Systems Programming in Unix/Linux

K. C. Wang

Systems Programming in Unix/Linux

K. C. Wang
School of Electrical Engineering
Washington State University
Pullman, WA, USA

ISBN 978-3-319-92428-1 ISBN 978-3-319-92429-8 (eBook)
https://doi.org/10.1007/978-3-319-92429-8

Library of Congress Control Number: 2018945425

This Springer imprint is published by the registered company Springer Nature Switzerland AG The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Dedicated
to Cindy

Preface

Systems programming is an indispensable part of Computer Science and Computer Engineering education. System programming courses in Computer Science/Engineering curriculums play two important roles. First, it provides students with a wide range of knowledge about computer system software and advanced programming skills, allowing them to interface with operating system kernel, perform file operations and network programming, and make efficient use of system resources to develop application programs. Second, it prepares students with needed background to pursue advanced studies in Computer Science/Engineering, such as operating systems, embedded systems, database systems, data mining, artificial intelligence, computer networks, and distributed and parallel computing. Due to its importance, systems programming in Unix/Linux has been a popular subject matter in CS/CE education and also for self-study by advanced programmers. As a result, there are a tremendous number of books and online articles in this area. Despite these, I still find it difficult to choose a suitable book for the Systems Programming course I teach at WSU. For many years, I had to use my own class notes and programming assignments in the course. After careful think ing, I have decided to put together the materials into a book form.

The purpose of this book is to provide a suitable platform for teaching and learning the theory and practice of systems programming. Unlike most other books, this book covers systems programming topics in greater depth and it stresses programming practice. It uses a series of programming projects to let students apply their acquired knowledge and programming skills to develop practical and useful programs. The book is intended as a textbook in technical-oriented systems programming courses. It contains detailed example programs with complete source code, making it suitable for self study also.

Undertaking this book project has proved to be another very demanding and time-consuming endeavor. While preparing the manuscripts for publica tion, I have been blessed with the encouragement and help from many people. I would like to take this opportunity to thank all of them. I want to especially thank Yan Zhang for his help in preparing figures for the book and proof reading the manuscript.

Special thanks go to Cindy for her continuing support and inspirations, which have made this book possible. Above all, I would like to thank my family for bearing with me with endless excuses of being busy all the time.

Sample solutions of programming projects in the book are available for download at http://wang.eecs.wsu.edu/~kcw. For source code, please contact the author by email.

K. C. Wang

Pullman, WA, USA April, 2018

Contents

About the Author

K. C. Wang received the BSEE degree from National Taiwan University in 1960 and the PhD degree in Electrical Engineering from Northwestern University, Evanston, IL, in 1965. He is currently a professor in the School of Electrical Engineering and Computer Science at Washington State University. His academic interests are in operating systems, distributed systems, and parallel computing.

Introduction 1

Abstract

This chapter presents an introduction to the book. It describes the book's scope, intended audience and its suitability as a textbook in Computer Science/Engineering curriculums. It presents a brief history of Unix, which includes early versions of Unix at

Bell Labs, AT&T System V and other developments of Unix, such as BSD, HP UX, IBM AIX and Sun/Solaris Unix. It describes the development of Linux and various Linux distributions, which include Debian, Ubuntu, Mint, Red Hat and Slackware. It lists both the hardware platforms and virtual machines for Linux. It shows how to install Ubuntu Linux to both VirtualBox and Vmware virtual machines inside the Microsoft Windows. It explains the startup sequence of Linux, from booting the Linux kernel to user login and command execution. It describes the Unix/Linux file system organization, file types and commonly used Unix/Linux commands. Lastly, it describes system administration tasks for users to manage and maintain their Linux systems.

1.1 About This Book

This book is about systems programming in the Unix/Linux (Thompson and Ritchie 1974, 1978; Bach 1986; Linux 2017) environment. It covers all the essential components of Unix/Linux, which include process management, concurrent programming, timer and time service, file systems, network program ming and MySQL database system. In addition to covering the functionalities of Unix/Linux, it stresses on programming practice. It uses programming exercises and realistic programming projects to allow students to practice systems programming with hands-on experiences.

1.2 Roles of Systems Programming

Systems programming is an indispensable part of Computer Science and Computer Engineering education. System programming courses in Computer Science/Engineering curriculums serve two main purposes. First, it provides students with a wide range of knowledge about computer system software and advanced programming skills, allowing them to interface with operating system kernel, make efficient use of system resources to develop application software. Second, it prepares students

with the needed background to pursue advanced studies in Computer Science/Engineering, such as operating systems, embedded systems, database systems, data mining, artificial intelligence, computer networks, network security, distributed and parallel computing.

1.3 Objectives of This Book

This book is intended to achieve the following objectives.

1.3.1 Strengthen Students Programming Background

Computer Science/Engineering students usually take introductory programming courses in their first or second year. Such courses typically cover basic programming in C, C++ or Java. Due to time limitations, these courses can only cover the basic syntax of a programming language, data types, simple program structures and using library functions for I/O. Students coming out of such course are rather weak in programming skills due to a lack of sufficient practices. For instance, they usually do not know much about the software tools, the steps of program development and run-time environment of program executions. The first object of this book is to provide students with the needed background and skills to do advanced programming. This book covers program development steps in detail. These include assembler, compiler, linker, link library, executable file contents, program execution image, function call conventions, parameter passing schemes, local variables, stack frames, link C program with assembly code, program termination and exception handling. In addition, it also shows how to use Makefiles to manage large programming projects and how to debug program executions by GDB.

1.3.2 Applications of Dynamic Data Structures

Most elementary data structure courses cover the concepts of link lists, queues, stacks and trees, etc. but they rarely show where and how are these dynamic data structures used in practice. This book covers these topics in detail. These include C structures, pointers, link lists and trees. It uses programming exercises and programming projects to let the students apply dynamic data structures in real life. Case in point: a programming exercise is for the students to display the stack contents of a program with many levels of function calls, allowing them to see the function calling sequence, passed parameters, local variables and stack frames. Another programming exercise is to let students imple ment a printf-like function with varying parameters, which requires them to apply parameter passing schemes to access varying parameters on stack. Yet another programming exercise is to print the partitions of a disk, in which the extended partitions form a “link list” in the disk, but not by the conventional pointers as they learned in programming languages. A programming project is to implement a binary tree to simulate the Unix/Linux file system tree, which supports pwd, ls, cd, mkdir, rmdir, creat, rm operations as in a real file system. Besides using dynamic data structures, the programming project allows them to apply a wide range of programming techniques, such as string tokenization, search for nodes in a tree, insert and delete tree nodes, etc. It also allows them to apply binary tree traversal algorithms to save binary trees as files and reconstruct binary trees from saved files.

1.3.3 Process Concept and Process Management

The major focus of systems programming is the abstract notion of processes, which is rather hard to grasp. This book uses a simple C program together with a piece of assembly code to let students see real processes in action. The program implements a multitasking environment to simulate process operations in an OS kernel. It allows the students to create processes, schedule processes by priority, run different processes by context switching, maintain process relations by a binary tree, use sleep and wakeup to implement wait for child process termination, etc. This concrete example makes it much easier for students to understand processes and process management functions. Then it covers process management in Unix/Linux in detail, which includes fork(), exit(), wait() and change process execu tion image by exec(). It also covers I/O redirection and pipes. It uses a programming project to let the students implement a sh simulator for command execution.

The sh simulator behaves exactly the same as the bash of Linux. It can execute all Unix/Linux commands with I/O redirections, as well as compound commands connected by pipes.

1.3.4 Concurrent Programming

Parallel computing is the future of computing. It is imperative to introduce parallel computing and concurrent programming to CS/CE students at an early stage. This book covers parallel computing and concurrent programming. It explains the concept of threads and their advantages over processes. It covers Pthreads (Pthreads 2017) programming in detail. It explains the various kinds of threads synchronization tools, such as threads join, mutex lock, condition variables, semaphores and barriers. It demonstrates applications of threads by practical examples, which include matrix computation, quicksort and solving system of linear equations by concurrent threads. It also introduces the concepts of race conditions, critical regions, deadlock and deadlock prevention schemes. In addition to using Pthreads, it uses a programming project to let students experiment with user-level threads, implement threads synchronization tools and practice concurrent programming by user-level threads.

1.3.5 Timer and Time Functions

Timer and time functions are essential to process management and file systems. This book covers the principle of hardware timers, timer interrupts and time service function in Unix/Linux in detail. It shows how to use interval timers and timer generated signals in systems programming. In addition, it uses a programming project to let students implement interval timer support for concurrent tasks.

1.3.6 Signals, Signal Processing and IPC

Signals and signal processing are keys to understanding program exceptions and Inter-Process Communication (IPC). This book covers signals, signal processing and IPC in Unix/Linux. It explains signal origins, signal delivery and processing in the OS kernel and the relation between signals and exceptions. It shows how to install signal catchers to handle program exceptions in user mode. It uses a programming project to let students use Linux signals and pipes to implement an IPC mechanism for concurrent tasks to exchange messages.

1.3.7 File system

Besides process management, file system is the second major component of an operating system. This book provides an in depth coverage of file systems. It describes file operations at different levels and explains their relationships. These include storage devices, file system support in Unix/Linux kernel, system calls for file operations, library I/O functions, user commands and sh scripts. It uses a series of programming exercises to let the students implement a complete EXT2 file system that is totally Linux compatible. These not only allow the students to have a thorough understanding of file operations but also provide them with the unique experience of writing a complete file system. Another programming project is to implement and compare the performances of different I/O buffer management algorithms. The simulation system supports I/O operations between processes and disk controllers. It allows the students to practice I/O programming, interrupts processing and synchronization between processes and I/O devices.

1.3.8 TCP/IP and Network Programming

Network programming is the third major component of an operating system. This book covers TCP/IP protocols, socket API, UDP and TCP programming using sockets and the server-client model in network computing. A programming project is for the students to implement a server-client based system for remote file operations across the Internet. As of now, WWW and Internet services have become an indispensable part of daily lives. It is essential to introduce CS/CE students to this technology. This book covers HTTP and Web programming in detail. It allows the students to configure the HTTPD server on their Linux machines to support HTTP programming. It shows how to use HTML for static Web pages and PHP to create dynamic Web pages. It covers the MySQL database system, shows how to create and manage databases in both command and batch modes and interface MySQL with both C and PHP. A programming project is for the students to implement a Web server to support file operations by HTTP forms and dynamic Web pages using CGI programming.

1.4 Intended Audience

This book is intended as a textbook for systems programming courses in technically oriented Computer Science/Engineering curriculums that emphasize both theory and programming practice. The book contains many detailed working example programs with complete source code. It is also suitable for self-study by advanced programmers and computer enthusiasts.

1.5 Unique Features of This Book

This book has many unique features which distinguishes it from other books.

1. This book is self-contained. It includes all the foundation and background information for studying systems programming. These include program development steps, program execution and termi nation, function call conventions, run-time stack usage and link C programs with assembly code. It also covers C structures, pointers and dynamic data structures, such as link lists, queues and binary trees. It shows applications of dynamic data structures in system programming.

2. It uses a simple multitasking system to introduce and illustrate the abstract notion of processes. The multitasking system allows students to create processes, schedule and run different processes by context switching, implement process synchronization mechanisms, control and observe process executions. This unique approach allows the students to have a concrete feeling and better understanding of process operations in real operating systems.

3. It describes the origin and roles of processes in Unix/Linux in detail, from system booting to INIT process, daemon processes, login processes and the user sh process. It describes the execution environment of user processes, such as stdin, stdout, stderr file streams and environment variables. It also explains how does the sh process execute user commands.
4. It covers process management functions in Unix/Linux in detail. These include fork(), exit(), wait (), changing process execution image by exec(), I/O redirection and pipes. Moreover, it allows the students to apply these concepts and programming techniques to implement a sh simulator for command execution. The sh simulator behaves exactly the same as the standard bash of Linux. It supports executions of simple commands with I/O redirections, as well as compound commands connected by pipes.
5. It introduces parallel computing and concurrent programming. It covers Pthreads programming by detailed examples, which include matrix computation, quicksort and solving systems of linear equations by concurrent threads, and it demonstrate the use of threads synchronization tools of join, mutex lock, condition variables and barriers. It also introduces the important concepts and problems in concurrent programming, such as race conditions, critical regions, deadlock and deadlock prevention schemes. In addition, it allows the students to experiment with user-level threads, design and implement threads synchronizing tools, such as threads join, mutex lock, semaphores and demonstrate the capabilities of user-level threads by concurrent programs. These allow the students to have a deeper understanding how threads work in Linux.
6. It covers timers and time service functions in detail. It also allows the reader to implement interval timers for concurrent tasks in a multitasking system.
7. It presents a unified treatment of interrupts and signals, treating signals as interrupts to processes, similar to interrupts to a CPU. It explains signal sources, signal types, signal generation and signal processing in great detail. It shows how to install signal catchers to allow processes to handle signals in user mode. It discusses how to use signals for IPC and other IPC mechanisms in Linux. It uses a programming project to let the students use Linux signals and pipes to implement an IPC mechanism for tasks to exchange messages in a multitasking system.
8. It organizes file operations into a hierarchy of different levels, ranging from storage devices, file system support in Unix/Linux kernel, system calls for file operations, library I/O functions, user commands and sh scripts. This hierarchical view provides the reader with a complete picture of file operations. It covers file operations at the various levels in detail and explains their relationships. It shows how to use system calls to develop file operation commands, such as ls for listing directory and file information, cat for displaying file contents and cp for copying files.
9. It uses a programming project to lead the students to implement a complete EXT2 file system. The programming project has been used in the CS360, System Programming course, at Washington State University for many years, and it has proved to be very successful. It provides students with the unique experience of writing a complete file system that not only works but is also totally Linux compatible. This is perhaps the most unique feature of this book.
10. It covers TCP/IP protocols, socket API, UDP and TCP programming using sockets and the server client model in network programming. It uses a programming project to let the students implement a server-client based system for remote file operations across the Internet. It also covers HTTP and Web programming in detail. It shows how to configure the Linux HTTPD server to support HTTP

programming, and how to use PHP to create dynamic Web pages. It uses a programming project to let the students implement a Web server to support file operations via HTTP forms and dynamic Web pages using CGI programming.

11. It emphasizes on programming practice. Throughout the book, it illustrates system programming techniques by complete and concrete example programs. Above all, it uses a series of program ming projects to let students practice system programming. The programming projects include (1) Implement a binary tree to simulate the Unix/Linux file system tree for file operations (Chap. 2).
    (2) Implement a sh simulator for command executions, including I/O redirections and pipes (Chap. 3).
    (3) Implement user-level threads, threads synchronization and concurrent programming (Chap. 4).
    (4) Implement interval timers for concurrent tasks in a multitasking system (Chap. 5).
    (5) Use Linux signals and pipes to implement an IPC mechanism for concurrent task to exchange messages (Chap. 6).
    (6) Convert file pathname to file's INODE in an EXT2 file system (Chap. 7). (7) Recursive file copy by system calls (Chap. 8).
    (8) Implement a printf-like function for formatted printing using only the basic operation of putchar() (Chap. 9).
    (9) Recursive file copy using sh functions and sh scripts (Chap. 10).
    (10) Implement a complete and Linux compatible EXT2 file system (Chap. 11). (11) Implement and compare I/O buffer management algorithms in an I/O system simulator (Chap. 12).
    (12) Implement a file server and clients on the Internet (Chap. 13).
    (13) Implement a file server using HTTP forms and dynamic Web pages by CGI programming (Chap. 13).
    Among the programming projects, (1)–(3), (6)–(9), (12)–(13) are standard programming assignments of the CS360 course at Washington State University. (10) has been the class project of the CS360 course for many years. Many of the programming projects, e.g. (12)–(13) and especially the file system project (10), require the students to form two-person teams, allowing them to acquire and practice the skills of communication and cooperation with peers in a team oriented working environment.

1.6 Use This Book As Textbook in Systems Programming Courses

This book is suitable as a textbook for systems programming courses in technically oriented Computer Science/Engineering curriculums that emphasize both theory and practice. A one-semester course based on this book may include the following topics.

