ls `-l` command is redirected to the file file1. If `file1` already exists, then its contents are replaced by the current output. The command being executed is not concerned with where the output is redirected.

Note: If file1 does not already exist, UNIX creates the file and copies the output.

If a user wants to append the output of the current command to the file, the token for redirection is two >> signs. If the file does not already exist, UNIX creates the file and redirects the output.

Consider an example of joining two files one after another:

```
$cat file1 >> file2
```

In the preceding example, the contents of **`file1`** will be copied to **`file2`**.

Safe I/O redirection with noclobber

The noclobber shell variable in c shell or noclobber option in bash and ksh prevents accidentally destroying the contents of a file. Reconsider the redirection example:

```
$ls -l > file1
```

In this example, if **`file1`** already exists, its contents are replaced by the output of the command **`ls -l`**. If noclobber option is turned on, it prevents destroying the contents of the file; instead, it generates an error message.

The command for setting noclobber is as follows:

```
% set noclobber ( C shell)
$ set -o noclobber
```

Note: If the user explicitly wants to replace an existing file during redirection, an exclamation (!) is required to be placed after the output redirection operator.

Example:

```
%  ls - l > ! file1
```

The following table shows the common standard I/O redirections:

Function	Csh	sh
Send stdout to file	prog > file	prog > file
Send stderr to file		prog 2> file
Send stdout and stderr to file	prog >& file	prog > file 2>&1

Take stdin from file	prog < file	prog < file
Send stdout to end of file	prog >> file	prog >> file
Send stderr to end of file		prog 2>> file
Send stdout and stderr to end of file	prog >>& file	prog >> file 2>&1
Read stdin from keyboard until c	prog << c	prog << c
Pipe stdout to prog2	prog | prog2	prog | prog2
Pipe stdout and stderr to prog2	prog |& prog2	prog 2>&1 | prog2

Table 7.4: *Common standard I/O redirections*

Input redirection

Similar to the output redirection, as a default, input to the program is given through the keyboard terminal. However, sometimes a large amount of input is required to execute the program. In this situation, the input can be stored in the file, and later it can be used as input to the program being executed. The input redirection operator is less than < symbol. Think of an arrow pointing to the command.

Example:

```
$ mail   George < mail_file
```

The preceding command mails the contents of the file **mail_file** to George.

Let us consider two exactly same commands:

```
Case 1:  wc -l file1
output: 10
Case 2:  wc -l < file1
output: 10 file1
```

In Case 1, the shell determines the command **wc** to be executed for counting the number of lines in the file **file1**; it opens the file **file1** and counts the number of lines.

In Case 2, **wc** executes in a slightly different manner. The shell determines the input redirection operator < when it scans the command line, the word following < symbol is the name of the file from which input is to be redirected. The shell executes **`wc`** command, redirecting its standard input from the file **`file1`**.

Note: Both input and output redirection cannot be used in a single command.

Redirecting error

A standard stream to display error generated by a program or command is by default combined with a standard output stream. Consider an example:

```
$ls  -l file1 file2
```

Output:

```
Cannot access file2. No such file or directory
-rwxr-xr-x 1 workshop acs 532 May 2015:31 . file1
```

Output and the error message are both displayed on the same screen. Users can redirect standard errors to the file and leave the output on the screen, or vice-versa is also possible. To do this user needs to specify the file descriptor along with the redirection operator. Let us reconsider the preceding example:

```
$ls -l  file file2 1> out_file
```

Output:

```
Cannot access file2. No such file or directory
$ cat out_file
-rwxr-xr-x 1 workshop acs 532 May 2015:31 . file1
```

The error message is shown on the screen to inform us that there is a problem while executing the command, and the output is redirected to the file **`out_file`**.

Both error message and output can be redirected to the file as follows:

```
$ls -l  file file2 1> out_file  2> err_file
$more err_file
Cannot access file2. No such file or directory
$more out_file
-rwxr-xr-x 1 workshop acs 532 May 2015:31 . file1
```

Redirecting to the one file

Both **STDOUT** and **STDERR** can be redirected to the same file. Consider an example:

```
$ls -l  1> out_file  2> &1
```

Discarding STDOUT or STDERR

Either or both **STDOUT** and **STDERR** can be discarded by redirecting to a null device accessed by a special write-only file **/dev/null**.

Example:

```
$ls -l > /dev/null 2>&1
```

Leftmost redirection sends **SDTOUT** to **/dev/null** file, and 2>$1 indicates that Channel 2 should be redirected to the same file as Channel 1.

Command grouping

Commands may be grouped in the following two ways:

- { command-list ; }
- (command-list)

In the first command, list is simply executed. The second form executes the command list as a separate process. For example, (**cd x; rm junk**) executes **rm junk** in the directory **x** without changing the current directory of the invoking shell.

The commands **cd x; rm junk** has the same effect but leaves the invoking shell in the directory **x**.

Command pipelining

Sometimes it is required to execute a series of commands to complete the task. Consider an example if the user wants to generate a hard copy of the list of files and directories in the current directory. To generate the list **ls -l** command will be used; the output of this command will be stored in some file using redirection, and then the **lpr** command will be used to print the file.

```
$ls -l > file1
$lpr file1
```

These two commands can be written as single command combining them using pipe (|) as follows:

```
$ ls -l | lpr
```

When a pipe is set up between two commands, the standard output from the first command is connected directly to the standard input of the second command. The output of the first command is not displayed on the screen because it is piped directly with the second command.

A pipe can be made between the two commands only if the first command writes its output to the standard output and the second command takes its input from the standard input.

Note: Pipe is an operator, not a command, which tells the shell to redirect the output of the first command as input to the second command.

Consider another example: if a list of files and directories in the current directory is very large, it scrolls up, and we can only see the last page of the output of `ls -l` command. We can use a pipe to see output one page at a time:

```
$ls -l | more
```

The tee command

The `tee` command copies output of the command or program to the standard output and, at the same time, copies it to one or more files. The tee command first creates the file to copy the output, and if the file is already existing, it overwrites the output. To prevent overwriting to an existing file, the `-a` option is used with the tee command. With `-a` option, the output is appended instead of overwritten.

Consider the following example:

```
$ls -l tee file_list
```

In the preceding example, a long list of files and directories in the current directory is displayed on the monitor as well it is copied to the file `file_list`.

UNIX environment variables

Environment variables are predefined variables. Environment variables control the user environment. In a shell environment, variables are commonly used to communicate to and from the shell. For example, setting a **`PATH`** variable tells the shell in which directory to look for the commands; the shell automatically sets the variable **`PWD`** when the current directory is changed.

