Yukun Liu
Yong Yue
Liwei Guo

UNIX Operating System

The Development Tutorial via UNIX Kernel Services

高等教育出版社
HIGHER EDUCATION PRESS

Springer

Yukun Liu
Yong Yue
Liwei Guo

UNIX Operating System

The Development Tutorial via UNIX Kernel Services

Yukun Liu
Yong Yue
Liwei Guo

UNIX Operating System

The Development Tutorial via UNIX Kernel Services

With 132 figures

高等教育出版社
HIGHER EDUCATION PRESS

Springer

Authors
Associate Professor Yukun Liu
College of Information Science and Technology
Hebei University of Science and Technology
Hebei 050018, China
E-mail: lyklucky@hebust.edu.cn

Professor Yong Yue
Faculty of Creative Arts, Technologies and Science
University of Bedfordshire
Park Square Luton Bedfordshire
LU1 3JU, United Kingdom
E-mail: yong.yue@beds.ac.uk

Professor Liwei Guo
College of Information Science and Technology
Hebei University of Science and Technology
Hebei 050018, China
E-mail: guoliwei@hebust.edu.cn

ISBN 978-7-04-031907-1
Higher Education Press, Beijing

ISBN 978-3-642-20431-9 e-ISBN 978-3-642-20432-6
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2011924668

© Higher Education Press, Beijing and Springer-Verlag Berlin Heidelberg 2011
This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilm or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer. Violations are liable to prosecution under the German Copyright Law.
The use of general descriptive names, registered names, trademarks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

This book presents UNIX as a practical case of operating systems for the reader to understand and master deeply and tangibly the theory and algorithms in operating systems. It gives discussions and studies on the hierarchical structure, principles, applications, shells, development, and management of the UNIX operation system multi-dimensionally, systematically and from the elementary to the profound. It brings readers to go into the inside of the UNIX operating system and lets them understand clearly what and how UNIX operating system functions.

Subject Matter

This book consists of 11 chapters. The first two chapters discuss the background of UNIX operating system (OS), and give a whole picture of what UNIX OS looks like and the evolution of UNIX.

Chapter 3 focuses on the editors that will be used frequently by UNIX users, no matter who are regular users or seasoned programmers.

Chapters 4, 5, 6, and 7 concentrate on the services of the UNIX kernel. Chapter 4 zooms in the process management, which is usually hidden from the final users. Chapter 5 is to discuss the UNIX memory management, which cooperates with the process management to accomplish the processes' concurrently running. Chapter 6 introduces the UNIX file management, which is involved almost in every service that the UNIX kernel provides to the users. Chapter 6, however, for UNIX users, is fundamentally useful to understand how UNIX works. Chapter 7 explores UNIX I/O, I/O redirection and piping. As UNIX treats its hardware devices as special files, this mechanism brings a whole different concept on UNIX input and output devices. I/O redirection and piping are two useful tools that can be used to deduce different commands to control the terminals through UNIX system calls.

UNIX has almost as many shells as versions of the UNIX operating system. Chapter 8 introduces some types of shells, shell evolution, and some common concepts in UNIX shells. As there are so many kinds of shells, it

is difficult to put all of them in one book. Hence, our strategy is to try to make one of them clear and integral in this book. Our choice is the primary shell for all other shells—Bourne shell. From this point, the readers can learn other shells by themselves from references. Therefore Chapters 9 and 10 focus on the discussion of Bourne shell as a programming language: Chapter 9 introduces basic facilities and Chapter 10 is for the advanced level.

Different from the studies in the previous chapters, which are concentrated on the local services and applications of UNIX in individual computers, Chapter 11 discusses the remote and network functions and services of UNIX in servers and workstations. Since the late 1960s, UNIX has had many original contributions to the development history of the computer networking and Internet.

Even though this book includes 11 chapters, it does not mean they are totally divided and irrelevant. Contrarily, they are coherent and relevant each other. Just like UNIX itself, its knowledge should be a link-up and cooperative "system". And we try very hard to unfold the information gradually and step by step in this book. When you, dear friends and readers, finish this book, you will have a relatively whole and systematical idea about UNIX. From that point, you can develop your applications on UNIX or other operating systems, or even build up a new operating system for a certain computer hardware system. This is just what the authors of this book really expect.

Historic and Active UNIX and Meaningfully UNIX Learning

As an open-source operating system, UNIX made its history during two decades of 1969 –1989. Maybe some say it has gone. However, UNIX's openness, which brought different groups of developers together to communicate their developing ideas and to respond feedback each other, really cultivated an excellent generation of operating system developers. We should remember these names: Dennis M. Ritchie, Ken Thompson, Robert S. Fabry, William N. Joy, Chuck Haley, Samuel J. Lefflerand, and more. The first two made their contribution to the premier UNIX System series and were honored by the ACM Turing Award in 1983 because of their work in UNIX, and the latter four did their great work on the primary UNIX BSD versions. Just as they made the earlier UNIX code open to the academic world and strived to move UNIX from one machine to another, UNIX grew and evolved. And even more, it left a lot of valuable academic papers about operating systems for the successors.

For its development process as an intact software system, UNIX, which presented solutions in detail, is unchallenged for generations of programmers. Compared to UNIX, its commercial counterparts usually provide a perfect environment that hides almost all the development details of lower levels of operating systems, which may leave a limited space for application program-

mers and also confine their imagination and creativity. This tendency can also affect the ability of system development newcomers to develop an intact software system that can handle software as well as hardware by restricting the field of vision to some detached modules or applications so as to result in software maintenance costly and complicated.

Further, just understanding the theory of operating systems, readers cannot image and understand well how the operating system works. With UNIX as a real case, readers can map the abstract algorithms, mechanisms and strategies of operating system theory into the real modules, functions and programs of UNIX implementation one-to-one. The abstract theory can be exemplified. In this way, as the promising programmers, readers can understand well and master these algorithms and mechanisms, practice them in their own development, and stimulate novel algorithms and mechanisms that may be more effective and efficient to their own context.

It seems as if a repetition of the old tale when considering the discussion on UNIX, which, all in all, reached its heyday around 1980s. In the latest two decades, however, due to commercial purposes and activities, there are no other operating systems like UNIX, which is so thoroughly open for the academic community to learn and do research.

In addition, there are plenty of references about UNIX that have been published, but most of them were originally published around 1980s. For the recent promising programmers, the published classics may be somewhat obscure because of the sparse context that might not be necessary for readers in those days but can be unfamiliar to nowadays readers. As the well-known rapid development of computer hardware in the latest decades, computer architecture and structure have made a big change. This change has also wielded a deep influence on the theories and concepts of computer, which makes the difficulty for recent readers to understand well descriptions and expressions in the published UNIX classics, and to map them properly into practical cases. It is possible to build an obstacle for readers to learn from them. Otherwise, for the operating system construction, which belongs to software developments but resides the one of the most exciting and integrated of software development, it would be a pity and defect if losing an operational means. Fortunately, this means can be gained by doing research on UNIX.

It is taken that UNIX has its own philosophy and several items in the philosophy are written in different references. If having the right, we can say that the most important one should be the UNIX programmers' dedication and passion to their work. UNIX is also deemed to a programmer's OS. UNIX programmers have done a wonderful work just as for tackling a necessary affair, from which others else really benefit. It is critical for the academic community.

UNIX benefited also from those days. If AT&T, at that time, could market computer products without a 1956 Consent Decree signed with the Federal Government, and if Bell Laboratories did not withdraw Ken Thompson and others from the MULTICS project, and if Professor Robert S. Fabry of the

University of California at Berkeley did not contact Ken Thompson at the Symposium on Operating Systems Principles at Purdue University in November 1973, we would have a totally different story about UNIX. It needs the open and free soil to breed an academic activity. The more relieved the outside environment is, the more natural the academic activity develops within the environment. UNIX was destined for being flourishing in its day.

Even though being just observers on this period of history, the authors of this book are impressed by the passion and concentration that UNIX developers had in the day. During five years of teaching UNIX in their campuses, the authors realized that if this fantastic piece of history was not introduced to more readers, it would be a pity for authors as well as readers. In this high-technology and high-material-civilization age, UNIX development process can give readers some new inspiration—a glowing motivation from inside to accomplish a meaningful work.

A General Survey of UNIX Development

Observing different versions of UNIX emerging, the authors and readers can discover that it is a process of constant development, amendment and enhancement. In this process, UNIX developers' thoughts were adjusted and enriched with the development of computer hardware and peripherals, and the proposal of new application demands. It resulted in UNIX' s being moved to execute on different hardware platforms and fitting in different projects, which also naturally made UNIX' s portability and scalability practice and reinforce repeatedly and concretized the concepts of portability and scalability in operating system theory.

UNIX drew on a lot of ideas of the earlier operating systems, and its openness made the idea-drawing expand into different UNIX versions and different groups of developers. For a new programmer, it is also necessary to derive the precursors' development thoughts and experiences. Only if learning from the precursors, newcomers can enrich their knowledge gradually and effectively, and the novel thinking can just grow from thick knowledge reserves.

For promising developers, the UNIX development process was also a training program. Linux is a successful example of UNIX derivatives. Through this training program with deducing mentally and programming physically, developers can get familiar with the computer system as a whole, including both hardware and software.

With the advent of commercial operating systems, most of the readers do their jobs on encapsulated and transparent operating systems. On the other hand, many students and graduate students of computer disciplines mostly start their studies from the theory of operating systems. A transparent, well-designed and inextricable operating system seems like saving the users a lot of time and effort, but it also cuts the exploring road towards the inside

of operating systems and the underlying hardware parts. For real developers and programmers, it may take a big risk to sit on a seemingly-transparent but unfamiliar system to do their developments—finally they may encounter some bugs that they cannot tackle. They have to experience something that can let them understand what really make the construction of an operating system, what the kernel of an operating system does for users, and how algorithms and mechanisms in the theory of operating systems are implemented. Even though the disassembled UNIX cannot tell all the story of a well-designed modern or future operating system, it can give the mapping or clues to different functions and services, which can be treated as an anatomy lecture of a promising surgeon.

In other words, a well-designed operating system may be daunting for a promising developer, which is complicated and confused. The simplicity and clarity of UNIX can help readers walk out of the swamp and sort out the confusion, and lead them to face and tackle more sophisticated problems.

Targets and Strategy of this Book

Knowledge needs to renew and information needs to update. The updating includes the expression of a convincing, successful and classical process in a proper, timely and new way. Maybe the UNIX story is old, but it can give different inspirations to people in different ages, which is still developing. The authors hope the developing can be reflected in this book.

One of the targets of this book is to let the UNIX philosophy propagate and carry on. Let more readers comprehend this philosophy's results—the fast development, maintainability and scalability of an operating system.

The authors also want to present readers (especially, programmers) two aspects of a whole operating system and any one of its subsystems, in other words, to give not only the inside implementation process, which is viewed by the system or application programmers, but also the outside application performance, which is usually felt by the end users. In this way, readers cannot only keep the view of the system constructors but also be considerate of the end users when developing their systems. During development, a system can benefit from that its developers can consider more for its end users.

For readers, it is easy to enter the learning from user interfaces of UNIX operating systems since they have usually had the experience of using one of operating systems in their daily works or lives. Thus, in this book, we take this strategy: when starting one topic, we present it from its user interface, and then go into the kernel with the system calls and lower-level algorithms if possible. In this way, readers can be brought from a familiar environment into elusive and deep techniques.

To describe algorithms, we try to use common English language rather than some computer language, such as C or assembly language. The primary

reason is: we try to make algorithms more readable and help readers save their time and effort. For a programmer, it is often time-consuming to read some code that is written by others.

Intended Audience

This book is written for the wide groups of readers who want to master the professional knowledge of operating systems through a real and open-source case. Its main readers include graduates, senior undergraduates and teachers of computer and software majors, and potential researchers on applicable computing and engineering modeling. The readers can also be ones who maybe have some or have not much knowledge related to Computer Science and Technology and Software Engineering, but have a strong interest in these fields and want to get into them quickly, acquire some useful and important knowledge and reach an advanced level in the relevant fields after learning. This book can help readers construct, not only as the users of operating systems but also in the view of the operating system designers, the knowledge on the UNIX operating system, and even on other kinds of operating systems. From this point, readers can build up their projects on an operating system. Or on this basis, readers can go deep into how UNIX and other operating systems to be designed and programmed because many versions of UNIX are open-source code, like Linux, and adjust and modify the operating systems on their own computer systems.

For readers whose mother tongues are not English, it may be more difficult to read and learn an English edition of the academic book than books written with their mother tongue. However, it is necessary for readers to have the ability to read the English editions of academic books, especially for computer and software professionals, because most papers on the advanced and update science and technology are written in English, especially in the field of computer hardware and software. Why not to try to gain this ability just from your learning process? Maybe it is difficult for you now. But nothing is easy when starting it. Maybe when you finish this book, you say, "It is not that hard, is it?" So try it now.

Yukun Liu
Yong Yue
Liwei Guo
January 2011

Acknowledgements

This book is funded by Academic Work Publication Fund of Hebei University of Science and Technology.

The authors of this book would like to give their thanks to Ms. Hongying Chen, Editor of High Education Press in China, who gave generously of her time and expertise to edit this book.

Contents

1 Background of UNIX Operating System

Before presenting the UNIX operating system, we describe the fundamental purpose of operating systems, different types of operating systems, and which type of operating systems UNIX belongs to. Then we discuss the development history of the UNIX, along with introducing different types of the UNIX.

1.1 Introduction of Operating System

An operating system (or OS) is usually sitting between the user's application programs and the computer hardware, which exploits the computer hardware resources to provide a set of services to the computer system users. The hierarchical view of a typical computer system is shown in Figure 1.1.

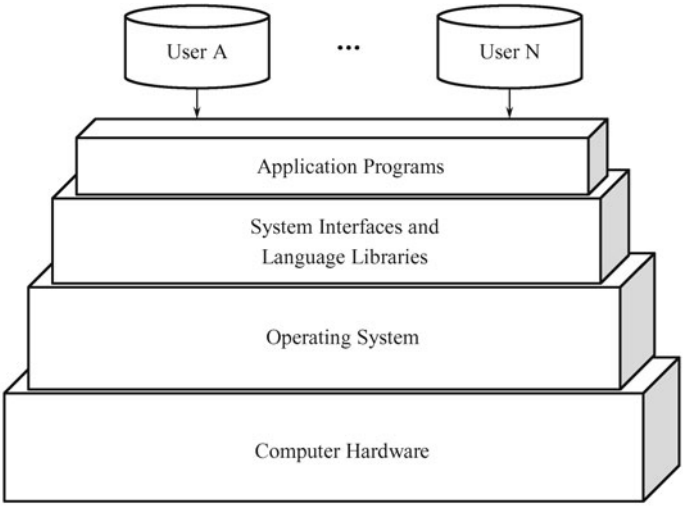

Fig. 1.1 The hierarchical view of a computer system.

The operating system functions as an interface between a user and a com-

puter. Usually, a common user is concerned about the applications rather than the architecture of computer. The application programs are developed by application programmers who develop the programs generally without responsibility of how to control the computer hardware. The reason beneath this is that there is an overwhelmingly complex task related to the computer hardware control, especially when it comes to the portability of the application programs from one computer system to another different one. To ease this task, a set of system programs should be built. Some of these system programs are frequently used as facilities, which implement functions, such as the file creation and management, I/O devices management, etc. An application programmer can use those functions to develop an application program. And once it is developed and executed in the environment where there are system programs, the application program can call some of the system programs to perform its own functions. And it is just the operating system that consists of a set of these system programs to take the responsibility of the details of the solution to the control and management of different computer resources, and builds a convenient software environment and interface for the application programmers to develop their applications for end users of computer systems.

Usually, the operating system provides some typical services for its users:

- Execute a program: When a program is executed, the operating system must load the instructions and data of the program into the main memory, initialize files and I/O devices, and prepare some other resources for its need.
- Create a program: The operating system should provide platforms and tools, such as editors, to assist the programmer in creating, editing and debugging programs.
- Operate files: For file management, the operating system must know not only the types of different files but also the characters of different storage devices. It also should provide the share and protection mechanisms of files.
- Control I/O devices: As there are so many types of I/O devices in the computer system and each of them has its own control instructions, the operating system must control I/O devices accurately when the application programs need them.
- Manage system and users: For multi-user computer systems, the operating system can not only let its users share system resources but also protect system resources, including CPU, primary memory, I/O devices and data, in order to make the execution of the operating system and applications smooth and proper. It also needs to make an account for each user in the system in order to collect some usage statistics and monitor the system performance.
- Detect and respond errors: As some errors can occur when the computers running, the operating system must detect errors and give an appropriate response in time. Different errors should be tackled in different ways, so

the operating system should give a variety of responses, such as reporting the error, retrying the operation, canceling the operation, or terminating the program that causes the error.

The UNIX operating system has solutions to the above services, which will be discussed detailed in the following chapters.

Operating systems are usually classified as three kinds according to numbers of users and the processes that operating systems can handle simultaneously.

- Single-user and single-process operating systems: Only one user at a time can use this kind of operating systems. And the user can run only one process at a time. Many older operating systems belong to this kind, such as MS-DOS, MacOS (Katzan 1970; Quarterman 1985; Stallings 1998; Tanenbaum 2005).
- Single-user and multiprocessing operating systems: Only one user at a time can use this kind of operating systems. But the user can run more-than-one processes at a time. Microsoft Windows XP Workstation is an example of this kind (Tanenbaum 2005).
- Multi-user and multiprocessing operating systems: More-than-one users at a time can use this kind of operating systems. And each of the users can run more-than-one processes at a time. They have examples, like UNIX, Linux, Microsoft Windows NT Server and Windows 2003 Server (Tanenbaum 2005).

On the other hand, as the evolution of operating systems, a contemporary operating system should have some major features, such as a multi-processing batch system and a time-sharing system. A multiprocessing batch system allows several programs to run concurrently, automatically without human intervention. UNIX and Linux execute programs in multi-processing batch mode, by running programs in the background (see Section 4.5.1). In a time-sharing system, several users access the system through different terminals simultaneously. So the operating system needs to switch the executing of each user program in a short period of computation in order to give a sole-user-in-system impression to each of the users. UNIX, Linux, Microsoft Windows NT Server, and Microsoft Windows 2003 Server are examples of the time-sharing systems.

1.2 Types of UNIX

In fact, the types of the UNIX operating system are almost countless (Kim 1999; Mann 1992; Martin 1995; Mckusick et al 2005; Perrone 1991; Perrone 1993; Riggs 1995). Since 1969, when Ken Thompson, Dennis Ritchie and others started working on an idle PDP-7 at AT&T Bell Labs and developed what was to become UNIX (Cooke 1999; McKusick 1999; Ritchie et al 1974; Sarwar 2006), many groups of developers and programmers, no matter who

came from the companies for commercial purposes or who were from universities for academic reasons, have been involved in different stages of UNIX development.

In the early 1970s, there was UNIX Time-Sharing System firstly (Cooke 1999; McKusick 1999; Ritchie 1974; Sarwar 2006). Since then, it has split into several branches. Even though there are assorted branches and some of their contributors moved from one branch to another or combined with others, two of the most influential are the System variant and BSD (Berkeley Software Distribution) variant. The former is the UNIX Time-Sharing System (Nieh et al 1997; Ritchie et al 1974), including six editions from System I to System VI. Even though at the very beginning it was developed just for programmers' own, this branch is more likely for the commercial purpose later, whose owner was AT&T. The latter was originated from the former because of one of the former developers, Ken Thompson, who installed one of UNIX operating system to a PDP-11/70 at the University of California, Berkeley (McKusick 1999). This branch tends more towards academic activities.

Linux is one of the popular, free members of the UNIX operating system. Another free derivative was 386BSD, which also led to FreeBSD (McKusick et al 2005), OpenBSD, and NetBSD. Now the UNIX operating system is widely applied in both servers and workstations. Linux and BSD are also expanding their application fields into such as consumer desktops, mobile phones, and embedded devices.

From 2007, the trademark UNIX® has been owned by the Open Group, an industry standards consortium, and only systems that are fully compliant with and certified to the Single UNIX Specification are qualified as "UNIX®" while others are called "Unix system-like" or "Unix-like". However, this result should be just for the commercial purpose. Thus, no matter whether it is "UNIX®" or "Unix-like", in this book, for academic research, it will be called one of the UNIX operating system, which may be more likely following the UNIX philosophy.

1.3 History of UNIX

As there are so many different types of the UNIX operating system and so many contributors involved into the UNIX development as well, it is necessary to learn the development history of the UNIX operating system. It can give a different angle to see how the UNIX development thoughts, which are usually brought into the basic theory of the textbooks of operating systems, influence the development of the whole operating systems.

No matter how many branches the UNIX operating system has, they have the same performances that are also called as the UNIX philosophy (Bach 2006; McKusick 1999; McKusick et al 2005; Quarterman et al 1985). These performances include: the use of plain text for data storage (see Section 3.1);

a hierarchical file system structure (see Section 6.1); treating devices as files (see Section 6.2); treating sockets that can be used inter-process communication as files (see Section 6.2); and the use of a large number of software tools that can be batched together through a command line interpreter by using pipes, creating a chain of producer-consumer processes (see Chapter 7); and UNIX shell which can be used to add new commands without changing the shell itself. Except these, the UNIX operating system is portable, multi-tasking and multi-user in a time-sharing configuration.

In the mid-1960s, AT&T Bell Labs worked on a collaborative project — an experimental operating system called Multics (Multiplexed Information and Computing Service). After AT&T Bell Labs pulled out of the project, one of the Multics developers, Ken Thompson, continued his development and led to start a new operating system for the PDP-7 at AT&T Bell Labs (Sarwar et al 2006; Thompson 1978). With the help of Dennis Ritchie, Ken Thompson and his team developed a multi-tasking operating system with a file system, which was the very beginning of the UNIX operating system.

During the 1970s, UNIX Time-Sharing System I to VI was developed sequentially. At that time, one breakthrough for UNIX and also for operating systems was to rewrite the UNIX operating system in the C programming language (Quarterman et al 1985; Ritchie et al 1974; Thompson 1978), contrary to the assembly language that, at that time, was considered as the general tool for such complex projects as an operating system, which must deal with time-critical events. And this also meant the UNIX operating system more portable — requiring only rewriting a relatively small amount of machine-dependent code in order to port the UNIX operating system from one computer system to another.

Another shinning idea, during that time, was related to modularizing the program code, which was resulted in scalability and expandability of the UNIX operating system.

In November 1973, the first UNIX paper of Ken Thompson and Dennis Ritchie was presented at the Symposium on Operating Systems Principles at Purdue University. During the conference, Professor Bob Fabry of the University of California at Berkeley obtained a copy of the UNIX system to test with at Berkeley. About one year later, Ken Thompson decided to take a one-year sabbatical as a visiting professor at the University of California at Berkeley, his alma mater, and brought up the latest UNIX, Version VI, to install in the PDP-11/70 at Berkeley. Later on, William N. Joy, Chuck Haley, Samuel J. Lefflerand , and others at Berkeley involved in the development of UNIX BSD versions. Meanwhile, Bob Fabry got a contract from Defense Advanced Research Projects Agency (DARPA) to add some features to BSD in order to meet the DARPA needs.

At the same time, the UNIX operating system went into two directions, one for the commercial use and the other for the academic purpose. The former was represented by UNIX System III and System V of AT&T; the latter were mainly Berkeley Software Distribution (BSD) releases of UNIX

developed at the University of California, Berkeley. During this time, BSD researchers brought to the UNIX operating system a few features such as the vi editor, C shell with job control, TCP/IP network code, and Berkeley Sockets application programming interface (API).

In the 1980s, some other companies started to develop their UNIX operating system for their own computers, such as Sun Microsystems' SunOS (Solaris later on), Microsoft's Xenix (SCO's SCO UNIX later on).

At the end of the 1980s, AT&T's SVR4 (System V Release 4) added new features, such as file locking, system administration, new forms of interprocess communication (IPC), the Remote File System, etc. And the UNIX-related standard — the IEEE's Portable Operating System Interface for UNIX POSIX (Portable Operating System Interface for UNIX) specification was established (Jespersen 1995; Walli 1995).

In the 1990s, splitting from BSD developers, William Jolitz led his team to develop the 386BSD, which was the free software pioneer for FreeBSD, OpenBSD, and NetBSD. At the same time, AT&T sold all its rights to UNIX to Novell, which developed the UnixWare. Linux, which was a typical UNIX-like system, was released as free software in 1992 and soon after became a popular operating system.

In the 2000s, many free UNIX operating systems are developed flourishingly, such as NetBSD and FreeBSD. And Sun Microsystems released its open-source OpenSolaris (German et al 2010). And most of the programs and scripts in this book have been debugged and passed on FreeBSD.

1.4 Summary

An operating system (OS) is usually in between the user's application programs and the computer hardware, which operates the computer hardware resources to provide a set of services for the computer system users. Usually, the operating system provides some typical services for its users: execute a program, create a program, operate files, control I/O devices, manage system and users, and detect and respond errors.

Operating systems usually have three types according to the numbers of the users and the processes that operating systems can handle simultaneously: single-user and single-process operating systems, single-user and multiprocessing operating systems, and multi-user and multiprocessing operating systems. UNIX, Linux, Microsoft Windows NT Server, and Windows 2003 Server belong to the multi-user and multiprocessing operating systems.

Among the UNIX operating systems, two of the most influential are System III and V variant and BSD variant. The former is more likely for the commercial use, whose owner was AT&T; the latter tends more towards academic use, whose place of origin was the University of California, Berkeley. Linux is one of the popular, free members of the UNIX operating system.

No matter how many branches the UNIX operating system has, the UNIX characters are the same, including: the use of plain text for data storage, a hierarchical file system structure, treating devices as files, treating sockets as files, the use of a large number of small programs that can be batched together through a command line interpreter using pipes, creating a chain of producer-consumer processes, and UNIX shell which can be used to add new commands without changing the shell itself. In addition, the UNIX operating system is portable, multi-tasking and multi-user in a time-sharing configuration.

Problems

Problem 1.1 What is an operating system? What typical services can operating systems provide?

Problem 1.2 How many kinds can operating systems usually be classified, according to the numbers of the users and the processes that operating systems can handle simultaneously? What are they? Which kind does the UNIX operating system belong to?

Problem 1.3 What are the two of the most influential UNIX operating systems?

References

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Cooke D, Urvan J, Hailton S (1999) UNIX and beyond: An interview with Ken Thompson. Computer 32(5): 58 – 64

German DM, Penta MD, Davies J (2010) Understanding and auditing the licensing of open source software distributions. IEEE 18th International Conference on Program Comprehension, Braga, Minho Portugal, June 2010: pp 84 – 93

Jespersen H (1995) POSIX retrospective. ACM, StandardView 3(1): 2 – 10

Katzan H Jr (1970) Operating system architecture. AFIPS: Spring Joint Computer Conference, Atlantic City, New Jersey, 5 – 7 May 1970: pp109 – 118

Kim T, Shin G, Hong E (1999) Experience with porting a UNIX version case tool to the corresponding Java version. APSEC'99: IEEE sixth Asia-Pacific software engineering conference, Takamatsu, December 1999: pp 622 – 629

Lawrence DC (1998) Internetnews server: Inside an open-source project. IEEE Internet Comput 2(5): 49 – 52

Mann D (1992) UNIX and the Am29000 microprocessor. IEEE Micro 12(1): 23 – 31

Martin VC (1995) There can be only one! A summary of the UNIX standardization movement. ACM, Crossroads 1(3): 9 – 11

McKusick MK (1999) Twenty years of Berkeley Unix: from AT&T-owned to freely redistributable. LINUXjunkies.org. http://www.linuxjunkies.org/articles/kirkmck.pdf. Accessed 20 Aug 2010

McKusick MK, Neville-Neil GV (2005) The design and implementation of FreeBSD operating system. Addison-Wesley, Boston

Nieh J, Lam MS (1997) Smart UNIX SVR4 support for multimedia applications. ICMCS'97: The IEEE International conference on multimedia computing and systems, Ottawa, Ontario, Canada, June 1997: pp 404 – 414

Perrone G (1991) Alternate versions of UNIX. Computer 24(11): 84 – 90

Perrone G (1993) The Macintosh A/Ux operating system release 3.0. Computer 26(2): 103 – 106

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading, Massachusetts

Riggs B, English E (1995) UNIX transformations. Computer 28(4): 8 – 9

Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17(7): 365 – 375

Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: The textbook, 2nd edn. China Machine Press, Beijing

Stallings W (1998) Operating systems: Internals and design principles 3rd edn. Prentice Hall, Upper Saddle River, New Jersey

Tanenbaum AS (2005) Modern operating systems, 2nd edn. China Machine Press, Beijing

Thompson K (1978) UNIX Implementation. Bell Syst Tech J 57(6), part 2: 1931 – 1946

Walli SR (1995) The POSIX family of standards. ACM, StandardView 3 (1): 11 – 17

2 How to Start

Before using the UNIX operating system, a brief introduction of the components and software structure of UNIX will be given. And it is also necessary for the users to know the working environment in UNIX in order to start to work in there and use its facilities. After learning the difference between a character user interface (CUI) and a graphical user interface (GUI) and also benefits of a CUI, it is helpful to learn the UNIX command lines and UNIX window systems. A UNIX shell is an important interface between the user and the operating system, and also a programming tool for users to write their own commands. This chapter will introduce only the shell setup files, and the detailed discussion on UNIX shells will be given in Chapter 8.

2.1 UNIX Software Architecture

Figure 2.1 shows a layered view for a UNIX-based computer system, identifying the system's software components and their logical relationship with the user and hardware. Following will be given each software layer from the bottom up.

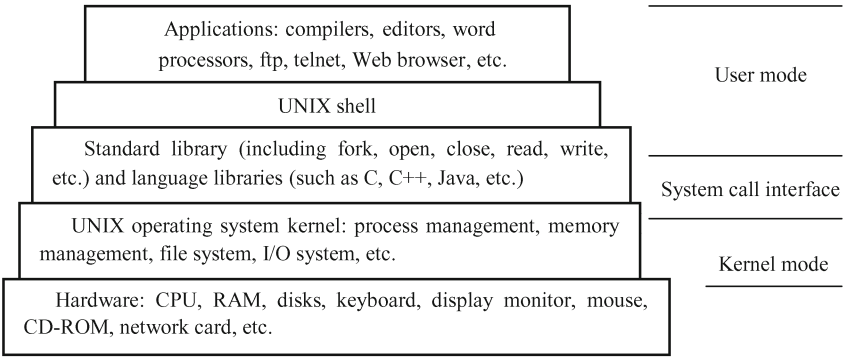

Fig. 2.1 The UNIX software architecture.

2.1.1 UNIX Kernel

The UNIX operating system kernel is the core of the system (Bach 2006; McKusick et al 2005; Nguyen et al 1997; Quarterman et al 1985; Spinellis 2008; Stallings 1998). Its main functions are shown in Figure 2.1, which are the process management, memory management, file system, and I/O system. Many application subsystems and programs that provide a high-level performance of the system, such as the shell and editors, have gradually been seen as features of the UNIX operating system. However, they all use lower-level services ultimately provided by the kernel, and they avail themselves of these services via a set of system calls.

2.1.1.1 Process Management

A program is an executable file while a process is an instance of the program in execution. Processes have "life" (Braams 1995; McKusick et al 2005; Stallings 1998). When it is created with the fork system call, a process is brought to be active in the system; when it is terminated for some reasons, it is dying out from the system. Many processes can execute concurrently on the UNIX operating system (this feature is also called multiprogramming or multitasking) with no limit to their number logically. Some system calls allow processes to create new processes, terminate processes, synchronize stages of process execution, and control reaction to various events.

In a UNIX system, the only active entities are the processes. Each process runs a single program and initially has a single thread of control. Process management of the UNIX operating system is typically responsible for tasks: to create, suspend and terminate processes, to switch their states, to schedule the CPU to execute multiple processes concurrently in a time-sharing system, and to manage the communication between processes (inter-process communication, IPC).

Pipes are one of IPC mechanisms of the UNIX operating system. A pipe is a temporary unidirectional channel in the main memory, which is created by the kernel for two processes, and into which one process can write a stream of bytes for the other to read. In other words, by using a pipe, two producer-consumer processes can be connected together so that the output from the producer becomes the input of the consumer in order to transfer data between them in the first-in-first-out (FIFO) way. Synchronization is attained because when a process tries to read from an empty pipe it is blocked until data are sent by the other.

Compared to the temporary pipe, a permanent channel that is located on the disk is called a named pipe which can be used for IPC by two or more processes that are running on the same computer.

Signals can be used for handling exceptional conditions, such as a keyboard interrupt, an error in a process, or a number of asynchronous events. Almost all the signals can be generated by the kill system call. With signals,

it is implemented to control processes asynchronously with events.

2.1.1.2 Memory Management

This part of kernel job is to keep 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 and deallocate it when they finish their tasks and put this memory space back in the free space pool so that it can be reallocated, and to manage swapping some processes between main memory and disk when main memory is not enough to hold all the processes (Ritchie et al 1974). It protects the memory space of a process from entrance by others. It also manages the share memory space for processes.

With a demand-paged virtual memory system (Bach 2006; McKusick et al 2005; Quarterman et al 1985; Rashid et al 1988), the kernel can implement paging for processes without size limitations. In other words, the kernel can load its related parts into main memory dynamically and execute it even though other portions are not in memory. In this way, more tasks can be kept in memory and swapping can be reduced to a minimum.

2.1.1.3 File System

Files are also managed by the kernel. How they are structured, named, accessed, used, protected, and implemented is handled by the file system of the kernel. File systems are stored on disks. Usually, in directories, there are files and sub-directories. In UNIX, a directory is also a file. The kernel manages files and directories (also called folders) by performing to create and remove files and directories, to change file and directory attribute, to protect files and directories according to their attribute and access modes, and to share files among several users, etc.

In UNIX, except directories and ordinary files, there are also special device files, sockets, pipes, and links kept in the file system (Bach 2006; Badger et al 1995; Quarterman et al 1985; Sarwar et al 2006). Thus users can access devices in the same way as to reference ordinary files via the file system.

In the storage, each file has its unique inode. When a file is accessed by a user, the kernel needs to load its inode from the disk into memory and give the user a file descriptor directing to its inode.

Sockets are another type of facilities in the UNIX kernel, which are used for a general interface, such as networking. Links, otherwise, are different references towards a certain file.

2.1.1.4 I/O System

In UNIX, all I/O devices are made to look like files and are accessed as such with the same read and write system calls that are used to access all ordinary files. I/O devices are those connected to the computers, such as, disks, printers, CD-ROM, keyboard, display, and network cards. UNIX operating systems integrate the devices into the file system as what are called special files (Carson et al 1992; Heindel et al 1995; Nelson et al 1996). Each I/O

device is assigned a path name (see Section 6.2). Various hardware devices have different drivers, such as the hard disk driver, floppy disk driver, CD-ROM driver, keyboard driver, mouse driver, and display driver. When a user command or application needs to perform a hardware-related operation, the UNIX kernel calls these driver programs via their special files. The user does not have direct access to these programs.

2.1.2 System Call Interface

As in UNIX, the kernel manages all system resources, any user application request that involves access to any system resource must be asked to the kernel code. For security reasons, user processes must not access to the kernel code. The UNIX operating system provide function calls, known as system calls, which user processes can use to invoke the execution of kernel code. So the system call interface provides entrances into the UNIX kernel. Some services of system calls include performances like creating and terminating processes, creating, deleting, reading and writing files, managing directories, and performing input and output. Among about 64 system calls in System V, nearly half of them is used frequently. The kernel body is composed of the set of internal algorithms that implement system calls. POSIX (Portable Operating System Interface for UNIX, IEEE) (Isaak 1990; Osaak et al 1998; Jespersen 1995) defines about 100 system calls, which determine most of what the operating system has to do. Programs that follow the POSIX standard can be easily moved from one hardware system to another. POSIX was based on UNIX operating system services, but it was programmed in a way that allows it to be implemented by other operating systems. Table 2.1 lists some of POSIX system calls.

Table 2.1 Some of POSIX system calls

Process management	
System call program	Service
fork	To create a child process identical to the parent
waitpid	To wait for a child to terminate
execve	To replace a process' core image
exit	To terminate process execution and return status
File management	
System call program	Service
open	To open a file for reading, writing or both
close	To close an open file
read	To read data from a file into a buffer
write	To write data from a buffer into a file
lseek	To move the file pointer
stat	To get a file's status information

Continued

Directory and file system management	
System call program	Service
mkdir	To create a new directory
rmdir	To remove an empty directory
link	To create a new entry and let it point to another entry in a directory
unlink	To remove a directory entry
mount	To mount a special file system
umount	To dismount a special file system
Miscellaneous	
System call program	Service
chdir	To change the working directory
chmod	To change a file's protection bits
kill	To send a signal to a process
time	To get the elapsed time since Jan. 1, 1970

Usually, the system calls allow users to write programs that do complex operations, and therefore, the UNIX kernel does not contain many functions that are part of the "kernel" in other operating systems, such as functions of compilers and editors, which are user-level programs (applications shown in Figure 2.1) in the UNIX operating system.

When a process executes a system call, the execution mode of the process is changed from user mode to kernel mode (as shown on the right column of Figure 2.1). At that moment, the operating system executes and attempts to service the user request, returning an error code if it fails. Even if no user process does request for system services, the UNIX operating system still does bookkeeping operations that are related to the user process, such as controlling process scheduling, managing memory, and so on. The differences between the two modes are as follows.

- Processes in user mode can access their own instructions and data but not kernel instructions and data. Processes in kernel mode, however, can access both kernel and user addresses. Assumedly, the virtual memory address space of a process may be divided between addresses that are accessible only in kernel mode and addresses that are accessible in either mode.
- Some machine instructions are only executed in kernel mode (for example, to manipulate the processor status register), and result in an error when executed in user mode.

Frequently, the hardware system supports the switch between kernel and user modes. Although the system executes in one of two modes, the kernel runs on behalf of a user process. The kernel is not a separate set of processes that run in parallel to user processes, but it runs as part of each user process. In other words, the system calls allow processes to do operations that are

otherwise forbidden to them.

2.1.3 Standard Libraries and Language Libraries

The user interface to UNIX is composed of the shell and the programs of the standard libraries. The POSIX standard (Isaak 1990; Osaak et al 1998; Jespersen 1995) specifies the syntax and semantics of file and directory manipulation commands, filters, and program development tools (such as editors and compilers). The idea of standardizing them is to make it possible for any programmer to write shell scripts that use these programs and work on all UNIX operating systems.

Language libraries are a set of functions available to programmers for use with the software that they develop. The availability and use of libraries can save programmers' time and energy because they do not need to write these functions from scratch and just link them to their own programs when using them. Those language libraries can be C, C++, Java, and FORTRAN.

2.1.4 UNIX Shell

The UNIX shell is a command line interface or CUI (see Section 2.3). When a user types a command and press Enter key on the keyboard, the shell interprets and executes the command. The shell searches for commands in a given sequence of directories changeable by user request per invocation of the shell. The shell usually executes a command synchronously, which means that it waits for the command to terminate before reading the next command line. But, it also allows asynchronous execution, where it reads the next command line and executes it without waiting for the prior command to terminate. Commands executed asynchronously can be implemented by putting them to execute in the background.

Every UNIX operating system comes with a variety of shells, among which the Bourne, Korn, and C shells are the most popular (Berkman 1995; McKusick et al 2005; Mohay et al 1997; Sarwar et al 2006). When logging on, one particular type of shell executes. This shell is the login shell. To use a different shell, do so by running a corresponding command available on the UNIX operating system.

2.1.5 Applications

The applications contain all the applications (such as, compilers, editors,

word processors, ftp, telnet, Web browser, etc.) that are available for users. A typical application is usually related to a program that can be executed by typing a command line and Enter key. When an application that needs to manipulate a system resource (e.g., an editor from which to read or write a file), it needs to invoke the kernel that performs the task. A user can use any command or application that is available on the system and interplay with the UNIX operating system through the interface known as the application user interface (AUI). Most of UNIX operating systems contain hundreds of applications; the commonly used applications will be discussed throughout this book. For example, editors will be introduced in Chapter 3 Text Editors.

2.2 UNIX Environment

When beginning to use UNIX, a user should be set up a UNIX account that is identified by the username. Sometimes, there will be a whole network of UNIX computers. So in addition to knowing the username, the user may also need to know the hostname of the computer that has the user account. Alternatively, the account may be shared among all computers on the local network, and the user may be able to log into any of them.

A user communicates with the computer from a terminal. To get into the UNIX environment, the user has two ways to do it: to log into the local computer, or to use a remote login command to connect to the remote computer.

To log in UNIX, the user needs to type in the username and password, which the UNIX uses to identify the user and prepare the working environment for the user according to the user account. When finishing work, the user should log out from UNIX.

Usually, the UNIX operating system is case sensitive, and most of the commands and file names use lowercase letters.

Two kinds of the UNIX environment can be logged into: a window system or a terminal system.

A typical log-in procedure is as follows:

```
login: username Enter
password: user_password Enter (Comment: In this book, Enter means
                    pressing the Enter key of the keyboard.)
```

When the login prompt appears on the screen, the user needs to keystroke the valid username, and then press Enter. When the password prompt appears, type in the valid user password, and then press Enter. Then some message, such as announcing news, important information, or having new mail, will appear on the screen. If logging into a terminal system, following is the shell prompt, for example, $ (which is Bourne shell default prompt), appearing on the screen. From there, the user can type in and execute commands. Therefore, once having logged into the account, the user will interact

with the UNIX operating system by typing commands at a command line to the shell. If it is a window system, a typical window that usually looks like Microsoft Windows will appear. From there, a terminal window where the user can interact with the shell and which looks like a mini-terminal, will be found.

If logging into a terminal system, to terminate the connection with the UNIX operating system, the user should type in logout after the shell prompt and press Enter, or press and hold on two keys CTRL-D on the keyboard after the shell prompt. Then log out the UNIX.

If logging into a window system, to end the connection with the UNIX, the user should click the main menu and choose the item of exit to log out the system.

2.3 Character User Interface Versus Graphical User Interface

To operating systems, usually, there are two kinds of user interfaces, a character user interface (CUI), also known as a command line interface, and a graphical user interface (GUI) (Sarwar et al 2006). Some operating systems have both of them; some don't, such as DOS. Among those that have both user interfaces, some have a CUI as their primary interface but allow running a GUI, such as most of UNIX operating systems, while others primarily offer a GUI, but have the capability to allow a user to enter a DOS- or UNIX-like command-line-based screen, for example, Microsoft Windows.

To a user of a computer system, user interfaces of operating systems give facilities to interact with the computer systems. With a CUI, the user can give commands to the operating system by an input device, such as a keyboard, to issue a command. Otherwise, via a GUI, the user can use a mouse as an input device to point and click some icon to issue commands.

Though UNIX operating systems usually have a CUI as their main user interfaces, they can process software based on the X Window System (Project Athena, MIT) that provides a GUI interface, which is similar as Microsoft Windows and users can get used to quickly if using it. Moreover, most of the UNIX operating systems now have an X-based GUI.

Even though a GUI brings a better-looking and easier use to the user, it gives less flexibility and slower task-processing. The reason underneath this is that a GUI is an extra layer above a CUI in the operating system between the user and the user's task to run on the computer. In fact, a CUI ultimately controls the computer system and processes programs that the user wants to run. For users, especially those programmers who want to get more control to the computer system and to run their programs faster, a CUI is more efficient and flexible.

As mentioned in Section 2.2, if logging into a terminal system, the user

will interact with a CUI; if logging into a window system, the user will work with a GUI.

Maybe for readers who have gotten used to Microsoft Windows, a CUI can be a challenge. However, for the promising professional programmers, it will pay off when they get accustomed to a CUI after several hands-on sessions and begin to enjoy its ability of control and flexibility.

2.4 UNIX Command Lines

As a text-based interface, or a CUI, is frequently a main interface for UNIX, it is important for a UNIX user to know how to type in a command correctly with the keyboard. In this section, UNIX command syntax will be introduced firstly, and then some common and useful commands will be studied separately. These commands and others that will be introduced in the following chapters in the whole book should be typed in on a command line in a terminal system or a terminal window. When a command is typed in and Enter pressed, the command will be interpreted and executed, and finally the result will appear on the screen and a new prompt will also appear on a new command line, where new commands can be typed in.

Modern UNIX shells can remember command lines typed before. This feature can save a lot of time retyping common commands. If the shell is configured as having this capability, try pressing up-arrow key, and then the previous command line can appear after the new shell prompt, just as it was typed before. If pressing up-arrow key again, the more previous command line can appear after the new shell prompt. With up-arrow key and down-arrow key together, make the commands forward and backward. To execute the command, press the Enter key no matter where the cursor is. To edit it, just do what to do. If not to execute it, just cancel it with pressing two keys CTRL-C.

If having made a typing mistake, just correct or retype it by pressing the following keys of the keyboard. It depends on the computer system and whether or not the following keys are configured to work in UNIX.

- Left-arrow key: Move the cursor left along the command line to the point where to make a change.
- Right-arrow key: Move the cursor right along the command line to the point where to make a change.
- BACKSPACE, DELETE, or CTRL-H: Erase the previous character and move the cursor to the previous character position.
- CTRL-U: Erase the whole current line and move the cursor to the beginning of the current line.
- CTRL-C: Terminate the current command and move the cursor to the beginning of the next line.
- CTRL-S: Stop scrolling of output on the screen.

- CTRL-Q: Restart scrolling of output on the screen that has been stopped by CTRL-S.

Each combination of two keys means to press and hold down the first key while pressing the second key. For example, CTRL-S is to press and hold down the CTRL key and to press the S key at the same time.

2.4.1 UNIX Command Syntax

A UNIX command may or may not have options and arguments. An argument can be a filename or not. And a command can be the name of a UNIX program (such as date), or it can be a command that's built into the shell (such as exit).

General format for UNIX command lines:

```
$ command [-option(s)] [argument(s)]
```

where $ is the shell prompt. There is not a total set of rules for writing UNIX commands, but basic rules in most cases are as the below:

- Type in commands in lowercase.
- Options change the way in which command works. Options are often single letters prefixed with a dash (-, also called the hyphen) and set off by any number of spaces or tabs. Multiple options in one command line can be set off individually, but not always. For example, -la is two options, l and a, which will function together.
- Some commands have options made from complete words and with two dashes, like -delete.
- The argument is the name of an object that the command works on. The argument can be a filename, but some commands have arguments that are not filenames. A command can have more than one argument simultaneously.
- There must be spaces between commands, options, and arguments.
- Options usually come before arguments. In a few cases, an option has its own argument associated with it; type this special argument just after its option. For example:

```
$ sort -o outputfile -n inputfile
```

which means sort reads the file named inputfile, and writes to the file named outputfile.

- Two or more commands can be written on the same command line, each separated by a semicolon (;). Commands entered this way are executed one by one by the shell.
- End one command typing by pressing the Enter key of the keyboard. The following commands are valid.

```
$ ls
$ ls -al
$ ls -l filename1
```

```
$ sort -o aa1 -n bb1
```

The first one has no option and argument. The second one has two options. The third one has one option and argument. The fourth one has two options with their own arguments, respectively.

If a mistake has been made when typing a command, no matter what type of mistake it is, an error message will be shown. This makes an important point about the execution of UNIX commands. If no error message displayed, the command is executed correctly, and the results may or may not appear on the screen, depending on whether or not the command actually has a display result. If an error message displayed, correct the error in order for the UNIX operating system to execute the command.

As most of the readers are familiar with directories and files, we will start from the directory and file operation commands. The next several sections will give them a try on how to use them. More details about how the kernel to service these commands will be discussed in the following chapters.

2.4.2 Directory Operation Commands

All directories on a UNIX operating system are organized into a hierarchical structure, like a family tree. The parent directory of the tree, which contains all other directories, is known as the root directory and is written as a forward slash (/). Figure 2.2 shows a directory tree of a UNIX file system where there is one root directory and some directories under the root.

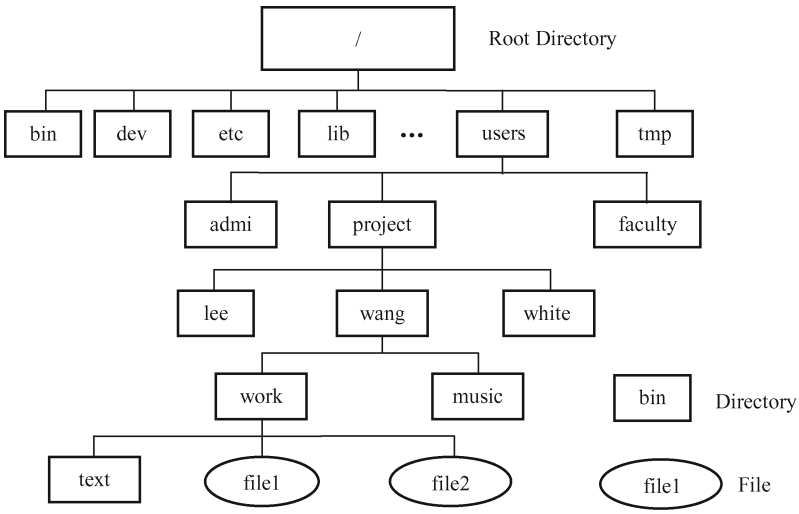

Fig. 2.2 Example of a UNIX directory tree.

In Figure 2.2, the files named file1 and file2 are stored in the directory named work. Beneath the directory work, there is a subdirectory called text.

On a UNIX operating system, each directory has only one parent directory, but it may have one or more subdirectories (such as the directory users in Figure 2.2). A subdirectory can have its own subdirectories, up to a limitless depth for practical purposes.

To specify a file or directory location, write its pathname. A pathname is like the address of the directory or file in the UNIX filesystem, which is a textual way of designating the location of a directory or file in the complete filesystem structure. A pathname uses slashes (/) between the directory names. For example, the path to the file file2 in Figure 2.2 is /users/project/wang/work/file2. The designation of the path begins at the root (/) of the entire file system, descends to the directory called users, and then descends to the directory called project, and then descends to the directory named wang, and then descends again to the directory named work.

Some commands that a user can use to operate directories are listed in different groups according to their functions.

2.4.2.1 Printing Working Directory

When logging in, a user is placed in a directory, called the home directory, associated with the username and password of the user. Whatever directory the user is presently in is known as the working directory (or current directory). Every time logging in, the home directory is the working directory. And if changing to another directory, the directory having been moved to becomes the working directory.

To find which the working directory is, use pwd (print working directory) command.

The syntax and function of the pwd command are as follows.

```
$ pwd
```

Function: to show the working directory on the screen.

Common options: none.

For example:

```
$ pwd
/users/project/wang
$
```

The output of this example shows the working directory is /users/project/wang. Each time typing in a command, readers just type the command name (with options and arguments if necessary) after the shell prompt (which pops out automatically) and press the Enter key.

2.4.2.2 Changing Directory

As the UNIX filesystem organizes its files and directories in an inverted tree structure with the root directory at the top, an absolute pathname tells the path of directories from the root to the directory where the file or subdi-

rectory is. A pathname uses slashes (/) between the directory names. The root is always indicated by the slash (/) at the start of the pathname. Therefore, an absolute pathname always starts with a slash. For example, /users/project/wang is an absolute pathname.

A relative pathname gives the location relative to the working directory. Relative pathnames start at the working directory, not the root directory. That is, a relative pathname never starts with a slash (/). For example, project/wang is a relative pathname.

To change the working directory to any directory, use cd command.

The syntax and function of cd command are as follows.

```
$ cd [pathname]
```

Function: to change the working directory to the directory that 'pathname' directs, or return to the home directory when without an argument.

Common options: none.

Note: The argument—pathname is an absolute or a relative pathname. cd can only change to another directory, but cannot to a filename.

For example:

```
$ cd /users/project/wang/work
$ pwd
/users/project/wang/work
$ cd text
$ pwd
/users/project/wang/work/text
$
```

The first command changes the working directory to the directory /users/project/wang/work. The output of pwd shows now the working directory is /users/project/wang/work. The third command changes the working directory to the directory /users/project/wang/work/text. The output of the fourth command pwd shows the execution result of the third command.

2.4.2.3 Creating a Directory

To create a new directory under the working directory, use the mkdir command.

The syntax and function of mkdir command are as follows.

```
   $ mkdir [option(s)] dirname(s) (Comment:(s) means there may be one
or more than one option (or argument). The rest of commands in this
book that adopt this kind of notation have the same meaning.)
```

Function: to create a directory or more named 'dirname(s)' in the working directory.

Common options:

-m MODE: to create a directory with given access modes;

-p: to create parent directories that do not exist in the pathnames specified in 'dirname(s)'.

For example:

```
$ mkdir myfile
```

```
$
```

The command creates a subdirectory named myfile under the working directory.

2.4.2.4 Deleting Directory

To delete a directory, use the rmdir command.

The syntax and function of rmdir command are as follows.

```
$ rmdir [option(s)] dirname(s)
```

Function: to delete an empty directory or more specified 'dirname(s)'.

Common options:

-p: to delete parent directories at the same time.

Note: rmdir can only remove empty directories, but cannot delete nonempty directories.

For example:

```
$ rmdir text
rmdir:  text:  Directory not empty
$ rm -r text
$
```

The first command tries to delete the directory named text, but as 'test' is not an empty directory, the error message that the rmdir command cannot delete the directory returns. So the second command empties the directory by using the rm command with the -r option. The -r option recursively descends down into the subdirectory and deletes every file in it before actually deleting the directory itself. However, be careful to use this command in order to avoid deleting all the files and subdirectories in accident. The second command deletes the directory text.

2.4.2.5 Listing Files in a Directory

To list the files and subdirectories contained in a directory, use the ls command.

The syntax and function of ls command are as follows.

```
$ ls [option(s)] [dirname(s) filename(s)]
```

Function: to display and list the names of the files and directories in the directories, or just names of files, which're specified in 'dirname(s) filename(s)'; or the names of the files and directories in the working directory without an argument.

Common options:

-l: to display long list that includes file access modes, link count, owner, group, file size (in bytes), and modification time;

-a: to display names of all the files, including hidden files;

-i: to display i-node numbers;

-F: to put a slash (/) after a directory, an asterisk (*) after an executable file and an at-sign (@) after a symbolic link.

Note: If a complete pathname specification is given to the argument of the ls command, the names of files and directories along that pathname will be listed.

For example:

```
$ ls
text file1 file2
$
```

This command lists the files and directories in the work directory.

Another example:

```
$ ls -la
Total 34
drwxr-xr-x     2  wang  project  512 Apr 15 15:11  .
drwxr-xr-x     2  admi  admi     512 Apr 12 15:01  ..
-rw-r--r--     2  wang  project  136 Jan 16 11:48  .exrc
-rw-r--r--     2  wang  project  833 Jan 16 14:51  .profile
-rw-r--r--     1  wang  project  797 Jan 16 15:02  file1
-rw-r--r--     1  wang  project  251 Jan 16 15:03  file2
drwxr-xr-x     2  wang  project  512 Apr 15 15:11  text
...
$
```

The second command has two options, l and a, which change the behavior of the command ls. So the output on the screen is a longer list of all ordinary and the hidden files (system files, such as .exrc and .profile), as well as provide other related information about the files. Do not forget to put the space character between the ls and -la.

The first row of the display result is 34 blocks of disk storage used by this directory. On many systems, a full block has 1024 bytes. A block can be partly full.

Please pay attention to the information in each of the following long rows. There are eight columns in each row. The detail is as follows.

- The first character of the first column: The type of file. b stands for block special file; c stands for character special file; d stands for directory; l stands for symbolic link; p stands for named pipe; s stands for socket; - stands for ordinary file.
- The rest characters of the first column (nine characters): The access permissions to that file for user, group, and others; three characters for each of them, respectively; r stands for read, w stands for write, and x stands for execution.
- The second column: The number of links to the file.
- The third column: The username of the owner of the file.
- The fourth column: The name of the group for the file.
- The fifth column: The number of bytes that the file occupies on disk.
- The sixth column: The date that the file was last modified.
- The seventh column: The time that the file was last modified.
- The eighth and final column: The name of the file.

The above ls command with options -la gives more complete information about the file. Its access modes can let the user know different types of users

having different rights to that file. More information about access modes will be discussed in Chapter 6.

The ls command can also be used to get the detailed information of a file. For example:

```
$ ls -la file1
-rw-r--r--     1  wang   project  797 Jan 16 15:02   file1
$
```

This command lists the information of the file1.

2.4.3 File Operation Commands

Both files and directories are identified by their names. As mentioned above, a directory is a special kind of file, so the rules for naming directories are the same as the rules for naming files. Some reserved characters are not good choices for names of files and directories, such as /. For a name of a file or directory, it is better to use only upper- and lowercase letters, numbers, dots, and underscore characters (_). Other characters, including spaces, are legal in a filename, but they can be confusing to use because the shell gives them special meanings. Do not use a dash as the first character of a filename because it will be confused with an option. Do not use a space in the filename because the shell breaks command lines into separate arguments at the spaces. To tell the shell not to break an argument at spaces, put quote marks (“) (”) around the argument (see Section 8.5).

Unlike some other operating systems, UNIX does not require a dot (.) in a filename as an extension added to a filename to identify a file as a member of a category, for example, .com or .exe for executable programs that MS-DOS can load and run. In the UNIX operating system, they can be used freely.

Some UNIX operating systems restrict filenames to 14 characters. Most newer UNIX operating systems allow much longer filenames. A filename must be unique inside one directory, but two directories may have files with the same names.

There are several commands that can be used to manage files. Some of them are discussed in this section. Some more will be introduced in different and proper sections in some following relevant chapters.

2.4.3.1 Copying File

To copy a file, use the cp command.

The syntax and function of cp command are as follows.

```
$ cp [option(s)] oldford newford
```

Function: to copy the source file or directory named oldford to the destination file or directory named newford.

Common options:

-p: to preserve file access modes and modification times;
-i: to prompt before overwriting if destination exists;
-r: to copy files and subdirectories recursively.

Note: cp command can put a copy of a file into the same directory (if different names for the files are used) or into another directory (it does not matter if the names are different or not). The latter needs a pathname for oldfile or newfile. cp does not affect the original file, so it's a good way to keep an identical backup of a file.

For example:

```
$ cp /etc/password password
$
```

This command makes a copy of the /etc/password file into a file called password in the working directory.

Another example:

```
$ cp oldfile edirectory
$
```

This puts a copy of the original file oldfile into an existing directory edirectory.

By using cp command, more than one file can be put into a single directory, such as

```
$ cp ../music/m1 ../music/m2 ../music/m3 text
$
```

This command copies three files called m1, m2 and m3 from /users/project/wang/music to a subdirectory called text (that is /users/project/wang/work/text) if now the working directory is /users/project/wang/work in Figure 2.2. The ".." means the parent directory of the working directory.

The following example can make a total copy of a directory in another directory.

```
$ cd /users
$ cp -r project/wang/work faculty/wang/work
$
```

This command uses the option -r, for recursive, to copy all files and subdirectories from /users/project/wang/work to /users/faculty/wang/work.

2.4.3.2 Renaming File

To rename a file, use the mv command (move). The mv command can also move a file from one directory to another.

The syntax and function of mv command are as follows.

```
$ mv [option(s)] oldfile newfile
$ mv [option(s)] ofile ddirectory
```

Function: to rename a file named oldfile into newfile; or to move ofile into the destination directory named ddirectory.

Common options:

-i: to prompt before overwriting if destination exists.

Note: Unlike cp, mv does affect the original file. After using mv command, the original file will not exist.

For example:

```
$ mv file1 file3
$
```

The command renames the file named file1 as file3. The file1 file has disappeared.

2.4.3.3 Deleting File

To delete a file, use the rm command.

The syntax and function of rm command are as follows.

```
$ rm [option(s)] filename(s)
```

Function: to delete a file or more specified 'filename(s)'.

Common options:

-i: to prompt before deleting files;

-r: to delete files and subdirectories recursively if filename(s) are directories.

Note: If doing cp with -r option for a directory fails, rm with -r option can be used to remove the directory cleanly. But be careful of using it because it can delete the storage information in the directory totally.

For example:

```
$ rm file2
$ rm -r text
$
```

The first command deletes the file named file2. The second command empties and deletes the directory text by using the rm command with the -r option.

2.4.3.4 Looking inside Files

There are different ways to look inside a file. Here, two ways will be given: one is to use the more command, the other is to use the less command. Maybe some UNIX operating systems do not have the less command, but most have the more command. The more and less commands can display one "page" (a terminal or a terminal window filled from top to bottom) of text at a time, so they are also called pager programs. In some UNIX operating systems, the pg command is available, which also has the same function.

The syntax and function of more command are as follows.

```
$ more [option(s)] filename(s)
```

Function: to display the files in 'filename(s)' on the screen, one screen at a time.

Common options:

-n number: to display the 'number' of lines per page;
-c: to clean the screen before displaying files;
-e: to exit more after displaying the last line;
-s: to compress several blank lines into one line to display.

Note: Since the more command shows the file one screen at a time, if the file has several pages, to proceed to view the next page, press the Space key on the keyboard. To see the next line, enter the Enter key. Press the Q key to quit the more command.

For example:

```
$ more file2
(The content of the file named file2 is displayed here.)
$
```

As different UNIX operating systems may have more, pg or less commands, the more command in different UNIX operating systems may have different options. To be sure what they have, check out their user manuals or their help document online.

The syntax and function of less command are as follows.

```
$ less [option(s)] filename(s)
```

Function: to display the files in 'filename(s)' on the screen, one screen at a time.

Common options:

-M: to make the less prompt show the filename and the current position in the file.

Note: The less command has a prompt and a command line at the bottom of one display page. The cursor sits to the right of the prompt, where less inside commands can be typed in. For example, press the Space key on the keyboard to proceed to display the next page. Type the Enter key to display the next line. Enter the H key to get the less help document. Press the Q key to quit the less command.

For example:

```
$ less file2
(The content of the file named file2 is displayed here.)
$
```

To see what the less inside commands and options available on the UNIX operating system, press the H key to get the less help while less is running. The more or less commands can also be combined with other commands through pipes (see Chapter 7), which will make them more useful and flexible.

2.4.3.5 Joining and Displaying Files

To display a file or several files in sequence, the cat (short for "concatenate") command can accomplish this job.

The syntax and function of the cat command are as follows.

```
$ cat [option(s)] filename(s)
```

Function: to join and display a file or more files specified 'filename(s)' in a terminal or a terminal window in sequence.

Common options:

-e: to display $ at the end of each line;

-n: to put line numbers on the displayed lines.

Note: Unlike the less and more commands, the cat command is not a pager program. If the whole content of the displayed files has several pages, it cannot stop until the final page is displayed in a terminal or terminal window. Also, it is not possible to go back to view the previous screens. But the cat command is really useful when combined with redirection (see Chapter 7).

For example:

```
$ cat file2
(The content of the file named file2 is displayed here.)
$ cat file1 file2
(The contents of the files named file1 and file2 are displayed in
sequence here.)
$
```

The first command displays the content of the file named file2. The second command shows the contents of the files called file1 and file2 sequentially in the terminal or terminal window.

2.4.3.6 Finding Files

To find files, use the find command.

The syntax and function of find command are as follows.

```
$ find pathname option(s)
```

Function: to search files or directories in the pathname.

Common options:

-type x: to search files if x=f; to search directories if x=d;

-name fordname: to search files or subdirectories specified 'fordname';

-user uname: to search all the files that belong to the user named 'uname';

-inum num: to search for files with inode number 'num';

-links num: to search for files with 'num' links;

-print: to display the searching result on the screen.

Note: If changed to the home directory, find command will start its search from there.

For example: if being in the wang directory in Figure 2.2, type in the following command:

```
$ find . -type f -name "file*" -print
./work/file1
./work/file2
./work/text/file3
$
```

As now being in the wang directory, the find command starts from there and finds three files in its subdirectories matched the option --name "file*".

The dot "." stands for the working directory, which is the searching pathname. The "*" is a wildcard standing for any number of characters in a filename (see Chapter 6).

2.4.3.7 Printing File on Printer

To print a file on a printer, use the lp or lpr command. Some UNIX operating systems have lp, such as System V, while others have lpr, such as FreeBSD or SunOS.

The syntax and function of lp command are as follows.

```
$ lp [option(s)] filename(s)
```

Function: to print a file or more specified 'filename(s)' on a printer.

Common options:

-d printer: to specify a given printer name if there is more than one printer at the local system;

-n number: to print the 'number' of copies of the file (the default is 1);

-m: to notify the sender by email when printing is done.

The syntax and function of lpr command are as follows.

```
$ lpr [option(s)] filename(s)
```

Function: to print a file or more specified 'filename(s)' on a printer.

Common options:

-p printer: to specify a given 'printer' name if there is more than one printer at the local system;

-number: to print the 'number' of copies of the file (the default is 1);

-m: to notify the sender by email when printing is done.

Note: The printer names are assigned by the system administrator. So ask the system administrator for the printer names before using the option -d (-p) printer.

Printers on UNIX operating systems are usually shared by a group of users. After the command to print a file entered, the shell prompt returns to the screen and the shell waits for another command. However, seeing the prompt does not mean that the file has been printed, but that it has been added to the printer queue to be printed in turn. This is a technique for the time-sharing systems, which is called SPOOLing (Simultaneous Peripheral Operation on-line). This technique allows several users to share a printer that was seen as an exclusively-occupied (unshared) device.

For example:

```
$ lp file2
request id is printer-101 (1 file)
$
```

or

```
$ lpr file2
request id is printer1-101 (1 file)
$
```

The above two commands show the ID of the job that the file called file2 will be printed, which can be used to check the status of the print job or cancel the job.

The following commands can be used to check the status of the print job and find out how many files or "requests" for output are ahead of this job: lpstat corresponding to the lp command; lpq to the lpr command.

For example:

```
$ lpstat -o
printer-99   wang   134866 Oct 6 10.27  on printer
printer-100  lee    86574  Oct 6 10:28
printer-101  wang   24547  Oct 6 10:30
$
```

or

```
$ lpq
printer is ready and printing
Rank     Owner   Job     Files       Total Size
active   wang    99      file1       134866 bytes
1st      lee     100     report1     86574 bytes
2nd      wang    101     file2       24547 bytes
$
```

The above two commands display their results in different ways. The lpstat lists request IDs, owners, and file sizes sequentially, and shows the request printer-99 is currently being printed on printer. The option -o is for all output requests to display. The lpq lists rank, owner, job (request ID), file (filename) and total size (of file) sequentially, and shows the request 99 is currently being printed (active).

The following commands can be used to terminate a print job: cancel terminates the lp job; lprm cancels the lpr job.

For example:

```
$ cancel printer-101
request "printer-101" cancelled
$
```

or

```
$ lprm 101
file2 dequeued
$
```

The cancel terminates the request printer-101 by specified the request ID. The lprm command also cancels the request 101, but the displayed result shows the actual filenames in the printing queue or on the printer.

2.4.4 Displaying Online Help

As different UNIX operating systems have some different features, it is helpful for the users to learn online the features of a certain UNIX operating system.

The man command can show the UNIX Reference Manual pages.

The syntax and function of man command are as follows.

```
$ man [option(s)] command(s)
$ man -k keyword(s)
```

Function: to display UNIX Reference Manual Pages for commands specified in 'command(s)' on the screen, or to display summaries of commands related to keywords in 'keyword(s)'.

Common options:

-a: to display all the pages that match the commands specified in 'command(s)';

-f: to display the pages that first matches the commands specified in 'command(s)';

-e command(s): to display summaries of commands;

-k keyword(s): to search and display summaries of commands related to keywords in 'keyword(s)';

-p pager: to display UNIX Reference Manual Pages for commands by a pager, such as more or pg;

-s n: to search section n for manual pages and display them.

Note: The two options, -a and -f, cannot be used at the same time.

UNIX Reference Manual Pages are divided into several sections, such as user commands, system calls, language library calls, devices and network interfaces, file formats, system maintenance-related commands, etc. Most end users find the pages they need in the section of user commands. Software developers mostly use the sections of library and system calls. Administrators likely refer to pages in the sections of devices and network interfaces, file formats and system maintenance-related commands. Some UNIX operating systems identify sections as numbers while some other systems differentiate them with characters, for example, SCO UNIX. And also, some UNIX systems have -s option while some other systems do not, for example, SCO UNIX and AIX. So when using -s option, be careful to follow the rules of the special UNIX operating system. However, when looking for some user commands, -f option can help find the first matched commands. And if the section option is not specified, the man command will search the first matches for the commands specified in command(s).

For example:

```
 $ man date
(The information of the date command is displayed here.)
```

The displayed information of the date command (to display the current date and time of the system) often include: name, synopsis, description, list of files, related information, errors, warnings, etc.

2.4.5 General Utility Commands

There are many general utility commands in the UNIX operating system, such as displaying the year or a month, displaying the system date, etc. Here are some of them.

2.4.5.1 Displaying Calendar

To display a calendar of the year or the month, use the cal command.

The syntax and function of cal command are as follows.

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

Function: to display a calendar on-screen.

Common options: none.

Note: The cal command can be used with or without arguments. The month argument can be a number between 1 and 12 and the year argument can be from 0 to 9999. If without argument, the cal command displays the calendar for the current month of the current year, which may be different for different versions of a UNIX operating system. If with only one argument, the argument is taken as a year and the command displays the calendar of that year. If with two arguments, it displays the calendar of that month in that year.

For example:

```
$ cal 1969
```

The calendar of 1969 is displayed.

```
$ cal 3 2003
```

The calendar of March 2003 is displayed.

2.4.5.2 Displaying the System Time

To display the current date and current time of the system, use the date command.

The syntax and function of date command are as follows.

```
$ date
```

Function: to display the system time.

Common options: none.

Note: The date command displays the current date (including year, month, date, and week) and the current time (including hour, minute and second).

For example:

```
$ date
Wed May 21 11:20:20 CST 2008
$
```

2.4.5.3 Creating the Aliases for Commands

To create the aliases for commands, use the alias command.

The syntax and function of alias command are as follows.

```
A alias [name [=string]]
B alias [name [string]]
```

Function: to create the alias 'name' for the command 'string'.

Common options: none.

Note: In the syntax of the command, A is the shell prompt for one of Bourne, Korn, and Bash shells; B is the shell prompt for C shell. The C shell allows you to create aliases from the command line while the Bourne, Korn, and Bash shells do not. Without any argument, the alias command can list the aliases for commands by default in the system.

For example:

In Bourne, Korn, or Bash shell, type the command. And the aliases are shown as follows:

```
$ alias
dir='ls -la \!*'
rename='mv \!*'
more= 'pg'
$
```

In C shell, type the command. The following is displayed:

```
% alias
dir 'ls -la \!*'
rename 'mv \!*'
more 'pg'
%
```

Opposite to the alias command, the unalias command is used to remove one or more aliases from the alias list. The -a option can be used to remove all aliases from the alias list.

For example:

```
$ unalias -a
$ alias
$
```

Command aliases can be placed in shell setup files or configuration files (see Section 2.6) , such as the .profile file (System V), the .1ogin file (BSD), the .bashrc file (Bash) and the .cshrc file (C shell). The .profile or .login file executes when a user logging in, and the .cshrc or .bashrc file executes every time the user starting a C or Bash shell. Table 2.2 lists some examples of aliases to put in one of these files. If set in the environment, these aliases allow to use the names dir, rename, and more as commands, substituting them for the actual commands given in quotes. The \!* string is substituted by the actual parameter passed to the dir command. For example, when the dir command is used, the shell actually executes the 1s -la command.

Table 2.2 Examples of aliases for different shells

Bourne, Korn and Bash Shells	C Shell
alias dir=‘ls -la \!*’	alias dir ‘ls -la \!*’
alias rename=‘mv \!*’	alias rename ‘mv \!*’
alias more=‘pg’	alias more ‘pg’

2.4.6 Summary for Useful Common Commands

In Section 2.4, some common useful commands have been introduced. For readers, it is helpful to give a brief summary of them. Table 2.3 lists all the commands introduced in Section 2.4 with their brief function descriptions.

Table 2.3 Summary of the commands in Section 2.4

Command	Function
alias	To create aliased for commands
cal	To display a calendar on-screen
cat	To join and display a file or more files in a terminal or a terminal window in sequence
cd	To change the working directory
cp	To copy files or directories
date	To display the system time
find	To search files or directories
less	To display the files on the screen, one screen at a time
lp or lpr	To print a file or more on a printer
ls	To display and list the names of the files and directories in the specified directories or the names of the files and directories in the working directory
man	To display UNIX Reference Manual Pages for commands
mkdir	To create a directory or more in the working directory
more	To display files on the screen, one screen at a time
mv	To rename a file; or to move it into the destination directory
pwd	To show the working directory on screen
rm	To delete a file or more
rmdir	To delete an empty specified directory or more

2.5 UNIX Window Systems

As mentioned in Section 2.3, although UNIX operating systems usually have a CUI as their main user interfaces, they can process software based on the X Window System (Project Athena, MIT) that provides a GUI interface (Sarwar et al 2006). X Window System, the most popular UNIX window system, is similar as Microsoft Windows. Thus users can get used to it quickly when using it. Moreover, most of the modern UNIX operating systems have an X-based GUI.

This section introduces X Window System, which is called X for short. This introduction should also help in window systems other than X.

In X, the appearance of windows, the way menus and icons work, as well as other features, are controlled by a program called the window manager. There are many different window managers; some have many features while others are simple and have just basic features. X may also have an optional desktop environment that provides even more features, such as support for drag-and-drop.

There are two popular desktop environments, GNOME with the Sawfish window manager and KDE with the kwm window manager.

2.5.1 Starting X

There are several possible situations when a computer that has X in its UNIX operating system is powered on, which are dependent on the version of that UNIX operating system.

If the UNIX operating system is set up to use the GUI interface, when the computer is powered on, there's a single window in the middle of the screen that has two prompts like "login:" and "password:". The cursor sits on the right of the "login:" line. To log in, type the username and press Enter, then do the same for the password. The system will be booted into the desktop environment just like the one that Microsoft Windows have at the starting of the operating system. From there, X will be ready to use.

If the screen shows something like a terminal, filling the whole screen (not in a separate window) and with a standard UNIX "login:" prompt, X is not running. This situation is like logging into a terminal system, so a CUI is waiting for interacting with the user. Log in and get a shell prompt. Next, to start X, type a command as follows:

```
$ startx
```

or

```
$ xinit
```

which one of them is the effect depends on that UNIX operating system

being used. So check on the user manual or ask the system administrator. After that, X will be ready to use.

2.5.2 Working with a Mouse and Windows

As a mouse on the UNIX operating system has three buttons, usually the first button means the leftmost button for right-handed users; the second button and the third one are the middle button and the rightmost button, respectively. For a mouse with only two buttons, the middle button is replaced by pressing the other buttons at once. Under X, a mouse can be set for either left-handed users or right-handed users.

Window systems use a mouse to move a pointer across the screen. The pointer can be used to select and move parts of the screen, copy and paste text, work with menus of commands, and more, which are familiar in Microsoft Windows. They also function in the similar way in X.

Some UNIX window systems have separated the cursor to show where text will be entered from the one to move all across the screen under control of the mouse. So in this book, to avoid confusion, call the former "cursor" and the latter "pointer".

The mouse pointer on the screen looks like a large "X" letter in X. When the mouse pointer moving from the desktop to other windows or menus, the shape of the pointer changes. Most of the time, it is shaped like a big X. It may change to an hourglass shape when some program is running. When a window is resized, the pointer could shape to a cross with the arrow.

Of all the windows on the screen, only one window receives the keystrokes. This window is usually highlighted. Frequently, choosing the window is called "setting the input focus". Most window managers can be configured to set the focus in one of the following ways: to simply move the pointer inside a window; to move the mouse pointer into a window and click a mouse button; to move the mouse pointer into a window and click on the title bar of the window.

A window manager can open windows of its own. But the main use of a window manager is to manage windows opened by other programs.

In X, windows also have title bars that hold the window title and some buttons, such as minimizing-, maximizing- and closing-window buttons. Windows have menus, too. The menus can be used to control the windows, such as to minimize, maximize, or close the window, to stack windows, to remember history position of the window by the history menu, etc. Having the experience of using Microsoft Windows, it is easy to get used to these features of X.

2.5.3 Terminal Window

In X, one of the most important windows is a terminal window. A terminal window has a UNIX session inside with a shell prompt, just like a miniature terminal. It is a CUI, from where command lines that are learned in Section 2.4 can be typed in and executed. It is just like the "command prompt" window in Microsoft Windows. Several terminal windows can be running at the same time, each doing something different. To enter a UNIX command or answer a prompt in a window, set the focus the terminal window and type a command or something appropriate. Programs in other windows will keep running; if they need input from the user, they will wait just as they would be on a standard terminal.

There are several programs that make terminal windows. One of the best-known programs is xterm. Others include GNOME Terminal and konsole. All do the same basic job: they allow the user to interact with UNIX from a shell prompt. If it is entered at the prompt, a UNIX command runs just as it would be on a terminal that is not under the X Window System.

To start other X-based window programs (X clients), it can be done by entering the program's name at a shell prompt in any terminal window. But it is a better way just to start new programs all from the terminal window that was open first. The reason is that if the shell has job control (see Chapter 4), it's easy to find and control all the other programs and their windows.

Another important operation should be noticed. When using a terminal window in which a program is already running, note that resizing the window may make trouble to the program currently running in it. It is best to set the terminal window size before running a program. Also, remember that standard terminals are 80 characters wide. If editing text in a window with a width that is not 80 characters, it can cause trouble later when the file is viewed on a standard terminal.

Some programs, such as ls, need to display in a fixed-width font, where every character is the same width. If the terminal window is set to a variable-width font, the columns will not line up correctly. So it is recommended to use fixed-width fonts, such as Courier, for terminal windows.

2.5.4 Using a Mouse in Terminal Windows

As modern UNIX, operating systems usually have window systems, and terminal windows are used a lot, it is useful for readers to know how to use a mouse in terminal windows.

In this section, it is about how to use the mouse to copy and paste text within an xterm window or between xterm windows.

When you move your mouse inside an xterm window, the pointer changes to an "I-beam" shape. There is also a block cursor that is the window's

insertion point, where text goes when typing on the keyboard. The block cursor does not follow the mouse because it is to show where text will be entered, which has been mentioned in Section 2.5.2.

The following lists two ways to select the text to be copied.

Select text for copying by using the I-beam pointer. Point to the first character of a command line (not the prompt) and click the first mouse button. Next, move the pointer to the end of the text to select and click the third button. Text between the two clicks should be highlighted. At this time, undo the selection by clicking the first button anywhere in the window if the wrong text has been selected.

Another way to select text is to point to the first character to copy with the mouse, then to hold down the first mouse button while dragging the pointer across the text. Release the mouse button at the end of the text to select. Then the whole area of text should be highlighted.

In an original xterm window program, there's no menu with a "Copy" command on it. The highlighted text is also automatically copied.

To paste the copied text, click the middle mouse button anywhere in the window with the block cursor at a shell prompt. Then the selected text will be inserted into the window at the block cursor. With a two-button mouse, press both buttons instead.

Some programs may not function in the way of xterm. They may just highlight text, but the text actually is not copied unless using the "copy" commands on the program's menu.

Another method for copying text between windows is to use a text editor to put the copied text into a temporary file. Then open the temporary file in a new window where to paste the text. Then copy it from that new window.

2.6 Shell Setup Files

As mentioned in Section 2.4.2, there may be some hidden files in the home directory, such as .cshrc and .profile. If not, there will probably be files named .login, .bashrc, .bash_profile, or .bash_login. These files are shell setup files or configuration files. Shell setup files contain commands that are automatically executed when a new shell starts – especially when a user logs in.

Usually the system administrator has set for a user the properties and appearance of the terminal window itself and the environment within which the commands are interpreted. The environmental settings are controlled by environment variables (the detail will be introduced in Chapter 8), which are set to their defaults when the user logs in. The environment controls which shell the user will use when typing commands. There are several kinds of popular shells: the Bourne shell (abbreviated sh), the Bourne Again shell (abbreviated bash), the Korn shell (abbreviated ksh), and the C shell (abbreviated csh). Each of them has its own shell setup files.

To find the shell that is running by default, type echo $SHELL at the shell prompt and then press Enter. Then the system replies by showing on the screen the path to the default shell.

In one of sh, bash and ksh, to view a list of the default environment variable settings, type set at the shell prompt and then press Enter. The settings of the sh, bash, or ksh environment variables will appear on the screen.

In csh, to see a list of the default environment variable settings, type setenv or printenv at the shell prompt and then press Enter. The setting of the csh environment variables will be displayed on the screen.

It is easy to change these environment variable settings for the duration of one session or for every subsequent session. For a beginner, it is not wise to change several of the environment variables.

In one of sh, bash and ksh, to set an environment variable for the current session, type variable=setting and then press Enter. Variable is a valid environment variable, and setting is a valid setting for that environment variable. For example, LESS=‘eMq’, which means which options of the less program are set every time it is used.

For example: Go to the home directory, and use less to display the file of .profile, which may look like follows:

```
PATH='/bin: /users/bin: /users/project/bin:'
LESS='eMq'
export PATH LESS
/users/project/wang
date
```

We will have the detailed explanation of the export command in Chapter 9.

In csh, to set an environment variable for the current session, type setenv variable setting and then press Enter. Variable is a valid environment variable, and setting is a valid setting for that environment variable. For example, setenv LESS ‘eMq’, which also means which options of the less program are set every time it is used.

For example: At the home directory, use less to display the file of .login, which may have the following information.

```
set path = (/bin /users/bin /users/project/bin .)
setenv LESS 'eMq'
/users/project/wang
date
```

To set an environment variable for all subsequent sessions, the shell setup files are needed to edit by using a text editor. A list of the names of the setup files in the home directory for UNIX shells are shown in Table 2.4; the tilde (∼) is a shorthand way of representing the home directory on most modern UNIX operating systems.

Table 2.4 Shell setup files

UNIX Shell (the abbreviated)	Names of the Setup Files
Bourne (sh), Korn (ksh)	/etc/.profile*, ∼/.profile
Korn (ksh)	∼ /.kshrc
Bash (bash)	∼ /.bashrc, ∼/.bash_profile
C (csh)	∼ /.login, ∼/.cshrc

Note: Do not try to edit the setup files. If it is necessary to change them, always make copies of the default setup files with the cp command before attempting to edit them. For example, to make a backup copy of the .cshrc file, type cp .cshrc .cshrc_b_2008Mar04 and then press Enter. After this command executes, there will be two identical files, one named .cshrc and the other named .cshrc_b_2008Mar04. In this way, once a mistake is made while changing .cshrc, it can be reverted to the .cshrc_bak_2008Mar04 version by using the mv command or doing operations as follows.

Quit from the text editor and type the following commands.

```
$ rm .cshrc
$ cp .cshrc_b_2008Mar04 .cshrc
```

The old .cshrc file has now been reinstalled.

If some changes are made in the setup files, these changes in the environment variables will take effect at the next time of logging on and the subsequent sessions. See Chapter 8 for a further description of how to change the environment variables in the appropriate setup file for a shell.

2.7 Summary

A typical UNIX operating system consists of several layers of software components. Close to the hardware is the UNIX kernel. The UNIX kernel is the core of the system, which provides several functions: process management, memory management, file system management, and I/O system management. The UNIX operating system provide system calls, which user processes can use to invoke the execution of kernel code. POSIX has about 100 system calls, which determine most of what the operating system has to do.

Above the UNIX kernel are the standard libraries and language libraries. The POSIX standard specifies the syntax and semantics of file and directory manipulation commands, filters, and program development tools. Language libraries are a set of functions available to programmers for use with the software that they develop.

At the top of the system are the UNIX shell and the applications. The UNIX shell is a command line interface, or CUI. When a user types a command, the shell interprets and executes the command. The applications contain all the ones that are available for users. A user can use any command or application that is available on the system and interplay with the UNIX

operating system through the interface known as AUI.

Within a text-based interface of UNIX, UNIX commands can be typed in and executed. Several classes of commands are discussed in this chapter. One group are directory operation commands, including: pwd that is used to find which the working directory is; cd that is used to change the working directory to any directory; mkdir that is used to create a new directory under the working directory; rmdir that is used to delete a directory; and ls that is used to list the files and subdirectories contained in a directory. All the definitions of specific directories are important, such as: home directory, working directory (or current directory), root directory, absolute pathname, and relative pathname.

The second group is file operation commands, including: cp that is used to copy a file; mv that is used to rename a file or move a file from one directory to another; rm that is used to delete a file; more or less that is used to look inside a file; cat that is used to display a file or several files in sequence; find that is used to find files; lp or lpr that is used to print a file on a printer; and man that is used to show the UNIX Reference Manual pages.

The third group is for general utility, such as: cal that is used to display a calendar of the year or the month; date that is used to display the current date and current time of the system; and alias that is used to create the aliases for commands.

Most versions of UNIX also work with window systems. They allow each user to have a single screen with multiple windows. A typical GUI in UNIX is X Window System. In X, one of the most important windows is a terminal window. A terminal window has a UNIX session inside with a shell prompt, just like a miniature terminal.

Shell setup files contain commands that are automatically executed when a new shell starts – especially when a user logs in. They are usually hidden files in the home directory, such as .cshrc, .profile, .login, .bashrc, .bash_profile, or .bash_login. The shell setup files have effect on the properties and appearance of the terminal window itself and the environment within which the commands are interpreted. The environment controls which shell the user will use when he or she types commands. There are several kinds of popular shells: the Bourne shell (abbreviated sh), the Bourne Again shell (bash), the Kom shell (ksh), and the C shell (csh). Each of them has its own shell setup files.

Problems

Problem 2.1 What are the common functions of the UNIX operating system kernel?

Problem 2.2 What are the system calls used for?

Problem 2.3 What are the differences between the user mode and the

kernel mode?

Problem 2.4 If logging in a terminal system, how can you log out the UNIX system?

Problem 2.5 For operating systems, usually, there are two kinds of user interfaces. What are they?

Problem 2.6 In the UNIX file system, what is the root directory? How is it indicated as?

Problem 2.7 What are the home directory and working directory? How can you find which the working directory is? How can you change the working directory? Please give an example to explain the detailed process.

Problem 2.8 What are absolute pathnames and relative pathnames? How can you distinguish between them when you type them on the command line?

Problem 2.9 How can you create a new directory under the working directory? How can you delete a directory? Please give an example for each of the two questions to explain the detailed process.

Problem 2.10 When you use the ls -l command, pay attention to the information in each of its long rows. What are the fields of each row?

Problem 2.11 Does UNIX require a dot (.) in a filename as an extension added to a filename to identify a file as a member of a category? What rules should be followed when creating a filename in UNIX?

Problem 2.12 To put more than one file into a single directory, how can you do? To make a total copy of a directory in another directory, how can you do?

