

# Linux内核完全注释

内核版本0.11

修正版 V3.0

赵 炯 编著



[www.oldlinux.org](http://www.oldlinux.org)

[gohigh@sh163.net](mailto:gohigh@sh163.net)

# Linux 内核 0.11 完全注释

A Heavily Commented Linux Kernel Source Code  
Linux Version 0.11

修正版 3.0  
(Revision 3.0)

赵炯  
Zhao Jiong

gohigh@sh163.net  
gohigh@gmail.com

www.plinux.org  
www.oldlinux.org

2007-06-07

## 内容简介

本书对早期 Linux 操作系统内核 (v0.11) 全部代码文件进行了详细全面的注释和说明,旨在让读者能够在尽量短的时间内对 Linux 的工作机理获得全面而深刻的理解,为进一步学习和研究 Linux 系统打下坚实的基础。虽然所选择的版本较低,但该内核已能够正常编译运行,并且其中已包括了 LINUX 工作原理的精髓。书中首先以 Linux 源代码版本的变迁为主线,概要而有趣地介绍了 Linux 系统的发展历史,同时着重说明了各个内核版本之间的主要区别和改进方面,给出了选择 0.11 (0.95) 版内核源代码作为研究对象的原因。然后在正式开始描述内核源代码之前,概要介绍了运行 Linux 系统的 PC 机的硬件组成结构、编制内核使用的汇编语言和 C 语言扩展部分,并且重点说明了 80X86 处理器在保护模式下运行的编程方法。接着我们详细介绍了 Linux 内核源代码目录树组织结构,并依据该组织结构对所有内核程序和文件进行了注释和详细说明。有关代码注释的章节安排基本上都为具体研究对象的概述、每个文件的功能介绍、代码内注释、代码中难点及相关资料介绍等部分逐步展开。为了加深读者对内核工作原理的理解,书中最后一章给出了围绕 Linux 0.11 系统的多个试验。试验中所使用的相关程序均可从本书配套网站 ([www.oldlinux.org](http://www.oldlinux.org)) 上下载。

本书适合作为高校计算机专业学生学习操作系统课程的辅助和实践教材,也适合 Linux 爱好者作为学习内核工作原理的自学参考书籍,还可供一般技术人员作为开发嵌入式系统的参考书使用。

## 版权说明

作者保留本电子书籍的修改和正式出版的所有权利。读者可以自由传播本书全部和部分章节的内容,但需要注明出处。由于目前本书仍然处于不断改进阶段,因此其中肯定还存在一些错误和不足之处。希望读者能踊跃给予批评指正或良好建议。反馈信息可以通过电子邮件发给我: [gohigh@sh163.net](mailto:gohigh@sh163.net) 或 [gohigh@gmail.com](mailto:gohigh@gmail.com), 也可直接来信至: 上海同济大学 电信工程学院计算机系 (江建慧教授收), 或机械电子工程研究所 (赵炯收) (上海四平路 1239 号, 邮编: 200092)。

© 2002 - 2007 by Zhao Jiong

© 2002 - 2007 赵炯 版权所有.

“RTFSC - Read The F\*\*king Source Code ☺!”