Environment variables are managed by the shell and are inherited by all processes, including the sub-shell. The new process gets its own copy of these variables. These local copies can be read, modified, and passed on to the children of the processes.

You can set environment variables with a command like this:

In Bourne and Korn shell:

```
$ NAME=value
```

In C shell:

```
% setend NAME value
```

PATH variable

The **`PATH`** environment variable lists directory searched for commands. When a command is given at the prompt, the directories listed in the **`PATH`** variable are searched in the order specified.

Syntax for setting **`PATH`** variable is as follows:

```
$ PATH= :/usr/bin
```

If there are multiple directories, each will be separated by: can be specified as a search path. An empty string specifies the current directory.

Note : $0 displays the current shell of the user.

The external program “env” prints all the environment variables.

HOME variable

The HOME environment variable is set by the login process. It specifies the login’s home directory. When **`cd`** command is executed without any argument, it automatically goes to the home directory.

Prompt string 1 (PS1) variable

The variable PS1 is a set of characters specified as a prompt symbol. Generally, it is a "$" sign. However, this can be changed to anything the user wants.

Prompt string 2 (PS2) variable

The PS2 is displayed when the system thinks that the new line started without completing the command. Generally, it is > sign.

MAIL variable

`MAIL` variable contains the address of the mailbox. `MAIL` variable is set during the login process in the user's profile.

CDPATH variable

When cd command is executed to change the directory, the shell searches the directory inside the current directory. Variable `CDPATH` contains the list of additional directories to be searched to change the directory.

```
$CDPATH= $HOME
```

Example:

```
$cd ch_dir
```

Shell looks for the directory `ch_dir` in the current directory; if `ch_dir` is not found in the current directory, the shell searches the list of directories defined in `CDPATH` variables.

MAILCHECK variable

`MAILCHECK` variable specifies the time and how often the shell checks for the mail. The default value of this variable is 10 minutes for the Bourne shell. If this variable is not set, shell disables the mail checking.

Note: Specifying the current directory as a search path may be dangerous sometime; consider a condition if somebody creates a

program in the current directory, then this trojan horse will be executed in place of the actual who command

Note: If MAILCHECK is set to 0, the shell checks the mail whenever it prints the primary "prompt" symbol.

MAILPATH variable

Bourne shell checks a single directory set in the **`MAIL`** variable to provide mail notification. If the user wants to watch multiple mailboxes, **`MAILPATH`** variable is required to be set. **`MAILPATH`** variable consists colon of a separate list of files or directories to be checked to provide mail notification.

```
$MAILPATH=/usr/spool/mail/George:/usr/spool/mail/cust_query
```

NOTE : Shell ignores MAIL variable if MAILPATH variable is set. Both can't be used together.

In addition, the login process defines several other environment variables. **`TERM`** defines the terminal type, **`USER`** or **`LOGNAME`** defines **`userID`**, **`SHELL`** defines the default shell, and **`TZ`** defines the time zone.

User's profile

When a user logs in to the UNIX system, the shell displays the prompt symbol and waits for the user to type the command. However, before displaying the prompt, symbol shells executes two files.

The first file is /etc/profile, set up by the system administrator. This file contains system-wide default settings for all the users who login using Bourne shell (/bin/sh). This file usually performs things such as checking whether MAIL is set, default file creation mask (umask), and setting standard exported variables. Command in this file is simple variable assignments.

A typical **`/etc/profile`** file is as follows:

```
#Set file creation mask
unmask 022
#Tell me where new mail arrives
MAIL=/usr/mail/$LOGNAME
```

```
#Add my /bin directory to the shell search sequence
PATH=/usr/bin:/usr/sbin:/etc::
#Set terminal type
TERM=lft
#Make some environment variables global
export MAIL PATH TERM
```

The second file is .profile, set up by the system administrator and automatically gets executed in the user's home directory after the login process adds **LOGNAME** and **HOME** variables to the environment. This file contains the definition of several shell variables and exports them to customize the user's own environment. Because .profile is stored in the user's home directory, user can change their .profile to include whatever commands to be executed when the user logs in. Commands in .profile can override the settings in the **/etc/profile** file.

A typical .profile is as follows:

```
PATH=/usr/bin:/etc:/home/bin1:/usr/lpp/tps4.0/user::
epath=/home/gsc/e3:
export PATH epath
csh
```

The .profile is simply a shell script that gets executed when the user logs in. Regular UNIX commands can also be included in the .profile. Modifying .profile makes a customized UNIX environment for the user.

Note: .profile is not shown in the listing of files and directories in the home directory because it is a hidden file. To list it, ls –a command is used.

Conclusion

The shell is an important component of the UNIX operating system. Users can interact with Kernel through the shell. It acts as a command interpreter for the UNIX Kernel. In addition to command interpretation, the UNIX shell also provides programming capabilities in the form of shell scripts. Most of the system administration tasks are performed using shell scripts. The shell also controls environment variables. This chapter discusses all possible features and functions of the UNIX shell. The next chapter includes the details about shell programming constructs and writing shell script in detail. The details contains the logics, control and other commands used to develop programs to solve problems using shell script writing.

Test your skills

1. **Which of the following command displays the login shell?**
   a. $SHELL
   b. $0
   c. Echo $SHELL
   d. Echo $0
2. **Which of the following command displays the current shell?**
   a. $SHELL
   b. $0
   c. Echo $SHELL
   d. Echo $0
3. **What is the descriptor of standard output stream?**
   a. 0
   b. 1
   c. 2
   d. 3
4. **What is the descriptor of standard input stream?**
   a. 0
   b. 1
   c. 2
   d. 3
5. **What is the descriptor of standard error stream.**
   a. 0
   b. 1

    c. 2
    d. 3

6. **Which of the following command creates the Bash subshell?**
    a. Bash
    b. bsh
    c. Csh
    d. None of the above
7. **Assuming that dat_file is a text file, what is the error (if any) in each of the following commands?**
    a. Date | dat_file
    b. Dat_file | date

Answers

1. **e**
2. **a**
3. **b**
4. **a**
5. **c**
6. **a**
7. **b**

Review Exercises

1. Put a listing of the files in your directory into a file called filelist. (Then delete it!)
2. Create a text file containing a short story, then use the spell program to check the spelling of the words in the file.
3. Redirect the output of the spell program to a file called errors.
4. Type the command ls -l and examine the format of the output. Pipe the output of the command ls -l to the word count program wc to obtain a count of the number of files in your directory. Create your own bin

subdirectory in your home directory and make it part of the search PATH.

