

A Heavily Commented Linux Kernel Source Code

Zhao Jiong



A Heavily Commented Linux Kernel Source Code

Kernel Version 0.12

(Chinese Revision 5)



Zhao Jiong
Jiong.zhao@TongJi.edu.cn

WWW.OLDLINUX.ORG

2019-01-24

Abstract

This book provides detailed and comprehensive comments and explanations on all source code of the early Linux kernel (V0.12), aiming to enable readers to gain a comprehensive and profound understanding of the working mechanism of Linux in a shortest possible time and to lay a solid foundation for further study of modern Linux systems. Although the version of the analysis is very low, the kernel has been able to compile and run, and it already includes the essence of the working principle of Linux.

The book first briefly introduced the development history of the Linux kernel, explained the main differences between the various kernel versions and improvements, and gave the reasons for choosing the 0.12 kernel source code as the study object. Then it gives the basic knowledge needed to read the source code, outlines the hardware structure of the PC running the Linux system, the assembly language used by the kernel, the extends of C language, and focuses on the 80X86 processor in protected mode. Then we introduced the kernel code overview, given the kernel source directory tree structure, and according to the organizational structure of all kernel, programs and files are described in detail. In order to deepen the reader's understanding of the working principle of the kernel, the last chapter gives a number of related operational debugging tests. All relevant information in the book can be downloaded from the website www.oldlinux.org.

This book suits as the assistant and practical teaching material of university computer major student study operating system course, also suitable for self-study reference book of Linux lovers as learning kernel operating principle, also can be used as the reference book that the general technical personnel develops the embedded system.

Copyright statement

The author retains all rights to modify and formally publish this book. Feedback from readers can be sent to me via e-mail: jiong.zhao@tongji.edu.cn or gohigh@gmail.com, or you can write directly to: School of Mechanical and Energy Engineering, Institute of Mechanical and Electronic Engineering, Tongji University, Address: Room B409, Machinery Building, 4800 Cao'an Road, Shanghai, China: 201804

Dedicated to:

*Sun Hongfang - my dear mother
Zhao Bichen - my beloved father*

*In teaching me to grow up
You have spent your hard life*

*Your son
Zhao Jiong*

献给

孙洪芳 - 我亲爱的母亲
赵碧臣 - 我敬爱的父亲

在教诲我成长过程中
度过了你们艰辛一生

你们的儿子
赵炯

"RTFSC - Read The F**king Source Code :)!"

-Linus Benedict Torvalds

Table of Contents

PREFACE.....	1	5.3	LINUX KERNEL MEMORY MANAGEMENT	178
THE MAIN GOAL OF THIS BOOK	1	5.4	INTERRUPT MECHANISM	193
FEATURES OF THIS BOOK	1	5.5	LINUX SYSTEM CALLS	199
OTHER BENEFITS OF READING EARLY KERNEL CODE....	2	5.6	SYSTEM TIME AND TIMING	201
THE IMPORTANCE AND NECESSITY OF READING THE		5.7	LINUX PROCESS CONTROL.....	203
COMPLETE CODE.....	3	5.8	HOW TO USE THE STACK IN LINUX.....	215
HOW TO SELECT THE KERNEL CODE VERSION TO READ	3	5.9	FILE SYSTEM FOR LINUX 0.12.....	219
THE BASIC KNOWLEDGE REQUIRED BY THE BOOK	4	5.10	DIRECTORIES OF KERNEL SOURCE CODE.....	220
IS READING AN EARLIER VERSION OUT OF DATE?	5	5.11	THE KERNEL CODE AND USER PROGRAMS .	229
EXT FILE SYSTEM AND MINIX FILE SYSTEM	5	5.12	LINUX/MAKEFILE.....	230
		5.13	SUMMARY	236
1 OVERVIEW	7	6 BOOTING SYSTEM.....	237	
1.1 THE BIRTH AND DEVELOPMENT OF LINUX	7	6.1	MAIN FUNCTIONS	237
1.2 CONTENT REVIEW	15	6.2	BOOTSECT.S	239
1.3 SUMMARY	20	6.3	SETUP.S	254
2 MICROCOMPUTER STRUCTURE	21	6.4	HEAD.S	286
2.1 THE MICROCOMPUTER COMPOSITION	22	6.5	SUMMARY	299
2.2 I/O PORT ADDRESSING & ACCESS CONTROL ..	23	7 INITIALIZATION PROGRAM (INIT)	301	
2.3 MAIN MEMORY, BIOS AND CMOS MEMORY	26	7.1	MAIN.C	301
2.4 CONTROLLERS AND CONTROL CARDS.....	28	7.2	ENVIRONMENT INITIALIZATION	316
2.5 SUMMARY	38	7.3	SUMMARY	318
3 KERNEL PROGRAMMING LANGUAGE AND		8 KERNEL CODE (KERNEL)	319	
ENVIRONMENT	39	8.1	MAIN FUNCTIONS	319
3.1 AS86 ASSEMBLER	39	8.2	ASM.S	322
3.2 GNU AS ASSEMBLER	46	8.3	TRAPS.C.....	329
3.3 C LANGUAGE PROGRAM.....	58	8.4	SYS_CALL.S.....	335
3.4 INTERWORKING BETWEEN C AND ASSEMBLY		8.5	MKTIME.C.....	349
LANGUAGE	67	8.6	SCHED.C.....	351
3.5 LINUX 0.12 OBJECT FILE FORMAT	76	8.7	SIGNAL.C.....	373
3.6 MAKE COMMAND AND MAKEFILE	87	8.8	EXIT.C	391
3.7 SUMMARY.....	93	8.9	FORK.C	404
4 80X86 PROTECTION MODE AND ITS		8.10	SYS.C.....	413
PROGRAMMING	94	8.11	VSPRINTF.C.....	429
4.1 80X86 SYSTEM REGISTERS AND SYSTEM		8.12	PRINTK.C	438
INSTRUCTIONS	94	8.13	PANIC.C	439
4.2 PROTECT MODE MEMORY MANAGEMENT ..	101	8.14	SUMMARY	440
4.3 SEGMENTATION MECHANISM	106	9 BLOCK DEVICE DRIVER	441	
4.4 PAGING	119	9.1	MAIN FUNCTIONS	442
4.5 PROTECTION	124	9.2	BLK.H	446
4.6 INTERRUPT AND EXCEPTION HANDLING	136	9.3	HD.C.....	451
4.7 TASK MANAGEMENT	147	9.4	LL_RW_BLK.C	477
4.8 THE INITIALIZATION OF PROTECTED MODE	157	9.5	RAMDISK.C.....	485
4.9 A SIMPLE MULTITASK KERNEL EXAMPLE ..	161	9.6	FLOPPY.C	491
4.10 SUMMARY	173	9.7	SUMMARY	521
5 LINUX KERNEL ARCHITECTURE	175	10 CHARACTER DEVICE DRIVER	523	
5.1 LINUX KERNEL MODE	175	10.1	MAIN FUNCTIONS	523
5.2 LINUX KERNEL SYSTEM ARCHITECTURE.....	176	10.2	KEYBOARD.S.....	535

10.3	CONSOLE.C	555	14.11	TERMIOS.H	947
10.4	SERIAL.C	594	14.12	TIME.H	954
10.5	RS_IO.S	603	14.13	UNISTD.H	956
10.6	TTY_IO.C	608	14.14	UTIME.H	963
10.7	TTY_IOCTL.C	626	14.15	FILES IN THE INCLUDE/ASM/ DIRECTORY	964
10.8	SUMMARY	635	14.16	IO.H	964
11	MATH COPROCESSOR (MATH)	637	14.17	MEMORY.H	965
11.1	FUNCTION DESCRIPTION	637	14.18	SEGMENT.H	966
11.2	MATH-EMULATION.C	647	14.19	SYSTEM.H	968
11.3	ERROR.C	660	14.20	FILES IN THE DIRECTORY INCLUDE/LINUX/ ..	973
11.4	EA.C	661	14.21	CONFIG.H	973
11.5	CONVERT.C	665	14.22	FDREG.H	975
11.6	ADD.C	670	14.23	FS.H	977
11.7	COMPARE.C	673	14.24	HDREG.H	982
11.8	GET_PUT.C	675	14.25	HEAD.H	985
11.9	MUL.C	682	14.26	KERNEL.H	986
11.10	DIV.C	684	14.27	MATH_EMU.H	987
11.11	SUMMARY	686	14.28	MM.H	991
12	FILE SYSTEM (FS)	689	14.29	SCHED.H	993
12.1	MAIN FUNCTIONS	689	14.30	SYS.H	1002
12.2	BUFFER.C	708	14.31	TTY.H	1004
12.3	BITMAP.C	728	14.32	HEADER FILES IN THE INCLUDE/SYS/ DIRECTORY	1008
12.4	TRUNCATE.C	735	14.33	PARAM.H	1008
12.5	INODE.C	738	14.34	RESOURCE.H	1009
12.6	SUPER.C	752	14.35	STAT.H	1011
12.7	NAMEI.C	763	14.36	TIME.H	1013
12.8	FILE_TABLE.C	793	14.37	TIMES.H	1014
12.9	BLOCK_DEV.C	793	14.38	TYPES.H	1015
12.10	FILE_DEV.C	798	14.39	UTSNAME.H	1016
12.11	PIPE.C	802	14.40	WAIT.H	1017
12.12	CHAR_DEV.C	807	14.41	SUMMARY	1018
12.13	READ_WRITE.C	810	15	LIBRARY FILES (LIB)	1019
12.14	OPEN.C	817	15.1	_EXIT.C	1020
12.15	EXEC.C	826	15.2	CLOSE.C	1021
12.16	STAT.C	845	15.3	CTYPE.C	1021
12.17	FCNTL.C	848	15.4	DUP.C	1022
12.18	IOCTL.C	852	15.5	ERRNO.C	1023
12.19	SELECT.C	854	15.6	EXECVE.C	1023
12.20	SUMMARY	868	15.7	MALLOC.C	1024
13	MEMORY MANAGEMENT (MM)	869	15.8	OPEN.C	1032
13.1	MAIN FUNCTIONALITIES	869	15.9	SETSID.C	1034
13.2	MEMORY.C	879	15.10	STRING.C	1034
13.3	PAGE.S	901	15.11	WAIT.C	1035
13.4	SWAP.C	902	15.12	WRITE.C	1036
13.5	SUMMARY	912	15.13	SUMMARY	1037
14	HEADER FILES (INCLUDE)	913	16	BUILDING KERNEL (TOOLS)	1039
14.1	FILES IN THE INCLUDE/ DIRECTORY	914	16.1	BUILD.C	1039
14.2	A.OUT.H	915	16.2	SUMMARY	1047
14.3	CONST.H	926	17	EXPERIMENTAL ENVIRONMENT SETTINGS AND USAGE	1048
14.4	CTYPE.H	926	17.1	BOCHS SIMULATION SOFTWARE	1049
14.5	ERRNO.H	928	17.2	RUNNING LINUX 0.1X SYSTEM IN BOCHS ..	1054
14.6	FCNTL.H	930	17.3	ACCESS INFORMATION IN A DISK IMAGE FILE	1059
14.7	SIGNAL.H	931	17.4	COMPILING AND RUNNING THE SIMPLE KERNEL	1062
14.8	STDARG.H	934			
14.9	STDDEF.H	936			
14.10	STRING.H	937			

17.5	USING BOCHS TO DEBUG THE KERNEL	1065
17.6	CREATING A DISK IMAGE FILE	1073
17.7	MAKING A ROOT FILE SYSTEM	1076
17.8	COMPILE KERNEL ON LINUX 0.12 SYSTEM	1084
17.9	COMPILE KERNEL UNDER REDHAT SYSTEM	1085
17.10	INTEGRATED BOOT DISK AND ROOT FS	1089
17.11	DEBUGGING KERNEL CODE WITH GDB AND BOCHS	1094
17.12	SUMMARY.....	1100

REFERENCES.....	1101
------------------------	-------------

APPENDIX.....	1103
----------------------	-------------

A1	ASCII CODE TABLE	1103
A2	COMMON C0, C1 CONTROL CHARACTERS	1104
A3	ESCAPE AND CONTROL SEQUENCES	1106
A4	THE FIRST SET OF KEYBOARD SCAN CODE	1109

Preface

Under the general trend of intelligent manufacturing and networking direct control of objects, the Linux operating system has become the most important basic platform for operation control in today's embedded systems. This book is a primer on the basic workings of the Linux operating system kernel.

The main goal of this book

The main goal of this book is to use a minimal amount of space or within a limited space to dissect the complete Linux kernel source code in order to obtain a full understanding of the basic functions and actual implementation of the operating system. To achieve a complete and profound understanding of the Linux kernel, a true understanding and introduction of the basic operating principles of the Linux operating system.

This book's readership is positioned to know the general use of Linux systems or has a certain programming basis, but it lacks the basic knowledge to read the current new kernel code and is eager to understand the working principle and actual code of the UNIX operating system kernel as soon as possible.

Features of this book

At the time of writing this book, there are books on the market that describe the Linux kernel that try to use the newer Linux kernel version (such as version 2.6.24 used by Fedora 8) to describe the kernel working mechanism. However, since the size of kernel source code is already very large (for example, 2.2.20 version has 2.68 million lines!), these books can only selectively explain and describe the Linux kernel source code, and many system implementation details are ignore. Therefore, it is difficult to have a clear and complete description of the Linux kernel.

The book "Linux Kernel Source Code Analysis" written by Scott Maxwell is basically oriented to the advanced level readers of Linux. It needs a more comprehensive basic knowledge to fully understand. And may be due to space limitations, the book does not comment on all the Linux kernel code, omitted a lot of kernel implementation details, such as the various header files used in the kernel (*.h), the tool to generate the kernel code image file The role of the program, each make file, and its implementation are not covered. Therefore, reading the book is difficult for readers who are at entry level.

The book "Leon's UNIX source code analysis" written by John Lions is a good book for learning UNIX source code of the operating system kernel, but because it uses the UNIX version V6, some of the code in the system call is With the assembler language of the long-deprecated PDP-11 series machine, it is difficult to conduct experiments when reading and understanding the source code related to the hardware part.

Andrew S. Tanenbaum's book "Operating Systems: Design and Implementation" is a good primer on operating system kernel implementation, but the MINIX system described in this book is a message-based kernel implementation mechanism, and Linux There are differences in the implementation of the kernel. Therefore, after learning this book, it is not very easy to start working on the newer Linux kernel source code.

When using these books for learning, there will be a feeling of "blind people feel like elephants". It is not

easy to understand the overall concept of the specific implementation of the Linux kernel system, especially when the Linux system beginners use those books to learn the principle of the kernel, the overall operating structure of the kernel. It cannot be clearly formed in the mind. This has profound experience in my many years of experience in the Linux kernel learning. In October 1991, Linux founder Linus Torvalds mentioned the same problem in an article written during the development of Linux version 0.03. In this article titled "LINUX--a free unix-386 kernel", he said: "The development of Linux is for the use, learning and entertainment of those operating system enthusiasts and computer science students." Today's popular Linux systems have become larger and more complex, so they are no longer suitable as a starting point for beginners learning the operating system.

In order to fill this vacancy, this book uses a minimal amount of space or within a limited space to conduct a complete dissection of the complete Linux kernel source code in order to obtain a full understanding of the basic functions and actual implementation of the operating system. To achieve a complete and profound understanding of the Linux kernel, a true understanding and introduction of the basic operating principles of the Linux operating system.

Other benefits of reading early kernel code

At present, there have been many kernel versions developed specifically for embedded systems based on Linux's early kernels, such as DJJ's x86 operating system, Uclinux, etc. Many people in the world also realize the benefits of learning through the early Linux kernel source code. At present, people in China are already organizing human annotations to publish books similar to this article. Therefore, by reading the source code of the Linux kernel version earlier, it is indeed an effective way to learn the Linux system, and it is also very helpful for the research and application of the Linux embedded system.

In commenting on early kernel source code, the author found that early kernel source code was almost a condensed version of the newer kernels in use today. It already includes almost all the basic functional principles of the current version. As Leland L. Beck, author of "System Software: An Introduction to System Programming," introduced system programs and operating system design, he introduced an extremely simplified Simple Instruction Computer (SIC) system to illustrate the design and implementation of all system programs. The principle, which not only avoids the complexity of the actual computer system, but also a thorough description of the problem. Here, select the early kernel version of Linux as a learning object, and its guiding ideology is the same as that of Leland. This is one of the best choices for beginners of Linux kernel learning. The basic working principle of the Linux kernel can be deeply understood in the shortest possible time.

For those who are already familiar with the working principle of the kernel, it is necessary to read the kernel source code in order to allow the actual operation mechanism of the system in the actual work to produce no feeling of castle in the air.

Of course, using the early kernel as a learning object also has its disadvantages. The selected Linux early kernel version does not include support for virtual file system VFS, support for network systems, support for only a.out executable files, and description of complex subsystems in some other existing kernels. However, since this book is an introductory textbook that is used as a working mechanism for the Linux kernel, this is one of the advantages of choosing an earlier kernel version. By studying this book, you can lay a solid foundation for further studying these advanced contents.

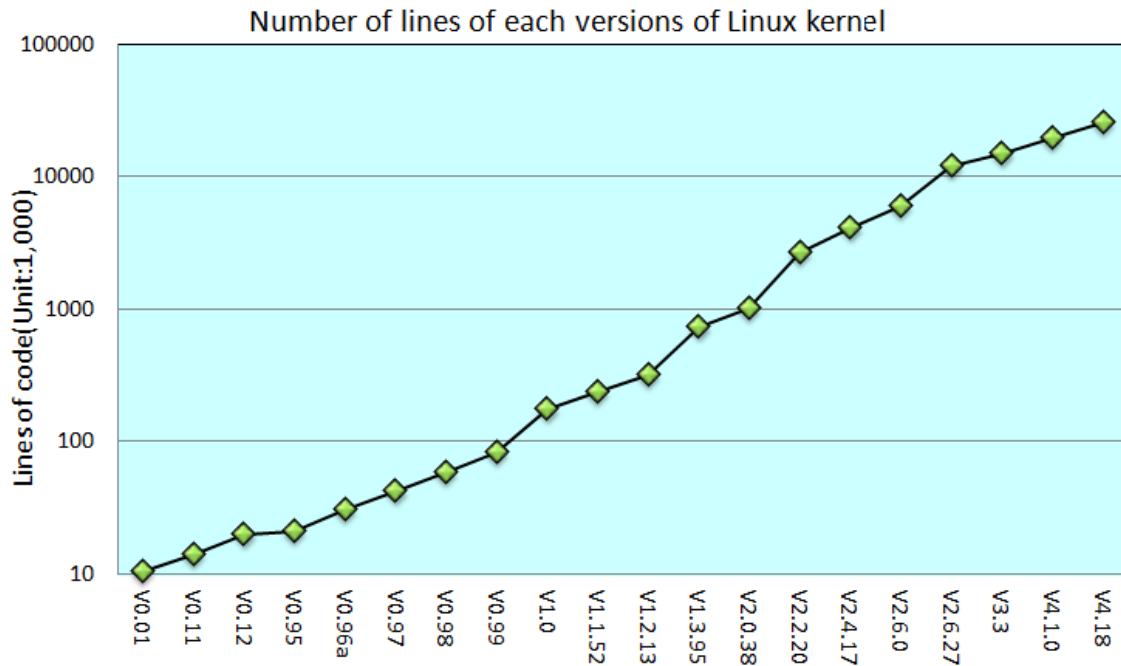
The importance and necessity of reading the complete code

Just as the founder of the Linux system stated in a newsgroup submission, to understand the true operating mechanism of a software system, be sure to read its source code (RTFSC – Read The F**king Source Code). The system itself is a complete whole, with many seemingly unimportant details. However, ignoring these details will make it difficult to understand the entire system and fail to truly understand the implementation method and means of an actual system.

Although some classic books on operating system principles (such as Mr. MJBach's "UNIX Operating System Design") can be used to theoretically guide the working principle of the UNIX-like operating system, the actual composition of the operating system is still not very clear. The understanding of the realization of internal relations is still not very clear. As Andrew S. Tanenbaum said, "many operating system textbooks are theoretical and light practice." "Most books and courses consume a lot of time and space for scheduling algorithms and completely ignore I/O. In fact, the former is usually less than one page of code. The latter often has to account for one-third of the total code of the entire system." A large number of important details in the kernel are not mentioned. Therefore, it does not allow readers to understand the true beauty of a real operating system. Only after reading the complete kernel source code in detail will there be a sense of openness to the system and a deep understanding of the entire system's operational process. When you choose the newest or newer kernel source code to learn later, you will not encounter major problems and basically will be able to understand the new code content smoothly.

How to select the kernel code version to read

So, how can we choose to meet the above requirements without being confused by too much content and choose a suitable version of the Linux kernel to learn and improve the efficiency of learning? After comparing and selecting a large number of kernel versions, the author finally chose the 0.12 kernel that is similar to the current basic functions of the Linux kernel and is very short, as the best version for getting started. The following figure shows the statistics for some major Linux kernel version lines.



The current Linux kernel source code amount is in the number of millions of lines, the 2.6.0 version of the kernel code line is about 5.92 million lines, and the 4.18.X version of the kernel code is extremely large, and it has exceeded 25 million lines! So it is almost impossible to fully annotate and elaborate on these kernels. The 0.12 version of the kernel does not exceed 20,000 lines of code, so it can be explained and commented clearly in a book. Small but complete. In order to have an inductive understanding of the system under study and to use experiments to deepen the understanding of the principle, the author has also specifically rebuilt the Linux 0.12 system that is based on this kernel. Since it contains the GNU gcc compilation environment, using this system can also do some simple development work.

In addition, the use of this version can avoid the use of existing newer kernel versions that have become more and more complicated to study the various subsystems (such as virtual file system VFS, ext2, or ext3 file systems, network subsystems, new complex Memory management mechanisms, etc.)

The basic knowledge required by the book

When reading this book, readers must have some basic C language knowledge and Intel CPU assembly language knowledge. The best reference for C language is still the book "The C Programming Language" written by Mr. Brain W. Kernighan and Mr. Dennis M. Ritchie. Assembler language data can refer to any assembly language textbook that explains Intel CPU. Also need some embedded assembly language information. The authoritative information about embedded assembly is contained in the GNU gcc compiler manual. We can also search for some valuable essays about embedded assembly from the Internet. The book also contains some basic syntax descriptions for inline assembly (Section 5.5).

In addition, I hope readers have the following basic knowledge or related reference books. One is knowledge or information about 80x86 processor architecture and programming. For example, the 80x86 programming manual (INTEL 80386 Programmer's Reference Manual) can be downloaded from the Internet; the second is about 80x86 hardware architecture and interface programming knowledge or information. There is a lot of information in this regard; the third should also have the simple skills of using the Linux system at the

beginning.

Since Linux kernel implementation was first developed according to the basic principles of the "UNIX operating system design" book, many variables or function names in the source code come from the book. Therefore, if you read this book properly, it will be easier to understand the kernel source code.

When Linus first developed the Linux operating system, he referred to the MINIX operating system. For example, the original Linux kernel version completely copied the MINIX 1.0 file system. Therefore, when reading this book, A.S. Tanenbaum's book "Operating System: Design and Implementation" also has great reference value.

Is reading an earlier version out of date?

On the surface, the book notes the contents of Linux's early kernel version as if the Linux operating system has just been released. Tanenbaum thinks that it is outdated (Linux is obsolete). However, by studying the content of this book, you will find that because of the small amount of early kernel source code and lean, using this book to learn the Linux kernel will have a very high learning efficiency, can do more with less, and get started quickly. And lay a solid foundation for continuing to further select the source code of the new kernel part. After completing this book, you will have a very complete and practical concept of how the system works. This complete concept makes it easy to further select and learn any part of the new kernel source code without having to Read the complete source code in the new kernel with a large amount of code.

Ext file system and MINIX file system

The Ext3 file system currently used on Linux systems was developed after kernel 1.x. Its function is detailed and its performance is also very complete and stable. It is the default standard file system on the current Linux operating system. However, as part of the introductory learning of the full working principle of the Linux operating system, in principle, the more streamlined the better. In order to achieve a complete understanding of an operating system, and without being overwhelmed by the complex and excessive details of the various subsystems, the principle of choosing the kernel version for learning is as simple as possible, as long as the system code can explain the actual working principle. The Linux kernel version 0.12 contained only the simplest MINIX 1.0 file system at the time, which is enough to understand the actual composition and working principle of a file system in an operating system. This is one of the main reasons to choose the early Linux kernel version for learning.

After reading this book in its entirety, I believe you will send this kind of sigh: "For the Linux kernel system, I'm finally getting started!" At this point, you should have enough confidence to further study the working principle and process of each part of the latest Linux kernel.

Dr. Zhao Jiong
Tongji University
2019.1

1 Overview

This chapter first reviews the process of the birth, development, and growth of the Linux operating system. This can be used to understand why the book chose an earlier version of the Linux system as a learning object. It then explains in detail the advantages and disadvantages of choosing an early version of the Linux kernel for learning and how to begin further learning. Finally, we briefly introduced the contents of each chapter.

1.1 The birth and development of Linux

Linux is a clone system of the UNIX operating system. It was born on October 5, 1991 (this is the time for the first official announcement). Since then, with the aid of the Internet, through the joint efforts of computer enthusiasts all over the world, it has now become the most widely used type of UNIX operating system in the world, and the number of users is still growing rapidly.

The birth, development, and growth of the Linux operating system depend on the following five pillars: the UNIX operating system, the MINIX operating system, the GNU project, the POSIX standard, and the Internet network. Based on these five basic clues, we follow the development history of Linux, its brewing process and its initial development. First of all, I will introduce the four basic elements, and then follow Linux founder Linus Torvalds to learn computer knowledge from her own interest in computers, start brewing her own operating system, release to the initial release of the Linux kernel version 0.01, and how difficult it will be. Step by step, with the help of hackers all over the world, the development of the more mature version 1.0 was finally introduced. It also describes the history of Linux's early development in detail.

Of course, the current Linux kernel version has been developed to version 4.18.x. The kernel used in most Linux systems is a stable 4.4.x-4.16.x kernel (where the second digit is an odd number, it means that it is being developed and cannot guarantee system stability). For the general history of Linux development, many articles and books have been introduced and will not be repeated here.

1.1.1 The birth of the UNIX operating system

The Linux operating system is a cloned version of the UNIX operating system. The UNIX operating system is a time-sharing operating system developed by Bell Labs's Ken. Thompson and Dennis Ritchie on the DEC PDP-7 minicomputer in the summer of 1969.

In order to be able to run his favorite Star Travel game on an idle PDP-7 computer, Ken Thompson developed UNIX operations within a month in the summer of 1969 while his wife returned home to California. The prototype of the system. At that time, the BCPL language (basic combination programming language) was used. After being rewritten by Dennis Ritchie in 1972 with a highly portable C language, the UNIX system was promoted in universities and colleges.

1.1.2 MINIX operating system

The MINIX system was developed by Andrew S. Tanenbaum (AST). AST is a mathematics and computer science system at Vrije University in Amsterdam, the Netherlands. He is a senior member of ACM and IEEE (only a few people in the world are senior members of the two associations). A total of more than 100 articles and 5 computer books were published.

Although AST was born in New York, it was a Dutch expatriate (his grandfather came to the United States in 1914). He studied at a high school in New York, a university at M.I.T, and a doctorate at the Berkeley campus of the University of California. Due to his postdoctoral studies, he came to his hometown of the Netherlands. Since then, it has been with the hometown. Later, I started teaching and graduate school at Vrije University. Amsterdam, the capital city of the Netherlands, is a year-round rainy city, but for the AST, this is best, because in this environment he can often stay at home to play with his computer.

MINIX was created in 1987 and is mainly used by students to learn operating system principles. By 1991 the version was 1.5. There are currently two major versions in use: Version 1.5 and Version 2.0. At that time, the operating system was free at university, but other uses were not. Of course, the current MINIX system is free and can be downloaded from many FTP sites.

For the Linux system, he later expressed his compliments to the developer Linus. However, he believes that the development of Linux is largely due to the fact that in order to keep MINIX small, he will be able to complete the learning within one semester, thus failing to accept the expansion requirements of MINIX from many people around the world. So under this premise inspired Linus to write a Linux system. Of course, Linus also just caught this good time.

As an operating system, MINIX is not an excellent one, but it also provides system source code written in C and assembly language. This is the first time that aspiring programmers or hackers have been able to read the operating system's source code. At the time, this source code was a secret that software vendors had been carefully guarding.

1.1.3 GNU Project

The GNU Project and the Free Software Foundation were founded by Richard M. Stallman in 1984 to develop a complete operating system similar to UNIX and free software: the GNU system (GNU is "GNU's Not". Recursive abbreviation for Unix, which is pronounced "guh-NEW"). Various GNU operating systems using Linux as the core are being widely used. Although these systems are often referred to as "Linux," Stallman believes that, strictly speaking, they should be referred to as GNU/Linux systems.

By the early 1990s, the GNU project had developed many high-quality free software, including the famous emacs editing system, bash shell program, gcc series compiler, gdb debugger and so on. These softwares create a suitable environment for the development of the Linux operating system. This is one of the foundations for the birth of Linux, so that many people now refer to the Linux operating system as the "GNU/Linux" operating system.

1.1.4 POSIX standard

POSIX (Portable Operating System Interface for Computing Systems) is a cluster of standards developed by IEEE and ISO/IEC. The standard is based on existing UNIX practices and experiences and describes the operating system's call service interface. The applications used to ensure the compilation can be ported and run on multiple operating systems at the source code level. It was based on the early work of a UNIX user group (usr/group) in the early 1980s. The UNIX user group originally attempted to re-integrate the distinction between AT&T's System V operating system and Berkeley CSRG's BSD operating system's call interface. In 1984, the /usr/group standard was customized.

In 1985, the IEEE Operating System Technical Committee Standards Subcommittee (TCOS-SS) began, under the aegis of ANSI, instructing the IEEE Standards Committee to establish a formal standard for program source code portability operating system service interfaces. In April 1986, the IEEE developed a trial standard. The first formal standard was approved in September 1988 (IEEE 1003.1-1988), and also the POSIX.1 standard

that is often mentioned later.

By 1989, POSIX's work was transferred to the ISO/IEC community and the 15 working group continued to develop it as an ISO standard. By 1990, POSIX.1, in conjunction with the already adopted C language standard, was formally approved as the IEEE 1003.1-1990 (also ANSI) and ISO/IEC 9945-1:1990 standards.

POSIX.1 only specifies system service application programming interfaces (APIs) and only summarizes basic system service standards. Therefore, the working group expects to establish standards for other functions of the system. So the work of IEEE POSIX began. Ten approval plans were in progress at the beginning, and nearly 300 people participated in the quarterly one-week meeting. The work that started was command and tool standard (POSIX.2), test method standard (POSIX.3), and real-time API (POSIX.4). In the first half of 1990, 25 plans were already in progress and 16 working groups were involved. At the same time, some organizations are also developing similar standards such as X/Open, AT&T, and OSF.

In the early 1990s, the formulation of the POSIX standard was in the final stage of voting, which was between 1991 and 1993. At this point, when Linux was just starting out, this UNIX standard provided extremely important information for Linux, enabling Linux to be developed under the guidance of standards and compatible with most UNIX operating systems. In the original Linux kernel source code (versions 0.1, 0.11, and 0.12) the Linux system was ready for compatibility with the POSIX standard. Several symbolic constants for POSIX standard requirements have been defined in the `/include/unistd.h` file of the Linux version 0.01 kernel, and Linus wrote in the comment: "OK, this may be a joke, but I'm working on it. It does."

On July 3, 1991, Linus mentioned on the post posted on `comp.os.minix` that it is collecting POSIX data. It revealed that he is working on the development of an operating system, and at the beginning of development, he had thought of the problem of compatibility with POSIX.

1.1.5 The birth of the Linux operating system

In 1981, IBM introduced the world-renowned microcomputer IBM PC. Between 1981 and 1991, the MS-DOS operating system was always the master of the microcomputer operating system. At this time, although the price of computer hardware has decreased year by year, software prices have remained high. At that time, Apple's MACs operating system can be said to be the best performance, but its price is so that no one can easily get close.

Another computer technology camp at the time was the UNIX world. However, the UNIX operating system is not only an expensive issue. In order to seek high profit margins, UNIX dealers have pushed prices extremely high, and PC users can't get close to it. The UNIX source code that once received permission from Bell Labs to be used for teaching in the university has also been carefully guarded against disclosure. For the majority of PC users, large vendors in the software industry have never given effective solutions to this problem.

At this time, the MINIX operating system appeared, and a book describing the principles of its design and implementation was issued at the same time. Since this book written by AST was very detailed and well-articulated, almost all computer enthusiasts around the world began to read this book in order to understand how the operating system works. It also includes Linus Benedict Torvalds, the founder of the Linux system. At that time (in 1991), he was a sophomore in the Department of Computer Science at the University of Helsinki and a self-taught computer hacker. The 21-year-old Finnish young man likes to drum up his computer and test the performance and limitations of the computer. But what he lacked at the time was a professional-level operating system.

During the same year, the GNU program has developed a number of software tools. The most anticipated

GNU C compiler has emerged, but the free GNU operating system has not yet been developed. Even the MINIX operating system used in teaching has begun to have copyright, and it is necessary to purchase it to get the source code. Although the GNU operating system HURD has been under development, it did not appear to have been completed within a few years.

To better learn computer knowledge (perhaps just for interest), Linus purchased a 386-compatible computer using Christmas lucky money and loans, and mailed a MINIX system software from the United States. While waiting for the MINIX software, Linus learned the hardware knowledge of Intel 80386 seriously. In order to be able to connect to the school's mainframe through a Modem dial-up, he uses assembly language and uses the multitasking features of the 80386 CPU to create a terminal emulation program. Later, in order to copy his own software on an old computer to a new computer, he also compiled drivers for floppy disk drives, keyboards, and other hardware devices.

Through programming practice and recognizing the many limitations of the MINIX system during the learning process (MINIX is good, but it is only a simple operating system for teaching purposes, rather than a powerful and practical operating system), Linus already has something similar. The code for the operating system device driver, so he began to have the idea of a new operating system. At this point, the GNU Project has developed many tools and software, among which the most anticipated GNU C compiler has appeared. Although GNU's free operating system HURD is under development. But Linus has waited for no hurry.

From April 1991, he began to develop his own operating system by modifying the terminal emulation program and hardware drivers. At the beginning, his purpose was simple, just to learn the programming techniques of the Intel 386 architecture protection mode operation. However, the development of Linux has completely changed its original intention. According to Linus's news release on the comp.os.minix newsgroup, we can see that he has gradually evolved from learning the MINIX system stage to developing his own Linux system.

Linus delivered the message to comp.os.minix for the first time on March 29, 1991. The title of the posted post is "gcc on minix-386 doesn't optimize". It is about the gcc compiler running optimized on the MINIX-386 system (MINIX-386 is an improvement from Bruce Evans using Intel 386 features 32 Bit MINIX system). From this it can be seen that Linus had already begun to study the MINIX system in depth in early 1991, and during this time there has been an improvement of the MINIX operating system. After further learning about the MINIX system, this idea gradually evolved into the idea of redesigning a new operating system based on the Intel 80386 architecture.

When he answered someone's question on MINIX, the first sentence said was "Read the F**ing Source Code :-)"). He thinks the answer lies in the source program. This also shows that for the learning system software, we not only need to understand the basic working principles of the system, but also need to combine the actual system to learn how to implement the actual system. After all, theory is a theory, in which many branches are omitted. Although these branch problems do not have much theoretical content, they are a necessary part of the system, just like a feather in a sparrow.

From April 1991, Linus spent almost all of his time researching the MINIX-386 system (Hacking the kernel) and trying to port GNU software to the system (GNU gcc, bash, gdb, etc.). And announced on comp.os.minix on April 13 that he had successfully ported bash to MINIX, and he could not afford to leave the shell software.

The first Linux-related news was released on comp.os.minix on July 3, 1991. (Of course, there was no such name as Linux at that time. Linus thought that the name might be FREAK , FREAX. The English meaning is grotesque, monsters, whimsical, etc.). It revealed that he is developing Linux system and he has already thought of the problem of compatibility with POSIX.

In another announcement by Linus (comp.os.minix, August 25, 1991), he asked all MINIX users "What do you most want to see in the MINIX system?" ("What would you like to see?" In minix?), in which he revealed for the first time that a (free) 386(486) operating system is being developed, and that he is only interested in it. The code will not be large and will not be as professional as GNU. I hope you will give us some feedback on what features the MINIX system likes and dislikes, and explain that due to practical and other reasons, the newly developed system is just like MINIX (and uses MINIX's file system). And it has successfully ported bash (version 1.08) and gcc (version 1.40) to the new system and it will be practical in months.

Finally, Linus stated that the operating system he developed does not use a single line of MINIX source code; because of the task switching feature of the 386, the operating system is not portable (no portability) and only AT hard disks are used. Linus did not consider the issue of Linux portability. But at present, Linux can run on almost any kind of hardware architecture.

On October 5th, 1991, Linus published a message on the comp.os.minix newsgroup, officially announcing the birth of the Linux kernel system (Free minix-like kernel sources for 386-AT). This news can be called the birth declaration of Linux, and has been widely circulated. Therefore, October 5 was a special day for the Linux community, and many later Linux versions had chosen this date. So RedHat chose this day to release its new system is not accidental.

1.1.6 Linux operating system version changes

Since the birth of the Linux operating system to the 1.0 release, a number of major releases have been released as shown in Table 1–1. Linus looked at all of the previous versions of 1.0 when he started learning to use the version management tool BitKeeper in September 2003. In fact, the Linux system does not have this version of 0.00, but since Linus' experiment on his own 80386 compatible machine succeeded in switching between two tasks under the control of clock interruption, he further enhanced his idea of developing his operating system to some extent. . Therefore we are also listed as a version. The Linux version of the kernel version was completed on September 17, 1991. However, Linus has no copyright awareness at all, so only one copy of copyright information appears in this version of the include/string.h file. The keyboard driver for this version of the kernel is hard-coded only into Finnish code, so only the Finnish keyboard is supported. Only 8MB physical memory is supported. Due to a mistake by Linus, the subsequent 0.02, 0.03 version of the kernel source code was destroyed and lost.

Table 1-1 Earlier major versions of the kernel

Version No.	Release date	Description
0.00	1991.2-4	The two processes display 'AAA...' and 'BBB...' on the screen, respectively. (Note: No release)
0.01	1991.9.17	The first official release of the Linux kernel version. Multi-threaded file system, segmentation, and paging memory management. Does not include floppy disk drivers yet.
0.02	1991.10.5	This version and version 0.03 is an internal version that is currently unavailable. Features the same as above.
0.10	1991.10	The Linux kernel version released by Ted Ts'o. Added memory allocation library functions. The boot directory contains a script that converts as86 assembler syntax to gas assembler syntax.
0.11	1991.12.8	Basically functioning kernel. Supports hard disk and floppy drive devices as well as

		serial communications.
0.12	1992.1.15	The more stable version mainly increases the software simulation program of the math coprocessor. Added job control, virtual console, file symbolic links, and virtual memory swapping capabilities.
0.95.x (ie 0.13)	1992.3.8	Virtual file system support was added in this version, but it still contains only one MINIX file system. Added login functionality. Improves the performance of floppy disk drivers and file systems. Changed hard disk naming and numbering. The original naming method is the same as that of the MINIX system. At this time, it is the same as the current Linux system. Support CDROM.
0.96.x	1992.5.12	Began to add UNIX Socket support. Added ext file system alpha tester. SCSI drivers are officially added to the kernel. Floppy disk type is automatically recognized. Improved serial driver, cache, memory management performance, support for dynamic link libraries, and the ability to run X-Windows programs. The keyboard driver written in the original assembly language has been rewritten with C. Compared with the 0.95 kernel code, there are great changes.
0.97.x	1992.8.1	Added support for new SCSI drivers; dynamic caching; msdos and ext file system support; bus mouse drivers. The kernel is mapped to the beginning of the linear address 3GB.
0.98.x	1992.9.28	Improve support for TCP/IP (0.8.1) networks and correct extfs errors. Rewritten memory management section (mm), each process has 4GB of logical address space (the kernel occupies 1GB). Starting from 0.98.4, each process can open 256 files at the same time (originally 32), and the process's kernel stack uses a single memory page independently.
0.99.x	1992.12.13	Re-design the process of the use of memory allocation, each process has 4G linear space. Constantly improving the network code. NFS support.
1.0	1994.3.14	The first official version.

The existing 0.10 version of the kernel code is a version of Ted Ts'o that was preserved at the time, Linus's own has also been lost. This version is a great improvement over the previous versions. On this version of the kernel system, GNU gcc has been used to compile the kernel, and has begun to support the operation of mounting/unmounting file systems. From this kernel version, Linus added copyright information for each file: "(C) 1991 Linus Torvalds". Some other changes in this version include: the original boot program boot/boot.s split into two programs boot/bootsect.s and boot/setup.s; 1 supports up to 16MB of physical memory; 2 drivers and memory management procedures Created their own subdirectories separately; 3 Added floppy driver; 4 Supported file read-ahead operations; 5 Supported dev/port and dev/null devices; 6 Rewritten kernel/signal.c code, added sigaction() Support etc.

Relative to the 0.10 version of the kernel, Linux 0.11 version of the changes are relatively small. However, this version is also the first stable version, and other people are beginning to participate in kernel development. The main additions in this version are: 1 load requirements for the execution program; 2 execute the /etc/rc initial file at startup; 3 build the math coprocessor simulation program frame program structure; 4 Ted Ts'o adds a script program The processing code; 5 Galen Hunt added support for multiple display cards; 6 John T Kohl modified the kernel/console.c program to enable the console to support tweet and KILL characters; 7 provides support for multilingual keyboards.

Linux 0.12 is a more satisfactory kernel version of Linus and a more stable kernel. During the Christmas

season in 1991, he compiled the virtual memory management code so that "large" software like gcc could be used on machines with only 2MB of memory. This version makes Linus feel that releasing the 1.0 kernel version is not something that is out of sight, so he immediately upgraded the next version (0.13 version) to version 0.95. Another implication of Linus's ability to do this is to make everyone not feel that they are still far from version 1.0. However, due to the hasty release of the 0.95 version, which also contains more errors, so when the 0.95 version was just released, there were more Linux enthusiasts encountered problems in use. At that time, Linus felt like he had encountered a catastrophe. However, he has accepted this lesson since then. Every time a new kernel version is released later, he will undergo more careful testing and let a few good friends try it out before officially publishing it. The main changes in the 0.12 version of the kernel are: 1 Ted Ts'o adds support for terminal signal processing; 2 can change the screen ranks used when starting up; 3 corrects a race condition caused by a file IO; 4 adds support for shared libraries Support, saving memory usage; 5 symbolic link handling; 6 deletion of directory system calls; 7 Peter MacDonald implements virtual terminal support, making Linux even superior to certain commercial versions of UNIX at the time; Function support, which was modified by Peter MacDonald based on patches provided by some people for MINIX, but MINIX did not adopt these patches; 9 re-executable system calls; 10 Linus compiled math coprocessor simulation code.

Version 0.95 was the first Linux kernel version to use the GNU GPL copyright. There are actually three sub-versions of this version. Due to some problems encountered when the first 0.95 release was released on March 8, 1992, another 0.95a version was immediately released in less than 10 days (March 17). And in a month later (April 9th), 0.95c+ was released again. The biggest improvement in this version is the introduction of the virtual file system VFS structure. Although only the MINIX file system was supported at the time, the program structure has been extensively adjusted to support multiple file systems. The code for the MINIX file system is put into a separate MINIX subdirectory. Some of the other changes in the 0.95 kernel include: 1 Added login interface; 2 Ross Biro added debugging code (ptrace); 3 Floppy disk drive track buffering; 4 Non-blocking pipeline file operations; 5 System restart (Ctrl-Alt-Del); Swapon() system call to select swap devices in real time; 6 support for recursive symbolic links; 7 support for 4 serial ports; 8 support for hard disk partitions; 9 support for more types of keyboards; 10 James Wiegand compiles initial parallel port drivers, etc. .

In addition, starting with the 0.95 release, many of the kernel's improvements (providing patches) were dominated by others, and Linus's main task began to become the maintenance of the kernel and decide whether to adopt a patch. Until now, the latest kernel version is version 4.16.16 released in June 2018. Its use of gz compressed source code package also has about 152MB! The latest version of each major stable release is shown in Table 1-2. Table 1-2

Table 1-2 New kernel source code size

Version number	Release date	Size (after gz compression)
2.0.40	2004.2.8	7.2 MB
2.2.26	2004.2.25	19 MB
2.6.25	2008.4.17	58 MB
3.0.10	2011.11.21	92MB
4.4.10	2016.5.11	127MB
4.16.16	2018.6.16	152MB

1.1.7 The reason for the Linux name

At the beginning of the Linux operating system, it was not called Linux. Linus named his operating system

FREAK. Its English meaning is grotesque, monster, whimsical. When he uploaded the new operating system to the ftp.funet.fi server, administrator Ari Lemke disliked the name very much. He believes that since it is Linus's operating system, take its homonym Linux as the operating system's directory, so the name of Linux began to pass down.

In Linus's autobiography "Just for Fun," Linus explains:

“Honest: I didn't want to ever release it under the name Linux because it was too egotistical . What was the name I reserved for any eventual release ? Freax. (Get it? Freaks with the requisite X.) In fact, some of the early make files-the files that describe how to compile the sources-included the word "Freax" for about half a year. But it really didn't matter. At that point I didn't need a name for it because I wasn't releasing it to anybody.”

“And Ari Lemke, who insured that it made its way to the ftp site, hated the name Freax. He preferred the other working name I was using-Linux-and named my posting: pub/OS/Linux. I admit that I didn't put up much of a fight. But it was his doing . So I can honestly say I wasn't egotistical, or half-honestly say I wasn't egotistical . But I thought, okay, that's a good name, and I can always blame somebody else for it, which I'm doing now.”

1.1.8 The main contributor to the development of early Linux systems

As can be seen from the early Linux source code, one of the most famous developers of the Linux system in addition to Linus himself is Theodore Ts'o (Ted Ts'o). He graduated from MIT Computer Science in 1990. In college time he actively participated in various student activities held in the school. He likes cooking, cycling, and of course Hacking on Linux. Later he began to like the amateur radio telegram campaign. He currently works at IBM on system programming and other important issues. He is also an IETF member of the International Network Design, Operations, Sales and Research Open Group.

The popularity of Linux in the world also has his great credit. As early as when the Linux operating system came out, he provided Maillist with great enthusiasm for the development of Linux. Almost since Linux was first released, he has been contributing to Linux. He was also the first person to add programs to the Linux kernel (the ramdisk.c virtual disk driver and the kernel memory allocation program kmalloc.c in the Linux kernel version 0.10). Until now he is still engaged in Linux-related work. In North America, he first established the Linux ftp site (tsx-11.mit.edu), and this site still provides services for the majority of Linux users. One of his biggest contributions to Linux was to propose and implement the ext2 file system. The file system has now become the de facto file system standard in the Linux world. Recently he introduced the ext3 file system. The system greatly improves the stability and access efficiency of the file system. As his admiration, the Linux Journal issue of the 97th issue (May 2002) used him as a cover character and interviewed him. He currently works for the IBM Linux Technology Center and is working on the Linux Standard Base (LSB).

Another famous person in the Linux community is Alan Cox. He originally worked at Swansea University College in Wales. At first, he particularly likes to play computer games, especially MUD (Multi-User Dungeon or Dimension). In the posts of games.mud news group in the early 90s you can find a lot of posts he posted. He even wrote a history of MUD development (rec.games.mud news group, March 9, 1992, A history of MUD). As MUD games are closely related to the internet, he slowly became fascinated with computer networks. In order to play the game and improve the speed of the computer running the game and the network transmission speed, he needs to choose a most satisfactory operating platform. So he began to contact various types of operating systems. Because of the lack of money, he could not afford even the MINIX system. When Linux 0.1x and 386BSD were released, he took a long time to purchase a 386SX computer. Since the 386BSD requires math

coprocessor support and the computer with the Intel 386SX CPU does not have a math coprocessor, he installed the Linux system. So he started to learn Linux with free source code and started to have interest in Linux systems, especially with regard to networking. In the discussion of Linux single-user mode of operation, he even praised Linux for being beautifully implemented.

After the release of Linux 0.95, he began writing patches (modification programs) for the Linux system (remembering that his two earliest patches were not adopted by Linus) and became the earliest users of TCP/IP network code on Linux systems. One. Later, he gradually joined the Linux development team and became one of the main responsible for maintaining the Linux kernel source code. It can also be said to be the most important figure in the Linux community after relaying Linus. Later Microsoft invited him to join, but he simply refused. Since 2001, he is responsible for maintaining the Linux kernel 2.4.x code. Linus is mainly responsible for the development of the latest development version of the kernel (odd version, such as 2.5.x version).

Michael K. Johnson, author of The Linux Kernel Hackers' Guide, was also one of the first people to contact the Linux operating system (from version 0.97). He is also one of the initiators of the well-known Linux Document Project (LDP). He once worked for Linux Journal and now works for RedHat.

The Linux system is not the only backbone that can develop into what it is today. There are many computer experts who have made great contributions to Linux. We will not list them here. The specific list of major contributors can be found in the CREDITS file in the Linux kernel, which lists in alphabetical order the list of more than 400 people who contributed significantly to Linux, including their email address and mailing address, home page, and major contributions. Deeds and other information.

Through the above explanation, we can sum up the above five pillars of Linux as follows:

- UNIX Operating System -- UNIX was born in Bell Labs in 1969. Linux is a UNIX clone system. The importance of UNIX goes without saying.
- The MINIX operating system -- The MINIX operating system is also a UNIX clone system. It was developed in 1987 by the famous computer professor Andrew S. Tanenbaum. Due to the emergence of the MINIX system and the availability of source code (which can only be used free of charge in universities), the whirlwind of learning the UNIX system was spurred by universities around the world. Linux first started development in 1991 with reference to the MINIX system.
- The GNU Project -- The development of the Linux operating system, and most of the software used on Linux is basically from the GNU program. Linux is only a kernel of the operating system. Without the GNU software environment (such as the bash shell), Linux will be difficult to move.
- POSIX Standard -- This standard has played an important role in the development of the Linux operating system after the formal development. It is the beacon of Linux's progress.
- Internet - If you don't have an Internet network and don't have the unselfish dedication of countless computer hackers all over the world, then Linux can only grow to a level of 0.13 (0.95).

1.2 Content review

This book will mainly describe and comment on the early Linux kernel version 0.12. The Linux-0.12 version was released on January 15, 1992. Include the following files when publishing:

bootimage-0.12.Z - a compressed boot image file with a U.S. keyboard code;
rootimage-0.12.Z - 1200kB compressed root file system image file;
linux-0.12.tar.Z - Kernel source code file. The size is 130KB, and only 463KB after expansion;

as86.tar.Z	- Bruce Evans' binary execution file. 16-bit assembler and loader;
INSTALL-0.11	- Updated installation information file.

bootimage-0.12.Z and rootimage-0.12.Z are compressed floppy image files. Bootimage is the boot image file, which mainly includes disk boot sector code, operating system loader, and kernel execution code. When the PC starts, the program in the ROM BIOS reads the boot sector code and data from the default boot drive into memory, and the boot sector code reads the operating system loader and kernel execution code into memory and then controls It is up to the operating system loader to further prepare the kernel for initialization, and the final loader will give control to the kernel code. Kernel code needs file system support to function properly. Rootimage is the root file system used to provide the most basic support to the kernel, including the operating system at least some configuration files and command execution procedures. For UNIX-based file system used in Linux system, it mainly includes some specified directories, configuration files, device drivers, development programs, and all other user data or text files. The combination of these two disks is equivalent to a bootable DOS operating system disk.

as86.tar.Z is a 16-bit assembler linker package. linux-0.12.tar.Z is the compressed Linux 0.12 kernel source code. INSTALL-0.11 is a simple installation documentation for the Linux 0.11 system. It also applies to Linux systems that use the 0.12 kernel.

At present, in addition to the original rootimage-0.12.Z file, the other four files can be found. However, the author has used the resources on the Internet to re-create a fully usable rootimage-0.12 root file system for Linux 0.12. The gcc 1.40 compiler that can be used in the 0.12 environment is recompiled and the available experimental development environment is configured. Currently, these files can be downloaded from the oldlinux.org website. The specific download directory location is:

- <http://oldlinux.org/Linux.old/images/> This directory contains the kernel image file bootimage and the root file system image file rootimage that have been created.
- <http://oldlinux.org/Linux.old/kernels/> This directory contains the kernel source code programs, including the Linux 0.12 kernel source code program described in this book.
- <http://oldlinux.org/Linux.old/bochs/> This directory contains Linux systems that have been set up to run under the computer simulation system bochs.
- <http://oldlinux.org/Linux.old/Linux-0.12/> This directory contains some of the other tools that can be used in the Linux 0.12 system and some of the original installation instructions.

This book mainly analyzes all the source code programs in linux-0.12 kernel in detail, and makes detailed comments on each source program file, including comments on Makefile files. The analysis process is mainly carried out according to the computer startup process. Therefore, the consistency of the analysis until the end of the initialization kernel starts calling the shell program. The rest of the programs are for their own analysis, there is no coherence, so you can read according to their own needs. However, some application examples are provided during the analysis.

In the process of analyzing all programs, if the author thinks it is difficult to understand the statement, it will give a detailed description of the relevant knowledge. For example, when an input/output operation to the interrupt controller is encountered, a detailed description of the Intel Interrupt Controller (8259A) chip will be given and the used commands and methods will be listed. This will help deepen the understanding of the code, but also better understand the use of the hardware used, the author believes that this method of interpretation than a separate chapter to the overall introduction of hardware or other knowledge is much more efficient.










Taking the Linux 0.12 kernel to "dissect" is to increase the efficiency of our understanding of Linux's

operating mechanism. Linux-0.12 version of the entire kernel source code is only about 463K bytes, including the content is basically the essence of Linux. The latest kernel version 2.6.XX is very large, 200 megabytes. Even if you spend a lifetime learning to read it may not be able to read all. Maybe you have to ask, "Since you want to start with Jane, why not analyze the smaller version of the Linux kernel source code for version 0.01? It's only about 240K bytes." The main reason is because the 0.01 version of the kernel code has too many shortcomings, not even Including drivers for floppy disks also does not relate well to the use of math coprocessors and instructions for login procedures. And the structure of the bootstrapping boot program is not the same as the current version, and the 0.12 boot boot program structure is basically the same as now. Another reason is that you can find the earlier version 1.22 of the already compiled and compiled kernel image file (bootimage-0.12), which can be used for boot demonstrations. If you add a simple root file system image (rootimage-0.12), it will be able to run normally.

There are also deficiencies in learning with Linux 0.12. For example, the kernel version does not include some very important code related to special process waiting queues, TCP/IP networks, etc. The allocation and use of memory is also different from the current kernel. Fortunately, the network code in Linux is basically self-contained, and the relationship with the kernel mechanism is not very large, so you can analyze the code after you understand the basic principles of Linux work.

This book describes all the code in the Linux kernel. In order to maintain the integrity of the structure, the description of the code is based on the structure of the source code in the kernel. Basically, the contents of each source code is a chapter. The order of the source files introduced can be found in the previous file list index. The directory structure of the entire Linux kernel source code is shown in Listing 1-1. All directory structures are based on Linux as the current directory.

List 1-1 Linux/ directory

	Name	Size	Last modified date (GMT)	Desc.
	boot/		1992-01-16 14:37:00	
	fs/		1992-01-16 14:37:00	
	include/		1992-01-16 14:37:00	
	init/		1992-01-16 14:37:00	
	kernel/		1992-01-16 14:37:00	
	lib/		1992-01-16 14:37:00	
	mm/		1992-01-16 14:37:00	
	tools/		1992-01-16 14:37:00	
	Makefile	3091 bytes	1992-01-13 03:48:56	

The content of this book can be divided into five parts. Chapters 1 to 4 are basics. The operating system is closely related to the hardware environment being run. If you want to thoroughly understand the entire operation of the operating system, then you need to understand its hardware operating environment, especially the processor multi-task operating mechanism. This part introduces in more detail the hardware composition of the microcomputer, the programming language used to compile the Linux kernel program, and the programming principle under Intel 80X86 protection mode; the second part includes chapters 5 through 7, describing the kernel boot boot and 32-bit operation. The preparation phase of the method should be fully read as a beginner to learn the kernel; the third part from Chapter 8 to Chapter 13 is the main part of the kernel code. The contents of Chapter 8 can be used as the main clue to read the subsequent chapters of this section. Chapters 14 to 16 are the

contents of the fourth section and can serve as reference for reading the third part of the source code. The last part includes only Chapter 17, which describes how to use the PC simulation software system Bochs to conduct various experimental activities on the Linux 0.12 kernel.

The second chapter is based on the hardware block diagram of the traditional microcomputer system. It mainly introduces the components of the IBM PC/AT386 microcomputer running on the Linux kernel. Describe the functions and relationships of each major section. At the same time, it is also compared with the block diagram of the latest microcomputer. This will provide enough relevant information for readers who have not learned the principles of computer composition.

Chapter 3 introduces the programming language, object file format, and compilation environment used in the Linux 0.12 kernel. The main goal is to provide the assembly language and GNU C language extension knowledge needed to read the Linux 0.12 kernel source code. This chapter first introduced the syntax and usage of as86 and GNU as assembler in more detail, and then explained the common C language extensions such as inline assembly, statement expressions, register variables, and inline functions in the GNU C language. The mutual calling mechanism between C and assembly functions is described in detail. Finally, the use of the Makefile is briefly described.

Chapter 4 describes the architecture of the 80X86 CPU and some basic knowledge of protected mode programming. It lays a solid foundation for preparing to read the Linux kernel source code based on the 80X86 CPU. These include: 80X86 basics, protected mode memory management, interrupt and exception handling, task management, and a simple multitasking kernel example.

Chapter 5 outlines the Linux operating system architecture, the organization of the kernel source code files, and the general functionality of each file. It also introduces the use of Linux for physical memory allocation, several stacks of the kernel, and how they are used, and the use of virtual linear addresses. Finally, it begins to comment the first file seen in the Linux/ directory in the kernel package, which is the contents of the overall Makefile of the kernel code. This file is the compilation management configuration file for all kernel source programs and is used by the build management tool software make.

Chapter 6 will explain in detail the three assembly language programs in the boot/ directory, including the bootdisk.ss of the disk boot program, the setup.s assembler that takes the parameters in the BIOS, and the 32-bit run start code program head.s. The three assembler programs complete the bootloading of the kernel from the block device into memory and detect system configuration parameters, completing all the work before entering the 32-bit protected mode. Prepare for the kernel system to perform further initialization work.

Chapter 7 mainly introduces the initialization program main.c of the kernel system in the init/ directory. It is a key point for the kernel to complete all initialization work and enter normal operation. After completing all the initialization of the system, a process for the shell is created. In the introduction of the program will need to see the other programs it calls, so the reading of the subsequent chapters can be performed in the order called here. Since memory management program functions are widely used in the kernel, this chapter should be read first. When you can really understand all the programs up to the main.c program, you should already have a certain understanding of the Linux kernel. It can be said that half of them are already started, but you also need to file systems, system calls, each Drivers, etc. for a deeper reading.

Chapter 8 mainly introduces all programs in the kernel/ directory. The most important part of the process is the process scheduler(), sleep_on(), and program related system calls. At this point you should already know some of the important programs. From the beginning of this chapter, we will encounter many assembly language statements embedded in C language programs. The basic syntax for embedded assembly statements is described in Chapter 3.

Chapter 9 explains the block device program in the kernel/blk_drv/ directory. This chapter mainly contains

drivers for block devices such as hard disks and floppy disks. It is mainly used to deal with file systems and high-speed buffers, and contains more hardware-related content. Therefore, you need to refer to some hardware information when reading this chapter. It's best to first look at the sections of the file system.

Chapter 10 Annotates the character device drivers in the `kernel/chr_drv/` directory. This chapter mainly deals with serial line drivers, keyboard drivers, and monitor drivers. These drivers constitute the serial terminal and console terminal devices supported by the 0.12 kernel. Therefore, this chapter also contains more hardware-related content. Need to refer to related hardware books when reading.

Chapter 11 introduces the math coprocessor simulation program in the `kernel/math/` directory. Due to the version of the kernel annotated in this book, coprocessors have not really started to be supported yet, so the content of this chapter is relatively small and relatively simple. Just have a general understanding.

The 12th chapter introduces the file system program in the `fs/` directory of the kernel source code. When reading this chapter, we recommend that you pause for a while to read about the MINIX file system in Andrew S. Tanenbaum's book "Operating System Design and Implementation". Chapters, because the original Linux system only supports MINIX file system, Linux 0.12 version is no exception.

Chapter 13 explains the memory management program in the `mm/` directory. To thoroughly understand this aspect, we need to have a sufficient understanding of the protection mode operation mode of the Intel 80X86 microprocessor. Therefore, when reading this chapter of the program, you can refer to the overview of the operation mode of the 80X86 protection mode included in the appropriate place in this chapter. In addition to the description, you should also refer to Chapter 4 at the same time. Since this chapter explains the use of examples in the source code as objects, you can better understand how memory management works.

Existing Linux kernel analysis books generally lack the description of the kernel header file, so for a beginner, there are many obstacles to reading the kernel program. Chapter 14 of this book details all the header files in the `include/` directory. Basically, each definition, each constant, or data structure is commented in detail. In order to facilitate reference during reading, this book also summarizes some important data structures and variables that are frequently used in the appendix, but these contents can actually be found in the header files of this chapter. Although the contents of this chapter are mainly used for reading the procedures in other chapters, if you want to thoroughly understand the kernel's operating mechanism, you still need to understand many of the details in these header files.

Chapter 15 describes all the files in the Linux 0.12 kernel source code `lib/` directory. These library function files mainly provide interface functions to the system programs such as the compilation system, which will help the future understanding of the system software. Because of this lower version, there is not much here, so we can read it quickly. This is one of the reasons why we chose the 0.12 version.

Chapter 16 introduces the `build.c` program in the `tools/` directory. This program is not included in the compiled and generated kernel image file. It is only used to connect the disk boot block in the kernel with other major kernel modules into a complete kernel image file.

Chapter 17 introduces the experimental environment for learning the kernel source code and the methods for hands-on experimentation. It mainly introduces the method of using and compiling Linux kernel under Bochs simulation system and the method of making disk image files. It also explains how to modify the syntax of the Linux 0.12 source code so that it can successfully compile the correct kernel under the RedHat 9 system.

The last is the appendix and index. The appendix gives some constant definitions and basic data structure definitions in the Linux kernel, as well as a concise description of the protection mode operating mechanism.

For ease of reference, the information on PC hardware used in the kernel is also listed separately in the appendix of this book. In the reference literature, we only provided books, articles, and other information that

we can refer to when reading the source code. We did not provide all kinds of complicated and messy literature lists. For example, when referring to a file in the LDP (Linux Document Project) of the Linux Documentation Project, we will explicitly list which HOWTO article we need to refer to, and not just the LDP's website address.

When Linus first developed the Linux operating system kernel, he mainly referred to three books. One is "UNIX Operating System Design" by M. J. Bach, which describes the working principle and data structure of the UNIX System V kernel. Linus uses the algorithms for many of the functions in the book. The names of many important functions in the Linux kernel source code are taken from the book. Therefore, when reading this book, this is an essential reference book on the working principle of the kernel. The other is "Programming the 80386" edited by John H. Crawford et al. and is a good book explaining the 80x86 protected mode programming method. There is also a first edition of the book "MINIX Operating System Design and Implementation" by Andrew S. Tanenbaum. Linus mainly uses the MINIX file system version 1.0 described in this book, and also supports only this file system in the early Linux kernel, so when reading this chapter about the file system, the working principle of the file system It is fully available from Tanenbaum's book.

In the explanation of each program, we first briefly explain the main purpose and purpose of the program, input and output parameters and the relationship with other programs, and then list the complete code of the program and make detailed comments on the code, the original The program code or text is not altered or deleted in any way, because C language is a kind of English language. The original small amount of English comments in the program also provides a lot of useful information for constant symbols, variable names, and so on. Behind the code is a more in-depth anatomy of the program and a description of some of the language or hardware related knowledge that appears in the code. If you look back through the program after reading this information, you will have a deeper understanding.

The introduction of some basic concept knowledge needed to read this book is scattered in the corresponding parts of the various chapters. This is mainly for the convenience of finding, and when you combine the source code reading, you can have a deeper understanding of some basic concepts.

The last thing to note is that when you have fully understood everything explained in this book, it does not mean that you have become a Linux expert. You just embarked on the journey of Linux, with some initial knowledge of becoming a Linux kernel master. At this point you should read more source code, preferably incrementally from version 1.0 up to the latest odd-numbered version under development. The latest Linux kernel at the time of revision of this book is version 4.16.16. When you can quickly understand the latest versions of these developments and even come up with their own suggestions and patches, I'm willing to take a plunge.

1.3 Summary

This chapter first elaborated on the indispensable pillars of the birth and development of Linux: UNIX's initial open source version provided the basic principles and algorithms for Linux implementation, and Richard Stallman's GNU program provided a variety of free and practical utilities for Linux systems. The emergence of tools and POSIX standards provides Linux with reference guides for implementing standards-compliant systems. AST's MINIX operating system has served as an indispensable reference for the birth of Linux, and the Internet is a necessary environment for Linux to grow and grow. Finally, the chapter outlines the basic content of the book.

2 Microcomputer structure

Any system can be seen as a model consisting of four basic parts, as shown in Figure 2-1. The input part is used to receive information or data entering the system; after being processed by the processing center, the output part is sent out. The energy section provides the energy supply for the operation of the entire system, including the input and output part of the energy required for operation.

The composition of the computer system is no exception, it is also mainly composed of these four parts. Internally, however, the channels or interfaces between the processing center and the input/output portion of the computer system can be used in common, and therefore (b) in Figure 2-1 should more appropriately abstractly represent a computer system. Of course, for computers or many complex systems, each of them can be regarded as a complete subsystem independently and can also be described using this model, and a complete computer system is composed of these subsystems.

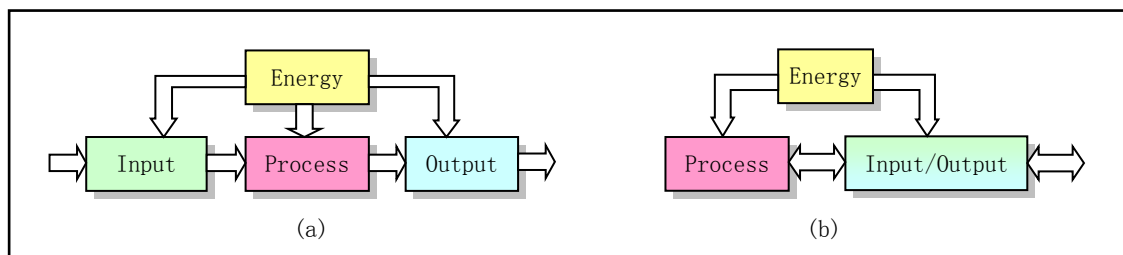


Figure 2-1 The basic composition of a system

Computer systems can be divided into hardware and software, but they are interdependent. The hardware part is the visible part of the computer system and is the platform for software operation and storage. Software is a stream of instructions that control hardware operations and actions. Just as information and thoughts stored in the human brain control the thoughts and actions of the human body, software can be seen as information and thinking in the "brain" of the computer. The theme described in this book is the operating mechanism of a computer system. It mainly explains the hardware composition principle of the processing center and the input/output part of the system and the realization of software control. On the hardware side, we outline the hardware system of an IBM PC microcomputer based on the Intel 80X86 CPU (Central Processing Unit) and its compatibles. The CPU chip of the computer can be regarded directly as the processing center of the system. The bus interface is connected with other parts; for the software running on it, we specifically describe the implementation of the Linux operating system kernel.

It can be seen that the operating system is closely related to the hardware environment being run. If you want to thoroughly understand the entire operating system, you need to understand its operating hardware environment. This chapter is based on the hardware block diagram of the traditional microcomputer system and introduces the functions of each major part of the microcomputer. These contents have basically established the hardware basis for reading the Linux 0.12 kernel. For ease of illustration, the term PC/AT will be used to refer to IBM PCs with 80386 or greater CPUs and their compatible microcomputers, while PCs are used generically to refer to all microcomputers, including IBM PC/XTs and their compatible microcomputers.

2.1 The Microcomputer Composition

From the perspective of overlooking, we illustrate the structure of a PC system with an 80386 or higher CPU. The structure of a conventional microcomputer hardware is shown in Figure 2-2. Among them, the CPU communicates with other parts of the system via a local bus (or internal bus) consisting of address lines, data lines, and control signal lines. The address line is used to provide the address of a memory or I/O device, which indicates the specific location where data needs to be read/written. Data lines are used to provide data transfer channels between the CPU and memory or I/O devices, while control lines are responsible for directing specific read/write operations. For PCs using the 80386 CPU, there are 32 internal address lines and data lines, respectively, which are all 32-bit. Therefore, the address space has 2^{32} bytes, ranging from 0 to 4GB.

In the figure, the upper controller and memory interface are usually integrated on a computer motherboard. These controllers are each a functional circuit composed mainly of a large-scale integrated circuit chip. For example, the interrupt controller is composed of Intel 8259A or its compatible chips; the DMA controller is usually composed of Intel 8237A chips; the timing counter is at the core of the Intel 8253/8254 timing chip; and the keyboard controller is using Intel 8042 chip with the keyboard. The scanning circuit communicates.

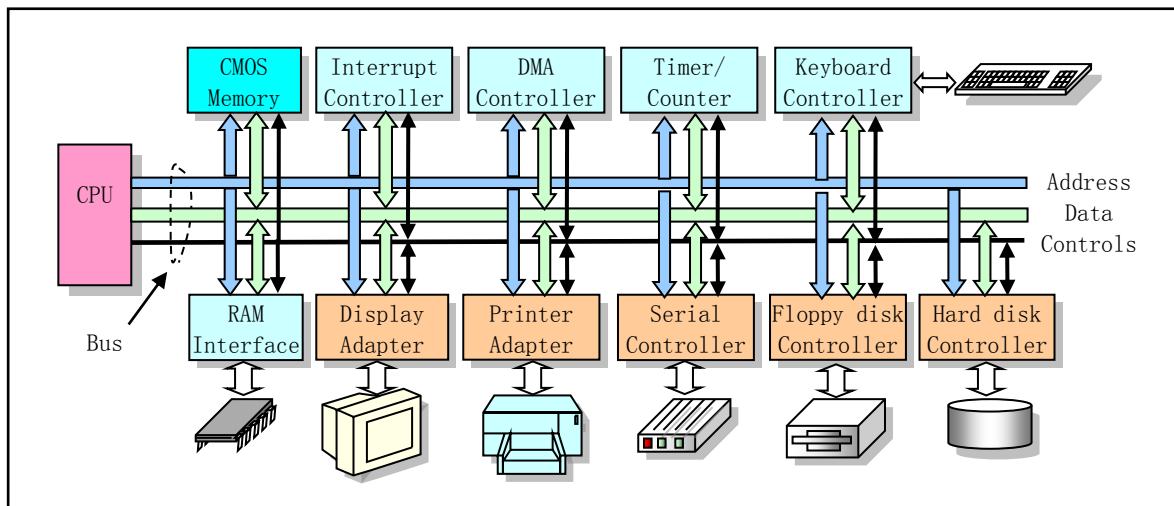


Figure 2-2 A block diagram of a traditional IBM PC and its compatibles

The control card (or adapter) in the lower part of the figure is connected to the system bus on the motherboard through an expansion slot. The bus slot is the standard connection interface to the extended device controller of the system address bus, data bus, and control line. These bus interface standards generally include an industry standard architecture ISA (Industry Standard Architecture) bus, an extended industrial standard architecture bus (EISA), a Peripheral Component Interconnect (PCI) bus, and an Accelerated Graphics Port (AGP) video. Bus and so on. The main difference between these bus interfaces is the data transfer rate and control flexibility. With the development of computer hardware, bus interfaces with higher transmission rates and more flexible control are still being introduced, such as the high-speed PCIE (PCI Express) bus using serial communication point-to-point technology. The original 80386 machine had only the ISA bus, so the system and external I/O devices can only use 16-bit data lines for data transfer.

With the development of computer technology, many functions (such as hard disk controller functions) that were originally implemented using control cards have been integrated in a few VLSI chips on a computer main

board. Several even one such chip is The main features and functions of the main board are determined, and the bus structure has undergone great changes in order to allow the different parts of the system to reach their highest transmission rates. The composition of modern PCs can often be described using Figure 2-3. In addition to the CPU, modern PC motherboards mainly use two chipsets or chipsets composed of ultra-large-scale chips: Northbridge chips and Southbridge chips. The Northbridge chip is used to interface with the CPU, memory, and AGP video. These interfaces have very high transmission rates. The North Bridge chip also plays a role in memory control. Therefore, Intel labels the chip as an MCH (Memory Controller Hub) chip. The South Bridge chip is used to manage low- and medium-speed components such as PCI bus, IDE hard disk interface, USB port, etc. Therefore, the name of the South Bridge chip is ICH (I/O Controller Hub). The reason for using the “South and North” bridges to collectively refer to these two chips is that they are located on the typical PC motherboards published by Intel Corporation. They are located at the lower and upper ends of the main version (that is, on the south and north of the map), and plays the role of channel bridging with the CPU.

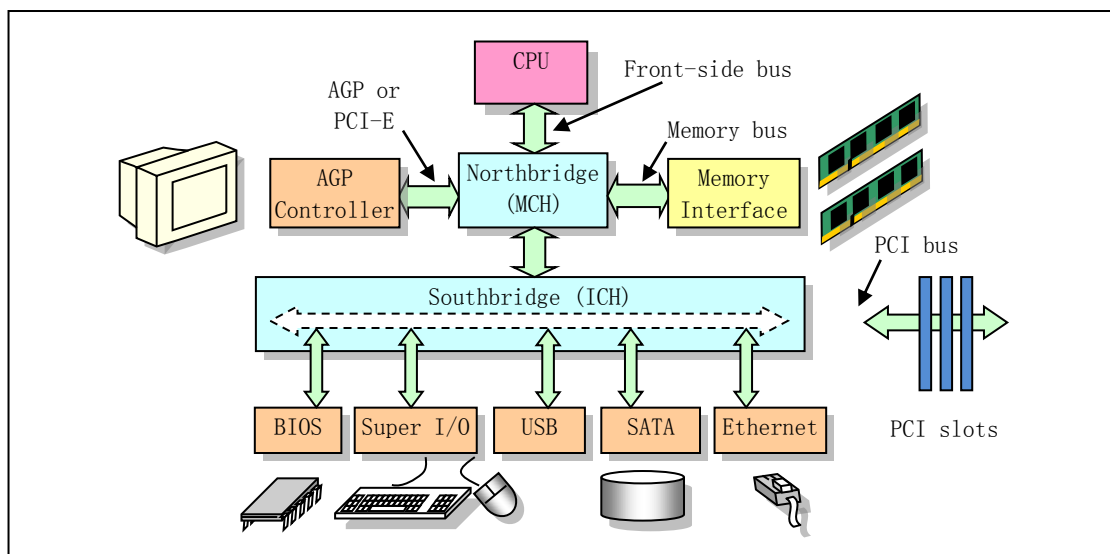


Figure 2-3 Modern PC chipset block diagram

Although the bus interface has undergone great changes, even the Northbridge and Southbridge will be combined in the future, but for our programmers, these changes are still compatible with the traditional PC architecture. Therefore, the program for the traditional PC hardware structure can still run on the current PC. This can be confirmed by Intel's development manual. Therefore, in order to facilitate the entry learning, we still discuss and study the composition and programming methods of the PC in the framework of the traditional PC architecture. Of course, these methods are still suitable for the modern PC architecture. Below we outline the working principle of each of the main controllers and control cards in Figure 2-2, and their actual programming methods are postponed until the corresponding kernel source code is read.

2.2 I/O Port addressing & access control

Before starting to transfer data between the CPU and the I/O adapter, it is necessary to first determine the I/O position of the communication adapter, that is, the port address. In the transmission of data between the CPU and the I/O interface, there may be a variety of transmission control modes. Generally, program loop query, interrupt processing, and DMA transfer may be used.

2.2.1 I/O ports and addressing

To access the data and status information on the I/O interface controller or control card, the CPU needs to specify their addresses first. This type of address is called an I/O port address or simply port. Usually an I/O controller includes a data port for accessing data, a command port for outputting commands, and a status port for accessing controller execution status. There are two ways to set the port address: unified addressing and independent addressing.

The principle of unified addressing of ports is to put the port address in the I/O controller into the memory addressing address space. Therefore, this addressing method also becomes memory image addressing. The operation of the CPU to access a port is the same as the operation of accessing the memory, and the instruction for accessing the memory is also used. The method of port independent address is to treat the addressing space of I/O controller and control card as a separate address space, which is called I/O address space. Each port has an I/O address corresponding to it and uses special I/O instructions to access the port.

The IBM PC and its compatible microcomputers mainly use an independent addressing mode and use a separate I/O address space to address and access the registers in the control device. Traditional PCs using the ISA bus architecture have I/O address space ranging from 0x000 to 0x3FF with 1024 I/O port addresses available. The default port address range used by each controller and control card is shown in Table 2-1. The use and programming methods of these ports will be described in detail later when the relevant hardware is specifically involved.

In addition, the IBM PC also partially uses the unified addressing mode. For example, the address of the display memory on the CGA display card directly occupies the memory address space 0xB800 -- 0xBC00 range. Therefore, if you want to display a character on the screen, you can directly use a memory operation instruction to perform a write operation to this memory area.

Table 2-1 I/O port address assignment

Address range	Allocation description
0x000 -- 0x01F	8237A DMA controller 1
0x020 -- 0x03F	8259A Programmable Interrupt Controller 1
0x040 -- 0x05F	8253/8254A Timer Counter
0x060 -- 0x06F	8042 Keyboard Controller
0x070 -- 0x07F	Access CMOS RAM/Real-Time Clock RTC Port
0x080 -- 0x09F	DMA page register access port
0x0A0 -- 0x0BF	8259A Programmable Interrupt Controller 2
0x0C0 -- 0x0DF	8237A DMA Controller 2
0x0F0 -- 0x0FF	Coprocessor access port
0x170 -- 0x177	IDE hard disk controller 1
0x1F0 -- 0x1F7	IDE hard disk controller 0
0x278 -- 0x27F	Parallel printer port 2
0x2F8 -- 0x2FF	Serial Controller 2
0x378 -- 0x37F	Parallel printer port 1
0x3B0 -- 0x3BF	Monochrome MDA display controller
0x3C0 -- 0x3CF	Color CGA display controller
0x3D0 -- 0x3DF	Color EGA/VGA display controller

0x3F0 -- 0x3F7	Floppy drive controller
0x3F8 -- 0x3FF	Serial Controller 1

For modern PCs using bus architectures such as EISA or PCI, 64 KB of I/O address space is available. The range of I/O addresses used by related controllers or settings can be obtained by looking at the `/proc/ioports` file under normal Linux systems. See the following:

```
[root@plinux root]# cat /proc/ioports
0000-001f : dma1
0020-003f : pic1
0040-005f : timer
0060-006f : keyboard
0070-007f : rtc
0080-008f : dma page reg
00a0-00bf : pic2
00c0-00df : dma2
00f0-00ff : fpu
0170-0177 : ide1
01f0-01f7 : ide0
02f8-02ff : serial(auto)
0376-0376 : ide1
03c0-03df : vga+
03f6-03f6 : ide0
03f8-03ff : serial(auto)
0500-051f : PCI device 8086:24d3 (Intel Corp.)
0cf8-0cff : PCI conf1
da00-daff : VIA Technologies, Inc. VT6102 [Rhine-II]
    da00-daff : via-rhine
e000-e01f : PCI device 8086:24d4 (Intel Corp.)
    e000-e01f : usb-uhci
e100-e11f : PCI device 8086:24d7 (Intel Corp.)
    e100-e11f : usb-uhci
e200-e21f : PCI device 8086:24de (Intel Corp.)
    e200-e21f : usb-uhci
e300-e31f : PCI device 8086:24d2 (Intel Corp.)
    e300-e31f : usb-uhci
f000-f00f : PCI device 8086:24db (Intel Corp.)
    f000-f007 : ide0
    f008-f00f : ide1
[root@plinux root]#
```

2.2.2 Interface access control

The PC I/O interface data transmission control mode can generally adopt program loop inquiry mode, interrupt processing mode and DMA transmission mode. As the name suggests, the cycle inquiry mode means that the CPU judges whether it can exchange data with the device by looping through the program to query the status in the specified device controller. This approach does not require excessive hardware support, and is relatively simple to use and program, but it consumes valuable CPU time. Therefore, this method should not be used in multitasking operating systems unless the waiting time is extremely short or necessary. In the Linux operating

system, this method is only used in a few places when the device or controller can immediately return information.

The interrupt handling control method needs the support of the interrupt controller. In this control mode, only when the I/O device requests a processing request from the CPU by interrupting, the CPU temporarily interrupts the currently executed program and executes the corresponding I/O interrupt processing service process. After executing the interrupt handling service process, the CPU will continue to execute the program that was just interrupted. When an I/O controller or device issues an interrupt request, the CPU addresses the entry address of the corresponding interrupt handling service process by using an interrupt vector table (or an interrupt descriptor table). Therefore, when using the interrupt control mode, it is necessary to first set the interrupt vector table and compile the corresponding interrupt processing service process. Most device I/O controls in the Linux operating system use interrupt handling.

The direct memory access (DMA) method is used for batch data transfer between the I/O device and the system memory. The entire operation process requires the use of a dedicated DMA controller without CPU intervention. Since there is no need for software intervention during the transmission, the operation is very efficient. In the Linux operating system, floppy disk drivers use interrupts and DMA methods to achieve data transfer.

2.3 Main memory, BIOS and CMOS memory

A typical PC often contains three types of memory, one is the main memory RAM (Random Access Memory) used to run programs and temporarily save data; the other is ROM (Read Only Memory) memory, stores the system boot diagnostics and initializes the hardware program; the third is a small amount of CMOS memory used to store the computer's real-time clock information and system hardware configuration information.

2.3.1 Main memory

When the IBM PC was first introduced in 1981, the system only had 640 KB of RAM main memory (referred to as memory). Since the 8088/8086 CPU used has only 20 address lines, the memory address range is up to 1024KB (1MB). At the time of the popular DOS operating system, the 640K or 1MB memory capacity was basically sufficient for ordinary applications. With the rapid development of computer software and hardware technology, current computers are usually configured with 512 MB or more of physical memory capacity, and all use Intel 32-bit CPUs, that is, PC/AT computers. Therefore, the CPU's physical memory addressing range has been up to 4GB (by using the new features of the CPU, the system can even address 64GB of physical memory capacity). However, in order to be compatible with the original PC in software, the allocation of physical memory below the system 1MB still remains basically the same as the original PC, but the BIOS of the original system ROM has always been the highest in the memory that the CPU can address. At the end location, the original location of the BIOS will be used as the shadow area of the BIOS when the computer is initially initialized, ie the BIOS code will still be copied to this area. See Figure 2-4.

When the computer is powered on, physical memory is set to a contiguous area starting at address 0. All memory except the range of addresses 0xA0000 to 0xFFFFF (384K to 1M total 384K) and 0xFFFFE0000 to 0xFFFFFFFF (the last 64K at 4G) can be used as system memory. These two specific ranges are used for I/O devices and BIOS programs. If we have 16MB of physical memory in our computer, 0-640K will be used to hold kernel code and data on Linux 0.1x systems. The Linux kernel does not use BIOS functions nor does it use the interrupt vector table set by the BIOS. The 384K between 640K and 1M is still reserved for the use indicated in the figure. Among them, the 128K starting from address 0xA0000 is used as the display memory buffer, and then the part is used for the ROM BIOS of other control cards or its mapping area, and 0xF0000 to 1M range is used

for the mapping area of the high-end system ROM BIOS. 1M-16M will be used by the kernel as an assignable main memory area. In addition, high-speed buffers and memory virtual disks also occupy a part of the memory area behind the kernel code and data. This area usually spans 640K to 1M.

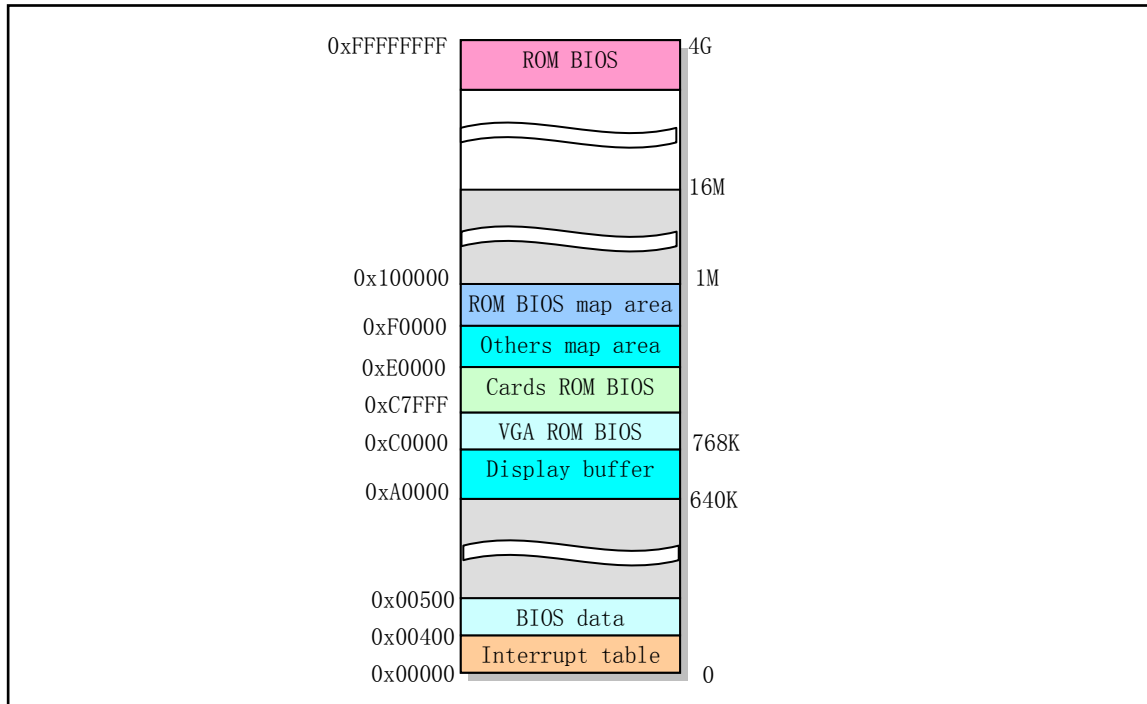


Figure 2-4 PC/AT machine memory area usage map

2.3.2 Basic input/output program BIOS

The system BIOS program stored in the ROM is mainly used to execute the self-check of various parts of the system when the computer is turned on, and various configuration tables that the operating system needs to use, such as an interrupt vector table and a hard disk parameter table, are established. It also initializes the processor and the rest of the system to a known state, and also provides hardware device interface services for operating systems such as DOS. However, since these services provided by the BIOS are not reentrancy (ie, the programs cannot be run concurrently), and considering the access efficiency, the Linux operating system runs at the same time, except that it uses the BIOS to provide some system parameters during initialization. Do not use the features in the BIOS.

When the computer system is powered on or a reset button on the chassis is pressed, the CPU automatically sets the code segment register CS to 0xF000, its segment base address is set to 0xFFFF0000, and the segment length is set to 64 KB. The IP is set to 0xFFFF0, so the CPU code pointer now points to 0xFFFFFFF0, which is the last 16 bytes of the last 64K in 4G space. From the above figure, this is where the system ROM BIOS is stored. And the BIOS will store here a jump instruction JMP to jump to an instruction in the 64KB range in the BIOS code to start execution. Since the BIOS capacity of PC/AT microcomputers is mostly 1MB to 2MB, and is stored in the Flash Memory ROM, it is far from the 0--1M address space in order to be able to execute or access the BIOS in more than 64 KB range. Other BIOS code or data, the BIOS program will first use 32-bit access to set the data segment register access range to 4G (rather than the original 64K), so that the CPU can execute and manipulate data in the range of 0 to 4G. After that, after the BIOS performs some column hardware detection and initialization operations, it will copy the 64 KB BIOS code and data compatible with the original PC to the 64K at

the low end of the 1M memory, and then jump to this place and let the CPU be real. Run in real address mode, as shown in Figure 2-5. Finally, the BIOS will load the operating system boot program from the hard disk or other block device into memory at 0x7c00 and jump to this location to continue the boot process.

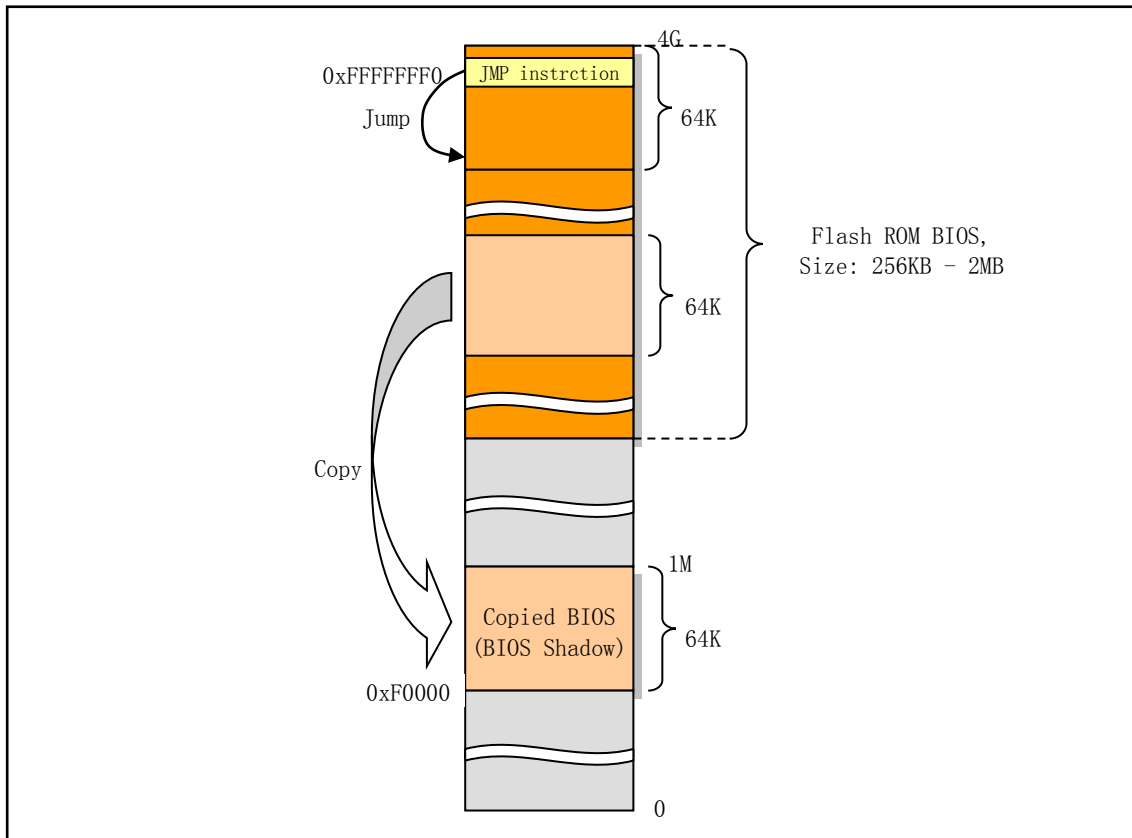


Figure 2-5 Flash ROM BIOS location and copy mapping area

2.3.3 CMOS memory

In PC/AT machines, in addition to the need to use memory and ROM BIOS, CMOS memory with little storage capacity (64 or 128 bytes) is used to store the real-time clock information and system hardware configuration information of the computer. This part of the memory is usually in an integrated block with the Real Time Chip. The address space of the CMOS memory is outside the basic memory address space and needs to be accessed using I/O instructions.