1. Introduction to Unix/Linux (Chap. 1)
2. Programming background (Chap. 2).
3. Process management in Unix/Linux (Chap. 3)
4. Concurrent programming (Parts of Chap. 4)
5. Timer and time Service functions (Parts of Chap. 5)
6. Signals and signal processing (Chap. 6)

7. File operation levels (Chap. 7)
8. File system support in OS kernel and system calls for file operations

(Chap. 8) 9. I/O library functions and file operation commands (Chap. 9)
10. Sh programming and sh scripts for file operations (Chap. 10)
11. Implementation of EXT2 file system (Chap. 11)
12. TCP/IP and network programming, HTTP and Web programming
(Chap. 13). 13. Introduction to MySQL database system (Parts of Chap. 14).

The problems section of each chapter contains questions designed to review the concepts and principles presented in the chapter. Many problems are suitable as programming projects to let students gain more experience and skills in systems programming.

Most materials of this book have been used and tested in the Systems Programming course, CS360, in EECS at Washington State University. The course has been the backbone of both Computer Science and Computer Engineering programs in EECS at WSU. The current CS360 course syllabus, class notes and programming assignments are available for review and scrutiny at

http://www.eecs.wsu.edu/~cs360

The course project is to lead students to implement a complete EXT2 file system that is totally Linux compatible. The programming project has proved to be the most successful part of the CS360 course, with excellent feedbacks from both industrial employers and academic institutions.

1.7 Other Reference Books

Due to its importance, systems programming in Unix/Linux has been a popular subject matter in CS/CE education and also for self-study by advanced programmers. As a result, there are a tremendous number of books and online articles in this area. Some of these books are listed in the reference section (Curry 1996; Haviland et al. 1998; Kerrisk 2010; Love 2013; Robbins and Robbins 2003; Rochkind 2008; Stevens and Rago 2013). Many of these books are excellent programmer's guides, which are suitable as additional reference books.

1.8 Introduction to Unix

Unix (Thompson and Ritchie 1974, 1978) is a general purpose operating system. It was developed by Ken Thompson and Dennis Ritchie on the PDP-11 minicomputers at Bell Labs in the early 70s. In 1975, Bell Labs released Unix to the general public. Initial recipients of this Unix system were mostly universities and non-profit institutions. It was known as the V6 Unix. This early version of Unix, together with the classic book on the C programming language (Kernighan and Ritchie 1988) started the Unix revolution on Operating Systems, with long-lasting effects even to this day.

1.8.1 AT&T Unix

Development of Unix at AT&T continued throughout the 1980s, cumulating in the release of the AT&T System V Unix (Unix System V 2017), which has been the representative Unix of AT&T.

System V Unix was a uniprocessor (single CPU) system. It was extended to multiprocessor versions in the late 80s (Bach 1986).

1.8.2 Berkeley Unix

Berkeley Unix (Leffler et al. 1989, 1996) refers to a set of variants of the Unix operating system developed by the Berkeley Software Distribution (BSD) at the University of California, Berkeley, from 1977 to 1985. The most significant contributions of BSD Unix are implementation of the TCP/IP suite and the socket interface, which are incorporated into almost all other operating systems as a standard means of networking, which has helped the tremendous growth of the Internet in the 90s. In addition, BSD Unix advocates open source from the beginning, which stimulated further porting and develop ment of BSD Unix. Later releases of BSD Unix provided a basis for several open source development projects, which include FreeBSD (McKusick et al. 2004), OpenBSD and NetBSD, etc. that are still ongoing to this day.

1.8.3 HP Unix

HP-UX (HP-UX 2017) is Hewlett Packard’s proprietary implementation of the Unix operating system, first released in 1984. Recent versions of HP-UX support the HP 9000 series computer systems, based on the PA-RISC processor architecture, and HP Integrity systems, based on Intel’s Itanium. The unique features of HP-UX include a built-in logical volume manager for large file systems and access control lists as an alternative to the standard rwx file permissions of Unix.

1.8.4 IBM Unix

AIX (IBM AIX 2017) is a series of proprietary Unix operating systems developed by IBM for several of its computer platforms. Originally released for the IBM 6150 RISC workstation, AIX now supports or has supported a wide variety of hardware platforms, including the IBM RS/6000 series and later POWER and PowerPC-based systems, IBM System I, System/370 mainframes, PS/2 personal computers, and the Apple Network Server. AIX is based on UNIX System V with BSD4.3-compatible extensions.

1.8.5 Sun Unix

Solaris (Solaris 2017) is a Unix operating system originally developed by Sun Microsystems (Sun OS 2017). Since January 2010, it was renamed Oracle Solaris. Solaris is known for its scalability, especially on SPARC systems. Solaris supports SPARC-based and x86-based workstations and servers from Oracle and other vendors.

As can be seen, most Unix systems are proprietary and tied to specific hardware platforms. An average person may not have access to these systems. This present a challenge to readers who wishes to practice systems programming in the Unix environment. For this reason, we shall use Linux as the platform for programming exercises and practice.

1.9 Introduction to Linux

Linux (Linux 2017) is a Unix-like system. It started as an experimental kernel for the Intel x86 based PCs by Linus Torvalds in 1991. Subsequently, it was developed by a group of people world-wide. A big milestone of Linux occurred in the late 90s when it combined with GNU (Stallman 2017) by incorporating many GNU software, such as the GCC compiler, GNU Emacs editor and bash, etc. which greatly facilitated the further development of Linux. Not long after that, Linux implemented the TCP/IP suite to access the Internet and ported X11 (X-windows) to support GUI, making it a complete OS.

Linux includes many features of other Unix systems. In some sense, it represents a union of the most popular Unix systems. To a large extent, Linux is POSIX compliant. Linux has been ported to many hardware architectures, such as Motorola, SPARC and ARM, etc. The predominate Linux platform is still the Intel x86 based PCs, including both desktops and laptops, which are widely available. Also, Linux is free and easy to install, which makes it popular with computer science students.

1.10 Linux Versions

The development of Linux kernel is under the strict supervision of the Linux kernel development group. All Linux kernels are the same except for different release versions. However, depending on distributions, Linux has many different versions, which may differ in distribution packages, user interface and service functions. The following lists some of the most popular versions of Linux distributions.

1.10.1 Debian Linux

Debian is a Linux distribution that emphasizes on free software. It supports many hardware platforms. Debian distributions use the .deb package format and the dpkg package manager and its front ends.

1.10.2 Ubuntu Linux

Ubuntu is a Linux distribution based on Debian. Ubuntu is designed to have regular releases, a consistent user experience and commercial support on both desktops and servers. Ubuntu Linux has several official distributions. These Ubuntu variants simply install a set of packages different from the original Ubuntu. Since they draw additional packages and updates from the same repositories as Ubuntu, the same set of software is available in all of them.

1.10.3 Linux Mint

Linux Mint is a community-driven Linux distribution based on Debian and Ubuntu. According to Linux Mint, it strives to be a “modern, elegant and comfortable operating system which is both powerful and easy to use”. Linux Mint provides full out-of-the-box multimedia support by including

some proprietary software, and it comes bundled with a variety of free and open-source applications. For this reason, it was welcomed by many beginner Linux users.

1.10.4 RPM-Based Linux

Red Hat Linux and SUSE Linux were the original major distributions that used the RPM file format, which is still used in several package management systems. Both Red Hat and SUSE Linux divided into commercial and community-supported distributions. For example, Red Hat Linux provides a community-supported distribution sponsored by Red Hat called Fedora, and a commercially supported distribution called Red Hat Enterprise Linux, whereas SUSE divided into openSUSE and SUSE Linux Enterprise

1.10.5 Slackware Linux

Slackware Linux is known as a highly customizable distribution that stresses on ease of maintenance and reliability over cutting-edge software and automated tools. Slackware Linux is considered as a distribution for advanced Linux users. It allows users to choose Linux system components to install and configure the installed system, allowing them to learn the inner workings of the Linux operating system.

1.11 Linux Hardware Platforms

Linux was originally designed for the Intel x86 based PCs. Earlier versions of Linux run on Intel x86 based PCs in 32-bit protected mode. It is now available in both 32-bit and 64-bit modes. In addition to the Intel x86, Linux has bee ported to many other computer architectures, which include MC6800 of Motorola, MIP, SPARC, PowerPC and recently ARM. But the predominant hardware platform of Linux is still the Intel x86 based PCs, although ARM based Linux for embedded systems are gaining popularity rapidly.

1.12 Linux on Virtual Machines

Presently, most Intel x86 based PCs are equipped with Microsoft Windows, e.g. Windows 7, 8 or 10, as the default operating system. It is fairly easy to install Linux alongside Windows on the same PC and use dual-boot to boot up either Windows or Linux when the PC starts. However, most users are reluctant to do so due to either technical difficulty or preference to stay in the Windows environment. A common practice is to install and run Linux on a virtual machine inside the Windows host. In the following, we shall show how to install and run Linux on virtual machines inside the Microsoft Windows 10.

1.12.1 VirtualBox

VirtualBox (VirtualBox 2017) is a powerful x86 and AMD64/Intel64 virtualization product of Oracle. VirtualBox runs on Windows, Linux, Macintosh, and Solaris hosts. It supports a large number of guest

operating systems, including Windows (NT 4.0, 2000, XP, Vista, Windows 7, Windows 8, Windows 10) and Linux (2.4, 2.6, 3.x and 4.x), Solaris and OpenSolaris, OS/2, and OpenBSD. To install virtualBox on Windows 10, follow these steps.

(1). Download VirtualBox.

At the VirtualBox website, http://download.virtualbox.org, you will find links to VirtualBox binaries and its source code. For Windows hosts, the VirtualBox binary is

VirtualBox-5.1.12-112440-win.exe

In addition, you should also download the

VirtualBox 5.1.12 Oracle VM VirtualBox Extension Pack

which provides support for USB 2.0 and USB 3.0 devices, VirtualBox RDP, disk encryption, NVMe and PXE boot for Intel cards.

(2). Install VirtualBox

After downloading the VirtualBox-5.1.12-112440-win.exe file, double click on the file name to run it, which will install the VirtualBox VM under Windows 10. It also creates an Oracle VM VirtualBox icon on the desktop.

(3). Create a VirtualBox Virtual Machine

Start up the VirtualBox. An initial VM window will appear, as shown in Fig. 1.1. Choose the New button to create a new VM with 1024 MB memory and a virtual disk of 12GB.

(4). Install Ubuntu 14.04 to VirtualBox VM

Insert the Ubuntu 14.04 install DVD to a DVD drive. Start up the VM, which will boot up from the DVD to install Ubuntu 14.04 to the VM.

(5). Adjust Display Size

For some unknown reason, when Ubuntu first starts up, the screen size will be stuck at the 640 480 resolution. To change the display resolution, open a terminal and enter the command line xdiagnose. On the X Diagnostic settings window, enable all the options under Debug, which consist of

Extra graphic debug message
Display boot messages
Enable automatic crash bug reporting

Although none of these options seems to be related to the screen resolution, it does change the resolution to 1024 768 for a normal Ubuntu display screen. Figure 1.2 shows the screen of Ubuntu on the VirtualBox VM.

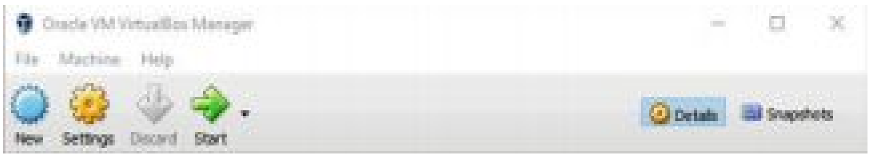
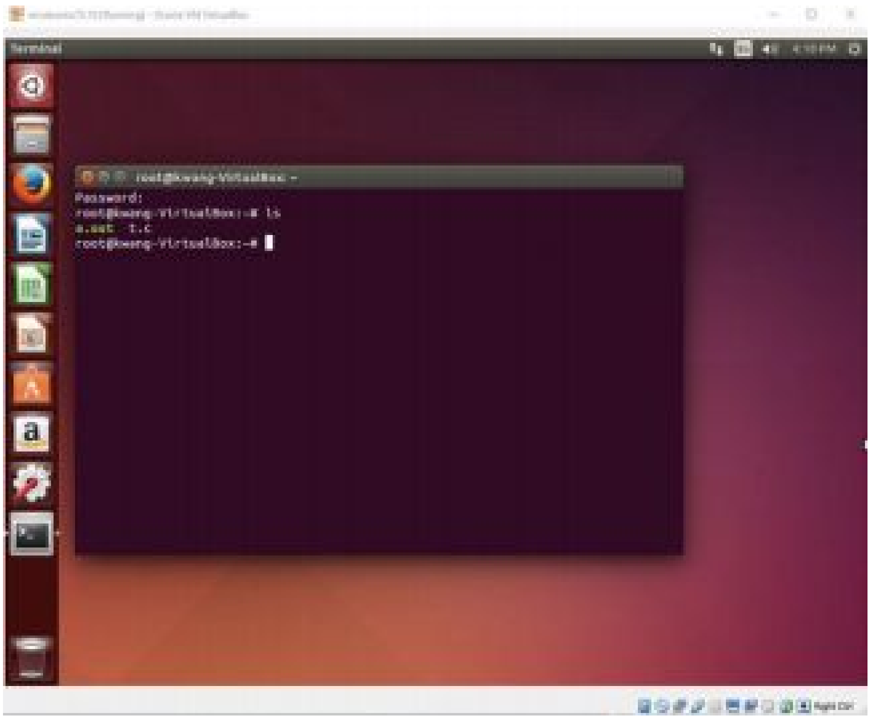


Fig. 1.1 VirtualBox VM Window



```
Password:
root@kwang-VirtualBox:~# ls
a.out  t.c
root@kwang-VirtualBox:~#
```

Fig. 1.2 Ubuntu Linux on VirtualBox VM

(6). Test C programming under Ubuntu

Ubuntu 14.04 has the gcc package installed. After installing Ubuntu, the user may create C source files, compile them and run the C programs. If the user needs emacs, install it by

```
sudo apt-get install emacs
```

Emacs provides an Integrated Development Environment (IDE) for text-edit, compile C programs and run the resulting binary executables under GDB. We shall cover and demonstrate the emacs IDE in Chap. 2.

1.12.2 VMware

VMware is another popular VM for x86 based PCs. The full versions of VMware, which include VM servers, are not free, but VMware Workstation Players, which are sufficient for simple VM tasks, are free.

(1). Install VMware Player on Windows 10

The reader may get VMware Workstation Player from VMware's download site. After downloading, double click on the file name to run it, which will install VMware and create a

VMware Workstation icon on the Desktop. Click on the icon to start up the VMware VM, which displays a VMware VM window, as shown in Fig. 1.3.

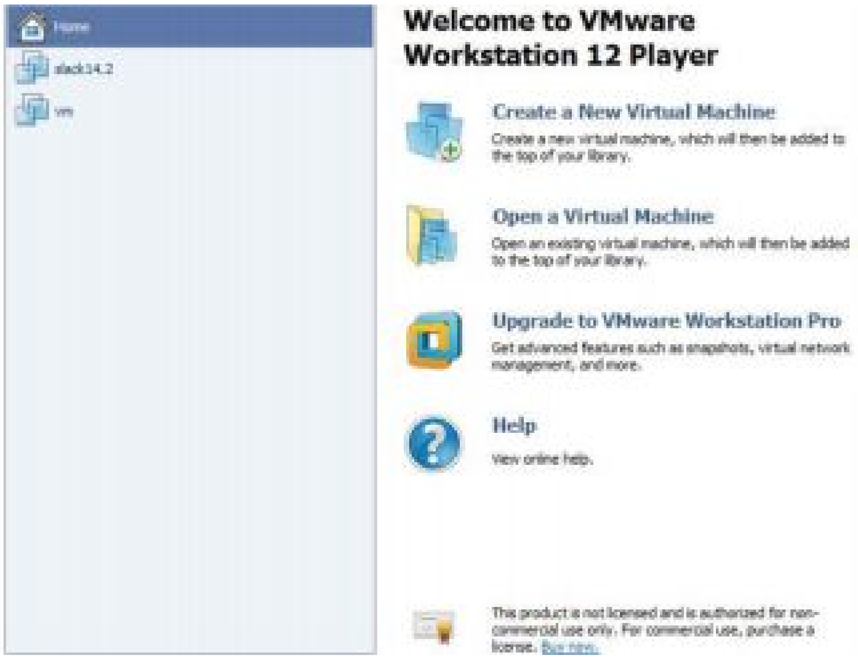

Home
slack14.2
vm
Welcome to VMware Workstation 12 Player
Create a New Virtual Machine
Create a new virtual machine, which will then be added to the top of your library.
Open a Virtual Machine
Open an existing virtual machine, which will then be added to the top of your library.
Upgrade to VMware Workstation Pro
Get advanced features such as snapshots, virtual network management, and more.
Help
View online help.
This product is not licensed and is authorized for non-commercial use only. For commercial use, purchase a license. Buy now.