5. Explain why we cannot use input indirection with cal, date, man, and which commands.
6. Explain why we cannot use output indirection with the echo command?
7. If we use the following commands in sequence: Command 1 | Command2
   a. What is the necessary requirement for Command1?
   b. What is the necessary requirement for Command2?
8. One of the following commands works, and the other does not. Which one works, and what is the problem with other commands?
   a. date | more
   b. more | date
9. Can we use input indirection with vi command?
10. Briefly explain the concept of pipelining here or give reference to the earlier chapter (if it has already been explained) before proceeding with the commands.

Lab practice

1. Execute the who command and display the result in the file called file1. Use more to see the contents of the file.
2. Display the long list of files and directories in the current directory and send the output to the printer.
3. What is the difference between backslash, a double quote, and a single quote?
4. How can we make a duplicate of standard output and send them to two different files?
5. How following two commands are different from each other.
   a. cal > out_file
   b. cal 1 > out_file

6. Consider the command ls –l | tee
    a. Where does the input of the tee command will come from?
    b. How many output files will be created?

Environment variables are managed by your shell. The difference between environment variables and regular shell variables (**6.8**) is that a shell variable is local to a particular instance of the shell (such as a shell script), while environment variables are “inherited” by any program you start, including another shell (**38.4**). That is, the new process gets its own copy of these variables, which it can read, modify, and pass on in turn to its own children. In fact, every UNIX process (not just the shell) passes its environment variables to its child processes.

CHAPTER 8
Shell Programming Using Bourne Shell

The UNIX operating system has emerged as a standard operating system for the development of the application. The essence of the UNIX operating system exists in its efficient environment for application development. The strength of UNIX lies in its long and sophisticated list of more than 200 commands and also the way these commands can be combined to perform a more useful function in the form of shell scripts.

A shell simply accepts user commands and converts them into a system understandable form. Shell also provides fundamental programming concepts to make decisions control the execution of scripts and functions.

Structure

In this chapter, we will cover the following topics:

- Introduction to shell
- Uses of UNIX shell
- Writing shell script
- Understanding basic commands used in shell programming
- How to control the flow of logic using control commands in shell programming

Objective

After reading this chapter, you will be able to learn to:

- Advanced concepts of the shell as a programming language
- Features and mode of execution of shell
- Shell programming contracts

- Control commands in shell programming

Introduction to shell programming

A shell is simply a program that acts as an interface to UNIX. The shell is often called as a command interpreter because it allows the system to understand the user's commands. There are several shells provided in the system, each has different features, but all of them affect how commands are interpreted. The shell provides tools to create the UNIX environment. The shell functionality includes control flow primitives, variable and string substitution, and parameter passing.

Shell processing is generally executed in five steps, as discussed in the previous chapter. The details of these steps are in the forthcoming section.

Writing shell scripts

Shell scripts are used to automate tasks that need to be done regularly. They are extensively used by the system administrator. Shell scripts save time as you do not have to type the commands every time and can just store them in a file as a shell script and run the shell script instead.

A shell script can be written directly on the terminal as follows:

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

We can write shell programs by creating scripts containing a series of shell commands into a file using an editor like vi. Later the file can be executed by the shell.

```
$vi my.sh
```

We also need to specify that the script is executable by setting the proper bits on the file with `chmod`:

For example:

```
$ chmod +x my.sh
```

We can then execute it on the shell prompt as follows:

```
$ my.sh
```

Or

```
$./my.sh
```

Following is an example of a simple shell script:

```
$vi my.sh
#my.sh
echo "My first shell script"
echo "Hello $LOGNAME"
echo "Date is `date`"
$chmod +x my.sh
$my.sh
```

My first shell script:

```
Hello usha
Date is Wed Feb 22 10:11:37 IST 2006
```

From the first shell script, we note that:

- Within the script, **#** indicates a comment from that point until the end of the line.
- **LOGNAME** is an environment variable, and **$LOGNAME** is accessing its value.
- We can substitute the output of a command using command substitution by using backquotes as in date. Here the date command is executed, and its value is substituted in the echo output.

Variables

As in all programming languages, the shell also allows to store values into variables. We can create our own variables in the shell and assign values. Conventionally, the shell variable name starts with an alphabet or underscore (_) character followed by zero or more alphanumeric values or underscore characters.

```
$param=value
```

Where **value** is any valid string and can be enclosed within quotations, either single (**value**) or double (**value**), to allow spaces within the string value. When enclosed with backquotes ("value"), the string is first evaluated by the shell, and the result is substituted. This is often used to run a command, substituting the command output for value.

The value stored in the shell variable can be displayed using the echo command. Shell uses a special $ symbol followed by the variable name to substitute its value, for example:

```
$ echo $param
```

Shell first substitutes value stored inside param and then executes the **`echo`** command. Values of more than one variable can be substituted at the same time:

```
$ echo $ param $count
```

In this case, the shell substitutes the value of param and count and then executes the echo command.

Shell variable can be used anywhere on the command line once it is declared:

```
$ my_dir=/users/home
$ ls $my_dir
```

In the preceding example, ls command will display the contents of directory users/home through the variable **`my_dir`**.

Shell variable can also store command name as value:

```
$ my_command=cp
$ $my_command file1 file2
```

In the preceding example, the shell first substitutes cp in place of **`my_command`** and then executes **`cp`** command to copy **`file1`** to **`file2`**.

If we try to display a variable without assigning any value, the shell does not give any error message; instead, it displays blank:

```
$ $var
```

Followings are the examples of displaying values of different shell variables:

```
// Command  : $day="Monday"
$echo $day
Output  : Monday
Command : $ day=`date +%a`
$ echo $day
Output: : Wed
```

After the parameter values have been assigned, the current value of the parameter is accessed using the **`$param`**, or **`${param}`** notation.

Some points are required to be considered while assigning values to variables:

- White spaces are not allowed before and after the equal to sign.

- In the shell, programming values are not bound to the data type as in other programming languages. The value of the shell variables is always considered as a string no matter what type of value is assigned to it. If Value 12 is assigned to the variable, it is considered as String 12, not Integer 12.
- As in other programming languages, the shell variable is not required to be declared before use because its value is not bound to the data type.

Comment

The details about the comment are already discussed in the previous chapter as conventional programming languages shell provides a way to write remarks inside the program. A special symbol # is provided to write comments. Whenever # is encountered in the program, the words following # till the end of the line are considered as a comment. If # starts, the line entire line is treated as a comment.

```
# this line is to write remarks
cp $file1 $file2
# Test for the correct argument
```

Quoting

Strings are quoted to control the way the shell interprets any parameters or variables within the string. The single (') and double (″) quotes can be used to quote strings.