2.4 Controllers and Control Cards

As can be seen from Figure 2-2, a PC contains a variety of control cards and controllers used to transfer data and control computer operations. These controllers and control cards mainly include interrupt controllers, DMA controllers, keyboard controllers, floppy/hard disk control cards, serial communication control cards, and display control cards. Here the term "controller" refers to a control component that is integrated on a computer motherboard, and "control card" refers to a control card component that is inserted into the computer through an expansion slot. Because the control device can exist in the form of an independent control card, or it can be integrated into the main board with the increase in the degree of computer integration, there is no substantial difference between the controller and the control card. Below we describe these control devices in detail.

2.4.1 Interrupt controller

The IBM PC/AT 80X86-compatible microcomputer uses two cascaded 8259A programmable interrupt control chips to form an interrupt controller for I/O device interrupt control data access, and can provide independent interrupts for 15 devices. The control function is shown in Figure 2-6. During the initial boot-up of the computer, the ROM BIOS initializes two 8259A chips and assigns 15 levels of interrupt priority to the clock timer, keyboard, serial port, print port, floppy disk controller, coprocessor, and hard disk. Use equipment or controllers. At the same time, an interrupt vector table is created in the 0x000-0xFFFF area at the beginning of the memory, and these interrupt requests are mapped to the interrupt vector number starting from 0x08, as shown in Table 2-2.

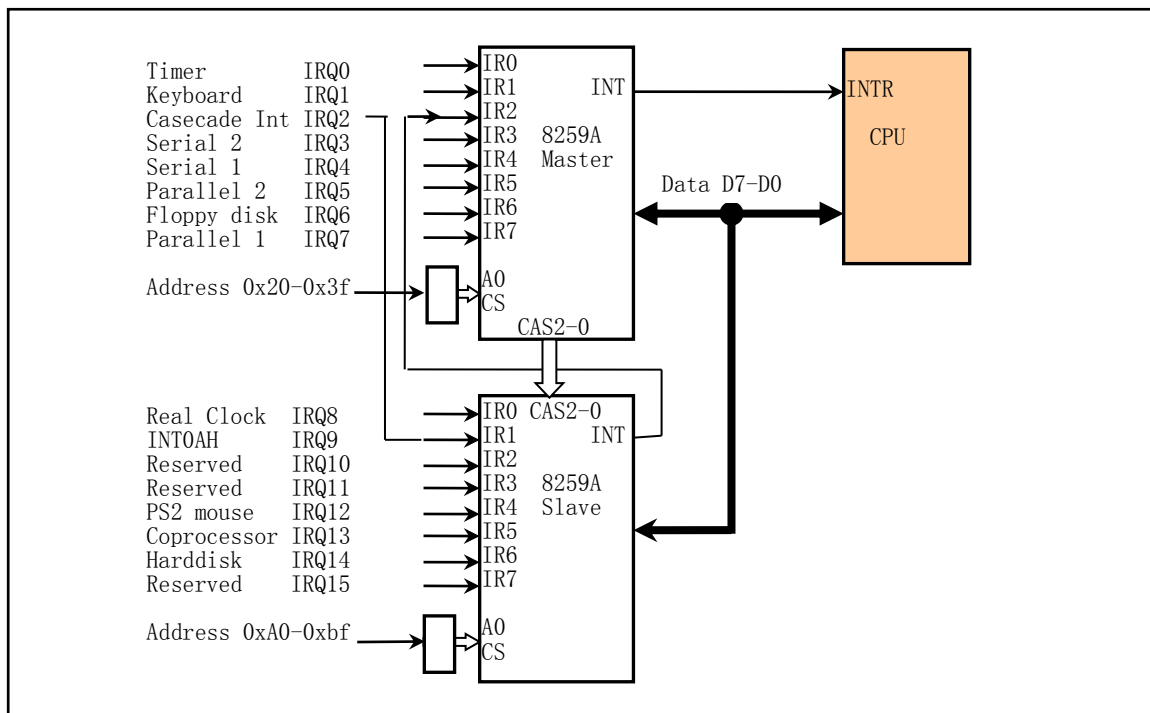


Figure 2-6 PC/AT microcomputer connected 8259 control system

However, since the interrupt number 0x00-0x1F belongs to Intel specifically reserved for the CPU, these BIOS settings conflict with Intel's requirements. To solve this problem, the Linux operating system does not directly use these interrupt numbers set by the BIOS. At power-on startup, the Linux operating system will re-set the 8259A during kernel initialization, mapping all system hardware interrupt request numbers to 0X20 and above interrupt numbers. See the subsequent sections for a detailed description of how the interrupt controller works and how to program it.

Table 2-2 Hardware request interrupt number set by ROM BIOS at power on

IRQ number	Interrupt number Set by the BIOS	Usage description
IRQ0	0x08 (8)	8250 issued 100HZ clock interrupt
IRQ1	0x09 (9)	Keyboard interrupt
IRQ2	0x0A (10)	The slave chip's interrupt

IRQ3	0x0B (11)	Serial port 2
IRQ4	0x0C (12)	Serial port 1
IRQ5	0x0D (13)	Parallel port 2
IRQ6	0x0E (14)	Floppy disk drive
IRQ7	0x0F (15)	Parallel port 1
IRQ8	0x70 (112)	Real-time clock interrupt
IRQ9	0x71 (113)	Change to INT 0x0A
IRQ10	0x72 (114)	Reserved
IRQ11	0x73 (115)	Reserved (network interface)
IRQ12	0x74 (116)	PS/2 mouse port interrupt
IRQ13	0x75 (117)	Math coprocessor interrupt
IRQ14	0x76 (118)	Hard disk controller interrupt
IRQ15	0x77 (119)	Reserved

2.4.2 DMA controller

As mentioned earlier, the main function of the DMA controller is to enhance the performance of the system by letting external devices transfer data directly to memory. Usually it is implemented by the Intel 8237 chip or its compatible chip on the machine. By programming the DMA controller, data transfer between peripherals and memory can be performed without CPU control. So during data transfer, the CPU can do other things.

In the PC/AT machine, two 8237 chips are used, so the DMA controller has 8 independent channels available. The last four of these are 16-bit channels. The floppy disk controller is specifically designated to use DMA channel 2. You must first set it before using a channel. This involves operations on three ports, the page register port, the (offset) address register port, and the data count register port. Since the DMA register is 8-bit, and the address and count value are 16-bit values, each needs to be sent twice.

2.4.3 Timer/counter

The Intel 8253/8254 is a Programmable Interval Timer (PIT) chip designed to handle precise time delays in computers. The chip provides three independent 16-bit counter channels. Each channel can work in different modes of operation, and these modes of operation can all be set using software. One way to perform a delay in software is to execute a loop operation statement, but doing so consumes CPU time. If the 8253/8254 chip is used in the machine, the programmer can configure the 8253 to meet its own requirements and use one of the counter channels to achieve the desired delay. After the time delay expires, 8253/8254 will send an interrupt signal to the CPU.

For the PC/AT and its compatible microcomputer system, the 8254 chip is used. The three timer/counter channels are used for time-of-day clocked interrupts, dynamic memory DRAM refresh timing circuits, and host speaker tone synthesis. The Linux 0.12 operating system only resets channel 0 so that the counter operates in mode 3 and sends a signal every 10 milliseconds to generate an interrupt request signal (IRQ0). The interrupt request generated at this interval is the pulse of the Linux 0.12 kernel. It is used to periodically switch the currently executed task and count the amount of system resources (time) used by each task.

2.4.4 Keyboard controller

The keyboard we are using now is a PC/AT compatible keyboard from IBM in 1984. It is usually called an AT-PS/2 compatible keyboard and has 101 to 104 buttons. There is a processor (Intel 8048 or compatible chip) on

the keyboard, which is called a keyboard encoder. It is used to scan and collect the status information (ie scan code) of all key presses and release, and sends it to the keyboard controller on the main board of the host computer. . When a key is pressed, the scan code sent by the keyboard is called Make code, or simply called the connect code; the scan code sent when a pressed key is released is called disconnected. Break code, or simply break code.

The host keyboard controller is specifically designed to decode the received keyboard scan code and send the decoded data to the operating system's keyboard data queue. Because the on and off codes of each key are different, the keyboard controller can determine which key the user is operating based on the scan code. The on and off codes of all the keys on the entire keyboard form a scan code set of the keyboard. According to the development of computers, there are currently three sets of scan codes available. They are:

- The first set of scan codes -- The original XT keyboard scan code set. The current keyboard has rarely sent such scan codes;
- The second set of scan codes -- The default scan code set used by modern keyboards, commonly referred to as the AT keyboard scan code set;
- The third set of scan codes -- PS/2 keyboard scan code set. The scan code set used by the original IBM launch of the PS/2 microcomputer has rarely been used.

The AT keyboard sends the second set of scan codes by default. In spite of this, the host keyboard controller will still convert all received second keyboard scan codes into the first scan code for compatibility with PC/XT software, as shown in Figure 2-7. Therefore, we usually only need to know the first set of scan codes when programming keyboard controllers. This is also the reason why only the XT keyboard scan code set is given when it comes to keyboard programming.

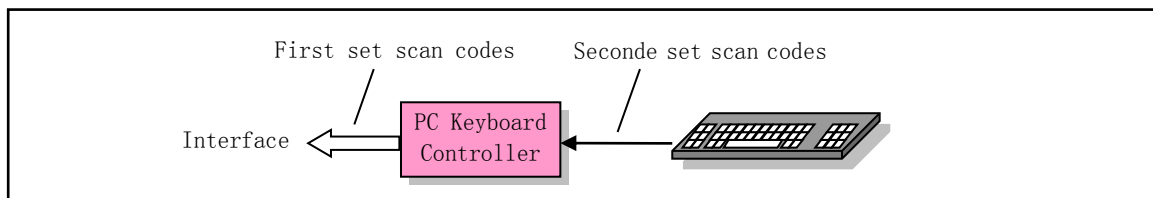


Figure 2-7 Keyboard controller conversion of scan code set

Keyboard controllers typically use Intel 8042 single-chip microprocessor chips or their compatible circuits. Today's PCs have integrated the keyboard controller in the motherboard chipset, but the functionality is still compatible with controllers that use the 8042 chip. The keyboard controller receives the 11-bit serial format data sent from the keyboard. The first bit is the start bit, the second 2-9 bits are the 8-bit keyboard scan code, the 10th bit is the parity check bit, and the 11th bit is the stop bit. See the description of the serial control card in the next section. After receiving the 11-bit serial data, the keyboard controller converts the keyboard scan code into a PC/XT standard keyboard-compatible system scan code, and then sends an interrupt request to the CPU through the IRQ1 pin of the interrupt controller. When the CPU responds to the interrupt request, the keyboard interrupt handler is invoked to read the XT keyboard scan code in the controller.

When a key is pressed, we can receive an XT keypad access code from the keyboard controller port. This scan code only indicates that the key at a location on the keyboard was pressed, but it has not yet been mapped to a character code. The connection code is usually one byte wide. For example, the key-on code for the key "A" is 30 (0x1E). When a pressed key is released, a break code is received from the keyboard controller port. For the XT keyboard (the scan code received by the keyboard controller programming port), the disconnection code is the

connection code when its connection code plus 0x80, that is, the most significant bit (bit 7) is set. For example, the break code for the above "A" key is $0x80 + 0x1E = 0x9E$.

But for those newly added ("extended") AT keyboard keys (such as the right Ctrl key and the right Alt key, etc.) for PC/XT standard 83-key keyboards, their on and off scan codes usually have 2 to 4 bytes, and the first byte must be 0xE0. For example, pressing the non-extended Ctrl key on the left produces a 1-byte passcode 0x1D, and pressing the Ctrl key on the right produces an extended 2-byte pass code 0xE0, 0x1D. Corresponding break codes are 0xE0, 0x9D. Table 2-3 shows several examples of turning on and off scan codes. In addition, the complete first set of scan codes is also given in the appendix.

Table 2-3 Example of the first scan code set received on the keyboard controller port

Pressed key	Connect scan code	Break scan code	Description
A	0x1E	0x9E	Non-expanding ordinary keys
9	0x0A	0x8A	Non-expanding ordinary keys
Function key F9	0x43	0xC3	Non-expanding ordinary keys
Arrow key right	0xE0, 0x4D	0xE0, 0xCD	Extended keys
Right Ctrl key	0xE0, 0x1D	0xE0, 0x9D	Extended keys
Left Shift + G	0x2A, 0x22	0xAA, 0xA2	Press and release Shift first

In addition, the output port P2 of the keyboard controller 8042 is used for other purposes. The P20 pin is used to implement the reset operation of the CPU, and the P21 pin is used to control the opening of the A20 signal line. When the output port bit 1 (P21) is 1, it turns on (gates) the A20 signal line, and 0 disables the A20 signal line. Today's motherboards no longer include a separate 8042 chip, but other integrated circuits on the motherboard will emulate the functionality of the 8042 chip for compatibility purposes. So now the keyboard programming is still using the 8042 programming method.

2.4.5 Serial control card

1. Asynchronous serial communication principle

Two computers/equipment exchange data, ie communication, must use the same language as people talk. In computer communication terminology, we refer to the "language" between a computer/device and a computer/device as a communication protocol. The communication protocol specifies the format for transmitting a unit of valid data length. Usually we use the term "frame" to describe this format. In order to allow the communication parties to determine the order of sending/sending and to perform some error detection operations, in addition to the necessary data, other information used for synchronization and error detection is also included in the transmitted one frame of information, for example, before the start of transmission of the data information. Send the start/synchronization or communication control information first, and then transmit some verification information after sending the required data information, as shown in Figure 2-8.

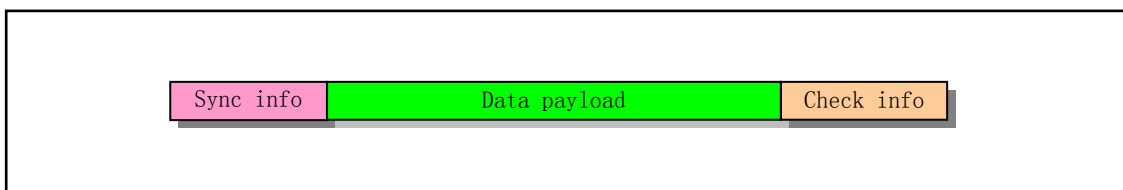


Figure 2-8 The general structure of communication frames

Serial communication refers to a communication method in which bit-bit data streams are transmitted one bit at a time on a line. Serial communication can be classified into asynchronous and synchronous serial communication. The main difference between them is the difference in the length of the communication units or frames that are synchronized during transmission. Asynchronous serial communication transmits a character as a communication unit or a frame, and synchronous serial communication transmits a sequence of a plurality of characters or bytes as one frame of data. If we use the dialogue between people as an analogy, then asynchronous communication is like the slow speed of conversation between the two parties. When speaking, a word is “worded out” and it can be paused for any length of time after each word is spoken. Synchronous communications, on the other hand, are like conversations between two parties in a consistent sentence. It can be seen that if we actually reduce the transmission unit to one bit (with letters!), then one-character asynchronous serial communication can also be regarded as a synchronous transmission of simultaneous transmission clock signals. way of communication.

2. Asynchronous serial transmission format

The frame format of asynchronous serial communication transmission is shown in Figure 2-9. Transmission of a character consists of a start bit, a data bit, a parity bit, and a stop bit. The start bit plays a role of synchronization and the value is always 0. The data bits are the actual data transmitted, ie a one-character code. Its length can be 5-8 bits. Parity bit is optional, set by the program. The stop bit is always 1, which can be set by the program to 1, 1.5, or 2 bits. Both parties must be set to the same format before communications begin sending information. If it has the same number of data bits and stop bits. In the asynchronous communication specification, the transmission 1 is called a MARK and the transmission 0 is called a SPACE. So we use these two terms in the following description.

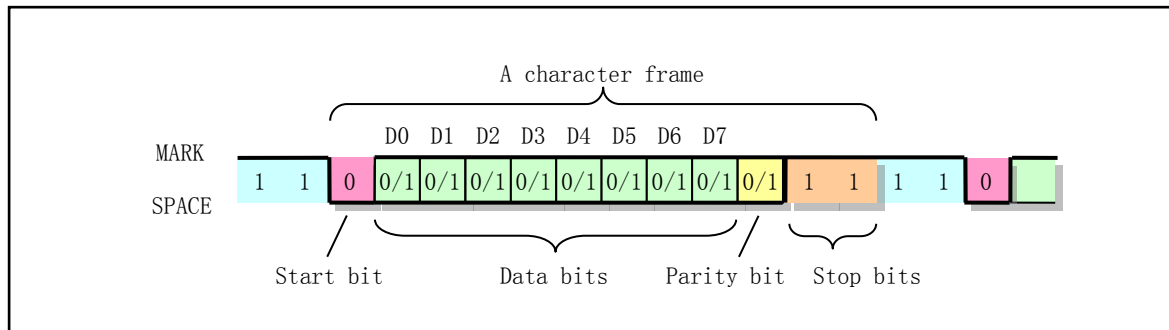


Figure 2-9 Asynchronous serial communication character transmission format

When there is no data transmission, the sender is in the MARK state, and sends 1 continuously. If it is necessary to send data, the sender needs to first send a space start bit of the bit interval. After receiving the space number, the receiver starts to synchronize with the sender and then receives the subsequent data. If the parity bit is set in the program, the parity bit needs to be received after the data transmission. Last is the stop bit. After a character frame is sent, the next character frame may be sent immediately, or the password may be temporarily sent, and the character frame may be sent after a moment.

When receiving a character frame, the receiver may detect one of three errors: (1) Parity error. At this point, the program should ask the other party to resend the character; (2) An overspeed error. This error occurs because the program fetches characters at a slower rate than the receiving speed. At this point you should modify the program to speed up the fetching of the character frequency; (3) The frame format is incorrect. This error occurs when the format information requested to be received is incorrect. For example, the empty number was received

when the stop bit should be received. Usually this kind of error is caused by the difference in frame format between the two parties except the line interference.

3. Serial controller

To achieve serial communication, PCs are usually equipped with two RS-232C-compliant serial interfaces and are processed using a serial controller consisting of a Universal Asynchronous Receiver/Transmitter (UART). Send and receive serial data. The serial interface on the PC usually uses a 25-pin or 9-pin DB-25 or DB-9 connector, which is mainly used to connect MODEM devices to work. Therefore, the RS-232C standard specifies many MODEM dedicated interface pins.

Previous PCs all use National Semiconductor's NS8250 or NS16450 UART chips. Today's PCs use the 16650A and its compatible chips, but are compatible with the NS8250/16450 chips. The main difference between the NS8250/16450 and the 16650A is that the 16650A chip also supports FIFO transfer. In this mode, the UART can cause an interrupt only after it has received or transmitted a maximum of 16 characters, which can relieve the system and CPU. When the PC is powered on, the RESET signal passes through the MR pin of the NS8250 to reset the UART internal registers and control logic. If you want to use the UART afterwards, you need to perform initial programming operations to set the UART's operating baud rate, data bits, and operating mode.

2.4.6 Display control

IBM PC/AT and its compatible computers can use color and monochrome video cards. IBM's earliest PC video system standards include monochrome MDA and color CGA standards, as well as EGA and VGA standards. All of the later advanced graphics cards (including today's AGP graphics cards) have extremely high graphics processing speeds and smart acceleration processing capabilities, but they all support these initial standards. The Linux 0.1x operating system uses only the text display methods supported by these standards.

1. MDA display standard

The monochrome display adapter MDA (Monochrome Display Adapter) only supports black and white display. And only supports the unique text character display mode (BIOS display mode 7). Its screen display specifications are 80 columns x 25 lines (column number $x = 0..79$; line number $y = 0..24$), and a total of 2000 characters can be displayed. Each character also has 1 attribute byte, so it takes 4 KB to display one screen (one frame). The even address byte stores the character code, and the odd address byte stores the display attribute. The MDA card is configured with 8KB of display memory. 8 KB space (0xb0000 — 0xb2000) starting from 0xb0000 is occupied in the memory address range of the PC. If the display screen number is `video_num_lines = 25`; the number of columns is `video_num_columns = 80`, then the position of the characters and attributes located at the screen column row values x, y in memory is:

```
Character byte position = 0xb0000 + video_num_columns * 2 * y + x * 2;  
Attribute byte position = Character byte position + 1;
```

In the MDA monochrome text display mode, the attribute byte format of each character is shown in Table 2-4. Among them, D7 is set to 1 will cause the character to flash; D3 is set to 1 to highlight the character. It is basically the same as the attribute byte of the color text character in Figure 2-10, but with only two colors: white (0x111) and black (0x000). Their combined effect is shown in the table.

Table 2-4 Monochrome display character attribute byte settings

Background color D6D5D4	Foreground color D2D1D0	Attribute value No flash low	display effect	example
0 0 0	0 0 0	0x00	Characters are not visible.	
0 0 0	1 1 1	0x07	White characters displayed on a black background (normal display).	Normal
0 0 0	0 0 1	0x01	White underlined characters displayed on a black background.	<u>Underline</u>
1 1 1	0 0 0	0x70	Black characters displayed on a white background (inverse).	Reverse
1 1 1	1 1 1	0x77	Show white squares.	■

2. CGA display standard

The color graphics adapter CGA (Color Graphics Adapter) supports seven kinds of color and graphics display (BIOS display 0—6). In the 80 column X 25 column text character display mode, there are two monochrome and 16 color display modes (BIOS display mode 2—3). The CGA card comes standard with 16KB of display memory (occupying the memory address range 0xb8000 — 0xbc000), so a total of 4 frames of display information can be stored therein. Similarly, in the 4KB display memory per frame, the even address byte stores the character code, and the odd address byte stores the character display attribute. However, only 8KB of display memory (0xb8000 — 0xba000) is used in the console.c program. In the CGA color text display mode, the definition of the attribute byte format for each display character is shown in Figure 2-10.

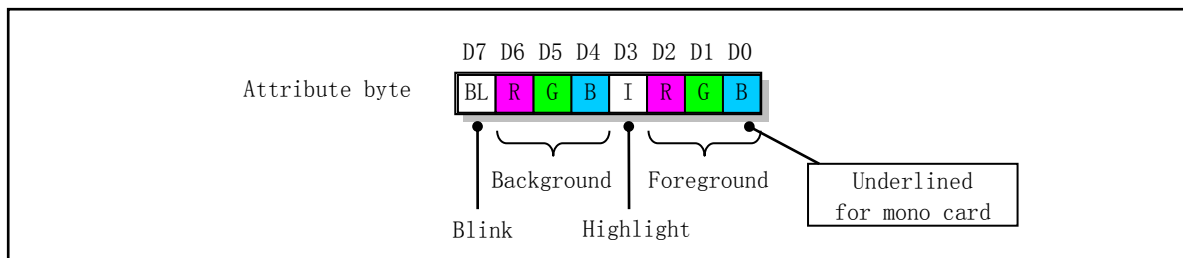


Figure 2-10 Character attribute format definition

As with the monochrome display, D7 is set to 1 to flash the display character; D3 is set to 1 to highlight the character; bits D6, D5, D4 and D2, D1, and D0 can be combined to create 8 colors. The combination of foreground and high-brightness bits can display the other 8 character colors. The color of these combinations is shown in Table 2-5.

Table 2-5 Foreground color and background color (left half)

I R G B	Value	Color name	I R G B	Value	Color name
0 0 0 0	0x00	Black	1 0 0 0	0x08	Dark grey
0 0 0 1	0x01	Blue	1 0 0 1	0x09	Light blue
0 0 1 0	0x02	Green	1 0 1 0	0x0a	Light green

0 0 1 1	0x03	Cyan	1 0 1 1	0x0b	Light cyan
0 1 0 0	0x04	Red	1 1 0 0	0x0c	Light red
0 1 0 1	0x05	Magenta	1 1 0 1	0x0d	Light magenta
0 1 1 0	0x06	Brown	1 1 1 0	0x0e	Yellow
0 1 1 1	0x07	Light grey	1 1 1 1	0x0f	White

3. EGA/VGA display standard

Enhanced Graphics Adapters (EGAs) and Video Graphics Adapters (VGAs) also support other display enhancements in graphics, in addition to or in addition to MDA and CGA support. In the MDA and CGA compatible display mode, the starting position and range of the occupied memory address are the same. However, EGA/VGA comes standard with at least 32KB of display memory. The physical memory address space starting from 0xa0000 is occupied graphically.

2.4.7 Floppy disk and hard disk controller

The floppy disk control subsystem of the PC consists of a floppy disk and a floppy disk drive. Because floppy disks can store programs and data and are easy to carry, floppy disk drives have long been one of the standard configuration devices on PCs. The hard disk also consists of a disk and a drive, but usually the hard disk's metal disk is fixed in the drive and cannot be removed. Because the hard disk has a large storage capacity, and read and write speed is very fast, it is the largest external storage device in the PC, usually also called external storage. Both floppy disks and hard disks use magnetic media to store information and have a similar storage operation. So here we use the hard disk as an example to briefly explain how they work.

The basic way to store data on a disc is to use a layer of magnetic media on the surface of the disc after magnetization. Floppy disks usually use polyester film as the substrate, while hard disk disks usually use metal aluminum alloy as the substrate. A floppy disk contains a polyester film disc. The upper and lower heads are used to read and write data on both sides of the disc. The disc rotation speed is about 300 rpm. For a floppy disk with a capacity of 1.44MB, the two sides of the disk are divided into 80 tracks, each track can store 18 sectors of data, so there are $2 \times 80 \times 18 = 2880$ sectors. Table 2-6 shows the basic parameters of several common types of floppy disks.

Table 2-6 Common floppy disk basic parameters

Disk type and capacity	tracks/face	Sectors/tracks	Total sectors	Rotate speed (r/min)	Data transmission rate (Kbps)
5¼ inch 360KB	40	9	720	300	250
3½ inch 720KB	80	9	1440	360	250
5¼ inch 1.2MB	80	15	2400	360	500
3½ inch 1.44MB	80	18	2880	360	500
3½ inch 2.88MB	80	36	5760	360	1000

The hard disk usually includes at least two or more metal disks, and thus has more than two read/write heads. For example, there are four physical heads for a hard disk that contains two disks, and eight read and write heads for a disk that contains four disks. See Figure 2-11. The hard disk rotation speed is usually fast at 4500 rpm to 10,000 rpm, so the hard disk data transfer speed is usually up to several megabits/second.

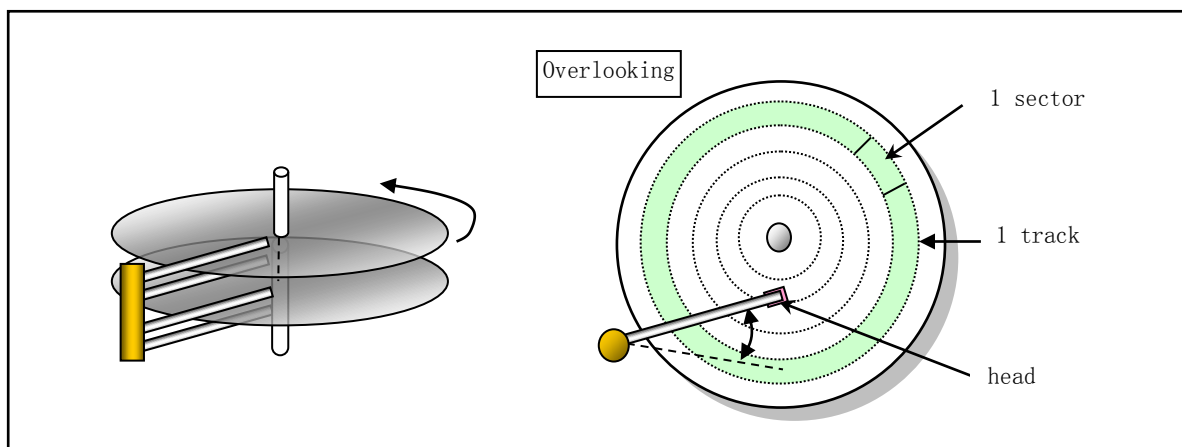


Figure 2-11 Typical hard disk internal structure with 2 disks

The magnetic head on the disk surface has a read coil and a write coil, respectively. During the read data operation, the head is first moved to a position on the rotating disk. Since the magnetic disk rotates, the magnetic medium moves at a uniform speed with respect to the magnetic head, so the magnetic head actually cuts magnetic lines of force on the magnetic medium. As a result, current is generated in the reading coil due to induction. Depending on the direction of the remanent state of the disk surface, the direction of the current induced in the coil is also different, so that 0 and 1 data recorded on the disk are read out, so that the bit stream can be sequentially read out from the disk. Since each track read by the head has a specific format for storing information, the disk circuit can discriminate and read the data in each sector on the track by recognizing the format in the read bit stream. See Figure 2-12 shows. Among them, GAP is an interval field used for isolation. Usually GAP is 12 bytes of 0. The address field of each sector address field stores the cylinder number, head number (face number), and sector number of the relevant sector, so a sector can be uniquely determined by reading the address information in the address field.

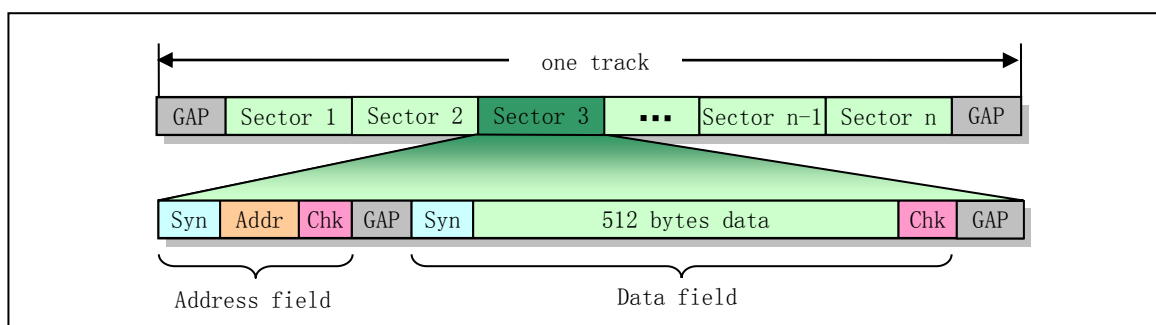


Figure 2-12 Disc track format

To read and write data on disks, you must use a disk controller. The disk controller is a logical interface circuit between the CPU and the driver. It receives request commands from the CPU, sends seeks, read/write, and control signals to the driver and controls and converts the data flow patterns. The data transferred between controller and driver includes the sector address information and timing and clock information in Figure 2-12. The controller must separate these address information and some encoding, decoding and other control information from the actual read/write data. In addition, the data transfer with the driver is a serial bit stream, so the controller needs to convert between parallel byte data and serial bit stream data.

The FDC (Floppy Disk Controller) in the PC/AT machine uses the NEC μ PD765 or its compatible chip. It is

mainly used to receive commands issued by the CPU and output various hardware control signals to the driver according to the command requirements, as shown in Figure 2-13. When performing a read/write operation, it needs to perform data conversion (string-parallel), encoding and verify operations, and constantly monitor the operating state of the drive.

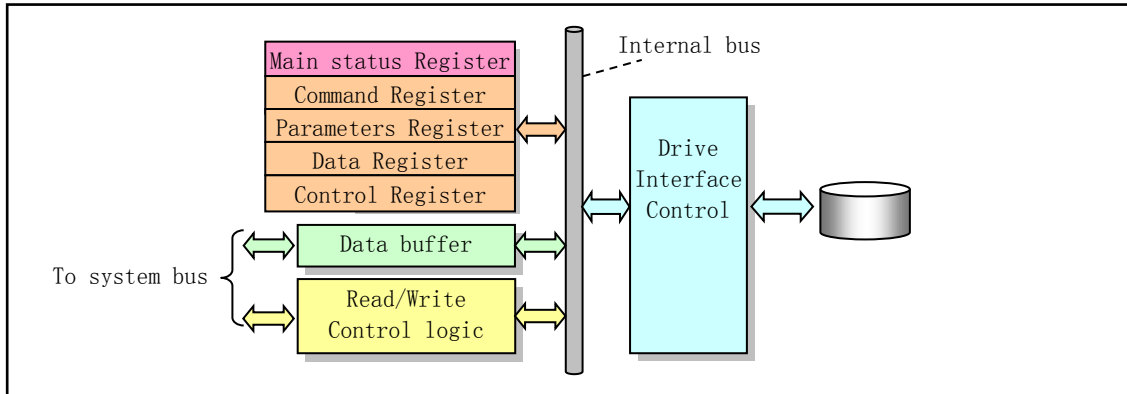


Figure 2-13 Disk controller internals

The programming process for the disk controller is to set the contents of the relevant registers in the controller through the I/O port and obtain the result information of the operation through the register. As for the transmission of sector data, the floppy disk controller is different from the PC/AT hard disk controller. The floppy disk controller circuit uses DMA signals and therefore requires the use of a DMA controller for data transfer. The AT hard disk controller uses high-speed data blocks for transmission without the intervention of a DMA controller. Because floppy disks are relatively vulnerable to damage (mould or scratches), floppy disk drives have not been deployed in computers at present. Instead, they use larger-capacity and more portable USB flash drives.

2.5 Summary

Hardware is the basic platform for operating systems. Understanding the hardware environment in which the operating system runs is a necessary condition for an in-depth understanding of the operating system on which it is running. This chapter briefly introduces each major part of the microcomputer based on the hardware composition of traditional microcomputers. In the next chapter, we describe the two assembly language syntaxes and related compilers used by the Linux kernel from a software perspective, and also introduce the contents of the GNU gcc syntax extensions used by the kernel.

3 Kernel programming language and environment

The language compilation process is the process of converting high-level languages that humans can understand into binary machine instructions that computer hardware can understand and execute. This conversion process usually generates some code that is not very efficient, so some of the code that requires high operating efficiency or has a large impact on performance is usually written directly in low-level assembly language, or the assembler generated by high-level language compilation. Then manually modify the optimization process. This chapter describes the programming language, object file format, and compilation environment used in the Linux 0.12 kernel. The main goal is to provide the assembly language and GNU C language extension knowledge needed to read the Linux 0.12 kernel source code. First of all, it introduces the syntax and usage of as86 and GNU as assembler in more detail. Then it uses C language extensions in kernel source code such as inline assembly, statement expressions, register variables, and inline functions in GNU C language. Introduced, and detailed description of the mutual calling mechanism between C and assembly functions. Because understanding the object file format is one of the most important prerequisites for understanding how the assembler works, when introducing two assembly languages, the basic format of the target file will be briefly described first, and Linux 0.12 will be given in more detail later in this chapter. The a.out object file format used in the system. Finally, the use of the Makefile is briefly described.

The content of this chapter is reference information when reading the Linux kernel source code. So you can take a brief look at the contents of this chapter, and then read the next section, and then refer back to this chapter when you encounter problems.

3.1 as86 Assembler

Two assemblers are used in Linux 0.1x systems. One is an as86 assembler that produces 16-bit code, using the companion ld86 linker; the other is the GNU assembler gas(as), which uses the GNU ld linker to link the resulting object files. Here we first describe how to use the as86 assembler, and the use of the assembler is described in the next section.

as86 and ld86 are Intel 8086, 80386 assembler compilers and linkers written by Bruce Evans, one of the main developers of MINIX-386. Linus already ported it to a Linux system when he first started developing the Linux kernel. Although it can compile 32-bit code for the 80386 processor, the Linux system uses it only to create the 16-bit boot sector program boot/bootsect.s and the binary setup code for the initial setup program boot/setup.s in real mode. The compiler is fast and compact, and has some features that GNU gas does not have, such as macros and more error detection methods. However, the compiler's syntax is incompatible with the syntax of the GNU as assembly compiler and more closely resembles the syntax of assemblers such as Microsoft's MASM, Borland's Turbo ASM, and NASM. These assemblers all use Intel's assembly language syntax (eg, the order of the operands is the opposite of GNU as, etc.).

The syntax of as86 is based on the assembly language syntax of the MINIX system, while the assembly syntax of the MINIX system is based on the assembler syntax of the PC/IX system. PC/IX was a UN*X operating system that was running on an Intel 8086 CPU long ago. Andrew S. Tanenbaum developed the MINIX system on a PC/IX system.

Bruce Evans is one of the major revision programmers for the 32-bit version of the MINIX operating system. He is a close friend of Linux founder Linus Torvalds. In the early days of Linux kernel development, Linus learned a lot from Bruce Evans about UNIX-like operating systems. The inadequacy of the MINIX operating system is also the result of two good friends discussing each other. These disadvantages of MINIX are just one of the main drivers that inspired Linus to develop a new concept operating system on the Intel 80386 architecture. Linus once said: "Bruce is my hero", so we can say that the birth of the Linux operating system and Bruce Evans also have a close relationship.

The source code for this compiler and linker can be downloaded from the FTP server `ftp.funet.fi` or from the website `www.oldlinux.org`. On modern Linux systems, RPM packages containing `as86/ld86` can be installed directly, for example `dev86-0.16.3-8.i386.rpm`. Since the Linux system only uses `as86` and `ld86` to compile and link the two 16-bit assemblers `bootsect.s` and `setup.s` mentioned above, only the assembler syntax and assembly commands (assembler) used in these two programs are described here. The role and use of indicator).

3.1.1 as86 assembly language syntax

The assembler is designed to compile low-level assembly language programs into binary programs or object files containing machine code. The assembler will compile an input assembly language program (such as `srcfile`) into an object file (`objfile`). The basic format of the command line for assembly is:

```
as [options] -o objfile srcfile
```

The options are used to control the compilation process to produce the target file with the specified format and settings. The input assembly language program `srcfile` is a text file. The contents of the file must consist of a series of lines of text that end with a newline character. Although GNU `as` can use semicolons to include multiple statements on a single line, it is common to include only one statement per line when programming assembly language programs.

Statements can be empty lines that contain only spaces, tabs, and line breaks, as well as assignment statements (or definition statements), pseudo-operator statements, and machine instruction statements. Assignment statements are used to assign a value to a symbol or identifier. It consists of an identifier followed by an equal sign followed by an expression, for example: `"BOOTSEG = 0x07C0"`. Pseudo-operator statements are indicators used by the assembler and usually do not generate any code. It consists of pseudo-opcodes and zero or more operands. Each opcode begins with a dot character `'.'`. The dot character `'.'` itself is a special symbol that represents the position counter during compilation. The value is the address of the first byte of the machine instruction where the dot symbol appears.

The machine instruction statement is a mnemonic of an executable machine instruction and consists of an operation code and 0 or more operands. In addition, any statement can be preceded by a label. The label consists of an identifier followed by a colon `'.'`. During compilation, when the assembler encounters a label, the current position counter value is assigned to this label. Therefore, an assembly statement usually consists of three fields: label (optional), instruction mnemonic (instruction name), and operand. The label is located in the first field of an instruction. It represents the address of its location and usually indicates the destination of a jump instruction. Finally, you can also follow the comment section that starts with the comment.

The object file `objfile` generated by the assembler compilation usually contains at least three segments or sections, namely, a text segment (`.text`), a data segment (`.data`), and an uninitialized data segment (`.bss`). A text segment (or a code segment) is an initialized segment that usually contains program execution code and read-only data. The data segment is also an initialized segment that contains read/write data. The uninitialized data segment

is an uninitialized segment. Usually the output object file generated by the assembler will not reserve space for the segment, but the operating system will initialize the contents of the segment to 0 when the object file is linked into the execution program. During compilation, statements that generate code or data in assembly language programs generate code or data in one of these three segments. Compiled bytes are stored starting from the '.text' section. We can use segment control pseudo operators to change the written segment. The target file format will be described in detail in the section "Linux 0.12 Object File Format" below.

3.1.2 as86 assembly programs

Below we use a simple framework sample program boot.s to illustrate the structure of the as86 assembler and the syntax of the statements in the program, and then give a compilation link and run method. Finally, use the as86 and ld86 usage methods and compilation options. The sample program is shown below. This example is a framework program for bootsect.s that compiles and generates the boot sector code. In order to demonstrate the use of certain statements, deliberately added a meaningless line 20 statements.

```
1 !
2 ! boot.s -- bootsect.s framework program. Replace 1 character in the string msg1
3 !           with code 0x07 and display it on the first line of the screen.
4 .globl begtext, begdata, begbss, endtext, enddata, endbss ! Global id used for ld86 links
5 .text ! Text segment
6 begtext:
7 .data ! Data segment
8 begdata:
9 .bss ! Uninitialized data segment
10 begbss:
11 .text ! Text segment
12 BOOTSEG = 0x07c0 ! Original segment address for the loaded bootsect code.
13
14 entry start ! Inform the linker the program starts executing from here.
15 start:
16     jmp     go, BOOTSEG ! Jump between segments. INITSEG indicates the jump
                        ! section address, the label go is the offset address.
17 go:     mov     ax, cs ! The value of the segment register cs --> ax is used
18         mov     ds, ax ! to initialize the data segment registers ds and es.
19         mov     es, ax
20         mov     [msg1+17], ah ! 0x07-> Replaces 1 dot in the string and beep once.
21         mov     cx, #20 ! 20 chars displayed, including cr & lf.
22         mov     dx, #0x1004 ! String displayed on screen at line 17, column 5.
23         mov     bx, #0x000c ! Character display attribute (red).
24         mov     bp, #msg1 ! Point to a string (required by interrupt call).
25         mov     ax, #0x1301 ! Write string and move cursor to the end of the string.
26         int     0x10 ! The BIOS interrupt call 0x10, function 0x13, subfunc 01.
27 loop1:  jmp     loop1 ! Dead cycle.
28 msg1:   .ascii "Loading system ..." ! Message to be displayed, total of 20 ASCII characters.
29         .byte 13, 10
30 .org 510 ! Indicates statement is stored from address 510 (0x1FE).
31         .word 0xAA55 ! Active boot sector flag, used by the BIOS.
32 .text
33 endtext:
34 .data
35 enddata:
```

```
36 .bss  
37 endbss:
```

We first introduce the function of the program, and then explain in detail the role of each statement. This program is a simple boot sector program. Compile and link the generated executable program can be placed in the first sector of the floppy disk directly used to boot the computer. After starting, the red string "Loading system .." is displayed at line 17 and column 5 of the screen, and the cursor moves down one line. Then the program loops endlessly on code line 27.

The first three lines of the program are comment statements. In an as86 assembly language program, statements beginning with an exclamation mark '!' or a semicolon ';' are followed by comment text. The comment statement can be placed after any statement, or it can start with a new line.

".globl" on line 4 is an assembly directive (or assembly directive, pseudo-operator). Assembler indicators start with a single character '.' and do not generate any code at compile time. Assembler directives consist of a pseudo opcode followed by zero or more operands. For example, 'globl' on the 4th line is a pseudo-opcode, and the labels following it 'begtext, begdata, begbss' and so on are its operands. The label is an identifier followed by a colon, for example 'begtext:' on line 6. However, there is no need to take a colon when referring to a label.

Usually an assembler supports many different pseudo-operators, but the following only describes the commonly used as86 pseudo-operators used in the Linux system bootsect.s and setup.s assembly language programs.

The '.globl' pseudo-operator is used to define that subsequent label identifiers are external or global and are mandatory to introduce even if they are not used.

In addition to the three labels defined on lines 5 to 11, three pseudo operators are defined: '.text', '.data', and '.bbs'. They respectively correspond to the assembler program to generate 3 segments in the target file, namely the text segment, the data segment, and the uninitialized data segment. '.text' is used to identify the start position of the text segment and switch to the text segment; '.data' is used to identify the starting position of the data segment, and the current segment is switched to the data segment; and '.bbs' is used Identifies the beginning of an uninitialized data segment and changes the current segment to the bbs segment. So lines 5--11 are used to define a label in each segment, and then switch to the text segment to start writing the following code. Here, all three segments are defined in the same overlapping address range, so the sample program is not actually segmented.

Line 12 defines an assignment statement "BOOTSEG = 0x07c0". The equal sign '=' (or the symbol 'EQU') is used to define the value represented by the identifier BOOTSEG, so this identifier can be referred to as a symbolic constant. This value, like the wording in C, can be used in decimal, octal, and hexadecimal.

The identifier 'entry' on the 14th line is a reserved key for forcing the linker ld86 to include in the generated executable file a label 'start' designated thereafter. Usually when linking multiple object files to generate an executable file, you should specify an entry label in the assembler with the keyword entry for debugging purposes. But in our example and in the Linux kernel boot/bootsect.s and boot/setup.s assembler we can omit this keyword because we don't want to include any symbol information in the generated pure binary executable file.

On line 16 is an inter-segment far jump statement, which jumps to the next instruction. When the BIOS loads the program into physical memory at 0x7c00 and jumps to it, the default value of all segment registers (including CS) is 0, that is, CS:IP=0x0000:0x7c00. Therefore, the inter-segment jump statement is used here to assign the segment value 0x7c0 to CS. After the statement is executed, CS:IP = 0x07C0:0x0005. The next two statements assign values to the DS and ES segment registers, respectively, so that they point to the 0x7c0 segment. This makes it easy to address data (strings) in the program.

The MOV instruction on line 20 is used to store the high byte (0x07) of the 0x7c0 segment value in the ah register to the last '.' position in the memory string msg1. This character will cause the BIOS interrupt to beep when the string is displayed. This statement is mainly used to illustrate the use of indirect operands. In as86, indirect operands require square bracket pairs. Some other addressing methods have the following:

```
! Direct register addressing. Jump to the address specified by bx, that is, copy bx to the IP.
    mov     bx, ax
    jmp     bx
! Indirect register addressing. The bx specifies the memory location as the address of the jump.
    mov     [bx], ax
    jmp     [bx]
! Put the immediate number 1234 into ax. Put the msg1 address value in ax.
    mov     ax, #1234
    mov     ax, msg1
! Absolute addressing. Put the contents of the memory address 1234 (msg1) into ax.
    mov     ax, 1234
    mov     ax, msg1
    mov     ax, [msg1]
! Index addressing. Put the value at the memory location indicated by the second operand into ax.
    mov     ax, msg1[bx]
    mov     ax, msg1[bx*4+si]
```

The statements on lines 21-25 are used to put immediate data in the appropriate registers. The # must be preceded by an immediate number, otherwise it will be used as a memory address to make the statement an absolute addressing statement. See the example above. In addition, when putting the address value of a label (such as msg1) into a register, it must be preceded by a '#', otherwise it will become the register at the address of msg1!

Line 26 is the BIOS screen display interrupt call int 0x10. Its function 19, sub-function 1 is used here. The purpose of this interrupt is to write a string (msg1) to the screen at the specified location. The register cx is a string length value, dx is a display position value, bx is a display used character attribute, and es:bp points to a character string.

Line 27 is a jump statement that jumps to the current instruction. So this is an endless loop statement. The endless loop statement is used here to allow the displayed content to stay on the screen without being deleted. Dead-loop statements are commonly used when debugging assembler programs.

Lines 28-29 define the string msg1. Defining a string requires the use of the pseudo-operator '.ascii' and enclosing the string in double quotes. The pseudo operator '.asciiz' also automatically adds a NULL(0) character after the string. In addition, line 29 defines carriage return and line feed (13, 10) characters. Defining characters requires the use of the pseudo-operator '.byte' and enclosing characters in single quotes. For example: "D". Of course, we can write the ASCII code of characters directly as in the example.

The pseudo-operator statement '.org' on line 30 defines the location of the current assembling. This statement will adjust the position counter value of the current segment during compilation of the assembler to the value given on the pseudo-operator statement. For this example program, this statement sets the location counter to 510 and places a valid boot sector flag word 0xAA55 here (line 31). The pseudo-operator '.word' is used to define a double-byte memory object (variable) at the current location, which can be followed by a number or an expression. Since there is no code or data, we can determine from this that the executable compiled by boot.s should be exactly 512 bytes.

Lines 32--37 place three more labels in each of the three segments. Used to indicate the end position of three segments. This setting can be used to distinguish between the start and end of each segment in each module when

linking multiple target modules. Because both the bootsec.s and setup.s programs in the kernel are separately compiled and linked, each expecting to generate a pure binary file does not link with other object module files. Therefore, the pseudo program for each segment is declared in the sample program. The characters (.text, .data, and .bss) can all be omitted. That is, the lines 4 - 11 and 32 - 37 of the program can all be deleted and the link can be compiled to produce the correct result.

3.1.3 as86 assembly language program compilation and link

Now we show how to compile the link sample program boot.s to generate the boot sector program we need to boot. Compiling and linking the above example program requires the following first two commands:

```
[/root]# as86 -O -a -o boot.o boot.s           // Compile. Generate the target file.
[/root]# ld86 -O -s -o boot boot.o             // link. Remove symbol information.
[/root]# ls -l boot*
-rwx--x--x  1 root    root          544 May 17 00:44 boot
-rw-----  1 root    root          249 May 17 00:43 boot.o
-rw-----  1 root    root          767 May 16 23:27 boot.s
[/root]# dd bs=32 if=boot of=/dev/fd0 skip=1    // Write to a floppy disk or Image file.
16+0 records in
16+0 records out
[/root]# _
```

Among them, the first one uses the as86 assembler to compile the boot.s program to generate the boot.o object file. The second command uses a linker ld86 to perform a link operation on the target file, and finally generates a MINIX structure executable file boot. The option '-O' is used to generate the 8086 16-bit target program; '-a' is used to specify that code that is compatible with the GNU as and ld parts is generated. The '-s' option is used to tell the linker to remove the symbol information from the last generated executable. '-o' specifies the name of the generated executable file.

As can be seen from the filenames listed above using the ls command, the last generated boot program is not exactly 512 bytes as stated earlier, but is 32 bytes long. These 32 bytes are the structure of the MINIX executable's header (see the "Creating a Kernel Component" chapter for a detailed structure description). In order to use this program to boot the machine, we need to manually remove the 32 bytes. There are several ways to remove this header structure:

- Use the binary editor to delete the first 32 bytes of the boot program and save it;
- Using the as86 compile linker on current Linux systems (eg RedHat 9), which have the option of generating a pure binary executable without the MINIX header structure, please refer to the online user manual (man as86) of the relevant system.
- Use the Linux system's dd command.

The third command listed above is to use the dd command to remove the first 32 bytes of the boot, and write the output directly to the floppy disk image file of the floppy disk or Bochs simulation system. (Please use the Bochs PC analog system. Refer to the last chapter). If we run this program in the Bochs simulation system, we can get the screen shown in Figure 3-1.

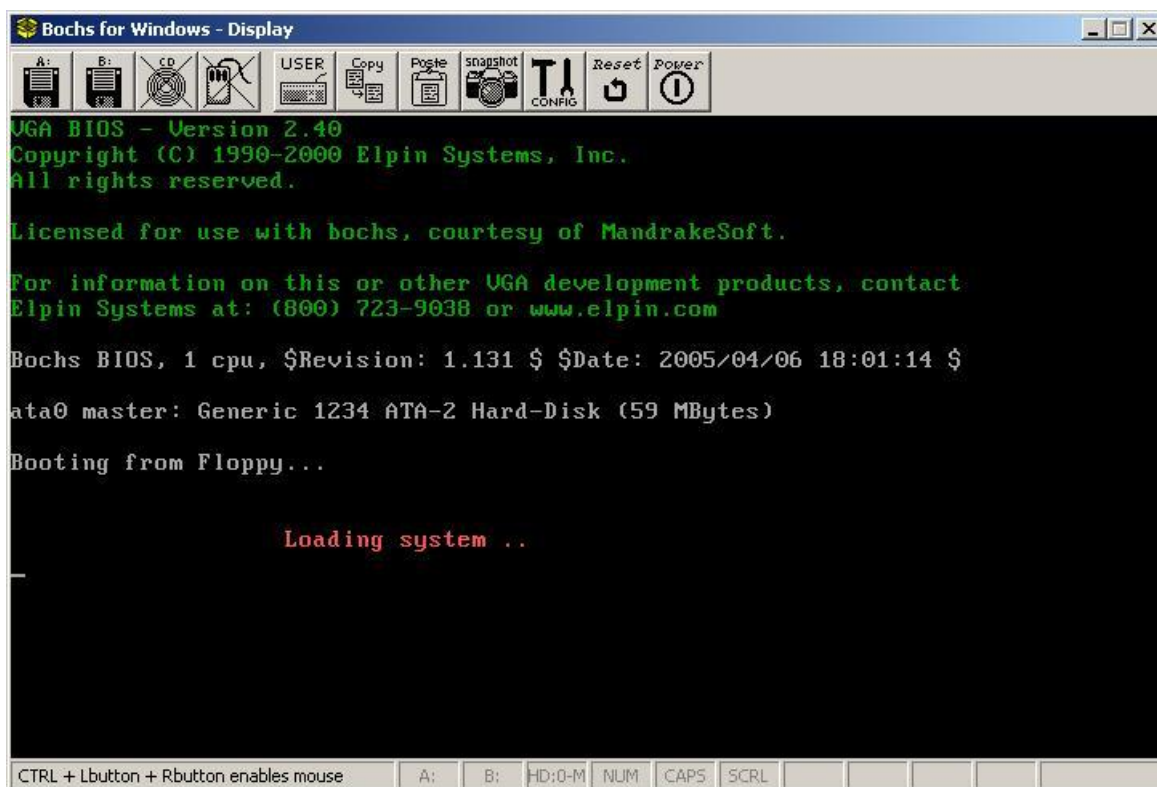


Figure 3-1 Running the boot program in the Bochs simulation system

3.1.4 as86 and ld86 usage methods and options

The usage methods and options of as86 and ld86 are as follows:

Usage and options of as:

```
as [-O3agjuw] [-b [bin]] [-lm [list]] [-n name] [-o objfile] [-s sym] srcfile
```

Default settings (Other than the defaults below, other options default to off or none; if you do not specify the a flag, there will be no output):

-3 Use the 80386 32-bit output;
list Display on standard output;
name The basic name of the source file (that is, does not include the extension after '.');

The meaning of each option:

-O Use 16-bit code segments;
-3 Use 32-bit code segments;
-a Open some compatibility options with GNU as, ld;
-b Generate binary files, followed by the file name;
-g Only global symbols are stored in the object file;
-j Make all jump statements long jumps;
-l Generate a list file, followed by the list file name;
-m Extend the macro definition in the list;
-n Followed by the module name (in place of the source file name into the target file);
-o Produce the target file, followed by the target file name (objfile);
-s Produce a symbol file followed by a symbol file name;
-u The undefined symbol as the symbol of the input unspecified segment;
-w No warning message is displayed;

ld linker usage syntax and options:

For version of generating Minix a.out format:

```
ld [-O3Mims[-]] [-T textaddr] [-llib_extension] [-o outfile] infile...
```

For version of generating GNU-Minix a.out format:

```
ld [-O3Mimrs[-]] [-T textaddr] [-llib_extension] [-o outfile] infile...
```

Default settings (except for the defaults below, other options are off or none by default):

-O3 32-bit output;

outfile a.out format output;

-O Generate a header structure with 16-bit magic numbers, use i86 subdirectory for -lx option;
-3 Generate a header structure with a 32-bit magic number, use i386 subdirectory for -lx option;
-M Display linked symbols on standard output devices
-T Followed by the text base address (using the format suitable for strtoul);
-i Separate instruction and data segment (I&D) output;
-lx Add the library /local/lib/subdir/libx.a to the linked file list;
-m Display linked modules on standard output devices
-o Specify the output file name;
-r Generate output suitable for further relocations;
-s Remove all symbols in the target file.

3.2 GNU as assembler

The as86 assembler introduced in the previous section is only used to compile the boot/bootsect.s boot sector program in the kernel and the boot/setup.s setup program in real mode. All other assembly language programs in the kernel (including those generated by the C language) are compiled with gas and linked with the modules generated by the C language program. This section describes the use of the assembler syntax and the GNU as assembler (as assembler) in the Linux kernel based on the 80X86 CPU hardware platform. We first introduce the syntax of as assembly language programs, and then give the meaning and use of common assembly directives (indicators). As assembly language program examples with detailed instructions will be given at the end of the next chapter.

Because many key code requirements of the operating system require high execution speed and efficiency, about 10% of the key assembly language programs are usually included in an operating system source code. The Linux operating system is no exception. Its 32-bit initialization code, all interrupt and exception handling interface programs, and many macro definitions all use as assembly language programs or extended embedded assembly statements. Whether or not you can understand the functions of these assembly language programs will undoubtedly become one of the key points for understanding the concrete implementation of an operating system.

When compiling a C program, the GNU gcc compiler first outputs an as assembly language file as an intermediate result, and then gcc calls the as assembler to compile the temporary assembly language program into a target file. That is, as the assembler was originally designed to assemble the intermediate assembly language program generated by gcc, rather than being used as a standalone assembler. Therefore, as assembler also supports many C language features, including characters, numbers, and constant representation methods as well as expression forms.

The GNU as assembler was originally developed following the assembler of BSD 4.2. The current as assembler can be configured to generate many different formats of object files. Although the compiled assembly

language program is not related to the target file that is used or generated in any format, if the target file format is involved in the following description, we will describe the a.out target file format adopted in the Linux 0.12 system. .

3.2.1 Compiling as assembly language program

The basic command line format for compiling an assembler program using the as assembler is as follows:

```
as [ options ] [ -o objfile ] [ srcfile.s ... ]
```

Where objfile is the target file name for the as compile output, and srcfile.s is the input assembly language program name for as. If you do not use the output file name, then as compiles the default destination file named a.out. After the as program name, the command line can include compilation options and file names. All options are free to place, but the result of compiling the filenames is closely related.

A program's source code can be placed in one or more files. How the program's source code is split across several files does not change the program's semantics. The source code of the program is the combined result of all these files in order. Every time you run the as compiler, it compiles only one source program. But a source program can consist of multiple text files (the terminal's standard input is also a file).

We can give zero or more input file names on the as command line. As will read the contents of these input files from left to right. If the parameters at any position on the command line do not have a specific meaning, they will be treated as an input file name. If no filename is given on the command line, as will attempt to read the input file content from the terminal or console standard input. In this case, if there is no content to input, you need to manually type Ctrl-D key combination to tell the as assembler. If you want to explicitly specify the standard input as an input file on the command line, you need to use the parameter '--'.

The output file of as is the binary data file compiled by the input assembly language program, ie the target file. Unless we specify the name of the output file with the option '-o', as will produce an output file named a.out. The target file is mainly used as an input file for the linker ld. The object file contains compiled program code, information that assists ld in producing an executable program, and possibly debugging symbol information. The a.out object file format used in the Linux 0.12 system will be described later in this chapter.

If we want to compile the boot/head.s assembler separately, we can type the following command on the command line:

```
[/usr/src/linux/boot]# as -o head.o head.s
[/usr/src/linux/boot]# ls -l head*
-rw-rwxr-x  1 root    root      26449 May 19 22:04 head.o
-rw-rwxr-x  1 root    root      5938  Nov 18  1991 head.s
[/usr/src/linux/boot]#
```

3.2.2 as assembly syntax

In order to maintain compatibility with the gcc output assembler, the assembler uses the AT&T System V assembler syntax (hereinafter referred to as AT&T syntax). This syntax is very different from the syntax used by the Intel assembler (Intel syntax for short), and there are several major differences between them:

- In the AT&T syntax, an immediate value is preceded by a character '\$'; the register operand name is preceded by the character percent sign '%'; absolute jump/invoke (relative to the program counter's jump/invoke) operands To add asterisk '*'. Intel assembly syntax does not have these limitations.

- The order of the source and destination operands used by AT&T syntax and Intel syntax is exactly the opposite. AT&T's source and destination operands are "source, destination" from left to right. For example Intel's statement 'add eax, 4' corresponds to AT&T's 'addl \$4, %eax'.
- The length (width) of the memory operand in AT&T syntax is determined by the last character of the opcode. The operand suffixes 'b', 'w', and 'l' indicate that the memory reference width is 8 bits byte, 16-bit words, and 32-bit long words, respectively. Intel syntax achieves the same purpose by using the prefixes 'byte ptr', 'word ptr', and 'dword ptr' before the memory operands. Therefore, Intel's statement 'mov al, byte ptr foo' corresponds to AT&T's statement 'movb \$foo, %al'.
- In the AT&T syntax, immediate and far-form calls in the immediate form are 'ljmp/lcall \$section, \$offset', while Intel's is 'jmp/call far section:offset'. Similarly, in the AT&T syntax, the far return instruction 'ret \$stack-adjust' corresponds to Intel's 'ret far stack-adjust'.
- The AT&T assembler does not provide support for multi-segment programs because UNIX-like operating systems require that all code be in one segment.

3.2.2.1 Assembler preprocessing

as assembler has a built-in simple preprocessing function for assembly language programs. This preprocessing function adjusts and removes extra space characters and tabs; removes all comment statements and replaces them with a single space or some newline character; converts character constants to their corresponding values. However, this preprocessing function does not process the macro definition nor handle the function of the include file. If this function is needed, then the assembly language program can use the uppercase suffix '.S' to make as use the gcc CPP preprocessing function.

Since the as assembly language program uses C comment statements (that is, '/' and '/'), it also uses the hash symbol '#' as a single-line comment start character, so if the program is not preprocessed before assembly, then All indicators or commands included in the program that begin with the hash sign '#' are treated as part of the comment.

3.2.2.2 Symbols, Statements, and Constants

Symbols are identifiers composed of characters, and the valid characters that make up the symbol are taken from the uppercase and lowercase character sets, numbers, and the three characters '_.\$'. Symbols are not allowed to start with numeric characters and the capitalization is different. There is no limit to the symbol length in as assembler, and all characters in the symbol are valid. Symbols use other characters (such as spaces, line breaks) or the beginning of the file to define the beginning and ending points.

The statement ends with a line break or a line break character (';'). The final statement of the file must end with a newline character.

If you use the backslash character '\' (before the newline) at the end of a line, you can use multiple lines for a statement. When as reads a backslash plus a newline, it ignores the two characters.

Statements start with zero or more labels, followed by a key symbol that determines the type of statement. The label consists of a symbol followed by a colon (':'). The key symbols determine the semantics of the rest of the statement. If the key symbol begins with a '.', the current statement is an assembly command (or directive, indicator). If the key symbol begins with a letter, the current statement is an assembly language instruction statement. So the general format of a statement is:

label:	.directive	followed by optional some comments
another_label:	#	This is an empty statement.
	instruction	operand_1, operand_2, ...

A constant is a number that can be divided into character constants and numeric constants. Character

constants can also be divided into strings and single characters; numeric constants can be divided into integers, large numbers, and floating-point numbers.

Strings must be enclosed in double quotes, and they can be escaped with special characters by using a backslash `\`. For example, `\\` indicates a backslash character. The first backslash is an escape indicator, which indicates that the second character is treated as a normal backslash character. The common escape sequences are shown in Table 3-1. If the backslash is followed by another character, the backslash will not work and the assembler will issue a warning message.

When using a single character constant in the assembler, you can write a single quotation mark before the character. For example, `'A'` indicates the value 65, and `'C'` indicates the value 67. The escape codes in Table 3-1 can also be used for single character constants. For example, `'\\'` indicates a common backslash character constant.

Table 3-1 As assembler supports escaped character sequences

Escape code	Description
<code>\b</code>	Backspace, value is 0x08
<code>\f</code>	Formfeed, value 0x0C
<code>\n</code>	Newline, value 0x0A
<code>\r</code>	Carriage-Return value is 0x0D
<code>\NNN</code>	Character code represented by 3 octal numbers
<code>\xNN...</code>	Hexadecimal number character code
<code>\\</code>	Represents a backslash character
<code>\"</code>	Represents a double quote in a string <code>""</code>

Integer numeric constants are represented in four ways, ie binary numbers starting with `'0b'` or `'0B'` ('0-1'); octal numbers starting with `'0'` ('0-7'); non-zero The decimal number starting with the digit ('0-9') and the hexadecimal number starting with `'0x'` or `'0X'` ('0-9a-fA-F'). To represent a negative number, just precede the negative `'-'`.

A Bignum is a number of bits more than 32 bits, which means that the method is the same as the integer. The representation of floating-point constants in assembler is basically the same as in C language. Since almost no floating point numbers are used in the kernel code, it is not described here.

3.2.3 Instruction statements, operands, and addressing

The instruction is the operation performed by the CPU. Usually the instruction is also called the opcode. The operand is the object of the instruction operation. The address is the position of the specified data in the memory. An instruction statement is a statement executed at the execution time of a program. It can usually consist of four components:

- Label (optional);
- Opcode (instruction mnemonic);
- Operands (specified by specific instructions);
- Comments

An instruction statement can contain zero or up to three comma separated operands. For an instruction statement with two operands, the first is the source operand and the second is the destination operand, ie the result of the instruction operation is stored in the second operand.

The operand can be an immediate value (that is, an expression whose value is a constant value), a register (value in the CPU's register), or memory (value in memory). An indirect operand (Indirect operand) contains the address value of the actual operand value. The AT&T syntax specifies an indirect operand by prepending the operand with a '*' character. Indirect operands can only be used by redirection/call instructions. See description of jump instructions below.

- A '\$' character prefix is required before immediate operands;
- A '%' character prefix needs to be preceded by a register name;
- The memory operand is specified by a variable name or a register containing the address of the variable. The variable name implicitly indicates the address of the variable and instructs the CPU to reference the contents of the memory at that address.

3.2.3.1 Name the instruction opcode

The last character of the instruction opcode name (ie, the instruction mnemonic) in AT&T syntax is used to indicate the width of the operand. The characters 'b', 'w', and 'l' specify byte, word, and long operands, respectively. If the instruction name does not have such a character suffix and the instruction statement does not contain a memory operand, as will try to determine the operand width based on the destination register operand. For example, the instruction statement 'mov %ax, %bx' is equivalent to 'movw %ax, %bx'. Similarly, the statement 'mov \$1, %bx' is equivalent to 'movw \$1, %bx'.

The names of almost all instruction opcodes in AT&T and Intel syntax are the same, but there are still a few exceptions. Both symbolic extensions and zero-extend instructions require two widths to indicate that the width needs to be specified for the source and destination operands. AT&T syntax is done by using two opcode suffixes. The basic opcode names for symbol expansion and zero extension in AT&T syntax are 'movs...' and 'movz...', respectively, in Intel are 'movsx' and 'movzx' respectively. Two suffixes are attached to the basic name of the opcode. For example, an AT&T statement that uses symbolic extensions to move from %al to %edx is 'movsbl %al, %edx', which is bl from byte to long, bw from byte to word, and wl from word to long. The correspondence between AT&T syntax and conversion instructions in Intel syntax is shown in Table 3-2.

Table 3-2 Correspondence between conversion command in AT&T syntax and Intel syntax

AT&T	Intel	Description
cbtw	cbw	Extend the byte value in %al to %ax
cwtl	cwde	Extend the %ax sign to %eax
cwtd	cwd	Extend the %ax sign to %dx:%ax
cltd	cdq	Extend the %eax sign to %edx:%eax

3.2.3.2 Instruction opcode prefix

The opcode prefix is used to modify subsequent opcodes. They are used to repeat string instructions, provide area overrides, perform bus lock operations, or specify operands and address widths. Normally, the opcode prefix can be used as an exclusive line of instructions without an operand and must be located directly before the affected instruction, but it is best to place it on the same line as the instruction it modifies. For example, the string scan command 'scas' uses prefixes to perform repeated operations:

```
repne scas %es:(%edi), %al
```

Some operand prefixes are listed in Table 3-3.

Table 3-3 Opcode prefix list

Opcode prefix	Description
cs, ds, ss, es, fs, gs	Section overrides the opcode prefix. Using the section:memory operands by specifying memory prefixes automatically adds this prefix.
data16, addr16	Operand/address width prefix. These two prefixes will change the 32-bit operand/address to a 16-bit operand/address. However, please note that as does not support 16-bit addressing.
lock	Bus latching prefix. Used to disable interrupts during instruction execution (only valid for some instructions, see the 80X86 manual).
wait	Coprocessor instruction prefix. Wait for the coprocessor to complete the execution of the current instruction. This prefix is not needed for the 80386/80387 combination.
rep, repe, repne	The prefix of the string instruction causes the string instruction to repeat the specified number of times in %ecx.

3.2.3.3 Memory reference

Indirect memory reference form of Intel syntax: section:[base + index*scale + disp]

Corresponds to the following AT&T syntax: section:disp(base, index, scale)

Base and index are optional 32-bit base registers and index registers, and disp is an optional offset value. Scale is a scale factor and its range is 1, 2, 4 and 8. Scale is multiplied by index to calculate operand address. If no scale is specified, the scale defaults to 1. Section specifies an optional segment register for the memory operand and overrides the current default segment register used by the operand. Note that if the specified section overwrite register is the same as the default operation section register, as does not output the same section prefix for the assembled instructions. The following are examples of memory references in several AT&T and Intel syntax forms:

<code>movl var, %eax</code>	# Put the contents at memory address var in the register %eax.
<code>movl %cs:var, %eax</code>	# Put the contents at var in the code segment into %eax.
<code>movb \$0x0a, %es:(%ebx)</code>	# Save byte value 0x0a to offset specified by %ebx in es segment.
<code>movl \$var, %eax</code>	# Put the address of var in %eax.
<code>movl array(%esi), %eax</code>	# Put contents at address determined by array+%esi into %eax.
<code>movl (%ebx, %esi, 4), %eax</code>	# Put contents at address determined by %ebx+%esi*4 in %eax.
<code>movl array(%ebx, %esi, 4), %eax</code>	# Put contents at address of array + %ebx+%esi*4 into %eax.
<code>movl -4(%ebp), %eax</code>	# Put contents at %ebp -4 in %eax, using the default segment %ss.
<code>movl foo(, %eax, 4), %eax</code>	# Put contents at foo+eax*4 into %eax, using default seg %ds.

3.2.3.4 Jump instruction

Jump instructions are used to move the execution point to another location in the program and continue execution. The destination of these jumps is usually represented by a label. When generating the object code file, the assembler will determine the address of all tagged instructions and encode the address of the jumped instruction into the jump instruction. Jump instructions can be divided into unconditional jumps and conditional jumps. The conditional jump instruction will depend on the state of a related flag in the flag register when the instruction is executed to determine whether to jump, and the unconditional jump does not depend on these flags.

JMP is an unconditional jump instruction and can be divided into two types: direct jump and indirect jump, whereas conditional jump instructions only have the form of a direct jump. For a direct jump instruction, the address of the jumped target instruction is directly encoded into the jump instruction as part of the jump instruction; for an indirect jump instruction, the jump destination is taken from a register or a Memory locations. The direct jump statement is written to give the label at the jump target; the indirect jump statement is written using a star character '*' as the prefix character of the operation indicator, and the operation indicator uses the same syntax as the movl instruction. . The following are some examples of direct and indirect jumps.

jmp NewLoc	# Jump directly. Unconditionally jump to label NewLoc to continue execution.
jmp %eax	# Indirect jump. The value of register %eax is the jump destination.
jmp *(%eax)	# Indirect jump. Read the jump destination from the address indicated by %eax.

Similarly, indirect call operands that are independent of the instruction counter PC must also have a '*' as the prefix character. If the '*' character is not used, the as assembler will select the jump label associated with the instruction count PC. Also, any other instruction that has a memory operand must use an opcode suffix ('b', 'w', or 'l') to indicate the size of the operand (byte, word, or long).

3.2.4 Sections and Relocation

Sections (also called segments) are used to represent an address range, and the operating system will treat and process the data information in that address range in the same way. For example, there may be a "read only" area, and we can only read data from this area and cannot write it. The concept of a zone is mainly used to indicate different information areas in a target file (or executable program) generated by a compiler, such as a text area or a data area in a target file. To properly understand and compile an assembly language program, we need to understand the format of the output object file produced by as. A detailed description of the a.out format object file format used by the Linux 0.12 kernel is given later in this chapter. Here, a brief introduction to the basic concepts of the zone is provided to understand the basic structure of the object file produced by the assembler.

The linker ld will combine the contents of the input object file according to a certain rule to generate an executable program. When the as assembler outputs a target file, the code in the target file is set by default to start at address 0. After that, ld will allocate different final address locations for each part of the different target files during the linking process. Ld moves the block of bytes in the program to the address where the program was run. These blocks are moved as fixed units. Their length and byte order will not be changed. Such a fixed unit is called a zone (or segment, part). The operation of allocating the address of the runtime for a zone is referred to as a relocation operation, which includes adjusting the addresses recorded in the target file so that they correspond to the appropriate runtime address.

The as-assembler creates and outputs an object file with at least 3 fields, which are called the text, data, and bss areas. Each district may be empty. If you do not use the assembler command to place the output in the '.text' or '.data' zone, these zones will still exist but the content is empty. In a target file, its text area starts at address 0, followed by the data area, followed by the bss area.

When a section is relocated, in order for the linker ld to know what data will change and how to modify the data, the assembler will also write the required relocation information to the target file. In order to perform a relocation operation, ld must know each time an address in the target file is involved:

- Where did the reference to an address in the target file come from?
- What is the length of the quoted byte?
- Which section is referenced by this address? What is the value of (address)-(start address of section)?
- Is the reference to the address related to the program counter PC (Program-Counter)?

In fact, all the addresses used by `as` can be expressed as: (section) + (offset in the section). In addition, most of the expressions evaluated by `as` have such zone-related characteristics. In the following description, we use the notation "{secname N}" to indicate the offset N in the secname of the zone.

In addition to the text, data, and bss areas, we also need to understand the absolute address area (absolute area). When the linker combines the various object files, the address in the absolute area will always be the same. For example, `ld` will "relocate" the address {absolute 0} to address 0 at runtime. Although the linker will never arrange the data areas in the two target files as overlapping addresses after linking, the absolute area in the target file must overlap and be overwritten.

There is also an Undefined section. It is not possible to determine at assembly that any address in the area is set to {undefined U}, where U will be filled in later. Because the value is always defined, the only way to present an undefined address involves only undefined symbols. A reference to a common block is such a symbol: Its value is unknown at assembly time, so it is in the undefined area.

Similarly, the section name is also used to describe the group of sections in the linked program. The linker `ld` will put the text section in all object files of the program at the adjacent address. The text area of the program that we are accustomed to refer to actually refers to the entire address area formed by the combination of the text sections of all of its object files. The same is true for the understanding of the data and bss sections in the program.

3.2.4.1 Linker involved sections

The linker `ld` only involves the following 4 types of sections:

- Text section, data section -- These two areas are used to save programs. `As` and `ld` treat them independently and equally. The description of the text section is also suitable for the data section. However, when the program is running, the usual text section will not change. The text section is usually shared by the process and contains the instruction code and constants. The contents of the data section usually change when the program is running. For example, C variables are usually stored in the data section.
- bss section -- This area contains 0 bytes when the program starts running. This area is used to store uninitialized variables or as a common variable storage space. Although the length information of the bss section of each target file of the program is very important, since the area stores zero-value bytes, there is no need to save the bss section in the target file. The purpose of setting the bss area is to explicitly exclude zero-value bytes from the target file.
- Absolute section -- The address 0 of this area is always "relocated" to the address 0 of the runtime. Use this section if you do not want `ld` to change the address you are referencing when relocating. From this point of view, we can refer to absolute addresses as "non-relocatable": they do not change during relocation operations.
- undefined section -- A reference to an object that is not in each of the previously mentioned sections belongs to this section.

An example of 3 idealized relocatable sections is shown in Figure 3-2. This example uses the traditional section names: '.text' and '.data'. The horizontal axis indicates the memory address. The specific operation of the `ld` linker will be described in detail later in this section.

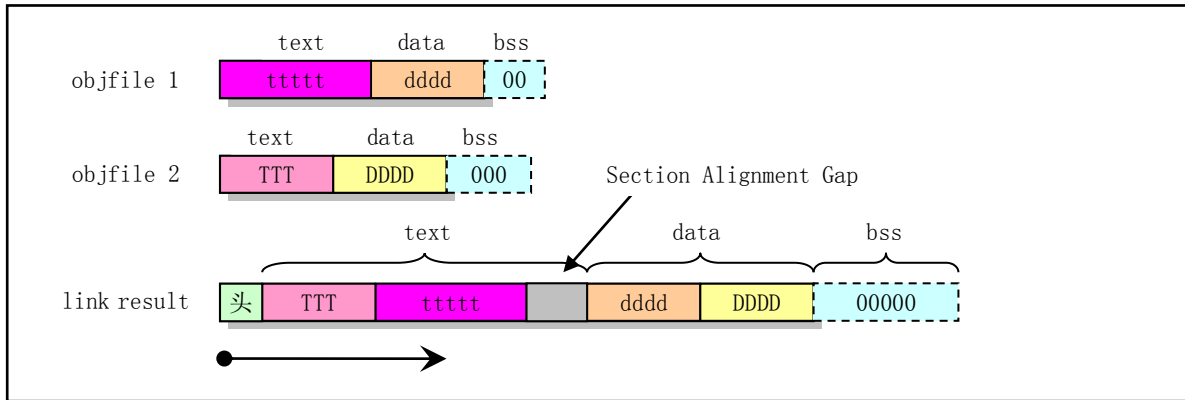


Figure 3-2 Example of linking two object files to generate a linked program

3.2.4.2 Subsection

The byte data that is assembled is usually located in the text or data section. Sometimes there may be some non-adjacent data groups in an area of an assembler source program, but you may want them to be stored together after assembly. The `as` assembler allows you to use subsections for this purpose. In each section, there may be a sub-area numbered 0–8192. Objects that are programmed in the same subsection are put together in the target file with other objects in that subsection. For example, the compiler may want to store constants in the text area, but do not want these constants to be scattered throughout the program being assembled. In this case, the compiler can use the `'.text 0'` subsection before each code area that is output, and use the `'.text 1'` subsection before each set of constants.

Using subsections is optional. If you do not use subsections, all objects are placed in subsection 0. Sub-sections appear in the destination file in the order of their numbers, but the destination file does not contain any information that represents sub-sections. The `ld` and other programs that process the target file do not see the traces of the subsections; they only see the text section consisting of all the text subsections; the data section consisting of all the data subsections. In order to specify which subsection area the subsequent statement is assembled into, you can use numeric parameters in `'.text expression'` or `'.data expression'`. The result of the expression should be an absolute value. If only `'.text'` is specified then `'.text 0'` is used by default. Similarly, `'.data'` indicates that `'.data 0'` is used.

Each section has a Location Counter that counts each byte that is assembled into the section. Because the subsections are only set up for ease of use by the assembler, there is no subsection counter. Although there is no direct way to manipulate a position counter, the assembly command `'.align'` can change its value, and any label definition will take the current value of the position counter. The location counter of the zone that is executing statement assembly processing is called the current activity counter.

3.2.4.3 bss section

The bss section is used to store local public variables. You can allocate space in the bss section, but you can't put data in it before the program runs. Because when the program starts executing, all bytes in the bss section will be cleared. The `'.lcomm'` assembly command is used to define a symbol in the bss section; `'.comm'` can be used to declare a public symbol in the bss section.

3.2.5 symbol

In the process of program compilation and linking, Symbol is an important concept. Programmers use symbols to name objects, linkers use symbols for link operations, and debuggers use symbols for debugging.

Label is a symbol followed by a colon. At this point the symbol represents the current value of the active

position counter and can, for example, be used as the operand of the instruction. We can use the equal sign '=' to assign an arbitrary value to a symbol.

The symbol name starts with one of the letters or '.' characters. Local symbols are used to help compilers and programmers use names temporarily. A total of 10 local symbol names ('0'...'9') are available for reuse in a program. To define a local symbol, simply write a label of the form 'N:' (where N represents any number). If you refer to the previously defined symbol, you need to write 'Nb'; if you want to refer to the next defined local label, you need to write 'Nf'. Where 'b' means backwards and 'f' means forwards. There are no restrictions on the use of local labels, but at any time we can only refer to the furthest 10 local labels forward/backward.

3.2.5.1 Special point symbol

The special symbol '.' indicates as the current address of the assembly. So the expression 'mylab: .long .' will define mylab to contain its own address value. Assigning a value to '.' is the same as the assembly command '.org'. So the expression ' .=. +4' is exactly the same as '.space 4'.

3.2.5.2 Symbol attributes

In addition to the names, each symbol has the "value" and "type" attributes. Depending on the format of the output, symbols can also have auxiliary attributes. If a symbol is used without definition, as will assume all its attributes are 0. This indicates that the symbol is an externally defined symbol.

The value of the symbol is usually 32 bits. For a symbol that marks a position in the text, data, bss, or absolute area, the value is the address value from the beginning of the area to the label. For the text, data, and bss areas, the value of a symbol will usually change during the linking process due to the change of the base address of the area, and the value of the symbol in the absolute area will not change. This is why they are called absolute symbols.

ld deals with the value of undefined symbols. If the value of the undefined symbol is 0, it means that the symbol is not defined in the assembler source program; ld will try to determine its value from other linked files. A symbol is generated when the program uses a symbol but does not define the symbol. If the value of an undefined symbol is not 0, then the symbol value represents the public memory space length that is required by the .comm public declaration. The symbol points to the first address of this memory space.

The type attribute of the symbol contains relocation information for the linker and the debugger, a flag indicating that the symbol is external, and some other optional information. For object files in a.out format, the symbol's type attribute is stored in an 8-bit field (n_type bytes). See the description of the include/a.out.h file for its meaning.

3.2.6 as assembler directives

Assembler directives are pseudo instructions that indicate the way an assembler operates. Assembler directives are used to require the assembler to allocate space for variables, determine the program start address, specify the current assembly sections, modify the position counter value, and so on. All assembler directives begin with a '.', the rest are characters, and the case is irrelevant. However, lowercase characters are generally used. Below we give a description of some common assembler instructions.

3.2.6.1 .align abs-expr1, abs-expr2, abs-expr3

.align is a storage-align assembler directive that sets (increments) the position counter value to the next specified memory boundary in the current subsection. The first absolute value expression abs-expr1 (absolute expression) specifies the required boundary alignment value. For an 80X86 system that uses a.out format object files, the value of this expression is the number of zero-valued bits on the rightmost binary value of the position counter after it has been incremented, that is, a power of two. For example, '.align 3' means to increase the position counter value to a multiple of 8. If the position counter value itself is a multiple of 8, then there is no need

to change it. But for 80X86 systems that use the ELF format, the expression value is directly the number of bytes required for it. For example, `'align 8'` is to increase the position counter value to a multiple of 8.

The second expression gives the byte value to use for alignment and padding. This expression and its preceding comma can be omitted. If omitted, the padding byte value is 0. The third optional expression, `abs-expr3`, is used to indicate the maximum number of bytes allowed for padding to be skipped by an alignment operation. If the number of bytes skipped by the alignment operation is greater than this maximum value, the alignment operation is canceled. If you want to omit the second parameter, you can use two commas between the first and third parameters.

3.2.6.2 .ascii "string"...

Allocate space for the string from the current location of the location counter and store the string. Multiple strings can be written separately using commas. For example, `'.ascii "Hellow world!", "My assembler"'`. The assembler instruction will have as assemble these strings at consecutive address locations, with no 0 (NULL) bytes added after each string.

3.2.6.3 .asciz "string"...

This assembler directive is just like `'.ascii'`, but each string is followed by a zero value byte. The `"z"` in `'.asciz'` stands for "zero".

3.2.6.4 .byte expressions

This directive expects zero or more expressions, separated by commas. Each expression is combined into the next byte.

3.2.6.5 .comm symbol, length

Declare a named public area in the bss section. During the ld link, a common symbol in one object file is merged with the common symbol with the same name in other object files. If ld does not find a symbol definition but only one or more common symbols, then ld will allocate uninitialized memory of length length bytes. Length must be an absolute value expression. If ld finds multiple common symbols with the same length but different names, ld allocates the space with the largest length.

3.2.6.6 .data subsection

This assembler directive tells as to assemble the following statements into the data subsection numbered subsection. If you omit the number, the number 0 is used by default. The number must be an absolute value expression.

3.2.6.7 .desc symbol, abs-expr

This directive sets the descriptor of the symbol to the low 16 bits of an absolute expression. It is only for a.out or b.out object format. See the description of the include/a.out.h file.

3.2.6.8 .fill repeat, size, value

This assembler directive will generate a repeat (repeat) of N bytes in size. The size value can be 0 or some value, but if size is greater than 8, it is limited to 8. The conetnts of each repeat bytes is taken from an 8-byte number. The highest order 4 bytes are zero, and the lowest 4 bytes are numeric values. The three parameter values are absolute values, size and value are optional. If the second comma and value are omitted, value defaults to 0; if the latter two parameters are omitted, size defaults to 1.

3.2.6.9 .global symbol (.globl symbol)

This assembler instruction will cause the linker ld to see the symbol. If symbol is defined in our object file, its value will be used by other object files in the link process. If the symbol is not defined in the object file, its attributes will be obtained from the symbol of the same name in other object files in the linking process. This is done by setting the external bit N_EXT in the symbol symbol type field. See the description in the include/a.out.h file.

3.2.6.10 .int expressions

The assembler directive sets 0 or more integer values in a certain area (80386 system is 4 bytes, same as .long). The value of each comma-separated expression is the value of the runtime. For example, .int 1234, 567, 0x89AB.

3.2.6.11 .lcomm symbol, length

The local common area specified for the symbol reserves space of length bytes. The value of the area and symbol that is located is the value of the new local common block. The allocated address is in the bss section, so these byte values are cleared at runtime. Since the symbol is not declared global, the linker ld is invisible.

3.2.6.12 .long expressions

Its meaning is the same as .int.

3.2.6.13 .octa bignums

This assembly directive specifies zero or more comma-separated 16-byte large numbers (.byte, .word, .long, .quad, .octa correspond to 1, 2, 4, 8 and 16 bytes, respectively).

3.2.6.14 .org new_lc, fill

This assembler directive sets the current section's location counter to the value new_lc. New_lc is an absolute value (expression), or an expression that has the same section as a subsection, ie it cannot use .org to span sections. If the section of new_lc is not correct, then .org will not work. Please note that the position counter is section-based, ie each section is used as a starting point for counting.

When the position counter value increases, the skipped bytes will be filled with the value fill. This value must be absolute. If you omit commas and fill, fill defaults to 0.

3.2.6.15 .quad bignums

This assembler directive specifies zero or more comma separated 8-byte large-number bignums. If the large number does not fit into 8 bytes, then take the lower 8 bytes.

3.2.6.16 .short expressions (same as .word expressions)

This assembler directive specifies zero or more comma separated 2-byte numbers in a section. For each expression, a 16-bit value is generated at runtime.

3.2.6.17 .space size, fill

The assembler directive generates size bytes, each of which is filled with fill. This parameter is an absolute value. If commas and fill are omitted, the default value of fill is 0.

3.2.6.18 .string "string"

Define one or more comma-separated strings. Escape characters can be used in strings. Each string is automatically appended with a null-terminated character. For example, .string "\n\nStarting", "other strings".

3.2.6.19 .text subsection

The notification as compiles the following statements into a subsection numbered subsection. If the number subsection is omitted, the default number value 0 is used.

3.2.6.20 .word expressions

For 32-bit machines, this assembly instruction has the same meaning as .short.

3.2.7 Writing 16-bit code

Although GNU as is usually used to write pure 32-bit 80X86 code, it also has limited support for writing code that runs in real mode or 16-bit protected mode after 1995. In order for as compile to generate 16-bit code, it is necessary to add the assembly instruction '.code16' before the instruction statement that is running in 16-bit mode, and use the assembly instruction '.code32' to switch the as-assembler back to 32-bit code assembly mode.

as does not distinguish between 16-bit and 32-bit assembler statements. Each instruction in 16-bit and 32-bit mode functions exactly the same way regardless of the mode. As always generates 32-bit instruction code for assembly statements regardless of whether the instruction will run in 16-bit or 32-bit mode. If the assembly instruction '.code16' is used to put as in 16-bit mode, then as will automatically add a necessary operand-width prefix to all instructions and let the instruction run in 16-bit mode. Note that since as adds extra address and operand-width prefixes for all instructions, the resulting code length and performance of the assembly will be affected.

as assembler did not support 16-bit code when developing the Linux kernel 0.12 in 1991, the as86 assembler described earlier was used when writing and assembling the boot startup code and initializing assembler in the 0.12 kernel real mode.

3.2.8 as assembler command line options

- a Turn on program listings.
- f Fast operation, skip whitespace and comment preprocessing.
- o objfile Name the object-file output from as *objfile*
- R Fold the data section into the text section.
- W Suppress warning messages.

3.3 C language program

GNU gcc has made some extensions to the C language described in ISO standard C89, some of which have been included in the ISO C99 standard. This section gives a description of some of the gcc extensions that are often used in the kernel. A brief explanation of the extended statement encountered will also be given at any time in the following section of program comments.

3.3.1 C program compiling and linking

The use of the gcc assembler to compile C programs usually goes through four stages of processing: the preprocessing stage, the compilation stage, the assembly stage, and the linking stage, as shown in Figure 3-3.