Fig. 1.3 VMware VM Window

(2). Install Ubuntu 15.10 on VMware VM

To install Ubuntu 15.10 on the VMware VM, follow the following steps.

1. Download Ubuntu 15.10 install DVD image; burn it to a DVD disc.
2. Download Vmware Workstation Player 12 exe file for Windows 10.
3. Install Vmware Player.
4. Start up Vmware Player

   Choose: Create a new virtual machine;

   Choose: Installer disc: DVD RW Drive (D:)

   ¼> insert the installer disc until it is ready to install

   Then, enter Next

   Choose: Linux

   Version: ubuntu

   Virtual machine name: change to a suitable name, e.g. ubuntu

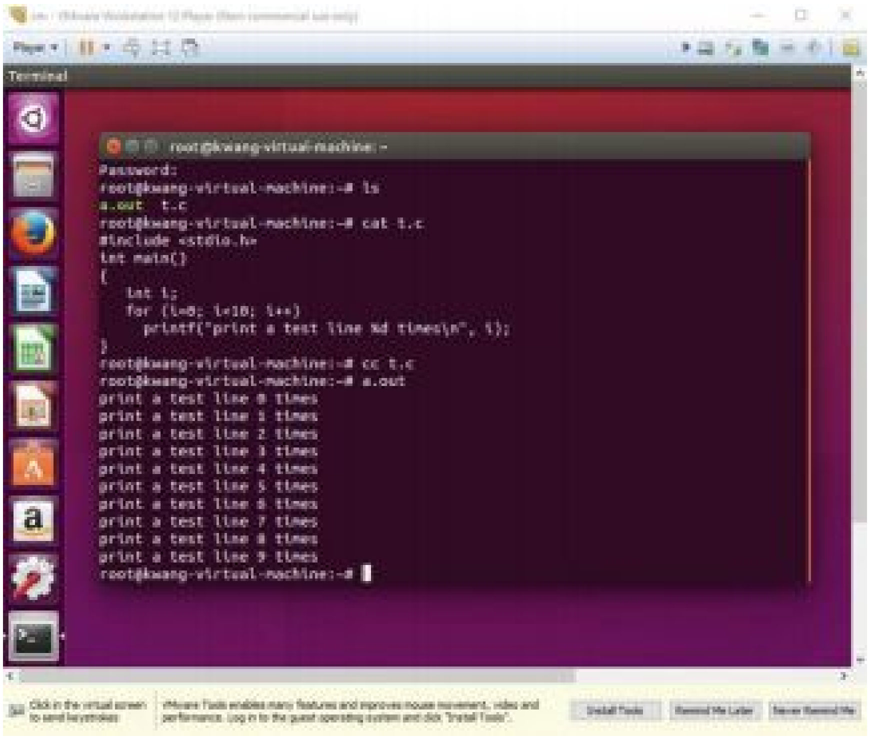
   Vmware will create a VM with 20GB disk size, 1GB memory, etc.

   Choose Finish to finish creating the new VM

   Next Screen: Choose: play virtual machine to start the VM.

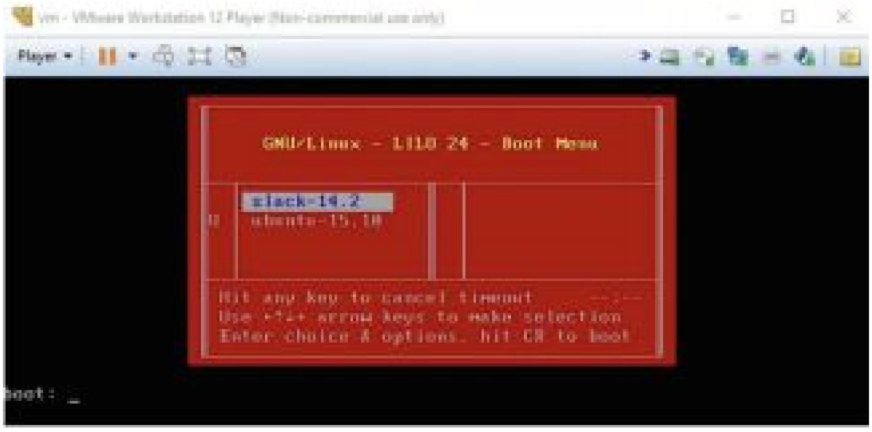
   The VM will boot from the Ubuntu install DVD to install Ubuntu.
5. Run C Program under Ubuntu Linux

Figure 1.4 shows the startup screen of Ubuntu and running a C program under Ubuntu.



```
root@kwang-virtual-machine: ~
Password:
root@kwang-virtual-machine:~# ls
a.out  t.c
root@kwang-virtual-machine:~# cat t.c
#include <stdio.h>
int main()
{
   int i;
   for (i=0; i<10; i++)
      printf("print a test line %d times\n", i);
}
root@kwang-virtual-machine:~# cc t.c
root@kwang-virtual-machine:~# a.out
print a test line 0 times
print a test line 1 times
print a test line 2 times
print a test line 3 times
print a test line 4 times
print a test line 5 times
print a test line 6 times
print a test line 7 times
print a test line 8 times
print a test line 9 times
root@kwang-virtual-machine:~#
```

Fig. 1.4 Ubuntu on VMware VM



```
GNU/Linux - LILO 24 - Boot Menu

slack-14.2
ubuntu-15.10

Hit any key to cancel timeout
Use ←↑↓→ arrow keys to make selection
Enter choice & options, hit CR to boot

boot: _
```

Fig. 1.5 Dual-Boot Window of Slackware and Ubuntu Linux

1.12.3 Dual Boot Slackware and Ubuntu Linux

It seems that free VMware Player only supports 32-bit VMs. The install steps are identical to above except the installer disc is the Slackware14.2 install DVD. After installing both Slackware 14.2 and Ubuntu 15.10, set up LILO to boot up either system when the VM starts up. If the reader installs Slackware first followed by installing Ubuntu, Ubuntu will recognize Slackware and configure GRUB for dual boot. The following figure shows the dual-boot menu of LILO (Fig. 1.5).

1.13 Use Linux

1.13.1 Linux kernel image

In a typical Linux system, Linux kernel images are in the /boot directory. Bootable Linux kernel images are named as

vmlinuz-generic-VERSION_NUMBER
initrd as the initial ramdisk image for the Linux kernel.

A bootable Linux kernel image is composed of three pieces:

| BOOT | SETUP | linux kernel |

BOOT is a 512-byte booter for booting early versions of Linux from floppy disk images. It is no longer used for Linux booting but it contains some parameters for SETUP to use. SETUP is a piece of 16-bit and 32-bit assembly code, which provides transition from the 16-bit mode to 32-bit protected mode during booting. Linux kernel is the actual kernel image of Linux. It is in compressed form but it has a piece of decompressing code at beginning, which decompresses the Linux kernel image and relocates it to high memory.

1.13.2 Linux Booters

The Linux kernel can be booted up by several different boot-loaders. The most popular Linux boot loaders are GRUB and LILO. Alternatively, the HD booter of (Wang 2015) can also be used to boot up Linux.

1.13.3 Linux Booting

During booting, the Linux boot-loader first locates the Linux kernel image (file). Then it loads

BOOT+SETUP to 0 90000 in real mode memory
Linux kernel to 1MB in high memory.

For generic Linux kernel images, it also loads an initial ramdisk image, initrd, to high memory. Then it transfers control to run SETUP code at 0 902000, which starts up the Linux kernel. When the Linux kernel first starts up, it runs on initrd as a temporary root file system. The Linux kernel executes a sh script, which directs the kernel to load the needed modules for the real root device. When the real root device is activated and ready, the kernel abandons the initial ramdisk and mounts the real root device as the root file system, thus completing a two-phase booting of the Linux kernel.

1.13.4 Linux Run-levels

The Linux kernel starts up in the single user mode. It mimics the run-levels of System V Unix to run in multi-user mode. Then it creates and run the INIT process P1, which creates the various daemon processes and also terminal processes for users to login. Then the INIT process waits for any child process to terminate.

1.13.5 Login Process

Each login process opens three file streams, stdin for input, stdout for output and stderr for error output, on its terminal. Then it waits for users to login. On Linux systems using X-windows as user interface, the X-window server usually acts as an interface for users to login. After a user login, the (pseudo) terminals belong to the user by default.

1.13.6 Command Executions

After login, the user process typically executes the command interpreter sh, which prompts the user for commands to execute. Some special commands, such as cd (change directory), exit, logout, &, are performed by sh directly. Non-special commands are usually executable files. For a non-special command, sh forks a child process and waits for the child to terminate. The child process changes its execution image to the file and executes the new image. When the child process terminates, it wakes up the parent sh, which prompts for another command, etc. In addition to simple commands, sh also supports I/O redirections and compound commands connected by pipes. In addition to built-in commands, the user may develop programs, compile-link them into binary executable files and run the programs as commands.

1.14 Use Ubuntu Linux

1.14.1 Ubuntu Versions

Among the different versions of Linux, we recommend Ubuntu Linux 15.10 or later for the following reasons.

(1) It is very easy to install. It can be installed online if the user has connection to the Internet. (2) It is very easy to install additional software packages by

```
sudo apt-get install PACKAGE
```

(3) It is updated and improved regularly with new releases.
(4) It has a large user base, with many problems and solutions posted in discussion forums online. (5) It provides easy connections to wireless networks and access to the Internet.

1.14.2 Special Features of Ubuntu Linux

Here are some helps on how to use the Ubuntu Linux.

(1) When installing Ubuntu on a desktop or laptop computer, it will ask for a user name and a password to create a user account with a default home directory /home/username. When Ubuntu boots up, it immediately runs in the environment of the user because it already has the default user logged in automatically. Enter Control-Alter-T to open a pseudo-terminal. Right click the Term icon and choose “lock to launcher” to lock the Term icon in the menu bar. Subsequently, launch new terminals by choosing the terminal->new terminal on the menu bar. Each new terminal runs a sh for the user to execute commands.
(2) For security reasons, the user is an ordinary user, not the root or superuser. In order to run any privileged commands, the user must enter

```
sudo command
```

which will verify the user's password first.

(3) The user's PATH environment variable setting usually does not include the user's current directory. In order to run programs in the current directory, the user must enter ./a.out every time. For convenience, the users should change the PATH setting to include the current directory. In the user's home directory, create a .bashrc file containing

```
PATH=$PATH:./
```

Every time the user opens a pseudo-terminal, sh will execute the .bashrc file first to set PATH to include the current working directory ./

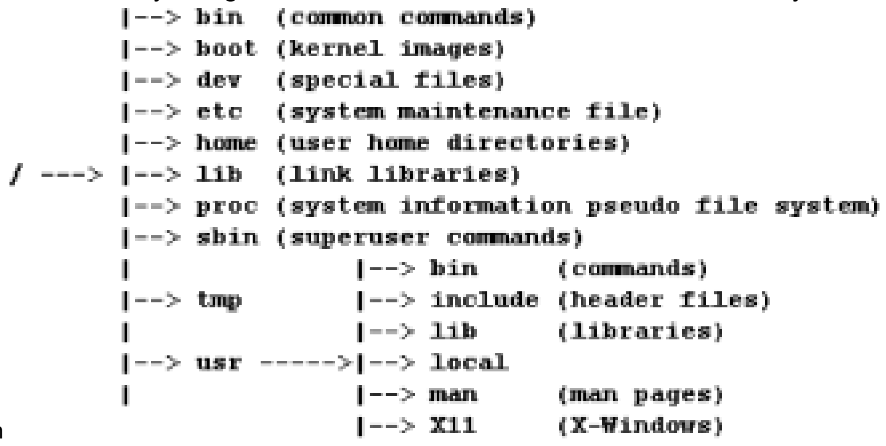
(4) Many users may have installed 64-bit Ubuntu Linux. Some of the programming exercises and assignments in this book are intended for 32-bit machines. In 64-bit Linux, use

```
gcc -m32 t.c # compile t.c into 32-bit code
```

to generate 32-bit code. If the 64-bit Linux does not take the -m32 option, the user must install additional support for gcc to generate 32-bit code.

(5) Ubuntu provides an excellent GUI user interface. Many users are so accustomed to the GUI that they tend to rely on it too much, often wasting time by repeatedly dragging and clicking the pointing device. In systems programming, the user should also learn how to use command lines and sh scripts, which are much more general and powerful than GUI.

(6) Nowadays, most users can connect to computer networks. Ubuntu supports both wired and wireless connections to networks. When Ubuntu runs on a PC/laptop with wireless hardware, it displays a wireless icon on top and it allows wireless connections by a simple user interface. Open the wireless icon. It will show a list of available wireless networks near by. Select a network and open the Edit Connections submenu to edit the connection file by entering the required login name and password. Close the Edit submenu. Ubuntu will try to login to the selected wireless network automatically.



```
         |--> bin  (common commands)
         |--> boot (kernel images)
         |--> dev  (special files)
         |--> etc  (system maintenance file)
         |--> home (user home directories)
/ ---> |--> lib  (link libraries)
         |--> proc (system information pseudo file system)
         |--> sbin (superuser commands)
         |               |--> bin     (commands)
         |--> tmp        |--> include (header files)
         |               |--> lib     (libraries)
         |--> usr ------>|--> local
         |               |--> man     (man pages)
                         |--> X11     (X-Windows)
```

Fig. 1.6 Unix/Linux File System Tree

1.15 Unix/Linux File System Organization

The Unix/Linux file system is organized as a tree, which is shown (sideway) in Fig. 1.6. Unix/Linux considers everything that can store or provide information as a file. In a general sense, each node of the file system tree is a FILE. In Unix/Linux, files have the following types.

1.15.1 File Types

(1). Directory files: A directory may contain other directories and (non-directory) files.

(2). Non-directory files: Non-directory files are either REGULAR or SPECIAL files, which can only be leaf-nodes in the file system tree. Non-directory files can be classified further as

(2).1 REGULAR files: Regular files are also called ORDINARY files. They contain either ordinary text or executable binary code.

(2).2 SPECIAL files: Special files are entries in the /dev directory. They represent I/O devices, which are further classified as

CHAR special files: I/O by chars, e.g. /dev/tty0, /dev/pts/1, etc.
BLOCK special files: I/O by blocks, e.g. /dev/had, /dev/sda, etc.
Other types such as network (socket) special files, named pipes, etc.

(3). Symbolic LINK files: These are Regular files whose contents are pathnames of other files. As such, they act as pointers to other files. As an example, the Linux command

```
ln -s aVeryLongFileName myLink
```

creates a symbolic link file, mylink, which points to aVeryLongFileName. Access to myLink will be redirected to the actual file aVeryLongFileName.

1.15.2 File Pathnames

The root node of a Unix/Linux file system tree, symbolized by /, is called the root directory or simply the root. Each node of the file system tree is specified by a pathname of the form

```
/a/b/c/d OR a/b/c/d
```

A pathname is ABSOLUTE if it begins with a /. Otherwise, it is RELATIVE to the Current Working Directory (CWD) of the process. When a user login to Unix/Linux, the CWD is set to the user's HOME directory. The CWD can be changed by the cd (change directory) command. The pwd command prints the absolute pathname of the CWD.