Single quotes

Single quotes are used to keep words together which are otherwise separated by whitespaces. Single quotes are also used to preserve white spaces. All special characters are ignored by the shell if they are enclosed inside single quotes.

As an example, let us consider a file **`per_data`**:

```
$ cat per_data
John miller 34 graduate
Bill Geirge 45 graduate
```

```
John Gates 50 graduate
```

Search the pattern Bill in the file per-data.

```
$ grep Bill per_data
Output : Bill Geirge 45 graduate
```

If the search string is John, then:

```
$grep John per_data
```

The output of the preceding command will be:

```
John miller 34 graduate
John Gates 50 graduate
```

But if Bill Gates is required to be found specifically, then:

The command is:

```
$ grep Bill Gates per_data
```

The output is an error.

```
grep can't open Gates
```

`grep` command takes second words as a file name. To search pattern containing two or more words, the search pattern must be enclosed in the single quotes.

```
$ grep  'Bill Gates' per_data
```

Let us take another example to show whitespace preservation using single quotes:

```
$ echo this     example   is to show use of   single    quotes
Output : this  example  is to show use of  single  quotes
```

To preserve white spaces, the string is to be displayed should be enclosed within single quotes:

```
$ echo 'this     example   is to show use of
single    quotes '
Output :  : this     example   is to show use of
single    quotes
```

Let us take an example to show ignorance of special characters:

```
$ var=23
$ echo $var
```

Output : **`23`** , $var is substituted by value of variable **`var`**.

```
$ echo '$var'
```

Output: **$var**, the substitution of value is ignored because **$var** is enclosed inside the single quotes.

Double quotes

Double quotes work similar to single quotes, but it does not ignore dollar signs, back quotes, and backslash characters.

As an example, let us use the variable, var, that has been assigned the value bat, and the constant string, man. If I wanted to combine these to get the result batman, I might try:

```
$varman
```

But this does not work because the shell will be trying to evaluate a variable called **varman**, which does not exist. To get the desired result, we need to separate it by quoting or by isolating the variable with curly braces ({}), as in:

```
"$var"man - quote the variable
$var""man - separate the parameters
$var"man" - quote the constant
$var''man - separate the parameters
$var'man' - quote the constant
$var\man - separate the parameters
${var}man - isolate the variable
```

These all work because ", ', \, {, and } are not valid characters in a variable name.

We could not use either of:

```
'$var'man
\$varman
```

Because it would prevent the variable substitution from taking place. When using the curly braces, they should surround the variable only and not include the **$**; otherwise, they will be included as part of the resulting string, for example:

```
$ echo {$var}man
{bat}man
```

The backslash

The backslash is used to quote a single character. When a single character is following a backslash, any special meaning assigned to that character is ignored.

For example:

```
$ echo >
```

In the preceding example, the shell assumes > as the indirection operator and generates an error because it does not find the corresponding file in which standard output will be transferred. To ignore special meaning of > operator, it should be preceded by \ symbol:

```
$ echo \>
Output : >
```

As discussed in the previous section, a double quote does not ignore backslash. This means that \ can be used inside a double quote to ignore the special meaning of the character, which is otherwise not ignored by double-quotes.

```
$ x=23
$ echo '\$x"
Output: :$x
```

Command substitution

Command substitution means inserting the standard output of the command at any point in the command line.

The shell performs command substitution by enclosing the command in the backquotes (`).

```
$ echo current directory : pwd
```

Output: `current directory`: `pwd`; in this command `pwd` is treated as a simple string.

```
$echo current directory; `ped`
```

Output: `current directory: /users/home`

When the shell finds a backquote while scanning the command line, it expects a command following the backquote. If the command is found, it first executes the command and substitutes the output on the command line.

Special shell variables

There are a number of variables automatically set by the shell when it starts. These allow you to reference arguments on the command line.

Variable	Usage
$#	Number of arguments on the command line.
$-	Options are supplied to the shell.
$?	The exit value of the last command executed.
$$	Process number of the current process.
$!	Process number of the last command done in the background.
$n	Argument on the command line, where *n* is from 1 to 9, reading left to right.
$0	The name of the current shell or program.
$*	All arguments are passed to the shell program on the command line ("$1 $2 … $9")
$@	All arguments on the command line, each separately
quoted ("$1" "$2" … "$9")	

Table 8.1: *Shell Script built-in-parameters with details*

We can illustrate these with some simple scripts as follows:

Example:

```
echo “$#:” $#
echo ‘$#:’ $#
echo ‘$-:’ $-
echo ‘$?:’ $?
echo ‘$$:’ $$
echo ‘$!:’ $!
echo ‘$3:’ $3
echo ‘$0:’ $0
echo ‘$*:’ $*
echo ‘$@:’ $@
```

When executed with some arguments it displays the values for the shell variables, for example:

```
$ ./variables.sh one two three four five
5: 5
```

```
$#: 5
$-:
$?: 0
$$: 12417
$!:
$3: three
$0: ./variables.sh
$*: one two three four five
$@: one two three four five
```

The shift command

The list of strings typed after shell script names on the command line are passed as arguments to the shell script. $1, $2–$9 refers to the positional arguments to the shell script. If the numbers of arguments are greater than nine, then the shift command is used.

The shift command is used to effectively left shift positional parameters. The argument $1 is lost; the value of $2 is shifted to $1, and so on. The inaccessible tenth parameter is available for $9.

Following is the example of a shift command:

```
vi shift.sh
echo "functioning of shift command'
echo $1
shift
echo $1
```

Save and execute shift.sh file as follows:

```
shift.sh shift command
```

The output of this shell script is as follows:

```
functioning of shift command
shift
command
```

The set command

The set command is used to explicitly initialize the values of positional parameters:

```
set  a b c
```

The **set** command will assign values **a**, **b**, and **c** to **$1**, **$2**, and **$3**, respectively.

Following is the example of the set command:

```
vi set.sh
set ONE TWO THREE
echo $1
echo $2
echo $3
```

Save and execute the script:

```
$ set.sh
Output   ONE     TWO      THREE:
1
EDITOR= /users/bin
HOME=/users/George
LOGNAME=Geore
PATH= /users/bin
```

The **set** command is also used to reassign the values of positional parameters. The positional parameters take values from strings passed on the command line at the time of execution of the shell script. The only way to change these values is to use set command during execution.

Interactive input

Shell scripts will accept interactive input to set parameters within the script.