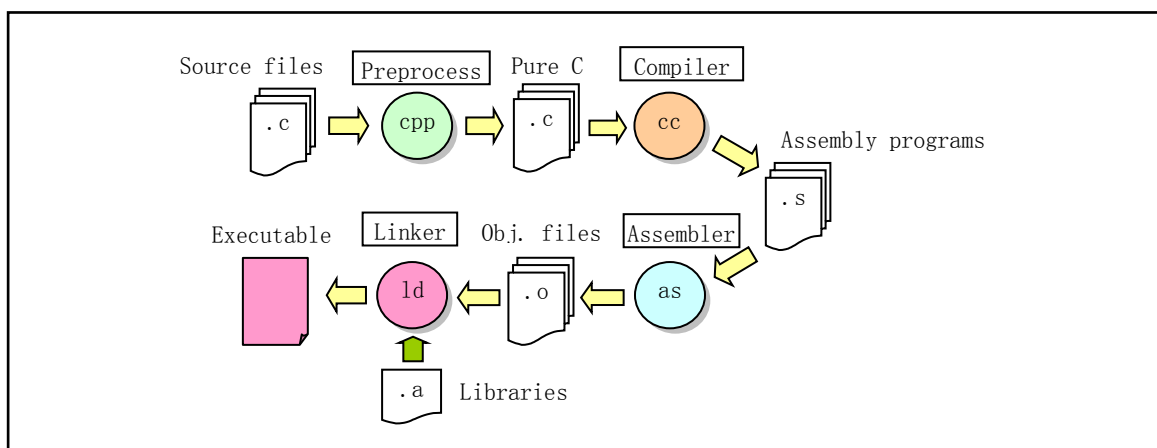


Figure 3-3 CC program compilation process

In the pre-processing stage, gcc passes the C program to the C preprocessor CPP, replaces the indicators and macros in the C language program, and outputs the plain C language code; at the compile stage, gcc compiles the C language program to generate the corresponding Machine-related assembly language code; Assembler stage,

assembler will convert the assembly code into machine instructions, and output in a specific binary format and save it in the target file; Finally, the GNU ld linker links the program's related target file combination Together, the program's executable image file is generated. The command line format for calling gcc is similar to the format for compiling assembly language:

```
gcc [ options ] [ -o outfile ] infile ...
```

Where infile is the input C language file; outfile is the compiled output file. For a compilation process, it is not necessary to execute all four stages. Using the command line option can cause the gcc compilation process to stop executing after a certain processing stage. For example, using the '-S' option allows gcc to stop after outputting the assembly language program for the C program; using the '-c' option allows gcc to only generate the target file without performing link processing, as shown below.

```
gcc -o hello hello.c      // Compile the hello.c to generate the execution file hello.
gcc -S -o hello.s hello.c // Compile hello.c to generate corresponding assembly hello.s.
gcc -c -o hello.o hello.c // Compile hello.c to generate target file hello.o without linking.
```

When compiling a large program such as the Linux kernel containing many source program files, the make tool is usually used to automatically manage the entire program's compilation process. See the description below.

3.3.2 Inline assembly language

This section describes inline assembly statements that are exposed in the kernel C language program. Since we usually use inline assembly code rarely used in the preparation of C programs, it is necessary here to explain the basic format and usage. In an assembler instruction using asm, you can specify the operands of the instruction using C expressions. This means we need not guess which registers or memory locations will contain the data you want to use, and we must specify an assembler instruction template, as it appears in the machine description, and specify an operand constraint string for each operand. The basic format of an inline assembly statement with input and output parameters is as following:

```
asm( "Assembly language statement"
    : Output register operands
    : Input register operands
    : Registers of clobbered or modified);
```

With the exception of the first line, lines with a colon after them can be omitted if they are not used. Among them, "asm" is an inline assembly statement keyword; "assembly statement" is where you write assembly instructions; "output register" indicates which registers are used to store output data after this embedded assembly is executed. Here, these registers correspond to a C expression value or a memory address, respectively; "input register" indicates the input value that should be stored in some of the registers specified here when the assembly code is started. They also correspond to a C variable, or constant value respectively. "Clobbered or Modified registers" means that you have changed the values in the registers listed there, and the gcc compiler can no longer rely on the values it originally loaded on these registers. Gcc needs to reload these registers if necessary. Therefore we need to list those register names that are not listed in the output/input registers section, but that are explicitly used or implicitly used in assembly statements.

For example, here is a fictitious 'combine' instruction:

```
asm( "combine %1, %0" : "=r" (result) : "r" (length));
```

Here length is the C expression for the input operand while result is that of the output operand. Each has "r" as its operand constraint, saying that a dynamically allocated general register is required. The '=' in '=r' indicates that the operand is an output; all output operands' constraints must use '='. The constraints use the same language used in the machine description as described in the gcc manual (section Operand Constraints in Chapter 16 Machine Descriptions).

As shown in the above example, each operand is described by an operand-constraint string followed by the C expression in parentheses. A colon separates the assembler template from the first output operand and another separates the last output operand from the first input, if any. Commas separate the operands within each group. The total number of operands is limited to ten or to the maximum number of operands in any instruction pattern in the machine description, whichever is greater. If there are no output operands but there are input operands, you must place two consecutive colons surrounding the place where the output operands.

Below we use a more detailed example to illustrate the use of inline assembly statements. Here is a block of code starting at line 22 in the kernel/traps.c file. In order to see more clearly, we have rearranged and numbered this code.

```
01 #define get_seg_byte(seg, addr) \
02 ({ \
03     register char __res; \           // A register variable __res is defined.
04     __asm__("push %%fs; \           // First save original value of fs (seg selector).
05           mov %%ax, %%fs; \         // Then use seg to set fs.
06           movb %%fs:%2, %%al; \     // Take 1 byte of seg:addr into the al register.
07           pop %%fs" \             // Restore the original contents of fs register.
08           : "=a" (__res) \         // Output register lists and constraints.
09           : "0" (seg), "m" (*(addr))) \ // Input register lists and constraints.
10     __res; })
```

This 10 lines of code defines an inline assembly language macro function. The most convenient way to use assembler statements is to place them in a macro. A compound statement enclosed in parentheses (a statement in curly braces): "({})" can be used as an expression, where the variable __res (line 10) on the last line is the output value of the expression, see The next section explains.

Because macro statements need to be defined on one line, these statements are concatenated using a backslash '\' here. This macro definition will be substituted into the program where the macro name is referenced. The first line defines the name of the macro, which is the macro function name get_seg_byte(seg,addr). Line 3 defines a register variable __res. This variable will be saved in a register for quick access and operation. If you want to specify a register (such as eax), then we can write this sentence as "register char __res asm ("ax");", where "asm" can also be written as "__asm__". The "__asm__" on line 4 indicates the beginning of the embedded assembly statement. The 4 statements from line 4 to line 7 are AT&T format assembler statements. In addition, in order to have a percent sign "%" in front of the register name in an assembly language program generated by gcc, two percent sign "%" must be written before embedding the assembly statement register name.

The eighth line is the output register. The meaning of this sentence is to place the value of the register represented by eax in the __res variable after the end of this code run, as the output value of this function, "a" in "=a" Called load code, "=" indicates that this is an output register and the value in it will be replaced by the output

value. Load code is a shorthand letter code for CPU registers, memory addresses, and some numeric values. Table 3-4 shows some of the register loading code and its specific meaning that we often use. Line 9 indicates that seg is placed in the eax register when this code starts to run, and "0" means that the same register as the one above is output. (*(addr)) represents a memory offset address value. In order to use this address value in the above assembler statement, the embedded assembler program specifies that the output and input registers are numbered sequentially, starting from the left and right top to bottom of the output register sequence starting with "%0", respectively, denoted as % 0, %1, ...%9. Therefore, the output register number is %0 (here only one output register), the first part of the input register ("0" (seg)) has the number %1, and the latter part has the number %2. The %2 above line 6 above represents (*(addr)) this memory offset.

Table 3-4 Common register load code description

Code	Description	Code	Description
a	Use register eax	m	Use memory address, any memory operand is allowed.
b	Use register ebx	o	Use memory address, and can add offset value
c	Use register ecx	I	Use constants 0-31
d	Use register edx	J	Use constants 0-63
S	Use register esi	K	Use constants 0-255
D	Use register edi	L	Use constants 0-65535
q	Use dynamically allocated byte addressable registers (eax、ebx、ecx 或 edx)	M	Use constants 0-3
r	Use any dynamically allocated register	N	Use 1 byte constant (0-255)
g	Any general register, memory or immediate integer operand is allowed (eax、ebx、ecx、edx or memory variable)	O	Use constants 0-31
A	Combine eax with edx (64-bit)	=	Output operands. The output value will replace the previous value
+	Indicates that the operand is readable and writable	&	Early-clobber operands. Indicates that the content will be modified before the operands are used

Now let's examine the function of code on lines 4-7. The first sentence puts the contents of the fs segment register on the stack; the second sentence assigns the segment value in eax to the fs segment register; the third sentence puts the byte specified by fs:(*(addr)) into the al register. . When the assembly statement is executed, the value of the output register eax will be put into __res as the return value of the macro function (block structure expression). It's simple, isn't it?

From the above analysis, we know that seg in the macro name represents a specified memory segment value, and addr represents a memory offset address amount. Until now, we should be very clear about the function of this program! The function of this macro function is to fetch one byte from the memory address of the specified segment and offset value. Then look at the next example.

```

01  asm("cld\n\t"
02      "rep\n\t"
03      "stol"
04      : /* No output register */

```

```
05      : "c"(count-1), "a"(fill_value), "D"(dest)
06      : "%ecx", "%edi");
```

The 1-3 lines are the usual assembler statements to clear the direction bit and repeatedly store the value. The characters "\n\t" in the first two lines are used to neatly set the gcc preprocessor output program list. The meaning of the characters is the same as in the C language. That is, the operation mode of gcc is to generate the assembler corresponding to the C program, and then call the assembler to compile it to generate the target code. If you want to look at the assembler corresponding to C when you write the program and debug the program, you need to get the pre The output of the assembler program that processes the program (this is commonly used when writing and debugging efficient code). In order to preprocess the assembler output in a neat format, you can use the two "\n\t" format symbols.

Line 4 shows that the inline assembler does not use the output register. The meaning of line 5 is: Load the count-1 value into the ecx register (the loading code is "c"), fill_value is loaded into eax, and dest is placed into edi. Why do we have to make the gcc compiler to load such register values without letting us do it ourselves? Because gcc can perform some optimization work while it is registering. For example, the fill_value value may already be in eax. If it is in a loop statement, gcc may keep eax in the entire loop operation, so you can use a movl statement in each loop.

The last line is to tell gcc that the values in these registers have changed. After gcc knows what you are doing with these registers, it can help with gcc's optimization. The following example does not allow you to specify which variable to use which register, but let gcc choose it for you.

```
01  asm("leal (%1, %1, 4), %0"
02      : "=r"(y)
03      : "0"(x));
```

The instruction "leal" is used to calculate the effective address, but it is used here for some simple calculations. The first assembler statement "leal (r1, r2, 4), r3" indicates $r1+r2*4 \Rightarrow r3$. This example can multiply x by 5 very quickly. Among them, "%0" and "%1" refer to the register that gcc automatically allocates. Here "%1" represents the register into which the input value x is to be placed, and "%0" represents the output value register. Be sure to add the equal sign before outputting the register code. If the code of the input register is 0 or is empty, then the same register as the corresponding output is used. So, if gcc specifies r as eax, then the meaning of the above assembly statement is:

```
"leal (eax, eax, 4), eax"
```

Note that when executing the code, if you do not want the assembly statement to be modified by GCC optimization, you need to add the keyword volatile after the asm symbol, as shown below. The difference between these two declarations lies in the aspect of program compatibility. It is recommended to use the latter way of declaration.

```
asm volatile (.....);
Or a more detailed explanation is:
__asm__ __volatile__ (.....);
```

The keyword volatile can also be placed before the function name to decorate the function to inform the gcc

compiler that the function will not return. This will allow gcc to produce better code. In addition, for functions that do not return, this keyword can also be used to prevent gcc from generating false warning messages. For example, the following statement in mm/memory.c shows that the functions `do_exit()` and `oom()` no longer return to the caller code:

```
31 volatile void do_exit(long code);
32
33 static inline volatile void oom(void)
34 {
35     printk("out of memory\n\r");
36     do_exit(SIGSEGV);
37 }
```

Here is a longer example. If you can read it, it means that inline assembly code is basically OK for you. This code is taken from the `include/string.h` file and is an implementation of the string comparison function for `strncmp()`. Similarly, the `"\n\t"` in each of these lines is set for the gcc preprocessor output list to look good.

```
//// String1 is compared with string2 in the first count characters.
// Paras: cs - String1, ct - String2, count - The number of characters to compare.
// %0 - eax(__res) return, %1 - edi(cs) String1 ptr, %2 - esi(ct)String2 ptr, %3 - ecx(count).
// Return: If string1 > string2, ret 1; string1 == string2, ret 0; string1 < string2, then ret -1.
extern inline int strncmp(const char * cs, const char * ct, int count)
{
register int __res ;                // __res is a register variable.
__asm__ ("cld\n"                  // Clear direction.
        "1:\tdecl %3\n\t"          // count--.
        "js 2f\n\t"               // If count<0, go forward to label 2.
        "lods b\n\t"              // Take string 2 character ds:[esi]=>al, and esi++.
        "scas b\n\t"              // Compare char in al and in string1 es:[edi] and edi++.
        "jne 3f\n\t"              // If they are not equal, go forward to label 3.
        "testb %%al, %%al\n\t"    // Is this character a NULL character?
        "jne 1b\n\t"              // No, go backward to label 1 and continue comparing.
        "2:\txorl %%eax, %%eax\n\t" // If it is a NULL char, eax is cleared (return value).
        "jmp 4f\n\t"              // Go forward to label 4 and end.
        "3:\tmovl $1, %%eax\n\t"   // eax is set to 1.
        "jl 4f\n\t"              // If the string2 chars < string1 chars, return 1 and end.
        "negl %%eax\n\t"          // Otherwise eax = -eax returns a negative value, ends.
        "4:"
        : "=a" (__res) : "D" (cs), "S" (ct), "c" (count) : "si", "di", "cx");
return __res;                    // Return the comparison result.
}
```

3.3.3 Combination statements in parentheses

The braces pair `{...}` is used to combine variable declarations and statements into a compound statement (combination statement) or a statement block so that these statements are semantically equivalent to a single statement. There is no need to use a semicolon after the closing brace of a compound statement. Combination statements in parentheses, ie statements of the form `"({...})"`, can be used as an expression in GNU C. This allows loops, switch statements, and local variables to be used in expressions, so this form of statement is often called a

statement expression. The statement expression has the following example form:

```
{ int y = foo(); int z;
  if (y > 0) z = y;
  else z = -y;
  3 + z; }
```

The last statement in a compound statement must be an expression followed by a semicolon. The value of this expression ("3 + z") is used as the value enclosed by the entire parenthesis. If the last statement is not an expression, the entire statement expression has a void type and therefore has no value. In addition, any local variables declared by statements in such an expression will expire after the entire block statement ends. This sample statement can be used like the following form of assignment statement:

```
res = x + ({omit...}) + b;
```

Of course, people usually don't write statements like the above, which are usually used to define macros. For example, the macro definition for reading CMOS clock information in the kernel source code `init/main.c` program:

```
69 #define CMOS_READ(addr) ({ \
70  outb_p(0x80|addr, 0x70); \           // First, output the addr to the I/O port 0x70.
71  inb_p(0x71); \                       // Then read the value from port 0x71 as the return value.
72 })
```

Look again at the macro definition of the read I/O port in the `include/asm/io.h` header file, where the value of the last variable `_v` is the return value of `inb()`.

```
05 #define inb(port) ({ \
06  unsigned char _v; \
07  __asm__ volatile ("inb %dx,%al":"=a" (_v):"d" (port)); \
08  _v; \
09  })
```

3.3.4 Register variables

Another extension of GNU to the C language allows us to put some variable values into the CPU registers, the so-called register variables. In this way, the CPU does not need to spend a long time to access the memory for value. There are two types of register variables: global register variables and local register variables. Global register variables hold registers dedicated to several global variables throughout the program's operation. In contrast, local register variables do not retain the specified registers, and special registers are used only as input or output operands in the inline asm assembly statement. The gcc compiler's data flow analysis capability is inherently capable of determining when a specified register has a value in use and when it can dispatch other fields. When the gcc data flow analysis function is considered to be stored when a local register variable value is useless, it may be deleted, and references to local register variables may also be deleted, moved, or simplified. Therefore, if you do not want gcc to make these optimization changes, it is best to add volatile keywords in the asm statement.

If you want to write the output of the assembler instruction directly to the specified register in an inline

assembler statement, it is convenient to use local register variables at this time. Since the Linux kernel usually only uses local register variables, we will only discuss the use of local register variables here. In GNU C programs we can define a local register variable in a function like this:

```
register int res __asm__("ax");
```

Here `ax` is the register that the variable `res` wants to use. Defining such a register variable does not specifically reserve this register for no other purpose. During program compilation, when the `gcc` data flow control determines that the value of a variable is no longer in use, the register may be dispatched for other purposes, and references to it may be deleted, moved, or simplified. In addition, `gcc` does not guarantee that the compiled code will keep the variable in the specified register. Therefore, it is better not to refer to this register explicitly in the portion of the instruction that is embedded in the assembly and it is assumed that the register must refer to this variable value. However, using this variable as an operand to `asm` ensures that the specified register is used as the operand.

3.3.5 Inline function

In a program, by declaring a function as an inline function, you can have `gcc` integrate the code of the function into the code that calls the function. This processing can remove the overhead of the entry/exit time when the function call is invoked, thus definitely speeding up execution. Therefore, the main purpose of declaring a function as an inline function is to be able to execute the function body as quickly as possible. In addition, if there is a constant value in an inline function, `gcc` may use it to perform some simplification during compilation, so not all inline function code will be embedded. The inline function method has no obvious effect on the length of the program code. Programs compiled using inline functions may generate longer or shorter target code, depending on the circumstances.

The operation of embedding an inline function in the caller's code is an optimization operation, so code embedding processing is performed only when an optimized compilation is performed. If the optimization option `"-O"` is not used during compilation, the code of the inline function is not actually embedded in the caller's code, but is only handled as an ordinary function call. The way to declare a function as an inline function is to use the keyword `"inline"` in the function declaration, such as the following function in the kernel file `fs/inode.c`:

```
01 inline int inc(int *a)
02 {
03     (*a)++;
04 }
```

The use of some of the statements in a function may prevent the replacement of an inline function from working properly, or may not be suitable for replacement operations. For example, variable parameters, memory allocation functions `malloc()`, variable-length data type variables, non-local `goto` statements, and recursive functions are used. Compiler can use the option `-Winline` to make `gcc` give warning information for functions marked as inline but cannot be replaced, and why they cannot be replaced.

When using both the `inline` keyword and the `static` keyword in a function definition, ie the definition of an inline function in the file `fs/inode.c` below, then all calls to the inline function are replaced if they are replaced. In the caller code, and the program does not refer to the address of the inline function, the assembly code of the inline function itself will not be referenced. In this case, unless we use the option `-fkeep-inline-functions` during compilation, `gcc` will no longer generate actual assembly code for the inline function itself. For some reason, some

calls to inline functions cannot be integrated into functions. In particular, the calling statement before the definition of the inline function is not replaced by the integration and cannot be a function defined by recursion. If there is a call that cannot be replaced by an integration, the inline function is compiled into assembly code as usual. Of course, if the program has a statement that references the address of an inline function, the inline function is also compiled into assembly code as usual. Because references to inline function addresses cannot be replaced.

```
20 static inline void wait_on_inode(struct m_inode * inode)
21 {
22     cli();
23     while (inode->i_lock)
24         sleep_on(&inode->i_wait);
25     sti();
26 }
```

Please note that inline function functions have been included in ISO standard C99, but the inline functions defined by this standard are quite different from those defined by gcc. The semantic definition of the inline function of the ISO standard C99 is equivalent to the definition of the combination keyword inline and static, which means “eliminating” the keyword static. If you need to use the C99 standard semantics in your program, you need to use the compile option `-std=gnu99`. However, in order to be compatible, it is still best to use inline and static combinations in this case. After that gcc will eventually use the definition of C99 by default. If you want to still use the semantics defined here, you need to use the option `-std=gnu89` to specify.

If the definition of an inline function does not use the keyword static, then gcc will assume that there is also a call to this function in other program files. Because a global symbol can only be defined once, the function can no longer be defined in other source files. Therefore, calls to inline functions cannot be replaced by integration here. Therefore, a non-static inline function is always compiled with its own assembly code. In this regard, the ISO standard C99 definition of an inline function that does not use the static keyword is equivalent to using the static keyword definition here.

If both inline and extern keywords are specified when defining a function, the function definition is only used for inline integration, and the function's own assembly code is not generated separately in any case, even if the function is explicitly referenced. The address will not be generated either. Such an address becomes an external reference, just as if you just declared a function without defining a function.

The combination of inline and extern is almost identical to a macro definition. Using this combination method is to put a function definition with a combination keyword in the .h header file, and put another definition of the same function without a keyword in a library file. The definition in the header file at this time causes most of the calls to the function to be embedded by substitution. If there is a call to the function that has not been replaced, then a copy of the program file or library is used (referenced). The file `include/string.h`, `lib/strings.c` in the Linux 0.1x kernel source code is an example of this use. For example, the following function is defined in `string.h`:

```
// Copy the string (src) to another string (dest) until it encounters a NULL character.
// Paras: dest - dest string ptr, src - source string ptr. %0 - esi(src), %1 - edi(dest).
27 extern inline char * strcpy(char * dest, const char *src)
28 {
29     __asm__ ( "cld\n"                // Clear direction.
30             "1:|t lodsb|n|t"        // Load DS: [esi] 1 byte => al and update esi.
```

```
31      "stosb\n\t"           // Store byte al=>ES:[edi] and update edi.
32      "testb %%al, %%al\n\t" // Just stored byte al is 0?
33      "jne 1b"              // If not, go back to label 1 or end.
34      :: "S" (src), "D" (dest): "si", "di", "ax");
35 return dest;              // Returns the destination string pointer.
36 }
```

In the kernel library directory, the lib/strings.c file defines the keywords inline and extern as empty, as shown below. Therefore, in fact, the kernel library contains a copy of all such functions in the string.h file, which in turn redefines these functions once and "eliminates" the effect of the two keywords.

```
11 #define extern              // Defined as empty.
12 #define inline              // Defined as empty.
13 #define LIBRARY
14 #include <string.h>
15
```

The above-defined strcpy() function in the library function now becomes the following:

```
27 char * strcpy(char * dest, const char *src) // Removed keywords inline and extern.
28 {
29     __asm__ ( "cld\n"           // Clear direction.
30             "l: |t lodsb\n\t"    // Load DS: [esi] 1 byte => al and update esi.
31             "stosb\n\t"         // Store byte al=>ES:[edi] and update edi.
32             "testb %%al, %%al\n\t" // Just stored byte al is 0?
33             "jne 1b"           // If not, go back to label 1 or end.
34             :: "S" (src), "D" (dest): "si", "di", "ax");
35 return dest;                  // Returns the destination string pointer.
36 }
```

3.4 Interworking between C and Assembly language

In order to improve the efficiency of code execution, some parts of the kernel source code directly use the assembly language. This will involve the invocation problem between two language programs. This section first describes the invocation mechanism of C language functions, and then uses an example to illustrate the calling method between the two functions.

3.4.1 C function call mechanism

After the Linux kernel program boot/head.s performs basic initialization operations, it will jump to execute the init/main.c program. So how does the head.s program transfer its execution control to the init/main.c program? That is how the assembler calls to execute C language programs? Here we first describe the C function call mechanism, control transfer mode, and then explain the head.s program jumps to the C program.

Function call operations include bidirectional data transfer and execution control transfer from one piece of code to another. Data passing is done through function parameters and return values. In addition, we also need to allocate storage space for the function's local variables when entering the function, and reclaim this space when exiting the function. The Intel 80x86 CPU provides simple instructions for control transfer, while the transfer of

data and the allocation and recovery of local variable storage space are achieved through stack operations.

3.4.1.1 Stack frame structure and control transfer method

Most program implementations use the stack to support function call operations. The stack is used to transfer function parameters, store return information, temporarily save the original values of the registers for recovery and to store local data. The portion of the stack used by a single function call operation is called the stack frame structure. Its general structure is shown in Figure 3-4. Both ends of the stack frame structure are specified by two pointers. The register `ebp` is usually used as a frame pointer, and `esp` is used as a stack pointer. During the execution of the function, the stack pointer `esp` moves with the data being pushed onto the stack. Therefore, most data accesses in the function are based on the frame pointer `ebp`.

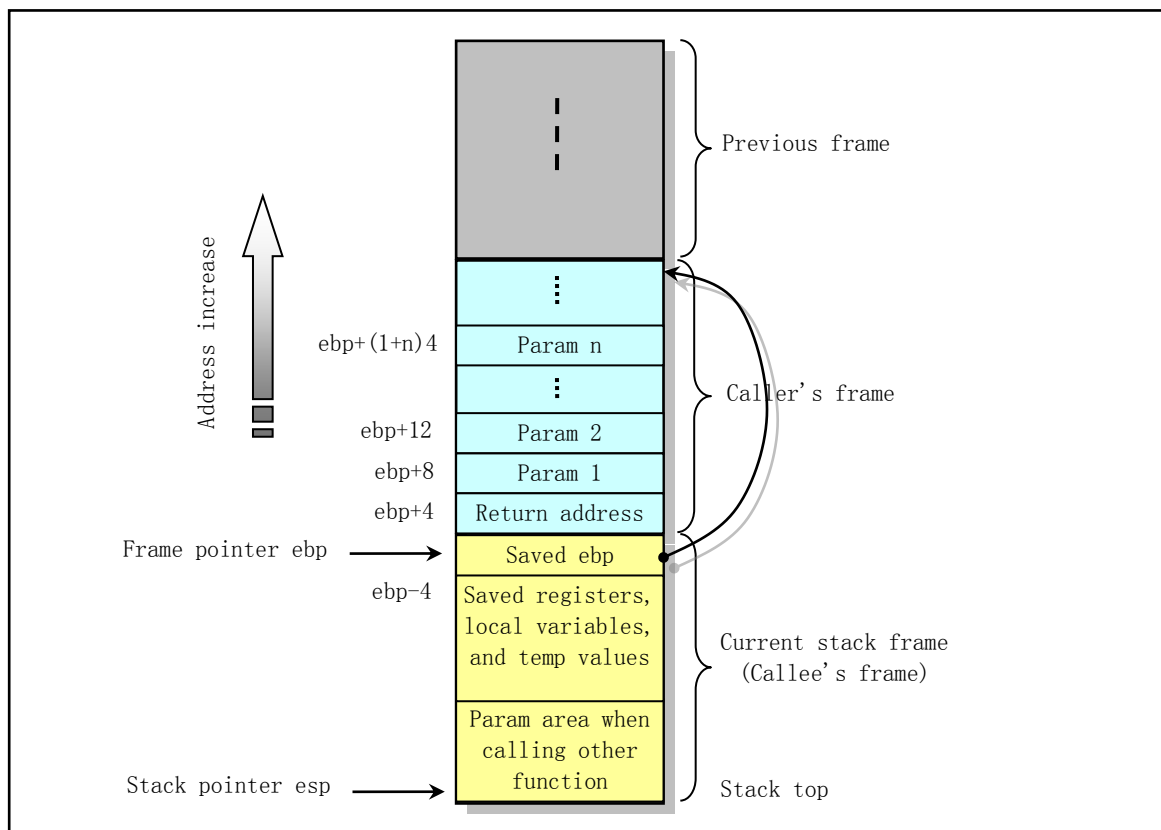


Figure 3-4 Frame structure in the stack

For the case where function A calls function B, the parameters passed to B are contained in A's stack frame. When A calls B, the return address of Function A (the address of the instruction that continues execution after the call is returned) is pushed onto the stack. This position on the stack also explicitly indicates the end of the A stack frame. The stack frame of B starts from the subsequent stack section, where the frame pointer (`ebp`) is stored. It is then used to store any saved register values and temporary values of the function.

The B function also uses the stack to hold local variable values that cannot be placed in registers. For example, because the normal CPU has a limited number of registers and cannot store all the local data of the function, or some local variables are arrays or structures, they must be accessed using an array or a structure reference. There is also the C language address operator `&` applied to a local variable, we need to generate an address for the variable, which allocates a space for the address pointer of the variable. Finally, the B function will use the stack to save the parameters that call any other function.

The stack is expanded toward the low (small) address, while the `esp` points to the element at the top of the current stack. By using the `push` and `pop` instructions we can push data onto the stack or pop it off the stack. For storage space that does not specify initial data, we can do this by decrementing the stack pointer by an appropriate value. Similarly, we can reclaim the allocated space on the stack by increasing the stack pointer value.

The instructions `CALL` and `RET` are used to handle function invocation and return operations. The effect of the instruction `CALL` is to push the return address onto the stack and jump to the beginning of the called function. The return address is the address of an instruction immediately following the instruction `CALL` in the program. So when the called function returns, it will continue from that position. The return instruction `RET` is used to pop up the address at the top of the stack and jump to that address. Before using this instruction, the contents of the stack should be processed correctly so that the current stack pointer is the same as the one returned by the previous `CALL` instruction. In addition, if the return value is an integer or a pointer, the register `eax` will be used by default to pass the return value.

Although only one function is executed at a time, we still need to make sure that when a function (caller) calls another function (the callee), the callee will not modify or overwrite the register contents that the caller will use in the future. Therefore, the Intel CPU adopts the uniform usage of registers that all functions must comply with. This convention indicates that the contents of registers `eax`, `edx`, and `ecx` must be held by the caller themselves. When function B is called by A, function B can use them arbitrarily without saving the contents of these registers without destroying any data needed by function A. In addition, the contents of registers `ebx`, `esi`, and `edi` must be protected by callee B. When the callee needs to use any of these registers, it must first save its contents on the stack and restore the contents of these registers on exit. Because caller A (or some higher-level function) is not responsible for saving these register contents, it may need to use the original value in future operations. There are also registers `ebp` and `esp` that must follow the second convention usage.

3.4.1.2 Function call example

As an example, let's observe the processing of the function call in the following C program `exch.c`. The program exchanges the values in the two variables and returns their difference.

```
1 void swap(int * a, int *b)
2 {
3     int c;
4     c = *a; *a = *b; *b = c;
5 }
6
7 int main()
8 {
9     int a, b;
10    a = 16; b = 32;
11    swap(&a, &b);
12    return (a - b);
13 }
```

The function `swap()` is used to exchange the values of two variables. The main program in the C program, `main()`, is also a function (described below). It returns the swapped result after calling `swap()`. The stack frame structure of these two functions is shown in Figure 35. As you can see, the function `swap()` gets its parameters from the caller's (`main()`) stack frame. The position information in the figure is relative to the frame pointer in the register `ebp`. The number to the left of the stack frame indicates the address offset value relative to the frame pointer. In debuggers such as `gdb`, these values are represented in 2's complement. For example, '-4' is represented as '0xFFFFFFFFC',

and '-12' is represented as '0xFFFFFFFF4'.

The stack frame structure of the caller `main()` includes the storage space for the local variables `a` and `b`, which are located at -4 and -8 offsets with respect to the frame pointer. Since we need to generate addresses for these two local variables, they must be stored on the stack and not simply stored in registers.

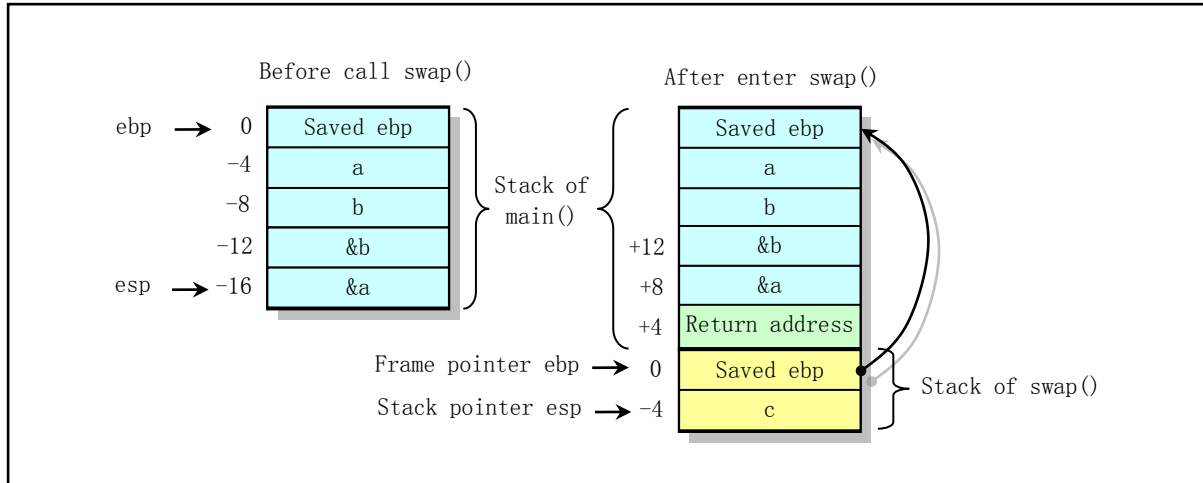


Figure 3-5 Stack frame structure when calling function `main` and `swap`

Use the command `"gcc -Wall -S -oexch.s exch.c"` to generate the assembler `exch.s` code for this C language program, as shown below (remove several lines of directives that are not relevant to the discussion).

```

1 .text
2 _swap:
3     pushl %ebp                # Save original ebp, set current function's frame pointer.
4     movl %esp, %ebp
5     subl $4, %esp            # Allocates space within the stack for the local variable c.
6     movl 8(%ebp), %eax        # Get 1st argument, which is a pointer to an integer value.
7     movl (%eax), %ecx         # Store value pointed by the pointer into variable c.
8     movl %ecx, -4(%ebp)
9     movl 8(%ebp), %eax        # Take 1st parameter again, and then take 2nd parameter.
10    movl 12(%ebp), %edx
11    movl (%edx), %ecx         # Put the content of the 2nd parameter into the place
12    movl %ecx, (%eax)         # pointed by the 1st parameter.
13    movl 12(%ebp), %eax        # Take the second parameter again, then place the content
14    movl -4(%ebp), %ecx        # of variable c at the position pointed by this pointer.
15    movl %ecx, (%eax)
16    leave                     # Restore the original ebp and esp.
17    ret
18 _main:
19    pushl %ebp                # Save original ebp, set current function's frame pointer.
20    movl %esp, %ebp
21    subl $8, %esp             # Allocates space in stack for local variables a and b.
22    movl $16, -4(%ebp)        # Assign initial values to variables (a=16, b=32).
23    movl $32, -8(%ebp)
24    leal -8(%ebp), %eax        # To call the swap(), push the address of variable b onto
25    pushl %eax                # stack. That is, push the second parameter first.
26    leal -4(%ebp), %eax        # Then push address of variable a as the first parameter.
27    pushl %eax

```

```

28      call _swap                # Call the function swap().
29      movl -4(%ebp),%eax        # Take the value of a - b.
30      subl -8(%ebp),%eax
31      leave                    # Restore the original ebp and esp.
32      ret

```

These two functions can be divided into three parts independently: "set" to initialize the stack frame structure; "body" to perform the actual calculation of the function; "end" to restore the stack state and return from the function. For the swap() function, the setting part code is 3--5 lines. The first two lines are used to set the caller's frame pointer and set the function's stack frame pointer. Line 5 allocates space for the local variable c by moving the stack pointer esp down by 4 bytes. Line 6-15 is the main part of the swap function. Lines 6 - 8 are used to retrieve the caller's first parameter, &a, and use this parameter as an address to fetch the contents of the memory into the ecx register and save it to the space allocated for the local variable (-4 (%ebp)) . Lines 9-12 are used to fetch the second parameter, &b, and take the parameter value as the address and take its contents to the address specified by the first parameter. Lines 13-15 store the value stored in the temporary local variable c at the address specified by the second parameter. The last 16-17 lines are the end of the function. The leave instruction is used to process the contents of the stack in preparation for return. Its role is equivalent to the following two instructions:

```

      movl %ebp,%esp            # Restores original esp (to beginning of stack frame).
      popl %ebp                # Restores original ebp (usually the caller's frame ptr).

```

This two lines of code restore the original values of the registers esp and ebp when entering the swap() function and executes the return instruction ret.

Lines 19-21 are the set part of the main() function. After saving and resetting the frame pointer, main() allocates space for the local variables a and b on the stack. Lines 22-23 assign values to these two local variables. You can see from lines 24-28 how main() calls the swap() function. The first step is to use the leal instruction (fetching an effective address) to get the addresses of the variables b and a and push them onto the stack, and then call the swap() function. The order in which the variable addresses are pushed onto the stack is exactly the opposite of the order of the parameters declared by the function. That is, the last parameter of the function is pushed onto the stack first, and the first parameter of the function is pushed onto the stack before the call to the function instruction call. Lines 29--30 subtract the two already exchanged numbers and place them in the eax register as the return value.

From the above analysis we can see that when C calls a function, it temporarily stores the value of the transferred function parameter on the stack. That is, C language is a value-based language. There is no direct method to modify the caller variable in the called function. value. Therefore, in order to achieve the purpose of modification, you need to pass a pointer to the variable (ie, the address of the variable) to the function.

3.4.1.3 Main() is also a function

The above assembler code is compiled using gcc 1.40. It can be seen that there are a few lines of extra code. It can be seen that the gcc compiler at that time could not produce the most efficient code. This is one of the reasons why some key code needs to be compiled directly in assembly language. In addition, the main program of the C program mentioned above is also a function. This is because it will be called as a function of the crt0.s assembler program when the link is compiled. crt0.s is a stub program. The name "crt" is an abbreviation for "C run-time". The program's target file will be linked at the beginning of each user's execution program, mainly used to set some initialization global variables. The crt0.s assembler program in Linux 0.12 is shown below. The global variable _environ is created and initialized for use by other modules in the program.

```
1 .text
2 .globl _environ                # Declare the global variable _environ. (correspond to
3                                # the environ variable in C program).
4 __entry:                      # Code entry label.
5     movl 8(%esp), %eax         # Get environment variable pointer envp, save in _environ.
6     movl %eax, _environ       # envp is set by execve() when executable file is loaded.
7     call _main                # Call main program. Its return status is in eax register.
8     pushl %eax                # Push return value as an argument to exit() and call it.
9 1:    call _exit
10     jmp 1b                   # Control should not arrive here.
11 .data
12 _environ:                    # Define variable _environ and assign it a long word space.
13     .long 0
```

When gcc is used to compile and link the executable file, gcc will automatically link the code of crt0.s as the first module in the executable program. Use the show details option '-v' at compile time to clearly see the linking process:

```
[/usr/root]# gcc -v -o exch exch.s
gcc version 1.40
/usr/local/lib/gcc-as -o exch.o exch.s
/usr/local/lib/gcc-ld -o exch /usr/local/lib/crt0.o exch.o /usr/local/lib/gnulib -lc
/usr/local/lib/gnulib
[/usr/root]#
```

So in the normal compilation process we don't need to specify the stub module crt0.o, but if we want to use the ld(gld) to generate the executable exch from the exch.o module manually from the assembly program given above, then we need to The crt0.o module is specified on the command line, and the link order should be "crt0.o, all program modules, library files."

In order to use ELF format object files and create a shared library module file, the current gcc compiler (2.x) has extended this crt0 into several modules: crt1.o, crt1.o, crtbegin.o, crtend.o, and Crtn.o. The link order of these modules is "crt1.o, crt1.o, crtbegin.o (crtbeginS.o), all program modules, crtend.o (crtendS.o), crtn.o, library module files". The gcc configuration file specfile specifies this link order. Where crt1.o, crt1.o, and crtn.o are provided by the C library, which is the C program's "boot" module; crtbegin.o and crenrend.o are the C++ language startup modules provided by the compiler gcc; and crt1.o It is similar to the effect of crt0.o, and is mainly used to do some initialization work before calling main(). The global symbol _start is defined in this module.

crtbegin.o and crenrend.o are mainly used in C++ languages to implement global constructors and destructor functions in the .ctors and .dtors sections. The roles of crtbeginS.o and crtendS.o are similar to those of the first two, but they are used to create shared modules. crt1.o is used to execute the initialization function init() in the .init section. The .init section contains the initialization code for the process, ie when the program starts executing, the system executes the code in .init before calling main(). Crtn.o is used to execute the process in the .fini area to terminate the processing function fini(), that is, when the program exits normally (main() returns), the system will arrange to execute the code in .fini.

In the kernel, lines 136--140 in the `boot/head.s` program are used to make preparations for jumping to the `main()` function in `init/main.c`. The instruction on line 139 pushes the return address on the stack, while line 140 presses the address of the `main()` function code. When `head.s` finally executes the `ret` instruction on line 218, the address of `main()` is popped up, and control is transferred to the `init/main.c` program.

3.4.2 Call C function in assembler code

The method of calling a C language function from an assembly code is actually given above. In the above assembler code for the C language example, we can see how the assembler statement calls the `swap()` function. Now we make a summary of the calling method.

When calling a C function in assembly code, you first need to push the function parameters into the stack in reverse order. That is, the last (rightmost) parameter of the function is pushed on the stack first, and the leftmost first parameter is pushed before the last instruction is called. See Figure 3-6. Then execute the `CALL` instruction to execute the called function. After the calling function returns, the program needs to clear the function parameters that were previously pushed onto the stack.

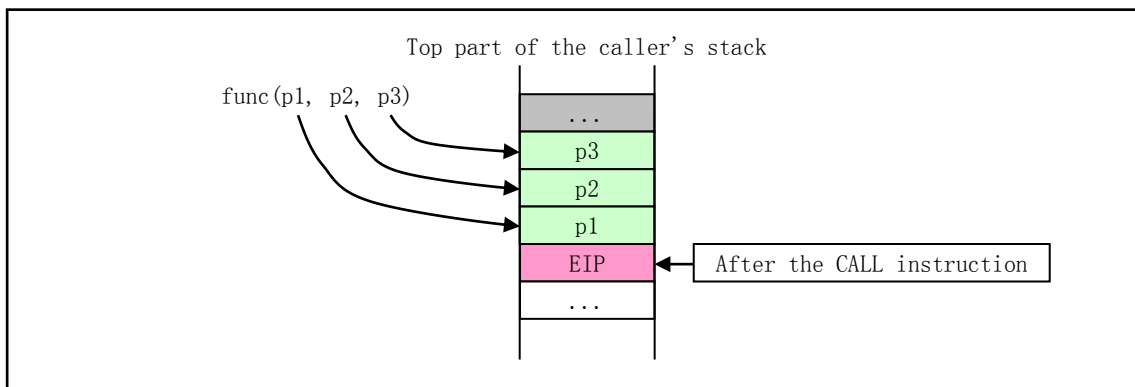


Figure 3-6 The parameters pushed into the stack when the function is called

When the `CALL` instruction is executed, the CPU pushes the address of the next instruction of the `CALL` instruction onto the stack (see EIP in the figure). If the call also involves code privilege level changes, the CPU will also perform a stack switch and push the current stack pointer, segment descriptor, and call parameters into the new stack. Since the Linux kernel uses only interrupt gates and trapdoors to handle privilege level changes during the call, and does not use the `CALL` instruction to handle privilege level changes, the use of the `CALL` instruction at the time of privilege level change is not described here. .

Calling C functions in assembly is relatively "free." As long as it is in the proper place in the stack, it can be used as a parameter for C functions. Here is still taking the function call with 3 parameters in Figure 3-6 as an example. If we do not call it directly by pushing the argument with `func()`, the `func()` function will still store the EIP position. The rest of the stack is used as its own parameter. If we explicitly press the first and second parameters for `func()` call, then the third parameter `p3` of the `func()` function will directly use the contents of the stack before `p2`. There are several places in this Linux 0.1x kernel code. For example, the `copy_process()` function (line 68 in `kernel/fork.c`) is called on line 231 in the `kernel/sys_call.s` assembly program. Although only five parameters are pushed onto the stack in the assembly function `_sys_fork`, `copy_process()` has a total of up to 17 parameters, as shown below:

```
// kernel/sys_call.s partial program: _sys_fork
```

```

226      push %gs
227      pushl %esi
228      pushl %edi
229      pushl %ebp
230      pushl %eax
231      call _copy_process      # Call the C function copy_process() (kernel/fork.c, 68).
232      addl $20,%esp          # Discard all the pushed content here.
233 1:      ret

```

```

// kernel/fork.c partial program.
68 int copy_process(int nr, long ebp, long edi, long esi, long gs, long none,
69                  long ebx, long ecx, long edx, long orig_eax,
70                  long fs, long es, long ds,
71                  long eip, long cs, long eflags, long esp, long ss)

```

We know that the later the parameter is pushed onto the stack, the closer it is to the left of the C function parameter. Therefore, the five register values that were actually pushed before the `copy_process()` was called are the five leftmost parameters of the `copy_process()` function. In order, they correspond to the values of the `eax(nr)`, `ebp`, `edi`, `esi`, and register `gs` that are stacked. The rest of the following parameters actually correspond directly to what is already on the stack. These contents are the values of various registers that are gradually added to the stack when the system call interrupt processing process is started until the system call process is called.

The parameter `none` is the return address of the next instruction when `_sys_fork` is called from the address jump table on line 99 of the `sys_call.s` program. The address jump table `sys_call_table[]` is defined in line 93 of the header file `include/linux/sys.h`. The following parameters are the registers `ebx`, `ecx`, `edx`, the original `eax`, and the segment registers `fs`, `es`, and `ds` that were pushed onto the stack at lines 85-91 just after entering `system_call`. The last five parameters are the CPU execution interrupt instruction push return address `eip` and `cs`, flag register `eflags`, user stack address `esp` and `ss`. Because the system call involves a change in program privilege level, the CPU pushes the flag register value and the user stack address onto the stack. After calling C function `copy_process()` returns, `_sys_fork` discards only 5 parameters pressed by itself, and the rest of the stack is also saved. Other functions that use the above usage include `do_signal()` in `kernel/signal.c`, `do_execve()` in `fs/exec.c`, etc. Please analyze it yourself.

In addition, we say that the assembly calls the C function is relatively free because we can use the `JMP` instruction to achieve the same purpose of calling the function without the `CALL` instruction. The method is to put the address of the instruction to be executed next into the stack manually after the parameter is pushed into the stack, and then directly use the `JMP` instruction to jump to the start address of the called function to execute the function. Afterwards, when the execution of the function is completed, the `RET` instruction will be executed to pop up the address of the next instruction we push manually into the stack, as the address returned by the function. There are also many ways to call this function in the Linux kernel, such as the case where the 62nd line of the `kernel/asm.s` program calls the `do_int3()` function in `traps.c`.

3.4.3 Call assembly function in C program

Calling an assembly function from a C program is the same as calling an C function in an assembler, but it is not often used in Linux kernel programs. The focus of the calling method is still on the determination of the location of function parameters in the stack. Of course, if the calling assembly language program is relatively short, it can be directly implemented in the C program using the inline assembly statement described above. Below we use an example to illustrate how to program this kind of program. The assembler `calle.s` containing two

functions is shown below.

```
/*
  This assembly language program uses the system call sys_write() to implement the display
  function int mywrite(int fd, char * buf, int count).
  The function int myadd(int a, int b, int * res) is used to perform the a+b = res operation.
  If the function returns 0, it means overflow.
  Note: If you compile under the current Linux system (such as RedHat 9), remove the
  underscore '_' before the function name.*/
SYSWRITE = 4                                # Sys_write() system call number.
.globl _mywrite, _myadd
.text
_mywrite:
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %ebx
    movl     8(%ebp), %ebx                  # Take the first argument of the caller: file descriptor fd.
    movl     12(%ebp), %ecx                # The second parameter: buffer pointer.
    movl     16(%ebp), %edx                # The third parameter: display character number.
    movl     $SYSWRITE, %eax               # Put system call number 4 in %eax.
    int      $0x80                         # Execute the system call.
    popl     %ebx
    movl     %ebp, %esp
    popl     %ebp
    ret
_myadd:
    pushl    %ebp
    movl     %esp, %ebp
    movl     8(%ebp), %eax                 # Get the first parameter a.
    movl     12(%ebp), %edx               # Get the second parameter b.
    xorl     %ecx, %ecx                   # If %ecx is 0, the calculation overflows.
    addl     %eax, %edx                   # Perform additions.
    jo       1f                           # Jump if it overflows.
    movl     16(%ebp), %eax               # Take the third parameter pointer.
    movl     %edx, (%eax)                 # Put the result in the position of the pointer.
    incl     %ecx                         # No overflow occurred, so set no overflow return value.
1:    movl     %ecx, %eax                  # %eax is the function return value.
    movl     %ebp, %esp
    popl     %ebp
    ret
```

The first function `mywrite()` in the assembly file uses the system interrupt `0x80` to call the system call `sys_write(int fd, char *buf, int count)` to display the information on the screen. The corresponding system call function number is 4 (see `include/unistd.h`). The three parameters are the file descriptor, the display buffer pointer, and the number of display characters. Before executing `int 0x80`, the caller function number (4) needs to be placed in the register `%eax`, and the registers `%ebx`, `%ecx`, and `%edx` should be stored as `fd`, `buf`, and `count`, respectively, according to the calling rules. The function argument `mywrite()` uses exactly the same number of parameters and uses as `sys_write()`.

The second function `myadd(int a, int b, int *res)` performs an addition operation. The parameter `res` is the result of the operation. The function return value is used to determine if an overflow has occurred. If the return value is 0, the calculation has overflowed and the result is not available. Otherwise the result of the calculation

will be returned to the caller via the parameter res.

Note that if you compile callee.s under the current Linux system (eg RedHat 9), remove the underscore '_' before the function name. The C program caller.c that calls these two functions is shown below.

```
/*
    Call assembly function mywrite(fd, buf, count) to display information;
    Call myadd (a, b, result) to perform addition. If myadd() returns 0, it indicates
    overflow. First, the start calculation information is displayed, and then the operation
    result is displayed.
*/
01 int main()
02 {
03     char buf[1024];
04     int a, b, res;
05     char * mystr = "Calculating...\n";
06     char * emsg = "Error in adding\n";
07
08     a = 5; b = 10;
09     mywrite(1, mystr, strlen(mystr));
10     if (myadd(a, b, &res)){
11         sprintf(buf, "The result is %d\n", res);
12         mywrite(1, buf, strlen(buf));
13     } else {
14         mywrite(1, emsg, strlen(emsg));
15     }
16     return 0;
17 }
```

The program first uses the assembly function mywrite() to display the information "Calculating..." on the screen, and then calls the addition calculation function myadd() to operate on the two numbers a and b, and on the third parameter res Returns the result of the calculation. Finally, use the mywrite() function to display the formatted result information string on the screen. If the function myadd() returns 0, it means that the overflow function has overflowed and the result of the calculation is invalid. The compilation and running results of these two files are shown below:

```
[/usr/root]# as -o callee.o callee.s
[/usr/root]# gcc -o caller caller.c callee.o
[/usr/root]# ./caller
Calculating...
The result is 15
[/usr/root]#
```

3.5 Linux 0.12 Object file format

To generate the kernel code, Linux 0.12 uses two compilers. The first is the assembler as86 and the corresponding linker (or linker) ld86. They are used exclusively for compiling and linking the 16-bit kernel boot sector program bootsect.s and the setup program setup.s running in real-address mode. The second is the GNU assembler as(gas) and the C compiler gcc and the corresponding linker gld. The compiler is used to generate the

corresponding binary code and data object file for the source program file. The linker is used to combine all related object files to form a target file that can be loaded by the kernel, ie, an executable file.

This section begins with a brief description of the compiler-generated object file structure, and then describes how the linker combines the object file modules that need to be linked together to generate a binary executable image file or a large module file. Finally, it explains the generation principle and process of the Linux 0.12 kernel binary code file Image. This gives information on the a.out object file format supported by the Linux 0.12 kernel. As86 and ld86 generate the MINIX-specific object file format, which we will present in the chapter on kernel creation tools that deal with this format. Because the MINIX object file structure is similar to the a.out object file format, it will not be described here. The basic working principle of the object file and linker program can be found in the book “Linkers & Loaders” by John R. Levine.

For convenience of description, the object file generated by the compiler is called an object module file (abbreviated as a module file), and the executable object file generated by the link program is called an executable file. And all of them are collectively referred to as object files.

3.5.1 Object file format

In the Linux 0.12 system, the UNIX module's traditional a.out format is used by both the GNU gcc or gas compiler output object module file and the linker generated executable file. This is an object file format called Assembly & Linker Editor Output. For a system with a memory paging mechanism, this is a simple and effective object file format. The a.out format file consists of a file header and subsequent code sections (also called text sections), initialized data sections (also called data sections), relocation information section, symbol tables and symbol names String composition, as shown in Figure 3-7. The code section and the data section are usually also referred to as a text segment (code segment) and a data segment, respectively.

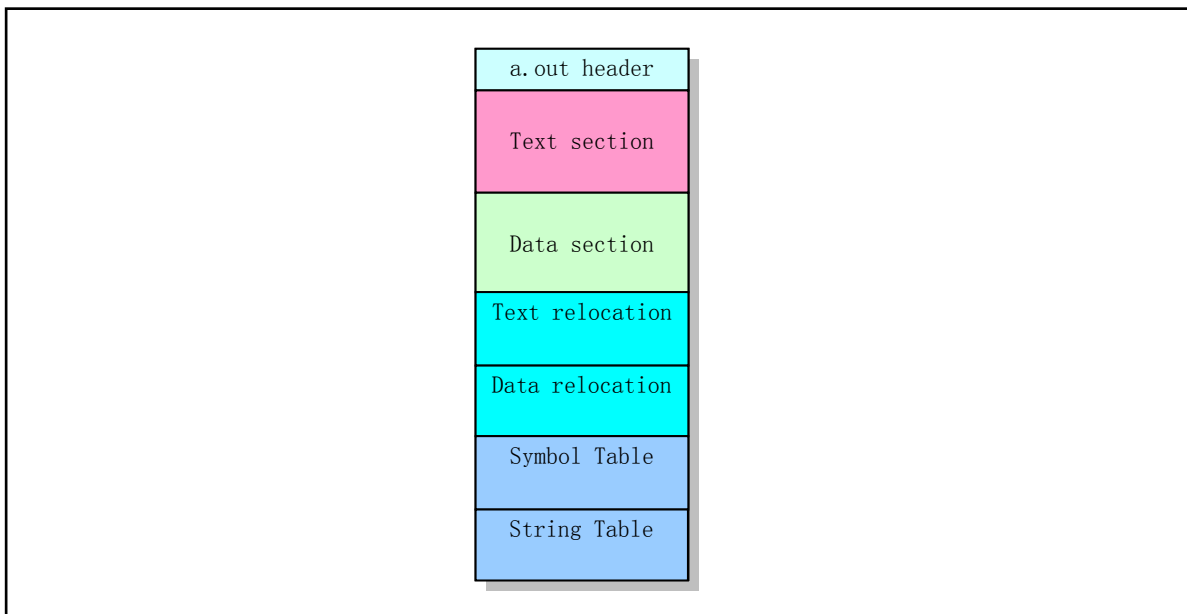


Figure 3-7 a.out format object file

The basic definitions and uses of the 7 sections of the a.out format are:

- Exec header. Execute file header section. This section contains some parameters (exec structure), which is the overall structure information about the target file. For example, the length of the code and data area, the length of the uninitialized data area, the corresponding source file name, and the target file creation time. The

kernel uses these parameters to load execution files into memory and execute them, and the linker (ld) uses these parameters to combine some of the module files into an executable file. This is the only necessary part of the target document.

- Text segment. The binary instruction code and data information generated by the compiler or assembler contains the instruction code and related data that are loaded into memory when the program is executed. Can be loaded as read-only.
- Data segment. Binary instruction code and data information generated by a compiler or assembler. This section contains data that has already been initialized and is always loaded into readable and writable memory.
- Text relocations. This section contains record data for use by the linker. Used to locate a pointer or address in a code segment when combining object module files. When the linker needs to change the address of the target code, it needs to be corrected and maintained.
- Data relocations. Similar to the role of the code relocation section, but used for relocation of pointers in the data segment.
- Symbol table. This section also contains record data for use by the linker. These record data hold the global symbols defined in the module file and the symbols that need to be input from other module files, or the symbols defined by the linker, used to cross named variables and functions (symbols) between module files. References.
- String table. This section contains the string corresponding to the symbol name. Used to debug program debugging target code, regardless of the linking process. This information can include source code and line numbers, local symbols, and data structure description information.

For a given target file, not all of the above information is necessarily included. Since the Linux 0.12 system uses the memory management function of the Intel CPU, it allocates a separate 64MB address space (logical address space) for each execution program. In this case, because the linker has processed the execution file to start from a fixed address, the relocation information is no longer needed in the relevant executable file. Below we explain some of the important areas or parts.

3.5.1.1 Executive header

In the header portion of the target file, there is a 32-byte exec data structure, commonly referred to as a file header structure or an execution header structure. Its definition is as follows. For more information about the a.out structure, see the introduction after the include/a.out.h file.

```
struct exec {
    unsigned long a_magic;           /* Use macros N_MAGIC, etc for access */
    unsigned a_text;                 /* length of text, in bytes */
    unsigned a_data;                 /* length of data, in bytes */
    unsigned a_bss;                  /* length of uninitialized data area for file, in bytes */
    unsigned a_syms;                 /* length of symbol table data in file, in bytes */
    unsigned a_entry;                /* start address */
    unsigned a_trsize;               /* length of relocation info for text, in bytes */
    unsigned a_drsiz;                /* length of relocation info for data, in bytes */
}
```

According to the value of the magic number field of the header structure in the a.out file, we can divide the a.out file into several types. The Linux 0.12 system uses two types: The module object file uses the OMAGIC (Old Magic) type of a.out format, which indicates that the file is an object file or an impure executable file. The

magic number is 0x107 (octal 0407). The executable file uses the ZMAGIC a.out format, which indicates that the file is an executable file for demand-driven (demand-paging, ie, load on demand) loading. The magic number is 0x10b (octal 0413). The main difference between these two formats is the way they allocate storage to each part. Although the total length of the structure is only 32 bytes, for a ZMAGIC type executable file, the beginning portion of the file requires a space of 1024 bytes for the head structure. Except 32 bytes occupied by the header structure, the rest is 0. The text segment and data segment of the program are only placed after 1024 bytes. For an OGMIC type .o module file, the 32-byte header structure at the beginning of the file is followed by the code area and the data area.

The `a_text` and `a_data` fields in the execution header structure indicate the byte lengths of the read-only code segment and the read-write data segment, respectively. The `a_bss` field indicates the length of the uninitialized data area (bss section) following the data segment when the kernel loads the target file. Since Linux automatically zeros memory when allocating memory, the bss section does not need to be included in a module file or an executable file. In order to visually represent that the target file logically has a bss segment, a dashed box will be used in the later illustration to represent the bss segment in the target file.

The `a_entry` field specifies the address at which the program code begins execution, while the `a_syms`, `a_trsize`, and `a_drsize` fields describe the size of the relocation information for the symbol table, code, and data segments after the data segment, respectively. Symbol tables and relocation information are not required for executable files, so unless the linker includes symbol information for debugging purposes, the fields in the execution file are usually zero.

3.5.1.2 Relocation information section

The Linux 0.12 system's module files and executable files are all object files in the a.out format, but only the compiler-generated module files contain relocation information for linking programs. The relocation information of the code segment and the data segment is composed of relocation records (items). The length of each record is 8 bytes. Its structure is as follows.

```
struct relocation_info
{
    /* Address (within segment) to be relocated.  */
    int r_address;
    /* The meaning of r_symbolnum depends on r_extern.  */
    unsigned int r_symbolnum:24;
    /* Nonzero means value is a pc-relative offset
       and it should be relocated for changes in its own address
       as well as for changes in the symbol or section specified.  */
    unsigned int r_pcrel:1;
    /* Length (as exponent of 2) of the field to be relocated.
       Thus, a value of 2 indicates 1<<2 bytes.  */
    unsigned int r_length:2;
    /* 1 => relocate with value of symbol.
       r_symbolnum is the index of the symbol
       in file's the symbol table.
       0 => relocate with the address of a segment.
       R_symbolnum is N_TEXT, N_DATA, N_BSS or N_ABS
       (the N_EXT bit may be set also, but signifies nothing).  */
    unsigned int r_extern:1;
    /* Four bits that aren't used, but when writing an object file
       it is desirable to clear them.  */
    unsigned int r_pad:4;
```

```
};
```

There are two functions for relocating items. One is that when the code segment is relocated to a different base address, the relocation item is used to indicate where it needs to be modified. Second, when there is a reference to an undefined symbol in the module file, the linker can use the corresponding relocation item to correct the value of the symbol when the undefined symbol is finally defined. It can be seen from the structure of the above relocation record items that each record item contains an address in the code section (code segment) and data section (data segment) of the module file that needs to be relocated at a length of 4 bytes and specifies how to weigh Positioning operation information. The address field `r_address` refers to the offset value of the relocatable item from the beginning of the code segment or data segment. The 2-bit length field `r_length` indicates the length of the relocated entry, and 0 to 3 indicates that the width of the relocated entry is 1 byte, 2 bytes, 4 bytes, or 8 bytes, respectively. The flag `r_pcrel` indicates that the relocated item is a "PC related" item, ie it is used as a relative address in the instruction. The external flag `r_extern` controls the meaning of `r_symbolnum` and indicates whether the relocation entry refers to a segment or a symbol. If the flag value is 0, the relocation entry is a normal relocation entry, and the `r_symbolnum` field specifies in which segment the positioning is addressed. If the flag is 1, then the relocation entry is a reference to an external symbol. In this case, `r_symbolnum` specifies a symbol in the symbol table in the target file and needs to be relocated using the value of the symbol.

3.5.1.3 Symbol Table and String Section

The last part of the target file is the symbol table and the related string table. The structure of the symbol table entry is as follows.

```
struct nlist {
    union {
        char      *n_name;           // String pointer,
        struct nlist *n_next;        // Or a pointer to another symbolic item structure,
        long       n_strx;           // Or the byte offset value of the symbol name in the table.
    } n_un;
    unsigned char n_type;             // This byte is divided into 3 fields. see a.out.h file.
    char          n_other;           // Usually not used.
    short         n_desc;            //
    unsigned long n_value;           // Symbol' s value.
};
```

Since the GNU gcc compiler allows arbitrary-length identifiers, the identifier strings are located in a string table after the symbol table. Each symbol table entry has a length of 12 bytes, where the first field gives the symbol name string (null-terminated) offset from the string table. The type field `n_type` indicates the type of symbol. The last bit of this field is used to indicate if the symbol is external (global). If this bit is 1, then the symbol is a global symbol. The linker does not need local symbol information, but it can be used by the debugger. The remaining bits of the `n_type` field are used to indicate the symbol type. The `a.out.h` header file defines these types of value constant symbols. The main types of symbols include:

- `text`, `data`, or `bbs` indicates the symbols defined in this module file. The value of the symbol at this time is the relocatable address of the symbol in the module.
- `abs` indicates that the symbol is an absolute (fixed) non-relocatable symbol. The value of the symbol is the fixed value.
- `undef` indicates that it is an undefined symbol in this module file. The value of the symbol at this time is usually 0.

However, as a special case, the compiler can use an undefined symbol to ask the linker to reserve a memory space for the specified symbolic name. If an undefined external (global) symbol has a non-zero value, then for the linker the value is the size of the memory that the program wishes to specify for symbolic addressing. During the linking operation, if the symbol is really not defined, then the linker creates a memory space for the symbol name in the bss section. The size of the space is the largest value of the symbol in all linked modules. This is the so-called Common block definition in the bss section. It is mainly used to support uninitialized external (global) data. For example, an uninitialized array defined in the program. If the symbol has already been defined in any module, the linker will use this definition and ignore the value.

3.5.2 Target file format in Linux 0.12

In the Linux 0.12 system, we can use the `objdump` command to view the specific values of the file header structure in the module file or executable file. For example, the following lists the specific values of the headers in the `hello.o` object file and its executable file.

```
[/usr/root]# gcc -c -o hello.o hello.c
[/usr/root]# gcc -o hello hello.o
[/usr/root]#
[/usr/root]# hexdump -x hello.o
00000000  0107  0000  0028  0000  0000  0000  0000  0000
00000010  0024  0000  0000  0000  0010  0000  0000  0000
00000020  6548  6c6c  2c6f  7720  726f  646c  0a21  0000
00000030  8955  68e5  0000  0000  e3e8  ffff  31ff  ebc0
00000040  0003  0000  c3c9  0000  0019  0000  0002  0d00
00000050  0014  0000  0004  0400  0004  0000  0004  0000
00000060  0000  0000  0012  0000  0005  0000  0010  0000
00000070  0018  0000  0001  0000  0000  0000  0020  0000
00000080  6367  5f63  6f63  706d  6c69  6465  002e  6d5f
00000090  6961  006e  705f  6972  746e  0066
0000009c
[/usr/root]# objdump -h hello.o
hello.o:
magic: 0x107 (407)machine type: 0 flags: 0x0 text 0x28 data 0x0 bss 0x0
nsyms 3 entry 0x0 trsize 0x10 drsize 0x0
[/usr/root]#
[/usr/root]# hexdump -x hello | more
00000000  010b  0000  3000  0000  1000  0000  0000  0000
00000010  069c  0000  0000  0000  0000  0000  0000  0000
00000020  0000  0000  0000  0000  0000  0000  0000  0000
*
0000400  448b  0824  00a3  0030  e800  001a  0000  006a
0000410  dbe8  000d  eb00  00f9  6548  6c6c  2c6f  7720
0000420  726f  646c  0a21  0000  8955  68e5  0018  0000
.....
--More--q
[/usr/root]#
[/usr/root]# objdump -h hello
hello:
magic: 0x10b (413)machine type: 0 flags: 0x0 text 0x3000 data 0x1000 bss 0x0
nsyms 141 entry 0x0 trsize 0x0 drsize 0x0
```

```
[/usr/root]#
```

It can be seen that the magic number of the hello.o module file is 0407 (OMAGIC), and the code segment immediately follows the header structure. In addition to the header structure, a code segment of 0x28 bytes in length and a symbol table with 3 symbol items and code segment relocation information of 0x10 bytes in length are included. The rest of the segments are 0 in length. The magic number of the corresponding execution file hello is 0413 (ZMAGIC), and the code segment is stored from the file offset position 1024 bytes. The lengths of the code and data segments are 0x3000 and 0x1000 bytes, respectively, with a symbol table containing 141 items. We can use the command strip to delete the symbol table information in the execution file. For example, below we delete the symbol information in the hello execution file. It can be seen that the length of the symbol table of the hello execution file becomes 0, and the length of the hello file is also reduced from the original 20591 bytes to 17412 bytes.

```
[/usr/root]# ll hello
-rwx--x--x  1 root    4096      20591 Nov 14 18:30 hello
[/usr/root]# objdump -h hello
hello:
magic: 0x10b (413)machine type: 0flags: 0x0text 0x3000 data 0x1000 bss 0x0
nsyms 141 entry 0x0 trsize 0x0 drsize 0x0

[/usr/root]# strip hello
[/usr/root]# ll hello
-rwx--x--x  1 root    4096      17412 Nov 14 18:33 hello
[/usr/root]# objdump -h hello
hello:
magic: 0x10b (413)machine type: 0flags: 0x0text 0x3000 data 0x1000 bss 0x0
nsyms 0 entry 0x0 trsize 0x0 drsize 0x0
[/usr/root]#
```

Figure 3-8 shows the correspondence between the areas of the a.out executable file in the process logical address space on the disk. The logical space size of a process in a Linux 0.12 system is 64 MB. For the ZMAGIC a.out executable file, its code area is an integer multiple of the memory page size. Since the Linux 0.12 kernel uses the Demand-paging technique, which means that a page of code is actually loaded into a physical memory page, it is only set for the fs/execve() function of the load operation. The paging mechanism of page directory entries and page table entries, so demand page technology can speed up the loading process.

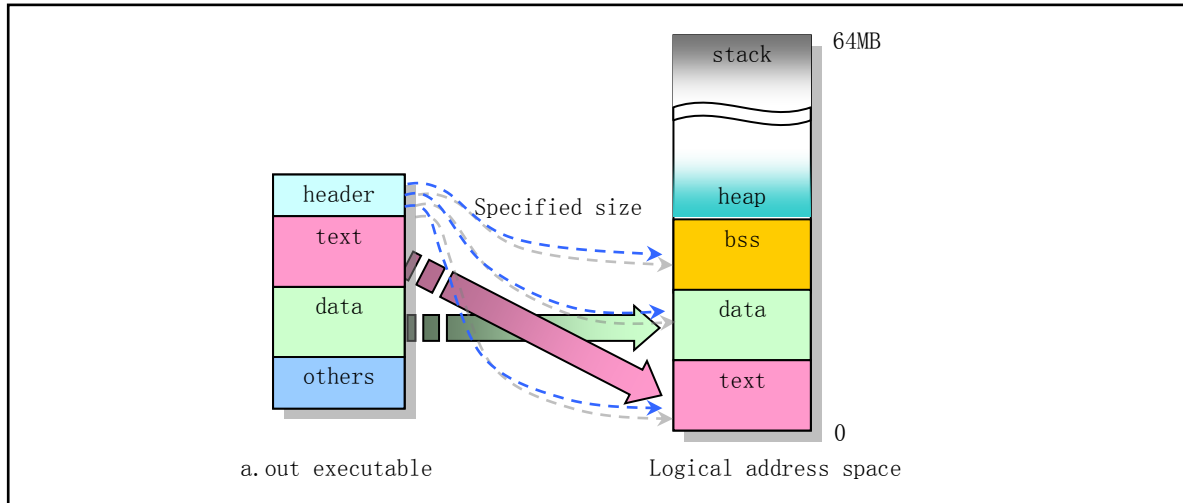


Figure 3-8 a.out execution file maps to process logical address space

In the figure, bss is the uninitialized data area of the process and is used to store static uninitialized data. The first page of bss memory will be set to all 0s at the beginning of program execution. The heap in the figure is a heap space area, which is used to allocate the memory space dynamically requested by the process during execution.

3.5.3 Linker output

The linker processes one or more input object files and related library function objects, and ultimately generates a corresponding binary execution file or a large module file composed of all modules. In this process, the link program's primary task is to perform storage space allocation operations for the execution file (or output module file). Once the storage location is determined, the linking program can continue to perform symbol-bonding operations and code revision operations. Because most of the symbols defined in the module file are related to the storage location in the file, there is no way to resolve the symbol before the corresponding position of the symbol is determined.

Each module file includes several types of segments. The second task of the linker is to join together the segments of the same type in all modules and form a single segment for the specified segment type in the output file. For example, the linker needs to merge the code segments from all the input module files into a single segment and place it in the output executable file.

For a.out format module files, since the segment types are known in advance, the linker can easily store and allocate a.out format module files. For example, for the case of having two input module files and the need to connect a library function module, its storage allocation is shown in Figure 3-9. Each module file has a code section, a data section, and a bss section. There may also be some common blocks that appear to be external (global) symbols. The linker collects the size of each of the module files, including code segments, data segments, and bss segments in any library function module. After all the modules have been read in and processed, any unresolved external symbols with a non-zero value will be treated as common blocks and their allocations will be stored at the end of the bss section.

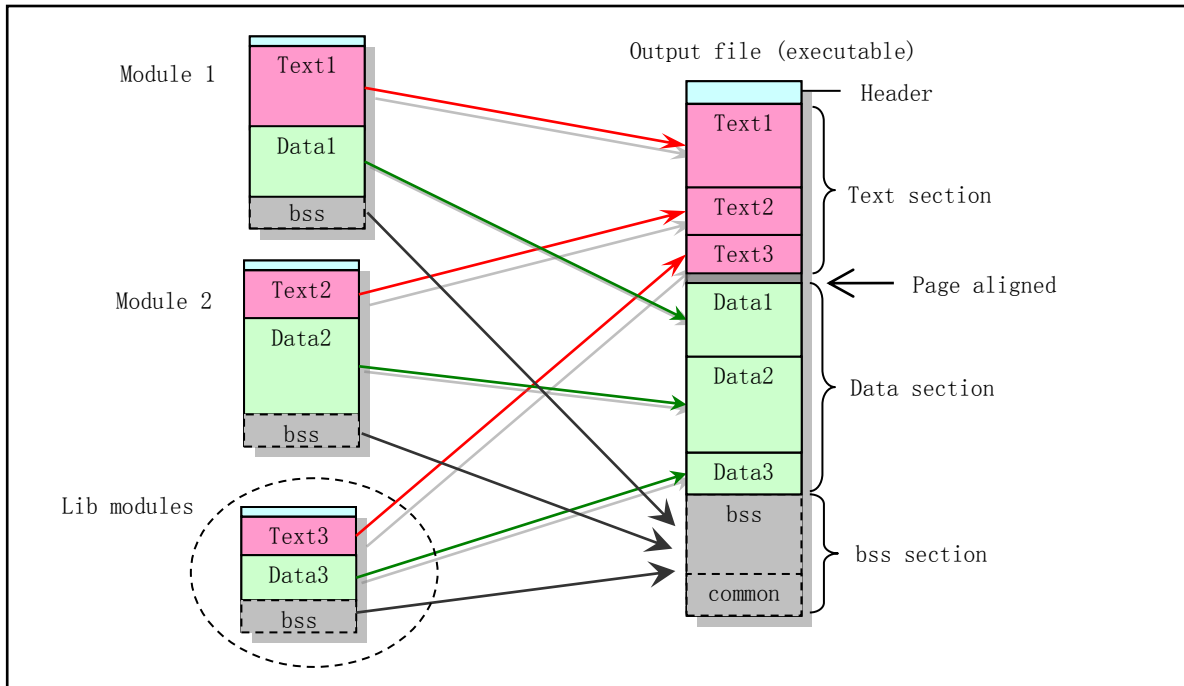


Figure 3-9 Object files link operation

The linker can then assign addresses to all segments. For the ZMAGIC type a.out format used in the Linux 0.12 system, the code segment in the output file is set to start at fixed address 0. The data segment starts from the next page boundary after the code segment. The bss section immediately follows the data segment. Within each segment, the linker stores the same type of segments in the input module file and aligns them by word.

When Linux 0.12 kernel loads an executable file, it will first determine whether the file is a suitable executable file based on the information in the file header structure, that is, if the magic number type is ZMAGIC, then the system is at the top of the user mode stack. The program sets the environmental parameters and parameter information blocks entered on the command line and builds a task data structure for it. Then use the stack return technique to execute the program after setting some related register values. The code and data in the execution program image file will be dynamically loaded into memory using Load on Demand when actually executed or used.

For the Linux 0.12 kernel compilation process, it is done using the make command to command the compiler and linker operations based on the kernel configuration file Makefile. In the build process, make also uses the build.c program in the kernel source code in the tools/ directory to compile and build a temporary tool program build that is used to combine all modules. Since the kernel is loaded into memory by the boot-up program using the ROM BIOS interrupt call, the execution header structure in the compiled kernel modules needs to be removed. The main function of the utility program build is to remove the execution header structures in the bootsect, setup, and system files, and then combine them sequentially to produce a kernel image file named Image.

3.5.4 Linker Predefined Variables

During linking, the linker ld and ld86 use variables to record the logical address of each segment in the execution program. Therefore, in the program, you can access the external variables to obtain the position of the program's middle segment. The linker's predefined external variables are usually at least etext, _etext, edata, _edata, end, and _end.

The addresses of the variable names `_etext` and `etext` are the first address after the program text segment ends; the addresses of `_edata` and `edata` are the first address after the initial data area; the addresses of `_end` and `end` are after the uninitialized data area (bss). The first address location. Names prefixed with the underscore `'_'` are equivalent to the underlined counterparts. The only difference between them is that the symbols `etext`, `edata`, and `end` are not defined in the ANSI, POSIX, and other standards.

When the program just begins to execute, its `brk` location is in the same position as `_end`. But system calls `sys_brk()`, memory allocation functions `malloc()`, and standard input/output operations will change this position. Therefore, the current `brk` position of the program needs to be obtained using `sbrk()`. Note that these variable names must be treated as addresses. Therefore, when accessing them, you need to use the address prefix `'&'`, such as `&end`. E.g:

```
extern int _etext;
int et;

(int *) et = &_etext;           // et contains the address after the end of text segment.
```

The following program `predef.c` can be used to display the addresses of these variables. It can be seen that the address value is the same for the `band` and the underscore `'_'` symbol.

```
/*
Print the symbols predefined by linker.
*/
extern int end, etext, edata;
extern int _etext, _edata, _end;
int main()
{
    printf("&etext=%p, &edata=%p, &end=%p\n",
           &etext, &edata, &end);
    printf("&_etext=%p, &_edata=%p, &_end=%p\n",
           &_etext, &_edata, &_end);
    return 0;
}
```

Running this program on a Linux 0.1X system gives the following results. Please note that these addresses are logical addresses in the program's address space, which is the address from when the execution program was loaded into the memory location.

```
[/usr/root]# gcc -o predef predef.c
[/usr/root]# ./predef
&etext=4000, &edata=44c0, &end=48d8
&_etext=4000, &_edata=44c0, &_end=48d8
[/usr/root]#
```

If you run this program on a modern Linux system (such as RedHat 9 or later), the following results can be obtained. We know that the program code in the Linux system is now stored from its logical address `0x08048000`, so we can see that the program's code segment length is `0x41b` bytes.

```
[root@plinux]# ./predef
&etext=0x804841b, &edata=0x80495a8, &end=0x80495ac
&_etext=0x804841b, &_edata=0x80495a8, &_end=0x80495ac
[root@plinux]#
```

When the Linux 0.1x kernel initializes the block device cache (fs/buffer.c), it uses the variable name `_end` to obtain the location of the kernel image file `Image` in memory at the end of the file, and from this position, the high speed is set. `Buffer`.

3.5.5 System.map file

When running the GNU linker `gld(ld)`, if the `-M` option is used, or the `nm` command is used, the link map information is printed on the standard output device (usually the screen). The target program memory address map information generated by the linker. It lists the location information of the program segment loaded into memory. Specifically, there is the following information:

- Locations of object files and symbol information mapped in memory;
- How public symbols are placed;
- All file members included in the link and their referenced symbols.

Usually we will redirect the link image information sent to the standard output device to a file (eg `System.map`). When compiling the kernel, the `System.map` file generated by the `linux/Makefile` file is used to store kernel symbol table information. The symbol table is a list of all kernel symbols and their corresponding addresses. Of course, it also includes address information of symbols such as `_etext`, `_edata`, and `_end` described above. With each kernel compilation, a new corresponding `System.map` file is generated. When an error occurs in the kernel, the variable name corresponding to an address value can be found by parsing the symbol table in the `System.map` file, or vice versa.

By using the `System.map` symbol table file, we can get more easily identified information when a kernel or related program fails. Examples of symbol tables are as follows:

```
c03441a0 B dmi_broken
c03441a4 B is_sony_vaio_laptop
c03441c0 b dmi_ident
c0344200 b pci_bios_present
c0344204 b pirq_table
```

Each line describes a symbol, the first column indicates the symbol value (address), the second column is the symbol type, indicates which section of the target file the symbol is located in or its attributes, and the third column is the corresponding symbol name.

The symbol type indicators in the second column usually have the types shown in Table 3-5, and there are some related to the target file format adopted. If the symbol type is a lowercase character, the symbol is local; if it is an uppercase character, the symbol is global (external). See the definition of the `nlist{}` structure `n_type` field in the file `include/a.out.h` (lines 110-185).

Table 3-5 The symbol type in the target file symbol list file

Symbol type	Name	Description
-------------	------	-------------

A	Absolute	The value of the symbol is an absolute value and will not be changed during further linking.
B	BSS	The symbols are in the uninitialized data section, ie in the BSS section.
C	Common	The symbol is public. Public symbols are uninitialized data. When linking, multiple public symbols may have the same name. If the symbol is defined elsewhere, the public symbol is treated as an undefined reference.
D	Data	The symbol is in the initialized data section.
G	Global	The symbol is in the initialized data section of the small object. The format of some object files allows more efficient access to small data objects such as a global integer variable.
I	Indirect	A symbol is an indirect reference to another symbol.
N	Debugging	The symbol is a debugging symbol.
R	Read only	The symbol is in a read-only data section.
S	Small	symbol in a small object's uninitialized data section.
T	Text	Symbols in code sections.
U	Undefined	The symbol is external and its value is 0 (undefined).
-	Stabs	The symbol is a stab symbol in the a.out object file and is used to save debugging information.
?	Unknwon	The type of symbol is unknown or related to a specific file format.

It can be seen that the variable named `dmi_broken` is located at kernel address `0xc03441a0`.

`System.map` is where the software that uses it (such as the kernel logging daemon `klogd`) can find it. When the system is started, `klogd` will search `System.map` in three places if the location of `System.map` is not given as `klogd` in the form of a parameter. as followed:

```
/boot/System.map
/System.map
/usr/src/linux/System.map
```

Although the kernel itself does not actually use `System.map`, other programs, such as `klogd`, `lsof`, `ps`, and other software like `dosemu`, require a correct `System.map` file. Using this file, these programs can find the corresponding kernel variable name based on the known memory address to facilitate debugging of the kernel.

3.6 Make Command and Makefile

As you can see from the examples given above, when creating an executable file that is generated by one or a few source programs, you only need to type a few simple lines of commands. But for a large program such as a kernel that consists of hundreds or even thousands of source files, compiling all code files by manually typing lines of commands is extremely complicated. The `make` command is the best tool designed to automatically handle this type of situation. Its main purpose is to be able to automatically determine which files need to be recompiled in a large project containing many source files and issue these files with a recompilation command. Let's take the compiling C program as an example to illustrate how to use `Make` briefly, but you can apply it to any programming language that can be compiled using shell commands. For detailed usage of `make`, please refer to "GNU make manual".

In order to use the `make` tool, we need to write a text file named `Makefile` (or `makefile`) for `make` execution.

The Makefile mainly contains some execution rules and commands required to tell make the relationships among files and what compile and link operations are required for the source files involved to generate the corresponding target files.

When "make" recompiles the source files, each modified source file is recompiled. If the header file has changed, each source file containing the header file will be recompiled. Each compilation generates an object file that corresponds to the source file. Finally, if all source files have been recompiled, all object files, whether newly created or saved from a previous compilation, will be linked together to produce a new executable.

3.6.1 Contents of the Makefile

A Makefile can include five elements: explicit rules, implicit rules, variable definitions, directives, and comments.

- Explicit rules are used to specify when and how to recompile one or more files called rule's targets. The rules explicitly list the other files on the target that depend on the prerequisites (or dependencies), as well as the commands for creating or updating targets.
- Implicit rules are based on the name of the target and the object to determine when and how to recompile one or more files called targets of the rule. This rule describes how the target depends on a file that is similar to the target name and will be given to create or update such a target file.
- Variable definitions define a text string for a variable on one line. This variable can be replaced in subsequent statements. For example, the variables object in the example below defines a list of all .o files.
- A directive is a command of make that indicates the specific operation that it performs when it reads a Makefile. These operations can include reading another makefile; determining whether to use or ignore a portion of the makefile and defining a variable from a string containing multiple lines.
- Comments are the parts of the text in the Makefile that begin with the '#' character. If we really need to use the '#' character, we need to escape it by adding a backslash character ('\#') before the character. Comments can appear anywhere in the Makefile. In addition, a command line script in the Makefile that begins with the TAB is passed to the shell in its entirety, and the shell determines whether it is a command or just a comment.

Once we have written an appropriate Makefile, we can simply type "make" each time we modify the source code to perform all necessary program updates. make determines which files need to be updated (recompiled) based on the contents of the Makefile and the last updated time of the file. For each file that needs to be updated, make executes the relevant commands recorded in the Makefile.

3.6.2 Rules in the Makefile File

A simple Makefile contains some of the following rules. These rules are mainly used to describe the dependencies between operating objects (source files and object files).

```
target ...: prerequisites ...
    command
    ...
    ...
```

Among them, the target object usually refers to the name of a file generated by the program, for example, it can be an executable file or an object file that ends with ".o". The target can also be the name of the activity to be

taken, for example "clean".

The prerequisite is a series of files or other targets necessary to create a target. The target usually depends on multiple such necessary or target files.

The command (command) refers to the operations performed by make, which are usually shell commands that generate the target. When the last modification time of one or more files in the prerequisite condition is newer than that of the target file, the command of the rule will be executed. In addition, there can be multiple commands in one rule, each command occupies a single line in the rule. Please note that we need to type a tab character (press Tab) before writing each command!

Typically, the command exists in a rule with prerequisites and is used to create a target file when any of the prerequisites change. However, the rules do not necessarily have preconditions. For example, a rule containing a delete command related to the target "clean" does not contain prerequisites.

A rule explains how and when to remake certain files, which are the targets of some specific rules. make will execute the command based on the prerequisites to create or update the target. A rule can also explain how and when to perform an operation.

In addition to the rules, a Makefile can also contain other text. However, for a simple Makefile it is usually sufficient to include only a few rules. Some rules may be more complex than the ones given earlier, but they are basically similar.

3.6.3 A Simple Makefile

Below we discuss a simple Makefile that describes how to compile and link a text editor program consisting of eight C source files and three header files.

When make recompiles C files based on the contents of the Makefile, only the modified C files are recompiled. Of course, if a .h header file is modified, then in order to ensure that the program is compiled correctly, every C code file that contains the header file is recompiled. Each compilation operation produces a target file that corresponds to the source program. The net result is that if any of the modified source code files are compiled, then all .o object files that are generated (including those that were just compiled and unmodified before the source code is compiled) need to be linked together to generate a new one. Executable editor program.

The contents of the Makefile example file describe how an executable named edit depends on 8 object files, and how the 8 object files depend on 8 C source files and 3 header files. In this example, all C files contain the "defs.h" header file, but only those C files that define the edit command contain "command.h" and only the underlying C file that changes the edit buffer contains "buffer.h" "head File.

```
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
    cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o

main.o : main.c defs.h
    cc -c main.c
kbd.o : kbd.c defs.h command.h
    cc -c kbd.c
command.o : command.c defs.h command.h
    cc -c command.c
display.o : display.c defs.h buffer.h
    cc -c display.c
insert.o : insert.c defs.h buffer.h
    cc -c insert.c
search.o : search.c defs.h buffer.h
    cc -c search.c
```

```
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

To use the Makefile to create the "edit" executable file, simply type “make” on the command line.

To use the Makefile to remove the compiled executable file and all object files from the current directory, just type “make clean”.

In the Makefile, the target of the rule includes the execution file "edit" and the .o object file "main.o", "kbd.o", and the like. Prerequisite (or dependent) files are source files such as "main.c" and "defs.h". In fact, we can see that each ".o" file is both a rule goal and a necessary prerequisite file for another rule. The commands include "cc -c main.c" and "cc -c kbd.c".

When the target is a file, then any dependent files in its prerequisites need to be recompiled or linked when they are modified. Of course, the file in the precondition that is itself an object should be updated first. In this example, "edit" depends on eight .o target files; and the .o target file "main.o" depends on the source file "main.c" and the header file "defs.h".

On the next line of the rule's target and prerequisites in the Makefile is the shell command. These shell commands indicate how to use the files in the prerequisite to update or generate the target object file. Note that we need to type a tab before each command line to distinguish the command line and other lines in the Makefile. What make does is execute the commands in the rule when the target needs to be updated.

The target "clean" is not a file, but just the name of an operation (activity). Because we generally do not require that this action be performed in its rules, "clean" is not a prerequisite for any other rule. The result is that make does not enforce this rule unless it is explicitly stated. Note that this rule (target) is not only a prerequisite for any other rule, it does not contain nor does it require any prerequisites. So the sole purpose of this rule is to execute the specified command. For such a rule, its target does not refer to or depend on any other file, but only indicates a specific operation. This target is called a phony target.

3.6.4 How make handles Makefile

By default, make will start from the first target in the Makefile (not including targets starting with '.'). This first goal is called the default goal of the Makefile. The ultimate goal is to make an effort to try to update the target.

In the above example, the default end goal is to update or create the execution program "edit", so we put the corresponding rules at the top of the Makefile. When we type the command make on the command line, make will read the Makefile and start processing the first rule. In the example, the first rule is to re-link to generate "edit", but before make can completely process this rule, it must first process the rules of the file that "edit" depends on. In the example, you need to first create or update those .o object files. Each .o file will be processed according to its own rules, that is, by compiling the respective source files to generate the respective .o object files. If any source or header file that is a target prerequisite is newer than the .o object files or the .o object files do not exist, recompilation is required to update or create the corresponding .o object file. .

Some of the other rules in the Makefile will also be processed if their goals (files) appear in the prerequisites of the final target. If the final goal (or any goal) does not depend on some other rule, make will not process these rules unless we actively request make to handle it. For example, when running make, we can give the target name of a specific rule in the Makefile on the command line to perform the specified update operation, such as using the

command "make clean".

Before recompiling a .o object, make first considers updating its prerequisites, source files, and header files. However, the Makefile does not specify any operation for the source and header files, ie, the source and header files are not the target of any rules, so make will not perform any processing on these source files.

After recompiling the desired .o object file, make decides whether to perform a relink to generate an updated edit program "edit". This will only be done if "edit" does not exist or if any .o target file is newer than "edit". If an .o object file has just been recompiled, it will be newer than "edit" and make will be relinked to generate a new "edit".

Therefore, if we modify the file "insert.c" and run make, make compiles the source file to update the corresponding "insert.o" and then links "edit". If we modify the header file "command.h" and run make, make will recompile the target files "kbd.o", "command.o" and "files.o" and then link to generate a new executable file "edit".

In general, make will use the contents of the Makefile to determine which .o object files need to be updated, and then determine which of the target files do need to be updated. If the .o object file is newer than all of its related files, the .o object is already up-to-date and no further updates are required. Of course, all necessary targets in the input condition (prerequisite) as the first final target are updated beforehand.

3.6.5 Variables in the Makefile

In the above example, we need to list all .o target files twice in the "edit" rule (see below):

```
edit : main.o kbd.o command.o display.o insert.o search.o files.o utils.o
      cc -o edit main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

This duplicate information is easy to make mistakes. If we add a new .o object file to the program, it is possible to add the .o object file name to a list but forget to add it in another place. By using a variable, we are likely to reduce the risk of this error, and also make the Makefile look more concise. Using variables allows us to define a text string once, which can then be replaced in several places.

For Makefiles, the typical practice is to define a variable named objects or OBJECTS to represent a list of all .o object files. We usually use the following line in the Makefile to define a variable objects:

```
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o
```

After that, in every place where you need to list the .o object files, you can replace the value of the variable by writing "\$(objects)".

3.6.6 Let make automatically deduce commands

We do not need to give the relevant commands in the rules in order to compile each C source program. Because make itself can judge it: it has an implied rule, which uses the 'cc -c' command according to the naming of the target file, and updates the corresponding .o file according to the corresponding .c file. For example, it compiles 'main.c' to 'main.o' using the command 'cc -c main.c -o main.o'. Therefore we can omit the commands in the .o object file rules.

When a .c file is used automatically in this way, it is automatically added to the prerequisites (dependencies). So we can omit the '.c' file in the rule preconditions --- Suppose we also omit the command. The following is a complete Makefile example that includes these two changes and uses variables:

```
objects = main.o kbd.o command.o display.o insert.o search.o files.o utils.o

edit : $(objects)
    cc -o edit $(objects)
main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

clean :
    -rm edit $(objects)
```

This is how we actually write the Makefile file. Because implied rules are so convenient, it is important. We will often see using them.