1.15.3 Unix/Linux Commands

When using an operating system, the user must learn how to use the system commands. The following lists the most often used commands in Unix/Linux.

ls: ls dirname: list the contents of CWD or a directory
cd dirname: change directory
pwd: print absolute pathname of CWD
touch filename: change filename timestamp (create file if it does not exist)
cat filename: display file contents
cp src dest: copy files
mv src dest: move or rename files
mkdir dirname: create directory
rmdir dirname: remove (empty) directory
rm filename: remove or delete file
ln oldfile newfile: create links between files
find: search for files
grep: search file for lines containing a pattern
ssh: login to remote hosts
gzip filename: compress filename to .gz file
gunzip file.gz: uncompress .gz file
tar –zcvf file.tgz . : create compressed tar file from current directory
tar -zxvf file.tgz . : extract files from .tgz file
man: display online manual pages
zip file.zip filenames : compress files to .zip file
unzip file.zip : uncompress .zip file

1.15.4 Linux Man Pages

Linux maintains online man (manual) pages in the standard /usr/man/directory. In Ubuntu Linux, it is in /usr/share/man directory. The man pages are organized into several different categories, denoted by man1, man2, etc.

```
/usr/man/
          |-- man1: commonly used commands: ls, cat, mkdir ...., etc.
          |-- man2: system calls
          |-- man3: library functions: strtok, strcat, basename, dirname etc.
```

All the man pages are compressed .gz files. They contain text describing how to use the command with input parameters and options. man is a program, which reads man page files and displays their contents in a user friendly format. Here are some examples of using man pages.

man ls : show man page of ls in man1
man 2 open : show man page of open in man2
man strtok : show man page of strtok in man 3, etc.
man 3 dirname : show dirname in man3, NOT that of man1

Whenever needed, the reader should consult the man pages for how to use a specific Linux command. Many of the so called Unix/Linux systems programming books are essentially condensed versions of the Unix/Linux man pages.

1.16 Ubuntu Linux System Administration

1.16.1 User Accounts

As in all Linux, user accounts are maintained in the /etc/passwd file, which is owned by the superuser but readable by anyone. Each user has a line in the /etc/passwd file of the form

```
loginName:x:gid:uid:usefInfo:homeDir:initialProgram
```

where the second field x indicates checking user password. Encrypted user passwords are maintained in the /etc/shadow file. Each line of the shadow file contains the encrypted user password, followed by optional aging limit information, such as expiration date and time, etc. When a user tries to login with a login name and password, Linux will check both the /etc/passwd and /etc/shadow files to authenticate the user. After a user login successfully, the login process becomes the user process by acquiring the user's gid and uid, changes directory to the user's homeDir and executes the listed initialProgram, which is usually the command interpreter sh.

1.16.2 Add New User

This may be a pathological case for most users who run Ubuntu Linux on their personal PCs or laptops. But let's assume that the reader may want to add a family member to use the same computer but as a different user. As in all Linux, Ubuntu supports an adduser command, which can be run as

```
sudo adduer username
```

It adds a new user by creating an account and also a default home directory /home/username for the new user. Henceforth, Ubuntu will display a list of user names in its "About The Computer" menu. The new user may login to the system by selecting the new username.

1.16.3 The sudo Command

For security reasons, the root or superuser account is disabled in Ubuntu, which prevents anyone from login as the root user (well, not quite; there is a way but I won't disclose it). sudo ("superuser do") allows a user to execute a command as another user, usually the superuser. It temporarily elevates the user process to the superuser privilege while executing a command. When the command execution finishes, the user process reverts back to its original privilege level. In order to be able to use sudo, the user's name must be in the /etc/sudoers file. To allow a user to issue sudo, simply add a line to sudoers files, as in

```
username ALL(ALL) ALL
```

However, the /etc/sudoers file has a very rigid format. Any syntax error in the file could breech the system security. Linux recommends editing the file only by the special command visudo, which invokes the vi editor but with checking and validation.

1.17 Summary

This chapter presents an introduction to the book. It describes the book's scope, intended audience and its suitability as textbook in Computer Science/Engineering curriculums. It presents a brief history of Unix, which includes early versions of Unix at Bell Labs, AT&T System V and other developments of Unix, such as BSD, HP UX, IBM AIX and Sun/Solaris Unix. It describes the development of Linux and various Linux distributions, which include Debian, Ubuntu, Mint, Red Hat and Slackware. It lists both the hardware platforms and virtual machines for Linux. It shows how to install Ubuntu Linux to both VirtualBox and Vmware virtual machines inside the Microsoft Windows. It explains the startup sequence of Linux, from booting the Linux kernel to user login and command execution. It describes the Linux file system organization, file types and commonly used Unix/Linux commands. Lastly, it describes some system administration tasks for users to manage and maintain their Linux systems.

References

Bach M., "The Design of the UNIX Operating System", Prentice-Hall, 1986
Curry, David A., Unix Systems Programming for SRV4, O'Reilly, 1996.
Haviland, Keith, Gray, Dian, Salama, Ben, Unix System Programming, Second Edition, Addison-Wesley, 1998. HP-UX, http://www.operating-system.org/betriebssystem/_english/bs-hpux.htm, 2017
IBM AIX, https://www.ibm.com/power/operating-systems/aix, 2017
Kernighan, B.W., Ritchie D.M, "The C Programming Language" 2nd Edition, 1988
Kerrisk, Michael, The Linux Programming Interface: A Linux and UNIX System Programming Handbook, 1st Edition, No Starch Press Inc, 2010.
Leffler, S.J., McKusick, M. K, Karels, M. J. Quarterman, J., "The Design and Implementation of the 4.3BSD UNIX Operating System", Addison Wesley, 1989
Leffler, S.J., McKusick, M. K, Karels, M. J. Quarterman, J., "The Design and Implementation of the 4.4BSD UNIX Operating System", Addison Wesley, 1996

Linux, https://en.wikipedia.org/wiki/Linux , 2017
Love, Robert, Linux System Programming: Talking Directly to the Kernel and C Library, 2nd Edition, O'Reilly, 2013. McKusick, M. K, Karels, Neville-Neil, G.V., "The Design and Implementation of the FreeBSD Operating System", Addison Wesley, 2004
Robbins Kay, Robbins, Steve, UNIX Systems Programming: Communication, Concurrency and Threads: Communica tion, Concurrency and Threads, 2nd Edition, Prentice Hall, 2003.
Pthreads: https://computing.llnl.gov/tutorials/pthreads, 2017
Rochkind, Marc J., Advanced UNIX Programming, 2nd Edition, Addison-Wesley, 2008.
Stallman, R., Linux and the GNU System, 2017
Stevens, W. Richard, Rago, Stephan A., Advanced Programming in the UNIX Environment, 3rd Edition, Addison Wesley, 2013.
Solaris, http://www.operating-system.org/betriebssystem/_english/bs-solaris.htm, 2017
Sun OS, https://en.wikipedia.org/wiki/SunOS, 2017
Thompson, K., Ritchie, D. M., "The UNIX Time-Sharing System", CACM., Vol. 17, No.7, pp. 365-375, 1974.
Thompson, K., Ritchie, D.M.; "The UNIX Time-Sharing System", Bell System Tech. J. Vol. 57, No.6, pp. 1905–1929, 1978
Unix System V, https://en.wikipedia.org/wiki/UNIX_System V, 2017

VirtualBox, https://www.virtualbox.org, 2017
Wang, K. C., Design and Implementation of the MTX Operating System, Springer International Publishing AG, 2015

Programming Background 2

Abstract
This chapter covers the background information needed for systems programming. It introduces several GUI based text editors, such as vim, gedit and EMACS, to allow readers to edit files. It shows how to use the EMACS editor in both command and GUI mode to edit, compile and execute C programs. It explains program development steps. These include the compile-link steps of GCC, static and dynamic linking, format and contents of binary executable files, program execution and termination. It explains function call conventions and run-time stack usage in detail. These include parameter passing, local variables and stack frames. It also shows how to link C programs with assembly code. It covers the GNU make facility and shows how to write Makefiles by examples. It shows how to use the GDB debugger to debug C programs. It points out the common errors in C programs and suggests ways to prevent such errors during program development. Then it covers advanced programming techniques. It describes structures and pointer in C. It covers link lists and list processing by detailed examples. It covers binary trees and tree traversal algorithms. The chapter cumulates with a programming project, which is for the reader to implement a binary tree to simulate operations in the Unix/Linux file system tree. The project starts with a single root directory node. It supports mkdir, rmdir, creat, rm, cd, pwd, ls operations, saving the file system tree as a file and restoring the file system tree from saved file. The project allows the reader to apply and practice the programming techniques of tokenizing strings, parsing user commands and using function pointers to invoke functions for command processing.

2.1 Text Editors in Linux

2.1.1 Vim

Vim (Linux Vi and Vim Editor 2017) is the standard built-in editor of Linux. It is an improved version of the original default Vi editor of Unix. Unlike most other editors, vim has 3 different operating modes, which are

. Command mode: for entering commands
. Insert mode: for entering and editing text
. Last-line mode: for saving files and exit

When vim starts, it is in the default Command mode, in which most keys denote special commands. Examples of command keys for moving the cursor are

h: move cursor one char to the left; l: move cursor one char to the right j: move cursor down one line; k: move cursor up one line

When using vim inside X-window, cursor movements can also be done by arrow keys. To enter text for editing, the user must switch vim into Insert mode by entering either the i (insert) or a (append) command:

i: switch to Insert mode to insert text;
a: switch to Insert mode to append text

To exit Insert mode, press the ESC key one or more times. While in Command mode, enter the : key to enter the Last-line mode, which is for saving texts as files or exit vim:

:w write (save) file
:q exit vim
:wq save and exit
:q! force exit without saving changes

Although many Unix users are used to the different operating modes of vim, others may find it somewhat unnatural and inconvenient to use as compared with other Graphic User Interface (GUI) based editors. The following editors belong to what is commonly known as What You See Is What You Get (WYSIWYG) type of editors. In a WYSIWYG editor, a user may enter text, move the cursor by arrow keys, just like in regular typing. Commands are usually formed by entering a special meta key, together with, or followed by, a letter key sequence. For example,

Control-C: abort or exit,
Control-K: delete line into a buffer,
Control-Y: yank or paste from buffer contents
Control-S: save edited text, etc.

Since there is no mode switching necessary, most users, especially beginners, prefer WYSUWYG type editors over vim.

2.1.2 Gedit

Gedit is the default text editor of the GNOME desktop environment. It is the default editor of

Ubuntu, as well as other Linux that uses the GNOME GUI user interface. It includes tools for editing both source code and structured text such as markup languages.

2.1.3 Emacs

Emacs (GNU Emacs 2015) is a powerful text editor, which runs on many different platforms. The most popular version of Emacs is GNU Emacs, which is available in most Linux distributions.

All the above editors support direct inputs and editing of text in full screen mode. They also support search by keywords and replace strings by new text. To use these editors, the user only needs to learn a few basics, such as how to start up the editor, input text for editing, save edited texts as files and then exit the editor.

Depending on the Unix/Linux distribution, some of the editors may not be installed by default. For example, Ubuntu Linux usually comes with gedit, nano and vim, but not emacs. One nice feature of Ubuntu is that, when a user tries to run a command that is not installed, it will remind the user to install it. As an example, if a user enters

```
emacs filename
```

Ubuntu will display a message saying “The program emacs is currently not installed. You can install it by typing apt-get install emacs”. Similarly, the user may install other missing software packages by the apt-get command.

2.2 Use Text Editors

All text editors are designed to perform the same task, which is to allow users to input texts, edit them and save the edited texts as files. As noted above, there are many different text editors. Which text editor to use is a matter of personal preference. Most Linux users seem to prefer either gedit or emacs due to their GUI interface and ease to use. Between the two, we strongly recommend emacs. The following shows some simple examples of using emacs to create text files.

2.2.1 Use Emacs

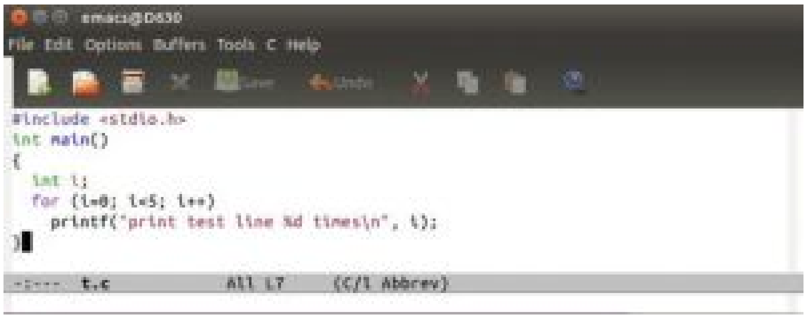
First, from a pseudo terminal of X-windows, enter the command line

```
emacs [FILENAME] # [ ] means optonal
```

to invoke the emacs editor with an optional file name, e.g. t.c. This will start up emacs in a separate window as show in Fig. 2.1. On top of the emacs window is a menu bar, each of which can be opened to show additional commands, which the user can invoke by clicking on the menu icons. To begin with, we shall not use the menu bar and focus on the simple task of creating a C program source file. When emacs starts, if the file t.c already exists, it will open the file and load its contents into a buffer for editing. Otherwise, it will show an empty buffer, ready for user inputs. Fig. 2.1 shows the user input lines. Emacs recognizes any .c file as source file for C programs. It will indent the lines in accordance with the C code line conventions, e.g. it will match each left { with a right }with proper indentations automatically. In fact, it can even detect incomplete C statements and show improper indentations to alert the user of possible syntax errors in the C source lines.

After creating a source file, enter the meta key sequence Control-x-c to save the file and exit. If the buffer contains modified and unsaved text, it will prompt the user to save file, as shown on the bottom line of Fig. 2.2. Entering y will save the file and exit emacs. Alternatively, the user may also click the Save icon on the menu bar to save and exit.

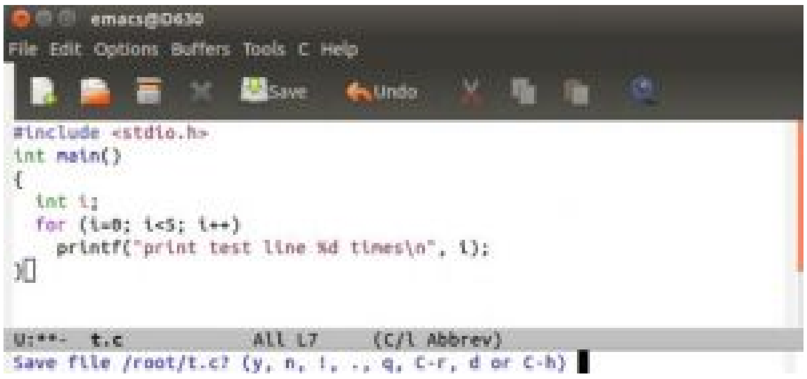
Fig. 2.1 Use emacs 1



```
#include <stdio.h>
int main()
{
  int i;
  for (i=0; i<5; i++)
    printf("print test line %d times\n", i);
}
```

```
-:---  t.c          All L7     (C/l Abbrev)
```

Fig. 2.2 Use emacs 2



```
#include <stdio.h>
int main()
{
  int i;
  for (i=0; i<5; i++)
    printf("print test line %d times\n", i);
}
```

```
U:**-  t.c          All L7     (C/l Abbrev)
Save file /root/t.c? (y, n, !, ., q, C-r, d or C-h)
```

2.2.2 Emacs Menus

At the top of the emacs window is a menu bar, which includes the icons

File Edit Options Buffers Tools C Help

The File menu supports the operations of open file, insert file and save files. It also supports printing the editing buffer, open new windows and new frames.
The Edit menu supports search and replace operations.
The Options menu supports functions to configure emacs operations.
The Buffers menu supports selection and display of buffers.
The Tools menu supports compilation of source code, execution of binary executables and debugging. The C menu supports customized editing for C source code.
The Help menu provides support for emacs usage, such as a simple emacs tutorial.