Shell uses the built-in command, read, to read in a line from standard input and assign whitespace delimited words to variables. If a number of words are more than the number of variables, then excess words are stored in the last variable.

The exit status of the read command is zero unless an end-of-file condition is encountered. If reading is performed from the terminal, the end-of-file is encountered when *Ctrl* + *D* is pressed. If reading is performed from the file, the end-of-line is encountered when there is no more data in the file.

Consider the following example:

```
$ vi add.sh
```

```
while  read v1 v2
do
echo $ ((v1 -v2))
done
When executed:
$ add.sh
23 13
10
25 5
20
Ctrl = d
$
```

Let us take another example:

```
$ read age name
39 Bill George
$ echo '$age\n $name'
39
Bill George
```

We can illustrate this with the simple script:

```
#!/bin/sh
echo "Input a phrase \c" # This is /bin/echo which requires
"\c" to prevent <newline>
read param
echo param=$param
```

When we run this script, it prompts for input and then echoes the results:

```
$ ./read.sh
Input a phrase hello mca01 # I type in hello mca01 <return>
param=hello mca01
```

Subshell

As discussed earlier, there are multiple shells in the system corresponding to each terminal. Moreover, the shell creates sub shell whenever a shell script is executed. A subshell is an entirely new shell for executions of the program. A new shell runs with its own shell environment and local variables. A subshell does not have knowledge of local variables assigned in its parent shell. A subshell cannot change the value of a variable in the parent shell. A subshell can also not change the values of the built-in environment variables in the parent shell.

Following is the example of a local variable in subshell:

```
vi loc_var.sh
echo initialize local variable x in sub shell
x=50
echo $x
$ loc_var.sh     execute loc_var.sh
$50     Sub shell terminated
$ echo $x
$
```

This will display blank because variable **x** is not assigned any value in the parent shell.

Following is the example of a subshell with environment variables:

```
vi shell_vars.sh
echo $PS1
PS!='=>'
```

Save and execute the file **shell_var.sh**:

```
$ shell_var.sh
$ $
=>
```

Now again, display PS1 from the parent shell:

```
$ echo $PS!
$ $
```

That means changes made in the environment variables of subshell are not reflected in the parent shell.

Exporting variables

As discussed earlier, a subshell does not have knowledge of local variables assigned in the parent shell. If local variables in the parent shell are required to be accessed in the subshell, they must be first exported. Exporting variables means passing down variables to the subshell.

Following is the example of local variables in sub shell:

```
$ x=200
$ echo 4x
$ 200
vi exp_var.sh
echo x = $x
```

Save and execute script **`exp_var`**; the output is **`x`**.

Since variable **`x`** is not available in the subshell to access, the output of the subshell is **`x`**, not **`200`**. To export **`x`** to the subshell and to access the value of **`x`** to the subshell, it needs to be exported to the subshell.

Following is the example for exporting local variable:

```
$ x=200
$ export x
$ exp_var.sh
x=200
```

This time variable **`x`** is exported to the subshell; therefore, this time, the output is **`200`**.

Test operator for comparisons

The built-in shell command test is used to test logical conditions:

Test expression

The expression represents the logical comparison. The test operator can be used for numeric comparisons, string comparisons, or file-related comparisons.

Conditional statements are evaluated for true or false values. This is done with the test, or its equivalent, the [] operators. If the condition evaluates to true, a zero (TRUE) exit status is set; otherwise, a non-zero (FALSE) exit status is set. If there are no arguments, the non-zero exit status is set.

Test operator for files: test [option] filename

Every shell script deals with a file. Due to this reason, test commands provide a set of options to test various attributes of the files. We can check whether a file exists, whether it is readable, and so on, using the test operator.

The options available for the test operator for files include:

- **-r**: True if it exists and is readable.
- **-w**: True if it exists and is writable.
- **-x**: True if it exists and is executable.
- **-f**: True if it exists and is a regular file (or for csh, exists and is not a directory).
- **-d**: True if it exists and is a directory.
- **-h** or **-L**: True if it exists and is a symbolic link.
- **-c**: True if it exists and is a character special file (that is, the special device is accessed one character at a time).
- **-b**: True if it exists and is a block special file (that is, the device is accessed in blocks of data).
- **-p**: True if it exists and is a named pipe (fifo).
- **-u**: True if it exists and is **setuid** (that is, has the set-user-id bit set, s or S in the third bit).
- **-g**: True if it exists and is **setgid** (that is, has the set-group-id bit set, s or S in the sixth bit).
- **-k**: True if it exists and the sticky bit is set (a t in bit 9).
- **-s**: True if it exists and is greater than zero in size.

There is a test for file descriptors:

- **-t [file_descriptor]**: True if the open file descriptor (default is **1**, **stdin**) is associated with a terminal.

All these operators are unary in nature. They take a single argument as the name of the file, including the directory.

Following is an example of the use of a file:

```
#check a file
echo "Enter a filename"
read fname
if  test  -d  $fname
then
 echo "$fname is a directory"
else
 echo "$fname is a file"
```

```
fi
```

Test for strings:

There are tests for strings: we can check if a string has a length of zero or compare two strings, as shown in the following table:

- **-z string:** True if the string length is zero.
- **-n string:** True if the string length is non-zero.
- **string1 = string2:** True if string1 is identical to string2.
- **string1 != string2:** True if string1 is non identical to string2.
- **string:** True if the string is not NULL.

Following is an example to use string check:

```
#check if string is null
echo “Enter a file name”
read fname
if test -z  $fname
then
 echo “Enter a non null filename”
fi
```

It will ask for the filename. If no filename is given, it will print **`Enter a non null filename`**.

Test for numeric comparisons:

A test operator can be used to compare two numbers. There are integer comparisons is given as follows:

- **n1 -eq n2:** True if integers n1 and n2 are equal.
- **n1 -ne n2:** True if integers n1 and n2 are not equal.
- **n1 -gt n2:** True if integer n1 is greater than integer n2.
- **n1 -ge n2:** True if integer n1 is greater than or equal to integer n2.
- **n1 -lt n2:** True if integer n1 is less than integer n2.
- **n1 -le n2:** True if integer n1 is less than or equal to integer n2.

Following is an example of a program for numeric comparisons:

```
#compare.sh
echo “Enter two numbers”
```

```
read n1 n2
if test $n1 -eq $n2
then
 echo “The numbers are equal”
elif test $n1 -gt $n2
then
 echo $n1 is greater than $n2
else
 echo $n2 is less than $n1
fi
```

It will be as two numbers to enter. If both the numbers are equal, it will print The numbers are equal; otherwise, then it will test the greater number and print the message accordingly.