3.6.7 Implicit rules in automatic variables

If one of the prerequisites (dependent objects) is searched for by searching the directory and is found in another directory, the command of the rule will be executed as scheduled. Therefore, we must carefully set the command so that the command can find the necessary prerequisites in this directory. This can be done by using automatic variables. Automatic variables that are implicit rules are variables that can be automatically replaced on the command line depending on the situation. The value of the automatic variable is set before the regular command is executed. For example, the value of the automatic variable '\$^' represents all the prerequisites for the rule, including the name of the directory they are in; the value of '\$<' represents the first prerequisite in the rule; '\$@' represents the target object (for other automatic variables, see the make manual). Sometimes, when we don't want to specify a header file on the command line, we can include these header files in the prerequisites. At this point, the automatic variable '\$<' is the first prerequisite.

```
foo.o : foo.c defs.h hack.h
    cc -c $(CFLAGS) $< -o $@
```

The '\$<' will be automatically replaced with foo.c and '\$@' will be replaced with foo.o.

In order for make to use idioms to update a target, we can eliminate the need for commands. Write a rule without a command or do not write a rule. At this point, make will determine which implicit rule to use based on the type of source file (file suffix).

In addition, there is an implicit rule called a suffix rule. It is an old-fashioned way of defining implicit rules for make (now that this rule is no longer used, instead using more general and clearer pattern matching rules). Since this rule is used in the Makefile of the Linux 0.1x kernel, here's a brief explanation. The following example is an application of a double suffix rule. Dual suffix rules are defined with a pair of suffixes: source suffix and target suffix. The corresponding implicit prerequisite is obtained by replacing the target suffix with the source suffix in the file name. Therefore, the following '\$<' value at this time is the '*.c' file name. The meaning of the positive make rule is to compile the '*.c' program into the '*.s' code.

```
.c.o:
$(CC) $(CFLAGS) \
-nostdinc -Iinclude -S -o $*.o $<
```

Usually the command belongs to a rule with preconditions (dependency objects) and is used to generate a target file when any of the prerequisites change. However, the rules that specify commands for goals do not necessarily have prerequisites. For example, rules related to the target 'clean' with a delete command do not require a prerequisite. At this point, a rule explains how and when to re-create certain files, which are the targets of specific rules. Make executes commands based on prerequisites to create or update targets. A rule can also explain how and when to perform an operation.

Makefiles can also contain text other than rules, but simple Makefiles need only contain the appropriate rules. The rules may seem much more complex than the template shown above, but they are basically consistent.

The Makefile file can also contain some referenced dependencies between the files. These dependencies are used by make to determine if a target needs to be rebuilt. For example, when a header file is changed, make recompiles all '*.c' files associated with the header file with these dependencies. For an example of a dependency, refer to the Makefile in the kernel source code.

3.7 Summary

In this chapter, several executable assembly language programs are used as description objects, and the basic language and usage of as86 and GNU as assembly language are described in detail. At the same time, the C language extensions used by the Linux kernel are described in detail. For learning the operating system, the object file structure supported by system has a very important role, so this chapter describes the a.out object file format used in Linux 0.12 in detail.

In the next chapter, we will detail the operating principle of the Intel 80X86 processor running in protected mode. Given an example of a protection mode multitasking program, reading this example will give us a basic understanding of how the operating system initially "rotates" and lays a solid foundation for continuing to read the full Linux 0.12 kernel source code.

4 80X86 Protection Mode and Its Programming

The Linux operating system introduced in this book is based on a PC system consisting of an Intel 80X86 processor and related peripheral hardware. The best reference for 80X86 CPU system programming is of course the three-volume "IA-32 Intel Architecture Software Developer's Manual" released by Intel Corporation, especially the third volume: "System Programming Guide". It is an essential reference for understanding the operating principle of the 80X86 CPU or system programming. These information can be downloaded for free from the Intel Corporation website. This chapter mainly describes the architecture of the 80X86 CPU and some basic knowledge of programming in protected mode, laying a solid foundation for preparing to read the source code of Linux kernel based on 80X86 CPU. Mainly include: 1. 80X86 CPU basic knowledge; 2. Protected mode memory management; 3. Various CPU protection methods; 4. Interrupt and exception handling; 5. Task management; 6. Initialization of protection mode programming; 7. A simple multitasking Kernel example.

A simple multitasking kernel program described in the last section of this chapter is a simplified example based on the Linux 0.12 kernel. This example is used to demonstrate the implementation of memory segmentation management and task management. It does not include paging mechanism content. However, if you thoroughly understand the operating mechanism of this example, you should not encounter any major problems when you read the Linux kernel source code later. If the reader is already familiar with this part of the content, you can directly read a runnable kernel sample program given at the end of this chapter. Of course, readers can refer back to this chapter at any time when reading kernel source code.

4.1 80X86 System Registers and System Instructions

To assist the processor in performing initialization and control system operations, the 80X86 provides a flag register, EFLAGS, and several system registers. In addition to some common status flags, EFLAGS also contains several system flags. These system flags are used to control task switching, interrupt handling, instruction tracking, and access permissions. System registers are used for memory management and control processor operations. They contain the base address of the system table for segmentation and paging processing, and the bit flags that control processor operations.

4.1.1 Flag Registers

The system flags and IOPL fields in the flag register EFLAGS are used to control I/O access, maskable hardware interrupts, debugging, task switching, and virtual-8086 modes, as shown in Figure 4-1. Normally only operating system code is allowed to modify these flags. The other flags in EFLAGS are some common flags (carry CF, parity PF, auxiliary carry AF, zero flag ZF, negative SF, direction DF, overflow OF). Here we describe only the system flags in the team EFLAGS.

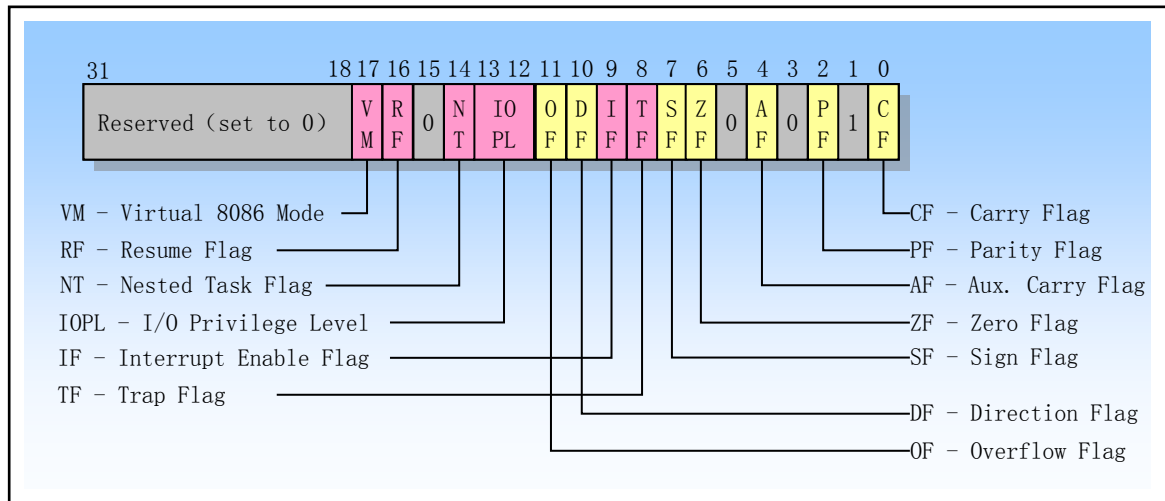


Figure 4-1 System Flags in EFLAGS

- TF** Bit 8 is the Trap Flag. When this bit is set, step-by-step execution can be started for the debug operation; single-step execution is prohibited when reset. In the single-step execution mode, the processor generates a debug exception after each instruction is executed, so that we can observe the state of the execute program after executing each instruction. If the program uses the POPF, POPFD, or IRET instructions to set the TF flag, the processor generates a debug exception after the subsequent instruction.
- IOPL** Bits 13-12 are the I/O Privilege Level field. This field indicates the I/O privilege level IOPL of the currently running program or task. The CPL currently running a program or task must be less than or equal to this IOPL to access the I/O address space. Only when the CPL is at privilege level 0 can the program use the POPF or IRET instructions to modify this field. IOPL is also one of the mechanisms that control the modification of the IF flag.
- NT** Bit 14 is a nested task flag. It controls the chain relationship between the interrupted task and the called task. The processor sets this flag on calls to a task initiated with a CALL instruction, an interrupt, or an exception. When returning from a task by using the IRET instruction, the processor checks and modifies this NT flag. This flag can also be modified using the POPF/POPFD instructions, but changing the state of this flag in the application can generate unexpected exceptions.
- RF** Bit 16 is the Resume Flag. This flag controls the processor's response to breakpoint instructions. When set, this flag temporarily disables the debug exception generated by the breakpoint instruction; when the flag is reset, the breakpoint instruction will generate an exception. The main function of the RF flag is to allow re-execution of an instruction after debugging an exception. Before the debug software uses the IRETD instruction to return to the interrupted program, the RF flag in the EFLAGS content on the stack needs to be set to prevent the instruction breakpoint from causing another exception. The processor automatically clears the flag after the instruction returns, again allowing instruction breakpoint exceptions.
- VM** Bit 17 is the Virtual-8086 Mode flag. When this flag is set, the virtual-8086 mode is turned on; when the flag is reset, it returns to the protected mode.

4.1.2 Memory Management Registers

The processor provides four memory management registers (GDTR, LDTR, IDTR, and TR) that specify the

base address of the system table used for memory segment management, as shown in Figure 4-2. The processor provides specific instructions for loading and saving these registers. For the role of the system table, see the detailed description in the next section, "Protection Mode Memory Management."

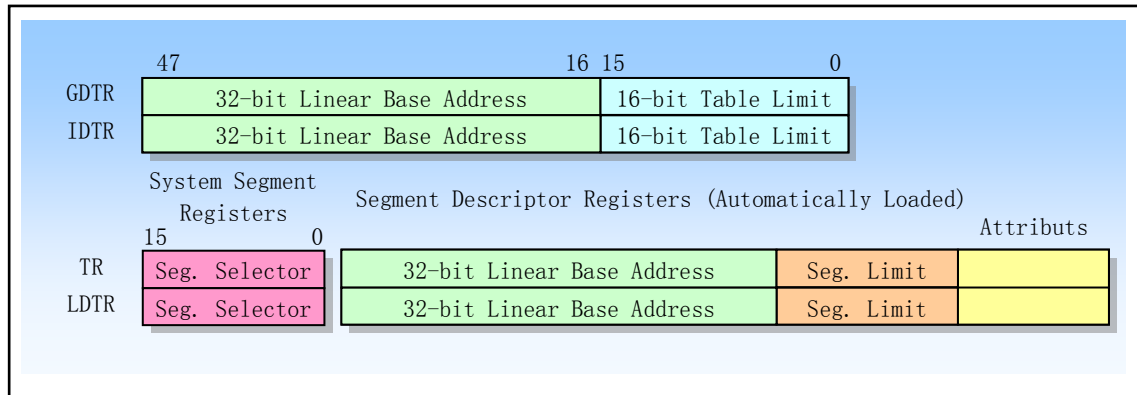


Figure 4-2 Memory management registers

GDTR, LDTR, IDTR, and TR are segment base registers that contain important information tables for the segmentation mechanism. GDTR, IDTR, and LDTR are used to address the segment where the descriptor table is stored. TR is used to address a special task state segment TSS (Task State Segment). The TSS segment contains important information about the currently executing task. Now we explain them in detail.

1. Global Descriptor Table Register (GDTR)

The GDTR register holds the 32-bit linear base address and the 16-bit limit value of the global descriptor table GDT. The base address specifies the address of byte 0 in the GDT table in the linear address space, and the table length indicates the byte length value of the GDT table. The instructions LGDT and SGDT are used to load and set the contents of the GDTR register, respectively. After the machine has just powered up or the processor is reset, the base address is set to 0 by default and the length value is set to 0xFFFF. The GDTR must be loaded with a new value during the initialization of the protection mode.

2. Interrupt Descriptor Table Register (IDTR)

Similar to GDTR, the IDTR register is used to store the 32-bit linear base address and 16-bit table length values of the interrupt descriptor table IDT. The instructions LIDT and SIDT are used to load and set the contents of the IDTR register, respectively. After the machine has just powered up or the processor is reset, the base address is set to 0 by default and the length value is set to 0xFFFF.

3. Local Descriptor Table Register (LDTR)

The LDTR register holds the 16-bit segment selector, 32-bit linear base address, 16-bit segment limit, and descriptor attribute values of the local descriptor table LDT. The instructions LLDT and SLDT are used to load and store the segment selector part of the LDTR register, respectively. The segment containing the LDT table must have a segment descriptor entry in the GDT table. When using the LLDT instruction to load selectors containing LDT segments into the LDTR, the segment base address, segment length, and descriptor attributes of the LDT segment descriptor are automatically loaded into the LDTR. When task switching occurs, the processor automatically loads the segment selector and segment descriptor of the new task LDT into the LDTR. After the machine powers up or the processor resets, the segment selector and base address are set to 0 by default, and the segment length is set to 0xFFFF.

4. Task Register (TR)

The TR register holds the 16-bit segment selector, 32-bit base address, 16-bit segment length, and descriptor

attribute values of the current task TSS segment. It references a TSS type descriptor in the GDT table. The LTR and STR instructions are used to load and save the segment selector portion of the TR register, respectively. When using the LTR instruction to load the selector into the task register, the segment base address, segment length, and descriptor attributes in the TSS descriptor are automatically loaded into the task register. When task switching is performed, the processor automatically loads the segment selector and segment descriptor of the TSS of the new task into the task register TR.

4.1.3 Control Registers

The control registers (CR0, CR1, CR2, and CR3) are used to control and determine the operating mode of the processor and the characteristics of the currently executing task, as shown in Figure 4-3. CR0 contains system control flags that control the operating mode and state of the processor; CR1 is reserved for use; CR2 contains a linear address that causes a page fault. CR3 contains the physical memory base address of the page directory table, so this register is also called the Page-Directory Base Address Register (PDBR).

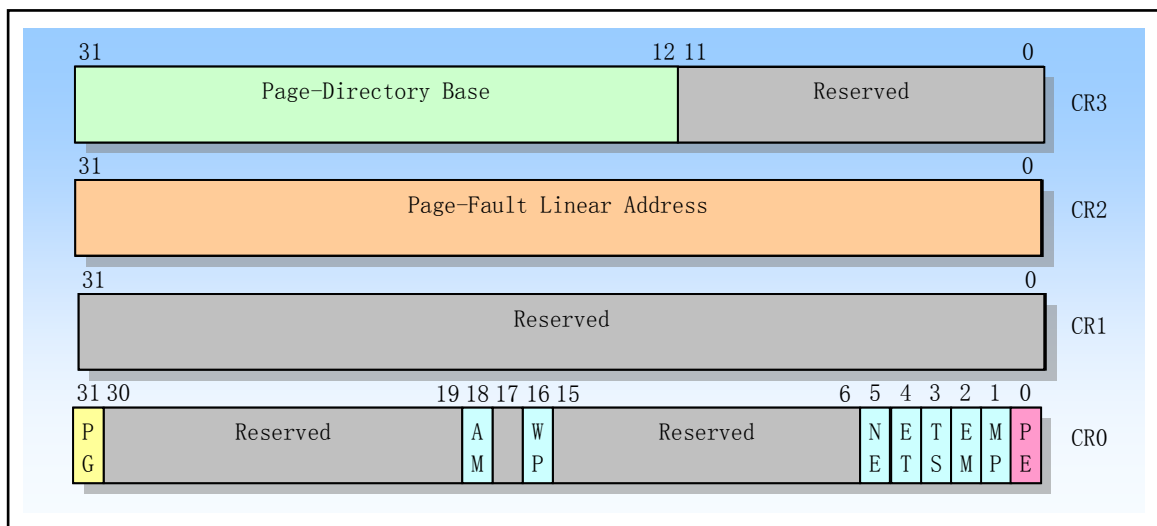


Figure 4-3 Control registers CR0--CR3

1. Coprocessor control bits in CR0

Four bits of CR0: Extended Type Bit (ET), Task Switching Bit (TS), Emulation Bit (EM), and Math Presence Bit (MP) are used to control the operation of the 80X86 floating point (math) coprocessor. For details on coprocessors, see Chapter 11. The ET bit (flag) of CR0 is used to select the protocol used to communicate with the coprocessor, ie to indicate whether the system is using the 80387 or 80287 coprocessor. The TS, MP, and EM bits are used to determine if a float instruction or WAIT instruction should generate a Device Not Available exception. This exception can be used to save and restore floating-point registers only for tasks that use floating-point operations. For tasks that do not use floating-point arithmetic, doing so can speed up the switching between them.

- ET** Bit 4 of CR0 is an Extension Type flag. When the flag is 1, it indicates that the system has a 80387 coprocessor and uses a 32-bit coprocessor protocol. ET=0 indicates use of the 80287 coprocessor. If simulation bit EM=1, this bit will be ignored. During a processor reset operation, the ET bit is initialized to indicate the type of coprocessor used in the system. If there is 80387 in the system, then ET is set to 1, otherwise if there is a 80287 or no coprocessor, ET is set to 0.
- TS** Bit 3 of CR0 is the Task Switched flag. This flag is used to postpone saving the coprocessor content of

the task switch until the new task begins to actually execute the coprocessor instruction. The flag is set by the processor each time the task is switched and tested when the coprocessor instruction is executed.

If the TS flag is set and the EM flag of CR0 is 0, a Device Not Available (DNA) exception is generated before any coprocessor instructions are executed. If the TS flag is set, but the MP0 and EM flags of CR0 are not set, no device exception will not occur until the coprocessor instruction WAIT/FWAIT is executed. If the EM flag is set, the TS flag has no effect on the execution of coprocessor instructions. See Table 4-1.

When the task is switched, the processor does not automatically save the context of the coprocessor but sets the TS flag. This flag causes the processor to generate a device-existent exception when it encounters a coprocessor instruction at any time while executing a stream of new tasks. A device-existent handler can use the CLTS instruction to clear the TS flag and save the coprocessor's context. If the task has never used a coprocessor, the corresponding coprocessor context does not need to be saved.

- EM Bit 2 of CR0 is an EMulation flag. When this bit is set, it means that the processor has no internal or external coprocessor. When the coprocessor instruction is executed, it will generates a device-not-available exception. When cleared, it means that the system has a coprocessor. Setting this flag forces all floating-point instructions to be simulated using software.
- MP Bit 1 of CR0 is the Monitor Coprocessor or Math Present flag. Used to control the interaction between the WAIT/FWAIT instruction and the TS flag. If MP=1 and TS=1, then executing the WAIT instruction will generate a device-not-available exception; if MP=0, the TS flag will not affect the execution of the WAIT.

Table 4-1 Influence of EM, MP and TS Combinations in CR0 on Coprocessor Actions

CR0 Flags			Instruction Type	
EM	MP	TS	Floating-Point	WAIT/FWAIT
0	0	0	Execute	Execute
0	0	1	DNA Exception	Execute
0	1	0	Execute	Execute
0	1	1	DNA Exception	DNA Exception
1	0	0	DNA Exception	Execute
1	0	1	DNA Exception	Execute
1	1	0	DNA Exception	Execute
1	1	1	DNA Exception	DNA Exception

2. Protection and Control bits in CR0

- PE Bit 0 of CR0 is the Protection Enable flag. When this bit is set, the protection mode is enabled; when reset, real address mode is entered. This flag only enables segment-level protection and does not enable paging. To enable the paging mechanism, both the PE and PG flags are set.
- PG Bit 31 of CR0 is a paging signature. When this bit is set, the paging mechanism is enabled; when reset, the paging mechanism is disabled. At this time, all linear addresses are equivalent to physical addresses. The PE flag must be turned on before turning this flag on. That is, to enable the paging mechanism, both the PE and PG flags are set.
- WP For Intel 80486 or higher CPUs, bit 16 of CR0 is the Write Protect flag. When this flag is set, the processor prohibits the superuser program (eg, a privilege level 0 program) from performing a write

operation to the user-level read-only page; when this bit is reset, it is the reverse. This flag is beneficial for UNIX-type operating systems to implement the Copy on Write technique when creating processes.

NE For Intel 80486 or higher CPUs, bit 5 of CR0 is the Numeric Error flag. When this flag is set, the internal reporting mechanism of the X87 coprocessor error is enabled; if the flag is reset, the X87 coprocessor error reporting mechanism in the form of a PC is used. When NE is in the reset state and there is a signal on the CPU's IGNNE input pin, the math coprocessor X87 error will be ignored. When NE is in the reset state and there is no signal on the CPU's IGNNE input pin, an unmasked math coprocessor X87 error will cause the processor to generate an interrupt externally through the FERR pin and perform the next wait form floating-point instruction Stop instruction execution immediately before or WAIT/FWAIT instruction. The FERR pin of the CPU is used to emulate the ERROR pin of the external coprocessor 80387, so it is usually connected to the interrupt controller input request pin. The NE flag, the IGNNE pin, and the FERR pin are used to implement an external error reporting mechanism in the form of a PC using external logic.

The Enable Protected (PE) bit (Bit 0) and the Paging bit (Bit 31) are used to control the segmentation and paging mechanisms, respectively. PE is used to control the segmentation mechanism. If PE=1, the processor operates in the context of the open segmentation mechanism, ie, operates in protected mode. If PE = 0, the processor turns off the segmentation mechanism and operates in the real address mode as in 8086. The PG is used to control the paging mechanism. If PG=1, the paging mechanism is turned on. If PG=0, the paging mechanism is disabled and the linear address is used directly as the physical address.

If PE=0, PG=0, the processor operates in real-address mode; if PG=0, PE=1, the processor operates in protection mode without paging mechanism; if PG=1, PE=0, this Since the paging mechanism cannot be enabled because it is not in protected mode, the processor generates a general protection exception. This flag combination is invalid; if PG=1, PE=1, the processor works under the protection mode with the paging mechanism enabled. .

We must be careful when changing the PE and PG bits. We can only change the setting of the PG bit when the execution program has at least some of the code and data in the linear address space and the physical address space with the same address. At this point, this part of the code with the same address acts as a bridge between paged and non-paged worlds. Regardless of whether the paging mechanism is turned on, this part of the code has the same address. In addition, the page cache TLB must be refreshed before paging is enabled (PG=1).

After the PE bit is modified, the program must immediately use a jump instruction to flush any instructions in the processor's execution pipeline that have acquired different modes. Before setting the PE bit, the program must initialize several system segments and control registers. On power up, the processor is reset to PE=0 and PG=0 (real mode state) to allow the boot code to initialize these registers and data structures before enabling the segmentation and paging mechanism.

3. CR2 and CR3

CR2 and CR3 are used for paging mechanism. CR3 contains the physical address of the page directory table page, so CR3 is also called PDBR. Because the page directory table page is page-aligned, only the upper 20 bits of this register are valid. The lower 12 bits are reserved for use by more advanced processors, so the lower 12 bits must be set to 0 when loading a new value into CR3.

CR2 is used to report error messages when a page exception occurs. When the report page is abnormal, the processor stores the linear address that caused the exception in CR2. Therefore, the page exception handler in the operating system can determine which page in the linear address space caused an exception by checking the contents of CR2.

The use of the MOV instruction to load CR3 has the side effect of invalidating the page cache. In order to reduce the number of bus cycles required for address translation, the most recently accessed page directory and page table are stored in the page cache of the processor, which is referred to as the Translation Lookaside Buffer (TLB). The page table entry is read from memory using an extra bus cycle only if the TLB does not contain the required page table entry.

Even if the PG bit in CR0 is in the reset state (PG = 0), we can load CR3 first to allow the paging mechanism to be initialized. When switching tasks, the contents of CR3 will also change. However, if the CR3 value of the new task is the same as that of the original task, the processor does not need to refresh the page cache. This allows tasks that share page tables to execute faster.

4.1.4 System Instructions

System instructions are used to handle system-level functions. Examples include loading system registers, managing interrupts, and so on. Most system instructions can only be executed by operating system software at privilege level 0. The remaining instructions can be executed at any privilege level so applications can use it. Table 4-2 lists some of the system instructions we will use. It also indicates whether they are protected.

Table 4-2 List of commonly used system instructions

Instruction	Description	Protected?	Description
LLDT	Load LDT Register	Yes	Load LDT segment selectors and segment descriptors from memory into the LDTR register.
SLDT	Store LDT Register	No	Save the LDT segment selector in LDTR to internal memory or general-purpose registers.
LGDT	Load GDT Register	Yes	Load the base address and length of the GDT table from memory into GDTR.
SGDT	Store GDT Register	No	Save the base address and length of the IDT table in GDTR to memory.
LTR	Load Task Register	Yes	Load TSS segment selectors (and segment descriptors) into the task register.
STR	Store Task Register	No	Save the current task TSS segment selector in TR to the memory or general register.
LIDT	Load IDT Register	Yes	The base address and length of the IDT table are loaded from memory into the IDTR.
SIDT	Store IDT Register	No	Store the base address and length of the IDT table in IDTR in memory.
MOV CR _n	Move Control Registers	Yes	Load and save control registers CR0, CR1, CR2, or CR3.
LMSW	Load Machine State Word	Yes	Load the machine status word (corresponds to CR0 bit 15--0). This instruction is for compatibility with the 80286 processor.
SMSW	Store Machine State Word	No	Save the machine status word. This instruction is for compatibility with the 80286 processor.
CLTS	Clear TS flag	Yes	Clears the task switched flag TS in CR0. There are no exceptions for handling devices (coprocessors).
LSL	Load Segment Limit	No	Load Segment Limit
HLT	Halt Processor	Yes	Stop the processor execution.

4.2 Protect Mode Memory Management

The following is a brief introduction to the definition of memory addressing, the use of segmentation and paging mechanism for the principle of the transformation between logical addresses, linear addresses and physical addresses, and the protection mechanism between tasks and privilege levels. Subsequent subsections elaborate on the working principles of each part.

4.2.1 Memory Addressing

Memory refers to an array of ordered bytes, each byte having a unique memory address. Memory addressing refers to locating the address of a specified data object stored in memory. Here, a data object is a numeric value or a string of a specified data type stored in memory. 80X86 supports multiple data types: 1-byte, 2-byte (1 word) or 4-byte (double-word or long-word) unsigned integer or signed integer, and multi-byte character strings. Usually the positioning or addressing of a certain bit in a byte can be addressed on a byte basis, so the addressing of the smallest data type is the positioning of 1-byte data (numeric values or characters). Normally, the memory address is addressed from 0. For the 80X86 CPU, the address bus width is 32 bits, so there are a total of 2^{32} different physical addresses. That is, the memory physical address space has 4G, a total of 4G bytes of physical memory can be addressed. For multi-byte data types (such as 2-byte integer data types), these bytes are stored in memory. The 80X86 first stores the low-value byte and then stores the high-value byte at the address. Therefore, the 80X86 CPU is a small-endian processor.

For the 80X86 CPU, one instruction consists mainly of the opcode and the operand. The operand can be located in a register or in memory. To locate an operand in memory, memory addressing is required. The 80X86 has many instruction operands that involve memory addressing, and there are many different addressing schemes to choose from depending on the type of data being addressed. For memory addressing, the 80X86 uses an addressing technique called **Segment**. Addressing a data object in memory requires the use of a segment's start address (that is, segment address) and an intra-segment offset address. The segment address part is specified using a 16-bit segment selector, of which 14 bits can select 2^{14} powers, ie 16384 segments. The intra-segment offset address portion is specified using a 32-bit value, so the intra-segment address can be 0 to 4G. That is, the maximum length of a segment can reach 4G. A 48-bit address or long pointer consisting of a 16-bit segment selector and a 32-bit offset in the segment thus forms a logical address (virtual address). It uniquely determines the segment address and segment offset address of a data object. An address specified by only a 32-bit offset address or pointer is based on the object address of the current segment. The segmentation mechanism also allows typing of segments so that the operations that may be performed on a particular type of segment can be restricted.

The 80X86 provides six segment registers for storing segment selectors: CS, DS, ES, SS, FS, and GS. Where CS is always used to address the code segment, and the stack segment specifically uses the SS segment register. The segment addressed by CS at any given moment is called the current code segment. At this time, the EIP register contains the offset address within the segment within the current code segment to be executed. Therefore, the address of the instruction to be executed can be expressed as CS:[EIP]. The inter-segment control branch instructions, which will be described later, can be used to assign new values to CS and EIP so that the execution position can be changed to other code segments, thus achieving control transfer of the program in different segments.

The segment addressed by the segment register SS is called the current stack segment. The top of the stack is specified by the contents of the ESP register. So the address at the top of the stack is SS:[ESP]. The other 4 segment registers are general segment registers. When the instruction does not specify a segment of the data to be operated, DS will be the default data segment register.

In order to specify the intra-segment offset address of a memory operand, the 80X86 instruction specifies many ways to calculate the offset, which is called instruction addressing. The instruction's offset consists of three parts: the base address register, the index register, and an offset constant. which is:

$$\text{Offset address} = \text{base address} + (\text{index} \times \text{scale factor}) + \text{offset}$$

4.2.2 Address Translation

Any complete memory management system contains two key parts: protection and address translation. Providing protection prevents one task from accessing another task or operating system's memory area. Address translation allows the operating system to have flexibility in allocating memory to tasks, and because we can make certain physical addresses unmapped by any logical address, memory protection is also provided during address translation.

As mentioned above, the physical memory in the computer is a linear array of bytes, each byte has a unique physical address; the address in the program is a logical address composed of a segment selector and an offset within the segment. This kind of logical address cannot be directly used to access physical memory, but it needs to be transformed or mapped to a physical memory address using an address translation mechanism. The memory management mechanism is used to translate this logical address into a physical memory address.

In order to reduce the information needed to determine the address translation, the translation or mapping usually uses memory blocks as the operating unit. Segmentation mechanism and paging mechanism are two widely used address translation techniques. They differ in how logical addresses are organized into mapped memory blocks, how the translation information is specified, and how programmers operate. Fragmentation and paging operations use tables that reside in memory to specify their respective translation information. These tables can only be accessed by the operating system to prevent unauthorized modifications by the application.

The 80X86 uses segmentation and paging in the translation from logical addresses to physical addresses, as shown in Figure 4-4. The first stage uses a segmentation mechanism to translate the logical address into an address in the processor linear address space. The second stage uses a paging mechanism to translate linear addresses into physical addresses. In the address translation process, the first-stage segmentation mechanism is always used, and the second-stage paging mechanism is optional. If no paging mechanism is enabled, the linear address space generated by the segmentation mechanism is directly mapped to the processor's physical address space. The physical address space is defined as the address range that the processor can generate on its address bus.

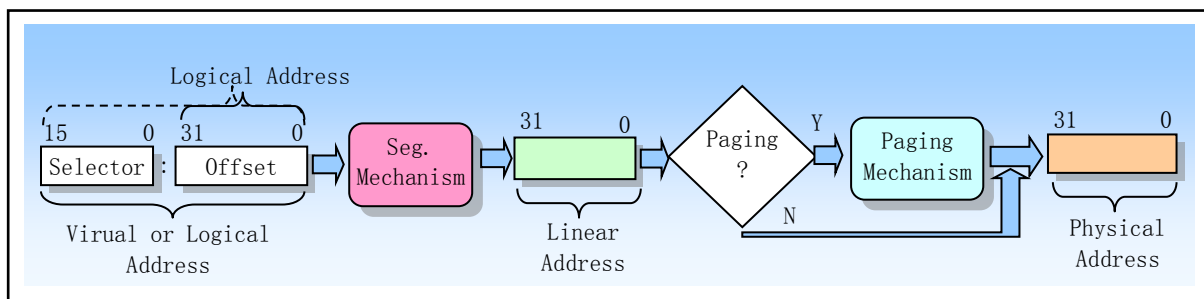


Figure 4-4 Logical address to physical address translation

1. Segmentation mechanism

Segmentation provides a mechanism to isolate individual code, data, and stack areas so that multiple

programs (or tasks) can run on the same processor without interfering with each other. The paging mechanism provides an implementation mechanism for traditional demand pages and virtual memory systems. The virtual memory system is used to realize that the program code is mapped to physical memory as required. The paging mechanism can of course also be used to provide isolation between multiple tasks.

As shown in Figure 4-5, segmentation provides a mechanism for dividing the processor addressable linear address space into smaller, protected address space regions called segments. Segments can be used to hold program code, data, and stacks, or to hold system data structures (such as TSS or LDT). If there are multiple programs or tasks running in the processor, each program can allocate its own set of segments. At this point the processor can enforce the boundaries between these segments and ensure that a program does not interfere with the execution of the program by accessing segments of another program. Segmentation also allows classification of segments. In this way, operations on specific types of segments can be limited.

All used segments in a system are contained in the processor's linear address space. In order to locate a byte in a specified segment, the program must provide a logical address. The logical address includes a segment selector and an offset. The segment selector is the unique identifier of a segment. In addition, the segment selector provides the offset of a data structure (called segment descriptor) in the segment descriptor table (eg, the global descriptor table GDT). Each segment has a segment descriptor. The segment descriptor specifies the size of the segment, the access rights and the privilege level of the segment, the segment type, and the position of the first byte of the segment in the linear address space (referred to as the segment's base address). The offset of the logical address is added to the base address of the segment to locate a byte in the segment. Therefore, the base address plus the offset form the address in the processor's linear address space.

The linear address space has the same structure as the physical address space. Compared to two-dimensional logical address space, both are one-dimensional address spaces. The virtual address (logical address) space can contain up to 16K segments, and each segment can be up to 4GB, resulting in a virtual address space capacity of 64TB (2^{46}). Both the linear and physical address spaces are 4GB (2^{32}). In fact, if the paging mechanism is disabled, the linear address space is the physical address space.

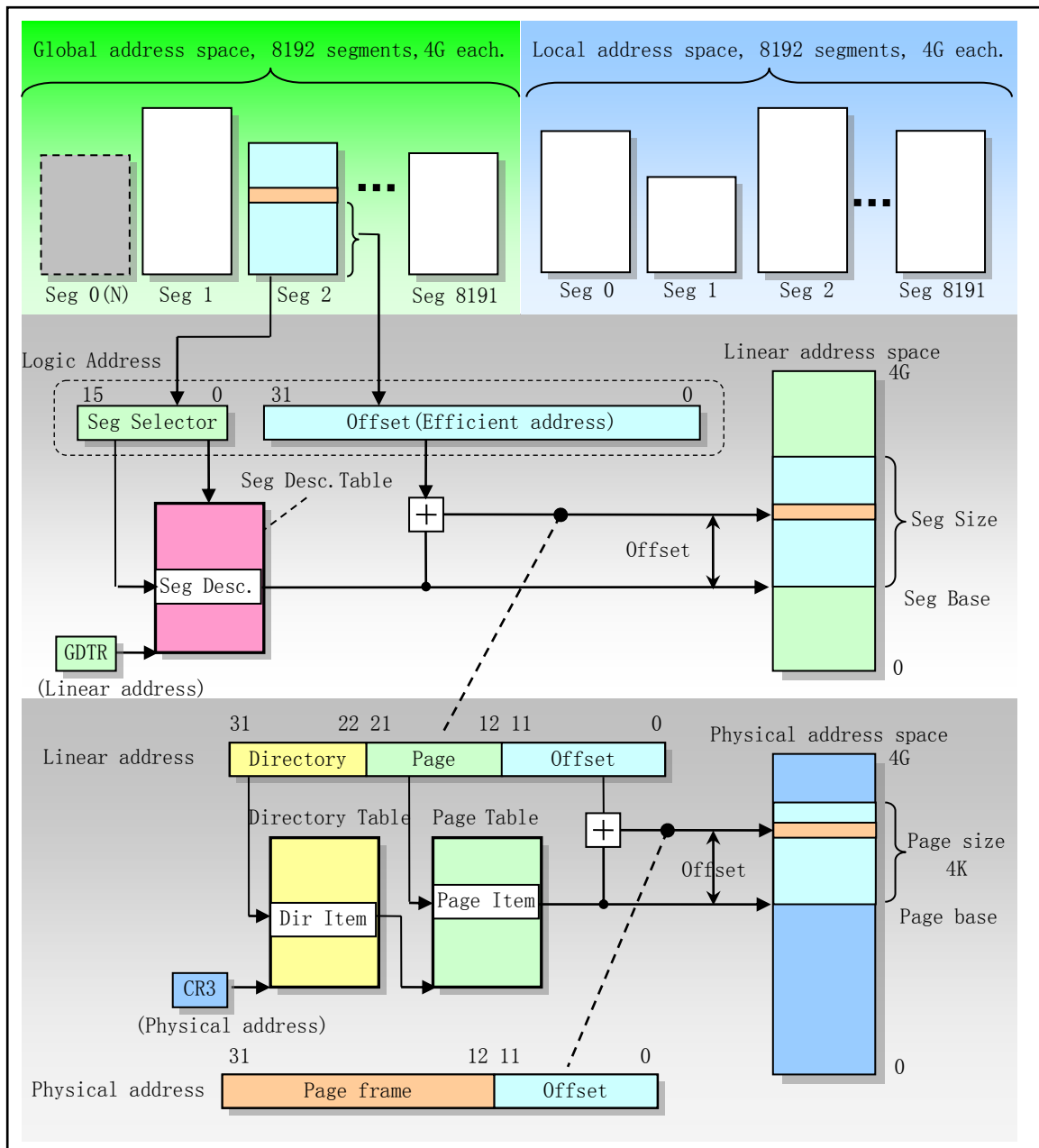


Figure 4-5 Logical, Linear, and Physical Addresses

2. Paging mechanism

Because the multi-tasking system usually defines a linear address space much larger than the physical memory it contains, it needs to use some kind of "virtualized" linear address space approach, that is, the use of virtual storage technology. Virtual storage is a memory management technology that allows programmers to create the illusion that the memory space is much larger than the actual physical memory capacity of the computer. With this illusion, we can program large programs at will without considering how much physical memory actually exists.

The paging mechanism supports virtual storage technology. In an environment using virtual storage, a large-capacity linear address space needs to be simulated using a small amount of physical memory (RAM or ROM) and some external storage space (such as a large-capacity hard disk). When paging is used, each segment is divided into pages (usually 4 KB in size per page) and the pages are stored in physical memory or on hard disk.

The operating system pays attention to these pages by maintaining a page directory and some page tables. When a program (or task) tries to access an address location in a linear address space, the processor uses the page directory and page table to convert the linear address to a physical address, and then performs the required operation at that memory location (reading Or write).

If the currently visited page is not in physical memory, the processor interrupts the execution of the program (by generating a page fault exception). The operating system can then read the page from the hard disk into physical memory and continue to execute the program that was just interrupted. When the operating system strictly implements the paging mechanism, the exchange of pages between the physical memory and the hard disk is transparent to the correctly executed program.

The 80X86 paging mechanism is best suited to support virtual storage technology. The paging mechanism uses fixed-size memory blocks, while segment management uses variable-sized blocks to manage memory. The use of fixed-size blocks for paging is more suitable for managing physical memory, whether in physical memory or on hard disks. On the other hand, the segmentation mechanism uses variable-sized blocks that are more suitable for processing logical partitions of complex systems. You can define memory cells that fit into the logical block size without being constrained by fixed-size pages. Each segment can be treated as a unit, simplifying the protection and sharing of segments.

Segmentation and paging are two different address translation mechanisms, all of which provide independent processing stages for the entire address translation operation. Although both mechanisms use a conversion table stored in memory, the table structure used is different. In fact, the segment table is stored in the linear address space, and the page table is stored in the physical address space. Therefore, the segment conversion table can be relocated by the paging mechanism without the information or cooperation of the segment mechanism. The segment conversion mechanism transforms virtual addresses (logical addresses) into linear addresses and accesses its own table in a linear address space, but it is not aware of the process of the paging mechanism converting these linear addresses to physical addresses. Similarly, the paging mechanism does not know the virtual address space where the program generates addresses. The paging mechanism simply translates linear addresses into physical addresses and accesses their own translation tables in physical memory.

4.2.3 Protection

The 80X86 supports two types of protection. One is to completely isolate each task by giving each task a different virtual address (logical address) space. The implementation principle is to provide each task with different logical address to physical address mapping. Another protection mechanism operates on tasks to protect the operating system memory segments and processor special system registers from being accessed by applications.

1. Protection between tasks

One important aspect of protection is to provide protection between applications' tasks. The method used by the 80X86 is to place each task in a different virtual address space and give each task a different mapping of logical addresses to physical addresses. The address translation function in each task is defined as the logical address in one task is mapped to a part of physical memory, and the logical address in another task is mapped to a different area in physical memory. In this way, all tasks are isolated because one task cannot generate parts of the physical memory that can be mapped to the corresponding logical addresses of other tasks. Just give each task a separate mapping table, and each task will have a different address translation function. In 80X86, each task has its own segment table and page table. When the processor switches to perform a new task, the key part of task switching is to switch to the new task's conversion table.

By arranging the same virtual-to-physical address mapping portion in all tasks and storing the operating

system in this common virtual address space portion, the operating system can be shared by all tasks. The same portion of the virtual address space that all this task has is called the Global address space. This is exactly how modern Linux operating systems use virtual address spaces.

The part of the virtual address space that is unique to each task is called the Local address space. The local address space contains private code and data that need to be distinguished from other tasks in the system. Because there is a different local address space in each task, references to the same virtual address in two different tasks will be converted to different physical addresses. This allows the operating system to give the same virtual address to each task's memory, but still isolate each task. On the other hand, all tasks' references to the same virtual address in the global address space will be translated to the same physical address. This provides support for the sharing of common code and data (such as operating systems).

2. Privilege level protection

In a task, four Privilege Levels are defined for restricting access to segments in a task based on the sensitivity of the data contained in the segment and the degree of trust of different portions of the program in the task. The most sensitive data is given the highest privilege level and they can only be accessed by the most trusted part of the task. Less sensitive data is given lower privilege levels and they can be accessed by less privileged code in the task.

The privilege level is represented by the numbers 0 to 3, with 0 having the highest privilege level and 3 being the lowest privilege level. Each memory segment is associated with a privilege level. This privilege level restricts programs with sufficient privilege level to access a segment. We know that the processor fetches and executes the instruction from the segment specified by the CS register, the current privilege level, that is, the CPL is the privilege level of the currently active code segment, and it defines the privilege level of the currently executing program. The CPL determines which segments can be accessed by the program.

Each time a program attempts to access a segment, the current privilege level is compared to the segment's privilege level to determine if there is an access permission. Programs executed at a given CPL level allow access to data segments of the same or lower level. Any references to high-level segments are illegal and raise an exception to notify the operating system.

Each privilege level has its own program stack to avoid the protection issues associated with using the shared stack. When the program is switched from one privilege level to another, the stack segment is also changed to the new level stack.

4.3 Segmentation Mechanism

The segmentation mechanism can be used to implement a variety of system designs. These designs range from flat models that only use minimum functionality of segmentations to protect programs to multi-segmented models that use segmentation to create a robust operating environment that can reliably run multiple programs (or tasks).

The simplest memory model for a system is the basic flat model, in which the operating system and programs have access to a continuous, unsegment address space. For the most part, this basic flat model hides the architecture's segmentation mechanism from system designers and application programmers. To implement a basic flat memory model, you must create at least two segment descriptors, one for the reference code segment and one for the reference data segment. However, both segments are mapped to the entire linear address space: that is, two segment descriptors have the same segment limit of 0 and 4 GB for the same base address value.

The multi-segment model can use the segmentation mechanism to provide full protection of hardware-enhanced code, data structures, programs, and tasks. In general, each program (or task) uses its own

segment descriptor table and its own segment. For programs, segments can be completely private or shared between programs. Access to all segments and each running program execution environment on the system is controlled by hardware.

Access checks can be used not only to protect references to addresses outside the boundaries of the segment, but also to prevent execution of impermissible actions in certain segments. For example, because the code segment is designed to be a read-only segment, you can use hardware to prevent writes to the code segment. The access rights information in the segment can also be used to set the protection ring or level. Protection levels can be used to protect operating system programs from unauthorized access by applications.

4.3.1 Segment definition

As mentioned in the overview of the previous section, the 80X86 provides 4GB of physical address space in protected mode. This is the address space that the processor can address on its address bus. This address space is flat and the address range is from 0 to 0xFFFFFFFF. This physical address space can be mapped to read and write memory, read-only memory, and memory-mapped I/O. The segmentation mechanism is to organize the virtual memory in the virtual address space into some variable-length memory block units called segments. The virtual address (logical address) in the 80X86 virtual address space consists of a segment portion and an offset portion. Segments are the basis for virtual address-to-linear address translation mechanisms. Each segment is defined by three parameters:

1. Base address specifies the starting address of the segment in the linear address space. The base address is a linear address and corresponds to offset 0 in the segment.
2. The segment limit is the maximum available offset within the segment in the virtual address space. It defines the length of the segment.
3. Attributes, specify the characteristics of the segment. For example, whether the segment is readable, writable, or executable as a program; the privilege level of a segment, and so on.

Segment length defines the size of the segment in the virtual address space. The segment base address and the segment limit length define the linear address range or region to which the segment is mapped. The address range from 0 to limit in the segment corresponds to the range base in the linear address to base + limit. A virtual address with an offset greater than the segment limit is meaningless and can cause an exception if used. In addition, exceptions can also result if you access a segment without permission from the segment attribute. For example, if you try to write a read-only segment, 80X86 will generate an exception. In addition, the range of multiple segments mapped into the linear address can partially overlap or cover, or even completely overlap, as shown in Figure 4-6. In the Linux 0.1x system introduced in this book, the length of the segment of a task's code segment and data segment is the same, and is mapped to the same area where the linear addresses are identical and overlap.

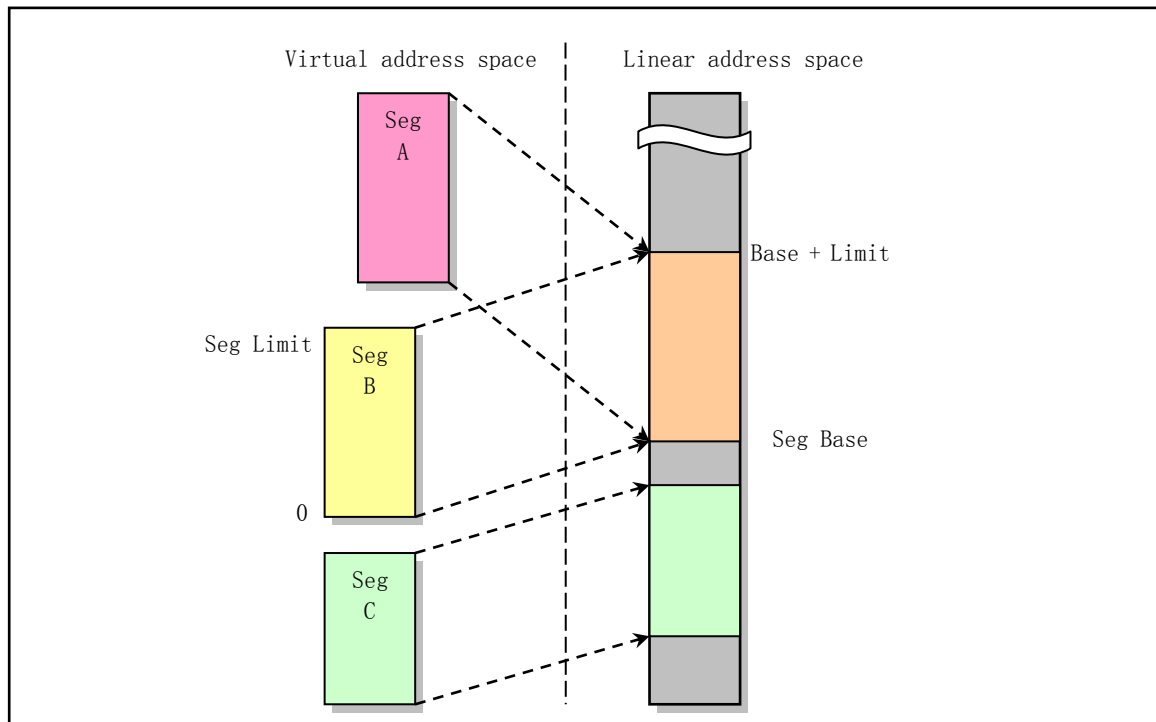


Figure 4-6 Segments in virtual map to linear address space

The base address, the limit length, and the protection attribute of a segment are stored in a structure item called a Segment Descriptor. This segment descriptor is used during the translation mapping from logical address to linear address. Segment descriptors are stored in the Descriptor table. The segment descriptor table is a simple array containing segment descriptor items. The segment selector described earlier is used to specify the corresponding segment by specifying the position of a segment descriptor in the table.

Even with the minimum functionality of the segment, each byte in the processor's address space can be accessed using the logical address. The logical address consists of a 16-bit segment selector and a 32-bit offset, as shown in Figure 4-7. The segment selector specifies the segment where the byte is located, and the offset specifies the position of the byte in the segment relative to the segment base address. The processor will convert each logical address to a linear address. A linear address is a 32-bit address in the processor's linear address space. Similar to the physical address space, the linear address space is also a flat 4GB address space with addresses ranging from 0 to 0xFFFFFFFF. The linear address space contains all the system-defined segments and system tables.

To convert the logical address into a linear address, the processor performs the following operations:

1. Use the offset value (segment index) in the segment selector to locate the corresponding segment descriptor in the GDT or LDT table. (This step is only needed if a new segment selector is loaded into a segment register.)
2. Examines the segment descriptor to check the access rights and range of the segment to insure that the segment is accessible and that the offset is within the limits of the segment.
3. Add the segment base address obtained in the segment descriptor to the offset and finally form a linear address.

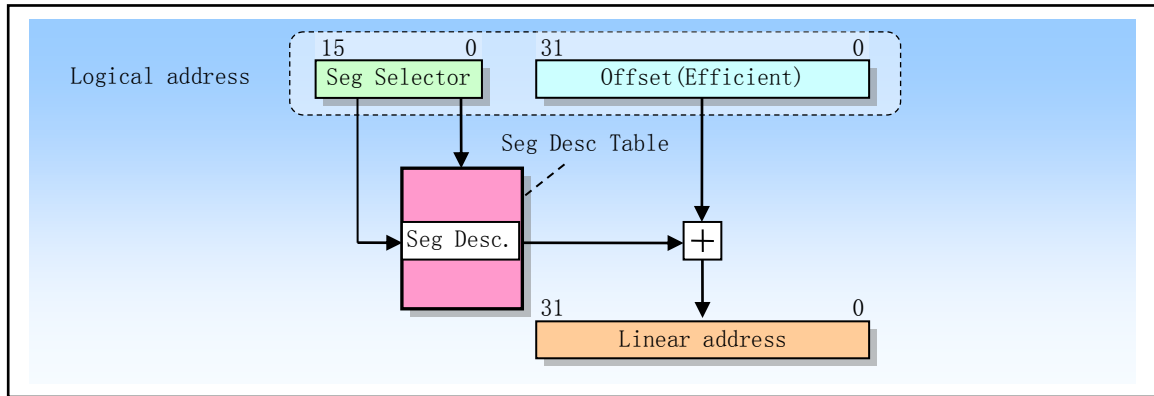


Figure 4-7 Logical address to linear address translation

If paging is not enabled, the processor directly maps the linear address to the physical address (ie, the linear address is sent to the processor's address bus). If the linear address space is paged, then a second level address translation will be used to translate the linear address into a physical address. The page conversion will be explained later.

4.3.2 Segment Descriptor Tables

The segment descriptor table is an array of segment descriptors, as shown in Figure 4-8. The descriptor table is variable in length and can contain up to 8192 8-byte descriptors. There are two descriptor tables: a global descriptor table (GDT) and a local descriptor table (LDT).

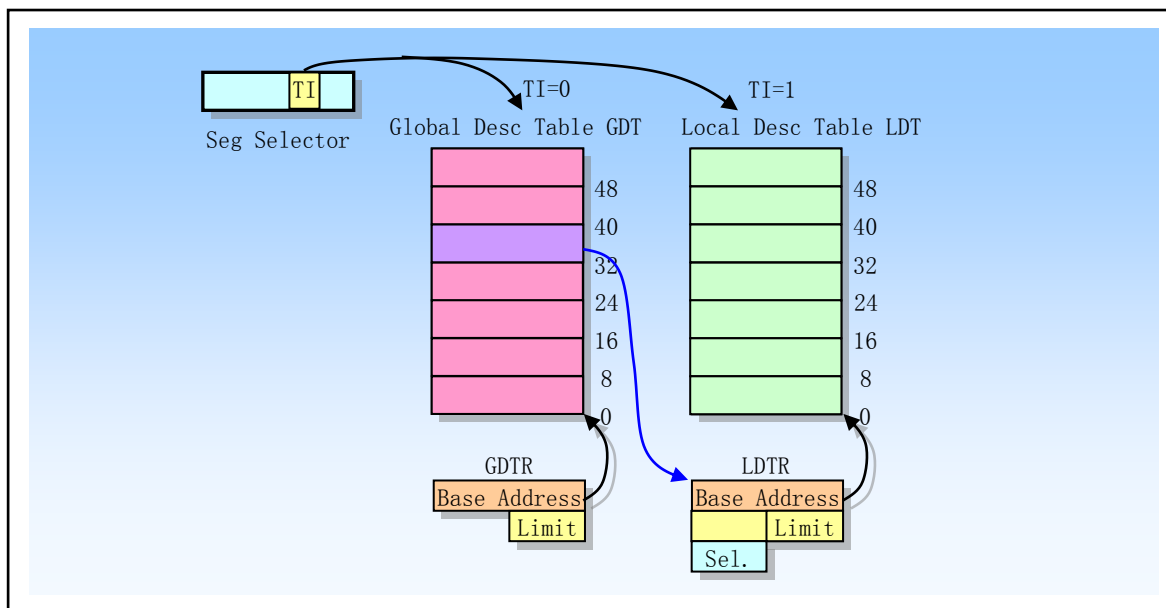


Figure 4-8 Global and Local Descriptor Tables

Each system must have one GDT that can be used for all programs and tasks in the system. Optionally, one or more LDTs may be defined. For example, an LDT can be defined for each individual task that is running, or some or all tasks can share the same LDT.

The GDT itself is not a segment; instead, it is a data structure in the linear address space. The basic linear address and limit of the GDT must be loaded into the GDTR register. The base address of the GDT should be

aligned on an eight-byte boundary to produce the best processor performance. The GDT limit is expressed in bytes. As with segmentation, limit values are added to the base address to get the address of the last valid byte. A limit of 0 results in only one valid byte. Because the segment descriptor is always 8 bytes long, the GDT limit should always be less than an integer multiple of 8 (ie, $8N-1$).

The LDT table is stored in the LDT type of system segment. At this point the GDT must contain the LDT segment descriptor. If the system supports multiple LDTs, then each LDT must have a segment descriptor and segment selector in the GDT. An LDT segment descriptor can be stored anywhere in the GDT table.

Accessing LDT requires its segment selector. In order to reduce the number of address conversions when accessing the LDT, the LDT's segment selector, base address, segment length, and access rights need to be stored in the LDTR register.

When the GDTR register is stored (using the SGDT instruction), a 48-bit "pseudo-descriptor" is stored in memory. In order to avoid alignment check errors in user mode (privilege level 3), dummy descriptors should be stored at an odd word address (ie, address MOD 4 = 2). This will cause the processor to store an aligned word first, followed by an aligned doubleword (at the 4-byte alignment). User-mode programs usually do not save dummy descriptors, but you can use this alignment to avoid the possibility of an alignment check error. The same alignment is also used when using the SIDT instruction to save the IDTR register contents. However, when saving LDTR or task registers (using SLTR or STR instructions, respectively), dummy descriptors should be stored at double-word aligned addresses (ie, address MOD 4 = 0).

Descriptor tables are stored in special data structures maintained by the operating system and referenced by the processor's memory management hardware. These special structures should be stored in a protected memory area that is only accessible by the operating system software to prevent the application from modifying the address translation information in it. The virtual (logical) address space is divided into two halves of equal size. Half is mapped to a linear address by the GDT and the other half is mapped by the LDT. The entire virtual address space contains 2^{14} segments: half of the space (that is, 2^{13} segments) is a global virtual address space mapped by GDT, and the other half is a local virtual address space mapped by LDT. By specifying a descriptor table (GDT or LDT) and description symbols in the table, we can locate a descriptor.

When a task switch occurs, the LDT will be replaced with a new task LDT, but the GDT will not change. Therefore, half of the virtual address space mapped by the GDT is common to all the tasks in the system, but the other half of the LDT mapping is changed when the task is switched. The segments shared by all tasks in the system are mapped by the GDT. Such segments typically include a section containing the operating system and each task's own special section containing LDTs. The LDT segment can be thought of as data belonging to the operating system.

The LDT contains descriptors for segments that are dedicated to a single task. Several tasks can share a common LDT. In this case, all these tasks can use the same set of segments because they have the same LDT, and all tasks share one GDT. Both tasks can also share a segment descriptor in their respective LDTs so that sharing a segment without having to put its descriptor in the GDT is shared for all tasks. In this case, the shared segment must be handled exclusively by the operating system because it has two descriptors in two different LDTs and must be updated together.

Figure 4-9 shows how the segments in a task can be separated between GDT and LDT. There are six segments in the figure for the two applications (A and B) and the operating system. Each application in the system corresponds to a task, and each task has its own LDT. Application A runs in Task A and has LDT_A to map segments Code_A and Data_A. Similarly, application B runs in task B and uses LDT_B to map the Code_B and Data_B segments. The two segments containing the operating system kernel, Code_{OS} and Data_{OS}, are mapped using the

GDT so that they can be shared by both tasks. Two LDT segments: LDT_A and LDT_B are also mapped using GDT.

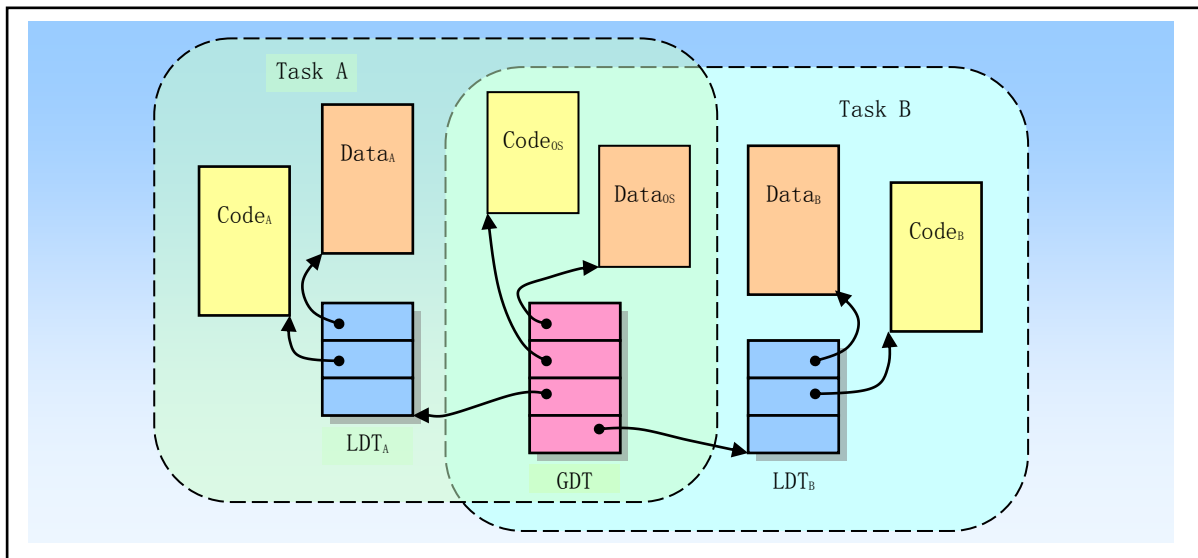


Figure 4-9 The segment types used by tasks

When task A is running, accessible segments include the $Code_A$ and $Data_A$ segments of the LDT_A map, plus the $Code_{OS}$ and $Data_{OS}$ segments of the GDT mapped operating system. When task B is running, the accessible segments include the LDT_B mapped $Code_B$ and $Data_B$ segments plus the GDT mapped segments.

This example demonstrates how virtual address space can be organized to isolate each task by having each task use a different LDT. When task A is running, the segment of task B is not part of the virtual address space, so task A cannot access task B's memory. Similarly, when task B is running, the segment of task A cannot be addressed. This method of isolating each application task using LDT is one of the key protection needs.

4.3.3 Segment Selectors

The segment selector is a 16-bit identifier for the segment, as shown in Figure 4-10. The segment selector does not point directly at the segment, but instead points to the segment descriptor that defines the segment in the segment descriptor table. The segment selector consists of 3 fields and the contents are as follows:

- Requested Privilege Level (RPL);
- Table Index (TI);
- Index.

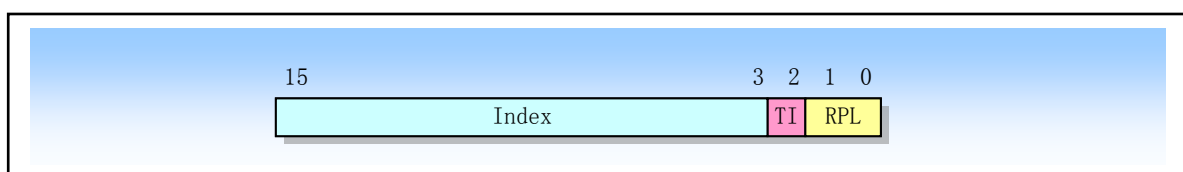


Figure 4-10 Segment selector structure

The request privilege level RPL provides segment protection information, which will be described later in detail. The table index TI is used to indicate the segment descriptor table GDT or LDT containing the specified

segment descriptor. TI=0 indicates that the descriptor is in the GDT; TI=1 indicates that the descriptor is in the LDT. The index field gives the index number of the descriptor in the GDT or LDT table. It can be seen that the selector specifies a segment by locating a descriptor in the segment table, and the descriptor contains all the information for accessing a segment, such as the base address of the segment, segment length, and segment attributes.

For example, the selector (0x08) in Figure 4-11(a) specifies a segment 1 with RPL=0 in the GDT. The index field value is 1, the TI bit is 0, and the GDT table is specified. The selector (0x10) in Figure 4-11(b) specifies the segment 2 with RPL=0 in the GDT. The index field value is 2, the TI bit is 0, and the GDT table is specified. The selector (0x0f) in Figure 4-11(c) specifies segment 1 with LPL=3 in the LDT. Its index field value is 1, the TI bit is 1, and the LDT table is specified. The selector (0x17) in Figure 4-11(d) specifies segment 2 with LPL=3 in the LDT. The index field value is 2, the TI bit is 1, and the LDT table is specified. In fact, the first four selectors in Figure 4-11: (a), (b), (c), and (d) are the kernel code snippets, kernel data snippets, task code snippets, and tasks of the Linux 0.1x kernel, respectively. The data segment selector. The selector (0xffff) in Figure 4-11(e) specifies segment 8191 with RPL=3 in the LDT table. Its index field value is 0b111111111111 (that is, 8191), the TI bit is equal to 1, and the LDT table is specified.

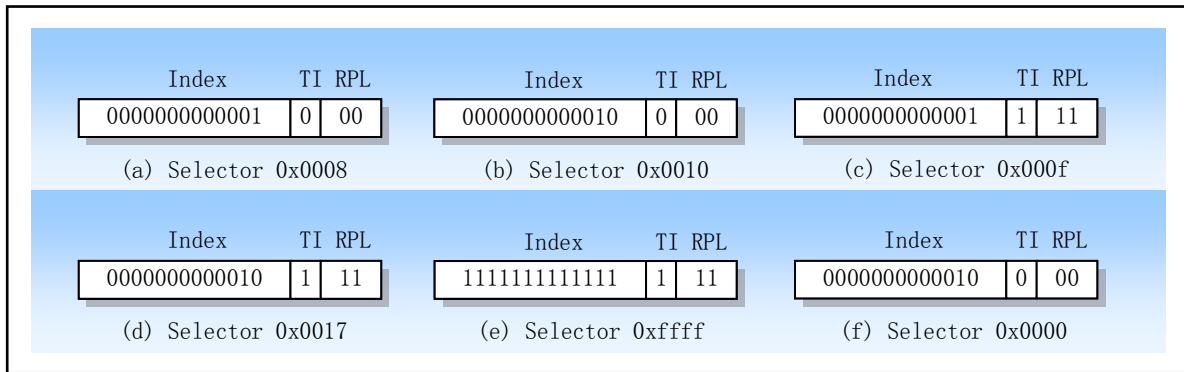


Figure 4-11 Segment selector examples

In addition, the processor does not use the first item in the GDT table. The selector to the GDT entry (that is, the index whose index value is 0 and the TI flag is 0) is used as a "null selector", as shown in Figure 4-11(f). When an empty selector is loaded into a segment register (other than CS and SS), the processor does not generate an exception. However, an exception occurs when using a segment register containing an empty selector to access memory. An exception will be caused when an empty selector is loaded into the CS or SS segment register.

The segment selector is visible to the application as part of the pointer variable, but the value of the selector is usually set or modified by the link editor or link loader, not the application. To reduce address translation time and programming complexity, the processor provides registers that hold up to six segment selectors (see Figure 4-12), that is, segment registers. Each segment register supports a specific type of memory reference (code, data, or stack). In principle, each program must at least load valid segment selectors into the code segment (CS), data segment (DS), and stack segment (SS) registers. The processor additionally provides three auxiliary data segment registers (ES, FS, and GS) that can be used to allow the currently executing program (or task) to access several other data segments.

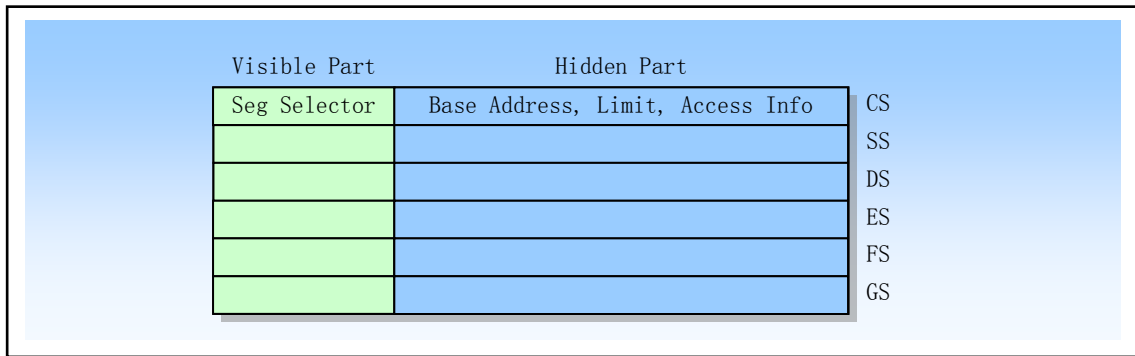


Figure 4-12 Segment register structure

For programs that access a segment, the segment selector must have been loaded into a segment register. Therefore, although a system can define many segments, only 6 segments are available for immediate access at the same time. To access other segments you need to load selectors for these segments.

In addition, to avoid reading the descriptor table every time you access the memory, to read and decode a segment descriptor, each segment register has a "visible" part and a "hidden" part (the hidden part is also called "Descriptor Buffer" or "Shadow Register"). When a segment selector is loaded into the visible portion of a segment register, the processor also loads the segment address, segment length, and access control information in the segment descriptor pointed to by the segment selector to the hidden portion of the segment register. The information buffered in the segment registers (visible and hidden portions) allows the processor to no longer spend time reading the base address and limit value from the segment descriptor when performing address translation.

Since the shadow register contains a copy of the descriptor information, the operating system must ensure that changes to the descriptor table should be reflected in the shadow register. Otherwise, the base address or limit of a segment in the descriptor table is modified, but the changes are not reflected in the shadow register. The simplest way to deal with this kind of problem is to reload 6 segment registers immediately after making any changes to the descriptors in the descriptor table. This will reload the corresponding segment information in the descriptor table into the shadow register.

There are two types of load instructions for loading segment registers:

1. Like MOV, POP, LDS, LES, LSS, LGS and LFS instructions. These instructions explicitly reference the segment register directly;
2. Implicitly loaded instructions such as CALL, JMP, and RET instructions using long pointers, IRET, INTn, INTO, and INT3 instructions. These instructions are accompanied by changes to the contents of the CS register (and some other segment registers) during operation.

The MOV instruction can of course also be used to store the contents of the visible part of the segment register in a general-purpose register.

4.3.4 Segment Descriptor

Earlier we explained that using a segment selector to locate a descriptor in the descriptor table. The segment descriptor is a data structure item in the GDT and LDT tables used to provide the processor with information about the position and size of a segment and the status of access control. Each segment descriptor is 8 bytes in length and contains three main fields: segment base address, segment length, and segment attributes. Segment descriptors are usually created by the compiler, linker, loader, or operating system, but are by no means applications. Figure 4-13 shows the general format of all types of segment descriptors.

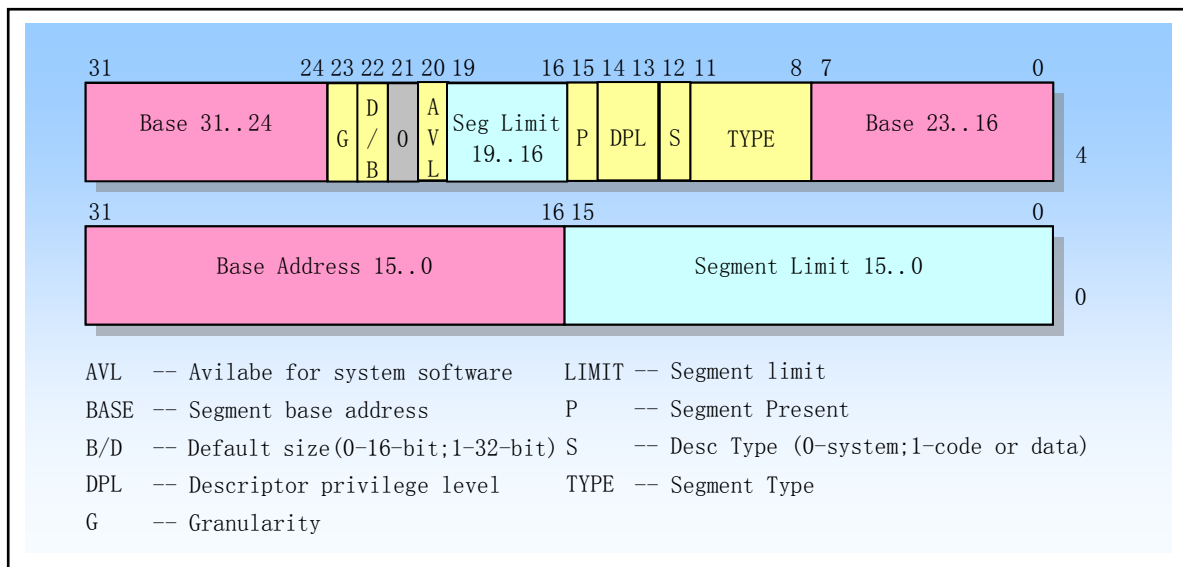


Figure 4-13 General format of segment descriptor

The meaning of fields and flags in a segment descriptor is as follows:

◆ Segment limit field (LIMIT)

The segment limit field is used to specify the size of the segment. The processor will combine the two segment limit fields in the segment descriptor into a 20-bit value, and specify the actual meaning of the segment limit length Limit value according to the granularity flag G. If G=0, the segment length Limit range can be from 1 byte to 1 MB byte in units of bytes. If G=1, the segment length Limit ranges from 4KB to 4GB and the unit is 4KB.

Depending on the segment extension direction flag E in the segment type, the processor uses the segment limit Length in two different ways. For an upward-extending segment (abbreviated as an upper-extending segment), the offset value in the logical address can range from 0 to the segment-limit value Limit. An offset greater than the limit length limit will produce a general protective exception. For the segment that is extended downward (abbreviated as the lower segment), the meaning of the segment limit Limit is reversed. Depending on the setting of the default stack pointer size flag B, the offset value can range from the segment limit length to 0xFFFFFFFF or 0xFFFF. An offset value less than the limit length Limit will produce a general protective exception. For the next expanded segment, reducing the value in the segment limit field allocates new memory at the bottom of the segment address space instead of at the top. The 80X86's stack is always scaled down, so this implementation is well suited to extending the stack.

◆ Base address field (BASE)

This field defines the location of a segment of byte 0 in the 4GB linear address space. The processor will combine 3 separate base address fields to form a 32-bit value. The segment base address should be aligned to a 16-byte boundary. Although this is not required, the best performance of the program can be achieved by aligning the code and data segments of the program on a 16-byte boundary.

◆ Type field (TYPE)

The type field specifies the type of the segment or gate, the type of access to describe the segment, and the extension direction of the segment. The interpretation of this field depends on the descriptor type flag S, indicating whether it is an application (code or data) descriptor or a system descriptor. The

encoding of the TYPE field is different for code, data, or system descriptors, as shown in Figure 4-14.

◆ Descriptor type flag (S)

The descriptor type flag S indicates whether a segment descriptor is a system segment descriptor (when S=0) or a code or data segment descriptor (when S=1).

◆ Descriptor privilege level (DPL)

The DPL field indicates the privilege level of the descriptor. The privilege level ranges from 0 to 3. The 0 privilege level is the highest and the 3 level is the lowest. DPL is used to control access to segments.

◆ Segment present (P)

The segment presence flag P indicates whether a segment is in memory (P=1) or not in memory (P=0). When the P flag of a segment descriptor is 0, then loading the selector pointing to this segment descriptor into the segment register will result in the generation of a segment without an exception. Memory management software can use this flag to control the actual need to load that segment into memory at a given time. This feature provides virtual storage with control beyond the paging mechanism. Figure 4-15 shows the segment descriptor format when P=0. When the P flag is 0, the operating system is free to save its own data using fields that are marked as Available in the format, such as information about where the segment actually does not exist.

◆ Default operation size/default stack pointer size and/or upper bound (D/B)

According to the segment descriptor describes an executable code segment, spread data segment or a stack segment, this mark has a different function. (For 32-bit code and data segments, this flag should always be set to 1; for 16-bit code and data segments, this flag is set to 0.)

- **Executable code segment.** This flag is called the D flag at this time and is used to indicate that the instruction in this segment refers to a valid address and the default length of the operand. If the flag is set, the default value is a 32-bit address and a 32-bit or 8-bit operand; if the flag is 0, the default value is a 16-bit address and a 16-bit or 8-bit operand. The instruction prefix 0x66 can be used to select a non-default operand size; the prefix 0x67 can be used to select an address size other than the default.
- **Stack segment (data segment pointed to by the SS register).** At this point, this flag is called the B (Big) flag and indicates the size of the stack pointer when an implicit stack operation (such as PUSH, POP, or CALL) occurs. If this flag is set, the 32-bit stack pointer is used and stored in the ESP register; if the flag is 0, the 16-bit stack pointer is used and stored in the SP register. If the stack segment is set to a lower extended data segment, this B flag also specifies the upper bound of the stack segment.
- **Expand the data segment.** At this point the flag is called the B flag and is used to indicate the upper limit of the segment. If this flag is set, the upper bound of the stack segment is 0xFFFFFFFF (4GB); if this flag is not set, the upper bound of the stack segment is 0xFFFF (64KB).

◆ Granularity (G)

This field is used to determine the unit of the segment-limited field value. If the granularity flag is 0, the unit of the segment limit value is bytes; if the granularity flag is set, the segment limit value uses 4 KB units. (This flag does not affect the granularity of the segment's base address. The base address's granularity is always in bytes.). If the G flag is set, the 12-bit least significant bit of the offset value is not checked when the segment length is used to check the offset value. For example, when G=1, the segment limit length of 0 indicates that the effective offset value is 0 to 4095.

◆ Available and reserved bits

Bit 20 of the second double word of the segment descriptor is available for use by system software; Bit

21 is a reserved bit and should always be set to 0.

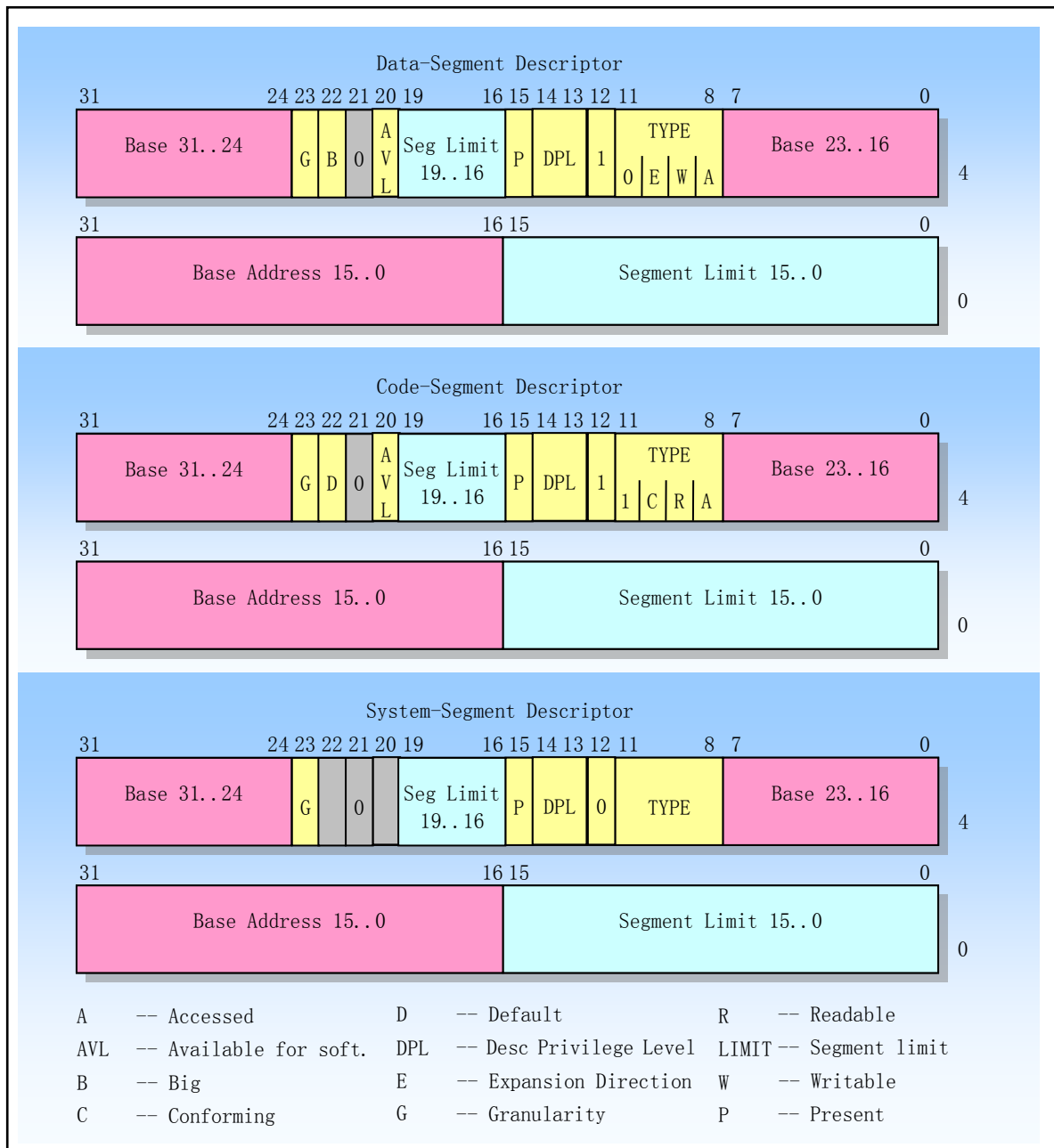


Figure 4-14 Code, data, and system segment descriptors formats

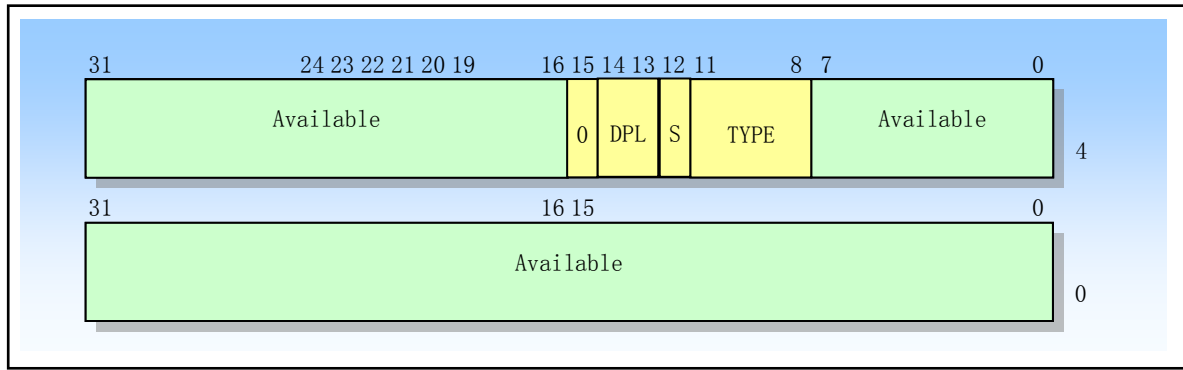


Figure 4-15 Segment Descriptor When bit P=0

4.3.5 Code and Data Segment Descriptor Types

When the S (descriptor type) flag is set in a segment descriptor, the descriptor is used for the code or data segment. At this point, the most significant bit in the type field (bit 11 of the second double word) is used to determine whether it is a data segment descriptor (reset) or a code segment descriptor (set).

For data segments, the lower 3 bits (bits 8, 9, 10) of the type field are used to indicate Accessed, Write-enable, and Expansion-direction, respectively. See Table 4-3 for descriptions of bit fields in the code and data segment type fields. According to the writable bit W setting, a data segment can be read-only or readable and writable.

Table 4-3 Code and Data Segment Descriptor Types

TYPE Field					Descriptor Type	Description
Decimal	11	10	9	8		
		E	W	A		
0	0	0	0	0	Data	Read-Only
1	0	0	0	1	Data	Read-Only, accessed
2	0	0	1	0	Data	Read/Write
3	0	0	1	1	Data	Read/Write, accessed
4	0	1	0	0	Data	Expand-down, Read-Only.
5	0	1	0	1	Data	Expand-down, Read-Only, accessed
6	0	1	1	0	Data	Expand-down, Read/Write
7	0	1	1	1	Data	Expand-down, Read/Write, accessed
		C	R	A		
8	1	0	0	0	Code	Execute-Only
9	1	0	0	1	Code	Execute-Only, accessed
10	1	0	1	0	Code	Execute/Read
11	1	0	1	1	Code	Execute/Read, accessed
12	1	1	0	0	Code	Conforming, Execute-Only
13	1	1	0	1	Code	Conforming, Execute-Only, accessed
14	1	1	1	0	Code	Conforming, Execute/Read-Only
15	1	1	1	1	Code	Conforming, Execute/Read-Only, accessed

Stack segment must be a read/write data segment. If a non-writable data segment selector is loaded into the

SS register, a general protection exception will result. If the length of the stack segment needs to change dynamically, the stack segment can be a data segment that is extended downwards (the expansion direction flag is set). Here, dynamically changing the segment limit will cause the stack space to be added to the bottom of the stack.

The accessed bit indicates whether a segment has been accessed since the last time the operating system resets the bit. Each time the processor loads a segment selector into the segment register, it sets this bit. This bit needs to be explicitly cleared, otherwise it remains set. This bit can be used for virtual memory management and debugging.

For code segments, the lower 3 bits of the type field are interpreted as Accessed, Read-enable, and Conforming. Depending on the setting of the readable R flag, the code segment can be execute-only or execute/read. An executable/readable code segment can be used when constants or other static data and instruction code are placed in a ROM. Here, we can read the data in the code segment by using an instruction with a CS override prefix or by loading the code segment selector for the code segment into a data segment register (DS, ES, FS, or GS). In protected mode, the code segments are not writable.

Code segments can be conforming or non-conforming. A transfer of execution control to higher privilege level conforming segment allows the program to continue execution at the current privilege level. A transfer to a non-conforming segment with a different privilege level will result in a general protection exception, unless a call gate or task gate is used- (for more information on consistent and non-conforming code segments, see "Directly Invoking or Jumping to Code segment"). System tools that do not access the protection facility and some exception types (such as errors, overflows) can be stored in the conforming segments. Tools that need to prevent access by low-privilege programs or procedures should be stored in non-conforming segments. Note that execution cannot be transferred by a call or a jump to a less-privileged (numerically higher privilege level) code segment, regardless of whether the target segment is a conforming or nonconforming code segment.

All data segments are non-conforming, meaning that they cannot be accessed by less privileged programs or procedures. However, unlike code segments, data segments can be accessed by higher privileged programs or procedures without the use of special access gates.

If a segment descriptor in the GDT or LDT is stored in the ROM, the processor will enter an infinite loop if the software or processor attempts to update (write) the segment descriptor in ROM. In order to prevent this problem, the accessed bit of all descriptors that need to be stored in the ROM should be set to the pre-set state. At the same time, delete any code in the operating system that attempts to modify the ROM segment descriptor.

4.3.6 System Descriptor Types

When the S flag (descriptor type) in a segment descriptor is in a reset state (0), the descriptor type is a system descriptor. The processor can recognize the following types of system descriptors:

- Local Descriptor Table (LDT) segment descriptor;
- Task-state segment (TSS) descriptor;
- Call-gate descriptor;
- Interrupt-gate descriptor;
- Trap-gate descriptor;
- Task-gate descriptor.

These descriptor types can be divided into two major categories: system segment descriptors and gate descriptors. The system segment descriptor points to the system segment (such as LDT and TSS segment). The gate descriptor is a "gate". For the call, interrupt or trap gate, it contains the selector of the code segment and the pointer of the program entry point in the segment; for the task gate, it contains the TSS segment selector. Table 4-4

shows the encoding of the system segment descriptor and gate descriptor type fields.

Table 4-4 System-Segment and Gate-Descriptor Types

TYPE Field					Description
Decimal	11	10	9	8	
0	0	0	0	0	Reserved
1	0	0	0	1	16-Bit TSS (Available)
2	0	0	1	0	LDT
3	0	0	1	1	16-Bit TSS (Busy)
4	0	1	0	0	16-Bit Call Gate
5	0	1	0	1	Task Gate
6	0	1	1	0	16-Bit Interrupt Gate
7	0	1	1	1	16-Bit Trap Gate
8	1	0	0	0	Reserved
9	1	0	0	1	32-Bit TSS (Available)
10	1	0	1	0	Reserved
11	1	0	1	1	32-Bit TSS (Busy)
12	1	1	0	0	32-Bit Call gate
13	1	1	0	1	Reserved
14	1	1	1	0	32-Bit Interrupt Gate
15	1	1	1	1	32-Bit Trap Gate

The use of TSS status segments and task gates will be explained in the task management section. The use of call gates will be described in the section on protection. The use of interrupts and trap gates will be used in interrupt and exception handling. Instructions are given in the section.

4.4 Paging

The paging mechanism is the second part of the 80X86 memory management mechanism. It completes the process of virtual (logical) address to physical address translation based on the segmentation mechanism. The segmentation mechanism translates logical addresses into linear addresses, while paging converts linear addresses into Physical addresses. Pagination can be used for any kind of segmented model. The processor paging mechanism divides the linear address space into which segments have been mapped, and these linear address space pages are then mapped to pages in the physical address space. Several page-level protection measures of the paging mechanism can be used in conjunction with the segment protection mechanism or replace the protection measures of the segmentation mechanism. For example, read/write protection can be enhanced on a page-based basis. In addition, on the page unit, the paging mechanism also provides user-superuser two-level protection.

The paging mechanism can be enabled by setting the PG bit in control register CR0. If PG=1, paging is enabled and the processor will use the mechanism described in this section to translate the linear address into a physical address. If PG=0, the paging mechanism is disabled, and the linear address generated by the segmentation mechanism is directly used as a physical address.

The previously described segmentation mechanism operates on various variable size memory regions. Unlike fragmentation, paging mechanisms operate on fixed-size blocks of memory (called pages). The paging mechanism divides the linear and physical address spaces into pages. Any page in the linear address space can be mapped to

any page in the physical address space. Figure 4-16 shows how the paging mechanism divides the linear and physical address spaces into pages and provides an arbitrary mapping between these two spaces. The arrows in the figure correspond the pages in the linear address space to the pages in the physical address space.

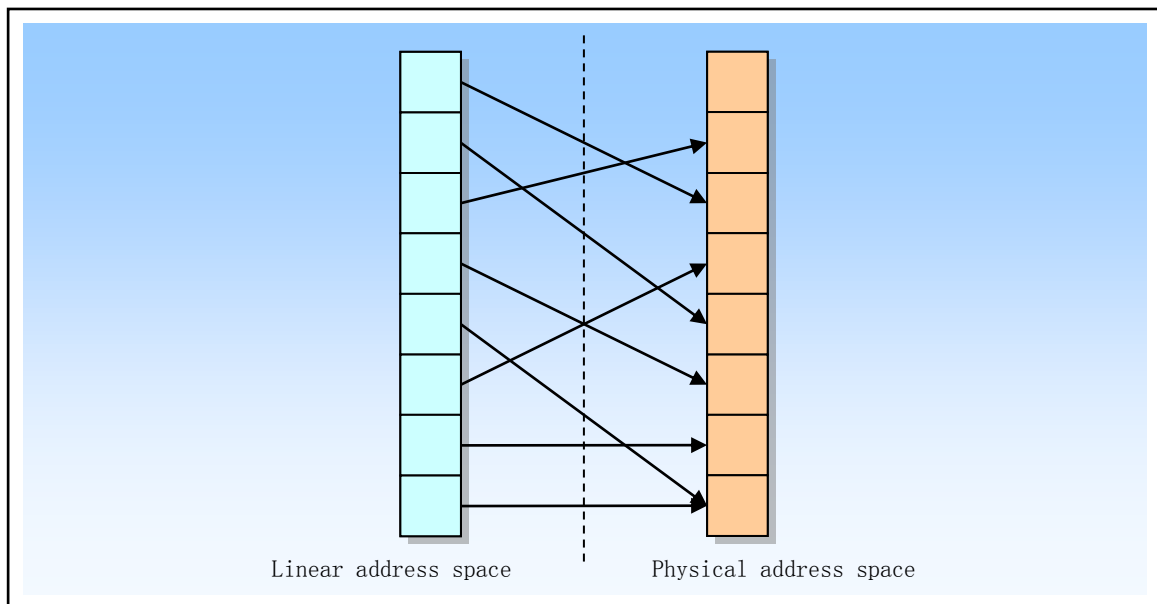


Figure 4-16 Illustration of correspondence between linear and physical address space pages

The 80X86 uses 4K (2^{12}) bytes of fixed-size pages and is aligned at the 4K address boundary. This means that the paging mechanism divides 2^{32} bytes (4GB) of linear address space into 2^{20} ($1\text{M} = 1048576$) pages. The paging mechanism operates by relocating pages in a linear address space into physical address space. Since a 4K page is mapped as a unit and aligned to a 4K boundary, the lower 12 bits of the linear address can be used as the in-page offset directly as the lower 12 bits of the physical address. The relocation function performed by the paging mechanism can be seen as converting the upper 20 bits of the linear address to the upper 20 bits of the corresponding physical address.