Problem 2.13 Lewis and Lee are users under /users/project directory. Lewis's home directory is /users/project/lewis. What will happen if a system administrator does the following two commands? Please explain them in detail and respectively.

```
$ cd /users/project/lewis
$ cp -r /users/project/lee/work work
```

Problem 2.14 When the cp f1 f2 command and the mv f1 f2 command are used, respectively, are their affections on the original file, f1, different? How do they affect on the original file, respectively?

Problem 2.15 How can you display a long file one screen (or one "page") at a time? Can the cat command accomplish this job?

Problem 2.16 Why will the file that you want to print be added to the printer queue after you enter the lp command? Please describe how the printer queue functions. How can you terminate a print job?

Problem 2.17 When you type the following commands in Bourne shell, what will happen?

```
$ alias
dir=‘ls -la \!*’
rename=‘mv \!*’
more=‘pg’
$
```

Problem 2.18 What are terminal windows?

Problem 2.19 What are shell setup files for? What are the names of Bourne shell setup files?

References

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Badger L, Sterne DF, Sherman DL et al (1995) Practical domain and type enforcement for UNIX. SP'95: The 1995 IEEE Symposium on Security and Privacy, Oakland, CA, 8 – 10 May 1995, pp 66 – 77

Berkman J (1995) A user-friendly menu interface for UNIX. The 23rd Annual ACM SIGUCCS Conference on User Services: Winning the Networking Game, St. Louis, Missouri, US. ACM, 1995, pp 51 – 54

Braams J (1995) Batch class process scheduler for UNIX SVR4. SIGMETERICS'95/PERFORMANCE'95: The 1995 ACM SIGMETERICS Joint International Conference on Measurement and Modeling of Computer Systems, Ottawa, Ontario, Canada. ACM, pp 301 – 302

Carson SD, Setia S (1992) Analysis of the periodic update write policy for disk cache. IEEE Software Eng 18(1): 44 – 54

Heindel LE, Kasten VA (1995) Real-time UNIX application filestores. RTAS'95: First IEEE Real-time Technology and Applications Symposium, Chicago, Illinois, 15 – 17 May 1995, pp 44 – 45

Isaak J (1990) The history of POSIX: a study of the standards process. Computer 23(7): 89 – 92

Isaak J, Lohnson L (1998) POSIX/UNIX standards: Foundation for 21st century growth. IEEE Micro 18(4): 88, 87

Jespersen H (1995) POSIX retrospective. ACM, StandardView 3(1): 2 – 10

McKusick MK (1999) Twenty years of Berkeley Unix: From AT&T-owned to freely redistributable. LINUXjunkies.org. http://www.linuxjunkies.org/articles/kirkmck.pdf. Accessed 20 Aug 2010

McKusick MK, Neville-Neil GV (2005) The design and implementation of FreeBSD operating system. Addison-Wesley, Boston

Mohay G, Zellers J (1997) Kernel and shell based applications integrity assurance. ACSAC'97: The IEEE 13th Annual Computer Security Applications Conference, San Diego, CA, 8 – 12, December 1997, pp 34 – 43

Nelson BL, Keezer WS, Schuppe TF (1996) A hybrid simulation-queuing module for modeling UNIX I/O in performance analysis. WSC'96: The 1996 IEEE Winter Simulation Conference, 8 – 11 December 1996, pp 1238 – 1246

Nguyen HQ, Bac C, Bernard G (1997) Integrating QoS management in a microkernel based UNIX operating system. The 23rd IEEE EUROMICRO Conference '97 new Frontiers of Information Technology, Budapest, Hungary, 1 – 4 September 1997, pp 371 – 378

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading, Massachusetts

Rashid R, Tevanian A, Michael JR et al (1988) Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. IEEE Comput 37(8): 896 – 908

Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17 (7): 365 – 375

Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: the textbook, 2nd edn. China Machine Press, Beijing

Spinellis D (2008) A tale of four kernels. ICSE'08: The 30th International Conference on Software Engneering, Leipzig, Germany, 10 – 18 May 2008, pp 381 – 390

Stallings W (1998) Operating systems: Internals and design principles, 3rd edn. Prentice Hall, Upper Saddle River, New Sersey

3 Text Editors

As the UNIX operating system is a text-driven operating system, it is necessary to learn how to use text editors, which are used to create and edit a plain-text file, such as programs, scripts, email messages, and so on (Bach 2006; Baecker 1986; Douglas et al 1983; Makatani et al 1986; Pelaprat et al 2002; Quarterman et al 1985; Ritchie et al 1974; Rosson 1985; Sarwar et al 2006; Stallings 1998; Walker et al 1988). We will introduce three popular text editors in this chapter, which are pico, emacs, and vi editors, in order that one of them is available in a certain version of UNIX.

3.1 Difference Between Text Editors and Word Processors

UNIX operating systems use plain-text files in many places: redirected input and output of UNIX programs (see Chapter 7), shell setup files (see Chapter 2), shell scripts (see Chapters 9 and 10), system configuration, and more. Here, "plain text" (Sarwar et al 2006) means a file with only letters, numbers, and punctuation characters in it. They let users add, change, and rearrange text easily. Therefore, users of UNIX must be familiar with one of text editors to enter a file. In addition, they must also be familiar with how to edit existing files efficiently, that is, to change their contents or otherwise modify them in some way. Text editors also allow users simply to view a file's contents, similar to the more and less commands (see Chapter 2).

Contrasted with text editors are word processors. When a word processor is used, although the screen may look as if the file is only plain text, the file may also have hidden codes (non-text characters) in the file, which are usually used for the document format or how to display it. For word processing, such as making documents, envelope, and so on, most UNIX operating systems also support word processors, including WordPerfect and StarOffice, which are compatible, more or less, with Microsoft word processors.

In this chapter, we will mainly discuss the text editors. Two common UNIX text editors are vi and emacs. Almost all UNIX operating systems

have vi, but emacs is also widely available. Pico is a simple editor that has been added to many UNIX operating systems. Although pico is much less powerful than emacs or vi, it's also a lot easier to learn.

The editors are usually full-screen-display editors. That is, in the terminal screen to display the file, a portion of the file, which fills most or all of the screen display, is shown. Any part of the text can be pointed to by moving the cursor. The displayed content of the file is usually held in a temporary storage area in primary memory called the editor buffer. If the file is larger than one screen, the buffer contents change as the cursor is moved through the file. The difference between a file to edit and a buffer is crucial. A file is stored on disk as a sequence of data. When editing that file, a copy that the editor creates is being edited and is in the editor buffer. Before saving the buffer, any change is just the change to the contents of the buffer. Once the file is saved, a new sequence of data is written to the disk. So before it is saved, any change has not been really realized. So it is a good idea to save the file in time.

It also should be mentioned that the text editors of UNIX operating systems are based on keystroke commands, whether they are a single keystroke or combinations of keys pressed simultaneously or sequentially, rather than mouse and GUI. Because the standard input device in UNIX is the keyboard, it is critical for users of UNIX to learn and master the correct syntax of keystroke commands, which is rigid to follow. It needs some time to be familiar with the rules. But once the user has become accustomed to it, this method of input is as efficient as the mouse-and-GUI input. Generally, the simpler text editor is easier to learn, such as pico. More powerful editors such as vi and emacs are capable of handling more complex editing tasks. This also means it takes more time to practice how they are implemented in order to master that knowledge.

3.2 Introduction of Pico Editor

The pico editor (Sarwar et al 2006), from the University of Washington, is easy to use. It allows doing simple editing on small text files efficiently. Its user interface is friendly and simple compared to the other editors later in this chapter. The pico editor is distributed free with the pine e-mail system. Like pine, pico is still evolving. Start pico by typing its name; the argument is the filename of the file to create or edit. If the pico program cannot be accessed by typing pico at the shell prompt, ask the system administrator where the shell can locate it. Then set the search path to include that location (see Chapter 8).

3.2.1 Start pico, Save File, Exit pico

For small and simple text files, such as short e-mail messages or shell scripts, the pico editor is easy and quick to learn, and also effective. As apparent in Figure 3.1, it has a straightforward screen display that has two main parts: a text area, where the user can enter and change text by typing, and a keystroke command area at the bottom of the screen, which shows valid keystroke commands. Like other UNIX commands, pico program can be started from a command line. The method to start pico from the command line is as follows.

```
$ pico [option(s)] [file]
```

Function: to start pico to create a new text file or edit an existing text file.

Common options:

-h: to list valid command line options;

-m: to enable mouse functionality under the X Window System;

-o dir: to set the operating directory only if files within dir are accessible;

-w: not to break lines at the right margin, but only if there is Enter at the right end.

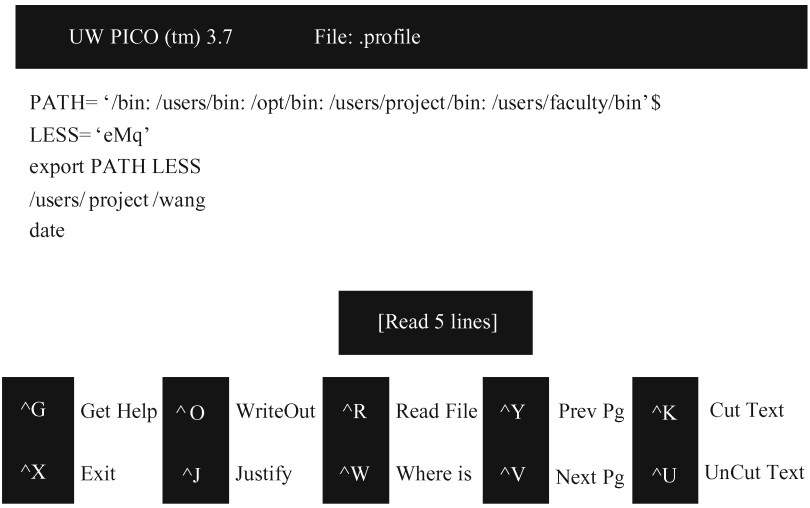

Fig. 3.1 Pico screen with an edited file.

Note: Without an argument, pico begins a new buffer, which will be a new file without filename yet at that moment. The filename must be given before exiting pico by saving the file with a name. If a line is longer than the right margin, pico -w marks the right end with a currency sign ($). When the cursor is moved over the currency sign, the next 80 characters of that one line are displayed.

For example:

```
$ pico -w .profile
```

The terminal screen fills with a copy of the file of .profile, shown in Figure 3.1.

The bottom two rows of the window list some pico commands. Note that, to the left of each keystroke command listed at the bottom of the screen, there are two characters (e.g., CTRL-O next to WriteOut and CTRL-G next to Get Help). CTRL-O means to hold down the first CTRL key on the keyboard while holding down the next character key O to execute the command to save the file (i.e., WriteOut a file).

To save at any point, hold down the CTRL and O keys at the same time. A prompt appears near the bottom of the screen asking to type in a filename to which the current text body will be saved. When you want to save the text that you have entered, use CTRL-O to confirm the filename that the text will be saved to. Then press Enter, and the current text will be saved to the filename that is specified. Note: Save the file frequently so that the work that has already been done will not be lost if the system crashes. Pico can save interrupted work in a file named pico.save or filename.save, where filename is the name of the file edited. But it is wise and safe to save by hand when the file is in a good state.

To exit pica, hold down the CTRL and X keys at the same time. If any changes have been made in the text since the last time the file is saved, type Y to save the file.

3.2.2 Create a New File with Pico

Let us take a tour through pico by creating a new file with pico. In this chapter, some operation examples will be given in order to show clearly and directly how to do some operations in different text editors, just like the following one. In this example, try to make a new file named example1.

Operation example 3.1 pico example1, showing steps of how to create a new file example1 and save it with the pico text editor. Following are steps that can be learned to practice step by step.

1) At the shell prompt, type the following command, and press Enter at its end.

```
$ pico example1
```

2) The pico screen display appears similar to Figure 3.1, but the middle of the screen is blank and ready for entering text.
3) On the blank text area, type some lines of text. Make some lines short and press Enter before the line gets to the right margin. Make others long; watch how pico wraps long lines.

If using a window system and having another terminal window open with some text in it, use the mouse to copy text from another window and paste it into the pico window by using the method of Section 2.5.4. To get a lot of text quickly, paste the same text more than once.

4) Hold down the CTRL and O keys on the keyboard at the same time to write out the file to the disk. Pico prompts to enter a filename. If just to save the file with the same name it had as started, press Enter. If to change the filename, edit the filename in place by adding or deleting characters in the name before pressing Enter to save it. Or backspace over the whole name and type a new one after the prompt (see Figure 3.2). The filename can be an absolute pathname.
5) Hold down the CTRL and X keys on the keyboard at the same time, and exit from pico.

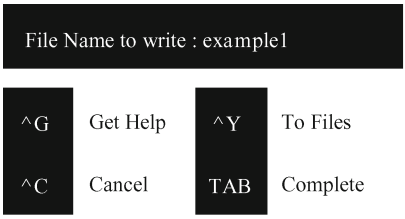

Fig. 3.2 Saving the file at the bottom of the pico screen.

3.2.3 Cursor-moving Commands in Pico

As pico works on all terminals, with or without a mouse, it will probably ignore the mouse when to move the cursor. Instead, use the keyboard to move the cursor. Practice moving around the file when editing a file with pico by using the following cursor-moving commands in Table 3.1.

Table 3.1 Cursor-moving commands in the pico

Keystroke Command	Action
CTRL-A	To move the cursor to the start of the current line
CTRL-B	To move the cursor backward a character
CTRL-E	To move the cursor to the end of the current line
CTRL-F	To move the cursor forward a character
CTRL-N	To move the cursor to the next line
CTRL-P	To move the cursor to the previous line
CTRL-V or PAGE DOWN	To move the cursor to the next page of text
CTRL-Y or PAGE UP	To move the cursor to the previous page of text

3.2.4 General Keystroke Commands in Pico

As shown at the bottom of pico screen in Figure 3.1, there are many general keystroke commands in pico, which are listed in Table 3.2.

Table 3.2 General keystroke commands in pico

Keystroke Command	Action
CTRL-C	To report the current cursor position as line n and character m
CTRL-G	To access to pico Help text
CTRL-J	To justify the selected text, to flow and fit neatly between the margins
CTRL-K	To cut the selected text
CTRL-O	To save the current text to a file
CTRL-R	To read in text from a file and paste the text at the current cursor position
CTRL-T	To check files and directories through a file browser
CTRL-U	To paste the current line of text
CTRL-V	To scroll one page down in the Help pages
CTRL-W	To search for a string of characters (where is)
CTRL-X	To exit from pico and allow to save any changes before exiting
CTRL-Y	To scroll one page up in the Help pages
CTRL-^ or CTRL-SHIFT-6	To begin to make a section of text for cutting out text

Some actions need to explain further. Here gives their operation steps, respectively.

- Cut a text: Move the cursor to the first character, then press CTRL-^ or hold down the CTRL key, and both the SHIFT and 6 keys at the same time. Then move the cursor to the last character and press CTRL-K. The pico cuts the text between the first character and the last character and remembers the text in order to paste it back later any many times anywhere until a new cut is done (or until exiting from pico.)
- Copy a text: Press CTRL-U just after pressing CTRL-K. That means, to paste the text back where cutting it, press CTRL-U to "uncut" or paste the text at current cursor position.
- Paste a text: Move the cursor to some spot where to insert the text and press CTRL-U there.
- Search for a word: Press CTRL-W ("where is" command) and a looked-for word. When pressing CTRL-W, the command list at the bottom of the display will change. The cursor sits after the word "Search:", waiting for typing in a word or characters to search for and Enter to do the search. That is shown in Figure 3.3.
- Justify one paragraph: Put the cursor somewhere in the paragraph and

press CTRL-J. Then the paragraph's lines flow and fit neatly between the margins.

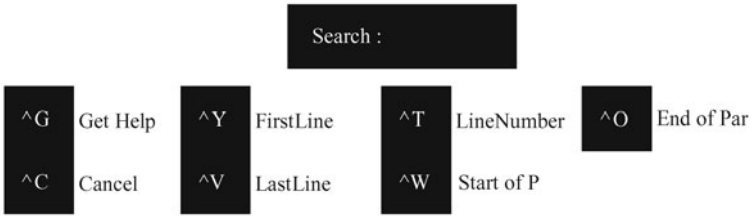

Fig. 3.3 Searching for a word at the bottom of the pico screen.

There are more functions. Check them out by pressing CTRL-G.

Operation example 3.2 pico example2. Follow the example steps and practice how to create a new file example2, use the keystroke commands to edit the file and correct mistakes, and save it as examples with the pico text editor.

1) At the shell prompt, type in the following command and press Enter at the end.

```
$ pico example2
```

2) In the text area of the pico screen, place the cursor on the first line and type:

This is a text that I have practiced to create in the pico editor.

Use the Delete, the arrow keys, CTRL-B and CTRL-F to correct any typing errors.

3) Move the cursor to the first character, then press CTRL-^. Then move the cursor to the last character and press CTRL-K. This action cuts the line of the text out of the current "buffer", or file to be edited. Press CTRL-U just after pressing CTRL-K. That means to copy not cut the text.
4) Press Enter. The cursor moves to the second line. Press CTRL-U. Another line of "This is a text that I have practiced to create in the pico editor." appears on the screen.
5) Press Enter. The cursor moves to the third line. Press CTRL-U. The third line of "This is a text that I have practiced to create in the pico editor." appears on the screen.
6) Change the characters of line 2 and 3 by using the Delete, the arrow keys, CTRL-B, CTRL-F, and so on, so that they read as shown in Figure 3.4.
7) Hold down the CTRL and O keys on the keyboard at the same time to write out the file. Pico prompts to enter a filename. At the prompt of File Name to Write:, enter a new filename example3 and then press Enter.
8) Hold down the CTRL and X keys on the keyboard at the same time, and exit from pico.

There are some more practices at the end of this chapter.

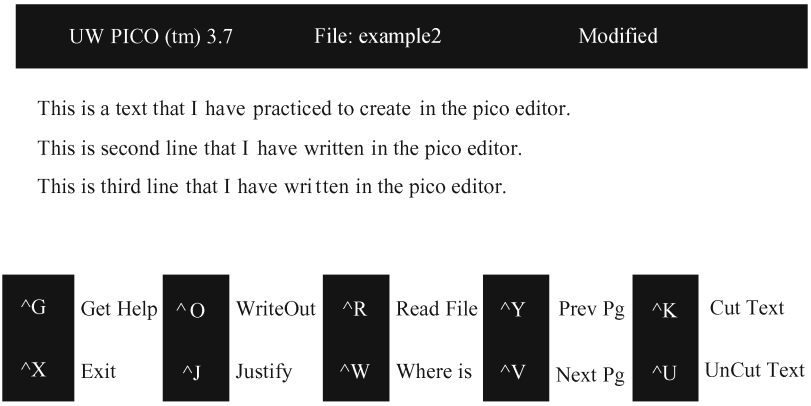

Fig. 3.4 Editing the file of example2 in pico.

3.3 The vi Editor and Modes

Around 1977, allegedly, with ADM-3a terminals that had screen-addressable cursors, William N. Joy could finally write the first version of vi editor at the University California, Berkeley (McKusick 1999; Ward 2003; Wilder1997). By the mid-1978, along with 2BSD, the vi editors were distributed.

The vi (visual) UNIX text editor is more complex than pico, but it has the ability to work on much larger files. It is more flexible and has more features than pico. Of course, it also takes more time and practices to get familiar with it than pico.

3.3.1 Three Modes of the vi and Switch Between Them

There are three general editing operation modes in the vi text editor: Command Mode, Insert Mode and Last Line Mode.

In Command Mode, the editor can enter some key sequences that are commands to accomplish certain actions, like to delete words or lines, to paste the deleted lines, etc., while in Insert Mode, the editor can input text of the edited file. The Last Line Mode allows the editor to do some actions, such as write out the file, quit the vi, switch between several opened files, or run shell commands without quitting the vi, and so on.

Figure 3.5 shows the organization of the vi text editor and how to switch from one mode to another.

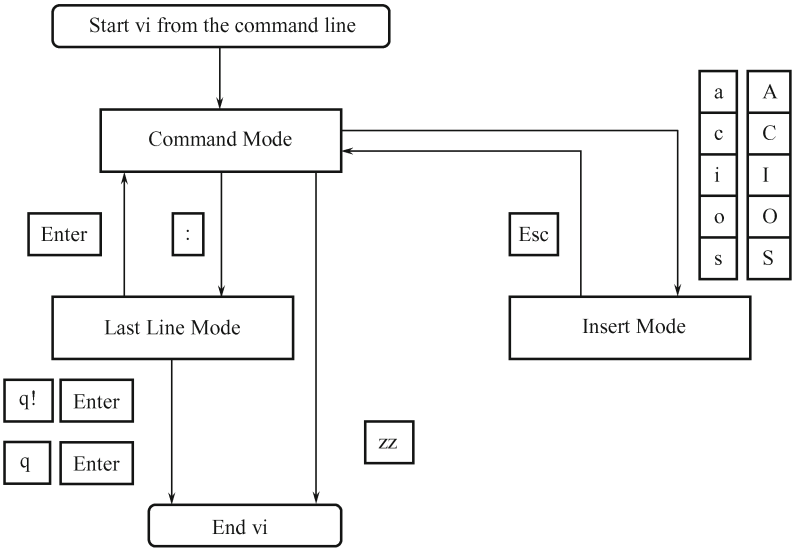

Fig. 3.5 Mode switch in vi.

To change from Command Mode to Insert Mode, type one of the following valid commands, A or a (to append text), C or c (to change text), I or i (to insert text), O or o (to open text), S or s (to substitute text), etc. The keystroke commands in vi are case-sensitive. The actions of the uppercase letter (such as A) and the lowercase letter (such as a) are slightly different, which will be explained later in this section. To change from Insert Mode to Command Mode, press the Esc key.

To change from Command Mode to Last Line Mode, type : (or SHIFT-: in some UNIX operating systems). When that is done, characters are echoed or shown on the last line on the screen. To terminate Last Line Mode, type Enter after commands.

3.3.2 Start vi, Create a File, Exit vi

Like pico, the vi text editor program can be started from a command line. The method to start vi from the command line is as follows.

```
$ vi [option(s)] [file(s)]
```

Function: to start vi to create a new text file or edit an existing text file.
Common options:
+/exp: to begin to edit at the first line in the file matching string exp;
+n: to begin to edit files(s) starting at line n.
Note: Command Mode is the first mode when starting the vi text editor.

For example:

```
$ vi shellscpt1
```

To start vi, at the shell prompt, type vi (with or without arguments) and then press Enter. The vi display appears on the screen, as shown in Figure 3.6. It is now in Command Mode. To enter Insert Mode, type A. The vi is ready to insert text on the first line of the file. After entering text, press the Esc key to return to Command Mode. From Command Mode, type : to go into Last Line Mode. Save the inserted text to a file on disk by typing w filename after the colon and pressing Enter, where filename is the identified name of the file to save the text to. To quit the editor, type q and Enter after the colon.

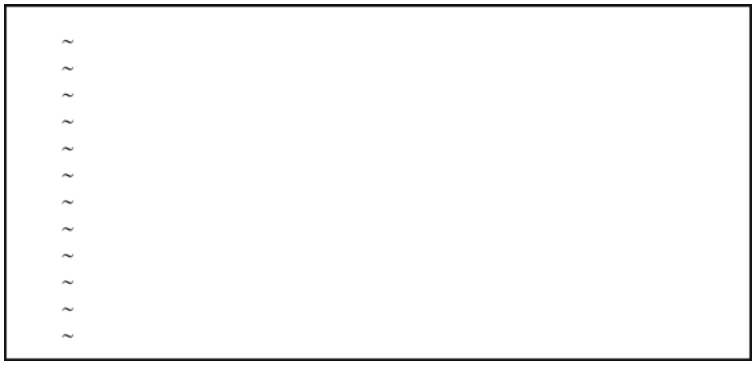

Fig. 3.6 The starting screen in the vi.

Operation example 3.3 vi shellscpt1, practicing how to create a script file of a group of UNIX commands in the vi editor, which are executed in sequence, as shown in Figure 3.7, and then to execute the script. For this example, it is assumed that the Bourne shell is the default shell, which usually is the case. If not, change shells by using the method in Chapter 8. And do not worry too much about making mistakes, and just go through the steps. The following sections will give more discussion about how to deal with the problems.

1) At the shell prompt, type the following command, and press Enter at the end.

   ```
   $ vi shellscpt1
   ```

   Then the vi screen appears on the display.
2) Type A, type pwd, and then press Enter. The first line shows on the screen as shown Figure 3.7.
3) Type ls -la and then press Enter. The second line appears on the screen.
4) Type date and then press Enter. The third line appears on the screen. At this point, the whole text has been entered. Press the Esc key to

return to Command Mode.

5) Type : to go into Last Line Mode. Then type wq and then press Enter to write out the shellscpt1 file, quit vi and return to the command line.
6) At the shell prompt, type in the following and press Enter at the end.

```
$ sh shellscpt1
```

Then the results of those commands appear on the display, one by one.

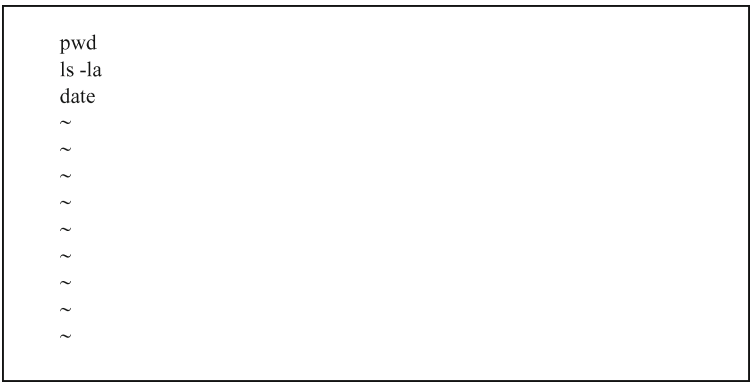

Fig. 3.7 A shell script displayed in vi screen.

3.3.3 Syntax of the vi Commands

In the vi editor, many commands can be used flexibly and variably by combining them with some numbers. The common syntax of keystrokes is

```
[number1] keystroke [number2] [unit]
```

Function: to do the action of the "keystroke" "number1" times in the "number2" of the "unit" of the text starting from the cursor. Here, "keystroke" can be one of the vi keystroke commands.

Options: (enclosed in the [] and optional)

number1: to specify how many times of actions of the "keystroke" are to be done;

number2: to specify how many "units" are affected by the action of the "keystroke";

unit: to specify a unit of the text on which the action of the "keystroke" is done.

Note that, the unit can be l (letter), w (word), etc. Some keystroke commands are not following the above syntax rigorously. Check out the help of the vi editor for using them. Table 3.3 lists some examples of this common syntax and variations.

Table 3.3 Examples of the common syntax of keystroke commands in vi

Command	Action
2dw	To delete two words starting at the cursor
d0	To delete the text from the first character of the current line to the character just before the cursor
5dd	To delete five lines starting at the current line
d3l	To delete three letters starting at the cursor to the right
d2,5	To delete lines two through five in the buffer
c3b	To change back three words
8o	To open eight new blank lines below the current line
10O	To open ten new blank lines above the current line
1G	To put the cursor on the first line of the file
9yy	To copy (yank) the next nine lines (starting with the current line) into a temporary buffer

3.4 Practicing in Insert Mode of the vi Editor

Since vi is always started in Command Mode, to enter Insert Mode from Command Mode, press one of the following keys to switch the mode, which are A, a, C, c, I, i, O, o, R, etc. Table 3.4 shows them and their actions.

Table 3.4 Important keystroke commands switching into Insert Mode

Keystroke Command	Action
a	To append text after the cursor
A	To append text after the last character of the current line
c	To begin a change operation, which means to delete the old text first and then to insert the new text
C	To change text from the cursor position to the end of the current line, which means to delete the old rest of the line first and then to insert the new text
i	To insert text before the cursor
I	To insert text at the beginning of the current line
o	To open a new blank line below the current line and put the cursor on that new line
O	To open a new blank line above the current line and put the cursor on that new line
R	To begin overwriting text on the cursor and the following to its right on the same line; an old letter is replaced by a new letter at a time; if the new text is longer than the old one, the rest will be appended to the end of the line
s	To substitute a specified number of characters starting from the cursor
S	To substitute the whole lines

The results of some actions need to explain in detail. Here gives the explanation.

- I: Insert text at the beginning of the current line. Figure 3.8 shows the situation before and after the I command pressed while Figure 3.9 displays the result of inserting int and a space before a.

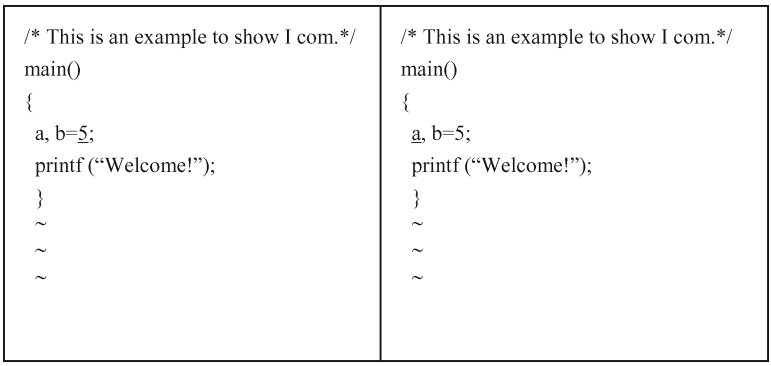

Fig. 3.8 The vi screens before (left) and after (right) typing I command.

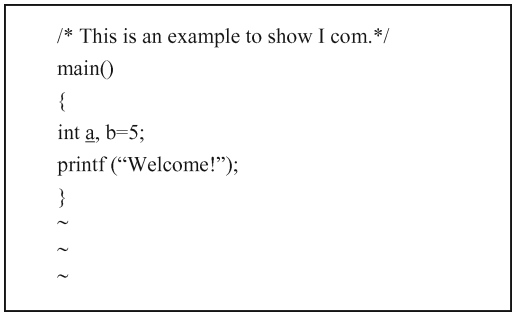

Fig. 3.9 The vi screen of inserting int and one space before a.

- O: Open a new blank line above the current line. Figure 3.10 shows the situation before and after entering O command while Figure 3.11 shows the result of inserting main().
- c: Begin a change operation. This command is usually used with the cursor-movement "unit" (see Section 3.3.3). Figure 3.12 shows the situation before and after entering cw command while Figure 3.13 shows the result of entering "an example" and pressing Esc.
- C: Change the text from the cursor position to the end of the current line. Figure 3.14 shows the situation before and after entering the C command while Figure 3.15 shows the result of entering "a program. */" and pressing the Esc key.
- r: Begin overwriting a specified number of characters starting from the

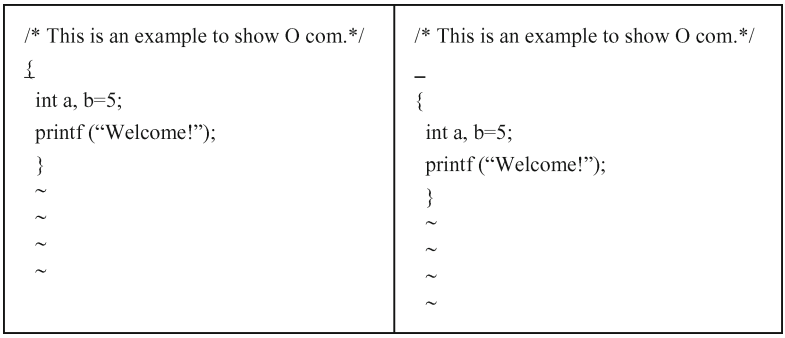

Fig. 3.10 The vi screens before (left) and after (right) typing O command.

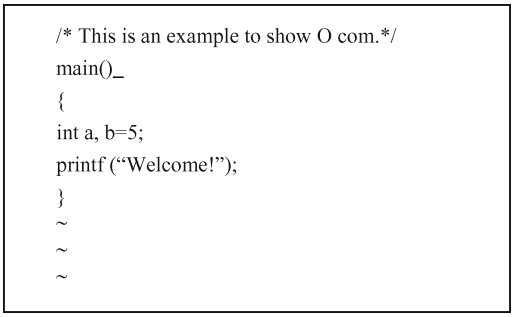

Fig. 3.11 The vi screen after inserting main().

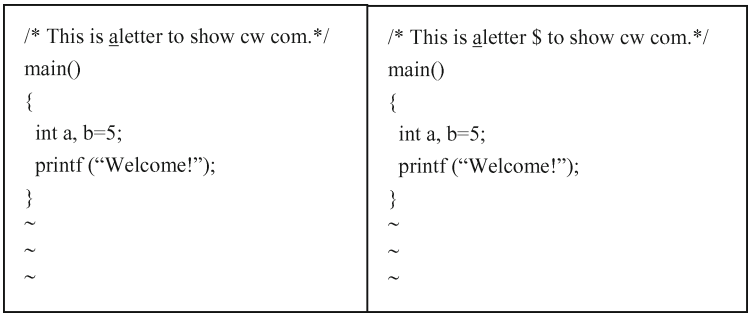

Fig. 3.12 The vi screens before (left) and after (right) typing cw command.

cursor. Figure 3.16 shows the situation before and after entering the 3rA command. In fact, r is not among the commands switching into Insert Mode. As it is similar to R in action, it is put here to compare easily with R.

- R: Begin overwriting text on the cursor and the following to its right on the same line. Figure 3.17 shows the situation before and after entering the 3RAA command and Esc. The AA replaces the aa and repeats three

```
/* This is an example_to show cw com.*/
main()
{
int a, b=5;
printf ("Welcome!");
}
~
~
~
```

Fig. 3.13 The vi screen after entering "an example and Esc".

```
/* This is an example to show C com.*/
main()
~
~
~
~
```

```
/* This is an example to show C com.*/ $
main()
~
~
~
~
```

Fig. 3.14 The vi screens before (left) and after (right) typing C command.

```
/* This is a program. */
main()
~
~
~
~
```

Fig. 3.15 The vi screen after entering "a program. */ " and Esc.

```
/* This is anexample to show r com.*/
main()
~
~
~
~
```

```
/* This is AAAxample to show r com.*/
main()
~
~
~
~
```

Fig. 3.16 The vi screens before (left) and after (right) typing 3rA command.

times, and the rest (aa) of the old text swifts right.

- s: Substitute a specified number of characters starting from the cursor.

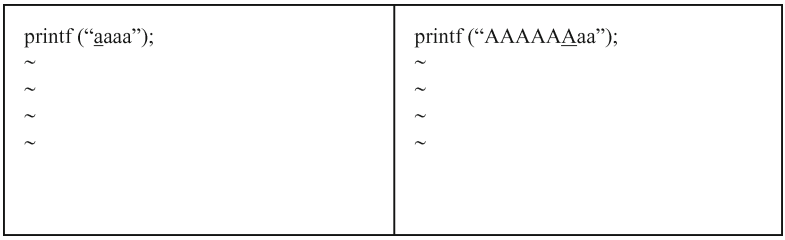

Fig. 3.17 The vi screens before (left) and after (right) typing 3RAA command.

Figure 3.18 shows the situation before and after entering the 3sA command and Esc.

The difference between commands of 3rA (in Figure 3.16) and 3sA is: the former just replaces the characters, but the latter replaces the characters and changes the mode (from Command Mode into Insert Mode). So s needs Esc to return Command Mode.

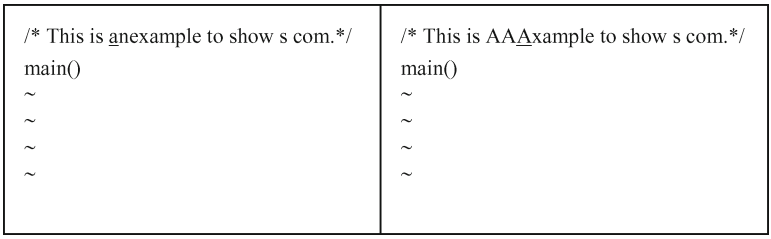

Fig. 3.18 The vi screens before (left) and after (right) typing 3sA command.

- S: Substitute the whole lines. Figure 3.19 shows the situation before and after entering the S command while Figure 3.20 is after entering “#include <stdio.h> Enter #include <stdlib.h>” and Esc. Like s command, S command causes to enter Insert Mode.

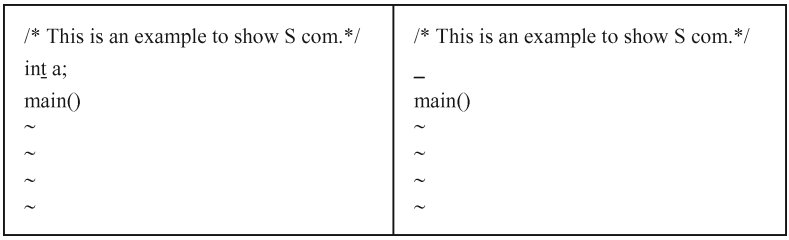

Fig. 3.19 The vi screens before (left) and after (right) typing S command.

Except inserting text, all the other operations have to be done within Command Mode or Last Line Mode, including editing the text, moving the cursor to a new position in the buffer, saving the buffer, and exiting the

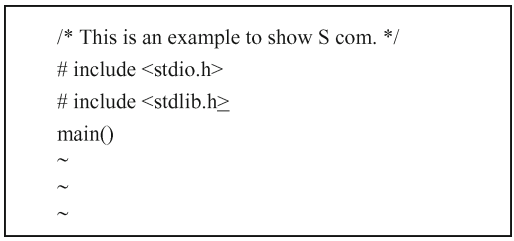

Fig. 3.20 The vi screen after entering two lines of "# include . . ." and Esc.

editor. To change from Insert Mode back to Command Mode, press the Esc key.

Operation example 3.4 vi ex1.c, practicing how to edit the file in vi.

1) At the shell prompt, type the following command and press Enter at the end.

```
$ vi ex1.c
```

   Then the vi screen appears on the display.
2) Type A, type "/*This is an example to show editing a file in vi.*/", and press Enter at the end of the line.
3) Type "# include <stdi.h>", and press Enter at the end of the line.
4) Type "# include <stdlib.h>", and press Enter at the end of the line.
5) Press Esc key.
6) Type :w, and press Enter. The screen display should be shown like Figure 3.21.
7) Use the arrow keys to position the cursor at the "." character on the second line of the file.
8) Type i and o.
9) Press Esc key.
10) Use the arrow keys to position the cursor on the third line of the file.
11) Type o, type main() and press Enter at the end of the line.
12) Press Esc key. The screen display should be shown like Figure 3.22.
13) Type :wq, and go back the shell prompt.

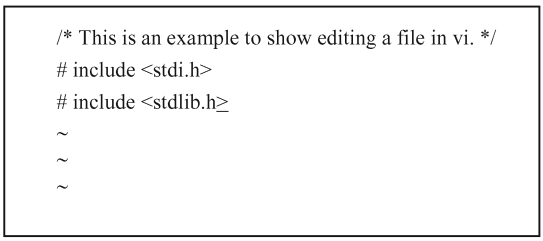

Fig. 3.21 The first one of the vi screen displays of editing the ex1.c file.

```
/* This is an example to show editing a file in vi. */
# include <stdio.h>
# include <stdlib.h>
main()
~
~
```

Fig. 3.22 The second one of the vi screen displays of editing the ex1.c file.

3.5 Practicing in Command Mode and Last Line Mode of the vi Editor

The general commands that are useful in Command Mode are shown in Table 3.5 while those executed from the status (last) line prefaced with a : character, which means Last Line Mode, are shown in Table 3.6. As known, character-at-a-time or line-at-a-time moves of the cursor can be accomplished easily with the arrow keys. There are also some more commands in vi that can be used to move the cursor more effectively, which are listed in Table 3.5 as well.

Table 3.5 Important commands in command mode

Keystroke Command	Action
d	To delete words, lines, etc., according to "number1", "number2" and "unit" (see Section 3.3.3)
dd	To delete the whole current line
D	To delete the text from the cursor position to the end of the current line
x	To delete the character at the cursor position
u	To undo the last edit
r	To replace the character at the current cursor location with what is typed next
ZZ	To quit vi, with saving the file only if changes were made since the last save
G	To move the cursor to the last line of the file
1G	To move the cursor to the first line of the file
0	To move (zero) the cursor to the first character of the current line
$	To move the cursor to the last character of the current line
CTRL-G	To report the position of the cursor in terms of line number and column number
w	To move the cursor forward one word at a time
b	To move the cursor backward one word at a time

Table 3.6 Important commands for Last Line Mode

Keystroke Command	Action
:r filename	To read and insert the contents of the file called filename at the current cursor position
:wq	To save the text in the buffer and quit vi
:q!	To quit vi without saving the text in the buffer
:w filename	To save the text in the current buffer to the file called filename
:w >> filename	To append the text in the current buffer to the end of the file called filename
:w! filename	To overwrite the file called filename with the current text

Note: When delete is done, the deleted text is stored in some buffer (see Section 3.6), which can be pasted somewhere later by using p or P commands (see Table 3.8).

Note: When pressing : in Command Mode, it enters into Last Line Mode, where the cursor sits on the last line of the vi screen and waits for further entering. Terminate a command of Last Line Mode by pressing Delete key.

Another benefit of Last Line Mode is to execute a shell command without quitting vi and then restarting it. Just do execute a shell command by typing the command after : with !, then pressing the Enter key. For example, to execute pwd command, just do as follows,

```
:!pwd
```

The pathname of the current directory will be displayed. And do this,

```
:!ls
```

The names of all the files in the current directory will appear on the screen as well. After executing a shell command, vi returns to Command Mode.

During editing the text of a file, some other commands of Last Line Mode can also help do a lot more, such as to move the cursor, to delete lines, to substitute a word, to switch between files, etc. Table 3.7 gives some of these

Table 3.7 More of important commands for Last Line Mode

Keystroke Command	Action
:3	To move the cursor to the first character of line 3
:$	To move the cursor to the first character of the final line
:+3	To move the cursor forward three lines
:10,100d	To delete the text from line 10 to line 100
:1,20s/red/green/g	To substitute the word red for each word green on lines 1–20
:e filename	To start to edit the file named filename, which is different from the file in the current buffer, so with warning before deleting the changed text in the buffer
:e #	To return to edit the last edited file

commands.

Note: These commands need pressing Enter at the end of commands to execute them. Terminate them by pressing Delete key. In the :1,20s/red/green/g command in Table 3.7, g means for global. In the :e # command, # means the last edited file while the currently edited file is remembered as % by vi.

Operation example 3.5 vi ex1.c, practicing how to edit the file in vi further.

1) At the shell prompt, type in the following command and press the Enter key at the end. In the rest of the content in this book, without extra mention, each command typed has to be followed by pressing Enter at the end to make it execute.

```
$ vi ex1.c
```

   Then the vi screen appears on the display.
2) Type G to move the cursor to the last line of the file.
3) Press CTRL-G at the same time to see the position of the cursor. The vi reports information, such as, "ex1.c" line4 of 4 –100%– col 1, on the last line of the screen display. The report is about the editing buffer (see Section 3.6), listing the current line number, the total number of lines in the buffer, the percentage of the buffer that this line represents, and the current column position of the cursor.
4) Type o to open a new line below the forth line of the file.
5) Type "print ("%d\n", a=c*2);" and press Esc key.
6) Type 0 to move the cursor to the first character of the current line.
7) Type $ to move the cursor to the last character of the current line.
8) Type O to open a new line above the fifth line of the file.
9) Type { and press Esc key.
10) Type o to open a new line below the fifth line of the file.
11) Type "int a, b= 110;" and press Esc key.
12) Move the cursor to the first 1 on the current line by using the arrow keys.
13) Type x to delete 1.
14) Move the cursor to the c on the seventh line by using the arrow keys.
15) Type r and b to replace the c with the b.
16) Type o to open a new line below the seventh line of the file.
17) Type} and press Esc key. The screen display should be shown like Figure 3.23.
18) Type :wq, and go back the shell prompt.

```
/* This is an example to show editing a file in vi. */
# include <stdio.h>
# include <stdlib.h>
main()
{
int a, b=10;
printf ("%d\n", a=b*2);
}
~
~
```

Fig. 3.23 The vi screen displays of editing the ex1.c file further.

3.6 Using Buffers of the vi Editor

As mentioned in Section 3.1, the displayed content of an edited file is usually stored in a temporary storage area in primary memory called the editor buffer. For vi, except the editor buffer, there are also several buffers working during editing files. In vi, the main buffer that works as the editing buffer is the main temporary storage area for the text to create or to modify from some previous permanently-stored file on disk. The general purpose buffer is somewhere that the most recent cut/copied text (or "ripped-out") is held. indexed buffers store more than one temporary string of text. It is helpful to know how to use these.

In vi, having been deleted or yanked, the text is stored in one of the buffers. Therefore, copying and pasting can be done with the vi commands y (yank) and p (put) while cutting and pasting can be accomplished with commands d (delete) and p (put).

Indexed buffers are named buffers that can be identified as a – z buffers or 1 – 9 buffers. It is easy to use indexed buffers to store different texts in different buffers, and to move and paste them in different places in a file. And even better, when switching the edited files, the contents in these buffers are not destroyed, so they can be pasted in different files. To access these buffers, double quotation mark (") is needed (see Table 3.8).

Table 3.8 Examples of yank and paste commands

Keystroke Command	Action
y3w	To yank three words to right, starting at the current cursor position
yy	To yank the whole current line
Y	To yank the text from the cursor position to the end of the current line
"ayy	To yank the whole current line and put it into the buffer named a

Continued

Keystroke Command	Action
p	To insert (paste) the yanked or deleted line(s) before the current line
P	To insert (paste) the yanked or deleted line(s) after the current line
"ap	To insert (paste) the yanked or deleted line(s) in the buffer named a before the current line

In default, the pasted text is from the non-named buffer that is the general purpose buffer, which holds the most recent cut/copied text. Pasting commands have two, p and P, as shown in Table 3.8.

Operation example 3.6 vi ex2.c ex3.c, practicing how to paste some lines of one file into another file in vi. It is assumed that the ex3.c file has already had several lines of text (similiar as Figure 3.22), which can be done by creating the file of ex3.c before doing this operation example. Following this example step by step, learn how to use the named buffers.

1) At the shell prompt, type the following command.

```
$ vi ex2.c ex3.c
```

Then the vi screen appears on the display.

2) Type A, type several lines, and press Enter at the end of each line. It is shown on the screen as Figure 3.24.

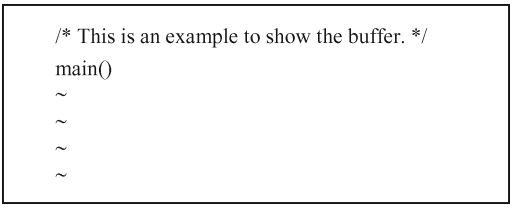

Fig. 3.24 The vi screen showing the context of the ex2.c file.

3) Type : to go into Last Line Mode. Then type w and then press Enter to write out the ex2.c file without quitting vi.
4) Type ": e ex3.c" and Enter. The whole command is as follows:

```
:e ex3.c
```

Instead of the file of ex2.c, the file of ex3.c is displayed on the screen, assumed that several lines has already been saved in the file, as shown in Figure 3.25.

5) Move the cursor to the first character of the second line. And type "ay2+, then the text from line 2 to line 4 is in buffer a until quitting vi.
6) Type : and e # Enter. The whole command is:

```
:e #
```

```
/* This is an example to show the buffer. */
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
main()
~
~
```

Fig. 3.25 The vi screen showing the context of the ex3.c file.

This time, the file of ex2.c is displayed on the screen again.

7) Move the cursor to the first character of the second line. And type "ap, then the text in buffer a is inserted before the current line, and the cursor is at the first character of the first inserted line, as shown in Figure 3.26.
8) Type : to go into Last Line Mode. Then type wq and then press Enter to write out the ex2.c file, quit vi and return to the shell command line.

```
/* This is an example to show the buffer. */
# include <stdio.h>
# include <stdlib.h>
# include <math.h>
main()
~
~
```

Fig. 3.26 The vi screen showing the context of the ex2.c file after pasted the text in buffer a.

3.7 The vi Environment Setting

The environment options of the vi editor can be customized, which influence the behavior of the vi. These options include the maximum line length, the cursor's automatically wrapping to the next line, the line number display, and the vi mode display. Almost every option has a full name and an abbreviated name.

There are two ways to set the vi environment options: one is within the vi by using the Last Line Mode command; the other is to set them in the file of .exrc. In the former case, the options are set for that session only until quitting the vi. In other words, after quitting the vi editor, the vi environment will be recovered to the original options before this setting. In the latter case, the options are customized to settings permanently.

To set these environment options, the set command should be used in Last Line Mode. Its syntax is

```
:set option(s)
```

Function: to set vi to function as the options.

Common options:

Autoindent (ai): to align the new line with the previous line at the line beginning.

ignorecase (ic): to ignore the case of a letter during the search process (with a / or the ? command).

number (nu): to display line numbers when a file being edited.

scroll: to set the number of lines to scroll when the CTRL-U command is used to scroll the vi screen up.

showmode (smd): to display the current vi mode in the bottom right corner of the screen.

wrapmargin (wm): to set the wrap margin in terms of the number of characters from the end of the line.

Note: In options, the abbreviated name for each option is in the brackets. And the detail for each option needs to explain. Here gives the explanation.

- autoindent: When using the :set ai command, the next line is aligned with the beginning of the previous line. Programmers usually use it to make the source codes neat and easy to recognize the hierarchy of a program.
- ignorecase: When using the :set ic command, search a string of characters ignoring the case of a letter, that is, the /chapter/ command searches for Chapter and chapter.
- number: When using the :set nu command, the line numbers are displayed when the file is being edited, but line numbers are not saved as part of the file.
- scroll: When using the :set scroll=10 command, the screen is scrolled by 10 lines at a time by using CTRL-D command to scroll the vi screen down and CTRL-U command to scroll the vi screen up.
- showmode: When using the :set smd command, the current vi mode is displayed in the bottom right corner of the screen.
- wrapmargin: When using the :set wm=5 command, the wrap margin is set to 5. That is, each line will be up to 75 characters long, assuming a line length of 80 characters.

To customize the vi environment permanently, put the option-setting command in the .exrc file in the home directory, as follows.

```
$ cat .exrc
...
set smd nu ic wm=4
...
$
```

All the options of displaying current mode, line number, ignoring case, and the wrap margin are set permanently.

In fact, there are some more features in the vi editor, which are worth learning from the practice with vi. The more you use it, the more you gain

from it, and the more you know it.

3.8 Introduction of the emacs Editor

Among the three editors of pico, vi and emacs, the emacs editor is the most complex and flexible one. It gives its users the most freedom to control the way of editing text files. There are two major "brands" of emacs: GNU Emacs and Xemacs, and they have very similar functions (Sarwar et al 2006). As the emacs editor is not the main part for this book, only a primary introduction about the emacs editor will be given. Several typical features of the emacs editor, especially different from the vi editor, will be discussed in this book. To learn more, readers can read the references listed at the end of this chapter.

Similar to the vi, within the emacs, the user can execute shell commands and scripts. As it can integrate a compiler and linker into it, the emacs is more convenient for the programmers, especially to program source code. Its shortcoming is that its keystroke commands are more complex to remember, especially for the new learners, than the vi. And some commands in the emacs editor have different functions from ones in the vi. But it may be the common case that the flexibility and the complexity are usually coming together in one system.

Additionally, like vi, emacs has also major modes of operation, such as Lisp mode and C mode, but they works in different way. In emacs, operation modes just help to format the special text in that operation. In vi, operation actions in the vi Command and Insert Modes are totally different.

The emacs program can be running either in a terminal or in a GUI, such as X Window System (see Section 2.6). When the program starts, a new window appears on-screen, looking similar to a typical window of Microsoft Windows in having some typical features, such as, menu bar, speed bar (for quick operations), status indicator, pull-down bar, scroll bar, etc. In the emacs window, the menu bar has items of File, Edit, Options, Buffers, Tools, and Help. Among them, File is for opening, saving, and closing buffers, files, windows, and frames; Edit is to modify text in buffers; Options are to make configuration changes; Buffers is a pull-down menu holding the currently open buffers; Tools are File and application functions; and Help is an extensive document and on-line manual for emacs. The speed bar contains buttons for file and buffer operations, editing operations (such as, cut, paste), and some other functions (such as printing, searching, and changing preferences). These features are so similar to those of Microsoft Windows that the users who are familiar to Microsoft Windows can get used to them quickly. Therefore, here, just focus on the different part of the emacs in UNIX from a window in Microsoft Windows.

3.8.1 Start emacs, Create File, Exit emacs

To start emacs from the command line, use the following command.

```
$ emacs [option(s)] [file(s)]
```

Function: to start emacs to create a new text file or edit an existing text file.

Common options:

+nw: to start emacs without opening a window;

-n: to begin to edit files(s) starting at line n.

Note: Without any option and argument, emacs starts with an empty buffer.

If a graphical emaes cannot be run and only a text-only console or terminal can work, the Menu Bar at the top of the emaes screen can be accessed by pressing ESC key on the keyboard and then pressing the single back-quote ` key. Move through the menu bar choices by pressing the letter key of the menu choice. For example, pressing the F key to access the File pull-down menu choices, and pressing the s key to save the current buffer. However, the speed button bar menu choices cannot be accessed from within a text-only display of emacs.

For example:

```
$ emacs
```

The emacs editor is a full-screen display editor. Within it, the user can use keystroke commands and also can use the GUI menus and graphical input methods to do programming. Some of important keystroke commands in the emacs are listed in Table 3.9.

Table 3.9 Important commands of emacs

Keystroke Command	Action
CTRL-X + CTRL-C	To exit emacs
CTRL-H	To access Help documentation
CTRL-X + CTRL-S	To save the current buffer
CTRL-X + CTRL-W	To save a buffer for a new file
CTRL-X l	To close all windows but this one (useful in Help documentation)
CTRL-X +CTRL-U	To undo the last edit and it can be used several times if necessary
CTRL-G	To cancel the current command
CTRL-X I	To insert text from a file at the current cursor position
CTRL-D	To delete (kill) the character at the cursor
ESC-D	To delete (kill) characters from the cursor to the end of the current word

Continued

Keystroke Command	Action
CTRL-K	To delete (kill) characters from the cursor to the end of the current line
ESC-DELETE	To delete (kill) the word before the cursor
CTRL-W	To delete (kill) the region
ESC-W	To copy the region to the Kill Ring, but not to delete the text from the current buffer
CTRL-Y	To paste into the buffer what had been deleted
CTRL-SPACE or CTRL-@	To place the mark before the cursor

Note: In vi, yanking copies text from the main buffer, but in emaes yanking pastes into the main buffer. The concept of a buffer in emacs is very important, and quite similar to it is in vi.

3.8.2 Buffers, Mark and Region in emacs

In the emacs, the buffer is a text object that is currently being edited by the emacs. The buffer is different from a file. The buffer is in the memory, but the file is a text object stored on disk. In other words, the object currently being modified and viewed in the emacs cannot be the same object stored on disk until the buffer is saved. The emaes can just work on text objects that are in the memory and not files. When the emacs with an argument of a file to edit is first launched, the buffer is created by emacs for that file in what is generally known as an emacs frame, with a single window open holding the buffer contents. A frame consists of one window, the pull-down and speed button bar menus, the Mode line, a minibuffer, and so on.

The minibuffer is a bar at the bottom of the emacs window, which is for information and questions/prompts from the emaes. It functions like the Last Line Mode line in the vi.

The emacs concepts of point and cursor location are different from the vi. In emacs, the point is at the left edge of the cursor, in other words, between the characters or the white space. It is also the location in the buffer where being edited. This difference is important for the cut/copy/paste operations.

For the emacs, point and mark are used to delimit a region for the cut/copy/paste operations. As mentioned above, the point is located in the white space before the character highlighted by the cursor. The mark is also in the white space before the character the cursor is highlighting, but it is set by placing the cursor over a character and then holding down CTRL-SPACE or CTRL-@. The mark is also a holder in the buffer. The region is all text between the point and the mark, which is the area of text to cut, copy and paste. For example, given a line of text “This is the example for editing a file”, highlight the character f in the word “file” and set the mark by holding

down CTRL-SPACE; then move the point before the e in the word "example" by highlighting the e in the word "example"; then the region is defined as "example for editing a ".

The Kill Ring is a buffer where the deleted (killed or cut) text is stored. As listed in Table 3.9, the killing (deleting or cutting) commands can put the text into the Kill Ring buffer. Then the text in the Kill Ring can be pasted into the current editor buffer by yanking it. The Kill Ring is a FIFO buffer, so the killing can be done several times sequentially, and then the yanking sequence will be done in the same order as the killing sequence.

For example, to cut four words from a current editor buffer and then paste them back at another position, move the point before the first word to cut and press ESC-D four times. The four words are then cut into the Kill Ring. Then move the point to where to put the four words and press CTRL-Y. The four words are pasted into the current editor buffer in the same order, left-to-right as they were cut.

The copying command can also put the copied text into the Kill Ring buffer. For example, to copy two words of text and then paste them at another position, set the mark by positioning the point after these two words, and then press CTRL-@. Then move the point before these two words; a region between the point and mark is defined. There is only one mark in the current editor buffer. Press ESC-W to put the text in the region into the Kill Ring; but the text is not deleted from the original place. To put the two words at another position, move the point there and press CTRL-Y. The two words are pasted into the new position.

3.8.3 Cursor Movement Commands

There are some cursor movement commands in the emacs that can be used in editing a file. They are listed in Table 3.10.

Table 3.10 Cursor movement commands of the emacs

Keystroke Command	Action
ESC-F	To move the cursor forward one word at a time
ESC-B	To move the cursor backward one word at a time
CTRL-A	To move the cursor to the beginning of the current line
CTRL-E	To move the cursor to the end of the current line
ESC-<	To move the cursor to the beginning of the buffer
ESC->	To move the cursor to the end of the buffer

3.8.4 Keyboard Macros

The keyboard macros in the emacs are collections of keystrokes that can be defined and recorded once and then accessed as many times as needed at any time later on. It is helpful for some repetitive multiple keystroke operations. Users can do it in this way: define a sequence of multiple keystroke operations as a single command, and then execute that command when necessary. The emacs contains a function that allows defining keyboard macros. The keystrokes can be emacs commands and other keyboard keys. A macro can be saved with a name, and even saved into a file. Table 3.11 shows a list of some of the keyboard macro commands.

Note: If a mistake is made when a macro being defined, hold down the CTRL-G keys to cancel the current macro definition. If the macro is a string of characters, pressing the CTRL-X + E keys can paste the string on the position of the cursor in the buffer.

Table 3.11 Some keyboard macro commands of the emacs

Keystroke Command	Action
CTRL-X + Shift+9	To begin the macro definition
CTRL-X + Shift+0	To end the macro definition
CTRL-X + E	To execute the last-defined keyboard macro
ESC-X + macro-name + Enter	To name the last-created macro
ESC-X + name	To repeat the named macro name

3.8.5 Search and Replace

There are two modes for global search and replace: one is unconditional, where every occurrence of old text to replace with new text is replaced without prompting; the other is interactive, where the emacs prompts before each occurrence of old text replaced with new text.

In the unconditional mode, for example, to replace the word "red" unconditionally with the word "black" from the current position of the point to the end of the current editor buffer, press ESC-X, type replace-string, and then press Enter. That is

```
ESC-X replace-string Enter
```

Then the emacs prompts for the old string. Type red and then press Enter. Then the emacs prompts for the new string. Type black and then press Enter. All reds are replaced without prompt.

In the interactive mode, press ESC-X, type query-replace, and then press Enter. That is

```
ESC-X query-replace Enter
```

Then input old and new strings. At each occurrence of the old string to replace, the emacs prompts whether or not to replace the old string with the new one. Table 3.12 shows the actions to take while in the midst of an interactive search and replace mode.

Table 3.12 Editing commands in interactive search and replace mode

Keystroke Command	Action
DELETE	To keep searching without this replacement
Enter	To stop replacing
SPACE	To do this replacement and then continue searching
, (comma)	To make and display this replacement, and then prompt for another command
. (period)	To make this replacement and then end searching
! (exclamation mark)	To replace this and all the rest unconditionally

Operation example 3.7 contains an example to create and edit a file in the emacs. Through it, practice different commands in the emacs.

3.8.6 Operation Example

In Section 2.4.5, the alias command has been discussed. With it, it is possible to create an alias for a command. As mentioned before, command aliases can be placed in shell setup files or configuration files, such as the .profile file (System V), the .1ogin file (BSD), and the .cshrc file (C shell). The .profile or .login file executes when logging in, and the .cshrc file executes every time starting a C shell. In Operation example 3.7, at the same time of practicing how to create and edit a file in emacs, it can help to enhance the knowledge of alias command and its application.

Since many commands in UNIX function in the similar way as DOS commands, it is possible to simulate DOS commands in the UNIX operating system by naming those UNIX commands as DOS command names. Once that done, executing those alias commands gives the feeling of running programs in a DOS environment.

Having used MS-DOS or Microsoft Windows, readers must know MS-DOS commands. In this example, through command aliases, simulate running DOS commands in the Bourne shell of the UNIX operating system. In this example, the Bourne shell is assumed to launch; if not, change the shell into the Bourne shell by typing sh after the shell prompt (the detailed will be discussed in Chapter 8). It is also necessary to modify an existing file, .profile (see Section 2.4.5), that is in the home directory so that the Bourne shell has been launched and being used, and the dosalias file in this example is in this

directory.

Operation example 3.7 DOS Aliases, practicing how to create a file to define aliases that allow typing DOS command names at the UNIX shell prompt to execute some of the common UNIX commands.

In this example, it also accomplishes to insert the alias file into the .profile file, so that these DOS-aliased commands will be available when logging in again. However, be careful that before practicing this example, make a copy of the original file of .profile by using the method introduced in Section 2.6. Follow the example step by step to practice:

1) At the shell prompt, type ls -la command to look for the .profile file in the home directory. If there is no .profile file in that home directory, then use emaes to create a new empty file named .profile. Check out which shell is the current shell by typing echo $SHELL after the shell prompt. If the current shell is the C shell, the shell prompt should be % and the system will respond with /bin/csh. If it is the Bourne shell, the shell prompt should be $ and the system will respond /bin/sh.
2) If the Bourne shell is the current shell, go to step 3 directly. If not, change the shell into the Bourne shell by typing sh after the shell prompt.
3) At the shell prompt, type the following command.

```
$ emacs dosalias
```

   Then the emacs screen appears on the display.
4) Type "# use DOS aliases to simulate DOS commands in UNIX", and press Enter at the end of the line.
5) Type "alias copy='cp\!*'", and press Enter at the end of the line.
6) Type "alias dir='ls -la\!*'", and press Enter at the end of the line.
7) Type "alias type='more\!*'", and press Enter at the end of the line.
8) Type "alias del='rm\!*'", and press Enter at the end of the line.
9) Press CTRL-X keys at the same time and press CTRL-S keys at the same time to save the current editor buffer into the dosalias file. The content of the dosalias file should be like Figure 3.27.
10) Press CTRL-H keys at the same time and press A key. The minibuffer area shows a prompt to obtain Help. Press CTRL-G keys at the same time to cancel the Help request.

```
#use DOS aliaese to simulate DOS commands in UNIX
alias copy= 'cp\!*'
alias dir='ls -la\!*'
alias type='more\!*'
alias del='rm\!*'
~
~
```

Fig. 3.27 The content of the dosalias file.

11) From the File pull-down menu, choose File>Open File. In the minibuffer, after the prompt Find file: ∼/, type .profile. A new buffer opens on-screen, which holds the contents of the .profile file. Position the cursor on a blank line in the .profile file.
12) Again, from the File pull-down menu, choose File>Insert File. In the minibuffer, erase anything on the line with the ← key, and type ∼/ dosalias, and then press Enter. The text of dosalias's DOS aliases should now be inserted into the .profile file.
13) Again, from the File pull-down menu, choose File>Save (current buffer) to save the .profile file.
14) Press CTRL-X keys at the same time and press CTRL-C keys at the same time to return to the shell prompt.

To test the new .profile file, log out and then log in the UNIX operating system again. Then in a terminal or console window, and at the shell prompt, type one of the aliased DOS commands and see the results. Without any mistake, for example, when the dir command is typed, the results of the ls -la command should be gotten.

3.8.7 Programming in emacs

For programmers, it is convenient to do multiple tasks within an editor. Except as the text editor, the emacs can assist to accomplish some other tasks, including program development in C, Java, and HTML, shell script execution, Internet work, and so on. For program development in a high-level language, it needs not only to write the source code, but also to compile, link, debug, and execute the program itself. With some of the built-in facilities, it can be easily done in the emacs. These all-in-one capabilities are a current trend because it is really convenient to accomplish common tasks from within one editor.

If these high-level-language facilities are available in the UNIX operating system, within emacs, users can type in the source code of a C program, indent the text, compile the source code, correct compile-time errors, link the source code, and even execute the program in a terminal window to test it. If the path for the current shell includes the working directory where the compiled and linked executable program is saved by the emacs, it is capable to run the program. Otherwise, it is necessary to include the path to this directory. To do so, check out the path variable by typing echo $PATH to see if the path display includes the current working directory where the emacs saves the program. The information about how to set the path variable will be discussed in Chapter 8.

3.9 Summary

Three useful text editors that UNIX operating system offers, which are the pico, vi, and emacs, have been discussed in this chapter. With one of them, some text editing chores can be done, such as editing script files, writing e-mail messages, or creating C language programs. As the UNIX operating system is a text-driven operating system, these editors consist of many keyboard commands to accomplish operations, such as starting a new file, opening an existing file, saving a file, quitting, cursor movement, cut- or copy-paste, deleting text, inserting text, and search and replace. Buffers are important for users of these editors. Different buffers provide different functions. Knowing their duties can assist to make the best use of them.

Problems

Problem 3.1 Compare the difference between a text editor and a word processor.

Problem 3.2 There are many general keystroke commands in the pico, which keystroke commands can be used to realize the operation of cut and paste a text and the operation of copy and paste a text?

Problem 3.3 Simulate the Operation example 3.1 to create a short text named myfile1 and save it.

Problem 3.4 Simulate the operation example 3.2 to create a short text named myfile2 and save it. Try to use those general keystroke commands as many times as possible in order to get used to them.

Problem 3.5 Edit the file named example2 that is created in operation example 3.2, and use the cursor movement keystroke commands to position the cursor at the beginning and end character of each line of text. Use the CTRL-C command to locate each character position when moving the cursor. Use the CTRL-T to check spelling. Try to check how many characters there are in the file, and what percent of the total file the beginning of each line is.

Problem 3.6 Execute the pico, and create and edit a short text for an e-mail message explaining some of the pico keystroke commands to your friend. Save it into a file called myunixemail.

Problem 3.7 How many modes are there in the vi text editor? What are they? In each mode, what can you do, respectively?

Problem 3.8 Give five keystroke commands in the vi and explain their actions, respectively.

Problem 3.9 Simulate the operation example 3.3 to create a short text named sehllscpt1 and save it. Type and execute the sehllscpt1 script on the command line, like the following, and see what are listed on the screen.

```
$ sh shellscpt1
```

Problem 3.10 In the vi editor, how can you switch from Command Mode to Insert Mode? And how can you switch from Insert Mode back to Command Mode?

Problem 3.11 Simulate the operation example 3.4 to create a short text named ex1.c and save it. Try to use the keystrokes that are shown in Table 3.4 as many times as possible to get familiar to them.

Problem 3.12 Simulate the operation example 3.5 to continue to edit a short text named ex1.c and save it. Try to use the general commands that are useed in Command Mode, shown in Table 3.5, as many times as possible to get familiar to them. Also try to use the general commands that are used in Last Line Mode, shown in Table 3.6.

Problem 3.13 With the vi, begin editing the file that is created in the operation example 3.4. Use the commands of G, $, 0 and w to move the cursor. Practice how to save a copy of the file with a different filename while still in vi without quitting this editing session. What happens if having accomplished four operations in vi, type 4u in Command Mode?

Problem 3.14 Edit the file created in Problem 3.13, and change the order of the text by using the yank, put, and D or dd commands.

Problem 3.15 What is an editor buffer? What is a general purpose buffer? And what is an indexed buffer used for?

Problem 3.16 Launch the vi and create a file named dosalias as the file in the operation example 3.7, practicing how to create a file to define aliases that allow typing DOS command names at the UNIX shell prompt to execute some of the common UNIX commands. Edit the file. Follow the example, make sure that the Bourne shell is launched. Make sure that the dosalias file is in this directory. Insert the dosalias file into the .profile file, so that these DOS-aliased commands will be available when logging in again. Log out. And log in again, check out the DOS aliased by typing and executing them.

Problem 3.17 In the emacs, what is a buffer for? Where is the buffer, in the memory or on the disk? What is the minibuffer for?

Problem 3.18 For the emacs, what are point and mark used for? Where are they located in the emace editor? How can you set the mark and point for a region of a cut/copy/paste operation? Give an example and notice the operation order of setting the mark and point.

Problem 3.19 In the emacs, what is the kill ring? How does it function? Give a copy-paste example by using the kill ring.

Problem 3.20 Launch the emacs from a console or terminal window by typing emacs -nw myfile. In the terminal window, a non-graphic emacs will open on the buffer of myfile. Practice to gain access to the Menu Bar menus at the top of the emacs screen by pressing the Esc key and then the back-quote ` key. Practice to make a choice by pressing a letter key of the menu.

Problem 3.21 With the emacs, simulate the operation example 3.7 to create a short text named dosalias that holds some DOS command aliases and save it. And do the following operations in the operation example 3.7 step by step carefully and see the execution results.

Problem 3.22 Give the names of three text editors introduced in this chapter and compare them.

References

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Baecker R (1986) Design principles for the enhanced presentation of computer program source text. CHI'86: Conference on Human Factors in Computer Systems, Boston, Massachusetts, US, 13 – 17 April 1986, ACM: 51 – 58

Douglas SA, Moran TP (1983) Learning text editor semantics by analogy. CHI'83: Conference on Human Factors in Computer Systems, Boston, Massachusetts, December 1983, ACM: 207 – 211

Makatani LH, Egan DE, Ruedisueli LW et al (1986) TNT: a talking tutor 'N' trainer for teaching the use of interactive computer systems. CHI'86: Conference on Human Factors in Computer Systems, Boston, Massachusetts, 13 – 17 April 1986, ACM: 29 – 34

McKusick MK (1999) Twenty years of Berkeley Unix: from AT&T-owned to freely redistributable. LINUXjunkies.org. http://www.linuxjunkies.org/articles/kirkmck.pdf. Accessed 20 Aug 2010

Pelaprat E, Shapiro RB (2002) User activity histories. CHI'02: Conference on Human Factors in Computer Systems, Minneapolis, Minnesota, April 2002, ACM: 876 – 877

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading, Massachusetts

Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17(7): 365 – 375

Rosson MB (1985) Effects of experience on learning, using, and evaluating a text editor. Human Factors 26: 463 – 475

Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: the textbook, 2nd edn. China Machine Press, Beijing

Stallings W (1998) Operating systems: internals and design principles 3rd edn. Prentice Hall, Upper Saddle River, New Jersey

Walker N, Olson JR (1988) Designing keybindings to be easy to learn and resistant to forgetting even when the set of commands is large. CHI'88: Conference on Human Factors in Computer Systems, Washington, D.C., US, May 1988, ACM: 201 – 206

Ward W (2003) Getting started with vi. Linux Journal 2003 (114): 5

Wilder D (1997) At last, an X-based vi. Linux Journal 1997(34): Article No. 6

4 UNIX Process Management

Since UNIX is one of multi-user and multiprocessing operating systems, UNIX has its solution to the system resource management. The UNIX kernel handles almost all the basic issues related to process management, memory management, file system, and I/O system, and provide well-defined system programs that have the clear-cut assignment of responsibility in order to allow user programs to call them with system calls (Bach 2006; McKusick et al 2005; Mohay et al 1997). This chapter will discuss the UNIX process management. Chapter 5 will introduce the UNIX memory management. Chapter 6 will focus on the UNIX file system. Chapter 7 will be related to the UNIX I/O system.

In this chapter, multiple processes' running concurrently will be introduced first. Then, process states, jobs, process and job attributes, and process and job control will be discussed. Foreground and background processes will be explained, too. And finally, UNIX system root and init process will be involved.

4.1 Multiple Processes' Running Concurrently

In a single-user and single-process operating system, programs must be executed one after another. In this environment, the execution of a program has a significant feature, that is, the execution is unable to break in half way by other programs and resume later without effect, except by the system interruptions. In other words, during the duration from the start to the end of a program execution, the program execution occupies the CPU and memory without sharing them with any other program until its execution finishes. The reason is that these kinds of operating systems do quite little work related to the management of CPU and primary memory. However, the exclusive execution of a program can be quite a waste. For example, when an I/O-intense program is executed, most of time, CPU is idle and just waiting for I/O device operations without doing anything. For most of modern computer systems that usually have much faster microprocessors and much

larger memory space than their ancestors, a single-user and single-program operating system cannot manage well and make the best use of all the system resources. Thus, operating system developers have developed multi-user and multi-processing operating systems.

In a multi-user and multi-processing operating system, not only the executions of several programs can share CPU in an interlaced way without any undesirable impact, but also several users can interact with the system without any delay (Andrews 1983; Bach 2006; Cmelik et al 1989; Sarwar 2006; Sielski 1992; Stallings 1998; Zhou 2009). The reason is that these kinds of operating systems do quite a lot of work on the management of CPU and primary memory in order to make the best use of the system resources. Included in the most important parts of system resource management are process management, memory management, file system, and I/O device system. Among them, process management tackles how to control and schedule CPU to accomplish the tasks of different users successfully.

In a multi-user and multi-processing operating system, the only active entities are the processes. A process is an execution process of a program. In a large multi-user time-sharing system, there may be hundreds or even thousands of processes running, and each user on the system may have several active processes at once. In fact, even though no users use the system that is running, dozens of background processes, called daemons, are executing.

As one of the multi-user and multi-processing operating systems, UNIX has its own solution to the system resource management. UNIX kernel handles almost all the basic problems related to process management, memory management, file system, and I/O device system. UNIX processes are very similar to the classical sequential processes. Each process runs a single program and initially has a single thread of control.

Multiprogramming not only allows the processor to handle multiple batch jobs at a time, but also be used to handle multiple interactive jobs. This is multiple processes' running concurrently. In a time-sharing system, the processor's time is shared among multiple users, and the execution of each user program is in a short period of time of computation. This short period of execution time for each user program is called a burst (or quantum, or time slice). In a UNIX operating system, the quantum is usually 0.1 – 1 sec. That is, one process is executed for a quantum, and then the CPU is taken away from it and given to another process. The new process executes for a quantum and then the CPU is given to the next process. The process management part of the UNIX kernel takes the responsibility to schedule and switch context for the next process that is ready to run.

A process has its lifetime, which starts from the moment when UNIX kernel creates it and ends at the time when it is removed from the system. Process creation with the fork system call and termination via the exit system call are the only mechanisms used by the UNIX system to execute external commands and programs. Every time a user runs an external command or program, the UNIX kernel creates a process for it to execute; once its execu-

tion is finished, the process is removed from the system.

4.1.1 Fundamental Concept for Scheduler and Scheduling Algorithm

As only one CPU is available in a uniprocessor system, a choice has to be made for which process to run next. It is the scheduler to make the choice (Anderson et al 1992; Braams 1995; DeBenedictis et al1993; Forrest et al 1996; Peachey et al 1984; Quarterman et al 1985; Ritchie et al 1974; Sielski et al 1992; Thompson 1978). The algorithm that the scheduler uses is called the scheduling algorithm (Bach 2006; Denning 1983; Devarakonda et al 1989). Because CPU time is a resource on a computer system, a good scheduler can make a big difference in perceived performance and user satisfaction.

On the other hand, nearly all processes alternate bursts of computation with disk I/O requests. For example, image this scenario: the CPU runs for a while without stopping, then a system call is made to read from a file; when the system call completes, the CPU computes again until it needs more data, and so on. In some situations, the processes spend most of their time computing. In others, the processes spend most of their time waiting for I/O devices. The former are called compute-bound processes; the latter are called I/O-bound processes. As the I/O-bound processes have more I/O requests, they should get more chance of being scheduled the CPU in order that they can issue disk requests and keep the disk I/O busy.

For any operating system, scheduling algorithm goals are usually three: to give each process fair share of the CPU, to monitor that the stated policy is carried out, and to keep all parts of the system busy.

The simplest of all scheduling algorithms is the first-come first-served (FCFS) (Stallings et al 1998). With this algorithm, processes are assigned the CPU in the order they request it. The first job enters the system and is allowed to run until it is blocked for some reason, and then the next ready job in the ready queue occupies the CPU. When a blocked process becomes ready, like a newly arrived job, it is put on the end of the queue.

In most of the time-sharing systems, the round robin (RR) scheduling algorithm is used (Stallings 1998). With this algorithm, each process is assigned a time slice to run. At the end of its quantum, the running process is preempted to use the CPU by another process. This algorithm is a natural choice for time-sharing systems, because it makes all users in the system are likely to see the progress of their processes.

The round robin scheduling makes the implicit assumption that all processes in the system are equally important. Otherwise, some users who own and operate multi-user computers have different ideas on which process is more important. In this way, it needs to take external factors into account to make a decision on who is more important in the system and who can

have the privilege to use the CPU first. This kind of scheduling is called the priority scheduling. That is, each process is assigned a priority, and the ready process with the highest priority is allowed to run. For example, a priority scheduling assigns a higher priority to an I/O-bound process than a compute-bound process. With this scheme, an I/O-bound process, such as a process running the vi editor, gets the higher priority.

There are some more scheduling algorithms for different operating systems. If interested, readers can see the operating system books.

For operating system programming, usually, the code known as the processor scheduler implements the CPU scheduling algorithm while the code known as the dispatcher accomplishes to take the CPU away from the running process and hands it over to the next scheduled process.

4.1.2 UNIX Scheduling Algorithm and Context Switch

As UNIX is a multi-user system, its scheduling algorithm was designed to provide a good response to interactive processes. Its scheduling algorithm includes two levels: the low-level algorithm is used to schedule the processes in the memory and ready to run; the high-level algorithm is used to make the decision on which process to bring in the memory from the disk and to be ready to run.

Usually, maybe slightly different for different UNIX operating systems, the low-level algorithm uses multiple waiting queues. Each queue has a unique range of non-overlapping priority values. Processes executing in user mode have positive values while processes executing in kernel mode have negative values. Negative values have the highest priority, and large positive values have the lowest, as shown in Figure 4.1. Only processes that are in memory and ready to run are located on these queues.

The low-level scheduler searches the queues starting from the highest priority until it finds a queue that is occupied. The first process on that queue is then chosen to run.

The processes within the same priority range are scheduled on the FCFS basis and share the CPU using a round robin algorithm. The running process is allowed to run for a maximum of one quantum or until it blocks. If the process uses up its quantum without being terminated, it is put back on the end of its queue. The scheduler searches again.

The priority value for every process in the system is recalculated every second. Recalculating priority values of all the processes every second changes process priorities dynamically. The formula used to compute the priority value is as follows.

$$\text{priority} = \text{base} + \text{nice} + \text{CPU\_usage} \tag{4.1}$$

The three components of the priority formula and their meanings are listed in Table 4.1.

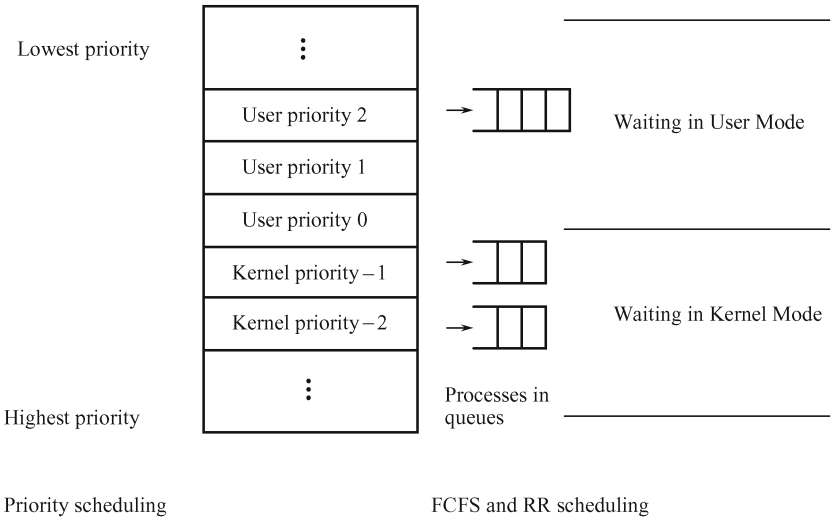

Fig. 4.1 UNIX low-level scheduling algorithm.

Table 4.1 Components of the priority formula

Components	Representations
base	An integer usually having a value of 40 or 60.
nice	A positive integer usually with a default value of 0, ranged from −20 to +20; each process has a nice value which can be set in the range 0 to 20 by using the nice command.
CPU_usage	The number of clock ticks for which the process has used the CPU

Note: The bigger the priority value of a process is, the lower priority the process has. Thus, if a user sets the nice value a positive integer, in fact, he or she is nice to the other users. Only the system administrator may ask for better (setting the nice value from −20 to −1) than other users' services.

Every time the clock ticks, the CPU usage counter increases the tick count for the process currently using the CPU by 1. This counter will ultimately be added to the process' priority giving it a higher numerical value and thus putting it on a lower-priority queue. However, CPU_usage value decays with time. Different versions of the UNIX operating system do the decay slightly differently. One way is that the tick count for every process is divided by 2 before process priorities are recalculated by using the formula shown above. In other words, before the next recalculation of the priority values, add the current value of the tick counter to the number of CPU_usage value acquired in the past and divide the result by 2. This algorithm weights the most recent value of the tick counter 1/2, the one before that by 1/4, and so on. The CPU_usage value therefore increases for the process using the CPU and

decreases for all other processes. This weighting algorithm is very fast because it just has one addition and one shift, but there are also some other weighting schemes being used.

The formula clearly indicates that UNIX gives higher priority to processes that have used less CPU time in the recent past. That is, the lower the recent CPU usage of a process is, the lower its priority value is, and the higher its priority is. This is why a text editor such as the vi editor gets higher priority than a process that just does computation by using the CPU. The vi editor is an I/O-bound process and spends most of the time in doing I/O (reading keyboard input and outputting it on the screen).

By the way, in order to make each process execute in its own environment without undesirable effect on the executed result when scheduled, along with each process scheduling, there is a context switch, which is to make a switch from the context of the current running process to the next running process.

The context of a process is typically composed of three portions: contents in hardware registers, contents of its user address space, and kernel data structures relevant to the process.

- The register context includes: the contents of the program counter, the processor status register, the stack pointer, and some general-purpose registers.
- The user address space context includes the instruction code, data, user stack, and shared memory space of the process.
- The kernel data structures include two parts: one is static, the other is dynamic. The former further includes: the process table entry of the process, the user structure of the process, and all the information of the virtual memory space belong to the process. The latter is a kernel context stack that adopts the pushing and popping operations on occurrences of various events from layer to layer.

The kernel context stack can be pushed a layer not only when an interrupt occurs, but also when a process makes a system call, or when the scheduling needs a context switch. Opposite, it can be popped a layer when the kernel returns from an interrupt service, when a process returns from kernel mode to user mode after a system call return, or when the scheduling needs a context switch. When scheduling, the kernel pushes one layer of the current running process, pops one layer of the next running process, and also stores some necessary information to recover the current context layer in the process table entry.

4.2 Process States

The operating system has the responsibility of controlling the execution of processes. At any time, a process is either being executed by a processor or not. Each process must be represented in some way so that the operating

system can keep track of it. That is, there must be some data structure holding the information relating to each process, including current state and location in memory. Processes that are not running must be kept in some sort of queue, waiting their turn to execute. Therefore, there must be a model that describes the states of all the processes that are in the operating system.

4.2.1 Fundamental Concept for Process States

If all processes were always ready to execute, the queuing discipline would be effective, assumed that the queue is a first-in, first-out list and the processor operates in the round-robin fashion on the available processes. However, in the queue, some processes might be ready to execute while others might be blocked, waiting for an I/O operation to complete. So the scheduler would have to scan the list looking for the process that is not blocked and that has been in the queue the longest. A natural way to handle this situation is to distinguish the process states in five states: running, ready, blocked, created, and terminated. The state transition of five-state process model is shown in Figure 4.2. The description of different states is listed as follows:

- Running: In this state, the process is currently being executed. At most one process can be in this state at a time in a single-processor system.
- Ready: In this state, a process is prepared to execute when given the opportunity.
- Blocked: In this state, a process cannot execute until some event occurs, such as the completion of an I/O operation.
- Created: In this state, a process has just been created but has not yet been entered into the ready queue.
- Terminated: In this state, a process has been released by the operating system, either for it halted or for it aborted for some reason.

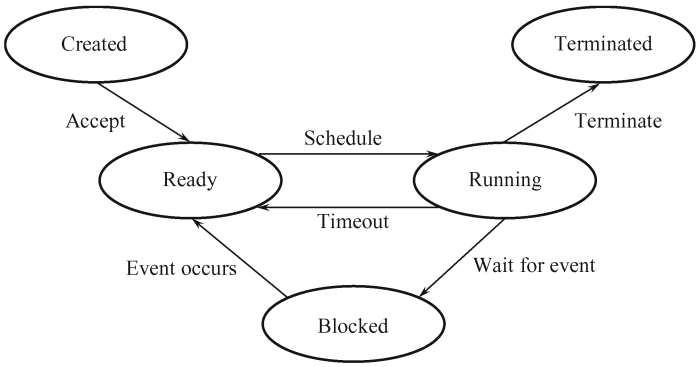

Fig. 4.2 State transition of five-state process model.

4.2.2 UNIX Process States

In a typical UNIX operating system, a process has more states than the five-state process model has, which are from being created to finishing its execution eventually and releasing from the system. Figure 4.3 shows a simplified UNIX process state transition, which is easier for readers to understand how a process to transit from one state to another in the UNIX operating system. Even though it has been simplified, it still has eight states. Compared to five-state process model of the previous section, the main differences are:

- The UNIX operating system has two running states: One is for a process that is running in user mode; the other is for a process that is running in kernel mode. Since there is only one process running in a single-processor system at any particular time, the running process is in one of these two running states at a certain time rather than in both of them.
- In UNIX, there are two states for ready processes: One is Ready in Memory that is for the processes that are ready to run as soon as the kernel schedules it; the other is Ready, Swapped for the processes are ready to run, but the swapper must swap the processes into main memory before the kernel can schedule them to execute.
- In UNIX, there are two states for blocked processes: One is Sleeping, in Memory that is for the processes that are unable to run until an event occurs; the other is Sleeping, Swapped for the processes that wait for the event occurring and have been swapped out to the secondary storage.