The test command can also be represented using [and] without the test operator.

For example:

```
if   [  -d  $fname ]
then
 Echo “$fname is a directory”
else
 Echo “$fname is a file”
fi
```

Logical operators with test

The following logical operators are also available:

The negation operator (!)

The ! unary operator can be used with a test command to negate the result of the evaluation of any expression.

```
[ ! -r /users/home/data]
```

Condition will evaluate true if directory **`/user/home/data`** is not readable.

The logical AND operator

The **-a** and (binary) operators perform logical AND of two expressions with test command.

```
[ -f  $file1 -a -r $file1 ]
```

The preceding condition will evaluate true if variable **file1** contains an ordinary file and it is readable.

The logical OR operator

The **-o** or (binary) performs logical or of two expressions with test command.

```
[ -f $file1 -o  -r $file1 ]
```

The preceding condition will evaluate true if variable **file1** contains either an ordinary file, or it is readable

Control commands

Bourne shell has control commands such as if-then-else construct, case construct, loops, and expression evaluation.

Conditional if

This is the basic control flow mechanism. It uses the **if**, **elif**, and **else** commands/keywords to control the flow of the process.

```
if condition1
then
command list if condition1 is true
[elif  condition2
then
command list if condition2 is true]
[else
command list if condition1 is false]
fi
```

The conditions to be tested for are usually done with the test command. The if and then must be separated, either with a <newline> or a semicolon (;).

Following is an example to check number of command line arguments:

```
#!/bin/sh
if  [  $# -ge 2  ]
```

```
then
echo $2
elif [  $# -eq 1  ]; then
echo $1
else
echo No input
fi
```

There are required spaces in the format of the conditional test, one after [and one before]. This script should respond differently depending upon whether there are zero, one, or more arguments on the command line. First with no arguments:

```
$ ./if.sh
No input
```

Now with one argument:

```
$ ./if.sh one
one
```

And now with two arguments:

```
$ ./if.sh one two
two
```

Conditional case

To choose between a set of string values for a parameter use case in the Bourne shell:

```
case parameter in
pattern1[|pattern1a]) command list1;;
pattern2) command list2
command list2a;;
pattern3) command list3;;
*) ;;
esac
```

You can use any valid filename meta-characters within the patterns to be matched. The ;; ends each choice and can be on the same line, or following a <newline>, as the last command for the choice.

Additional alternative patterns to be selected for a particular case are separated by the vertical bar, |, as in the first pattern line in the preceding example. The wildcard symbols,: ? to indicate any one character and * to match any number of characters, can be used either alone or adjacent to fixed strings.

This simple example illustrates how to use the conditional case statement:

```
#menu.sh
clear
echo  “Press 1 to see directory listing “
echo  “Press 2 to see users logged in”
echo  “Press 3 to quit”
echo “Enter your choice”
read choice
case $choice in
1)ls
;;
2)who
;;
3)exit
;;
*) echo “Unknown option”
;;
esac
```

The while command

The **while** command lets you loop as long as the condition is true. The condition is tested; if its exit status is zero, the command list enclosed between doing and done is executed repeatedly till the condition remains true.

```
while condition
do
command list
[break]
[continue]
done
```

The command list enclosed between do and done may never be executed if the exit status of the condition is non-zero.

A simple script to illustrate a while loop is as follows:

```
#!/bin/sh
while [ $# -gt 0 ]
do
echo $1
shift
done
```

This script takes the list of arguments, echoes the first one, then shifts the list to the left, losing the original first entry. It loops through until it has shifted all the arguments on the argument list.

```
$ ./while.sh one two three
one
two
three
```

The until command

The **`until`** is a control command used for executing a set of statements until the condition is true. As soon as the condition becomes false, the control moves out of the until loop.

```
until condition
do
command list while condition is false
done
```

The condition is tested at the start of each loop, and the loop is terminated when the condition is true. The until construct is similar to while only it executes command list as long as condition after until returns non-zero status.

A script equivalent to the preceding example is as follows:

```
#!/bin/sh
until [ $# -le 0 ]
do
echo $1
shift
done
```

Notice, though, that here, we are testing for less than or equal, rather than greater than or equal, because the until the loop is looking for a false condition. Both the until and while loops are only executed if the condition is satisfied. The condition is evaluated before the commands are executed.

Same as while the command list between do and done might never be executed if the condition returns zero status the first time it is tested.

The for command

One way to loop through a list of string values is with the for command.

```
for variable [in list_of_values]
do
command list
done
```

The **`list_of_values`** is optional, with **`$@`** assumed if nothing is specified. Each value in this list is sequentially substituted for a variable until the list is emptied. Wildcards can be used and are applied to file names in the current directory.

Here is a for loop for copying all files ending in old to similar names ending in .new. In these examples, the basename utility extracts the base part of the name so that we can exchange the endings.

```
#!/bin/sh
for file in *.old
do
newf=`basename $file .old`
cp $file $newf.new
done
```

The break command

The **`break`** command is used to make an immediate exit from the loop.

When **`break`** command is executed, control immediately switches to the next statement written just after the loop, and then the program continues as normal.

Condition status within the loop construct serves no purpose but returns zero or non-zero depending upon the loop in which the break command is used.

Following is an example using the **`break`** command:

```
while :
do
  read  -p  "Enter integer number ; ` a
  if  [ $a  -eq -1 ]
  then
  break
  fi
done\
  echo $a
```

The continue command

The **`continue`** command passes control to the next iteration of the loop bypassing the remaining command in the loop body.

Following is an example of the shell script to display even numbers:

```
for number in 1 2 3 4 5 6 7 8
do
case $number in (1| 3|5 |7) continue
esac
echo $number
done
```

It will give the output **`2   4   6   8`**.

Special string operations

We can also find out the length of the string, extract a substring or locate a character in a string using the expr operator.

To calculate the string length: use expr with ".*"

```
$expr "abcdefghijk"  :  '.*'
12
```

Prints the length of the string.

Use the following to extract a substring:

```
$expr "abcd" :  '..\(..\)'
cd
```

The number of dots corresponds to the number of characters. This means to retrieve characters cd in the string **`abcd`**, we skip the first two characters and then extract two characters.

Next, let us locate a character in a string:

```
$expr "abcdefg" :  '[^d]*d'
4
```

This command is used to locate the position of character **`d`** in the string **`abcdefg`**. The **`expr`** command indicates that skip all characters that are not d till it reaches the character **`d`**.