When paging is used, the processor divides the linear address space into fixed-size pages (length 4KB) that can be mapped into physical memory and/or disk storage. When a program (or task) references a logical address in memory, the processor translates the logical address into a linear address and then uses a paging mechanism to translate the linear address into a corresponding physical address. If a page containing a linear address is not currently in physical memory, the processor generates a page fault exception. A page fault exception handler usually causes the operating system to load the corresponding page into physical memory from disk (it may also write different pages in physical memory to disk during operation). After the page is loaded into physical memory, the return from the exception handler causes the instruction that caused the exception to be re-executed. The information used by the processor to translate linear addresses into physical addresses and to generate page fault exceptions (if necessary) is contained in page directories and page tables stored in memory.

The biggest difference between paging and segmentation is that paging uses fixed-length pages. If you only use segmented address translation, a data structure stored in physical memory will contain all of its parts. However, if pagination is used, one data structure can be stored in physical memory in part and another part in disk.

In order to reduce the number of bus cycles required for address translation, the most recently accessed page directory and page table are stored in a processor buffer, known as the Translation Lookaside Buffer (TLB). The

TLB can satisfy most read page directories and page table requests without using bus cycles. Only when the TLB does not contain the required page table entries will an extra bus cycle be used to read the page table entries from memory. This usually occurs when a page table entry has not been accessed for a long time.

4.4.1 Page Table Structure

The paging translation is described by a table that resides in memory. This table is called a page table and is stored in a physical address space. The page table can be seen as a simple array with 2^{20} items. The linear to physical address mapping function can simply be seen as an array search. The upper 20 bits of the linear address form the index of this array and is used to select the physical (base) address of the corresponding page. The lower 12 bits of the linear address give the offset in the page, plus the base address of the page eventually forms the corresponding physical address. Since the page base address is aligned on the 4K boundary, the bottom 12 bits of the page base address must be 0. This means that the 20-bit page base address and the 12-bit offset connection are combined to get the corresponding physical address.

Each page table entry in the page table has a size of 32 bits. Since only 20 bits are needed to store the physical base address of the page, the remaining 12 bits can be used to store attribute information such as whether the page exists or not. If the linear address index page table entry is marked as existing, then the item is valid, we can get the physical address of the page. If the item indicates that it does not exist, an exception will be generated when accessing the corresponding physical page.

4.4.1.1 Two-Level Page Table Structure

A page table contains 2^{20} (1M) entries, each occupying 4 bytes (32 bits). If they are only stored as one table, they will occupy up to 4MB of memory. Therefore, in order to reduce the memory footprint, the 80X86 uses two levels of tables. Thus, the conversion of a high 20-bit linear address to a physical address is also performed in two steps, using 10 bits per step.

The first level table is called a page directory. Occupies one page with 2^{10} (1K) entries of 4 bytes in length. These entries point to the corresponding secondary table. The top 10 bits (bits 31 - 22) of the linear address are used as index values in the primary table (page directory) to select one of the 2^{10} secondary tables.

The second level table is called a page table. Its length is also a page, and it contains at most 1K 4-byte entries. Each 4-byte table entry contains the 20-bit physical base address of the associated page. The secondary page table uses the middle 10 bits of the linear address (bits 21--12) as the index of the entry to obtain the entry containing the 20-bit physical base address of the page. The 20-bit page physical base address and the lower 12 bits (in-page offset) in the linear address are combined to obtain the output value of the page conversion process, ie, the corresponding final physical address.

Figure 4-17 shows the two-level table lookup process. The CR3 register specifies the base address of the page directory table. The upper 10 bits of the linear address are used to index this page directory table to obtain a pointer to the associated second-level page table. The central 10 bits of the linear address are used to index the secondary page table to obtain the upper 20 bits of the physical address. The lower 12 bits of the linear address are directly used as the physical address low 12 bits to form a complete 32-bit physical address.

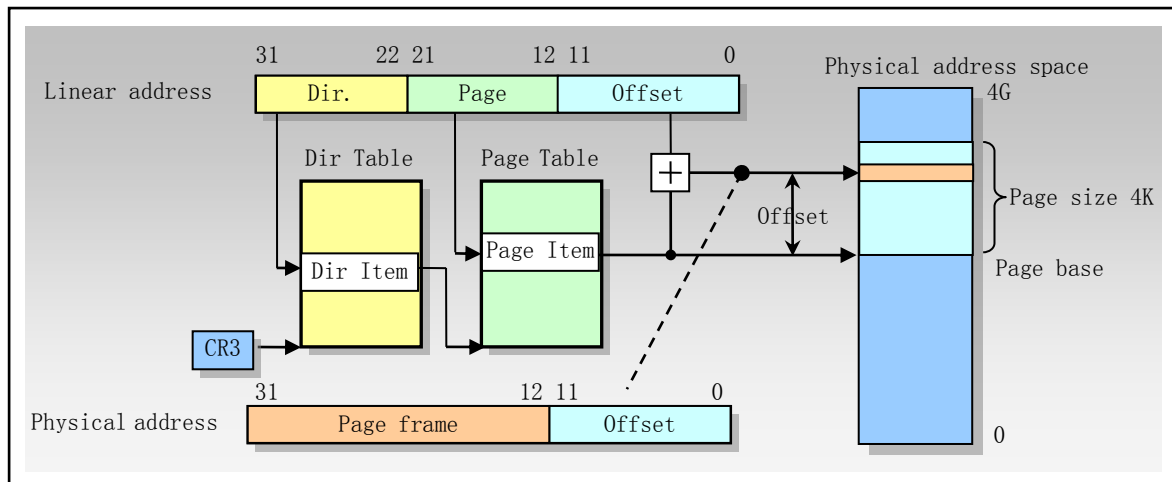


Figure 4-17 Linear Address Translation

4.4.1.2 Nonexistent Page Tables

By using the two-level table structure, page tables are allowed to be spread across the pages of memory without being stored in consecutive 4MB memory blocks. In addition, there is no need to allocate secondary page tables for portions that are not present or unused in the linear address space. Although directory pages must always exist in physical memory, secondary page tables can be redistributed as needed. This makes the size of the page table structure correspond to the actual use of the linear address space size.

Each table entry in the page directory table also has a present attribute, similar to the table entry in the page table. The presence attribute in the page directory entry indicates whether the corresponding secondary page table exists. If the directory entry indicates that the corresponding secondary page table exists, then by accessing the secondary table, the second step of the table lookup process will continue as described above. If there is a bit indicating that the corresponding secondary table does not exist, the processor will generate an exception to notify the operating system. The presence attribute in the page directory entries allows the operating system to allocate secondary page table pages based on the actual linear address range used.

The presence bits in the page directory entries can also be used to store secondary page tables in virtual memory. This means that only part of the secondary page table needs to be stored in physical memory at any time, while the rest can be stored on disk. The page directory entries corresponding to the page tables in physical memory will be marked as present to indicate that they can be paged. The page directory entry for the page table on disk will be marked as not present. An exception caused by the absence of a secondary page table informs the operating system to load the missing page table from disk into physical memory. Storing page tables in virtual memory reduces the amount of physical memory needed to save the paging translation tables.

4.4.2 Page-Directory and Page-Table Entries

The format of the page directory entry and page table entry is shown in Figure 4-18. The bits 31--12 contain the upper 20 bits of the physical address and are used to locate the physical base address of a page (also called a page frame) in the physical address space. The lower 12 bits of the table entry contain page attribute information. We have already discussed the existence attributes. Here we briefly describe the functions and uses of the remaining attributes.

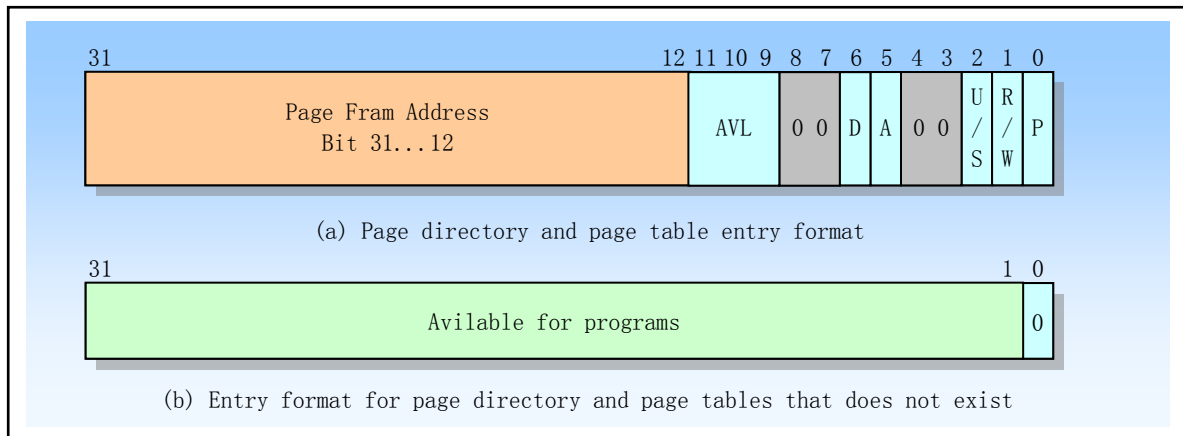


Figure 4-18 Page directory and page table entry format

- P – Bit 0 is Present flag. It indicates whether the page or page table pointed to by the table entry is currently loaded into physical memory. After the flag is set, it indicates that the page is in physical memory and performs address translation. When the flag is cleared, it means that the page is not in memory. If the processor tries to access the page, a page fault exception will be generated. At this point the operating system can use the rest of the entry to store information such as the location of the page in the disk system.
- R/W -- Bit 1 is the Read/Write flag. If set to 1, it means the page can be read, written or executed. If 0, the page is read-only or executable. The R/W bit has no effect when the processor is running at the superuser privilege level (level 0, 1 or 2), refer to the U/S flag below. The R/W bit in the page directory entry acts on all pages it maps to.
- U/S -- Bit 2 is the User/Supervisor flag. If set to 1, then the program running on any privilege level can access the page. If 0, the page can only be accessed by programs running on the superuser privilege level (0, 1, or 2). The U/S bit in the page directory entry acts on all pages it maps to.
- A -- Bit 5 is the Accessed flag. This flag of the page table entry is set to 1 when the processor accesses the page mapped by the page table entry. This flag of the page directory entry is set to 1 when the processor accesses any page mapped by the page directory entry. The processor is only responsible for setting this flag, and the operating system can count the usage of the page by periodically resetting the flag.
- D -- Bit 6 is the page modified (Dirty) flag. When the processor performs a write operation on a page, the D flag corresponding to the page table entry is set. The processor does not modify the D flag in the page directory entry.
- AVL – Available field. This field is reserved for program use. The processor will not modify these bits, and the future upgrade processor will not.

4.4.3 Virtual Memory

The presence flag P in the page directory and page table entries provides the necessary support for virtual storage using paging technology. If the page in the linear address space exists in the physical memory, the flag P=1 in the corresponding entry and the corresponding physical address is included in the entry. A table whose page is not in physical memory has its flag P = 0. If the program accesses a page that does not exist in physical memory, the processor generates a page fault exception. At this point, the operating system can use this exception handling process to transfer the missing pages from the disk to the physical memory, and store the corresponding physical address in the table entry. Finally, the flag P=1 is set before the return program re-executes the instruction that caused the exception.

The accessed flag A and the modified flag D can be used to effectively implement virtual memory technology. By periodically checking and resetting all A flags, the operating system can determine which pages have not been accessed recently. These pages can be candidates for removal to disk. Suppose that when a page is read into memory from disk, its dirty flag D=0, then when the page is moved out to disk again, if the D flag is still 0, the page does not need to be written to disk. If D=1 at this time, the page content has been modified, so the page must be written to disk.

4.5 Protection

In protected mode, the 80X86 provides segment and page level protection. This protection mechanism provides access restrictions on certain segments and pages based on privilege levels (4 levels of protection and level 2 page protection). For example, operating system code and data are stored in segments that have a higher privilege level than normal applications. The processor's protection mechanism will then limit the application's access to the operating system's code and data in a controlled and regulated manner.

Protection mechanisms are required for a reliable multitasking environment. It can be used to protect individual tasks from mutual interference. Segment and page level protection can be used at any stage of software development to assist in finding and detecting design issues and errors. When the program performs an undesired reference to the error memory space, the protection mechanism can block such operations and report such events.

Protection mechanisms can be used for segmentation and paging mechanisms. The 2 bits of the processor register define the privilege level of the currently executing program, called the Current Privilege Level (CPL). During segmentation and paging address translation, the processor will verify the CPL.

By setting the PE flag (bit 0) of the control register CR0, the processor can be operated in protected mode, thus turning on the segmentation protection mechanism. Once in protected mode, there is no clear control flag in the processor to stop or enable the protection mechanism. However, the privilege-level protection mechanism part can be implicitly turned off by setting the privilege level of all segment selectors and segment descriptors to level 0. This approach can prohibit privilege-level protection barriers between segments, but other segment length and segment type checks and other protection mechanisms still work.

Setting the PG flag (bit 31) of the control register CR0 enables the paging mechanism and also enables the paging protection. Similarly, there are no associated flags in the processor to disable or enable the page level protection mechanism in the paging open condition. But by setting the read/write (R/W) flag and the user/superuser (U/S) flag for each page directory entry and page table entry, we can disable page-level protection. Setting these two flags allows each page to be arbitrarily read/write, so page-level protection is actually disabled.

For the segment level protection, the processor performs protection verification using the selectors (RPL and CPL) in the segment register and the fields in the segment descriptor. For the paging mechanism, the R/W and U/S flags in the page directory and page table entries are mainly used to implement the protection operation.

4.5.1 Segment Protection

When the protection mechanism is used, each memory reference is checked to verify that the memory reference meets various protection requirements. Because the check operation is concurrent with the address translation, processor performance is not affected. The protection checks performed can be divided into the following categories:

- Segment limit checks;
- Segment type checks;
- Privilege level checks;
- Restriction of addressable domain;

- Restriction of procedure entry-points;
- Restriction of instruction set.

All violations of protection will result in an exception. The following sections describe the protection mechanisms in protected mode.

4.5.1.1 Segment Length Limit Check

The segment limit length field of the segment descriptor is used to prevent program or process addressing to the location outside the segment. The effective value of the segment length depends on the setting state of the granularity G flag. For data segments, the segment length is also related to the flag E (extended direction) and the flag B (default stack pointer size and/or upper bound). The E flag is a bit of the type field in the segment descriptor of the data segment type.

When the G flag is cleared (byte granularity), the effective segment length is the value of the segment descriptor length field Limit of the 20-bit segment descriptor. In this case, Limit ranges from 0 to 0xFFFFF (1MB). When the G flag is set (4KB page granularity), the processor multiplies the value of the Limit field by a factor of 4K. In this case, the valid Limit range is from 0xFFF to 0xFFFFFFFF (4GB). Note that when the G flag is set, the lower 12 bits of the segment offset (address) are not checked against Limit. For example, when the segment length Limit is equal to 0, the offset values 0 to 0xFFF are still valid.

Except for the expand-down data segment, the value of the valid Limit for all other segment types is the last address allowed to be accessed in the segment, which is one byte smaller than the segment length. Any valid address range specified beyond the segment length field will result in a general protection exception.

For the expand-down data segment, the segment length has the same function, but its meaning is different. Here, the segment length specifies the last address in the segment that is not allowed to access, so in the case where the B flag is set, the effective offset range is from (valid segment offset +1) to 0xFFFF FFFF; when B is cleared The valid offset value range is from (valid segment offset +1) to 0xFFFF. When the segment length of the next expansion segment is 0, the segment will have the maximum length.

In addition to checking the segment length, the processor also checks the length of the descriptor table. The GDTR, IDTR, and LDTR registers contain a 16-bit limit value that the processor uses to prevent the program from selecting descriptors outside of the descriptor table. The limit length value of the descriptor table indicates the last valid byte in the table. Since each descriptor is 8 bytes long, the table containing N descriptor entries should have a limit value of 8N-1.

The selector can have a value of zero. Such a selector points to the first unused descriptor item in the GDT table. Although this null selector can be loaded into a segment register, any attempt to reference memory using this descriptor will result in a general protection exception.

4.5.1.2 Segment Type Checking

The segment descriptor contains type information in two places, namely the S flag in the descriptor and the type field TYPE. The processor uses this information to detect programming errors caused by illegal use of segments or gates.

The S flag is used to indicate whether a descriptor is of a system type or a code or data type. The TYPE field additionally provides 4 bits for defining various types of code, data, and system descriptors. The table in the previous section gives the encoding of the code and data descriptor TYPE fields; the other table gives the encoding of the system descriptor TYPE field.

When the segment selector and descriptor are manipulated, the processor will check the type information at any time. The type information is checked mainly in the following two cases:

1. When a segment selector is loaded into a segment register. Certain segment registers can contain only certain descriptor types, for example:

- The CS register can only be loaded with a selector for an executable code segment;
 - The selector of the unreadable executable segment cannot be loaded into the data segment register;
 - Only selectors of writable data segments can be loaded into the SS register.
2. When instructions access segments whose descriptors are already loaded into segment registers. Certain segments can be used by instructions only in certain predefined ways, for example:
- No instruction can write an executable segment;
 - No instruction can write into a data segment where the writable bit is not set;
 - No instruction can read an executable segment unless the executable the readable flag is set.

4.5.1.3 Privilege Levels

The segment protection mechanism of the processor can identify 4 privilege levels (or privilege layers), 0 to 3 levels. The greater numbers mean lesser privileges. Figure 4-19 shows how these privilege levels can be interpreted as protection ring forms. The center (retained for the most advanced code, data, and stack) is used for segments that contain the most important software, usually for the core part of the operating system. The middle two rings are used for more important software. Systems using only 2 privilege levels should use privilege levels 0 and 3.

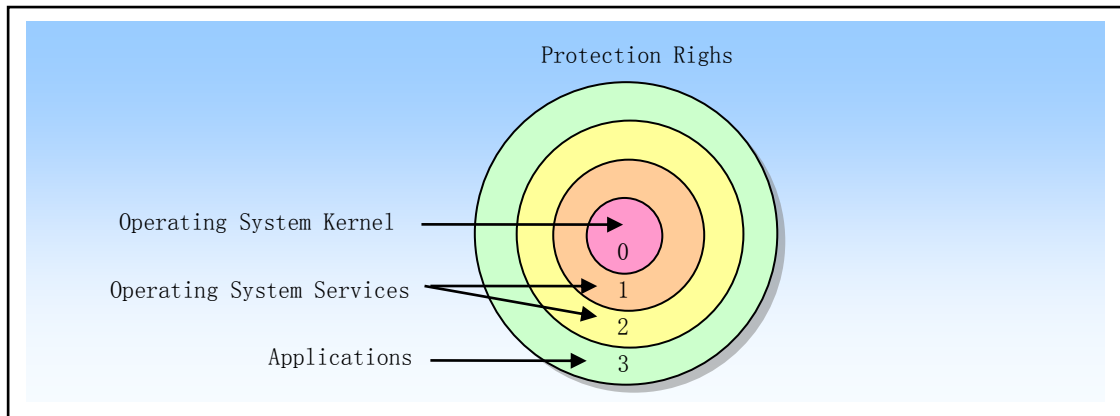


Figure 4-19 Protection Level Rings

The processor utilizes a privilege levels to prevent a program or task running at a lower privilege level from accessing a segment with a higher privilege level, except under controlled conditions. When the processor detects an operation that violates a privilege level, it generates a general protection exception.

In order to perform privilege level checking between individual code segments and data segments, the processor can recognize the following three types of privilege levels:

- **Current Privilege Level (CPL).** The CPL is the privilege level of the currently executing program or task. It is stored in bits 0 and 1 of the CS and SS segment registers. Normally, the CPL is equal to the privilege level of the code segment from which instructions are being fetched. The processor changes the CPL when program control is transferred to a code segment with a different privilege level. The CPL is treated slightly differently when accessing conforming code segments. Conforming code segments can be accessed from any privilege level that is equal to or numerically greater (less privileged) than the DPL of the conforming code segment. Also, the CPL is not changed when the processor accesses a conforming code segment that has a different privilege level than the CPL.
- **Descriptor Privilege Level (DPL).** The DPL is the privilege level of a segment or gate. It is stored in the DPL field of the segment or gate descriptor for the segment or gate. When the currently executing

code segment attempts to access a segment or gate, the DPL of the segment or gate is compared to the CPL and RPL of the segment or gate selector (as described later in this section). The DPL is interpreted differently, depending on the type of segment or gate being accessed:

- ◆ **Data Segment.** Its DPL indicates the numerically highest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a data segment is 1, only programs running at a CPL of 0 or 1 can access the segment.
- ◆ **Nonconforming code segment (without using a call gate).** The DPL indicates the privilege level that a program or task must have to access the segment. For example, if the DPL of a nonconforming code segment is 0, then only programs running at CPL 0 can access this segment.
- ◆ **Call Gate.** Its DPL indicates the numerically highest privilege level at which the current executing program or task accessing the call gate can be. (This is the same as the access rule for the data segment.)
- ◆ **Conforming code segment and nonconforming code segment accessed through a call gate.** The DPL indicates the numerically lowest privilege level that a program or task can have to be allowed to access the segment. For example, if the DPL of a conforming code segment is 2, programs running at a CPL of 0 or 1 cannot access the segment.
- ◆ **Task status segment TSS.** Its DPL indicates the numerically highest privilege level at which the current executing program or task accessing the TSS can be. (This is the same as the access rule for the data segment.)
- **Request privilege level RPL.** The RPL is an override privilege level assigned to a segment selector, which is stored in bits 0 and 1 of the selector. The processor checks both the RPL and the CPL to determine if access to a segment is allowed. Even if a program or task has sufficient privilege level (CPL) to access a segment, access will be denied if the provided RPL privilege level is insufficient. That is, if the RPL of the segment selector has a value greater than the CPL, the RPL will overwrite the CPL (and use the RPL as the privilege level for checking comparisons), and vice versa. That is, the privilege level with the largest value in the RPL and CPL is always taken as the comparison object when accessing the segment. Therefore, RPL can be used to ensure that high privileged code does not access a segment on behalf of the application unless the application itself has access to the segment.

The privilege level check operation is performed when the segment selector of the segment descriptor is loaded into a segment register, but the check method for data access is different from the one for checking the program control transfer between code segments. Therefore, the following two access situations are considered.

4.5.2 Privilege Level Check When Accessing Data Segments

To access operands in a data segment, the segment selector for the data segment must be loaded into the data-segment registers (DS, ES, FS, or GS) or into the stack-segment register (SS). (Segment registers can be loaded with the MOV, POP, LDS, LES, LFS, LGS, and LSS instructions.) Before the processor loads a segment selector into a segment register, it performs a privilege check (refer to Figure 4-20) by comparing the privilege levels of the currently running program or task (the CPL), the RPL of the segment selector, and the DPL of the segment's segment descriptor. The processor loads the segment selector into the segment register if the DPL is numerically greater than or equal to both the CPL and the RPL. Otherwise, a general protection fault is generated and the segment register is not loaded.

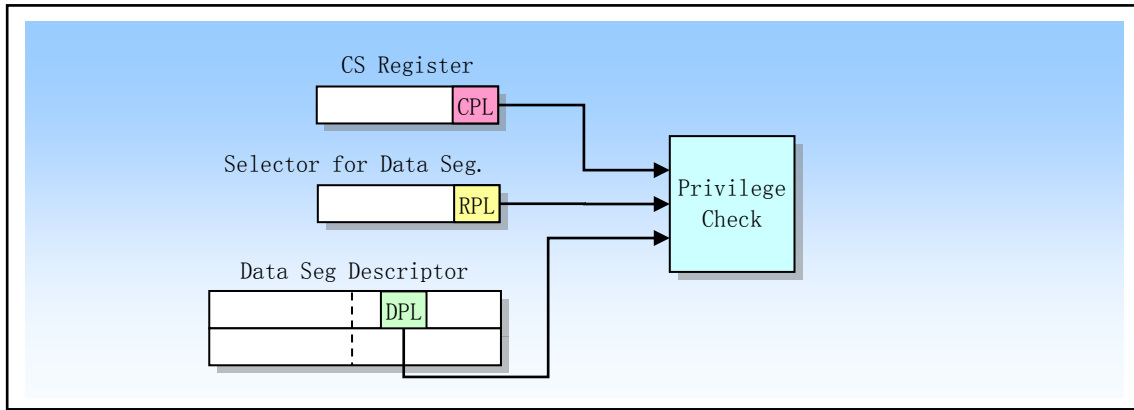


Figure 4-20 Privilege level check when accessing data segments

It can be seen that a program or task addressable area changes as its CPL changes. When the CPL is 0, the data segments at all privilege levels can be accessed at this time; when the CPL is 1, only the data segments at the privilege levels 1 to 3 can be accessed; when the CPL is 3, only the privileged level The data segment of 3 can be accessed.

In addition, it may desirable to access data structures that are contained in a code segment, for example, when the code and data are in ROM. So sometimes we will need access to the data in the code segment. At this point you can use the following methods to access the data in the code segment:

1. Load the selector of a nonconforming, readable, code segment into a data segment register.
2. Load the selector of a conforming, readable, code segment into a data segment register.
3. Use the code segment override prefix (CS) to read a readable code segment whose selector is already in the CS register.

The same rules for accessing data segments also apply to Method 1. Method 2 is always valid because the privilege level of the consistent code segment is equivalent to the CPL, regardless of the DPL of the code segment. Method 3 is also always valid because the DPL of the code segment selected by the CS register is the same as the CPL.

A privilege level check is also performed when the SS segment register is loaded using the stack segment selector. All privilege levels associated with the stack segment here must match the CPL. That is, the CPL, the RPL of the stack segment selector, and the DPL of the stack segment descriptor must all be the same. If the RPL or DPL is different from the CPL, the processor will generate a general protection exception.

4.5.3 Privilege Level Checking When Transferring Program Control

Between Code Segments

For transferring program control from one code segment to another, the segment selector for the target code segment must be loaded into the code segment register (CS). As part of this loading process, the processor detects the segment descriptor for the target code segment and performs various limit, type, and privilege level checks. If these checks pass, the target code segment selector is loaded into the CS register, and control of the program is transferred to the new code segment, and the program will begin execution at the instruction pointed to by the EIP register.

The control transfer of the program is implemented using the instructions JMP, RET, INT, and IRET as well as exception and interrupt mechanisms. Exceptions and interrupts are special implementations that will be described later. This section discusses only the JMP, CALL, and RETS instructions. A JMP or CALL instruction

can reference another code segment in one of four ways:

- The target operand contains the segment selector for the target code segment;
- The target operand points to a call gate descriptor, which contains the selector for the target code segment;
- The target operand points to a TSS, which contains the selector for the target code segment;
- The target operand points to a task gate that points to a TSS, which contains the selector for the target code segment;

The first two reference types are described below, the latter two of which are described in the section on task management.

4.5.3.1 Direct Calls or Jumps to Code Segments

The near forms of the JMP, CALL, and RET instructions simply performs program control transfers in the current code segment, so privilege level checks are not performed. The far forms of the JMP, CALL, or RET instructions transfer control to another code segments, so the processor must perform privilege level checks.

When transferring program control to another code segment without going through a call gate, the processor verifies four kinds of privilege level and type information as shown in Figure 4-21:

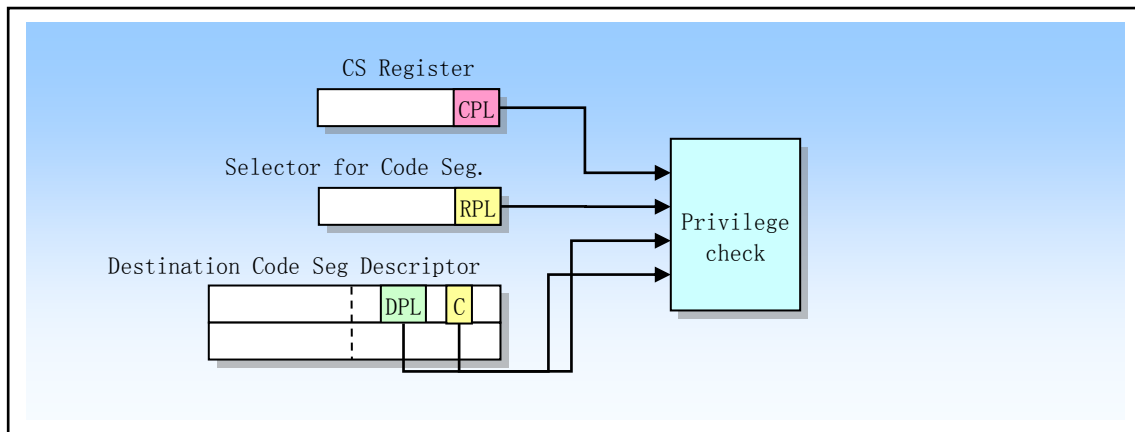


Figure 4-21 Privilege check when calling or jumping directly to another code segment

- Current privilege level CPL. (Here, CPL is the privilege level of the code segment that executes the call, that is, the code segment that executes the call or jump.)
- The descriptor privilege level DPL of the segment descriptor for the destination code segment that contains the called procedure.
- Request privilege level RPL in the segment selector of the destination code segment.
- The conforming flag C in the destination code segment descriptor. It determines whether a code segment is a non-conforming code segment or a consistent code segment.

The rules for the processor to check the CPL, RPL, and DPL depend on the setting state of the flag C. When accessing a non-conforming code segment (C=0), the CPL of the caller (program) must be equal to the DPL of the destination code segment, otherwise a general protection exception will be generated. The RPL of the segment selector pointing to a non-conforming code segment has a limited effect on the check. The RPL must be numerically less than or equal to the caller's CPL in order for the control transfer to complete successfully. When the segment selector of a non-conforming code segment is loaded into the CS register, the privilege level field does not change, ie it is still the caller's CPL. This is true even if the RPL of the segment selector is different from the CPL.

When accessing a conforming code segment ($C = 1$), the caller's CPL may be numerically greater than or equal to the DPL of the destination code segment. The processor will only generate a general protection exception when $CPL < DPL$. For access to conforming code segments, the processor ignores the check for RPL. For a conforming code segment, DPL represents the numerically lowest privilege level at which the caller can make a successful call to the code segment.

When program control is transferred to a conforming code segment, the CPL does not change, even if the DPL of the destination code segment is numerically less than the CPL. This is the only case where the CPL may not be the same as the current code segment DPL. Also, since the CPL has not changed, the stack will not switch.

Most code segments are non-conforming code segments. For these segments, control of the program can only be transferred to code segments with the same privilege level, unless the transfer is through a call gate, as explained below.

4.5.3.2 Gate Descriptors

To provide controlled access to code segments with different privilege levels, the processor provides a special set of descriptors called gate descriptors. There are four kinds of gate descriptors:

- Call Gate (TYPE=12);
- Trap Gate (TYPE=15);
- Interrupt Gate (TYPE=14);
- Task Gate (TYPE=5).

The task gate is used for task switching and will be explained later in the Task Management section. Trap gates and interrupt gates are special classes for calling gates that are used to call handlers for exceptions and interrupts, as explained in the next section. This section only describes how to use the call gate.

Call gates are used to implement controlled program control transfers between different privilege levels. They are usually only used in operating systems that use privilege-level protection mechanisms. Figure 4-22 shows the format of the call gate descriptor. The call gate descriptor can be stored in the GDT or LDT, but cannot be placed in the interrupt descriptor table IDT. A call gate has the following main functions:

- Specifies the code segment to be accessed;
- Defines an entry point for a procedure (program) in a specified code segment;
- Specifies the privilege level that the caller of the access procedure needs to have;
- If a stack switch occurs, it specifies the number of optional parameters that need to be copied between the stacks;
- Indicates whether the call gate descriptor is valid.

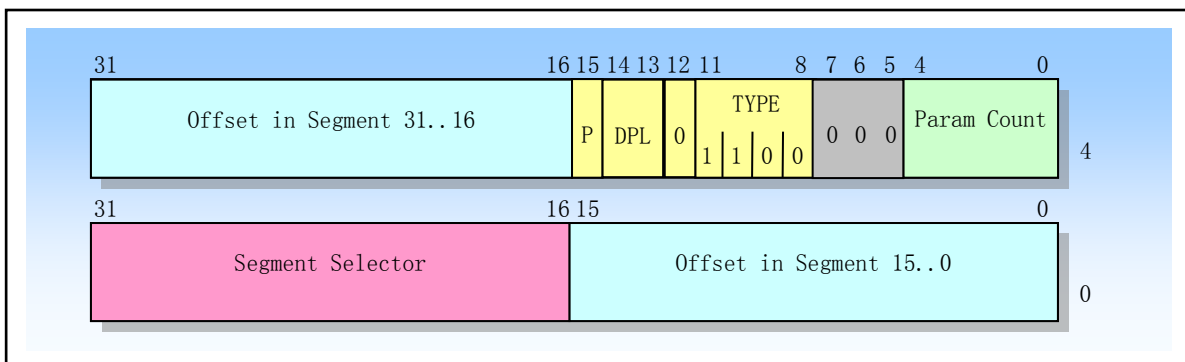


Figure 4-22 Call gate descriptor format

The segment selector field in the call gate specifies the code segment to be accessed. The Offset Value field specifies the entry point in the segment. This entry point is usually the first instruction in the specified process. The DPL field specifies the privilege level of the call gate, thereby specifying the privilege level required to access a particular procedure through the call gate. The flag P indicates whether the call gate descriptor is valid. The Parameter Count field (Param Count) indicates the number of parameters copied from the caller stack to the new stack when a stack switch occurs.

The call gate is not used in the Linux kernel. The description of the call gate is intended to prepare for the processing of interrupts and exception gates in the next section.

4.5.3.3 Accessing a Code Segment Through a Call Gate

In order to access the call gate, we need to provide a far pointer for the operand of the CALL or JMP instruction. The segment selector in this pointer is used to specify the call gate. The offset value of the pointer is needed, but the processor does not use it. This offset value can be set to any value. See Figure 4-23.

When the processor accesses the call gate, it uses the segment selector in the call gate to locate the segment descriptor for the destination code segment. The processor then combines the base address of the code segment descriptor with the offset value in the call gate to form the linear address of the specified program entry point in the code segment.

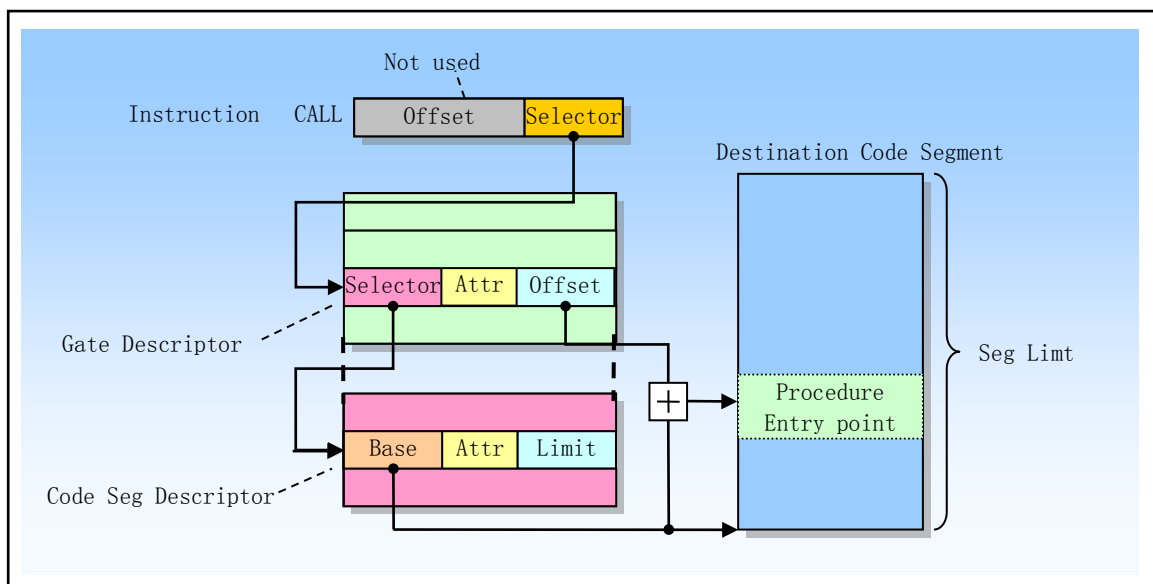


Figure 4-23 Call-gate operation process

When a program control transfer is made by a call gate, the CPU checks four different privilege levels to determine the validity of the control transfer, as shown in Figure 4-24.

- The current privilege level CPL;
- The requestor's privilege level RPL of the call gate's selector;
- The descriptor privilege level DPL of the call gate descriptor;
- The DPL of the segment descriptor of the destination code segment.

In addition, the conforming flag C in the destination code segment descriptor will also be checked.

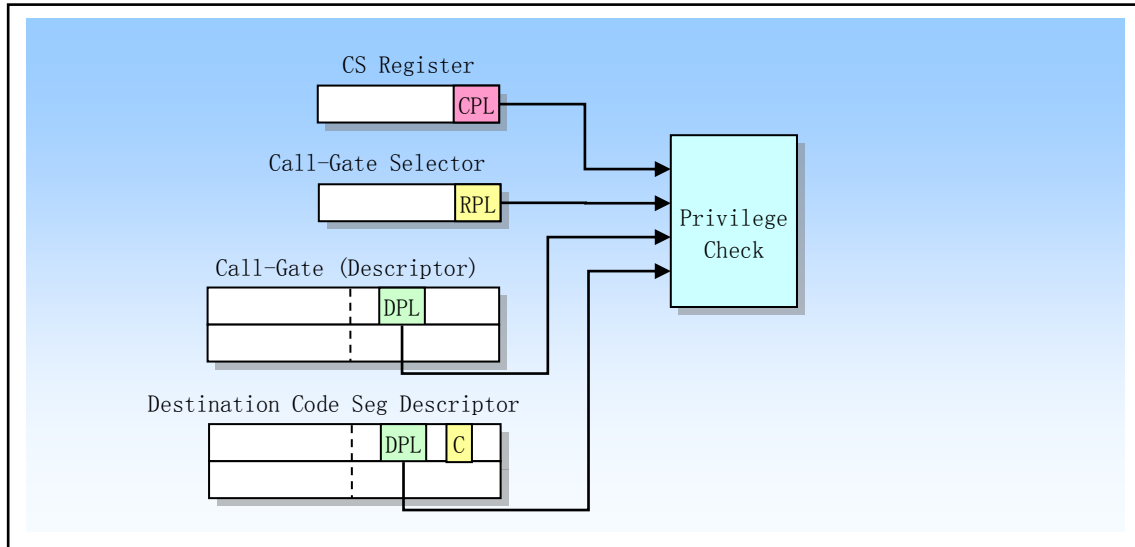


Figure 4-24 Privilege level check for control transfer with call-gate

The privilege level checking rule is different for controlling transfer using CALL instruction or JMP instruction, as shown in Table 4-5. The DPL field of the call gate descriptor indicates the numerically maximum privilege level (least privilege level) at which the caller can access the call gate. That is, in order to access the call gate, the privilege level CPL of the caller program must be less than or equal to the DPL of the call gate. The RPL of the segment selector of the call gate also needs to follow the same rules as the CPL that invokes this, ie the RPL must also be less than or equal to the DPL of the calling gate.

Table 4-5 Privilege level check rules for CALL and JMP instructions

Instruction	Privilege Check Rules (numerically)
CALL	CPL ≤ Call gate DPL; RPL ≤ Call gate DPL Destination conforming & nonconforming code segments DPL ≤ CPL
JMP	CPL ≤ Call gate DPL; RPL ≤ Call gate DPL Destination conforming code segment DPL ≤ CPL; Destination nonconforming code segment DPL = CPL

If the privilege level check between the caller and the call gate succeeds, the CPU will then compare the caller's CPL with the DPL of the code segment descriptor. In this regard, the CALL instruction and the JMP instruction check rules are different. Only the CALL instruction can use call gate to transfer program control to the more privileged (numerically lower privilege level) non-conforming code segment, that is, it can be transferred to the non-conforming code segment with a DPL less than the CPL. A JMP instruction can use a call gate only to transfer program control to a nonconforming code segment with a DPL equal to the CPL. However, both the CALL instruction and the JMP instruction can transfer control to a conforming code segment of a higher privilege level, that is, to a conforming code segment where the DPL is numerically less than or equal to the CPL.

If a call transfers control to a non-conforming code segment of a higher privilege level, the CPL is set to the DPL value of the destination code segment and causes a stack switch. But if a call or jump transfers control to a higher privileged level conforming code segment, the CPL does not change and does not cause a stack switch.

Call gate allows a procedure in a code segment to be accessed by a program of a different privilege level. For example, operating system code located in a code segment may contain code that the operating system itself and

application software allow to access (such as code that handles character I/O). So you can set up a call gate for all of these procedures that all privilege level code can access. In addition, some higher privilege level call gates can be set specifically for code that is only used by the operating system.

4.5.3.4 Stack Switching

Whenever the call gate is used to transfer program control to a more privileged non-conforming code segment, the CPU automatically switches to the stack of the privilege level of the destination code segment. The purpose of the stack switch operation is to prevent more privileged programs from crashing due to insufficient stack space, and also to prevent low privilege level programs from intentionally or unintentionally interfering with high privilege level programs through the shared stack.

Each task must define up to 4 stacks. One is for application code running at privilege level 3, and the other is used for privilege levels 2, 1, and 0, respectively. If only two privilege levels of 3 and 0 are used in a system, then only two stacks need to be set for each task. Each stack is in a different segment and is specified using the segment selector and the offset value in the segment.

When the privilege level 3 program is executing, the segment selector and stack pointer of the privilege level 3 stack are stored in the SS and ESP, respectively, and are saved on the stack of the called procedure when a stack switch occurs.

The initial pointer values for the stacks of privilege levels 0, 1, and 2 are stored in the TSS segment of the currently running task. These pointers in the TSS segment are read-only values. The CPU does not modify them while the task is running. When a higher privilege level program is called, the CPU uses them to build a new stack. When returning from the calling procedure, the corresponding stack does not exist. The next time the procedure is called, a new stack is created again using the initial pointer values in the TSS.

The operating system is responsible for establishing stack and stack segment descriptors for all used privilege levels and setting the initial pointer value in the task's TSS. Each stack must be readable and writable and have enough space to hold some of the following information:

- The contents of the SS, ESP, CS, and EIP registers for the calling process;
- The parameters of the called procedure and the space required for the temporary variables;
- The EFLAGS register and error code, when implicit calls are made to an exception or interrupt handler.

Since one procedure can call other procedures and the operating system can support nesting of multiple interrupts, each stack must have enough space to accommodate multiple frames of the above information.

When a privilege level change is made by a call to a gate, the CPU performs the following steps to switch the stack and begin executing the called procedure on the new privilege level (see Figure 4-25):

1. Select the pointer to the new stack from the TSS using the DPL of the destination code segment (ie the new CPL). The segment selector and stack pointer of the new stack are read from the current TSS. Any error that violates the segment boundary will result in an invalid TSS exception during the process of reading the stack segment selector, stack pointer, or stack segment descriptor;
2. Check if the stack segment descriptor privilege level and type are valid. If invalid, an invalid TSS exception is also generated.
3. Temporarily save the current values of the SS and ESP registers, and load the segment selector and stack pointer of the new stack into the SS and ESP. Then push the temporarily saved SS and ESP content onto the new stack.
4. Copy the specified number of parameters in the call gate descriptor from the calling procedure stack to the new stack. The value of the parameter in the call gate is up to 31. If the number is 0, it means no parameter and no copy is needed.
5. Push the return instruction pointer (ie the current CS and EIP content) onto the new stack. The new

(destination) code segment selector is loaded into the CS, and the offset value (new instruction pointer) in the call gate is loaded into the EIP. Finally, the execution of the called process begins.

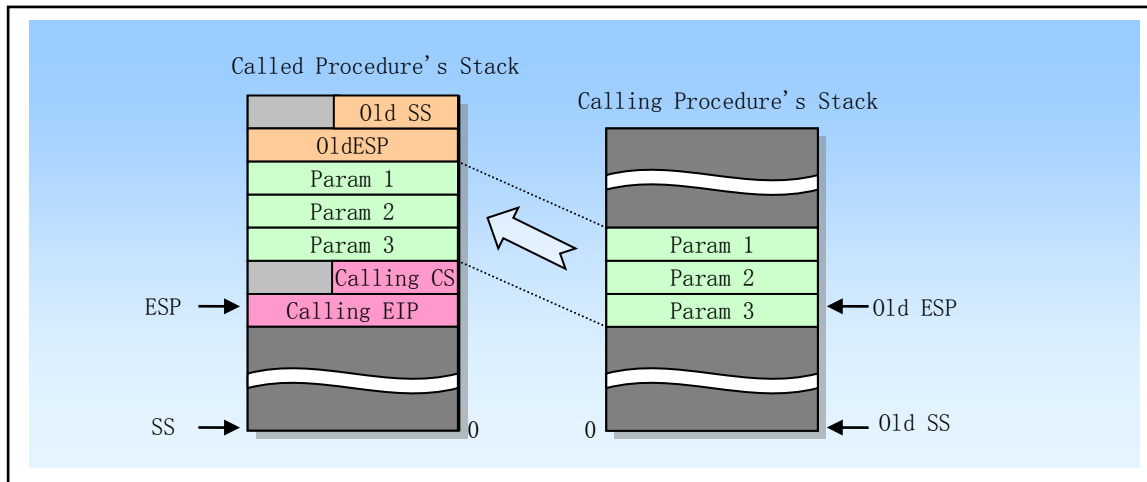


Figure 4-25 Stack switch when calling between different privilege levels

4.5.3.5 Returning from a Called Procedure

The instruction RET is used to perform near return, far return with privilege level, and far return with different privilege levels. This instruction is used to return from the procedure called with the CALL instruction. Near return only transfers program control in the current code segment, so the CPU only performs boundary checking. For far return of the same privilege level, the CPU simultaneously pops the selector of the return code segment and the return instruction pointer from the stack. Since these two pointers are normally pushed onto the stack by CALL instructions, they are valid because of this. However, the CPU still performs a privilege level check to cope with situations where the current process may modify the pointer value or if there is a problem with the stack.

The far return that would result in a privilege level change is only allowed to return to the low privilege level program, ie the code segment DPL returned is numerically greater than the CPL. The CPU uses the RPL field of the selector in the CS register to determine if a low privilege level is required. If the value of the RPL is larger than the CPL, a return operation between privilege levels is performed. When the execution returns far to a calling process, the CPU performs the following steps:

1. Check the RPL field value in the saved CS register to determine if the privilege level needs to be changed on return.
2. Pop up and load the CS and EIP registers using the values on the called procedure stack. The privilege level and type checking of the code segment descriptor and the code segment selector RPL are performed during this process.
3. If the RET instruction contains a parameter count operand and the return operation changes the privilege level, then the parameter count value is added to the ESP register after the CS and EIP values in the pop-up stack to skip the caller stack's parameter. At this point the ESP register points to the pointer SS and ESP of the originally saved caller stack.
4. Load the saved SS and ESP values into the SS and ESP registers to switch back to the caller's stack. At this time, the SS and ESP values of the caller stack are discarded.
5. If the RET instruction contains a parameter number operand, the parameter value is added to the ESP register value to skip (discard) the parameters on the caller stack.

6. Check the contents of the segment registers DS, ES, FS and GS. If there is a segment pointing to a DPL that is smaller than the new CPL (except for the consistent code segment), then the CPU loads the segment register with the NULL selector.

4.5.4 Page-Level Protection

The read/write flags R/W and the user/supervisor flag U/S in the page directory and page table entries provide a subset of the segmentation mechanism protection attributes. The paging mechanism only recognizes two levels of permissions. Privilege levels 0, 1, and 2 are classified as superuser level, while privilege level 3 is classified as a normal user level. Normal user level pages can be marked as read only/executable or readable/writable/executable. Superuser-level pages are always readable/writable/executable for superusers, but are not accessible to ordinary users, as shown in Table 4-6.

For the segmentation mechanism, programs executed at the outermost user level can only access user-level pages, but programs executed at any super user level (0, 1, 2) can not only access the user layer's page, but also access the super. User layer page. Unlike the segmentation mechanism, programs executed at the inner superuser level have readable/writable/executable permissions on any page, including those that are marked as read-only/executable at the user level.

Table 4-6 Normal and super user access restrictions on the page

U/S	R/W	User Access Rights	Supervisor Access Rights
0	0	None	Read/Write/Execute
0	1	None	Read/Write/Execute
1	0	Read/Execute	Read/Write/Execute
1	1	Read/Write/Execute	Read/Write/Execute

Just as the paging mechanism is implemented after the segmentation mechanism in the entire 80X86 address translation mechanism, page-level protection also plays a role in the protection after the the segmentation mechanism. First, all segment level protection is checked and tested. If you pass the check, the page level protection check will be performed. For example, a byte in memory can be accessed by a program on level 3 only when a byte is in a segment accessible by the program on level 3 and is marked as a user-level page. A write to a page can only be performed when both segmentation and paging are allowed to be written. If a segment is a read/write type segment, but the corresponding page corresponding to the address is marked as read-only/executable, then the page cannot be written. If the type of the segment is read-only/executable, the page always has no write permission regardless of the protection attribute given to the corresponding page. It can be seen that the protection mechanism for segmentation and paging is like a serial line in an electronic circuit, in which the switch does not open without a connection.

Similarly, the protection attribute of a page consists of the "serial" or "and operation" of the table entry and the entries in the page table, as shown in Table 4-7. The U/S flag and the R/W flag in the page table entry are applied to a single page of the entry mapping. The U/S and R/W flags in the page directory entry act on all pages mapped to the directory entry. The combined protection attribute of the page directory and the page table is composed of the AND operation of the two attributes, so the protection measures are very strict.

Table 4-7 Page directory and page table entries combined protection of the page

Dir Entry U/S	Page Entry U/S	Combined U/S	Dir Entry R/W	Page Entry R/W	Combined R/W
0	0	0	0	0	0

0	1	0	0	1	0
1	0	0	1	0	0
1	1	1	1	1	1

4.5.4.1 Software issues for modifying page table entries

To avoid having to access the page table that resides in memory each time a memory reference is accessed, the most recently used linear to physical address translation information is stored in the page conversion cache within the processor. The processor first uses the information in the buffer cache before accessing the page table in memory. The processor searches the page directory and page table in memory only when the necessary conversion information is not in the cache. Another term for page conversion caching is called Translation Lookaside Buffer (TLB).

The 80X86 processor does not maintain the dependency of the page conversion cache and the data in the page table, but requires operating system software to ensure they are consistent. That is, the processor does not know when the page table has been modified by the software. Therefore, the operating system must refresh the cache after changing the page table to ensure that the two are consistent. By simply reloading register CR3, we can complete the refresh operation on the cache.

There is a special case where modifying the page table entry does not require refreshing the page conversion cache. That is, when the entry of the non-existing page is modified, even if the P flag is changed from 0 to 1 to mark the entry to be effective for page conversion, there is no need to refresh the cache. Because invalid entries are not stored in the cache, we don't need to refresh the page conversion cache when we call a page from disk into memory to make the page exist.

4.5.5 Combining page and segment protection

When the paging is enabled, the CPU first performs segment-level protection before processing page-level protection. If the CPU detects a protection violation error at any level, it will discard the memory access and generate an exception. If it is an exception generated by the segment mechanism, then no more page exception will be generated.

Page level protection cannot be used to replace or override segment level protection. For example, if a code segment is set to be non-writable, then after the code segment is paged, the page will not be writable even if the page's R/W flag is set to be readable and writable. At this point the segment protection check will block any attempt to write to the page. Page level protection can be used to enhance segment level protection. For example, if a readable and writable data segment is paged, the page level protection mechanism can be used to write protect individual pages.

4.6 Interrupt and Exception Handling

Interrupts and Exceptions are events that occur somewhere in the system, processor, or current executor (or task) that need to be processed by the processor. Often, such events can cause execution control to be forced from the currently running program to a special software function or task called interrupt handler or exception handler. The action taken by the processor in response to an interrupt or exception is called an interrupt/exception service (processing).

Typically, an interrupt occurs at a random time in the execution of the program in response to a signal from the hardware. The system hardware uses interrupts to handle external events, such as requiring service to external devices. Of course, the software can also generate interrupts by executing the INT n instruction.

An exception occurs when the processor executes an instruction and an error condition is detected, such as an

error condition divided by zero. The processor can detect various error conditions, including violations of protection mechanisms, page faults, and internal machine errors.

For applications and operating systems, the 80X86 interrupt and exception handling mechanisms transparently handle interrupts and exceptions that occur. When an interrupt is received or an exception is detected, the processor automatically suspends the currently executing program or task and begins running the interrupt or exception handler. When the handler completes, the processor resumes and continues executing the interrupted program or task. The recovery process of the interrupted program does not lose the continuity of program execution unless it is impossible to recover from the exception or the interrupt causes the current program to be terminated. This section describes the processing mechanisms for processor interrupts and exceptions in protected mode.

4.6.1 Sources of Interrupts

The processor receives interrupts from two places:

- External (hardware generated) interrupts;
- Software generated interrupts.

External interrupts are received by two pins (INTR and NMI) on the processor chip. When the pin INTR receives an external interrupt signal, the processor reads the interrupt vector number provided by the external interrupt controller (such as the 8259A) from the system bus. When the pin NMI receives a signal, it generates a non-maskable interrupt. It uses a fixed interrupt vector number of 2. Any external interrupt received through the INTR pin of the processor is referred to as a maskable hardware interrupt, including interrupt vector numbers 0 through 255. The IF flag in the flag register EFLAGS can be used to mask all of these hardware interrupts.

The INT n instruction can be used to generate an interrupt from software by providing the interrupt vector number in the instruction operand. For example, the instruction INT 0x80 will execute the Linux system interrupt call interrupt 0x80. Any of the vectors from 0 to 255 can be used as a parameter in this instruction. However, if a processor predefined NMI vector is used, the processor's response to it will be different from the normal-generated NMI interrupt. If the NMI vector number 2 is used for the INT instruction, the NMI interrupt handler is called, but the processor's NMI processing hardware is not activated at this time.

Note that interrupts generated in software using the INT instruction cannot be masked by the IF flag in EFLAGS register.

4.6.2 Sources of Exceptions

There are also two sources of exceptions received by the processor:

- Processor-detected program-error exceptions;
- Software-generated exceptions.

One or more exceptions are raised if the processor detects a program error during the execution of an application or operating system. The 80X86 processor defines a vector for each exception it detects. Exceptions can be further classified as faults, traps, and aborts, as explained later.

The INTO, INT 3, and BOUND instructions can be used to generate exceptions from software. These instructions check for special exception conditions that are executed at specified points in the instruction stream. For example, the INT 3 instruction will generate a breakpoint exception.

The INT n instruction can be used to simulate a specified exception in software, with one limitation. If the operand n in the INT instruction is one of the 80X86 exception vector numbers, then the processor will generate an interrupt for the vector that will execute the exception handler associated with the vector. But since this is actually an interrupt, the processor does not push an error number onto the stack, even if the vector-related

interrupt generated by the hardware usually produces an error code. For exceptions that generate an error code, the exception handler will attempt to pop the error code from the stack while processing. Therefore, if an INT instruction is used to emulate the generation of an exception, the handler will pop off and discard the EIP (just at the missing error code location) onto the stack, causing a return position error.

4.6.3 Exception Classifications

Exceptions can be subdivided into Faults, Traps, and Aborts depending on how the exception is reported and whether the instruction that caused the exception can be re-executed with no loss of program or task continuity.

- A Fault is an exception that can usually be corrected and can continue to run once it has been corrected. When a Fault occurs, the processor will restore the state of the machine to the state it was in prior to the instruction that generated the Fault. At this point, the return address of the exception handler will point to the instruction that generated the Fault, not the one that follows. Therefore, the instruction that generated the fault after returning will be re-executed.
- Trap is an exception that causes a trap to be reported immediately after the execution of the trapping instruction. Trap also enables programs or tasks to execute without loss of continuity. The return address for the trap handler points to the next instruction, so the next instruction is executed after the return.
- Abort is an exception that does not always report the exact location of the instruction that caused the exception, and does not allow the program that caused the exception to resume execution. Abort is used to report serious errors such as hardware errors and inconsistencies or illegal values in the system tables.

4.6.4 Exception and Interrupt Vectors

To help handle exceptions and interrupts, each defined exception and interrupt condition that needs to be specially processed by the processor is given an identification number called a vector. The processor uses the vector as an index number in the Interrupt Descriptor Table (IDT) to locate an exception or interrupt handler entry point location.

The allowed vector number ranges from 0 to 255. 0 to 31 are reserved for the exceptions and interrupts defined by the 80X86 processor, but currently the vector numbers in this range are not defined for each function, and the vector number of the undefined function will be reserved for future use.

Vector numbers ranging from 32 to 255 are used for user-defined interrupts. These interrupts are typically used for external I/O devices so that they can send interrupts to the processor through an external hardware interrupt mechanism.

The vectors assigned to the exceptions and NMI interrupts defined for 80X86 are given in Table 4-8. For each exception, the table gives the exception type and whether an error code is generated and saved on the stack. Each pre-defined exception and NMI interrupt source is also given.

Table 4-8 Exceptions and interruptions in protected mode

Vector No.	Mnemonic	Description	Type	Error Code	Source
0	#DE	Divide Error	Fault	No	DIV and IDIV instructions.
1	#DB	Debug	Fault/Trap	No	Any code or data reference or the INT 1 instruction.
2	--	NMI Interrupt	Interrupt	No	Nonmaskable external interrupt.
3	#BP	Breakpoint	Trap	No	INT 3 instruction.
4	#OF	Overflow	Trap	No	INTO instruction.

5	#BR	BOUND Range Exceeded	Fault	No	BOUND instruction.
6	#UD	Invalid Opcode (Undefined)	Fault	No	UD2 instruction or reserved (new for P6)
7	#NM	Device Not Available (No Math Coprocessor)	Fault	No	Floating-point or WAIT/FWAIT instruction.
8	#DF	Double Fault	Abort	Yes(Zero)	Any instruction that can generate an exception, an NMI, or an INTR.
9	--	Coprocessor Segment Overrun (reserved)	Fault	No	Floating-point instruction (not for CPU after 386)
10	#TS	Invalid TSS	Fault	Yes	Task switch or TSS access.
11	#NP	Segment Not Present	Fault	Yes	Loading segment registers or accessing system segments.
12	#SS	Stack-Segment Fault	Fault	Yes	Stack operations and SS register loads.
13	#GP	General Protection	Fault	Yes	Any memory reference and other protection checks.
14	#PF	Page Fault	Fault	Yes	Any memory reference.
15	--	(Intel reserved. Do not use.)		No	
16	#MF	Floating-Point Error (Math Fault)	Fault	No	Floating-point or WAIT/FWAIT instruction.
17	#AC	Alignment Check	Fault	Yes(Zero)	Any data reference in memory.
18	#MC	Machine Check	Abort	No	Error codes (if any) and source are model dependent.
19	#XF	Streaming SIMD Extensions	Fault	No	SSE and SSE2 floating-point instructions. (for PIII cpu)
20-31	--	Intel reserved. Do not use.			
32-255	--	User Defined (Nonreserved) Interrupts	Interrupt		External interrupt or INT n instruction.

4.6.5 Program or Task Restart

In order for a program or task to resume execution after an exception or interrupt has been processed, all exceptions except Abort can report the exact instruction location, and all interrupt guarantees occur on the instruction boundary.

For fault class exceptions, the return instruction pointer saved when the processor generates an exception points to the faulting instruction. Therefore, when a program or task is restarted after the fault handler returns, the original faulting instruction is re-executed. Re-execution of the instruction that caused the fault is usually used to handle the case where the access instruction operand is blocked. The most common example of a Fault is a page fault exception. This exception occurs when a program references an operand that is not on a page in memory. When a page fault exception occurs, the exception handler can load the page into memory and resume program execution by re-execution of the faulting instruction. To ensure that re-execution is transparent to the current execution program, the processor saves the necessary register and stack pointer information so that it can return to the state it was in prior to executing the faulting instruction.

For a Trap class exception, the return pointer saved when the processor generates an exception points to the next instruction that caused the trap operation. If a Trap is detected during the execution of an instruction that

performs a control transfer, the return instruction pointer reflects the transition of the control. For example, if a Trap exception is detected while executing a JMP instruction, the return instruction pointer points to the target location of the JMP instruction, not to the next instruction of the JMP instruction.

The abort class exceptions do not support reliable restarting of programs or tasks. A handler that aborts an exception is typically used to collect diagnostic information about the state of the processor when the exception occurred, and to close the program and system as appropriately as possible.

Interrupts strictly support restarting of interrupted programs without losing any continuity. The return instruction pointer saved for an interrupt points to the next instruction boundary that will be executed when the processor acquires the interrupt. If the instruction just executed has a repeat prefix, the interrupt will occur when the current iteration ends and the register has been set for the next iteration.

4.6.6 Enabling and Disabling Interrupts

The Interrupt Enable Flag (IF) of the Flag Register EFLAGS can disable the servicing of maskable hardware interrupts received on the INTR pin of the processor. When IF = 0, the processor disables the interrupt sent to the INTR pin; when IF = 1, the interrupt signal sent to the INTR pin is processed by the processor.

The IF flag does not affect the non-masked interrupts sent to the NMI pin, nor does it affect the exception generated by the processor. As with the other flags in EFLAGS, the processor clears the IF flag (IF=0) in response to a hardware reset operation.

The IF flag can be set or cleared using the instructions STI and CLI. These two instructions can only be executed when the program's CPL ≤ IOPL, otherwise a general protective exception will be raised. The IF flag is also affected by the following operations:

- The PUSHF instruction can store the EFLAGS contents on the stack, where they can be examined and modified. The POPF instruction can be used to put the contents of the modified flags back into the EFLAGS register.
- The task switch, POPF, and IRET instructions load the EFLAGS register. Therefore, they can be used to modify the setting of the IF flag.
- When an interrupt is processed through the interrupt gate, the IF flag is automatically cleared (reset), which disables the maskable hardware interrupt. However, if an interrupt is handled through the trap gate, the IF flag will not be reset.

4.6.7 Priority of exceptions and interrupts

If there are multiple exceptions or interrupt pending processing at the boundary of an instruction, the processor processes them in the specified order. Table 4-9 shows the priority of the exception and interrupt source classes. The processor first processes exceptions or interrupts in the highest priority class. Low priority exceptions are discarded, while low priority interrupts are held waiting. When the interrupt handler returns to a program or task that generated an exception and/or interrupt, the discarded exception will reoccur.

Table 4-9 Priority of exceptions and interrupts

Priority	Description
1(Highest)	Hardware reset: RESET
2	Task switching trap: T flag is set in TSS
3	External hardware intervention
4	Previous instruction trap: breakpoint, debug trap exception
5	External interrupt: NMI interrupt, maskable hardware interrupt

6	Code breakpoint error
7	Take an instruction error: violation of code segment limit, code page error
8	The next instruction decode error: instruction length >15 bytes, invalid opcode, coprocessor does not exist
9(Lowest)	Execution instruction error: overflow, boundary check, invalid TSS, segment not present, stack error, general protection, data page, alignment check, floating point exception

4.6.8 Interrupt Descriptor Table (IDT)

The Interrupt Descriptor Table (IDT) associates each exception or interrupt vector with a gate descriptor of their process or task, which is used to handle related exceptions and interrupts. Similar to the GDT and LDT tables, IDT is also an array of 8-byte long descriptors. Unlike GDT, the first item in the table can contain descriptors. To form an index into the IDT table, the processor puts the vector number of the exception or interrupt $\times 8$. Since there are at most 256 interrupt or exception vectors, the IDT does not need to contain more than 256 descriptors. IDTs can contain fewer than 256 descriptors because descriptors are only needed for exceptions or interruptions that may occur. However, all empty descriptor entries in the IDT should have their presence bit (flag) set to zero.

The IDT table can reside anywhere in the linear address space, and the processor uses the IDTR register to locate the location of the IDT table. This register contains the 32-bit base address of the IDT table and the 16-bit length (length limit) value, as shown in Figure 4-26. The IDT table base address should be aligned on an 8-byte boundary to improve processor access efficiency. The limit length value is the length of the IDT table in bytes.

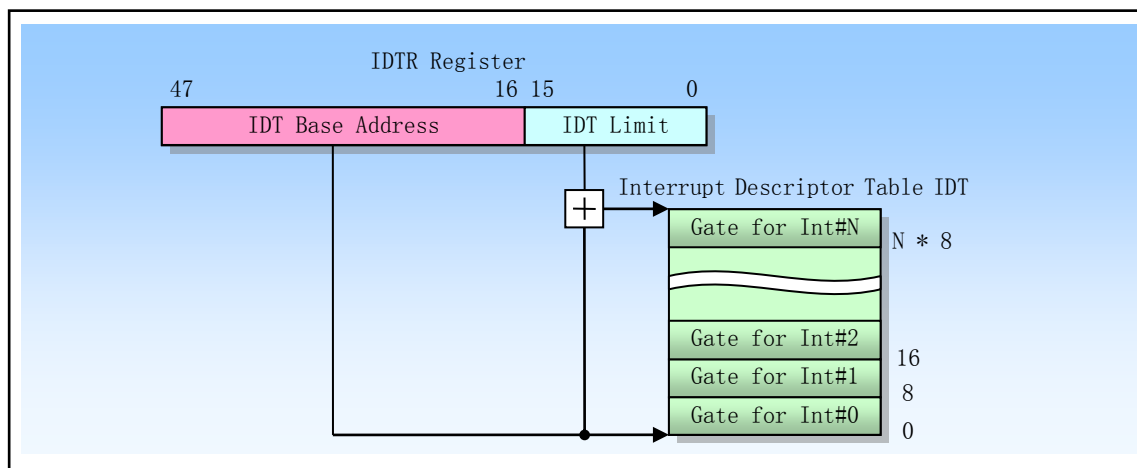


Figure 4-26 Interrupt Descriptor Table IDT and Register IDTR

The LIDT and SIDT instructions are used to load and store the contents of the IDTR register, respectively. The LIDT instruction loads the limit and the base address operand in memory into the IDTR register. This instruction can only be executed by code whose current privilege level CPL is 0, and is usually used in the operating system initialization code when IDT is created. The SIDT instruction is used to copy the base address and the limit content in the IDTR into the memory. This instruction can be executed at any privilege level. If the descriptor referenced by the interrupt or exception vector exceeds the bounds of the IDT, the processor generates a general protection exception.

4.6.9 IDT Descriptors

4.6.10 Exception and Interrupt Handling

The processor's method of invoking exceptions and interrupt handlers is similar to calling a procedure and task using the CALL instruction. When responding to an exception or interrupt, processor uses the vector of the exception or interrupt as an index in the IDT table. If the index value points to an interrupt gate or trap gate, the processor invokes an exception or interrupt handler using a method similar to the CALL instruction operation call gate. If the index value points to the task gate, the processor performs a task switch using a method similar to the CALL instruction operation task gate, and performs an exception or interrupted processing task.

An exception or interrupt gate references an exception or interrupt handler that runs in the context of the current task, as shown in Figure 4-28. The segment selector in the gate points to the executable code segment descriptor in the GDT or the current LDT. The offset field in the gate descriptor points to the beginning of the exception or interrupt handling process.

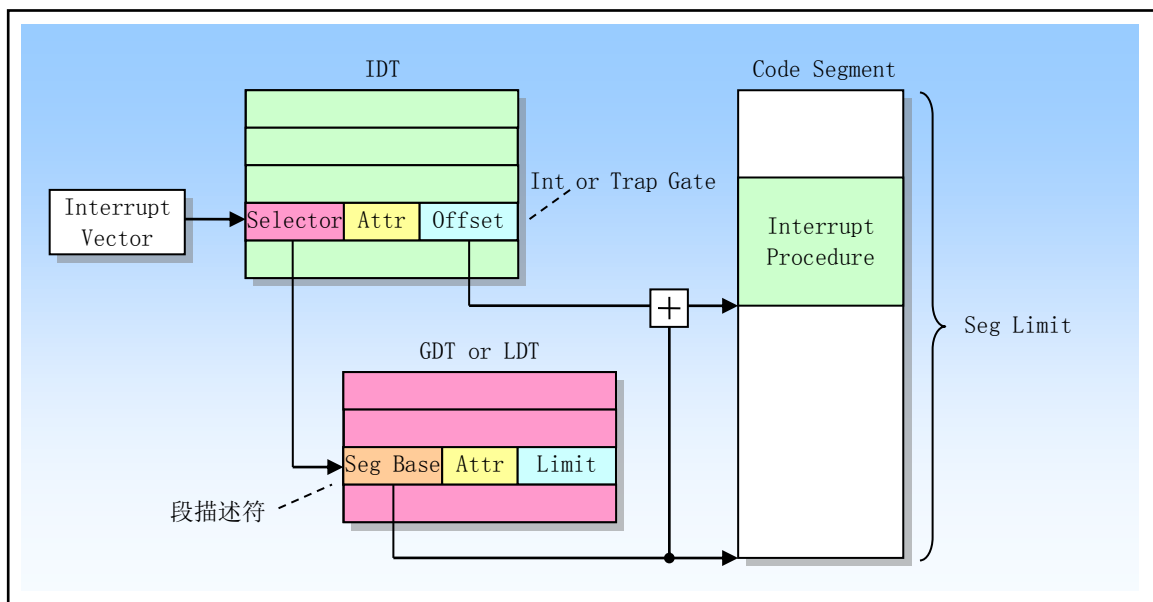


Figure 4-28 Interrupt Procedure Call

When the processor executes an exception or interrupt handler call, the following actions are taken:

- If the handler procedure will be executed at a high privilege level (such as level 0), then a stack switch operation will occur. The stack switching process is as follows:

The processor gets the segment selector and stack pointer of the stack used by the interrupt or exception handler from the TSS segment of the currently executing task (eg `tss.ss0`, `tss.esp0`). The processor then pushes the stack selector and stack pointer of the interrupted program (or task) onto the new stack, as shown in Figure 4-29. The processor then pushes the current values of the EFLAGS, CS, and EIP registers onto the new stack. If the exception generates an error code, the error code will also be pushed to the new stack.
- If the handler procedure will run on the same privilege level as the interrupted task, then:

The processor saves the current values of the EFLAGS, CS, and EIP registers on the current stack. If the exception generates an error code, the error code will also be pushed to the new stack.

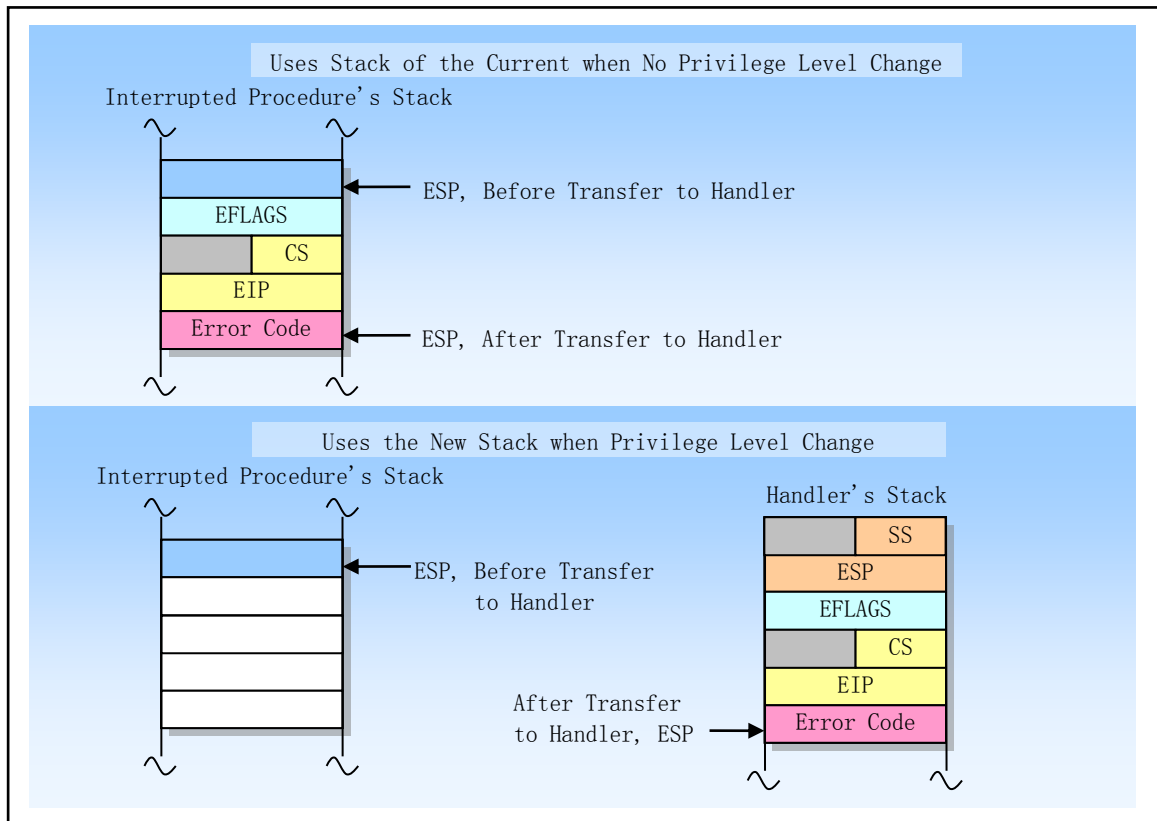


Figure 4-29 Stack usage method when transferring to interrupt processing

To return from an exception- or interrupt-handler procedure, the handler must use the IRET (or IRETD) instruction. The IRET instruction is similar to the RET instruction except that it restores the saved flags into the EFLAGS register. The IOPL field of the EFLAGS register is restored only if the CPL is 0. The IF flag is changed only if the $CPL \leq IOPL$. If a stack switch occurred when calling the handler procedure, the IRET instruction switches back to the interrupted procedure's stack on the return.

4.6.10.1 Protection of exceptions and interrupt handler procedures

The privilege level protection mechanism for exception and interrupt handler procedures is similar to calling a normal procedure through a call gate. The processor does not allow control to be transferred to interrupt handler procedure that is lower than the CPL privileged code segment, otherwise a general protection exception will be generated. In addition, the protection mechanism for interrupts and exceptions is different from the general call gate procedure in the following respects:

- Because the interrupt and exception vectors do not have an RPL, the RPL is not checked when the exception and interrupt handler procedures are implicitly called.
- The processor checks the DPL in the interrupt or trap gate only when an exception or interrupt is generated using the INT n, INT 3, or INTO instruction. At this time, the CPL must be less than or equal to the DPL of the gate. This restriction prevents applications running at privilege level 3 from using software interrupts to access important exception handling procedures, such as page fault handling, assuming that these processes have been placed in a higher privilege level code segment. For hardware-generated interrupts and processor-detected exceptions, the processor ignores the DPL in the interrupt gate and trap gate.

Because exceptions and interrupts do not usually occur on a regular basis, these rules about privilege levels

effectively enhance the privilege level limits that exceptions and interrupt handlers can run. We can use one of the following techniques to avoid violating privilege level protection:

- Exception or interrupt handlers can be stored in a consistent code segment. This technique can be used for handlers that only need to access data available on the stack (for example, divide error exceptions). If the handler needs data in the data segment, then privilege level 3 must be able to access this data segment. But there is no protection at all.
- The handler can be placed in a nonconforming code segment with privilege level 0. This handler can always be performed regardless of the current privilege level CPL of the interrupted program or task.

4.6.10.2 Flag usage by Exception or Interrupt Handler Procedure

When an exception or interrupt handler is accessed through an interrupt gate or trap gate, the processor clears the TF flag in EFLAGS after saving the contents of the EFLAGS register on the stack. Clearing the TF flag prevents the instruction trace from affecting the interrupt response. The subsequent IRET instruction will restore the original TF flag of EFLAGS with the contents of the stack.

The only difference between an interrupt gate and a trap gate is the way the processor operates the IF flag of the EFLAGS register. When an exception or interrupt handler is accessed through the interrupt gate, the processor resets the IF flag to prevent other interrupts from interfering with the current interrupt handler. Subsequent IRET instructions will restore the IF flag of the EFLAGS register with the contents stored on the stack. Accessing the handler procedure through the trap gate does not affect the IF flag.

4.6.11 Interrupt handler Tasks

Task switching occurs when an exception or interrupt handler is accessed through the task gate in the IDT. Using separate tasks to handle exceptions or interruptions has the following benefits:

- The complete context of the interrupted program or task is automatically saved;
- A new TSS permits the handler to use a new privilege level 0 stack when handling the exception or interrupt. If an exception or interrupt occurs when the current privilege level 0 stack is corrupted, accessing the handler through a task gate can prevent a system crash by providing the handler with a new privilege level 0 stack.
- The handler can be further isolated from other tasks by giving it a separate address space. This is done by giving it a separate LDT.

The disadvantage of using a separate task to handle exceptions or interrupts is that the amount of machine state must be saved during task switching, making it slower than using interrupt gates, resulting in increased interrupt latency.

The task gate in the IDT will reference the TSS descriptor in the GDT, as shown in Figure 4-30. The process of switching to a handler task is the same as the normal task switching process. The back link to the interrupted task will be saved in the previous task link field of the handler task TSS. If an exception generates an error code, the error code is copied to the new task stack.

When an exception or interrupt handler task is used in an operating system, there are actually two mechanisms for dispatching tasks: operating system software scheduling and hardware scheduling of the processor interrupt mechanism. The software scheduler needs to accommodate interrupt tasks that may be dispatched when interrupts are enabled.

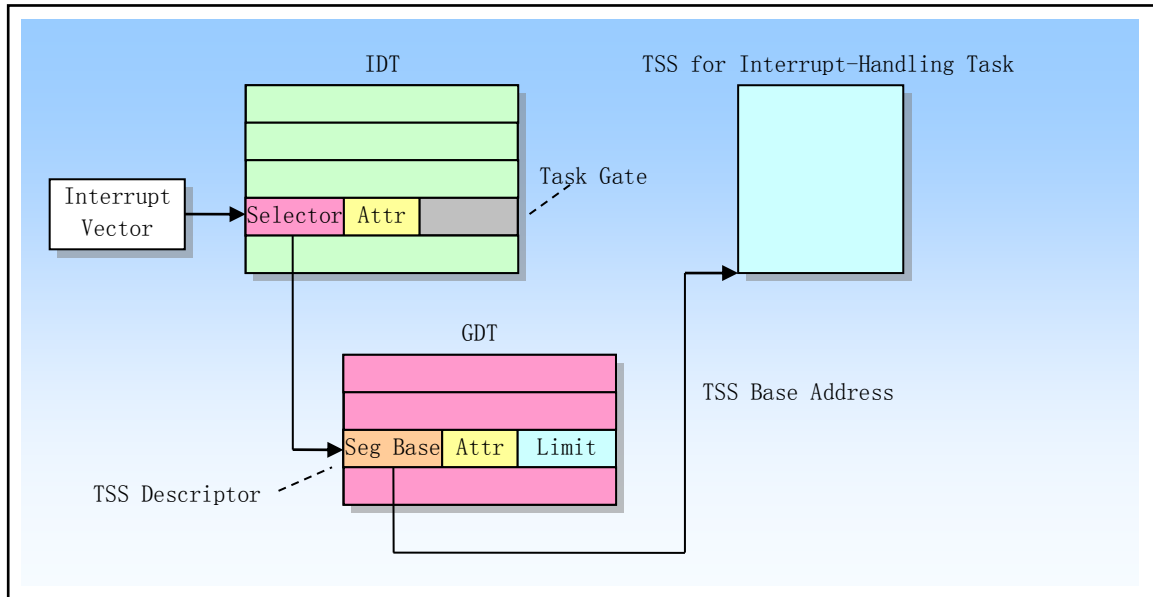


Figure 4-30 Interrupt Task Switch

4.6.12 Error Code

When an exception condition is associated with a particular segment, the processor pushes an error code onto the stack of exception handler. The format of the error code is shown in Figure 4-31. The error code is much like a segment selector, but the lowest 3 bits are not TI and RPL fields, but the following 3 flags:

- Bit 0 is the external event (EXT) flag. When set, indicates that an event external to the program caused an exception, such as a hardware interrupt.
- Bit 1 is a descriptor location (IDT) flag. When this bit is set, the index portion indicating the error code points to a gate descriptor in the IDT. When this bit is reset, it indicates that the index refers to a descriptor in the GDT or LDT.
- Bit 2 is the GDT/LDT table select flag TI. Only useful when IDT (bit 1) is 0. When the TI=1, the index portion indicating the error code points to a descriptor in the LDT. When TI=0, it indicates that the index part in the error code points to a descriptor in the GDT table.

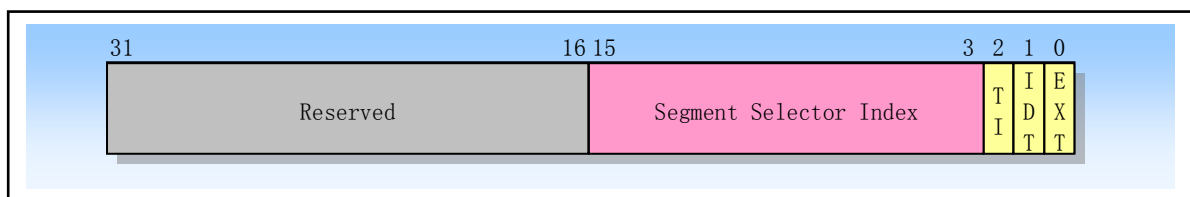


Figure 4-31 Error Code format

The Segment Selector Index field provides the index into the IDT, GDT, or current LDT to the segment or gate selector being referenced by the error code. In some cases the error code is null (ie, the lower 16 bits are all clear). A null error code indicates that the error was not caused by referencing a particular segment, or that a null segment descriptor was referenced in the operation.

The error code format of the Page-fault exception is different from the above, as shown in Figure 4-32. Only the lowest 3 bits are useful, and their names are the same as the last three bits in the page table entry (U/S, W/R,

P). The meanings and effects are:

- Bit 0 (P), the exception is caused by a page not being present or violating access privileges. P=0, indicating that the page does not exist; P=1 means that the page-level protection authority is violated.
- Bit 1 (W/R), the exception is due to a memory read or write operation. W/R=0, indicating that it is caused by a read operation; W/R=1, indicating that it is caused by a write operation.
- Bit 2 (U/S), the code level at which the CPU executes when an exception occurs. U/S=0, indicating that the CPU is executing the super user code; U/S=1, indicating that the CPU is executing the general user code.

In addition, the processor also stores the linear address used to cause the page fault exception to be stored in CR2. The page fault exception handler can use this address to locate the relevant page directory and page table entry.

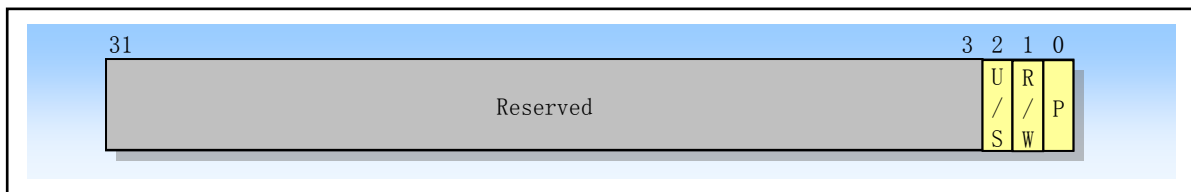


Figure 4-32 Page fault error code format

Note that the error code is not automatically popped off the stack by the IRET instruction, so the interrupt handler must clear the error code on the stack before returning. In addition, although some exceptions generated by the processor will generate an error code and will be automatically saved to the stack of the handler procedure, an external hardware interrupt or an exception generated by the program executing the INT n instruction will not push the error code onto the stack.

4.7 Task Management

A task is a unit of work that a processor can allocate to schedule, execute, and suspend. It can be used to execute programs, tasks or processes, operating system services, interrupt or exception handling procedures, and kernel code. A task is a running program or a program waiting to be run.

The 80X86 provides multi-tasking hardware support for saving the status of tasks, dispatching tasks, and switching from one task to another. When working in protected mode, all of the processor's operations are in the task. Even a simple system must define at least one task. More complex systems can use the task management capabilities of the processor to support multitasking applications.

We can perform a task by interrupt, exception, jump or call with a specified entry in a descriptor table. There are two kinds of task-related descriptors in the descriptor table: task state segment descriptors and task gates. When execution rights are passed to any of these kinds of descriptors, task switching occurs. Task switching is much like a procedure call, but task switching saves more processor state information. Task switching will completely transfer control to a new execution environment, the execution environment of the new task. This transfer operation requires saving the current contents of almost all registers in the processor, including the flag register EFLAGS and all segment registers. However, the task cannot be reentrant. Task switching does not push any information onto the stack. The processor's state information is stored in a data structure called a task state segment (TSS) in memory.

4.7.1 Task Structure and State

A task consists of two parts: the task execution space and the task state segment (TSS). The task execution space includes a code segment, a stack segment, and one or more data segments, as shown in Figure 4-33. If an operating system uses the processor's privilege level protection mechanism, then the task execution space needs to provide a separate stack space for each privilege level. The TSS specifies the segments that make up the task execution space and provides storage for the task state information. In a multitasking environment, TSS also provides a way to handle links between tasks.

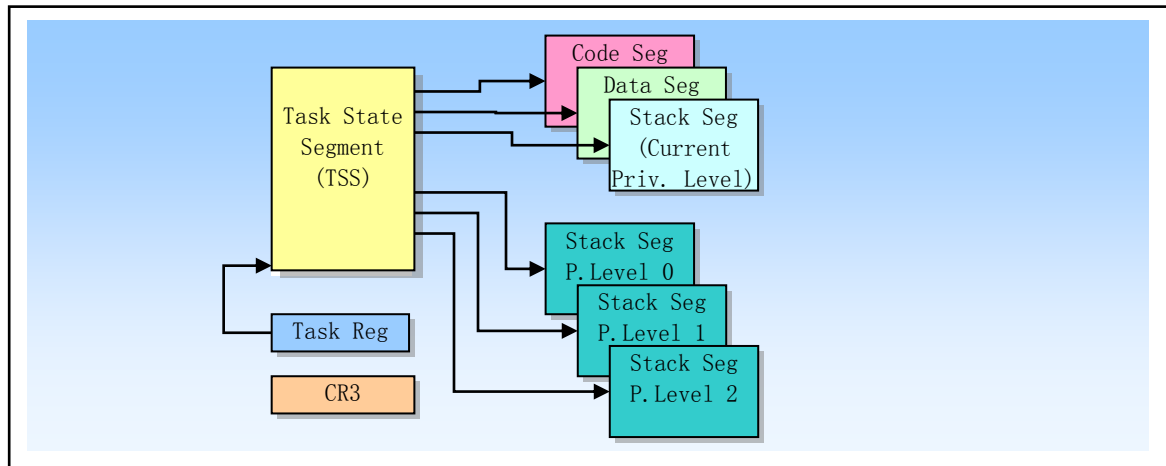


Figure 4-33 Structure of a Task

A task is specified using a segment selector that points to its TSS. When a task is loaded into the processor for execution, then the segment selector, base address, segment length, and TSS segment descriptor attributes of the task are loaded into the task register (TR). If the paging mechanism is used, the page directory base address used by the task is loaded into control register CR3. The state of the currently executing task consists of the following:

- The task's current execution space, defined by the segment selectors in the segment registers (CS, DS, SS, ES, FS, and GS);
- The state of the general purpose registers;
- The state of the EFLAGS, EIP, control register CR3, task register and LDTR register;
- The I/O map base address and I/O map (contained in the TSS);
- Stack pointers to the privilege 0, 1, and 2 stacks (contained in the TSS);
- Link to previously executed task (contained in the TSS).

Before distributing a task, all of these items are included in the task's TSS, except for the status of the task registers. In addition, the complete contents of the LDTR register are not included in the TSS and only contain the segment selector of the LDT.

4.7.2 Execution of Tasks

The software or processor can dispatch a task for execution in one of the following methods:

- A explicit call to a task with the CALL instruction;
- A explicit jump to a task with the JMP instruction (the way the Linux kernel uses);
- An implicit call (by the processor) to an interrupt-handler task;

- An implicit call to an exception-handler task;
- A return (initiated with an IRET instruction) when the NT flag in the EFLAGS register is set.

All of these methods of scheduling task execution use a selector that points to the task gate or task TSS segment to determine a task. When dispatching a task using a CALL or JMP instruction, the selector in the instruction may either directly select the task's TSS or the task gate that holds the selector for the TSS. When dispatching a task to handle an interrupt or exception, then the interrupt or exception entry in the IDT must contain a task gate that holds the selector for the TSS of the interrupt or exception handler task.

When a task is dispatched for execution, a task switching operation occurs automatically between the currently running task and the scheduled task. During a task switch, the execution environment (called the state or context of the task) that is currently running the task is saved to its TSS and the execution of the task is suspended. The context of the newly scheduled task is then loaded into the processor and a new task is executed starting from the instruction pointed to by the loaded EIP.

If the currently executing task (the caller) invokes the scheduled new task (the callee), then the caller's TSS segment selector is stored in the callee TSS, providing a link back to the caller. For all 80X86 processors, tasks are not recursively called, ie tasks cannot be called or jumped to themselves.

Interrupts or exceptions can be handled by switching to a handler task. In this case, the processor not only can perform a task switch to handle interrupts or exception, but also automatically switch back to the interrupted task when the interrupt or exception handler task returns. This mechanism can handle interrupts that occur during interrupt tasks.

As part of the task switching operation, the processor also switches to another LDT, allowing each task to have a different logical-to-physical address mapping for LDT-based segments. At the same time, the page directory register CR3 is also reloaded at the time of switching, so each task can have its own set of page tables. These protections can be used to isolate individual tasks and prevent them from interfering with each other.

It is optional to use the processor's task management capabilities to handle multitasking applications. We can also use software to implement multitasking so that each software-defined task is executed in the context of a single 80X86 architecture task.

4.7.3 Task Management Data Structures

The processor defines the following registers and data structures that support multitasking:

- Task-state segment (TSS);
- TSS descriptor;
- Task register (TR);
- Task-gate descriptor;
- NT flag in the EFLAGS register.