As usual, clicking on a menu item will display a table of submenus, allowing the user to choose individual operations. Instead of commands, the user may also use the emacs menu bar to do text editing, such as undo last operation, cut and past, save file and exit, etc.

2.2.3 IDE of Emacs

Emacs is more than a text editor. It provides an Integrated Development Environment (IDE) for software development, which include compile C programs, run executable images and debugging

program executions by GDB. We shall illustrate the IDE capability of emacs in the next section on program development.

2.3 Program Development

2.3.1 Program Development Steps

The steps of developing an executable program are as follows.

(1). Create source files: Use a text editor, such as gedit or emacs, to create one or more source files of a program. In systems programming, the most important programming languages are C and assembly. We begin with C programs first.

Standard comment lines in C comprises matched pairs of /* and */. In addition to the standard comments, we shall also use // to denote comment lines in C code for convenience. Assume that t1.c and t2.c are the source files of a C program.

```
/********************** t1.c file *****************************/ int g = 100; // initialized global
variable int h; // uninitialized global variable static int s; // static global variable

main(int argc, char *argv[ ]) // main function
{
   int a = 1; int b; // automatic local variables
   static int c = 3; // static local variable
   b = 2;
   c = mysum(a,b); // call mysum(), passing a, b
   printf("sum=%d\n", c); // call printf()
}
/********************** t2.c file ****************************/ extern int g; // extern global
variable
int mysum(int x, int y) // function heading
{
     return x + y + g;
}
```

2.3.2 Variables in C

Variables in C programs can be classified as global, local, static, automatic and registers, etc. as shown in Fig. 2.3.

Global variables are defined outside of any function. Local variables are defined inside functions. Global variables are unique and have only one copy. Static globals are visible only to the file in which they are defined. Non-static globals are visible to all the files of the same program. Global variables can be initialized or uninitialized. Initialized globals are assigned values at compile time. Uninitialized globals are cleared to 0 when the program execution starts. Local variables are visible only to the function in which they are defined. By default, local variables are automatic; they come into existence

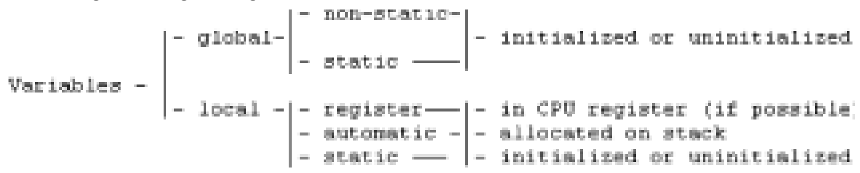


Fig. 2.3 Variables in C

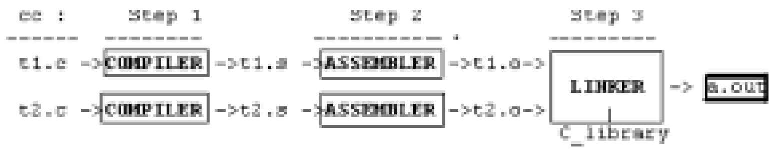


Fig. 2.4 Program development steps

when the function is entered and they logically disappear when the function exits. For register variables, the compiler tries to allocate them in CPU registers. Since automatic local variables do not have any allocated memory space until the function is entered, they cannot be initialized at compile time. Static local variables are permanent and unique, which can be initialized. In addition, C also supports volatile variables, which are used as memory-mapped I/O locations or global variables that are accessed by interrupt handlers or multiple execution threads. The volatile keyword prevents the C compiler from optimizing the code that operates on such variables.

In the above t1.c file, g is an initialized global, h is an uninitialized global and s is a static global. Both g and h are visible to the entire program but s is visible only in the t1.c file. So t2.c can reference g by declaring it as extern, but it cannot reference s because s is visible only in t1.c. In the main() function, the local variables a, b are automatic and c is static. Although the local variable a is defined as int a ¼ 1, this is not an initialization because a does not yet exist at compile time. The generated code will assign the value 1 to the current copy of a when main() is actually entered.

2.3.3 Compile-Link in GCC

(2). Use gcc to convert the source files into a binary executable, as in

```
gcc t1.c t2.c
```

which generates a binary executable file named a.out. In Linux, cc is linked to gcc, so they are the same.

(3). What's gcc? gcc is a program, which consists of three major steps, as shown in Fig. 2.4.

Step 1. Convert C source files to assembly code files: The first step of cc is to invoke the C COMPILER, which translates .c files into .s files containing assembly code of the target machine. The C compiler itself has several phases, such as preprocessing, lexical analysis, parsing and code generations, etc, but the reader may ignore such details here.

Step 2. Convert assembly Code to OBJECT code: Every computer has its own set of machine instructions. Users may write programs in an assembly language for a specific machine. An ASSEMBLER is a program, which translates assembly code into machine code in binary form. The resulting .o files are called OBJECT code. The second step of cc is to invoke the ASSEMBLER to translate .s files to .o files. Each .o file consists of

. a header containing sizes of CODE, DATA and BSS sections
. a CODE section containing machine instructions
. a DATA section containing initialized global and initialized static local variables . a BSS section containing uninitialized global and uninitialized static local variables . relocation information for pointers in CODE and offsets in DATA and BSS . a Symbol Table containing non-static globals, function names and their attributes.

Step 3: LINKING: A program may consist of several .o files, which are dependent on one another. In addition, the .o files may call C library functions, e.g. printf(), which are not present in the source files. The last step of cc is to invoke the LINKER, which combines all the .o files and the needed library functions into a single binary executable file. More specifically, the LINKER does the following:

. Combine all the CODE sections of the .o files into a single Code section. For C programs, the combined Code section begins with the default C startup code crt0.o, which calls main(). This is why every C program must have a unique main() function.

. Combine all the DATA sections into a single Data section. The combined Data section contains only initialized globals and initialized static locals.

. Combine all the BSS sections into a single bss section.

. Use the relocation information in the .o files to adjust pointers in the combined Code section and offsets in the combined Data and bss sections.

. Use the Symbol Tables to resolve cross references among the individual .o files. For instance, when the compiler sees c ¼ mysum(a, b) in t1.c, it does not know where mysum is. So it leaves a blank (0) in t1.o as the entry address of mysum but records in the symbol table that the blank must be replaced with the entry address of mysum. When the linker puts t1.o and t2.o together, it knows where mysum is in the combined Code section. It simply replaces the blank in t1.o with the entry address of mysum. Similarly for other cross referenced symbols. Since static globals are not in the symbol table, they are unavailable to the linker. Any attempt to reference static globals from different files will generate a cross reference error. Similarly, if the .o files refer to any undefined symbols or function names, the linker will also generate cross reference errors. If all the cross references can be resolved successfully, the linker writes the resulting combined file as a.out, which is the binary executable file.

2.3.4 Static vs. Dynamic Linking

There are two ways to create a binary executable, known as static linking and dynamic linking. In static linking, which uses a static library, the linker includes all the needed library function code and data into a.out. This makes a.out complete and self-contained but usually very large. In dynamic linking, which uses a shared library, the library functions are not included in a.out but calls to such functions are recorded in a.out as directives. When execute a dynamically linked a.out file, the operating system loads both a.out and the shared library into memory and makes the loaded library code accessible to a.out during execution. The main advantages of dynamic linking are:

. The size of every a.out is reduced.
. Many executing programs may share the same library functions in memory. . Modifying library functions does not need to re-compile the source files again.

Libraries used for dynamic linking are known as Dynamic Linking Libraries (DLLs). They are called Shared Libraries (.so files) in Linux. Dynamically loaded (DL) libraries are shared libraries which are loaded only when they are needed. DL libraries are useful as plug-ins and dynamically loaded modules.

2.3.5 Executable File Format

Although the default binary executable is named a.out, the actual file format may vary. Most C compilers and linkers can generate executable files in several different formats, which include

(1) Flat binary executable: A flat binary executable file consists only of executable code and initialized data. It is intended to be loaded into memory in its entirety for execution directly. For example, bootable operating system images are usually flat binary executables, which simplifies the boot-loader.
(2) a.out executable file: A traditional a.out file consists of a header, followed by code, data and bss sections. Details of the a.out file format will be shown in the next section.
(3) ELF executable file: An Executable and Linking Format (ELF) (Youngdale 1995) file consists of one or more program sections. Each program section can be loaded to a specific memory address. In Linux, the default binary executables are ELF files, which are better suited to dynamic linking.

2.3.6 Contents of a.out File

For the sake of simplicity, we consider the traditional a.out files first. ELF executables will be covered in later chapters. An a.out file consists of the following sections:

(1) header: the header contains loading information and sizes of the a.out file, where tsize ¼ size of Code section;
dsize ¼ size of Data section containing initialized globals and static locals;
bsize ¼ size of bss section containing uninitialized globals and static locals; total_size ¼ total size of a.out to load.
(2) Code Section: also called the Text section, which contains executable code of the program. It begins with the standard C startup code crt0.o, which calls main().
(3) Data Section: The Data section contains initialized global and static data. (4) Symbol table: optional, needed only for run-time debugging.

Note that the bss section, which contains uninitialized global and static local variables, is not in the a. out file. Only its size is recorded in the a.out file header. Also, automatic local variables are not in a.out. Figure 2.5 shows the layout of an a.out file.

In Fig. 2.5, _brk is a symbolic mark indicating the end of the bss section. The total loading size of a. out is usually equal to _brk, i.e. equal to tsize+dsize+bsize. If desired, _brk can be set to a higher value for a larger loading size. The extra memory space above the bss section is the HEAP area for dynamic memory allocation during execution.

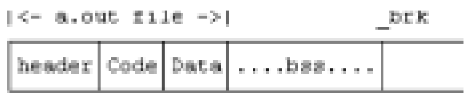


Fig. 2.5 Contents of a.out file

2.3.7 Program Execution

Under a Unix-like operating system, the sh command line

```
a.out one two three
```

executes a.out with the token strings as command-line parameters. To execute the command, sh forks a child process and waits for the child to terminate. When the child process runs, it uses a.out to create a new execution image by the following steps.

(1) Read the header of a.out to determine the total memory size needed, which includes the size of a stack space:

```
TotalSize = _brk + stackSize
```

where stackSize is usually a default value chosen by the OS kernel for the program to start. There is no way of knowing how much stack space a program will ever need. For example, the trivial C program

```
main(){ main(); }
```

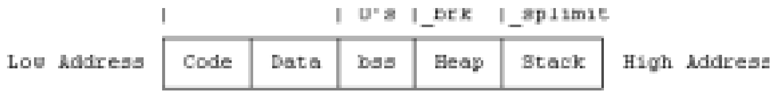
will generate a segmentation fault due to stack overflow on any computer. So the usual approach of an OS kernel is to use a default initial stack size for the program to start and tries to deal with possible stack overflow later during run-time.

(2) It allocates a memory area of TotalSize for the execution image. Conceptually, we may assume that the allocated memory area is a single piece of contiguous memory. It loads the Code and Data sections of a.out into the memory area, with the stack area at the high address end. It clears the bss section to 0, so that all uninitialized globals and static locals begin with the initial value 0. During execution, the stack grows downward toward low address.

(3) Then it abandons the old image and begins to execute the new image, which is shown in Fig. 2.6.

In Fig. 2.6, _brk at the end of the bss section is the program's initial "break" mark and _splimit is the stack size limit. The Heap area between bss and Stack is used by the C library functions malloc()/free() for dynamic memory allocation in the execution image. When a.out is first loaded, _brk and _splimit may coincide, so that the initial Heap size is

zero. During execution, the process may use the brk (address) or sbrk(size) system call to change _brk to a higher address, thereby increasing the Heap size. Alternatively, malloc() may call brk() or sbrk() implicitly to expand the Heap size. During execution, a stack overflow occurs if the program tries to extend the stack pointer below _splimit. On machines with memory protection, this will be detected by the memory management hardware as an error, which traps the process to the OS kernel. Subject to a maximal size limit, the OS kernel may grow the stack by allocating additional memory in the process address space, allowing the execution to continue. A stack
Fig. 2.6 Execution image



overflow is fatal if the stack cannot be grown any further. On machines without suitable hardware support, detecting and handling stack overflow must be implement in software.

(4). Execution begins from crt0.o, which calls main(), passing as parameters argc and argv to main(), which can be written as

```
int main( int argc, char *argv[ ] ) { ... . }
```

where argc ¼ number of command line parameters and each argv[ ] entry points to a corresponding command line parameter string.

2.3.8 Program Termination

A process executing a.out may terminate in two possible ways.

(1). Normal Termination: If the program executes successfully, main() eventually returns to crt0.o, which calls the library function exit(0) to terminate the process. The exit(value) function does some clean-up work first, such as flush stdout, close I/O streams, etc. Then it issues an _exit(value) system call, which causes the process to enter the OS kernel to terminate. A 0 exit value usually means normal termination. If desired, a process may call exit(value) directly without going back to crt0.o. Even more drastically, a process may issue an _exit(value) system call to terminate immediately without doing the clean-up work first. When a process terminates in kernel, it records the value in the _exit(value) system call as the exit status in the process structure, notifies its parent and becomes a ZOMBIE. The parent process can find the ZOMBIE child, get its pid and exit status by the system call

```
pid = wait(int *status);
```

which also releases the ZMOBIE child process structure as FREE, allowing it to be reused for another process.
(2). Abnormal Termination: While executing a.out the process may encounter an error condition, such as invalid address, illegal instruction, privilege violation, etc. which is recognized by the CPU as an exception. When a process encounters an exception, it is forced into the OS kernel by a trap. The kernel's trap handler converts the trap error type to a magic number, called a SIGNAL, and delivers the signal to the process, causing it to

terminate. In this case, the exit status of the ZOMBIE process is the signal number, and we may say that the process has terminated abnormally. In addition to trap errors, signals may also originate from hardware or from other processes. For example, pressing the Control_C key generates a hardware interrupt, which sends the number 2 signal SIGINT to all processes on that terminal, causing them to terminate. Alternatively, a user may use the command

```
kill -s signal_number pid # signal_number = 1 to 31
```

to send a signal to a target process identified by pid. For most signal numbers, the default action of a process is to terminate. Signals and signal handling will be covered later in Chap. 6.

2.4 Function Call in C

Next, we consider the run-time behavior of a.out during execution. The run-time behavior of a program stems mainly from function calls. The following discussions apply to running C programs on 32-bit Intel x86 processors. On these machines, the C compiler generated code passes parameters on the stack in function calls. During execution, it uses a special CPU register (ebp) to point at the stack frame of the current executing function.

2.4.1 Run-Time Stack Usage in 32-Bit GCC

Consider the following C program, which consists of a main() function shown on the left-hand side, which calls a sub() function shown on the right-hand side.

```
----------------------------------------------------------
  main() | int sub(int x, int y)
  { |{
     int a, b, c; | int u, v;
     a = 1; b = 2; c = 3; | u = 4; v = 5;
     c = sub(a, b); | return x+y+u+v;
     printf("c=%d\n", c); | }
  } |
----------------------------------------------------------
```

(1) When executing a.out, a process image is created in memory, which looks (logically) like the diagram shown in Fig. 2.7, where Data includes both initialized data and bss. (2) Every CPU has the following registers or equivalent, where the entries in parentheses denote registers of the x86 CPU:

PC (IP): point to next instruction to be executed by the CPU.
SP (SP): point to top of stack.
FP (BP): point to the stack frame of current active function.
Return Value Register (AX): register for function return value.

(3) In every C program, main() is called by the C startup code crt0.o. When crt0.o calls main(), it pushes the return address (the current PC register) onto stack and replaces PC with the entry address of main(), causing the CPU to enter main(). For convenience, we shall show the stack contents from left to right. When control enters main(), the stack contains the saved return PC on top, as shown in Fig. 2.8, in which XXX denotes the stack contents before crt0.o calls main(), and SP points to the saved return PC from

where crt0.o calls main().