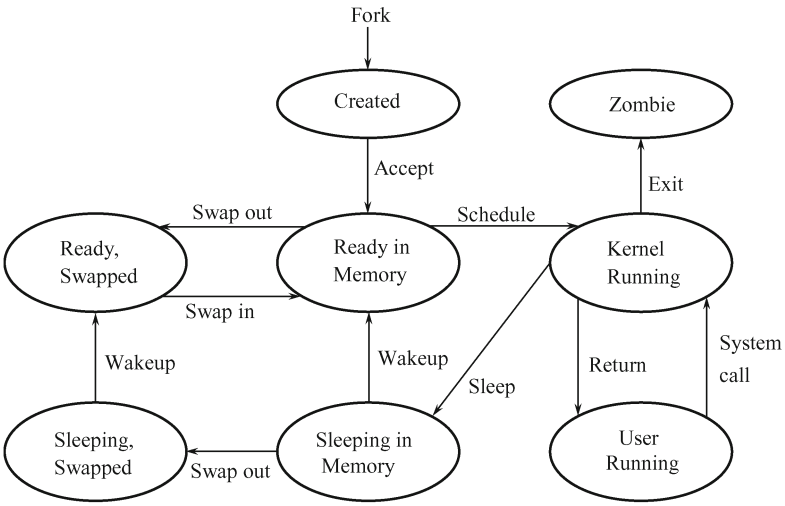

Fig. 4.3 Simplified UNIX state transition.

In Figure 4.3, all the process states in the circles should be recorded in a field of some data structure that represents a process, all the actions besides

the directed line segment outside the circles are the system programs in the UNIX kernel, and both of them compose of the process management part of the UNIX kernel.

Table 4.2 lists the description of the process states in the simplified UNIX process state transition.

Table 4.2 Simplified UNIX process states

Process States	Description
Created	Process is newly created and not yet ready to run
Ready in Memory	Process is ready to run as soon as the kernel schedules it
Kernel Running	Process is running in kernel mode
User Running	Process is running in user mode
Sleeping, in Memory	Process is in memory and awaiting an event
Sleeping, Swapped	Process is in the secondary storage and unable to run until an event occurs
Ready, Swapped	Process is ready to run, but needs the swapper to swap it into the main memory before the kernel can schedule it to execute
Zombie	Process is dying and leaves a record for its parent process to collect

Note: In the UNIX operating system, a process can be temporarily put on the swap space of the disk because of not enough memory available at this time.

Two processes, Processes 0 and 1 are unique in UNIX operating system, which are the processes whose process identifiers are 0 and 1, respectively. Process 0 is the process that is created when the system boots and predefined as a data structure loaded at boot time, and later on it is replaced with the swapper. Process 0 produces Process 1, the init process; all other processes in the system have Process 1 as a parent process. When a new interactive user logs on the UNIX operating system, Process 1 creates a user process for that user. Subsequently, the user process can create its child processes in a branching tree, so that any particular application can include a number of related processes.

A process can terminate normally when it finishes its work and exits the system while it can also terminate abnormally when it exits the system because of an exception (error) condition or intervention by its owner or the super-user. The owner of the process can use a command or a particular keystroke to terminate the process. The relative commands and keystrokes will be discussed later in the chapter.

4.3 Process Image and Attributes

To manage and control a process, UNIX operating system must know where the process is located and other attributes of the process. In other words, each process has a number of attributes that are used by UNIX for process control. These attributes, typically, are a collection of process identification, processor state information, and process control information. All the attributes are referred as a data structure called process control block (PCB). In fact, the UNIX kernel handles the process control block in two portions: process table and user structure. In addition, as a process is an executable entity, each process must include a set of instruction code, a set of data locations for local and global variables and any defined constants, and a stack that is used to keep track of procedure calls and parameter passing between procedures. All the information, including the set of instruction code, data, stack, and attributions, can be referred as the process image.

4.3.1 UNIX Process Attributes in Kernel

The UNIX kernel maintains a process through two key data structures, the process table and the user structure (also called u structure or u area). Every process has an entry in the kernel process table, and is allocated a user structure that contains private data manipulated only by the kernel. The process table is in the memory all the time and contains information needed for all processes. The user structure can be swapped out onto the disk when its associated process is not in memory in order to save memory for other processes. The process table entry can include the main information, listed in Table 4.3.

Table 4.3 Main information in the process table entry

Category	Elements
Process identification	Process ID, PID of the parent process, and IDs of the owner and group of the process
Processor state information	Process state
	Pointers to the instruction code, data, and stack segments in the memory or the disk; and the pointer to its user structure
	Process priority, amount of CPU usage time, and the nice value that the user sets
Process control information	Flags and signals that may be associated with communication between two independent processes
	Scheduling information
	Event that the process is awaiting before it can be resumed
	Links to other processes in a queue

The user structure is an extension for the process table entry. Some information of a process must be accessed by the kernel no matter whether or not the process is in the memory while some information cannot be needed by the kernel when the process is swapped out. The latter should be hold in the user structure. In other words, the user structure contains information that is not needed when the process is not in memory. For example, the event information must be in the process table since the process is waiting for it no matter whether or not the process is in memory; the information about file descriptors can be kept in the user structure because it is not possible for a process to read a file when it is swapped out. Table 4.4 lists the main information included in the user structure.

Table 4.4 Main information in the user structure

Category	Elements
Process identification	The real and effective user IDs of the process that allow different users to have various privileges to the file according to the situation Pointer to the process table entry associated to this user structure
Process control and environment information	Machine registers that are saved when a trap to the kernel occurs System call information that includes the system call parameters and return values Kernel stack that is a fixed stack for use by the kernel part of the process File descriptor table that indexes all the open files related to the process File system environment of the process, including current directory and current root Pointer to a table that keeps track of the CPU time in user mode and kernel mode used by the process Pointer to an array that indicates how the process to react to signals The terminal where the user who executes the process logs in

4.3.2 UNIX Process Attributes from User Angle

From the user angle, a process in UNIX operating system has its attributes, including owner's ID, process name, process ID (PID), process state, PID of the parent process, priority, and length of time the process has been running. When it is considered to controlling a process by using process control commands, PID of a process is important. Process control commands will be discussed later in this chapter. Now we introduce a command that can be used to check the states of the processes in the system.

Checking on the Process State

To view the attributes of processes in a system, use the ps command (report process state). Different versions of UNIX may have their own variants of ps command, and the outputs of the commands are slightly different.

The syntax and function of ps command are as follows.

```
$ ps [option(s)]
```

Function: to display and list processes and their attributes running in the system.

Common options:

-l: to display long list that may include 13 columns: state, user ID, PID, Parent PID, CPU usage, priority, nice value, address, size, wait channel, terminal, time, and command;

-a: to display information about the processes in the system except the session leader (the login shell);

-e: to display information about all the processes in the system;

-u UIDlist: to display information about processes belongs to the users listed in 'UIDlist' (UIDs separated by commas);

-g GIDlist: to display information about processes belonging to the groups listed in 'GIDlist' (Group-IDs separated by commas).

Note: The ps command without options will display four fields of information of all the processes belong to the user who ran the command, including PID, terminal, time, and command. Table 4.5 lists all the attributes of the processes of the output of the ps command with l option.

Table 4.5 Fields and their meanings of output of ps -l

Field	Description
STAT	State: R – Currently running; S – Sleeping for an event; T – Stopped (background process, suspended, or being traced); Z – Zombie process
UID	User ID: is process owner's user ID
PID	Process ID
PPID	Parent PID: is PID of the parent process
C	CPU Usage: is recent CPU usage, a parameter used in computing a process's priority for scheduling purposes
PRI	Priority value: indicates the priority with which the process is scheduled; the smaller the priority value of a process is, the higher its priority
NI	Nice value: is another parameter used in the computation of a process's priority value
ADDR	Address: indicates the memory address of a process if it is in the memory, or the disk address of a process if it is swapped out
SZ	Size: indicates the size of the memory image of a process in blocks
WCHAN	Wait channel: shows the event the process is waiting for if the process is a waiting or sleeping process, or shows null if the process is running processes or processes that are ready to run

Continued

Field	Description
TT	Terminal: shows the name of the terminal, which a process is attached to
TIME	Time: indicates the length of time (in minutes and seconds) a process has been running, except sleeping or stopping time
CMD	Command: shows the command used to launch this process

For example:

```
$ ps
PID        TT      TIME      CMD
777        U0      0:02      -sh
796        U0      0:00      ps
797        U0      0:09      vi
$
```

The last column of the output of the ps command shows that three processes are running at the terminal U0: -sh (Bourne shell), ps, and vi (vi editor) belong to the user who ran the command. The - in front of sh indicates that Bourne shell is the login shell. The first column lists the PIDs of sh, ps, and vi that are 777, 796, and 797, respectively. The third column shows that -sh, ps, and vi have executed for 2, 0, and 9 sec, respectively.

Second example: when using the -a option,

```
$ ps -a
PID        TT      TIME      CMD
797        U0      9:09      vi
990        U0      0:00      ps
$
```

This command displays all processes except the session leader, the login shell (-sh), compared with the former example.

Third example: use the -u option.

```
$ ps -u 112
UID     PID        TT     TIME    CMD
112     777        U0     0:02    -sh
112     797        U0     1:09    vi
112     990        U0     0:00    ps
$
```

This command outputs the information of all the processes belonging to the users with user ID 112.

Final example: use -l option.

```
$ ps -l
UID  PID PPID  C  PRI NI ADDR SZ  WCHAN  STAT TT TIME  CMD
1001 766 757   1  60  20 9e43 476 wait    S   U0 0:02  -sh
1001 771 777   12 66  20 feaf 288         R   U0 0:00  ps
1001 782 777   0  64  24 cbe9 356         T   U0 1:09  vi
$
```

This command shows the long list of processes on the system. Here, using three of the fields in the output of the ps -1 command, show how the UNIX

scheduler works (see Section 4.1.2). These three fields are C (CPU usage), NI (nice value), and PRI (priority value of the process). Remember that to compute the priority value, the formula (4.1) is used.

$$\text{priority} = \text{base} + \text{nice} + \text{CPU\_usage}.$$

In terms of the fields listed in Table 4.5 and corresponding to the terms of the formula (4.1), C is related to CPU_usage value, NI is nice value, and PRI is priority values.

In the final example in this section, the output of the ps -l command shows that the vi process is with a C value of 0 because it is currently stopped, has a NI value of 24 and a PRI value of 64. The shell process (-sh) has a C value of 1, a NI value of 20 and a PRI value of 60. By using the C, PRI, and NI values of the ps command, which are 12, 66, and 20, respectively, compute the base value used by this UNIX system:

$$\text{base} = 66 - 20 - 6 = 40.$$

Remember that the CPU_usage value should be given by C/2 before this calculation.

Assuming a process with a nice value of 20 and a CPU usage of 20, use this base value to compute its priority value:

$$\text{PRI (priority)} = 40 + (20/2) + 20 = 70.$$

Thus, the priority value of this process is 70.

By the way, as the higher the value of C for a process is, the higher its priority value is, and the lower its scheduling priority is, at the next time the UNIX scheduling, the kernel will assign the CPU to the shell process (-sh). In other words, the more CPU time a process has used in the recent past, the lower its priority becomes.

4.4 Creating a Process in UNIX

As mentioned above, a process has a lifetime starting from being created, and the UNIX operating system just uses the mechanism of process creation and termination to execute external commands and programs. This is also shown in Figure 4.3. System call fork is the only way to create a new process in UNIX.

4.4.1 Fork System Call

In Chapter 2, Table 2.1 has listed some of the major POSIX system calls. Among them is there the system call fork. The fork creates an exact copy

of the original process (the parent process), including all the file descriptors, registers, and everything else. All the variables have identical values at the time of the fork, but since the entire parent core image is copied to create the child's, subsequent changes in one of them do not affect the other one. In other words, after the fork, the original process (the parent process) and the copy (the child process) go their separate ways. The fork call returns a value, which is zero in the child, and equal to the child's PID in the parent. With the returned PID, the two processes can show their parent-child relationship. Figure 4.4 gives a C-style illumination of the return part of the program of fork system call.

```
if (the running process is the parent process)
{                                   /* the executing process is the parent process */
  if (free memory space is available)
  {
    Mark child process state as "ready in memory";
  }
  else
  {                                 /* swap the child process out on the disk*/
    Allocate the swap space on the disk for the child process;
    Put the child process on the swap space;
    Mark child process state as "ready, swapped ";
  }
  return (child's PID);             /* from the system call to user mode */
}
else
{                                   /* the executing process is the child process */
  Initialize timing fields in the user structure;
  return (0);                       /* to user mode */
}
```

Fig. 4.4 The illumination of return part of fork system call program.

After a fork, the child process will usually need to execute different instrument code from the parent process. For instance, a user types in a shell command. The shell reads this command from the terminal, creates a child process by using fork, waits for the child process to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent (the shell) executes a waitpid system call (see Table 2.1), which waits until the child process terminates (any child if more than one exists). The waitpid has three parameters. The first one allows the caller to specify the child to wait for. If it is -1, the parent process will wait for the first child to terminate. The second parameter is the address of a variable that will hold the exit value and show the child's exit status (normal or abnormal termination). The third parameter is used to determine whether

the caller blocks or returns if no child is terminated.

4.4.2 How UNIX Kernel to Execute Shell Commands

In the case of the shell, the child process executes the command typed by the user. It does this by using the execve system call (see in Table 2.1; different visions of UNIX can have different execve program), which causes its entire core image to be replaced by the file named in its first parameter. In general, the execve system call has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment array. The C-style illumination of the part of the shell program involving the use of the fork, waitpid, and execve is shown in Figure 4.5.

```
while (TRUE)
{
   Display  the shell prompt on the screen;
   Read the input  line from keyboard;
   pid = fork();                        /* create a child process */
   if (pid<0)
   {
    Display error message on the screen;
     continue;
    }
    if (pid  != 0)
     {                                   /* code for the parent process*/
      waitpid (-1,&staus, 0);          /*  the parent process waits the child process */
     }
    else
    {                                 /* code for child process*/
      execve (command, parameters, 0);   /* execute  the command */
    }
}
```

Fig. 4.5 The C-style illumination of part of the shell program.

To explain how the UNIX kernel to execute shell commands, some relevant examples will be given to display how the shell program to work. Before that, some relative concepts should be introduced.

Shell commands can be divided into two groups: internal commands and external commands. The internal commands are part of the shell process (built-in commands), so they are always in the memory. Some of the internal commands are called dot (.) commands. The external commands should be created processes for them when they are needed to execute in order to bring into the memory. Both of two groups include different elements, respectively.

- Internal commands: include commands, such as bg, cd, continue, echo, exec, exit, export, fg, jobs, pwd, read, readonly, return, set, shift, test, times, trap, umask, unset, and wait.
- External commands: consist of commands, such as cat, cp, ftp, grep, lp, ls, mkdir, more, ps, rmdir, sort, and telnet.

Executable files, like a binary code program and a shell script, are executed in the same way as external commands are.

In UNIX, a process can create another process by using the fork system call, which creates an exact copy of the original process in the main memory. After the fork, parent and child processes continue execution differently. The following examples can help explain how the shell to execute its commands with the fork system calls.

First example: Figure 4.6 displays the whole operation process that a Bourne shell creates a child process — a copy of the Bourne shell, the kernel replaces the child process with the cp command, and finally it returns the original shell prompt. In the user view, this is just the accomplishment of the execution of the cp command. Notice that Figure 4.6 and other figures in this section are just illustrative, so they may not follow the conventions rigorously.

Figure 4.6 shows that when the user typing in the cp file1 file4 command and Enter on the command line after the shell prompt from the keyboard, at the first step, a Bourne shell creates a child process — a copy of the Bourne shell by the fork system call; at the second step, the kernel replaces the child process with the cp command through the execve system call; at the third step, the parent process (the original Bourne shell) waits for the cp process to finish and terminate, and finally the parent process resumes, and the shell prompt displays on the screen again.

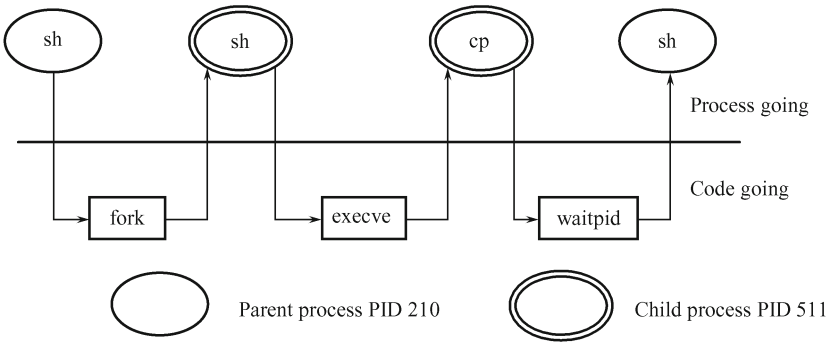

Fig. 4.6 Steps of executing the cp command on the shell.

The execution of a shell script, which usually includes a series of shell commands in a script, is just like to run those commands one by one. There-

fore, the parent shell creates a child shell and lets the child shell execute the first command in the shell script, and then execute the second command, and then execute the third, and so on. In the script, the external commands are created and replaced by the fork and execve system calls while internal commands are executed by the child shell directly. When the child shell executes commands in the script file, the parent shell is waiting. Once the child shell reaches the eof (End-of-File Marker) in the script file (see Section 6.3.7), it terminates. After the child shell terminates, the parent shell resumes its execution again.

Second example: Assumed a script named tested has the following two lines of the content:

```
pwd
ls
```

Figure 4.7 displays the operation process of tested's execution. That is, a Bourne shell creates a child process — a copy of the Bourne shell; the child process runs the pwd command first; after that, the kernel replaces the child process with the ls command, and finally it returns the old shell prompt. For the user, it is just the accomplishment of the execution of the ls command.

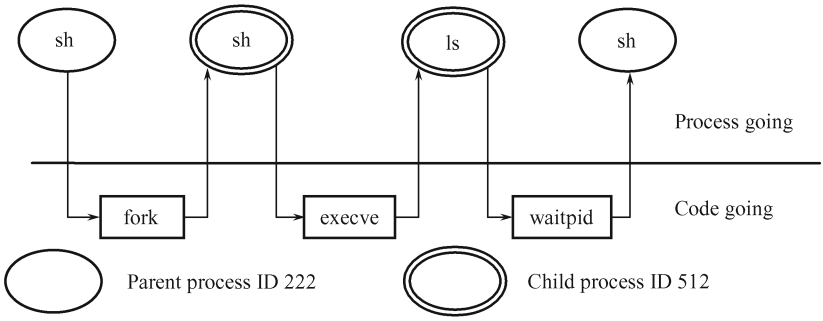

Fig. 4.7 Steps of executing the tested script on the shell.

Figure 4.7 shows that when the user typing in tested and Enter on the command line after the shell prompt from the keyboard, first, a Bourne shell creates a child process – a copy of the Bourne shell by the fork system call; second, the child shell runs the pwd command that is an internal command; third, the kernel replaces the child process with the ls command through execve system call; forth, the parent process (the original Bourne shell) waits for the ls process and the script to finish and terminate, and finally the parent process resumes, and the shell prompt displays on the screen again.

If the user does not run another shell in the original shell or does run a script of the same shell as the original shell, assumed a Bourne shell, the child process is just a copy of a Bourne shell.

Third example: If a user shifts the shells from the original shell to a different shell, assumed from the Bourne shell to the Korn shell, and then

types a Korn shell command, the kernel creates a child process of Korn shell, executes the more command, terminates its child process, prompts the Korn shell and waits for another command. If the user wants to come back the parent shell — in this case, it is the Bourne shell, use CTRL-D on a new command line to terminate the Korn shell that is the child process of the Bourn shell and come back to the Bourne shell. Figure 4.8 shows the steps.

The sh command is to run the Bourne shell, the csh to run the C shell, and the ksh to run the Korn shell.

Shell scripts can also be written for different shells and be executed under different shells. To do so, simply indicate clearly the full pathname of the shell, for which the script is written, in the first line of the script. For example, the following line specifies that the script including this line is for the C shell.

```
#!  /bin/csh
```

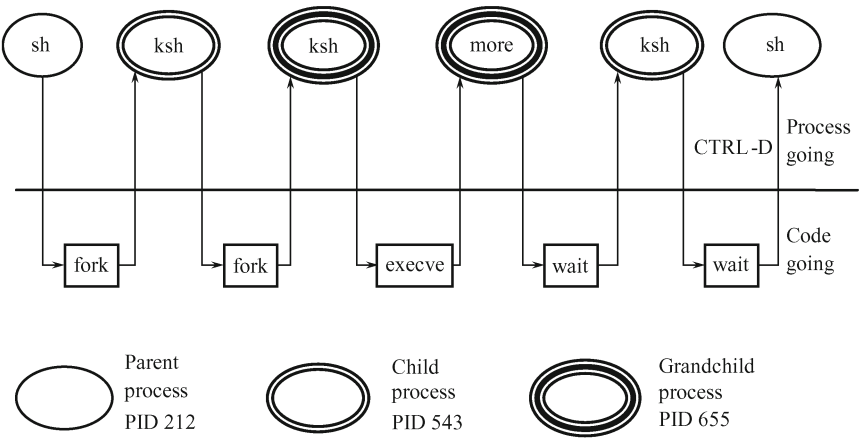

Fig. 4.8 Steps of shifting the shells.

4.5 Process Control

In an operating system, process control has several tasks to do, including process creation, process switching, process exiting, etc. Therefore the contents of the previous sections in this chapter should be part of process control. And most of these tasks relative to process control are hidden from the users of the operating system and are done by the operating system itself in order to give users the user-friendly feeling.

For users of UNIX, the UNIX operating system provides several convenient commands to interact with the UNIX kernel in order to control processes, such as process creation, executing processes in the foreground and

background, moving processes between foreground and background, suspending processes, process termination, and so on. In this section, some of these commands will be discussed.

4.5.1 Running Command in Foreground or in Background

Most UNIX operating systems let users run more than one program at the same terminal. This is called job control. However, even in a window system, the user can use job control to do several tasks at the same terminal window. This is also called multitasking. On a single-tasking operating system, such as MS-DOS, the user enters a command and must wait for the command finishing and the system prompt return, and then can enter the next command. In UNIX, the user can get another shell prompt immediately and enter the next command in the foreground right after putting the first command in the background. Even better, the user can put several jobs in the background and at the same time, type in new commands on the command line. The reason for this is that UNIX can do many jobs concurrently by dividing the processor's time between the jobs so quickly that it looks as if everything is running at the same time.

A UNIX user can use this mechanism to switch between several jobs from the terminal.

Usually, when the user types in a command and Enter, the shell executes the command and returns by displaying the shell prompt. While the command executes, the shell prompt can not appear until the command finishes. So the user cannot execute the next command until the current command finishes and the shell prompt appears on the screen, which means this command is running in the foreground. Obviously, when a command executes in the foreground, it takes input from the keyboard and sends output to the display screen.

If some command running needs a lot of time, it is annoying for a user to wait for its finishing without doing anything. UNIX operating systems allow the user to get the shell prompt back and run other tasks while executing a time-consuming command. This capability is to execute the command in the background. Usually, put the commands that are CPU-time-consuming and without I/O operation in the background, such as find (some files in a huge directory) and sort (lines in a huge file). Commands that involve a lot of user interventions, such as the vi editor, are not good for background execution. The reason is that such a process stops when it reads input from the keyboard. If the foreground is used to do some job and at the same time, some other job is running in the background, the background process may generate output sending to the display screen and hence, the screen may be interrupted. But this distortion cannot influence the foreground job.

One way to run a command in the background is simply put an ampersand

(&) at the end of the command, like

```
$ sort longfile > longfile.sorted &
[1] 18799
$
```

The sort command is used as the perfect example if the sorted file is huge. The output redirection operator > sends the output of the command of the sort longfile into the file longfiel.sorted rather than the standard input — the screen. The I/O redirection will be discussed in Chapter 7. The line shown after the executed command before the new shell prompt is returned by the shell. The number shown in brackets [1] is the job number for this sorting process; the number 18799 is its PID. The PID can be used to check the status of a background process, or to cancel it. The PID numbers do not need to remember because UNIX operating systems have commands to check on the running process on the system (see them later in this chapter).

A job is a process that is executed in the background or suspended (not running in the foreground). A job is related to the terminal where it is put in the background or suspected, so it can be accessed only at that terminal. Background processes usually have larger nice values and thus, lower priorities. Of course, they get to use the CPU only when no higher priority process needs it.

Another way to run a command in the background is to use the bg command. To support for a user to switch between several jobs from the terminal, the UNIX operating system provides several other commands along with bg. Here, some of them are introduced.

4.5.1.1 Putting Process in Background

To move the foreground and suspended processes to the background, use the bg command.

The syntax and function of bg command are as follows.

```
$ bg [% job[s]]
```

Function: to resume execution of suspended processes (or jobs) with job numbers in the background or to put the current process to execute in the background.

Common values for %job:

% or %+: to put the current job in the background;

%-: to put the previous job in the background;

% "cname": to put the job whose command name is cname in the background;

%?name: to put the command whose command line contains "name" in the background;

%n: to put the job whose job number is n in the background.

Note: The bg command without an argument resumes execution of the current process in the background. The job using the CPU at the current time is called the current job.

4.5.1.2 Bringing Process to Foreground

To move the background processes to the foreground or to resume the suspended processes, use the fg command.

The syntax and function of fg command are as follows.

```
$ fg [% job[s]]
```

Function: to move background processes (or jobs) with job numbers to the foreground or resume execution of suspended processes (or jobs) with job numbers to the foreground.

Common values for %job:

% or %+: to move the current job to the foreground;

%-: to move the previous job to the foreground;

% "cname": to move the job whose command name is cname to the foreground;

%?na: to move the command whose command line contains "na" to the foreground;

%n: to move the job whose job number is n to the foreground.

Note: The fg command without an argument brings the current process to the foreground.

The user can use the suspend keystroke command (usually CTRL-Z) to suspend a program running in the foreground. The program pauses, and the user gets a new shell prompt and can do anything else. And then the user can put the suspended program into the background by using the bg command and can also bring a suspended or background process to the foreground by using the fg command.

4.5.1.3 Displaying Status of Processes in Background or Suspended

To display the status of the background and suspended processes, use the jobs command.

The syntax and function of jobs command are as follows.

```
$ jobs [option(s)] [% job[s]]
```

Function: to display the status of the background and suspended processes (or jobs) with job numbers.

Common options:

-l: to display the process status and PID.

Note: The jobs command without an argument displays the status of current job.

If there are multiple suspended processes and several background processes, the fg command without an argument brings the current process into the foreground, and the bg command without an argument resumes execution of the current process in the background. The jobs command can be used to display the job numbers of all suspended (stopped) and background processes, and the current process and the previous process. The process with a + is the current process and the one with a – is the previous process.

To show how these commands to work, here give some examples. The following sequence of commands can be typed in on the keyboard constantly by a user, but are here broken down into several pieces in order to explain some of them nearly and clearly.

```
$ sort longfile > longfile.sorted &
[1] 18799
$ find . \(-name file1 -o -name '*.c' \) -print > mf&
[2] 18800
```

The first command is a copy of the example at the beginning of this section. The second command works in this way: to search files that are named file1 or have names ended with ".c" in the work directory and send the output into the file mf, and put the command in the background. In a UNIX command, options can be grouped together logically. -a or a space represents logically AND, and -o substitutes logically OR. Here the option -o is used. Also in a UNIX command, a complex expression can be enclosed in parentheses, \(and \). Continue.

```
$ sort encyclopedia > encyclopedia.st 2> /dev/null &
[3] 18801
```

The third command has sorted another huge file named encyclopedia, sent the sorting result into the file encyclopedia.st, and would send the error messages into the file /dev/null if there were errors while the command running. We will discuss in Chapter 7 that stderr is the standard error file and its standard file descriptor is 2. So 2> /dev/null means make the standard error redirect to the file /dev/null. The file /dev/null is a special UNIX file where anything that goes into will not go out.

```
$ ps -a
PID   TT TIME CMD
18779 U0 0:10 sort longfile > longfile.sorted
18800 U0 0:06 find . \(-name file1 -o -name '*.c' \) -print > mf
18801 U0 0:02 sort encyclopedia > encyclopedia.st 2> /dev/null
18802 U0 0:02 ps
```

Remember that the ps command with the -a option displays the information of the executing processes in the system except the session leader (see section 4.3.2). The three processes are just put in the background. Continue.

```
$ jobs
[1] Running sort longfile > longfile.sorted&
[2] - Running find . \(-name file1 -o -name '*.c' \) -print >mf&
[3] + Running sort encyclopedia > encyclopedia.st 2> /dev/null&
```

Check out the status of processes in the system. The process with a + is the current process and the one with a – is the previous process. Processes executing in the background are in the running state (shown as Running). Continue.

```
$ vi ex4.c
...       /*editing the file ex4.c*/
...
CTRL-Z
```

```
[4] + stopped (SIGTSTP) vi ex4.c
```

This time, the user uses the vi editor to edit the file ex4.c at the foreground for a while. Then suspend the vi program by pressing CTRL-Z. The final line shows that the system displays the current job status. Continue.

```
$ fg %1
sort longfile > longfile.sorted
CTRL-Z
[1] + stopped (SIGTSTP) sort longfile > longfile.sorted
```

The user moves the job with job number 1 to the foreground by using fg with %1 and then suspend it by pressing CTRL-Z. Continue.

```
$ jobs -l
[1] + 18779 Stopped (SIGTSTP) sort longfile > longfile.sorted
[2] - 18800 Running find . \(-name file1 -o -name '*.c'\) -print > mf&
[3] 18801 Running sort encyclopedia > encyclopedia.st 2> /dev/null&
[4] 18834 Stopped (SIGTSTP) vi ex4.c
```

The jobs command with option -l displays the process status and PID. Processes executing in the background are in the running state and the suspended processes are in the stopped state (shown as Stopped; SIGTSTP field indicates the reason why the process to stop, and here the reason is the process received the stop signal of the user's pressing CTRL-Z). See more information about signals in Sections 4.5.4 and 4.5.5. There are two processes suspended, which are processes with job number 1 and 4. Keep going.

```
$ fg %"vi ex4.c"
...       /*continue to edit the ex4.c file*/
...
:q!
$
```

This time, the user brings back the vi editor to the foreground by using the string "vi ex4.c" instead of the job number, finishes editing ex4.c file, and then quits the vi editor.

4.5.2 More Concepts about Process Concurrently Execution in UNIX

In addition to the multitasking and multiprogramming concepts discussed in section 4.1, the UNIX operating system gives more concepts associated with the process concurrently execution in other aspects. For example, group and execute several commands in different ways to realize running them in different orders and modes. These modes are rough divided into three according to how the kernel to handle the commands. A detailed discussion about them will be given in the following.

Concurrently execution: Type in commands in a single command line and make every command end with an ampersand (&).

The syntax for concurrently execution is

```
$ command1 & command2 & command3 & ...
```

Function: to make the kernel execute all the commands: command1, command2, command3, etc., concurrently.

Note: The spaces before or after the commands are not necessary, but they can make the command line look neat. This command line can make the kernel fork each command a child process, respectively, so the kernel can schedule the execution between them. For example:

```
$ sort longfile > longfile.sorted & find . -name file1 -print > mf&
[1] 18799
[2] 18800
$
```

The sort and find commands execute concurrently in the background. The job number and PID of the sort command are 1 and 18779; the job number and PID of the find command are 2 and 18800, respectively. The order how the kernel to schedule them and the amount of time each takes to execute are dependent on the current situation of the system and the scheduler.

Sequentially execution: Put all the commands in a single command line.

The syntax for sequentially execution is

```
$ command1; command2; command3; ...
```

Function: to make the kernel execute all the commands: command1, command2, command3, etc., one by one, in sequence.

Note: The spaces before or after the commands are not necessary, but they can make the command line clearer. And the last command is followed without a semicolon. This command line can make the kernel fork a subshell (or a child process) for one command and execute it, and then fork a subshell (or a child process) for the next command and execute it, in this way, until all the commands are finished. For example:

```
$ echo "Welcome to UNIX World!"; echo "Enjoy using UNIX!"; pwd; date
Welcome to UNIX World!
Enjoy using UNIX!
/users/project/wang
Wed Aug 10 11:20:56 CST 2008
$
```

Two echo commands, pwd and date commands execute in sequence in the foreground.

Grouped execution: Put several commands in a group as one process by enclosing them in parentheses, and separating them with semicolons.

The syntax for grouped execution is

```
$ (command1; command2; command3; ...)
```

Function: to make the kernel fork one subshell for all the commands (command1, command2, command3, etc.) and execute them in sequence.

Note: This command line can make the kernel fork one subshell (or a child process) for all the commands and execute them in sequence until all the commands are finished and come back the original shell. For example:

```
$ (echo "Welcome!"; echo "Enjoy using UNIX!"; pwd); date
Welcome!
Enjoy using UNIX!
/users/project/wang
Wed Aug 10 11:20:56 CST 2008
$
```

In this command line, two echo commands and pwd are grouped into one process to execute in sequence and the date command is executed in another process in the foreground. And the two processes are executed in sequentially execution mode.

In the following example, several commands are executed in a group and put in the background.

```
$ (sort lfile>lfile.sorted; find . -name file1 -print>mf) & date
[1] 18799
Wed Aug 10 13:10:16 CST 2008
$
```

This time, the sort and find commands use the same process with the job number 1, and the date command is executed in another process and displays its result on the screen, firstly.

Command groups can also be nested, such as ((echo “Welcome!”; echo “Enjoy using UNIX!”); (pwd; date)).

In addition, for most of the shells, to put several commands in the background at the same time, just enclose the command sequence in parentheses before adding the ampersand at the end; in the C shell, put semicolons (;) in between the commands and put the & at the end of the command line. That is

```
(command1 command2 command3) &        for most of shells
(command1; command2; command3) &        for the C shell
```

So the following syntax can work in all shells:

```
(command1; command2; command3) &
```

As (command1; command2; command3) & can put several commands in the background at the same time, the bg command can move multiple suspended jobs into the background at the same time, too. For example, the bg %1%4 command can be used to move jobs 1 and 4 into the background.

4.5.3 UNIX Inter-Process Communication

In UNIX, there are two types of inter-process communication (IPC) for the local terminal: one is the asynchronous signaling of events; the other is the synchronous transmission of messages between processes. We will discuss the

former in the next section. In this section, the synchronous mechanisms are introduced.

The synchronous mechanisms, which originate from the UNIX System V, include messages, shared memory, and semaphores (Bach 2006; Leffler et al 1983; McKusick 1999; McKusick et al 2005; Quartrman et al 1985; Thompson 1978). All the three mechanisms adopt some common features: an array, data structures, several operations, and the strategy of permission checking, getting one entry before using it, and removing one entry when it is no use any more (which simulates the open and close system calls in the file system).

4.5.3.1 Messages

The kernel manages the messages in several linked queues. Each queue has a unique descriptor that can be used to index into an array of message queue structures. Message mechanism is operated with four system calls: msgget, msgctl, msgsnd, and msgrcv.

The message queue structure stores some information about the queue, including the current number of messages on the queue, the current and upper-limit numbers of bytes on the queue, the pointers of the fore and aft messages on the queue, the PIDs of the last message-sending and -receiving processes, and the last times of msgsnd, msgrcv, and msgctl operations.

When a user process wants to communicate with other processes via the messages, it needs to start from the msgget system call. If it is an existent message queue, msgget returns its descriptor; if not, msgget creates a new one for the user process and returns its descriptor.

Once the message queue is established, the message-sending process can use the msgsnd system call to send its messages while the message-receiving process can receive the messages via the msgrcv system call.

No matter the messages are sent or received, a data buffer that can hold a character array is needed.

In the msgsnd system call, after passing several condition tests, the kernel allocates the buffer space, copies the message data from the user space to the buffer, puts the message header that records the type, size, and location of the message at the end of the queue, and updates the message queue structure, such as increasing the current number of messages on the queue, the current number of bytes on the queue, and the last PID and last time of msgsnd. And the kernel wakes up the processes that are sleeping for messages on the queue. If the queue is full, the message-sending process will sleep until the messages on the queue are removed.

In the msgrcv, if all the tests are successful, the kernel finds the first message on the queue, copies the message to the user data space, updates the message queue structure, such as reducing the current number of messages on the queue, the current number of bytes on the queue, and the last PID and last time of the msgrcv, and frees the buffer space that stored the message. If some processes sleeping because of no room for more messages to put in,

the kernel wakes them up.

The msgctl system call can be used to check and set the states of a message descriptor, and remove a message descriptor (which is just like the close system call in the file system).

When necessary, the permission verification is done by each of msgget, msgsnd, and msgrcv.

4.5.3.2 Shared memory

The kernel can make part of the virtual memory space shared by several processes. With the shared memory, processes can read and write the data in it in order to communicate with each other. Like message mechanism, shared memory mechanism also has four system calls: shmget, shmat, shmdt, and shmctl.

The kernel manipulates the shared memory with the shared memory array (or table).

When a user process needs the shared memory, it should use the shmget system call first. The kernel checks the shared memory array with a key value that is given by the user process to see if or not it exists. If it does, shmget returns its descriptor. If not, and the size that the user wants is appropriate, the kernel creates a new region for it with some memory system calls and records its information in the shared memory array entry, such as access permission, size, and a pointer to the memory region table entry. The kernel marks the region table entry as shared memory in order not to remove before the last process sharing it exits.

After getting the shared memory, the process should attach it to its virtual memory space via the shmat system call. The shmat system call can attach the shared memory with its descriptor to the address that the user wants to attach, specify its access permission, and finally return the virtual address where the shared memory is attached. The attached address should be careful to specify in order to avoid overlapping other regions of the virtual memory space, especially when the data region or stack grows. This returned address can be used later on when the process does not use this shared memory again the kernel detaches it. If the calling process is the first to attach the region, the kernel allocates memory page tables and others, which shows that part of virtual memory is really allocated.

Once the shared memory is attached to the virtual memory space of a process, the shared memory becomes part of the virtual memory of the process, and the process can read and write it with the same way to read and write the regular memory.

When a shared memory is out of use for a process, the process can detach the shared memory via the shmdt system call with the virtual memory address that is returned by the previous shmat system call.

The shmctl system call can be used to check or change the states of shared memory descriptors, and also to remove a shared memory. However before doing removal, the kernel checks if there is a process that attaches to the

shared memory. If not, the kernel will free the shared memory entry, the memory region table entry, and the memory region. It is just like what the unlink system call does on the file system.

4.5.3.3 Semaphores

In UNIX, the semaphore mechanism is the adjusted Dijkstra EW's Dekker algorithm in which a semaphore is an integer variant with two atomic operations: P and V operations (Dijkstra 1968-1; Dijkstra 1968-2; Kosaraju 1973; Lauesen 1975). The atomicity of an operation means that if it does, the operation must do all its actions successfully; if it cannot, the operation does not do anything. In the Dekker algorithm, the P operation decrements the value of semaphore if it is greater than zero while the V operation increments the value. With the atomicity, a P or V operation can mostly accomplish its action on a semaphore at any time.

In UNIX System V, the semaphore mechanism makes some adjustment. The operation atomicity is retained. However, the added or subtracted value of operations at a time can be greater than 1. And even more, processes can do multiple semaphore operations simultaneously to avoid the deadlock problems when several processes compete for a number of different resources simultaneously.

In UNIX, a semaphore consists of some fields, including the semaphore value, the last process ID to operate the semaphore, the number of processes in the process waiting queue for the semaphore value to increase, and the number of processes in the process waiting queue for the semaphore value to become zero.

There are three semaphore system calls: semget, semop, and semctl. And also there is a semaphore structure table. In one entry of the semaphore structure table is a pointer to an array of semaphores that can be used to request several system resources in one task.

Similar to messages and shared memory, the semget system call is called by the user processes when the users need to communicate each other with the semaphore mechanism in order to ask for some system resource. The semget system call can allocate an entry in the semaphore structure table with a specified number of elements in the semaphore array and return a unique descriptor that can be used by semop and semctl system calls.

With the semaphore descriptor as one of its arguments, the semop system call can be invoked to operate on the semaphores by processes. Another argument of the semop is a pointer to an array of semaphore operations. The data structure of the semaphore operation array is composed of several fields, such as the index number of the operated semaphore in the semaphore array, the operation number, and flags.

The kernel reads the list of semaphore operations from the user virtual memory space, tests their verification, and does the changes of several semaphore values if it passes the test. For atomicity, if the kernel is blocked in the middle of the list of operations, it must restore semaphores' original

values; when it is waked up again, the kernel can read the operation array from the user virtual memory space again and does its semaphore job from the very beginning again.

The value of the operation number is designed to determine the action on the semaphore value. There are several possible situations: if the value of the operation number is positive, the kernel increments the semaphore value and awakes all processes in the process waiting queue for the semaphore value to increase; if zero, and the semaphore value is also zero, the kernel continues the following operations in the operation array; if zero, but the semaphore value is not zero, the kernel increments the number of processes in the process waiting queue for the semaphore value to become; if negative, and its abstract value is equal to the semaphore value, the kernel zeros the semaphore value and awakens all processes in the process waiting queue for the semaphore value to become zero; if negative, and its abstract value is less than the semaphore value, the kernel subtracts the abstract value of operation number from the semaphore value; if negative, and its abstract value is greater than the semaphore value, the kernel puts the process to the process waiting queue for the semaphore value to increase.

Sometimes, a sudden termination of a process that has locked some resource with a semaphore can cause that the process would never reset the semaphore via a system call to unlock the resource. To avoid this problem, the flags of the semop system call can be set as SEM_UNDO, which means that the kernel takes records on the semaphore operations when the process makes the semaphore value changes. If the process terminates in half way, the kernel will restore the actions with these records.

The semctl system call can be used to check or set the control information of a semaphore, read or set the semaphore value, and also remove a semaphore with its descriptor. Before removing a semaphore, the kernel restores the semaphore value and awakens the processes in the process waiting queues. And once the processes resume their execution, they will find the semaphore is not existent and return an error.

4.5.4 UNIX Signals

The signals can help practice the asynchronous communication between two processes. As mentioned in Section 4.3.1, the UNIX kernel maintains a process through two data structures, the process table and the user structure. Every process has an entry in the kernel process table, and is allocated a user structure that contains private data. In the process table entry, there are fields that hold signals that can be used between two independent processes.

In other words, in UNIX, processes can communicate through signals (also known as software interrupts). Signals can make processes notice the occurrence of events. A process can send a signal to another process. A process

can tell also the system what it wants to happen through a signal. To send a signal to a process, the kernel sets a bit in the signal field of the process table entry, corresponding to the type of signal received. For example, if a process receives a hangup signal and a kill signal, the kernel sets the appropriate bits in the process table signal fields. If the process is asleep, the kernel awakens it. Then the job of the sender (a process or the kernel) is complete. A process can remember different types of signals. The kernel checks for receipt of a signal and handles it for a process when the process is about to return from kernel mode to user mode.

In different visions of UNIX, there can be different numbers of signals. Some signals that are caused by a process itself are known as internal signals, and other signals caused by events external to a process are called external signals. They can be rough classified as follows.

- Signals relative to the termination of a process, such as when a process exits, or when a process invokes the kill system call to terminate its child process.
- Signals relative to exceptions, such as, when a process accesses an address without the privilege, or when an unexpected error condition happens during a system call.
- Signals caused by terminal interaction, such as when a user hangs up a terminal, or when a user presses the Break or Delete keys on a terminal keyboard.
- Signals for tracing execution of a process.

When a signal arrives, a process can take one of three actions:

- To accept the signal and terminate the process (the default for most signals).
- To ignore the signal and do nothing with the process.
- To catch the signal and switch a user-defined handler. As it is treated as a software interrupt, the process must have a signal handling procedure. When a signal arrives, the system will switch to a handler. When the handler is finished and returns, the control goes back to where it interrupted, similar to hardware I/O interrupts.

In the Bourne shell, a user-defined handler can be specified in a shell script by using the trap command. In the C shell, the onintr instruction can do the similar job. In a C program, it can be done by using the signal system call. Usually, C libraries in UNIX provide a signal program who makes the signal system call. Programmers can link it into their own code if necessary.

The kill command can be used to send any type of signal to a process. In the next section, the kill command and other related commands will be introduced.

4.5.5 Termination of Processes

As known, a process can terminate normally when it finishes its work and exits while it can also terminate abnormally because of an exception (error) condition or intervention by its owner or the superuser. The owner of the process can use a command or a particular keystroke (such as CTRL-C) to terminate the process. The command that can terminate a process is the kill command.

Sometimes it is necessary to terminate a process abnormally. For example, a user does a copy of a directory recursively, but he gives a wrong pathname. Or, if the copy command seems to be taking a long time, the user can stop cp with CTRL-Z and check out the filesystem. If no problem, resume the cp command by using bg. If something wrong happens, the user can terminate the cp command by using the fg command first and then pressing CTRL-C.

It needs different ways to terminate a background process from to cancel a foreground process. Here, list the ways used to terminate a process for different situations.

- To terminate a foreground process: press CTRL-C when it running.
- To terminate a background process: bring it back by using the fg command and them press CTRL-C when it running.
- To terminate a background process: check out the job number or PID of the process by using the ps command and then terminate it by using the kill command with its job number or PID.

4.5.5.1 Canceling a Process

To cancel a process, use the kill command. Otherwise, the primary purpose of the kill command is to send a signal to a process.

The syntax and function of kill command are as follows.

```
$ kill [signal] [process[s]]
```

Function: to send the signal represented by [signal] to processes whose PIDs or job numbers that must start with % are specified in 'process[s]'.

Common values for signal:

1: to generate the hangup signal;
2: to generate the interrupt signal of CTRL-C;
3: to generate the quit signal of CTRL-\;
9: to sure kill;
15: to generate software signal (default signal number);
18: to generate the stop signal CTRL-Z.

```
$ kill -l
```

Function: to return a list of all signals and their names (depend on the UNIX versions).

Note: The kill command without any signal number sends signal number

15 to the process whose PID is specified as an argument. 15 is the default signal number, which can cause termination of the specified processes. The hangup signal with number 1 is often generated when the user logs out, by pressing CTRL-D after a shell prompt. The interrupt signal with number 2 is usually stimulated when the user pressing CTRL-C. The quit signal with number 3 is generated when the user pressing CTRL-\. The stop signal with number 18 is stimulated when the user pressing CTRL-Z. There are some other signals. Readers can check out by using the man command.

Remember, in Section 4.5.1, the display result of the jobs command showed that the job 1 of sort and the job 4 of vi were stopped by pressing CTRL-Z have been identified with the stop signal SIGTSTP.

As mentioned in the previous section, the process receiving the signal may ignore the signal or catches the signal to switch a user-defined handler. To make sure termination of a process, kill the process with signal number 9, known as sure kill. If being the owner of all the processes, a user can terminate them by using the command kill 9 with arguments of PIDs of these processes. But be careful not to terminate the shell process that is used to log in UNIX. Here, give some examples to show how to use the kill command.

```
$ ps
PID    TT   TIME   CMD
15678  U0   0:03   -sh
18779  U0   3:10   sort longfile > longfile.sorted
18800  U0   5:06   find . \(-name file1 -o -name ‘*.c’ \) -print > mf
18801  U0   10:02  sort encyclopedia > encyclopedia.st 2> /dev/null
18802  U0   0:02   ps
$ kill 18779
[1] + Killed   sort longfile > longfile.sorted &
$
```

This kill command terminates the process with PID 18779.

```
$ kill -2 %3
[3] - Killed   sort encyclopedia > encyclopedia.st 2> /dev/null &
$
```

This kill command sends the signal 2 (CTRL-C) to the process with job number 3.

```
$ kill -9 %2
[2] + Killed   find . \(-name file1 -o -name ‘*.c’ \) -print>mf&
$
```

The last kill command terminates surely the process with job number 2. The first and third commands can also combine into one command, like the following:

```
$ kill -9 %1 %2
```

4.5.5.2 Waiting for Specified Period of Time

Sometimes, a process or command needs to put off for a period of time to start. The sleep command can be used to do this job.

The syntax and function of sleep command are as follows.

```
$ sleep sec
```

Function: to create a process that makes the system time vanish for "sec" seconds.

Common options: None.

Note: If sleep is used in a shell script, the process of the script is put in the sleeping state for specified time.

For example, create a process running in the background that reminds the user to wake up in 15 minutes:

```
$ (sleep 900; echo "Wake up!") &
```

In the following example, the sleep command simply causes a process to go to sleep for 60 seconds.

```
$ (sleep 60; pwd) &
[1] 19011
$ ps
PID    TT  TIME  CMD
15678  U0  0:03  -sh
19011  U0  0:00  sleep
19012  U0  0:02  ps
$ kill 19011
[1] + Killed   sleep 60 &
/users/project/wang
$
```

As the sleep 60 command is terminated ahead of schedule, the following command pwd is executed immediately and puts its output on the screen.

4.5.5.3 Ignoring Hangup Signal

Usually, if a user logs out UNIX, the processes of this user will receive a hangup signal (signal number 1) and be terminated. If a user wants to log out and wants some processes to keep running after logout because these processes need more time to finish without any user intervention, the user can use the nohup command to do this job.

The syntax and function of nohup command are as follows.

```
$ nohup command(s)
```

Function: to run the command, ignoring the hangup signal.

Common options: None.

Note: If nohup is programmed in a shell script to specify a process and the script is executed, the process can be being executed after the user has logged out. A nohup command can be used for several commands separated by semicolons.

The nohup command can be used for processes that take a long time to finish, such as to find some files in a huge directory, or to sort some big file. The following command can make the sort command run ignoring the hangup signal. Hence, when the user logs out, the sort command will run until finish. If the output of the sort command is not redirected, it will be appended to the nohup.out file by default.

For example:

```
$ nohup sort encyclopedia > encyclopedia.st &
[1] 19022
$
```

The following example uses one nohup for two commands.

```
$ nohup sort longfile > longfile.st; find . -name '*.c' -print >mf&
[2] 19023
$
```

4.5.6 Daemons — UNIX Background "Guardian Spirits"

In the beginning of this chapter, the term of daemon has been mentioned, but has not been explained yet.

Processes in UNIX can be divided into three groups: user processes, daemon processes, and kernel processes. Most processes are usually user processes, related to users at terminals. Daemon processes do not belong to any users but do system facilities (Bach 2006; Sarwar et al 2006), such as a printer spooling, control and management of networks, executions of time-dependent administrations, and so on. Some daemon processes can exist throughout the lifetime of the system while other daemon processes can be invoked by user processes to make system calls in order to access system services. Like daemon processes, kernel processes provide system services, too. Otherwise, as part of the kernel, kernel processes can have more control and access kernel algorithms and data structures directly without using system calls. However, considering programming and compiling, kernel processes are not as flexible as daemon processes because to change the kernel code, the whole UNIX kernel must be changed and recompiled.

In UNIX, a daemon is often placed in the background and is activated only when needed. Hence, even if the user is absent, in a UNIX operating system, many daemons can be running in the background, which may have been started automatically at UNIX system boot time.

Daemons are frequently used to provide various types of system utility services to users and handle system administration tasks. For example, when a program hits an error and cannot recover from the error, an error-recovering daemon can be activated to correct the error and recover the system.

The finger services are handled by the finger daemon, fingerd. Finger is an Internet utility that enables a user to obtain information of other users who may be at other sites (if those sites permit access by finger). For example, if given an e-mail address, finger returns the user's full name, an indication of whether or not the user currently logs in, and any other information the user has chosen to supply as a profile. Or if given a first or last name, finger returns the logon names of users whose first or last names match.

The inetd, commonly known as the UNIX superserver, handles various

Internet-related services by creating several daemons when the UNIX operating system is booted. Physically, a superserver is usually a network server with especially high capabilities for speed and data storage. Here, the UNIX superserver means high capabilities of Internet utility services in UNIX.

The printing services are usually provided by the printer spooling daemon.

Usually, daemons are hidden from the UNIX users. For readers, it is useful to know UNIX daemons in order to learn them, if necessary, more deeply in future.

4.6 UNIX System Boot and Init Process

As mentioned in Section 4.2.2, the UNIX operating system has two unique processes, Process 0 and 1. Process 0 is the process that is created when the system boots. Process 0 produces Process 1, the init process; all other processes in the system have Process 1 as a parent process.

Different versions of UNIX can be booted slightly differently, but some common tasks are necessary for any UNIX operating system. Here, give a brief review of UNIX system booting.

To initialize a computer system from startup or reset, the first sector of the boot disk – the boot block (block 0) (see Section 6.5.1) is read and loaded into memory and executed. The program in the boot block loads the kernel from the file /unix (see Section 6.1.1). After the kernel is loaded in memory, the boot program moves control to the start address of the kernel, and the kernel starts running.

The kernel does initialization, including building up its internal data structures in memory. After finishing the initialization, it mounts the root file system onto root (/) in memory and creates environment for process 0, such as initializing the process table entry 0, assigning a user structure, having root as the working directory of process 0, and so on.

Then, the system begins to do configuration for I/O devices, exploring the devices to see which ones actually are present and to add the present devices to a table of attached devices.

At this time, the system is running as process 0. Process 0 forks process 1. Process 1 does the execve system call and is replaced by init program (from the file /etc/init), which is responsible for initialization of new processes.

The init process creates processes that allow users to log into the system, etc. Init reads the file /etc/inittab to check out a state identifier (marking single user, multi-user, and so on) along with other information. If the multi-user state is marked, init forks a process to execute the /etc/rc program, which can do file system consistency checks. It also reads /etc/ttys, which lists the terminals and some of their attributes. For each active terminal, it forks a getty process (from the file /etc/getty), which is used to monitor the terminals. Then, getty sets the terminal properties according to the file

/etc/termcap and prompts the following information on terminal screens:

```
Login:
Password:
```

When a user types in the user name and password from the keyboard, getty verifies the password against the encrypted password in the /etc/passwd file. If it is correct, getty executes /bin/login, the login program, and do the execve system call for a login shell. Meanwhile, init executes the wait system call, monitoring the termination of its child processes (see Section 4.4.1: fork system call).

The init may also spawns some daemon processes that exist throughout the lifetime of the system.

Later on, the process 0 creates other kernel processes, and then becomes a swapper process.

Once in the login shell, the user can do any work and terminate the shell by pressing CTRL-D. When doing so, the shell process terminates and control goes back to the getty process, by displaying the login: prompt again.

Figure 4.9 illustrates the UNIX system boot and processes' creation.

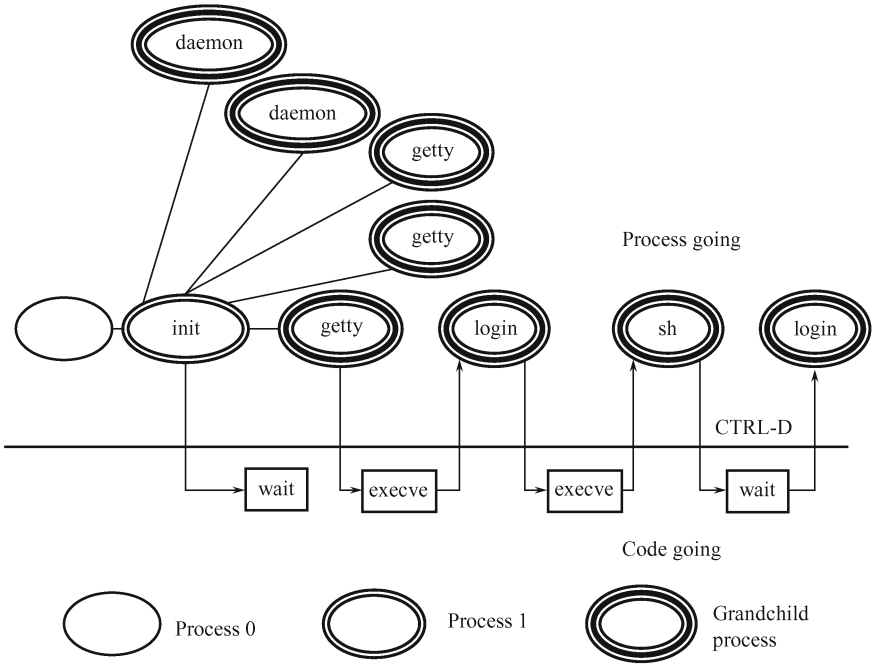

Fig. 4.9 UNIX system boot and processes' creation.

Therefore, in UNIX, the swapper and init processes have the lifetime of the system from system boot to turning off the computer system. The getty process lives as long as the terminal attached to the system. All other

processes are as long as a command or program's execution.

For the whole system, there is always a certain process running in the system from its booting to its power off. That is, during the session, the CPU keeps running for a certain process, no matter the process belongs to some user or the kernel. Thus, in the operating system, there is always some process being executed without stop from the system booting until the system is terminated. Therefore, we can even see in this way that there is a large task occupying the CPU. It represents user A now, and does work for user B then, and does the kernel work sometime later on. But each of them is just one part of the large task. When switching from one job to another, the kernel does context switch for them to make different processes execute in their own appropriate environments, accomplish their own execution in a correct way and give the desired results even though each of them does their own execution intermittently.

Now, remember the kill command. As Process ID 0 creates all the processes after login, the command of kill 0 can terminates all processes resulting from the current login.

To see the process tree of a running process in a graphical form, use the ptree command. To learn how to use it, check out this command on the system by using man -s1 ptree.

4.7 Summary

As only one CPU is available in an uniprocesssor system, multiple processes' running concurrently allows the processor to handle multiple batch jobs at a time, but also be used to handle multiple interactive jobs. It is the scheduler to make the choice which process to run next. The algorithm that the scheduler uses is called the scheduling algorithm.

The simplest of all scheduling algorithms is the first-come first-served (FCFS). With this algorithm, processes are assigned the CPU in the order they request it. In most of the time-sharing systems, the round robin (RR) scheduling algorithm is used. With this algorithm, each process is assigned a time slice to run. With the priority scheduling, each process is assigned a priority, and the ready process with the highest priority is allowed to run next.

As UNIX is a multiprogramming system, its scheduling algorithm was designed to provide good response to interactive processes. Its scheduling algorithm includes two levels: the low-level algorithm is used to schedule the processes in the memory and ready to run; the high-level algorithm is used to make the decision on which process to bring in the memory from the disk and to be ready to run. Usually, in UNIX, the low-level algorithm uses multiple queues, each of which has a unique range of non-overlapping priority values. The queue searching of the low-level scheduler is started from the highest

priority until a queue that is occupied is found. The first process on that queue is then chosen to run. The processes within the same priority range are scheduled on the FCFS basis and the round robin algorithm.

In the system, some processes might be ready to execute while others might be blocked, waiting for an I/O operation to complete. A natural way to handle this situation is to distinguish the process states in five states: running, ready, blocked, created, and terminated.

A simplified UNIX process state transition has been discussed. Differently, the UNIX has two running states, two states for ready processes, and two states for blocked processes.

To manage and control a process, UNIX must know where the process is located and other attributes of the process. These attributes are referred as a process control block (PCB). All the information, including the set of instruction code, data, stack, and attributions, can be referred as the process image. The PCB is handled by the UNIX kernel in two data structures, the process table and the user structure. The PCB is composed of both of them. From the user angle, a process in UNIX has its attributes, including owner's ID, process name, process ID (PID), process state, PID of the parent process, priority, and length of time the process has been running. Process control commands, such as executing processes in the foreground and background, moving processes between foreground and background, suspending processes, process termination, and so on, have been discussed in this chapter. The ps command is used to report process attributes.

System call fork is the only way to create a new process in UNIX. After fork, parent and child processes continue execution differently.

In UNIX, the user can get another shell prompt immediately and enter the next command in the foreground by putting the first command in the background. Users can also run several commands in one command line in different modes: concurrently execution, sequentially execution, and grouped execution modes.

UNIX processes can communicate each other synchronously with one of three mechanisms: messages, share memory and semaphores, or asynchronously through signals. The kill command can be used to send any type of signals to a process.

In UNIX, a daemon is often placed in the background and is activated only when needed. And there are two unique processes, Processes 0 and 1. Process 0 is the process that is created when the system boots and becomes the swapper later on. Process 0 produces Process 1, the init process; all other processes in the system have Process 1 as a parent process.

Problems

Problem 4.1 Try to explain multiple processes' running concurrently. What is a process? In a time-sharing system, what is a time slice?

Problem 4.2 In an operating system, what is a scheduler? And what is the scheduling algorithm used for? What is the FCFS scheduling algorithm? What is the round robin scheduling algorithm? Where is usually the round robin scheduling algorithm used? What is the priority scheduling?

Problem 4.3 What feature does a compute-bound process have, compared to an I/O-bound process?

Problem 4.4 How many levels does the UNIX scheduling algorithm include? What are they? In its low-level algorithm, how is the priority value for every process computed? What does each of the three components in the priority formula mean, respectively? How does the priority formula indicate that UNIX gives higher priority to processes that have used less CPU time in the recent past? Explain the reason by describing the computation process of the priority formula.

Problem 4.5 Why does the system administrator's process get higher priority than other users' services when the system administrator sets the nice value from -20 to -1?

Problem 4.6 Try to create an algorithm program that can accomplish the UNIX scheduling mechanism that is described in Section 4.1.2.

Problem 4.7 Please describe the five states in the five-state process model shown in Figure 4.2. What are the main differences between a simplified UNIX process state transition shown in Figure 4.3 and the five-state process model?

Problem 4.8 In UNIX, what is the process control block composed of? Where are they possibly located, respectively, when the swapping is considered?

Problem 4.9 From the user angle, what do a process' attributes indicate? How can a user view the attributes of processes running on a system? Give an example.

Problem 4.10 Assume that in a UNIX system, there is a process, called PA, with the C, NI, and PRI values of 14, 20, and 68. Try to compute the base value of this UNIX operating system. Assume there is a process, called PB, with a nice value of 20 and a CPU_usage of 22 in the same system. Use the above base value to compute the priority value of PB.

Problem 4.11 In UNIX, how can the kernel create a new process?

Problem 4.12 To wait for the child process to finish, which system call should the parent (or the shell) process execute?

Problem 4.13 When the child process has to execute a command typed after a shell prompt by a user, what can the execve system call do?

Problem 4.14 How many types of shell commands are there? What are they? How does the kernel execute them, respectively?

Problem 4.15 What is process control? What is the job control in UNIX?

Problem 4.16 If you execute a command in the foreground, can you execute the next command before the first command finishes? When you put a command in the background, what happens on the terminal screen? Give two ways to execute a command in the background.

Problem 4.17 What is the current job? How can you move the background processes to the foreground or to resume the suspended processes? How can you suspend a process that is running in the foreground? How can you put the suspended program into the background? How can you check the status of the background and suspended processes?

Problem 4.18 When a command line is typed in the way of concurrently execution, how does the kernel do for this command line? When a command line is typed in the way of sequentially execution, how does the kernel handle this command line? How does the kernel tackle a command line typed in the way of grouped execution?

Problem 4.19 What mechanisms can the processes use to make the interprocess communication in a local system?

Problem 4.20 Try to create your version of the message mechanism to accomplish the functions that have been explained in Section 4.5.3.1.

Problem 4.21 Give your version of the semaphore mechanism to accomplish the operations that have been described in Section 4.5.3.3.

Problem 4.22 In UNIX, what are signals used for? How can signals be classified? When a signal arrives, what kinds of actions can a process take? In the Bourne shell, how can a user-defined handler be specified in a shell script?

Problem 4.23 How can you terminate a process? Give several different ways for different situations.

Problem 4.24 How can a process be put off for a period of time to start?

Problem 4.25 In the UNIX operating system, what is a daemon?

Problem 4.26 In the UNIX operating system, what are Process 0 and Process 1, respectively? What does a getty process do when a user logs in or logs out the system?

References

Anderson TE, Bershad BN, Lazowska ED et al (1992) Scheduler activations: Effective kernel support for the user-level management of parallelism. ACM T Comput Syst 10(1): 53 – 79

Andrews GR (1983) Concepts and notations for concurrent programming. ACM Comput Surv 15(1): 3 – 43

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Braams J (1995) Batch class process scheduler for UNIX SVR4. SIGMETERICS'95/PERFORMANCE'95: The 1995 ACM SIGMETERICS joint international conference on measurement and modeling of computer systems, Ottawa,

Ontario. ACM, pp 301 – 302
Cmelik RF, Gehani NH, Roome WD (1989) Experience with multiple processor versions of concurrent C. IEEE T Software Eng 15(3): 335 – 344
DeBenedictis EP, Johnson SC (1993) Extending UNIX for scalable computing. Computer 26(11): 43 – 53
Denning PJ (1983) The working set model for program behavior. Commun ACM 26(1): 43 – 48
Devarakonda MV, Iyer PK (1989) Predictability of process resource usage: A measurement-based study on UNIX. IEEE T Software Eng 15(12): 1579 – 1586
Dijkstra EW (1968) Cooperating sequential processes, in programming languages. ACM, New York
Dijkstra EW (1968) The structure of the "THE"-multiprogramming system. Commun ACM 11(5): 341 – 346
Forrest S, Hofmeyr SA, Somayaji A et al (1996) A sense of self for UNIX processes. SP'96: The 1996 IEEE Symposium on Security and Privacy, Oakland, CA, 6 – 8 May 1996, pp 120 – 128
Kosaraju SR (1973) Limitations of Dijkstra's semaphore primitives and Petri nets. SOSP'73: The Fourth ACM Symposium on Operating System Principles, ACM, pp 122 – 126
Lauesen S (1975) A large semaphore based operating system. Commun ACM 18(7): 377 – 389
Leffler SJ, Fabry RS, Joy WN (1983) A 4.2BSD interprocess communication primer. UNIX Programmer's Manual, 4.2 Berkeley Software Distribution. Computer Systems Research Group, Depat. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Aug 1983
McKusick MK (1999) Twenty years of Berkeley UNIX: From AT&T-owned to freely redistributable. LINUXjunkies.org. http://www.linuxjunkies.org/articles/kirkmck.pdf. Accessed 20 Aug 2010
McKusick MK, Neville-Neil GV (2005) The design and implementation of FreeBSD operating system. Addison-Wesley, Boston
Mohay G, Zellers J (1997) Kernel and shell based applications integrity assurance. ACSAC'97: The IEEE 13th Annual Computer Security Applications Conference, 1997, pp 34 – 43
Peachey DR, Bunt RB, Williamson CL et al (1984) An experimental investigation of scheduling strategies for UNIX. SIGMETRICS: 1984 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pp 158 – 166
Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading, Massachusetts
Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17 (7): 365 – 375
Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: the textbook, 2nd edn. China Machine Press, Beijing
Sielski KL (1992) Implementing Ada tasking in a multiprocessing, multithreaded UNIX environment. The Conference on TRI-Ada'92, Orlando, Florida. ACM, 512 – 517
Stallings W (1998) Operating systems: internals and design principles, 3rd edn. Prentice Hall, Upper Saddle River, New Jersey
Thompson K (1978) UNIX implementation. Bell Syst Tech J 57(6) Part 2: 1931 – 1946
Zhou Y (2009) UNIX process, merging unified process and extreme programming to benefit software development practice. 2009 IEEE First International Workshop on Education Technology and Computer Science, Wuhan, Hubei, China, 7 – 8 March 2009, pp 699 – 702, doi 10.1109/ETCS.2009.690

5 UNIX Memory Management

In this chapter, we will focus on the memory management in UNIX, which is one of the most important services of UNIX kernel. In a computer system, CPU must cooperate with the memory to accomplish any computing. The main memory has scarce space and cannot contain all the programs on the disk. However, a process cannot execute if it is not brought in memory. Thus, the memory management becomes quite important, especially when the sizes of application programs become fairly large. And the memory management has a close relationship with the process management. We will introduce the outline of memory management, process swapping in UNIX, and demand paging in UNIX in this chapter.

5.1 Outline of Memory Management

With the development of computer hardware technology, the size of physical memory has been evolving from very small one (less than one half Mbytes, such as PDP-11) to quite large one (more than 1 Gbyte, such as HP xw4200). At the very beginning, operating system developers had to try very hard to figure out how to use the limited memory space to accomplish as many user applications as possible. To do so, many schemes to manage memory were addressed and among them, the swapping and paging were two of the most important ones and still work well in modern operating systems. The swapping mechanism was adopted in the UNIX System versions of AT&T Bell Lab at first (Ritchie et al 1974) while the paging technique was added to UNIX BSD variants of the University of California at Berkeley at the start (Quarterman et al 1985). With the size of physical memory increasing, the tension of memory space in the computer system, however, has not been eliminated. The reason is that the growing speed of the size of application programs is sometimes faster than the development rate of the physical memory.

Because the physical memory space is always limited and programs become larger and larger, memory must be carefully managed, and the memory management becomes more and more important. Typically, the memory man-

agement is responsible to allocate the portion of memory for new processes, keep track of which parts of memory are in use, deallocate parts of memory when they are unused, and manage swapping between main memory and disk and demand paging when main memory is not enough to hold all the processes.

5.1.1 Evolution of Memory Management

As in a single-process operating system only one process at a time can be running, there is just one program sharing the memory, except the operating system. The operating system may be located at the lower-addressed space of the memory and the user program at the rest part. Thus, the memory management is quite simple. That is, there is not too much work to do for the memory management in the single-process operating system. The memory management just handles how to load the program into the user memory space from the disk when a program is typed in by a user and leaves the process management to accomplish the program execution. When a new program name is typed in by the user after the first one finishes, the memory management also loads it into the same space and overwrite the first one.

In multiprocessing operating systems, there are many processes that represent different programs to execute simultaneously, which must be put in different areas of the memory. Multiprogramming increases the CPU utilization, but needs complex schemes to divide and manage the memory space for several processes in order to avoid the processes' interfering with each other when executing and make their execution just like single process executing in the system.

It may be the simplest scheme to divide the physical memory into several fixed areas with different sizes. When a task arrives, the memory management should allocate it the smallest area that is large enough to hold it and mark this area as used. When the task finishes, the management should deallocate the area and mark it as free for the later tasks. A data structure is necessary to hold the information for each size-fixed area, including its size, location and use state. Since a program has a starting address when it is executed and the initial addresses for various areas are different, the memory management should also have an address transformation mechanism to handle this issue. The two biggest disadvantages for this scheme are that the fixed sizes cannot meet the needs of the number increasing of the tasks brought in the system simultaneously and the size growing of application programs. The former problem can be handled by swapping and the latter one via paging.

Mentioned in last chapter, sometimes some processes wait for I/O devices in memory without doing anything. Otherwise, some other processes are ready to run, but there is not enough memory to hold them. Thus, operating system developers consider if the memory management can swap the

waiting-for-I/O processes out of the memory and put them in the disk temporarily to save the memory space for other processes that are ready to run. When the memory is available for the processes swapped out on the disk, the system checks which processes swapped out are ready to run and swap the ready processes in memory again. This memory management strategy is called swapping. The memory is allocated to processes dynamically. As the swapping is fast enough to let the user not to realize the delay and the system can handle more processes, the performance of the whole system becomes better. Since the program is kept in a continuous memory space, the swapping is just done on a whole process.

When the size of an application program becomes too big to load in the memory as a whole at a time to execute, the memory paging is needed. Paging technique can divide the main memory into small portions with the same size — pages, whose size can be 512 or 1024 bytes (Bach 2006; Belay et al 1969; McKusick et al 2005; Quarterman et al 1985; Rashied et al 1988; Stallings 1998). When a long program is executed, the addresses accessed over any short period of time are within an area around a locality. That is, only a number of pages of the process are necessarily loaded in main memory over a short period of time. When some page of the program is needed and not in memory yet, it is acceptable if that page is loaded in main memory fast enough when demanded, which is called demand paging. In this situation, bringing in or out of memory is with pages, rather than a whole process. Demand paging combined with page replacement and swapping implements the virtual memory management in 4.2BSD and UNIX System V (Bach 2006; Bartels et al 1999).

At the very beginning of UNIX development in the early 1970s, UNIX System versions adopted swapping as its memory management strategy. It was designed to transfer entire processes between primary memory and the disk. Swapping was quite suitable for the hardware system at that time, which had a small size memory, such as PDP-11 (whose total physical main memory was about 256 Kbytes). With swapping as the memory management scheme, the size of the physical memory space restricts the size of processes that can be running in the system. However, the swapping is easy to implement and its system overhead is quite small.

Around the second half of 1970s, with the advent of the VAX that had 512-byte pages and a number of gigabytes of virtual address space, the BSD variants of UNIX first implemented demand paging in memory management. Demand paging transfers pages instead of a whole process between the main memory and the disk. To start executing, a process is not necessarily to load in the memory as a whole, but several pages of its initial segment. During its execution, when the process references the pages that are not in memory, the kernel loads them in memory on demand. The demand paging allows a big-sized process to run in a small-sized physical memory space and more processes to execute simultaneously in a system than just swapping. Even though demand paging is more flexible than just swapping, demand paging

implementation needs the swapping technique to replace the pages.

From UNIX System V, UNIX System versions also support demand paging.

Another way used in memory management is segmentation, which divides the user program into logical segments that are corresponding to the natural length of programming and have unequal sizes from one segment to another. As it is used less in UNIX kernel, we won't discuss it deeply in this book.

5.1.2 Memory Allocation Algorithms in Swapping

With swapping, the memory is assigned to processes dynamically when a new process is created or an existent process has to swap in from the disk. The memory management system must handle it. A common method used to keep track of memory usage is linked lists. The system can maintain the allocated and free memory segments with one linked list or two separate lists. The allocated memory segments hold processes that reside currently in memory, and the free memory segments are empty holes that are in between two allocated segments.

With one list, the segment list can be sorted by address and each entry in the list can be specified as allocated or empty with a marker bit.

With two separate lists for allocated and free memory segments respectively, each entry in one of the lists holds the address and size of the segment, and two pointers. One pointer points to the next segment in the list; the other points to the last segment in the list.

With the lists of memory segments, the following algorithms can be used to allocate memory for a new process or an existent process that has to swap in. Having one mixed list, the algorithms search the only list; with two separate lists, the algorithms scan the list of free memory segments (or the free list), and after allocated, the chosen segment is transferred from the free list to the allocated list.

- First-fit algorithm. It scans the list of memory segments from the beginning until it finds the first free segment that is big enough to hold the process. The chosen free segment is tried to split into two pieces. One piece is just enough for the process. If the rest piece is greater than or equal to the minimum size of free segments, one piece is assigned to the process and the other remains free. If the rest piece is less than the minimum size of free segments, the whole segment is assigned to the process without being divided. The list entries are updated.
- Next-fit algorithm. It works almost the same way as first fit does, except that at the next time it is called to look for a free segment, it starts scanning from the place where it stopped last time. Thus, it has to record the place where it finds the free segment every time. And when the searching reaches the end of the list, it goes back to the beginning of the list and

continues.

- Best-fit algorithm. It searches the whole list, takes the smallest free segment that is enough to hold the process, assigns it to the process, and updates the list entries.
- Quick-fit algorithm. It has different lists of the memory segments and puts the segments with the same level of size into a list. For example, it may have a table with several entries, in which the first entry is a pointer to a list of 8 Kbyte segments, which are the segments whose sizes are less than or equal to 8 Kbytes; the second entry is a pointer to a list of 16 Kbyte segments, which are the segments whose sizes are less than or equal to 16 Kbytes and greater than 8 Kbytes; and so on. When called, it just searches the list of the segments with the size that is close to the requested one. After allocated, the list entries should be updated.

First-fit algorithm is simple and fast. When all the processes do not occupy all the physical memory space, Next-fit algorithm is faster than the first-fit algorithm because first-fit the algorithm does searching always from the beginning of the list (Bach 2006; Bays 1977). When the processes fill up the physical memory and some of them are swapped out on the disk, next-fit algorithm does not necessarily surpass first-fit algorithm. Best-fit algorithm is slower than first-fit and next-fit algorithms because it must search the entire list every time. Quick-fit algorithm can find a required free segment faster than other algorithms (Bach 2006; Bays 1977).

When deallocating memory, the memory management system has to do merge free segments to avoid memory split into a large number of small fragments. That is, if the neighbors of the newly deallocated segment are also free, they are merged into one bigger segment by revising the list entries.

Two separate lists for allocated and free segments can speed up all the algorithms, but make the algorithms more complicated and merging free segments when deallocating memory more costly, especially for quick-fit algorithm.

The early UNIX System versions adopted the fist-fit algorithm to carry out allocation of both main memory and swap space.

5.1.3 Page Replacement Algorithms in Demand Paging

For demand paging, only a number of pages of a process reside in memory (Braams 1995; Brawn et al 1970; Christensen et al 1970; Cmelik et al 1989; Denning 1970; Iftode et al 1999; Park et al 1996; Peachey et al 1984; Weizer et al 1969). When some page has to be referenced but not brought in memory yet, it is called a page fault. When a page fault occurs and there is no room available in memory, the memory management system has to choose a page to remove from memory to make room for the page that has to be brought in. In a virtual memory system, pages in main memory may be either clean or

dirty. If it is clean, the page has not been modified and will not be rewritten to the disk when it is removed from the memory because the disk copy is the same as the one in memory. If it is dirty, the page has been changed in memory, and must be rewritten to the disk if it is removed from the memory. Then the new page to be read will be brought in memory and overwrite the old page.

Several important replacement algorithms (Midorikawa et al 2008; Quarterman et al 1985; Ritchie et al 1974; Stallings 1998) are as follows. When considering how good a page replacement algorithm is, we can examine how frequent the thrashing happens when the algorithm used. The thrashing is the phenomenon that the page that has been just removed from memory is referenced and has to be brought in memory again (Denning 1968).

- The optimal page replacement algorithm. It is an ideal algorithm, and of no use in real systems. But traditionally, it can be used as a basis reference for other realistic algorithms. It removes the optimal page that will be referenced the last among processes currently in memory. But it is difficult and costly to look for this page.
- The first-in-first-out (FIFO) page replacement algorithm. It removes the page that is in memory for the longest time among all the processes currently residing in memory. The memory management system can use a list to maintain all pages currently in memory, and put the page that arrives the most recently at the end of the list. When a page fault happens, the first page in the list, which is the first comer, is removed. The new page is the most recent comer, so it is put at the end of the list.
- The least recently used (LRU) page replacement algorithm. It assumes that pages that have not been used for a long time in the past would remain unused for a long time in the future. Thus, when a page fault occurs, the page unused for the longest time in the past will be removed. To implement LRU paging, it is necessary to have a linked list of all pages in memory. When a page is referenced, it is put at the end of the list. Thus, for a while, the head of the list is the least recently used page, which will be chosen as the one to be removed when a page fault happens. The implementation can also be performed with hardware. One way is to equip each page in memory with a shift register, as shown in Figure 5.1.

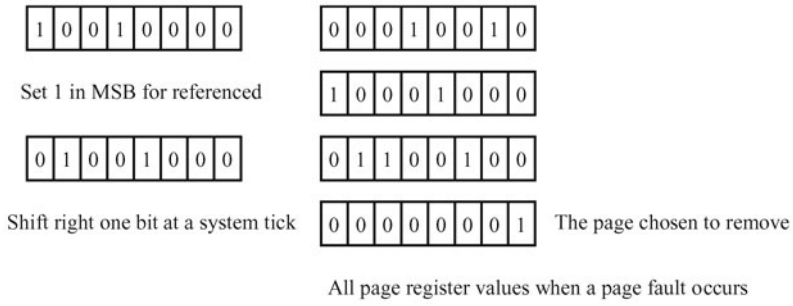

Fig. 5.1 The shift registers for the least recently used page replacement algorithm.

Every time a page is referenced, its register MSB is set. All the registers are shifted right one bit in each slice of system time. When a page fault occurs, the page with the lowest register value is chosen to remove.

- The clock page replacement algorithm. It puts all the pages in memory in a circular list and maintains the list in a clock-hand-moving order, as shown in Figure 5.2. Usually, the virtual memory in a computer system has a status bit associated with each page, for example, R, which is set when the page is referenced. If a page fault occurs, the system begins its page scanning along the clock-like list. If its R bit is zero, the page is chosen to remove, and replaced with the new page. Then the searching pointer, which works like the clock hand, moves to point to next page in the list and stops there until the next page fault. If R is one, the system clears the R bit and moves the searching pointer to the next page in the list. The pointer motion is repeated until a page with zero R bit is found. Then the system does the page replacement.

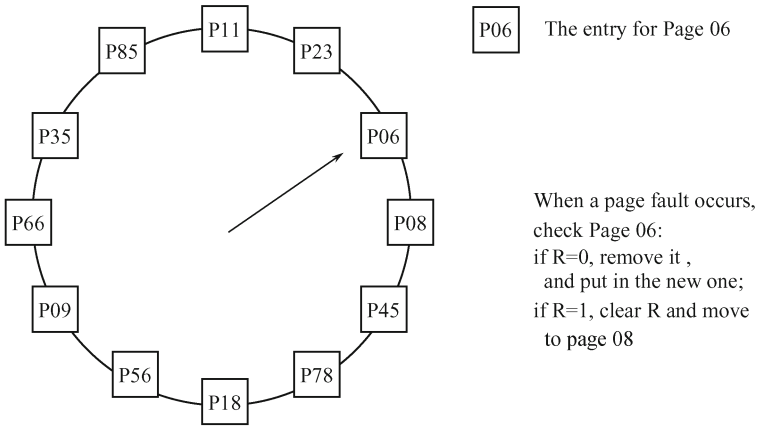

Fig. 5.2 The clock page replacement algorithm.

- The revised clock page replacement algorithm. If the virtual memory management in a computer system has two status bits associated with each page, one for reference (R) and the other for modification (M). That is, R is set when the page is referenced, and M is set when the page is modified (written or dirty). The bits can be contained in each page table entry. If the hardware does not have these bits, they can be simulated with a data structure. Pages in memory can have four cases: R=0 and M=0, in which pages are not referenced recently and not modified; R=0 and M=1, in which pages are not referenced recently but modified; R=1 and M=0, in which pages are referenced recently but not modified; R=1 and M=1, in which pages are referenced recently and modified.

Now there are four cases to choose. Obviously, the case of R=0 and M=0 is the first choice; the case of R=0 and M=1 is the second one. Like the clock

page replacement algorithm, when a page fault happens, the system begins its page scanning along the clock-like list. If both of its R and M bits are zero, the page is chosen to remove, and replaced with the new page. The searching pointer moves to point to next page in the list and stops there until the next page fault.

If the searching takes a whole circle of the clock-like list without finding one page with R=0 and M=0, it starts second searching circle. In the second searching, it does two operations: one is looking for a page with R=0 and M=1; the other is clearing R for pages with R=1. If it encounters a page with R=0 and M=1, it rewrites it to the disk and replaces it with the new page. The searching pointer moves to point to next page in the list and stops there until the next page fault.

If a page with R=0 and M=1 is not found in the second searching circle, the third searching circle has to start. Since in the second circle, R bits have been cleared, a page with R=0 must be found in the third searching circle. Then the system does the page replacement.

Compared to the clock page replacement algorithm, the revised clock page replacement algorithm allows the pages to stay in memory longer and reduce thrashing.

Because the VAX has no page reference bit in hardware memory management, the 4.2BSD that was developed on VAX used an algorithm that is slightly different from the clock page replacement algorithm (Quarterman et al 1985). The missing hardware reference bit is substituted with a field of a data structure. A software clock hand scans all pages in memory repeatedly and labels each page as invalid when it reaches the page for the first time, which is like clearing reference bit. The page can become valid again if it is accessed before the clock hand arrives at it again. But if the page has not been accessed when it is encountered again, it is replaced by a new page.

UNIX System V follows a least recently used page replacement algorithm that was implemented differently from the LRU algorithm introduced above. We will discuss it in Section 5.3.

5.2 Process Swapping in UNIX

As UNIX memory management started from swapping the whole processes out of or in memory, in this section, we will first discuss how to swap a process as a whole out of or in memory.

5.2.1 Swapped Content

We know the swapping moves a whole process between the memory and the

swap space on the disk. What does the swapped content of a process consist of?

In UNIX, since processes can execute in user mode or kernel mode, typically, the major data associated with a process consist of the instruction segment, the user data segment, and the system data segment. Except the private code, the instruction segment may include the shared code, which can be used by several processes. The user data segment includes user data and stack. The system data segment is composed of kernel data and stack. Either data or stack in both user and system segments can grow during the process executing.

The sharable code is not necessary to swap because it is only to read and there is no need to read in a piece of shared code for each process if the kernel has already brought it in memory for some process. On the other hand, multiple processes using the same code can save the memory space. In UNIX, shared code segments are treated with an extra mechanism.

Except the shared code, all the other segments, including the private code of the instruction segment, the user data segment, and the system data segment, can be swapped if necessary.

To swap easily and fast, all the segments have to keep in a contiguous area of memory. Contiguous placement of a process can cause serious external fragmentation of memory. However, in demand paging, it is not necessary to put a process in a contiguous area of memory.

5.2.2 Timing of Swapping

In UNIX kernel, the swapper is responsible for swapping processes between the memory and the swap area on the disk. The swapper is Process 0 (see Section 4.2.2).

The swapper is awaked at least once in a set slice of time (for example 4 seconds) to check whether or not there are processes to be swapped in or out. The swapper does examine the process control table to search a process that has been swapped out and ready to run. If there is free memory space available, the kernel allocates main memory space for the process, copies its segments into memory, and changes its state from ready swapped into ready in memory, and puts it in the proper priority queue to compete CPU with other processes that are also ready in memory.

If the kernel finds the system does not have enough memory to make the process to be swapped in, the swapper will examine the process control table to find a process that sleeps in memory waiting for some event happens and put it in the swap space on the disk. Then the swapper is back to search a process to swap in. The free memory space is allocated to that process.

Except when there is not enough room in memory for all the existing processes, swapping-out can also happen if one of two cases occurs: one is

some segments of a process increase and its old holder cannot accommodate the process; the other is a parent process creating a child process with a fork system call.

As known, both the user and system data segments may grow and exceed the original scope during the process execution. If there is enough memory to allocate a new memory space for the process, the allocation will be done directly by the invocation of the brk system call that can set the highest address of a process's data segment and its old holder will be freed by the kernel.

If not, the kernel does an expansion swap of the process, which includes: to allocate the swap space on the disk for the new size of the process, to modify the address mapping in the process control table according to the new size, to swap the process out on the swap space, to initiate the newly expanded space on the disk, and to modify its state as "ready, swapped". When the swapper is invoked again, the process will be swapped in memory and finally resume its execution in a new larger memory space.

When a child process is created via the fork system call, the kernel should allocate main memory space for it (see Figure 4.4). However, if there is no room in memory available, the kernel will swap out the child process onto the swap space without freeing the memory because the memory has not been allocated yet and set the child in "ready, swapped". Later on, the swapper will swap it in memory.

5.2.3 Allocation Algorithm

As mentioned before, the first-fit algorithm was adopted in UNIX to allocate memory or the swap space on the disk to processes.