Using these data structures, the processor can switch from one task to another while preserving the context of the original task to allow the task to re-execute. When operating in protected mode, a TSS and TSS descriptor must be created for at least one task, and the segment selector for the TSS must be loaded into the task register (using the LTR instruction).

4.7.3.1 Task-State Segment (TSS)

The processor state information for restoring a task execution is saved in a segment called a task state segment TSS (Task state segment). Figure 4-34 shows the format of the TSS used by the 32-bit CPU. The fields in the TSS segment can be divided into two broad categories: dynamic fields and static fields.

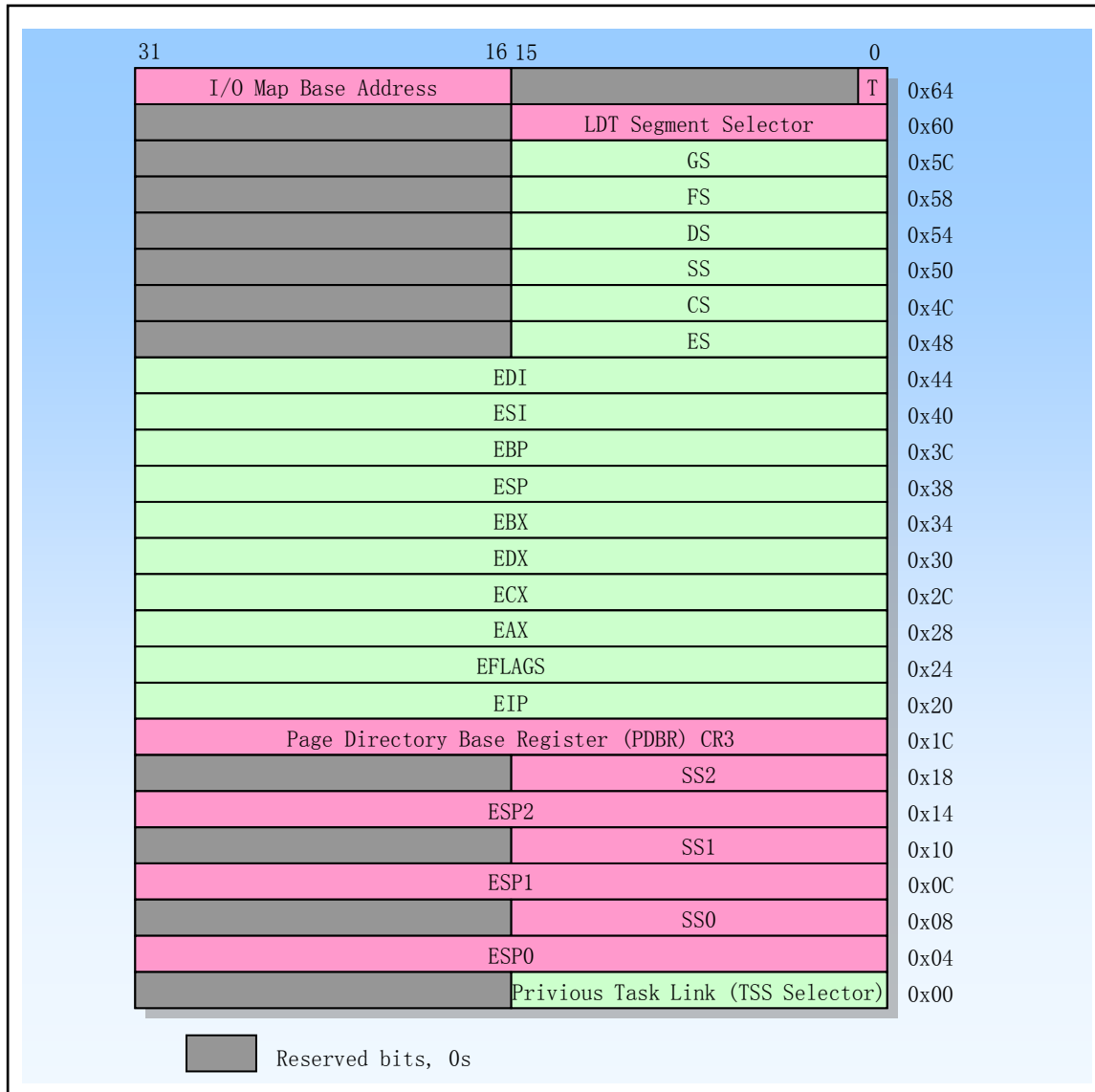


Figure 4-34 32-bit task status segment TSS format

- Dynamic fields. When the task is switched and suspended, the processor updates the contents of the dynamic field. These fields include:
 - General purpose register field. Used to save the contents of the EAX, ECX, EDX, EBX, ESP, EBP, ESI, and EDI registers.
 - Segment selector field. Used to save the contents of the ES, CS, SS, DS, FS, and GS segment registers.
 - Flag register EFLAGS field. Save EFLAGS before switching.
 - Instruction pointer EIP field. Save the contents of the EIP register before switching.
 - The previous task link field. Contains the previous task TSS segment selector (updated during a task switch of a call, interrupt, or exception fire). This field (also commonly referred to as the Back link field) allows the task to switch to the previous task using the IRET instruction.
- Static fields. The processor reads the contents of static fields, but usually does not change them. These field contents are set when the task is created. These fields are:
 - LDT segment selector field. Contains the segment selector for the task's LDT.
 - CR3 Control Register Field. Contains the physical base address of the page directory used by the task.

The control register CR3 is also commonly referred to as a page directory base register (PDBR).

- Stack pointer field for privilege levels 0, 1, and 2. These stack pointers consist of stack segment selectors (SS0, SS1, and SS2) and offset pointers in the stack (ESP0, ESP1, and ESP2). Note that the values of these fields are constant for a given task. Therefore, if a stack switch occurs in a task, the contents of the registers SS and ESP will change.
- Debug Trap T flag field. This field is located at byte 0x64 bit 0. When this bit is set, then a debug exception will be generated when the processor switches to the task.
- I/O bitmap base address field. This field contains the 16-bit offset value from the beginning of the TSS segment to the I/O permission bitmap. When present, these maps are stored at the higher address of the TSS. The I/O mapping base address points to the beginning of the I/O permission bitmap and the end of the interrupt redirection bitmap.

If a paging mechanism is used, the memory page boundaries should be avoided in the TSS segment of the processor operation (in the first 104 bytes) during task switching. If the TSS part contains a memory page boundary, then the pages on both sides of the boundary must exist simultaneously and continuously in physical memory. In addition, if the paging mechanism is used, the pages related to the original task TSS and the new task TSS, and the corresponding descriptor table entries should be readable and writable.

4.7.3.2 TSS Descriptor

Like other segments, the task status segment TSS is also defined using a segment descriptor. Figure 4-35 shows the format of the TSS descriptor. TSS descriptors can only be stored in the GDT.

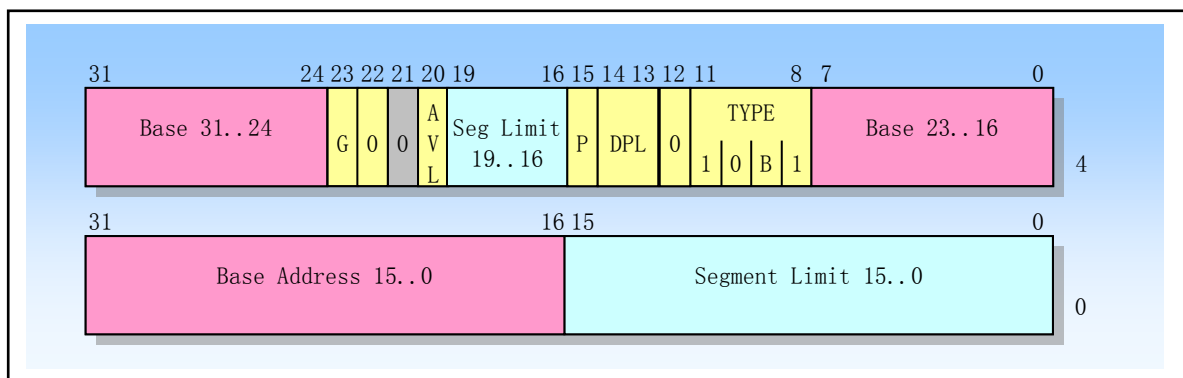


Figure 4-35 TSS segment descriptor format

The busy flag (B) in the type field TYPE is used to indicate whether the task is busy. A busy task is a task that is currently executing or a task that is waiting to be executed (suspended). A type field with a value of 0b1001 indicates that the task is inactive; a type field with a value of 0b1011 indicates that the task is busy. The processor uses the busy flag to detect an attempt to call a task whose execution has been interrupted. To insure that there is only one busy flag is associated with a task, each TSS should have only one TSS descriptor that points to it.

The base address, limit, the descriptor privilege level DPL, the granularity G, and the present flags have the same functions as the corresponding fields in the data segment descriptor. When G=0, the limit field must have a value equal to or greater than 103 (0x67), that is, the minimum length of the TSS segment must not be less than 104 bytes. If the TSS segment also contains an I/O permission bitmap, the TSS segment length needs to be larger. In addition, if the operating system still wants to store some other information in the TSS segment, the TSS segment needs to be larger.

With a call or jump instruction, any program that can access the TSS descriptor can cause a task switch.

Programs that can access the TSS descriptor must have a CPL that is numerically less than or equal to the DPL of the TSS Descriptor. In most systems, the DPL field of the TSS descriptor should be set to less than 3. In this way, only privileged software can perform task switching operations. However, in a multitasking application, the DPL for some TSSs can be set to 3 so that task switching operations can also be performed at the user privilege level.

The ability to access a TSS segment descriptor does not give the program the ability to read and write the descriptor. If you want to read or modify a TSS segment descriptor, you can use a data segment descriptor (ie, an alias descriptor) that maps to the same location in memory. Loading the TSS descriptor into any segment register will result in an exception. Attempting to access the TSS segment using the selector set by the TI flag (ie, the selector in the current LDT) will also cause an exception.

4.7.3.3 Task Register

The task register TR (Task Register) stores a 16-bit segment selector and the entire descriptor (invisible portion) of the current task TSS segment. This information is copied from the TSS descriptor of the current task in the GDT. The processor uses the invisible portion of the task register TR to buffer the contents of the TSS segment descriptor.

The instructions LTR and STR are used to load and save the visible portion of the task register, ie the selector of the TSS segment, respectively. The LTR instruction can only be executed by a privileged level 0 program. The LTR instruction is typically used to load the initial value of the TR register during system initialization (eg, the TSS segment selector for task 0), and then during system operation, the contents of the TR are automatically changed upon task switching.

4.7.3.4 Task-Gate Descriptor

The task gate descriptor provides an indirect, protected reference to a task, as shown in Figure 4-27. The task gate descriptor can be stored in a GDT, LDT, or IDT table.

The TSS Segment Selector field in the Task Gate Descriptor points to a TSS Segment Descriptor in the GDT. The RPL field in this TSS segment selector is not used. The DPL in the task gate descriptor is used to control access to the TSS segment at the time of task switching. When a program makes a call or jump to a task through a task gate, the CPL and the RPL field of the gate selector pointing to the task gate must be less than or equal to the DPL of the task-gate descriptor. Note that when using the task gate, the DPL of the target TSS segment descriptor is ignored.

The program can access a task through a task gate descriptor or a TSS segment descriptor. Figure 4-36 shows how the task gates in the LDT, GDT, and IDT tables all point to the same task.

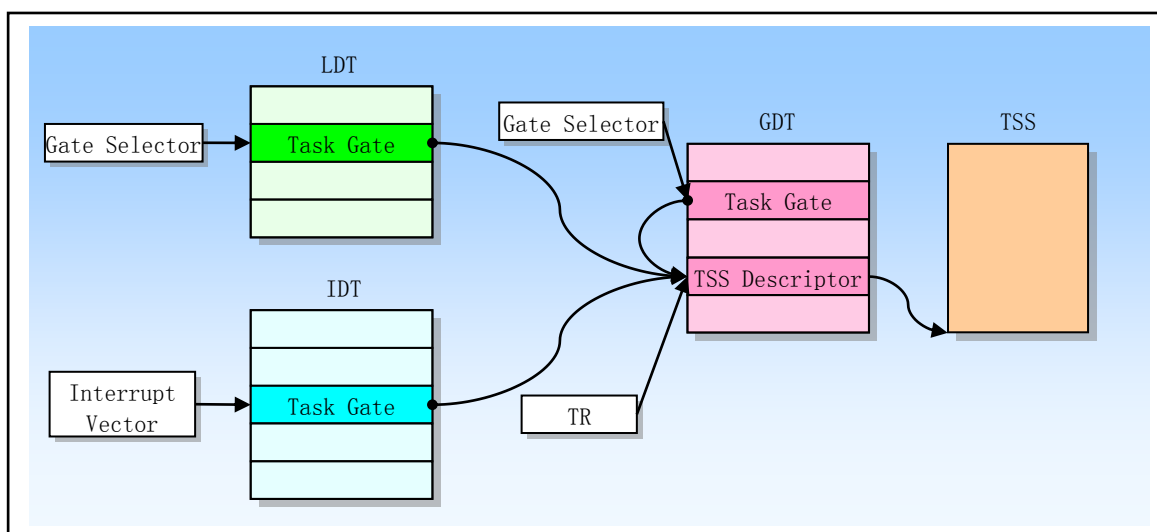


Figure 4-37 Task switching operation diagram

When switching to a new task, the processor does the following:

1. Obtain a TSS segment selector for a new task as the operand of the JMP or CALL instruction, or from the task gate, or from the previous task link field of the current TSS (for task switching caused by IRET).
2. Check if the current task is allowed to switch to the new task. Apply data access privilege rules to JMP and CALL instructions. The CPL of the current task, and the RPL of the new task segment selector must be less than or equal to the DPL of the TSS descriptor or task gate being referenced. Regardless of the DPL of the target task gate or TSS descriptor, exceptions, interrupts (except for interrupts generated using the INT n instruction), and IRET instructions allow task switching. The interrupt generated by the INT n instruction will check the DPL.
3. Check that the TSS descriptor for the new task is marked as present (P=1) and that the limit is valid (greater than 0x67). Any changes to the state of the processor are resumed when attempting to execute an instruction that will generate an error. This causes the return address of the exception handler to point to the error instruction instead of the next instruction of the error instruction. Therefore the exception handler procedure can handle the error condition and re-execute the task. The intervention of the exception handler procedure is completely transparent to the application.
4. If the task switch is generated from a JMP or IRET instruction, the processor will reset the busy flag B in the current task (old task) TSS descriptor; if the task switch is generated by a CALL instruction, exception or interrupt, the busy flag B Will not be changed.
5. 5. If the task switch is initiated with an IRET instruction, the processor resets the NT flag in a temporarily saved EFLAGS image; if the task switch is initiated with a CALL, JMP instruction, or an exception or interrupt, the NT flag is left unchanged in the saved EFLAGS image.
6. Save the state of the current (old) task to the TSS of the current task. The processor retrieves the base address of the current task TSS from the task register and copies the states of the following registers into the current TSS: all general purpose registers, the segment selector in the segment register, the flag register EFLAGS, and the instruction pointer EIP.
7. If the task switch was initiated with a CALL instruction, an exception, or an interrupt, the processor sets the NT flag in the EFLAGS image stored in the new task's TSS; if initiated with an IRET instruction, the processor restores the NT flag from the EFLAGS image stored on the stack. If initiated with a JMP instruction, the NT flag is left unchanged.
8. If the task switch was initiated by a CALL, JMP instruction, or exception or interrupt, the processor sets the busy flag B in the new task TSS descriptor. If the task switch is generated by the IRET, the B flag is not changed.
9. Load the task register TR (including the hidden portion) using the segment selector and descriptor of the new task TSS. Set the TS flag in the control register CR0 image stored in the new task's TSS.
10. 10. Load the TSS status of the new task into the processor. This includes the LDTR register, the PDBR (CR3) register, the EFLAGS register, the EIP register, and general purpose registers and segment selectors. Any errors detected during this time will appear in the context of the new task.
11. Begins executing the new task. (To an exception handler, the first instruction of the new task appears not to have been executed.)

When the task switching operation is successfully performed, the state of the currently executing task is always saved. When the task resumes execution, the task will execute from the instruction pointed to by the saved EIP, and all registers will be restored to the value when the task was suspended.

When performing a task switch, the privilege level of the new task has nothing to do with the privilege level of the original task. The new task starts running at the privilege level specified by the CPL field of the CS register, which is loaded from the TSS. Because each task is isolated from each other by their independent address space and TSS segments, and the privilege level rules already control access to the TSS, the software does not need to perform privilege level checks at the time of task switching.

The task switching flag TS in the control register CR0 is set each time the task is switched. This flag is very useful for system software. The system software can use the TS flag to coordinate operations between the processor and the floating point coprocessor. The TS flag indicates that the context in the coprocessor may be different from that of the current task.

4.7.5 Task Linking

The TSS's previous task link field (Backlink) and the NT flag in EFLAGS are used to return execution to the previous task. The NT flag indicates whether the currently executing task is nested within the execution of another task, and the previous task link field of the current task holds the TSS selector for the higher level task in the nesting hierarchy, if there is one (see figure 4-38).

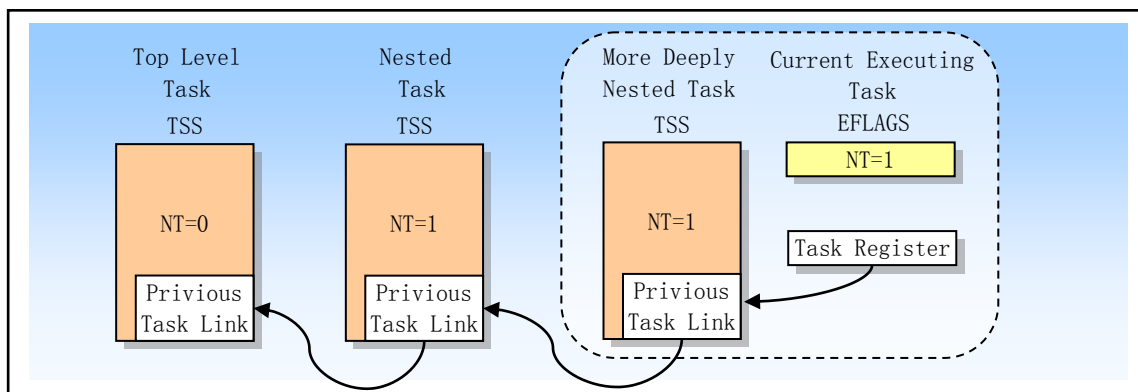


Figure 4-38 Nested Tasks

When a CALL instruction, interrupt, or exception causes a task switch, the processor copies the selector of the current TSS segment to the previous task link field of the new task TSS segment, and then sets the NT flag in EFLAGS. The NT flag indicates that the saved TSS segment selector is stored in the previous task link field of the TSS. If the software suspends a new task using the IRET instruction, the processor will return to the previous task using the value in the previous task link field and the NT flag. That is, if the NT flag is set, the processor will switch to the task specified in the previous task link field to execute.

Note that when the task switch is caused by a JMP instruction, the new task will not be nested. That is, the NT flag will be set to 0 and the previous task link field will not be used. JMP instructions are used in task switching where nesting is not desired.

Table 4-10 summarizes the usage of busy flag B (in the TSS segment descriptor), the NT flag, the previous task link field, and the TS flag (in CR0) during task switching. Note that programs running at any privilege level can modify the NT flag, so any program can set the NT flag and execute the IRET instruction. This approach will cause the processor to perform the tasks specified in the previous task link field of the current task TSS. In order to avoid successful execution of such forged task switching, the operating system should initialize this field of each TSS to zero.

Table 4-10 Effects of a task switch on Busy flag, NT flag, previous task link field, and TS flag

Flag or Field	Effect of JMP	Effect of CALL or Interrupt	Effect of IRET
Busy (B) flag of new task.	Flag is set. Must have been clear before.	Flag is set. Must have been clear before.	No change. Must have been set.
Busy flag of old task.	Flag is cleared.	No change. Flag is currently set.	Flag is cleared.
NT flag of new task.	No change.	Flag is set.	Restored to value from TSS of new task.
NT flag of old task.	No change.	No change.	Flag is cleared.
Previous task link field of new task.	No change.	Loaded with selector for old task's TSS	No change.
Previous task link field of old task.	No change.	No change.	No change.
TS flag in CR0	Flag is set.	Flag is set.	Flag is set.

4.7.6 Task Address Space

The address space of a task consists of the segments that the task can access. These segments include the code referenced in the TSS, the data, the stack and system segments, and any other segments accessed by the task code. These segments are mapped to the processor's linear address space and then mapped to the processor's physical address space (either directly or through paging).

The LDT field in the TSS can be used to give each task its own LDT. For a given task, by putting all the segment descriptors associated with the task into the LDT, the task's address space can be isolated from other tasks. Of course, several tasks can also use the same LDT. This is a simple and effective way to allow certain tasks to communicate or control each other without having to discard the entire system's protective barrier. Because all tasks have access to the GDT, it is also possible to create shared segments that are accessed through this table.

If the paging mechanism is enabled, the CR3 register (PDBR) field in the TSS allows each task can also have its own set of page table for mapping linear addresses to physical addresses. Or, several tasks can share the same set of page tables.

4.7.6.1 Mapping Tasks to the Linear and Physical Address Spaces

There are two ways to map tasks to linear address spaces and physical address spaces:

- All tasks share a linear to physical address space mapping. This method can only be used when the paging mechanism is not enabled. When paging is not turned on, all linear addresses are mapped to the same physical address. When the paging mechanism is turned on, we can use this mapping from linear to physical address space by having all pages use a single page directory. If the demand page virtual storage technology is supported, the linear address space can exceed the size of the existing physical address space.
- Each task has its own linear address space and is mapped to the physical address space. We can use this mapping form by having each task use a different page directory. Because PDBR (Control Register CR3) is loaded every time a task is switched, each task can have a different page directory.

Linear address spaces for different tasks can be mapped to completely different physical addresses. If the entries (table entries) of different page directories point to different page tables, and the page tables also point to different pages in the physical address, then each task will not share any physical address.

For both methods of mapping task linear address space, the TSSs for all tasks must be stored in the shared physical address space area, and all tasks can access this area. In order for the processor to perform task switching

and the TSS address mapping does not change when reading or updating the TSS, this mapping method is required. The linear address space mapped by the GDT should also be mapped to the shared physical address space. Otherwise, the role of the GDT is lost.

4.7.6.2 Task Logical Address Space

To share data between tasks, use one of the following methods to establish a shared logical-to-physical address space mapping for the data segment:

- By using the segment descriptor in the GDT. All tasks must be able to access the segment descriptors in the GDT. If some segment descriptors in the GDT point to segments in the linear address space and these segments are mapped into the physical address space shared by all tasks, then all tasks can share the code and data in those segments.
- Through a shared LDT. Two or more tasks can use the same LDT if their LDT fields in their TSSs point to the same LDT. If some segment descriptors in a shared LDT point to segments mapped to a common area of the physical address space, then all tasks sharing the LDT can share all of the code and data in those segments. This kind of sharing is better than sharing through GDT, because doing so can limit sharing to certain tasks. There are other tasks in the system that do not have access to these shared segments.
- Segment descriptors in different LDTs mapped to the common address area in the linear address space. If this common area in the linear address space maps each task to the same area of the physical address space, then these segment descriptors allow tasks to share the segments. Such segment descriptors are often referred to as alias segments. This sharing method is better than the one given above, because other segment descriptors in the LDT can point to separate unshared linear address regions.

4.8 The Initialization of Protected Mode

When the machine is powered on or hardware reset, the processor operates in the 8086-compatible real-address mode and executes the software initialization code starting at physical address 0xFFFFFFF0 (usually in EPROM). The software initialization code must first set the necessary data structure information for basic system function operations, such as real-mode IDT tables (ie, interrupt vector tables) that handle interrupts and exceptions. If the processor will still work in real mode, the software must load the operating system modules and corresponding data to allow the application to run reliably in real mode. If the processor is going to work in protected mode, then the operating system software must load the data structure information necessary to protect the mode and then switch to protected mode.

4.8.1 First Instruction Executed

As described above, the first instruction acquired and executed after the hardware reset is located at the physical address 0xFFFFFFF0. This address is at the 16 bytes of the processor's highest physical address. This is usually the address range in which the EPROM firmware containing the software initialization code is located.

In real address mode, the address 0xFFFFFFF0 is outside the processor's 1 MB addressable range. The processor initializes to the starting address in the following manner. The CS register has two parts: the visible segment selector portion and the hidden base portion. In real address mode, the base address is typically formed by shifting the 16-bit segment selector value to the left by 4 bits to produce a 20-bit base address. However, during hardware reset, the segment selector in the CS register is loaded as 0xF000 and the base address is loaded as 0xFFFF0000. Therefore, the start address is formed by adding the base address to the EIP register (ie, $0xFFFF0000 + 0xFFF0 = 0xFFFFFFF0$).

When the CS register first loads a new value after a hardware reset, the processor will follow the normal

rules for address translation in real-address mode (ie, [CS base address = CS segment selector * 16]). To ensure that the base address in the CS register remains unchanged until the EPROM-based software initialization code is completed, the code must not contain a far-hop or remote call or allow an interrupt to occur (this will cause the CS selector value to change).

4.8.2 Initialization operation when entering protection mode

The processor is in real address mode after a hardware reset. Some basic data structures and code modules must be loaded into physical memory during initialization to support further initialization of the processor. Some of the data structures required for protected mode are determined by the processor memory management function. The processor supports a segmentation model that can be used from a single, unified address space flat model to a highly structured multi-segment model with several protected address spaces per task. The paging mechanism can be used to process large pieces of data structure information that is partially in memory and partially on disk. Both forms of address translation require the operating system to set the required data structure for the memory management hardware in memory. Therefore, before the processor can be switched to protected mode, the operating system's loading and initialization software (bootsect.s, setup.s, and head.s) must first set the basics of the data structure used in protected mode in memory. These data structures include the following:

- A protected-mode interrupt descriptor table IDT;
- A global descriptor table GDT;
- A task status segment TSS;
- A local descriptor table LDT;
- If paging is enabled, at least one page directory and one page table need to be set;
- A code segment containing execution code for the processor to switch to protected mode;
- Code modules that contain interrupts and exception handlers.

The software initialization code must also set the following system registers before being able to switch to protected mode:

- Global descriptor table base address register GDTR;
- Interrupt descriptor table base address register IDTR;
- Control register CR1--CR3;

After initializing these data structures, code modules, and system registers, the processor can be switched to protected mode by setting the protection mode flag PE (bit 0) of the CR0 register.

4.8.2.1 Protection Mode System Structure Table

The protected mode system table set in memory during software initialization relies primarily on the type of memory management that the operating system will support: flat, flat with paging, segmentation, or segmentation with paging.

In order to implement a flat memory model without paging, the software initialization code must at least set up a GDT table with one code segment and one data segment. Of course, the first item of the GDT table also needs to place a null descriptor. The stack can be placed in normal readable and writable data segments, so no special stack descriptors are needed. A flat memory model that supports the paging mechanism also requires a page directory and at least one page table. Before the GDT table can be used, the base address and limit for the GDT must be loaded into the GDTR register using the LGDT instruction.

Multi-segment models also require additional segments for the operating system, as well as segments and LDT table segments for each application. The segment descriptors of the LDT table are required to be stored in the GDT table. Some operating systems will allocate new segments and new LDT segments for the application. This approach provides maximum flexibility for dynamic programming environments, such as the Linux

operating system. An embedded system like a process controller can pre-allocate a fixed number of segments and LDTs for a fixed number of applications, which is a simple and efficient way to implement the real-time system software environment structure.

4.8.2.2 Exceptions and Interrupt Initialization in Protected Mode

The software initialization code must set a protection mode IDT, which at least needs to contain the gate descriptor corresponding to each exception vector that the processor may generate. If an interrupt or trap gate is used, the gate descriptor can all point to the same code segment containing the interrupt and exception handling. If a task gate is used, each exception handling process that uses the task gate requires a TSS and associated code, data, and stack segments. If the hardware is allowed to generate an interrupt, then the gate descriptor must be set in the IDT for one or more interrupt handlers.

The IDT table base address and limit length must be loaded into the IDTR register using the LIDT instruction before the IDT can be used.

4.8.2.3 Paging initialization

The paging mechanism is set by the PG flag in the control register CR0. When this flag is cleared to 0 (ie, the state at the time of hardware reset), the paging mechanism is turned off; when the PG flag is set, the paging mechanism is turned on. The following data structures and registers must be initialized before setting the PG flag:

- Software must create at least one page directory and one page table in physical memory. If the page directory table contains a entry that points to itself, then you can eliminated the use of page table. At this point, the page directory table and the page table are stored on the same page.
- Load the physical base address of the page directory table into the CR3 register (also known as the PDBR register).
- The processor is in protected mode. If all other restrictions are met, the PG and PE flags can be set at the same time.

To maintain compatibility, the following rules must be observed when setting the PG flag (and PE flag):

- Instructions that set the PG flag should follow a JMP instruction immediately. The JMP instruction following the MOV CR0 instruction changes the execution stream, so it clears the instructions that 80X86 processor has taken or decoded. However, the Pentium and above processors use the Branch Target Buffer (BTB) for branch code orientation, thus eliminating the need to refresh the queue for branch instructions.
- The code that sets the PG flag to the jump instruction JMP must come from a page on the peer mapping (that is, the linear address before the jump is the same as the physical address after the paging is turned on).

4.8.2.4 Multitasking Initialization

If the multitasking mechanism is to be used and/or the privilege level is allowed to change, then the software initialization code must have at least one TSS and the corresponding TSS segment descriptor (because the stack segment pointers for privilege levels 0, 1, and 2 need to be taken from the TSS). Do not mark the TSS descriptor as busy (do not set the busy flag), which is only set by the processor when performing task switching. Like the LDT segment descriptor, the TSS descriptor is also stored in the GDT.

After the processor switches to protected mode, the selector of the TSS segment descriptor can be loaded into the task register TR using the LTR instruction. This command marks the TSS as busy ($B = 1$), but does not perform a task switch operation. The processor can then use this TSS to locate the stack of privilege levels 0, 1, and 2. In protected mode, the selector of the TSS segment must be loaded first before the software performs the first task switch, because the task switch will copy the current task state into the TSS.

Subsequent to the LTR instruction execution, subsequent operations on the task register are performed by

task switching. Similar to other segments and LDTs, TSSs and TSS descriptors can be pre-allocated or allocated when needed.

4.8.3 Mode Switching

In order for the processor to operate in protected mode, mode switching must be performed from real address mode. Once in protected mode, the software usually no longer needs to go back to real address mode. In order to be able to run programs programmed for real-address mode, it is usually more convenient to run in virtual-8086 mode than to switch back to real mode.

4.8.3.1 Switching to Protected Mode

Before switching to protected mode, you must first load some minimum system data structures and code modules. Once these system tables are created, the software initialization code can be switched to protected mode. By executing the MOV CR0 instruction that sets the PE flag in the CR0 register, we can enter the protection mode. (In the same instruction, the PG flag of CR0 can be used to enable the paging mechanism.) When running in protected mode, the privilege level CPL is 0. In order to ensure the compatibility of the program, the switching operation should be carried out as follows:

1. Disable interrupts. Maskable hardware interrupts can be disabled using the CLI instruction. The NMI is disabled by hardware circuitry. At the same time, the software should ensure that no exceptions or interruptions occur during mode switching operations.
2. Execute the LGDT instruction to load the base address of the GDT table into the GDTR register.
3. Execute the MOV CR0 instruction that sets the PE flag (optional setting of the PG flag) in the control register CR0.
4. Execute a far jump JMP or a far call CALL instruction immediately after the MOV CR0 instruction. This operation is usually a far jump to or far from the next instruction in the instruction stream.
5. If a local descriptor table is to be used, execute the LLDT instruction to load the LDT segment selector into the LDTR register.
6. Execute the LTR instruction to load the task register TR with the segment selector of the initial protected mode task or the segment descriptor of the writable memory area. This writable memory area is used to store the TSS information of the task when the task is switched.
7. After entering protected mode, the segment register still contains the contents in real address mode. The JMP or CALL instruction in step 4 resets the CS register. Do one of the following to update the contents of the remaining segment registers: (1) Reload registers DS, SS, ES, FS, and GS. If the ES, FS, and/or GS registers are not used, load them with a null selector. (2) Execute a JMP or CALL instruction on a new task that automatically resets the value of the segment register and branches to the new code segment.
8. Execute the LIDT instruction to load the base address and limit of the protected mode IDT table into the IDTR register.
9. Execute the STI instruction to turn on the maskable hardware interrupt and perform the necessary hardware operations to turn on the NMI interrupt.

In addition, the JMP or CALL instruction immediately following the MOV CR0 instruction changes the execution flow. If the paging mechanism is enabled, the code between the MOV CR0 instruction and the JMP or CALL instruction must come from a page on the peer mapping (ie, the linear address before the jump is the same as the physical address after paging). The target instruction to which the JMP or CALL instruction jumps does not need to be on the peer mapping page.

4.8.3.2 Switch back to real address mode

If you want to switch back to real address mode, you can use the MOV CR0 instruction to clear the PE flag in control register CR0. The process of re-entering the real address mode should follow these steps:

1. Disable interrupts. Maskable hardware interrupts can be disabled using the CLI instruction. The NMI is disabled by hardware circuitry. At the same time, the software should ensure that no exceptions or interruptions occur during mode switching operations.
2. If the paging mechanism is enabled, you need to execute:
 - Transfer the program control to linear address of the peer map (ie the linear address is equal to the physical address).
 - Make sure the GDT and IDT are on the peer mapped page.
 - Clear the PG flag in CR0.
 - Set to 0x00 in the CR3 register to refresh the TLB buffer.
3. Transfer the control of the program to a readable segment of length 64KB (0xFFFF). This step loads the CS register using the segment limit required by the real mode.
4. Load the SS, DS, ES, FS, and GS segment registers with a selector that points to a descriptor with the following set values.
 - Limit = 64KBytes (0xFFFF).
 - Byte granularity (G = 0).
 - Expand up (E=0).
 - Writable (W=1).
 - Present (P=1).
5. Execute the LIDT instruction to point to the real address mode interrupt table in the 1MB real mode address range.
6. Clear the PE flag in CR0 to switch to real address mode.
7. Execute a far jump instruction to jump to a real mode program. This step refreshes the instruction queue and loads the appropriate base address and access value for the CS register.
8. The SS, DS, ES, FS, and GS registers are loaded as needed for the real address mode code. If any of the registers are not used in real address mode, write 0 to them.
9. Execute the STI instruction to turn on the maskable hardware interrupt and perform the necessary hardware operations to turn on the NMI interrupt.

4.9 A Simple Multitask Kernel Example

As a summary of this chapter and the previous chapters, this section provides a complete description of the design and implementation of a simple multitasking kernel. This kernel example contains two privilege level 3 user tasks and a privilege level 0 system call interrupt procedure. We first explain the basic structure of this simple kernel and the basic principles of load operation, then we describe how it is loaded into the machine's RAM memory and how the two tasks are switched. Finally, we give the source code to implement this simple kernel: boot boot.s and protected mode multitasking kernel program head.s.

4.9.1 Multitasking program structure and working principle

The kernel example given in this section consists of two source files. One is the bootloader boot.s compiled with the as86 language, which is used to load kernel code into the memory from the boot disk when the computer system is powered up; the other is the kernel program head.s compiled with GNU as assembly language. It

implements two tasks running on the privilege level 3 to switch between each other under the control of the clock interrupt, and also implements a system call for displaying characters on the screen.

We refer to these two tasks as task A and task B (or task 0 and task 1). They call the system call to display the characters 'A' and 'B' on the screen, respectively, and switch to another task every 10 milliseconds. Task A continuously calls the system call to display the character 'A' on the screen; task B always displays the character 'B'. To terminate this kernel instance program, you will need to restart the machine or shut down the running simulated PC runtime environment software.

The `boot.s` program generates a total of 512 bytes of code that will be stored in the first sector of the floppy image file, as shown in Figure 4-39. When the PC is powered on, the program in the ROM BIOS will load the first sector on the boot disk to the beginning of the physical memory 0x7c00 (31KB) position, and transfer the execution control to 0x7c00 to start running the boot code.

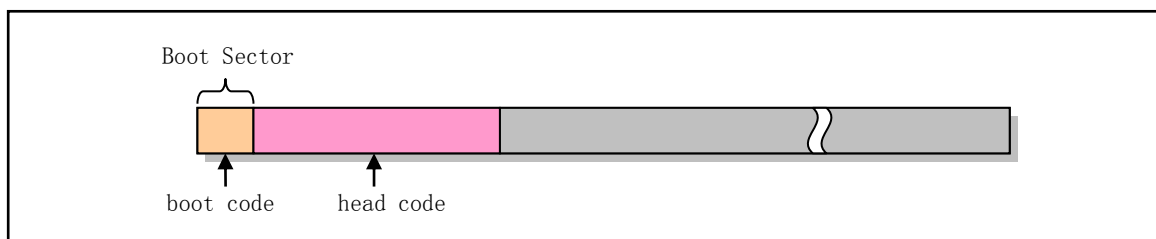


Figure 4-39 Floppy disk image file

The main function of the boot program is to load the head kernel code in the floppy disk or image file to a specified location in the memory. After setting the temporary GDT table and other information, set the processor to run in protected mode. Then jump to the head code to run the kernel code. In fact, the `boot.s` program will first use the ROM BIOS interrupt `int 0x13` to read the head code in the floppy disk to the beginning of the memory location 0x10000 (64KB), and then move the head code to the beginning of the memory location 0. Finally, the enable protection operation mode flag in the control register `CRO` is set, and jumps to the memory location 0 to start executing the head code. A schematic diagram of the boot code moving the head code in memory is shown in Figure 4-40.

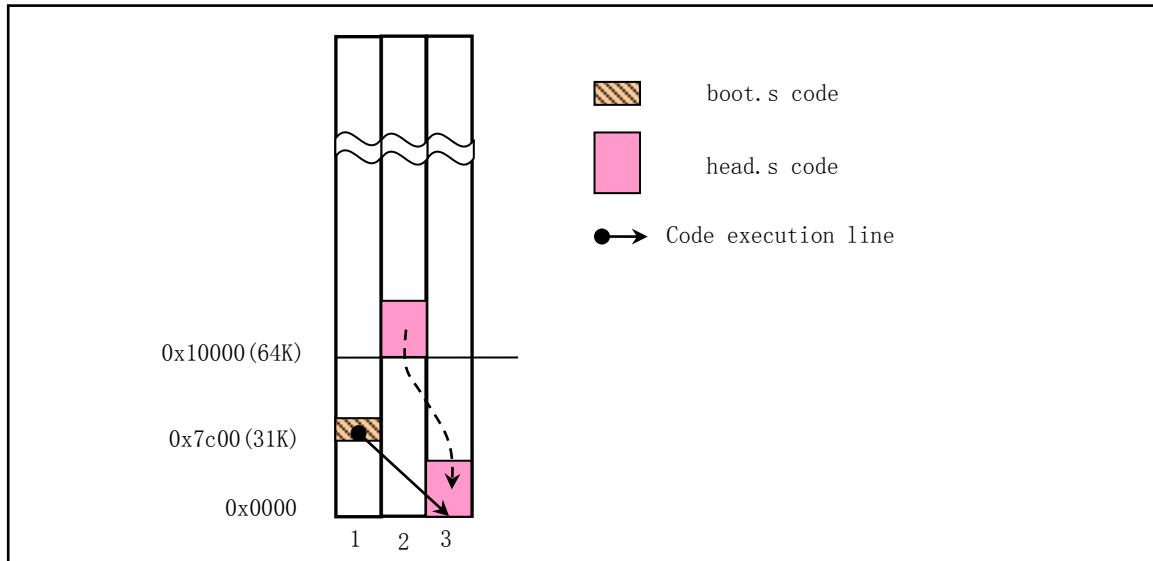


Figure 4-40 The movement and distribution of kernel code in physical memory

The main reason for moving the head kernel code to the beginning of physical memory 0 is that it is simpler to set up the GDT table, so it can also make the head.s program as short as possible. But we can't let the boot program load the head code directly from the floppy or image file into memory location 0. Because the interrupt vector table used by the BIOS is at the beginning of memory 0, and the load operation requires the use of the interrupt process provided by the ROM BIOS. So if you load the head code directly into memory 0, the BIOS interrupt vector table will be destroyed.

Of course, we can also load the head code into the memory 0x10000 and then jump directly to the location to run the head code. The source program using this method can be downloaded from the oldlinux.org website, as explained below.

The head program runs in 32-bit protected mode, which mainly includes the code of the initial setting, the process code of the clock interrupt int 0x08, the process code of the system call interrupt int 0x80, and the code and data of task A and task B. The initial setting work mainly includes: (1) resetting the GDT table; (2) setting the system timer chip; (3) resetting the IDT table and setting the clock and system call interrupt gate; (4) moving to the task A for execution.

In the virtual address space, the kernel and task code allocation diagram for the head.s program is shown in Figure 4-41. In fact, all the code and data segments in this kernel example correspond to the same area of physical memory, that is, the area starting from physical memory location 0.

The contents of the global code segment and the data segment descriptor in the GDT are set to as following:

- The base address is 0x0000;
- The segment limit value is 0x07ff. Since the granularity is 1, the actual segment length is 8MB.

The global display data segment is set as following:

- The base address is 0xb8000;
- The segment limit length is 0x0002, so the actual segment length is 8KB, corresponding to the display memory area.

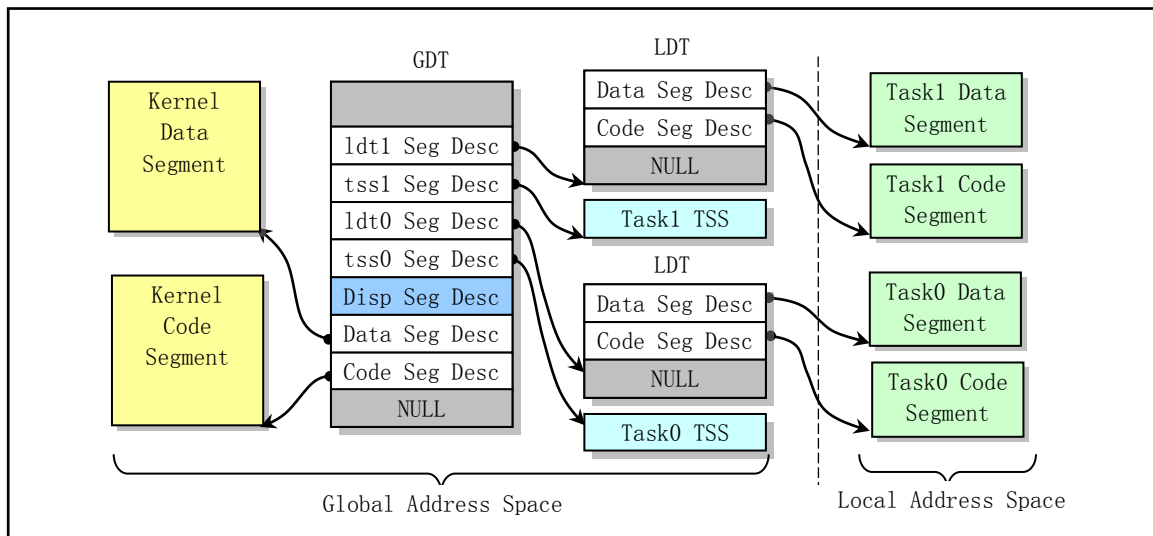


Figure 4-41 Diagram of the allocation of kernel and tasks in the virtual address space

The contents of the code segment and data segment descriptors in the LDT for both tasks are also set as follows:

- The base address is 0x0000;
- The segment length is 0x03ff and the actual segment length is 4MB.

Therefore, in the linear address space, the code and data segments of this "core" and the code and data segments of the task start from linear address 0 and since they do not use the paging mechanism, they all directly correspond to the beginning of physical address 0. In the object file compiled by the head program and the resulting floppy image file, the organization of the code and data is shown in Figure 4-42.

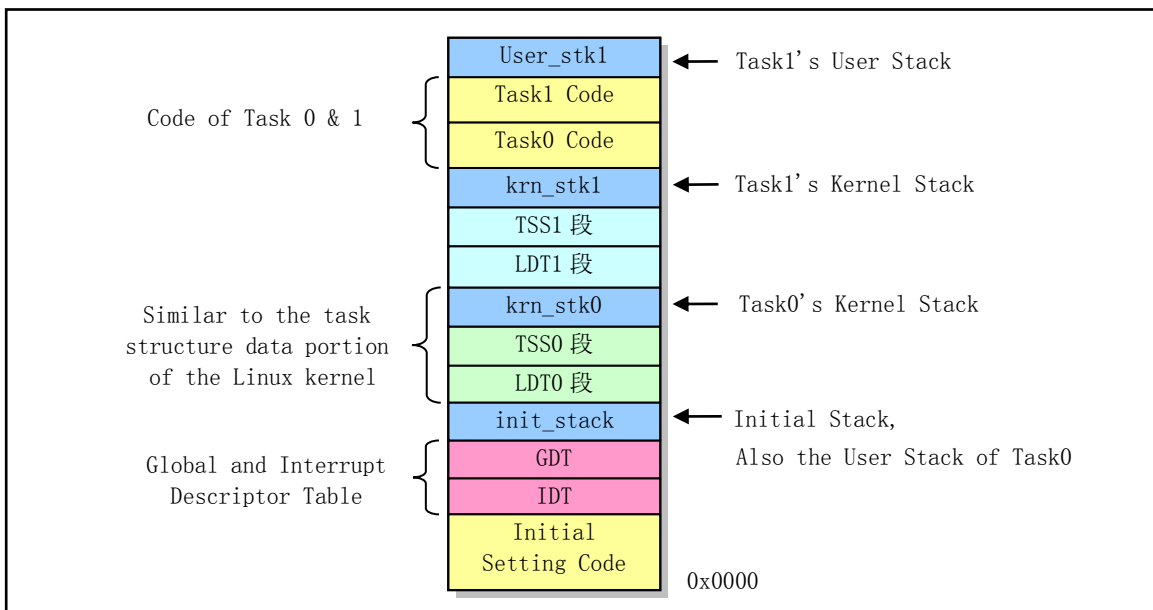


Figure 4-42 Kernel image file and in-memory head code and data diagram

Code at privilege level 0 cannot be directly transferred to the code of privilege level 3,

but control transfer can be achieved by using the interrupt return operation. So when the GDT, IDT, and timing chips are initialized, we use the interrupt return instruction IRET to start the first task.

The specific implementation method is to manually set a return environment in the initial stack `init_stack`. That is, after loading the TSS segment selector of task 0 into the task register LTR and loading the LDT segment selector into the LDTR, the user stack pointer of task 0 (`0x17: init_stack`), code pointer (`0x0f: task0`), and the flag register value is pushed onto the stack, and then the interrupt return instruction IRET is executed.

The IRET instruction pops the stack pointer on the stack as the task 0 user stack pointer, restores the contents of the hypothetical task 0 flag register, and pops the code pointer in the stack into the CS:EIP register, thus starting the execution of the task 0 code. This completes the control transfer from privilege level 0 to privilege level 3 code.

In order to switch the running task every 10 milliseconds, the channel 0 of the timer chip 8253 is set in the `head.s` program to send a clock interrupt request signal to the interrupt control chip 8259A every 10 milliseconds. When PC is powered on, the ROM BIOS program has set the clock interrupt request signal to the interrupt vector 8 in the 8259A, so we need to perform the task switching operation in the interrupt 8 handler procedure. The task switching method is implemented by looking at the current running task number in the `current` variable. If `current` is currently 0, the TSS selector of task 1 is used as the operand to execute the far jump instruction, thereby switching to task 1 for execution, otherwise vice versa.

Each task will first put the ASCII code of a character into the register AL, and then call the system interrupt to call `int 0x80`. The system call processing process will call a simple character write screen subroutine, display the characters in the register AL on the screen, and record the next position of the screen displaying the characters as the screen position for displaying the characters next time. After a character has been displayed, the task code is delayed for a period of time using the loop statement, and then jumps to the beginning of the task code to continue the loop until a timed interrupt occurs for 10 milliseconds. At this point the code will switch to another task to run. For task A, the character 'A' will always be stored in the register AL, and the character 'B' will always be stored in the AL during the task B runtime. Therefore, when the program is running, we will see a series of characters 'A' and a series of characters 'B' displayed continuously on the screen at intervals, as shown in Figure 4-43.

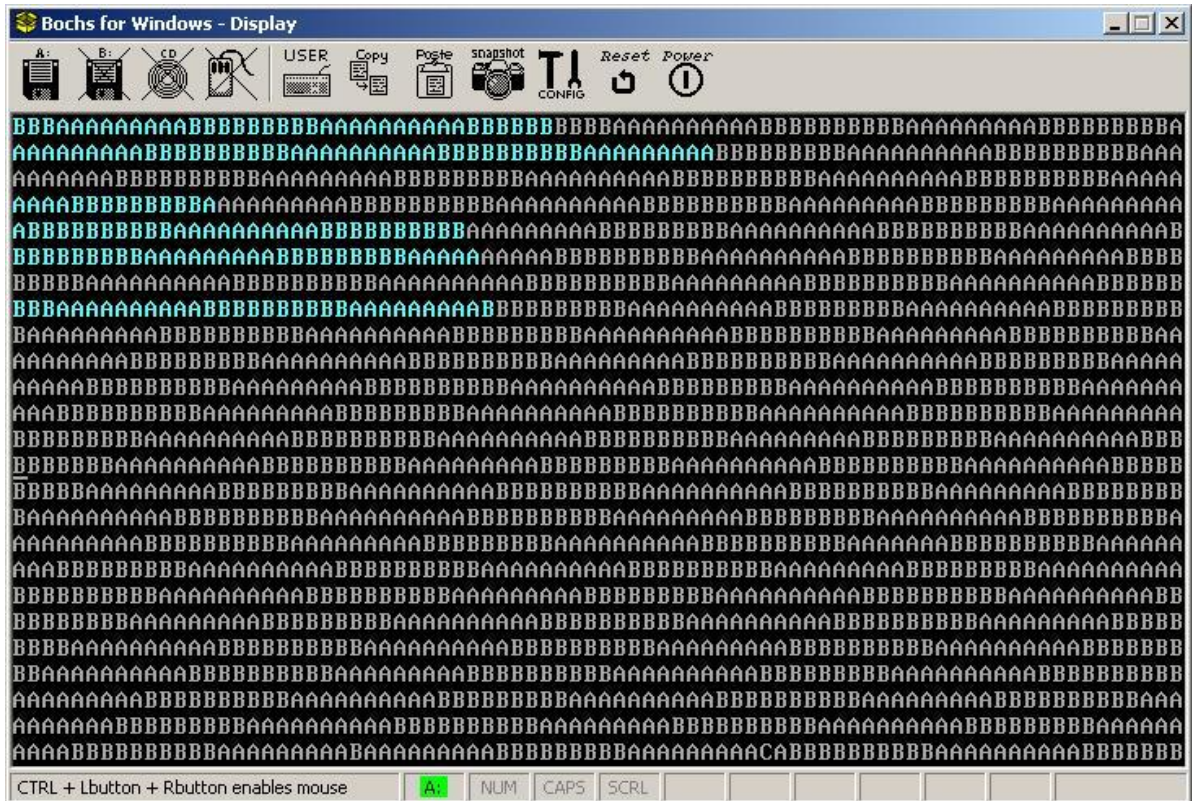


Figure 4-43 Simple kernel running screen display

Figure 4-43 shows the screen display of this running kernel in Bochs. Careful readers will find that a character 'C' is displayed on the bottom line of the figure. This is because the PC accidentally generates another interrupt that is not a clock interrupt or a system call interrupt. Because we have installed a default interrupt handler for all other interrupts in the program. When a different interrupt occurs, the system will run the default interrupt handler with code to display character 'C', so a character 'C' will be displayed on the screen and the interrupt will be exited.

Detailed comments on the boot.s and head.s programs are given below. For the compilation and operation of this simple kernel example, please refer to the section "Compiling and Running a Simple Kernel Sample Program" in the last chapter of this book.

4.9.2 Boot Startup Program boot.s

In order to make the program as simple as possible, this bootloader can only load the head code of no more than 16 sectors, and directly use the interrupt vector number set by the ROM BIOS, that is, the interrupt number of the timer interrupt request number is still 8. This is different from what is used in Linux systems. The Linux system resets the 8259A interrupt control chip during kernel initialization and maps the clock interrupt request signal to interrupt 0x20. See the chapter "Kernel Boot Program" for details.

01 ! boot.s program

02 ! First use BIOS interrupt to load head code into memory 0x10000, and then move it to memory 0.

03 ! Finally enter protected mode, jump to the beginning of head code at 0 to continue running.

```

04 BOOTSEG = 0x07c0          ! This program is loaded into memory at 0x7c00 by BIOS.
05 SYSSEG  = 0x1000          ! The head is first loaded to 0x10000 and moved to 0x0.
06 SYSLEN  = 17              ! Max num of disk sectors occupied by the kernel.
07 entry start
08 start:
09     jmp     go, #BOOTSEG    ! Jump between segments to 0x7c0:go. All segment reg
10 go:     mov     ax, cs      ! are 0 when runs. This jump ins load CS with 0x7c0.
11         mov     ds, ax      ! Both DS and SS point to the 0x7c0 segment.
12         mov     ss, ax
13         mov     sp, #0x400  ! Set temp stack pointer. Its value needs to be larger
14                               ! than this program and has a certain space.
15 ! Load the kernel code to the beginning of memory at address 0x10000.
16 load_system:
17     mov     dx, #0x0000     ! Use BIOS int 0x13 func2 to load head code from bootdisk.
18     mov     cx, #0x0002     ! DH-head no; DL-drive no; CH-10 bit track no low 8 bits
19     mov     ax, #SYSSEG     ! CL-Bits7,6 track num high 2 bits ,bit 5-0 start sector
20     mov     es, ax          ! ES:BX - Read in buffer location (0x1000:0x0000).
21     xor     bx, bx          ! AH-read sector func num; AL- num of sectors read (17).
22     mov     ax, #0x200+SYSLEN
23     int     0x13
24     jnc     ok_load         ! If no error occurs, then continues, otherwise dead.
25 die:     jmp     die
26
27 ! Move kernel code to memory location 0. Total of 8KB bytes are moved (kernel code <8kb).
28 ok_load:
29     cli                     ! Disable interrupts.
30     mov     ax, #SYSSEG     ! Move from DS:SI(0x1000:0) to ES:DI(0:0).
31     mov     ds, ax
32     xor     ax, ax
33     mov     es, ax
34     mov     cx, #0x1000     ! Set the move 4K times, one word at a time.
35     sub     si, si
36     sub     di, di
37     rep     movw            ! Execute the repeat move instruction.
38 ! Load IDT and GDT base address registers IDTR and GDTR.
39     mov     ax, #BOOTSEG
40     mov     ds, ax          ! Let DS point to 0x7c0 segment again.
41     lidt    idt_48          ! Load IDTR. 2 byte table limit, 4 byte linear base addr.
42     lgdt    gdt_48          ! Load GDTR. 2 byte table limit, 4 byte linear base addr.
43
44 ! Set CRO to enter protection mode. The seg selector value 8 refers to 2nd descriptor in GDT.
45     mov     ax, #0x0001     ! Set the protection mode flag PE (bit 0) in CRO.
46     lmsw    ax              ! Jump to segment specified by the selector, offset 0.
47     jmp     0, 8            ! Seg value is now a selector. The linear base addr is 0.
48
49 ! The following is the content of GDT. It has 3 seg descriptors. The first one is not used,
    ! the other two are code and data segment descriptors.
50 gdt:     .word    0, 0, 0, 0    ! First descriptor not used. Occupies 8 bytes.
51
52         .word    0x07FF         ! Descriptor 1. 8Mb - limit=2047 (2048*4096=8MB).
53         .word    0x0000         ! Segment base address = 0x00000.
54         .word    0x9A00         ! Code segment, readable/executable.
55         .word    0x00C0         ! Segment attribute granularity = 4KB, 80386.

```

```
56
57     .word    0x07FF          ! Descriptor 1. 8Mb - limit=2047 (2048*4096=8MB).
58     .word    0x0000          ! Segment base address = 0x00000.
59     .word    0x9200          ! Data segment, readable and writable.
60     .word    0x00C0          ! Segment attribute granularity = 4KB, 80386.
61 ! The following are the 6-byte operands of the LIDT and LGDT instructions, respectively.
62 idt_48: .word    0           ! The IDT table length is 0.
63     .word    0,0             ! The linear base address of IDT table is also zero.
64 gdt_48: .word    0x7ff       ! GDT limit is 2048 bytes, can hold 256 descriptors.
65     .word    0x7c00+gdt,0    ! Linear base of GDT is at offset gdt of seg 0x7c0.
66 .org 510
67     .word    0xAA55          ! Boot sector flag. Must be at last 2 bytes of boot sector.
```

4.9.3 Multitasking Kernel program head.s

After entering the protection mode, the main reason for the head.s program to re-establish and set the IDT and GDT tables is to make the program clearer in structure, and to be consistent with the setting of the two tables in the Linux 0.12 kernel source code. Of course, for this program, we can directly use the IDT and GDT table locations set in boot.s, and fill in the appropriate descriptor items.

```
01 # Head.s contains code for 32-bit protected mode init, clock & system call interrupts, and two
02 # tasks code. After initialization, the program moves to task 0 to start execution, and the
   # switching operation between tasks 0 and 1 is performed under the clock interrupt.
03 LATCH      = 11930          # Timer count, interrupt is sent every 10 ms.
04 SCR_N_SEL  = 0x18           # The segment selector for the screen display memory.
05 TSS0_SEL   = 0x20           # TSS segment selector for task 0.
06 LDT0_SEL   = 0x28           # LDT segment selector for task 0.
07 TSS1_SEL   = 0x30           # TSS segment selector for task 1.
08 LDT1_SEL   = 0x38           # LDT segment selector for task 1.
09 .text
10 startup_32:
11 # First load DS, SS, and ESP. The linear base address of all segments is 0.
12     movl $0x10,%eax          # 0x10 is the data segment selector in the GDT.
13     mov %ax,%ds
14     lss init_stack,%esp
15 # Re-set the IDT and GDT tables at new location.
16     call setup_idt           # Setup IDT.
17     call setup_gdt           # Setup GDT.
18     movl $0x10,%eax          # Reload all segment registers after changing GDT.
19     mov %ax,%ds
20     mov %ax,%es
21     mov %ax,%fs
22     mov %ax,%gs
23     lss init_stack,%esp
24 # Set 8253 timing chip. Channel 0 is set to generate an interrupt request every 10 ms.
25     movb $0x36, %al          # Control word: Channel 0 in mode 3, Count in binary.
26     movl $0x43, %edx          # 0x43 is write port of control word register.
27     outb %al, %dx
28     movl $LATCH, %eax         # Init count set to LATCH (1193180/100), freq. 100HZ.
```



```

29      movl $0x40, %edx          # The port of channel 0.
30      outb %al, %dx            # Write initial count value to channel 0 in two steps.
31      movb %ah, %al
32      outb %al, %dx
33 # Set the timer interrupt gate descriptor at item 8 of the IDT table.
34      movl $0x00080000, %eax    # EAX high word set to kernel code seg selector 0x0008.
35      movw $timer_interrupt, %ax # Set timer int gate descriptor. Get handler address.
36      movw $0x8E00, %dx        # Interrupt gate type is 14, plevel is 0 or hardware used.
37      movl $0x08, %ecx         # Clock interrupt vector no. set by BIOS is 8.
38      lea idt(,%ecx,8), %esi    # Put IDT Descriptor 0x08 address into ESI and set it.
39      movl %eax, (%esi)
40      movl %edx, 4(%esi)
    # Set the system call trap gate descriptor at item 128 (0x80) of the IDT table.
41      movw $system_interrupt, %ax # Set system call gate descriptor. Get handler address.
42      movw $0xef00, %dx         # Trap gate type is 15, code of plevel 3 is executable.
43      movl $0x80, %ecx         # System call vector no. is 0x80.
44      lea idt(,%ecx,8), %esi    # Put IDT Descriptor 0x80 address into ESI and set it.
45      movl %eax, (%esi)
46      movl %edx, 4(%esi)
47 # Now, to use IRET to move to task 0 (A), we manually prepare to setup the stack contents.
    # See Figure4-29 for the stack contents we need to setup. Refer to include/asm/system.h.
48      pushfl                   # Reset NT flag in EFLAGS to disable task switch when
49      andl $0xffffbfff, (%esp) # execute IRET instruction.
50      popfl
51      movl $TSS0_SEL, %eax      # Load task0's TSS seg selector into task register TR.
52      ltr %ax
53      movl $LDT0_SEL, %eax      # Load task0's LDT seg selector into LDTR.
54      lldt %ax                 # TR and LDTR need only be manually loaded once.
55      movl $0, current         # Save current task num 0 into current variable.
56      sti                      # Enable int, build a scene on stack for int returns.
57      pushl $0x17              # Push task 0 data (stack) seg selector onto stack.
58      pushl $init_stack        # Push the stack pointer (same as push ESP).
59      pushfl                   # Push the EFLAGS.
60      pushl $0x0f              # Push current local space code seg selector.
61      pushl $task0             # Push task 0 code pointer onto stack.
62      iret                    # This causes execution moves to task0 in plevel 3.
63
64 # The following are the subroutines for setting descriptor items in GDT and IDT.
65 setup_gdt:                   # GDT table position & limit are set using
66     lgdt lgdt_opcode          # 6-byte operand lgdt_opcode.
67     ret
    # The following code is used to temporarily set all 256 interrupt gate descriptors in the
    # IDT table to the same default value. All use the default interrupt handler ignore_int.
    # The specific method of setting is: first set the contents of 0-3 bytes and 4-7 bytes of
    # the default interrupt gate descriptor into the eax and edx register pairs. Then, using
    # this register pair, the interrupt descriptor is cyclically filled into the IDT table.
68 setup_idt:                   # Set all 256 int gate descriptors to use default handler.
69     lea ignore_int,%edx       # The same way as setting timer int gate descriptor.
70     movl $0x00080000,%eax     # The selector is 0x0008.
71     movw %dx,%ax
72     movw $0x8E00,%dx         # Interrupt gate type is 14, plevel is 0.
73     lea idt,%edi
74     mov $256,%ecx            # Loop through all 256 gate descriptor entries.

```

```

75 rp_idt: movl %eax, (%edi)
76         movl %edx, 4(%edi)
77         addl $8, %edi
78         dec %ecx
79         jne rp_idt
80         lidt lidt_opcode          # IDTR register is loaded with a 6-byte operand.
81         ret
82
83 # Display characters subroutine. Get current cursor position & display char in AL.
  # The entire screen can display 80 X 25 (2000) characters.
84 write_char:
85     push %gs                      # First save the register to be used, EAX is
86     pushl %ebx                   # saved by the caller.
87     mov $SCRN_SEL, %ebx          # Then let GS point to display mem seg (0xb8000).
88     mov %bx, %gs
89     movl scr_loc, %bx            # Get current char display position from scr_loc.
90     shl $1, %ebx                 # Since each char has one attribute byte, so actual
91     movb %al, %gs:(%ebx)         # display memory offset should multiplied by 2.
92     shr $1, %ebx                 # After putting char into display memory, divide the
93     incl %ebx                    # position value by 2 plus 1 to get the next position.
94     cmpl $2000, %ebx             # If position is greater than 2000, it is reset to 0.
95     jb 1f
96     movl $0, %ebx
97 1:    movl %ebx, scr_loc          # Finally save this position value (scr_loc),
98     popl %ebx                   # and pop up the contents of saved register, return.
99     pop %gs
100    ret
101
102 # The following are 3 interrupt handlers: default, timer, and system call interrupt.
103 # Ignore_int is default handler. If system generates other interrupts, it display char 'C'.
104 .align 2
105 ignore_int:
106     push %ds
107     pushl %eax                   # Let DS point to the kernel data segment because
108     movl $0x10, %eax            # the interrupt handler belongs to the kernel.
109     mov %ax, %ds
110     movl $67, %eax              # Put 'C' in AL, call write_char to display on screen.
111     call write_char
112     popl %eax
113     pop %ds
114     iret
115
116 # This is the timer interrupt handler. The main function is to perform task switching operations.
117 .align 2
118 timer_interrupt:
119     push %ds
120     pushl %eax
121     movl $0x10, %eax            # First let DS point to the kernel data segment.
122     mov %ax, %ds
123     movb $0x20, %al             # Then send EOI to 8259A to allow other interrupts.
124     outb %al, $0x20
125     movl $1, %eax               # Then check current task to switch task 0 and 1.
126     cmpl %eax, current

```

```

127     je 1f
128     movl %eax, current      # If current task is 0, save 1 in current and jump to
129     ljmp $TSS1_SEL, $0     # task 1 to execute. The offset of jump is useless.
130     jmp 2f
131 1:    movl $0, current      # If current task is 1, save 0 in current and jump to
132     ljmp $TSS0_SEL, $0     # task 0 to execute.
133 2:    popl %eax
134     pop %ds
135     iret
136
137 # The system call int 0x80 handler. This example has only one display char function.
138 .align 2
139 system_interrupt:
140     push %ds
141     pushl %edx
142     pushl %ecx
143     pushl %ebx
144     pushl %eax
145     movl $0x10, %edx        # First let DS point to the kernel data segment.
146     mov %dx, %ds
147     call write_char         # Then call routine write_char to display char in AL.
148     popl %eax
149     popl %ebx
150     popl %ecx
151     popl %edx
152     pop %ds
153     iret
154
155 /*****/
156 current:.long 0             # Store current task number (0 or 1).
157 scr_loc:.long 0            # Store screen current display position.
158
159 .align 2
160 lidt_opcode:
161     .word 256*8-1           # 6-byte operand for set IDTR : table size & base.
162     .long idt
163 lgdt_opcode:
164     .word (end_gdt-gdt)-1   # 6-byte operand for set IDTR : table size & base.
165     .long gdt
166
167 .align 3
168 idt:   .fill 256,8,0        # IDT. 256 gate descriptors, each 8 bytes, total 2KB.
169 # The following is GDT table contents (of descriptors).
170 gdt:   .quad 0x0000000000000000 # [0] The first segment descriptor is not used.
171       .quad 0x00c09a00000007ff # [1] Kernel code descriptor. Its selector is 0x08
172       .quad 0x00c09200000007ff # [2] Kernel data descriptor. Its selector is 0x10
173       .quad 0x00c0920b80000002 # [3] Display mem descriptor. Its selector is 0x18
174       .word 0x68, tss0, 0xe900, 0x0 # [4] TSS0 descriptor. Its selector is 0x20.
175       .word 0x40, ldt0, 0xe200, 0x0 # [5] LDT0 descriptor. Its selector is 0x28
176       .word 0x68, tss1, 0xe900, 0x0 # [6] TSS1 descriptor. Its selector is 0x30
177       .word 0x40, ldt1, 0xe200, 0x0 # [7] LDT1 descriptor. Its selector is 0x38
178 end_gdt:
179     .fill 128,4,0           # Initial kernel stack space.

```

```

180 init_stack:                                # Stack pointer when first enter protected mode.
181     .long init_stack                        # Stack segment offset position.
182     .word 0x10                             # Stack segment, same as kernel data seg (0x10).
183
184 # Below is the local segment descriptor in the LDT table segment of task 0.
185 .align 3
186 ldt0:   .quad 0x0000000000000000           # [0] The first descriptor is not used.
187         .quad 0x00c0fa00000003ff          # [1] The local code descriptor, its selector is 0x0f
188         .quad 0x00c0f200000003ff          # [2] The local data descriptor, its selector is 0x17
189 # Content of TSS seg for task 0. Note fields such as labels do not change when task switches.
190 tss0:   .long 0                           /* back link */
191         .long krn_stk0, 0x10               /* esp0, ss0 */
192         .long 0, 0, 0, 0, 0               /* esp1, ss1, esp2, ss2, cr3 */
193         .long 0, 0, 0, 0, 0               /* eip, eflags, eax, ecx, edx */
194         .long 0, 0, 0, 0, 0               /* ebx esp, ebp, esi, edi */
195         .long 0, 0, 0, 0, 0, 0            /* es, cs, ss, ds, fs, gs */
196         .long LDT0_SEL, 0x8000000         /* ldt, trace bitmap */
197
198         .fill 128, 4, 0                   # This is the kernel stack space for task 0.
199 krn_stk0:
200
201 # Task 1 LDT table content and TSS segment content.
202 .align 3
203 ldt1:   .quad 0x0000000000000000           # [0] The first descriptor is not used.
204         .quad 0x00c0fa00000003ff          # [1] The selector is 0x0f, base = 0x00000.
205         .quad 0x00c0f200000003ff          # [2] The selector is 0x17, base = 0x00000.
206
207 tss1:   .long 0                           /* back link */
208         .long krn_stk1, 0x10               /* esp0, ss0 */
209         .long 0, 0, 0, 0, 0               /* esp1, ss1, esp2, ss2, cr3 */
210         .long task1, 0x200                 /* eip, eflags */
211         .long 0, 0, 0, 0                   /* eax, ecx, edx, ebx */
212         .long usr_stk1, 0, 0, 0           /* esp, ebp, esi, edi */
213         .long 0x17, 0x0f, 0x17, 0x17, 0x17, 0x17 /* es, cs, ss, ds, fs, gs */
214         .long LDT1_SEL, 0x8000000         /* ldt, trace bitmap */
215
216         .fill 128, 4, 0                   # This is the kernel stack space for task 1. Its user
217 krn_stk1:                                # stack uses the initial kernel stack space directly.
218
219 # The programs of tasks 0 and 1, which cyclically display chars 'A' and 'B', respectively.
220 task0:
221     movl $0x17, %eax                       # DS point to the local data segment of the task.
222     movw %ax, %ds                          # No local data, these 2 instructions can be omitted.
223     movl $65, %al                          # Put 'A' into the AL register.
224     int $0x80                              # Execute system call to display it.
225     movl $0xffff, %ecx                     # Execute a loop, act as a delay.
226 1:    loop 1b
227     jmp task0                              # Jump to start of task 0 to continue displaying.
228 task1:
229     movl $66, %al                          # Put 'B' into the AL register.
230     int $0x80                              # Execute system call to display it.
231     movl $0xffff, %ecx                     # Execute a loop, act as a delay.
232 1:    loop 1b

```

```
233         jmp task1
234
235         .fill 128,4,0           # This is user stack space for task 1.
236 usr_stk1:
```

4.10 Summary

This chapter describes the protection mode memory management and programming principles of the Intel 80X86 CPU. It describes in detail the specific meanings of the global and local descriptor tables, segment descriptors, and segment selectors. It also gives the transformation relationships between program logical address, CPU linear address, and physical memory address. Finally, a simple kernel sample program is given and introduced at the end of this chapter. Based on an understanding of the sample program, we can roughly explain how well we master the protection mode programming.

Below we use a whole chapter to provide a comprehensive overview of the hardware settings, memory allocation and usage of the Linux kernel and the function of the task data structure, and then classify all the source code in the kernel source tree, let the reader first Have a general understanding of the entire kernel code file structure. Then, in the following chapters, the source code files in the kernel are described and annotated in detail.

5 Linux kernel architecture

This chapter can be seen as a general overview of the kernel source code and can be used as a reference for reading subsequent chapters. For content that is difficult to understand, you can skip it first. When you read the related content in the following chapters, return to refer to this chapter. Please review or learn about how the 80X86 protected mode mode of operation works before reading this chapter.

This chapter begins with an overview of the Linux kernel's composition modes and architecture, and then details the source file organization in the kernel source directory, as well as the main functions of the various code files in the subdirectories and the hierarchical relationships of the basic calls. Then directly cut into the topic, starting from the first file Makefile in the kernel source file Linux / directory to explain each line of code in detail. We will briefly describe the basic architecture and main components based on Linux 0.12 kernel source code. It also explains several important data structures that appear in the source code. Finally, the method of building the Linux 0.12 kernel compilation experimental environment is described.

From a layered perspective, a complete operating system can be composed of four parts: hardware, operating system kernel, operating system services, and user applications, as shown in Figure 5-1. User applications are those word processors, Internet browser programs, or various applications compiled by users themselves; Operating system services are those that provide services to users and are considered part of the operating system. In the Linux operating system, these programs include X window system, shell command interpretation system and system programs such as kernel programming interface; The operating system kernel is the part of this book that is of interest. It is mainly used to abstract hardware resources and schedule management of all system resources.

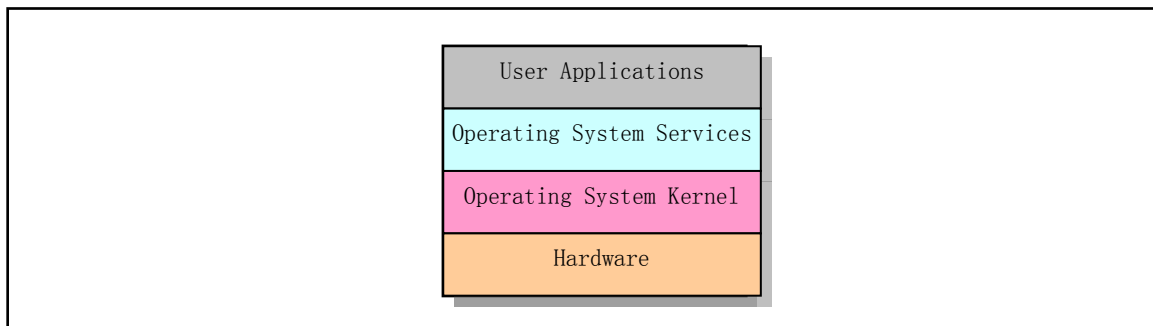


Figure 5-1 Operating system components

The main purpose of the Linux kernel is to interact with computer hardware, implement interface operations and programmatic control of components, schedule access to hardware resources, and provide an easy-to-use execution environment and a common hardware virtual interface for user programs on the computer. .

5.1 Linux kernel mode

At present, the structural mode of the operating system kernel can be mainly divided into a monolithic

single-core model and a hierarchical micro-kernel model, and a mixed mode of the two. The Linux 0.12 kernel annotated in this book uses a single-core mode.

In a monolithic single-core system model, the service process provided by the operating system is: the application program executes the system call instruction (int x80) with the specified parameters, so that the CPU switches from the user mode to the core state (Kernel Model).), then the operating system calls the specific system call service procedure according to the specific parameters, and these service procedures call some of the underlying support functions to complete the specific functions as needed. After completing the services required by the application, the operating system switches the CPU back from the kernel mode to the user mode, and returns to the application to continue executing the following instructions. So in summary, the single-core mode kernel can also be roughly divided into three levels: the main program layer that calls the service, the service layer that executes the system call, and the underlying functions that support the system call. See Figure 5-2. The main advantage of the monolithic model is that the kernel code is compact and fast, and the disadvantages are mainly that the hierarchy is not strong.

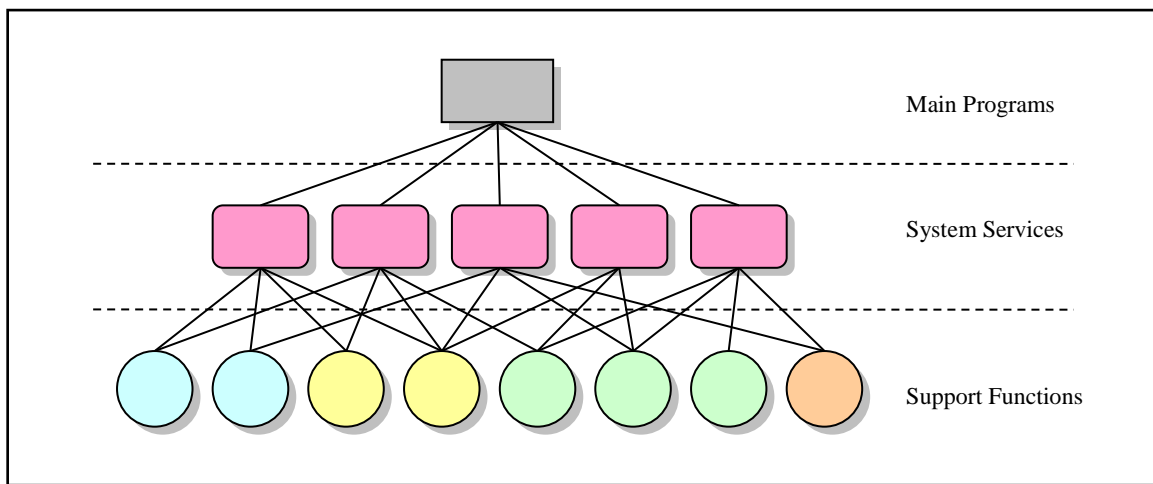


Figure 5-2 Simple structural of monolithic model

For the microkernel architecture model, its main features are function modularization and messaging between service threads or processes. The system core provides a basic hardware abstraction management layer and key system service functions. These key functions are main process/inter-thread communication services, virtual memory management, and process scheduling. The rest of the operating system functions in user space in a variety of modular forms. Therefore, the advantage of the microkernel structure is that the system service coupling is low, which facilitates system improvement, expansion, and porting. The main disadvantage is that a large amount of message passing and synchronization operations between system service modules are required during the running, and these operations cause communication resource consumption and time delay. Typical microkernel architecture systems are the MINIX operating system and the Mac OS system with the Mach kernel.

5.2 Linux kernel system architecture

The Linux kernel consists of five modules: process scheduling module, memory management module, file system module, interprocess communication module, and network interface module, as shown in Figure 5-3.

The process scheduling module is responsible for controlling the use of CPU resources by the process. The

scheduling strategy adopted is that each process can access the CPU fairly and reasonably, while ensuring that the kernel can perform hardware operations in a timely manner. The memory management module is used to ensure that all processes can safely share the main memory area of the machine. At the same time, the memory management module also supports the virtual memory management mode, so that the Linux support process uses more memory capacity than the actual memory space. You can use the file system to swap unused memory blocks to an external storage device and exchange them when needed. File system modules are used to support the drive and storage of external devices. The Virtual File System module hides the different details of various hardware devices by providing a common file interface to all external storage devices. This provides and supports multiple file system formats that are compatible with other operating systems. The interprocess communication module is used to support the exchange of information between multiple processes. Network interface modules provide access to a variety of network communication standards and support many network hardware.

The dependencies between these modules are shown in Figure 5-3. The connections represent the dependencies between them, and the dashed and dashed boxes represent the unimplemented parts of Linux 0.12 (the virtual file system is gradually implemented from the Linux 0.95 version, while the network interface support is only available in version 0.96 or later).

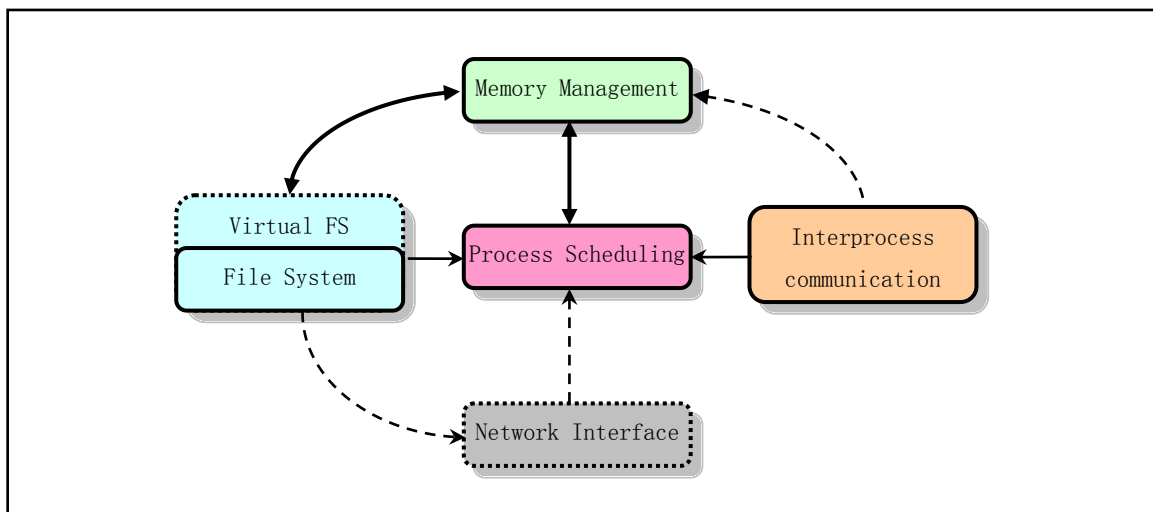


Figure 5-3 Linux kernel module structure and interdependence

As can be seen from the figure, all modules have dependencies on the process scheduling module. Because they all rely on the process scheduler to suspend (pause) or re-run their processes. Typically, a module will be suspended while waiting for hardware operations, and will continue to run until the operation is complete. For example, when a process attempts to write a block of data to a floppy disk, the floppy disk driver may place the process in a suspend wait state during the boot floppy disk rotation, and cause the process to continue after the floppy disk enters the normal rpm. run. The other three modules are also dependent on the process scheduling module for similar reasons.

The dependencies of several other modules are somewhat less obvious, but they are also important. The process scheduling subsystem needs to use memory management to adjust the physical memory space used by a particular process. The interprocess communication subsystem relies on a memory manager to support shared memory communication mechanisms. This communication mechanism allows two processes to access the same area of memory for the exchange of information between processes. The virtual file system also uses the

network interface to support the Network File System (NFS), as well as the memory management subsystem to provide memory ramdisk devices. The memory management subsystem also uses the file system to support the swapping of memory blocks.

From the monolithic structure model, we can also draw the kernel main modules into the block diagram structure shown in Figure 5-4 according to the structure of the Linux 0.12 kernel source code.

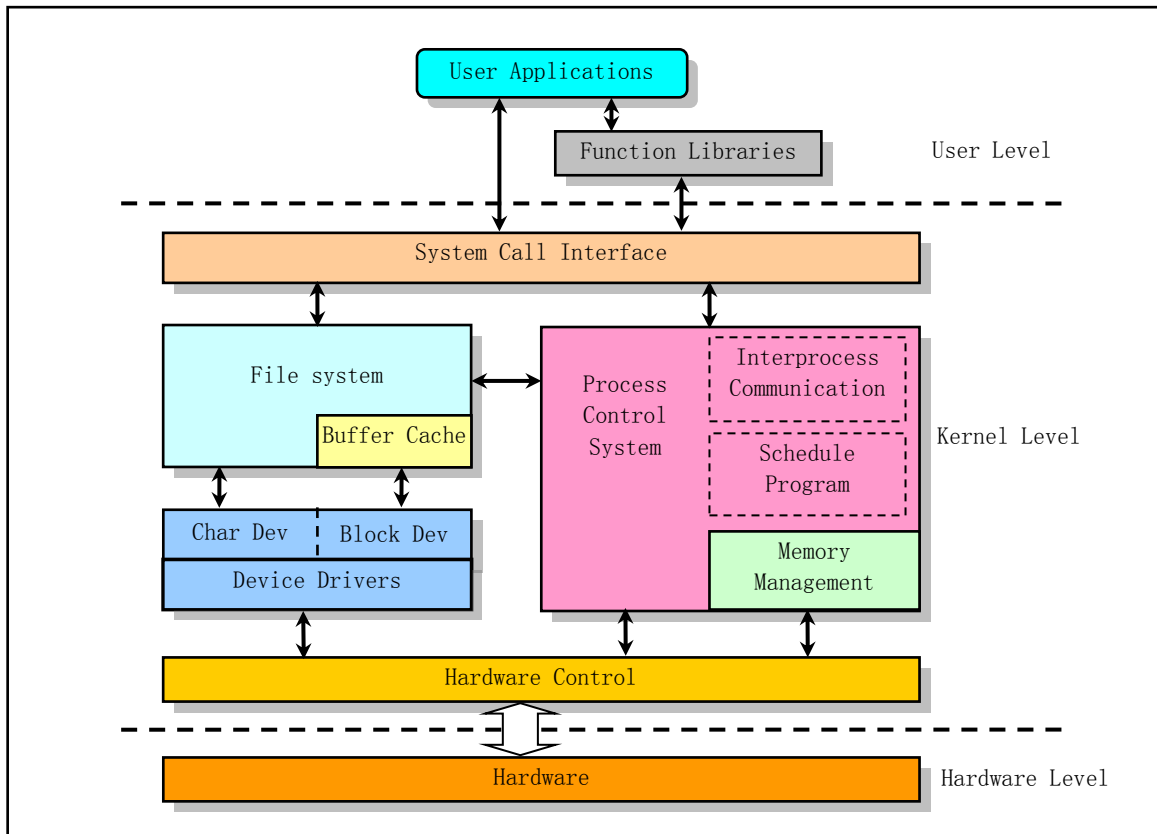


Figure 5-4 Kernel block diagram

In several boxes in the kernel level, the thick line boxes other than the hardware control block correspond to the directory organization of the kernel source code. In addition to the dependencies already given in these diagrams, all of these modules also rely on common resources in the kernel. These resources include memory allocation and reclaim functions, print warning or error message functions, and some system debugging functions.

5.3 Linux kernel memory management

This section first describes the relatively straightforward physical memory usage in Linux 0.12 systems. Then combined with the application situation in the Linux 0.12 kernel, it outlines the segmentation and paging management mechanism of the memory, as well as the CPU multitasking operation and protection mode. Finally, we will comprehensively explain the correspondence between the kernel code and data in the Linux 0.12 system and the code and data of each task in the virtual, linear and physical address space.

The description in this section can be seen as a summary or review of memory management. See Chapter 4 for a detailed description of memory management in protection mode.

5.3.1 Physical address

In the Linux 0.12 kernel, in order to effectively use the physical memory in the machine, the memory is divided into several functional areas during the system initialization phase, as shown in Figure 5-5.

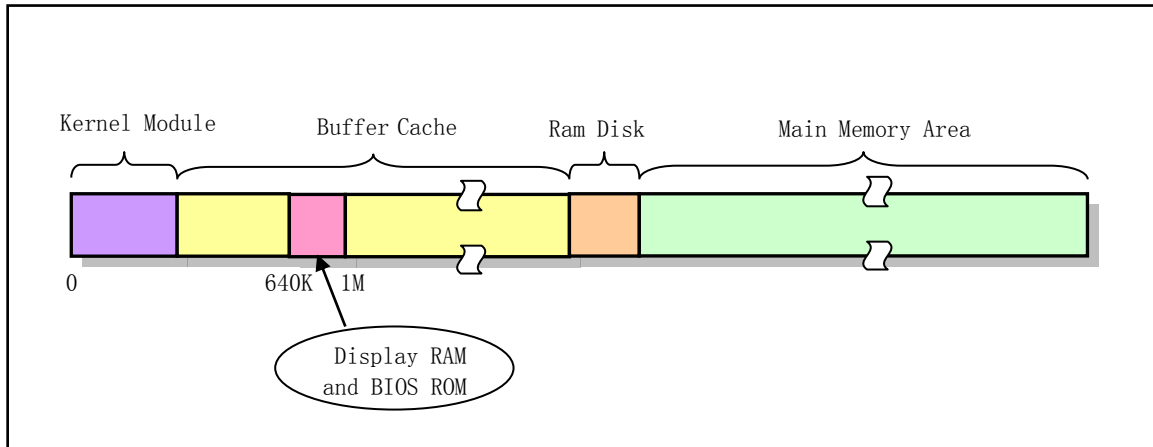


Figure 5-5 Functional distribution of physical memory usage

Among them, the Linux kernel occupies the beginning of the physical memory, followed by the high-speed buffer portion for the block device such as the hard disk or floppy disk (in which the memory address range of the display card memory and the ROM BIOS is 640K--1MB). When a process or task needs to read data from a block device, the system first reads the data into the cache. When there is data to be written to the block device, the system first puts the data into the cache, and then the block device driver writes to the corresponding device. The last part of the memory is the main memory area that all programs can request and use at any time. When the kernel program uses the main memory area, it also needs to apply to the kernel memory management module first, and can use the memory after the application is successful. For systems with RAM virtual disks, the main memory area header is also partially removed for the virtual disk to store data.

Because the actual physical memory capacity contained in the computer system is limited, the CPU usually provides a memory management mechanism to effectively manage the memory in the system. In the Intel 80386 and later CPUs, two memory management (address translation) systems are provided: the Memory Segmentation and the Paging System. The paging management system is optional and is programmed by the system programmer to determine whether to adopt. In order to use physical memory effectively, the Linux system uses both memory segmentation and paging management mechanisms.

5.3.2 Memory Address Space Concept

In the Linux 0.12 kernel, when performing address mapping, we need to first distinguish between three types of addresses and the concept of transformation between them: a). virtual and logical addresses of programs (processes); b). linear addresses of CPUs; c). actual physics Memory address.

Virtual Address refers to the address generated by the program consisting of two parts: the segment selector and the offset address within the segment. Because the two parts of the address are not directly used to access the physical memory, but need to be processed or mapped by the segmentation address translation mechanism to correspond to the physical memory address, such an address is called a virtual address. The virtual address space consists of the global address space mapped by the GDT and the local address space mapped by the LDT. The index portion of the selector is represented by 13 bits, plus one bit that distinguishes

between GDT and LDT, so the Intel 80X86 CPU can index a total of 16384 selectors. If the length of each segment takes a maximum of 4G, the maximum virtual address space is $16384 * 4G = 64T$.

Logical Address is the portion of the offset address associated with the sections generated by the program. In Intel protected mode, it refers to the offset address within the program execution code section limit length (assuming the code section and data section are identical). Application programmers only need to deal with logical addresses, and the segmentation and paging mechanism is completely transparent to him, only by system programmers. However, some materials do not distinguish between the concept of logical addresses and virtual addresses, but collectively refer to them as logical addresses.

The Linear Address is the middle layer between the virtual address and physical address translation and is the address in the processor's addressable memory space (called the linear address space). The program code will generate a logical address, or an offset address in the segment, plus a base address for the corresponding segment to generate a linear address. If the paging mechanism is enabled, the linear address can be transformed to produce a physical address. If the paging mechanism is not enabled, the linear address is directly the physical address. The Intel 80386 has a linear address space of 4G.

The Physical Address is the address signal indicating the addressed physical memory on the CPU external address bus, which is the final result address of the address translation. If the paging mechanism is enabled, the linear address is transformed into a physical address using the items in the page directory and page table. If the paging mechanism is not enabled, the linear address becomes the physical address directly.