(4) Upon entry, the compiled code of every C function does the following:
. push FP onto stack # this saves the CPU's FP register on stack.
. let FP point at the saved FP # establish stack frame
. shift SP downward to allocate space for automatic local variables on stack
. the compiled code may shift SP farther down to allocate some temporary working space on the stack, denoted by temps.

Fig. 2.7 Process execution image

Fig. 2.8 Stack contents in function call

Fig. 2.9 Stack contents: allocate local variables

Fig. 2.10 Stack contents: passing parameters

For this example, there are 3 automatic local variables, int a, b, c, each of sizeof(int) ¼ 4 bytes. After entering main(), the stack contents becomes as shown in Fig. 2.9, in which the spaces of a, b, c are allocated but their contents are yet undefined.

(5) Then the CPU starts to execute the code a¼1; b¼2; c¼3; which put the values 1, 2, 3 into the memory locations of a, b, c, respectively. Assume that sizeof(int) is 4 bytes. The locations of a, b, c are at -4, -8, -12 bytes from where FP points at. These are expressed as -4(FP), -8(FP), -12(FP) in assembly code, where FP is the stack frame pointer. For example, in 32-bit Linux the assembly code for b¼2 in C is

```
movl $2, -8(%ebp) # b=2 in C
```

where $2 means the value of 2 and %ebp is the ebp register.

(6) main() calls sub() by c ¼ sub(a, b); The compiled code of the function call consists of . Push parameters in reverse order, i.e. push values of b¼2 and a¼1 into stack. . Call sub, which pushes the current PC onto stack and replaces PC with the entry address of sub, causing the CPU to enter sub().

When control first enters sub(), the stack contains a return address at the top, preceded by the parameters, a, b, of the caller, as shown in Fig. 2.10.

(7) Since sub() is written in C, it actions are exactly the same as that of main(), i.e. it . Push FP and let FP point at the saved FP;
. Shift SP downward to allocate space for local variables u, v.
. The compiled code may shift SP farther down for some temp space on stack. The stack contents becomes as shown in Fig. 2.11.

Fig. 2.11 Stack contents of called function

Fig. 2.12 Function stack frame

Fig. 2.13 Function call sequence

2.4.2 Stack Frames

While execution is inside a function, such as sub(), it can only access global variables, parameters passed in by the caller and local variables, but nothing else. Global and static local variables are in the combined Data section, which can be referenced by a fixed base register. Parameters and automatic locals have different copies on each invocation of the function. So the problem is: how to reference parameters and automatic locals? For this example, the parameters a, b, which correspond to the arguments x, y, are at 8(FP) and 12(FP). Similarly, the automatic local variables u, v are at -4(FP) and -8(FP). The stack area visible to a function, i.e. parameters and automatic locals, is called the Stack Frame of a function, like a frame of movie to a person. Thus, FP is called the Stack Frame Pointer. To a function, the stack frame looks like the following (Fig. 2.12).

From the above discussions, the reader should be able to deduce what would happen if we have a sequence of function calls, e.g.

```
crt0.o --> main() --> A(par_a) --> B(par_b) --> C(par_c)
```

For each function call, the stack would grow (toward low address) one more frame for the called function. The frame at the stack top is the stack frame of the current executing function, which is pointed by the CPU's frame pointer. The saved FP points (backward) to the frame of its caller, whose saved FP points back at the caller's caller, etc. Thus, the function call sequence is maintained in the stack as a link list, as shown in Fig. 2.13.

By convention, the CPU's FP ¼ 0 when crt0.o is entered from the OS kernel. So the stack frame link list ends with a 0. When a function returns, its stack frame is deallocated and the stack shrinks back.

2.4.3 Return From Function Call

When sub() executes the C statement return x+y+u+v, it evaluates the expression and puts the resulting value in the return value register (AX). Then it deallocates the local variables by

Fig. 2.14 Stack contents with reversed allocation scheme

.copy FP into SP; # SP now points to the saved FP in stack.
.pop stack into FP; # this restores FP, which now points to the caller's stack
frame, # leaving the return PC on the stack top.

(On the x86 CPU, the above operations are equivalent to the leave instruction). .Then, it executes the RET instruction, which pops the stack top into PC register, causing the CPU to execute from the saved return address of the caller.

(8) Upon return, the caller function catches the return value in the return register (AX). Then it cleans the parameters a, b, from the stack (by adding 8 to SP). This restores the stack to the original situation before the function call. Then it continues to execute the next instruction.

It is noted that some compilers, e.g. GCC Version 4, allocate automatic local variables in increasing address order. For instance, int a, b; implies (address of a) < (address of b). With this kind of allocation scheme, the stack contents may look like the following (Fig. 2.14).

In this case, automatic local variables are also allocated in "reverse order", which makes them consistent with the parameter order, but the concept and usage of stack frames remain the same.

2.4.4 Long Jump

In a sequence of function calls, such as

```
main() --> A() --> B()-->C();
```

when a called function finishes, it normally returns to the calling function, e.g. C() returns to B(), which returns to A(), etc. It is also possible to return directly to an earlier function in the calling sequence by a long jump. The following program demonstrates long jump in Unix/Linux.

```
/** longjump.c file: demonstrate long jump in Linux **/
#include <stdio.h>
#include <setjmp.h>
jmp_buf env; // for saving longjmp environment

int main()
{
   int r, a=100;
   printf("call setjmp to save environment\n");
   if ((r=setjmp(env)) == 0){
      A();
      printf("normal return\n");
   }
```

```
   else
      printf("back to main() via long jump, r=%d a=%d\n", r, a);
}

int A()
{ printf("enter A()\n");
     B();
     printf("exit A()\n");
}

int B()
{
   printf("enter B()\n");
   printf("long jump? (y|n) ");
   if (getchar()==’y’)
      longjmp(env, 1234);
   printf("exit B()\n");
}
```

In the longjump program, setjmp() saves the current execution environment in a jmp_buf structure and returns 0. The program proceeds to call A(), which calls B(). While in the function B(), if the user chooses not to return by long jump, the functions will show the normal return sequence. If the user chooses to return by longjmp(env, value), execution will return to the last saved environment with a nonzero value. In this case, it causes B() to return to main() directly, bypassing A(). The principle of long jump is very simple. When a function finishes, it returns by the (callerPC, callerFP) in the current stack frame, as shown in Fig. 2.15.

If we replace (callerPC, callerFP) with (savedPC, savedFP) of an earlier function in the calling sequence, execution would return to that function directly. In addition to the (savedPC, savedFP), setjmp() may also save CPU's general registers and the original SP, so that longjmp() can restore the complete environment of the returned function. Long jump can be used to abort a function in a calling sequence, causing execution to resume from a known environment saved earlier. Although rarely used in user mode programs, it is a common technique in systems programming. For example, it may be used in a signal catcher to bypass a user mode function that caused an exception or trap error. We shall demonstrate this technique later in Chap. 6 on signals and signal processing.

2.4.5 Run-Time Stack Usage in 64-Bit GCC

In 64-bit mode, the CPU registers are expanded to rax, rbx, rcx, rdx, rbp, rsp, rsi, rdi, r8 to r15, all 64-bit wide. The function call convention differs slightly from 32-bit mode. When calling a function, the first 6 parameters are passed in rdi, rsi, rdx, rcx, r8, r9, in that order. Any extra parameters are passed through the stack as they are in 32-bit mode. Upon entry, a called function first establishes the stack frame (using rbp) as usual. Then it may shift the stack pointer (rsp) downward for local variables and working spaces on the stack. The GCC compiler generated code may keep the stack pointer fixed,

Fig. 2.15 Function return frame

with a default reserved Red Zone stack area of 128 bytes, while execution is inside a function, making it possible to access stack contents by using rsp as the base register. However, the GCC compiler generated code still uses the stack frame pointer rbp to access both parameters and locals. We illustrate the function call convention in 64-bit mode by an example.

Example: Function Call Convention in 64-Bit Mode

(1) The following t.c file contains a main() function in C, which defines 9 local int (32-bit) variables, a to i. It calls a sub() function with 8 int parameters.

```
/********* t.c file ********/
#include <stdio.h>
int sub(int a, int b, int c, int d, int e, int f, int g, int h) {
    int u, v, w;
    u = 9;
    v = 10;
    w = 11;
    return a+g+u+v; // use first and extra parameter, locals
}

int main()
{
    int a, b, c, d, e, f, g, h, i;
    a = 1;
    b = 2;
    c = 3;
    d = 4;
    e = 5;
    f = 6;
    g = 7;
    h = 8;
    i = sub(a,b,c,d,e,f,g,h);
}
```

(2) Under 64-bit Linux, compile t.c to generate a t.s file in 64-bit

assembly by gcc –S t.c # generate t.s file

Then edit the t.s file to delete the nonessential lines generated by the compiler and add comments to explain the code actions. The following shows the simplified t.s file with added comment lines.

```
#------------ t.s file generated by 64-bit GCC compiler -------------– .globl sub
sub: # int sub(int a,b,c,d,e,f,g,h)

# first 6 parameters a, b, c, d, e, f are in registers
# rdi,rsi,rdx,rcx,r8d,r9d
# 2 extra parameters g,h are on stack.
```

```
# Upon entry, stack top contains g, h
# ----------------------------------
# ... ...| h | g | PC | LOW address
# ---------------|------------------
# rsp

# establish stack frame
            pushq %rbp
            movq %rsp, %rbp
# no need to shift rsp down because each function has a 128 bytes # reserved stack
area.
# rsp will be shifted down if function define more locals

# save first 6 parameters in registers on stack
            movl %edi, -20(%rbp) # a
            movl %esi, -24(%rbp) # b
            movl %edx, -28(%rbp) # C
            movl %ecx, -32(%rbp) # d
            movl %r8d, -36(%rbp) # e
            movl %r9d, -40(%rbp) # f

# access locals u, v, w at rbp -4 to -12
            movl $9, -4(%rbp)
            movl $10, -8(%rbp)
            movl $11, -12(%rbp)

# compute x + g + u + v:
            movl -20(%rbp), %edx # saved a on stack
            movl 16(%rbp), %eax # g at 16(rbp)
            addl %eax, %edx
            movl -4(%rbp), %eax # u at -4(rbp)
            addl %eax, %edx
            movl -8(%rbp), %eax # v at -8(rbp)
            addl %edx, %eax
```

```
# did not shift rsp down, so just popQ to restore rbp
        popq %rbp
        ret

#====== main function code in assembly ======
        .globl main
main:
# establish stack frame
        pushq %rbp
        movq %rsp, %rbp

# shit rsp down 48 bytes for locals
        subq $48, %rsp
```

```
# locals are at rbp -4 to -32
        movl $1, -4(%rbp) # a=1
        movl $2, -8(%rbp) # b=2
        movl $3, -12(%rbp) # c=3
        movl $4, -16(%rbp) # d=4
        movl $5, -20(%rbp) # e=5
        movl $6, -24(%rbp) # f=6
        movl $7, -28(%rbp) # g=7
        movl $8, -32(%rbp) # h=8

# call sub(a,b,c,d,e,f,g,h): first 6 parameters in registers movl -24(%rbp), %r9d # f
        in r9
        movl -20(%rbp), %r8d # e in r8
        movl -16(%rbp), %ecx # d in ecx
        movl -12(%rbp), %edx # c in edx
        movl -8(%rbp), %esi # b in esi
        movl -4(%rbp), %eax # a in eax but will be in edi
# push 2 extra parameters h,g on stack
        movl -32(%rbp), %edi # int h in edi
        pushq %rdi # pushQ rdi ; only low 32-bits = h
        movl -28(%rbp), %edi # int g in edi
        pushq %rdi # pushQ rdi ; low 32-bits = g

        movl %eax, %edi # parameter a in edi
        call sub # call sub(a,b,c,d,e,f,g,h)

        addq $16, %rsp # pop stack: h,g, 16 bytes
        movl %eax, -36(%rbp) # i = sub return value in eax

        movl $0, %eax # return 0 to crt0.o
        leave
        ret

# GCC compiler version 5.3.0
        .ident "GCC: (GNU) 5.3.0"
```

2.5 Link C Program with Assembly Code

In systems programming, it is often necessary to access and control the hardware, such as CPU registers and I/O port locations, etc. In these situations, assembly code becomes necessary. It is therefore important to know how to link C programs with assembly code.
2.5 Link C Program with Assembly Code 41

2.5.1 Programming in Assembly

(1) C code to Assembly Code

```
/************* a.c file ********************/
#include <stdio.h>

extern int B();

int A(int x, int y)
{
   int d, e, f;
   d = 4; e = 5; f = 6;
   f = B(d,e);
}
```

```
======= compile a.c file into 32-bit assembly code ======
cc -m32 -S a.c ===> a.s file

====================================================
            .text
            .globl A
A:
             pushl %ebp
             movl %esp, %ebp

             subl $24, %esp
             movl $4, -12(%ebp) # d=4
             movl $5, -8(%ebp) # e=5
             movl $6, -4(%ebp) # f=6

             subl $8, %esp
             pushl -8(%ebp) # push e
             pushl -12(%ebp) # push d
             call B

             addl $16, %esp # clean stack
             movl %eax, -4(%ebp) # f=return value in AX

             leave
             ret

====================================================
```

Explanations of the Assembly Code
The assembly code generated by GCC consists of three parts:

(1). Entry: also called the prolog, which establishes stack frame, allocates local variables and working space on stack

(2). Function body, which performs the function task with return value in AX register (3). Exit: also called the epilog, which deallocates stack space and return to caller

The GCC generated assembly code are explained below, along with the stack contents

A: # A() start code location (1). Entry Code:

```
        pushl %ebp
        movl %esp, %ebp # establish stack frame
```

The entry code first saves FP (%bp) on stack and let FP point at the saved FP of the caller. The stack contents become

```
              SP
---------|--------------------------- LOW address
 xxx |PC|FP|
---------|---------------------------
              FP
```

```
        subl $24, %esp
```

Then it shift SP downward 24 bytes to allocate space for locals variables and working area.

(2). Function Body Code:

```
        movl $4, -20(%ebp) // d=4
        movl $5, -16(%ebp) // e=5
        movl $6, -12(%ebp) // f=6
```

While inside a function, FP points at a fixed location and acts as a base register for accessing local variables, as well as parameters. As can be seen, the 3 locals d, e, f, each 4 bytes long, are at the byte offsets -20, -16, -12 from FP. After assigning values to the local variables, the stack contents become

```
              SP --- -24 ----> SP
               | |
---------- -4 -8 -12 -16 -20 -24|------------------ LOW address xxx |PC|FP|? |? | 6|5|4 |? |
---------|---------f---e---d----|------------------
              FP
```

# call B(d,e): push parameters d, e in reverse order:

```
        subl $8, %esp # create 8 bytes TEMP slots on stack pushl -16(%ebp) # push e
        pushl -20(%ebp) # push d
```

```
                                                                                  SP
 -------------|-4 -8 -12 -16 -20 -24|- TEMP -|-------|--- LOW address xxx |retPC|FP|? |? | 6|5|4 |?
 | ??| ?? | e|d |
 ----------|---------f---e---d-----|-------------------
                   FP
```

2.5 Link C Program with Assembly Code 43call B # B() will grow stack to the RIGHT

```
# when B() returns:
        addl $16, %esp # clean stack
        movl %eax, -4(%ebp) # f = return value in AX
```

(3). Exit Code:

```
# leave
        movl %ebp, %esp # SAME as leave
        popl %ebp

        ret # pop retPC on stack top into PC
```

2.5.2 Implement Functions in Assembly

Example 1: Get CPU registers. Since these functions are simple, they do not need to establish and deallocate stack frames.

```
#============== s.s file ===============
        .global get_esp, get_ebp
get_esp:
        movl %esp, %eax
        ret
get_ebp:
        movl %ebp, %eax
        ret
#=====================================

int main()
{
   int ebp, esp;
   ebp = get_ebp();
   esp = get_esp();
   printf("ebp=%8x esp=%8x\n", ebp, esp);
}
```

Example 2: Assume int mysum(int x, int y) returns the sum of x and y. Write mysum() function in ASSEMBLY. Since the function must use its parameters to compute the sum, we show the entry, function body and exit parts of the function code.

```
# =========== mysum.s file ===========================
        .text # Code section
        .global mysum, printf # globals: export mysum, import printf mysum:
```

(1) Entry:(establish stack frame)

```
        pushl %ebp
        movl %esp, %ebp

# Caller has pushed y, x on stack, which looks like the following # 12 8 4 0
#---------------------------------------------------
# | y | x |retPC| ebp|
#---------------------------|---------------------–
# ebp

# (2): Function Body Code of mysum: compute x+y in AX register movl 8(%ebp),
        %eax # AX = x
        addl 12(%ebp), %eax # AX += y

# (3) Exit Code: (deallocate stack space and return)
        movl %ebp, %esp
        pop %ebp
        ret
# =========== end of mysum.s file ==========================

int main() # driver program to test mysum() function
{
   int a,b,c;
   a = 123; b = 456;
   c = mysum(a, b);
   printf("c=%d\n", c); // c should be 579
}
```

2.5.3 Call C functions from Assembly

Example 3: Access global variables and call printf()

```
int a, b;
int main()
{
    a = 100; b = 200;
    sub();
}

#========== Assembly Code file ================
        .text
        .global sub, a, b, printf
sub:
        pushl %ebp
        movl %esp, %ebp
```

```
        pushl b
        pushl a
        pushl $fmt # push VALUE (address) of fmt
        call printf # printf(fmt, a, b);
        addl $12, %esp

        movl %ebp, %esp
        popl %ebp
        ret

        .data
fmt: .asciz "a=%d b=%d\n"
#====================================
```

2.6 Link Library

A link library contains precompiled object code. During linking, the linker uses the link library to complete the linking process. In Linux, there are two kinds of link libraries; static link library for static linking, and dynamic link library for dynamic linking. In this section, we show how to create and use link libraries in Linux.