Parameter substitution

You can reference parameters abstractly and substitute values for them based on conditional settings using the operators defined following. Again we will use the curly braces ({}) to isolate the variable and its operators.

- **$parameter:** Substitute the value of the parameter for this string
- **${parameter}:** Same as above. The brackets are helpful if there is no separation between this parameter and a neighboring string.
- **$parameter=:** Sets parameter to null.
- **${parameter-default}:** If a parameter is not set, then use default as the value here. The parameter is not reset.
- **${parameter=default}:** If the parameter is not set, then set it to default and use the new value
- **${parameter+newval):** If the parameter is set, then use **`newval`**; otherwise, use nothing here. The parameter is not reset.
- **${parameter?message}:** If the parameter is not set, then display the message. If the parameter is set, then use its current value.

There are no spaces in the preceding operators. If a colon (:) is inserted before the -, =, +, or ? then a test if first performed to see if the parameter has a non-null setting.

To illustrate some of these features, we will use the following test script:

```
#!/bin/sh
param0=$0
test -n "$1" && param1=$1
test -n "$2" && param2=$2
test -n "$3" && param3=$3
```

```
echo 0: $param0
echo "1: ${param1-1}: \c" ;echo $param1
echo "2: ${param2=2}: \c" ;echo $param2
echo "3: ${param3+3}: \c" ;echo $param3
```

In the script, we first test to see if the variable exists; if so, we set a parameter to its value. As follows, we report the values, allowing substitution.

In the first run through the script we will not provide any arguments:

```
$ ./parameter.sh
0: ./parameter.sh # always finds $0
1: 1: # substitute 1, but don't assign this value
2: 2: 2 # substitute 2 and assign this value
3: : # don't substitute
```

In the second run through the script, we will provide the arguments:

```
$ ./parameter one two three
0: ./parameter.sh # always finds $0
1: one: one # don't substitute, it already has a value
2: two: two # don't substitute, it already has a value
3: 3: three # substitute 3, but don't assign this value
```

Functions

The Bourne shell has a function facility too. These are somewhat similar to aliases in the C shell but allow more flexibility. The function is reusable code that can be later called from UNIX code.

A function has the form:

```
fcn () { command; }
```

Where the space after {, and the semicolon (;) are both required; the latter can be dispensed with if a <newline> precedes the }. Additional spaces and <newline>'s are allowed.

```
ls() { /bin/ls -sbF "$@";}
ll() { ls -al "$@";}
```

The first one redefines **ls** so that the options **-sbF** are always supplied to the standard **/bin/ls** command and acts on the supplied input, **"$@"**. The second one takes the current value for **ls** (the previous function) and tacks on the **-al** options. Functions are very useful in shell scripts.

Passing arguments to function

Positional parameters $1, $2, …, $*n* are used arguments to the function. White spaced set of strings is used to assign the values to the positional parameters during the function call. For example:

```
Shell__fun ()
{
   echo $1
   echo $2
}
```

To assign values to **$1** and **$2**, the function will be called as follows:

```
Shell_fun "hell" World"
```

Returning from function

Shell function returns the control in four different ways:

- Changes the states of the local variables within the function.
- Using **exit** command to terminate shell script.
- Using the return command to exit from the function and return a value to the calling section of the shell script.
- echo the output to the terminal.

Following is an example script to explain returning from a function:

```
fun_ret.sh
fun1_ret()
{
c=`expr $1 = $2'
return
}
Fun1_ret 10 20
ret=$?
echo   sum is $ret
```

Save and execute script **fun_ret.sh**:

Output is:

```
Sum is 30
```

Trapping signals

A process running in UNIX can receive signals and respond to them. For example, when we give the **`kill -9`** command, we send signal Number 9 to the process to abort it. UNIX lets you alter the effects of signals using a trap.

Trap command signal list

Trap waits for the signal given in the signal list. If a signal shows up, it executes the command.

Consider the following example:

```
#myfile.sh
file=$1
pat=$2
trap 'rm -f  test.$$' 1 2 15
if grep $pat $file > test.$$
then
  echo "Pattern found"
else
  echo  "Pattern not found"
fi
rm -f test.$$
```

So, if we send an interrupt before the program reaches the trap line, the interrupt will act in the usual way; that is, it will terminate the program. If the interrupt is sent after the program has read this line, the commands in the trap will be executed.

Arrays

Bourne and C shell do not support arrays, but bash and Korn shell support arrays. The array facility provided by Korn shell is limited to one dimension and can have a maximum of 1,024 elements.

Array elements can be assigned in two ways. One is to use the standard variable assignment method with the array index operator []. For example:

```
my_arr [3]= arr3
```

The value **`arr3`** is assigned to the element of **`my_arr`** with Index **`3`**.

The second method to assign values to the array elements is using a variant of the set command. For example:

```
set -A my_arr arr1 arr2 arr3---
```

This will assign arr1 to the variable **`my_arr[0]`**, **`arr2`** to the variable **`my_arr[1]`**, and so on. If the array is not already created, it first creates and then assigns values to the corresponding array indices.

An array element can be referred to in the same manner as other shell variables. For example, i-th index of array **`my_arr`** can be referred as **`$ my_arr [i]`**. Index **`i`** can be generated using mathematical expressions. If **`*`** is placed in the place of the index, it will refer to all elements of the array separated by space. Omitting i will automatically be replaced by index 0.

The shell provides **`#`** operator to count total elements in the array. For example **`$ { # my_arr [*]}`** will refer to all the elements of array **`my-arr`**.

`$ksh` invokes the Korn shell.

We can declare and set values to an array as follows:

```
$myarr={10 23 14 15 16}
```

To print all the elements of the array:

```
$echo ${myarr[*]}
```

Output will be: **`10 23 14 15 16`**.

To print the size of the array:

```
$echo ${#myarr[*]}
```

Output: `5`.

To assign values to individual elements, use:

```
$myarr[0]=15; myarr[1]=18
```

Conclusion

UNIX is a large collection of commands. Users can build their own complex commands by combining these simple sets of commands. The notion of software development in UNIX started with the shell at the top level of the program. The UNIX shell is a full-featured programming language with a shell environment, shell variable, command-line arguments, conditional, and control constructs. In addition to the basic tools of programming, the shell also provides array, functions, and signal trapping with limited capacity. The bourn shell is considered as a primary shell for UNIX operating system.