In UNIX, the swap space on the disk is allocated to a process with sequential blocks. To do so is for several reasons: first, the use of the swap space on the disk is temporary; second, the speed of transferring processes between the memory and the swap space on the disk is crucial; third, for I/O operations, several contiguous blocks of data transfer are faster than several separate blocks.

The kernel has a mapping array to manage the swap space. Each entry of the mapping array holds the address and number of blocks of a free space. At first, the array has only one entry that consists of the total number of blocks that the swap space in the system can have. After several times of swapping in and out, the mapping array can have many entries.

The malloc system call is used to allocate the swap space to the process to be swapped out. With the first-fit algorithm, the kernel scans the mapping array for the first entry that can make the process fit in. If the size of the process can cover all the blocks of entry, all the blocks are allocated to the process, and this entry is removed from the array. If the process cannot use

all the blocks, the kernel breaks up the blocks into two sequential groups, one that are enough for the process are allocated to the process; the other becomes a new entry with modified address and number of blocks in the mapping array.

To free the swap space, the kernel does some merge just like what it does when deallocating memory. If one or both of the front and back neighbors are free, the newly freed entry is merged with them. The address or the number of blocks of the new entry should be modified, and some entry may be deleted according to the situation. If the newly freed entry is separate, the kernel adds one entry into an appropriate position of the mapping array and fills in its address and number of blocks.

5.2.4 Selection Principle of Swapped Processes

The selection principles for the processes to be swapped out or in are slightly different.

To swap in processes, the swapper has to make a decision on which one to be swapped in earlier than others. Two rules are used.

- It examines processes that have been swapped out and are ready to run.
- It tests how long a process stays on the disk. The longer time a process stays than others, the earlier it will be swapped in.

To swap out processes, the swapper chooses a process according to the rules:

- It examines the processes that are sleeping in memory for some event to occur.
- It checks how long a process stays in memory. The process for the longest time will be first swapped out.

5.2.5 Swapper

The swapper or Process 0 is a kernel process that enters an infinite loop after the system is booted. It tries to swap processes in memory from the swap space on the disk or swap out processes from memory onto the swap space if necessary, or it goes to sleep if there is no process suitable or necessary to swap. The kernel periodically schedules it like other processes in the system. When the swapper is scheduled, it examines all processes whose states are “ready, swapped”, chooses one that has been out on the disk for the longest time. If there is free memory space available and enough for the chosen process, the swapper does the swapping in for the process. If successful, the swapping-in repetition continues to look for other process in “ready, swapped” state and swap them in one by one until no process on the swap space is in “ready, swapped” or there is no room in memory available for

the process to be swapped in. If there is no room in memory for the process to be swapped in, the swapper enters in swapping-out searching, in which it checks the processes that are sleeping in memory, and chooses the one that has been in memory for the longest time to swap out on the swap space of the disk. Then the infinite loop goes back to look for the processes to be swapped in. If there is no chosen process, the swapper goes to sleep. The summary procedure of swapper is shown in Figure 5.3.

```
while (true)  {
   for (all processes in "ready, swapped")  {
    Look for the process swapped out on disk for the longest time;
    if (there is no such process)  {
     sleep (event for there is process to be swapped in);
     continue;
    }
    if (there is free memory space enough for the chosen process)  {
     Allocate free memory space to the process;        /* swap in the process */
    Read the process from the swap space on the disk;
    Mark the process state as "ready, in memory";
    Free the swap space;
     continue;
   }
  }
  for (all processes in "sleeping in memory") {
    Look for the process in memory for the longest time;
    if (there is no such process) {
     sleep (event for there is process to be swapped in);
     continue;
    }
    else {
      Allocate the swap space to the process; /* swap out the chosen process */
      Write the process to the swap space;
      Mark the process state as "sleeping, swap ped";
      Free the memory space;
       continue;
    }
  }
}
```

Fig. 5.3 The C-style summary procedure of swapper.

The procedure of swapper also indicates that the swapping has to handle three parts: the swap device allocation, swapping processes in memory, and swapping out of memory.

5.2.6 Swapping Effect

As known, the swapping is simple and appropriate for the system with the small main memory space. But it has some flaws.

As the swap space is on the disk, the swapping can seriously intensify the file system traffic and increase the usage of disk.

When the kernel wants to swap in a process but there is no free memory space for it, it has to swap out a process. If the size of the process to be swapped out is much less than the process to be swapped in, one time of swapping out cannot make the swapping-in successful. This may cause swapping to delay longer, especially considered that it involves I/O operations.

Further more, there is an extreme case existing for swapping. If some of the swapped-out processes are ready to run, the swapper is invoked to try to swap them in. But if there is not enough room in memory now, the swapper has to swap out some process in memory. However, if there is no room in the swap space on the disk, either, and at the same time some new process is created, a stalemate can happen. If the processes in memory are time-consuming, the deadlock can keep much longer.

5.3 Demand Paging in UNIX

Since the advent of computer systems with the virtual memory, such as VAX, in the early 1970s, operating systems have been developed to manage the memory in a new way related to virtual memory address space that is different from the physical memory space, and even better, expands the memory space into a virtual larger scope and allows more processes to execute in the system simultaneously. Demand paging in virtual memory even enhances the system throughput and makes the concurrently execution of multiple processes in a uni-processor system implement well.

As known, the locality principle was first addressed by Peter J. Denning in 1968 (Denning 1983), which uncovered the fact that the working set of pages that the process references in a short period of time are limited to a few of pages. Because the working set is a little dynamical part of a process, if the memory management handles this dynamicity as soon as possible, the system potentially can increase its throughput and allow more processes concurrently executing. We also know that the swapping needs to transfer the whole processes and may aggravate the disk I/O traffic. Demand paging transfers only some pages of the processes between the memory and the swap space on the disk, and has some mechanisms to reduce the I/O traffic that will be discussed in this section.

Demand paging usually cooperates with page replacement in the virtual memory management. The former tackles how and when to bring one or more pages of a process in memory; the latter handles how to swap out some pages

of a process periodically in order to allow new pages of a process to enter the memory. When a process references a page that is not in memory, it causes a page fault that invokes demand paging. The kernel puts the process in sleeping until the needed page is read in memory and is accessible for the process. When the page is loaded in memory, the process resumes the execution interrupted by the page fault. As there is always some new page to be brought in memory, the page replacement must dynamically swap out some pages onto the swap space. Thus, we will discuss demand paging and page replacement in separate sections.

5.3.1 Demand Paging

As known, in the virtual memory address space, the pages of a process are indexed by the logical page number, which indicates the logical order in the process. We also have a physical memory address space in the system, which is also divided into pages. To avoid making readers confused by two context-related pages, the pages in the virtual memory space are still called pages, but the pages in physical memory are usually called frames. When a page of a process is put in a frame, this page is really allocated in physical memory.

Three data structures in the UNIX kernel are needed to support demand paging: page table, frame table, and swap table. And two handlers are used to accomplish demand paging for different situations.

5.3.1.1 Page Table

Entries of the page table are indexed by the page number of a process, and one entry is for a page. Each entry of the page table has several fields: the physical address of the page, protection bits, valid bit, reference bit, modify bits, copy-on-write bit, age bits, the address on the disk, and disk type (see in Figure 5.4).

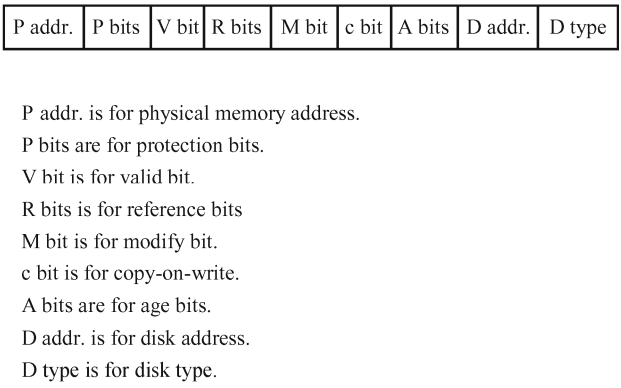

Fig. 5.4 Fields of entry in the page table.

Here gives the explanation of the fields:

- Physical address is the address of the page in the physical memory, which is the frame address that the page occupies.
- Protection bits are the access privileges for processes to read, write or execute the page.
- Valid bit indicates whether or not the content of a page is valid.
- Reference bits indicate how many processes reference the page.
- Modify bit shows whether or not the page is recently modified by processes.
- Copy-on-write bit is used by fork system call when a child process is created.
- Age bits show how long the page is in memory and are used for page replacement.
- Disk address shows the address of the page on the disk, including the logical device number and block number, no matter whether it is in the file system or the swap space on the disk.
- Disk type includes four kinds: file, swap, demand zero, and demand fill. If the page is in an executable file, its disk type is marked as file and its disk address is the logical device number and block number of the page in the file on the file system. If the page is on the swap space, its disk type is marked as swap and its disk address is the logical device number and block number of the page in the swap space on the disk. If the page is marked as "demand zero", which means bss segment (block started by symbol segment) that contains data not initialized at compile time, the kernel will clear the page when it assigns the page to the process. If the page is marked as "demand fill", which contains the data initialized at compile time, the kernel will leave the page to be overwritten with the content of the process when allocating the frame to the process.

5.3.1.2 Frame Table

Frame table is used for physical memory management. One frame of physical memory has an entry in the frame table, and the frame table is indexed with the frame number in physical memory. Entries in frame table can be arranged on one of two lists: a free frame list or a hash frame queue.

The frames on the free frame list are reclaimable. When a frame is put at the end of the free frame list, it will be allocated to a new page of a process if no process does reference it again in a period of time. However, a process may cause a page fault that is found still on the free frame list, and it can save once I/O operation of reading from the swap space on the disk.

The hash frame table is indexed with the key that is the disk address (including the logical device number and block number). One entry of the hash frame table is corresponding to a hash queue with one unique key value. With the key value, the kernel can search for a page on the hash frame table quickly.

When the kernel allocates a free frame to a page of a process, it removes

an entry at the front of the free frame list, modifies its disk address, and inserts the frame into a hash frame queue according to the disk address.

To support the frame allocation and deallocation, each entry in the frame table has several fields:

- Frame state can be several situations, for example, reclaimable, on the swap space, in an executable file, being underway of reading in memory, or accessible.
- The number of referencing processes shows how many processes access the page.
- Disk address where the page is stored in the file system or the swap space on the disk includes the logical device number and block number.
- Pointers to the forward and backward neighbor frames on the free frame list or a hash frame queue.

5.3.1.3 Swap Table

The swap table is used by page replacement and swapping. Each page on the swap space has an entry in the swap table. The entry holds a reference field that indicates how many page table entries point to the page.

5.3.1.4 Page Fault

We have known that when a process references a page that is not in memory, it causes a page fault that invokes demand paging. In fact, demand paging is a handler that is similar to general interrupt handlers, except that the demand paging handler can go to sleep but interrupt handlers cannot. Because the demand paging handler is invoked in a running process and it will be back to the running process, its execution has to be in the context of the running process. Thus, demand paging handler can sleep when I/O operation is done for the page read or swapped in memory.

Since page faults can occur in different situations during a process execution, in the UNIX virtual memory management, there are two kinds of page faults: protection page faults and validity page faults. Thus, there are two demand paging handlers, protection handler and validity handler, which handle protection page faults and validity page faults, respectively.

The protection page faults are often caused by the fork system call. The validity page faults can be resulted from several situations depending on the different stages of the execution of a process, and mostly related to the execve system call. Thus, later we will discuss these two, respectively.

Protection page fault

In the UNIX System V, the kernel manages processes with the per process region table that is usually part of process control block of the process. Each of its entries represents a region in the process and holds a pointer to the starting virtual address of the region. The region contains the page table of this region and the reference field that indicates how many processes reference the region. The per process region table consists of shared regions and private

regions of the process. The former holds one part of the process that can be shared by several processes; the latter contains the other part that is protected from other processes' references.

When the demand paging handler is invoked during the fork system call, the kernel increments the region reference field of shared regions for the child process. For each of private regions of the child process, the kernel allocates a new region table entry and page table. The kernel then examines each entry in page table of the parent process. If a page is valid, the kernel increments the reference process number in its frame table entry, indicating the number of processes that share the page via different regions rather than through the shared region in order to let the parent and child processes go in different ways after the execve system call. Similarly, if the page exists on the swap space, it increments the reference field of the swap table entry for this page. Now the page can be referenced through both regions, which share the page until one of the parent or child processes writes to it.

Then the kernel copies the page so that each region has a private version. To do this, the kernel turns on the copy-on-write bit for each page table entry in private regions of the parent and child processes during the fork system call. If either process writes the page, it causes a protection page fault that invokes the protection handler. Now we can see that the copy-on-write bit in a page table entry is designed to separate a child process creation from its physical memory allocation. In this way, via protection page fault, the memory allocation can postpone until it is needed.

The protection page fault can be caused in two situations. One is when a process references a valid page but its permission bits do not allow the process access, and the other is when a process tries to write a page whose copy-on-write bit is set by the fork system call. The kernel has to check first whether or not permission is denied in order to make a decision about what to do next, to signal an error message or to invoke the protection handler. If the latter, the protection handler is invoked.

When the protection handler is invoked, the kernel searches for the appropriate region and page table entry, and locks the region so that the page cannot be swapped out while the protection handler operates on it. If the page is shared with other processes, the kernel allocates a new frame and copies the contents of the old page to it; the other processes still reference the old page. After copying the page and updating the page table entry with the new frame number, the kernel decrements the process reference number of the old frame table entry.

If the copy-on-write bit of the page is set but the page is not shared with other processes, the kernel lets the process retain the old frame. Then the kernel separates the page from its disk copy because the process will write the page in memory but other processes may use the disk copy. Then it decrements the reference field of the swap table entry for the page and if the reference number becomes 0, frees the swap space. It clears the copy-on-write bit and updates the page table entry. Then it recalculates the process

priority because the process has been raised to a kernel-level priority when it invokes the demand paging handler in order to smooth the demand paging process. Finally, before returning to the user mode, it checks signal receipts that reached during handling the demand paging.

Through the processing above, we can see that the page copying of the child process is deferred until the process needs it and causes a protection page fault, rather than when it is created.

BSD systems used demand paging before System V and had their solution to the separate memory allocation for a child process. In BSD, there are two versions of fork system calls: one is the regular one that is just the fork system call; the other is the vfork system call that does not do physical memory allocation for the child process. The fork system call makes a physical copy of the pages of the parent process, which is a wasteful operation if it is closely followed by an execve system call. However the vfork system call, which assumes that a child process will immediately invoke the execve system call after returning from the vfork call, does not copy page tables so it is faster than the fork system call of System V. The potential risk of vfork is that if a programmer uses vfork incorrectly, the system will go into danger. After vfork system call, the child process uses the physical memory address space of the parent process before execve or exit is called, and can ruin the parent's data and stack by accident and make the parent not to be able to go back into its working context.

Validity page fault

When the execve system call is invoked, the kernel reads an executable file whose name is one of the arguments of the execve system call into memory from the file system on the disk. Readers now should realize that the disk in UNIX is divided into two portions: one is used statically for the file system; the other is allocatable dynamically for the swap space. In the memory management with demand paging, the kernel does not pre-allocate memory to the whole executable file, but assigns physical memory and reads in pages when demanded.

The kernel finds all the disk block numbers of the executable file from the inode on the file system during execve system call and puts the list in the in-core inode table entry that is in memory. We will discuss inode in Chapter 6.

To manage the memory in the demand paging way, the kernel has to assign the page tables for the executable file first, and setting the disk type field of each page table entry as demand zero or demand fill. When setting up the page tables for the executable file, the kernel fills in the disk address for each entry with the logical device number and block number containing the page. The starting logical block number of an executable file is usually 0. The validity handler will use the disk address to find and bring the page in memory from the file system on the disk.

When reading in each page of the file, a validity page fault incurs, and

the validity handler is invoked. If it is set as demand zero, the content of the frame is cleared; if it is marked as demand fill, the content of the frame is replaced with one page of the file.

The above is typically what the execve system call does. But how and what does the validity handler do for processes? The validity handler is invoked when a process tries to access a page whose valid bit is not set.

As known, if the page is valid, its content can be used rather than processed halfway. Thus, except in the valid state, in any other state during processing of the page the instruction or data of the page cannot be used to execute.

In fact, the virtual memory management in UNIX does not set the valid bit of a page table entry if the page is outside the virtual address space of a process or if the page is assigned with the virtual address space but does not have an assigned frame. Usually the hardware with virtual memory provides the function to cause a page fault when a page in an invalid state is accessed. The kernel searches for the page in the page table. If it finds it, the kernel locks the region containing the page table entry to prevent the page from being swapped out and allocates a frame of physical memory to bring in the page content from the executable file or the swap device. If not, the reference is invalid and an error signal is sent to the process. It is what the validity handler mainly does.

As analyzed above, the page that can cause the validity page fault has a zero of the valid bit of its page table entry, and typically has five possibilities: file, swap, demand zero, or demand fill in the disk type field of its page table entry, or on the free frame list in memory.

If we put it in the order that the page is processed from far from to close to the valid state of the page, we can have the following sequence of five different invalid states.

1) File

In this situation, the kernel cannot find the page on the free frame list but finds its disk address on the file system from the page table and its disk type is file. It means now the page is in the executable file on the file system and has never been read in memory before. The kernel has to read the page from its original position in the file system, which can be found from its in-code inode.

The validity handler looks into the page table for the disk address of the page, and also finds the in-core inode from its region table. The logical block number is used as an offset to find the disk block number where the page is stored in the file system by adding it to the starting logical address of the array of disk block numbers in the in-core inode. Then the kernel takes a frame table entry from the head of the free frame list, puts the frame table entry on a hash queue, reads the page content from the file system, and modifies the physical memory address field of the page table entry with the frame number. The validity handler sleeps until the disk I/O operation finishes. Then it wakes up the processes sleeping for the content of the page

to be read in.

Finally, the validity handler sets the valid bit and resets the modify bit of the page. It also recalculates the process priority and checks for signals for the same reasons as the protection handler.

2) Demand zero

In this situation, the kernel cannot find the page on the free frame list but finds that it is set as demand zero in its disk type field. It means that the page has to be cleared before being used.

The validity handler allocates a frame table entry, and clears the content of the frame. Finally, the kernel sets the valid bit, and updates the page table and frame table. It also recalculates the process priority and checks for signals.

3) Demand fill

Similar to demand zero, in this case, the kernel cannot find the page on the free frame list but finds that it is set as demand fill in its disk type field. It means that the content of the page has to be overwritten with the content of the executable file.

The validity handler allocates a frame table entry to the page, and copies the page of the file into the frame. And the kernel also does the rest part of the work like the demand zero case.

4) On the free frame list

In this case, even though the page table entry indicates that the page is swapped out, it is still on the free frame list without being reassigned to the other page.

The validity handler finds the page on the free frame list, reassigns the page table entry to the page, and increments its page reference bits. The kernel sets its valid bit, and updates the page table and frame table.

In this situation, the kernel does not need any disk I/O operation.

5) Swap, not in memory

In this situation, the kernel cannot find the page on the free frame list but finds its disk address on the swap space from the page table. It means that the page was in memory but now is swapped out on the disk.

The validity handler takes a free frame table entry, and reads the page content in the frame from the swap space. Then the validity handler sleeps until the disk I/O operation finishes. Then it wakes up the processes sleeping for the content of the page to be read in.

Finally, the validity handler sets the valid bit, and updates the page table and frame table. It also recalculates the process priority and checks for signals.

5.3.2 Page Replacement

As known, demand paging should cooperate with page replacement to implement the virtual memory management. When a process executes, its working

pages change dynamically. Some of its pages in memory should be swapped out dynamically and replaced by new pages to let the process keep executing until its work finishes.

Page replacement is similar to the swapping, except that it swaps out pages of a process rather than a whole process.

In UNIX, there are two solutions to page replacement: one is the page stealer of System V; the other is the pagedaemon of the 4.2BSD. We will discuss them in this section, respectively.

5.3.2.1 Page Stealer

Since the system's booted, the page stealer has been created. The kernel sets two threshold values for the number of free frames: one is the low value for the least number of free frames; the other is the high value for the most number. When the number of the free frames is fewer than the low value, kernel awakens the page stealer to swap out the eligible pages on the swap space of the disk. When the number of the free frames increases more than the high value, the page stealer goes to sleep. The two threshold values can be configured to reduce thrashing by system administrators.

The page stealer adopts a least recently used page replacement algorithm, and its implementation is slightly different from the ones introduced in Section 5.1.3.

When the page stealer is invoked, it does several turns of examinations on the all valid pages, for instance, four times, which can be set. At the first turn, the page stealer resets the reference bit of each valid page and begins to count how many turns of examinations have gone since the page was last referenced. Within the four turns, if no new reference to the page, its count number increments once for each turn and so that finally becomes four, which means the page has aged enough to be swapped up. Thus, the page stealer swaps out the page. However, within the four turns, when any new reference appears to the page, its count number is reset. And at the forth turn, its count number is less than four, and the page is unsuitable for swapping. The count number is recorded in the age bits of the page table entry. Thus the maximum number of examination turns can be constrained by the width of the age bit field in the page table entry.

Before the page stealer really swaps out a page, it has to see whether or not the swapping is necessary. There are two situations:

- If there is a copy of the page on the swap space and the modify bit of the page table entry is clear, it does not need to swap out the page really. Thus the page stealer just clears its valid bit of the page table entry, and decrements its reference number of the frame table entry. If the reference number becomes 0, the page stealer puts the frame entry on the free frame list.
- If there is no copy of the page on the swap space or there is a copy on the swap space but the page is modified, the page stealer puts the page on a list of pages to be swapped out, clears its valid bit of the page table entry,

and also decrements its reference number of the frame table entry. If the reference number becomes 0, the page stealer puts its frame entry on the free frame list. When the list of pages to be swapped out is full, the kernel does swap out all the pages in the list on the swap space. The collection of the pages to be swapped out can reduce the disk I/O operations.

The System V retains the swapping in its virtual memory management to reduce thrashing when the kernel cannot allocate pages for a process because the working pages in memory are larger than the physical memory space and the page replacement handlers cannot make the page replacement timely. In this situation, the swapper can swap out entire processes and alleviate the system overload quickly and effectively. When the swapped-out processes become "ready, swapped", the kernel brings them in memory with page faults rather than directly swapping in.

5.3.2.2 Pagedaemon

In the virtual memory management of the 4.2BSD, the page replacement is implemented with the pagedaemon. The pagedaemon is Process 2 of the system, along with Process 0 (called scheduler in BSD systems, swapper in System V) and Process 1 (init). The pagedaemon is used to keep the free frame list contain an enough number of frames during the system running.

There are three threshold values used in the BSD to help control when the pagedaemon or swapper to be invoked. The three threshold values are lotsfree that is a less number of free frames in the free frame list, minfree that is the least number of free frames in the free frame list, and desfree that is the average desirable number of free frames. Usually, the three values have the comparison of minfree $<$ desfree $<$ lotsfree. The pagedaemon spends most of its time sleeping, but the kernel checks periodically the number of free frames. When the number of free frames is below lotsfree, the kernel wakes up the pagedaemon. When the number of free frames increases to lotsfree, the pagedaemon stops and goes to sleep again.

If the system goes into an extreme situation with high overload, the number of free frames drops below minfree, and the average number of free frames over a period of recent time is less than desfree, the swapper is awakened to swap out a process as a whole.

The 4.2BSD also uses two lists to manage frames in physical memory: one is the free frame list that contains the free frames; the other is called the core map (cmap) that holds the used frames recorded with the disk block numbers, which indicates which process pages are mapped into the frames.

As mentioned in Section 5.1.3, the 4.2BSD adopts a slightly different clock page replacement algorithm. An algorithm clock hand scans the frames in cmap cyclically. Each examination turn includes two cycles of scanning all the frames in cmap. If some conditions indicate that a frame is in process or untouched, the frame is skipped. Otherwise, the page table entry corresponding to the frame is located and checked whether or not it is valid. If it is valid, the kernel clears its valid bit and makes the frame reclaimable. If the

page is not referenced again before the next scanning cycle reaches it, it can be allocated. If the page has been modified, it must first be written to disk before the frame is added to the free frame list.

If the page is referenced again before the next scanning cycle reaches it, a page fault occurs and the page is made valid again. It will not be put on the free frame list next time it is scanned.

With VAX's main memory space increasing, one scanning cycle of the algorithm clock hand can take so long time to make the second scanning arrival at a page have less relevance to the operations done in the first cycle. In the 4.3BSD, a second algorithm clock hand is adopted, which starts out later after the first clock hand. The first hand makes a page invalid, later on the second hand lets it reclaimed. The whole process takes less time than a whole cycle of all the frames on the free frame list.

5.4 Summary

In this chapter, we have discussed the outline of memory management, process swapping, and demand paging in UNIX.

In multiprocessing operating systems, there are many processes that must be put in different areas of the memory. Multiprogramming needs complex schemes to divide and manage the memory space for several processes in order to avoid the processes' interfering with each other when executing and make their execution just like single process executing in the system.

Considered that some processes wait for I/O devices in memory without doing anything, swapping is used as a memory management strategy. If the swapping is fast enough to let the user not to realize the delay, the system can handle more processes and the performance of the whole system becomes better.

When the size of an application program becomes too big to load in the memory as a whole at a time, the memory paging is needed.

There are several common memory allocation algorithms. They are first-fit, next-fit, best-fit, and quick-fit algorithms.The first-fit algorithm is simple and fast. next-fit algorithm is often faster than the first-fit algorithm. But when the processes fill up the memory and some of them are swapped out on the disk, the next-fit algorithm does not necessarily surpass first-fit algorithm. The best-fit algorithm is slower than the first-fit and next-fit algorithms. The quick-fit algorithm can find a required free segment faster than other algorithms.

Five common page replacement algorithms have been introduced, which are the optimal, FIFO, LRU, clock, and revised clock page replacement algorithms. The UNIX System V uses a least recently used page replacement algorithm that was implemented differently. The 4.2BSD used a slightly different clock page replacement algorithm.

In swapping in UNIX, swapped context of a process, the timing of swapping, and the selection principle of swapped processes are important. The swapper is a kernel process that tries to swap processes in memory from the swap space on the disk or swap out processes from memory onto the swap space if necessary.

In demand paging in UNIX, three data structures are used to support the demand paging and page replacement, which are page table, frame table, and swap table. When page faults occur, two kinds of handlers can be invoked. The protection handler services for protection page faults that are usually caused by the folk system call. The validity handler tackles the validity page faults that often occur during the execve system call. There are two solutions for page replacement in UNIX: one is the page stealer used in UNIX System V; the other is the pagedaemon existing in the 4.2BSD.

Problems

Problem 5.1 For the memory management scheme that divides the physical memory into different size-fixed areas, what are the biggest problems? How can they be solved?

Problem 5.2 Please explain the swapping mechanism? Why is the swapping mechanism brought into memory management? What is the shortcoming of the swapping?

Problem 5.3 What is the demand paging? For what reason can the demand paging be used in memory management? Compared to the swapping, what advantages does the demand paging have?

Problem 5.4 In swapping, how does the kernel keep track of memory usage? If you are the designer of the memory management system, how can you manage the physical memory of your system in the swapping scheme? Please give your algorithm in detail.

Problem 5.5 In this chapter, several memory allocation algorithms have been discussed. What are they? Please describe them in your way.

Problem 5.6 If you are the developer, please choose one of the three algorithms for your memory management system: the first-fit, next-fit or best-fit algorithm and implement it. Please explain why you choose it and how you will implement it.

Problem 5.7 If you are the developer, how can you implement the quick-fit algorithm? Please describe the practicable implementation in detail.

Problem 5.8 When you use the quick-fit algorithm for the memory allocation, how will you carry out the deallocation of the memory when some process exits the system and needs to free its memory space?

Problem 5.9 We have discussed several page replacement algorithms in this chapter. Try to introduce and compare them in your way.

Problem 5.10 If you are the developer, please use the least recently used

algorithm and implement it. You can use software or hardware to do your implementation. Please explain how you will do.

Problem 5.11 For a system, what reasons can cause the swapper invoked too frequently? If you are the developer of the system, how can you handle it?

Problem 5.12 Please give your version of the malloc system call program to allocate the swap space for the process to be swapped out. You can descript it with a C-like algorithm.

Problem 5.13 If you have to design a virtual memory management with demand paging, please design an algorithm to manipulate the hash frame table for the used frames.

Problem 5.14 Please compare the two handlers: protection handler and validity handler. Give your implementation versions of these handlers, respectively.

Problem 5.15 Please give a solution to deallocate the swap space on the disk in detail.

References

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Bartels G, Karlin A, Anderson D et al (1999) Potentials and limitations of fault-based Markov prefetching for virtual memory pages. SIGMETRICS'99: The 1999 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, Atlanta, Georgia, June 1999: 206 – 207

Bays C (1977) A comparison of next-fit, first-fit, and best-fit. Commun ACM 20(3): 191 – 192

Belay LA, Nelson RA, Shedler GS (1969) An anomaly in space-time characteristics of certain programs running in a paging machine. Commun ACM 12(6): 349 – 353

Braams J (1995) Batch class process scheduler for UNIX SVR4. SIGMETERICS'95/PERFORMANCE'95: The 1995 ACM SIGMETERICS Joint International Conference on Measurement and Modeling of Computer Systems, Ottawa, 1995, Canada. ACM, pp 301 – 302

Brawn BS, Gustavson FG, Mankin ES (1970) Sorting in a paging environment. Commun ACM 13(8): 483 – 494

Christensen C, Hause AD (1970) A multiprogramming, virtual memory system for a small computer. AFIPS'70 (spring): Spring Joint Computer Conference, Atlantic City, New Jersey, 5 – 7 May 1970: pp 683 – 690

Cmelik RF, Gehani NH, Roome WD (1989) Experience with multiple processor versions of concurrent C. IEEE T Software Eng 15(3): 335 – 344

Denning PJ (1968) Thrashing: its causes and prevention. AFIPS'68 (Fall, part I): Fall Joint Computer Conference, part I, San Francisco, California, 9 – 11 December 1968: pp 915 – 922

Denning PJ (1970) Virtual memory. ACM Compt Surv 2(3): 153 – 189

Denning PJ (1983) The working set model for program behavior. Commun ACM 26(1): 43 – 48

Iftode L, Blumrich M, Dubnicki C et al (1999) Shared virtual memory with automatic update support. ICS'99: The 13th International Conference on Supercomputing, Rhodes, Greece, May 1999, ACM: 175 – 183

McKusick MK, Neville-Neil GV (2005) The design and implementation of FreeBSD operating system. Addison-Wesley, Boston

Midorikawa ET, Piantola RL, Cassettari HH (2008) On adaptive replacement based on LRU with working area restriction algorithm. ACM SIGOPS Operating Systems Review 42(6): 81 – 92

Park Y, Scott R (1996) Virtual memory versus file interfaces for large, memory-intensive scientific applications. Supercomputing'96: The 1996 ACM/IEEE Conference on Supercomputing (CDROM), Pittsburgh, Pennsylvania, November 1996, IEEE computer society: Article No.: 53

Peachey DR, Bunt RB, Williamson CL et al (1984) An experimental investigation of scheduling strategies for UNIX. SIGMETRICS'84: ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, August 1984, 12(3): pp 158 – 166

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading, Massachusetts

Rashid R, Tevanian A, Michael JR et al (1988) Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. IEEE T Comput 37(8): 896 – 908

Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17 (7): 365 – 375

Stallings W (1998) Operating systems: internals and design principles, 3rd edn. Prentice Hall, Upper Saddle River, New Jersey.

Weizer N, Oppenheimer G (1969) Virtual memory management in a paging environment. Spring Joint Computer Conference, ACM, Boston, Massachusetts, 14 – 16 May 1969: pp 249 – 256

6 UNIX File System

When using a computer system, users are mostly performing file-related operations: reading, writing, modifying, creating, or executing files. And these operations are interacting with the file system. Therefore, readers need to understand the file system and file concept in UNIX, how they are managed and represented in the operating system, and how they are stored on the disk. In this chapter, we will discuss the file system structure, file concept in UNIX, how to manage file and file system, and the file representation and storage. This chapter will also focus on the local file system.

6.1 UNIX File System Structure

The UNIX file system can be characterized with the hierarchical structure (Ashley et al 1999; Bach 2006; Cai et al 2009; McKusick 1999; McKusick et al 2005; Quarterman et al 1985; Ritchie et al 1974; Thompson 1978), consistent treatment of file data, ability to create and delete files, dynamic growth of files, file data protection, and treatment of peripheral devices (such as terminals and disk) as files. For users, it is easy to understand the UNIX file system from three aspects: how files in the system are organized, how files are stored on the secondary storage, and how files are read, and written.

The UNIX kernel keeps regular files and directories on block devices such as disks. The system may have several physical disk units. Partitioning a disk into several parts makes it easier for administrators to manage the data stored there. Otherwise, the kernel deals on a logical level with file systems rather than with disks, treating each one as a logical device identified by a logical device number.

Since the UNIX kernel provides its services transparently and hides the device distinction from users, even though a computer system has several disk drives that contain user and system files, for a user, it is not necessary to worry about which disk drive contains the file that is needed to access. Users who are familiar with MS-DOS and Microsoft Windows know that there can be several disks or partitions in thses operating systems, such as C:, D:, E:,

etc. In UNIX, however, these disks or partitions hide from its users (Sarwar et al 2006). All the several disk drives or disk partitions can be mounted on the same file system structure, allowing their access as directories and not as named drives C:, D:, E:, etc. Files and directories on these disks or partitions can be accessed by specifying their pathnames as if they are part of the file system on one single partition of one single disk. The benefit of this mechanism is for users not to remember in what drive or partition files (and directories) are.

6.1.1 File System Organization

As known in Section 2.4.2, the UNIX file system is organized as an upside-down tree with a single root node called the root directory (written as "/"); every non-leaf node of the file system structure is a directory of files, and files at the leaf nodes of the tree are either directories, regular files or special device files in the file system hierarchy (Bach 2006; Floyd et al 1989; Sarwar et al 2006). Thus, the file system structure starts with one root directory, and can have any number of files and subdirectories under it. This structure leads to a hierarchy relationship between a directory and its subdirectories and files. A typical UNIX system is organized as shown in Figure 6.1.

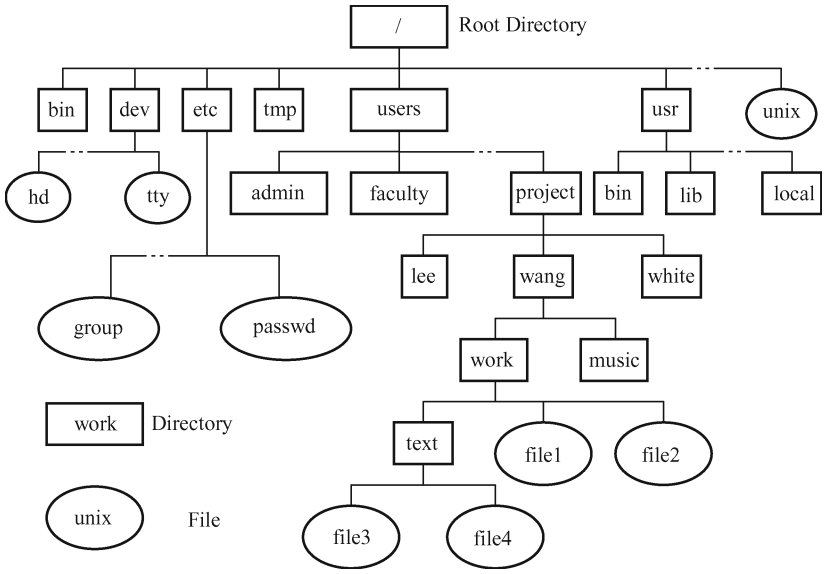

Fig. 6.1 An example of UNIX directory tree.

In Figure 6.1, some of the standard files and directories are shown. These directories and files exist in every UNIX operating system, so it is useful for

users to understand what they are, what they contain and where they are used. Here Table 6.1 gives a summary of some of these directories.

Table 6.1 Summary of some UNIX standard directories

Directory	Function	Content
/	Root directory	The main directory for the whole file system
/bin	The binary directory that contains binary (executable) programs of most UNIX commands or links to executable files in some other directories	cat, chmod, cp, csh, date, kill, ksh, ln, ls, mail, mkdir, more, mv, ping, ps, rm, rmdir, sh, tar, vi, etc.
/dev	The device directory that contains special files for the I/O devices (such as terminals, disk drives, CD-ROM drive, tape drive, modem, network, printer, etc.) connected to the computer	cdrom (CD-ROM drive), console (the console), fd (floppy drive), hd (hard disk or a partition on a hard disk), isdn (an ISDN connection), lp (line printer), midi (a midi interface), pty (pseudo terminal), ram (RAM disk), tty (a terminal), etc.
/etc	The administrator directory that contains commands and files for system administration.	group, init, inetd.conf, login, passwd, printcap, .profile, rc.d, services, termcap, etc.
/lib	The library directory that contains a collection of files related to a high-level language in a single file called an archive; the archive file for a language can be used by applications developed in the language	Libraries for C, C++, FORTRAN, etc.
/tmp	The temporary directory that contains temporary files, used by several commands and applications; the life of a file in this directory is set by the administrator and is usually only a few minutes	All the files in this directory are deleted periodically in order to prevent the disk from being filled with temporary files
/users	The user directory that contains the home directories of all the users of the system	The files of users
/usr	It contains subdirectories that hold the UNIX utilities, tools, language libraries, manual pages, etc. Two of those subdirectories are bin and lib, which contain binary programs of most UNIX commands and the language libraries, respectively	UNIX commands, language libraries, etc.

Note: The /bin directory is often a link to the /usr/bin directory. The /etc directory is of the system administrators, so common users cannot access into this directory. The /lib directory is often a link to the /usr/lib directory.

In the UNIX file system, there are several files that are worth the detailed

explanation: one is passwd; the other is unix.

The passwd file in /etc directory holds several lines of user information, each line for each user on the system. Each line has seven fields, separated by colons. The format of the line is as follows.

```
login_name:password:user_ID:group_ID:user_info:home_directory:
login_ shell
```

The meaning for each field is listed in the following:

- login_name. It is the login name that the user types in when logging in and with which the system identifies the user.
- password. It is the password for the user, which contains the encrypted characters in older UNIX operating systems and the dummy x or * in newer systems (starting with System VR4). The newer versions of the UNIX operating system store encrypted passwords in /etc/shadow.
- user_ID. It is an integer between 0 to 65535 assigned to the user, where 0 is assigned to the superuser and 1—99 are reserved.
- group_ID. It identifies the group that the user belongs to, and it also is an integer between 0 and 65535, with 0—99 reserved.
- user_info. It contains information about the user, such as the user's full name.
- home_directory. It holds the absolute pathname for the user's home directory.
- login_shell. It holds the absolute pathname for the user's login shell. When the user logs on, the system executes the command corresponding to the pathname listed in this field.

For example, the following line from the etc/passwd file on the system is for the user of wang.

```
wang:x:120:101:wang wei:/users/project/wang:/usr/bin/sh
```

where the login name is wang, the password field contains x, the user ID is 120, the group ID is 101, the personal information is wang wei, the home directory is/users/project/wang, and the login shell is /usr/bin/sh, which is the Bourne shell.

Introduced in Section 4.6, the init file in the /etc directory is the parent of all the other user processes in the system.

The libc.a in the /lib directory is the C library, which consists of utility routines and system call interfaces.

In the /usr/include directory, there are system parameter files. Among them, the /usr/include/stdio.h holds the parameters of the standard I/O system.

The unix file in the root directory holds the binary program of the UNIX kernel to be loaded into memory at system booting time. For different UNIX operating systems, the unix file can have different names and also can be located in different directories. For example, in BSD, it is vmunix in the root directory; in Sun's Solaris system, it is unix in /kernel directory.

There are some more standard directories and files, about which readers

can read the on-line help of UNIX.

6.1.2 Home and Working Directories

As mentioned in Chapter 2, when a user logs in, the UNIX system puts the user in the home directory. The directory in which the user is working is called the working directory (also known as the current directory). For example, the directory called wang in Figure 6.1 is the home directory for the username of wang for login. For the C and Korn shells, the tilde (∼) character can be used as the shell metacharacter for the home directory. The working directory can be substituted with . (dot). The parent of the working directory is substituted with .. (double dots).

6.1.3 Absolute and Relative Pathnames

As the UNIX file system is constructed in a hierarchy, the pathnames are helpful for access to directories and files.

To specify a file or directory location, use its pathname. A pathname functions like the address of the directory for file in the UNIX file system. As the UNIX file system organizes its files and directories in an inverted tree structure with the root directory at the top, an absolute pathname directs the path of directories to travel from the root to the directory or file the user wants to go into. To locate a file or directory, it can also be done with a relative pathname, which gives the location relative to the working directory. A relative pathname always goes through from the current directory to the directory or file the user wants to go into. The root is indicated by the slash (/) at the start of the pathname, so an absolute pathname always starts with a slash (/). However, as all relative pathnames start from the working directory, a relative pathname never starts with a slash. An absolute pathname can be used by any user from anywhere in the file system structure, but a relative pathname can have different destinations for different users.

In a pathname, put slashes (/) between the directory names. Pathnames can be written in three ways as follows:

- Starting from the root directory. That means, it is an absolute pathname.
- Starting with the working directory. That means, it is a relative pathname.
- Starting with the user's home directory or the parent directory of the working directory. That means, it is also a relative pathname.

For example, to locate the file of file3 under the subdirectory of text in Figure 6.1, assuming now the working directory is work, and the user has logged in with the username of wang, use one of the following pathnames. Before that,

firstly, when a user logs in with the username of wang, the system puts the user into the home directory, /users/project/wang. Then the user moves to the working directory, work, by using the cd work command. Then the valid pathnames for locating the file of file3 are:

```
/users/project/wang/work/text/file3      (an absolute pathname)
text/file3                (a relative pathname)
./text/file3              (a relative pathname)
~ /work/text/file3        (a relative pathname)
../work/text/file3        (a relative pathname)
```

In Section 2.4, some commands used to determine the home and working directories and interact with the UNIX file system have been discussed.

6.1.4 UNIX Inodes and Data Structures for File System

For the UNIX kernel, the internal representation of a file is given by an inode, which is a data structure and contains a description of the disk layout of the file data, the file owner, access permissions, and access times. The term inode is a contraction of the term index node. Every file has one inode, but it may have several names called links, all of which map into the inode via a pathname. When a file is referenced by a process with the name of the file that is typically a pathname having several components separated by slashes, the kernel parses the file name one component at a time, checks that the process has permission to search the directories in the path, and eventually retrieves the inode for the file. Inode list is stored in the file system on the disk (see Section 6.5), but the kernel reads them into an in-core inode table that is in the main memory when accessing files.

Two other data structures in the kernel, the file table and the per-process file descriptor table, are also used to manage the file system (see Section 6.5.4 later on in this chapter). The file table is a global kernel structure, but the per-process file descriptor table is allocated to each process. When a process on behalf of a user opens or creates a file with the open or creat system call (see Section 2.1.2), the kernel allocates an entry from each table, corresponding to the file's inode. Entries in the three structures — per-process file descriptor table, file table, and inode table — manage the state of the file and the user's access to it. The file table holds the information including the access rights to the open process, and the byte offset in the file where the user's next read or write will start. The user file descriptor table contains all the open files for a process. When the open or creat system call is invoked, the kernel returns a file descriptor, which is an index into the user file descriptor table. When executing read and write system calls (see Section 6.3), the kernel uses the file descriptor to access the user file descriptor table and follows its pointer to the file table. Then it follows the pointers in the file table to inode table entries. And finally, from the inode, the kernel finds the data in the file.

Later on in Section 6.5 of this chapter, some more information about the inode will be discussed.

6.2 UNIX File Concept and Types of Files

Different from other kinds of operating systems, in UNIX, the concept of files is extended, uniform and transparent, which covers not only common files but also devices, including almost all the system resources (Mohay et al 1997; Ousterhout et al 1985; Quarterman et al 1985). Therefore, in UNIX, all the things, including all input and output devices (such as a keyboard, a display terminal, a disk drive, a network interface card, a printer, etc.), a directory, of course, an ordinary file, are treated as files. All the things belong to the UNIX file system. In short, there are different kinds of files in the UNIX file system.

6.2.1 Types of Files

UNIX operating system supports the following types of files:

- Ordinary files
- Directories
- Special files (for I/O devices), including block device special files and character device special files
- Pipes
- Sockets
- Link Files

6.2.2 Ordinary Files

Ordinary files are the files defined in other operating systems, such as MS-DOS and Microsoft Windows. They hold information and data, and are stored on a secondary storage device (or a block device), such as a disk. As known, ordinary files can include source programs (in C, C++, Java languages, etc.), executable programs (applications of some programming tools, such as compilers, database tools, desktop publishing tools, graphing software, etc.), pictures, graphics, audio, etc.

UNIX file system treats every file as a sequence of bytes. In other words, it tackles these files in the same way. It does not specify a structure to a file according to its contents. A file's contents are identified by the application of some programming tools. That means, for UNIX, a C program file, an HTML

file for a Web page, and a file for a video clip are all the same, ordinary files. However, a C compiler (e.g., cc) treats a C program totally different from other kinds of ordinary files.

The UNIX file system almost does not impose any naming conventions on files of any type, except the situation that can bring some confusion to the system, such as a slash (/), which is reserved as the separator between files and directories in a pathname. Additionally, the following are other unwise choices for a filename.

- Use a dash (-) as the first character of a filename because it will be confused with an option.
- Use a space (space or tab) in a filename because the shell breaks command lines into separate arguments at the spaces.
- Use nonprintable characters in a filename because they are difficult for the shell to deal with as part of a file name.
- Use other shell metacharacters (see Chapter 8) in a filename because they are difficult for the shell to deal with as part of a file name.

Furthermore, unlike Microsoft Windows, UNIX does not require a dot (.) in a filename for extensions; in fact, file names can have as many dots as wanted, such as, it..is and it.is.a.game that are legal filenames in UNIX. However, it is recommended to use an .exe extension to an executable program and a .doc extension to a document, which can give the explainable meaning in the file names, especially for the users who have been already familiar with Microsoft Windows. Additionally, it largely depends on the application tools if or not to use an extension in a file name. For instance, all C compilers require that C source program files have a .c extension, but not all Web browsers require an .html extension for files for Web pages. Here gives some extensions in Table 6.2, and other situations need the readers to check out the relative references.

Table 6.2 Some extensions for some application tools

Extension	File
.asm	Assembly language code
.bin	MacBinary file
.bmp, .gif, .jpg	Graphics file
.c	C Source code
.C, .ccp, .cc	C ++ Source code
.exe	Executable program
.java	Java source code
.html, .htm	File for a Web page
.o	Object code
.z, .gz	Compressed file

Some UNIX operating systems, such as System V, limit filenames to 14 characters. Most modern systems allow much longer filenames, for example, 255 letters in BSD.

A filename must be unique inside one directory, but different directories may have files with the same names. For example, in two directories, project/wang/work/text and faculty/wang/work, there are two files named the same name file3 in them, respectively.

6.2.3 Directories

In UNIX, files are organized into directories. A directory is actually a special kind of file where the file system stores information about other files. A directory can be thought as a place, where files are to be contained. So a directory is similar to a folder used in other operating systems.

A directory file consists of an array of directory entries. Each entry holds the information of a file or a subdirectory under the directory. In UNIX, a directory entry is the structure shown in Figure 6.2.

Fig. 6.2 A directory entry structure.

As mentioned in Section 6.1.4, every file has one inode. The Inode of a file contains file attributes such as file owner ID, access permissions, file size (in bytes), etc. The inode number (or i-number) is an index value in the inode list on the disk.

In the early UNIX System versions, the width of each directory entry was 16 bytes. The first two held the inode number, so filenames were constrained to 14 characters. From the 4.2BSD, filenames are expanded up to 255 characters. The longer filenames require more overhead for the kernel, but give the users more flexibility with their file names.

6.2.4 Special Files

Both types of special files, block device special files and character device special files, specify devices, and therefore, their file inodes do not reference any data. Each special file is a means of accessing hardware devices, including character devices (such as the keyboard and printer) and block devices (such as hard disk). Each hardware device is corresponding to at least one special file. So to access a device, use the command or system call that accesses its special file.

The inode for a special file contains two numbers known as the major and minor device numbers. The major number indicates a device type such as terminal or disk, and the minor number indicates the unit number of the

device. Some special files are such as, fd1 (for floppy drive 1), hdc (for hard drive c), lp3 (line printer 3), and tty (for teletype-terminal).

Special files are placed in the /dev directory (see Table 6.1). This directory contains at least one file for each device connected to the computer. User applications read and write peripheral device files in the same way that they read or write an ordinary file. The UNIX kernel recognizes that a given file is a regular file or a device, but hides the distinction from user processes. That is also called device-independent. Various special devices simulate physical devices and are therefore known as logical devices. The logical devices allow users to interact with a UNIX system without maneuvering directly the physical devices connected to the computer system. The mechanism is becoming more and more important because it allows use of a UNIX system via a network or modem, or with virtual terminals in a window system such as the X Window System.

6.2.5 Pipes

Shown in Chapter 4, UNIX provides several mechanisms for local processes to communicate with each other. Except these mechanisms, pipes can be used to do the similar work.

A pipe is an area in the kernel memory (a kernel buffer) that allows two processes, which are running on the same computer system and related to each other, to communicate with each other, for example, one produces data and the other consumes data.

A named pipe is a file that allows two processes to communicate with each other if the processes are on the same computer, but do not have to be related to each other.

The pipe sometimes called a FIFO (for the "first-in-first-out"), differs from a regular file in that the data space in a pipe is temporary: once data is read from a pipe, it cannot be read again. Also, the data are read in the order that they were written to the pipe, and the system allows no change for that order. With the pipe system call, the kernel can store data in a named pipe in the same way it stores data in an ordinary file, except that it uses only the direct address, not the indirect address.

6.2.6 Sockets

Sockets are socket files that processes use to communicate each other according to some kind of protocol suite. A socket can be used by processes on the same computer or on different computers, which can be on the Intranet or on the Internet. For example, a socket of AF_INET address is used for communi-

cation by using the transport level protocols in the TCP/IP protocol suite. A socket with AF_INET is known as the Internet domain socket. A socket with AF_UNIX address can be used for communication between processes running on the same machine under a UNIX operating system, which is known as a UNIX domain socket.

6.2.7 Link Files

The concept of a link in UNIX is for file sharing. A link file is created by the system when a symbolic link is created to an existing file. The link file points to the existing file, which allows the users to rename an existing file and share it without duplicating its contents. As introduced in Section 6.1.4, for the kernel, every file has one inode, but it may have several names of links, all of which map into the inode.

The concept of the symbolic link in UNIX is a creation of BSD but is presently available on almost all versions of UNIX (Ousterhout et al 1985; Quarterman et al 1985). The symbolic link is also called soft link. A symbolic link may be a file containing the pathname of another file or directory, and it can cross file system boundaries. In contrast to soft link, it is a hard link. There are two typical hard links: one is written as "."; the other is ".." (see Section 6.3.2). "." in a directory is a hard link to the directory itself, and ".." of the directory is a hard link to its parent directory.

6.3 Managing Files and Directories

In Chapter 2, some commands related files and directories have been introduced. In this section, some of them are reviewed more detailed and some more commands will be brought in gradually. Having the file system concept of the UNIX kernel in mind, readers can know them more clearly and deeply. The discussion will cover some commands for browsing through the UNIX file system, creating files and directories, determining file attributes, displaying the absolute pathname for the home directory, displaying the pathname for the working directory, and displaying the type of a file. In the meantime, some examples will be given in order to help readers know these commands. In Section 6.3, the discussion will be based on the file Structure shown in Figure 6.1.

6.3.1 Displaying Pathname for Home Directory and Changing Directories

The following is a description of the echo command. In fact, echo command has been used in Section 2.6.

The syntax and function of echo command are as follows.

```
$ echo [string]
```

Function: to display 'string' on the display screen; 'string' can be strings, shell variables, and file references.

Common options: none.

Command arguments:

\c: print line without newline;

\t : tab character;

\n: newline.

Note: echo can terminate with newline, but can also let the cursor stay at the end of the same line by using \c. Without an argument, the command displays a blank line on the screen.

As known, when logging in, the user is put in the home directory, assumed the user name is wang. At this time, to see which one is the absolute pathname of the home directory, use the echo or pwd command. Notice that if moving the directory, use pwd command to check out the working directory rather than the home directory because every time logging in, the home directory is the working directory. The pwd command displays the absolute pathname of the current directory.

For example:

```
$ echo How are you!  \c
How are you!
```

To display the absolute pathname of the home directory, use the following command:

```
$ echo $HOME
/users/project/wang
$
```

Or use this command:

```
$ pwd
/users/project/wang
$
```

As SHELL, HOME is also a shell variable (also called placeholder), which the shell uses to store the absolute pathname of the home directory (see in Chapter 8). There is another shell variable PWD that is related to the following cd command. The shell variable PWD is set after each execution of the cd command, and the pwd command uses the value of this variable to display the absolute pathname of the working directory.

To go from the home directory to other directories in the file system, use

the cd (change directory).

The syntax and function of cd command are as follows.

```
$ cd [pathname]
```

Function: to change the working directory to the directory that 'pathname' directs, or return to the home directory when without the argument—'pathname'.

Common options: none.

For example:

```
$ cd /users/project/wang/work
$ pwd
/users/project/wang/work
$ cd text
$ pwd
/users/project/wang/work/text
$
```

The first command changes the working directory to the directory /users/project/wang/work. The output of pwd shows now the working directory is /users/project/wang/work. The third command changes the working directory to the directory /users/project/wang/work/text. The output of the fourth command pwd shows the execution result of the third command.

6.3.2 Viewing Directories and File Attributes

After getting into a directory, to view its contents (the names of files or subdirectories in it), provided that the user has the permissions, use the ls command. In fact, ls command is such a powerful command that with different arguments and options, it can list different types of content relate to the directories and files in the file system. The use of the ls command with various options will be displayed in the rest of this chapter and other chapters of the book.

The syntax and function of ls command are as follows.

```
$ ls [option(s)] [dirname(s)-and-filename(s)]
```

Function: to display and list the names of the files and directories in the directories, or just names of files, which are specified in 'dirname(s)-and-filename(s)'; or the names of the files and directories in the working directory without the 'dirname(s)-and-filename(s)' argument.

Common options:

-l: to display long list that includes file access modes, link court, owner, group, file size (in bytes), and modification time;

-a: to display names of all the files, including hidden files;

-i: to display i-node numbers.

Note: If a complete pathname specification is given to the argument of

the ls command, the names of files and directories along that pathname will be listed.

For example:

```
$ cd ..
$ pwd
/users/project/wang/work
$ ls
text file1 file2
$ ls ~
work music
$ ls $HOME
work music
$ cd
$
```

If the ls command is used without any argument as the third command, it displays the names of files and directories in the working directory, which is the work directory here. The ls ~ and ls $HOME commands display the names of the files and directories in the home directory. The cd command (without any argument) lets move back to the home directory. Now, the home directory is the current directory.

Usually, for the regular users of UNIX, the ls command displays the names of all the files and directories in the current directory, except the hidden files (system files). But for the superusers of the system and the users who are given the permissions to access all the files and directories in the system, they can access many important files and directories related to system administration and to other users' files and directories. A superuser has the permission to go to the root directory and list its contents. The following commands can list more information for a superuser.

For example:

```
$ cd /
$ ls
bin dev etc install lost+found tmp users usr unix
$ cd
$ ls -a
. .. .cshrc .exrc .pinerc work music
$
```

The first command moves to the root directory. The second command displays the content of the root directory. The third command moves to the home directory. The final command displays all the names of files and directories in the home directory including the hidden files. Among all the directories and files, the . (dot) and . . (dot dot) are the working directory and the parent of the working directory. Some of the hidden files have been discussed in the previous chapters. Here gives some more for readers to know what they are.

- .addressbook is address book for pine;
- .bshrc is setup for Bash shell;
- .cshrc is setup for C shell;
- .exrc is setup for vi;

- .login is setup for C or TC shell and executed at login time;
- .mailrc is setup and address book for mail and mailx;
- .profile is setup for Bourne or Korn shells and executed at login time.

The ls command can also be used to determine the attributes of files. Its options can be used together, but their order does not matter. For example, use the -l option to get a long list of a directory that gives the attributes of files as follows:

```
$ ls -la
Total 32
drwxr-xr-x   2   wang   project   512   Apr   15   15:11    .
drwxr-xr-x   2   admi   admi      512   Apr   12   15:01    ..
-rw-r--r--   2   wang   project   136   Jan   16   11:48    .exrc
-rw-r--r--   2   wang   project   833   Jan   16   14:51    .profile
-rw-r--r--   1   wang   project   797   Jan   16   15:02    file1
-rw-r--r--   1   wang   project   251   Jan   16   15:03    file2
drwxr-xr-x   2   wang   project   512   Apr   15   15:11    text
...
$
```

The second line of the display result is the attributes for the working directory (.). The third line is the information for the parent directory of the working directory (..). The owner of the files is wang, who belongs to the group project. The more detailed explanation about the information can be seen the description of the ls command in Chapter 2.

Here, pay more attention to the first column of the last line about the information of the text directory. There is slightly difference between the access permissions of a file and the permissions for a directory. The access permissions to a file allow the user whether or not to do action on file's contents. The access permissions on the directory control the user whether or not to do action on the directory's contents, that is, whether or not to rename or remove the files under it.

Read permission for a file decides whether or not the user can read file's contents. Owning write permission makes the user have the privilege to change file's contents. Execute permission is just for the files that are executable.

Read permission on a directory allows the user to read a directory; write permission allows a user to create new directory entries or remove old ones, thereby changing the contents of the directory; execute permission allows a user to search the directory for a file name (not to execute a directory as to do a file). So if to access a directory, a user must have execute permission to the directory and all of its parent directories up to the root.

If using the option of F, the file type is displayed after each file. For example:

```
$ ls -F /
bin/ dev/ etc/ install@ temp/ usr/ users/ unix*
$
```

If using the option of i, the inode number of a file is displayed before each file. For example:

```
$ ls -i
12450 file1 12567 file2 22345 text
$
```

This example shows that the inode numbers for file1, file2, and text are 12450, 12567, and 22345, respectively.

6.3.3 Creating Directories and Files

As mentioned in Chapter 2, there is the mkdir command that can be used for creating a directory. However, there is no UNIX command that is specified for creating a file even though there are several other methods for creating a file. For example, copy a new duplicate of an existing file by the cp command. Another method of creating a file is by using one of application tools, such as a compiler, or one of the text editors that we have discussed in Chapter 3.

We have introduced the mkdir and rmdir commands in Chapter 2. In this section, some more examples will be given in order to know the application of them.

When the user of wang logs on the system, Wang can use the following command to create a subdirectory, called homework, in the home directory, as follows:

```
$ mkdir homework
$ ls
homework music work
$
```

Then, Wang can create a directory called text2 in the /users/project/wang/work by using the following command:

```
$ mkdir /users/project/wang/work/text2
$
```

To create the directory called test1 in the lab directory that has not been created yet, the user can type this command:

```
$ mkdir -p lab/test1
$
```

To remove an empty directory, use the rmdir command directly. If the directory is not empty, remove the files and subdirectories in it first. Another way to do so is to use the rm command with r option (see Chapter 2).

The following command removes the test1 directory from the working directory.

```
$ rmdir test1
$
```

If test1 is a file, the output of the command is the error message: rmdir: test1: Path component not a directory. If test1 is not empty, the result of the command is the error message: rmdir: test1: Directory not empty.

If the ∼/lab directory holds only one empty subdirectory called test1, the following command can remove both of them at the same time.

```
$ rmdir -p ∼/lab/test1
$
```

6.3.4 Displaying Type of a File

As UNIX file system does not support the extensions of filenames, it is difficult to determine what a file contains from its name. However, the file command can help find the type of a file's contents. The file command tests every argument in order to classify them. Usually, this command is used to distinguish between a text file and a binary file. Text files can be displayed on the screen, but binary files cannot because the system may interpret some of the binary values as control codes.

The syntax and function of file command are as follows.

```
$ file [options] file(s)
```

Function: to determine the type of files.

Common options:

-f ffile: to treat ffile as a file that contains a list of files to be examined.

Note: some of classifications that the file command displays are directory, symbolic link, ascii text, C program text, Bourne shell script text, empty, nroff/troff, command text, PostScript, sccs, setuid executable, setgid executable, etc.

For example:

```
$ file file1
file1:  ascii text
$ file shellscpt1
shellscpt1:  command text
$ file text
text:  directory
$
```

It is a good idea for users to read the online man pages for the UNIX commands on UNIX in order to get more information about them.

6.3.5 Making Lines in File Ordered

Sometimes, in some situations, such as an index of a book, a glossary of a textbook, student registration list, or a customer list of a company, it is necessary to make the content of a file in a certain order, such as in ascending, descending or alphabetical order. For example, to make a set of strings {cherry, orange, pear, strawberry, grape, peach} in alphabetical order, it will

be {cherry, grape, orange, peach, pear, strawberry}. Another example, to sort {122, 134, 156, 178, 111, 190} in ascending order, it will be {111, 122, 134, 156, 178, 190}.

To make lines of an ASCII file ordered, use the sort command.

The syntax and function of sort command are as follows.

```
$ sort [options] file(s)
```

Function: to sort lines in the ASCII files according to a collated sequence and display the result on the screen; by default, to sort according to the entire input line.

Common options:

+n1 –n2: to specify a field as the sort key, starting with n1 and ending at n2;

+n1: to specify a field as the sort key, starting with n1 and ending at the end of line;

-d: to sort in alphabetical order, only comparing letters, digits and blanks;

-f: to treat lowercase and uppercase letters as equals;

-i: to ignore nonprintable characters;

-r: to collate in reverse order;

-b: to ignore leading blank characters when determining the starting and ending positions of a sort key.

Note: The sort command just displays the sorting result, but does not change the content of the original file. Without any argument, sort takes input from the keyboard. Without redirection, the output of the sort command is sent to the standard output – the screen.

For the sort command, the lines in a file are strings separated by the newline character; fields are usually groups of characters separated by spaces; it is based on a field or grouped fields to sort lines in a file. The field numbers start with 0. This field or grouped fields are called the sort key. Sometimes, several fields can combine into one sort key. The sort key is defined by option +n1 [-n2] in a sort command. Without the sort key specification, sort takes each line starting with the beginning of the line to be the sort key.

The lines in a file are compared by using the sort keys to determine the position of each line in the sorted list. Therefore, the sort command rearranges the lines of the file according to the sort keys from left to right.

When an ordering option appears without any key field specification, the rule of this ordering option is applied globally to all sort keys.

Here, some examples of the sort command are given.

First example: The freshman file is a record file for a project group (each record line for one student), and fields are divided each other by one or more space characters. Field 0 is family name; Field 1 is given name; Field 2 is cell-phone number; Field 3 is the email address. Note all the records are imaginary and not real.

```
$cat freshman
Jones David      1358999998      jonesdavid@hebust.edu.cn
```

```
Huo Song          1341212121      huosong88@126.com
Dang Ailin        1500011223      dangailin@163.com
Jones David       1357777888      jdavid87@sina.com
Liang Fanghua     1310000555      liangfh88@163.com
Miller Hill       1322223434      millerhill8806@sina.com
Gao Hill          1361010101      gaoyuling@hebust.edu.cn
Nan Xee           1393219123      nanxee1987@aaa.com
Wang Feng         1311212123      wangfeng89@hebust.edu.cn
$ sort freshman
Dang Ailin        1500011223      dangailin@163.com
Gao Hill          1361010101      gaoyuling@hebust.edu.cn
Huo Song          1341212121      huosong88@126.com
Jones David       1357777888      jdavid87@sina.com
Jones David       1358999998      jonesdavid@hebust.edu.cn
Liang Fanghua     1310000555      liangfh88@163.com
Miller Hill       1322223434      millerhill8806@sina.com
Nan Xee           1393219123      nanxee1987@aaa.com
Wang Feng         1311212123      wangfeng89@hebust.edu.cn
$
```

As this sort command is used without the sort key, the lines are sorted in alphabetical order and ascending order by all letters and digits from left to right.

Second example: Continue the first example. This time, sort the freshman file by identifying Field 2 (cell-phone number) as the sort key and in reverse order.

```
$ sort +2 -3 -r -b freshman
Dang Ailin      1500011223      dangailin@163.com
Nan Xee         1393219123      nanxee1987@aaa.com
Gao Hill        1361010101      gaoyuling@hebust.edu.cn
Jones David     1358999998      jonesdavid@hebust.edu.cn
Jones David     1357777888      jdavid87@sina.com
Huo Song        1341212121      huosong88@126.com
Miller Hill     1322223434      millerhill8806@sina.com
Wang Feng       1311212123      wangfeng89@hebust.edu.cn
Liang Fanghua   1310000555      liangfh88@163.com
$
```

The result of this sort command is the lines rearranged in reverse according to the cell-phone number. Remember the field number starting from 0. The sort key that is specified by (+2 –3) is the string of characters starting from cell-phone number and ending before the e-mail address. As the fields in the freshman file are separated by different numbers of space in different lines, it is important to use the -b option that tells to sort without considering the leading spaces between fields. As sorting is according to the ASCII values and the space character has smaller one than all letters and digits have, blank spaces being included will generate unexpected output.

Third example: Continue the first example. Here, sort the freshman file with Field 1 (given name) identified as the first sort key, and Field 2 (cell-phone number) and Field 3 (e-mail address) together as the second sort key.

```
$ sort +1 -2 +2 -b freshman
Dang Ailin      1500011223      dangailin@163.com
Jones David     1357777888      jdavid87@sina.com
Jones David     1358999998      jonesdavid@hebust.edu.cn
Liang Fanghua   1310000555      liangfh88@163.com
```

```
Wang Feng       1311212123      wangfeng89@hebust.edu.cn
Miller Hill     1322223434      millerhill8806@sina.com
Gao Hill        1361010101      gaoyuling@hebust.edu.cn
Huo Song        1341212121      huosong88@126.com
Nan Xee         1393219123      nanxee1987@aaa.com
$
```

This command does sorting, firstly, according to the first sort key (+1 –2), Field 1 (given name), and then does the second sorting according to the second sort key (+2), from Field 2 to the end of lines, when the results of the first sorting for more than one line are the same. For example, two given names are "Hill" and two other given names are "David"; they are finally sorted by the second sort key.

6.3.6 Searching Strings in Files

Sometimes, the user wants to search some string of characters in a file, such as to search the record of some student whose family name is Huo in the freshman file of the last section. The greg command can be used to do this. The name "grep" comes from the g/re/p command of the ed (a Unix line editor). The g/re/p means "globally search for a regular expression and print all lines containing it." The regular expressions are a set of UNIX rules that can be used to specify one or more items in a singly character string. The regular expressions work like wildcards (in fact, wildcards belong to the regular expressions), that is, they can save typing in a lot of characters by using representation rules. But the level of support for regular expressions is not the same for different tools.

The grep command searches a file or files for lines that have a certain pattern. The syntax is:

```
$ grep [options] character-string file(s)
```

Function: to search the files for lines including character-string that can be a string, pattern or expression and display the result on the screen.

Common options:

-c: to display only the number of matching lines;

-i: to match either upper- or lowercase;

-l: to display only the names of files that have matching lines;

-n: to display the matched lines with their line number;

-v: to display all lines that do not match pattern.

Note: Without any argument, grep takes input from the keyboard. Without redirection, the output of the grep command is sent to the screen.

Forth example: Continue the first example of last section. Here, grep those lines that contain Hill in the freshman file.

```
$ grep Hill freshman
Miller Hill     1322223434      millerhill8806@sina.com
```

```
    Gao Hill          1361010101      gaoyuling@hebust.edu.cn
$
```

Fifth example: Continue the first example of last section. This time, use the command grep with -n option to filter those lines that contain David in the freshman file.

```
$ grep -n David freshman
1:  Jones David     1358999998     jonesdavid@hebust.edu.cn
4:  Jones David     1357777888     jdavid87@sina.com
$
```

The command grep with -n option displays the matched lines with their line number. So at the beginning of each line, the line number is shown.

Sixth example: Continue the first example of last section. This time, use the command grep with -v option to display those lines that do not contain David in the freshman file.

```
$ grep -v "David" freshman
Huo Song       1341212121      huosong88@126.com
Dang Ailin     1500011223      dangailin@163.com
Liang Fanghua 1310000555       liangfh88@163.com
Miller Hill    1322223434      millerhill8806@sina.com
Gao Hill       1361010101      gaoyuling@hebust.edu.cn
Nan Xee        1393219123      nanxee1987@aaa.com
Wang Feng      1311212123      wangfeng89@hebust.edu.cn
$
```

Here, the command grep with -v option filters out the lines that have David.

Seventh example: Search for lines that has main in all the files that have .c as the end of filenames in the working directory.

```
$ grep -n main *.c
ex1.c 4:  main()
ex2.c 5:  main()
ex3.c 5:  main()
$
```

Eighth example: Continue the seventh example. This time, the grep command is used with -l option. Search for lines that has main in all the files that have .c as the end of filenames in the working directory.

```
$ grep -l main *.c
ex1.c
ex2.c
ex3.c
$
```

The command grep with -l option shows the names of the files with matching lines, but does not display the lines.

6.3.7 The eof and CRTL-D

The eof is an end-of-file marker for UNIX files. When UNIX commands read

their input from files, reading the eof marker means that the end of a file is reached. Usually, for the stored files, the value of the eof marker is a negative integer such as -1.

When the input file is redirected to a keyboard (see later in Section 6.5.5), the end-of-file marker is pressing CTRL-D on a new line. That is why use pressing CTRL-D on a new line to terminate the commands such as cat while reading input from the keyboard. For these commands, pressing CTRL-D on a new line means to send end-of-input signal to them. If CTRL-D is pressed at a shell prompt, which generates a "hangup" signal (see Section 4.5.5), it may close the terminal window or log out of UNIX.

6.4 File and Directory Wildcards

When many files named in series (for example, file1 to file10) or filenames with common characters (such as abin, aben, abong and abone), the wildcards can be used to specify many files at once. These wildcards are asterisk (*), question mark (?), and square brackets ([]). These can save typing for a long filename or to choose many files at once. Given as an argument on a command line, the following can be used:

- * (asterisk) stands for any number of characters in a filename. For example, ab* can match abin, aben, abong, abone, etc. A single * means all file and subdirectory names in a directory.
- ? (question mark) stands for a single character. For example, ab?n matches abin, aben, and abon, but not abean.
- [ ] (square brackets) can be used to surround a choice or a range of digits or letters. For example, [ad]bin would match abin and dbin; [Aa]bin would match Abin and abin; file[1-3] would match file1, file2, and file3.

The wildcards belong to the shell metacharacters that will be discussed in Chapter 8. The wildcards can be used in arguments of the ls command. For example, the command ls /users/project/* displays the names of all the files and directories in the /users/project directory.

The wildcards can also be used to specify a particular set of files and directories. For example:

```
$ ls -l ~/work/file[^6]
```

This command can be used to display the long lists for all the files in the ~/work directory whose names start with the string file followed by zero or more characters, with the condition that the first of these characters cannot be 6. In the Bourne shell, the negative character ^ should be replaced with the !^ character.

Similarly,

```
$ ls -l [a-zA-Z]?[1-7].html
```

This command can be used to display the inode numbers and names of all the files in the current directory whose names start with one letter, followed by any one character, and end with a digit from 1 through 7 and an .html extension.

Another example:

```
$ more ~/[^0-9]*.[c,C]
```

This command can be used to display the contents of all the files in the home directory whose names end with .c or .C and do not start with a digit.

The following command shows names and types of all the files in the root directory.

```
$ file /*
all.backup:      POSIX tar archive
bin:       directory
dev:       directory
etc:       directory
...
tmp:       directory
usr:       directory
unix:      ELF 32-bit LSB executable
$
```

6.5 UNIX File Storage and File System Implementation

A file system consists of sequence of logical blocks, each containing 512, 1024, 2048, or any multiple of 512 bytes, depending on the system implementation. The size of a logical block is fixed within a file system, but different file systems in different computer systems may have different logical block size dependent on the system configuration. Using large logical blocks increases the effective data transfer rate between disk and memory, because the kernel can transfer more data per disk operation and therefore make fewer time-consuming operations that involve a lot of mechanical motions of the I/O device. For example, reading 1K bytes for a disk in one read operation is faster than reading 512 bytes twice. However, if a logical block is too large, effective storage capacity may drop because it can cause a lot of fragmentation storage space unused, especially for small-sized files.

6.5.1 File System Physical Structure and Allocation Strategies

In UNIX System versions, a file system has the following structure shown in Figure 6.3. BSD has a different solution to the file system physical structure, which will be discussed in Section 6.5.1.2.

- Boot block: This block occupies the beginning of a file system, usually the first sector, and may contain the bootstrap loader that is read into

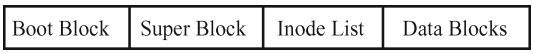

Fig. 6.3 File system structure.

the machine to initialize the operating system when the system's booted. Although only one boot block is needed to boot the system, every file system has a boot block that can be empty.

- Super block: This block describes the information of the file system, including how large it is, how many files it can store, the information of free space on the file system, etc.
- Inode list: This is a list of inodes that follows the super block in the file system. The size of the inode list can be configured in the file system configuration by administrators. The kernel references inodes by index (which is called i-number) into the inode list. One inode is the root inode of the file system. It is the inode by which the directory structure of the file system is accessible after execution of the mount system call.
- Data blocks: These blocks start at the end of the inode list and contain file (and directory) data and administrative data. One and only one file can be corresponding with one data block in the file system.

6.5.1.1 Allocation Strategy in UNIX System Versions

In UNIX System versions, to manage free blocks, the super block contains the free inode list to hold the numbers of free inodes. The free blocks are also kept in the free block list in the super block. The free inode list is filled by the kernel now and then. If the free inode list is empty, the kernel searches the inode list on the disk for the free inodes, from the lowest i-number to the highest one, puts the free inodes on the free inode list, and finally notes the highest i-number of the free inodes. If a process needs a new inode, the kernel assigns the process the free inode with the highest i-number from the free inode list.

The free blocks are managed in number-fixed groups, for instance, each group has 100 free blocks. And each group has just one head block that is incorporated in the free block list. In the free block list, each head block keeps the disk address of the next head block in the list and also holds the disk addresses of up to 100 blocks in the next group. When the free blocks are required, the contents of the head block of one group are brought into the memory and the kernel begins assignment reversely from the last block of the group, like a stack pop. If the group is empty, the next group starts to be assigned because its disk addresses can be found in the head block of the previous group. When a block is deallocated, its address is put at the end in the head block of the recent group in memory in a positive order, like a stack push. If the head block is full, the new deallocated block will be the head block of the next group and the contents of the recent group in memory will be copied into it. Therefore, the disk space is allocated dispersedly and

arbitrarily. The longer the file system is used, the more disordered the block numbers in a file become.

6.5.1.2 Allocation Strategy in BSD

The BSD file system has a different solution. In the BSD file system, except the boot block, the logical disk is divided into cylinder groups that can occupy one or more consecutive cylinders (see Section 6.5.3). Each cylinder group has a superblock, a cylinder block, an array of inodes, and data blocks. The information of free blocks does not exist in the superblock but in the cylinder block. A block bit map for free blocks and fragments and an inode bit map for free inodes in the cylinder block can help manage the free blocks and free inodes, respectively. According to the logical number of a block in the cylinder group, each block is allocated a unique bit in the block bit map. When a block is free, its bit is set as 1. When a block is assigned to a process to write, its corresponding bit is cleaned. Similarly, every inode has a unique bit in the inode bit map in terms of index number. If an inode is free, its bit is 1. Otherwise, its bit is 0.

For the BSD file system, there are three levels of free block allocation routines. The local allocation routine tries to find a block near to the requested one first. If not, it tries to find a block in the same cylinder. If not, it then tries to find a block in the same cylinder group. If not, the quadratic rehash routine is invoked to search for a block in other cylinder groups. If not, the exhaustive routine begins to search for a block. If a block is found in the local allocation routine, the quadratic rehash and exhaustive routines are saved, which is usually the case.

As the disk blocks are grouped in cylinder groups in BSD, the inode for a new file or directory can be allocated to the fullest extent possible from the cylinder group where its parent directory's inode is, which can make the ls command read all the inodes of a directory more easily. And the data disks for a file can also be allocated from the same cylinder or as closely together as possible, which can make the disk head seeking time shorter.

And in BSD, the header information of each cylinder group is put in a varied place of the cylinder group rather than at the beginning of the cylinder group. This scheme can reduce the risk of the whole file system crash caused by a single disk head damage.

In UNIX System versions, the data blocks are fixed-size, typically 512 or 1024 bytes. But in BSD, there are two units for the data blocks: one is still called block; the other is fragment (Quarterman et al 1985). The former is larger; the latter is smaller. The block and fragment sizes can be configured when the file system is created. Typically, the ration of block and fragment is 8/1. For example, the block size is 8192 bytes while the fragment size is 1024 bytes. The BSD solution can have two benefits: one is if the file has a larger size, it can reduce the number of block addresses of the file (which can also occupy the disk space) with the bigger block; the other is the fragment size can reduce the internal fragmentation in the last block of a file (which

can cause the disk space waste).

6.5.2 Inode, Inode List and Inode Table

As mentioned in Section 6.1.4, inode is a data structure on the disk that stores the attributes of a stored file. Each file has a unique inode that is allocated to the file when it is created by the system. All the inodes are in the inode list (or i-list) that is an array of inodes on the disk.

When a file is opened, its inode is allocated an entry in the in-code inode table. The in-core inode table is used by the UNIX kernel for all open files in the memory. The inodes of files that are not open reside on the disk.

The i-list and inode table are indexed by the file inode number. The inode number is used to index the inode table when the kernel on behalf of some process accesses to the attributes of an open file.

When the attributes of a file are manipulated, its inode in the main memory is updated; disk copies of the inodes will be updated later on at fixed intervals. Some of the fields of an inode data structure are shown in Figure 6.4.

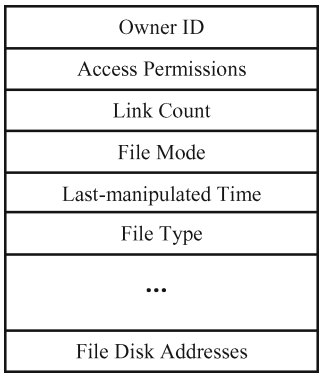

Fig. 6.4 Data structure of an inode.

The following is the explanation of some fields in Figure 6.4:

- Owner ID: This field stores the ID number of the owner of the file.
- Access permissions: This field holds the users who can access the file for what type of operation (see the explanation for the ls command in Chapter 2).
- Link count: This field stores the number of different names (links) the file has within the file system.
- File mode: This field is for what (read, write, etc.) the file is opened.
- Last-manipulated time: This field holds the time that the file is manipulated the most recently.

- File type: This field contains the file type, such as ordinary file, directory, block device special file, character device special file, pipe, socket, link, or free.
- File disk addresses: This field stores a number of direct and indirect pointers to disk blocks containing file data.

In UNIX System versions, one inode has 13 disk addresses, among which the first ten are direct addresses pointing to the first ten blocks of a file, the 11th address is a single indirect address, the 12th address is a double indirect address, and the 13th address is a triple indirect address (Bach 2006; Quarterman et al 1985). For one block size is 512 bytes and one address occupies four bytes, the direct address can accommodate a file that has a size less than 5 120 bytes. The direct address and the single indirect address can hold a file that has a size less than 70 656 bytes but more than 5 120 bytes because the single indirect address contains the addresses of another 128 blocks of the file. If a file size is more than 70 560 bytes but less than 8 459 264 bytes, the double indirect address has to be added, which can provide 128 address blocks that totally point to 16 384 blocks of a file. If a file size is more than 8 459 264 bytes, the triple indirect address can be used, which can provide the addresses for 128^3 of blocks at most.

In BSD, one inode contains 15 disk addresses, in which the first dozen are direct addresses, the 13th is a single indirect address, the 14th is a double indirect address, and the 15th is a triple indirect address. The capacity for each address can be computed in the same way as the UNIX System version.

6.5.3 Disk Physical Structure and Mapping Pathname to Inode

Usually, there are several disk drives in a computer system. Disks are organized into cylinders, each one containing as many concentric circles called tracks as the number of heads stacked vertically. The tracks are subdivided into sectors with the number of sectors along the circumference typically being 8 to 32 on floppy disks, and up to several hundred on hard disks (Heindel et al 1995; Jonge et al 1993; Stallings 1998). The number of heads varies from 1 to 16.

For the kernel, disk I/O access takes place in terms of one sector, also called a disk block.

The address of a sector is a four-dimensional address shown in Figure 6.5.

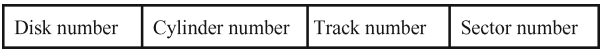

Fig. 6.5 The address of a sector.

This four-dimensional address forms into a linear (one-dimensional) block number that is a logical address used by the UNIX file system because the

block number is relatively easy to deal with.

Usually, these blocks are numbered logically starting from sector 0 of the topmost track of the outermost cylinder, and assigning it block number 0. The block numbers increase with the sector number on the track growing in the circumference direction, with the track number in the cylinder growing downward in the axis direction, with the cylinder number on the disk growing towards the center in the radial direction, and finally with the disk number growth. File space is divided in clusters of two, four, or eight 512-byte disk blocks, which are dependent on different systems.

The way to access a file that is open is shown in Figure 6.6. Assuming that the file named file1 has been open, the kernel, firstly, finds this file and its inode number in the directory of /user/project/wang/work. Then it uses this inode number to index the in-core inode table to get to the inode of the file1 file. Then it uses the file disk addresses in this inode to find the disk blocks where the file1 file is located. Then it accesses (reads or writes) the contents of file1 on the disk.

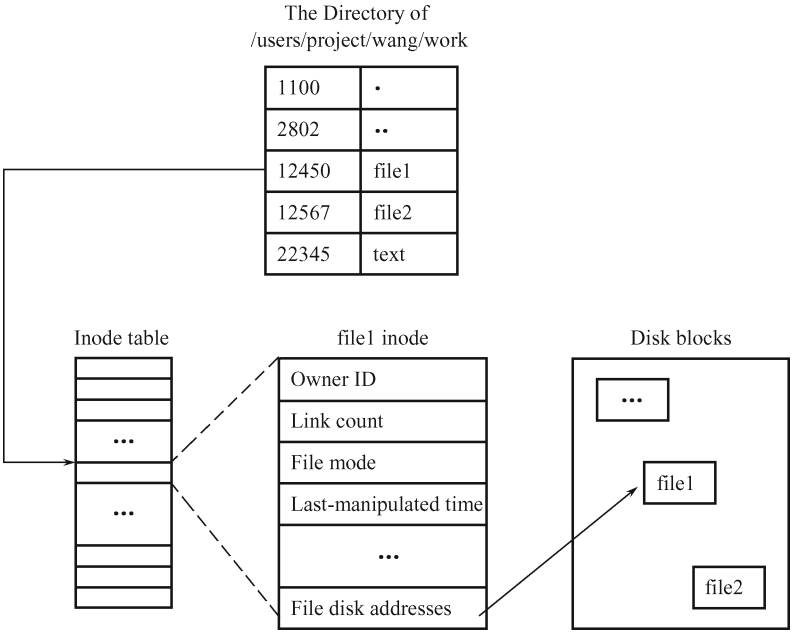

Fig. 6.6 Accessing the file named file1.

As known, the user references a file with a pathname, but the kernel identifies a file with its inode. Thus, it is necessary for the kernel to map a pathname of a file to its inode. If an absolute pathname (such as /user/project/wang/work/file1) is used by the user, the root directory (/) is the starting directory to search for the file1. If a relative pathname (such as work/file1) is given by the user, the working directory (such as /user/project/wang) is taken as

the starting directory by the kernel. Firstly, some examinations have to be passed, such as testing the directory existence, the user's access permissions to the starting directory, etc. Having passed, the kernel finds the inode of the starting directory in the same way shown in Figure 6.6. Then the kernel takes the next item between first two slashes or before the first slash in the pathname (such as the user for the absolute pathname, or work for the relative pathname). With the item as a filename, the kernel looks into the starting directory file to find the inode for the directory (user for the absolute pathname, or work for the relative pathname) if there is no error with this item. The kernel repeats the above process for each the following item between two adjacent slashes in the pathname until the inode of the file1 is found. This is a typical process that the kernel maps a pathname to an inode.

6.5.4 File Descriptors

Mentioned in Chapter 2, the UNIX kernel manages all system resources, and any user application request that involves access to any system resource must be asked to the kernel. The system call interface provides entrances into the UNIX kernel. Services of file system calls include creating, deleting, reading and writing files, managing directories, etc.

In fact, the UNIX file system handles an I/O operation on a file by starting from opening the file first with the open system call and then performing the file operation (read, write, seek, etc.). In other words, the open system call is the first step a process must take to access the data in a file. The open system call takes a pathname and a permission mode as arguments, and returns an integer called the user file descriptor. Other operations, such as reading, writing, seeking, duplicating and closing the file, use the file descriptor that the open system call returns, as illustrated in Figure 6.7.

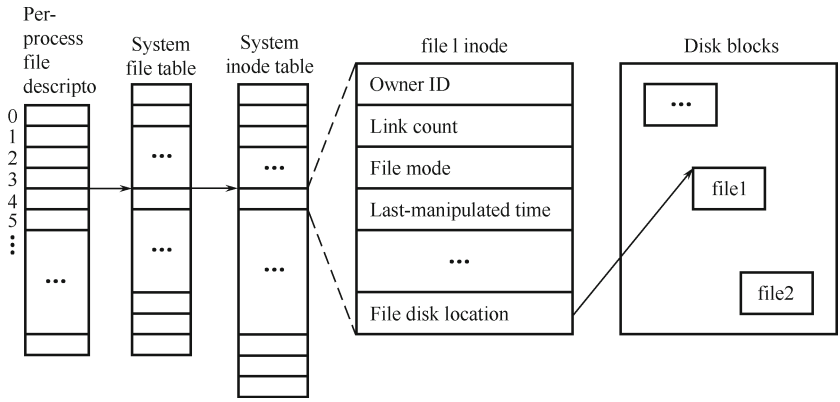

Fig. 6.7 The file descriptor's function when accessing the file named file1.

After the file is opened, the kernel uses a file descriptor to index the per-process file descriptor table to get a pointer to the system file table. The file table contains a pointer to the file inode entry in the inode table. With the file disk addresses in the inode for the file, the required file operation can be performed by accessing appropriate disk block(s) for the file by using the direct and indirect pointers.

6.5.5 System Calls for File System Management

Except the open system call, the creat system call can be used to create a new file and also returns the file descriptor (Bach 2006; Isaak et al 1998; Quarterman et al 1985). The close system call can close an open file with its file descriptor as argument, and free all the data structures that were built by open or creat system call.