Assume that we have a function

```
// musum.c file
int mysum(int x, int y){ return x + y; }
```

We would like to create a link library containing the object code of the mysum() function, which can be called from different C programs, e.g.

```
// t.c file
int main()
{
    int sum = mysum(123,456);
}
```

2.6.1 Static Link Library

The following steps show how to create and use a static link library.

(1). gcc –c mysum.c # compile mysum.c into mysum.o (2). ar rcs libmylib.a mysum.o # create static link library with member mysum.o
(3). gcc -static t.c -L. –lmylib # static compile-link t.c with libmylib.a as link library
(4). a.out # run a.out as usual

In the compile-link step (4), -L. specifies the library path (current directory), and -l specifies the library. Note that the library (mylib) is specified without the prefex lib, as well as the suffix .a

2.6.2 Dynamic Link Library

The following steps show how to create and use a dynamic link library.

```
(1). gcc –c -fPIC mysum.c # compile to Position Independent Code mysum.o
(2). gcc –shared -o libmylib.so mysum.o # create shared libmylib.so with mysum.o
(3). gcc t.c -L. –lmylib # generate a.out using shared library libmylib.so
(4). export LD_LIBRARY_PATH=./ # to run a.out, must export LD_LIBRARY=./
(5). a.out # run a.out. ld will load libmylib.so
```

In both cases, if the library is not in the current directory, simply change the –L. option and set the LD_LIBRARY_PATH to point to the directory containing the library. Alternatively, the user may also place the library in a standard lib directory, e.g. /lib or /usr/lib and run ldconfig to configure the dynamic link library path. The reader may consult Linux ldconfig (man 8) for details.

2.7 Makefile

So far, we have used individual gcc commands to compile-link the source files of C programs. For convenience, we may also use a sh script which includes all the commands. These schemes have a major drawback. If we only change a few of the source files, the sh commands or script would still compile all the source files, including those that are not modified, which is unnecessary and time consuming. A better way is to use the Unix/Linux make facility (GNU make 2008). make is a program, which reads a makefile, or Makefile in that order, to do the compile-link automatically and selectively. This section covers the basics of makefiles and shows their usage by examples.

2.7.1 Makefile Format

A make file consists of a set of targets, dependencies and rules. A target is usually a file to be created or updated, but it may also be a directive to, or a label to be referenced by, the make program. A target depends on a set of source files, object files or even other targets, which are described in a Dependency List. Rules are the necessary commands to build the target by using the Dependency List. Figure 2.16 shows the format of a makefile.

2.7.2 The make Program

When the make program reads a makefile, it determines which targets to build by comparing the timestamps of source files in the Dependency List. If any dependency has a newer timestamp since last

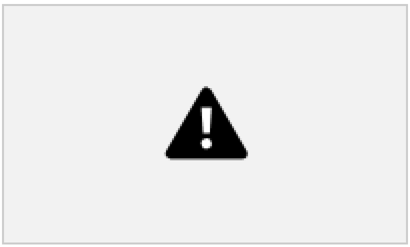


Fig. 2.16 Makefile format

build, make will execute the rule associated with the target. Assume that we have a C program consisting of three source files:

```
(1). type.h file: // header file
        int mysum(int x, int y) // types, constants, etc

(2). mysum.c file: // function in C
        #include <stdio.h>
        #incldue "type.h"
        int mysum(int x, int y)
        {
                return x+y;
        }

(3). t.c file: // main() in C
        #include <stdio.h>
        #include "type.h"
        int main()
        {
                int sum = mysum(123,456);
                printf("sum = %d\n", sum);
        }
```

Normally, we would use the sh command

```
gcc –o myt main.c mysum.c
```

to generate a binary executable named myt. In the following, we shall demonstrate compile-link of C programs by using makefiles.

2.7.3 Makefile Examples

Makefile Example 1

(1). Create a makefile named mk1 containing:

```
myt: type.h t.c mysum.c # target: dependency list
        gcc –o myt t.c mysum.c # rule: line MUST begin with a TAB
```

The resulting executable file name, myt in this example, usually matches that of the target name. This allows make to decide whether or not to build the target again later by comparing its timestamp against those in the dependency list.

(2). Run make using mk1 as the makefile: make normally uses the default makefile or Makefile, whichever is present in the current directory. It can be directed to use a different makefile by the –f flag, as in

```
make –f mk1
```

make will build the target file myt and show the command execution as

```
gcc –o myt t.c mysum.c
```

(3). Run the make command again. It will show the message

```
make: ‘myt’ is up to date
```

In this case, make does not build the target again since none of the files has changed since last build.

(4). On the other hand, make will execute the rule command again if any of the files in the dependency list has changed. A simple way to modify a file is by the touch command, which changes the timestamp of the file. So if we enter the sh commands

```
touch type.h // or touch *.h, touch *.c, etc.
make –f mk1
```

make will recompile-link the source files to generate a new myt file

(5). If we delete some of the file names from the dependency list, make will not execute the rule command even if such files are changed. The reader may try this to verify it.

As can be seen, mk1 is a very simple makfile, which is not much different than sh commands. But we can refine makefiles to make them more flexible and general.

Makefile Example 2: Macros in Makefile

(1). Create a makefile named mk2 containing:

```
CC = gcc # define CC as gcc
CFLAGS = -Wall # define CLAGS as flags to gcc
OBJS = t.o mysum.o # define Object code files
INCLUDE = -Ipath # define path as an INCLUDE directory

  myt: type.h $(OBJS) # target: dependency: type.h and .o files $(CC) $(CFLAGS) –o t
                                    $(OBJS) $(INCLUDE)
```

In a makefile, macro defined symbols are replaced with their values by $(symbol), e.g. $(CC) is replaced with gcc, $(CFLAGS) is replaced with –Wall, etc. For each .o file in the dependency list,

make will compile the corresponding .c file into .o file first. However, this works only for .c files. Since all the .c files depend on .h files, we have to explicitly include type.h (or any other .h files) in the dependency list also. Alternatively, we may define additional targets to specify the dependency of .o files on .h files, as in

```
t.o: t.c type.h # t.o depend on t.c and type.h
         gcc –c t.c
```

```
mysum.o: mysum.c type.h # mysum.o depend type.h
            gcc –c mysum.c
```

If we add the above targets to a makefile, any changes in either .c files or type.h will trigger make to recompile the .c files. This works fine if the number of .c files is small. It can be very tedious if the number of .c files is large. So there are better ways to include .h files in the dependency list, which will be shown later.

(3). Run make using mk2 as the makefile:

```
          make –f mk2
```

(4). Run the resulting binary executable myt as before.

The simple makefiles of Examples 1 and 2 are sufficient for compile-link most small C programs. The following shows some additional features and capabilities of makefiles.

Makefile Example 3: Make Target by Name
When make runs on a makefile, it normally tries to build the first target in the makefile. The behavior of make can be changed by specifying a target name, which causes make to build the specific named target. As an example, consider the makefile named mk3, in which the new features are highlighted in bold face letters.

```
# ------------------ mk3 file --------------------–
CC = gcc # define CC as gcc
CFLAGS = -Wall # define CLAGS as flags to gcc
OBJS = t.o mysum.o # define Object code files
INCLUDE = -Ipath # define path as an INCLUDE directory all: myt install # build all

listed targets: myt, install

  myt: t.o mysum.o # target: dependency list of .o files $(CC) $(CFLAGS) –o myt
                                        $(OBJS) $(INCLUDE)

t.o: t.c type.h # t.o depend on t.c and type.h
            gcc –c t.c
mysum.o: mysum.c type.h # mysum.o depend mysum.c and type.h gcc –c
          mysum.c

install: myt # depend on myt: make will build myt first echo install myt to /usr/local/bin
            sudo mv myt /usr/local/bin/ # install myt to /usr/local/bin/
```

```
run: install # depend on install, which depend on myt echo run executable image myt
            myt || /bin/true # no make error 10 if main() return non-zero

clean:
          rm –f *.o 2> /dev/null # rm all *.o files
          sudo rm –f /usr/local/bin/myt # rm myt
```

The reader may test the mk3 file by entering the following make commands:

```
(1). make [all] –f mk3 # build all targets: myt and install (2). make install –f mk3 # build
target myt and install myt (3). make run –f mk3 # run /usr/local/bin/myt
(4). make clean –f mk3 # remove all listed files
```

Makefile Variables: Makefiles support variables. In a makefile, % is a wildcard variable similar to * in sh. A makefile may also contain automatic variables, which are set by make after a rule is matched. They provide access to elements from the target and dependency lists so that the user does not have to explicitly specify any filenames. They are very useful for defining general pattern rules. The following lists some of the automatic variables of make.

$@ : name of current target.
$< : name of first dependency
$^ : names of all dependencies
$* : name of current dependency without extension
$? : list of dependencies changed more recently than current target.

In addition, make also supports suffix rules, which are not targets but directives to the make program. We illustrate make variables and suffix rules by an example.

In a C program, .c files usually depend on all .h files. If any of the .h files is changed, all .c files must be re-compiled again. To ensure this, we may define a dependency list containing all the .h files and specify a target in a makefile as

```
DEPS = type.h # list ALL needed .h files
    %.o: %.c $(DEPS) # for all .o files: if its .c or .h file changed $(CC) –c –o $@ # compile
                                    corresponding .c file again
```

In the above target, %.o stands for all .o files and $@ is set to the current target name, i.e. the current .o file name. This avoids defining separate targets for individual .o files.

Makefile Example 4: Use make variables and suffix rules

```
# ---------- mk4 file -------------
CC = gcc
CFLAGS = -I.
OBJS = t.o mysum.o
AS = as # assume we have .s files in assembly also
DEPS = type.h # list all .h files in DEPS
```

```
.s.o: # for each fname.o, assemble fname.s into fname.o
          $(AS) –o $< -o $@ # -o $@ REQUIRED for .s files

.c.o: # for each fname.o, compile fname.c into fname.o
          $(CC) –c $< -o $@ # -o $@ optional for .c files

%.o: %.c $(DEPS) # for all .o files: if its .c or .h file changed $(CC) –c –o $@ $< #
          compile corresponding .c file again

myt: $(OBJS)
          $(CC) $(CFLAGS) -o $@ $^
```

In the makefile mk4, the lines .s.o: and .c.o: are not targets but directives to the make program by the suffix rule. These rules specify that, for each .o file, there should be a corresponding .s or .c file to build if their timestamps differ, i.e. if the .s or .c file has changed. In all the target rules, $@ means the current target, $< means the first file in the dependency list and $^ means all files in the dependency list. For example, in the rule of the myt target, -o $@ specifies that the output file name is the current target, which is myt. $^ means it includes all the files in the dependency list, i.e. both t.o and mysum.o. If we change $^ to $< and touch all the .c files, make would generate an "undefined reference to mysum" error. This is because $< specifies only the first file (t.o) in the dependency list, make would only recompile t.c but not mysum.c, resulting a linking error due to missing mysum.o file. As can be seen from the example, we may use make variables to write very general and compact makefiles. The downside is that such makefiles are rather hard to understand, especially for beginning programmers.

Makfiles in Subdirectories

A large C programming project usually consists of tens or hundreds of source files. For ease of maintenance, the source files are usually organized into different levels of directories, each with its own makefile. It's fairly easy to let make go into a subdirectory to execute the local makefile in that directory by the command

```
(cd DIR; $(MAKE)) OR cd DIR && $(MAKE)
```

After executing the local makefile in a subdirectory, control returns to the current directory form where make continues. We illustrate this advanced capability of make by a real example.

Makefile Example 5: PMTX System Makefiles

PMTX (Wang 2015) is a Unix-like operating system designed for the Intel x86 architecture in 32-bit protect mode. It uses 32-bit GCC assembler, compiler and linker to generate the PMTX kernel image. The source files of PMTX are organized in three subdirectories:

Kernel : PMTX kernel files; a few GCC assembly files, mostly in C
Fs : file system source files; all in C
Driver : device driver source files; all in C

The compile-link steps are specified by Makefiles in different directories. The top level makefile in the PMTX source directory is very simple. It first cleans up the directories. Then it goes into the Kernel subdirectory to execute a Makefile in the Kernel directory. The Kernel Makefile first generates .o files

for both .s and .c files in Kernel. Then it directs make to go into the Driver and Fs subdirectories to generate .o file by executing their local Makfiles. Finally, it links all the .o files to a kernel image file. The following shows the various Makefiles of the PMTX system.

```
#------------– PMTX Top level Makefile -------------–
all: pmtx_kernel

pmtx_kernel:
```

```
              make clean
              cd Kernel && $(MAKE)

clean: # rm mtx_kerenl, *.o file in all directories

#------------– PMTX Kernel Makefile ---------------–
AS = as –Iinclude
CC = gcc
LD = ld
CPP = gcc -E -nostdinc
CFLAGS = -W -nostdlib -Wno-long-long -I include -fomit-frame-pointer

KERNEL_OBJS = entry.o init.o t.o ts.o traps.o trapc.o queue.o \ fork.o exec.o wait.o io.o
syscall.o loader.o pipe.o mes.o signal.o \ threads.o sbrk.o mtxlib.o

K_ADDR=0x80100000 # kernel start virtual address

all: kernel

 .s.o: # build each .o if its .s file has changed
              ${AS} -a $< -o $*.o > $*.map

pmtx_kernel: $(KERNEL_OBJS) # kernel target: depend on all OBJs cd ../Driver &&
              $(MAKE) # cd to Driver, run local Makefile
              cd ../Fs && $(MAKE) # cd to Fs/, run local Makefile

# link all .o files with entry=pm_entry, start VA=0x80100000 ${LD} --oformat
              binary -Map k.map -N -e pm_entry \
                     –Ttext ${K_ADDR} -o $@ \
                       ${KERNEL_OBJS} ../DRIVER/*.o ../FS/*.o
clean:
              rm -f *.map *.o
              rm –f ../DRIVER.*.map ../DRIVER/*.o
              rm –f ../FS/*.map ../FS/*.o
```

The PMTX kernel makfile first generates .o files from all .s (assembly) files. Then it generates other .o files from .c files by the dependency lists in KERNEL_OBJ. Then it goes into Driver and Fs directories to execute the local makefiles, which generate .o files in these directories. Finally, it links all the .o files to generate the pmtx_kernel image file, which is the PMTX OS kernel. In contrast, since

all files in Fs and Driver are in C, their Makefiles only compile .c files to .o files, so there are no .s or ld related targets and rules.

```
#------------– PMTX Driver Makefile ---------------–
CC=gcc
CPP=gcc -E -nostdinc
CFLAGS=-W -nostdlib -Wno-long-long -I include -fomit-frame-pointer

DRIVER_OBJS = timer.o pv.o vid.o kbd.o fd.o hd.o serial.o pr.o atapi.o driverobj:
${DRIVER_OBJS}
```

```
#------------– PMTX Fs Makefile ------------------–
CC=gcc
CPP=gcc -E -nostdinc
CFLAGS=-W -nostdlib -Wno-long-long -I include -fomit-frame-pointer

  FS_OBJS = fs.o buffer.o util.o mount_root.o alloc_dealloc.o \ mkdir_creat.o cd_pwd.o rmdir.o
          link_unlink.o stat.o touch.o \ open_close.o read.o write.o dev.o mount_umount.o
fsobj: ${FS_OBJS}
#-------------- End of Makefile ------------------–
```

2.8 The GDB Debugger

The GNU Debugger (GDB) (Debugging with GDB 2002; GDB 2017) is an interactive debugger, which can debug programs written in C, C++ and several other languages. In Linux, the command man gdb displays the manual pages of gdb, which provides a brief description of how to use GDB. The reader may find more detailed information on GDB in the listed references. In this section, we shall cover GDB basics and show how to use GDB to debug C programs in the Integrated Development Environment (IDE) of EMACS under X-windows, which is available in all Linux systems. Since the Graphic User Interface (GUI) part of different Linux distributions may differ, the following discussions are specific to Ubuntu Linux Version 15.10 or later, but it should also be applicable to other Linux distributions, e.g. Slackware Linux 14.2, etc.