Review Exercise

1. Write a shell script to display a word repeatedly a number of times. [Command line arguments: word, number]
2. Write a shell script to check for a leap year. [Command line argument: four digit year]
3. Write a shell script to delete all lines from a file that contains the word UNIX. [Command line argument: filename]
4. Write a shell script to compare two files and delete the second file if found equal. [Command line arguments: file1, file2]
5. Write a shell script to find the sum of digits of a number. [Command line argument: number]
6. Write a shell script to find the factorial of a number.
7. Write a shell script to prefix each line of a file by line number. [Command line argument: filename]
8. Write a shell script to convert all lower case characters in a file to upper case. [Command line argument: filename]
9. Write a shell script to copy the date without time into a file.
10. Write a shell script to check whether the given file is a regular file or a directory.
11. Write a shell script that displays logged-in users after every 15 seconds.
12. Write a shell script to count down from a number to 0. If no arguments are provided, the starting number should be taken as 10. [Command line argument: number]
13. Write a shell script to change all uppercase filenames in the current directory to lowercase.
14. Write a shell script to change the extension of all text files in the current directory, from .txt to .doc
15. Write a shell script to add a message (c) MCA 2006-2007 at the end of all text files in the current directory

16. Write a shell script to repeatedly ask for the capital of India. The program should terminate once the user enters Delhi.
17. Write a shell script to check whether a user is logged in or not.
18. Write a shell script to echo Good Morning or Good Afternoon or Good Evening according to the time of day.
19. Write a shell script to delete files passed as command-line arguments
20. Write a shell script to count the number of lines in a file that contain a given word. [Command line arguments: word, filename]
21. What are system and user variables? Discuss any five system variables.
22. Write a shell script to reverse the digits of an input number *n*.
23. Explain different shell programming constructs with suitable examples.
24. Explain the usage of the following variables:
    a. $#
    b. $*
    c. $$
    d. (d)$?
    e. PS1
25. Write a shell script to identify whether an odd or an even number of parameters is passed.

Solutions to some of the shell scripts are provided here:

```
#Shell Script to display a word repeatedly
if [ $# -ne 2 ]
then
echo "Usage: a1 word number"
exit
else
i=1
while [ $i -le $2 ]
do
echo $1
i=`expr $i + 1`
```

```
done
fi
```

#Shell Script to check for a leap year

```
if [ $# -ne 1 ]
then
echo “Usage: $0 year”
exit
fi
if [ `expr $1 % 4` -eq 0 -a \(`expr $1 % 100` -ne 0 -o `expr
$1 % 400`
-eq 0 \)]
then
echo $1 is a Leap Year
else
echo $1 is not a Leap Year
fi
```

#Shell Script to delete all lines containing the word UNIX

```
if [ $# -ne 1 ]
then
echo “Usage: $0 filename”
exit
else
grep -v “UNIX” $1 > tmp
mv tmp $1
fi
```

#Shell Script to compare 2 files and delete the second file if
#found equal

```
if [ $# -ne 2 ]
then
echo “Usage: $0 file1 file2”
exit
else
cmp -s $1 $2
if [ $? -eq 0 ]
then
rm -f $2
else
```

```
echo “Files $1 and $2 are not identical”
fi
fi
```

# Shell Script to find the sum of digits of a number

```
if [ $# -ne 1 ]
then
echo “Usage: $0 number”
exit
else
sum=0
num=$1
while [ $num -ne 0 ]
do
tmp=`expr $num % 10`
sum=`expr $sum + $tmp`
num=`expr $num / 10`
done
echo The sum of digits is $sum
fi
```

#Shell Script to find factorial of a number

```
if [ $# -ne 1 ]
then
echo “Usage: $0 number”
exit
else
fact=1
count=1
while [ $count -le $1 ]
do
fact=`expr $fact \* $count`
count=`expr $count + 1`
done
echo “Factorial is $fact”
fi
```

# Shell Script to prefix each line of a file by line number

```
if [ $# -ne 1 ]
then
```

```
echo "Usage: $0 filename"
exit
else
nl $1 > tmp
mv tmp $1
fi
```

# Shell Script to convert all lower case characters to upper case

```
if [ $# -ne 1 ]
then
echo "Usage: $0 filename"
exit
else
tr "[a-z]" "[A-Z]" < $1 > tmp
mv tmp $1
fi
```

#Shell Script to copy date without time in a file

```
date|cut -d" " -f1-3,6 >file
cat file
```

#Shell Script to check for a regular file or directory

```
if [ $# -ne 1 ]
then
echo "Usage: $0 filename"
exit
fi
if [ -d $1 ]
then
echo It is a Directory
else
if [ -f $1 ]
then
echo It is a Regular File
else
echo It is neither a directory nor any regular file
fi
fi
```

```
#Shell Script to display logged in users after every 15
seconds
while :
do
who -H
sleep 15
done
# Shell Script to count down from a number to 0
if [ $# -eq 0 ]
then
start=10
else
start=$1
fi
while [ $start -ge 0 ]
do
echo $start
start=`expr $start - 1`
done
#Shell Script to change all uppercase filenames to lowercase
for oldfile in *
do
if [ $newfile != $oldfile ]
then
mv $oldfile $newfile
fi
done
# Shell Script to change all .txt extension to .doc extension
for file in *.txt
do
lname=`basename $file txt`
mv $file ${lname}doc
done
#Shell Script to add a message (c) MCA 2006-2007 at end of all
#text files in current directory
for file in *.txt
do
```

```
echo "(C) MCA 2006-2007" >>$file
done
```

#Shell Script to check capital of India

```
capital=delhi
ans=empty
while [ $ans != $capital ]
do
echo "Enter capital of India? "
read ans
done
```

#Shell Script to check for a logged in user

```
if [ $# -ne 1 ]
then
echo "Usage: a17 login_name"
exit
fi
who|grep $1 >/dev/null
if [ $? -eq 0 ]
then
echo $1 is currently logged in
else
echo $1 is currently not logged in
fi
```

#Shell Script to greet a user according to time

```
h=$(date|cut -c 12-13)
if [ $h -ge 4 -a $h -lt 12 ]
then
echo Good Morning
else
if [ $h -ge 12 -a $h -lt 17 ]
then
echo Good Afternoon
else
echo good Evening
fi
fi
```

#Shell Script to delete files passed as command line arguments

```
if [ $# -lt 1 ]
then
echo “Usage: $0 file1 file2…”
exit
fi
max=$#
n=1
while [ $n -le $max ]
do
rm -r $1
shift
n=`expr $n + 1`
done
```

#Shell Script to count the number of lines which contain a word

```
if [ $# -ne 2 ]
then
echo “Usage: $0 word filename”
exit
fi
grep -c $1 $2
```

Index