The read or write system call takes the file descriptor, a pointer to a buffer and the buffer size as its arguments to transfer data from or to the disk file. Each time a read or write system call is invoked, the kernel keeps the file offset. The lseek system call can be used to set the offset in the file for the next read or write. Since it is possible for several processes to reference a file simultaneously, each of them has to keep their own file offset. The file offset of each process is usually kept in the file table separately. When a parent process creates a child process with fork system call, the child process inherits the file table entry of the parent process. So they may have the same file offset for a file. On the other hand, when a file is referenced, its disk inode is copied into memory and becomes an in-core inode. The in-core inode is added to some more fields, such as, a reference count that records how many file table entries point to it. Similarly, the file table entry also has a reference count that records how many per-process file descriptor entries point to it.

The link system call can create hard links to an existent file while the symlink system call can make new soft links to the file. The unlink system call can remove both hard and soft links. And the process that invokes the unlink system call to remove the last hard link of a file does delete the file and free its inode. A process references the deleted file with its left symbolic link will cause an error.

The chown system call can set the owner and group of a file. The chmod system call is used to change the access permission modes of a file. The stat system call can be used to check the attributes of a file with the file name as argument. The fstat system call can also be used to examine the properties of a file but with its file descriptor as argument.

The mkdir system call can create directories while the rmdir system call is used to remove directories. The chdir system call can set the current (working) directory. The mknod system call can be used to create device special files.

The mount and umount system calls can be used to mount or dismount a

file system of a block special device onto a directory of the existing file system. Thus, the user can access data in a block special device through a file system. The mount system call takes two pathnames as arguments: one pathname is for a block special file (such as /dev/da0); the other is for the directory in the existing file system where the new file system will be mounted (such as /mnt), which is called the mount point. After the mount system call is invoked, the root of the mounted file system, which has to have a super block, inode list and root inode, is accessed via the directory of the mount point in the existing file system (such as /mnt). A mount table contains entries for all the mounted devices, and each entry has some fields, including the mounted device number, a pointer to the memory space where holds the super block of the mounted file system, a pointer to the root inode of the mounted files system, and a pointer to the inode of the directory of the mount point. When the directory of the mount point is referenced, the kernel searches the mount table for the device number of the mounted device, and the root inode of the mounted file system. From the root directory, the user can access other files and subdirectories in the mounted file system. When the mounted file system is no use any more, it can be disconnected with the existing file system via the umount system call.

6.5.6 Standard Files

When running UNIX commands, the kernel automatically opens three files, which are called standard files: stdin is the standard input file, stdout is the standard output file, and stderr is the standard error file.

When a command is typed in on the terminal keyboard after the shell prompt, the command is displayed on the terminal screen. As a command runs, the results are usually displayed on the terminal screen. The terminal keyboard is the command's standard input; the terminal screen is the command's standard output. By default, each of the commands takes its input from the standard input and sends the results to the standard output. And simultaneously, these standard files are attached to the terminal. That is, the shell makes the command input come from the terminal keyboard and its output and error messages go to the terminal screen (or a terminal window, like an xterm in a UNIX system running the X Window System, as discussed in Chapter 2). The command reads input from stdin file and sends its output and error messages to stdout and stderr files, respectively. These files function like terminal I/O transmission buffers.

The UNIX allows changing standard files into alternate files for a single execution of a command, including a shell script. These default files can be varied to other files by using the redirection operators: < for input redirection and > for output and error redirection, which will be discussed in Chapter 7.

The integer values 0, 1, and 2 are the file descriptors for stdin, stdout, and stderr, respectively, known as standard file descriptors, and are the first three entries in per-process file descriptor table shown in Figure 6.7. The other descriptors usually range from 3 through 19 and are called user-defined file descriptors. Chapter 7 will give the more detailed discussion on standard files.

As every device, including a terminal keyboard and screen, is represented by a special file in UNIX, the stdin also is related to the terminal keyboard and stdout and stderr are related to the terminal screen. Figure 6.8 depicts the logical relationship between files and file descriptors, assumed that files file1 and file2 are open and have descriptors 3 and 4, respectively, that the open system call returned when the files were opened. As shown in Figure 6.7, the file descriptors and the files are relating together also through a file table, inode table, and file disk storage in between.

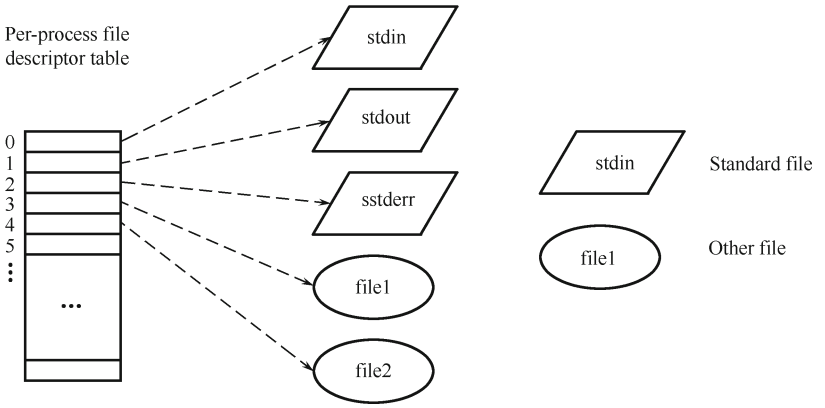

Fig. 6.8 The logical relationship between the file descriptors and the files.

The concept of the standard files and file descriptors is important for readers to understand UNIX commands that take input from standard input or send output to standard output. It is also important for the correct use of commands and system calls.

6.6 Summary

The UNIX file system can be featured by the hierarchical structure, consistent treatment of file data, ability to create and delete files, dynamic growth of files, protection of file data, and treatment of peripheral devices (such as terminals and disk) as files. It is organized as an upside-down tree with a single root directory; every non-leaf node of the file system structure is a directory of files, and files at the leaf nodes of the tree are either directories,

regular files or special device files in the file system. This structure leads to a hierarchy relationship between a directory and its subdirectories and files.

To specify a file or directory location, use its pathname. As the UNIX file system organizes its files and directories in an inverted tree structure with the root directory at the top, an absolute pathname directs the path of directories to travel from the root to the directory or file the user wants to go into. To locate a file or directory, it can also be done with a relative pathname, which gives the location relative to the working directory. The internal representation of a file is given by an inode, which contains a description of the disk layout of the file data, the file owner, access permissions, and access times. Every file has one inode, but it may have several names called links, all of which map into the inode. Inode list is stored in the file system on the disk, but the kernel reads them into an in-core inode table that is in main memory when accessing files. The file table and the per-process file descriptor table are also used to manage the file system.

UNIX operating system supports the different types of files: ordinary files, directories, special files (for I/O devices), including block device special files and character device special files, pipes, sockets and link files. UNIX file system treats every file as a sequence of bytes. And UNIX does not require a dot in a filename for extensions. By using commands, users can browse through the UNIX file system, create files and directories, determine file attributes, display the absolute pathname for the home directory, display the pathname for the working directory, and display the type of a file.

There are three wildcards: asterisk (*), question mark (?), and square brackets ([]), which can save typing for a long filename or choose many files at once.

Physically, a file system has four parts on disks in sequence: the boot block, super block, inode list, and data blocks. Usually, there are several disk drives in a computer system. Disks are organized into cylinders, each one containing as many concentric circles called tracks as heads stacked vertically. The tracks are subdivided into sectors with the number of sectors along the circumference. For the kernel, disk I/O access takes place in terms of one sector, also called a disk block.

To the free inode and block allocation, the UNIX System and BSD versions have different solutions. The former adopts two lists in the super block; the latter uses two bit maps in the cylinder blocks.

An inode is a data structure on the disk that stores the attributes of a stored file. Each file has a unique inode that is allocated to the file when it is created by the kernel. All the inodes are in the inode list that is an array of inodes on the disk. Along other attributes, the file disk addresses in an inode can tell where the kernel can find the blocks of a file on the disk. To fit in different sizes of files, UNIX file system provides direct and indirect addresses. The direct addresses can meet the needs of small-sized files, the single indirect address can hold the middle-sized files, and the double and triple indirect addresses can accommodate the large-sized files.

As the user references a file with a pathname but in the disk a file is represented with an inode, the kernel has to map a pathname to an inode before accessing the file.

The UNIX file system handles an I/O operation on a file by starting from opening the file first and then performing the file operation. The open system call returns an integer called the user file descriptor. Other operations, such as reading, writing, seeking, duplicating, or closing the file, use the file descriptor that the open system call returns. To access a file that is open, the kernel firstly finds this file and its inode number in the directory. Then it uses this inode number to index the in-core inode table to get to the inode of the file. Then it uses the file disk location in the inode to find the disk blocks where the file is located. Then it accesses (reads or writes) the contents of the file on the disk.

The kernel provides different system calls to do different operations on files and directories. The read or write system call can be used to read from or write to a file. The lseek system call can set the file offset. The link and unlink system calls can create or delete hard links to a file. The mkdir and rmdir can create or remove directories. The mount and umount can be used to mount or dismount a file system of a block special device onto a directory of the existing file system. UNIX file system has some more system calls as well.

When running UNIX commands, the kernel automatically opens three files, which are called standard files: stdin is the standard input file, stdout is the standard output file, and stderr is the standard error file. The integer values 0, 1, and 2 are the file descriptors for stdin, stdout, and stderr, respectively, known as standard file descriptors, and are the first three entries in per-process file descriptor table.

Problems

Problem 6.1 What information does the /etc/passwd file contain? What meaning does each field on each line of this file have? View the /etc/passwd file on the UNIX operating system to determine your user ID. Write down the line in the /etc/passwd file that contains information about user's login. What are the login shell, user ID, home directory, and group ID? Does your system contain the encrypted password in the /etc/passwd or /etc/shadow file?

Problem 6.2 For the UNIX kernel, what is the internal representation of a file? How many inode does a file have? And how many link names can a file have? Why?

Problem 6.3 Where is the inode list stored? And where is the in-core inode table stored? What is the latter for?

Problem 6.4 What is the file table? And what is the per-process file de-

scriptor table? Try to explain how to use them when accessing a file?

Problem 6.5 What types of files does UNIX operating system support?

Problem 6.6 What is a directory treated in UNIX? What does a directory entry consist of?

Problem 6.7 How many types of special files are there? What are they? What types of devices do they represent, respectively?

Problem 6.8 The inode for a special file contains two numbers, what are they? What do they represent, respectively?

Problem 6.9 Move to the /dev directory on your system and identify one character special file and one block special file.

Problem 6.10 What is a link in UNIX? When and for what is it created?

Problem 6.11 Just after login, run the following commands. What do they help you determine, respectively?

```
$ echo ~
$ echo $SHELL
```

Problem 6.12 What do the **.** (dot) and**. .** (dot dot) represent, respectively?

Problem 6.13 As UNIX file system does not support the extensions of filenames, it is difficult to determine what a file contains from its name. But there is a command to determine what type a file is. What is it? Execute the file /etc/* command to identify types of all the files in this directory.

Problem 6.14 Create a directory, called college, in the home directory. Go into this directory and create a file college.projects by using one of the editors. Give three pathnames for this file.

Problem 6.15 Continue Problem 6.14. Give a command line for creating a subdirectory lecture under the college directory.

Problem 6.16 Continue Problem 6.14. Make a copy of the file college. projects and put it in the home directory. Name the copied file part-time. projects. Give two commands for accomplishing this task.

Problem 6.17 Continue Problem 6.14. Draw a diagram like that shown in Figure 6.6 for the college directory to show all directory entries, including inode numbers for all the files and directories in it.

Problem 6.18 Continue Problem 6.14. Give the command for deleting the college directory. Give the command to show that the directory has been deleted?

Problem 6.19 Try to do the following job and write down the result of the final command. Use the cd command to go to the /usr/bin directory and run the ls -F command to identify symbolic links and binary files.

Problem 6.20 Try to do the following job. Run the ls -1 command in the home directory and write down the information of some of the files.

Problem 6.21 Check out the inode numbers of the root and the home directory when you log on your machine. Give the commands that can be used to find these inode numbers.

Problem 6.22 Give three commands that can be used to list the absolute pathname of the home directory.

Problem 6.23 Create the freshman file like in Section 6.3.5, and do index and filter on the file with the following commands and see their results.

```
$ sort freshman
$ sort +2 -3 -r -b freshman
$ grep -n David freshman
```

Problem 6.24 What is an end-of-file marker for a UNIX file? What is it used for? What is an end-of-file marker for the input from the keyboard? What is it used for?

Problem 6.25 How many kinds of wildcards are there in UNIX systems? What are they?

Problem 6.26 Give a command line to display all the files in the home directory that start with the word file, and are followed by a digit 1, 2, 6, 8, or 9.

Problem 6.27 Continue Problem 6.26. Give a command line to display all the files in the lecture directory that do not start with letters d, t, V, or X and the second character in the name is neither a digit and nor a letter (uppercase or lowercase).

Problem 6.28 What fields does the file system physical structure contain? Try to explain each of them.

Problem 6.29 In the data structure of an inode, what does the field of link count represent?

Problem 6.30 If you are a file system designer, how do you manage the free inodes? Please give your algorithm.

Problem 6.31 If you have to design an algorithm to handle the free block allocation, how can you make all the files in one directory as closely as possible? Please explain your algorithm.

Problem 6.32 How is a disk organized, usually? What unit is a track subdivided into? How does the kernel do disk I/O access when it references a file?

Problem 6.33 Please implement the algorithm that does map a pathname to an inode with one of the high-level languages.

Problem 6.34 In the UNIX file system, the open system call is the first step a process must take to access the data in a file. The open system call returns an integer. What do we call this integer? How can access a file that is open? Try to describe the access process in detail. And describe the file descriptor's function when accessing a file.

Problem 6.35 How does UNIX build up its file system if there is one physical disk device in the system? How about if there are several physical disk devices? If UNIX has been booted, a new disk device is mounted. How does the kernel treat the new disk device in order to allow the user to access it?

Problem 6.36 When running UNIX commands, the kernel automatically opens three files. What are they? What are their file descriptors, respec-

tively? Usually, what is the command's standard input? And what is the command's standard output?

References

Ashley P, Vandenwauver M (1999) Using SESAME to implement role based access control in UNIX file systems. IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, Palo Alto, California, 16 – 18 June 1999, pp 141 – 146

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Cai X, Gui Y, Johnson R (2009) Exploiting UNIX file-system races via algorithmic complexity attacks. 2009 30th IEEE Symposium on Security and Privacy, Oakland, California, US, 17 – 20 May 2009, pp 27 – 41

Floyd RA, Ellis CS (1989) Directory reference patterns in hierarchical file systems. IEEE T Knowl and Data En, 1(2): 238 – 247

Heindel LE, Kasten VA (1995) RTAS'95: Real-time UNIX application filestores. First IEEE real-time technology and applications symposium, Chicago, Illinois, 15 – 17 May 1995, pp 44 – 45

Isaak J, Lohnson L (1998) POSIX/UNIX standards: foundation for 21st century growth. IEEE Micro 18(4): 88, 87

Jonge W, Kaashoek MF, Hsieh WC (1993) The logical disk: a new approach to improving file systems. SOSP'93: The fourteenth ACM Symposium on Operating Systems Principles, Asheville, North Carolina, December 1993, pp 15 – 28

McKusick MK (1999) Twenty years of Berkeley Unix: from AT&T-owned to freely redistributable. LINUXjunkies.org. http://www.linuxjunkies.org/articles/kirkmck.pdf. Accessed 20 Aug 2010

McKusick MK, Neville-Neil GV (2005) The design and implementation of FreeBSD operating system. Addison-Wesley, Boston

Mohay G, Zellers J (1997) Kernel and shell based applications integrity assurance. ACSAC'97: The IEEE 13th Annual Computer Security Applications Conference, San Diego, 8 – 12, December 1997, pp 34 – 43

Ousterhout JK, Costa HD, Harrison D et al (1985) A trace-driven analysis of the UNIX 4.2 BSD file system. SOSP'85: The Tenth ACM Symposium on Operating Systems Principles, Orcas Island, Washington, December 1985, pp 15 – 24

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading, Massachusetts

Ritchie DM, Thompson K (1974) The UNIX time-sharing system. Commun ACM 17(7): 365 – 375

Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: The textbook, 2nd edn. China Machine Press, Beijing

Stallings W (1998) Operating systems: internals and design principles 3rd edn. Prentice Hall, upper saddle River, New Jersey

Thompson K (1978) UNIX implementation. Bell Sys Tech J 57(6) Part 2: 1931 – 1946

7 UNIX I/O System, I/O Redirection and Piping

Known from Chapter 6, in UNIX, it is through a special file to access one of hardware devices, including character devices (such as the keyboard and printer) and block devices (such as the hard disk). Each hardware device is corresponding to at least one special file. To access a device, use the command or system call that accesses its special file. All I/O devices in the UNIX are treated as files and are accessed as such with the almost same read and write system calls that are used to access all ordinary files (Isaak et al 1998; Jespersen 1995; Sarwar et al 2006). The difference is that device parameters must be set by using a special system call.

Because of the mechanisms that UNIX uses to handle I/O devices, usually, standard files, I/O redirections, and pipes and filters are more relative to the file system in UNIX. In this book, these topics are combined together with I/O devices just to show readers some inherent relationship between I/O devices and the file system.

As we have discussed some concepts about standard files in previous chapters, it is easy for readers to get into I/O system from the standard input and output devices. Thus, we will begin from standard input and output, and standard files in this chapter. Then we will introduce I/O redirections, and pipes, and filters. Finally, I/O system implements in UNIX will be discussed.

7.1 Standard Input and Output, Standard Files

Some UNIX programs or commands read input, some write output, and some do both. Sometimes, the programs or commands read input from a terminal keyboard or write output to a terminal screen; sometimes, they read from a file or write to a file. When a user types in a command on the keyboard, if making some mistakes, such as a command needs a directory argument but the user typing in a file argument, what will happen? Usually, the error is prompted on the terminal screen. However if the command is running in the background and the user does not want the error prompt popping out when

doing other jobs, how can the user do to meet the need?

As known in UNIX, devices are treated as special files and those terminal keyboard and screen are I/O devices. As a big interface between users and terminal devices, UNIX file system must be passed through for users to interact with the computer. Hence, what kind of files is associated with the terminal keyboard and screen? In UNIX, there are three standard files where a command reads its input and sends its output and error messages, called standard input, standard output, and standard error, respectively (Sarwar et al 2006). The input, output, and errors of a command can also be redirected to other files by using the redirection facilities in UNIX. In this chapter, we will discuss how to change the input or output of a command between the terminal and a file, and how to use redirection and pipes to combine some commands together to do more complex jobs.

7.1.1 Standard Input and Output

In Section 6.5.6, standard input and standard output have been defined. In this chapter, we can learn them from the I/O system aspect rather than from the file system. Standard input and standard output are I/O devices. So they have their own special files. By default, the terminal keyboard is the standard input, and the terminal screen is the standard output. As running a command is the typical mission for the UNIX shell, the terms of input and output are defined according to a running command. That is, the standard input is where a command receives its input data from while the standard output is the place where a command sends its output data to. It is also helpful for readers to understand the redirection and pipe concepts in the rest of this chapter.

7.1.2 Standard Input, Output and Error Files

When UNIX commands are executed, the kernel opens three standard files: stdin is the standard input file, stdout is the standard output file, and stderr is the standard error file. By default, each of the commands takes its input from the standard input and sends the results to the standard output. These standard files are attached to the terminal where the user types in commands. When a command is running on the UNIX shell, by default, the command input comes from the terminal keyboard and its output and error messages go to the terminal screen (or a terminal window). The command reads input from stdin file and sends its output and error messages to stdout and stderr files, respectively. The illustrative relationships between standard I/O and standard files when running a command are shown in Figure 7.1.

Mentioned in Section 6.5, every open file in UNIX file system has a file descriptor. And standard files have their file descriptors, too. They are 0 for stdin, 1 for stdout, and 2 for stderr. Figure 7.2 shows the logical relationship of the file descriptors, standard files and standard I/O.

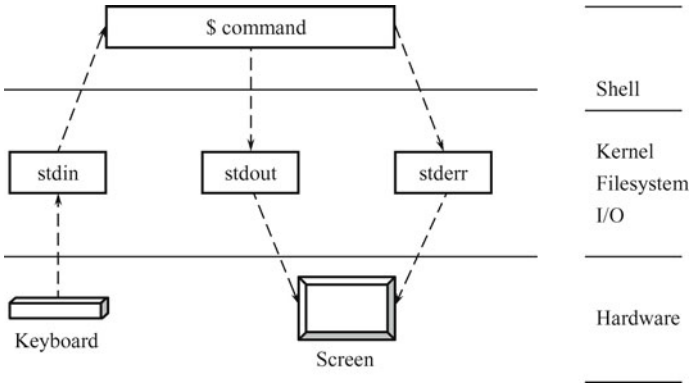

Fig. 7.1 Illustrative relationships between standard files and standard I/O in command default execution.

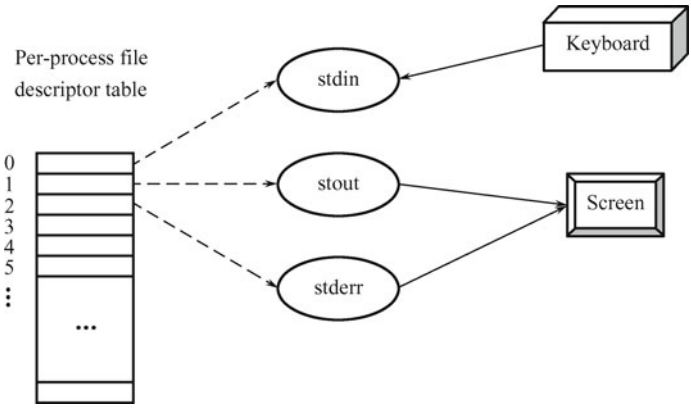

Fig. 7.2 The logical relationship of the file descriptors, standard files and standard I/O.

7.2 Input Redirection

In UNIX, input redirection means to let a command input direct to a file rather than to standard input – a terminal keyboard. So once done, the redirected command receives the input from the specified file not from the terminal keyboard. There are two different ways to do input redirection. One

is to use the less-than symbol (<); the other is to use the < combined with the file descriptor of the standard input file, stdin. They will be discussed, respectively. The effect of the input redirection is illustrated in Figure 7.3.

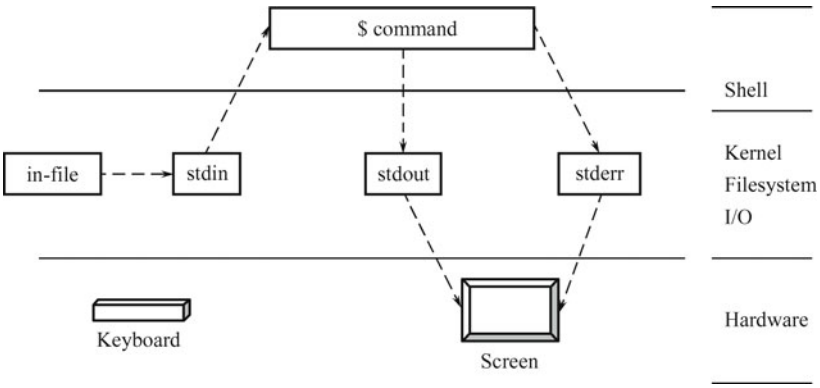

Fig. 7.3 Illustrative relationships between standard files and standard I/O when an input-redirected command executing.

7.2.1 Input Redirection with < Operator

The syntax and function of the input redirection operator < are as follows.

```
$ command < in-file
```

Function: to make the command input come from the in-file rather than the terminal keyboard.

Note: The command, here, should need the input from the keyboard by default. That is, if a command does not need an input naturally, the input redirection operator in the command makes no sense.

For example,

```
$ grep "Hill" < freshman
Miller Hill     1322223434      millerhill8806@sina.com
Gao Hill        1361010101      gaoyuling@hebust.edu.cn
$
```

As known in Chapter 6, when using grep command without any argument, grep takes input from the keyboard. This command is different from the command grep Hill freshman in Section 6.3.6. This command is without argument and takes the file freshman as input, but the command grep Hill freshman has an argument freshman. Without other redirections, the output of this grep command is sent to the standard output – the screen.

Another example is as follows.

```
$ cat < file6
```

```
...
$
```

Known from the previous example, this cat command takes its input from the file file6 and displays the content of file6 on the screen. Even though the result is the same as the command cat file6, two commands are different.

7.2.2 Input Redirection with File Descriptor

Described above, the file descriptor of the standard input file is 0. This integer number can be used to redirect a command input. Most of UNIX shells allow to use file descriptors to open files and associate file descriptors, but the C shell does not use file descriptors with redirection operators. Thus, standard input can be redirected by using the 0< operator in most shells but not in C shell.

The syntax and function of the 0< operator are as follows.

```
$ command 0< in-file
```

Function: to make the command input come from the in-file rather than the terminal keyboard.

Note: The command, here, should also need the input from the keyboard by default.

Two examples in the previous section can be written down as the following and their effects are the same as the previous two.

```
$ grep "Hill" 0< freshman
Miller Hill         1322223434      millerhill8806@sina.com
Gao Hill            1361010101      gaoyuling@hebust.edu.cn
$ cat 0< file6
...
$
```

As these two commands have the same functions as the ones in the previous section, the previous are simpler, but these two are clearer.

7.3 Output Redirection

As input redirection, output redirection means to let a command output direct to a file rather than to standard output — a terminal screen. So once done, the redirected command sends the output to the specified file not to the terminal screen. There are also two different ways to do output redirection. One is to use the greater-than symbol (>); the other is to use the > combined with the file descriptor of the standard output file, stdout. They will be discussed, respectively. The effect of the output redirection is illustrated in Figure 7.4.

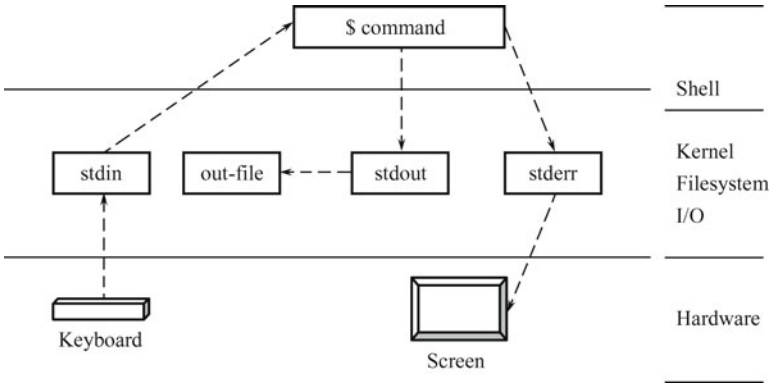

Fig. 7.4 Illustrative relationships between standard files and standard I/O when an output-redirected command executing.

7.3.1 Output Redirection with > Operator

The syntax and function of the output redirection operator > are as follows.

```
$ command > out-file
```

Function: to make the command output go to the out-file rather than the terminal screen.

For example,

```
$ grep Hill freshman > gfreshman
$ cat < gfreshman
Miller Hill          1322223434      millerhill8806@sina.com
Gao Hill             1361010101      gaoyuling@hebust.edu.cn
$
```

With the output redirection, the output of this grep command is sent to the gfreshman file. So the result cannot be seen on the screen. But it can be displayed on the screen by using the cat command as shown.

Another interesting example is as follows.

```
$ cat > file7
... typing the content of file7
CTRL-D
$
```

Remember that the cat command sends its output to standard output – the display screen by default. The standard input of this command is still the keyboard. Therefore, when this command is executed, it creates a file called file7 and waits for the user to type in on the keyboard. When finishing the text of file7, press Ctrl-D in the first column of a new line to quit the cat command. If the file7 exists, by default it will be overwritten. Now, another

way to create a new file has been learned.

7.3.2 Creating a File with Output Redirection

Many methods to create a file have been learned in the previous chapters, such as using a text editor, using the copy command, etc. In this section, a new method to create a small text file with output redirection will be introduced in detail.

Assuming that a file for a work diary, named td-workdiary is needed to create, the steps are given as follows.

```
$ cat > td-workdiary
Read computer books 30 pages in the morning
Do the science research in laboratory in the afternoon
Write work notes in the evening
CTRL-D
$
```

The first part is to create the td-workdiary file by using the cat command without an argument and with output redirection to the td-workdiary file. The cat command without an argument takes the input from the keyboard. Type in some lines of text and terminate the command by pressing CTRL-D on a new line. Then continue the following steps.

```
$ date > td-time
$
```

The second part is to make the td-time file store the time by using the date command with output redirection to the td-time file.

```
$ cat td-time td-workdiary > myworkdiary
$ cat myworkdiary
Tues Aug 19 08:04:56 GMT 2008
Read computer books 30 pages in the morning
Do the science research in laboratory in the afternoon
Write work notes in the evening.
$
```

The third part is first to create the myworkdiary file by using the cat command with two arguments (td-time and td-workdiary) and with output redirection to the myworkdiary file. Then display the content of the myworkdiary file by using the cat command. Now, the work diary, named myworkdiary, is created.

7.3.3 Output Redirection with File Descriptor

Described above, the file descriptor of the standard output file is 1. This integer number can be used to redirect a command output. By using file

descriptor, standard output can be redirected by using the 1> operator in most shells.

The syntax and function of the 1> operator are as follows.

```
$ command 1> out-file
```

Function: to make the command output go to the out-file rather than the terminal screen.

Two examples in the previous section can be rewritten down as the following and their effects are the same as the previous two.

```
$ grep Hill freshman 1> gfreshman
$ cat 0< gfreshman
Miller Hill         1322223434      millerhill8806@sina.com
Gao Hill            1361010101      gaoyuling@hebust.edu.cn
$
$ cat 1> file7
... typing the content of file7
CTRL-D
$
```

As input redirection, the previous two commands are simpler, but these two are more obvious.

7.4 Appending Output Redirection

By default, output and error redirections overwrite contents of the specified files. If a user does not expect to overwrite the previous contents of the destination files, UNIX provides a solution to deal with this problem — by using the appending redirection operator >>.

7.4.1 Appending Output Redirection with >> Operator

The syntax and function of the >> operator are as follows.

```
$ command >> out-file
```

Function: to make the command output go to the out-file rather than the terminal screen and append the new content to the end of the out-file if it is existent.

For example,

```
$ grep Hill freshman >> gfreshman
$ cat 0< gfreshman
Miller Hill         1322223434      millerhill8806@sina.com
Gao Hill            1361010101      gaoyuling@hebust.edu.cn
Miller Hill         1322223434      millerhill8806@sina.com
Gao Hill            1361010101      gaoyuling@hebust.edu.cn
$
```

The result of the grep command is interesting. In the gfreshman file, there are two same lines for "Miller Hill" and "Gao Hill". That result is assuming that this grep command is done after the previous grep command in the last section. This grep command does not overwrite the previous content of the gfreshman file but appends the new contents at the end of the file.

7.4.2 Appending Output Redirection with the File Descriptor

The syntax and function of the 1>> operator are as follows.

```
$ command 1>> out-file
```

Function: to make the command output go to the out-file rather than the terminal screen and append the new content to the end of the out-file if it is existent.

Note: As the > operator, the default effect of the >> operator is the output redirection. So the result of a command with the 1>> operator is equivalent to the one with the >> operator. But file descriptor 2 can be used to append errors to a file, which will be discussed later in this chapter.

The example in the previous section can be rewritten down as follows and their effects are the same.

```
$ grep Hill freshman 1>> gfreshman
$ cat 0< gfreshman
Miller Hill       1322223434      millerhill8806@sina.com
Gao Hill          1361010101      gaoyuling@hebust.edu.cn
Miller Hill       1322223434      millerhill8806@sina.com
Gao Hill          1361010101      gaoyuling@hebust.edu.cn
$
```

7.5 Standard Error Redirection

As output redirection, standard error redirection means to let the error caused by the command execution direct to a file rather than to standard output – a terminal screen. Once done, the redirected command sends the execution error to the specified file not to the terminal screen. As the standard error file and standard output file are both associated with the terminal screen by default, the standard error is redirected only by using the > operator combined with the file descriptor 2 of the standard error file, stderr. The effect of the standard error redirection is illustrated in Figure 7.5.

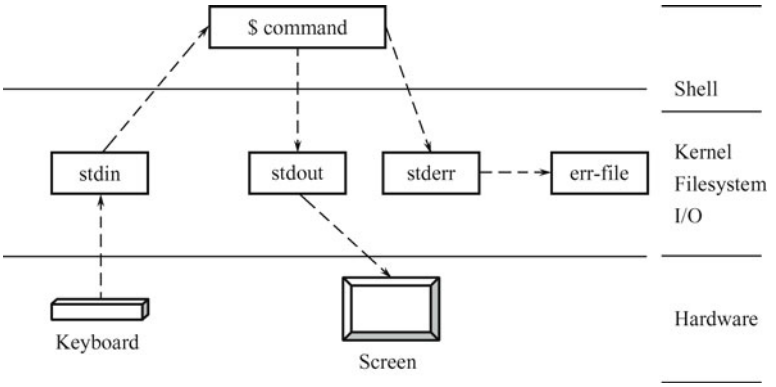

Fig. 7.5 Illustrative relationships between standard files and standard I/O when an error-redirected command executing.

7.5.1 Error Redirection by Using File Descriptor

As mentioned above, most of UNIX shells allow to use file descriptors to open files and associate file descriptors, but the C shell does not use file descriptors with redirection operators. The standard error redirection in the C shell will be discussed in the next section. In most UNIX shells, standard error output can be redirected by using the 2> operator.

The syntax and function of the 2> operator are as follows.

```
$ command 2> err-file
```

Function: to make the command execution error go to the err-file rather than the terminal screen.

For example,

```
$ grep Hill freshman 2> efreshman
Miller Hill       1322223434      millerhill8806@sina.com
Gao Hill          1361010101      gaoyuling@hebust.edu.cn
$
```

Without the output redirection, the output of this grep command is sent to the terminal screen. If efreshman does not exist, it is created; otherwise, it is overwritten.

Another example,

```
$ sort encyclopedia > encyclopedia.st 2> /dev/null &
[3] 18801
$
```

Remember this is the example in Section 4.5.1. This command sorted a huge file named encyclopedia, sent the sorting result into the file encyclopedia.st, and would send the error messages into the file /dev/null if there

were errors while the command running. And 2> /dev/null means make the standard error redirect to the file /dev/null. The /dev/null is an interesting file, and anything goes into it will disappear. This command is running at the background.

However, sometimes, it is useful to keep the standard error attached to the display screen because it can prompt the user instantly if the command is not typed in a correct way, such as typing a nonexistent directory as a command argument, or the user without the read permission for a directory.

7.5.2 Appending Error Redirection by Using File Descriptor

As the 1>> operator, the 2>> operator can avoid to overwrite an existent file.

The syntax and function of the 2>> operator are as follows.

```
$ command 2>> err-file
```

Function: to make the command execution error go to the err-file rather than the terminal screen and append the new content to the end of the err-file if it is existent.

For example,

```
$ grep Hill freshman 2>> efreshman
Miller Hill       1322223434      millerhill8806@sina.com
Gao Hill          1361010101      gaoyuling@hebust.edu.cn
$
```

The output of this grep command is sent to the terminal screen. If efreshman does not exist, it is created; otherwise, it is appended to its end with the new error messages of this command execution.

7.5.3 Error Redirection in C Shell

In the C shell, the input, output, and appending redirection operators (<, >, >>) do work as well as they do in other shells. But these operators combined with file descriptors do not function well in the C shell. In the C shell, there is not an operator to redirect the stderr alone. But the >& operator can be used for both the output and error redirections at the same time.

The syntax and function of the >& operator is as follows.

```
% command >& outerr-file
```

Function: to make the command output and error messages go to the outerr-file rather than the terminal screen.

```
For example,
```

```
% ls − l >& detailandeofd
%
```

This ls command redirects its output and error messages to the detailandeofd file. Note: % sign is the C shell prompt, just like $ is the Bourn shell prompt.

Even though the C shell does not have an operator just for error redirection, the C shell solution is to use the redirection operators in parentheses so that the command can be executed in different subshells (see Section 4.5.2). For example,

```
% (sort encyclopedia > encyclopedia.st ) >& err-sort &
%
```

As mentioned in Chapter 5, when an external command is executed on the shell, the kernel forks a subshell that inherits the standard files of the parent shell and executes the command. The stdout and stderr of the parent shell are redirected to the err-sort first, and then the subshell that is created for a sort in parenteses is redirected the stdout to the encyclopedia.st. Therefore, the command in the parentheses sends its output to the encyclopedia.st file and remains the error messages go into the err-sort file.

In the C shell, the >>& operator can be also used. As the >> operator, the >>& operator can avoid to overwrite an existent file.

The syntax and function of the >>& operator is as follows.

```
% command >>& outerr-file
```

Function: to make the command output and execution error go to the out-file rather than the terminal screen and append the new contents to the end of the out-file if it is existent.

For example,

```
% ls −l >>& detailandeofd
%
```

This ls command appends its output and error messages to the detailandeofd file.

7.6 Combining Several Redirection Operators in One Command Line

In the most UNIX shells, some redirection operators can also be used in one command line. Some combinations will be discussed.

7.6.1 Combining Input and Output Redirections in One Command Line

Input and output redirection operators can be used together by combining in one command. When both input and output operators are used in one command, the effect of the redirected command is illustrated in Figure 7.6.

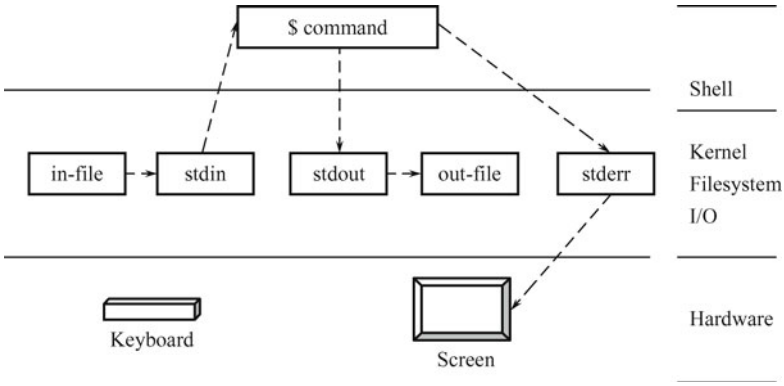

Fig. 7.6 Illustrative relationships between standard files and standard I/O when an input-and-output-redirected command executing.

The syntax and function of combining the input and output redirection operators are as follows.

```
$ command < in-file > out-file
$ command > out-file < in-file
$ command 0< in-file 1> out-file
$ command 1> out-file 0< in-file
```

Function: to make the command input come from the in-file rather than the terminal keyboard and the command output go to the out-file than the terminal screen.

For example,

```
$ grep "Hill" < freshman > gfreshman
$
```

This command is without argument, takes the file freshman as input, and sends its output to the gfreshman file. If the gfreshman file exists, overwrites it; if not, creates it.

Another example is as follows:

```
$ cat < file6 > file7
$
```

This cat command takes its input from the file file6 and sends the content of file6 to the file7 file. Its function seems like the command cp file6 file7, but two commands are slightly different. If the file7 exists, the cat < file6 >file7 command first empties the file7 file and copies the contents of file6 into it.

In this way, the file7 file still has the original file attributes, such as the user permissions for the file. But the cp file6 file7 command makes the file7 file from the inode copy. So the file7 and file6 files are totally the same, not only in their contents but in their attributes. If the file7 does not exist, the two commands will have the same file7.

7.6.2 Combining Output and Error Redirections in One Command Line

The output and error messages of a command can be redirected in one command line.

The syntax and function of combining the output and error redirection operators are as follows.

```
$ command 1> out-file 2> err-file
$ command 2> err-file 1> out-file
```

Function: to make the command output and error messages go to the out-file and err-file, respectively, rather than the terminal screen.

Note: Both of two given commands function the same, and the effect of them is shown in Figure 7.7.

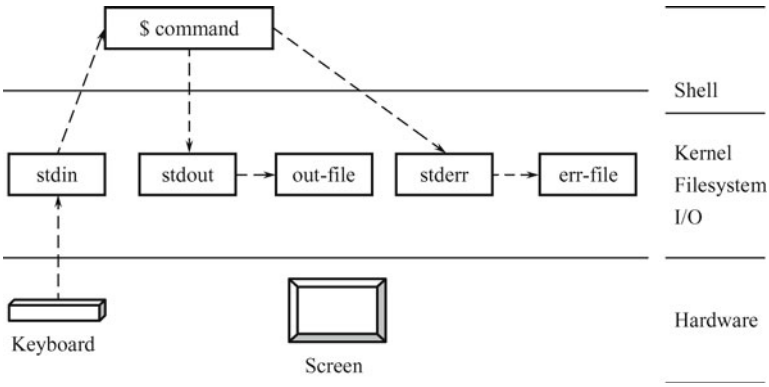

Fig. 7.7 Illustrative relationships between standard files and standard I/O when an output-and-error-redirected command executing.

For example,

```
$ grep "Hill" freshman 1> gfreshman 2> efreshman
$
```

The output of this grep command is sent to the gfreshman file and its error messages are sent to the efreshman file. If gfreshman and efreshman don't exist, they are created; otherwise, overwritten.

To put the output and error messages of a command into one file, use the

following command syntax.

```
$ command 1> outerr-file 2>&1
$ command 2> outerr-file 1>&2
```

Function: to make the command output and error messages go to the same file of outerr-file rather than the terminal screen.

Note: Both of two given commands have the same function – to put the command output and error messages into one file. But the 2>&1 operator in the first command means to copy descriptor 1 to descriptor 2, and the 1>&2 operator in the second command means to copy descriptor 2 to descriptor 1. Usually a UNIX shell interprets and executes a command from left to right. Therefore, if one term is dependent on another in a command, be careful with this rule. In this case, as one descriptor is a copy of another, redirections must be specified in left-to-right order. That is, it is necessary to let the copy source goes before the copy destination. In other words, to use 2>&1, it is necessary to let the 1> operator goes first. The effect of them is shown in Figure 7.8.

For example:

```
$ grep "Hill" freshman 1> gefreshman 2>&1
$
```

or

```
$ grep "Hill" freshman 2> gefreshman 1>&2
$
```

These two commands send the output and error messages into the same gefreshman file, if some error produced.

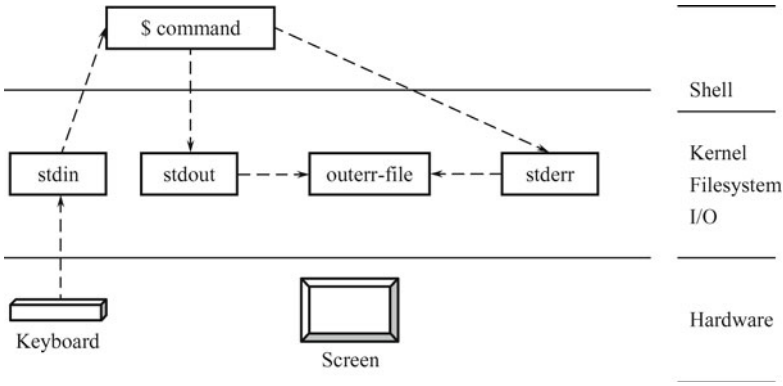

Fig. 7.8 Illustrative relationships between standard files and standard I/O when another output-and-error-redirected command executing.

7.6.3 Combining Input, Output and Error Redirections in One Command Line

The input, output and error messages of a command can also be redirected in one command line.

The syntax and function of combining the input, output and error redirection operators are as follows.

```
$ command 0< in-file 1> out-file 2> err-file
```

Function: to make the command input from the in-file file rather than from the terminal keyboard, and make the command output and error messages go to the out-file and err-file, respectively, rather than the terminal screen.

Note: In this syntax, the order of the redirection operators is also important. If the in-file does not exist, the error message will be sent to the terminal screen because the error redirection has not been done. The effect of this syntax is shown in Figure 7.9.

For example:

```
$ grep "Hill" 0> freshman 1> gfreshman 2> efreshman
$
```

This command gets its input from the freshman file, sends the output and error messages into the gefreshman file and efreshman file, respectively, if some error produced.

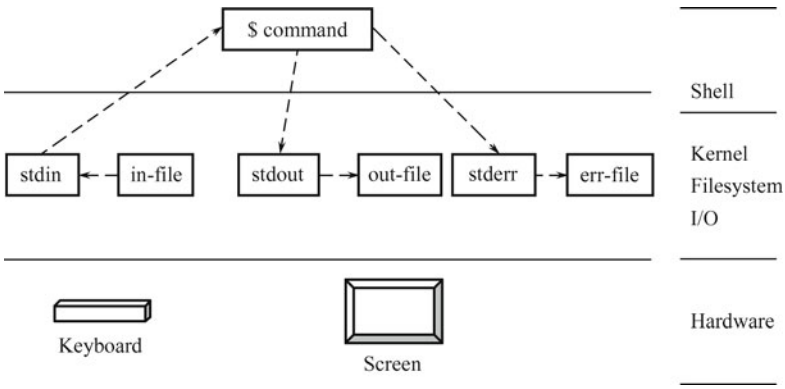

Fig. 7.9 Illustrative relationships between standard files and standard I/O when an input-output-and-error-redirected command executing.

7.6.4 Combining Appending Redirection with Other Redirections in One Command Line

The appending redirection >> operators can also be combined together with themselves or other redirection operators in one command line. In this way, it can avoid overwriting some files by accident. The following are given some of examples.

```
$ grep Hill freshman 1>> gfreshman 2>> efreshman
$
```

This grep command appends its output to the gfreshman file and its error messages to the efreshman file.

```
$ ls -la 1>> lsoutput 2> lserror
$
```

This ls command appends its output to the lsoutput file and sends its error messages to the lserror file. If the lserror file does not exist, create it; if it exists, overwrite it.

```
$ cat file1 file2 file3 >> file4 2> cerror
$
```

This cat command adds file1, file2, and file3 to the end of the file4 file. The file1, file2, and file3 files are the input of the command. The file4 file is the output destination of the command. And the error messages are redirected to the cerror file. If the cerror file does not exist, create it; if it exists, overwrite it.

In the C shell and Korn shell, it can also avoid overwriting an existent file to set the noclobber option. If putting this set command in the. profile or. cshrc, the noclobber option can be set permanently. The. profile file has been discussed in Chapter 2.

If the noclobber variable is set, the command cat filel file2 file3 > file4 generates an error message if file4 exists. If file4 does not exist, it is created and the files of filel, file2, and file3 are copied into it. When noelobber is set, the command cat filel file2 file3 >> file4 functions well if file4 exists while an error message is generated if file4 does not exist.

However, the >!, >>!, and >>&! (each operator with an exclamation mark) operators can override the effect of the set noelobber variable. Therefore, even if the noclobber variable is set and file4 exists, the command cat filel file2 file3 >! file4 copies the files of filel, file2, and file3 into it.

7.7 UNIX Pipes and Filters

As mentioned above, the input or output of a command can be redirected to a file by using the input or output redirection operators. It is also possible in UNIX to connect the output from one command with the input to another

command. Hence, the output of the first command becomes the input of the second command. In other words, the UNIX allows the stdout of one command to be connected to the stdin of another command.

7.7.1 Concepts of Pipe and Filter

To make a connection, the pipe operator (a vertical bar |) can be used. The syntax of the pipe operator is as follows.

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

Function: to make the command1 output connect to the command2 input, and the command 2 output connect to the command3 input.

Note: In fact, the command list can go on and on for several commands in one command line if the connections between each pair of all the commands are meaningful. The effect of this syntax is shown in Figure 7.10.

When a pipe is set up between two commands, the standard output of the command to the left of the pipe operator becomes the standard input of the command to the right of the pipe operator. Any two commands can form a pipe as long as the first command writes to standard output, and the second command reads from standard input.

If a command takes its input from another command, does some operation on that input, and writes the result to the standard output, which may be piped to yet another command, the command is called a filter.

Most UNIX commands can be used to form pipes. In the following sections, some commands that can be used as filters will be discussed.

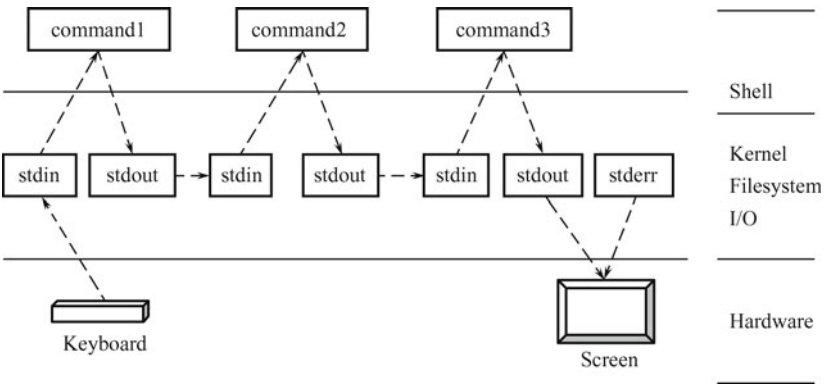

Fig. 7.10 Illustrative relationships between standard files and standard I/O when piped commands executing.

7.7.2 Examples of Pipes and Filters

In UNIX, pipes and filters are usually used to perform some complicated tasks that cannot be done with a single command. Commonly, some commands, such as cat, grep, lp, pr, sort, tr, and wc, can be used as filters.

For example,

```
$ ls -la
Total 32
drwxr-xr-x    2   wang   project   512    Apr   15   15:11   .
drwxr-xr-x    2   admi   admi      512    Apr   12   15:01   ..
-rw-r--r--    2   wang   project   136    Jan   16   11:48   .exrc
-rw-r--r--    2   wang   project   833    Jan   16   14:51   .profile
-rw-r--r--    1   wang   project   797    Jan   16   15:02   file1
-rw-r--r--    1   wang   project   251    Jan   16   15:03   file2
drwxr-xr-x    2   wang   project   512    Apr   15   15:11   text
...
$ ls -la | grep "wang"
drwxr-xr-x    2   wang   project   512    Apr   15   15:11   .
-rw-r--r--    2   wang   project   136    Jan   16   11:48   .exrc
-rw-r--r--    2   wang   project   833    Jan   16   14:51   .profile
-rw-r--r--    1   wang   project   797    Jan   16   15:02   file1
-rw-r--r--    1   wang   project   251    Jan   16   15:03   file2
drwxr-xr-x    2   wang   project   512    Apr   15   15:11   text
...
$
```

The above command line executes ls -la command to list the working directory. The output is piped to the grep "wang" command that only displays on the terminal screen the lines that contain the string wang, that is, the files whose owner is wang.

Second example:

```
$ ls -la | more
......
$
```

The difference between this command line and the ls -la command is that this command line can display the information of the working directory one screen at a time because the more command is a pager. If the list of the directory information is so long that it covers several screens, this command line will be helpful to see the detailed information one screen by one screen.

7.7.3 Combining Pipes and I/O Redirections in One Command Line

Two or more commands can be connected with pipes alone. And also, pipes and I/O redirections can be used in one command line. The related examples will be given in the next section. But it is impossible to realize to redirect the output of a command to a file and connect it to the input of another

command at the same time by just using pipes and I/O redirections in one command line. But it can be done with the help of the tee operator. Its syntax is as follows:

```
$ command1 | tee file1 file2 file3 |command2
```

Function: to make the command1 output connect to the tee input, and the tee command send its input to the files file1 through file3 and to the command3 input.

Note: The tee operator tells the shell to send the output of a command to one or more files, as well as to another command at the same time. And the file list after the tee operator can go on and on for several files in one command line. The effect of this syntax is shown in Figure 7.11.

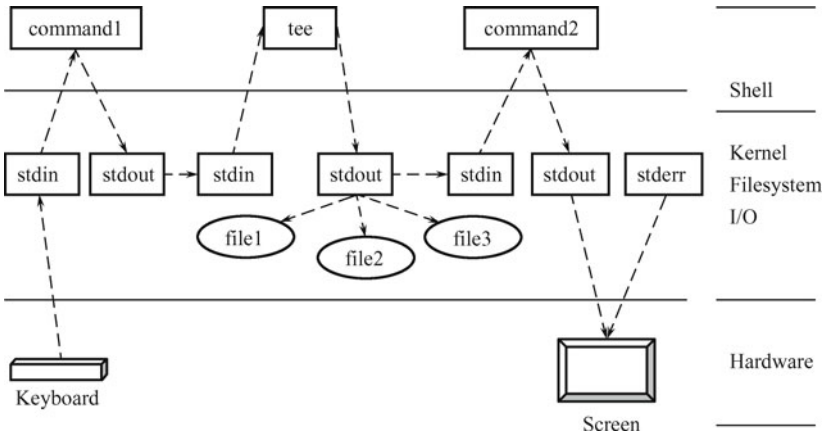

Fig. 7.11 Illustrative relationships between standard files and standard I/O when executing commands with tee.

The command is executed and its output is stored in the files of file1, file2, and file3, and sent to command2 as its input. The following is an example.

```
$ grep -v "David" freshman | tee fresh1 fresh2 | sort +1 -2 -b > fresh3
$
```

In this command line, the output of the grep -v “David” freshman command is piped to the input of tee, which puts copies of these lines in the files of fresh1 and fresh2, and sends them to the input of the sort +1 -2 -b > fresh3 command. Finally, the result of the sort command is sent to the fresh3 file. Thus, those lines in the freshman file that do not contain David are saved in the files of fresh1 and fresh2. If later on there are two different contents for the two files, respectively, it is useful to have these two copies.

7.7.4 Practical Examples of Pipes

In the examples of the previous section, pipes are implemented in the main memory but not on disk, which is an I/O device. To see the difference clearly, here give another example that is equivalent to the second example of Section 7.7.2.

```
$ ls −la > rsofls
$ more rsofls
......
$
```

Known before, the I/O access can slow down significantly the execution of the commands by using the read and write system calls for the rsofls file on disk. But the real situation is not that bad for the I/O access because of the cache buffer for the disk access of the UNIX kernel.

Second example: Here, by using the example of Section 6.3.5 in Chapter 6 as another example, do the following process.

```
$ cat freshman
Jones David          1358999998      jonesdavid@hebust.edu.cn
Huo Song             1341212121      huosong88@126.com
Dang Ailin           1500011223      dangailin@163.com
Jones David          1357777888      jdavid87@sina.com
Liang Fanghua        1310000555      liangfh88@163.com
Miller Hill          1322223434      millerhill8806@sina.com
Gao Hill             1361010101      gaoyuling@hebust.edu.cn
Nan Xee              1393219123      nanxee1987@aaa.com
Wang Feng            1311212123      wangfeng89@hebust.edu.cn
$ cat freshman | grep -v "David" | sort +1 -3 -b
Dang Ailin           1500011223      dangailin@163.com
Liang Fanghua        1310000555      liangfh88@163.com
Wang Feng            1311212123      wangfeng89@hebust.edu.cn
Miller Hill          1322223434      millerhill8806@sina.com
Gao Hill             1361010101      gaoyuling@hebust.edu.cn
Huo Song             1341212121      huosong88@126.com
Nan Xee              1393219123      nanxee1987@aaa.com
$
```

The first cat freshman command displays the lines in the freshman file. In the second command line, the output of the cat freshman command is sent to the input of the grep -v “David”, which should display those lines that do not contain David in the freshman file; then the output of the grep -v “David” is piped to the input of the sort +1 -3 -b command, which does sorting according to the first sort key (+1 -3), given name (Field 1) and cell-phone number, and displays the result on the screen.

Now, the power of the pipe can be seen, which is to perform the functions of three commands in one command line by forming a pipeline of commands and is equivalent to the following command sequence.

```
$ cat freshman > tempfresh1
$ grep -v "David" tempfresh1 > tempfresh2
$ sort +1 -3 −b tempfresh2
$ rm tempfresh1 tempfresh2
```

```
$
```

The command line with pipes does not need any disk file as a temporary storage place for processing the data, but the command sequence uses two temporary disk files and four or more disk I/O (read and write) operations depending on the file system.

Third example: I/O redirection and pipes can be used in one command line. Here is the previous example with different processes.

```
$ cat freshman
Jones David          1358999998      jonesdavid@hebust.edu.cn
Huo Song             1341212121      huosong88@126.com
Dang Ailin           1500011223      dangailin@163.com
Jones David          1357777888      jdavid87@sina.com
Liang Fanghua        1310000555      liangfh88@163.com
Miller Hill          1322223434      millerhill8806@sina.com
Gao Hill             1361010101      gaoyuling@hebust.edu.cn
Nan Xee              1393219123      nanxee1987@aaa.com
Wang Feng            1311212123      wangfeng89@hebust.edu.cn
$ cat < freshman | grep -v "David"
Huo Song             1341212121      huosong88@126.com
Dang Ailin           1500011223      dangailin@163.com
Liang Fanghua        1310000555      liangfh88@163.com
Miller Hill          1322223434      millerhill8806@sina.com
Gao Hill             1361010101      gaoyuling@hebust.edu.cn
Nan Xee              1393219123      nanxee1987@aaa.com
Wang Feng            1311212123      wangfeng89@hebust.edu.cn
$
```

In the second command line, the input of the cat command is from the freshman file and the output of the cat < freshman command is piped to the input of the grep -v "David", which displays on the screen those lines that do not contain David in the freshman file.

Forth example: Here is also the previous example with different processes.

```
$ cat < freshman | grep -v "David" | sort +1 -3 -b > stfreshman
$
```

In this command line, the input of the cat command is from the freshman file and the output of the cat < freshman command is piped to the input of the grep -v "David"; then the output of the grep -v "David" is piped to the input of the sort +1 -3 -b command, which redirects its output to the stfreshman file.

7.7.5 Pipes in C Shell

In the C shell, the | operator can connect the output of one command to the input of another one. Otherwise, it also allows the output and error messages of one command to be attached to the input of another command with the |& operator. Its syntax is displayed as follows.

```
% command1 | command2
```

Function: to connect the command1's output to the command2's input.

```
% command1 |& command2
```

Function: to let the command1's output and error messages sent to the command2 as its input.

For example:

```
% cat freshman | sort -1 +2 -b > sfreshman
%
```

The output of the cat command is attached to the input of the sort command. Hence, the output of the cat command is sent to the sort command as its input.

Another example:

```
% cat freshman |& sort -1 +2 -b > sfreshman
%
```

The output and error messages of the cat command are attached to the input of the sort command. Hence, the sort command reads the output of the cat command, or any error produced by its executing, such as, if freshman does not exist.

Third example:

```
% cat freshman | grep -v "David" |& sort +1 -2 -b >stfreshman
%
```

In this command line, the output of the cat command is sent to the input of the grep command. Then, the output and error messages of the grep command are piped to the sort command as its input.

7.7.6 Named Pipes

For the kernel, a pipe is an area in the kernel memory that allows two processes, which are running on the same computer system and related to each other, to communicate with each other. This has been discussed in Section 6.2.5. Thus, a pipe can be used in an inter-process communication.

The pipe differs from a regular file in that the data in a pipe is temporary: once data is read from a pipe, it cannot be read again. Also, the data is read in the order that it was written to the pipe. In other words, the pipe communication is one-way. For example, in the command line of ls -la | grep "wang", the output of ls -la is read by grep "wang" as input. Here, the one-way communication is from ls to grep. For a bidirectional communication between processes, two pipes are needed. This cannot be performed at the shell level, but can be done by using the pipe system call in the C or C++ programming.

A named pipe is a file that allows two processes to communicate with each other if the processes are on the same computer, but do not have to be

related to each other.

The kernel stores data in a named pipe in the same way it stores data in an ordinary file, except that it uses only the direct address, not the indirect address.

Processes that are communicating with pipes usually have the same parent process while processes that are communicating with named pipes can be independently on one system.

The UNIX operating system provides the mkfifo command to create named pipes. Its syntax is as follows.

```
$ mkfifo [option] file[s]
```

Function: to create named pipes; file[s] can be several files with pathnames.

Common options:

-m mode: to set access permissions for named pipes to 'mode'; the 'mode' is set in the same way as with the chmod command (see Section 9.5.1), such as 777 for giving all users the read, write and execute permissions to the created named pipes.

Note: As the mkfifo command can create the named pipes alone, it means the commands that use the named pipes do not necessarily exist and execute in one command line. A named pipe is on disk like a file with a filename. Hence, it can be used like a file, such as accepting the operations of the open, close, read, and write system calls. The execution of the mkfifo command causes the mknod system call that can be used to create a new special file, directory, or named pipe.

For example:

```
$ mkfifo firstfifo
$ mkfifo —m 666 secondfifo
$ ls -l
Total 42
drwxr-xr-x  2  wang   project   512   Apr   15   15:11   .
drwxr-xr-x  2  admi   admi      512   Apr   12   15:01   ..
...
prw-r--r--  1  wang   project   0     Jan   18   15:02   firstfifo
prw-rw-rw-  1  wang   project   0     Jan   18   15:03   secondfifo
...
$
```

The first mkfifo command has created a named pipe, called firstfifo, with default permissions. The second command has created a named pipe, called secondfifo, with read and write permissions for all the users and execute permission for nobody. The ls -l command has displayed the access permissions of the two named pipes.

When two commands use the named pipe called firstfifo to communicate with each other, they can run separately as the following sequence:

```
$ command1 < firstfifo &
$ command2 > firstfifo
```

Note: The first command is put in the background so that the shell

prompt can be appear on the screen again and the second command can be typed in. When the first command is typed in, it is blocked because the fistfifo is empty. When the output of the second command is sent to the firstfifo, the first command starts to read from the firstfifo and process the data in it. The output of the first command displays on the screen. The effect of the named pipe is shown in Figure 7.12.

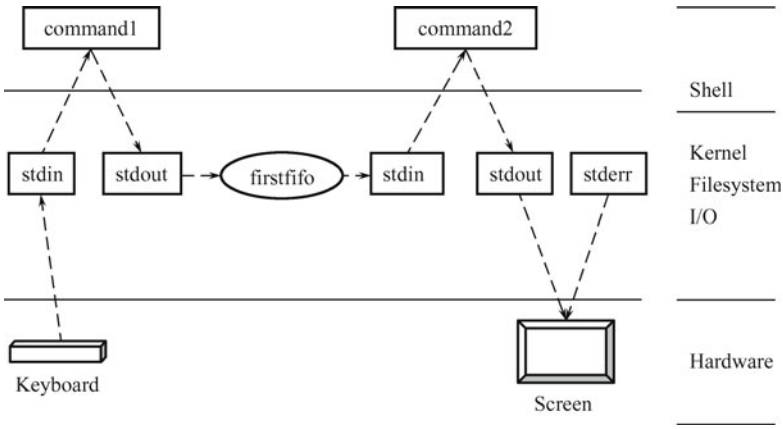

Fig. 7.12 Illustrative relationships between standard files and standard I/O when executing commands with a named pipe.

For example:

```
$ cat firstfifo &
$ more secondfifo &
$ grep -v "David" freshman |tee firstfifo secondfifo | sort +1 -3 -b
......
$
```

The first two commands, cat and more, are blocked until the firstfifo and secondfifo are written. The grep command sends those lines that do not contain David in the freshman file to the tee, which redirects its output to the two named pipes as well as sends it to the sort command. Thus, the sort, cat, and more commands display their outputs on the screen, respectively. As the scheduling of the processes in the system is dependent on the kernel, the outputs of the commands may not appear on the screen in an expected order, but all of their outputs will display on the screen finally.

When a named pipe is no longer needed, it can be removed by the way to remove an ordinary file — using the rm command to remove it from the file system. For example, to remove the firstfifo and secondfifo, use the following command:

```
$ rm firstfifo secondfifo
$
```

If the ls command is used to check, the two named pipes will disappear from the list.

7.8 UNIX Redirection and Pipe Summary

In fact, many commands can be combined together with pipes and redirection operators to perform some complex functions. For readers, it needs some time and practices to know how powerful the pipes and redirection operators are. And also, it is not just for those commands given in this chapter that can combine with pipes and redirection operators. In UNIX, there are some more, especially in the networks.

In this section, a summary of UNIX redirection and pipe operators in the Bourne and C shells is given in Table 7.1. As in the Korn shell, most of the redirection and pipe operators are the same as in the Bourne shell, they are not shown in the table.

Table 7.1 Important redirection and pipe operators in Bourne and C shells

Operator	Use in Bourne Shell	Use in C Shell
< file	Input redirection	Input redirection
0< file	Input redirection	
> file	Output redirection	Output redirection
1> file	Output redirection	
>> file	Append standard output to the file	Append standard output to the file
1>> file	Append standard output to the file	
2> file	Error message redirection	
2>> file	Append standard error to the file	
>& file		Output and error message redirection
>>& file		Append output and error message to the file
m>& n	Copy file descriptor n to file descriptor m	
>! file		Output redirection with ignoring noclobber
>>! file		Append standard output to the file with ignoring noblobber; if the file is inexistent, create it
>>& ! file		Append standard output and error message to the file with ignoring noblobber; if the file is inexistent, create it
comm1 | comm2	Connect the output of the comm1 command to the input of the comm2 command	Connect the output of the comm1 command to the input of the comm2 command

Continued

Operator	Use in Bourne Shell	Use in C Shell
comm1 | & comm2		Connect the output and error message of the comm1 command to the input of the comm2 command

7.9 I/O System Implementation in UNIX

In a computer system, there are peripheral devices attached to it, such as disks, terminals, keyboards, printers, and networks. The UNIX kernel modules that control devices are known as device drivers.

7.9.1 I/O Mechanisms in UNIX

In UNIX, devices are divided into two types, block devices and character devices (Bach 2006; Nelson et al 1996). Block devices, such as disks, are random access storage devices. Character devices include all other devices such as terminals and network media.

A block special file consists of a sequence of numbered blocks. The key property of the block special file is that every block can be accessed with its own address. In other words, a process can open a block special file and read, for example, block 1024 without going through blocks 0 to 1023 first. That is also the meaning of random access storage.

Character special files are usually used for devices with which human beings interact with the computer system directly. Compared to computers, human beings act far slower. Bridging between human beings and computers, these devices input or output a character stream. Keyboards, printer, networks, mice, and most of the other I/O devices belong to these devices.

In UNIX, these I/O devices are integrated into the file system, so the user interface to devices needs to go through the file system to control the I/O devices. Every device has a name that looks like a filename and is accessed like a file. The device special file also has an inode and occupies a node in the directory tree of the file system. The special file is different from regular files by the file type stored in its inode, either "block special" or "character special", corresponding to device it represents.

In the UNIX file system, each I/O device is assigned a pathname, usually in /dev. For example, a disk may be /dev/hd1, a printer may be /dev/lp, a terminal may be /dev/tty2, and the network may be /dev/net.

System calls for regular files, such as open, close, read, and write, have an appropriate meaning for devices. Processes can open, read, and write

special files in the same way as they do regular files. So for users, no special mechanism is needed for doing I/O.

Of course, each device driver does not necessarily support every system call interface. I/O devices have their own system calls that are not applicable to regular files. For example, the system calls of enabling and disabling of character echoing, and conversion between carriage return and line feed are specified for some character special files.

In UNIX, I/O is operated by a set of device drivers. The function of the drivers is to isolate the rest of the system from the specific controls on the hardware. By providing standard interfaces between the drivers and the rest of the operating system, most of the I/O system can be put into the machine-independent part of the kernel, which is important for operating system portability.

UNIX may contain one disk driver to control all disk drives connected to the system and one terminal driver to control all terminals in the system. That is the one-to-one relationship between device drivers and device types. The device driver should distinguish among many devices it controls, that is, output for one terminal must not be sent to another.

In other words, associated with each special file is a device driver that handles the corresponding device. Each driver has what is called a major device number that serves to identify it and is related to the device type. If a driver controls several disks of the same type, each disk has a minor device number that identifies it and is used to distinguish it among devices of one device type. Together, the major and minor device numbers uniquely identify a single I/O device.

The driver interface that the kernel builds between the file system , and the device drivers includes two tables: the block device switch table (bdevsw) and character device switch table (cdevsw). The former is for block special files; the latter for character special files. Figure 7.13 shows the driver interface between the file system and the device drivers. Figure 7.14 displays the simplified tables of bdevsw and cdevsw.

For system calls that use file descriptors (such as read and write system calls), the kernel follows pointers from the user file descriptor to the kernel file table and inode, where it checks out the file type. From the inode, it also gets the major and minor numbers. With the file type, the kernel knows which table (the bdevsw or cdevsw table) should be accessed. Furthermore, the kernel uses the major number as an index into the appropriate table (bdevsw or cdevsw), and calls the corresponding driver according to the system call being made, passing the minor number as a parameter.

In the user view, when a user accesses a special file through system calls, the file system examines whether it is a block special file or a character special file and determines its major and minor device numbers. The major device number is used to index into the entry of the corresponding table, bdevsw table for block special files or cdevsw table for character special files. The entry located contains pointers to the programs to call for opening the

device, reading the device, writing the device, and so on. The minor device number that tells which unit of this type of devices is chosen is passed as a parameter.

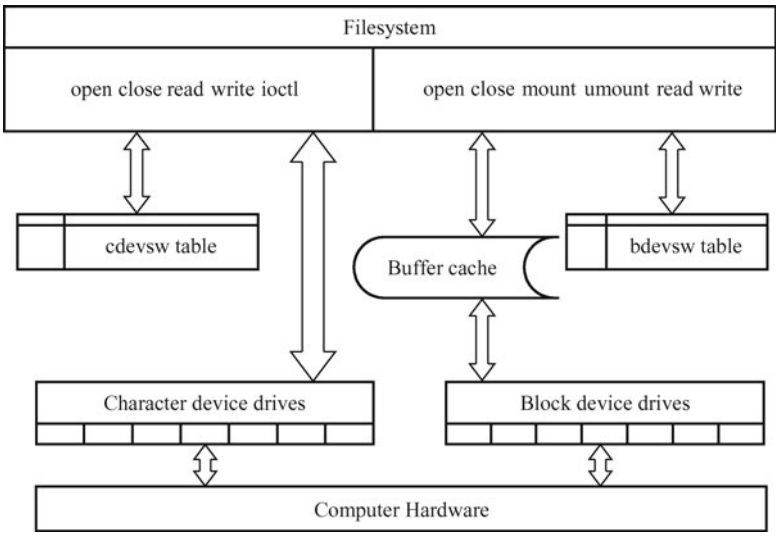

Fig. 7.13 The UNIX interface between the file system and the device drivers.

	Block device switch table		
	open	close	strategy
0			
1			
2			
3			

	Character device switch table				
	open	close	read	write	ioctl
0					
1					
2					
3					

Fig. 7.14 Simplified tables of bdevsw and cdevsw.

Also in UNIX, to add a new device type to the system is just to insert a new entry in one of these tables and to supply the corresponding programs to handle the various operations on the device.

Each driver is divided into two parts. One part runs in user mode with interfaces to the UNIX kernel. The other one runs in kernel mode and interacts with the device. Drivers are allowed to make calls to kernel processes for other operations, such as memory allocation, timer management, DMA control, and so on.

In classical UNIX operating systems, UNIX device drivers had been statically linked into the kernel, so they were all present in memory when the system was booted every time, which was really inflexible.

7.9.2 Block Special Files and Buffer Cache

As the disk special files are the main part of the block special files, the disk drivers will be paid more attention to. The disk driver converts a file system address, including a logical device number and block number, into a particular sector on the disk.

Since I/O operations involve in many mechanical actions and those actions take a lot of time (Bach 2006; Carson et al 1992; Heindel et al 1995; Quarterman et al 1985; Stallings 1998), the goal of block special files is to minimize the number of actual transfers between the memory and I/O devices. To accomplish this goal, UNIX uses a buffer cache between the disk drivers and the file system. On the other hand, system files, commands, and directories are usually reused frequently. If their data blocks are in the buffer cache, it is fast and effective for the kernel to find them directly in the cache when they are referenced because it saves to retrieve them from the disk and makes the cache hit rate high (see Figure 7.13).

Because the accessed block may be or not be in the buffer cache, the driver can get the address in one of two ways. One is that the strategy procedure (see Figure 7.14) uses a buffer from the buffer pool and the buffer header contains the device and block number; the other is that the read and write system calls are passed the logical (minor) device number as a parameter, with which they convert the byte offset saved in the user structure to the appropriate block address. The disk driver uses the device number to identify the physical drive and particular section to be used.

In the UNIX kernel, the strategy system call (see Figure 7.14) is used to transmit data between the buffer cache and a disk device. The strategy system call can queue I/O jobs for a device on a work list or do even more jobs. Drivers can set up data transmission for one physical address or more. The kernel passes a buffer header address to the driver strategy program; the header contains a list of addresses and sizes for transmission of data from or to the device. To the buffer cache, the kernel transmits data from one data address; when swapping, the kernel transmits data from many data addresses. We now discuss buffer headers in detail.

7.9.2.1 Buffer Headers and Lists

The buffer cache consists of many buffers that are managed in several different lists according to their used statuses. Each buffer is identified with its buffer header. The buffer header holds the following information:

- The device number and block number, where the block of a file that occupies the buffer now is on the disk.
- A pointer to where the buffer is located in the physical memory.
- The size of the location of the buffer in the physical memory.
- The memory amount in the location of the buffer that contains the file data.
- The dirty bit, which marks if the contents of the buffer element are mod-

ified.

When the system is booted, chucks of memory pages are specified as the buffer pool. Each page in the buffer pool is initiated as a buffer with a buffer header linked in a list. The size of the buffer pool and the number of buffer headers are relied on the available space in the primary memory.

The buffer headers are used to link the buffers of the buffer cache together so that the kernel can manage them easily. There are usually several different lists to link the buffers.

- Reserved region, which contains blocks that hold the resident part of the system that the kernel accesses frequently.
- Cache list, which holds blocks that are referenced recently and may be reused in the future.
- Assignable list, which collects blocks that have a lower possibility to be reused than ones in the cache list.

The cache and assignable lists are manipulated in the least recently used mechanism. The buffer headers are also hashed with the device and block numbers in the way like the one used in the hash frame queues of the memory management (see Section 5.3.1.2), which can speed up searching a buffer.

Except the above lists, some buffer headers can be empty and without disk blocks associated with yet, which are put in the free list and can be used to make up the buffer insufficiency when needed. And also, some buffers can be in process when some block device operations are being done. These buffers are not in any of the above lists, but in some waiting queue related to some block device.

Since in BSD there are two units of the data blocks on the disk (see Section 6.5.1.2): blocks and fragments, the maximum size of a buffer can be a block size (typically 8192 bytes), and the minimum one is a fragment size (1024 bytes). A buffer header can tell the buffer size, which may be a block size or less-than-eight contiguous fragments. That is, the buffer size can be varied. When a buffer is allocated to a block of a file and found that it cannot fit in the size of the block of the file, some appropriate free buffer has to be taken from the free list, and the previously-allocated buffer will be free and put on the free list.

The free list can also be complemented by the truncate system call, which can be used to reduce the size of a file. When the size of a file is reduced, the buffer that the file occupies will be decreased. For this reason, the kernel will allocate a smaller but appropriate free buffer to the file, and free the older one onto the free list.

7.9.2.2 Read from and Write to Disk Block

When a user process starts to read a file, the kernel first transfers a block on the file system in the buffer cache, and then copies the buffer to the user process's memory space. Similarly, when a user process begins to write a file, the kernel allocates a buffer to the process, and then the user process's

writing data are copied from the user process's memory space into the buffer. Both read and write are not done on the disk directly.

Therefore, when a process reads a block from a disk, the kernel first looks into the cache list to see if or not the block is there. If so, it is taken from there and a disk access is saved. In this way, the buffer cache greatly improves I/O performance.

If the block is not in the buffer cache, the kernel removes a buffer from the assignable list, updates its buffer header with the device number and block number of the block, reads the contents of the block first from the disk into the buffer, and copies from the buffer to where it is needed. If no buffers are in the assignable list, one buffer on the cache list can be chosen to allocate according to the least recently used mechanism. And if the chosen buffer is dirty, it has to write to the disk before being replaced with the new contents. With the least recently used mechanism, whenever a buffer is accessed, it is moved to the head of the list. When a block must be removed from the cache to make room for a new block, the one at the end of the chain is selected.

When the end of a buffer is referenced, the buffer is put on the assignable list because it is assumed that if the reference reaches the end of one block of a file, which is in the buffer now, the next reference may start at the following block of the file more possibly, which may cause a new block to be read in the buffer cache. If the end of the buffer has not been referenced, the buffer is more likely reused in the future so that it is still in the cache list.

The buffer cache also works for writes. When a process writes a block, it writes to the cache, not to the disk, temporarily. And the kernel marks the dirty bit of its buffer header. When the cache fills up, some blocks in the buffer cache must be forced out. If it is marked dirty, the buffer should be written to the disk. So it is added to the queue waiting for the disk I/O and the buffer is put on the assignable list. It is called the delayed write mechanism (Carson et al 1992; Ritchie et al 1974; Stallings 1998).

For the read, the temporary substitution of a disk block operation with a buffer operation cannot make any serious problems when the system crashes. However, for the write, the modified data in the buffer may not be really written to the file system on the disk if the system crashes unexpectedly, which can result in the data in the buffer lost and the data on the disk incorrect. To avoid this problem, the sync system call is periodically invoked to do write back to the disk — all the dirty blocks are usually written to the disk every 30 seconds. If all the blocks of a file have to be written to the disk at a time, for instance, in the situation for high data consistency, the fsync system call can replace the sync to do the write out.

Since the incorrect data in a directory, in-core inode or superblock can make even more serious problem, in UNIX, the kernel writes through to the disk each time it writes a directory, inode, or superblock in the buffer cache.

7.9.2.3 Character Device Drive for Disk

We know the block devices are structured, which are called structured I/O more properly. Therefore, the rest part of devices in the I/O system belongs to unstructured I/O. In UNIX, the devices in the system are divided into two groups: block devices and character blocks. Thus, except the block devices introduced above, all the others in the system are incorporated into the character devices.

Since many of the read and write operations are done by the interaction between the user and the system through the user interface of UNIX, which is a terminal or terminal window, the disk manipulating implementation inescapably involves the character devices and the character device files. In fact, the whole disk manipulating implementation includes three parts: a character device drive, a block device drive, and a swap device drive. The block device drive has been discussed in the last section, and the swapping has been introduced in Section 5.2. Thus, here we discuss the character device drive for the disk. In BSD versions, sometimes the character device drive is called raw device interface (Quarterman et al 1985).

In UNIX, disk manipulation is implemented via a queue of operation notes, which are called transaction records. Each transaction record holds the following fields:

- A flag, which tells whether the record is for reading or writing.
- A memory address, where the transfer is in the primary memory.
- A disk address, where the transfer is on the disk.
- The byte number of the transfer, which indicates the number of the transfer in bytes.

The character device drive for disk makes the transaction records according to the user process requests. It actually does the mapping of the transferred data from the user process memory space to the buffer. According to the transaction records, the block device drive accomplishes to transfer the data between the buffer and the disk, and makes its best to sort the transaction records in order to enhance the disk I/O operations. According to a transaction record, the swap device drive can do swapping if necessary.

7.9.3 Character Special Files and Streams

As the terminal special files are the main part of the character special files, the terminal drivers will be paid more attention to in this section. Terminals are special in the feature that they are the user hardware interface to the computer system. Since character special files are usually used for devices with which human beings interact with the computer system directly and deal with character streams, they do not move chunks of data between memory and disk. Therefore, the buffer cache is not needed for character special files,

as shown in Figure 7.13.

In fact, basically, there are two solutions to character drivers.

7.9.3.1 Line Discipline Solution

This solution was originated from BSD (Bach 2006; McKusick et al 2005).

To do interactive utility, terminal drivers have an internal interface to line discipline modules that interpret input and output.

For the internal interface, there are two modes that can be chosen: one is a raw mode, the other is a canonical mode. In canonical mode, the line discipline converts the raw character sequence typed at the keyboard to a processed character sequence that a process can recognize before sending the data to a receiving process; the line discipline also converts raw output sequences written by a process to a format that the human being is familiar with. In raw mode, the line discipline passes data between processes and the terminal directly without any interpretation. The raw mode is especially useful when sending binary data to other computers over a serial line or for GUIs. The main functions of a line discipline are listed in Table 7.2.

Table 7.2 Main functions of the line discipline

Function	Description
To receive the input	In a canonical mode, to generate signals to processes for terminal hangup, or in response to a user pressing control characters, such as the Delete, CTRL-S, or Enter key In a raw mode, to forbid interpreting special characters such as erase (Backspace or Delete), kill (CTRL-U) or carriage return (Enter)
To process	To parse input strings from the keyboard into lines To process erase characters (such as Backspace and Delete) To process a kill character (such as CTRL-U) that invalidates all characters typed so far on the current line
To send the output	In a canonical mode, to echo received characters to the terminal screen with interpretation In a raw mode, to echo received characters to the terminal screen without interpretation

The data structures handled by line disciplines are called c-lists, which represent character lists. A c-list is a linked list of cblocks with a count of the number of characters on the list. A cblock contains three fields: a pointer to the next cblock on the linked list, a small data block, and two offsets indicating the position of the valid data in the cblock. The start offset indicates the first location of valid data in the data block, and the end offset is the first location of non-valid data.

Clists provide a simple buffer holding the small volume of data that is fit for transmission of slow devices such as terminals. They allow dealing with one character at a time or groups of cblocks.

For terminal drivers, there are three clists associated with them: one is used to store data for output to the terminal screen, the second one is to store the raw input data provided by the terminal interrupt handler as the

user typed it in, and a third one is to store the processed input data that the line discipline converts the data of the raw clist into.

Therefore, when a user process reads from /dev/tty, the characters pass through the line discipline. The line discipline accepts the raw character sequence from the terminal driver, "cooking" it, and producing the cooked character sequence. The cooked sequence is passed to the process. However, if the process wants to interact on every character, it can put the line in raw mode, in which case the raw character sequence will be passed to the process directly without being cooked.

Output works in a reverse way, such as, converting a line feed to a carriage return plus a line feed, adding filler characters following carriage returns for slow mechanical terminals, and so on. Like input, output can go through the line discipline in the raw mode or canonical mode.

7.9.3.2 Streams Solution

This solution was from System V, and devised by Dennis Ritchie (Bach 2006; Ritchie 1984). It was issued at the background of some drawbacks of device drivers. That is, different drivers tended to duplicate functionality, such as network drivers which include a device-dependent portion and a protocol portion, but it was difficult to realize in the reality because the kernel did not provide the mechanisms for common use. Therefore, to get more flexibility and greater modularity, it is a mechanism to break down the modularity of the UNIX I/O system. That is Ritchie's solution.

The basic idea of a stream is to break down a whole device drive into several modules connecting from a user process to a driver and be able to insert processing modules into the stream dynamically for a certain utility. So a stream functions like pipelines in user space.

A stream is a duplex, multi-joint connection between a process and a device driver. It consists of a set of linearly linked queue pairs. Each queue pair has two members: one is for input, the other for output. When a process reads data from a stream, the kernel transfers the input data from the device driver up to the process through the input queues. In reverse, when a process writes data to a stream, the kernel sends the data down the output queues. The queues also pass messages to neighboring queue by an appropriate interface. Each queue pair belongs to an instance of a specific kernel module, such as a driver, line discipline, or protocol. Modules manipulate data passed through their queues. The stream infrastructure in the UNIX kernel defines the interfaces of modules so well that different modules can be plugged together.

A typical queue is a data structure that contains some elements shown in Table 7.3.

Table 7.3 Main elements of a stream queue

Element	Description
procedures	An open procedure that is called during an open system call
	A close procedure that is called during a close system call
	A put procedure that is called to pass a message into the queue
	A service procedure that is called when a queue is scheduled to execute
pointers	A pointer that points to the next queue in the stream
	A pointer that points to a list of messages awaiting service
	A pointer that points to a private data structure that maintains the state of the queue
flags and marks	Flags and high- and low-water marks that are used to control the data flow, schedule and maintain the queue state

A device with a stream driver is usually a character device that has a special field in the cdevsw table that points to a stream initialization structure, containing the addresses of procedures and high- and low-water marks shown in Table 7.3. For the first open system call of a stream driver, the kernel allocates two pairs of queues, one is for the stream-head, and the other is for the drive. The stream-head module is identical for all instances of open streams. It has generic put and service procedures and is the interface to higher-lever kernel modules that implement the read, write, and other system calls. Many modules, if needed, can be inserted into a stream by issuing ioctl system calls (see Figure 7.14). As queue pairs are bidirectional, each module maintains a read queue and a write queue; one is for reading, the other for writing. When

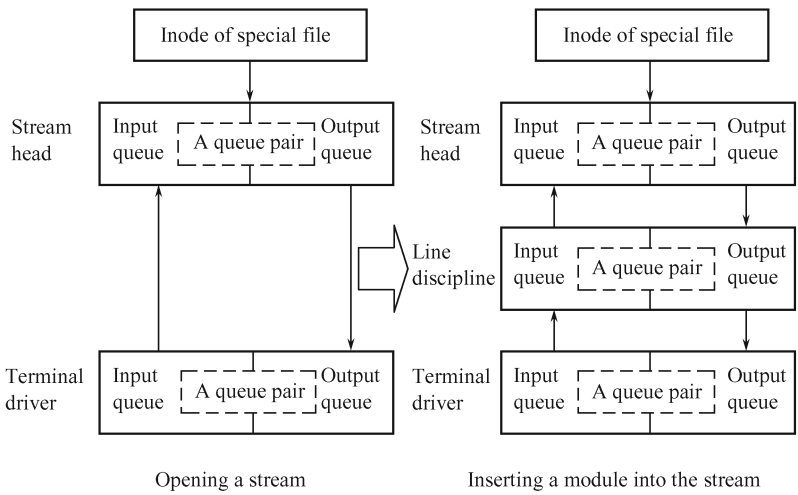

Fig. 7.15 A simple stream being created.

a user process reads from a stream, the stream head interprets the system call and packs the data into stream buffers that are passed from module to module upward. With each module performing their part in the transformation job correctly, the user process can read the data that it wants. A simple stream being created is displayed in Figure 7.15.

7.9.4 Sockets for Networks in UNIX

Networks are another type of I/O devices in UNIX. They are usually treated as sockets, which were started from BSD versions (Bach 2006). Sockets are used to handle network utilities, such as mailboxes, remote login, and remote file transfer. As shown in Section 4.5.3, in UNIX, there are many methods to maneuver inter-process communication even on different machines. Sockets can make processes communicate with other processes on other computers according to protocols and media.

For the socket mechanism, the kernel structure has three parts, which are shown in the order from the higher level down to the lower level in Table 7.4.