Virtual storage (or virtual memory) is the amount of memory that a computer presents to be much larger than the actual amount of memory it has. It therefore allows the programmer to program and run programs that are much larger than the actual system has. This allows many large projects to be implemented on systems with limited memory resources. A very appropriate analogy is that you don't need a long track to get a train from Shanghai to Beijing. You only need a long enough rail (say 10 km) to complete this task. The method is to immediately lay the rear rails in front of the train. As long as your operation is fast enough and can meet the requirements, the train can run like a complete track. This is the task that virtual memory management needs to accomplish. In the Linux 0.12 kernel, each program (process) is divided into a virtual memory space with a total capacity of 64MB. Therefore, the program's logical address range is 0x0000000 to 0x4000000.

As mentioned above, sometimes we also refer to logical addresses as virtual addresses. Because the logical address is similar to the concept of virtual memory space, and is also independent of the actual physical memory capacity.

5.3.3 Memory Segmentation Mechanism

In a memory segmentation system, the logical address of a program is automatically mapped (transformed) into the 4GB (2^{32}) linear address space of the middle layer by a segmentation mechanism. Each reference to memory by the program is a reference to memory in the memory segment. When a program references a memory address, a corresponding linear address is formed by adding the corresponding segment base address to the logical address visible to the programmer. If the paging mechanism is not enabled at this time, the linear address is sent to the CPU's external address bus for direct addressing of the corresponding physical memory. See Figure 5-6.

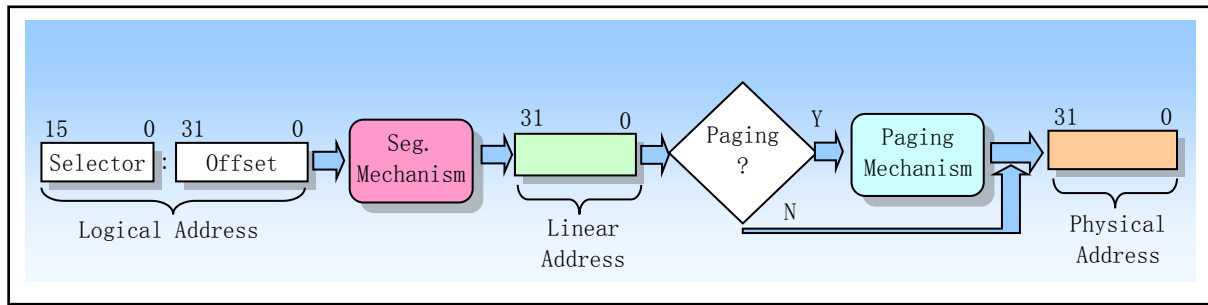


Figure 5-6 Logical address to physical address translation

The main purpose of the CPU to perform address translation (mapping) is to solve the mapping problem from virtual memory space to physical memory space. The meaning of virtual memory space refers to a method that uses secondary or external storage space to enable programs to use memory without being limited by the actual amount of physical memory. Usually the virtual memory space is much larger than the actual physical memory.

So how is virtual storage management implemented? The principle is similar to the analogy of the above train operation. First, when a program needs to use a memory that does not exist (that is, the corresponding memory page is not in memory in the memory page table entry), the CPU needs a way to know the situation. This is achieved by the 80386 page fault exception interrupt. When a process references a memory address in a non-existing page, it triggers the CPU to generate a page fault exception interrupt and places the linear address that caused the interrupt into the CR2 control register. Therefore, the process of processing the interrupt can know the exact address of the page exception, so that the page requested by the process can be loaded into the physical memory from the secondary storage space (such as the hard disk). If the physical memory is already occupied at this time, you can use a part of the secondary storage space as a swap buffer (Swapper) to swap the temporarily unused pages in the secondary buffer, and then transfer the requested page to the memory. in. This is the page fault loading mechanism of memory management. It is implemented in the program `mm/memory.c` in the Linux 0.12 kernel.

The Intel CPU uses the concept of segment to address the program. Each segment defines information such as an area in memory and the priority of access. Assuming that everyone knows the memory addressing principle in real mode, we now use the comparison method to briefly explain the main features of memory addressing under the 32-bit protected mode operating mechanism according to the different addressing modes of the CPU in real mode and protected mode.

In real mode, addressing a memory address primarily uses segment and offset values, segment values are stored in segment registers (such as DS), and the length of the segment is fixed at 64 KB. The intra-segment offset address is stored in any register that can be used for addressing (eg, SI). Therefore, based on the values in the segment register and the offset register, the actual pointed memory address can be calculated, as shown in Figure 5-7 (a).

In the protected mode mode, the segment register is no longer the base address of the addressed segment, but the index value of a descriptor entry in the segment descriptor table. The segment descriptor item specified by the index value contains information such as the base address of the memory segment to be addressed, the limit length of the segment, and the access privilege level of the segment. The addressed memory location is a combination of the segment base address specified in the segment descriptor entry and an intra-segment offset. The limit length of the segment is variable and is specified by the content in the descriptor. It can be seen that, compared with the addressing in the real mode, the segment register value is replaced with the index value of

the corresponding segment descriptor in the segment descriptor table, and the segment table selection bit and the privilege level, which is called a Segment Selector, but The offset value still uses the concept in the original mode. Thus, addressing a memory address in protected mode requires one more procedure than in real mode, which requires the use of a segment descriptor table. This is because there is more information to access a memory segment in protected mode, and a 16-bit segment register can't hold much of this content. The schematic is shown in Figure 5-7 (b). Note that if you do not define a memory linear address space area in a segment descriptor, the address area will not be addressed at all and the CPU will deny access to the address area.

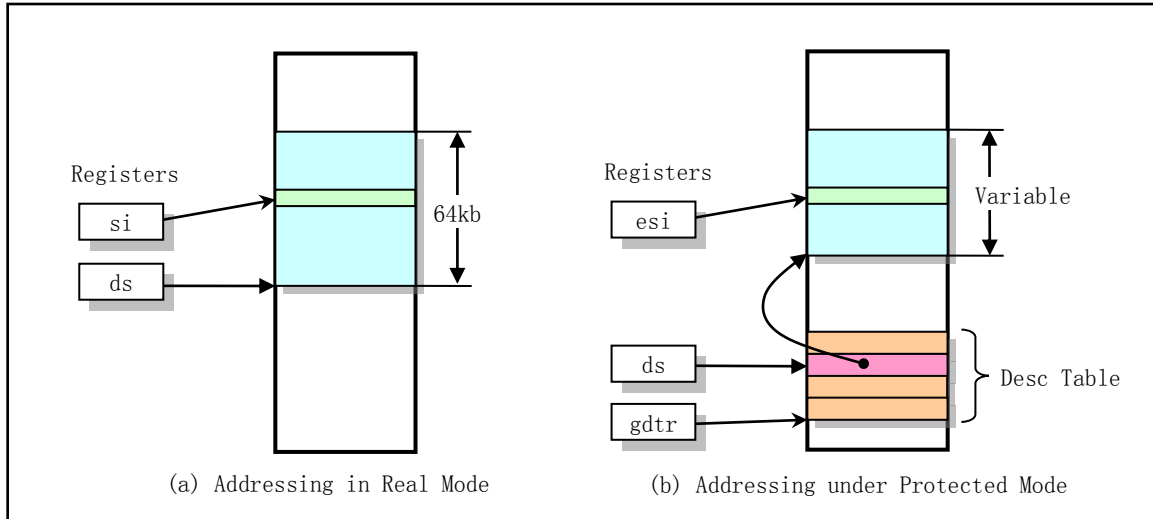


Figure 5-7 Comparison of addressing in real mode and protected mode

Each descriptor occupies 8 bytes, containing the starting address (base address) of the described segment in the linear address space, the limit length of the segment, the type of the segment (such as code segments and data segments), the privilege level of the segment, and Some other information. The maximum length a segment can define is 4GB.

There are three types of descriptor tables for saving descriptor items, each for a different purpose. The Global Descriptor Table (GDT) is the main base descriptor table that can be used by all programs to reference a memory segment. The Interrupt Descriptor Table (IDT) holds a segment descriptor that defines the interrupt or exception handling process. The IDT table directly replaces the interrupt vector table in the 8086 system. In order to operate normally in 80X86 protected mode, we must define a GDT table and an IDT table for the CPU. The last type of table is the Local Descriptor Table (LDT). This table is used in multitasking systems, usually using one LDT table per task. As an extension to the GDT table, each LDT table provides more available descriptor entries for the corresponding task, thus providing a range of addressable memory spaces for each task.

These tables can be saved anywhere in the linear address space. In order to allow the CPU to locate the GDT table, the IDT table, and the current LDT table, three special registers of GDTR, IDTR, and LDTR need to be set for the CPU. The 32-bit linear base address of the corresponding table and the limit-length byte value of the table are stored in these registers. The table length value is the length value of the table -1.

When the CPU is to address a segment, the selector in the 16-bit segment register is used to locate a segment descriptor. In an 80X86 CPU, the value in the segment register is shifted to the right by 3 bits, which is the index value of a descriptor in the descriptor table. A 13-bit index value can locate up to 8192 (0--8191)

descriptor entries. The selector bit 2 (TI) is used to specify which table to use. If the bit is 0 then the selector specifies the descriptor in the GDT table, otherwise it is the descriptor in the LDT table.

Each program can consist of several memory segments. The logical address (or virtual address) of the program is used to address the specific address locations in these segments. In Linux 0.12, the translation process of program logical address to linear address uses the global segment descriptor table GDT and the local segment descriptor table LDT. The address space mapped by the GDT is called the global address space, and the address space mapped by the LDT is called the local address space, and the two constitute the space of the virtual address. The specific usage is shown in Figure 5-8.

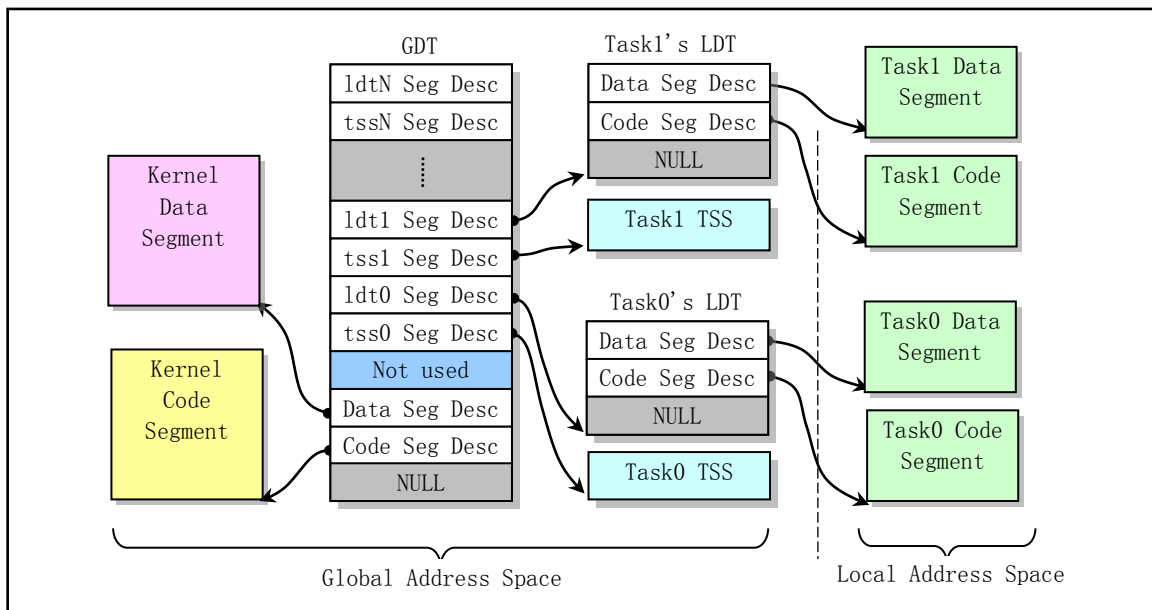


Figure 5-8 Virtual address space allocation map in Linux

The figure shows the situation when there are two tasks. It can be seen that the local descriptor table LDT of each task is also a memory segment defined by the descriptor in the GDT, in which the code segment and the data segment descriptor of the corresponding task are stored, so the LDT segment is very short. The segment length is usually as long as it is larger than 24 bytes. Similarly, the task status segment TSS for each task is also a memory segment defined by the descriptors in the GDT, and the segment length limit is sufficient as long as it satisfies the ability to store a TSS data structure.

对 In the interrupt descriptor table IDT, it is stored in the kernel code segment. In the Linux 0.12 kernel, since the code segment and data segment of the kernel and each task are respectively mapped to the same base address in the linear address space, and the segment length is the same, the code segment and the data segment of the kernel are overlapped. The code segment and data segment of each task are also overlapped respectively, as shown in Figure 5-10 or Figure 5-11. The Task State Segment (TSS) is used to automatically save or restore the current execution context (CPU current state) of the related task when the task is switched. For example, for a task that is switched out, the CPU saves its register and other information in the TSS segment of the task, and the CPU uses the information in the TSS segment newly switched into the task to set each register to restore the execution environment of the new task.

In Linux 0.12, the TSS segment content of each task is saved in the task data structure of the task. In addition, the fourth descriptor in the GDT table (the syscall descriptor entry in the figure) is not used in the Linux 0.12 kernel. From the original English comment on line 201 in the include/linux/sched.h file shown

below, it can be guessed that Linus had designed the kernel to place the code for the system call in this specialized section when designing the kernel.

```
200 /*  
201  * Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall  
202  * 4-TSS0, 5-LDT0, 6-TSS1 etc ...  
203 */
```

5.3.4 Memory paging management

If paging is used, then the linear address is only an intermediate result, and it needs to be transformed by using the paging mechanism, and finally mapped to the actual physical memory address. Similar to the segmentation, paging mechanism allows us to redirect (transform) each memory reference to accommodate our particular requirements. The most common use of the paging is when the system memory is actually divided into a lot of messy blocks, it can create a large and continuous memory space image, so that the program does not have to worry about and manage these scattered memory blocks. The paging mechanism enhances the performance of the segmentation mechanism. In addition, the page address transformation is based on the segment transformation. The protection measures of any paging mechanism will not replace the protection measures of the segment transformation but only carry out further checking operations.

The basic principle of the memory paging is to divide the entire linear memory area of the CPU into 4096 bytes of a page of memory pages. When the program requests to use memory, the system allocates it in units of memory pages. The memory paging is implemented in a similar way to the segmentation mechanism, but not as sophisticated as the segmentation. Because the paging is implemented on top of the segmentation, the result is very flexible control over system memory, and the paging protection mechanism is added to the memory protection of the segmentation mechanism. In order to use the paging in the 80X86 protected mode, the highest bit (bit 31) of the control register CR0 needs to be set.

When using this memory paging method, each executing process (task) can use a contiguous address space that is much larger than the actual memory capacity. In order to map linear addresses to a relatively small physical memory space using the paging, the 80386 uses page directory tables and page tables. Page directory entries are basically the same format as page table entries, occupying 4 bytes, and each page directory table or page table contains only 1024 page table entries. Therefore, a page directory table or a page table occupies a total of 1 page of memory. The small difference between a page directory entry and a page table entry is that the page table entry has a written bit D (Dirty), while the page directory entry does not.

The transfer process from linear address to physical address is shown in Figure 5-9. The control register CR3 in the figure holds the base address of the current page directory table in physical memory (hence CR3 is also referred to as the page directory base address register PDBR). The 32-bit linear address is divided into three parts for locating the corresponding entries in the page directory table and the page table, and specifying the offset position within the page in the corresponding physical memory page. Because a page table can have 1024 entries, a page table can map up to $1024 * 4KB = 4MB$ memory; and because a page directory table has up to 1024 entries, corresponding to 1024 secondary page tables, a page directory table can be up to $Maps\ 1024 * 4MB = 4GB$ of memory. That is, a page directory table can map the entire linear address space range.

Since the kernel and all tasks in the Linux 0.1x system share the same page directory table, the mapping function of the processor linear address space to the physical address space is the same at any time. Therefore, in order for the kernel and all tasks to not overlap and interfere with each other, they must be mapped from the virtual address space to different locations of the linear address space, that is, occupying different linear address

space ranges.

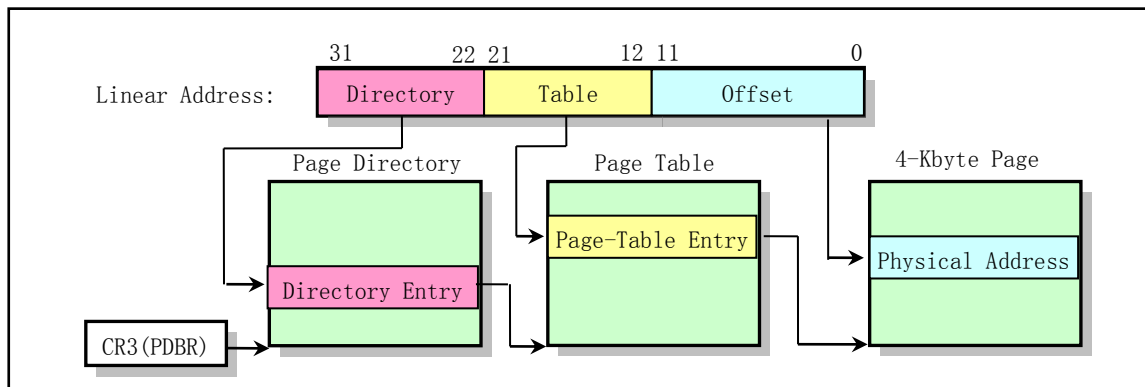


Figure 5-9 Diagram of linear address to physical address transformation

For Intel 80386 systems, the CPU can provide up to 4G of linear address space. The virtual address of a task needs to be first transformed into the address in the entire linear address space of the CPU by its local segment descriptor, and then mapped to the actual using the page directory table PDT (primary page table) and page table PT (secondary page table). On the physical address page. To use actual physical memory, the linear address of each process is dynamically mapped to different physical memory pages in the main memory area by the secondary memory page table.

Since Linux 0.12 defines the maximum available virtual memory space per process as 64MB, the logical address of each process can be converted to an address in linear space by adding (task number) * 64MB. However, in the code comments in this book, we sometimes simply refer to such addresses in a process as logical addresses or linear addresses without confusing them.

For Linux 0.12 kernel, the maximum number of segment descriptor entries set in the GDT is 256. Two of them are not used and two are system entries, and each process or task uses two. Therefore, at this point the system can accommodate up to $(256-4)/2 = 126$ tasks, and the virtual address range is $((256-4)/2) * 64\text{MB}$ is approximately equal to 8G. However, the maximum number of tasks manually defined in the 0.12 kernel is `NR_TASKS = 64`, the logical address range of each task is 64M, and the starting position of each task in the linear address space is (task number) * 64MB. So the linear address space used by all tasks is $64\text{MB} * 64 = 4\text{G}$, as shown in Figure 5-10.

The figure shows the situation when the system has 4 tasks. The kernel code segment and data segment are mapped to the beginning 16MB portion of the linear address space, and both the code and data segments are mapped to the same region, completely overlapping each other. The first task (task 0) is started by the kernel "manually". The code and data are contained in the kernel code and data, so the linear address space used by this task is quite special. The length of the code segment and the data segment of task 0 is a range of 640 KB from the linear address 0, and the code and the data segment also completely overlap, and overlap with the kernel code segment and the data segment. In fact, the instruction space I (Instruction) and the data space D (Data) of all tasks in Linux 0.12 use one piece of memory. That is, all the code, data, and stack parts of a process are in the same memory segment, which is a way of using I&D without separation.

Task 1 has a linear address space starting at address 64MB and only 640KB in length. The detailed correspondence between them will be described later. Task 2 and Task 3 are mapped to the linear addresses 128MB and 192MB, respectively, and their logical address ranges are 64MB. Since the 4G address space range

is exactly the linear address space range of the 32-bit CPU, it is also the addressable maximum physical address space range. Moreover, when the logical address range of task 0 and task 1 is regarded as 64 MB, the system may have The logical address range of the task is also 4GB, so it is easier to confuse the three address concepts in the 0.12 kernel.

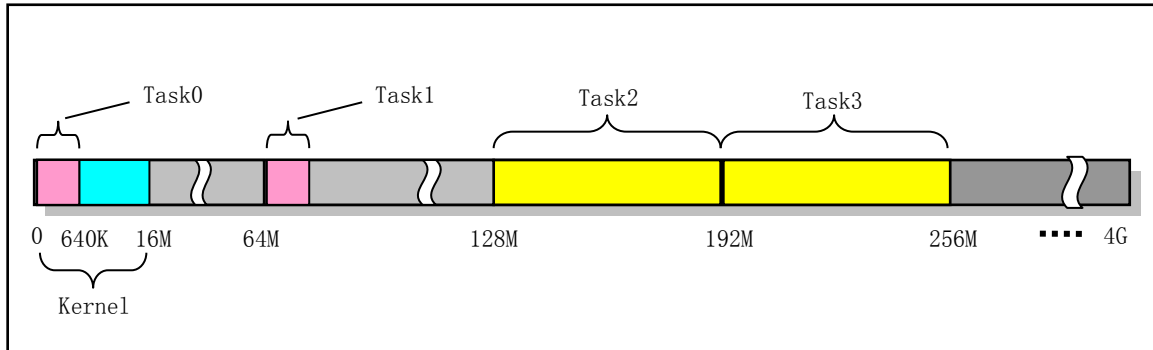


Figure 5-10 Diagram of the use of Linux 0.12 linear address space

If the tasks in the virtual space are also arranged according to the order of the tasks in the linear space, then we can have the system shown in Figure 5-11 and have a schematic diagram of all tasks in the virtual address space, and the virtual space range is also 4 GB. It does not consider the scope of kernel code and data in the virtual space. In addition, in the figure, the positions of the code segments and data segments (including data and stack contents) in the respective logical spaces are also given for task 2 and task 3.

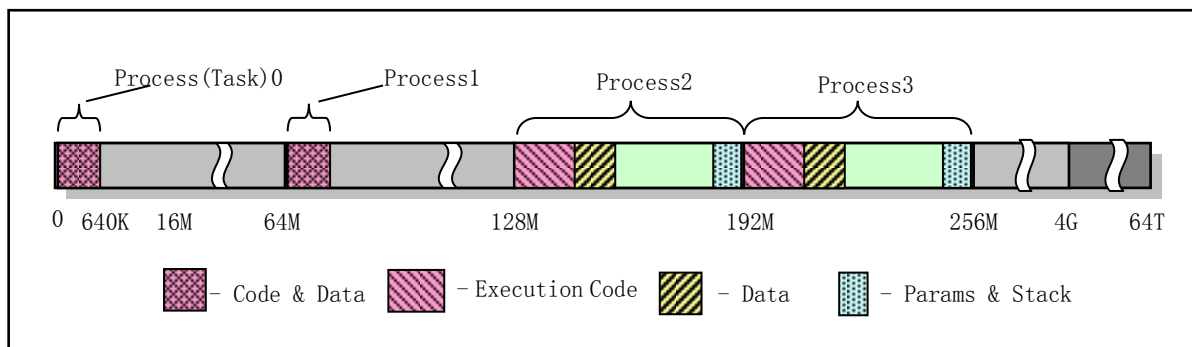


Figure 5-11 The spatial extent of tasks in virtual space in Linux 0.12

It should also be noted that the concept of Code Section and Data Section in the task logical address space is not the same concept as the code segment and data segment in the CPU segmentation mechanism. In the segmentation of the CPU, the concept of a segment determines the purpose of a segment in a linear address space and the constraints and restrictions that are enforced or accessed. Each segment can be placed anywhere in the 4GB linear address space, and they can be independent of each other. They can also overlap completely or partially. The code section and data section of a task refer to the code area, initialization & uninitialization data area, and stack area specified in the process logic space specified by the compiler when compiling the program and when the operating system loads the program. The structure of the code segment and data segment in the task logical address space is shown in Figure 5-12. The nr in the figure is the process or task number, and start_code is the starting location of the process in the linear address space. All other variables contain values in the process logic space.

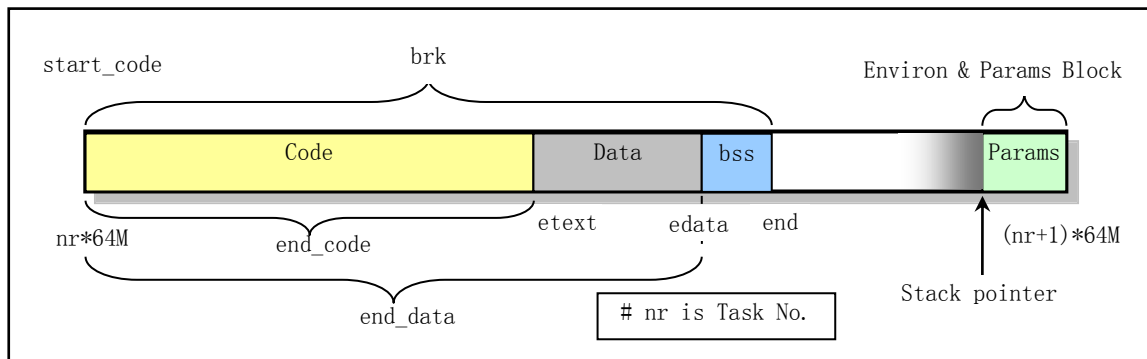


Figure 5-12 The distribution of task code and data in its logical address space

5.3.5 CPU multitasking and protection

The protection mechanism of the 80X86 CPU has four protection levels, with level 0 having the highest priority and level 3 having the lowest priority. The Linux 0.12 operating system uses two protection levels of CPU, 0 and 3. Each task has its own code and data area. These two areas are stored in the local address space, so other tasks in the system are invisible (not accessible). The kernel code and data are shared by all tasks, so it is stored in the global address space. A schematic of this structure is shown in Figure 5-13. The concentric circles in the figure represent the protection level of the CPU (protection layer), and only the 0 and 3 levels of the CPU are used here. Radial rays are used to distinguish between tasks in the system. Each radial ray indicates the boundary of each task. Except for the global address area of each task virtual address space, the address in task 1 is independent of the same address in task 2.

When a task (process) executes a system call and is executed in kernel code, we call the process in kernel operation (or simply kernel mode). At this point the processor is executed in the highest privileged level (level 0). When the process is in kernel mode, the executed kernel code uses the kernel stack of the current process and each process has its own kernel stack. When the process is executing the user's own code, it is said to be in the user's running state (user mode). That is, the processor is now running in the lowest privileged level (level 3) user code.

When the user program is being executed and suddenly interrupted to execute the interrupt handler procedure, the user program can also be symbolically referred to as the kernel state of the process. Because the interrupt handler will use the kernel stack of the current process. This is somewhat similar to the state of a process in kernel mode. The kernel state and user mode of the process are described in more detail later in the section on process running states.

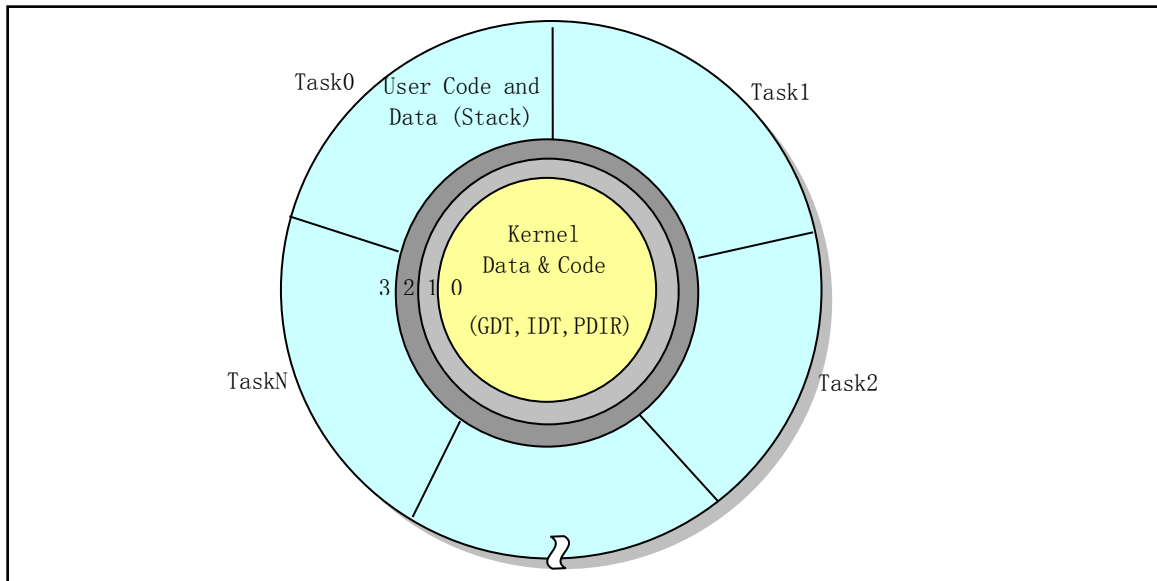


Figure 5-13 Multitasking protection system

5.3.6 Virtual Addresses, Linear Addresses, and Physical Addresses

Earlier we explained the CPU memory management method according to the memory segmentation and paging mechanism. Now let's take the Linux 0.12 system as an example to explain the correspondence between the kernel code and data and the code and data of each task in the virtual address space, linear address space and physical address space. Since the generation or creation process of Task 0 and Task 1 is special, we will describe them separately.

Kernel code and data address

For Linux 0.12, both the kernel code segment and the data segment have been set to a segment of length 16MB in the initialization operation of the head.s program. The range of the two segments overlaps in the linear address space, starting from linear address 0 to address 0xFFFFF for a total of 16MB address range. This range contains all the kernel code, kernel segment tables (GDT, IDT, TSS), page directory tables and secondary page tables, kernel local data, and kernel temporary stack (which will be used as the user stack for task 0). Its page directory table and secondary page table have been set to map the linear address space of 0--16MB to the physical address one by one, occupying 4 directory entries, that is, 4 secondary page tables. So for kernel code or data addresses, we can think of them directly as addresses in physical memory. At this time, the relationship between the virtual address space, the linear address space, and the physical address space of the kernel can be represented by figure 5-14.

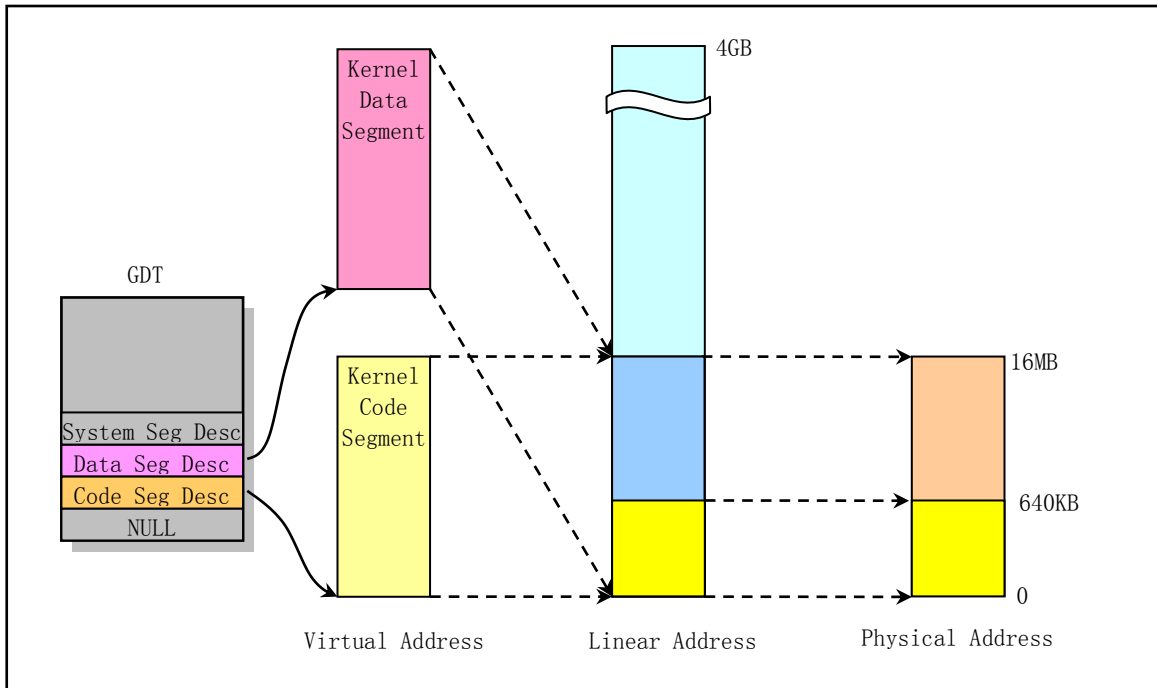


Figure 5-14 Kernel code and data segments in three address spaces

Therefore, by default, the Linux 0.12 kernel can manage up to 16MB of physical memory, with a total of 4096 physical pages (page frames), 4KB per page. Through the above analysis, it can be seen that:

- ◆ The kernel code and the data segments region are the same in the linear and physical address space. This setting can greatly simplify the initialization of the kernel.
- ◆ GDT and IDT are in the kernel data segment, so their linear addresses are also equal to their physical addresses. In the setup.s program initialization operation in real mode, we have set the temporary GDT and IDT, which must be set before entering the protection mode. Since the two tables were in physical memory at about 0x90200, the kernel system module was in the physical memory 0 start position after entering protected mode, and the space at 0x90200 would be used for other purposes (for caching). So after entering protected mode, we need to reset the two tables in the first program head.s that are running. That is, setting GDTR and IDTR to point to the new GDT and IDT, the descriptor also needs to be reloaded. However, since the position of the two tables does not change when the paging mechanism is turned on, there is no need to re-establish or move the table position.
- ◆ Except for task 0, the physical memory pages used by all other tasks are at least partially different from the pages in the linear address, so the kernel needs to dynamically map them in the main memory area to dynamically create page directory entries and page table entries. . Although the code and data of task 1 are also in the kernel, since it needs to be allocated separately to obtain memory, it also needs its own mapping table entry.

Although Linux 0.12 can manage 16MB of physical memory by default, it is not necessary to have such physical memory in the system. As long as there are 4MB (or even 2MB) of physical memory in the machine, you can run Linux 0.12 system. If the machine has only 4MB of physical memory, then the kernel 4MB--16MB address range will be mapped to the non-existing address. But this does not hinder the operation of the system. Because the kernel memory manager knows the exact amount of physical memory in the machine at

initialization time, it does not let the CPU paging mechanism map the linear address page to the 4MB--16MB that does not exist. The default setting in the kernel is mainly to facilitate the expansion of the system's physical memory, and actually does not use the physical memory area that does not exist. For the system has more than 16MB of physical memory, because the initialization of the `init/main.c` program limits the use of more than 16MB of memory, and here the kernel only maps the memory range of 0--16MB. Therefore, physical memory above 16MB will not be used.

Of course, we can extend this limitation by adding some page tables to the kernel here and making minor changes to the `init/main.c` program. For example, if there are 32MB of physical memory in the system, we need to create 8 secondary page table entries for the kernel code and data segments to map the 32MB linear address range to physical memory.

The address correspondence of task 0

Task 0 is the first task manually initiated in the system. Its code and data segment length are set to 640KB. The code and data for this task are included directly in the kernel code and data, and are 640KB of content starting from linear address 0. Therefore, it can directly use the page directory and page table that the kernel has set to perform paging address translation. Similarly, its code and data segments overlap in the linear address space. The corresponding task status segment TSS0 is also manually pre-set and located in the task 0 data structure information. See the data starting with line 156 in the `include/linux/sched.h`. The TSS0 segment is located in the code of the kernel `sched.c` file and has a length of 104 bytes. For details, see the "Task 0 Structure Information" item in Figure 5-24. The mapping correspondence in the three address spaces is shown in Figure 5-15.

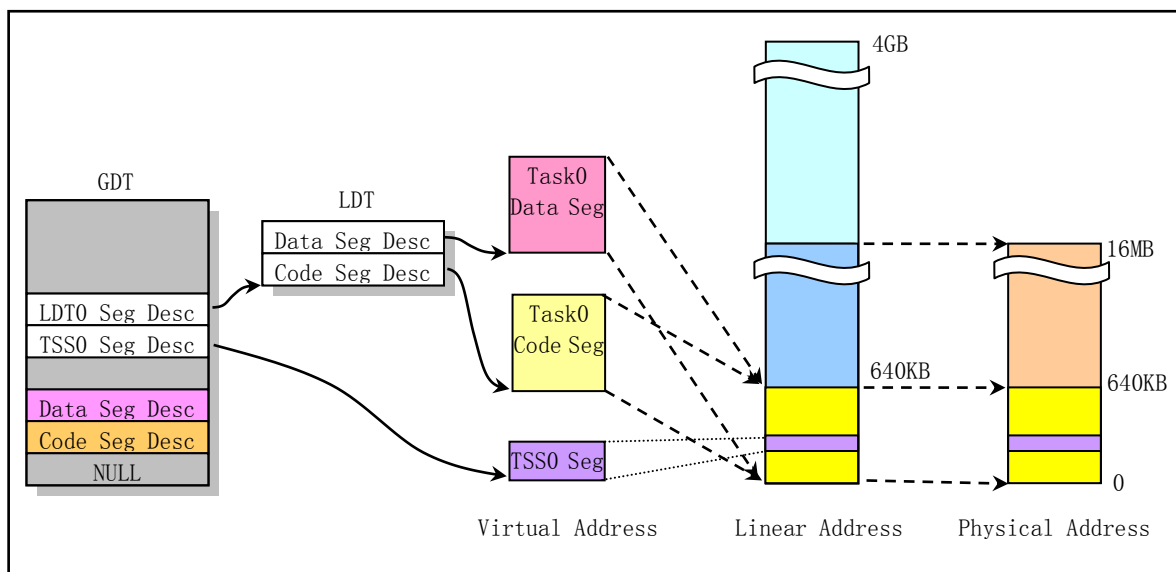


Figure 5-15 The relationship of task 0 in three address spaces

Since task 0 is directly included in the kernel code, there is no need to allocate additional memory pages for it. The kernel-mode stack and user-mode stack space required for its operation are also in the kernel code area, and since the kernel page initialization (`head.s`), the properties of these kernel pages in the page table entry have been set to `0b111`. That is, the corresponding page user can read and write and exist. Therefore, although the user stack `user_stack[]` space is in kernel space, task 0 can still read and write to it.

Task 1 address correspondence

Similar to task 0, task 1 is also a special task. Its code is also in the kernel code area. Different from task 0, in the linear address space, when the task (init process) is created using `fork()`, the system stores a page of memory in the main memory area for storing the secondary page table of task 1. The page directory and the secondary page table entry of the parent process (task 0) are copied. Therefore, task 1 has its own page directory and page table entry, which maps the linear space range of task 1 from 64MB to 128MB (actually 64MB to 64MB + 640KB) to the physical address 0--640KB. At this time, the length of task 1 is also 640KB, and its code segment and data segment overlap, occupying only one page directory entry and one secondary page table. In addition, the system will also request a page of memory for task 1 to store its task data structure and kernel stack space. The task data structure (also called process control block PCB) information includes the TSS segment structure information of task 1. See Figure 5-16.

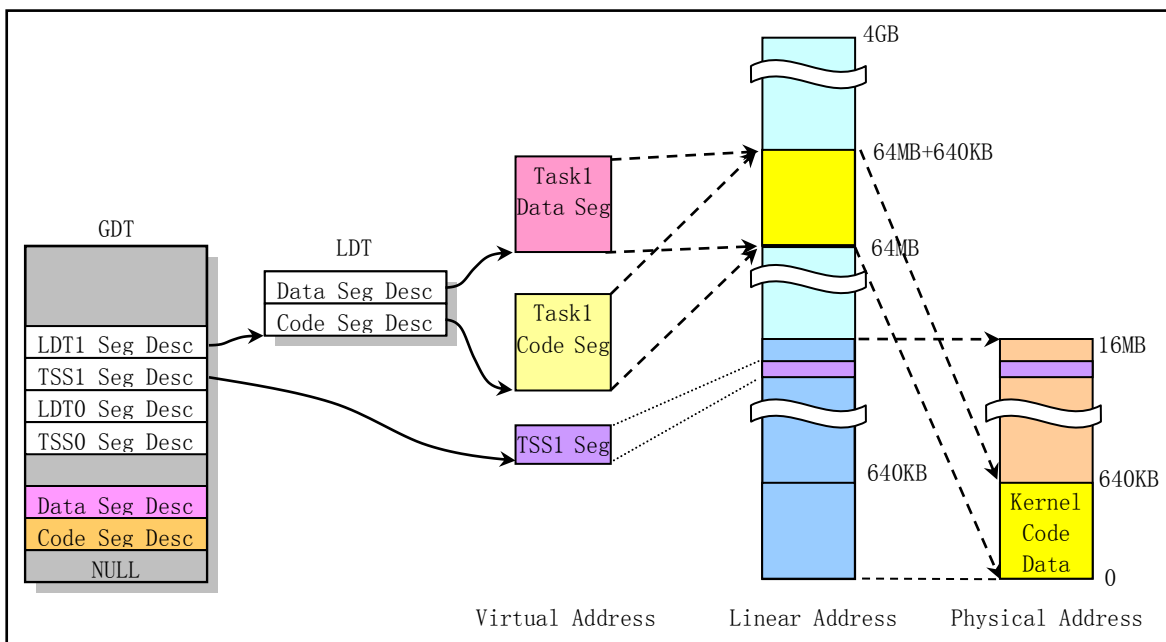


Figure 5-16 Task 1 relationship in three address spaces

The user-mode stack space of task 1 will directly share the user-mode stack space `user_stack[]` of task 0 in kernel code and data region (linear address 0--640KB) (see `kernel/sched.c`, lines 82--87). Therefore this stack needs to be "clean" until Task 1 is actually used to ensure that the stack being copied for Task 1 does not contain useless data. At the beginning of the creation of Task 1, the user-mode stack `user_stack[]` of Task 0 is shared with Task 1, but when Task 1 starts running, the page table entry mapped to `user_stack[]` is set to read-only. This causes Task 1 to cause a write page exception when performing a stack operation, so that the kernel allocates the main memory area page as the user stack space.

Address correspondence of other tasks

For the other tasks that were created starting from Task 2, their final parent processes are all `init` (task 1) processes. We already know that there are 64 processes in the Linux 0.12 system. Below we use task 2 as an example to illustrate the use of address space by any other task.

Starting from task 2, if the task number is represented by `nr`, the starting position of task `nr` in the linear address space will be set at $nr * 64\text{MB}$. For example, the starting position of task 2 = $nr * 64\text{MB} = 2 * 64\text{MB} =$

128MB. The maximum length of the task code segment and data segment is set to 64MB, so Task 2 occupies a linear address space ranging from 128MB to 192MB, occupying a total of $64\text{MB}/4\text{MB} = 16$ page directory entries. The task code segments and data segments in the virtual space are mapped to the same range of linear address spaces, so they also completely overlap. Figure 5-17 shows the correspondence between the code segment of the task 2 and the data segment in the three address spaces.

After task 2 is created, the `execve()` function will be run in it to execute the shell program. When the kernel just created task 2 through replication of task 1, in addition to occupying a linear address space range (128MB--128MB+640KB), the relationship between the code and data of task 2 in the three address spaces is similar to that of task 1. When the code of task 2 (`init()`) calls the `execve()` system call to start loading and executing the shell program, the system call releases the page directory and page table entries and corresponding memory pages copied from task 1. Then re-create the relevant page directory and page table entries for the new executor shell. Figure 5-17 shows the situation when task 2 starts executing the shell program, that is, the case where the code and data of task 2 were originally copied by the code segment and data segment of the shell program. The figure shows a situation where one page of physical memory pages has been mapped. Note here that when executing the `execve()` function, although the system allocates a 64MB space range for Task 2 in the linear address space, the kernel does not immediately allocate and map physical memory pages for it. The memory manager allocates and maps a page of physical memory into its linear address space in the main memory area only when an exception occurs due to a lack page fault when task 2 begins execution. This method of allocating and mapping physical memory pages is called load on demand. See the related description in the Memory Management chapter.

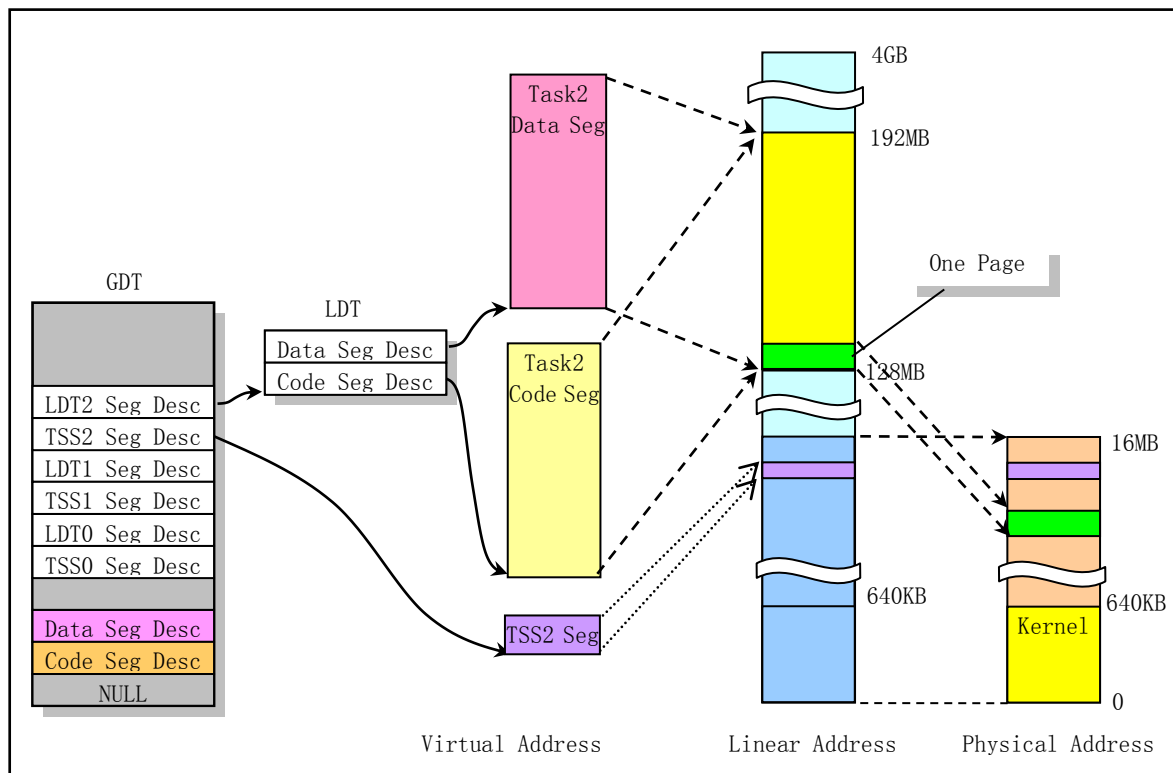


Figure 5-17 Correspondence in other task address spaces

Since the Linux kernel version 0.99, the use of memory space has changed. By using the page directory table independently, each process can enjoy the entire 4G address space range. If we can understand the

memory management concept described in this section, then the memory management principles used in the Linux 2.x kernel that is currently used can be immediately understood. Due to space limitations, this will not be explained here.

5.3.7 User application for memory dynamic allocation

When the user program uses the memory allocation function `malloc()` in the C library to apply for memory, the memory capacity or size of these dynamic applications is managed by the high-level C library function `malloc()`, and the kernel itself does not intervene. Because the kernel has allocated 64MB of space in the 4G linear address space of the CPU for each process (except tasks 0 and 1, which are resident in memory with the kernel code). Therefore, as long as the range of the task or process execution is within the 64MB range, the kernel will also automatically allocate the physical memory page and map the operation for the corresponding page through the memory page fault management mechanism.

But the kernel maintains a current position variable `brk` for the code and data space used by the process. This variable value is stored in the data structure of each process. It indicates the end position of the process code and data (including the dynamically allocated data space) in the process address space. When the `malloc()` function allocates memory for the program, it notifies the kernel of the length of the space requested by the program by the system call `brk()`. The kernel code can update the value of `brk` based on the information provided by `malloc()`. However, the physical memory page is not mapped for the newly requested space at this time. Only when the program addresses an address that does not have a corresponding physical page, the kernel performs mapping operations on the relevant physical memory page.

If the page where a certain data is addressed by the process code does not exist, and the location of the page belongs to the process heap scope, that is, it does not belong to the memory range corresponding to the executable file image file, the CPU generates a page fault exception. And allocate and map a page of physical memory pages for the specified page in the exception handler. As for the memory size of the application and the specific location in the corresponding physical page, the memory allocation function `malloc()` in the C library is responsible for management. The kernel allocates and maps physical memory in units of pages. This function specifically records how many bytes of memory is used by the user program. The remaining capacity will be reserved for use when the program re-applies for memory.

When the user program uses the function `free()` to dynamically release the requested memory block, the memory management function in the C library marks the released memory block as free, in case the program requests the memory again. The physical page allocated by the kernel for this process will not be released during this process. Only when the process ends up will the kernel fully reclaim all physical memory pages that have been allocated and mapped to the process's address space.

The specific code implementation for the library functions `malloc()` and `free()` can be found in the `lib/malloc.c` program in the kernel library.

5.4 Interrupt mechanism

This section describes the basic principles of the interrupt mechanism and related programmable controller hardware logic, as well as methods for using interrupts in Linux systems. For the specific programming method of the programmable controller, please refer to the description after the `setup.s` program in the next chapter.

5.4.1 Principle of Interrupt Operation

Microcomputer systems typically include input and output devices. One way the processor provides

services to these devices is to use polling. In this method the processor sequentially queries each device in the system and "queries" if they need service. The advantage of this method is that software programming is simple, but the disadvantage is that it consumes processor resources and affects system performance.

Another way to provide services to a device is to make a request to the processor itself when the device needs service. The processor also serves the device only when requested by the device. When the device makes a service request to the processor, the processor will respond to the device's request as soon as the current instruction is executed, and then execute the relevant service program of the device. When the service program is executed, the processor will continue to do the program that was just interrupted. This type of processing is called an interrupt method, and the service request that the device sends to the processor is called an IRQ - Interrupt Request. The device-related program that the processor executes in response to the request is called an interrupt service routine or an ISR.

The Programmable Interrupt Controller (PIC) is the administrator that manages device interrupt requests in a microcomputer system. It accepts the Terminal Service Request signal from the device through an interrupt request pin connected to the device. When the device activates its interrupt request IRQ signal, the PIC will detect it immediately. In the case of receiving interrupt service requests from several devices at the same time, the PIC will prioritize them and select the highest priority interrupt request for processing. If the processor is currently executing an interrupt service routine for a device, the PIC also needs to compare the selected interrupt request with the priority of the interrupt request being processed and determine whether to issue an interrupt to the processor based on the comparison. When the PIC issues an interrupt to the processor's INT pin (INTR pin in Figure 5-18), the processor immediately stops what was done at that time and asks the PIC which interrupt service request to execute. The PIC informs the processor which interrupt service process to execute by sending an interrupt number corresponding to the interrupt request to the data bus. The processor obtains the interrupt vector of the relevant device (ie, the address of the interrupt service routine) by querying the interrupt vector table (or the interrupt descriptor table IDT in the 32-bit protected mode) according to the read interrupt number and starts executing the interrupt service routine. When the execution of the interrupt service routine ends, the processor continues to execute the program interrupted by the interrupt signal.

What has been described above is the interrupt service processing of the input/output device. But the interrupt method is not necessarily hardware-dependent, it can also be used in software. By using the INT instruction and using its operand to indicate the interrupt number, the processor can be executed to perform the corresponding interrupt processing. The PC/AT series of microcomputers provide support for 256 interrupts, most of which are used for software interrupts or exceptions. The exceptions are interrupts generated by the processor detecting errors during processing. Only some of the interrupts mentioned below are used on the device.

5.4.2 Interrupt subsystem of 80X86 PC

The 8259A programmable interrupt controller chip is used in a microcomputer system composed of 80X86. Each 8259A chip can manage eight interrupt sources. Through multi-chip cascading, the 8259A can form a system that manages up to 64 interrupt vectors. In the PC/AT series compatible PC, two 8259A chips are used to manage 15 levels of interrupt vectors. A schematic diagram of the cascade is shown in Figure 5-18. The INT pin of the slave chip is connected to the IR2 pin of the master chip, that is, the interrupt signal sent by the 8259A slave chip will be the IRQ2 input signal of the 8259A master chip. The port base address of the master 8259A chip is 0x20, and the slave chip is 0xA0. The IRQ9 pin functions the same as the IRQ2 of the PC/XT. That is, the PC/AT machine uses the hardware circuit to redirect the IRQ2 pin of the device using IRQ2 to the IRQ9 pin of the PIC, and uses the software in the BIOS to interrupt the IRQ9. Int 71 redirects to IRQ2 interrupt int 0x0A

interrupt handler procedure. This allows any 8-bit card with PC/XT using IRQ2 to function properly under the PC/AT machine. The backward compatibility of the PC series has been achieved.

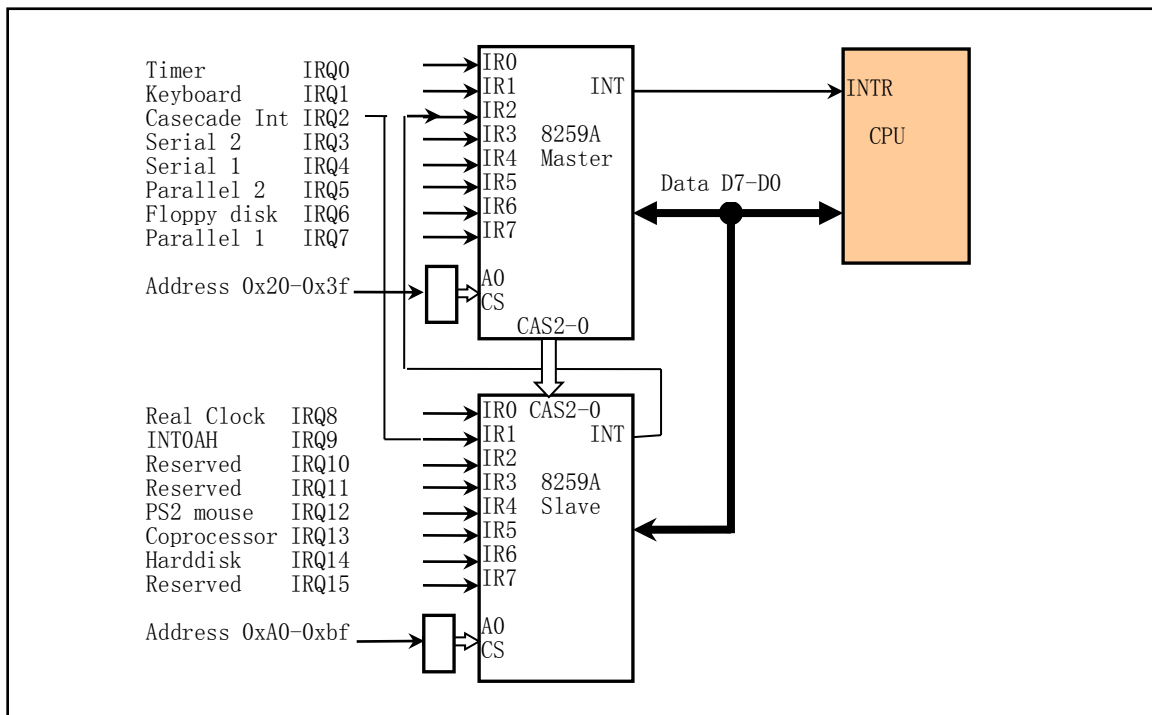


Figure 5-18 PC/AT microcomputer cascaded 8259 control system

Under the control of the bus controller, the 8259A chip can be in the programming state and operating state. The programming state is the state in which the CPU initializes the 8259A chip using the IN or OUT instruction. Once the initialization programming is completed, the chip enters the operating state. At this time, the chip can respond to the interrupt request (IRQ0 – IRQ15) proposed by the external device at any time, and the system can also modify the interrupt processing mode at any time by using the operation command word. Through the interrupt arbitration selection mechanism, the chip will select the current highest priority interrupt request as the interrupt service object, and notify the CPU of the interrupt request by the CPU pin INT. After the CPU responds, the chip sends the programmed interrupt number of the current service object from the data bus D7-D0, and the CPU acquires the corresponding interrupt vector value and executes the interrupt service routine.

5.4.3 Interrupt Vector Table

The previous section has indicated that the CPU fetches the interrupt vector value based on the interrupt number, which corresponds to the entry address value of the interrupt service routine. Therefore, in order for the CPU to find the corresponding interrupt vector from the interrupt number, it is necessary to create a lookup table in the memory, that is, the interrupt vector table (in the 32-bit protection mode, the table is called the interrupt descriptor table IDT, see below).

The 80X86 microcomputer supports 256 interrupts, and an interrupt service routine is required for each interrupt. In the 80X86 real mode mode, each interrupt vector consists of 4 bytes. These 4 bytes indicate the segment value and the intra-segment offset value of an interrupt service routine. Therefore the length of the entire vector table is $4 * 256 = 1024$ bytes. When the 80X86 microcomputer starts up, the program in the ROM

BIOS initializes and sets the interrupt vector table at the physical memory start address 0x0000:0x0000, and the default interrupt service routine for each interrupt is given in the BIOS. Since the vectors in the interrupt vector table are arranged in the order of the interrupt numbers, given an interrupt number N , the position of the corresponding interrupt vector in memory is 0x0000: $N * 4$, that is, the corresponding interrupt service program entry address is stored in Physical memory 0x0000: $N * 4$ position.

When the BIOS performs the initialization operation, it sets the 16 hardware interrupt vectors supported by the two 8259A chips and the interrupt calling function provided by the BIOS with the interrupt number 0x10-0x1F. For interrupts that are not actually used, the vector is filled with the address of the temporary dummy interrupt service routine. Later, when the system boots the operating system, some interrupt vector values will be modified according to actual needs. For example, for the DOS operating system, it will reset and modify the interrupt vector value of interrupt 0x20-0x2F. For Linux systems, in addition to the display and disk read interrupts provided by the BIOS when you first load the kernel, a new interrupt vector table is created. That is, the 8259A chip is reinitialized in the setup.s program, and an interrupt vector table (interrupt descriptor table) is re-established in the head.s program. Therefore, the Linux completely abandons the BIOS interrupt vector table after the kernel is running normally.

When the Intel CPU is running in 32-bit protected mode, you need to use the Interrupt Descriptor Table (IDT) to manage interrupts or exceptions. IDT is a direct replacement for the interrupt vector table used in the Intel 8086 - 80186 CPU. Its role is similar to the interrupt vector table, except that each interrupt descriptor entry contains information about the privilege level and descriptor class in addition to the address of the interrupt service routine. The Linux operating system works in 80X86 protected mode, so it uses the interrupt descriptor table to set and save the "vector" information for each interrupt.

5.4.4 Linux kernel interrupt handling

For the Linux kernel, interrupt signals are usually divided into two categories: hardware interrupts and software interrupts (or exceptions). Each interrupt is identified by a number between 0-255, called the interrupt number. For interrupts INT0--INT31 (0x00--0x1f), the function of each interrupt is fixed or reserved by Intel Corporation, as shown in Table 5-1. As can be seen from the above section, the range of interrupt numbers set by the BIOS conflicts with it.

Table 5-1 Exceptions and Interrupts reserved by Intel Co.

Vector No	Name	Type	Error Code	Signal	Source
0	Divide error	Fault (Error)	No	SIGFPE	DIV and IDIV instructions.
1	Debug	Fault/Trap	No	SIGTRAP	Any code or data reference or the INT instruction.
2	nmi	Interrupt	No		Non maskable external interrupt.
3	Breakpoint	Trap	No	SIGTRAP	INT 3 instruction.
4	Overflow	Trap	No	SIGSEGV	INTO instruction.
5	Bounds check	Fault	No	SIGSEGV	BOUND instruction.
6	Invalid Opcode	Fault	No	SIGILL	UD2 instruction or reserved opcode.
7	Device not available	Fault	No	SIGSEGV	Floating-point or WAIT/FWAIT instruction.
8	Double fault	Abort	Yes(0)	SIGSEGV	Any instruction that can generate an exception, NMI, or an INTR.

9	Coprocessor seg overflow	Abort	No	SIGFPE	Floating-point instruction.
10	Invalid TSS	Fault	Yes	SIGSEGV	Task switch or TSS access.
11	Segment not present	Fault	Yes	SIGBUS	Loading segment registers or accessing system segments.
12	Stack segment	Fault	Yes	SIGBUS	Stack operations and SS register loads.
13	General protection	Fault	Yes	SIGSEGV	Any memory reference and other protection checks.
14	Page fault	Fault	Yes	SIGSEGV	Any memory reference.
15	Intel reserved		No		
16	Coprocessor error	Fault	No	SIGFPE	Floating-point or WAIT/FWAIT
17	Alignment check	Fault	Yes (0)		Any data reference in memory.
20-31	Intel reserved.				
32-255	User Defined interrupts	Interrupt			External interrupt or INT n instruction.

These interrupts are soft interrupts, but Intel calls them exceptions. Because these interrupts are caused by an abnormal condition detected when the CPU executes the instruction. It can usually be divided into two categories: faults and traps. The interrupt INT32--INT255 (0x20--0xff) can be set and defined by the user. The classification of all interrupts and the way the CPU operates after execution are shown in Table 5-2.

Table 5-2 Interrupt classification and how the CPU handles it

Interrupt	Name	CPU Check Method	Processing Method
Hardware	Maskable	CPU pin INTR	Clear the IF maskable interrupt flag of EFLAGS.
	Nonmaskable	CPU pin NMI	Non-Maskable Interrupts.
Software	Fault	Detected before error occurred	CPU re-executes the instruction that caused the error.
	Trap	Detected after error occurred	CPU continues to execute the following instruction.
	Abort	Detected after error occurred	Programs that caused this error should be terminated.

In Linux systems, INT32--INT47 (0x20--0x2f) corresponds to the hardware interrupt request signal IRQ0--IRQ15 issued by the 8259A interrupt control chip (see Table 5-3), and set the software interrupt issued by the user program to INT128 (0x80), called the system call (System Call) interrupt. System call interrupt is the only interface for user programs that use operating system resources.

Table 5-3 List of interrupt numbers for Linux system interrupt requests

Interrupt Request No.	Interrupt No.	Purpose
IRQ0	0x20 (32)	100HZ clock interrupt signal from 8253 chip
IRQ1	0x21 (33)	Keyboard interrupt
IRQ2	0x22 (34)	Cascade to slave chip
IRQ3	0x23 (35)	Serial port 2
IRQ4	0x24 (36)	Serial port 1
IRQ5	0x25 (37)	Parallel port 2
IRQ6	0x26 (38)	Floppy disk drive
IRQ7	0x27 (39)	Parallel port 1

IRQ8	0x28 (40)	Real clock
IRQ9	0x29 (41)	Reserved
IRQ10	0x2a (42)	Reserved
IRQ11	0x2b (43)	Reserved (Network interface)
IRQ12	0x2c (44)	PS/2 mouse
IRQ13	0x2d (45)	Coprocessor
IRQ14	0x2e (46)	Harddisk
IRQ15	0x2f (47)	Reserved

At system initialization, the kernel first uses a dummy interrupt vector (interrupt descriptor) to default settings for all 256 descriptors in the interrupt descriptor table (IDT). This dummy interrupt vector points to a default "no interrupt" handler procedure. The message "Unknown interrupt" is displayed when an interrupt occurs and the interrupt vector has not been reset. For some interrupts that need to be used in the system, the kernel will re-edit the interrupt descriptor items of these interrupts during the process of continuing initialization, so that they point to the corresponding actual handler procedure. Usually, the exception interrupt processing (INT0 -- INT 31) is reset in the initialization function of traps.c, and the system call interrupt int128 is reset in the scheduler initialization function.

In addition, the Linux kernel uses both the interrupt gate and the trap gate descriptors when setting the interrupt descriptor table IDT. The difference between them is the effect on the interrupt enable flag IF in the flag register EFLAGS. The interrupt executed by the interrupt gate descriptor resets the IF flag, so other interrupts can be prevented from interfering with the current interrupt processing. The subsequent interrupt end instruction IRET will restore the original value of the IF flag from the stack. Interrupts that are executed through the trap gate do not affect the IF flag.

5.4.5 Interrupt Flag of Flag Register

In order to avoid contention and disruption of the critical code area, the CLI and STI instructions are used in many places in the Linux 0.12 kernel code. The CLI instruction is used to reset the interrupt flag IF in the CPU flag register so that the system does not respond to external interrupts after executing the CLI instruction. The STI instruction is used to set the interrupt flag in the flag register to allow the CPU to recognize and respond to interrupts from external devices.

When entering a code area that may cause race conditions, the kernel will use the CLI instruction to turn off the response to the external interrupt, and the kernel will execute the STI instruction to re-allow the CPU to respond to the external interrupt when the content code area is executed. For example, when modifying the lock flag of the file super block and the task entry/exit wait queue operation, it is necessary to first use the CLI instruction to disable the CPU from responding to the external interrupt, and then use the STI instruction to enable the response to the external interrupt after the operation is completed. If you do not use the CLI, STI instruction pair, that is, when you need to modify a file super block without using the CLI to disable the response to the external interrupt, then before the modification, it is judged that the super block lock flag is not set and you want to set this flag. If the system clock interrupt occurs just at this time and switches to another task to run, and it happens that other tasks also need to modify the super block, then this other task will first set the lock flag of the super block and modify the super block. When the system switches back to the original task, at this time, the task will not judge the lock flag and will continue to execute the lock flag of the set super block, thereby causing two tasks to simultaneously perform multiple operations on the critical code area, causing the super block data. Inconsistency can lead to kernel system crashes in severe cases.

5.5 Linux system calls

5.5.1 System Call Interface

System calls (commonly referred to as syscalls) are the only interfaces that the Linux kernel can communicate with upstream applications, as shown in Figure 5-4. From the description of the interrupt mechanism, the user program can use kernel resources, including system hardware resources, by calling interrupt int 0x80 directly or indirectly (through the library function) and specifying the system call function number in the EAX register. However, usually the application uses the kernel's system calls indirectly using functions in the C library with standard interface definitions, as shown in Figure 5-19.

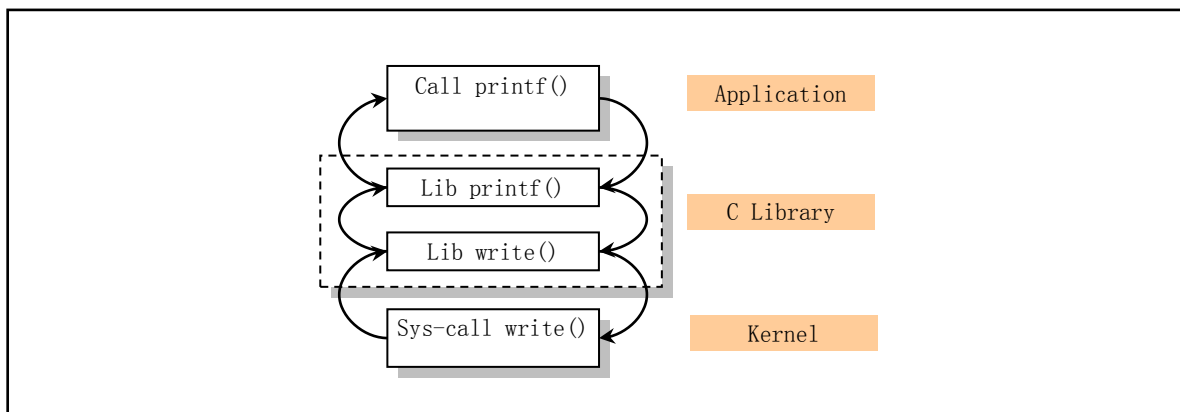


Figure 5-19 Relationship between user programs, library functions, and kernel system calls

Usually system calls are made using a functional form, so they can take one or more parameters. For the result of the system call execution, it will be represented in the return value. Usually a negative value indicates an error and a 0 indicates success. In the case of an error, the wrong type code is stored in the global variable `errno`. By calling the library function `perror()`, we can print out the error string information corresponding to the error code.

In the Linux kernel, each system call has a unique system call function number. Kernel 0.12 has a total of 87 system call functions (0-86). These feature numbers are defined at the beginning of line 62 in the file `include/unistd.h`. For example, the `write` system call has a function number of 4, defined as the symbol `__NR_write`. These system call function numbers actually correspond to the index of the items in the system call handler pointer array table `sys_call_table[]` defined in `include/linux/sys.h`. So the handler pointer for the `write()` system call is at item 4 of the array.

When we want to use these system call symbols directly in our own program, we need to define the symbol `__LIBRARY__` before including the file `"<unistd.h>"` as shown below.

```

#define __LIBRARY__
#include <unistd.h>
  
```

In addition, we can see from `sys_call_table[]` that the names of all system call handlers in the kernel basically start with the symbol `'sys_'`. For example, the implementation function of the system call `read()` in the kernel source code is `sys_read()`.

5.5.2 System Call Processing

When the application issues an interrupt INT 0x80 to the kernel via a library function, a system call is initiated. The system call number is stored in the register EAX, and the carried parameters can be stored in the registers EBX, ECX and EDX in turn. Therefore, the user program in the Linux 0.12 kernel can pass up to three parameters directly to the kernel. Of course, you can also take no parameters. The process of handling the system call interrupt INT 0x80 is the `system_call` in the program `kernel/system_call.s`.

In order to facilitate the execution of system calls, the kernel source code defines the macro function `_syscalln()` in the `include/unistd.h` file (150-200 lines), where `n` represents the number of parameters carried, which can be 0 to 3 respectively. Therefore, up to 3 parameters can be passed directly. If you need to pass large chunks of data to the kernel, you can pass the pointer of the chunk data. For example, for the `read()` system call, its definition is:

```
int read(int fd, char *buf, int n);
```

If we execute the corresponding system call directly in the user program, the macro of the system call is:

```
#define __LIBRARY__
#include <unistd.h>

_syscall3(int, read, int, fd, char *, buf, int, n)
```

So we can use the above `_syscall3()` directly in the user program to execute a system call `read()` instead of mediating through the C library. In fact, the form of the function call in the C function library is the same as that given here.

For each system call macro given in `include/unistd.h`, there are $2+2*n$ parameters. The first parameter corresponds to the type of the system call return value; the second parameter is the name of the system call; followed by the type and name of the parameter carried by the system call. This macro will be extended to a C function that contains inline assembly statements, as shown below.

```
int read(int fd, char *buf, int n)
{
    long __res;
    __asm__ volatile (
        "int $0x80"
        : "=a" (__res)
        : "0" (__NR_read), "b" ((long) (fd)), "c" ((long) (buf)), "d" ((long) (n)));
    if (__res >= 0)
        return int __res;
    errno = -__res;
    return -1;
}
```

It can be seen that this macro is expanded to be a concrete implementation of a read system call. It uses the embedded assembly statement to execute the Linux system interrupt call 0x80 with the function number `__NR_read(3)`. This interrupt call returns the actual number of bytes read in the EAX (`__res`) register. If the returned value is less than 0, it means that the read operation error occurs, so the error number is inverted and

stored in the global variable `errno`, and the value of -1 is returned to the calling program.

If a system call requires more than 3 parameters, the kernel usually uses the parameter as a parameter buffer block and passes the pointer of the buffer block as a parameter to the kernel. So for system calls with more than 3 parameters, we only need to use the macro `_syscall1()` with one argument to pass the pointer of the first argument to the kernel. For example, the `select()` function system call has 5 arguments, but we only need to pass a pointer to its first argument, see the description of the `fs/select.c` program.

When entering the kernel call handler `kernel/sys_call.s` in the kernel, the `system_call` code will first check if the system call function number in `EAX` is within the valid system call number range. Then execute the corresponding system call handler according to the `sys_call_table[]` function pointer table call.

```
call _sys_call_table(,%eax, 4)           // kernel/sys_call.s 第 99 行。
```

The meaning of this assembly statement operand is: indirectly call the function at `_sys_call_table + %eax * 4`. Since the `sys_call_table[]` pointer is 4 bytes each, it is necessary to multiply the system call function number by 4. Then use the resulting value to get the address of the called handler from the table.

5.5.3 Parameter Passing Method of Linux System Call

Regarding Linux user processes passing parameters to the system interrupt call procedure, Linux systems use general-purpose register transfer methods such as registers `EBX`, `ECX`, and `EDX`. A significant advantage of this method of using register-passing parameters is that when the system interrupt service routine is entered and the register values are saved, the registers that pass the parameters are also automatically placed on the kernel-state stack. Therefore, there is no need to specialize the special processing of the registers that pass the parameters. This is the simplest and fastest method of parameter transfer that Mr. Linus knew at the time. There is also a parameter transfer method using the system call gate provided by the Intel CPU, which automatically copies the passed parameters in the process user state stack and the kernel state stack. But the method used in this method is more complicated.

In addition, the parameters passed should be verified in each system call handler to ensure that all parameters are legal and valid. In particular, user-supplied pointers should be rigorously reviewed to ensure that the range of memory regions pointed to by the pointer is valid and has appropriate read and write permissions.

5.6 System time and timing

5.6.1 System time

In order to allow the operating system to automatically and accurately provide current time and date information, battery-powered real-time RT (Real Time) circuit support is provided in the PC/AT microcomputer system. Usually, this part of the circuit is integrated on a single chip with a small amount of CMOS RAM that holds system information, so this part of the circuit is called an RT/CMOS RAM circuit. Motorola's MC146818 chip is used in the PC/AT microcomputer or its compatible machine.

At initialization, the Linux 0.12 kernel reads the current time and date information stored in the chip and converts it to the current time in seconds from 0:00 on January 1, 1970. We call this the UNIX calendar time. This time determines the calendar time at which the system starts running and is saved in the global variable `startup_time` for all kernel code to use. The user program can use the system call `time()` to read the value of

startup_time, while the superuser can modify the system time value by calling stime().

In addition, the program can uniquely determine the current running time by the following system tick value jiffies counted from the system start. Each tick is generated by a timer as described below. Since each tick timing value is 10 milliseconds, a macro is defined in the kernel code to facilitate access to the code at the current time. This macro is defined on line 192 of the include/linux/sched.h file and has the following form:

```
#define CURRENT_TIME (startup_time + jiffies/HZ)
```

Among them, $HZ = 100$, is the core system clock frequency. The current time macro `CURRENT_TIME` is defined as the system boot time `startup_time` plus the time `jiffies/100` of the system running after booting. This macro is used when modifying the time when a file is accessed or when its i-node is modified.

5.6.2 System Timing

The basic timing beat of the system is generated by the timing chip. During the initialization of the Linux 0.12 kernel, the counter channel 0 of the PC's programmable timing chip Intel 8253 (8254) is set to operate in mode 3, and the initial count value `LATCH` is set to emit a square wave rising edge at output `OUT` every 10 milliseconds. Since the clock input frequency of the 8254 chip is 1.193180 MHz, the initial count value `LATCH=1193180/100` is approximately 11931. Since the `OUT` pin is connected to level 0 of the programmable interrupt control chip, the system issues a clock interrupt request (`IRQ0`) every 10 milliseconds. This time beat is the pulse of the operating system, we call it a system tick or a system clock cycle. Therefore, every a tick time elapses, the system will call the clock interrupt handler (`timer_interrupt`).

The clock interrupt handler `timer_interrupt` is mainly used to accumulate the number of clock ticks that have passed since the system was started by the `jiffies` variable. The `jiffies` value is incremented by one each time a clock interrupt occurs. Then call the C language function `do_timer()` for further processing. The parameter `CPL` with the call is obtained from the segment selector of the interrupted program (the `CS` segment register value stored in the stack) to obtain the current code privilege level `CPL`.

The `do_timer()` function accumulates the current process runtime based on the privilege level. If `CPL=0`, it means that the process is interrupted when it runs in kernel mode, so the kernel will increase the kernel state running time `stime` of the process by one, otherwise it will increase the running value of the process user state by one. If the floppy disk program `floppy.c` adds a timer during the operation, the timer list is processed. If a timer expires (equal to 0 after decrement), the handler of the timer is called. Then the current process running time is processed, and the current process running time slice is decremented by one. A time slice is the CPU time that a process can continue to run before being switched out. The unit is the number of ticks defined above.

If the process time slice value is decremented and is still greater than 0, it means that its time slice has not been used up, so it exits `do_timer()` and continues to run the current process. If the process time slice has been decremented to 0 at this time, it means that the process has used up the time slice of the CPU, and the program will determine the further processing method according to the level of the interrupted program. If the interrupted current process is working in user mode (privilege level is greater than 0), then `do_timer()` will call the scheduler `schedule()` to switch to another process to run. If the interrupted current process is working in kernel mode, that is, it is interrupted while running in the kernel program, `do_timer()` will exit immediately. Therefore, this way of processing determines that the Linux system process will not be switched by the scheduler when running in kernel mode. That is, the process is nonpreemptive when running in a kernel mode program.

Note that the above timer code is dedicated to floppy motor turn-on and turn-off timing operations. This

kind of timer is similar to the dynamic timer in modern Linux systems and is only used by the kernel. Such timers can be dynamically created when needed and dynamically revoked when the timing expires. In the Linux 0.12 kernel, there are up to 64 timers at the same time. The processing code for the timer is in the sched.c program 283--368 lines.

5.7 Linux Process Control

A program is an executable file, and a process is an instance of a program that is executing. With time-sharing technology, multiple processes can be run simultaneously on the Linux operating system. The basic principle of time-sharing technology is to divide the running time of the CPU into time slices of a specified length, so that each process runs in one time slice. When the time slice of the process runs out, the system uses the scheduler to switch to another process to run. So in fact, for a machine with a single CPU, only one process can be run at a time. But since each process runs a short time slice (for example, 15 system tick = 150 milliseconds), it appears as if all processes are running at the same time.

For the Linux 0.12 kernel, the system can have up to 64 processes at the same time. Except for the first process which created "manually", the rest are new processes created by existing processes using the system call fork. The created process is called the child process, and the creator is called the parent process.

The kernel program uses the process ID (process ID, pid) to identify each process. A process consists of executable instruction code, data, and a stack sections. The code and data parts in the process correspond to the code segments and data sections in one execution file respectively. Each process can only execute its own code and access its own data and stack area. Communication between processes needs to be done through system calls. For systems with only one CPU, only one process can be running at a time. The kernel schedules each process to run in a time-sharing manner through the scheduler.

We already know that a process in a Linux system can be executed in kernel mode or user mode, and each uses its own separate kernel state stack and user state stack. The user stack is used by the process to temporarily save the parameters of the calling function, local variables, etc. in the user state; the kernel stack contains the information when the kernel program executes the function call.

Also in the Linux kernel, processes are often referred to as tasks, and programs running in user space are called processes. This book will mix these two terms while trying to follow this default rule.

5.7.1 Task Data Structure

The kernel program manages the process through the process table, and each process occupies one item in the process table. In a Linux system, a process table entry is a task_struct task structure pointer. Some books refer to it as process control block (PCB) or process descriptor (PD). It holds all the information used to control and manage the process. It mainly includes the status information of the current running of the process, the signal, the process number, the parent process number, the running time accumulated value, the file being used, the local descriptor of the task, and the task status segment information. The task data structure is defined in the header file include/linux/sched.h. The specific meaning of each field of the structure is as follows.

```
struct task_struct {  
    long state;           // -1 unrunnable, 0 runnable (ready), > 0 stopped.  
    long counter;         // Task run time tick (decrement), run time slice.  
    long priority;        // Priority. When task starts running, counter=priority.  
    long signal;          // Signal bitmap, each bit is a signal( = bit offset + 1).
```

```

struct sigaction sigaction[32]; // Signal attribute struct. Signal operation and flags.
long blocked;                  // Process signal mask (Bitmap of masked signal).
int exit_code;                  // Exit code after task stops, its parent will get it.
unsigned long start_code;       // Code start location in linear address space.
unsigned long end_code;         // Code length or size (bytes).
unsigned long end_data;         // Code size + data size (bytes).
unsigned long brk;              // Total size (number of bytes).
unsigned long start_stack;      // Stack bottom location.
long pid;                       // Process identifier.
long pgrp;                      // Process group number.
long session;                   // Process session number.
long leader;                    // Leader session number.
int groups[NGROUPS];            // Group numbers. A process can belong to more groups.
task_struct *p_pptr;            // Pointer to parent process.
task_struct *p_cptr;            // Pointer to youngest child process.
task_struct *p_ysptr;           // Pointer to younger sibling process created afterwards.
task_struct *p_osptr;           // Pointer to older sibling process created earlier.
unsigned short uid;             // User id.
unsigned short euid;            // Effective user id.
unsigned short suid;            // Saved user id.
unsigned short gid;             // Group id.
unsigned short egid;            // Effective group id.
unsigned short sgid;            // Saved group id.
long timeout;                   // Kernel timing timeout value.
long alarm;                     // Alarm timing value (ticks).
long utime;                     // User state running time (ticks).
long stime;                     // System state runtime (ticks).
long cutime;                    // Child process user state runtime.
long cstime;                    // Child process system state runtime.
long start_time;                // Time the process started running.
struct rlimit rlim[RLIM_NLIMITS]; // Resource usage statistics array.
unsigned int flags;              // per process flags.
unsigned short used_math;        // Flag: Whether a coprocessor is used.
int tty;                        // The tty subdevice number used. -1 means no use.
unsigned short umask;            // The mask bit of the file creation attribute.
struct m_inode *pwd;             // Current working directory i node structure pointer.
struct m_inode *root;           // Root i-node structure pointer.
struct m_inode *executable;     // The pointer to i-node structure of the executable file.
struct m_inode *library;        // The loaded library i-node structure pointer.
unsigned long close_on_exec;     // A bitmap flags that close file handles on execution.
struct file *filp[NR_OPEN];     // File structure pointer table, up to 32 items.
                                // The index is the value of file descriptor.
struct desc_struct ldt[3];       // LDT. 0-empty, 1-code seg cs, 2-data & stack seg ds & ss.
struct tss_struct tss;          // The task status segment structure TSS of the process.
};

```

◆long state -- The state field contains current state of the process. At some point, a Linux process can be in one of five states and can transition between these states under the operation of the kernel scheduler. The five states are: running state (TASK_RUNNING), interruptible sleep state (TASK_INTERRUPTIBLE), uninterruptible sleep state (TASK_UNINTERRUPTIBLE), zombie state (TASK_ZOMBIE), and stopped state (TASK_STOPPED). The way the kernel changes the state of the process is described in the next section.

◆long counter -- The counter field holds the number of time ticks that the process can execute before it is

temporarily stopped. That is, it usually takes several system clock cycles to switch to another process. The scheduler uses the counter value of the process to select the next process to execute, so the counter can be thought of as a dynamic feature of a process. The initial value of the counter is equal to the priority when a process has just been created.

◆ **long priority** -- Priority is used to assign the initial value to the counter. In Linux 0.12 this initial value is 15 system clock cycle times (15 ticks). When needed, the scheduler will use the value of priority to assign an initial value to the counter, see the `sched.c` and the `fork.c` programs. Of course, the unit of priority is also the number of time ticks.

◆ **long signal** -- The signal field is a bitmap of the signal currently received by the process. The bitmap has 32 bits, each bit represents a signal, and the signal value = bit offset value + 1. So the Linux kernel has up to 32 signals. At the end of each system call process, the signal is preprocessed using the signal bitmap.

◆ **struct sigaction sigaction[32]** -- The sigaction structure array is used to store the operations and attributes used to process each signal. Each item of the array corresponds to one signal.

◆ **long blocked** -- The blocked field is a blocking bitmap of the signal that the process does not currently want to process. Similar to the signal field, each bit represents a blocked signal.

◆ **int exit** -- The exit field is used to save the exit code when the program terminates. After the child process ends, the parent process can query its exit code.

◆ **unsigned long start_code** -- The start_code field is the starting address of the process code in the CPU linear address space. In the Linux 0.1x kernel, its value is an integer multiple of 64MB.

◆ **unsigned long end_code** -- The end_code field holds the byte length value of the process code.

◆ **unsigned long end_data** -- The end_data field holds the code length of the process + the total byte length value of the data length.

◆ **unsigned long brk** -- The brk field is also the total byte length value (pointer value) of the process code and data, but also includes the uninitialized data area bss, as shown in Figure 5-12. This is the initial value of brk when a process starts executing. By modifying this pointer, the kernel can add and release dynamically allocated memory for the process. This is usually done by the kernel by calling the `malloc()` function and by calling the `brk` system call.

◆ **unsigned long start_stack** -- The start_stack field value points to the beginning of the stack in the process's logical address space. See also the stack pointer location in Figure 5-12.

◆ **long pid** -- Pid is the process identification number. It is used to uniquely identify the process.

◆ **long pgrp** -- Pgrp refers to the process group number to which the process belongs.

◆ **long session** -- Session is the session number of the process, which is the process ID of the session.

◆ **long leader** -- The leader is the first process number of the session. For the concept of process groups and sessions, see the instructions following Chapter 7, Program Listing.

◆ **int groups[NGROUPS]** -- Groups is an array of group numbers for each group to which the process belongs. A process can belong to more than one group.

◆ **task_struct *p_pptr** -- p_pptr is a pointer to the parent process's task structure.

◆ **task_struct *p_cptr** -- p_cptr is a pointer to the most recent subprocess's task structure. That is the youngest child's task structure. Refer to figure 5-20.

◆ **task_struct *p_ysptr** -- p_ysptr is a pointer to an adjacent process created later than itself. That is pointer to the younger sibling process.

◆ **task_struct *p_osptr** -- *p_osptr is a pointer to an adjacent process created earlier than itself. That is point to the older sibling process. See Figure 5-20 for the relationship between the above four pointers. In the task data structure of the Linux 0.11 kernel, there is a parent process number field `father`, but it is not used in the

0.12 kernel. At this point we can use the process's `pptr->pid` to get the process number of the parent process.

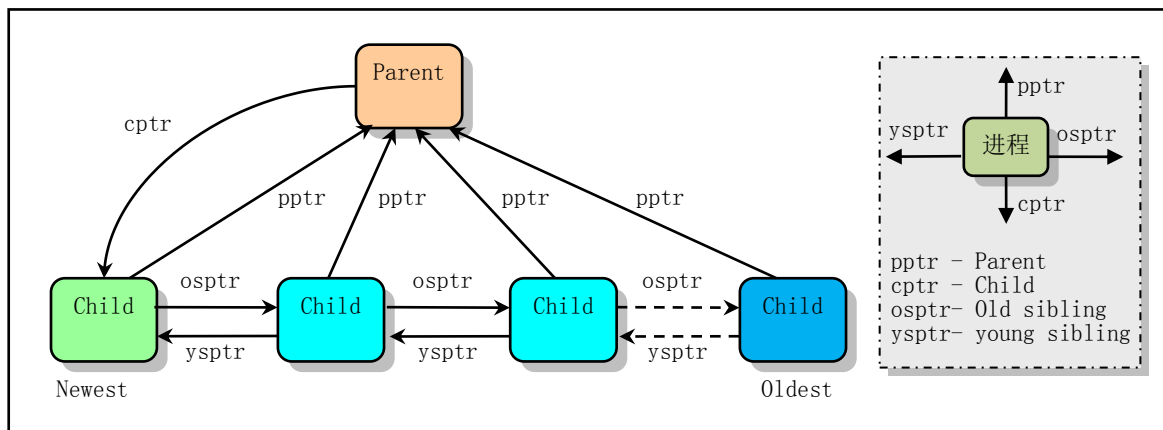


Figure 5-20 Relationship between process pointers

- ◆ unsigned short uid -- Uid is the user identification number (user id) that owns the process.
- ◆ unsigned short euid -- Euid is the effective user identification number that indicates the permissions to access the file.
- ◆ unsigned short suid -- Suid is the saved user identification number. When the set-user-ID flag of the execution file is set, the suid of the execution file is saved in the suid. Otherwise suid is equal to the euid of the process.
- ◆ unsigned short gid -- Gid is the group identification number (group id) to which the user belongs. It identifies the user group that owns the process.
- ◆ unsigned short egid -- Egid is an effective group identification number that indicates the permissions of the group of users to access the file.
- ◆ unsigned short sgid -- Sgid is the saved user group identification number. When the set-group-ID flag of the execution file is set, the gid of the execution file is stored in the sgid. Otherwise sgid is equal to the process's egid. For a description of these user id and group id, see the overview of the `sys.c` program in Chapter 5.
- ◆ long timeout -- Kernel timing timeout value.
- ◆ long alarm -- Alarm is the alarm timing value (number of ticks) for the process. If the process sets the field value using the system call `alarm()`, then when the system time ticking value exceeds the alarm field value, the kernel sends a `SIGALRM` signal to the process. By default this signal will terminate the execution of the program. Of course, we can also use the signal capture function (`signal ()` or `sigaction ()`) to capture the signal for the specified operation. The function `alarm()` starts at line 370 of `kernel/sched.c`. The kernel converts the function value of the function in seconds into a tick value, which is stored in the field after the current time tick value of the system.
- ◆ long utime -- Utime is the cumulative time (ticks) that the process runs in user state.
- ◆ long stime -- Stime is the cumulative time (ticks) that the process runs in the system state.
- ◆ long cutime -- Cutime is the cumulative time (ticks) that child processes runs in user state.
- ◆ long cstime -- Cstime is the cumulative time (ticks) that child processes runs in system state.
- ◆ long start_time -- Start_time is the time when the process is generated and starts running.
- ◆ struct rlimit rlim[RLIM_NLIMITS] -- The resource usage statistics array for the process.
- ◆ unsigned int flags -- Its the flag for each process, and 0.12 kernel is not yet in use.
- ◆ unsigned short used_math -- It is a flag indicating whether the process uses a coprocessor.
- ◆ int tty -- It is the subdevice number of the process using the tty terminal. -1 means no use.

◆**unsigned short umask** -- It is the 16-bit attribute mask word used by the process to create a new file (each bit represents a file), that is, the access attribute set by the new file. If a bit of the mask word is set, it means that the corresponding attribute is disabled (masked). This attribute mask word is used with the attribute value given when the file was created (mode & ~umask) as the actual access attribute of the newly created file. See the `include/fcntl.h` and `include/sys/stat.h` files for the specific meaning of the masked word and file attributes.

◆**struct m_inode * pwd** -- Pwd is a pointer to the i-node structure of the current working directory of the process. Each process has a current working directory that resolves relative path names and can be changed using the system call `chdir`.

◆**struct m_inode * root** -- Root is the process's own root i-node structure. Each process can have its own specified root directory for parsing absolute path names. Only the superuser can modify this root directory by calling `chroot`.

◆**struct m_inode * executable** -- Executable is the pointer to the i-node structure in memory for the execution file of the process. The system can use this field to determine if there is another process running the same executable file in the system. If so, then the in-memory i-node reference count value of `executable->i_count` will be greater than 1. When the process is created, the field is given the same value as the same field of the parent process, which means that the same program is being run with the parent process. When the kind of `exec()` function is called to execute a specified new executable file, the field value is replaced with the memory i-node pointer of the new program executed by the `exec()` function. When the process calls the `exit()` function and performs exit processing, the reference count of the memory i node pointed to by this field is decremented by 1, and the field will be blanked. The main role of this field is reflected in the `share_page()` function of the `memory.c` program. This function code can determine whether there are multiple copies of the currently running program in the system (at least 2) according to the reference count of the node pointed to by the execution of the process. If so, try a page sharing operation between them.