2.8.1 Use GDB in Emacs IDE

1. Source Code: Under X-window, open a pseudo-terminal. Use EMACS to create a Makefile, as shown below.

Makefile:

```
t: t.c
        gcc –g –o t t.c
```

Then use EMACS to edit a C source file. Since the objective here is to show GDB usage, we shall use a very simple C program.

```
/******** Source file: t.c Code ********/
#include <stdio.h>
int sub();
int g, h; // globals

int main()
{
   int a, b, c;
   printf("enter main\n");
   a = 1;
```

```
    b = 2;
    c = 3;
    g = 123;
    h = 456;
    c = sub(a, b);
    printf("c = %d\n", c);
    printf("main exit\n");
}

int sub(int x, int y)
{
    int u,v;
    printf("enter sub\n");
    u = 4;
    v = 5;
    printf("sub return\n");
    return x+y+u+v+g+h;
}
```

2. Compile Source Code: When EMACS is running, it displays a menu and a tool bar at the top of the edit window (Fig. 2.17).

Each menu can be opened to display a table of submenus. Open EMACS Tools menu and select Compile. EMACS will show a prompt line at the bottom of the edit window

```
make –k
```

and waits for user response. EMACS normally compile-link the source code by a makefile. If the reader already has a makefile in the same directory as shown above, press the Enter key to let EMACS continue. In instead of a makefile, the reader may also enter the command line manually.

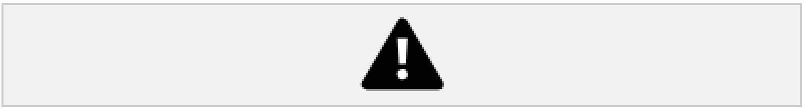


Fig. 2.17 EMACS menu and tool bar

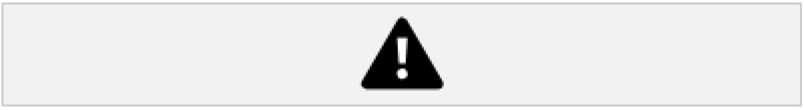


Fig. 2.18 GDB menu and tool bar

```
gcc -g -o t t.c
```

In order to generate a binary executable for GDB to debug, the –g flag is required. With the –g flag, the GCC compiler-linker will build a symbol table in the binary executable file for GDB to access variables and functions during execution. Without the –g flag, the resulting executable file can not be debugged by GDB. After compilation finishes, EMACS will show the compile results, including warning or error messages, if any, in a separate window below the source code window.

3. Start up GDB: Open EMACS Tools menu and select Debugger.

EMACS will show a prompt line at the bottom of the edit window and wait for user

response. `gdb –i=mi t`

Press Enter to start up the GDB debugger. GDB will run in the upper window and display a menu and a tool bar at the top of the EMACS edit window, as shown in Fig. 2.18.

The user may now enter GDB commands to debug the program. For example, to set break points, enter the GDB commands

```
b main # set break point at main
b sub # set break point at sub
b 10 # set break point at line 10 in program
```

When the user enters the Run (r) command (or choose Run in the tool bar), GDB will display the program code in the same GDB window. Other frames/windows can be activated through the submenu GDB-Frames or GDB-Windows. The following steps demonstrate the debugging process by using both commands and tool bar in the multi-windows layout of GDB.

4. GDB in Multi-Windows: From the GDB menu, choose Gud ¼> GDB-MI ¼> Display Other Windows, where ¼> means follow a submenu. GDB will display GDB buffers in different windows, as shown in Fig. 2.19.

Figure 2.19 shows six (6) GDB windows, each displays a specific GDB buffer.

Gud-t: GDB buffer for user commands and GDB messages
t.c: Program source code to show progress of execution
Stack frames: show stack frames of function calling sequence
Local Registers: show local variables in current executing function
Input/output: for program I/O
Breakpoints: display current break points settings

It also shows some of the commonly used GDB commands in a tool bar, e.g. Run, Continue, Next line, Step line, which allows the user to choose an action instead of entering commands.

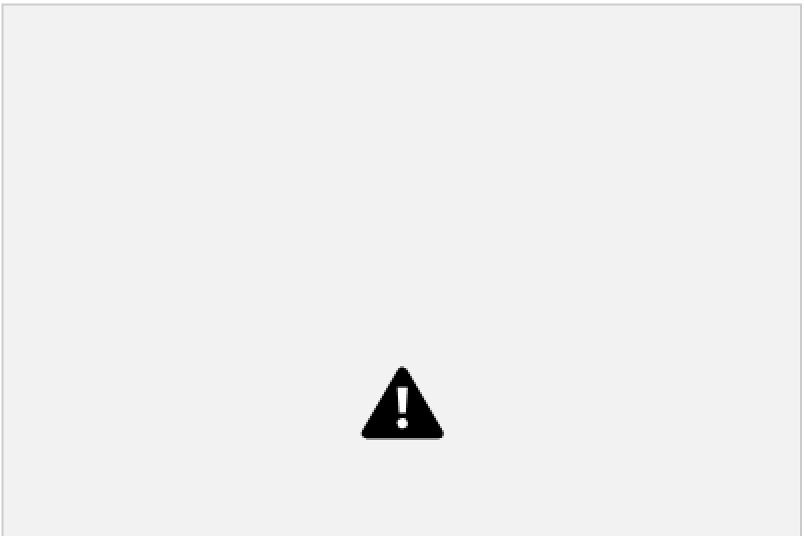


Fig. 2.19 Multi-windows of GDB

While the program is executing, GDB shows the execution progress by a dark triangular mark, which points to the next line of program code to be executed. Execution will stop at each break point, providing a break for the user to interact with GDB. Since we have set main as a break point, execution will stop at main when GDB starts to run. While execution stops at a break point, the user may interact with GDB by entering commands in the GDB window, such as set/clear break points, display/change variables, etc. Then, enter the Continue (c) command or choose Continue in the tool bar to continue program execution. Enter Next (n) command or choose Next line or Step line in the GDB tool bar to execute in single line mode.

Figure 2.19 shows that execution in main() has already completed executing several lines of the program code, and the next line mark is at the statement

```
g ¼ 123;
```

At the moment, the local variables a, b, c are already assigned values, which are shown in the Locals registers windows as a¼1, b¼2, c¼3. Global variables are not shown in any window, but the user may enter Print (p) commands

```
p g
p h
```

to print the global variables g and h, both of which should still be 0 since they are not assigned any values yet.

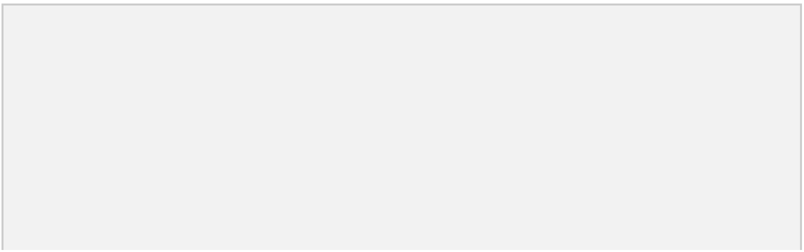


Fig. 2.20 Multi-windows of GDB

Figure 2.19 also shows the following GDB windows:

. input/output window shows the outputs of printf statements of main().
. stack frames window shows execution is now inside the main() function.
. breakpoints window shows the current breakpoints settings, etc.

This multi-windows layout provides the user with a complete set of information about the status of the executing program.

The user may enter Continue or choose Continue in the tool bar to continue the execution. When control reaches the sub() function, it will stop again at the break point. Figure 2.20 shows that the program execution is now inside sub() and the execution already passed the statements before

```
printf("return from sub\n");
```

At this moment, the Locals Registers window shows the local variables of sub() as u¼4 and v¼5. The input/output window shows the print results of both main() and sub(). The Stack frames window shows sub() is the top frame and main() is the next frame, which is consistent with the function calling sequence.

(5). Additional GDB Commands: At each break point or while executing in single line mode, the user may enter GDB commands either manually, by the GDB tool bar or by choosing submenu items in the Gud menu, which includes all the commands in the GDB tool bar. The following lists some additional GDB commands and their meanings.

Clear Break Points:

```
   clear line# : clear bp at line#
   clear name : clear bp at function name
```

Change Variable Values

```
   set var a¼100 : set variable a to 100
   set var b¼200 : set b to 200, etc.
```

Watch Variable Changes:

```
   watch c : watch for changes in variable c; whenever c changes, it will display its old value
             and new value.
```

Back trace (bt):

```
   bt stackFrame# to back trace stack frames
```

2.8.2 Advices on Using Debugging Tools

GDB is a powerful debugger, which is fairly easy to use. However, the reader should keep in mind that all debugging tools can only offer limited help. In some cases, even a powerful debugger like the GDB is of little use. The best approach to program development is to design the program's algorithm carefully and then write program code in accordance with the algorithm. Many beginning programmers tend to write program code without any planning, just hoping their program would work, which most likely would not. When their program fails to work or does not produce the right results, they would immediately turn to a debugger, trying to trace the program executions to find out the problem. Relying too much on debugging tools is often counter-productive as it may waste more time than necessary. In the following, we shall point out some common programming errors and show how to avoid them in C programs.

2.8.3 Common Errors in C programs

A program in execution may encounter many types of run-time errors, such as illegal instruction, privilege violation, divide by zero, invalid address, etc. Such errors are recognized by the CPU as exceptions, which trap the process to the operating system kernel. If the user has not made any provision to handle such errors, the process will terminate by a signal number, which indicates the cause of the exception. If the program is written in C, which is executed by a process in user mode, exceptions such as illegal instruction and privilege violation should never occur. In system program ming, programs seldom use divide operations, so divide by zero exceptions are also rare. The predominate type of run-time errors are due to invalid addresses, which cause memory access exceptions, resulting in the dreadful and familiar message of segmentation fault. In the following, we list some of the most probable causes in C programs that lead to memory access exceptions at run-time.

(1). Uninitialized pointers or pointers with wrong values: Consider the following code segments, with line numbers for ease of reference.

```
1. int *p; // global, initial value = 0
   int main()
   {
2. int *q; // q is local on stack, can be any value
3. *p = 1; // dereference a NULL pointer
4. *q = 2; // dereference a pointer with unknown value }
```

Line 1 defines a global integer pointer p, which is in the BSS section of the run-time image with an initial value 0. So it's a NULL pointer. Line 3 tries to dereference a NULL pointer, which will cause a segmentation fault.

Line 2 defines a local integer pointer q, which is on the stack, so it can be any value. Line 4 tries to dereference the pointer q, which points at an unknown memory location. If the location is outside of the program's writable memory area, it will cause a segmentation fault due to memory access violation. If the location is within the program's writable memory area, it may not cause an immediate error but it may lead to other errors later due to corrupted data or stack contents. The latter kind of run-time error is extremely difficult to diagnose because the errors may have propagated through the program execu tion. The following shows the correct ways of using these pointers. Modify the above code segment as shown below.

```
int x, *p; // or int *p = &x;
int main()
{
     int *q;
     p = &x; // let p point at x
     *p = 1;
     q = (int *)malloc(sizeof(int); // q point at allocate memory *q = 2;
 }
```

The principle is very simple. When using any pointer, the programmer must ensure the pointer is not NULL or has been set to point to a valid memory address.

(2). Array index out of bounds: In C programs, each array is defined with a finite number of N elements. The index of the array must be in the range of [0, N-1]. If the array index exceeds the range at run-time, it may cause invalid memory access, which either corrupt the program data area or result in a segmentation fault. We illustrate this by an example. Consider the following code segment.

```
   #define N 10
1. int a[N], i; // An array of N elements, followed by int i int main()
   {
2. for (i=0; i<N; i++) // index i in range
            a[i] = i+1; // set a[ ] values = 1 to N
3. a[N] = 123456789; // set a[N] to a LARGE value 4. printf("i = %d\n", i); // print
current i value
5. printf("%d\n", a[i])); // segmentation fault !
   }
```

Line 1 defines an array of N elements, which is followed by the index variable i. Line 2 represents the proper usage of the array index, which is within the bounds [0, N-1]. Line

3.sets a[N] to a large value, which actually changes the variable i because a[N] and i are in the same memory location. Line 4 prints the current value of i, which is no longer N but the large value. Line 5 will most likely to cause a segmentation fault because a[123456789] tries to access a memory location outside of the program's data area.
(3). Improper use of string pointers and char arrays: Many string operation functions in the C library are defined with char * parameters. As a specific example, consider the strcpy() function, which is defined as

```
char * strcpy(char *dest, char *src)
```

It copies a string from src to dest. The Linux man page on strcpy() clearly specifies that the dest string must be large enough to receive the copy. Many programmers, including some 'experienced' graduate students, often overlook the specification and try to use strcpy() as follows.

```
char *s; // s is a char pointer
strcpy(s, " this is a string");
```

The code segment is wrong because s is not pointing at any memory location with enough space to receive the src string. If s is global, it is a NULL pointer. In this case, strcpy() will cause a segmentation fault immediately. If s is local, it may point to an arbitrary memory location. In this case, strcpy() may not cause an immediate error but it may lead to other errors later due to corrupted memory contents. Such errors are very subtle and difficult to diagnose even with a debugger such as GDB. The correct way of using strcpy() is to ensure dest is NOT just a string pointer but a real memory area with enough space to receive the copied string, as in

```
char s[128]; // s is a char array
strcpy(s, " this is a string");
```

Although the same s variable in both char *s and char s[128] can be used as an address, the reader must beware there is a fundamental difference between them.
(4). The assert macro: Most Unix-like systems, including Linux, support an assert(condition) macro, which can be used in C programs to check whether a specified condition is met or not. If the condition expression evaluates to FALSE (0), the program will abort with an error message. As an example, consider the following code segments, in which the mysum() function is designed to return the sum of an integer array (pointed by int *ptr) of n<¼128 elements. Since the function is called from other code of a program, which may pass in invalid parameters, we must ensure the pointer ptr is not NULL and the array size n does not exceed the LIMIT. These can be done by including assert() statements at the entry point of a function, as shown below.

```
#define LIMIT 128
int mysum(int *ptr, int n)
{
     int i = 0, sum = 0;
     assert(ptr != NULL); // assert ptr not NULL
     assert(n <= LIMIT); // assert n <= LIMIT
     while(i++ < n)
          sum += *ptr++
}
```