Table 7.4 Kernel structure for the socket mechanism

Layer	Function
Socket layer	To provide the interface between the system calls and the lower layers
Protocol layer	To contain the protocol modules used for communication (such as TCP and IP)
Device layer	To contain the device drivers that control the network devices (such as the network adapter card)

Appropriate combinations of protocols and drivers should be defined in the system configuration. Processes communicate using the client-server model (which will be discussed in Chapter 11), by which a server process listens to a socket at one end of a bidirectional communication path, and client processes communicate to the server process by another socket on another machine at the other end of the communication path. The kernel handles internal connections and routes data from client to server.

The socket mechanism involves several system calls. The socket system call establishes the end point of the communication link with the type of communication and the protocol to control the communication and returns a socket descriptor. With the socket descriptor, the bind system call associates an address that points to a structure holding the information of the communication domain and protocol. Before a socket can be used for networking, it must have an address bound to it. The address can be in one of several naming domains. The most common domain is the Internet naming domain. The connect system call sets up a connection for an existing socket. The listen

system call sets the queue length. The accept call allows incoming requests for a connection to a server process. The send and recv system calls send and receive data through a connected socket. The shutdown system call closes a socket connection. And so on.

Each socket supports a certain type of communication properties and follows a certain protocol. Here, two common types are listed:

- Virtual circuit type (or reliable connection-oriented byte stream): This socket type allows two processes on different machines to establish the equivalent of a pipe between them. Byte stream is sent in one end of the pipe and comes out in the same order at the other end. The system guarantees that all bytes sent out can reliably arrive at the receiver end.
- Datagram type (or unreliable packet-oriented transmission): This is especially useful for real-time utilities (such as, on-line movies). This type provides the unduplicated delivery, that is, it does not guarantee sequenced and reliable delivery. Packets sent out may be lost or reordered by the network. In some situation, higher performance is more important than reliability (for example, for multimedia delivery, fast transmission is more important than being correct).

The most popular protocol for virtual circuit type is Transmission Control Protocol (TCP). The common choice for the datagram type is User Datagram Protocol (UDP). Both protocols are basic for the Internet. We will discuss them in Chapter 11.

After sockets have been created on two computers, a connection can be established between them (for reliable connection). One side makes a listen system call on the server socket, which creates a buffer and blocks until data arrive. The other side makes a connect system call on the client socket, providing parameters such as the file descriptor for the client socket and the address of the server socket. If the server socket accepts the call, the system then establishes a connection between the two sockets. Once a connection established, it functions like a pipe. A process can read from and write to it using the file descriptor for its client socket. When the communication is no longer needed, it can be closed with the close system call.

7.10 Summary

Because of the mechanisms that UNIX uses to handle I/O devices, standard files, I/O redirections, and pipes and filters are closer to the file system in UNIX. Input redirection means to let a command input direct to a file rather than to standard input – a terminal keyboard. There are two different ways to do input redirection. One is to use the less-than symbol (<); the other is to use the < combined with the file descriptor 0 of the standard input file, stdin. Output redirection means to let a command output direct to a file rather than to standard output – a terminal screen. There are also two

different ways to do output redirection. One is to use the greater-than symbol (>); the other is to use the > combined with the file descriptor 1 of the standard output file, stdout. Expecting not to overwrite the previous contents of the destination files, use the appending redirection operator >>. As output redirection, standard error redirection means to let the error caused by the command execution direct to a file rather than to standard output – a terminal screen. The standard error is redirected only by using the > operator combined with the file descriptor 2 of the standard error file, stderr.

In the C shell, the input, output, and appending redirection operators (<, > and >>) do work as well as they do in other shells, but there is not an operator to redirect the stderr alone. The >& operator can be used for both the output and error redirections at the same time.

In most UNIX shells, the input, output, or error messages of a command can be redirected in one command line. The appending redirection >> operators can also be combined together with themselves or other redirection operators in one command line.

In UNIX, pipes and filters are usually used to perform some complicated tasks that cannot be done with a single command. The pipe operator (a vertical bar |) can be used to make the standard output of the command to the left of the pipe operator become the standard input of the command to the right of the pipe operator. If a command takes its input from another command, does some operation on that input, and writes the result to the standard output, the command is called a filter. Pipes and I/O redirections can be used in one command line.

In the C shell, the | operator can connect the output of one command to the input of another one. Otherwise, it also allows the output and error messages of one command to be attached to the input of another command with the |& operator.

Processes that are communicating with pipes usually have the same parent process while processes that are communicating with named pipes can be independently on one system. UNIX provides the mkfifo command to create named pipes. A named pipe can be removed from the file system by using the rm command, when it is no longer used.

In UNIX, the I/O devices are integrated into the file system, so the user interface to devices needs to go through the file system to control the I/O devices. That is, it is through a special file to access one of hardware devices, including block devices and character devices. To access a device, use the command or system call that accesses its special file. All I/O devices in UNIX are treated as files and are accessed as such with the same read and write system calls that are used to access all ordinary files. The driver interface that the kernel builds between the file system and the device drivers includes two tables: the block device switch table (bdevsw) and character device switch table (cdevsw). The former is for block special files; the latter is for character special files. A block special file consists of sequence of numbered blocks; the key property of the block special file is that every block can be accessed

with its own address. Character special files are usually used for devices with which human beings interact with the computer system directly.

To minimize the number of actual transfers between the CPU and I/O devices, UNIX uses a buffer cache between the disk drivers and the file system. The strategy system call can be used to transmit data between the buffer cache and a device. Usually, the buffers in the cache are linked together in a list, which adopts the least recently used mechanism to manage its buffers. When the cache fills up, some block in the buffer must be forced out. In UNIX, it is periodic to do write all the modified blocks back to the disk.

Actually, the whole disk manipulating implementation includes a character device drive, a block device drive, and a swap device drive.

There are two basic solutions to character drivers. One is the line discipline solution that was from BSD. For this solution, terminal drivers have an internal interface to line discipline modules that interpret input and output. The other is the streams solution that was from the System V. It is to break down a whole device drive into several modules connecting from a user process to a driver and be able to insert processing modules into the stream dynamically for a certain utility. A stream is a duplex, multi-joint connection between a process and a device driver. Modules have well-defined interfaces, defined by the streams infrastructure in the UNIX kernel, so different modules can be plugged together.

Networks are another type of I/O devices in UNIX. They are usually treated as sockets, which were started from BSD. Sockets are used to handle network utilities. Sockets can make processes communicate with other processes on other computers according to protocols and media.

Problems

Problem 7.1 What are standard files? State their purposes, respectively.

Problem 7.2 Describe input, output, and error redirection briefly. Write two commands of each to show single and combined use of the redirection operators.

Problem 7.3 How can the input, output, and error redirection operators be combined with the file descriptors of standard files to perform redirection in the Bourne shell? Give your examples.

Problem 7.4 Give a command sequence to create a short text file.

Problem 7.5 Give a command line to combine the pipe and output redirection operators.

Problem 7.6 What is the purpose of the tee? Give an example to use the tee.

Problem 7.7 What is the UNIX pipe? What is a filter? How different is pipe from output redirection? Give an example to illustrate.

Problem 7.8 What is a named pipe? Write down a single command to

create three named pipes, called myfifo1, myfifo2, and myfifo3. Write a command sequence by using these named pipes.

Problem 7.9 In UNIX, how can a process access one of hardware devices? How many kinds are hardware devices in UNIX divided in? What are they?

Problem 7.10 What is a block special file? What is a character special file?

Problem 7.11 Why is the delayed write mechanism adopted in UNIX?

Problem 7.12 What does the driver interface that the kernel builds between the file system and the device drivers include? How do they work?

Problem 7.13 In UNIX, what is a buffer cache? What is the buffer cache used for? How is the buffer cache managed?

Problem 7.14 What does the strategy system call do?

Problem 7.15 In UNIX, how many solutions are there to character drivers? What are they? Please explain how they work, respectively.

Problem 7.16 What are sockets used for? The socket mechanism involves several system calls. Explain how they function.

References

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Carson SD, Setia S (1992) Analysis of the periodic update write policy for disk cache. IEEE T Software Eng 18(1): 44 – 54

Heindel LE, Kasten VA (1995) Real-time UNIX application filestores. RTAS'95: First IEEE Real-time Technology and Applications Symposium, Chicago, Illinois, 15 – 17 May 1995, pp 44 – 45

Isaak J, Lohnson L (1998) POSIX/UNIX standards: Foundation for 21st century growth. IEEE Micro 18(4): 88, 87

Jespersen H (1995) POSIX retrospective. ACM, StandardView 3 (1): 2 – 10

McKusick MK, Neville-Neil GV (2005) The design and implementation of FreeBSD operating system. Addison-Wesley, Boston

Nelson BL, Keezer WS, Schuppe TF (1996) A hybrid simulation-queuing module for modeling UNIX I/O in performance analysis. WSC'96: The 1996 IEEE Winter Simulation Conference, 8 – 11 December 1996, pp 1238 – 1246

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading

Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17 (7): 365 – 375

Ritchie DM (1984) A stream input output system. AT&T Bell Lab Tech J, 63(8) Part 2: 1897 – 1910

Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: the textbook, 2nd edn. China Machine Press, Beijing

Stallings W (1998) Operating systems: internals and design principles, 3rd edn. Prentice Hall, Upper Saddle River, New Jersey

8 UNIX Shell Introduction

Known in Chapter 2, UNIX provides a text-based interface or a CUI interface (see Figure 2.1). This interface is a shell, which bridges between the UNIX kernel and users. In other words, when typing in a command line in a terminal system or a terminal window, users of UNIX work on one of the shells. When a user logs on and enters a terminal system, UNIX starts running a program that is a UNIX shell (Bach 2006; Miller et al 2000; Mohay et al 1997; Quarterman et al 1985; Ritchie et al 1974; Sarwar 2006). When the shell starts running, it gives a shell prompt ($ for the Bourne or Korn shell, or % for the C shell) and waits for the user to type in commands (Bourne 1978; Bourne 1983; Joy 1980; Korn 1983; Rosenblatt et al 2002). The UNIX shell executes commands that the user types on the keyboard.

The primary purpose of the shells is as the UNIX command interpreter to interpret commands that the user types in. UNIX shells can execute programs and commands, and control program and command input and output. They can also execute sequences of programs and commands. In this chapter, we will discuss a variety of UNIX shells, UNIX shell as the command interpreter, environment variables, switching between UNIX shells, and shell metacharacters.

UNIX shells are also high-level programming languages. For the kernel, the shells are just ordinary programs that can be called by a UNIX kernel or commands. They contain some characters (such as variables, control structures, and so on) that make it similar to a programming language. Users can use one of shells to write their own short programs, called shell scripts, to accomplish particular functions, and run them on that particular shell. We will learn how to program with Bourne shell in the next two chapters.

8.1 Variety of UNIX Shells

There are many kinds of UNIX shells, just as the situation that there are many versions of the UNIX operating system. Among them, Bourne, Korn, and C shells are the most popular ones. And there are also some else, such

as the Bash, TC, and Z shells (Sarwar et al 2006). And users can add their own utilities to their shells and even create totally a different shell if they like and have enough competence.

8.1.1 Shell Evolution

Usually, in a UNIX operating system, there are several different shells existing, which are likely the Bourne, C, and Korn shells. As programs have been developed more and more with time to meet the needs of users, shell programs are evolving and growing.

The Bourne shell was developed for UNIX System versions by Stephen Bourne at AT&T Bell Labs (Bourne 1978), and it is the development fundamental of other shells else and the most basic shell. The C shell was developed for BSD by William N. Joy at the University of California, Berkeley (Joy 1980). The C shell is different from the Bourne shell and mostly, simulates the C language in some way. The Korn shell was developed by David Korn at AT&T Bell Labs (Korn 1983), which is compatible with the Bourne shell and have more functions than the Bourne shell.

In Linux, there is the Bash shell, which means Bourn Again Shell. The Bash shell evolves from the Bourne shell, but includes some features of the C and Korn shells. Most of the Bourne scripts can run in the Bash shell without any modification. There are also TC and Z shells for Linux systems (Sarwar et al 2006).

The simplified shell evolution is shown in Figure 8.1, which also displays the tendency that the functions are increasing – the shell at the right of the figure has more functions than the shell at the left usually. The Bourne shell is the ancestor for all the other shells, and the Korn shell is a relatively younger and has more functions. The Figure 8.1 is just illustrative and helps readers know the shell evolution process, so they may not follow the conventions rigorously. A solid line roughly represents the closer relationship between the right and left items than a dot line.

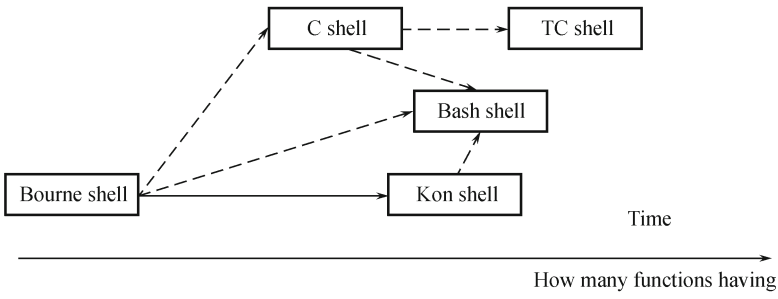

Fig. 8.1 The simplified shell evolution.

8.1.2 Login Shell

When logging on, only one shell launches. This shell is called the login shell, which is determined by the system administrator. Recall that in Section 6.1.1, the passwd file in /etc directory holds several lines of user information, each line for each user on the system. And the following is the format of the line:
Login_name:password:user_ID:group_ID:user_info:home_directory:login_shell

The last field is a login_shell, which holds the absolute pathname for the user's login shell. When the user logs on, the system executes the command corresponding to the pathname in this field. The following is the example that was also given in Section 6.1.1.

```
wang:x:120:101:wang wei:/users/project/wang:/usr/bin/sh
```

where, the login shell is /usr/bin/sh, which is the Bourne shell.

To check out which shell is the login shell on the system, we can also use the echo $SHELL command just after logging in the system (see Section 6.3.1).

Usually, a shell is usually located in a corresponding directory in the file system. Table 8.1 shows a list of the most common shells, their pathnames on the file system, and the shell program names. Note: The pathnames shown here are typical for most systems, but they may be slightly different for different UNIX operating systems.

Table 8.1 Shell pathnames and program names

Shell	Pathname	Program Name
Bourne	/usr/bin/sh	sh
C	/usr/bin/csh	csh
Korn	/usr/bin/ksh	ksh
Bash	/bin/bash	bash
TC	/usr/bin/tcsh	tcsh
Z	/usr/local/bin/zsh	zsh

8.2 UNIX Shell as a Command Interpreter

When a user logs in UNIX, a login shell acts as an interface between the user and the UNIX kernel. In Chapter 2, it has been known that after logging on UNIX, some of the UNIX operating system directly provide a text-based interface or a CUI interface and some others bring the user into a GUI interface, but having a terminal window that functions like the text-based interface. In the latter situation, the user interacting environment is also in the login shell, which bridges between the UNIX kernel and the user. Therefore, when the user types in a command on a command line in a terminal system or terminal window, the user works on the login shell. When a shell starts running, it

gives a shell prompt ($ for the Bourne or Korn shell, or % for the C shell) and waits for the user to type in commands. The UNIX shell executes the commands that the user types on the keyboard. From the knowledge in the previous several chapters, it is known that the user types a command and presses Enter after a shell prompt, and the shell interprets the command and executes it. That is why the shell is called the UNIX command interpreter.

8.2.1 Shell Internal and External Commands

Shell commands can be divided into two groups: the internal (built-in) commands and the external commands. As being also called built-in commands, the internal command code is part of the shell process. It means that they are part of a shell program, and the instruction is run without going out of the shell code. Usually, the commands that are the most useful and relatively shorter are embedded in the shell code.

The external commands are usually programs that are stored as binary executable files. When they are executed, the kernel needs the fork and execve system calls to create a child-process and do the execution (see Section 4.4).

When the shell is being executed, it can be terminated by the user's pressing CTRL-D on a new command line. When the shell receives this keystroke combination CTRL-D, it terminates itself and makes the user log off the UNIX. The system then prompts the login: again, and the user can use it to log in the system again.

The above is a typical whole process that a shell works.

8.2.2 Shell's Interpreting Function

For UNIX users, the shell functions like a user interface. All it needs is the ability to read from and write to the terminal, and to execute other programs.

In the kernel, the execution mechanisms for internal commands and external commands are different. For the internal commands, it is simple — just to let the instruction jump to the beginning of that part of code. For the external commands, it needs the fork and execve system calls. So the shell's interpreting function typically indicates the external command execution.

When the external command is executed, the shell interprets first the command. Typically, it treats the command line that the user just types in the following order:

- The first field, which contains the name of the executed command;
- The second field, which holds the options that are starting with a hyphen (-);
- The third field, which accommodates the command arguments.

This order is just the same as a shell program needs and following the UNIX command syntax shown in Section 2.4.1.

After reading the command line, the shell process determines whether the command is an internal or external command. It performs all internal commands by jumping to the corresponding code segments that are within its own code.

To execute an external command, the shell searches several directories in the file system structure, looking for a file that has the command name. The kernel then transfers control to an instruction located at the beginning of the file code and executes the code.

8.2.3 Searching Files Corresponding to External Commands

The pathnames of the directories that a shell uses to search the program file of an external command are called the search paths. The search paths are stored in the shell variable PATH (or path in the C shell, the former is capital-lettered, the latter is small-lettered).

To check out the PATH variable, use the echo command, which has been discussed in Section 6.3.1. The following displays the echo command again, but here with shell variables as its arguments.

```
$ echo $PATH
% echo $path
```

Function: to display shell variables on the display screen; the first one is used in Bourne, Korn, and Bash shells; the second one is used in C shell.

Note: echo can be used to send strings, file references, or shell variables to the standard output. As the search paths are discussed here, shell variables are focused on.

For example:

```
$ echo $PATH
/usr/bin:.:/users/project/wang/bin:/users/project/wang:/usr/include:
/usr/lib:/etc:/usr/etc:/usr/local/bin:/usr/local/lib:/bin
$
% echo $path
/usr/bin . /users/project/wang/bin /users/project/wang /usr/include
/usr/lib/etc /usr/etc /usr/local/bin /usr/local/lib /bin
%
```

The results of the two commands are different. The first command is for the Bourne, Korn, and Bash shells, and it separates pathnames by colons; the second command is in the C shell and separates pathnames by spaces.

8.3 Environment Variables

Some environment variables have been used in the previous chapters and sections, such as PATH, SHELL, and HOME. In this section, a list of environment variables is given and how to set them will be discussed, too.

8.3.1 Some Important Environment Variables

Different shells have their own environment variables to determine how the shells to act, how to execute commands, how to handle I/O, how to program, and the user environment. Table 8.2 lists some important environment variables for the popular shells.

Table 8.2 Some important environment variables of the Bourn, Korn and C shells

Bourne, Korn Shells	C Shell	Use
CDPATH	cdpath	Its value is used by the cd command as the pathnames to search for its argument directory; if it is not set, the cd command searches the working directory
ENV		Its value is used by UNIX as the pathname of the configuration files
EDITOR		Its value is used by some programs, such as the e-mail program and pine, as the default editor
HOME	home	Its value is the name of the user's home directory
MAIL	mail	Its value is the name of the system mailbox file
MAILCHECK		Its value is a period at which the shell should check user's mailbox for new mail and inform user
PATH	path	Its value is the search path that a shell uses to search for an external command or program
PPID		Its value is the process ID of the parent process
PS1	prompt	Its value is the shell prompt that appears on the command line. For Bourne or Korn shell, it is $; for C shell, it is %
PS2	prompt2	Its value is the secondary shell prompt displayed on second line of a command if the command is not finished. The default value is "> "(a greater-than symbol and a space). The user can continue to type in characters after the secondary shell prompt to finish the command
PWD	cwd	Its value is the name of the current directory
SHELL		Its value is the pathname for the current shell
TERM		Its value is the type of the user's console terminal

The environment variables in Table 8.2 are writable. There are also some

other environment variables, such as the positional arguments, which are read-only. For the Bourne shell, Chapter 9 will give a detailed discussion.

In Section 2.6, the shell setup files have been discussed. They are some-hidden files in the home directory, such as .cshrc (for the C shell), .profile (for the Bourne or Korn shell), .login (for the C shell), .bashrc, .bash_profile, or .bash_login. They are shell setup files or configuration files. Shell setup files contain commands that are automatically executed when a new shell starts – especially when a user logs in. They also contain the initial settings of important environment variables for the shell and something else, for instance, .profile in System V and .login in BSD. And some hidden files for specific shells are executed when to start a particular shell, for example, .cshrc for C shell and .bashrc for Bash. Usually, it is the responsibility of the system administrator to modify the hidden files. But for readers, it has been introduced a little in Chapter 3 and will be discussed more in the following chapters how to change them.

In UNIX, there are also some other hidden files that are for other setup and configuration purposes rather than for shells. For all the hidden files, the same is that their names start with a dot (.).

8.3.2 How to Change Environment Variables

The environment variables can be set temporarily on the command line by using the set command and set permanently in some hidden files. In Chapter 9, detailed discussion on the set command will be given. Here, take the PATH variable as an example to practice setting the value of the environment variable.

In Table 8.2, the environment variable PATH (for Bourne or Korn shell) or path (for C shell) holds the search path that a shell uses to search for an external command or program and is set in the .profile (on System V) or .login (on BSD) file. To change the search path, set the new value of this variable.

To change the search path temporarily for the current session, just change the value of PATH at the command line. For example:

```
$ PATH=~/bin:$PATH:.
$
```

In this command, $PATH means remaining the default search path and ~/bin:$PATH:. means adding two directories (~/bin and .) into the current search path. Be careful! Don't lose the default search path settings.

For a permanent change, change the value of the PATH variable in the corresponding hidden file.

8.3.3 Displaying the Current Values of Environment Variables

To display the current values of environment variables, use the env command.

The syntax and function of env command are as follows.

```
$ env
```

Function: to display the current values of environment variables.

Common options: none.

Note: It depends on if or not the env command is available in the system. If not, the echo command can do the same job, even though the echo command shows one environment variable at a time.

For example:

```
$ env
EDITOR=/usr/ucb/vi
HOME=/users/project/wang
LOGNAME=wang
PATH=/usr/bin:.:/users/project/wang/bin:/users/project/wang:/bin
SHELL=/usr/bin/sh
......
$
```

The result of this command displays the environment variables and their values.

8.4 Switching Between UNIX Shells

Usually, in a UNIX operating system, there is more than one shell that is available. And most of the shells have some common facilities. As mentioned above, all a shell needs is the ability to read from and write to the terminal, and to execute other programs. They must have the ability to handle the I/O in order to interact with the user and the ability to execute the internal and external commands or programs. They also must be used to program in order to execute a sequence of programs and commands.

8.4.1 Why to Change Shell

Even though shells usually have some common functions mentioned above, some programs or external commands can be programmed in different shells. Hence, some of them can be executed in a specific shell while some others can be run in a different shell because not every shell is compatible to other shells well.

In fact, each shell has its own strengthens and shortcomings. For UNIX users, especially programmers, it is wise to learn the detail of some popular

shells. If they learn more, they will realize the carefully learning can help them not only to know how to use the shells but also to understand how the shells to function, and even better, to design their own version of shell in the future. In addition, as there are so many shells, and programs or scripts that users get someday in the future may run in different shells, knowing the shells well can help the users execute the programs smoothly and well. By the way, it is common for UNIX users to interact with more than one shell during a session, especially among shell programmers.

8.4.2 How to Change Shell

There are two different methods to change the shell.

One way to change the shell is to execute some corresponding commands on the command line. But before changing the shell, check out which shell is the current one, use the echo command like this:

```
$ echo $SHELL
/usr/bin/sh
$
```

This command shows the pathname of the current shell.

To change from Bourne shell to C shell, use the following command:

```
$ csh
%
```

The prompt has changed from $ to %.

To change from C shell to Bourne shell, use the following command:

```
% sh
$
```

The prompt has turned from % to $.

As learned from the third example in Section 4.4.2, a parent process can fork and execute a child process for another shell that can be different from the parent process. Hence, the above method is just to create a child process that is running the new shell and the login shell is not changed, which is the parent process of the new shell.

To terminate this temporary shell and return to the original login shell, press CTRL-D on a new command line, like

```
% CTRL-D
$
```

On some systems, this method may not function well. If not, do the following:

```
% exit
$
```

The above two commands are used for the situation that the original login

shell is Bourne shell. If the original login shell is C shell, the difference is the prompt changed from $ to %, not from % to $.

For other shells, the program names and pathnames can be seen in Table 8.1.

The other way to make the current shell change is to use the chsh command.

For some UNIX operating systems, the chsh command can do shell change. But it is not guaranteed for all UNIX operating systems. If it is available, the shell then prompts to type in the pathname of the new shell, which can be chosen from Table 8.1. For instance, the pathname of the C shell is /usr/bin/csh.

If the above methods cannot work well, the reasons can be one of the following: the C shell is not available on the system, it is not accessible, or the search path does not include /usr/bin. The final problem can be solved by using /usr/bin/csh instead of csh, or puting /usr/bin in the search path.

8.4.3 Searching for a Shell Program

For users, sometimes, it is necessary to know firstly whether a shell exists in the system and where it is located in the file system in order to change the current shell. To search for a program in a file system, use the whereis or which command. Here, try to use these commands to check out if or not a certain shell program is available on the system.

The syntax and function of whereis command are as follows.

```
$ whereis [option(s)] filename(s)
```

Function: to find and display on the screen the location of the binary, source, and man page files for a command.

Common options:

-b: to search only for binaries;

-m: to search only for manual sections;

-i: to search only for sources.

For example:

```
$ whereis csh
/usr/bin/csh
$
```

The result of this command is the absolute pathname of csh.

The which command can do the same job as the whereis command does.

The syntax and function of the which command are as follows.

```
$ which filename(s)
```

Function: to find and display on the screen the pathname or alias of a command.

Common options: none.
For example:

```
$ which csh
/usr/bin/csh
$
```

The result of this command is also the absolute pathname of csh.

8.5 Shell Metacharacters

In Section 6.4, three kinds of wildcards that are asterisk (*), question mark (?), and square brackets ([]) have been discussed. They can be used to save typing for a long filename or to choose many files at once. These wildcards belong to the shell metacharacters.

Shell metacharacters are some characters rather than letters and digits, and have special meaning to the shell. For their special uses, they cannot be used in filenames. As the wildcards, the shell metacharacters can also be used to specify many files in many directories in one command line. It will save users a lot of time in the future to know well how to use them in commands.

No space is required before or after a metacharacter when these characters are used in commands. However, it will be clear to use spaces before and after a shell metacharacter. Table 8.3 lists some shell metacharacters and their functions. Table 8.4 displays some shell metacharacters only used in C and Korn shells. In Table 8.3, most of the examples in the third column are the practical examples that have been used in the previous chapters. And some other examples can be used in the following chapters.

Table 8.3 Some shell metacharacters

Metacharacter	Function	Practical example
Enter	To end a command line and start a new line	
Space/ Tab	To separate elements on a command line	$ ls $HOME
#	To start a comment in a shell script	# This is a C shell script
$	To end line To substitute a shell variable	$ ls $HOME
&	To put a running command in background	$ sort longfile > longfile.sorted &
;	To separate commands in sequentially execution	$ (echo “Welcome!”; pwd); date
“ ”	To quote multiple characters To allow substituting a shell variable	$ grep -v “David” freshman $ ls “$dirct1”
‘ ’	To quote multiple characters	‘2008-8-21’
`	To substitute a command	cmd1=`ls -al`

Continued

Metacharacter	Function	Practical example
^	To begin a line To negate the following characters	$ ls ~/work/file[^6]
( )	To execute a command list in a subshell	$ (echo "Welcome!"; pwd); date
{ }	To execute a command list in the current shell	$ {echo "Welcome!"; pwd}
[ ]	To surround a choice or a range of digits or letters	$ ls -i [a-zA-Z]?[1-7].html
*	To stand for any number of characters in a filename	$ more ~/[^0-9]*.[c,C]
?	To stand for a single character	$ ls -i [a-zA-Z]?[1-7].html
<	Input redirection operator	$ cat < gfreshman
>	Output redirection operator	$ date > td-time
|	To connect the output of one command to the input of another command	$ ls -la | grep "wang"
/	The root directory The separator symbol in a pathname	/usr/bin
\	To escape a single character To escape Enter character to allow continuation of a shell command on the next line	$ find. \(-name file1 -o -name '*.c' \) -print > myfilepn &

Table 8.4 Some shell metacharacters only for C and Korn shells

Metacharacter	Function	Practical example
%	The C shell prompt The starting character for specifying a job number	% or % 3
~	The home directory	~ /.profile

Except wildcards have been discussed in Chapter 6 and redirection operators and pipes have been introduced in Chapter 7, some other metacharacters are used in Chapter 4. Some new metacharacters are introduced here.

In the practical example column of Table 8.3, there is cmd1=`ls -al`. Here, the ls -al command is enclosed in the backquotes (` `). The backquote is the second-top most-left key on the keyboard.

Also in Table 8.3, the function of backslash (\) is to escape the single character after it. Usually, the following character is one of the metacharacters. As a metacharacter has its special meaning for the shell, to avoid confusing the shell and to use the character, put a back slash in front of the character to let the shell know that the character does not have the special meaning here. For example:

```
$ cat test\&1
```

```
......  The text of the test&1 is displayed here.
$
```

8.6 Summary

There are many kinds of UNIX shells. The most popular shells can be Bourne, Korn, and C shells. And there are also some else, such as Bash, TC, and Z shells.

When logging on the system, only one shell starts execution, which is called the login shell. When it starts running, a shell gives a shell prompt ($ for Bourne or Korn shell, or % for C shell) and waits for the user to type in commands. The UNIX shell interprets and executes the command.

Shell commands can be divided into two groups: the internal (built-in) commands and the external commands. The internal command code is part of the shell process. The external commands are usually programs that are stored as binary executable files. When the external commands are executed, the kernel needs the fork and execve system calls to create a child-process do the execution.

The pathnames of the directories that a shell uses to search the program file of an external command are called the search paths. The search paths are stored in the shell variable PATH (or path). Along with PATH, SHELL, and HOME, there are many environment variables. Different shells have their own environment variables to be used to customize for a user the environment, including how the shells to act, how to execute commands, how to handle I/O, and how to program. Shell setup files and some other hidden files, for instance, .profile in System V and. login in BSD, contain the initial settings of important environment variables for the shell and something else. The environment variables can be set temporarily on the command line by using the set command and set permanently in some hidden files. Using the env command can display the current values of environment variables.

Usually, in a UNIX operating system, there is more than one shell that is available. We can change the current shell with some methods: one is done by running some corresponding command on the command line; the other is by using the chsh command.

Shell metacharacters are some characters rather than letters and digits and have special meaning to the shell. As the wildcards, the shell metacharacters can also be used to specify many files in many directories in one command line.

Problems

Problem 8.1 What is a UNIX shell? Describe its main purposes and give some popular shell examples. What is a shell script for?

Problem 8.2 What is the login shell? How can you know which shell is your login shell? Give two ways of doing this and write down the login shell on your system?

Problem 8.3 How can you terminate the execution of a subshell? How about to terminate the execution of the login shell?

Problem 8.4 Why is a shell called as the UNIX command interpreter?

Problem 8.5 Give some examples of shell commands. How does the shell execute them? How can the shell execute an external command? Describe the process.

Problem 8.6 What is the search path for a shell? How can you find the search path? Check it out on your system and write down the displayed result.

Problem 8.7 What are environment variables of a shell for? What files can be used to set important environment variables for the shell initially and automatically? How can you display the current values of environment variables? Write down some of the environment variables' values on your system.

Problem 8.8 If we want to change the search path temporarily for the current session and remain the default search path, please recommend your solution to this problem.

Problem 8.9 At the shell prompt, type the set | more command, see what happens on the screen, and write down the displayed result.

Problem 8.10 At the shell prompt, type the csh command. Next, type the setenv | more command, see what happens on the screen, and write down the displayed result. Finally, press CTRL-D on a new line.

Problem 8.11 How can you change the current shell? Give your solutions for the temporary and permanent changes, respectively.

Problem 8.12 Using the whereis command, test the pathnames of the various shells listed in Table 8.1. Are all these shells available on your system? If they are, write down their pathnames.

Problem 8.13 What are the shell metacharacters? Give three examples of the shell metacharacters.

References

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Bourne SR (1978) The UNIX shell. T Bell Syst Tech J, 57(6) Part 2: pp 1971 – 1990

Bourne SR (1983) The UNIX system. Addison-Wesley, Reading, Massachusetts

Joy WN (1980) An introduction to the C shell. UNIX Programmer's Manual, 4.2 Berkeley Software Distribution. Computer systems research group, Depat. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Aug 1980

Korn D (1983) KSH - a shell programming language. USENIX Conference Proceedings, Toronto, Ontario, June 1983, pp 191 – 202

Miller RC, Myers BA (2000) Integrating a command shell into a web browser. 2000 USENIX Annual Technical Conference, San Diego, California, 18 – 23 June 2000. http://www.usenix.org/events/usenix2000/general/generaltechnical. html. Accessed 24 Sep 2009

Mohay G, Zellers J (1997) Kernel and shell based applications integrity assurance. ACSAC'97: The IEEE 13th Annual Computer Security Applications Conference, 1997, pp 34 – 43

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading, Massachusetts

Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17 (7): 365 – 375

Rosenblatt B, Robbins A (2002) Learning the Korn shell, 2nd edn. O'Reilly & Associates, Sebastopol

Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: the textbook, 2nd edn. China Machine Press, Beijing

9 How to Program in Bourne Shell (1)

Considering there are so many shells that two chapters cannot accommodate them totally (maybe a whole book can hold them), as a language, we will just discuss Bourne shell in detail and readers can find some references about other shells at the end of the chapter.

In this chapter and next one, we will discuss how to program in Bourne shell. This chapter will cover Bourne shell scripts, shell variables, Bourne shell variable commands, shell scripts' argument transport, how to execute a Bourne shell script, and two of program control flow statements: if and for. The next chapter will continue the discussion about Bourne shell as a high-level language.

9.1 Bourne Shell Scripts

As known, Bourne shell is not only a command interpreter but also a programming language (Bach 2006; Bourne 1978; Bourne 1983; Miller et al 2000; Mohay et al 1997; Quarterman et al 1985; Ritchie et al 1974; Sarwar et al 2006). It can be used by UNIX users to write shell scripts in order to perform various tasks that cannot be done by any single UNIX command.

A shell script is a program that can be stored in disk as a file, which can contain any of UNIX commands that can be typed in on a command line and be executed like an external command (Bourne 1978; Joy 1980; Korn 1983; Rosenblatt et al 2002). Like any language program, shell scripts have a basic structure. Before learning how to program in Bourne shell, it is helpful to get a whole picture that a shell script looks like. In this way, the readers know at least what their scripts have in order to make the shells execute them properly.

9.1.1 Simplified Structure of Bourne Shell Scripts

Typically, the basic structure and execution of Bourne shell scripts looks like what Figure 9.1 shows.

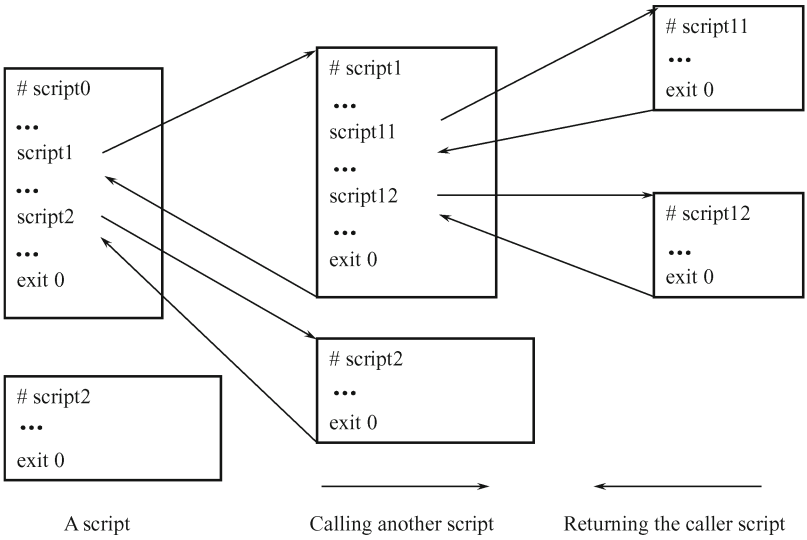

Fig. 9.1 The simplified structure and execution of Bourne shell scripts.

Bourne shell allows a script to call other scripts. Even though the Bourne shell does not have an explicit call statement, it does give users the feeling that the caller script does make a call for a called script when the called script name appears in the caller script code and the shell is running at that point of the caller script code. Or, we can say it works like that the caller script makes a call to the called script because exactly speaking, the shell does make the fork and execve system calls to allow the called script to execute. And when the task of the called script is performed, it can go back to the calling point in the caller script by using the exit command and the caller script execution resumes. Also, Bourne shell allows the called script programmed in its code to call another script. That means multi-level calling, or nesting the calls. There are different levels of the callers and the called scripts in the Bourne shell scripts. In fact, the shell in which commands and scripts are running can be seen as the top caller.

Users can use one of the text editors that have been introduced in Chapter 3 to create a Bourne script. Here is a simple Bourne script.

```
$ cat dptest0
fname0=file1
echo $fname0
exit 0
```

```
$ dptest0
file1
$
```

In the following sections, it will be given about what the content of the dptest0 script means.

9.1.2 Program Headers and Comments

With the background of software engineering, programmers know that the comments and program headers have no use for program execution, but they are helpful to resist forgetting and share programs with others. Thus, it is a good programming habit to put comments in every program in order to describe the purpose of a particular section of the code or to describe the purpose of a variable or assignment statement.

A program header is usually a group of comments at the top of a shell script, which is used to describe the main purpose of the whole script (Sarwar et al 2006).

Program headers and comments can also help programmers in a project group cooperate and check their programs each other easily. They can make programmers understand the programs quickly.

A comment can follow a command separated by # sign, like:

```
$ cat dptest0
fname0=file1 # to assign file1 to the fname0 variable
echo $fname0
exit 0
$
```

A comment can also occupy a whole line itself, such as:

```
$ cat dptest0
# This script is to show how to use the fname0 variable.
fname0=file1
echo $fname0
exit 0
$
```

In the second example, the comment line is just like a simple program header. However, a program header contains more information, such as the name, created time of the script, the author of the script, the last-modified time, the purpose of the script, etc. For example, the following is a program header for a data_backup script.

```
# Script name:  /users/faculty/data_backup
# Purpose: to make two copies for the database, databack and datasort,
#          one is not sorted; the other is sorted.
# Author:  Liu Yukun
# Date Created:  Sept 22, 2006
# Date Last Modified:  Aug 24, 2008
```

9.1.3 Exit Command

For users, the point that the exit command appears in a script code can be seen as the return point to the calling script or the shell that the script is running. In the kernel, the exit command is used to transfer control to the calling process.

The syntax and function of the exit command in Bourne shell are as follows.

```
$ exit [n]
```

Function: to exit from the script and return to its caller with an exit status number n.

Note: The exit command has only one argument that is optional and an integer number. The argument is returned to the calling process as the exit status of the called process. When the argument is 0, it means that the termination of the called process is success. If the argument is a nonzero number, it means that the termination of the called process is failure. Programmers can use different nonzero numbers to indicate different kinds of failure terminations of a script execution. When the exit command is without an argument, the UNIX kernel sets the exit status value for the script according to the execution of the script. All UNIX commands return the exit status in this way.

The exit status value of a command is stored in the $? environment variable that can be checked by the calling process. In shell scripts, the exit status of a script is commonly checked by the calling script in order to determine which action is taken subsequently. The use of $? will be shown in some shell scripts later in this chapter.

9.2 Shell Variables

Shell variables can be divided into two types: shell environment variables and user-defined variables.

In Chapter 8, some of the environment variables have been introduced. They are used to make the environment settings for proper execution of shell commands. Most of these variables are initialized in some of the system setup files. When a user logs in, the system setup files (such as the .profile file) are executed and accomplish the initialization of those environment variables. Usually, the system administrator writes the system setup files to set up a common environment for all users of the system. The users, however, can customize their own shell environment by assigning different values to some or all of these variables in the setup file according to their demands.

When a command is executed as a child process of the login shell, a copy of environment variables is passed to it by the kernel.

Except the writable environment variables introduced in Chapter 8, there are some more environment variables that are usually read-only. These variables can be read, but cannot be written. For example, positional arguments ($1-$9) and the environment variables that are used to hold the PID of the current process ($$), the exit status of the most recent command ($?), etc.

User-defined variables are created by users according to their demands when they program shell scripts. They are temporary and their values can be changed. They also can be defined as read-only by users.

A variable, no matter which is an environment variable or a user-defined variable, is a main memory location with a name. It allows the user to refer the memory location by using its name.

For environment variables, they have their definite and unchangeable names. But, UNIX users can define their own users-defined variables in their shell scripts. The name of a shell variable can be composed of letters, digit, and underscore. The first character of a variable must be a letter or underscore. As the main memory is the random access memory (RAM), the user can access a variable's value.

For a Bourne shell, the value of a variable is always treated as a string of characters (Bourne 1978; Sarwar et al 2006), even if only digits are in it. The length of a variable's value is unrestricted, theoretically.

In a Bourne shell, the shell variables do not need to declare and initialize. If not initialized explicitly by the user, a shell variable is initialized by the kernel to a null string by default.

To display environment variables, the env command can be used (see Section 8.3.3). Also, the set command can do the same job. And even better, the set command without any argument can display all the shell variables, including the user-defined variables. Except this function, the set command has also other functions, such as to set new values for some environment variables.

For example, compare the set command with the env command.

```
$ env
EDITOR=/usr/ucb/vi
HOME=/users/project/wang
LOGNAME=wang
PATH=/usr/bin:.:/users/project/wang/bin:/users/project/wang:/bin
PWD=/users/project/wang/work/text
SHELL=/usr/bin/sh
.....
$
```

The result of this command displays some environment variables and their values.

```
$ set
.....
EDITOR=/usr/ucb/vi
ERRNO=9
FCEDIT=/usr/bin/ed
HOME=/users/project/wang
IFS=‘’
```

```
LOGNAME=wang
PATH=/usr/bin:.:/users/project/wang/bin:/users/project/wang:/bin
PPID=12764
PS1=‘ ’
PS2=‘> ’
PWD=/users/project/wang/work/text
SHELL=/usr/bin/sh
TMOUT=0
......
$
```

The result of this command displays all the shell variables and their values.

9.3 Bourne Shell Variable Commands

There are several commands that can be used to access shell variables. In this and the following sections, these commands will be discussed in several groups according to their function relativity. And some practical examples will be following closely.

9.3.1 Reading Shell Variables

In the previous chapters, the echo command has been discussed several times. Here, its function related to shell variables and script programming will be focused on. Here is a more detailed description of the echo command.

The syntax and function of echo command are as follows.

```
$ echo [string]
```

Function: to display ‘string’ on the display screen; ‘string’ can be strings, file references, and shell variables.

Common options:

-n: to print line without newline;

-e: to explain the special characters after the backward slash.

Special characters in command arguments:

\b: to backspace;

\c: to print line without newline;

\f: to form feed;

\n: to put the cursor on a new line;

\r: to make the carriage return;

\t : to make the tab character;

\v: to make the vertical tab;

\\: to escape the special meaning of backslash;

\N: to display the character whose ASCII number is octal N, the N number must start with 0 (zero).

Note: In some UNIX operating systems, the -e option may be default. But in some others, the special characters can function well only if the echo command is followed with the -e option. The above special characters can be recognized within any of the arguments. Without an argument, echo displays a blank line on the screen. The current value of a variable can be referred to by using the echo command with inserting a dollar sign ($) before the name of each shell variable, which will be shown in the next section.

9.3.2 Assignment Statement

To assign the value to a variable, use the assignment statement.

The syntax and function of assignment statement in the Bourne shell are as follows.

```
$ va1=v1 [va2=v2 va3=v3 ]
```

Function: to assign the value of v1 to the variable of va1, the value of v2 to the variable of va2, and the value of v3 to the variable of va3.

Note: The statement vai=vi can go on and on for a number of statements in one command line if needed. There is no space before and after the equals sign (=) in this syntax. If a value contains space, it must be enclosed the value in quotes. Single quotes (‘ ’), double quotes (“ ”) and back-quotes (` `) work differently (see Table 8.3). The following examples will show the difference. These examples can be seen as a sequence.

```
$ uname1=Wang
$ echo $uname1
Wang
$
```

The Wang value is assigned to the uname1 variable. The echo command shows this value.

```
$ uname2=Wang Fang
Fang:  not found
$
```

This command causes an error message because the Wang Fang value has a space in between and is not enclosed in quotes. The shell takes the second word Fang in the value as a valid command and tries to execute it, but it is not a valid command at all.

```
$ uname2="Wang Fang"
$ echo $uname2
Wang Fang
$
```

The Wang Fang value is assigned to the uname2 variable. The echo command shows this value.

```
$ fname1=file?
```

```
$ echo ‘$fname1’
$fname1
$
```

This echo command displays the $fname1 as a common string because the single quotes just hold the whole string without processing it.

```
$ echo "The searched file is $fname1."
The searched file is file?.
$
```

Different from the last echo command, this echo command not only displays the sentence, but also substitutes the fname1 variable with its value (file?). The substitution is the double quotes' function.

```
$ echo $fname1
file1 file2 file3 file4
$
```

As the fname1 variable is not quoted, the echo command displays all the matched files in the working directory. The ? sign in the file? value is taken as the shell metacharacter.

```
$ echo \$fname1
$fname1
$
```

The escape metacharacter (\ ) makes the $ not to have its special meaning here and the echo command just takes $fname1 as a common string and displays it directly.

```
$ cmd1=date
$ $cmd1
Tues Aug 19 08:04:56 CST 2008
$ cmd2=ddd
$ $cmd2
Sh:  ddd:  command not found
$
```

The date value is assigned to the cmd1 variable. The shell treats the command $cmd1 or $comd2 in this way: it firstly substitutes the cmd1 (or cmd2) variable with the date (or ddd) value and then tries to execute it. If the value is a valid command, the result of the execution will display on the screen. If not, an error message will be shown.

```
$ cmd3=`date`
$ echo "Today is:  $cmd3."
Today is:  Tues Aug 19 08:04:56 CST 2008.
$ echo "Today is:  $cmd1"
Today is:  date
$
```

As the value of the cmd3 variable is `date` (date is in the back-quotes), that means the execution result of date is assigned to the cmd3 variable. Hence, when the shell executes the echo command, it executes the date command, substitutes the position of the date with its result (Tues Aug 19 08:04:56 CST 2008) and then displays the whole sentence on the screen.

However, as the date value of the cmd1 variable is not in the back-quotes, the shell just shows its value directly.

```
$ echo "Today is:  `date`."
Today is:  Tues Aug 19 08:04:56 CST 2008.
$
```

This command functions the same as the echo command with the cmd3 variable.

9.3.3 Resetting Variables

Having been known, in Bourne shell, if not initialized explicitly by the user, a shell variable is initialized by the kernel to a null string by default. As a variable can be assigned a value, the value of a variable can also reset to null. It can be explicitly assigned to null. For example:

```
$ fname4=
$ echo "$fname4"
$
```

As the uname4 variable is assigned to null, the echo command display an empty line.

In Bourne shell, there is a specific command to have the reset-variable function. It is an unset command.

The syntax and function of the unset command in Bourne shell are as follows.

```
$ unset val1 val2 val3
```

Function: to reset the variables of val1, val2, and val3 to null.

Note: The vali list can go on and on for a number of variables in one command line if necessary.

For example:

```
$ fname4=file3 uname4=Hua
$ echo "$fname4 $uname4"
file3 Hua
$ unset fname4 uname4
$ echo "$fname4"
$ echo "$uname4"
$
```

When using the unset command to reset the variables of fname4 and uname4, the values of them become null.

9.3.4 Exporting Variables

When a variable is created and assigned with a value, it is local and its value

is not known to subsequent shells. In other words, after a variable is created and assigned with a value, some script that uses the variable is executed without the value passed to it and may get an invalid value.

Thus, in a UNIX shell, to make the subsequent execution of scripts get a valid value of a variable, use the export command to pass the value of a variable to subsequent shells before using the variable in some scripts.

In a UNIX shell, to execute an external command, the kernel forks a child shell and executes the external command in the child shell. When the command execution is finished, the child process is terminated, and the control comes back to the parent shell. For users, it functions like that the parent shell is a caller and the external command executed in the child shell is called by the parent shell. Remember that the login shell is the parent process of all the processes for all the commands executed under it.

A shell script is executed in the same way as an external command. Thus, when a shell script is called and executed in the parent shell or another shell script, it does not get automatic access to the variables defined in the caller (the parent shell or the original script) unless they are passed to it. If necessary, a variable, no matter which is a user-defined variable or an environment variable, must be passed to the subsequent scripts by using the export command. If readers have not realized that, part of the content of the .profile file that has been introduced in Section 2.6 is copied here again.

```
PATH='/bin: /users/bin: /users/project/bin:'
LESS='eMq'
export PATH LESS
/users/project/wang
date
```

The third line uses the export command to pass the PATH and LESS variables. Just because they are exported at the time they are initialized, all writable shell environment variables are available to every command, script, and subshell.

The export command's function works like to make a variable global.

The syntax and function of the export command in the Bourne shell are as follows.

```
$ export val1 val2 val3
```

Function: to export the variables of val1, val2 and val3 and their values to every execution of commands or scripts from this point on.

Note: The vali list can go on and on for a number of variables in one command line if needed.

The following several examples present how the export command to be used.

```
$ fname2=file2
$ export fname2
$
```

Some more examples will be given in the following to show the function of the export command. By using one of the text editors that have been

discussed in Chapter 3, create a Bourne shell script called dptext1, which is similar to the detext0 script in Section 9.1.1.

```
$ cat dptest1
echo $fname2
exit 0
$ dptest1
file2
$ fname2=file1
$ dptest1
$
```

As the first fname2 assignment is exported, the first execution of dptest1 displays file2. But the second assignment of the fname2 to file1 is not exported, and the fname2 in the dptest1 script is not initialized and automatically set to null. Hence, the result of the second execution of the dptext1 is an empty line.

Now let us create two other scripts like the dptest2 and dptest3 scripts. The dptest2 script assigns a new value file3 to the fname2 variable without exporting it. The dptest3 script assigns a new value file4 to the fname1 variable and exports it.

```
$ cat dptest2
echo $fname2
fname2=file3
echo $fname2
exit 0
$ cat dptest3
fname2=file4
export fname2
dptest1
dptest2
dptest1
exit 0
$ dptest3
file4
file4
file3
file4
$ echo $?
0
$
```

When the dptest3 script is executed, firstly, the value of the fname2 variable is assigned to file4 and exported. The dptest1 is called and displays that the value of the fname2 variable is file4. Then the dptest2 is called. It, firstly, displays the file4 value of the fname2. It changes the value of the fname2 to file3, but does not export the variable. The dptest2 continues to display the value of the fname2 again, but this time, its value is file3 that is local to the dptest2 script. When the dptext2 finishes and returns to the dptest3, the dptest3 executes the next dptest1. As the file3 value of the local fname2 in the dptest2 script is not exported, it is not known by the subsequent execution of the dptest3 script. So the dptest1 displays that the value of the fname2 variable is still file4. Hence, the whole result of the dptest3 execution is file4 file4 file3 file4. The final echo command shows that the exit status of

the dptest3 is success.

9.3.5 Making Variables Read-only

Having had the experience of C language programming, programmers know that symbolic constants sometimes are needed, especially for the situation that some certain constant can be used in many places scattered in the code (Sarwar et al 2006; Stallings 1998). When the literal value of the constant needs to change, with a symbolic constant, the change can be made at one place only, but the constant will be changed everywhere in the code along with. For the operating system kernel, this mechanism can help make the code portable. In other words, if some constant is dependent on the hardware device, programmers can make a symbolic constant for it when programming and then they can change its literal value that is appropriate to the hardware device where the software should be moved on when the software is applied.

In the Bourne shell, the readonly command can make a variable function like a symbolic constant.

The syntax and function of the readonly command in the Bourne shell are as follows.

```
$ readonly val1 val2 val3
```

Function: to make the variables of val1, val2 and val3 read-only.

Note: The vali list can go on and on for a number of variables in one command line if needed. If the readonly command is executed without arguments, it displays all read-only variables and their values.

The following several examples present how the readonly command to be used.

```
$ fname3=.profile
$ uname3=Superuser
$ readonly fname3 uname3
$ echo "$fname3 $uname3"
.profile Superuser
$ uname3="Root"
uname3:  is read only
$
```

The third command makes the fname3 and uname3 variables read-only, so when a change of the value of the uname3 is tried in the fifth command, an error message is shown.

9.3.6 Reading Standard Input

To create a script that can allow the user to interact with it through the

terminal, it needs the read command. An interactive shell script can prompt and wait for the user's input from the keyboard, and store the user input in a shell variable.

The syntax and function of the read command in the Bourne shell are as follows.

```
read val1 val2 val3
```

Function: to read one line from standard input and assign words in the line to the variables of val1, val2, and val3.

Note: The vali list can go on and on for a number of variables in one command line if needed. For the read command with several arguments, one word is read unit and the word alignment order from left to right is the read-in order. The words in a line are separated by white spaces (Space or Tab keys, depending on the value of the shell environment variable IFS; see the result of the set command in Section 9.2).

The words are assigned to the variables one by one in the order of the variable alignment in the command and the input order from the terminal, from left to right. If the number of variables in the list is less than the number of words in the input line, the last variable is assigned the rest words. If the number of variables is greater than the number of words in the input line, the rest variables are reset to null.

For example: create a Bourne script named readtest by using a text editor and execute it.

```
$ cat readtest
echo "Enter a sentence with three words:  \c"
read word1 word2 word3
echo "The first word you typed is:  $word1"
echo "The second word you typed is:  $word2"
echo "The rest of the line you typed is:  $word3"
exit 0
$ readtest
Enter a sentence with three words:  Hello, welcome to the UNIX world!
The first word you typed is:  Hello,
The second word you typed is:  welcome
The rest of the line you typed is:  to the UNIX world!
$
```

The first command in the readtest script is to let the user type in three words after the prompt: "Enter a sentence with three words:". The special character "\c" means to print line without newline (see Section 9.3.1), so the cursor prompt waits for the user to type in the words on the same line. The second command is to read in three words from the standard input and store the words in the variables of word1, word2, and word3, respectively. The third command displays on the screen "The first word you typed is: $word1", and the value of the word1 variable replaces the position of the word1 variable on the screen. The forth command displays on the screen "The second word you typed is: $word2", and the value of the word2 variable replaces the position of the word2 variable on the screen. And the fifth command displays "The rest of the line you typed is: $word3", and the value of the word3 variable is

displayed at the position of the word3 variable on the screen.

When the readtest script is executed, the user types in "Hello, welcome to the UNIX world!" Hence, the following two lines displayed on the screen are "The first word you typed is: Hello," and "The second word you typed is: welcome." As the number of the words that the user types in is more than three, the value of the word3 variable is the rest of the words — "to the UNIX world!" and the final line displayed on the screen is "The rest of the line you typed is: to the UNIX world!"

9.4 Shell Scripts' Argument Transport

Mentioned in Section 9.2, except the writable environment variables, there are still some more environment variables that are usually read-only, for example, positional parameters, the environment variables that are used to hold the exit status of most recent command, the PID of the current process, etc. Table 9.1 displays some of the read-only environment variables.

Table 9.1 Some of the read-only environment variables

Environment Variable	Use
	Positional parameters
$0	Name of the command program or script
$1 $2 $3 $4 $5 $6 $7 $8 $9	Values of the command line arguments, from argument 1 to argument 9, respectively
$*	Values of all command line arguments
$@	Values of all command line arguments; but if $@ is enclosed in quotes, such as in "$@", each argument individually quoted
$#	The number of the total command line arguments
	Others
$?	Exit status of the most recent command
$$	Process ID (PID) of the current process
$!	PID of the most recent background process

The set command can be used to change the values of some of the read-only environment variables.

9.4.1 Shell Positional Parameters

The shell positional parameters are used to pass command line arguments to shell scripts. The biggest number of the arguments is nine. That means, a shell script can store its arguments, at most, in nine variables, from $1 to

$9. Therefore, programmers can use the names of these variables to refer to the values of these arguments. If it is not passed an argument, the positional parameter has a value of null string.

The $0 variable contains the command name of the script file. The $# variable contains the total number of arguments passed in an execution of a script. Both of the variables $* and $@ contain the values of all of the arguments. But $@ has its special function that when it is used as "$@", each individual argument is divided in quotes.

An example is given here to show how the positional parameters to function. Create a shell script named dptext4 by using a text editor.

```
$ cat dptext4
echo "Command name is: $0."
echo "Number of command arguments passed as parameters is: $#."
echo "Values of the command arguments are: $1 $2 $3 $4 $5 $6 $7 $8 $9."
echo "If using the \$* to show them, values are: $*."
echo "If using the \$@ to show them, values are: "$@"."
cat $1 $2 $3 $4 $5 $6 $7 $8 $9
exit 0
$ dptext4 file1 file2 file3 file4
Command name is: dptext4.
Number of command arguments passed as parameters is: 4.
Values of the command arguments are: file1 file2 file3 file4.
If using the $* to show them, values are: file1 file2 file3 file4.
If using the $@ to show them, values are: "file1" "file2" "file3" "file4".
...... The content of file1 is shown here.
...... The content of file2 is shown here.
...... The content of file3 is shown here.
...... The content of file4 is shown here.
$
```

The first line of the dptext4 script is to show the command name of the script. The second line is to display the number of command line arguments that are passed to the script. The third line is to show the values of all the command line arguments. The forth and fifth lines are also to display the values of all the command line arguments, but the forth line uses the $* variable and the fifth line uses the "$@" variable. As the "$@" variable is in quotes, the values of all the command line arguments are divided by quotes, too. The second last line shows how the positional parameters can be used to substitute the arguments of a UNIX command. When the dptext4 is executed with four arguments as shown, the expected result is displayed on the screen.

9.4.2 Setting Values of Positional Parameters

The set command can be used to assign values to the variables of $1, $2, $3, $4, $5, $6, $7, $8, and $9, but the set command cannot do the same work to the $0 variable. The positional parameters can also be set as the output of some commands.

The syntax and function of the set command in Bourne shell are as follows.

```
set [options] [a list of arguments]
```

Function: Without an argument, the set command displays names of all shell variables and their current values; with a list of arguments, the set command sets values of the positional parameters to the arguments in the list.

Note: There are two ways to set the positional parameters by using the set command. One is that the set command is used to set the values of positional parameters directly by using a list of arguments. For example:

```
$ cat dptext5
set file1 file2 file3 file4
cat $1 $2 $3 $4 $5 $6 $7 $8 $9
exit 0
$ dptext5
...... The content of file1 is shown here.
...... The content of file2 is shown here.
...... The content of file3 is shown here.
...... The content of file4 is shown here.
$
```

The first command line in the dptext5 script sets the values of positional arguments, $1, $2, $3, and $4, to file1, file2, file3 and file4, respectively. The second command line concatenates them and displays them on the screen.

The other one is that the set command is used to take the output of one command as the arguments and pass these arguments to the positional parameters one word after another, from left to right. For example:

```
$ cat dptext6
date
set `date`
echo "$*"
echo "$2 $3, $6"
exit 0
$ dptext6
Thurs August 28 11:24:11 CST 2008
Thurs August 28 11:24:11 CST 2008
August 28, 2008
$
```

The second line, the set `date` (in back-quotes) command, in the dptext6 script sets the values of positional arguments, $1 through $9, to the output of the date command, one parameter to one field of the output. The displayed result of the date command has six fields. Here, $1 is set to Thurs, $2 to August, $3 to 28, $4 to 11:24:11, $5 to CST, and $6 to 2008. The third command displays the values of all the positional arguments on the screen. The forth command line displays the values of the variables, $2, $3, and $6, which are the same as the time of that day in a commonly used form.

Another example:

```
$ cat dptext7
filename="$1"
ls -l `$filename`
set `ls -l $filename`
owner="$4"
size="$6"
```

```
echo "Filename\tOwner\tSize"
echo "$filename\t$owner\t$size"
exit 0
$ dptext7 file1
-rw-r--r-- 1 wang project 797 Jan 16 15:02 file1
Filename   Owner  Size
file1      wang   797
$
```

The first line in the dptext7 script assigns the filename variable the value of positional argument $1. The second line lists the information of the file that is typed in from the keyboard as the argument of the dptext7 script. The third line sets the values of positional parameters, $1 through $9, to the output of the same command as the second line. The forth line and fifth line assign the owner and size variables the values of $4 and $6, respectively. The seventh line displays the values of the filename, owner and size variables. The special character \t in the sixth and seventh echo commands is used to display a Tab character (see the echo command in Section 9.3.1).

When the dptext7 script is executed with the file1 arguments (the file1's information can be seen in Section 6.3.2), the values of the filename, owner and size variables are assigned file1, wang and 797, respectively, shown as the above result, which are the name, owner and size of the file1.

9.4.3 Shift Command

Although there are at most nine command line arguments assigned to a script at a time, scripts can take more than nine arguments. To do so, use the shift command.

The syntax and function of the shift command in Bourne shell are as follows.

```
shift [n]
```

Function: to shift the command line arguments n positions to the left.

Note: Without the number argument n, the shift command makes the command line arguments move to the left by one position, having $2 become $1, $3 become $2, and so on, and the first argument, $1, shift out. If the n is greater than one, the number of positions to be shifted to left is more than one. Once shifted out, the arguments cannot be restored.

For example:

```
$ cat dptext8
echo "Command name is: $0."
echo "Values of the command arguments are: $1 $2 $3 $4 $5 $6 $7 $8 $9."
shift
echo "Values of the command arguments are: $*."
shift 2
echo "Values of the command arguments are: $*."
shift 3
```

```
echo "Values of the command arguments are: $*."
exit 0
$ dptext8 A B C D E F G H I J K L M N
Command name is: dptext8.
Values of the command arguments are: A B C D E F G H I.
Values of the command arguments are: B C D E F G H I J.
Values of the command arguments are: D E F G H I J K L.
Values of the command arguments are: G H I J K L M N.
$
```

The first line of the dptext8 script is to show the command name of the script. The second line is to show the values of all the command line arguments. The third line is to shift the command line arguments one position to the left. The fifth line is to shift the command line arguments two positions to the left. The seventh line is to shift the command line arguments three positions to the left. Therefore, the second line in the result of the execution of the dptext8 shows that the A is shifted out; the third line in the result of the execution of the dptext8 displays that the B and C are shifted out; the forth line in the result of the execution of the dptext8 is that the D, E, and F are not there. And the total number of the arguments is 14, greater than 9.

9.5 How to Execute a Bourne Shell Script

A Bourne shell script can be executed in two different situations: One is in Bourne shell; the other is in a different shell. Two common ways to execute a Bourne shell script are given in this section. There are some more ways that can be used to execute a script, which are more dependent on the version of the UNIX operating system. Readers can check out the on-line user manual.

Before the discussion of the first way, the chmod command should be introduced.

9.5.1 Setting File Access Permissions

To make a script file executable, add the execute permission to the access permissions for the file. To do so, use the chmod command.

The syntax and function of the chmod command are as follows.

```
$ chmod [options] octal-mode file[s]
$ chmod [options] symbolic-mode file[s]
```

Function: to change or set permissions for files in the arguments.

Common options:

-f: to force specified access permissions; if the file's owner does the change, no error messages will be prompted.

-R: to change permissions recursively descending through directories for

all of the files and subdirectories under each directory.

The access permissions have been introduced simply when the ls command is discussed in Chapter 2.

For the octal mode, three octal numbers are needed, which represent the access permissions for all the users of a file.

There are three types of users and three types of permissions in the UNIX operating system. If 1 bit represents a permission type, 3 bits are needed to indicate file permissions for one type of users (user, group, or others). Hence, the whole permissions for a UNIX file can be represented by a number with nine bits. Each bit can be 1 (permission allowed) or 0 (permission not allowed). One type of users of a file can have one of the eight possible types of permissions for this file. Eight 3-bit values of permissions can be represented by octal numbers from 0 through 7 if 0 means no permissions, and 7 means all (read, write, and execute) permissions, shown in Table 9.2.

Table 9.2 Octal numbers, access permission values and meanings for a user of a file

Octal Number	r	w	x	Meaning
0	0	0	0	No permission
1	0	0	1	Execute permission, no read and write permissions
2	0	1	0	Write permission, no read and execute permissions
3	0	1	1	Write and execute permissions, no read permission
4	1	0	0	Read permission, no write and execute permissions
5	1	0	1	Read and execute permissions, no write permission
6	1	1	0	Read and write permissions, no execute permission
7	1	1	1	Read, write and execute permissions

As there are three types of users and each type needs one octal number (or three bits), the total of three octal numbers (or 9 bits) are needed to express permissions for all three typed of file users. Hence, the possible access permission values are ranged from 000 through 777 (as octal numbers). The first octal digit specifies permissions for the owner of the file, the second digit specifies permissions for the group that the owner of the file belongs to, and the third digit are permissions for others else.

Another way to express access permissions is: a dash (-) represents a permission bit value of 0, and r, w, or x represents a value of 1, depending on the position of the bit according to Table 9.2. Thus, 0 in octal (no permissions allowed) for a user of a file can be written as - - - and 7 (all three permissions allowed) can be expressed rwx. The outputs of the ls -l commands just displays the access permissions in this way (see the ls command in Section 2.4.2).

The symbolic mode, also called mode control word, has three parts: the first part is for who, the third part is for access permission, and in between the first part and the third part is the second part — operator. "Who", "operator", and "permission" can have the values that are shown in Table

9.3.

Table 9.3 Values for who, operator and permission in symbolic mode

Value	Meaning
	Who
u	User (the owner of the file)
g	Group
o	Others
	Operator
+	To add permissions
–	To remove permissions
=	To set permissions
	Permission
r	Read bit
w	Write bit
x	Execute bit for files and search bit for directories
u	User's current permissions
g	Group's current permissions
o	Others' current permissions

The combination of the different values of who, operator, and permission should be meaningful. For example, as a permission, the value of u (g or o) can only be used with the = operator, which means to set some who's permissions equal to user's current permissions (group's current permissions or others' current permissions). Two or three values can be used for 'who' and 'permission' at the same time, such as uo for the 'who' field and rw for the 'permission' field. To show how the octal mode and symbolic mode to function, some examples of the chmod command are given as follows.

Before using the chmod command to change the access permissions for files and seeing how the use of the chmod commands to affect the permissions, use the ls -l command to check the current permissions for the files in the working directory.

```
$ ls -l
-rw-r--r--   1   wang   project   245  Aug  16  15:02    dptext1
-rw-r--r--   1   wang   project   234  Aug  16  15:23    dptext2
-rw-r--r--   1   wang   project   345  Aug  16  15:42    dptext3
-rw-r--r--   1   wang   project   276  Aug  16  16:03    dptext4
-rw-r--r--   1   wang   project   255  Aug  16  17:02    dptext5
-rw-r--r--   1   wang   project   356  Aug  16  17:43    dptext6
-rw-r--r--   1   wang   project   275  Aug  16  18:22    dptext7
-rw-r--r--   1   wang   project   336  Aug  16  18:43    dptext8
-rw-r--r--   1   wang   project   797  Jan  16  15:02    file1
-rw-r--r--   1   wang   project   251  Jan  16  15:03    file2
-rw-r--r--   1   wang   project   255  Aug  16  14:02    readtest
drwxr-xr-x   2   wang   project   512  Apr  15  15:11    text
$ chmod 700 dptext1 dptext2
$ ls -l dptext1 dptext2
-rwx------   1   wang   project   245  Aug  16  15:02    dptext1
-rwx------   1   wang   project   234  Aug  16  15:23    dptext2
$
```

This chmod command sets access permissions for the files of dptext1 and dptext2 to read, write and execute permissions for the owner, and no access permissions to group and others. The following command has the same effect as this command does.

```
$ chmod u=rwx, go= dptext1 dptext2
```

This command sets the user's access permissions for the two files of dptext1 and dptext2 to read, write and execute permissions (u=rwx), and for group and others to none (go=).

```
$ chmod 740 *
$ ls -l
-rwxr-----   1   wang   project   245  Aug  16  15:02    dptext1
-rwxr-----   1   wang   project   234  Aug  16  15:23    dptext2
-rwxr-----   1   wang   project   345  Aug  16  15:42    dptext3
-rwxr-----   1   wang   project   276  Aug  16  16:03    dptext4
-rwxr-----   1   wang   project   255  Aug  16  17:02    dptext5
-rwxr-----   1   wang   project   356  Aug  16  17:43    dptext6
-rw-r-----   1   wang   project   275  Aug  16  18:22    dptext7
-rw-r-----   1   wang   project   336  Aug  16  18:43    dptext8
-rw-r-----   1   wang   project   797  Jan  16  15:02    file1
-rw-r-----   1   wang   project   251  Jan  16  15:03    file2
-rw-r-----   1   wang   project   255  Aug  16  14:02    readtest
drwxr-----   2   wang   project   512  Apr  15  15:11    text
$
```

This chmod command sets access permissions for all the files and subdirectories in the working directory to read, write and execute permissions for the owner, only read permission for the group, and no access permissions for others.

```
$ chmod a-rw dptext3
$ ls -l dptext3
---x------   1   wang   project   345  Aug  16  15:42    dptext3
$
```

This chmod command removes read and write permissions for the dptext3 file of all the users, including owner, group, and others. The a represents all. The following command has the same function as this command has.

```
$ chmod ugo-rw dptext3
```

To make all the users have the read and execute permissions for the dptext4 file, use the following command.

```
$ chmod a+rx dptext4
$ ls -l dptext4
-r-xr-xr-x   1   wang   project   345  Aug  16  15:42    dptext4
$
```

To let the group have the same access permissions for the dptext5 file as the owner does, use the following command.

```
$ chmod g=u dptext5
```

The access permissions for all the files and directories under one or more directories can be set by using the chmod command with the -R option. For

example:

```
$ chmod -R 711 ~/work
```

This command sets access permissions for all the files and directories under the directory called work to 711 recursively, which means set access permissions to the read, write and execute permissions for the owner, the execute permission for group and others.

For the octal mode, one or two octal digits can be used at one time. For example:

```
$ chmod 1 dptext1
$ chmod 71 dptext2
$ ls -l dptext1 dptext2
---------x   1   wang   project   245  Aug  16  15:02    dptext1
----rwx--x   1   wang   project   234  Aug  16  15:23    dptext2
$
```

The first chmod command sets others' access permissions for the dptext1 file to 1 (--x) and the access permissions for the owner and group to 0 (---). The second chmod command sets access permissions of the owner, group and others for the dptext2 file to 0 (---), 7 (rwx) and 1 (--x), respectively. The ls -l command shows the results for these commands.

9.5.2 One Way to Make Bourne Shell Script Executable

The first method is to make the script file executable by adding the execute permission to the existing access permissions for the file. To do so, use the chmod command to make the script executable for the user. For example, to make all the scripts that have been created in Section 9.4 executable, execute the following command:

```
$ chmod a+x readtest dptext?
$
```

This command can make all the users of these files have the privilege to execute these files. If the owner just wants only himself to have the execute permission, he can use the following command, instead.

```
$ chmod u+x readtest dptext?
$
```

The next step is to execute these scripts after a shell prompt on the command line. For example, type the following on the keyboard to execute the readtest file,

```
$ readtest
Enter a sentence with three words:  Hello, welcome to the UNIX world!
The first word you typed is:  Hello,
The second word you typed is:  welcome
The rest of the line you typed is:  to the UNIX world!
$
```

As described in Chapter 4, the script is executed as a child process of the current shell. Note that, with the above method, the script executes properly if it is just in Bourne shell or Korn shell, but not if it is in any other shell. If it is now in C shell, execute the following commands, instead:

```
% /bin/sh
$ readtest
```

The first command does change the shell.

Sometimes, it seems that it does not work to use the chmod command to add the execute permission to a file. The reason may be that the search path of UNIX does not include the pathname of the directory that holds that file. So check out the search path by using echo $PATH if the file cannot be executed directly after its execute permission has been added.

9.5.3 Another Way to Make Bourne Shell Script Executable

The second method of executing a shell script is to execute the /bin/sh command with the script file as its parameter. Thus, the following command executes the dptext6 and dptext8 scripts.

```
$ /bin/sh dptext6
Thurs August 28 11:24:11 CST 2008
Thurs August 28 11:24:11 CST 2008
August 28, 2008
$ /bin/sh dptext8 A B C D E F G H I J K L M N
The command name is:  dptext8.
The values of the command line arguments are:  A B C D E F G H I.
The values of the command line arguments are:  B C D E F G H I J.
The values of the command line arguments are:  D E F G H I J K L.
The values of the command line arguments are:  G H I J K L M N.
$
```

If the search path (the PATH variable, see Section 8.2.3) includes the /bin directory (commonly, it is so), simply use the sh command, such as:

```
$ sh dptext6
Thurs August 28 11:24:11 CST 2008
Thurs August 28 11:24:11 CST 2008
August 28, 2008
$
```

There are also some other methods to execute a script, but they are not as common as the above two and usually dependent on the versions of the UNIX operating system.

9.6 Program Control Flow Statement (a): if Statement

The program control flow statements are used to determine the code execution sequence in a shell script. For programming, usually, there are two

types of statements for controlling the code flow: branching statements and repetitive execution statements. In the Bourne shell statements, for branching, there are if and case statements; for repetitive execution, there are for, while, and until statements.

The simplest use of if statement is for the two-way branching, but it can also be used for more than two of branches. In the following subsections, the discussion will be going from the simplest to the more complex according to the structure and function of if statements.

9.6.1 The Simplest if Statement

The simplest use of if statement is for the two branches.

The syntax, function, and flow diagram of the simplest if statement are as follows.

```
if expression
  then
   command lines
fi
```

Function: to branch the code sequence in the two ways.

Flow diagram is shown in Figure 9.2.

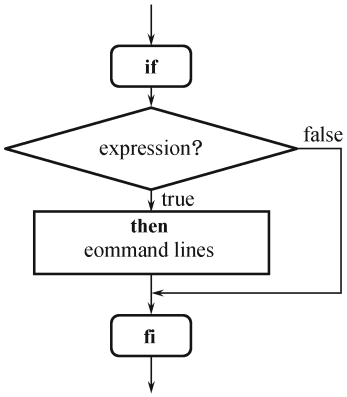

Fig. 9.9 The flow diagram of the simplest if statement.

Note: The words of if, then, and fi are keywords, which must be used in the syntax. The expression can be a list of commands. The execution of commands in expression returns a status of true (success) or false (failure). When if statement begins to execute, the command in expression is executed, firstly. If the return result of expression is true, the command lines after then keyword are executed and then the code execution goes out of the if statement through fi keyword; if false, the code execution goes directly out

of the if statement through fi keyword.

9.6.2 The test Command

The test command can be used as the expression. It evaluates an expression and returns true or false.

There are two syntaxes that have the same function for test command as follows.

```
test expression
[ expression ]
```

Function: to evaluate expression and return true or false status.

Note: The first syntax needs the test keyword; the second expression syntax must include the brackets. At least one space is required before and after an operator, an operand, a parenthesis, or a bracket. If a test expression needs more than one line, before going to the next line by pressing Enter, a backslash (\) must be used in order to let the shell not to treat the next line as a separate command.

There are four groups of operators used in test command: one group is for file testing, one group for integer testing, one group for string testing, and one group for logically connecting two or more expressions to form complex expressions. Tables 9.4, 9.5, 9.6, and 9.7 show the meanings of these operators, respectively. Most of them are supported by test command on most UNIX operating systems.

Table 9.4 Operators for file testing

Expression	Function
-r filename	If the file with the filename exists and is set read permission for the user, its return status is true; otherwise, it is false
-w filename	If the file with the filename exists and is set write permission for the user, its return status is true; otherwise, it is false
-x filename	If the file with the filename exists and is set execute permission for the user, its return status is true; otherwise, it is false
-f filename	If the file with the filename exists and is an ordinary file, its return status is true; otherwise, it is false
-d filename	If the file with the filename exists and is a directory, its return status is true; otherwise, it is false
-p filename	If the file with the filename exists and is a named pipe, its return status is true; otherwise, it is false
-s filename	If the file with the filename exists and is not an empty file, its return status is true; otherwise, it is false
-t file descriptor	If the file descriptor is associated with the terminal, its return status is true; otherwise, it is false. The default file descriptor is 1

Table 9.5 Operators for integer testing

Expression	Function
n –eq m	If the integers n and m are equal, its return status is true; otherwise, it is false
n –ne m	If the integer n is not equal to the integer m, its return status is true; otherwise, it is false
n –lt m	If the integer n is less than the integer m, its return status is true; otherwise, it is false
n –le m	If the integer n is less than or equal to the integer m, its return status is true; otherwise, it is false
n –gt m	If the integer n is greater than the integer m, its return status is true; otherwise, it is false
n –ge m	If the integer n is greater than or equal to the integer m, its return status is true; otherwise, it is false

Table 9.6 Operators for string testing

Expression	Function
s	If the string s is not an empty string, its return status is true; otherwise, it is false
s1 = s2	If two strings s1 and s2 are the same, its return status is true; otherwise, it is false
s1 != s2	If two strings s1 and s2 are not the same, its return status is true; otherwise, it is false
-z s	If the length of the string s is zero, its return status is true; otherwise, it is false
-n s	If the length of the string s is not zero, its return status is true; otherwise, it is false

Table 9.7 Operators for complex expression forming

Operator	Function
! e1	Logical NOT operator: if the e1 expression is false, its return status is true; otherwise, it is false.
e1 -a e2	Logical AND operator: if the expressions e1 and e2 are true, its return status is true; otherwise, it is false.
e1 -o e2	Logical OR operator: if the expression e1 or e2 is true, its return status is true; otherwise, it is false.
(e1)	Parentheses for grouping expressions; at least there is one space before and one after each parenthesis

Here, write a Bourne shell script with if statement.

```
$ cat dpif0
if test -f "$1"
then echo "$1 is an ordinary file."
fi
exit 0
$ dpif0 file1
```

```
file1 is an ordinary file.
$
```

In the dpif0 script, the first line is to test if or not the first argument is an ordinary file. If so, the echo command after the then keyword is executed and displays the argument is an ordinary file.

The second example:

```
$ cat dpif1
# Test whether or not the script is typed with an argument.
# If not, tell the user to type in an argument and exit with status 1.
# If so, test whether or not the argument is an ordinary file.
# If not, tell the user to type in an argument that is an ordinary file
# and exit with status 2.
# If so, display the information of the file.
if test $# -eq 0
then echo "This script must be followed by an argument."
exit 1
fi
if test -f "$1"
then
filen="$1"
set `ls -l $filen`
owner="$4"
size="$6"
echo "$1 is an ordinary file."
echo "Filename\tOwner\tSize"
echo "$filen\t$owner\t$size"
exit 0
fi
echo "$1 must be an ordinary file."
exit 2
$ dpif1
This script must be followed by an argument.
$ dpif1 fil
fil must be an ordinary file.
$ dpif1 file1
file1 is an ordinary file.
Filename      Owner     Size
file1         wang      797
$
```

In the dpif1 script, the first six lines are comment lines that explain how the script to function. Readers can read the comments carefully and know how the script functions. Even though this script just uses the simplest if statements, it realizes three branches: one is for the user's typing the dpif1 script program on the command line with an ordinary file argument, which exits with the success status of 0; one is for the user' s typing the dpif1 script program on the command line without an argument, which exits with the exit status of 1; the third one is for users' typing the dpif1 script program on the command line with an argument that is not an ordinary file, which exits with the exit status of 2. The exit status can be checked out by using the echo $? command.

The third example uses the operators to form the complex expression.

```
$ cat dpif2
# Let the user type a number ranged from 1 to 60 on the keyboard.
```

```
# Test whether or not the number is greater than 60 or less than 1.
# If so, tell the user that it is a wrong number and exit with the
# status of 1.
# If not, test whether or not the number is greater than or equal to 31.
# If not, display the information "You belong to C1" on the screen.
# If so, display the information "You belong to C2" on the screen
# and exit with the status of 0.
echo "Type in a number (1 -- 60):  \c"
read num
if [ "$num" -gt 60 -o "$num" -lt 1 ]
then echo "The number that you have just typed in is a wrong number."
exit 1
fi
if [ "$num" -ge 31 ]
then
echo "You belong to C2."
exit 0
fi
echo "You belong to C1."
exit 0
$ dpif2
Type in a number (1 -- 60):  67
The number that you have just typed in is a wrong number.
$ dpif2
Type in a number (1 -- 60):  3
You belong to C1.
$ dpif2
Type in a number (1 -- 60):  32
You belong to C2.
$
```

In the dpif2 script, the first seven lines are comment lines that have explained how the script to function. Readers can read the comments carefully to know how the script functions. Though this script also uses the simplest if statements, it realizes also three branches: one is for the user's typing a number that is out of the range of 1 – 60, which exits with the status of 1; one is for the user's typing a number that is greater than 30, which exits with the exit status of 0; the third one is for users' typing a number that is less than 31, which exits with the exit status of 0. The executions of the final three command lines (dpif2) show how the script functions. This script simulates to let users type their ID number to go into the system.

9.6.3 The if Statement with the else Keyword

The if statement with the else keyword is also for the two branches. The difference is that if statement with the else keyword can have some command lines after the else keyword in order to do some commands different from the command lines in the then block. In the simplest if statement, however, the code execution goes directly out of if statement with doing nothing if the result of the expression is false.

The syntax, function, and flow diagram of the if statement with the else keyword are as follows.

```
if expression
  then
   command lines
else
   command lines
fi
```

Function: to branch the code sequence in the two ways.
Flow diagram is shown in Figure 9.3.

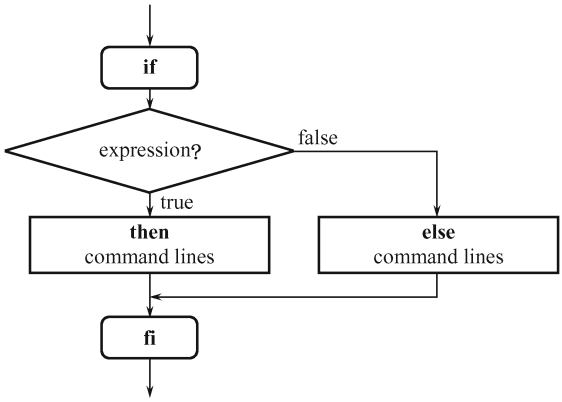

Fig. 9.3 The flow diagram of the if statement with the else keyword.

Note: The word of else is keyword, too, which is optional in the syntax. If the return result of expression is true, the command lines after then keyword are executed and then the code execution goes out of the if statement through fi keyword; if false, the command lines after else keyword are executed and then the code execution also goes out of the if statement through fi keyword.

Here, write the dpif1 and dpif2 scripts in Section 9.6.2 again with the if statement with the else keyword.

```
$ cat dpif1_1
# Test whether or not the script program is with an argument.
# If not, tell the user to type in an argument and exit with status 1.
# If so, test whether or not the argument is an ordinary file.
# If not, tell the user to type in an argument that is an ordinary file
# and exit with status 2.
# If so, display the information of the file.
if test $# -eq 0
then echo "This script must be followed by an argument."
exit 1
fi
if test -f "$1"
then
filen="$1"
set `ls -l $filen`
owner="$4"
size="$6"
echo "$1 is an ordinary file."
echo "Filename\tOwner\tSize"
echo "$filen\t$owner\t$size"
exit 0
```

```
else
echo "$1 must be an ordinary file."
exit 2
fi
$ cat dpif2_1
# Let the user type a number ranged from 1 to 60 on the keyboard.
# Test whether or not the number is greater than 60 or less than 1.
# If so, tell the user that it is a wrong number and exit with the
# status of 1.
# If not, test whether or not the number is greater than or equal to 31.
# If not, display the information "You belong to C1" on the screen.
# If so, display the information "You belong to C2" on the screen
# and exit with the status of 0.
echo "Type in a number (1 -- 60):  \c"
read num
if [ "$num" -gt 60 -o "$num" -lt 1 ]
then echo "The number that you have just typed in is a wrong number."
exit 1
fi
if [ "$num" -ge 31 ]
then
echo "You belong to C2."
else
echo "You belong to C1."
fi
exit 0
$
```

The dpif1_1 and dpif2_1 scripts have the same functions as the dpif1 and dpif2 do, respectively.

9.6.4 Integral Structure of if Statement

The integral structure and function of if command are as follows.

```
if expression1
   then
      command lines
elif expression2
   then
      command lines
elif expression3
   then
      command lines
......
else
      command lines
fi
```

Function: to branch the code sequence in the multiple ways.

Flow diagram is shown in Figure 9.4.

Note: The word of elif is keyword, too, which is optional in the syntax. If the return result of expression1 is true, the command lines after the first then keyword are executed and then the code execution goes out of the if statement through fi keyword; if false, tests the expression2 after the first

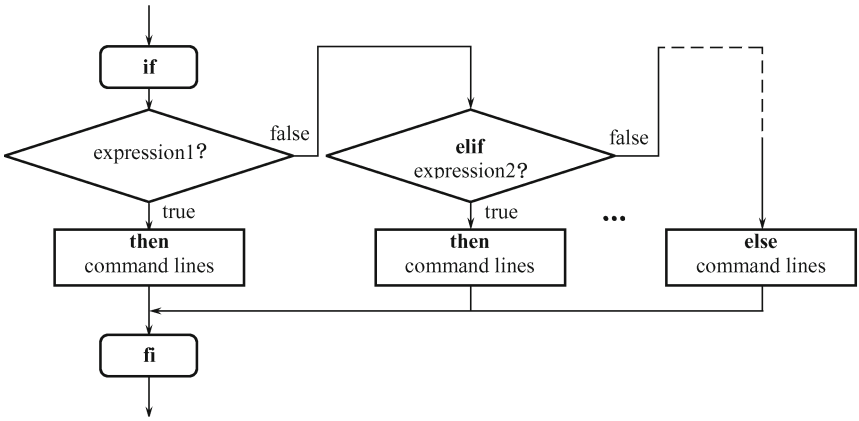

Fig. 9.4 The flow diagram of the integral if statement.

elif keyword. If the return result of expression2 is true, the command lines after then keyword (that is just after the first elif keyword) are executed and then the code execution goes out of the if statement through fi keyword; if false, tests the expression3 after the second elif keyword. In this way, go on and on through all the elif bodies. And finally, if the return result of the final expressionn is false, the command lines after else keyword are executed and then the code execution also goes out of the if statement through fi keyword.