- Linus Benedict Torvalds

# 目录

序言 .....	1	5.5 LINUX 的系统调用 .....	160
本书的主要目标 .....	1	5.6 系统时间和定时 .....	162
现有书籍不足之处 .....	1	5.7 LINUX 进程控制 .....	164
阅读早期内核的其他好处 .....	2	5.8 LINUX 系统中堆栈的使用方法 .....	172
阅读完整源代码的重要性和必要性 .....	2	5.9 LINUX 0.11 采用的文件系统 .....	176
如何选择要阅读的内核代码版本 .....	3	5.10 LINUX 内核源代码的目录结构 .....	177
阅读本书需具备的基础知识 .....	3	5.11 内核系统与应用程序的关系 .....	184
使用早期版本是否过时? .....	4	5.12 LINUX/MAKEFILE 文件 .....	184
EXT2 文件系统与 MINIX 文件系统 .....	4	5.13 本章小结 .....	189
第 1 章 概述 .....	5	第 6 章 引导启动程序 (BOOT) .....	191
1.1 LINUX 的诞生和发展 .....	5	6.1 总体功能 .....	191
1.2 内容综述 .....	12	6.2 BOOTSECT.S 程序 .....	193
1.3 本章小结 .....	16	6.3 SETUP.S 程序 .....	203
第 2 章 微型计算机组成结构 .....	17	6.4 HEAD.S 程序 .....	221
2.1 微型计算机组成原理 .....	17	6.5 本章小结 .....	234
2.2 I/O 端口寻址和访问控制方式 .....	19	第 7 章 初始化程序 (INIT) .....	235
2.3 主存储器、BIOS 和 CMOS 存储器 .....	21	7.1 MAIN.C 程序 .....	235
2.4 控制器和控制卡 .....	23	7.2 环境初始化工作 .....	247
2.5 本章小结 .....	31	7.3 本章小结 .....	249
第 3 章 内核编程语言和环境 .....	32	第 8 章 内核代码 (KERNEL) .....	251
3.1 AS86 汇编器 .....	32	8.1 总体功能 .....	251
3.2 GNU AS 汇编 .....	38	8.2 MAKEFILE 文件 .....	254
3.3 C 语言程序 .....	48	8.3 ASM.S 程序 .....	256
3.4 C 与汇编程序的相互调用 .....	55	8.4 TRAPS.C 程序 .....	262
3.5 LINUX 0.11 目标文件格式 .....	63	8.5 SYSTEM_CALL.S 程序 .....	267
3.6 MAKE 程序和 MAKEFILE 文件 .....	72	8.6 MKTIME.C 程序 .....	279
第 4 章 80X86 保护模式及其编程 .....	75	8.7 SCHED.C 程序 .....	281
4.1 80X86 系统寄存器和系统指令 .....	75	8.8 SIGNAL.C 程序 .....	300
4.2 保护模式内存管理 .....	81	8.9 EXIT.C 程序 .....	311
4.3 分段机制 .....	85	8.10 FORK.C 程序 .....	318
4.4 分页机制 .....	96	8.11 SYS.C 程序 .....	326
4.5 保护 .....	99	8.12 VSPRINTF.C 程序 .....	333
4.6 中断和异常处理 .....	110	8.13 PRINTK.C 程序 .....	341
4.7 任务管理 .....	120	8.14 PANIC.C 程序 .....	342
4.8 保护模式编程初始化 .....	128	8.15 本章小结 .....	343
4.9 一个简单的多任务内核实例 .....	131	第 9 章 块设备驱动程序 (BLOCK DRIVER) .....	345
第 5 章 LINUX 内核体系结构 .....	141	9.1 总体功能 .....	346
5.1 LINUX 内核模式 .....	141	9.2 MAKEFILE 文件 .....	349
5.2 LINUX 内核系统体系结构 .....	142	9.3 BLK.H 文件 .....	351
5.3 LINUX 内核对内存的管理和使用 .....	144	9.4 HD.C 程序 .....	355
5.4 中断机制 .....	157	9.5 LL_RW_BLK.C 程序 .....	378
		9.6 RAMDISK.C 程序 .....	384
		9.7 FLOPPY.C 程序 .....	390

## 第 10 章 字符设备驱动程序(CHAR DRIVER) ..... 417

10.1 总体功能 .....	417
10.2 MAKEFILE 文件.....	427
10.3 KEYBOARD.S 程序 .....	429
10.4 CONSOLE.C 程序.....	448
10.5 SERIAL.C 程序 .....	474
10.6 RS_IO.S 程序 .....	483
10.7 TTY_IO.C 程序.....	487
10.8 TTY_IOCTL.C 程序.....	499

## 第 11 章 数学协处理器(MATH)..... 507

11.1 MAKEFILE 文件.....	507
11.2 MATH-EMULATION.C 程序.....	509

## 第 12 章 文件系统(FS)..... 511

12.1 总体功能 .....	511
12.2 MAKEFILE 文件.....	527
12.3 BUFFER.C 程序 .....	530
12.4 BITMAP.C 程序.....	547
12.5 TRUNCT.C 程序.....	553
12.6 INODE.C 程序 .....	555
12.7 SUPER.C 程序 .....	567
12.8 NAMEI.C 程序 .....	577
12.9 FILE_TABLE.C 程序.....	601
12.10 BLOCK_DEV.C 程序 .....	601
12.11 FILE_DEV.C 程序.....	605
12.12 PIPE.C 程序.....	608
12.13 CHAR_DEV.C 程序 .....	612
12.14 READ_WRITE.C 程序.....	615
12.15 OPEN.C 程序 .....	621
12.16 EXEC.C 程序 .....	627
12.17 STAT.C 程序 .....	647
12.18 FCNTL.C 程序 .....	649
12.19 IOCTL.C 程序.....	652

## 第 13 章 内存管理(MM)..... 655

13.1 总体功能 .....	655
13.2 MAKEFILE 文件.....	661
13.3 MEMORY.C 程序.....	662
13.4 PAGE.S 程序.....	679