At system initialization, the execution of all tasks created by the system is 0 before the first call to execute the `execve()` function. These tasks include Task 0, Task 1, and all tasks created directly by Task 1 that have not yet executed `execve()`. That is, the executable field of all tasks directly included in the kernel code are 0. Because the code for task 0 is included in the kernel code, it is not loaded by the system from the file system. Therefore, the executable code has a fixed value of 0 in the kernel code. In addition, when creating a new process, `fork()` will copy the task data structure of the parent process, so task 1's executable is also 0. But after running `execve()`, the executable is given a pointer to the memory i-node of the file being executed. After that, this value of all tasks will not be 0.

◆**unsigned m_inode * library** -- Library is the i-node structure pointer of the library file that is loaded when the program is executed.

◆**unsigned long close_on_exec** -- It is a file descriptor (file handle) bitmap flag for a process. Each bit represents one file descriptor that is used to determine the file descriptor that needs to be closed when the system call `execve()` is called (see `include/fcntl.h`). When a program creates a child process using the `fork()`, it usually calls the `execve()` function in the child process to load another new program. At this point the child process will be completely replaced by the new program and the new program will start executing in the child process. If the corresponding bit of a file descriptor in `close_on_exec` is set, then the file descriptor corresponding to the open file will be closed when the child process executes the `execve()`. That is, the file descriptor is closed in the new program, otherwise the file descriptor will always be open.

◆**struct file * filp[NR_OPEN]** -- It is a table of file structure pointers for all open files used by the process, up to a maximum of 32 entries. The value of the file descriptor is the index value in the structure table. Each of

these is used for file descriptors to locate file pointers and access files.

◆ `struct desc_struct ldt[3]` -- It is the process local descriptor table structure LDT. It defines the code segment and data segment of the task in the virtual address space. Where array item 0 is a null item, item 1 is a code segment descriptor, and item 2 is a data segment (including data and stack) descriptor.

◆ `struct tss_struct tss` -- It is the task state segment TSS information structure of the process. The `tss_struct` structure holds all register values of the current processor when the task is switched out from execution. When the task is re-executed, the CPU uses these values to restore to the state when the task was switched out and starts execution.

When a process is executing, the values in all registers of the CPU, the state of the process, and the contents of the stack are called the context of the process. When the kernel needs to switch to another process, it needs to save all the state of the current process, that is, save the context of the current process, so that when the process is executed again, it can be restored to the state before the switch. In Linux, the current process context is stored in the task's task data structure. When an interrupt occurs, the kernel executes the interrupt service routine in the kernel state in the context of the interrupted process. At the same time, all the resources that need to be used are retained, so that the execution of the interrupted process can be resumed when the interrupt service ends.

5.7.2 Process Running States

A process can be in a different set of states during its lifetime, called the process state, as shown in Figure 5-21. Each circle in the figure with a different number represents a different state. As mentioned earlier, the process state is saved in the state field of the process task structure.

If a process is waiting for using CPU or is running, it is said to be in a ready or running state. At this point, the process state field value is `TASK_RUNNING`. If a process is asleep while waiting for system resources or event to occur, it is said to be in an interruptible sleep state, or an uninterruptible sleep state. At this time, the state field of the process may be `TASK_INTERRUPTIBLE` or `TASK_UNINTERRUPTIBLE`, respectively. If a process has been terminated, but it has not completely released the kernel resources, it is said to be in a dead state. At this point, the state field value of the process is `TASK_ZOMBIE`. If a process is temporarily stopped, it is said to be in a suspended state. At this point, the state field value of the process is `TASK_STOPPED`.

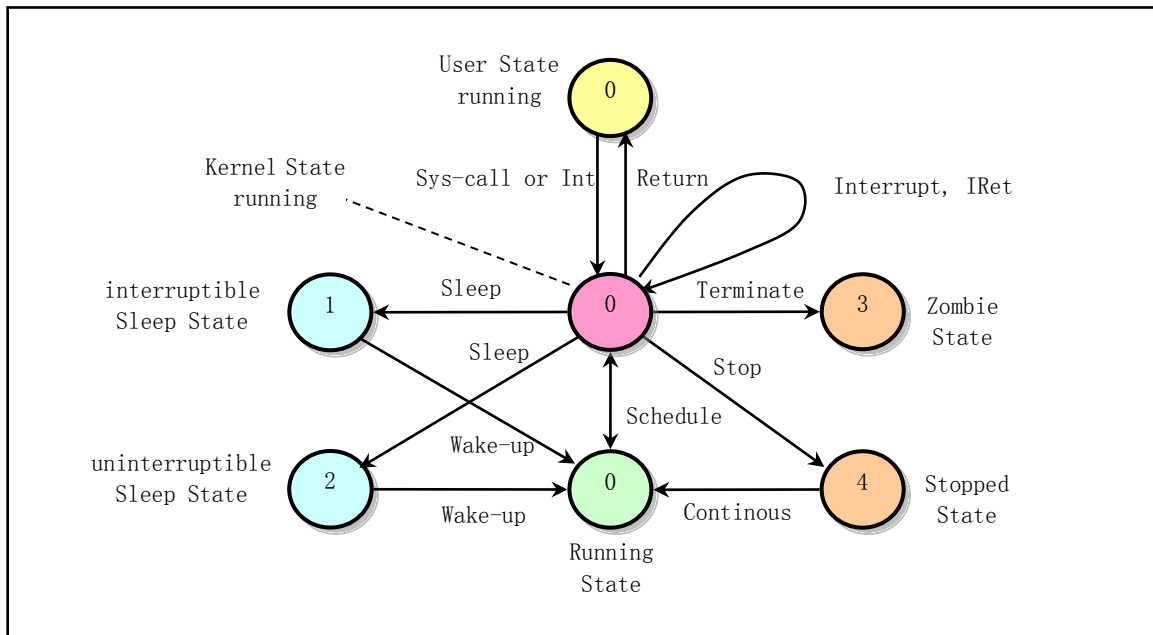


Figure 5-21 Process status and conversion relationship

The constant symbol names used for the five states of the Linux process are shown below. They are defined in line 46-50 in the file include/linux/sched.h.

```

// This defines the state of a process.
46 #define TASK_RUNNING      0 // is running or ready to run
47 #define TASK_INTERRUPTIBLE 1 // is in an interruptible wait state.
48 #define TASK_UNINTERRUPTIBLE 2 // uninterruptible wait state for wait I/O operation.
49 #define TASK_ZOMBIE       3 // is in a zombie state and has been terminated.
50 #define TASK_STOPPED      4 // The process has stopped.

```

◆ Running status (0, TASK_RUNNING)

When a process is being executed by the CPU, or is ready to be executed by the scheduler at any time, the process is said to be running state. If the process is not executed by the CPU at this time, it is said to be in the ready-to-run state, as shown in Figure 5-21. In the figure, the middle three circles from top to bottom contains the same value 0, which means that they are all ready or running states. Processes can run in kernel mode or in user mode. When a process is running in kernel code, we call it kernel running state, or simply kernel mode; when a process is executing its own code, we call it user running state (user mode). When system resources are available, the process wakes up and enters the ready-to-run state, which is called the ready state. These states (the middle column in the figure) represent the same method in the kernel and are said to be in the TASK_RUNNING state. When a new process has just been created, it is in this state (the bottom 0).

◆ Interruptible sleep state (1, TASK_INTERRUPTIBLE)

When a process is in an interruptible wait (sleep) state, the system does not schedule the process to execute. When the system generates an interrupt or releases the resource that the process is waiting for, or the process receives a signal, it can wake up the process to change to the ready state (that is, the running state).

◆ Uninterruptible sleep state (2, TASK_UNINTERRUPTIBLE)

This state is similar to the interruptible sleep state except that it is not woken up by the receipt of a signal.

However, a process in this state can only be converted to a runnable ready state when it is explicitly awake using the `wake_up()` function. This state is typically used when a process needs to wait undisturbed or when a waiting event occurs quickly.

◆Zombie state (3, `TASK_ZOMBIE`)

When a process has stopped running, but its parent has not called `wait()` to ask for its status, the process is said to be in a dead state. In order for the parent process to get the information that it stopped running, the task data structure information of the child process needs to be retained. Once the parent process calls `wait()` to get the information of the child process, the task data structure of the process in that state is released.

◆Stop status (4, `TASK_STOPPED`)

When the process receives the signal `SIGSTOP`, `SIGTSTP`, `SIGTTIN` or `SIGTTOU`, it will enter the stop state. The `SIGCONT` signal can be sent to the process to transition to a runnable state. Any signal received by the process during debugging will enter this state. In Linux 0.12, the conversion processing for this state has not been implemented. Processes in this state will be processed as process termination.

When a process runs out of time, the system uses the scheduler to force a switch to another process to execute. In addition, if the process needs to wait for a certain resource of the system when it executes in kernel mode, the process will call `sleep_on()` or `interruptible_sleep_on()` to voluntarily give up the usage rights of the CPU, and let the scheduler execute other processes. The process goes to sleep (`TASK_UNINTERRUPTIBLE` or `TASK_INTERRUPTIBLE`).

The kernel will only perform process switching operations when the process moves from "kernel running state" to "sleep state". Processes running in kernel mode cannot be preempted by other processes, and one process cannot change the state of another process. In order to avoid kernel data errors during process switching, the kernel will disable all interrupts when executing critical area code.

5.7.3 Process initialization

In the `boot/` directory, the bootloader loads the kernel from disk into memory and puts the system into protected mode, then starts the system initialization program `init/main.c`. The main program first determines how to allocate and use the system's physical memory, and then calls the initialization functions of the various parts of the kernel to initialize the memory management, interrupt handling, block devices, character devices, process management, and hard disk and floppy disk hardware. After these operations are completed, the various parts of the system are operational. Program control is then "manually" moved to task 0 (process 0), and process 1 is created for the first time using the `fork()` call. In process 1, the program will continue to initialize the application environment and execute the shell login program. The original process 0 will be scheduled to execute when the system is idle. At this point task 0 only executes the `pause()` system call, which in turn executes the scheduler function.

The process "Move to task 0 execution" is done by the macro `move_to_user_mode` (`include/asm/system.h`). It runs the `main.c` program execution flow from kernel mode (privilege level 0) to task 0 of user mode (privilege level 3). Before the move, the system first sets the running environment of task 0 in the initialization process (`sched_init()`) of the scheduler. This includes manually pre-setting the values of the fields of the task 0 data structure (`include/linux/sched.h`), adding the task state segment (TSS) descriptor of task 0 and the local descriptor table (LDT) in the global descriptor table. The segment descriptors are loaded into the task register `tr` and the local descriptor table register `ldtr`, respectively.

It should be emphasized here that kernel initialization is a special process, and the kernel initialization code is also the code of task 0. From the initial data set in the task 0 data structure, the base address of the code

segment and the data segment of task 0 is 0, and the segment length is 640 KB. The base address of the kernel code segment and the data segment is 0, and the segment length is 16 MB. Therefore, the code segment and data segment of task 0 are included in the kernel code segment and the data segment, respectively. The kernel initialization program `main.c` is the code for task 0, except that the system is running the main code with kernel mode privilege level 0 before moving to task 0. The function of the macro `move_to_user_mode` is used to move the running privilege level from level 0 of the kernel state to level 3 of the user mode, but still continue to execute the original code instruction stream.

During the move to task 0, the macro `move_to_user_mode` uses the interrupt return instruction to cause a privilege level change. The use of this method for control transfer is caused by the CPU protection mechanism. The CPU allows low-level (such as privilege level 3) code to be called or transferred to high-level code by calling a gate or interrupt, a trap gate, but not vice versa. So the kernel uses a method that simulates IRET to return low-level code. The main idea of this method is to construct the contents of the interrupt return instruction in the stack, and set the segment selector of the return address to the task 0 code segment selector with a privilege level of 3. Thereafter, executing the interrupt return instruction IRET will cause the system CPU to jump from privilege level 0 to the privilege level 3 of the outer layer. See Figure 5-22 for a diagram of the interrupt return stack structure when the privilege level changes.

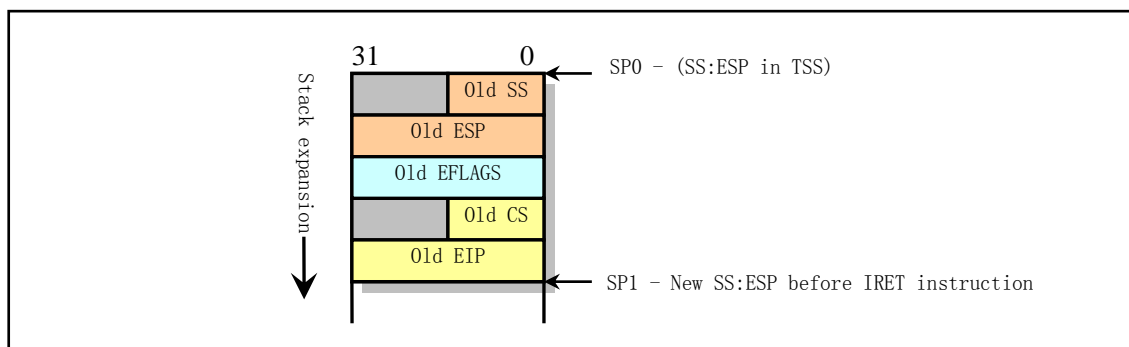


Figure 5-22 Interrupt return stack structure when privilege level changes

The macro `move_to_user_mode` first pushes the task 0 stack segment (ie, data segment) selector and kernel stack pointer into the kernel stack. Then push in the contents of the flag register. Finally, the task 0 code segment selector and the offset of the next instruction to be executed after the "interrupt return" is pushed in.

When the IRET instruction is executed, the CPU sends the return address to the CS:EIP and pops up the contents of the flag register in the stack. Since the CPU judges that the privilege level of the destination code segment is 3, it is different from the level 0 of the current kernel state. The CPU then pops the stack segment selector and stack pointer in the stack into SS:ESP. As the privilege level changes, the values of the segment registers DS, ES, FS, and GS become invalid, and the CPU clears these segment registers. Therefore, these segment registers need to be reloaded after the IRET instruction is executed. Thereafter, the system begins running on privilege level 3 on the code of task 0. At this point, the original stack used before the move is used as the subscriber station stack for task 0. The kernel state stack is specified by the contents of the TSS of task 0. It is specified beginning at the top of the page where the task data structure is located (`PAGE_SIZE + (long) & init_task`). Since the task data structure of task 0, including its user stack pointer, needs to be copied later when creating a new process, the user-mode stack of task 0 is required to remain "clean" until task 1 (process 1) is created.

5.7.4 Creating new processes

Creating a new process in a Linux system requires the use of the `fork()` system call. All processes created after initialization originates from process 0, which are child processes of process 0.

In the process of creating a new process, the system first finds an empty item (empty slot) in the task data structure array that has not been used by any process. If the system already has 64 processes running, the `fork()` system call will return an error because there are no empty items available in the task array table. Otherwise, the system will apply for a page of memory in the main memory area for the new process to store its task data structure information, and copy all the contents of the current process task data structure as a template for the new process task data structure. In order to prevent this new process that has not been processed from being executed by the scheduler, the new process state should be immediately set to the uninterruptible wait state (`TASK_UNINTERRUPTIBLE`).

The copied task data structure is then modified. Set the current process to be the parent of the new process, clear the signal bitmap and reset the statistics of the new process, and set the initial runtime slice value to 15 system ticks (150 milliseconds). Next, the values of the registers in the task status segment (TSS) are set according to the current process. Since the new process return value should be 0 when the process is created, you need to set `tss.eax = 0`. The new process kernel state stack pointer `tss.esp0` is set to the top of the memory page where the new process task data structure is located, and the stack segment `tss.ss0` is set to the kernel data segment selector. `Tss.lidt` is set to the index value of the local table descriptor in the GDT. If the current process uses a coprocessor, you also need to save the full state of the coprocessor to the `tss.i387` structure of the new process.

After that, the system sets the code and data segment base address and limit length of the new task, and copies the current process memory page table. Note that at this point the system does not allocate the actual physical memory page for the new process, but instead lets it share the memory page of its parent process. Only when any of the parent process or the new process has a write memory operation will the system allocate the relevant memory page for the write operation. This type of processing is called Copy On Write technology. For a detailed description of this technique, see the Write-Only Replication mechanism in the Memory Management chapter.

Later, if there are files in the parent process that are open, you should increase the number of open times of the corresponding file by one. The TSS and LDT descriptor entries for the new task are then set in the GDT, where the base address information points to `tss` and `ldt` in the new process task structure. Finally, set the new task to a runnable state and return a new process number.

Also note that creating a new child process and loading an executable file is two different concepts. When a child process is created, it completely copies the code and data area of the parent process and executes the code for the child process part there. When executing a program on a block device, it is generally run in the child process by running the `exec()` system call. After entering `exec()`, the original code and data area of the child process will be cleared (released). When the child process starts running a new program, since the kernel has not loaded the code of the program from the block device at this time, the CPU immediately generates an exception of page does not exist. At this point, the memory manager loads the corresponding code page from the block device, and the CPU re-executes the instruction that caused the exception. Until now, the code of the new program will actually start to be executed.

5.7.5 Process Scheduling

The scheduler in the kernel is used to select the next process to run in the system. This selective operating

mechanism is the basis of a multitasking operating system. The scheduler can be thought of as management code that allocates CPU runtime between all running processes. As can be seen from the foregoing description, the Linux process is preemptive, but the preempted process is still in the `TASK_RUNNING` state, but it is not temporarily run by the CPU. The preemption of the process occurs when the process is in the user state execution phase and cannot be preempted when executed in kernel state.

In order to enable the process to effectively use the system resources and allow the process to have a faster response time, it is necessary to adopt a certain scheduling strategy for the process switching scheduling. A scheduling strategy based on priority queuing is adopted in Linux 0.12.

The scheduler

The `schedule()` function first scans the task array. For each ready state (`TASK_RUNNING`) task, determine which process currently runs the least by comparing their run time count counter values. Which one has a large value means that the running time is not long, so the process is selected and the task switching macro function is used to switch to the process.

If all the time slices of the process in the `TASK_RUNNING` state have been used up at this time, the system will recalculate the time slice (counter) required for each process (including the sleeping process) in the system according to the priority value of each process. The calculation formula is:

$$counter = \frac{counter}{2} + priority$$

Thus, for a sleeping process, they have a higher time slice counter value when they are woken up. Then, the `schedule()` function rescans all processes in the `TASK_RUNNING` state in the task array and repeats the process until a process is selected. Finally, `switch_to()` is called to perform the actual process switch operation.

If no other processes can run at this time, the system will select process 0 to run. For Linux 0.12, process 0 calls `pause()` to put itself into an interruptible sleep state and call `schedule()` again. However, when scheduling a process, `schedule()` does not care what state of process 0 is. As long as the system is idle, the process 0 is scheduled to run.

Process switching

Whenever a new runnable process is selected, the `schedule()` function calls the `switch_to()` macro to perform the actual process switch operation. `Switch_to()` is defined in `include/asm/system.h`. This macro replaces the current process state (context) of the CPU with the state of the new process. Before switching, `switch_to()` first checks if the process to be switched to is the current process, and if so, does nothing and exits directly. Otherwise, the kernel global variable `current` is first set to the pointer of the new task, and then jumps to the address of the task state segment TSS of the new task, causing the CPU to perform the task switching operation. At this point, the CPU saves the state of all its registers to the `tss` structure of the current process task data structure pointed to by the TSS segment selector in the current task register `TR`. The register information in the `tss` structure in the new task data structure pointed to by the new task status segment selector is then restored to the CPU. After that, the system officially started the task of running the new switch. This processing can be seen in Figure 5-23.

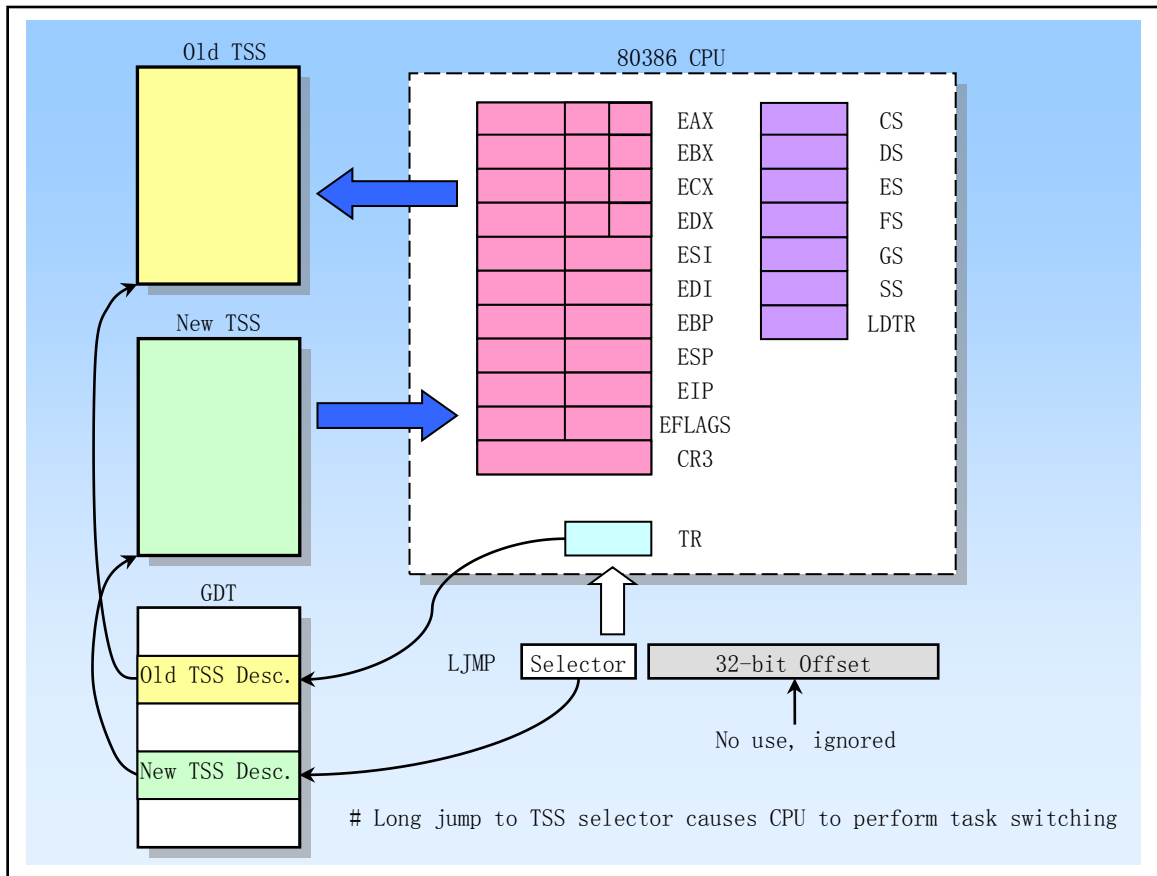


Figure 5-23 Task switching operation diagram

5.7.6 Terminating a process

When a process finishes running or terminates running halfway, the kernel needs to free up system resources occupied by the process. This includes files opened while the process is running, memory used, and so on.

When a user program calls the `exit()` system call, the kernel function `do_exit()` is executed. The function first releases the memory page occupied by the process code segment and the data segment, closes all files opened by the process, and synchronizes the current working directory, the root directory, and the i-node running the program used by the process. If the process has child processes, let the init process be the parent of all its child processes. If the process is a session header process and has a control terminal, the control terminal is released and the hang up signal `SIGHUP` is sent to all processes belonging to the session. This usually terminates all processes in the session. Then set the process state to `TASK_ZOMBIE`. And send a `SIGCHLD` signal to its original parent process to inform that some of its child processes have terminated. Finally `do_exit()` calls the scheduler to execute other processes. It can be seen that when the process is terminated, its task data structure remains, because its parent process also needs to use the information.

During the execution of a child process, the parent process usually waits for a child process to terminate using the `wait()` or `waitpid()` function. When the waiting child process is terminated and is in a zombie state, the parent process accumulates the time spent by the child process into its own process. Eventually, the memory page occupied by the child process task data structure is freed, and the pointer items occupied by the child process in the task array are blanked.

5.8 How to use the stack in Linux

This section provides an overview of how the Linux kernel uses the stack from boot-up to system uptime. The description of this part is closely related to the kernel code and can be skipped first. You can come back and study it when reading the relating code.

Four types of stacks are used in the Linux 0.12 system. One is the stack that is temporarily used during system boot initialization; the other is the stack that is used to initialize the kernel after entering protected mode, located at a fixed location in the kernel code address space. This stack is also the user-mode stack used by Task 0 later; the next is the stack used by each task to execute the kernel code through system calls, which we call the kernel-level stack of the task. Each task has its own independent kernel-mode stack; the last is the stack that the task executes in user mode, at the near end of the task (process) logical address space.

There are two main reasons for using multiple stacks or using different stacks in different situations. The first is that since the real mode enters the protection mode, the CPU has changed the memory access mode, so it is necessary to re-adjust the setup stack area. In addition, in order to solve the protection problem caused by the use of the stack by different privilege levels of the CPU, it is necessary to use a different stack for executing the level 0 kernel code and the level 3 user code. When a task enters kernel mode, it uses the stack pointer `tss.ss0`, `tss.esp0`, which is the privilege level 0 given in its TSS section, which is the kernel stack. The original user stack pointer will be saved in the kernel stack. When returning from user mode to kernel mode, the user-mode stack is restored. They are explained separately below.

5.8.1 Initialization phase

Boot initialization (bootsect.s, setup.s)

When the bootsect code is loaded by the ROM BIOS to physical memory 0x7c00, the stack segment is not set. Of course, the program does not use the stack. After the bootsect is moved to 0x9000:0, the stack segment register SS is set to 0x9000, and the stack pointer esp register is set to 0xff00. That is, the top of the stack is at 0x9000:0xff00, see line 61 of boot/bootsect.s. The stack section set in bootsect is also used in the setup.s program. This is the stack that is temporarily used during system initialization.

When entering protected mode (head.s)

From the head.s program, the system is officially running in protected mode. At this point the stack segment is set to the kernel data segment (0x10), the stack pointer esp is set to point to the top of the user_stack array (see head.s, line 31), and one page of memory (4K) is reserved for use as a stack. The user_stack array is defined in lines 67--72 of sched.c and contains a total of 1024 long words. Its location in physical memory can be seen in Figure 5-24 below. At this point this is the stack used by the kernel itself. The addresses given in the figure are approximate values, which are related to the actual setup parameters at compile time. These address locations can be found in the system.map file generated when the kernel is compiled.

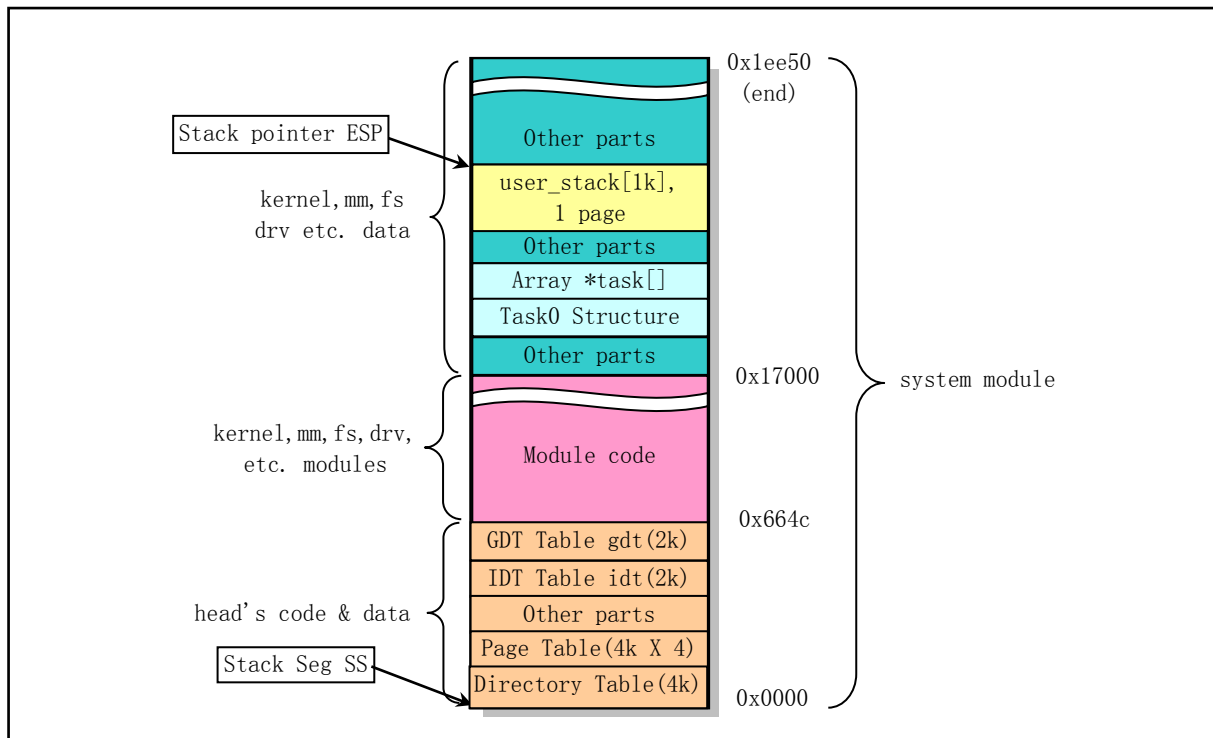


Figure 5-24 Stack diagram used by the kernel when entering protection mode

Initialization (main.c)

In the `init/main.c` program, the system always uses the above stack before executing the `move_to_user_mode()` code to transfer control to task 0. After the `move_to_user_mode()` is executed, the code of `main.c` is "switched" to task 0. By executing the `fork()` system call, `init()` in `main.c` will be executed in task 1 and use the stack of task 1. `Main()`, after being "switched" to task 0, continues to use the kernel's own stack as the user-mode stack for task 0. A detailed description of the stack used by task 0 is described later.

5.8.2 Stack of tasks

Each task has two stacks for the execution of user-mode and kernel-mode programs, and is called the user-mode stack and the kernel-state stack, respectively. In addition to being in different CPU privilege levels, the main difference between the two stacks is that the kernel mode stack of the task is small, and the amount of data saved cannot exceed $(4096 - \text{task data structure block})$ bytes, which is about 3K bytes. The user-mode stack of the task can extend within the user's 64MB space.

When running in user mode

Each task (except task 0 and 1) has its own 64MB address space. When a task (process) has just been created, its user state stack pointer is set nearly at the end of its address space (64MB top). In fact, the end part also includes the parameters of the execution program and environment variables, and then the user stack space, as shown in Figure 5-25. The application always uses this stack when it is running in user mode. The physical memory actually used by the stack is determined by the CPU paging mechanism. Since Linux implements copy-on-write, if the process and its parent process do not use the stack after the process is created, the two share the physical memory page corresponding to the same stack. The kernel memory manager allocates new memory pages for the write process only when one of the processes performs a stack write operation (such as a push operation). The user stacks for tasks 0 and 1 are special, as explained below.

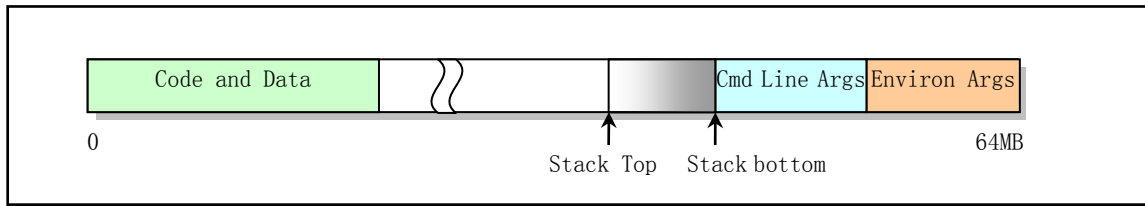


Figure 5-25 User state stack in logical space

When running in kernel mode

Each task has its own kernel-mode stack for the task to execute during execution in kernel code. The location in its linear address is specified by the `ss0` and `esp0` fields in the task TSS segment. `ss0` is the segment selector for the task kernel state stack, and `esp0` is the stack stack low pointer. Therefore, whenever a task is transferred from user code into kernel code, the kernel state stack of the task is always empty. The task kernel state stack is placed at the end of the page where its task data structure is located, that is, on the same page as the task's task data structure (`task_struct`). This is when the new task is created, the `fork()` program is set in the kernel-level stack fields (`tss.esp0` and `tss.ss0`) of the task `tss` segment, see `kernel/fork.c`, line 92:

```
p->tss.esp0 = PAGE_SIZE + (long)p;
p->tss.ss0 = 0x10;
```

Where `p` is the task data structure pointer for the new task and `tss` is the task state segment structure. The kernel requests memory for a new task to hold its `task_struct` structure data, and the `tss` structure (segment) is a field in the `task_struct`. The kernel stack segment value `tss.ss0` for this task is also set to `0x10` (ie, the kernel data segment selector), while `tss.esp0` points to the end of the page where the `task_struct` structure is saved, as shown in Figure 5-26. In fact, `tss.esp0` is set to point to the top byte of the page (outside) (at the bottom of the stack in the figure). This is because the Intel CPU performs the stack operation by first decrementing the stack pointer `esp` value and then saving the contents of the stack at the `esp` pointer.

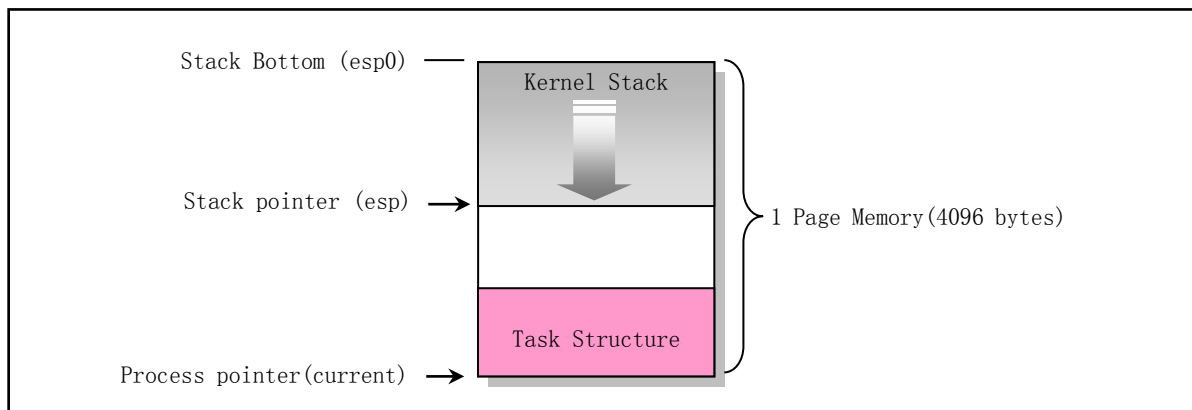


Figure 5-26 Kernel state stack diagram of a process

Why can a page of memory used to save the task data structure from the main memory area be set to be the data in the kernel data segment, that is, why can `tss.ss0` be set to `0x10`? This is because the user's kernel state stack still belongs to the kernel data space. We can illustrate this from the length of the kernel code segment. At the end of the `head.s` program, the kernel code and data segment descriptors are set respectively, and the segment length are all set to 16MB. This segment length is the maximum physical memory length that the

Linux 0.12 kernel can support (see `head.s`, notes beginning with line 110). Therefore, the kernel code can address anywhere in the entire physical memory range, including the main memory area. Whenever a task executes a kernel code and needs to use its kernel stack, the CPU uses the TSS structure to set its kernel state stack to consist of two values, `tss.ss0` and `tss.esp0`. When the task is switched, the kernel stack pointer `esp0` of the old task is not saved. For the CPU, these two values are read-only. So every time a task enters kernel mode execution, its kernel state stack is always empty.

Stacks of task 0 and task 1

The stack of task 0 (idle process) and task 1 (init process) is special and needs special explanation. The code segments and task segments of task 0 and 1 are the same, and the limit length is also 640 KB, but they are mapped to different linear address ranges. The segment base address of task 0 starts from linear address 0, while the segment base address of task 1 starts from 64 MB, but they all map to the physical address 0--640KB range. This address range is where the kernel code and basic data are stored. After the `move_to_user_mode()` is executed, the kernel state stacks of task 0 and task 1 are respectively located at the end of the page where the respective task data structures are located, and the user state stack of task 0 is the stack used before entering the protected mode, at the location of the `user_stack[]` array in `sched.c`. Since Task 1 replicated the user stack for Task 0 at the time of creation, Task 0 and Task 1 share the same user stack space at the beginning. However, when task 1 starts running, since the page table entry mapped by task 1 to `user_stack[]` is set to read-only, task 1 will cause a write page exception when performing a stack operation. Thus, the kernel will use the copy-on-write mechanism to allocate the main memory area page for task 1 as the stack space. Only then will Task 1 start using its own separate user stack memory page. Therefore, the stack of task 0 needs to be kept "clean" until task 1 actually starts to use, that is, task 0 cannot use the stack at this time to ensure that the copied stack page does not contain the data of task 0.

The kernel state stack for task 0 is specified in its manually set initialization task data structure. Its user-mode stack is set in the stack before the simulated `iret` return when executing `move_to_user_mode()`, as shown in Figure 5-22. We know that when a privilege level change occurs, the destination code uses the new privilege level stack, and the original privilege level code stack pointer is saved in the new stack. Therefore, the task 0 user stack pointer is first pushed into the stack currently at privilege level 0, and the code pointer is also pushed onto the stack. The execution of the `IRET` instruction then implements the transfer of control from the privilege level 0 code to the privilege level 3 code of task 0. In this manually set stack, the original `esp` value is set to still be the original position in `user_stack`, and the original `ss` segment selector is set to 0x17, that is set to the data segment selector in the user local table LDT. Then the task 0 code segment selector 0x0f is pushed onto the stack as the selector of the original CS segment in the stack, and the pointer of the next instruction is pushed onto the stack as the original EIP. In this way, by executing the `IRET` instruction, it can be "returned" to the code of task 0 to continue execution.

5.8.3 Switching between the kernel stack and user stack

In Linux 0.12, all interrupt service routines belong to the kernel code. If an interrupt is generated while the task is executing in user code, then the interrupt will cause a change in the CPU privilege level from level 3 to level 0, at which point the CPU will switch between the user state stack and the kernel state stack. The CPU will get the segment selector and offset value of the new stack from the TSS of the current task. Because the interrupt service routine is in the kernel and belongs to level 0 privilege level code, the 48-bit kernel state stack pointer is obtained from the `ss0` and `esp0` fields of the TSS. After locating the new stack (the kernel state stack), the CPU first pushes the original user state stack pointers `ss` and `esp` into the kernel state stack, and then pushes the contents of the flag register `eflags` and the return positions `cs` and `eip` into the kernel state stack.

The system call is a software interrupt, so when a task calls the system call, it enters the kernel and executes the interrupt service code in the kernel. At this point the kernel code will operate using the kernel-mode stack of the task. Similarly, when entering the kernel, as the privilege level changes (from user mode to kernel mode), the stack segment and stack pointers of the user-mode stack and eflags are stored in the kernel-state stack of the task. When the IRET instruction is executed to exit the kernel and return to the user program, the user state stack and eflags are restored. This process is shown in Figure 5-27.

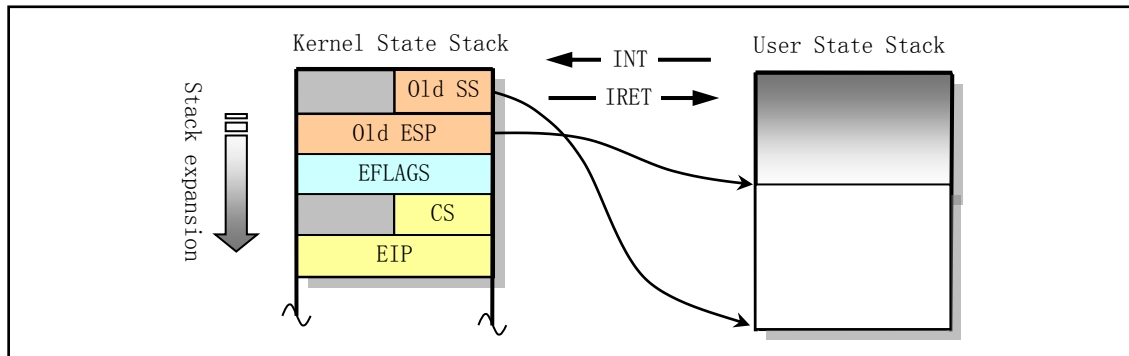


Figure 5-27 Kernel state and user state stack switching

If a task is running in kernel mode, then the stack switch operation is no longer needed if the CPU responds to the interrupt. Because the kernel code that the task is running at this time is already using the kernel state stack, and does not involve changes in priority levels. Therefore, the CPU only pushes the eflags and interrupt return pointers cs and eip into the current kernel state stack, and then executes the interrupt service process.

5.9 File System for Linux 0.12

File system support is required for the kernel to function properly. The root file system is used to provide the most basic information and support to the kernel. That is, when the Linux system boots and starts, the default file system is the root file system. This includes some of the operating system's minimum configuration files and command execution programs. For UNIX-like file systems used in Linux systems, it mainly includes some specified directories, configuration files, device drivers, executable programs, user applications, data or text files. These generally include the following subdirectories and files:

etc/	The directory mainly contains some system configuration files;
dev/	Contains device special files for file operation statement on devices;
bin/	Store system execution programs. Such as sh, mkfs, fdisk, etc.;
usr/	This directory stores library functions, manuals, and other files;
usr/bin	These directories store commands commonly used by users;
var/	This directory is used to store system runtime data or log information.

The device that holds the file system is the file system device. For example, for the commonly used Windows 10 operating system, the hard disk C is the file system device, and the files stored on the hard disk according to certain rules constitute the file system. Windows 10 has NTFS, FAT32 and other file systems of

different formats. Linux also has EXT2, EXT3 and other file systems of different formats to choose from. The file system supported by the Linux 0.12 kernel is the MINIX 1.0, which is created by Andrew Tanenbaum, the author of MINIX operation system. But the most widely used on current Linux systems is the ext3 or ext4 file system.

For the Linux 0.12 system running on a floppy disk described in Chapter 1, it consists of two simple floppy disks: the bootimage disk and the rootimage disk. Bootimage is the boot image file, which mainly includes the disk boot sector code, operating system loader and kernel binary code. Rootimage is the root file system used to provide the most basic support to the kernel. These two disks are equivalent to a bootable DOS operating system disk.

When the Linux boot disk loads the root file system, the root file system is loaded from the specified device according to the device number in a word (ROOT_DEV) at the 509th and 510th bytes of the boot sector on the disk. If the device number is 0, it means that the root file system needs to be loaded from the current drive where the boot disk is located. If the device number is a hard disk partition device number, the root file system is loaded from the specified hard disk partition.

Currently, the Filesystem Hierarchy Standard, which is maintained by the Linux Foundation (LF), is the specification file in this regard. Based on the original UNIX file system organization structure and content, the FHS specification has been studying the standardization of the structure and content of the Linux distribution system file system since 1994. FHS is now part of the official ISO LSB standard proposed by the Linux Standard Library (LSB).

5.10 Directories of kernel source code

Since the Linux kernel is a single-core mode system, almost all of the code in the kernel is closely related. The data dependencies and invocation relationships between them are very close. So when reading a source code file, you often need to refer to other files. Therefore, it is necessary to familiarize yourself with the directory structure and arrangement of the source code files before starting to read the kernel source code.

Here we first list the complete source directory of the kernel, including its subdirectories. Then introduce the main functions of the programs included in each directory one by one, so that the entire kernel source code arrangement can establish a general framework in our minds, so as to facilitate the source code reading work in the next chapter.

When we use the tar command to unpack the linux-0.12.tar.gz file, the kernel source files are placed in the linux/ directory. The directory structure is shown in Figure 5-28:

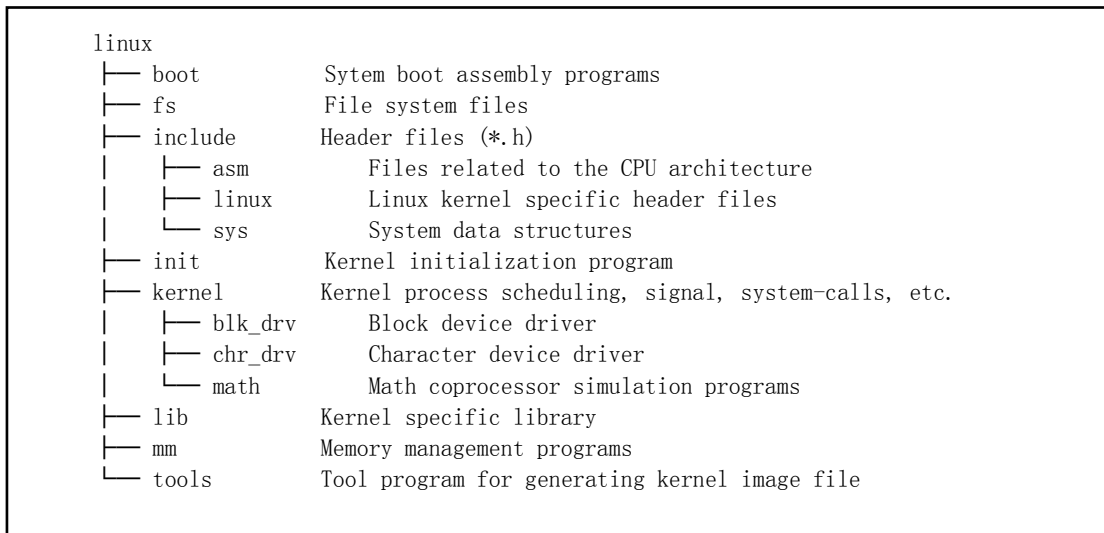


Figure 5-28 Linux kernel source code directory structure

The source code directory for this version of the kernel contains 14 subdirectories, including a total of 102 code files. The contents of these subdirectories are described one by one below.

5.10.1 Kernel home directory linux

The linux directory is the main directory of the source code. In addition to including all 14 subdirectories, the home directory contains a unique Makefile. This file is the parameter configuration file for the compile aid software make. The main purpose of the make tool is to automatically determine which files need to be recompiled in a system with multiple source files by identifying which files have been modified. Therefore, the make tool is a management software for a program project.

The Makefile in the linux directory also nests the Makefiles contained in all subdirectories. This way, make will recompile any files in the linux directory (including subdirectories) that have been modified. So in order to compile all the source code files of the entire kernel, just run the make software once in the linux directory.

5.10.2 The boot program directory

The boot directory contains three assembly language files, which are the first programs compiled in the kernel source code file. The main function of these three programs is to boot the kernel when the computer is powered up, load the kernel code into memory, and do some system initialization before entering the 32-bit protection mode.

- The bootsect.s program is a disk boot block program that resides in the first sector of the disk after compilation (boot sector, 0 track (cylinder), 0 head, 1st sector). After the PC is powered up and the ROM BIOS self-test, it will be loaded into the memory 0x7C00 for execution.
- The setup.s program is primarily used to read the machine's hardware configuration parameters and move the kernel module system to the appropriate memory location.
- The head.s program will be compiled and connected to the foremost part of the kernel system module, mainly for the hardware device's probe settings and the initial setup of the memory management page.

The bootsect.s and setup.s programs need to be compiled using the as86 software, using the as86 assembly language format (similar to Microsoft's). Head.s needs to be compiled with GNU as, using the assembly language of AT&T format. These two assembly languages are briefly described in the code comments in the next chapter and in the descriptions that follow the code listing.

5.10.3 File System Directory fs

The file system of the Linux 0.12 kernel uses the version 1.0 MINIX file system, which is because Linux was developed on the MINIX system. The MINIX file system is easy to cross-compile, and Linux partitions can be loaded from MINIX. Although the MINIX file system is used, Linux handles it differently than the MINIX system. The main difference is that MINIX uses a single-threaded approach to the file system, while Linux uses a multi-threaded approach. Due to the multi-threaded processing method, Linux programs must deal with the race conditions and deadlocks caused by multi-threading. Therefore, the Linux file system code is much more complicated than the MINIX system. In order to avoid the occurrence of competitive conditions, the Linux system has strictly checked the resource allocation. And when running in kernel mode, if the task does not actively sleep (call sleep()), the kernel will not be allowed to switch tasks.

The fs/ directory is a directory of file system implementations and contains a total of 18 C programs. The main references and dependencies between these programs are shown in Figure 5-29. Each box in the figure represents a file, placed from top to bottom in a basic reference relationship. The file name is omitted from the suffix .c. The program files in the virtual box are not part of the file system. Lines with arrows indicate reference relationships, and thick lines indicate mutual reference relationships.

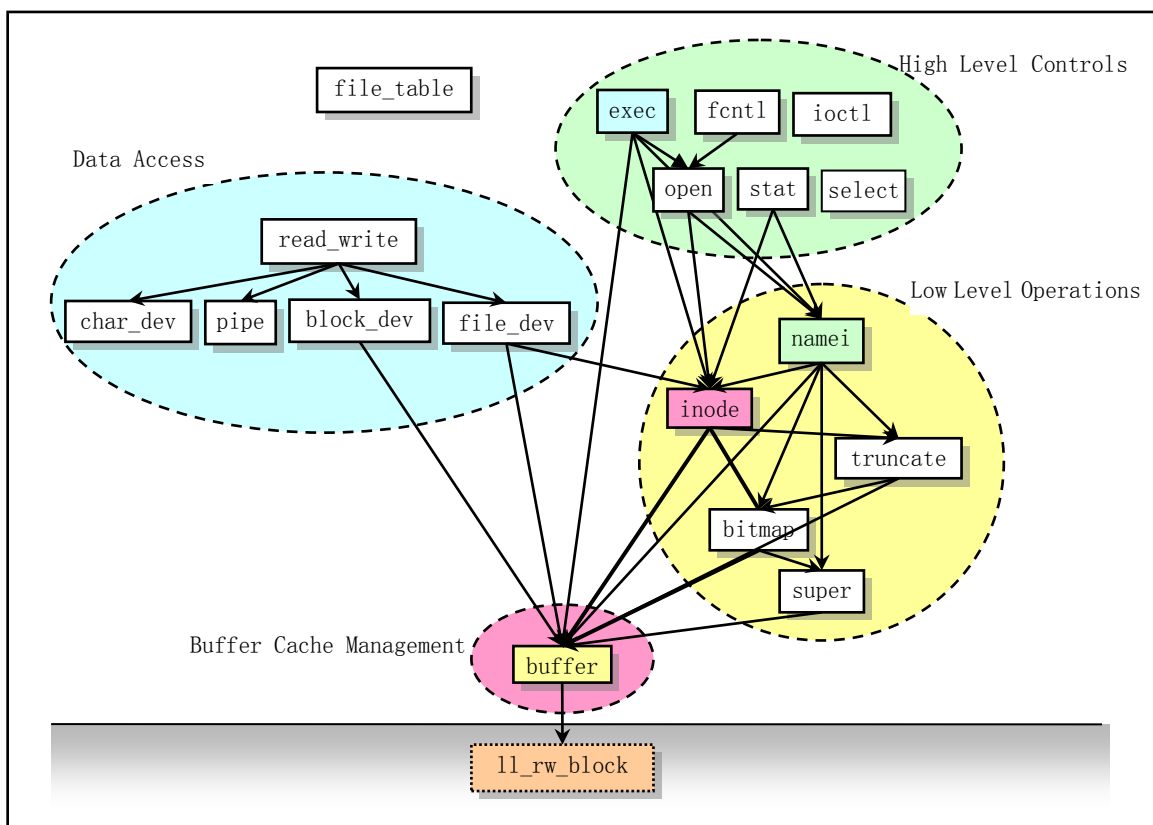


Figure 5-29 The relationship of each program in the fs directory.

As can be seen from the figure, the program in this directory can be divided into four parts: cache management, low-level file operations, file data access, and file high-level functions. When annotating the files in this directory, we will also describe them in four parts.

For the file system, we can think of it as an extension of the memory cache. All access to data in the file system needs to be first read into the cache. The programs in this directory are mainly used to manage the

allocation of buffer blocks in the cache and the file system on the block device. The program that manages the cache is `buffer.c`, while other programs are mostly used for file system management.

- The `file_table.c` file currently has only one file handle (descriptor) structure array defined.
- The `ioctl.c` file will reference the functions in `kernel/chr_drv/tty.c` to implement the io control function of the character device.
- The `exec.c` file mainly contains an executive function `do_execve()`, which is the main function in all `exec()` function clusters.
- The `fcntl.c` program is used to implement the system call function for file i/o control.
- The `read_write.c` program is used to implement file read/write and locate three system call functions.
- The `stat.c` program implements two system calls that get file state information.
- The `open.c` program mainly contains system call functions that implement modification of file attributes and creation and closing of files.
- The `char_dev.c` mainly contains the character device read and write function `rw_char()`.
- The `pipe.c` program contains pipe read and write functions and system calls to create pipes.
- The `file_dev.c` program contains file read and write functions based on the i-node and descriptor structure.
- The `namei.c` program mainly includes operation functions and system calls for directory names and file names in the file system.
- The `block_dev.c` contains block data read and write functions.
- The `inode.c` program contains functions for file system i-node operations.
- The `truncate.c` program is used to release the device data space occupied by files when deleting files.
- The `bitmap.c` file is used to process bitmaps of i-nodes and logical blocks in the file system.
- The `super.c` program contains handlers for file system superblocks.
- The `buffer.c` program is mainly used to process the memory cache.
- The `select.c` program is mainly used to effectively handle the problem of simultaneous I/O operations on multiple files.

The `ll_rw_block` in the virtual box is the underlying read function of the block device. It is not in the `fs` directory, but the block device read and write driver function in `kernel/blk_drv/ll_rw_block.c`. Putting it here just makes us clearly see that the file system needs to read and write the data in the block device through the high-speed buffer cache and the block device driver (`ll_rw_block()`). The file system's program itself does not directly interact with the driver of the block device.

In the process of annotating the program, we will additionally give the call hierarchy between the main functions in these files.

5.10.4 Header file directory include

There are a total of 36 `.h` header files in the header file directory. There are 13 in the main directory, 4 in the `asm` subdirectory, 11 in the `Linux` subdirectory, and 8 in the `sys` subdirectory. The respective functions of these header files are briefly described below. See the Notes on Header for specific actions and information.

- `<a.out.h>` The `a.out` header file defines the `a.out` executable file format and some macros.
- `<const.h>` The constant symbol file currently defines only the flags of `i_mode` field in the i-node.
- `<ctype.h>` The character type header file. Defines some macros for character type conversion.
- `<errno.h>` Error number header file. Contains various error numbers in the system. (Linus was introduced from `minix`).
- `<fcntl.h>` File control header file. The definition of the operation control constant symbol used for the

file and its descriptors.

- `<signal.h>` Signal header file. Define signal symbol constants, signal structures, and signal manipulation function prototypes.
- `<stdarg.h>` Standard parameter header file. Define a list of variable parameters in the form of macros. It mainly describes one type (`va_list`) and three macros (`va_start`, `va_arg` and `va_end`) for the `vsprintf`, `vprintf`, and `vfprintf` functions.
- `<stddef.h>` The standard definition header file. `NULL`, `offsetof(TYPE, MEMBER)` is defined.
- `<string.h>` String header file. Mainly defines some embedded functions about string operations.
- `<termios.h>` Terminal input and output function header file. It mainly defines the terminal interface that controls the asynchronous communication port.
- `<time.h>` Time type header file. The most important of these is the definition of the `tm` structure and some function prototypes related to time.
- `<unistd.h>` Linux standard header file. Various symbol constants and types are defined and various functions are declared. If `__LIBRARY__` is defined, it also includes the system call number and the inline assembly `_syscall0()`.
- `<utime.h>` User time header file. The access and modification time structures and the `utime()` prototype are defined.

include/asm -- Architecture related header file subdirectory

These header files primarily define data structures, macro functions, and variables that are closely related to the CPU architecture.

- `<asm/io.h>` Io header file. Defines the function that operates on the io port in the form of a macro's embedded assembler.
- `<asm/memory.h>` Memory copy header file. Contains `memcpy()` embedded assembly macro functions.
- `<asm/segment.h>` Segment operation header file. An embedded assembly function is defined for segment register operations.
- `<asm/system.h>` System header file. An embedded assembly macro that defines or modifies descriptors/interrupt gates, etc. is defined.

include/linux -- Linux kernel dedicated header file subdirectory

- `<linux/config.h>` Kernel configuration header file. Define keyboard language and hard disk type (`HD_TYPE`) options.
- `<linux/fdreg.h>` Floppy disk file. Contains some definitions of floppy disk controller parameters.
- `<linux/fs.h>` File system header file. Define the file table structure (`file`, `buffer_head`, `m_inode`, etc.).
- `<linux/hdreg.h>` Hard disk parameter header file. Define access to the hard disk register port, status code, partition table and other information.
- `<linux/head.h>` Head header file. A simple structure for the segment descriptor is defined, along with several selector constants.
- `<linux/kernel.h>` Kernel header file. Contains prototype definitions of some of the commonly used functions of the kernel.
- `<linux/math_emu.h>` Coprocessor emulation header file. A coprocessor data structure and a floating point representation structure are defined.
- `<linux/mm.h>` Memory management header file. Contains page size definitions and some page release function prototypes.
- `<linux/sched.h>` The scheduler header file defines the task structure `task_struct`, the data of the initial

task 0, and some embedded assembly function macro statements about the descriptor parameter settings and acquisition.

- <linux/sys.h> The system calls the header file. Contains 72 system call C function handlers, starting with 'sys_'.
- <linux/tty.h> The tty header file defines parameters and constants for tty_io, serial communication.

include/sys -- System-specific data structure subdirectory

- <sys/param.h> Parameter file. Some hardware-related parameter values are given.
- <sys/resource.h> Resource file. Contains information on the limits and utilization of system resources used by processes.
- <sys/stat.h> File status header file. Contains file or file system state structures stat{ } and constants.
- <sys/time.h> The timeval structure and the itimerval structure are defined.
- <sys/times.h> Defines the running time structure tms and the times() function prototype in the process.
- <sys/types.h> Type header file. The basic system data types are defined.
- <sys/utsname.h> System name structure header file.
- <sys/wait.h> Wait header file. Define the system call wait() core waitpid() and related constant symbols.

5.10.5 init -- kernel initialization program directory

This directory contains only one file, main.c, for performing all initialization work on the kernel. Then control transfers to user mode to create a new process and run the shell program on the console device.

The program first allocates the buffer memory capacity based on the amount of physical memory in the machine. If the system also sets up a virtual disk to use, it also leaves space behind the buffer memory. All hardware initialization is then performed, including manually creating the first task (task 0) and setting the interrupt enable flag. After the control moves from the kernel state to the user state, the system first calls the create process function fork() to create a process for running init(). In this child process, the system will set up the console environment and generate a child process to run the shell program.

5.10.6 kernel -- kernel program main directory

The linux/kernel directory contains a total of 32 files and 3 subdirectories (blk_drv, chr_drv, and math). There are 12 code files and a Makefile in the kernel directory, 5 files in the subdirectory kernel/blk_drv, 6 files in kernel/chr_drv, and 9 files in kernel/math.

All programs that handle tasks are stored in the kernel/ directory, which including forks, exits, schedulers, and some system callers. It also includes the procedure services of handling interrupt exceptions and traps. Subdirectories include low-level device drivers such as get_hd_block and tty_write. Because of the complex calling relationships between the code in these files, the reference relationship diagram between the files is not detailed here. However, it is still possible to perform an approximate classification, as shown in Figure 5-30.

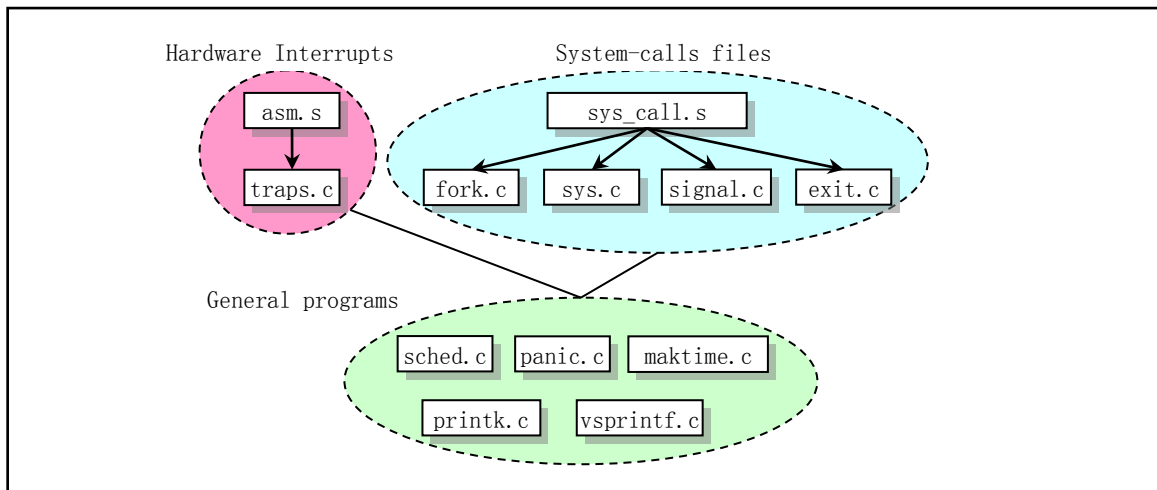


Figure 5-30 Call hierarchy of each file

- The `asm.s` program is used to handle interrupts caused by system hardware exceptions.
- The `traps.c` file is used for the actual handling of hardware exceptions. In each interrupt processing, the corresponding C language processing function in `traps.c` will be called separately.
- The `exit.c` program mainly includes system calls for processing process termination, including process release, session (process group) termination, and program exit processing functions, as well as system call functions such as killing processes, terminating processes, and suspending processes.
- The `fork.c` program gives two C language functions used in the `sys_fork()` system call: `find_empty_process()` and `copy_process()`.
- The `mktime.c` program contains a time function `mktime()` used by the kernel to calculate the number of seconds from 0:00 on January 1, 1970 to the day of booting. It is only called once in `init/main.c`.
- The `panic.c` file contains a function `panic()` that displays kernel error messages and stops.
- The `printk.c` program contains a kernel-specific information display function `printk()`.
- The `sched.c` program includes basic functions for scheduling (`sleep_on`, `wakeup`, `schedule`, etc.), as well as some simple system call functions. There are also several floppy disk operation functions related to timing.
- The `signal.c` program includes four system calls for signal processing and a function `do_signal()` that processes the signal in the corresponding interrupt handler.
- The `sys.c` program includes many system call functions, some of which are not yet implemented.
- The `system_call.s` program implements the interface processing of the Linux system call (`int 0x80`). The actual processing is contained in the corresponding C language functions of each system call. These processing functions are distributed throughout the Linux kernel code.
- The `vsprintf.c` program implements a string formatting function that is now included in the standard library functions.

kernel/blk_drv -- Block device driver subdirectory

Typically, a user accesses a device through a file system, and a device driver implements an access interface for the file system. When a block device is used, a cache mechanism is used between the user process and the block device in order to efficiently use the data on the block device due to its large data throughput. When accessing data on a block device, the system first reads the data on the block device into the cache in the form of a data block, and then copies the data into user's buffer space.

The `blk_drv` subdirectory contains a total of 4 c files and 1 header file. The header file `blk.h` is dedicated to

the block device program and is therefore placed with the C file. The approximate relationship between these documents is shown in Figure 5-31.

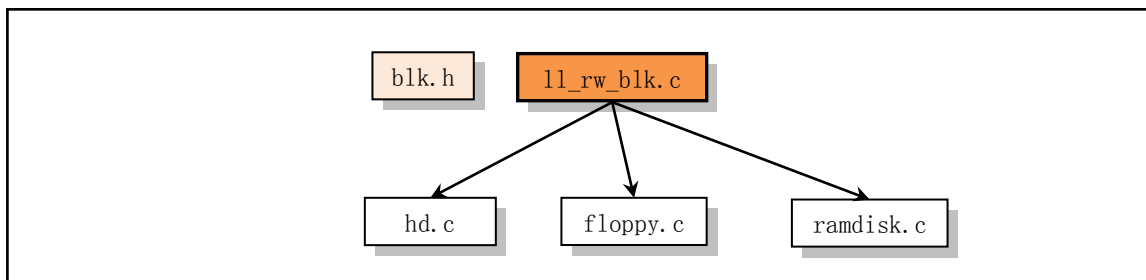


Figure 5-31 Hierarchical relationship of files in the blk_drv directory.

- blk.h block device special header file. The block device structure and data block request structure shared by several C programs are defined in it.
- The hd.c program mainly implements the underlying driver function for reading/writing hard disk data blocks, mainly the do_hd_request() function.
- The floppy.c program mainly implements the read/write drive function for floppy data blocks, mainly the do_fd_request() function.
- The ramdisk.c program is a memory virtual disk driver. The main function is do_rd_request(). A virtual disk device is a technology that uses physical memory to simulate an actual disk storage medium. Its main advantage is that it can greatly improve the speed of data access operations.
- The program in ll_rw_blk.c implements the low-level block device data read/write function ll_rw_block(). All other programs in the kernel use this function to access data from block devices. You will see that the function is called in many places where the block device data is accessed, especially in the cache file fs/buffer.c.

kernel/chr_drv -- Character device driver subdirectory

The character device subdirectory contains a total of 4 C language programs and 2 assembler files. These files enable drivers for serial port rs-232, serial terminals, keyboards, and console terminals. Figure 5-32 is the approximate call hierarchy between these files.

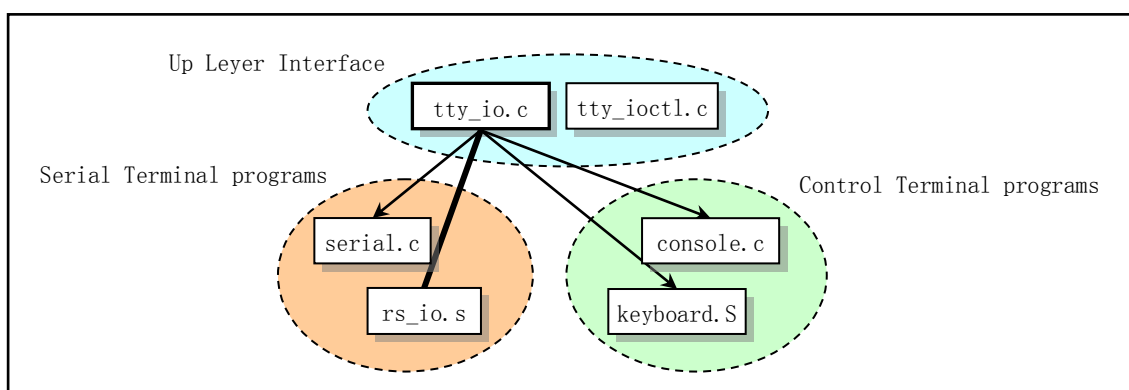


Figure 5-32 Schematic diagram of the relationship between character device programs

- The tty_io.c program contains the tty character device read function tty_read() and the write function tty_write(), which provides the upper layer access interface for the file system. It also includes the C

function `do_tty_interrupt()` called during serial interrupt processing. This function will be called when the interrupt type is a read character.

- The `console.c` file mainly contains the console initialization program and the console write function `con_write()`, which is used by the tty device. It also includes an initialization setup program `con_init()` for screen display and keyboard interrupts.
- The `rs_io.s` program is used to implement interrupt handlers for two serial interfaces. The interrupt handler processes each of the four interrupt types retrieved from the interrupt identifier register (port 0x3fa or 0x2fa) and calls `do_tty_interrupt()` in the code that handles the interrupt type as a read character.
- `Serial.c` is used to initialize the asynchronous serial communication chip UART and set the interrupt vector of the two communication ports. Also included is the `rs_write()` function that tty uses to output to the serial port.
- The `tty_ioctl.c` program implements tty's io control interface function `tty_ioctl()`, and reads and writes to the `termio(s)` terminal io structure, and is called in the `fs/ioctl.c` program that implements the system call `sys_ioctl()`.
- The `keyboard.S` program mainly implements the keyboard interrupt handler procedure `keyboard_interrupt`.

kernel/math -- Coprocessor emulator subdirectory

This subdirectory contains the math coprocessor emulation handler file, which has a total of 9 C files. When there is no math coprocessor in the machine, and the CPU executes the coprocessor instruction, it will cause the device not exist interrupt INT 7. Therefore, using this interrupt, software can be used to simulate the function of the coprocessor. These programs are closely related to the CPU hardware, but have little to do with the rest of the kernel implementation. The relevant knowledge in the program is very helpful for implementing system-level programs such as multiprocessor programming and assembly and disassembly.

- The `math_emulate.c` program is the main program of coprocessor emulation, which implements the device not exist exception handler, floating point instruction emulation main function `do_emu()` and other auxiliary functions.
- The `error.c` program is used to process the error signal sent by the coprocessor. Its main function is `math_error()`.
- The `ea.c` program is used to calculate the effective address used by the operands in the instruction when simulating floating-point instructions.
- The `convert.c` program is used to implement the data type conversion operation between the user data format and the temporary real number format in the simulation calculation process.
- The `add.c` program implements the addition in the simulation process and performs the conversion between the mantissa symbolization and the non-symbolization.
- The `compare.c` program is used to simulate the size of two temporary real numbers in the coprocessor compare accumulator.
- The `get_put.c` program implements access to data in the user's memory.
- The `mul.c` program is used to simulate multiply instructions in the coprocessor.
- The `div.c` program is used to simulate the division of the coprocessor.

5.10.7 lib -- Kernel library directory

Unlike normal user programs, kernel code cannot use standard C libraries and other library code. The main

reason is that the complete C function library is large and complex to implement. Therefore, there is a special `lib/` directory in the kernel source that provides some functions that the kernel needs to use. The kernel function library is used to provide call support for the kernel initialization program `init/main.c` running in user mode (process 0, 1). It is exactly the same as the implementation of a normal static library. Readers can learn the basic structure of the general `libc` library.

There are 12 C language files in the `lib/` directory. Except for a `malloc.c` programmed by Mr. tytso, other files are very short, and some have only one or two lines of code. They implement interface functions for some system calls. These files mainly include the exit function `_exit()`, the close file function `close(fd)`, the copy file descriptor function `dup()`, the file open function `open()`, the write file function `write()`, the execute program function `execve()`. The memory allocation function `malloc()`, wait for the child process state function `wait()`, create the session system call `setsid()`, and all string manipulation functions implemented in `include/string.h`.

5.10.8 mm -- Memory Management Directory

This directory contains 3 code files. It is mainly used to manage the application's use of the main memory area, and implements the mapping operation from the logical to linear address and linear to physical memory address. And through the memory paging mechanism, a correspondence is established between the virtual memory page and physical memory page of the main memory area. At the same time, virtual storage technology has also been realized.

The Linux kernel handles memory in both fragmented and paged ways. The first is to divide the 80X86 4G virtual address space into 64 segments (64MB per segment). The kernel program occupies the first segment and the physical address is the same as the segment linear address. Then each task is assigned a segment to use. The paging mechanism is used to map the specified physical memory page into the segment, detect any duplicate copies created by the fork, and perform a copy-on-write mechanism.

The `page.s` file includes the memory page abort (int 14) handler, which is mainly used to handle page exceptions caused by page faults and page protection caused by accessing illegal addresses.

The `memory.c` program includes the function `mem_init()` that initializes the memory, and the `do_no_page()` and `do_wp_page()` functions called by the memory interrupt procedure in `page.s`. When you create a new process and perform a copy process operation, you use a memory handler to allocate the management memory space.

The `swap.c` program is used to manage page swapping between physical pages in main memory and high-speed secondary storage (hard disk) space. When the main memory space is not enough, you can save the temporarily unused memory page to the hard disk. When a page fault exception occurs, first check whether the requested page is in the hard disk swap space. If it exists, the page is directly read into the memory from the swap space.

5.10.9 tools -- Kernel Tools Directory

The `build.c` program in this directory is used to build a complete kernel module. It combines the objects compiled in the Linux directory into a runnable kernel image file image. The specific functions can be found in the next chapter.

5.11 The Kernel Code and User Programs

In Linux systems, the kernel can provide service support for user programs in two ways. One is through the system call interface, that is, the interrupt calls int 0x80; on the other hand, through the development

environment library functions or kernel library functions. However, the kernel library functions are only used by task 0 and task 1 created by the kernel. They eventually still call the system call. Therefore, the kernel actually only provides a unified service interface of system calls to all user programs or processes. The implementation method of the kernel library function code in the `lib/` directory is basically the same as in the C function library `libc`. In order to use kernel resources, the kernel system call is eventually invoked through inline assembly code, as shown in Figure 5-4.

System calls are primarily used for system software programming or for implementation of library functions. The program developed by the average user accesses kernel resources by calling functions in libraries such as `libc`. The functions or resources in these libraries are often referred to as application programming interfaces (APIs). It defines a set of standard programming interfaces used by the application. By calling functions in these libraries, the application code can perform a variety of common tasks, such as opening and closing access to files or devices, performing scientific calculations, error handling, and accessing system information such as group and user ID numbers.

In the UNIX-like operating system, the most commonly used API interface based on the POSIX standard, and Linux is no exception. The difference between API and system call is that in order to implement an application interface standard, such as POSIX, the API may correspond to a system call, or may be implemented by several system call functions. Of course, some API functions may not need to use system calls at all, that is, do not use the services provided by the kernel. Therefore, the function library can be regarded as a main interface that implements the POSIX standard. The application does not care what is the relationship between it and system call. No matter how much difference between the system calls provided by the two operating systems, as long as it provides compliance with the same API standard, the application can be portable between these operating systems.

The system call is the highest level of the interface between the kernel and the outside world. In the kernel, each system call is often implemented as a macro and has a serial number (defined in the `include/unistd.h` header file). Applications should not use system calls directly, otherwise the portability of the program will be worse. Therefore, the current Linux standard base (LSB) and many other standards do not recommend that applications directly access system call macros. Documentation for system calls can be found in Part 2 of the online manual for the Linux operating system.

Library files typically include user-level functions such as input/output and string manipulation functions that are not provided by the C language to perform advanced functions. Some library functions are just enhancements to system calls. For example, the standard I/O library functions `fopen` and `fclose` provide similar functionality to the system calls `open` and `close`, but at a higher level. In this case, system calls usually provide slightly better performance than library functions, but library functions provide more functionality and are more error-detecting. The library functions provided by the system can be found in the online manual section 3 of the operating system.

5.12 linux/Makefile

From this section, we begin to annotate the kernel source code files. First let's comment on the first file `Makefile` encountered in the `Linux` directory. Subsequent sections will be annotated in a similar description structure.

5.12.1 Function Description

The `Makefile` is equivalent to a batch file during program compilation. It is the default compilation settings

input file for the utility program make at runtime. Just type the make command in the directory containing the Makefile, it will call the compiler and linker to compile, link, or install the source code or the target code file according to the settings in the Makefile.

The make utility automatically determines which files in a program package containing multiple source files need to be recompiled and issues commands to compile those program files. So before using make, we need to write a Makefile text file. This file describes the relationship between the programs in the entire package and gives specific control commands for each file that needs to be updated. Typically, executable target are updated based on their object files, which are created by the compiler. Once you have written a suitable Makefile, you can perform all necessary recompilation tasks after each modification of some source code files in the program package. The make tool uses the Makefile file and the last-modification time of code files to determine which files need to be updated. For each file that needs to be updated, it will issue the appropriate command based on the information in the Makefile. In the Makefile, lines starting with '#' are comment lines. The '=' assignment statement at the beginning of the file defines abbreviations for some parameters or commands.

The main function of this Makefile in the kernel directory is to instruct the make program to eventually connect and merge all kernel compiled code into a runnable kernel image file using the build executable in the tools/directories that are independently compiled. The specific process is: (1) use the 8086 assembler to compile the bootsect.s and setup.s in boot/, and generate their respective objects. (2) Compile all other programs in the source code using GNU compiler gcc/gas, and link to generate module system. (3) Finally, use the build tool to combine the three parts into a kernel image file image.

The build tool is a stand-alone executable compiled from the tools/build.c source file. It is not compiled and linked into the kernel code. It is only used as a tool in the process of building a kernel image file. The basic compilation link/combination structure is shown in Figure 5-33.

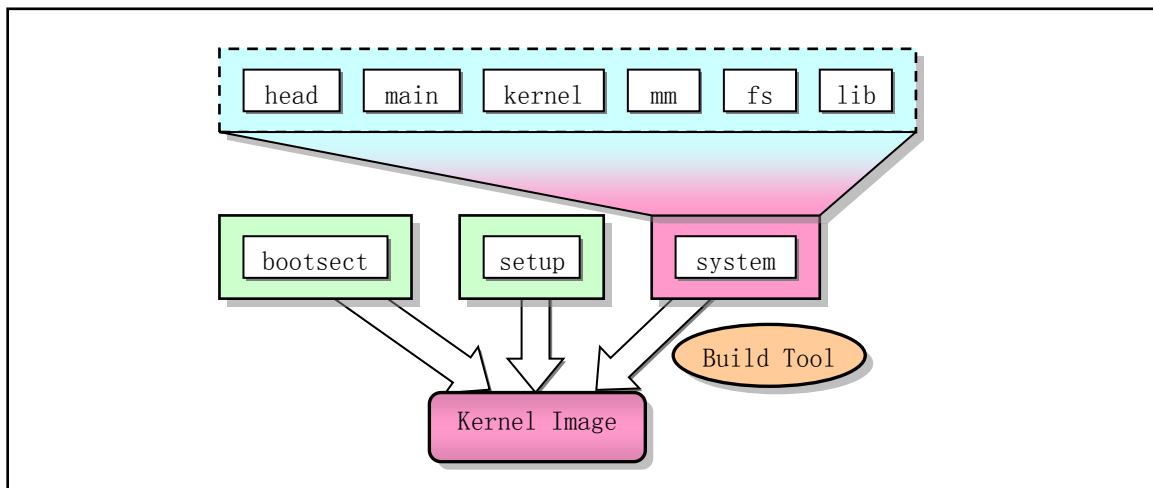


Figure 5-33 Kernel compilation link/combination structure

In the Linux kernel source code, except for the tools/, init/, and boot/ directories, each of the subdirectories contains a corresponding Makefile, which is identical in structure. Due to space limitations, only one annotation of the Makefile is given in the book. Program 5-1 is a detailed comment of the file.

Note that the statement with line number in the source file is the original statement, and statement without line number is the comment of the author of this book.

5.12.2 Program Annotation

Program 5-1 linux/Makefile

```

1 #
2 # if you want the ram-disk device, define this to be the
3 # size in blocks.
4 #
   # Define the size of the block if you want to use a RAM disk device. The default RAMDISK
   # is not defined here (commented out), otherwise gcc will be compiled with the option
   # '-DRAMDISK=512', see line 13 below.
5 RAMDISK = #-DRAMDISK=512
6
7 AS86      =as86 -O -a      # 8086 assembly and linker. The meaning of parameter is: -O
8 LD86      =ld86 -O        # generates 8086 object; -a code compatible with gas and gld.
9
10 AS        =gas            # GNU assembler and linker.see chapter 3 for more information.
11 LD        =gld
   # gld options: -s all symbolic info omitted in output file; -x removes all local symbols;
   # -M indicates that a link map is required, which is a memory address map generated by the
   # linker, which lists the location information that the code block is loaded into memory.
12 LDFLAGS =-s -x -M

   # gcc is the GNU compiler. For UNIX-like scripts, when you reference an identifier, you
   # need to precede it with a sign and enclose the identifier in parentheses.
13 CC        =gcc $(RAMDISK)

   # gcc options: -Wall prints all warnings; -O optimizes code. '-f flag' specifies flag.
   # Where, -fstrength-reduce is used to optimize loop statements; -fomit-frame-pointer
   # indicates that do not leave frame pointer in register for functions that do not require
   # frame pointer. This avoids operation and maintenance of frame pointer in the function.
   # -fcombine-regs is used to indicate that the compiler combines instructions that copy
   # one register to another. -mstring-insns is an option that Linus adds to gcc. It is used
   # by gcc-1.40 to copy string structures using 386 CPU string instructions and can be removed.
14 CFLAGS =-Wall -O -fstrength-reduce -fomit-frame-pointer \
15 -fcombine-regs -mstring-insns

   # cpp is gcc's preprocessor,used for macro substitution, conditional compilation, and
   # inclusion files specified with '#include'. All lines starting with '#' need to be
   # processed by preprocessor. All macros defined by '#define' will be replaced with their
   # definitions. All conditional lines such as '#if', '#ifdef', '#ifndef', and '#endif' are
   # used to determine whether to include statements in their specified range.
   # The option '-nostdinc -Iinclude' means not searching standard header file directory,
   # ie not using files in /usr/include/, but using directory specified by the '-I' option
   # or searching in the current directory.
16 CPP      =cpp -nostdinc -Iinclude
17
18 #
19 # ROOT_DEV specifies the default root-device when making the image.
20 # This can be either FLOPPY, /dev/xxxx or empty, in which case the
21 # default of /dev/hd6 is used by 'build'.
22 #
   # Here /dev/hd6 corresponds to first partition of the sceond hard disk. This is the

```



```

# location where the root file system is located when Linus develops the Linux kernel.
# /dev/hd2 represents 2nd partition of 1st hard disk and is used as a swap partition.
23 ROOT_DEV=/dev/hd6
24 SWAP_DEV=/dev/hd2
25
# Below are the object files generated in the kernel, mm, and fs directories. For ease of
# reference, they are represented here by the ARCHIVES identifier.
26 ARCHIVES=kernel/kernel.o mm/mm.o fs/fs.o

# Block and character device library files. '.a' indicates that the file is an archive
# file, that is, a library file containing a collection of executable binary code
# subroutines, usually generated by GNU's ar program. Ar is a GNU binary file tool for
# creating, modifying, and extracting files from archive files.
27 DRIVERS =kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a
28 MATH     =kernel/math/math.a
29 LIBS     =lib/lib.a           # A generic library compiled from files in lib/ dir.
30
# Here is the old-fashioned implicit suffix rule for make. This line instructs make to
# compile all '.c' files into '.s' assembly using the commands of rule after ':'. The
# whole sentence means that gcc uses the option specified by the CFLAGS and only uses the
# header file in include/ dir, and stops (-S) compiling, thereby generating an assembly
# file corresponding to each input C file. By default, the resulting assembly file is the
# original C file with suffix '.c' replaced with '.s'. '-o' indicates the output file.
# Where '$*.s' (or '$@') is automatic object variable, and '$<' represents the first
# prerequisite, here is the file that meets the condition '*.c'.
# The following three rules are used for different requirements. If the target is a .s
# file and the source is a .c file, the first rule will be used; if target is .o and the
# original is .s, the second rule will be used; if the target is a .o file, the original
# is a c file, then the third rule is used.
31 .c.s:
32     $(CC) $(CFLAGS) \
33     -nostdinc -Iinclude -S -o $*.s $<
34 .s.o:
35     $(AS) -c -o $*.o $<
36 .c.o:
37     $(CC) $(CFLAGS) \
38     -nostdinc -Iinclude -c -o $*.o $<
39
# The following 'all' means to create the topmost target of the Makefile, here is the
# Image file. It is the boot disk image file. If you write it to a floppy disk, you can
# use the floppy disk to boot the Linux system. See the line 46 for commands to write an
# Image to a floppy disk under Linux. The software rawrite.exe can be used under DOS.
40 all:    Image
41
# The target (Image) is generated by 4 elements behind the colon, which are the bootsect
# and setup files in boot/, the system and build files in tools/. Lines 43--44 are commands
# executed. Line 43 indicates that the bootsect, setup, and system files are assembled
# to be kernel image file, with $(ROOT_DEV) device using build utility in tools directory.
# The sync cmd on line 45 forces buffer to immediately write disk and update super block.
42 Image: boot/bootsect boot/setup tools/system tools/build
43     tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) \
44     $(SWAP_DEV) > Image

```

```
45         sync
46
47 # Indicates that the disk target is created from Image. dd is a standard cmd: copy a file,
48 # convert and format it according to the options. bs= the number of bytes read/written at
49 # a time. If= the input file, and of= the file to be output. Here /dev/PS0 refers to first
50 # floppy disk drive (device file). Use /dev/fd0 under current linux system.
51 disk: Image
52     dd bs=8192 if=Image of=/dev/PS0
53
54 tools/build: tools/build.c          # Create executable build tool.
55     $(CC) $(CFLAGS) \
56     -o tools/build tools/build.c
57
58 boot/head.o: boot/head.s            # Generate head.o object using the .s.o rule.
59
60 # Indicates that the tools/system is to be generated by the elements to the right of the
61 # colon. Line 57--62 is cmds to generate system object. The last '> System.map' means that
62 # gld needs to redirect the link info into the System.map file.
63 tools/system: boot/head.o init/main.o \
64     $(ARCHIVES) $(DRIVERS) $(MATH) $(LIBS)
65     $(LD) $(LDFLAGS) boot/head.o init/main.o \
66     $(ARCHIVES) \
67     $(DRIVERS) \
68     $(MATH) \
69     $(LIBS) \
70     -o tools/system > System.map
71
72 # The archive file math.a is built by cmds on line 64: cd into kernel/math/; run make.
73 kernel/math/math.a:
74     (cd kernel/math; make)
75
76 kernel/blk_drv/blk_drv.a:           # Create block driver archive file.
77     (cd kernel/blk_drv; make)
78
79 kernel/chr_drv/chr_drv.a:           # Character driver archive file.
80     (cd kernel/chr_drv; make)
81
82 kernel/kernel.o:                   # kernel object file.
83     (cd kernel; make)
84
85 mm/mm.o:                          # Memory management object file.
86     (cd mm; make)
87
88 fs/fs.o:                          # File system object file.
89     (cd fs; make)
90
91 lib/lib.a:                         # Internal lib.a
92     (cd lib; make)
93
94 # Compile setup.s file to generate setup.o using the 8086 assembler and linker. The option
95 # -s indicates that the symbol information in the target file need to be removed.
96 boot/setup: boot/setup.s
97     $(AS86) -o boot/setup.o boot/setup.s
```

```

88      $(LD86) -s -o boot/setup boot/setup.o
89
90      # Execute preprocessor, replace macro in *.S file to generate the corresponding *.s file.
91      boot/setup.s:    boot/setup.S include/linux/config.h
92                      $(CPP) -traditional boot/setup.S -o boot/setup.s
93
94      boot/bootsect.s:    boot/bootsect.S include/linux/config.h
95                      $(CPP) -traditional boot/bootsect.S -o boot/bootsect.s
96
97      boot/bootsect:    boot/bootsect.s
98                      $(AS86) -o boot/bootsect.o boot/bootsect.s
99                      $(LD86) -s -o boot/bootsect boot/bootsect.o
100
101      # When 'make clean' is executed, the commands on lines 101--107 are executed, and all
102      # files generated are removed. 'rm' is a file deletion cmd, and the option -f means to
103      # ignore files that do not exist and does not display deletion messages.
104      clean:
105          rm -f Image System.map tmp_make core boot/bootsect boot/setup \
106              boot/bootsect.s boot/setup.s
107          rm -f init/*.o tools/system tools/build boot/*.o
108          (cd mm;make clean)
109          (cd fs;make clean)
110          (cd kernel;make clean)
111          (cd lib;make clean)
112
113      # The rule will first execute the above clean rule, then compress the linux/ directory to
114      # generate a 'backup.Z' compressed file. 'tar cf - linux' means to execute the tar archiver
115      # on the linux/ directory. '-cf' indicates creating a archive file. '|compress -' means that
116      # the execution of the tar program is passed to the compressor compress by the pipeline
117      # operation, and the output of the compressor is saved as a backup.Z file.
118      backup: clean
119          (cd .. ; tar cf - linux | compress - > backup.Z)
120      sync
121
122      dep:
123      # This goal or rule is used to generate dependencies between files. These dependencies are
124      # created to let the make command use them to determine if a target object needs to be rebuilt.
125      # For example, when a header file has been changed, make can recompile all *.c files related
126      # to it through the generated dependencies. The specific method is as follows:
127      # Use the string editor sed to process the Makefile (here, this file), the output is to
128      # delete all the lines after the '### Dependencies' line in the Makefile, that is, delete
129      # all lines from 122 to the end of the file, and generate a temporary file tmp_make (also
130      # known as 114 lines). # Then perform a gcc preprocessing operation on each C file (in fact,
131      # only one file main.c) in the specified directory (init/). The flag '-M' tells the preprocessor
132      # cpp to output rules describing the relevance of each object file, and these rules conform
133      # to the make syntax. For each source file, the preprocessor outputs a rule whose result
134      # is the target file name of the corresponding source file plus its dependencies, that is,
135      # a list of all the header files contained in the source file. Then add the pre-processing
136      # results to the temporary file tmp_make, and finally copy the temporary file into a new Makefile.
137      # The '$$i' on line 115 is actually '$(i)'. Here 'i' is the value of the shell variable
138      # 'i' in front of this sentence.
139      sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
140      (for i in init/*.c;do echo -n "init/";$(CPP) -M $$i;done) >> tmp_make

```

```
116      cp tmp_make Makefile
117      (cd fs; make dep)
118      (cd kernel; make dep)
119      (cd mm; make dep)
120
121 ### Dependencies:
122 init/main.o : init/main.c include/unistd.h include/sys/stat.h \
123 include/sys/types.h include/sys/time.h include/time.h include/sys/times.h \
124 include/sys/utsname.h include/sys/param.h include/sys/resource.h \
125 include/utime.h include/linux/tty.h include/termios.h include/linux/sched.h \
126 include/linux/head.h include/linux/fs.h include/linux/mm.h \
127 include/linux/kernel.h include/signal.h include/asm/system.h \
128 include/asm/io.h include/stddef.h include/stdarg.h include/fcntl.h \
129 include/string.h
```

5.13 Summary

This chapter provides an overview of the kernel modes and architecture of the early Linux operating systems. First, the Linux 0.12 kernel usage and management memory methods, kernel state stack and user state stack settings and usage methods, interrupt mechanism, system clock timing, and process creation, scheduling, and termination methods are given. Then according to the directory structure of the source code, the basic functions and hierarchical relationships of the code files in each subdirectory are introduced in detail. It also explains the target file format used by Linux 0.12. Finally, we started with the makefile in the Linux kernel home directory and began to comment on the kernel source code.

This chapter can be seen as a summary of the important information about the Linux 0.12 kernel, so it can be used as a reference for reading subsequent chapters. From the bootloader in the next chapter, we formally began to annotate the source code of the kernel.