Here, write the dpif1 and dpif2 scripts in Section 9.6.2 again with the integral if statement.

```
$ cat dpif1_2
# Test whether or not the script program is with an argument.
# If not, tell the user to type in an argument and exit with status 1.
# If so, test whether or not the argument is an ordinary file.
# If not, tell the user to type in an argument that is an ordinary file
# and exit with status 2.
# If so, display the information of the file.
if test $# -eq 0
then echo "This script must be followed by an argument."
exit 1
elif test -f "$1"
then
filen="$1"
set `ls -l $filen`
owner="$4"
size="$6"
echo "$1 is an ordinary file."
echo "Filename\tOwner\tSize"
echo "$filen\t$owner\t$size"
exit 0
else
echo "$1 must be an ordinary file."
exit 2
fi
$ cat dpif2_2
# Let the user type a number ranged from 1 to 60 on the keyboard.
# Test whether or not the number is greater than 60 or less than 1.
# If so, tell the user that it is a wrong number and exit with the
```

```
# status of 1.
# If not, test whether or not the number is greater than or equal to 31.
# If not, display the information "You belong to C1" on the screen.
# If so, display the information "You belong to C2" on the screen
# and exit with the status of 0.
echo "Type in a number (1 -- 60):  \c""
read num
if [ "$num" -gt 60 -o "$num" -lt 1 ]
then echo "The number that you have just typed in is a wrong number."
exit 1
elif [ "$num" -ge 31 ]
then
echo "You belong to C2."
else
echo "You belong to C1."
fi
exit 0
$
```

The dpif1_2 and dpif2_2 scripts have the same functions as the dpif1 and dpif2 do, respectively.

9.7 Program Control Flow Statement (b): for Statement

In Bourne shell, there are three statements that are used for repetitive execution on a block of commands in a shell script. These statements are for, while, and until statements. They are also called loops. In this section, for statement is introduced. The other two will be discussed in the next chapter.

9.7.1 The for Statement with a Word List

There are two formats for the for statement. They will be discussed in this section and the next section, respectively.

The syntax, function, and flow diagram of the for statement with a word list are as follows.

```
for variable in w1 w2 w3 w4 ...
do
  command lines
done
```

Function: to execute commands in between the do and done keywords as many times as the number of words (w1, w2, w3, w4, ...).

Flow diagram is shown in Figure 9.5.

Note: The words of the for, do, done, and in are keywords. The words in word list (w1, w2, w3, w4, ...) are assigned to variable one by one, and the command lines in between the do and done, also known as the body of the loop, are executed for every assignment. The process allows the execution of commands in command lines as many times as the number of words (w1, w2,

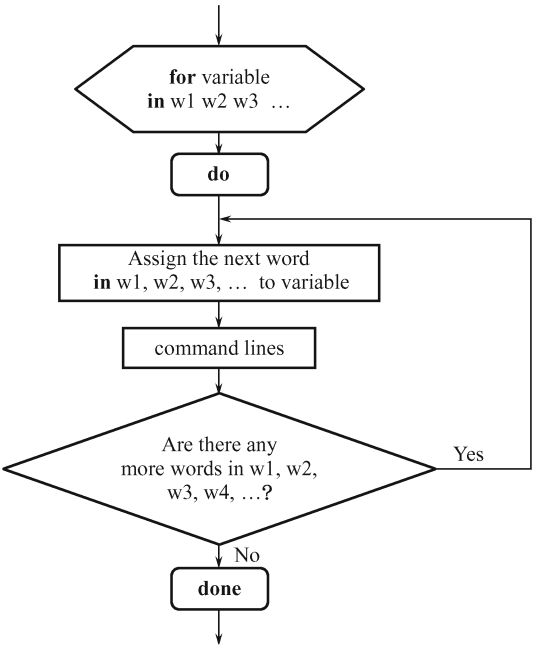

Fig. 9.5 The flow diagram of the for statement with a word list.

w3, w4, . . .).

For example:

```
$ cat dpfor0
for weekday in Monday Tuesday Thursday Wednesday Friday
do
  echo "It is $weekday."
done
exit 0
$ dpfor0
It is Monday.
It is Tuesday.
It is Thursday.
It is Wednesday.
It is Friday.
$
```

In the dpfor0 script, the weekday variable is assigned the words of Monday, Tuesday, Wednesday, Thursday, and Friday one by one and each time, the value of the variable is echoed until no word remains in the list. After that, control goes out of the for statement, and the command following done is executed, which, here, is the exit command.

9.7.2 The for Statement without a Word List

The syntax, function, and flow diagram of the for statement without a word list are as follows.

```
for variable
do
  command lines
done
```

Function: to execute commands in between the do and done keywords as many times as the number of positional parameters.

Flow diagram is shown in Figure 9.6.

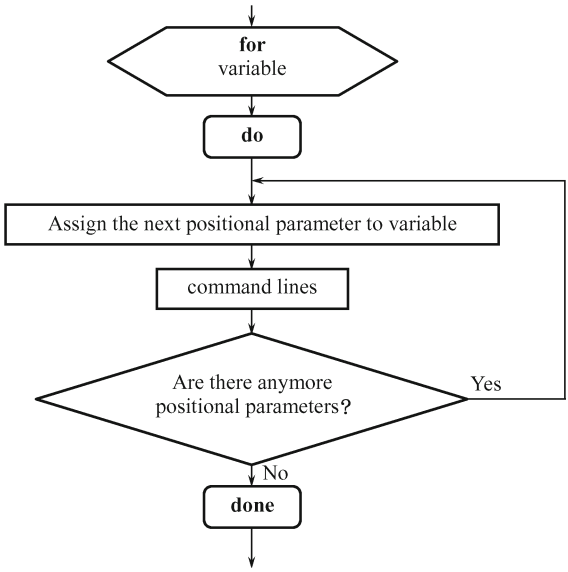

Fig. 9.6 The flow diagram of the for statement without a word list.

Note: The positional parameters are assigned to variable one by one, and the command lines in between the do and done are executed for every assignment. The process allows the execution of commands in command lines as many times as the number of the positional parameters.

For example:

```
$ cat dpfor1
# display the contents of files under the working directory
# $1, $2 -- the name of files
for name
do
  if [ -f $name ]
    then cat $name
      echo "End of $name."
    else echo "$name is an invalid file name."
  fi
```

```
done
exit 0
$ dpfor1 file1 file2 file3 fil
...... The content of the file1 is displayed here.
End of file1.
...... The content of the file2 is displayed here.
End of file2.
...... The content of the file3 is displayed here.
End of file3.
Fil is an invalid file name.
$
```

In the dpfor1 script, the name variable is assigned the positional parameters one by one. The positional parameters should be file names. Each time the value of the name variable is assigned, if it is a valid file name, the content of the corresponding file is displayed on the screen and the sentence of "End of $name." is displayed at the end of the file. If it is not a valid file name, the sentence of "$name is an invalid file name." is displayed. When there is not any more positional parameter, control goes out of for statement, and the command following done is executed, which is the exit command here.

The for statement can also be written in a user-interacting script. For instance,

```
$ cat dpfor2
# display the contents of files under the working directory
echo "Enter the name(s) of file(s) separated by spaces:  \c"
read fname
for fn in $fname
do
   if [ -f $fn ]
        then cat $fn
             echo "End of $fn."
        else echo "$nf is an invalid file name."
   fi
done
exit 0
$ dpfor2
Enter the name(s) of file(s) separated by spaces: file1 file2 file3 fil
...... The content of the file1 is displayed here.
End of file1.
...... The content of the file2 is displayed here.
End of file2.
...... The content of the file3 is displayed here.
End of file3.
Fil is an invalid file name.
$
```

The dpfor2 script can do the similar job as the dpfor1 script does. But, in the dpfor2 script, the fn variable of the for statement is assigned the value of the fname variable that is read from the standard input. If the value of the fname is a list of words that are separated by spaces, the for statement takes the list as the list of words that the value of the fn variable is replaced with one at a time. After that, the execution of the script firstly waits for the user's input. The script also tests if or not the input words are filenames. The words that the user types in are expected file names. Each time the value of the fn variable is assigned, if it is a valid file name, the content of the corresponding

file is displayed on the screen and the sentence of "End of $fn." is displayed at the end of the file. If it is not a valid file name, the sentence of "$fn is an invalid file name." is displayed. When there is not any more word in the list, control goes out of the for statement, and the command following done is executed.

9.8 Summary

The Bourne shell is not only a command interpreter but also a programming language. It can be used by UNIX users to write shell scripts in order to perform various tasks that cannot be done by any single UNIX command. A shell script is a program that can be stored in disk as a file. Bourne shell allows a script to call other script. The caller script makes a call for a called script when the script name appears in the caller script code and the shell is running at that point of the caller script code. And when the task of the called script is performed, it can go back to the calling point in the caller script by using the exit command and the caller script execution resumes. Also, Bourne shell allows multi-level calling. There are different levels of the callers and the called scripts in the Bourne shell scripts. The exit command has only one argument that is optional and an integer number. The argument is returned to the calling process as the exit status of the called process. When the argument is 0, it means that the termination of the called process is success; if a nonzero number, the termination is failure.

A program header is usually a group of comments at the top of every shell script, which is used to put together the comments and describe the main purpose of the whole script. Shell variables can be divided into two types: shell environment variables and user-defined variables. Except the writable environment variables introduced in Chapter 8, there are still some more environment variables that are usually read-only, such as positional parameters and those environment variables that are used to hold the PID of the current process, the exit status of most recent command, etc. User-defined variables are created by UNIX users according to their needs. They are temporary and their values can be changed. They also can be defined as read-only by the users. For Bourne shell, the value of a variable is always treated as a string of characters, even if only digits are in it. To display environment variables, the env and set commands can be used. The echo command can be used to read shell variables, too. To assign value to variable, use the assignment statement. In an assignment statement, single quotes (' '), double quotes (" ") and back-quotes (` `) work differently. To make the subsequent execution of scripts get a valid value of a variable, use the export commands to pass the value of a variable to subsequent shells. The read command can make an interactive shell script prompt and wait for the user's input from the keyboard, and store the user input in a shell variable.

The shell positional parameters are used to pass command line arguments to shell scripts. The $0 variable contains the command name of the script file. The $# variable contains the total number of arguments passed in an execution of a script. Both of the variables $* and $@ contain the values of all of the arguments. The set command can be used to assign values to the variables of $1, $2, $3, $4, $5, $6, $7, $8, and $9, but the set command cannot do the same work to the $0 variable. The positional parameters can also be set as the output of some commands. Although there are at most nine command line arguments assigned to a script at a time, scripts can take more than nine arguments by using the shift command.

To make a script file executable, add the execute permission to the access permissions for the file by using the chmod command. A Bourne shell script can be executed in two different situations: one is in the Bourne shell; the other is in a different shell. Two common ways to execute a Bourne shell script are given in this chapter.

The program control flow statements are used to determine the code execution sequence in a shell script. For programming, commonly, there are two types of statements for controlling the code flow: branching statements and repetitive execution statements. In the Bourne shell statements, for branching, there are if and case statements; for repetitive execution, there are for, while, and until statements. The test command can be used to evaluate an expression and return true or false. In this chapter, only the if and for statements have been discussed. Others will be introduced in the next chapter.

Problems

Problem 9.1 What is a shell script? How can a shell script call another script? When the called script is finished, how can it return its caller program?

Problem 9.2 When we put an exit command in a script code, for the script's caller, what can the point that the exit command appears in the script code be seen as? What is the exit status of the exit command used for? How can a caller script check out the exit status value of this script?

Problem 9.3 Why is it a good programming habit to put a program header and comments in every program?

Problem 9.4 What is a shell variable? How many types can shell variables be divided? What are they? What is a read-only variable? Give some examples to explain them. How can you make a user-defined variable read-only?

Problem 9.5 For the Bourne shell, what data type is the value of a variable always treated as? If not initialized explicitly by the user, by default, what is a shell variable initialized to by the kernel?

Problem 9.6 Try to list the environment variables in your current UNIX operating system by using the env and set commands. Write down their displayed results and compare them.

Problem 9.7 What differences do they have when the echo command has an argument that has some words that are enclosed in single quotes (‘ ’), double quotes (“ ”) or back-quotes (` `)? Give some examples to explain them.

Problem 9.8 Type the following commands on your system, and write down each of their results shown on the terminal screen. Explain the difference between the variables of cmd and cmd1.

```
cmd=‘date’
$ cmd1=date
$ echo "Today is:  $cmd."
$ echo "Today is:  $cmd1."
```

Problem 9.9 What happens when typing the following sequence of shell commands?

```
name=pwd
$ echo $name
$ echo `$name`
```

Problem 9.10 Create the dptest1, dptest2, and dptest3 scripts (in Section 9.3.4) on your system. See how the export command functions, and give your explanation.

Problem 9.11 Create a Bourne shell script as the following, execute it, and see its result. Write down the result and give your explanation how the script functions.

```
$ cat dptext5
filename="$1"
set `ls -l $filename`
owner="$4"
size="$6"
echo "Filename\tOwner\tSize"
echo "$filename\t$owner\t$size"
exit 0
$
```

Problem 9.12 Write a Bourne shell script that takes an ordinary file as an argument and removes the file if its size is zero. Otherwise, the script displays file’s name, size, and owner on one line.

Problem 9.13 Write a Bourne shell script that takes the positional parameters of the command line as the variable of for statement. The positional parameters should be file names. If the positional parameter is a valid file name, the content of the corresponding file is displayed on the screen. If it is not a valid file name, send the wrong message to the screen. When there is not any more positional parameter, control goes out of for statement and exit the script.

Problem 9.14 Write a Bourne shell script that prompts some message to let the user type in eight integers firstly. If the number of the typed

integers is not enough or greater than eight, the script displays the error message. If the number is eight, the script does the following work: it checks the data one by one; it finds the even integers and add them together; it finds the odd integers and multiplies them one by one. And it finally displays the sum with some descriptive message and the product with some explanation, respectively.

References

Bach MJ (2006) The design of the UNIX operating system. China Machine Press, Beijing

Bourne SR (1978) The UNIX shell. Bell Syst Techn J, 57(6) Part 2: pp 1971 – 1990

Bourne SR (1983) The UNIX system. Addison-Wesley, Reading, Massachusetts

Joy WN (1980) An introduction to the C shell. UNIX Programmer's Manual, 4.2 Berkeley Software Distribution. Computer Systems Research Group, Depat. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Aug 1980

Korn D (1983) KSH - a shell programming language. USENIX Conference Proceedings, Toronto, Ontario, June 1983, pp 191 – 202

Miller RC, Myers BA (2000) Integrating a command shell into a web browser. 2000 USENIX Annual Technical Conference, San Diego, California, USA, 18 – 23 June 2000. http://www.usenix.org/events/usenix2000/general/generaltechnical.html. Accessed 24 Sep 2009

Mohay G, Zellers J (1997) Kernel and shell based applications integrity assurance. ACSAC'97: The IEEE 13th Annual Computer Security Applications Conference, 1997, pp 34 – 43

Quarterman JS, Silberschatz A, Peterson JL (1985) Operating systems concepts, 2nd edn. Addison-Wesley, Reading, Massachusetts

Ritchie DM, Thompson K (1974) The Unix time-sharing system. Commun ACM 17 (7): 365 – 375

Rosenblatt B, Robbins A (2002) Learning the Korn shell, 2nd edn. O'Reilly & Associates, Sebastopol

Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: the textbook, 2nd edn. China Machine Press, Beijing

Stallings W (1998) Operating systems: internals and design principles, 3rd edn. Prentice Hall, Upper Saddle River, New Jersey

10 How to Program in Bourne Shell (2)

As a programming language, the Bourne shell has been discussed partly in Chapter 9. In order to program more advanced scripts, it is necessary to learn more statements and commands that can be used to program. In this chapter, the discussion on program control flow statements will be continued, which includes case, while, until, break, and continue statements. Except these statements, there are some more topics: how to process numeric data, the Bourne shell support of functions, and how to debug shell scripts.

10.1 Program Control Flow Statement (c): case Statement

Mentioned in Section 9.6, in the Bourne shell statements, there are two statements for branching, which are if and case statements. The if statement has been discussed in Chapter 9. Here, introduce the case statement.

The case statement provides a multi-way branching method similar to an integral if statement with several elif blocks. However, the structure provided by the case statement is more clear and readable.

The syntax, function, and flow diagram of case statement are as follows.

```
case string in
   pattern1)
      command lines
       ;;
   pattern2)
      command lines
        ;;
      ......
   patternn)
      command lines
   ;;
esac
```

Function: to branch the code sequence in the multiple ways.

Flow diagram is shown in Figure 10.1.

Note: The words of case, in, and esac are keywords, which must be used in the syntax. The single right parenthesis ()) after each pattern is a must,

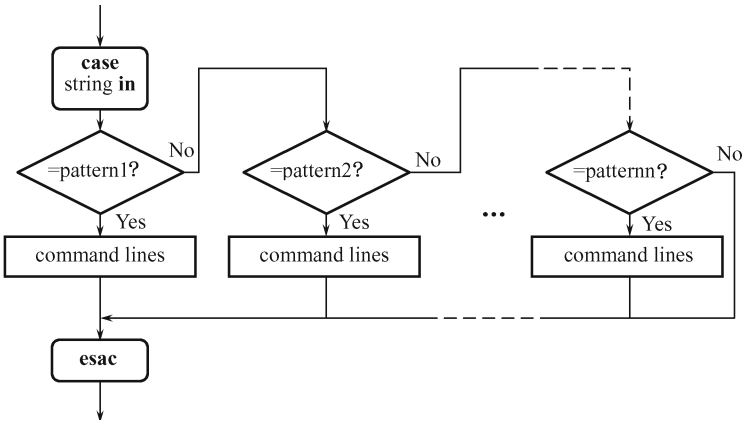

Fig. 10.1 The flow diagram of the case statement.

too. Double semicolons (;;) must be used to delimit a group of command lines for each pattern. Without them (;;), the shell will be confused about where to stop for one pattern and mistakenly make the first command in the command lines of the next pattern execute and result in unusual behavior. When the case statement is executed, firstly, the case statement compares the value of string with the values of all the patterns one by one until either a match is found or there is no matched pattern. If a match is found, the command lines just after the corresponding pattern are executed and then the instruction goes out of the case statement. If no matched pattern is found, the instruction goes out of the case statement directly. Sometimes, a wildcard (*) pattern is used as the default case, which can match any value of string (Bourne 1978; Sarwar et al 2006). The default case allows execution of a special set of command lines that are used to handle an exception (error) condition here, that is, for all the situations in which the value of string does not match any of defined patterns. However, never put the default case in the first pattern. If the default case is at the first pattern, whatever option is made will match the default case because the default case matches any value of string and the following patterns will not be matched again.

Two examples are given as follows.

```
$ cat dpcase1
echo "The following is a command menu. You can make your choice."
echo "       c:  To display a calendar of this month."
echo "       d:  To display today's date and present time."
echo "       l:  To list the files in the working directory."
echo "       p:  To see what is the pathname of the working directory."
echo "       q:  To quit this script."
echo "Type your option and press Enter:  \c."
read opt
case "$opt" in
  c) cal
     ;;
  d) date
```

```
    ;;
  l) ls
    ;;
  p) pwd
    ;;
  q) exit 0
    ;;
  *) echo "There is not an item for your choice."
    exit 1
    ;;
esac
exit 0
$ dpcase1
The following is a command menu. You can make your choice.
      c:  To display a calendar of this month.
      d:  To display today's date and present time.
      l:  To list the files in the working directory.
      p:  To see what is the pathname of the working directory.
      q:  To quit this script.
Type your option and press Enter:  d.
Mon Sept 1 17:20:20 CST 2008
$ dpcase1
The following is a command menu. You can make your choice.
      c:  To display a calendar of this month.
      d:  To display today's date and present time.
      l:  To list the files in the working directory.
      p:  To see what is the pathname of the working directory.
      q:  To quit this script.
Type your option and press Enter:  x.
There isn't an item for your choice.
$
```

The dpcase1 script functions like a command mini-menu. It displays a menu of command options and prompts the user to type an option. The choice of the user is read into a variable called opt. The case statement then matches the choice with one of the five valid patterns one by one. When a match is found, the corresponding command is executed. The first execution example shows that the user chooses the d item, the date command is executed and the today date is displayed, and then the instruction goes out of the case statement and exits the script program. The second execution example shows that the user chooses the x item that is not available in this script, so the default case is executed, the error message is displayed, and the script is exited with a failure status.

To accept the capital letters, the dpcase1 script should be adjusted as follows.

```
$ cat dpcase2
echo "The following is a command menu. You can make your choice."
echo "    c or C: To display a calendar of this month."
echo "    d or D: To display today's date and present time."
echo "    l or L: To list the files in the working directory."
echo "    p or P: To see what is the pathname of the working directory."
echo "    q or Q: To quit this script."
echo "Type your option and press Enter:  \c."
read opt
case "$opt" in
  c|C) cal
  ;;
```

```
  d|D) date
  ;;
  l|L) ls
  ;;
  p|P) pwd
  ;;
  q|Q) exit 0
  ;;
  *) echo "There is not an item for your choice."
    exit 1
  ;;
esac
exit 0
$
```

The dpcase2 script functions just like the dpcase1 script, except that the dpcase2 script accepts the capital letters. Here, the vertical line (|) represents the logical OR operator.

10.2 Program Control Flow Statement (d): while Statement

In the Bourne shell statements, for repetitive execution, there are for, while, and until statements. The for statement has been discussed in Chapter 9. In this chapter, introduce the while and until statements.

The while statement makes a block of code execute repeatedly according to the condition of an expression.

The syntax, function, and flow diagram of the while statement are as follows.

```
while expression
do
  command lines
done
```

Function: to execute command lines as long as expression is logically true.

Flow diagram is shown in Figure 10.2.

Note: The words of while, do, and done are keywords, which must be used in the syntax. If the expression is true, the command lines in between the do and done keywords are executed. In the command lines, there are some lines that are used to assign the new value to the variable in the expression and the expression is evaluated again. One execution of expression evaluation and command lines is called once iteration. The iteration is repeated until the expression is false. After that, the instruction comes out of the while statement and the command following done is executed.

The while statement must be programmed so correctly that the condition that the expression represents can become logically true or false expectedly and properly. Sometimes, poor programming can cause infinite loops without termination.

To terminate a process with an infinite loop, use CTRL-C or the kill

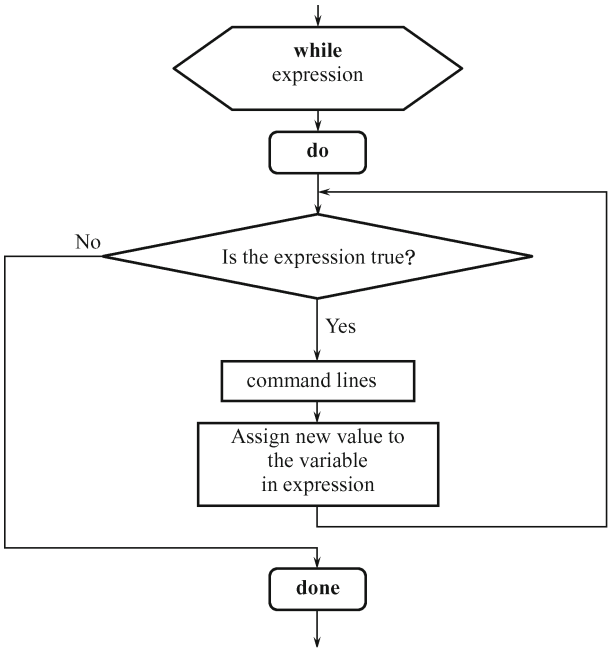

Fig. 10.2 The flow diagram of the while statement.

command (see Section 4.5.5). If the process is running in the foreground, pressing CTRL-C can terminate it. If the process is in the background, use the kill -9 command to terminate it.

For example:

```
$ cat dpwhile1
# Arguments of script should be valid filenames in working directory.
# If valid, the contents of files are displayed one by one.
# If not, an error message is shown.
while [ "$1"!= "" ]
do
   if [ -f $1 ]
        then echo "This is $1."
           cat $1
       else echo "$1 is not a valid file name."
   fi
   shift
done
exit 0
$ dpwhile1 file1 file2 file3 file4 fil
This is file1.
...... The content of the file1 is displayed here.
This is file2.
...... The content of the file2 is displayed here.
This is file3.
...... The content of the file3 is displayed here.
This is file4.
...... The content of the file4 is displayed here.
fil is not a valid file name.
```

```
$
```

This dpwhile1 script takes if or not the positional parameter is null as the condition. If the position parameter $1 is not null, the while statement is executed. In the body of the iteration, if statement makes two branches: one is for the case that the $1 is a filename; the other is for the case that the $1 is not a valid file name. For the case that the $1 is a filename, the script program displays the sentence "This is $1." and the contents of the file. For the case that the $1 is not a filename, the script program displays the sentence "$1 is not a valid file name." Then, move the positional parameters to let $2 be $1, $3 be $2, and so on, by using the shift command. The next iteration begins. And this time, in the evaluation of the condition, the positional argument $2 is evaluated. In this way, in once iteration, a positional parameter is evaluated until there is no more positional parameter, at all. The instruction goes out of the while statement and executes the exit command.

The expression of the while statement can accept a variable from the read command. For example:

```
$ cat dpwhile2
# User's first input should be a valid filename in working directory.
# The user's following inputs until typing in "end" are appended
# to the end of the file with the above filename.
# The whole content of the file is displayed.
echo "Type a filename:  \c"
read fname
echo "You can continue to type several words to form a sentence."
echo "If you want to finish, just enter end."
echo "Type one word:  \c"
read word
while [ "$word" != "end" ]
 do
  echo $word >> $fname
  echo "Continue to type another word:  \c"
  read word
 done
cat $fname
exit 0
$ copy file1 fileforwhile
$ dpwhile2
Type a filename:  fileforwhile
You can continue to type several words to form a sentence.
If you want to finish, just enter end.
Type one word:  UNIX
Continue to type another word:  is
Continue to type another word:  an
Continue to type another word:  interesting
Continue to type another word:  OS.
Continue to type another word:  end
...... The content of the file1 is displayed here.
UNIX
is
an
interesting
OS.
$
```

The dpwhile2 script reads in a filename and stores it into the fname vari-

able. It continues to read in a word and stores it into the word variable. The while statement tests if or not the word is "end". If not, the word is appended to the end of the file with the above filename; the next word is read in and stored into the word variable; the iteration keeps repeating and testing the word. If it is "end", the while statement is terminated and the cat command is executed. The whole content of that file is displayed and the script program is exited.

Before execution of the dpwhile2 script, the cp command creates a new copy, fileforwhile, of the file1 file that exists. When the script is executed, the user types in fileforwhile firstly. Then "UNIX is an interesting OS." are entered before "end" is typed in, too.

10.3 Program Control Flow Statement (e): until Statement

Like for and while statements, until statement is used for repetitive execution. The until statement also makes a block of code execute repeatedly according to the condition of an expression. But there is difference between the while and until statements. For the while statement, the iteration keeps repeating as long as the expression is true; for the until statement, the iteration keeps repeating as long as the expression is false.

The syntax, function and flow diagram of until statement are as follows.

```
until expression
do
  command lines
done
```

Function: to execute command lines as long as expression is logically false.

Flow diagram is shown in Figure 10.3.

Note: The words of until, do, and done are keywords, which must be used in the syntax. If the expression is false, the command lines in between the do and done keywords are executed. In the command lines, there are some lines that are used to assign the new value to the variable in the expression and the expression is evaluated again. One execution of expression evaluation and command lines is one iteration. The iteration is repeated until the expression is true. At that time, the instruction comes out of the until statement and the command following done is executed.

Here, use the until statement to realize the functions of the dpwhile1 and dpwhile2 files in the previous section.

```
$ cat dpuntil1
# Arguments of script should be valid filenames in working directory.
# If valid, the contents of files are displayed one by one.
# If not, an error message is shown.
until [ "$1" = "" ]
do
   if [ -f $1 ]
```

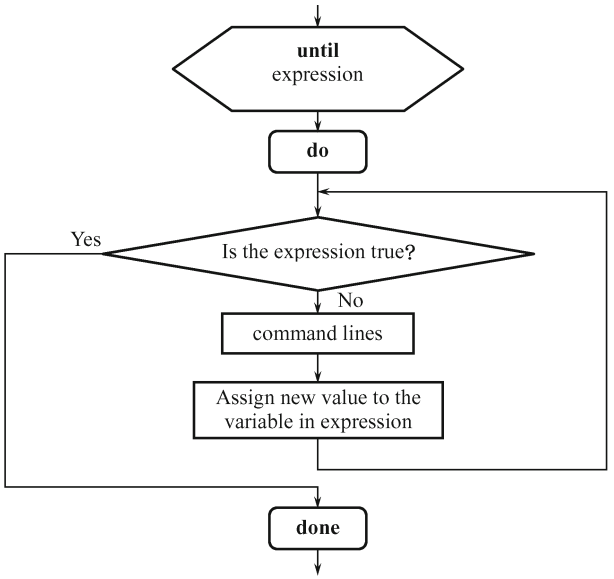

Fig. 10.3 The flow diagram of the until statement.

```
         then echo "This is $1."
            cat $1
         else echo "$1 is not a valid file name."
   fi
   shift
done
exit 0
$ dpuntil1 file1 file2 file3 file4 fil
This is file1.
...... The content of the file1 is displayed here.
This is file2.
...... The content of the file2 is displayed here.
This is file3.
...... The content of the file3 is displayed here.
This is file4.
...... The content of the file4 is displayed here.
fil is not a valid file name.
$ cat dpuntil2
# User's first input should be a valid filename in working directory.
# The user's following inputs until typing in "end" are appended
# to the end of the file with the above filename.
# The whole content of the file is displayed.
echo "Type a filename:  \c"
read fname
echo "You can continue to type several words to form a sentence."
echo "If you want to finish, just enter end."
echo "Type one word:  \c"
read word
until [ "$word" = "end" ]
   do
       echo $word >> $fname
       echo "Continue to type another word:  \c"
       read word
```

```
    done
cat $fname
exit 0
$ copy file1 fileforuntil
$ dpuntil2
Type a filename:  fileforuntil
You can continue to type several words to form a sentence.
If you want to finish, just enter end.
Type one word:  UNIX
Continue to type another word:  is
Continue to type another word:  an
Continue to type another word:  interesting
Continue to type another word:  OS.
Continue to type another word:  end
...... The content of the file1 is displayed here.
UNIX
is
an
interesting
OS.
$
```

Notice that the difference of the dpuntil1 script from the dpwhile1 scripts is that "!=" is replaced by "=" and "while" is replaced by "until". The difference between the dpuntil2 and dpwhile2 scripts is similar.

10.4 Program Control Flow Statement (f): break and continue Commands

The break and continue commands can be used to break or make a "short-cut" in the repeated execution of the iteration when using for, while, or until statements for repetitive execution. In this section, they are discussed, respectively.

10.4.1 The break Command

The break command can be used to break the normally repeated execution of the iteration in for, while, or until statements, terminate the repetitive execution, and execute the command following the keyword done.

The syntax, function, and flow diagram of the break statement are as follows.

```
while expression
do
command lines
......
break [n]
......
command lines
done
```

Function: to terminate the repetitive execution, and transfer control to the command after the keyword done.

Flow diagram is shown in Figure 10.4.

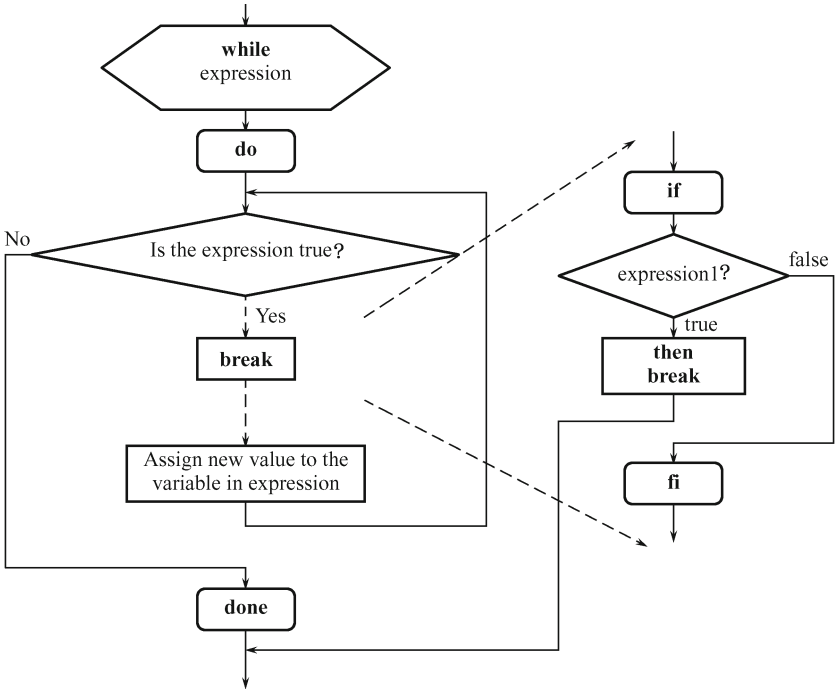

Fig. 10.4 The flow diagram of while statement with the break command.

Note: As the commands in the iteration following the break command are not executed, the break command is usually used as part of a conditional statement such as the if statement. The break command can have an integral argument to represent the number of the nested levels that are broken. The default value of the argument is 1. The command break 3 means three levels of repetitive executions are broken if there are more than three levels of nested repetitive executions.

For example:

```
$ cat dpbreak1
echo "Enter several numbers; to exit the program, type in 5 or null."
echo "Type in a number:  \c"
read num
while [ "$num" != "" ]
do
   if [ "$num" -eq 5 ]
        then break
   fi
   echo "Type in a number:  \c"
   read num
done
```

```
exit 0
$ dpbreak1
Enter several numbers; to exit the program, type in 5 or null.
Type in a number:  4
Type in a number:  3
Type in a number:  2
Type in a number:  5
$
```

In the dpbreak1 script, the while statement can be terminated by reading in null for the num variable or through the break command when reading in 5 for the num variable. Both of these two ways can be used to exit the script program.

10.4.2 The continue Command

The continue command is used to make a "shortcut" in the execution of one iteration in for, while, or until statements, transfer control to the top of the repetitive statement, and make the expression for the condition be tested again and continue the iteration.

The syntax, function, and flow diagram of the continue statement are as follows.

```
while expression
do
command lines
......
continue [n]
......
command lines
done
```

Function: to transfer control to the top of the repetitive statement, and make the expression be tested again and continue the iteration.

Flow diagram is shown in Figure 10.5.

Note: As the break command, the continue command makes the commands in the iteration following the continue command be jumped over and the continue command is also usually used as part of a conditional statement such as the if statement. The continue command can also have an integral argument to represent the number of the nested levels that are jumped over. The default value of the argument is 1. The command continue 3 means three levels of repetitive executions are jumped over if there are more than three levels of nested repetitive executions.

For example:

```
$ cat dpcontinue1
for num in 1 2 3 4 5 6 7
do
 if [ "$num" -eq 5 ]
   then continue
 fi
```

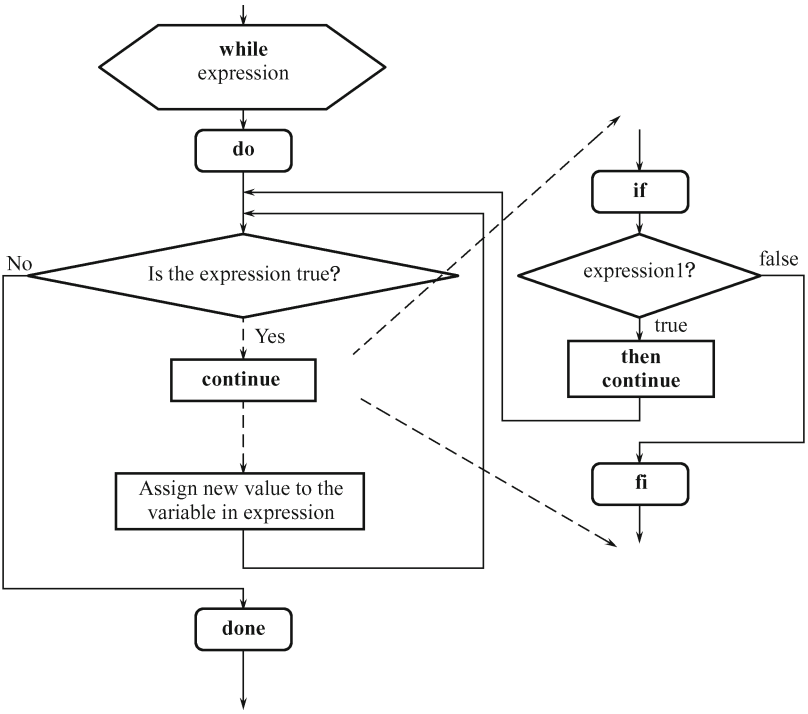

Fig. 10.5 The flow diagram of while statement with the continue command.

```
 echo "$num"
done
exit 0
$ dpcontinue1
1
2
3
4
6
7
$
```

In the dpcontinue1 script, the for statement does repetition according to the number list: 1, 2, 3, 4, 5, 6, and 7. But in the iteration body, the if statement makes a branch that contains a continue command when the value of the num variable becomes 5. The continue command makes a jump back to the expression testing, which makes that the echo command is jumped over and the 5 number is not displayed.

10.5 Processing Numeric Data

Because in Bourne shell, the values of all variables are stored as character strings, numbers are actually stored in the form of character strings, which makes numeric data processing a little bit complex. To perform arithmetic and logic operations on them, it is necessary to convert them into integers, and be sure to convert the operated result back into a character string in order to store it in a shell variable. The expr command can do this job. The Bourne shell provides just the operations on integers.

The syntax and function the expr statement are as follows.

```
$ expr n1 operator n2
```

Function: to perform the arithmetic or comparison operations and send the result to standard output; n1 and n2 can be an integer or a variable.

Common operators:

+, -, *, /, %: Integer arithmetic operators: add, subtract, multiply, integer divide (return quotient), and remainder.

=, !=, \ >, \ >=, \ <, \ <=: Integer comparison operators: equal, not equal, greater than, greater than or equal to, less than, and less than or equal to.

Note: In the expr command, there must be a space in between n1 and operator, and operator and n2. As * is a shell metacharacter and there is a special meaning for it, to use it as the multiply operator, put a backslash (\) before it to escape the special meaning. So do the operators of \ >, \ >=, \ < and \ <=. When the expr command consists of an expression of some comparison operation, if the result of the expression is true, the output of the expr command is integer 1; if it is false, the output is integer 0.

For example:

```
$ expr 1 + 20
21
```

The expr command performs the addition of two integers.

```
$ va1=20
$ expr $va1 + 2
22
```

The expr command accomplishes the addition of a variable and an integer.

```
$ va1=`expr $va1 \* $va1`
$ echo $va1
400
```

The expr command performs the multiplying of two variables.

```
$ va2=`expr $va1 / 20`
$ va3=`expr $va1 % 20`
$ echo "$va2, $va3"
20, 0
$
```

The first expr command performs the integer division on the va1 vari-

able. The second expr command performs the integer remainder on the va1 variable.

The following example uses the expr command in the dpexpr1 script.

```
$ cat dpexpr1
# This script is used to make the sum of some numbers
# and display the result on the screen.
if [ $# = 0 ]
then
echo "You should type in several integers as arguments of this script."
exit 1
fi
total="$#"
count=0
num="$1"
sum=0
while [ $count -lt $total ]
do
   count=`expr $count + 1`
   sum=`expr $sum + $num`
   shift
   num="$1"
done
echo "The sum is $sum."
exit 0
$ dpexpr1
You should type in several numbers as arguments of this script.
$ dpexpr1 2 3 4 5 7 8
The sum is 29.
$
```

The dpexpr1 script can be used to make a sum of a group of integers and display the result on the screen. In this script, the if statement checks if or not the user types in the script name with some integer arguments. If not, the message of "You should type in several integers as arguments of this script." is prompted and the script terminates, just like the result of the first execution shown above. If so, the total variable is assigned the value of $# that is the total number of positional parameters that the user types in. The num variable is assigned the value of $1. The count and sum variables that represent the number and the sum of integers, respectively, are initialized as 0. In the while statement, the sum is added to and the count variable counts the number of integers that has been added to the sum variable. Once the integer stored in $1 is added, the positional parameters are shifted and the $2 is assigned to the num variable. Iteration is finished once. In this way, when all the integers that the user types in are added to the sum variable, the instruction goes out of the while statement. The following echo command is executed, which displays the result of the sum and the script program is terminated. The result of the second execution of the dpexpr1 script is shown above, in which the sum of 2, 3, 4, 5, 7, and 8 is 29.

10.6 The exec Command

In Section 4.4.2 it has been discussed, in the UNIX kernel, how a process is executed. An external command is executed by the kernel in this way: firstly, a Bourne shell creates a child process — a copy of the Bourne shell by fork system call; secondly, the kernel replaces the child process with that external command through execve system call; thirdly, the parent process (the original Bourne shell) waits for the child process to finish and terminate, and finally the parent process resumes, and the shell prompt displays on the screen again. Here, it is assumed that the external command is cp command and Figure 4.6 is copied here as Figure 10.6.

The exec command that is introduced in this section makes an execve system call. Notice that there is no the fork system call to create a child process – a copy of the Bourne shell, in which the command is expected to replace the shell copy and to be executed. Therefore, the exec command is normally used to execute a command by replacing the current shell without creating a child process. In other words, it overwrites the current shell with the code of the executed command.

Except the function to execute a command, the exec command has the use of I/O redirection. In summary, the exec command can be used in two ways:

- To execute a command or program by replacing the current shell where the exec command is running;
- To redirect standard input and output by opening and closing file descriptors.

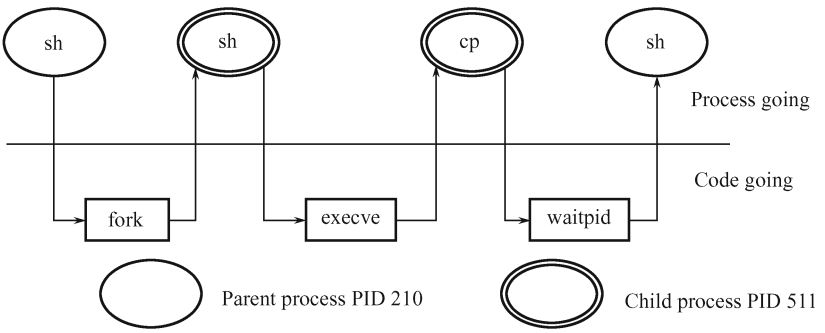

Fig. 10.6 Steps of executing the cp command on the shell.

The above two functions will be discussed in the following two sections, respectively.

10.6.1 Execution Function of the exec Command

To execute a command without creating a child process, use the exec command.

The syntax and function the exec statement are as follows.

```
$ exec command
```

Function: to make the command run in place of the parent process without creating a child process, by overwriting the code on top of the parent process that executes the exec command for the command to execute.

Note: As the exec command executes a command without creating a child process and just instead of the parent process that is usually a shell, once the execution of the command is finished, control transfers to the grandparent process – the parent process of the original shell and cannot go back to the original shell. Therefore, if the original shell is the login shell, control will go back to the getty process (see Section 4.6) when the exec command finishes, for instance:

```
$ exec date
Thurs Sept 04 11:04:56 CST 2008
login:
```

The steps of the execution of the exec date command on the login shell are illustrated in Figure 10.7.

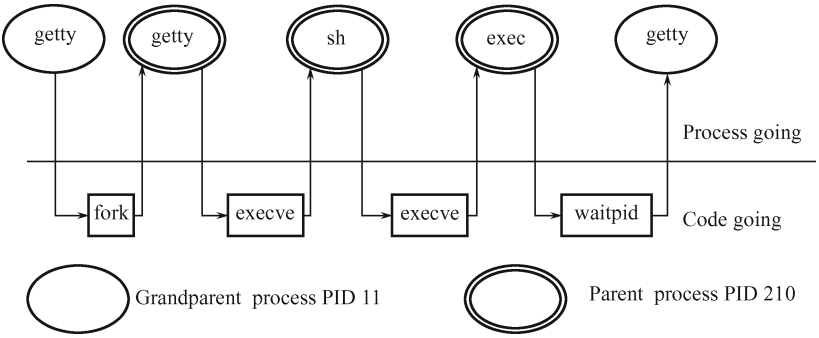

Fig. 10.7 Steps of executing the exec date command on the login shell.

To avoid the above situation happening, the exec date command can be run on a subshell of the logging shell. For example, before the execution of the exec command, type in ksh command on the command line, which means the execution of the Korn shell on the logging shell, say, the Bourne shell. At this situation, when the exec command is finished, control goes back to the login shell – the Bourne shell, not the getty process, as shown in Figure 10.8. To see the difference, use the ps command before and after the execution of the exec command.

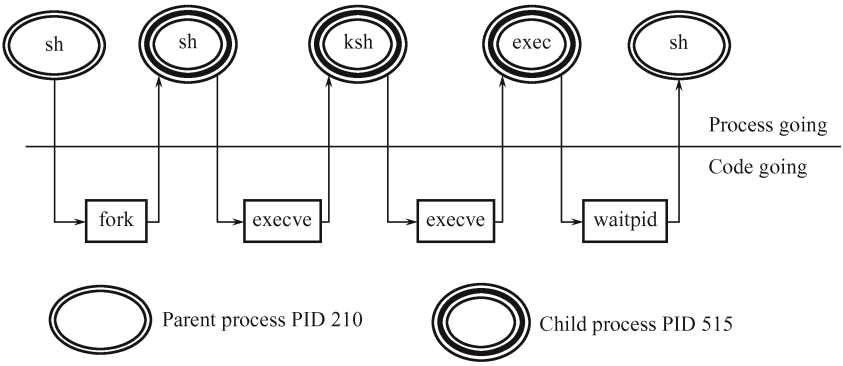

Fig. 10.8 Steps of executing the exec date command on the subshell.

10.6.2 Redirection function of the exec Command

In Chapter 7, the input and output redirections have been discussed. Input redirection means to let a command input direct to a file rather than to standard input — a terminal keyboard while output redirection makes a command output direct to a file rather than to standard output — a terminal screen. The exec command can be used with redirection operators to make commands and shell scripts to read from or write into any type of files, including devices. Here, the common uses of the exec command with different redirection operators are discussed, respectively.

10.6.2.1 Input Redirection of the exec Command

The syntax and function of the exec command with the input redirection operator < are as follows.

```
$ exec < in-file
```

Function: to open in-file for reading and redirect standard input of the process to in-file.

Note: The behavior of the exec < in-file can be slightly different when it is executed on the command line or in a shell script.

When it is executed from the command line, the current shell takes each line in the in-file file as a command and executes it. The reason beneath this case is that the exec command is executed by the shell process, and the primary purpose of the shell is to read, interpret, and execute the commands from stdin. As the in-file is attached to the stdin, the shell reads, interprets, and executes the commands from this file, and finally, the shell terminates itself and goes back to its parent process after executing the last line in file. Therefore, to avoid going back to the getty process, execute another shell

before executing the exec < in-file command.

It is important to know that the exec < /dev/tty command must be executed after having used the exec < in-file in order to reattach stdin to the standard input — the terminal keyboard.

For example:

```
$ cat execfile1
date
pwd
echo "This is a testing file."
$ echo $SHELL
/usr/bin/sh
$ ksh
$ echo $SHELL
/usr/bin/ksh
$ exec < execfile1
Thurs Sept 04 11:04:56 CST 2008
/users/project/wang/work
This is a testing file.
$ exec < /dev/tty
$ echo $SHELL
/usr/bin/sh
$
```

The first cat command shows the content of the execfile1 file that is used as the redirection file for the exec command later on. Each line of the execfile1 file is a command. Three of the echo $SHELL are used to display the current shell, which can tell the change of the current shell. Before the ksh command and after the exec < execfile1 command, the shell is the Bourne shell. Between these two commands, the shell is the Korn shell. Remember that it is necessary to reattach stdin to the standard input – the terminal keyboard after the execution of the exec < execfile1 command.

When it is executed within a shell script, the exec < in-file command makes the stdin of the remainder of the script to be attached to the in-file. It is also necessary to run the exec < /dev/tty command after having used the exec < in-file in order to reattach stdin to the standard input – the terminal keyboard.

10.6.2.2 Output Redirection of the exec Command

The syntax and function of the exec command with the output redirection operator > are as follows.

```
$ exec > out-file
```

Function: to open out-file for writing and redirect standard output of the process to out-file.

Note: The behavior of the exec > out-file can be slightly different when it is executed on the command line or in a shell script.

When the exec > out-file command is executed from the command line, it redirects the outputs of all subsequent commands executed on the shell to the out-file file (normally to the standard output — the terminal screen). Thus, the output of any command does not appear on the screen.

To see the output on the screen again, it is necessary to execute the exec > /dev/tty command. To do so can make it possible to see the contents of the out-put file to check out the outputs of all the commands executed before this command.

For example:

```
$ exec > execfile2
$ date
$ pwd
$ echo "This is a testing file."
$ exec > /dev/tty
$ cat execfile2
Thurs Sept 04 11:04:56 CST 2008
/users/project/wang/work
This is a testing file.
$
```

The exec > execfile2 command makes the outputs of the following commands, date, pwd, and echo commands, go to the execfile2 file. The exec > /dev/tty makes the output of the command come back to the standard output – the screen. Thus, the cat command displays its result that is the contents of the execfile2 file on the screen.

When the exec > out-file command is executed within a shell script, it redirects the outputs of all following commands within the script to the out-file file (normally to the standard output — the terminal screen) until the exec > /dev/tty command is executed later on within the shell script.

10.6.2.3 Appending Redirection of the exec Command

The syntax and function of the exec command with the >> operator are as follows.

```
$ exec >> out-file
```

Function: to open out-file for writing, redirect standard output of the process to out-file, and append standard output to out-file.

For example, continue to use the execfile2 file for this example:

```
$ exec >> execfile2
$ date
$ pwd
$ echo "This is a testing file."
$ exec > /dev/tty
$ cat execfile2
Thurs Sept 04 11:04:56 CST 2008
/users/project/wang/work
This is a testing file.
Thurs Sept 04 11:24:06 CST 2008
/users/project/wang/work
This is a testing file.
$
```

The difference of this execution from the previous execution of the exec > execfile2 command is that the execution results of the following commands are appended to the end of the execfile2 file, rather than overwriting the execfile2 file. Notice, when the redirection is not used any more, don't forget

to use the exec > /dev/tty command to make the output of the command come back to the standard output — the screen.

10.6.2.4 n> Redirection of the exec Command

The syntax and function of the exec command with the n> operator are as follows.

```
$ exec n> out-file
```

Function: to open out-file for writing, and allocate the file descriptor "n" to it.

The error redirection by using file descriptor should be familiar to readers who have read Section 7.5.1. The standard error file and standard output file are both associated with the terminal screen by default. The standard error file, stderr, has the file descriptor 2. Here give an example that uses the exec command with the 2> operator.

```
$ exec > execfile3 2> execfile4
$
```

In this command, the output and error messages from the following commands on the command line are directed to the execfile3 and execfile4 files, respectively. Remember that the exec > /dev/tty command makes the output of the command go back to the standard output – the screen. Similarly, the exec 2> /dev/tty command makes the error message of the command sent to the standard output – the screen, which must be done if the redirection of the error message is no use any more.

10.6.2.5 n>> Redirection of the exec Command

The syntax and function of the exec command with the n>> operator are as follows.

```
$ exec n>> out-file
```

Function: to open out-file for writing, allocate it the file descriptor "n", and append the new data to the end of the file.

This command is similar to the exec >> out-file, but this can be redirected the output of the following commands to a file with a file descriptor n, which can be not only 1 and 2, but also other logical numbers.

10.6.2.6 Some More Redirection of the exec Command

The syntax and function of the exec command with the n< operator are as follows.

```
$ exec n< in-file
```

Function: to open in-file for reading, assign it the file descriptor "n".

This command is similar to the exec < in-file, but this can be redirected the input of the subsequent commands to a file with a file descriptor n, which can be not only 0, but also other logical numbers.

The exec command can also be used with the redirection to a file that is allocated with a file descriptor n rather than 0, 1, and 2. At the end of this section, an integral example will be given, which is a little bit complex and shows this use combined with other facilities.

The syntax and function of the exec command with the n>&m operator are as follows.

```
$ exec n>&m
```

Function: to make the file with the file descriptor n a duplicate of the file with the file descriptor m.

Note: This command can make the two file descriptors m and n direct the same real file. For users, it seems like that one is the copy of the other.

Mentioned at the beginning of Section 10.6, the exec command can be used in two ways, and the second function includes opening and closing file descriptors. The above is mostly about opening file descriptors. Now, how to close file descriptors will be discussed.

10.6.2.7 Closing File Descriptor Function of the exec Command

The syntax and function of the exec commands with the <&-, >&-, n<&-, and n>&- operators are as follows.

```
$ exec <&-
```

Function: to close standard input.

```
$ exec >&-
```

Function: to close standard output.

```
$ exec n<&-
```

Function: to close the file with the descriptor n attached to stdin.

```
$ exec n>&-
```

Function: to close the file with the descriptor n attached to stdout.

The following script can be used to compare two files. If some difference is found in two files, the script is terminated with some message. Before the expression of the dpexec1 script, three tested files should be given, which are as follows:

```
$ cat ftext1
This is a tested file.
It is different from the ftext2.
It is different from the ftext3.
$ cat ftext2
This is a tested file.
It is different from the ftext1.
$ cat ftext3
This is a tested file.
It is different from the ftext1.
$
```

The following is the dpexec1 script that can be used to compare two files

on the Bourne shell. There are some detailed explanations in it, which can tell how the script works. The exec commands in this script are used to open and close files. The exec 3< "filename3" and exec 4< "filename4" commands open the files for reading and allocate them file descriptors 3 and 4. From this point on, to read the two files, the script uses these descriptors. The read line3 0<&3 and read line4 0<&4 commands read the next lines from the files with fie descriptors 3 and 4, respectively. The exec 3<&- and exec 4<&- commands close the two files, respectively.

```
$ cat dpexec1
# The first part of script is used to check out if or not the user
# types in the script program in a correct way.
# The correct way is:  dpexec1 file1 file2
if [ $# != 2 ]
  then
 echo "This command needs two filename arguments."
 exit 1
fi
if [ -f "$1" ]
then
 if [ -f "$2" ]
  then
   echo "Wait a second."
    else
   echo "The argument $2 must be a filename."
   exit 1
 fi
else
 echo "The argument $1 must be a filename."
 exit 1
fi
# The second part is used to assign the values of
# positional parameters of $1 and $2 to two local variables,
# filename3 and filename4, respectively.
filename3="$1"
filename4="$2"
# The third part is to open the files for reading and
# allocate them file descriptor 3 and 4, and assign 0 to mark.
exec 3< "$filename3"
exec 4< "$filename4"
mark=0
# The forth part is to read one line after another
# from each of both files and to compare them.
# The inputs of the read commands in the while and if statements
# are redirected to files with file descriptors of 3 and 4,
# respectively, by using 0<&3 and 0<&4.
# If both reach EOF at the same time, files are the same.
# Otherwise, different.
while read line3 0<&3
  do
  if read line4 0<&4
  then
    if [ "$line3" = "$line4" ]
      then
        continue
    else  # If different lines exist, two files are not the same.
       mark=1
       break
    fi
   else  # If EOF of file2 is first reached, file1 is bigger than file2.
```

```
    mark=2
    break
fi
 done
# The fifth part is to continue to read in line4
# from the file with the file descriptors of 4, by using 0<&4.
# If EOF of file1 is first reached, file2 is bigger than file1.
# Otherwise, two files are the same.
if [ "$mark" -eq 0 ]
then
 if read line4 0<&4
      then
  echo "$2 is bigger than $1."
 else
  echo "$1 and $2 are the same."
 fi
else
    if [ "$mark" -eq 1 ]
  then
      echo "There are different lines in $1 and $2."
      echo " $1:  $line3"
      echo " $2:  $line4"
   else
      echo "$1 is bigger than $2."
    fi
fi
# The sixth part is to close files with file descriptors 3 and 4.
exec 3<&-
exec 4<&-
exit 0
$ dpexec1
This command needs two file arguments.
$ dpexec1 ftext1 ftext2
Wait a second.
There are different lines in ftext1 and ftext2.
ftext1:  It is different from the ftext2.
ftext2:  It is different from the ftext1.
$ dpexec1 ftext2 ftext3
Wait a second.
ftext2 and ftext3 are the same.
$
```

10.7 Bourne Shell Functions

From the programming experience in high-level languages, it is known that functions and routines can save programmers a lot of time to type the same block of code. In Bourne shell, a shell script can call another shell script to perform some regular tasks.

In Chapter 9, Figure 9.1 shows that Bourne shell allows a script to call other scripts. As the script file is on the disk, to call the script requires loading the script file from the disk into the main memory, which costs a lot of time.

Compared to a script, a Bourne shell function is in the main memory (Bourne 1978; Sarwar et al 2006). Thus, to call a function is faster than to call a script.

In Bourne shell, as shell scripts, functions have their own name and are

composed of a series of commands, called the function body. They can be invoked by using the function names.

A function is normally used to form a block of code that is called at various places in a script. The benefit of a function is to write the code just once, but to be used several times (Bourne 1978; Bourne 1983; Joy 1980; Korn 1983; Rosenblatt et al 2002). Thus if a block of code is used at many different places in a script, it is effective to create a function for it and call it where it is to be inserted by using the function name.

10.7.1 Defining Function

Before using them, the functions need to be defined.

The syntax of function definition is as follows:

```
function_name ()
{
command lines
}
```

The function_name is the name for the function that the programmer can choose. The function name must be followed by the parentheses (()). The function name and the opening brace ({) can be written on the same line. The command lines compose the function body.

There are three ways to define Bourne shell functions. Here lists them:

- To define functions in the ∼/.profile file (see Chapter 2). This method is used for the general-purpose functions. In this way, the shell brings them in its environment when the users log in and allows the users to call them while they use the system.
- To define functions on the command line. These functions are valid before logging out.
- To define functions in a shell script. These functions are specific to the script.

Functions can be made available to all the subshells of the shell that contains these functions by using the export command. Remember that any executed command, except the exec command, firstly, is forked a copy of the current shell, and then replaces the copy to run.

To define a function on a command line, type the function name and parentheses after the shell prompt, followed by a { on a line, one command per line, and a } on the last line. For example:

```
$ funct1 ()
> {
> date
> echo "This is a testing function."
> }
$
```

Notice, the > is the secondary prompt (see Table 8.2).

10.7.2 Calling Function

The commands in a function body are not executed until the function is called. To call a function, we can type its name without the parentheses (()) as a command. If a function is called on the command line, the commands in the function body are executed one by one. And then the control goes back to the shell after the function accomplishes its execution. When a function is called in a script, the commands in the function body are also executed one by one. And when the function is finished, the control returns to the command following the function in the script.

A function can also return to its caller by using the return command.

The syntax and function of the return command in Bourne shell are as follows.

```
return [n]
```

Function: to return from the function to its caller with an exit status number n.

Note: Like the exit command, the return command has only one argument that is optional and an integer number. The argument is returned to the calling process as the exit status of the called process. The exit status value of a function is stored in the $? environment variable that can be checked by the calling process. Without the argument n, the exit status value of a function is the exit status value of the final command in the function.

The variables of a function can be passed to the function by the script in which the function is. The positional parameters, $1 through $9, for a function are equivalent to the positional parameters for a script.

For example:

```
$ cat dpfunct2
# funct2 is a function name, which does multiplying operation and
# echoes the values of variables and arguments.
funct2()
{
z= `expr $x \* $y`
echo "This is a testing function."
echo $x $y $z
echo $1 $2 $3
echo "This is function end."
}
x=5
y=10
funct2 AAA BBB CCC
set `date`
echo " Today is $1, $2 $3, $6."
exit 0
$ dpfunct2
This is a testing function.
5 10 50
AAA BBB CCC
This is function end.
Today is Thurs, Sept 04, 2008.
$
```

In the dpfunct2 script, there is a function called funct2. The variables, x and y, are passed to the funct2 function. And when the script calls the function, the function name is with the values of three positional parameters, AAA, BBB, and CCC. When the function is executed, the z variable is evaluated first, the values of the three variables, x, y, and z, are displayed, and then the values of positional parameters, AAA, BBB, and CCC, are displayed. After returning to the caller, the set command and the following commands are executed.

10.8 How to Debug Shell Scripts

When writing a long and complex script, it is natural to make some mistakes, and it is difficult to look for some mistakes in a long and complex script that cannot function properly. The Bourne shell provides debugging tools to solve this problem. The sh command with some options can be used to do so.

The syntax and function of the sh command for debugging scripts in the Bourne shell are as follows.

```
$ sh [option] scriptname
```

Function: to debug a shell script with the name of scriptname.

Common options:

-n: to read and check the syntax error of the commands in the script, but not to execute it;

-v: (verbose) to display each line of the script as it is written in the script file and its execution result;

-x: (echo) to display each line of the script with its arguments after variable substitution and its execution result.

To explain the functions of the three options: -n, -v, and -x, give one example for each of them as follows.

```
$ cat -n dpdebug1
1 # dpdebug1 is a script, which prompts the user for a capital letter.
2 # If user enters a letter between A and Z, it displays
     a message.
3 # Write an error intentionally on the "then" line.
4 echo "Enter a capital letter (A -- Z): \c"
5 read in1
6 if [ "$in1" = [A-Z] ]
7 # then # This line is written in this way intentionally.
8 echo "Your entering is correct!"
9 fi
10 exit 0
$ sh -n dpdebug1
dpdebug1:  syntax error at line 9 ‘fi’ unexpected
$
```

The cat command with -n option displays each line with its line number. At line 7, then is commented out intentionally, which results in the syntax error. The sh command with -n option shows the syntax error. Correct it by

deleting the # sign, and then the dpdebug1 script can work properly.

```
$ cat dpdebug2
echo "This is the $0 script."
echo "The total number of arguments of this script is $#."
echo "The first argument is $1."
exit 0
$ sh -v dpdebug2 this is testing
echo "This is the $0 script."
This is the dpdebug2 script.
echo "The total number of arguments of this script is $#."
The total number of arguments of this script is 3.
echo "The first argument is $1."
The first argument is this.
exit 0
$
```

In the result of the sh command with -v option, there are two lines for each line in the script: one line displays a line that is written in the script; the next line is for the execution result of the same line in the script.

```
$ sh -x dpdebug2 this is testing
+ echo This is the dpdebug2 script.
This is the dpdebug2 script.
+ echo The total number of arguments of this script is 3.
The total number of arguments of this script is 3.
+ echo The first argument is this.
The first argument is this.
+ exit 0
$
```

In the result of the sh command with -x option, there are also two lines for each line in the script: one line with the + sign displays a line that is in the script but after variable substitute; the next line is for the execution result of the same line in the script.

The -x option can be used together with the -v option in one sh command, such as sh -xv dpdebug2. When the -x or -v option is used to check the syntax error in a script and the script has a syntax error, the script program goes to the point where the syntax error exists, the syntax error is displayed, and the program terminates.

Another method to debug a script is to use the set command with -x and -v options. Just put the set command with these two options at the beginning of a script or at the point in a script, where to start to be debugged.

The syntax and function of the set command for debugging scripts in the Bourne shell are as follows.

```
set -x[v]
```

Function: to set the beginning point for debugging in a shell script.

Common options:

-v: (verbose) to display each line of the script as it is written in the script file and its execution result;

-x: (echo) to display each line of the script with its arguments after variable substitution and its execution result.

Note: The options -v and -x in the set command have the same functions as they do in the sh command.

10.9 Summary

In this chapter, the topic of programming with Bourne shell is progressing into a more advanced stage than in Chapter 9. In Bourne shell, there are two statements for branching, if and case. The case statement provides a multi-way branching method similar to an integral if statement with several elif blocks. For repetitive execution, there are for, while, and until statements. The while and until statements make a block of code execute repeatedly according to the condition of an expression, but there is difference between the while and until statements. For the while statement, the iteration keeps repeating as long as the expression is true; for the until statement, the iteration continues repeating as long as the expression is false. The break and continue commands can be used to break or make a "shortcut" in the repeated execution of the iteration when using for, while or until statements for repetitive execution. As the commands in the iteration following the break and continue commands are not executed, the break and continue commands are usually used as part of a conditional statement such as if statement.

Because in Bourne shell, the values of all variables are stored as character strings, numbers are actually stored in the form of character strings. To perform arithmetic and logic operations on them, it is necessary to convert them into integers, and be sure to convert the operated result back into a character string in order to store it in a shell variable by using the expr command. Bourne shell provides the operations only on integers. The exec command can be used in two ways: to execute a command or program by replacing the current shell in which the exec command is running; to redirect standard input and output by opening and closing file descriptors. Functions are normally used to form a block of code — function body — that can be called at various places in a script. If a block of code is used at many different places in a script, it is effective to create a function and call it anywhere it is inserted by using the function name.

For writing a long and complex script, the Bourne shell provides debugging tools to check the script syntax errors. The sh command and the set command with the -x and -v options can be used to check syntax errors in shell scripts. The former can be used on the command line; the latter can be inserted in the checked scripts. When the scripts pass debugging, remove the set command from the scripts.

Problems

Problem 10.1 In the case statement, what is the default case for? Where should the default case be put in the case statement? Why?

Problem 10.2 Write a script with the case statement, which functions like a command mini-menu. Try to find some simple UNIX commands as the items for the command mini-menu.

Problem 10.3 For the repetitive statements, what is the iteration? What can cause infinite loops?

Problem 10.4 What is the difference between the while and until statements when they are executed? Try to write two scripts with the while and until statements, respectively. The scripts accept the words that the user types from the keyboard, store the words in a file, and finally, display the contents of the file.

Problem 10.5 When they are executed, what is the difference between the break and continue commands? Write two scripts with the break and continue commands, respectively. One is for checking if or not the user types in a q letter; if it is, exit the script. The other is for checking if or not a g letter in the tested list; if a g is encountered, ignore it and continue checking until all the items in the list are checked.

Problem 10.6 In the Bourne shell, what data form are the values of all variables stored as? What is the expr command used for?

Problem 10.7 Write a script that can accept a group of numbers as its positional parameters, square each of the numbers, sum all the squared values, and display the arithmetic result.

Problem 10.8 Why does the execution of the exec command on the login shell make the user log out? How can you do to solve this problem?

Problem 10.9 Write the following shell script by using a text editor, and execute it to see what happens. Write down the lines shown on the terminal screen and explain them.

```
$ cat dpexec0
cat > execf1
exec < execf2
cat > execf3
exec < /dev/tty
cat > execf4
$ chmod 577 dpexec0
$ dpexec0
```

Problem 10.10 What are the two kinds of functions of the exec command?

Problem 10.11 If you use the exec > exfile1 command to redirect the output of the subsequent commands to the exfile1 file firstly, when the redirection is no use any more, how can you do to make the output go back to the terminal screen?

Problem 10.12 Write your vision of the dpexec1 script in Section 10.6.2 to compare two files.

Problem 10.13 Considered their locations on the computer system, what

is the difference between a script and a function? Comparing a script and a function, which is faster in execution?

Problem 10.14 Before using them, the functions need to be defined. How many ways can be used to define a function? What are they?

Problem 10.15 Write a script including functions, which can do arithmetic operations of two integers, including add, subtract, multiply, and integer divide, respectively. Use the sh command with the -x and -v options to debug the script. Write another script including a function, which can do sort a group of integers. Use the set command with the -x and -v options to debug the second script.

References

Bourne SR (1978) The UNIX shell. Bell Syst Tech J 57(6) Part 2, pp 1971 – 1990

Bourne SR (1983) The UNIX system. Addison-Wesley, Reading, Massachusetts

Joy WN (1980) An introduction to the C shell. UNIX Programmer's Manual, 4.2 Berkeley Software Distribution. Computer Systems Research Group, Depat. of Electrical Engineering and Computer Science, Univ. of California, Berkeley, Aug, 1980

Korn D (1983) KSH - a shell programming language. USENIX Conference Proceedings, Toronto, Ontario, June 1983, pp 191 – 202

Rosenblatt B, Robbins A (2002) Learning the Korn shell, 2nd edn. O'Reilly & Associates, Sebastopol.

Sarwar SM, Koretesky R, Sarwar SA (2006) UNIX: the textbook, 2nd edn. China Machine Press, Beijig

11 UNIX in Internet and Computer Networking

In previous chapters, the local applications and services of UNIX in individual computers have been introduced. This chapter will present the remote and network functions and services of UNIX in servers and workstations. After explanation of general concepts about the Internet and computer networking, Transmission Control Protocol/Internet Protocol (TCP/IP) suit will be presented, and contents associated with layers of TCP/IP model from down to up will be discussed. Finally, we will explain how to use some commands in the application layer of the TCP/IP model, such as telnet, ping, ftp, etc.

11.1 UNIX's Contribution to Development of Computer Networking

In the reality, the UNIX operating system is widely applied in servers and workstations, and have many original contributions to the development history of the computer networking and Internet. With the background information of computer networking, readers should have known that at the very beginning, the development of computer networking was funded by Advanced Research Projects Agency (ARPA, also named Defense Advanced Research Projects Agency, DARPA) in 1960s. This project resulted in the ARPANet, in which the UNIX time-sharing system allegedly performed well as an ARPA network mini-host according to RFC 681 (Request for Comments, which will be introduced in the following section) (Holmgren 1975). Within the document of RFC 681, it is detailed explained why UNIX was chosen as the operating system of an ARPANet mini-host. One reason is that UNIX has many beneficial features, among which are the network control program (NCP) integrated in the UNIX kernel and network connections accessible through UNIX I/O standard system calls. In the ARPANet, the first connected nodes included UCLA, Stanford University, University of California at Santa Barbara, and the University of Utah. In 1980s, BSD added the TCP/IP network function to the UNIX kernel, which made the Internet infrastructure expand

TCP/IP networks into not only the U.S. military and academic institutions but also commercial services.

Since most of the networking protocols were initially implemented on UNIX and most of the Internet services are provided by server processes running on the UNIX operating system, UNIX has a fundamental and profound influence on computer networking.

11.2 General Concepts of Computer Networks and Internet

As Computer Networking is also one of the most important disciplines in computer science and technology and there are many published books specialized to the Computer Networking topics, this chapter will present some general concepts of the computer networks and Internet, which are close to or involved with operating systems, and the detailed information about Computer Networking should be referred to the special books listed at the end of this chapter (Comer 1998; Comer et al 1998; Stevens 2002; Wright et al 2002). And having the knowledge of Computer Networking will be helpful for readers to understand this chapter.

11.2.1 Request for Comments

In the late 1960s, the first Request for Comments (RFC) was used by authors in the ARPA-funded projects to take notes of their research findings and circulated their notes with other ARPA researchers. In modern computer network engineering, the RFCs are memos published by the Internet Engineering Task Force (IETF) and the Internet Architecture Board (IAB) in order to describe research findings and achievements that can be applied to the computer network systems and Internet. Today, RFCs are the official publications that can help exchange information among the global computer network researchers.

As their main goals were requests for comments, the early RFCs were different from the modern RFCs, written not necessarily in a formal way and often left questions open for other researchers' extension. Nowadays, this type of writing style can be used for Internet Draft documents before being approved and revised as an RFC. The RFCs that are adopted by the IETF from proposals can be published as Internet standards while the RFCs can also be used for computer scientists and engineers in the Internet Society to publish their latest research achievements. Readers can refer to the RFCs through the Internet. Some of them are listed in the references of this chapter.

Another benefit of RFCs is that they are standards directly from the

experience and practices of computer scientists and engineers who have been doing researches on computer networking by themselves. For this reason, these standards are operational and pragmatic. It is also the reason why compared to the ISO model, the TCP/IP model is viable and popular in the computer world.

11.2.2 Computer Networks and Internet

Before the advent of computer networks, communication between computers and even between the computing system and I/O systems was relied on human beings moving storage media (usually tapes) between them. Today, computer networks are the core of modern communication. The following advantages that computer networks have brought to the modern world may be so familiar for us to take them for granted.

- To share computer resources among different users. Users in a computer network can share printers, disks, files, and even CPUs.
- To make cooperation of researchers in different locations to develop one big project. Researchers with different talents in different cities can work together for one project.
- To enhance the reliability of a whole computer system. If one computer in a network is broken, other computers in the same network can undertake its work.

As a reader, you can count more items of the advantages. And even more, just to image the situation if without the computer networks and Internet, you will realize how consequential their seeping effect on our real activities is.

As known, computer networking supports the communication between computer systems in or between computer networks. Computer networks are formed by connecting two or more computer hardware resources which can be computers, printers, scanners, screens, etc. If given an intuitive definition, a computer network is a group of computers or devices connected to each other in order to exchange data, and computer networking is to connect different computers with devices and media in order to make them communicate each other. In computer networking, a host can be a computer holding information resources as well as application software for providing network services; and routers or gateways are dedicated computers that are responsible to connect other computers or networks. According to their scales, there are three types of computer networks: LAN, MAN, and WAN:

- Local area network (LAN), a small network is located in a small geographic area, such as a building or a campus, and provides services to a small group of people who belong to one community. LANs can adopt a peer-to-peer or client-server networking model, which will be discussed in the following section.

- Metropolitan area network (MAN), a medium-sized network usually covers a city. When many LANs over a specific geographical area (for example a city) are integrated into one larger network, a MAN for that area is built up. Over a MAN, the flow volume of communications increases with the network growth.
- Wide area network (WAN), a large network often crosses different cities, which is a network composed of numerous smaller networks with a wide variety of resources. The Internet is the giant WAN. Through the Internet, a multi-national corporation can build up a WAN to interconnect their branch offices in different countries. Also, a wide variety of technologies involve in the communications in WANs, for example, Asynchronous Transfer Mode (ATM), Frame Relay, Point-to-Point Protocol (PPP), and Synchronous Optical Network (SONET). These technologies are distinguished by their switching capabilities and data transmission speeds.

As the technologies in telecommunications, computer science and computer engineering developing, the communication media in the computer networking are not only a variety of tangible wires (such as coaxial cable, twisted-pair copper wire cable, power lines, and optical fiber) but also wireless technologies. Therefore, the Wireless LANs and WANs are alternatives for LANs and WANs, which make computer networking mobile. A wireless LAN or WAN is almost the same as a LAN or WAN, except without wires between computers or devices. The data are transmitted via radio transceivers. Communications within large areas can be performed through communications satellites, cellular network, wireless local area network (WLAN, such as Wi-Fi) or Wireless local loop. According to the capabilities of antennas, Wireless LANs or WANs can cover areas ranged from hundreds of meters to a few kilometers. And Bluetooth makes the devices within a few meters communicate with each other wirelessly. Wireless technologies can save the cost and inconvenience of laying cables.

According to the geographic reference to a network, networks can also be divided into three types of networks: Intranet, Extranet, and Internet.

- Intranet, a network is usually for trust communications inside an organization, university, or enterprise. It is only accessible by authorized users, such as the faculty of the university or employees of the enterprise. For security, Intranets are generally limited to connect to the Internet.
- Extranet, an outside extension of an intranet allows secure communications to users outside an organization, university, or enterprise, such as the visitors of the university or the customers of the enterprise. With the Virtual Private Network (VPN) technology, Intranets and Extranets can be securely joined the Internet and escape the access of general Internet users.
- Internet, a generalized inter-network that is a ubiquitous network of networks. In other words, The Internet is a platform where a set of Internet Service providers and consumers are interconnected together and provide them the Internet access. The Internet usually includes end users, enter-

prises, businesses, and organizations interconnected by Internet Service Providers (ISP, also referred as Internet Access Providers, IAP), which are companies that offers its customers access to the Internet. The Internet Services include email accounts, remote data files storage and transmission, on-line chatting, commerce transactions, etc.

Along with electronic commerce development, commerce transactions can be done over the Internet via the communication security mechanism. The typical communication styles include business-to-business (B2B), business-to-consumer (B2C), and consumer-to-consumer (C2C). Along with the social needs' evolving, communication types extend to consumer-to-business (C2B), government-to-business (G2B), business-to-government (B2G), etc. Here give the descriptions of business-to-business, business-to-consumer and consumer-to-consumer.

- Business-to-business (B2B), the communication for electronic commerce transactions between businesses or enterprises. It can be between a manufacturer and a wholesaler, or between a wholesaler and a retailer.
- Business-to-consumer (B2C), the communication for the electronic commerce activities of businesses' serving end consumers with products or services. In a supply chain, there can be several B2B transactions (typically one from a manufacturer to a wholesaler, and one from a wholesaler to a retailer) but only one B2C transaction (one from a retailer to an end customer).
- Consumer-to-consumer (C2C), the communication for the electronic commerce transactions between consumers through the third party. The popular online auction is a C2C example. eBay is an example of the third party for C2C.

These communication models need the support of the communication security mechanism over the Internet and the services of networking operating systems. As UNIX usually works in servers, these issues have been considered in its development. And in the following sections, the services will be discussed gradually.

11.2.3 Client-server vs Peer-to-peer Models

In the computer networking, there are two types of networking models, the client-server and peer-to-peer models, both of which are based-on distributed computing.

11.2.3.1 Client-server Model

In a network with the client-server model, every client (usually a computer) is connected to the server (usually a high-performance host) and also each other. The resources in a server can be shared with clients, but a client does not share any of its resources with servers and other clients. A server

computer executes one or more server programs waiting for clients' requests. A client initiates communication with a server for some service request. For one instance of a client program, a client can send a data request to one or more servers connected to it. In turn, the corresponding servers can receive the request, process it, and return the requested data to the client. A web browser is a typical client program through which a user can access a web server on the Internet. For instance, if a student wants to access remotely the library database server in a university to enquire some author's articles, he may first access the web server of the university library through a web browser from his computer and send a request to the web server. The server program may send the request to its own database client program that in turn forwards a request to a database server in the campus to retrieve the information of the articles. The information is then sent back to the database client, which in turn returned it to the web browser client displaying the enquired results on the screen of the student.

In addition to the web browser, the client-server model has a wide variety of applications. The main application protocols of the Internet, such as Hypertext Transfer Protocol (HTTP), Simple Mail Transfer Protocol (SMTP), File Transfer Protocol (FTP), Telnet (Telnet remote Protocol), and DNS (Domain Name System) (protocols will be discussed in the later sections), adopt this model. With these protocols, functions such as web access, email transfer, and database access, can be built. The corresponding servers include web, mail, ftp, name, file, database, application, print, and terminal servers; and the clients include web browsers, email clients, chat clients, and so on. Some servers can undertake just one task like a web server and play the role of single-service servers while other servers may perform the integration of more than one task, such as the combination of ftp server and file server, or the composition of the computing function and database service, dependent on the capacity of the hardware and the correlation between the tasks. However, as a user of a computer network or the Internet, a connected computer can be installed with different client programs and initiate each of them when needed. Although the client-server model can be used in a variety of applications, the architecture is basically the same.

The client-server model has its advantages and disadvantages. The advantages are:

- Its architecture is maintainable when the volume of servers and clients change. As the roles of clients and servers are concise, their responsibilities are also clear. Therefore, it is easy to replace, repair, and add a new service to it without a high impact on the main part of the network.
- The security cost is low. As all data storage is concentrated in the servers, the security can be controlled more easily than the data distributed in different computers.
- The system update is convenient. Also as all data storage is centralized in the servers and the roles in the model are definite, the data update can be maintained just in the servers and the function update can be done in

the clients or servers relied on the division of responsibilities.

The disadvantages of the client-server model are:

- The bottleneck for traffic on the client-server model can be caused at the server when the volume of client requests increases rapidly. If a large number of client requests are initiated at the same time, the server can be so overloaded that the response from the server may be sluggish and even fail.
- The reliability of the client-server model is lower than the peer-to-peer model. If the server without the backup system is disrupted, the service can be broken off and even all the data can be lost.
- The scalability of the client-server model is less than the peer-to-peer model. As the capability of a central server is limited, the extension of the whole network can be restrained.

11.2.3.2 Peer-to-peer Model

In a network with the peer-to-peer (abbreviated to P2P) model, the resources of each computer peer can be shared with other computer peers. The shared resources can be files, disk storage, network bandwidth, and CPUs. In the peer-to-peer architecture, there is not a host that works as the central coordination server. On the contrary to the client-server model where servers provide resources or services of which clients make use, the status of each computer peer in the peer-to-peer model is equal to others' and can not only provide its resources to others but also make use of the resources of others.

File sharing has made the peer-to-peer model popular and is also criticized drastically for the copyright protection. In file sharing systems (such as Napster, Freenet, and Gnutella), users can connect their computers to a peer-to-peer network with some software and search for shared files on other computer peers that are in the network as well. Files of interest can then be transferred directly from other computer peers. To speed up the transference, large files can be decomposed into smaller parts, obtained from different peers simultaneously, and then reassembled after transference. At the same time, one peer can also upload the parts it has received to other peers.

Beside the file sharing, there are many other applications of a peer-to-peer model. A peer-to-peer network can be used for indexing and resource discovery. For example, since resources in a network are manifold, a peer-to-peer model can be applied to fulfill the efficient resource discovery for grid computing systems, which usually undertake the responsibility to manage resources and schedule applications. Resource discovery involves searching for the suitable resource types to match the application demands of the user.

A network with the peer-to-peer model is usually implemented in the application layer of TCP/IP and runs over the underlying Internet Protocol (IP) network (the later sections will give detailed discussion on TCP/IP and IP).

The peer-to-peer model and client-server model can be merged into one

network, which is usually called a hybrid peer-to-peer architecture. There are two types of hybrid peer-to-peer networks: one divides their clients into two groups – client nodes and overlay nodes – in which overlay nodes are used to coordinate the peer-to-peer structure; the other has central servers to index the peers and direct the traffic among the peers.

Compared to the client-server networks, peer-to-peer networks have their advantages and disadvantages, too. The advantages are:

- The reliability of the peer-to-peer model is higher than the client-server model. As peers in a peer-to-peer model have equivalent status and responsibility, if one peer shuts down, the rest of the peers are still capable to communicate each other.
- The requests on the peer-to-peer model are settled in balance. As it can distribute data among peers and there is not a central server in it, the bottleneck for traffic can be prevented in the peer-to-peer model.
- The resources of the peers in a peer-to-peer model can be utilized effectively to enhance the performance of the whole network. As the peers in a peer-to-peer model take both roles of providers and consumers of resources, each of them can contribute their parts to a cooperating execution.
- The peer-to-peer model can be scalable. As a peer-to-peer model is usually formed dynamically, the addition or removal of peers brings no significant influence to the whole network.

The disadvantages of the peer-to-peer networks are:

- The security of the peer-to-peer model is lower than a client-server model. As it is easy and convenient for a user to engage in the peers of a peer-to-peer model, the insecurity can also be brought easily into the network.
- The existence time of a peer-to-peer network is largely dependent on the peer loyalty and engagement. Since a user can join to a peer-to-peer network as their wish, peers may give up the network when it loses the attraction.
- The reliance on the common interest among the peers can restrain the applications of a peer-to-peer model. Because the effectiveness and efficiency of the peer-to-peer model come from the peers' cooperation, the search for some rare topic may fail in a peer-to-peer model.

Both client-server and peer-to-peer architectures are widely used today. In a workgroup on the same LAN, users can share printers and database servers via the client-server model. With common interests, users located arbitrarily can share a set of servers over the Internet through the client-server model and also share resources via the peer-to-peer software.

11.2.4 TCP/IP and ISO models

In computer networking, networks can be treated in a physical or logical

perspective. In a physical perspective, just like the above discussion, it involves geographic locations, physical connection, and the network elements that are connected via varied media. As the technologies in the computer, telecommunication, and other relevant fields have developed so quickly in these decades, the technologies involved in the computer networking and Internet are diversified and complicated. However, no matter how complex and variant the technologies become, there are always some basic logical principles to support them. And logical networks in the TCP/IP and Open Systems Interconnection (OSI) models can map onto one or more physical media. In this section, it will be introduced how to view the computer networking in a logical way.

When considering the layout of networks, the below protocols (or rules) and facilities should be followed and established:

- The protocols are to implement the detail of software for the particular applications such as telnet, ftp, etc.
- The protocols are to fulfill the application data transportation between two processes over a network.
- The protocols are to carry out routing the application data between hosts over a network.
- The protocols are to execute access to the physical transmission medium for a host's data transmission on a network.
- The type of network topology, such as bus, ring, star, tree, or mesh, is adopted as the infrastructure layout of a network.
- The type of physical communication medium, such as coaxial cable, twisted-pair copper wire cable, power lines, optical fiber, or wireless technologies, is chosen as the connection between the resources over a network.

There are two popular models that are often referred in the computer networking; they are the Open Systems Interconnection (OSI) Reference Model and the TCP/IP Model (or Internet Protocol Model). The OSI model was recommended by the International Standards Organization in 1981 and the TCP/IP model was started by the Department of Defense Advanced Research Projects Agency (ARPA), which has been mentioned at the very beginning of this chapter, in the late 1960s.

The OSI model is composed of seven layers, which are Application, Presentation, Session, Transport, Network, Data Link, and Physical layers from top to bottom. Each layer focuses on its own task that can be mapped onto one of the above six protocols and facilities. Typically, the application, presentation, and session layers undertake all the detailed tasks of application software. In detail, the application layer defines the medium with which application software communicates to other computers; the presentation layer defines file formats (such as binary, gif, jpeg, and text) in order to display a file in an appropriate way; the session layer is to initiate, control, and end communications. The transport layer takes the responsibility of the application data transportation via a network. The network layer realizes routing between hosts via a network. The data link layer carries out access to the

transmission medium for the data transmission on a network. The physical layer combines the tasks of selection and layout of the types of network topology and physical communication medium.

The TCP/IP model consists of four layers, which are Application, Transport, Network, and Link layers, shown in Figure 11.1. Each layer in this model can be also mapped onto one or more items of the above six items of protocols and facilities. The application layer carries out the tasks of application protocols. The transport layer undertakes the tasks of transmission protocols. The network layer realizes the missions of the routing protocols. And the link layer settles all the issues including the access to the transmission medium and the construction of communication device and network infrastructure, which are mostly involved in hardware control on the data flow from the origin host to the destination host over a network. In this four-layer model, the application layer focuses on the details of the applications without considering the communication while the other three layers do not care about the application but concentrate on how to move the data across the network.

As it is applied in the Internet, the TCP/IP model will be focused on in the following sections.

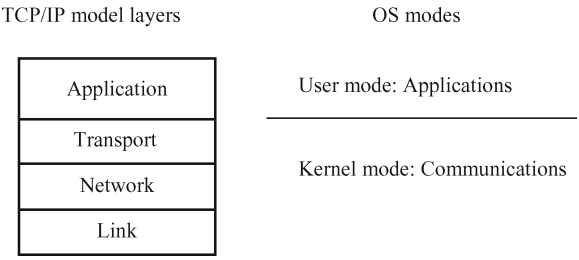

Fig. 11.1 TCP/IP model layers with process execution modes in OS.

11.2.5 TCP/IP Protocol Suite

Each layer of the TCP/IP model should fulfill the task that is defined by the protocols and facilities and in addition, there are usually several protocols that are associated with one of the layers in the TCP/IP model. These protocols make up the TCP/IP suite (commonly called the Internet Protocol Suite). Figure 11.2 shows the popular protocols in the four layers in the TCP/IP model. Some of them can implement the task of one layer, such as TCP and User Datagram Protocol (UDP), which can fulfill the mission in the transport layer; some of them can only practice part of the task of one layer, which means that they need some other protocol or facility to cooperate with to fulfill the task of their corresponding layer, such as Address Resolution

Protocol (ARP) and Reverse Address Resolution Protocol (RARP).

In this section, it will give general concepts about the protocols of the TCP/IP protocol suite. In the later sections, some important protocols, such as TCP, UCD, and IP protocols, will be discussed further. And some protocols of the application layer will be introduced along with the corresponding software in the UNIX operating system in the final several sections.

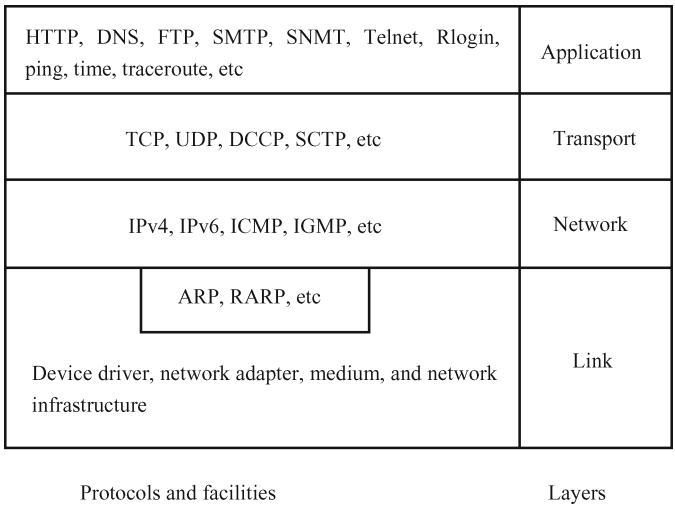

Fig. 11.2 The TCP/IP protocol suite.

11.2.5.1 Protocols in Application Layer

In the application layer, there are some protocols that are really well-known for the Internet users, for example, Hypertext Transfer Protocol (HTTP), Telnet and Rlogin (remote login), File Transfer Protocol (FTP), Simple Mail Transfer Protocol (SMTP), Domain Name System (DNS), Ping (testing a host reachable), Trivial File Transfer Protocol (TFTP), and Simple Network Management Protocol (SNMP). Each of these protocols is involved in one application that can be utilized by the Internet users to invoke one Internet service, interact with the Internet and do some activity over the Internet, such as Web browsing, sending and receiving email, chatting on-line, transmitting files between different hosts over the Internet, logging remotely in a host, etc.

11.2.5.2 Protocols in Transport Layer

As the purpose of the transport layer is to provide a flow of data between two hosts for the application layer above, the transport layer includes protocols that provide various functions, such as: segmentation, reassembly, and error recovery. In the transport layer, TCP and UDP are two dominant protocols.

The Transmission Control Protocol (TCP) along with the Internet Protocol (IP) constitutes the original of the Internet Protocol Suite (the TCP/IP protocol suite). In their cooperation for the message transmission over the

Internet, the IP tackles the lower-lever communication between hosts and the TCP handles the task only at a higher level of the two end systems, such as a Web browser and a Web server for the Web browsing application. The TCP provides reliable and sequenced delivery of a stream of bytes from one computer to another. Besides the common tasks of connection establishment, data transfer, reliable transmission, error detection, and connection termination, the tasks of the TCP also include the management of maximum segment size, flow control, congestion control, selective acknowledgments, window scaling, TCP timestamps, out-of-band-data, forcing data delivery, etc. A typical process of data transmission from one host to another with the TCP is to divide the data up from the application layer into the proper-sized packets for the below network layer, to send the packets via the IP, to acknowledge received packets, and to set timeouts in order to make sure the other end acknowledges packets to be sent. Since the TCP provides the reliable flow of data, the application layer supported by the TCP can ignore this issue. The first specifications of the TCP were written in RFC 675 in 1974. Since then, the TCP has been developed in many ways and the enhancements in its different aspects are published in some of the subsequent RFCs.

The User Datagram Protocol (also called the Universal Datagram Protocol, abbreviated as UDP), on the other hand, provides a much simpler service to the application layer without implicit hand-shaking dialogues for guaranteeing reliability and data integrity. In other words, the UDP just sends packets of data — datagrams — from one host to the other, but it does not make sure that the datagrams reach the other end. With the UDP at the transport layer, any desired reliability must be added by the application layer. However, in some situations, the UDP is more useful than the TCP. For example, for the time-tight applications, such as the streaming media systems of the Internet Protocol television (IPTV) and Voice over IP (VoIP), the rate of data transmission is more important than the data integrity, but the connection establishing for the reliable, error-free, and in-order delivery of data can make a significant delay. UDP was defined in RFC 768 in 1980.

Therefore, the UDP loads different applications in the application layer from the TCP. UDP can be applied in broadcasting and multicasting, DNS, etc. while TCP applications can be Telnet and Rlogin, FTP, SMTP, etc.

There are also some other transport layer protocols, such as DCCP and SCTP. Different from the TCP, Datagram Congestion Control Protocol (DCCP) provides a way with which to access to congestion control mechanisms without their implementation at the application layer and to allow for flow-based semantics without the reliable in-order delivery. This scheme quite fits into the applications time-limited on the data delivery, in which the delay of the reliable sequenced delivery and congestion control can make the data delivery useless to the receiver. The first implementation of DCCP was released in Linux Kernel Version 2.6.14 in 2005. And as a proposed standard, DCCP has been published in RFC 4340 (Kohler et al 2006) by the IETF in 2006. Stream Control Transmission Protocol (SCTP) provides not only some

of the same services of TCP and UDP, but also multiple streams (which mean the delivery of data chunks with independent streams to eliminate unnecessary delay of the TCP's head-of-line blocking) and multi-homing (which means that the system has multiple network interfaces to meet the need of highly-available data transfer) support. SCTP is introduced and defined in RFC 3286 (Ong et al 2002) and 4960 (Stewart 2007).

11.2.5.3 Protocols in Network Layer

The Internet Protocol (IP) is the foremost protocol in the network layer of the TCP/IP protocol suite and takes the job of delivering different protocol datagrams (or packets) from one host to another host just based on their addresses. For this objective, the IP describes addressing mechanics and datagram encapsulation structures. There are many versions of the IP, but among them, Internet Protocol Version 4 (IPv4) and Internet Protocol Version 6 (IPv6) are officially adopted. IPv4 uses 32-bit addressing structure, which has a capacity of more than 4 billion addresses while the IPv6 employs 128-bit addressing structure that holds about 340 undecillion addresses. IPv4 was defined in RFC 791 (Postel 1981.1) by IETF in 1981 and IPv6 was described in RFC 2460 around 1998. Today, IP4 is still the most widely exploited network protocol, although its successor, IPv6 is being supported by the worldwide infrastructure.

With the datagram encapsulation, the IP can be used to connect networks with different link layer implementations, such as Ethernet, token ring, ATM, Fiber Distributed Data Interface (FDDI), Wi-Fi, etc. Since different link layer implementation may have distinguished addressing strategies, it can be the most complex for IP to address and route. IP addressing is about how to allocate an IP address to an end host and how to group sub-networks of IP host addresses. IP routing can be performed by all hosts, but mostly by dedicated routers, which adopt either interior gateway protocols or external gateway protocols relied on their responsibilities in their network in order to help make IP datagram forwarding decisions across IP networks.

The IP provides an unreliable (or best effort) delivery. With this strategy, the benefits are reducing the network complexity and having routers along the transmission path just forward packets to the next gateway that matches the routing prefix of the destination address. And it also meets the need of the surroundings in the Internet, that is, no central monitoring facility exists and availability of links and nodes dynamically changes.

As the main protocol performing routing at the network layer, the IP is used for routing by both TCP and UDP. Any chunk of TCP and UDP data transferred around the Internet needs to pass through the network layer with the IP at both end systems and at every passed router.

Usually, the IP is used for communicating data across packet-switched networks. The packet switching is a communications technology that divides all transmitted data into packets regardless of their content and type. Contrary to the circuit switching applied in public switched telephone networks,

which sets up a dedicated connection in a constant bit rate and with constant delay between nodes for exclusive use during the communication session, the packet switching is to deliver sequences of packets through network adapters and routers over networks in a variable bit rate and with variable latency because of the dynamic of availability of links and nodes, network traffic load, and the packet buffering and queuing. The packet switching, partly, determines the unreliability of the Internet Protocol service.

In the Internet, there are two major packet switching modes: connectionless packet switching (also called datagram switching) and connection-oriented packet switching (also known as virtual circuit switching). For the former, each packet holds address information, with which the packets are routed individually, sometime in different paths and out of order. For the latter, a connection must be first established before any packet is transferred, via which the packets can be delivered in order and reliably by acknowledging after successful delivery and automatically repeating request for missing data or detected bit-errors. Compared to the connection-oriented packet mode whose connections are always unicast (for one single destination host), the connectionless packet switching mode has broadcast function (for all hosts in a network) and multicast function (for a set of hosts belonging to a multicast group), which can save network resources when the same data have to be transmitted to several destinations. IP and UDP are connectionless protocols while TCP is connection-oriented.

The Internet Control Message Protocol (ICMP) is an auxiliary for the IP. It is mainly utilized by the network layer to exchange error messages (such as unavailability of a requested service and unreachability of a host) and other vital information with the network layer in another host or router. It is not used directly in user common network applications, except two popular diagnostic tools, ping and traceroute. The ICMP (also called the ICMPv4) is for the IPv4 while for the IPv6 is the ICMPv6. The ICMP is defined in RFC 792 (Postel 1981.2), and the ICMPv6 is in RFC 4443 (Conta et al 2006).

The Internet Group Management Protocol (IGMP) is used to send a UDP datagram to multiple hosts – multicast. The IGMP can be applied in the establishment and management of the multicast group memberships for IP hosts and neighboring multicast routers. The IGMP can also enhance resource utilization when used in the applications such as online streaming video and gaming. The IGMP is only used with the IPv4 because the IPv6 has a different solution to multicast. Three versions of the IGMP are defined respectively in RFC 1112 (Deering 1989), RFC 2236 (Fenner 1997), and RFC 3376 (Cain et al 2002).

11.2.5.4 Protocols in Link Layer

As the services of the lower layers have to be transparent to the higher layers, the implementation detail of services in the link layer is hidden from the applications in order to make the services easy to be called when some application is raised. However, since it consists of software and hardware simultaneously,

the implementation of the link layer is different from and harder than those of the other layers. The implementation of services in the link layer of the TCP/IP protocol suite includes the interface between the network layer and link layer, the driver of network adaptor, and the network interface to the local network or router. Therefore, the protocols in the link layer are manifold and complex. For this reason, only some of them will be introduced in this book, and more information related to them can be referred to the references at the end of this chapter.

The Address Resolution Protocol (ARP) and Reverse Address Resolution Protocol (RARP) are dedicated protocols used to convert between the addresses executed by the network layer and the addresses by the hardware interface in the link layer for certain types of network interfaces (such as Ethernet and token ring). The ARP is described in RFC 826 (Plummer 1982) and the RARP in RFC 903 (Finlayson et al 1984). These two protocols are crucial for both the local network and the inter-network routing. For the latter, they can be used to make a decision on which router will be passed through next.

As mentioned above, the ARP and RARP have been implemented in many different types of networks, such as Ethernet, token ring, and many others. However, because of the prevailing of the IPv4 and Ethernet in networking, ARP is mostly used to map an IP address onto an Ethernet Media Access Control (MAC) address, which is a unique identifier assigned to most network adapters by the manufacturer for identification. The IP addresses are 32-bit, but the Ethernet MAC addresses (sometimes called as Ethernet Hardware Address, EHA) are 48-bit.

APR can have a function to test whether or not an IPv4 address is already in use, which is called APR probe. It is useful to make sure that an IPv4 address is available before using it.

In the link layer of the TCP/IP protocol suite, Ethernet protocol is also popular. Ethernet protocol, promoted by three companies (DEC, Intel, and Xerox) and published by the Institute of Electrical and Electronics Engineers (IEEE) around the early 1980s, uses 48-bit addresses and the bit rate of 10 Mbps. Its access method is called as the Carrier Sense, Multiple Access with Collision Detection (abbreviated as CSMA/CD). After that, the IEEE 802 Committee published a set of standards. Among these standards, 802.3 defines Ethernet networks, 802.4 defines token bus networks, and 802.5 defines token ring networks. Now Ethernet becomes one of the prevalent technologies in LANs. One of the most popular wired LAN implementations is the combination of the Ethernet over twisted pair (used in connection between end systems and the network) with the Ethernet over fiber optic (used in establishment of site backbones).

11.2.5.5 An Example for TCP/IP Protocol Suite

From the above sections, it is known that there are so many protocols in different layers of the TCP/IP protocol suite. Then, how do these protocols

work together? Figure 11.3 gives a typical example to show how the protocols in different layers cooperate together to fulfill file transfer between two hosts with client-server model over the Internet.

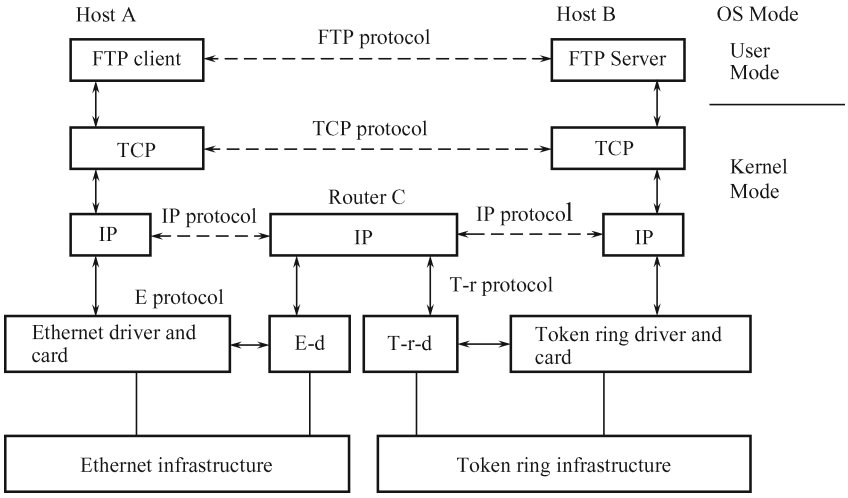

Fig. 11.3 Protocols used in file transmission between Host A and Host B via Router C (E-d replaces Ethernet driver and card, T-r-d replaces taken ring driver and card, E protocol means Ethernet protocol, and T-r protocol means token ring protocol in this figure).

In this case, the FTP protocol is executed in the application layer, and the TCP protocol is practiced in the transport layer of two hosts: A and B. As Host A is equipped with Ethernet adaptor and located in Ethernet LAN but Host B is within Token ring LAN, a router is necessary to connect these two different types of LANs. And the IP protocol is used in the network layer of Host A, Router C, and Host B. It is also seen that Ethernet protocol is running in the link layer of the Ethernet LAN and Token ring protocol is running within the same layer of the Token ring LAN. With the software modules that implement these protocols transmitting data down or up in sequence, the FTP client in Host A initiates the file transfer request and the FTP server in Host B responds and provides the corresponding service.

11.3 Encapsulation and Demultiplexing

The software implementation of the TCP/IP protocol suite is also called the protocol stack. As the TCP/IP protocol suite has several layers, the protocol stack has several modules, each of which implements the function of its corresponding layer. When a certain application sends data through each layer with some protocols, the data is delivered down via each module

of the protocol stack until it is transmitted as a stream of bits across the network. During the delivery process, each layer adds headers in front of and sometimes information at the end of the data received from the upper layer. In this way, the higher level objects can be hidden from the underlying structures and functions can be logically separated between the layers. This technique is called encapsulation. Figure 11.4 shows the encapsulation of the application data down through the protocol stack.

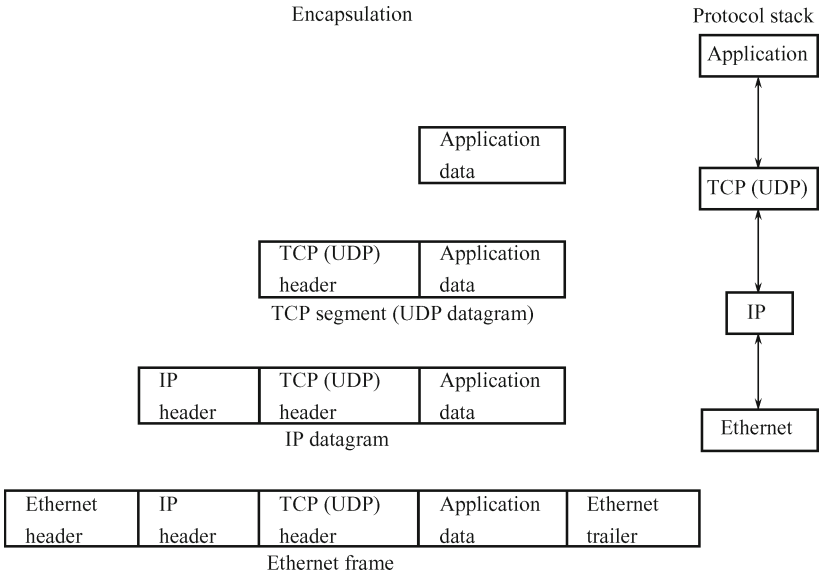

Fig. 11.4 Encapsulation of data.

If the TCP is used in the transport layer, the application data is encapsulated into TCP segments. A TCP segment has a 20-byte TCP header that is added to the application data. If the UDP is applied in the transport layer, the application data is encapsulated into UDP datagrams. A UDP datagram has an 8-byte UDP header. TCP segments (or UDP datagrams) are in turn encapsulated into IP datagrams via the network layer. The IP datagrams have 20-byte IP headers added to TCP segments (or UDP datagrams). When the Ethernet is used in the underlying layer, IP datagrams are then encapsulated into Ethernet frames before going across the network. The Ethernet frames have 14-byte Ethernet headers and 4-byte Ethernet trailers added to the IP datagrams. The size of an Ethernet frame is between 46 and 1500 bytes.

Known from Figure 11.2, there are several protocols in each layer of the TCP/IP protocol suite. These protocols need to be identified when the data travel up from layer to layer at the destination host. This issue can be tackled with the headers in different layers. In a header of one layer, there is one type of identifier to point out which protocol is used in the next upper layer. For example, to differentiate among the TCP, UDP, IMCP, and IGMP in the

transport layer, the IP headers consist of an 8-bit protocol field, which is equal to 6 for the TCP, 17 for the UDP, 1 for the ICMP, and 2 for the IGMP. As some applications use the TCP and some other applications use the UDP, TCP or UDP headers have 16-bit port numbers to distinguish these applications. In the Ethernet header, there is a 16-bit frame type field to identify the protocols used in the layer or sublayer above it. When the data reaches its destination host, these identifiers are quite useful.

Opposite to the encapsulation at the origin host, at the destination host, an Ethernet frame needs to be demultiplexed the same number of times as its encapsulation layer by layer until it rises at the application layer. At each layer, the proper protocol is used to remove the corresponding header and look at the identifier in the header to make a decision about which protocol should be used in the next upper layer.

11.4 Networking Operating Systems

Networking operating systems came along with the computer networking and Internet. In other words, networking operating systems coexist with the computer networking and Internet. Therefore, networking operating systems should include the protocol stack in their architecture in order to provide the facilities for the computer networking and Internet. Also for the distributed computing, modern operating systems usually can not only support the independent local applications but also meet the needs of computer networking. These types of operating systems can be configured with different function modules, such as clients or servers, according to their real roles in the networking. Therefore, networking operating systems should be more scalable, flexible, and dynamical.

As known in Section 11.2.4, when the networking is seen in the logical way, its functions can be divided into different layers of the TCP/IP or OSI models. From the view of the operating system, it is a good way to see networks in a logical perspective. In this way, we can put different modules associated with the protocols in different layers into corresponding layers of the operating system structure. As described in Figure 2.1, operating systems have their own hierarchy from hardware (just like physical layer in the OSI model) up to applications (just like application layer in the TCP/IP model). These natural relations can help the reader know in which level of the operating system an implementation module of one protocol should be put. It is important for the reader to have this perspective because this book is discussing UNIX operating systems now. For an operating system administrator or designer, it is meaningful and effective to make the networking modules fit into the operating system architecture.

As an operating system usually makes the physical detail hidden from the users, it is natural to put it in the kernel of the operating system how to

make the communication between two hosts. Therefore, when a user initiates a networking application, a user process is created for the user and then the operating system makes the corresponding system call to invoke the appropriate protocol modules in sequence according to the user application and the protocol stack. On the right side of Figures 11.1 and 11.3, at the application layer is running a user process while the functions of the lower three layers are usually executed in the kernel of the operating system. It is typical the way done in UNIX.

Since networking operating systems are a much younger and flourishing member of operating system family, there are still new emerging characters and potential for networking operating systems. However, as the benefits of sharing computer resources among users, providing a development cooperation platform for variedly located researchers, and enhancing the reliability of the whole computer system have been brought into the reality by the computer networking, some certain features of networking operating systems can be listed as follows.

Inside operating systems, in addition to the common services that operating systems usually have and have been introduced in previous chapters, some other facilities should be included in networking operating systems. No matter what role in a network is played by the computer with the networking operating system — a client, server, or router, the following are must-be:

- The modules are implemented for the protocols that carry out routing the application data over a network.
- The modules are practiced for the protocols that execute access to the physical transmission medium over a network.

For a client or server, the following is also necessary:

- The modules are realized for the protocols that fulfill the application data transportation between two processes on two hosts over a network.

While the above should be implemented in the kernel of operating systems, some network applications, attached to the interface between users and operating systems, can be provided. These applications can, typically, be:

- Network system administration, which is used to manage the resources over the network with high availability and fault tolerance.
- Network user administration, which is used to add, remove, and manage users who are expected to use resources on the network.
- Remote login, which is to allow users to access resources over the network.
- File transmission, which is used to transmit files between different hosts over a network.
- Email service, which is used for users to send and receive emails.
- Web browsing, which provides websites browse service.
- Data network storage and backup services, which can enlarge the storage space and provide expending data backup, besides data sharing.
- On-line chatting that provides a public or private platform for users to communicate with each other at real time.
- Network printer service, which can let several users share a number of

printers via a network.
- Network resource security, which can give users the proper authorization and authentication for login restriction and access control.
- Network database management system and database access logic, which can let differently-located users share some database via a network.
- Name service and directory service, which can help direct users to the hosts and directories over a network of interest.
- And other emerging network applications.

As mentioned in Sections 11.2.3 and 11.2.5, the software implementation of these applications is usually divided into two parts — client and server, and located in different computers, which relies on the roles played by the computers in a network. And according to the inner relationship between applications and lower-layer protocols, some of applications will be discussed along with their corresponding protocols in the following sections.

11.5 IP Protocol

The Internet Protocol (IP) is quite popular in the computer networking world. All the TCP, UDP, ICMP, and IGMP datagrams are transmitted with the IP. However, the IP provides an unreliable and connectionless delivery service.

Being unreliable, the IP just makes a best effort for its delivery service. If some mistakes happen in the delivery, the IP can send back an ICMP message to the source host. Thus, if a delivery needs reliability, it requires the support of the upper-layer protocol, such as the TCP. For being connectionless, the IP does not have the function of real connection between the source and destination hosts. The IP lets its datagrams deliver independently no matter whether or not the datagrams are linked with their context. Therefore, when they arrive at the destination host, the datagrams may be out of order. This issue can be handled by some fields in the IP header.

There are two UNIX commands: ping and traceroute, which have an inherent relationship with the IP and will be discussed in the following sections.

11.5.1 IP Header

As known, there are two official versions of IP, IPv4 and IPv6. They have different IP headers. Without any option, the regular size of the IPv4 header is 20 bytes (160 bits), shown in Figure 11.5, while the size of the IPv6 header is 40 bytes (320 bits), shown in Figure 11.6.

Here gives the explanation of the fields in the IPv4 header.
- Version: for IPv4, it is 4.
- Header length (H-length): it is the number of 32-bit words in an IP header

	0-3 bits	4-7 bits	8-15 bits	16-31 bits
0-31	Version	H-length	Type of service	Total length in bytes
32-63	Identification			Flags / Fragment offset (51-63 bits)
64-95	Time to live		Protocol	Header checksum
96-127	Source IP address			
128-159	Destination IP address			
	Options (bytes)			
	Data			

Fig. 11.5 IPv4 header and IPv4 datagram (The numbers in the blocks of the first row are the bit numbers that the blocks cover in one 32-bit word; Flags field and Fragment offset field in the third row cover three bits and 13 bits, respectively. The numbers in the blocks at the left column are the numbers of bits composing a regular IPv4 header of 20 bytes).

with any options. For a regular IPv4 header without any option, it is 5. As a 4-bit binary number is limited up to 15 in decimal system, the greatest length of an IPv4 header is 60 bytes.

- Type of service (TOS): its first 3 bits are usually ignored, the middle 4 bits are TOS bits, and the final bit is always 0. Among four bits of TOS, the first bit is minimize delay bit; the second one is maximize throughput bit; the third one is maximize reliability bit; the final one is minimize monetary cost bit. The names of these four bits can tell their functions. All the four bits can be 0 simultaneously, but only one of them can be 1. The minimum delay bit should be set 1 when used in remote login applications such as Telnet and Rlogin while the maximum throughput bit must be 1 when used in file transfer with FTP.
- Total length: it is the number of total bytes of an IPv4 datagram. Since it covers 16 bits, the greatest length of an IPv4 datagram is 65 535 bytes.
- Identification: it is a unique identifier for each datagram. It is added by one each time a datagram is sent between hosts or routers via a network.
- Flags: from high order to lower order, its first bit is reserved and always 0; the second bit is Don't Fragment (DF); and the third bit is More Fragments (MF).
- Fragmentation offset: it is the offset in 8-byte units of a particular fragment from the beginning of the original IP datagram.
- Time to live (TTL): it is an upper limit of the number of routers through which a datagram can travel. It is set to a certain value by the sender and decremented by one by each router via which the datagram passes. When the number is equal to 0 and the destination host is not reached, the datagram is discarded. In this way, it can prevent packets' forever routing over a network with limiting the lifetime of the datagram.

- Protocol: it is an identifier for the upper protocols that send the application data to the IP. It allocates 1 to the ICMP, 2 to the IGMP, 6 to the TCP, and 17 to the UDP.
- Header checksum: it is used to calculate the sum of an IP header when the datagram arrives at the destination in order to check whether or not the delivery is correct. When calculated, the IP header is considered as a sequence of 16-bit words. It only checks the IP header, and the upper protocol headers should be checked in their own layers.
- Source IP address: it is a unique 32-bit address for the source host. The IP address will be discussed in the following section.
- Destination IP address: it is a unique 32-bit address for the destination host.
- Options: it is the extended datagram information, such as security and handling restrictions, record route, timestamp, loose source routing, or strict source routing. Dependent on the need of the extended information, the length of the option list is variable.

As the link layer has an upper limit on the size of the frame that can be transmitted, the IP performs fragmentation if the datagram size is larger than this frame size. The fragments of an IP datagram are reassembled until they reach the destination. For fragmentation, the value of the identification field of an IP datagram is copied into the identification field of each of its fragments. The More Fragments of the flags field of all its fragments, except the final fragment, is set to 1. The fragment offset field of each fragment holds its offset from the beginning of the original IP datagram. The total length field of each fragment is the size of this fragment, not the whole IP datagram. This technique makes it possible for the receiver to reassemble the fragments in the correct way with the information in the IP header even though the fragments of a datagram arrive at the final destination out of order.

In Figure 11.5, the most significant bit is on the left and the least significant bit is on the right. IP datagrams are transmitted in the big-endian byte order. That means, the transmitting order is: $0-7$ bits are transmitted at the first, $8-15$ bits at the second, $16-23$ bits at the third and $24-31$ bits at the forth.

As shown in Figure 11.6, the IPv6 datagram has a total different header. The most significant feature of the IPv6 is that the IP address is extended from 32 bits of IPv4 to 128 bits. As the focus of this book is put on the IPv4, the details of other fields are beyond this book and can be referred in RFC 2460 (Deering et al 1989).

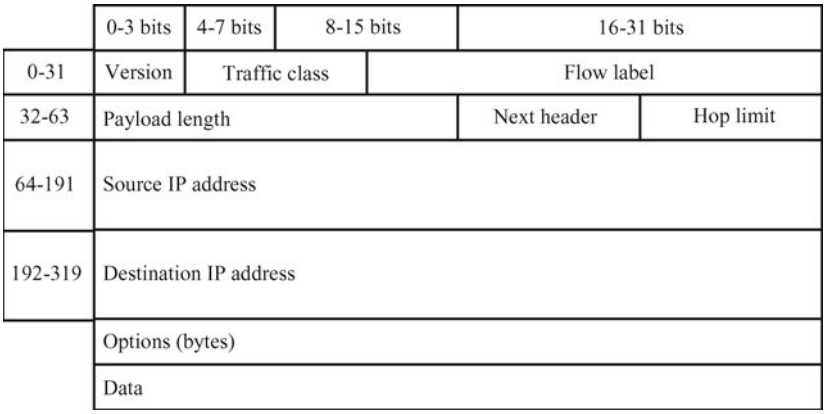

Fig. 11.6 IPv6 header and IPv6 datagram.

11.5.2 IPv4 Addressing

Every interface in the Internet must have a unique Internet address (IP address). For IPv4, the IP addresses are 32-bit. Instead of using a linear address space, an IPv4 address is composed of four decimal parts, each of which covers one byte of the address, for instance, 192.252.124.231. This is also called dotted-decimal notation. There are five classes in the Internet addresses. The class difference of an address can be identified in the first number of its dotted-decimal address, shown in Figure 11.7.

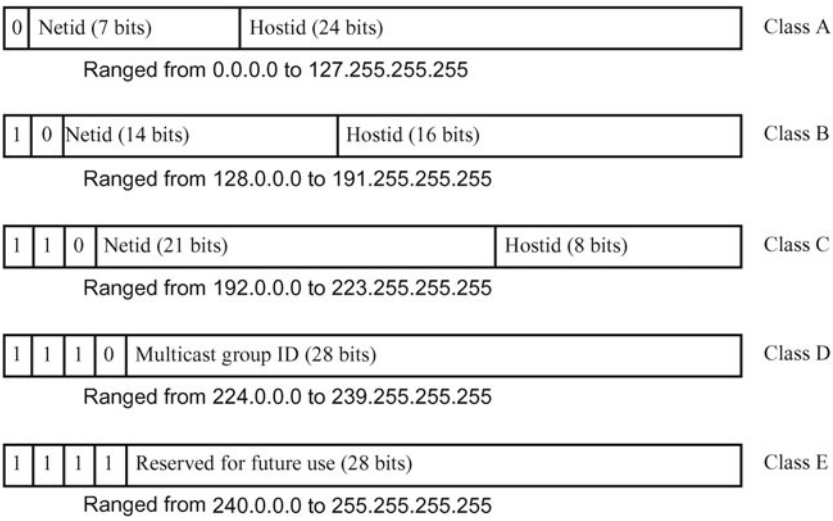

Fig. 11.7 Five classes of Internet addresses and their ranges.

11.5.3 IPv4 Routing

Mentioned in Section 11.2.2, when computer networking, a computer can take the role of a host or router. In the network layer, a computer with a multi-user operating system such as UNIX can be configured to act not only as a host but also as a router. Being just a host in a network, a computer cannot forward a datagram; as a router, a computer can do datagram forward.

In general, the network layer can receive a datagram from the upper layer that may perform one of the protocols, including TCP, UPD, ICMP, and IGMP, and send the datagram down to the link layer. Also, it can receive a datagram from a network interface. And then it sends the datagram to the upper layer if the computer IP address does match the datagram destination IP address. Or it forwards the datagram if the computer does not match the datagram destination IP address and has been configured as a router. Or it discards the datagram if the computer has not been configured as a router and the IP address does not match.

In the network layer, there is a routing table to support the IP routing. The data structure of each entry of the routing table typically consists of the following fields:

- IP address for the destination, which can be a host address or a network address. If it is a host address, the host flag bit is set. If it is a network address, it has a zero hostid (shown in Figure 11.7) and maps into all the hosts in the same network, and the host flag bit is zero.
- IP address in the networking that can be a next-hop router or a directly connected network, which is used to forward a datagram. If it is a next-hop router, the router flag bit is set.
- Flags, two of which are the host flag bit and router (or gateway) flag bit.
- Reference count, which displays the reference count of the route entry. For a connection-oriented protocol such as TCP, the route is held when the connection is established. Known that Telnet is an application protocol of TCP in Section 11.2.5, a Telnet execution can increase the reference count of the route entry by one.
- Use count, which shows the number of packets passed through the route. When a datagram-sending protocol such as ping is used, the use count is added by one for each action of the protocol performance.
- Name of the local network interface, which can be used by the kernel to map it into a socket device that is usually corresponding to an attached Ethernet card or Token ring card.

In the routing table, there are two special entries: default route and loop-back interface.

The default route allows any other network reachable through a single router. For default route entry in the routing table, the field of the destination IP address is identified as default, its router flag is set, and the IP address of the next-hop router is usually the gateway through which the user can access

the Internet.

The loop-back interface is used by the host to communicate to itself. It lets a client and server on the same host to communicate each other by using TCP/IP. For loop-back interface in the routing table, two fields of the destination IP address and the directly connected network have the same value: 127.0.0.1. By convention, the Class A network ID 127 (as Netid shown in Figure 11.7) is reserved for the loop-back interface.

The IP routing algorithm is shown in Figure 11.8. Typically, the IP routing algorithm makes at most three iterations of match searching in the routing table.

The first iteration is to search the routing table for the entry that the host flag is set and the IP address of the destination host matches both the network ID (the class number plus Netid in Figure 11.7) and the host ID (Hostid in Figure 11.7). If the matched entry is found and its route flag is set, the packet is sent to the specified next-hop router. If the route flag of the matched entry is reset, the packet is sent to the directly connected interface.

If the first iteration does not find the matched entry, the second iteration starts to search. The second iteration is to search the routing table for the entry that its host flag is reset and its IP address of destination network is matched. If the matched entry is found and its route flag is set, the packet is sent to the specified next-hop router. If the route flag of the matched entry is reset, the packet is sent to the directly connected interface.

If the second iteration does not find the matched entry, either, the third iteration starts to search. This time, it searches the routing table for the default entry. Usually, the default entry can be found and the packet is sent to the specified next-hop router. If even the default entry cannot be found, the error message, which can show "host unreachable" or "network unreachable", will be displayed.

As IP routing is done from hop to hop, IP protocol usually does not know the complete route to a destination if the destination is not connected directly to the sending host. All that IP routing does is to provide the IP address of the next-hop router to which the datagram is sent. It is assumed that the next-hop router is closer to the destination than the sending host is, and that the next-hop router is directly connected to the sending host. Another feature of IP routing is that the routing is specified to a network. That is, the routing table is just a subset of all the options of possibly routing for a datagram. In this way, it simplifies the routing for any datagram.

In UNIX, there is a routing daemon, called routed, to do initialize the routing table and decide which routes are selected into the routing table. Known in Section 4.5.6, daemons are usually started when the system is booting. The routing daemon is also started at the system booting and does updating the routing table periodically (typically once each 30 seconds). Except the routing daemon, the ifconfig command can be used to set an interface's address, and the route command can do the similar task. In some UNIX operating systems, the /etc/defaultrouter file holds a default router,

```
while (the end of routing table is not reached)
   {              /* the first search for a matching host address */
  if (the host flag is set and IP address of destination host is completely matched)      {
     if (the route flag is set)          {
            send the packet to the next-hop router;
            return;
            }
            else
            {
               send the directly connected interface;
               return;
            }
     }
   }
while (the end of routing table is not reached)
    {                  /* the second search for a matching net work address */
  if (the host flag is reset and IP address of destination network is matched)  {
     if (the route flag is set)          {
            send the packet to the next-hop router;
            return;
            }
            else
            {
               send the directly connected interface;
               return;
            }
      }
    }
while (the end of routing table is not reached)
    {                  /* the third search for a default route */
      if (it is not the default entry)  {
           continue;
          }
      send the packet to the next-hop router;
       return;
  }
Display error me ssage to show "host unr eachable" or "net work un reachable";
return;
```

Fig. 11.8 The C-style illumination of IP routing algorithm.

which is added to the routing table when the system is booting.

11.5.4 Two commands, ping and traceroute

Here, we will introduce two commands, ping and traceroute, which can be used to check the status of the network connected to our computers.

11.5.4.1 Testing a Host Connection

Ping command is usually used to test a network connection. A user can execute the ping program on a host, which sends the echo requests with a client to a server on the tested host. And the client waits for the response from the server. When the response comes back to the user host, the client displays the testing result on the user's screen.

In UNIX, if a process invokes the ping program, the kernel sets the identifier field in the ICMP (shown in Section 11.2.5.3) message to the process ID of the process, which makes the ping program distinguish the returned responses if there are several processes of ping executing simultaneously on the same host.

The syntax and function of ping command are as follows.

```
$ ping [-options] hostname
```

Function: to send an IP datagram to the host with the 'hostname' name to test whether it is connected to the network (or Internet) and display the tested result on the screen.

Common options:

-c n: to set the number of packets that will be sent and received to 'n';

-s p_size: to set the size of packets that will be sent to 'p_size' bytes.

Note: without any option, the ping program just displays "hostname is alive" if the hostname is available or "no answer" if the hostname is not found. By default, the size of packets to send is 56 bytes.

For example:

```
$ ping hebust.edu.cn
hebust.edu.cn is alive
$
```

The above ping command without any option and it displays just a simple sentence.

Another example:

```
$ ping -c 3 beds.ac.uk
PING beds.ac.uk (194.80.218.20):  56 data bytes
64 bytes from 194.80.218.20:  icmp_seq=0 ttl=255 time=347ms
64 bytes from 194.80.218.20:  icmp_seq=1 ttl=255 time=348ms
64 bytes from 194.80.218.20:  icmp_seq=2 ttl=255 time=348ms
--- beds.ac.uk PING Statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 347/348/348 ms
$
```

The second ping command with the option of -c 3 sends and receives three messages. The first line of the ping output displays the IP address of

the tested host. The DSN (Domain Name System) resolver has mapped the given hostname into the IP address, which will be discussed simply in Section 11.7.2.

We can see that the ICMP sequence number, TTL, and the round-trip time are following. The ICMP sequence number begins with 0 and is added by one each time a new echo request is sent. As the ICMP header is 8-byte, the whole size of each package is 64 bytes. As known, IP is a best effort datagram delivery service and its packets can be lost, reordered, or duplicated. The ping program displays the sequence number of each returned packet, having us check if packets are lost, reordered, or duplicated. If the echo reply for a certain sequence number does not appear in the displayed list, it is lost. If sequence number M is shown before sequence number M-1, they are reordered. When sequence number M appears twice or more in the list, it is duplicated.

Shown in Section 11.5.1, TTL is the time-to-live field of the IP header, which is an upper limit of the number of routers that a datagram can go through.

The ping program can also calculate the round-trip time with subtracting the current time when the reply returns by the sending time that has been stored in the ICMP message when the echo request was sent.

The third example:

```
$ ping -c 3 -s 500 beds.ac.uk
PING beds.ac.uk (194.80.218.20):  500 data bytes
508 bytes from 194.80.218.20:  icmp_seq=0 ttl=255 time=348ms
508 bytes from 194.80.218.20:  icmp_seq=0 ttl=255 time=347ms
508 bytes from 194.80.218.20:  icmp_seq=0 ttl=255 time=348ms
--- beds.ac.uk PING Statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 347/348/348 ms
$
```

The third ping command with the -c and -s options sends and receives three messages of the 508-byte size. Compared to the second ping command, since the packet size increases not too much, the round-trip time does not change a lot, either.

11.5.4.2 Tracing the Route from one Host to Another

Traceroute command can be used to trace the route from one host to another. It also uses ICMP and the TTL filed in the IP header. As the TTL field in the IP header occupies 8 bits, the maximum TTL value is 255. Each time the datagram is passed to a router, the TTL value is subtracted by one or the number of seconds that the router holds the datagram, which is dependent on the router. Mostly, the TTL value is subtracted by one by each passed router.

Therefore, the traceroute program takes the mechanism: firstly, it sends an IP datagram with a TTL of 1 to the destination host. The first encountered router subtracts the TTL value by 1 and makes it zero. Explained in Section

11.5.1, when the value of TTL is equal to 0 and the destination host is not reached, the datagram is discarded. And the ICMP shows time out. The first round-trip can identify the IP address of the first router in the path to the destination host. Then the traceroute program does the second test. It sends a datagram with a TTL of 2, and the IP address of the second router can be found. In this way, many times of round trips with increasing TTL values are taken until the destination host is reached. At the final round trip when the destination host is reached, as the TTL is equal to 1, the time is not exceeded and the datagram is not discarded by the destination host. Hence, the last reply returned to the sending host is different from the previous ones.

The syntax and function of traceroute command are as follows.

```
$ traceroute [-options] hostname
```

Function: to trace the host with the 'hostname' name and to display the routers in the path from the sending host to the destination host with the 'hostname' name on the screen.

Common options:

-n: to display the IP addresses rather than the hostnames.

Note: By default, the size of packets to send is 32 bytes.

For example:

```
$ traceroute hebust.edu.cn
traceroute to hebust.edu.cn (222.223.188.179), 30 hops max, 40 byte
packets
1 pc2010bvh.campus (192.168.189.2) 20 ms 10 ms 10 ms
2 hebust.edu.cn (222.223.188.179) 30 ms 30 ms 40 ms
$
```

The first line of the traceroute output displays the name and IP address of the destination and the maximum value of TTL that the traceroute can set is 30. The datagram size is 40 bytes with the 20-byte IP header and 8-byte UDP header. The rest 12-byte data include a sequent number, the time at which the datagram was sent, and a copy of the initial TTL value.

The following lines contain the TTL value, the name and IP address of the router or host, and the times taken of three packets sent by traceroute as they go from one router to the next. For any sent datagram, an asterisk is displayed if no reply is received within a set slice of time.

Because the traceroute command can threaten the network security by displaying a route to a host on the Internet and letting out the internal structure of the network to which the host is connected and the IP addresses of some routers on the network, its execution is usually disabled in most systems.

11.6 TCP Protocol and Applications

As a connection-oriented, reliable, and byte-stream protocol, Transmission control protocol (TCP) must establish a connection between two applications before providing other services. The two applications usually are at two different end points on a network.

11.6.1 TCP Segment, Header and Services

As shown in Figure 11.4, the unit of data that TCP transfers is called a TCP segment, and each segment has a 20-byte TCP header. Figure 11.9 gives the detail of a TCP header and segment.

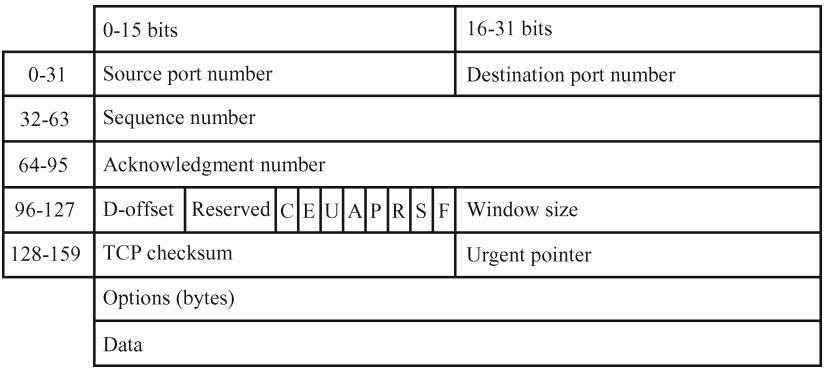

Fig. 11.9 TCP header and segment (as shownin Figure 11.5, the numbers in the blocks of the first row are the bit numbers that the blocks cover in one 32-bit word; D-offset in the fifth row covers four bits and represents Data offset, followed by four reserved bits; each Letter among C, E, U, A, P, R, S, and F in the fifth row covers one bit and represents CWR, ECE, URG, ACK, PSH, RST, SYN, and FIN, respectively. The numbers in the blocks at the left column are the numbers of bits composing a regular TCP header of 20 bytes).

The following is the description for each field in a TCP header.

- Source port number: it is to specify the sending application. Combined with the source IP address in the IP header for a datagram, the source port number can be used to call a socket (shown in Section 7.9.4), which is an interface provided by UNIX for the user to communicate with the network. The source couple of IP address and port number make up a client identifier that has two portions: a client IP address and a client port number, both of which can be used by the kernel when further networking services are provided.
- Destination port number: it is to specify the receiving application. Like the source port number, along with the destination IP address in the IP

header for a datagram, the destination port number can be used to call a socket at the destination host. And the destination couple of IP address and port number can compose a server identifier that has two portions: a server IP address and a server port number, both of which can be used by the kernel when further networking services are provided at the receiving host. A socket pair (including the source couple and destination couple) uniquely identifies a TCP connection between two end points in a network and supports inter-process communication on two machines in the network via the client-server model.

- Sequence number: it is a byte index number in the data stream transferred from the source host to the destination host. For the sending host, the TCP transfers segments with a unidirectional byte stream and marks each byte in sequence. If the SYN flag is set (which means a new connection being established), the segment number is an initial segment number and the actual first byte is with the segment number that is one more than the initial number, in which the acknowledged segment is considered. If the SYN flag is reset, it is the accumulated sequence number of the first byte of this segment for this data transference. As sequence number field covers 32 bits, it is reset to 0 when it is bigger than $2^{32} - 1$. As TCP is a full-duplex service to the application layer, each end of a TCP connection has to maintain two sequence numbers for the data flowing in two directions, sending and receiving, respectively.
- Acknowledgment number: it is a byte index number used by the receiving end. If the ACK flag is set, the acknowledgment number is the next sequence number that the receiving end is expecting. It is one more than the sequence number of the last successfully received byte by the receiver and acknowledges receipt of all the previous bytes. However, the first ACK segment acknowledges just the initial sequence number itself received by the receiver, but not data bytes.
- Data offset: it is the size of the TCP header (including options) in 32-bit words and also indicates the offset of the actual data from the start of the TCP segment. Without any option, the minimum header size is 20 bytes. Because the data offset field covers four bits, the maximum header size is 60 bytes, allowing for up to 40 bytes of options in the header.
- Eight flags: there are eight flag bits, CWR, ECE, URG, ACK, PSH, RST, SYN, and FIN. Congestion Window Reduced (CWR) is set if the sending end has received a TCP segment with the ECE flag set and has responded with congestion control mechanism. ECE is set for Explicit Congestion Notification-Echo (ECN-Echo). URG is set if the urgent pointer is valid. ACK is set if the acknowledgment number is valid, and ACK after the initial SYN segment should be always set. Push function (PSH) is set if the receiver should push the buffered data to the receiving application as soon as possible. RST is set to reset the connection. Synchronize sequence numbers (SYN) is set to initiate a new connection and only the first segment sent from each end has SYN set. FIN is set if the sender has

finished data sending.

- Window size: it gives the size of the receiving window, which specifies the number of bytes (starting from the sequence number in the acknowledgment filed) that the receiving end is currently willing to receive. As this field covers 16 bits, the window size is limited to 65 535 bytes.
- TCP checksum: it is used to make error-checking of the TCP header and data. TCP checksum of a TCP segment is calculated and stored by the sending end, and verified by the receiving end.
- Urgent pointer: it is an offset of the sequence number of the last byte of urgent data from the sequence number that is stored in the sequence number field of the segment. The urgent pointer is valid only if the URG flag is set.
- Options: it is variable from 0 to 320 bits. Different portions of options have different functions. For example, the first 8 bits are used for end of option list, the second 8 bits are used for padding, the closely following 32 bits are used for Maximum segment size (MSS), etc. For the detail, readers should refer to the references at the end of this chapter.

TCP's first specification was described in RFC 675 in 1974 (Cerf 1974). Later on RFC 793 (Postel 1981.3), 1122, 2581, and 3168 have given some clarifying and supplementary information for TCP.

TCP typically provides services including segmenting, sending, receiving, checking, and rearranging data. When some upper layer application does request for TCP's services, TCP firstly makes a TCP connection between the two ends according to the socket pair. It breaks application data into segments. Then it sends segments in a stream of bytes across the TCP connection between the two ends. When sending a segment, the sending TCP uses a timer, which is set how long the sender will wait for an acknowledgement receipt from the other end. If the acknowledgement receipt is not received in time, the segment is retransmitted. At the other end of the connection, the receiving TCP receives the segment, and makes a checksum on its header and data. If the segment is examined invalid, TCP discards it. If the segment is verified, TCP makes acknowledgment for this segment. Because TCP segments are transferred as IP datagrams and IP datagrams can be out of order or duplicated when being transmitted, the received TCP segments can be out of order or duplicated. The receiving TCP has to rearrange the data if necessary. If IP datagrams are duplicated, the receiving TCP has to discard the duplicated data. Then TCP passes the received data in the correct order to the appropriate application according to the destination port number in the TCP header.

Since each end of a TCP connection provides a finite amount of buffer space, the receiving TCP only allows the sending end to transmit as much data as the receiver's buffers can contain. In this way, TCP makes a flow control.

However, like UNIX's treating the contents of a file as a stream, TCP transfers a stream of bytes between two ends of a TCP connection and does

not know the contents of the transmitted bytes at all. It relies on the applications on each end of the connection to interpret the contents of the byte stream.

11.6.2 TCP Applications

In this section, we will discuss some popular applications that perform their network services for users via TCP. We will introduce telnet and rlogin, ftp, and email.

11.6.2.1 Remote Login: Telnet and Rlogin

Remote login is a fundamental application for some other applications: such as remote command execution. When we log in a remote host with telnet or rlogin, we can do some tasks just like what we do on a local host, including searching for some files or directories, copying files, execute a program, etc.

Remote login provides a facility that a user can log on a remote host across a network or the Internet, which may be thousands of miles far away, without having a terminal directly connected to the remote host. Remote login adopts the client-server model. There are two popular remote login application protocols: telnet and rlogin. Telnet is a standard application protocol that exists in almost every TCP/IP implementation and almost all the operating systems. The specification of telnet protocol can be found in RFC 854 (Postel et al 1983) that was published in 1983. Rlogin was originally designed for 4.2BSD and to work between UNIX operating systems only. But in some systems that support BSD, rlogin can also work well. The specification of rlogin protocol was given in RFC 1282 (Kantor 1991) that was available from 1991.

Telnet

The syntax and function of telnet are as follows.

```
$ telnet [-options] hostname
```

Function: to connect and log in a remote host with the 'hostname' name via a network.

Common options:

-a: to try to login automatically;

-l: to login with a specified user name.

Note: The hostname can be the name of the destination host or its IP address in dotted decimal notation. Usually a user has to have a valid account on the remote host if the user wants to log in the host, but some remote hosts allow a user to log in without an authorized account.

Since the telnet works in the client-server model, the user who is logging in the remote host with telnet interacts with the telnet client. The telnet

client provides two modes, command mode and input mode.

If a user types in the telnet command with a hostname as an argument, the telnet client prompts the 'login:' to let the user type in user name and password for the account on the remote host.

When a user logs in the remote host via the telnet command without an argument (a hostname), the telnet client brings the user into the command mode and prompts the 'telnet>' on the user's terminal. Then the user can type in some telnet commands after the prompt, such as to make a telnet connect to the remote host with hostname by entering 'open hostname', to close the telnet connection by entering 'close' or 'quit', to get a telnet command list by entering '?' or 'help', etc.

After a connection is built up between the client at the local host and the server at the remote host, every keystroke on the local terminal is sent to the remote host and the telnet goes into the input mode. In the input mode, there are also two options of entering style: one is a character at a time, which is the default option; the other is a line at a time. To go back to the command mode, the user can type the telnet escape key (^]).

In the input mode, as the user is interacting with the telnet client, what the user types in will be treated as commands for the operating system on the remote host, and the telnet server on the remote host will interpret them to take the corresponding actions.

Since most of systems should be accessed with authority for security issues, it is not wise to make an account known to public. However, for academic studies, we build up a temporary network of two hosts: one is a client host, called chost; the other is a server host, called shost. And we have a user name, wang, on both hosts. The following command is used on the client host (chost) to log on the server host (shost).

```
$ telnet
telnet> mode line
telnet> open shost
Trying 192.168.11.104...
Connected to shost.
Escape character is '^]'
...
Login:  wang
Password:
...           (typing in some command lines to be sent to the shost)
$ date        (here $ is the login shell prompt of shost)
^]
telnet> z
$ fg
telnet shost
^]
telnet>quit
$
```

In the above example, we first enter the telnet command. The telnet> prompt is displayed on the screen, which means it goes into the command mode of the telnet client. When we type in mode line after telnet>, we switch the entering style to line-a-time mode. We type in open shost to try

to make a connection between the chost and shost. Then it lets us enter the username and password for the account on the shost. When the connection is established, the telnet client enters the insert mode and sends whatever we type in after that point. We can type in some command lines just like what we do on the local shell because we now interact with the login shell on the shost. We type in date command and the system time of the shost is displayed on the screen.

After a while, we press CTRL-] back to the command mode of the telnet mode. Then we enter z that is a telnet client command for suspending the telnet execution and returning to the local host. The fg command can bring the telnet execution back to the foreground, just like what we do here.

Then we press CTRL-] back to the command mode of the telnet client. Finally, we enter quit to close the connection and quit the telnet operation.

Next example gives some experience of how to make a connection to some server via the Internet with the telnet client on the local host.

```
$ telnet mit.edu 79
Trying 18.9.22.69...
Connected to mit.edu.
Escape character is '^]'
guest
Student data loaded as of Oct 6, Staff data loaded as of Oct 6.
Notify Personnel or use WebSIS as appropriate to change your information.
Our on-line help system describes
  How to change data, how the directory works, where to get more info.
  For a listing of help topics, enter finger help@mit.edu. Try finger
  help_about@mit.edu to read about how the directory works.
 Directory bluepages may be found at http://mit.edu/communications/bp.
There were 3 matches to your request.
Complete information will be shown only when one individual matches
your query. Resubmit your query with more information.
For example, use both firstname and lastname or use the alias field.
        name:  Guest, Arthur Norman
  department:  Aeronautics And Astronautics
        year:  G
       alias:  A-guest
        name:  Guest, Iestyn W.
  department:  Lincoln Laboratory
       title:  Administrative Staff
       alias:  I-guest
        name:  Guest, Lindsay
  department:  Medical Department
       title:  Mental Health Resources Coordinator
       alias:  L-guest
Connection closed by foreign host.
$
```

In this example, we use a well-known port number (79) as an optical argument. 79 is the port number of finger server. As shown in Figure 11.9, every application has a unique port number in the TCP header, sometimes called well-known port number. Finger is also a TCP/IP application protocol (which was defined in RFC 1288 in 1991) (Zimmerman 1991) and has its corresponding command with the same name. Finger can be used to check the user information on a local or remote host. Here, we use guest as the user name, and the information shows that there are three matches.

The port number of the telnet server is 23. For readers to understand well the telnet protocol, it is necessary to know how the telnet client and server work. Figure 11.10 shows the function diagram of telnet client and server.

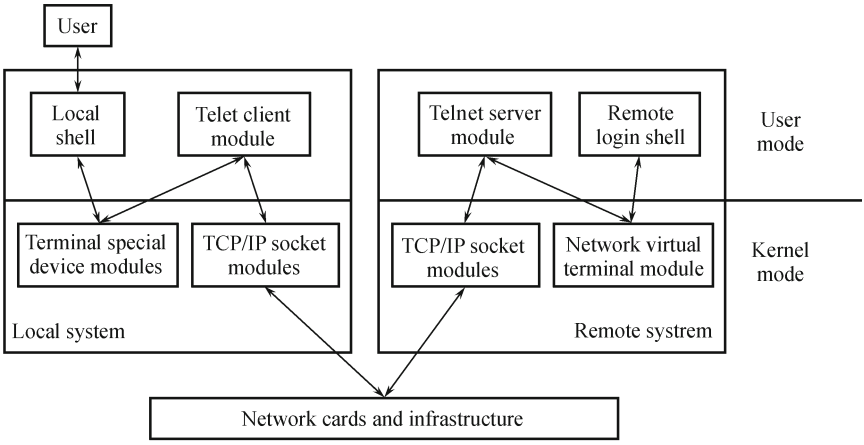

Fig. 11.10 The function diagram of telnet client and server.

As shown in Figure 11.10, when a connection is established between the local host and the remote host, the telnet client begins to interact with the user at the local terminal through a local shell. Via the TCP/IP protocols, the telnet client sends what the user typing in from the local terminal to the remote system and receives the response from the telnet server on the remote system. Then the reply from the telnet server is displayed on the local screen via the local shell.

The network virtual terminal (NVT), which was first defined in RFC 854, is an imaginary character device with an input device like keyboard and output device like a printer. The telnet client maps the user's input from the local terminal into the NVT, and the telnet server maps NVT into its input and its output. The client echoes what the user types to the output device by default. And data typed by the user on the local keyboard is sent to the server, and data received from the server is printed on the local screen.

The telnet server makes the NVT attached to the login shell on the remote host, where some remote interaction applications, such as text editors, can be used by the local user.

Rlogin

The syntax and function of rlogin are as follows.

```
$ rlogin [-options] hostname
```

Function: to connect and log in a remote UNIX host with the 'hostname' name via a network.

Common options:

-l: to login with a specified user name.

Note: The hostname can be the name of the destination host or its IP address in dotted decimal notation. If having a valid account on the remote host and logging in the remote host successfully, the user is put in the home directory and the login shell is executed. And all the hidden files for a local login are also executed for the remote login. If the user wants to interact just with the rlogin client rather than sending what entering to the rlogin server, the user can type a tilde key (~) as the first character of the line and followed by CTRL-Y (to suspend the client input and execute next commands on the local host), or CTRL-Z (to suspend the rlogin client), or CTRL-D (to terminate the rlogin). When the remote login is no use any more, typing in logout (or ~ CTRL-D) can make the remote login close and return to the local system.

We can rewrite the start of the first example of the telnet command as the following.

```
$ rlogin shost
Login:  wang
Password:  ******
...
$
```

11.6.2.2 File Transfer: FTP

FTP is another popular TCP/IP application protocol, which is the Internet standard for file transfer or a complete file copied from one host to another. FTP also needs an account to log in the file server, but anonymous FTP allows a user to access a server.

FTP occupies a well-known port, numbered 21. FTP also uses the client-server model, but needs two types of TCP connections: control connection and data connection. Firstly, a FTP server on the remote host makes an open on the port 21, and waits for a connection from a FTP client on some local host. The FTP client creates a control connection by making an open to the port 21. The FTP holds the control connection for the whole time when the client and server are communicating to each other. Each time a file is transmitted between the client and server, a data connection is just produced. IP does different treatments for the control connection and data connection. Shown in Figure 11.5, there is a type-of-service field in the IP header. For a control connection, IP sets the minimize delay bit of its type-of-service field. For a data connection, IP sets the maximize throughput bit of the type-of-service field.

FTP supports a number of file types, such as ASCII or binary, and file structures, like byte-stream or record-oriented. Specification of FTP was published in RFC 959 (Postel et al 1985) in 1985.

In UNIX and other operating systems, there is a command, named ftp, corresponding to FTP application.

The syntax and function of ftp are as follows.

```
$ ftp [-options] hostname
```

Function: to log in and transfer files from or to a remote UNIX host with the 'hostname' name via a network.

Common options:

-i: to disable prompting during multiple file transfers;

-v: to display all the responses from the remote host.

Note: Because of the way that the FTP makes a connection between the ftp client and server, if the ftp server on the remote host has not done an open on the 21 port before the ftp client on the local host does a connection request, the connection cannot be established and an error message is displayed by ftp command. Also because of security issues, a user should have a verified account and access permission to transfer files from or to a remote host, and mostly, ftp servers allow a user more likely to download files from than to upload files to them.

Table 11.1 lists some of the ftp commands.

Table 11.1 Some of ftp commands

Command	Function
ascii	To set the ftp to ASCII mode in order to transfer ASCII-type files like text files
binary	To set the ftp to binary mode in order to transfer non-ASCII files including executable files and image files
cd	To change the working directory on the remote host
close	To close the ftp connection to the remote host but stay in the ftp command
dir rdir lfile	To store the listing of rdir directory on the remote host into lfile file on the local host
get rfile [lfile]	To download rfile file from the remote host to the lfile file in the current directory on the local host, or to the rfile file in the current directory on the local host without lfile argument
help [command]	To show the information of command, or a summary list of all ftp commands without command argument
ls [rdir]	To list the files in the rdir directory on the remote host, or in the current directory on the remote host without rdir argument
mget rfiles	To download multiple files listed in rfiles from the remote host to the current directory on the local host
mput lfiles	To upload multiple files listed in lfiles from the local host to the current directory on the remote host
open [hostname]	To try to open a connection to the remote host
put lfile [rfile]	To upload lfile file from the local host to the rfile file in the current directory on the remote host, or to the lfile file in the current directory on the remote host without rfile argument
quit	To end the ftp command
user [login_name]	To log in the remote host with a user name, login_name
! command	To execute the command on the local host

To try some ftp commands, we can use anonymous FTP. The site of ftp.pku.edu.cn is an anonymous FTP site, so you can use anonymous as a user name and your e-mail address as the password to access it, like the following example.

For example:

```
$ ftp ftp.pku.edu.cn
Connected to vineyard.pku.edu.cn.
220-Welcome to public FTP service in Peking University
220
User < vineyard.pku.edu.cn:<none>>:  anonymous
331 Please specify the password.
Password:
230-
230- Max 3 connections an IP
230-
230- Only downloading permitted
230-
230- @ Peking University
230-
230 Login successful
ftp> ls
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
Linux
lost+found
mnt
open
welcome.msg
226 Directory send OK.
ftp:  received 43 bytes, time 0.00seconds 43000.00Kbytes/sec.
ftp>cd open
250 PORT command successful. Consider using PASV.
ftp>ls -l
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
drwxr-xr-x 2 ftp    ftp  4096 Mar 04 2010 389ds
...
drwxr-xr-x 2 ftp    ftp  4096 Feb 25 2010 OpenGTS
...
drwxr-xr-x 5 ftp    ftp  4096 Apr 20 05:50 ftp
...
drwxr-xr-x 2 ftp    ftp  4096 Jul 19 2010 openca
...
226 Directory send OK.
ftp:  received 5243 bytes, time 0.02seconds 327.69Kbytes/sec.
ftp> cd openca
250 Directory successfully changed.
ftp>ls -l
200 PORT command successful. Consider using PASV.
150 Here comes the directory listing.
-rw-r- - r-- 1 ftp ftp 9267407 Feb 25 2010 openca-base-1.0.2.tar.gz
-rw-r- - r-- 1 ftp ftp 397801 Feb 25 2010 openca-tools-1.1.0.tar.gz
226 Directory send OK.
ftp:  received 165 bytes, time 0.00seconds 165000.00Kbytes/sec.
ftp>get openca-tools-1.1.0.tar.gz
200 PORT command successful. Consider using PASV.
150 Opening BINARY mode data connection for openca-tools-1.1.0.tar.gz
<397801 bytes>.
226 Transfer complete.
ftp:  received 397801 bytes, time 2.27seconds 175.63Kbytes/sec.
```

```
ftp>quit
221 Goodbye.
$
```

In this example, we use the ftp command to connect to ftp.pku.edu.cn with the user name and password of anonymous and the author's email address, respectively. After making a connection successfully, we use the ls command to list the home directory on the remote host, and move the directory to the open directory. We use the ls -l command to make a long list of the open directory. Then we move the current directory to the subdirectory of open – openca. We find a compressed file – openca-tools-1.1.0.tar.gz – over there. Then we download the file to the current directory on the local system. Finally we quit the fit command.

11.6.2.3 Email Protocols and Programs

Email (electronic mail) is a popular application in the Internet. Involved in email are some protocols and the email user and transfer agent programs, which will be discussed in this section. Before we discuss the protocols and programs involved in email, we give a whole picture about how the email technology to operate over a network or the Internet. Figure 11.11 displays the function diagram of email technology.

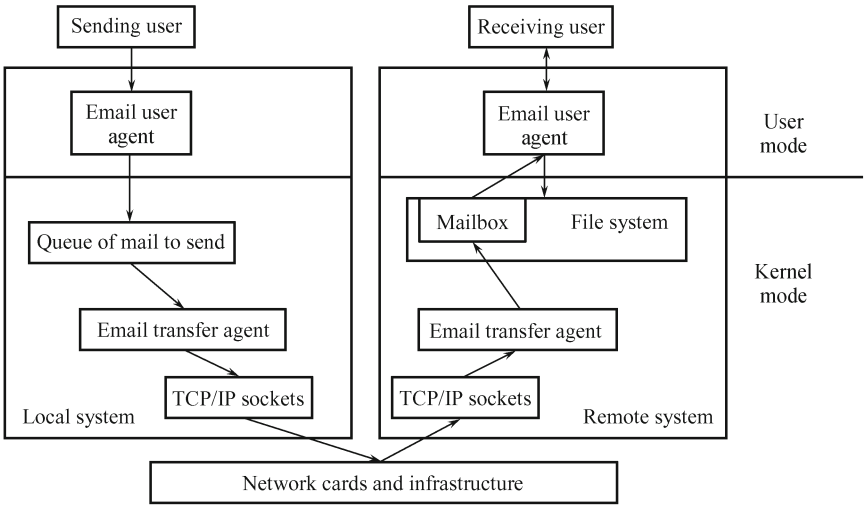

Fig. 11.11 The function diagram of email technology.

Typically, the process of sending an email from the sending host to receiving host is started from an email sender's launching an email user agent on a local host. The email user agent on the local host puts the email on the queue of mail to be sent. The email transfer agent (also called message transfer agent, or abbreviated MTA) on the local host manages the queue of email to be sent and sends the email through the TCP/IP sockets and a

network (or the Internet) to the remote host. On the remote host, another email transfer agent receives the email and puts it in the mailbox that is usually a directory in the remote file system, and sends a message "you have mail" to the email receiver. The receiver uses the email user agent to dispose the email, or to store the email in a specified directory, or forward it, or reply to its sender, or delete it according to his own wishes. The email is simply a file stored in the directory.

Email Protocols

There are several protocols that support email application. The most important one is Simple Mail Transfer Protocol (SMTP). RFC 821 (Postel 1982) published in 1982 gives the detail of SMTP protocol, which describes how to transfer the message between the local and remote hosts, or how two email transfer agents communicate each other via a TCP connection. RFC 822 (Crocker 1982) in 1982 is also about the email technology, and defines the format of an email message, including an envelope, a header, and a body, which is transferred by using RFC 821. SMTP protocol also adopts the client-server model.

There are two common fashions to access emails: one is from the specified host that stores emails; the other is from different hosts that can access the specified email host that contains emails. The former uses the Post Office Protocol (POP). The latter adopts the Internet Message Access Protocol (IMAP, or called Interactive Mail Access Protocol). Nowadays, the latter is the more popular and convenient fashion, which allows a user, no matter who is at home, or school, or office, using different computers at different locations, on which different email user agents may be available, to access the email transfer agent on an email server that adopts the IMAP protocol. This is the most common fashion that the Internet Service Providers (ISPs) adopt.

POP is described in RFC 918 of 1984 (Reynolds 1984), and IMAP protocol is in RFC 1064 of 1988 (Crispin 1988). And both of them are updated in several later RFCs, respectively.

RFC 1521 of 1993 describes Multipurpose Internet Mail Extensions (MIME) or Multimedia Internet Mail Standard (MIMS) (Borenstein et al 1993). For email application, MIME is another important protocol that defines the extensions to the email body structure. It adds some new headers to what the RFC 822 defines, which tell the recipient the structure of the body. In fact, it makes multimedia – files including images, sounds, movies, and compute programs – to be able to attach to an email. The body can still be transmitted as SMTP, regardless of the mail contents. Because of MIME, the size of an email can become quite large, maybe several megabytes. It also needs the email user agent to adopt the extensions on the email body structure.

Email Message Structure

Email messages consist of three parts: envelope, header, and body.

The envelope includes two fields:

- To: field, which specifies the destination address of the email;
- From: field, which identifies the source address of the email.

The envelope is used by the email transfer agent to transfer and deliver the email via a TCP connection. In UNIX, each user has a user name and the user name is also the email address of the user, which can be used in the destination address or source address of an email. When sending an email across the Internet, the email address should follow the rules of the Internet Domain Name System.

The above two fields also belong to the header. Except them, the header includes several other fields:

- Attach: field, which contains a list of attachments (files) that will be sent along with the message;
- CC: field, which means carbon copies and identifies other destination addresses of the email;
- Date: field, which holds the date when the email is sent;
- Reply-to: field, which holds the address where an original email is received from and this reply-to email will be sent to;
- Subject: field, which specifies the subject of the message.

Each field contains a name with a colon and the field value. There are some more header fields that are added automatically by the email agents. They are used by the agents rather than the user and are not explicitly given here. The attachment files can be ASCII text files or multimedia types and subtypes of files, including image files (such as jpeq and gif files), audio files (such as basic and mp3 files), and video files (such as mpeg files). The multimedia types and subtypes of files are supported by MIME.

The body is the main content of the message.

When the email user agent on the local host gets the body of the message typed by the user, it adds some header fields to the body, and passes the whole to the email transfer agent. Then, the email transfer agent adds the envelop and some more header fields to what the email user agent sends to it, and sends the total result to the email transfer agent on the remote host.

Email Programs

Shown in Figure 11.11, protocols involved in the email technology are implemented with two parts, that is, two types of email programs: one is email user agent; the other is email transfer agent. The former is used to edit, read, and dispose the messages; the latter is applied to transfer messages between the local host and remote host via a network or the Internet, and deliver the messages. There are many email user agents available in UNIX, such as mail command, pine, and KMail.

The pine program is developed for UNIX by the University of Washington.

Like the pico editor, introduced in Section 3.2, pine has the really similar user environment as the pico does, where a user can compose, send or read messages, and manage the address book. If the pine program is available in the system, users can get familiar to it quickly after using the pico.

KMail is a GUI program and executes under the KDE desktop environment (see in Section 2.5). Having the experience of Microsoft Windows, it is easy for readers to learn how to use KMail by themselves. In this book, we will discuss just the mail command.

In UNIX, the most popular email transfer agent is Sendmail. The system administrator sets up the local email transfer agent, and users can make a choice on which email user agent they use. The email transferring usually occupies the port 25.

Figure 11.11 shows a mailbox in the file system. In UNIX, the mailbox is usually the directory /usr/mail or some other directory, and emails are files in the directory. Each time a user logs in the system, if there is new email for him, the system gives notice "you have mail" on the screen. Users can use the mail command or pine program to read the mail in the mailbox.

Mail Command

In BSD, the mail program is called Mail and the following mail command can invoke Mail. In UNIX System V, the mail program is mailx. Here introduces the mail command. The syntax and function of mail are as follows.

```
$ mail [-options] [destination_address]
```

Function: to send or receive email.
Common options:
-b ad1: to specify the address ad1 where a blind carbon copy is sent to;
-c ad2: to specify the address ad2 where a carbon copy is sent to;
-P: to make all messages displayed with full headers;
-s: to specify the subject for email to be sent.

When sending email, just type the mail command with the destination addresses as arguments and in the way of entering other UNIX commands, like the following:

```
$ mail wangwei@hostname
```

After entering the mail command, it goes into the mail operation. You can type in the main body of the email message line by line. When finished, you press CTRL-D at a beginning of a new line to end the mail command and make it send the email.

When reading email, you enter mail without argument, just like:

```
$ mail
```

If there are some new emails waiting for your disposition, a list of message headers is displayed on the screen.

11.7 UDP Protocol and Applications

Different from TCP protocol that is connection-oriented, reliable, and byte-stream, UDP is a transport layer protocol that is simple, unreliable, and datagram-oriented. With the features of UDP, the applications based on UDP are different from ones of TCP.

11.7.1 UDP Protocol

RFC 768 (Postel 1980) published in 1980 gives the detailed description of the specification of UDP. From the studies in Section 11.6, we have known that TCP has to segment the application data. However, each UDP datagram is usually just one operation output of a process. And each UDP datagram is often sent with one IP datagram. UDP passes the application datagrams to the network layer without guarantee that all the datagrams can reach their destination.

Figure 11.12 displays the UDP header and datagram.

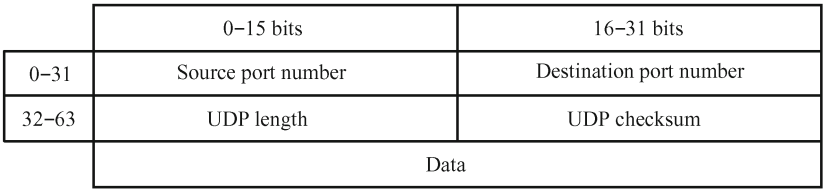

Fig. 11.12 UDP header and segment (as shown in Figure 11.5, the numbers in the blocks of the first row are the bit numbers that the blocks cover in one 32-bit word; the numbers in the blocks at the left column are the numbers of bits composing a regular UDP header of 8 bytes).

Compared to TCP header, UDP header is quite simple. In UDP header, the source and destination port numbers have the same meaning as in TCP header. Since the protocol field in IP header specifies the protocol (shown in Figure 11.5), the source and destination port numbers can identify UDP applications or TCP applications independently.

The UDP length field gives the datagram length in bytes, including the UDP header and UDP data. Similar to TCP checksum, the UDP checksum makes error-checking on both the UDP header and UDP data. However, the UDP checksum is optional rather than mandatory such as in TCP.

11.7.2 UDP Applications

Discussed in Section 11.2.5.3, unicast is always a connection-oriented pocket mode. The connectionless packet switching mode has broadcast and multicast functions, which can save network resources if the same data have to be transmitted to several destinations. TCP is connection-oriented, which means an explicit connection established between two application ports on two hosts. Therefore, broadcasting and multicasting can only be based on UDP, which is connectionless.

Considered within a LAN or MAN, many hosts share a network. When a host in the network wants to deliver a datagram to every other host on the network, the broadcasting can do this task. When a host on the network wants to send a datagram just to a group of the hosts on the network, multicasting can deal with it. A set of hosts on the network can be divided into a multicast group.

In fact, broadcasting or multicasting is relying on the filtering function of the TCP/IP protocol stack (shown in Figure 11.4). When a datagram reaches a host via the network or the Internet, it will go through all the modules in the protocol stack until up to the application layer. Let's take the right column in Figure 11.4 as an example. Firstly, when a datagram reaches a host, the network card (here an Ethernet card) on the host and Ethernet protocol check if the destination address of the datagram matches the hardware address of the card, the broadcast address or multicast address, and makes the first filtering. If the datagram passes the first filtering, it goes up to the IP layer. The IP examines the datagram according to the IP address in the IP header and does the second filtering. If the datagram also passes the second filtering, it goes up to the transport layer and accepts the examination of UDP in terms of destination port number and checksum in the UDP header. It is the third filtering. At any filtering, if the datagram does not match, it will be discarded.

Finally, we discuss another important protocol in the TCP/IP suite, Domain Name System (DNS), even though it is not dedicated to UDP applications. DNS is usually used to map between hostnames and IP addresses, and to provide routing information for other TCP/IP applications. And DNS is usually done first by an application because the application must convert a hostname given by a user to an IP address before it can request UDP to send its data or ask TCP to make a connection. DNS also adopts the client-server model.

DNS provides a distributed database whose data are scattered in different web sites on the Internet. As the Internet is giant, dynamic and distributed, any single site can not contain all the IP addresses of all the hosts in the Internet. Typically, each site manages its own DNS database and provides a server program to support the query from the clients on other hosts via the Internet.

BSD has its solution to DNS, called Berkeley Internet Name Domain

(BIND), including resolver and BIND server (also called named). The resolver is an interface for users provided as a DNS client. TCP/IP applications can access DNS through the resolver. The resolver invokes gethostbyname program to map a hostname into an IP address and invokes getnamebyaddr program to map an IP address into a hostname.

11.8 Summary

In this chapter, we have discussed UNIX in the Internet and computer networking. UNIX has many original contributions to the development history of the computer networking and Internet. Most of the networking protocols were initially implemented on UNIX and most of the Internet services are provided by server processes running on the UNIX operating system.

Since the late 1960s, in computer network engineering, the Request for Comments (RFCs) have become the official publications that can help exchange information among the global computer network researchers.

According to their scales, computer networks can be divided into three types: LAN, MAN, and WAN. In a network with the client-server model, every client is connected to the server and also each other. The resources in a server can be shared with clients, but a client does not share any of its resources with servers and other clients.

Logically, a network can be mapped into the TCP/IP model. The TCP/IP model consists of four layers, which are Application, Transport, Network, and Link layers. The application layer carries out the tasks of application protocols. The transport layer undertakes the tasks of transmission protocols. The network layer realizes the missions of the routing protocols. And the link layer settles all the issues including the access to the transmission medium and the construction of communication device and network infrastructure.

Each layer of the TCP/IP model should fulfill the task that is defined by the protocols and facilities. These protocols make up the TCP/IP suite. The software implementation of the TCP/IP protocol suite is also called the protocol stack. As the TCP/IP protocol suite has several layers, the protocol stack has several modules, each of which implements the function of its corresponding layer. When a certain application sends data through each layer with some protocols, the data is delivered down via each module of the protocol stack until it is transmitted as a stream of bits across the network.

Since networking operating systems are a much younger and flourishing member of operating system family, there are still new emerging characters and potential for them. However, networking operating systems basically should have the benefits of sharing computer resources among users, providing a development cooperation platform for variedly located researchers, and enhancing the reliability of the whole computer system, except the common services that operating systems usually have.

The Internet Protocol (IP) is quite popular protocol. All the TCP, UDP, ICMP, and IGMP datagrams are transmitted with the IP. However, the IP provides an unreliable and connectionless delivery service.

Along with IP protocol, two commands, ping and traceroute, have been discussed. Ping is usually used to test a network connection. Traceroute can be used to trace the route form one host to another.

The Transmission Control Protocol (TCP) is a connection-oriented, reliable, and byte-stream protocol, which must establish a connection between two applications at two different end points on a network before providing other services. We also have discussed some popular applications: telnet and rlogin, ftp, and email. Telnet and rlogin are two popular remote login application protocols. Telnet is a standard application protocol that exists in almost every TCP/IP implementation and almost all the operating systems. Rlogin was originally designed for 4.2BSD and to work between UNIX operating systems only. The File Transfer Protocol (FTP) is another popular TCP/IP application protocol, which is the Internet standard for file transfer. FTP needs an account to log in the file server, but anonymous FTP allows a user to access a server.

There are several protocols that support email application. The most important one is SMTP. There are two common fashions to access emails: one is from the specified host that stores emails; the other is from different hosts that can access the specified email host that contains emails. The former uses the Post Office Protocol (POP). The latter adopts IMAP. MIME is another important protocol that defines the extensions to the email body structure.

The User Datagram Protocol (UDP) is a transport layer protocol that is simple, unreliable, and datagram-oriented. Broadcasting and multicasting can only be based on UDP because it is connectionless.

The Domain Name System (DNS) is usually used to map between hostnames and IP addresses, and to provide routing information for other TCP/IP applications.

Problems

Problem 11.1 Try to refer to RFC 681 by yourself via the Internet and read its contents.

Problem 11.2 List some more benefits that the computer networking brings to us, except ones that have been given in Section 11.2.2.

Problem 11.3 Try to inquire into the computer network that your campus or company adopts. What operating system, model, and protocols are used in your computer network?

Problem 11.4 Give some application protocols that adopt the client-server model. Explain how a client and server cooperate to work with each other.

Problem 11.5 Try to find an application that uses a peer-to-peer model and explain how it works.

Problem 11.6 Try to find RFCs for TCP and UDP protocols via the web site given in the references and do some studies on them. Compare them in detail. Give some examples about where they are applied to, respectively?

Problem 11.7 Try to find Stream control transmission protocol (SCTP) via the web site in the references and do some researches on how SCTP works.

Problem 11.8 Please analyze the reasons of the unreliability of IP delivery. How can make a reliable delivery?

Problem 11.9 What are ARP and RARP used for? Try to do some research on ARP and RARP protocols via the web site given in the references.

Problem 11.10 Please analyze how the telnet protocol does remote login with the assistance of TCP, IP, and Ethernet protocols by referring to Figure 11.3.

Problem 11.11 Try to create a program that can implement the encapsulation of data from the application layer to transport layer.

Problem 11.12 How can you make the protocol stack fit into the hierarchy structure of the operating system? Please depict the hierarchy structure of your version of networking operating system.

Problem 11.13 Explain how the IP handles the datagram fragmentation after doing some research on the IPv4. Try to create a program to implement the IP datagram fragmentation.

Problem 11.14 Give a description on how IP does routing.

Problem 11.15 Try to program to accomplish some main functions of TCP, including segmenting, sending, receiving, checking, and rearranging data, after doing a detailed research on TCP protocol.

Problem 11.16 Refer to RFC 854 and do researches on how NVT (network virtual terminal) works.

Problem 11.17 Try to search some anonymous FTP sites over the Internet. Log in one of them with the anonymous user name and try some harmless ftp commands on the site.

Problem 11.18 Try to do researches on RFCs 821, 822, and 1521, and analyze the difference between two body structures of SMTP and MIME.

Problem 11.19 Try to study TCP and UDP protocols and analyze why UDP is unreliable when compared to TCP.

Problem 11.20 How can DNS handle information of the hosts existing in the Internet?

References

Borenstein N, Freed N (1993) MIME (Multipurpose Internet Mail Extensions) Part One: Mechanisms for specifying and describing the format of Internet message bodies. RFC 1521.http://tools.ietf.org/html/rfc1521. Accessed 10 Oct 2010

Cain B, Deering S, Kouvelas I et al (2002) Internet Group Management Protocol, Version 3. RFC 3376. http://tools.ietf.org/html/rfc3376. Accessed 10 Oct 2010

Cerf V, Dalal Y (1974) Specification of Internet Transmission Control Program. RFC 675. http://tools.ietf.org/html/rfc675. Accessed 10 Oct 2010

Comer DE (1998) Computer networks and Internets. Prentice Hall, Upper Saddle River, New Jersey

Comer DE, Stevens DL (1998) Internetworking with TCP/IP vol III: Client-server programming and applications, windows sockets version. Prentice Hall, Upper Saddle River, New Jersey

Conta A, Deering S, Gupta M (ed) (2006) Internet Control Message Protocol (ICMPv6) for the Internet Protocol Version 6 (IPv6) Specification. RFC 4443. http://tools.ietf.org/html/rfc4443. Accessed 10 Oct 2010

Crispin M (1988) Interactive mail access protocol, 2nd edn. RFC 1064. http://tools.ietf.org/html/rfc1064. Accessed 13 Oct 2010

Crocker DH (1982) Standard for the format of ARPA Internet text messages. RFC 822. http://tools.ietf.org/html/rfc822. Accessed 13 Oct 2010

Deering S (1989) Host Extensions for IP Multicasting. RFC 1112. http://tools.ietf.org/html/rfc1112. Accessed 10 Oct 2010

Deering S, Hinden R (1998) Internet Protocol, Version 6 (IPv6) Specification. RFC 2460. http://tools.ietf.org/html/rfc2460. Accessed 10 Oct 2010

Fenner W (1997) Internet Group Management Protocol, 2nd edn. RFC 2236. http://tools.ietf.org/html/rfc2236. Accessed 10 Oct 2010

Finlayson R, Mann T, Mogul JC at el (1984) A Reverse Address Resolution Protocol. RFC 903. http://tools.ietf.org/html/rfc903. Accessed 10 Oct 2010

Holmgren S (1975) Network UNIX. RFC 681. http://tools.ietf.org/html/rfc681. Accessed 10 Oct 2010

Kantor B (1991) BSD rlogin. RFC 1282. http://tools.ietf.org/html/rfc1282. Accessed 13 Oct 2010

Kohler E, Handley M, Floyd S (2006) Datagram congestion control protocol (DCCP). RFC 4340. http://tools.ietf.org/html/rfc4340. Accessed 10 Oct 2010

Ong L, Yoakum J (2002) An introduction to the stream control transmission protocol (SCTP). RFC 3286. http://tools.ietf.org/html/rfc3286. Accessed 10 Oct 2010

Plummer DC (1982) An Ethernet Address Resolution Protocol, or converting network protocol address to 48 bit Ethernet address for transmission on Ethernet hardware. RFC 826. http://tools.ietf.org/html/rfc826. Accessed 10 Oct 2010

Postel JB (1980) User datagram protocol. RFC 768. http://tools.ietf.org/html/rfc768. Accessed 10 Oct 2010

Postel JB (ed) (1981) Internet Protocol. RFC 791. http://tools.ietf.org/html/rfc791. Accessed 10 Oct 2010

Postel JB (1981) Internet Control Message Protocol. RFC 792. http://tools.ietf.org/html/rfc792. Accessed 10 Oct 2010

Postel JB (1981) Transmission Control Protocol. RFC 793. http://tools.ietf.org/html/rfc793. Accessed 10 Oct 2010

Postel JB (1982) Simple Mail Transfer Protocol. RFC 821. http://tools.ietf.org/html/rfc821. Accessed 13 Oct 2010

Postel JB, Peynolds JK (1983) Telnet Protocol specification. RFC 854. http://tools.ietf.org/html/rfc854. Accessed 13 Oct 2010

Postel JB, Reynolds JK (1985) File Transfer Protocol (FTP). RFC 959. http://tools.ietf.org/html/rfc959. Accessed 13 Oct 2010
Reynolds JK (1984) Post Office Protocol. RFC 918. http://tools.ietf.org/html/rfc918. Accessed 13 Oct 2010
Stevens WR (2002) TCP/IP illustrated, volume 1: The protocols. China Machine Press, Beijing
Stevens WR (2002) TCP/IP illustrated, volume 3: TCP for transactions, HTTP, NNTP and UNIX ® domain protocols. China Machine Press, Beijing
Stewart R (ed) (2007) Stream control transmission protocol. RFC 4960. http://tools.ietf.org/html/rfc4960. Accessed 10 Oct 2010
Wright GR, Stevens WR (2002) TCP/IP illustrated, volume 2: The implementation. China Machine Press, Beijing
Zimmerman D (1991) The finger user information protocol. RFC 1288. http://tools.ietf.org/html/rfc1288. Accessed 13 Oct 2010

Index

E

F

G

H

I

J

K

L

M

N

O

P

Q