## 第 14 章 头文件(INCLUDE) ..... 683

14.1 INCLUDE/目录下的文件 .....	683
14.2 A.OUT.H 文件.....	684
14.3 CONST.H 文件 .....	695
14.4 CTYPE.H 文件 .....	695
14.5 ERRNO.H 文件 .....	697
14.6 FCNTL.H 文件 .....	699
14.7 SIGNAL.H 文件 .....	701
14.8 STDARG.H 文件.....	703
14.9 STDDEF.H 文件 .....	704
14.10 STRING.H 文件 .....	705
14.11 TERMIOS.H 文件 .....	715

14.12 TIME.H 文件.....	722
14.13 UNISTD.H 文件.....	724
14.14 UTIME.H 文件 .....	729
14.15 INCLUDE/ASM/目录下的文件 .....	731
14.16 IO.H 文件 .....	731
14.17 MEMORY.H 文件.....	732
14.18 SEGMENT.H 文件.....	733
14.19 SYSTEM.H 文件.....	735
14.20 INCLUDE/LINUX/目录下的文件 .....	739
14.21 CONFIG.H 文件 .....	739
14.22 FDREG.H 头文件 .....	741
14.23 FS.H 文件.....	744
14.24 HDREG.H 文件.....	749
14.25 HEAD.H 文件 .....	752
14.26 KERNEL.H 文件.....	753
14.27 MM.H 文件.....	754
14.28 SCHED.H 文件.....	754
14.29 SYS.H 文件 .....	761
14.30 TTY.H 文件.....	763
14.31 INCLUDE/SYS/目录中的文件.....	766
14.32 STAT.H 文件 .....	766
14.33 TIMES.H 文件.....	767
14.34 TYPES.H 文件.....	768
14.35 UTSNAME.H 文件.....	769
14.36 WAIT.H 文件.....	770

## 第 15 章 库文件(LIB)..... 773

15.1 MAKEFILE 文件 .....	774
15.2 _EXIT.C 程序 .....	776
15.3 CLOSE.C 程序 .....	777
15.4 CTYPE.C 程序 .....	777
15.5 DUP.C 程序 .....	778
15.6 ERRNO.C 程序.....	779
15.7 EXECVE.C 程序.....	779
15.8 MALLOC.C 程序 .....	780
15.9 OPEN.C 程序 .....	789
15.10 SETSID.C 程序.....	790
15.11 STRING.C 程序.....	791
15.12 WAIT.C 程序.....	791
15.13 WRITE.C 程序 .....	792

## 第 16 章 建造工具(TOOLS)..... 795

16.1 BUILD.C 程序.....	795
----------------------	-----

## 第 17 章 实验环境设置与使用方法 ..... 802

17.1 BOCHS 仿真系统 .....	802
17.2 在 BOCHS 中运行 LINUX 0.11 系统.....	806
17.3 访问磁盘映像文件中的信息.....	813
17.4 编译运行简单内核示例程序.....	815
17.5 利用 BOCHS 调试内核.....	817
17.6 创建磁盘映像文件.....	824
17.7 制作根文件系统.....	827
17.8 在 LINUX 0.11 系统上编译 0.11 内核.....	834
17.9 在 REDHAT 9 系统下编译 LINUX 0.11 内核 ..	835

17.10 内核引导启动+根文件系统组成的集成盘	838	附录 2 ASCII 码表 .....	863
17.11 从硬盘启动：利用 SHOELACE 引导软件....	843	附录 3 常用 C0、C1 控制字符表.....	864
17.12 利用 GDB 和 BOCHS 调试内核源代码 .....	846	附录 4 常用转义序列和控制序列 .....	865
<b>参考文献.....</b>	<b>853</b>	附录 5 第 1 套键盘扫描码集.....	868
<b>附录.....</b>	<b>855</b>	<b>索引 .....</b>	<b>869</b>
附录 1 内核数据结构.....	855		

# 序言

本书是一本有关 Linux 操作系统内核基本工作原理的入门读物。

## 本书的主要目标

本书的主要目标是使用尽量少的篇幅或在有限的篇幅内，对完整的 Linux 内核源代码进行解剖，以期对操作系统的基本功能和实际实现方式获得全方位的理解。做到对 linux 内核有一个完整而深刻的理解，对 linux 操作系统的基本工作原理真正理解和入门。

本书读者群的定位是一些知晓 Linux 系统的一般使用方法或具有一定的编程基础，但比较缺乏阅读目前最新内核源代码的基础知识，又急切希望能够进一步理解 UNIX 类操作系统内核工作原理和实际代码实现的爱好者。这部分读者的水平应该界于初级与中级水平之间。目前，这部分读者人数在 Linux 爱好者中所占的比例是很高的，而面向这部分读者以比较易懂和有效的手段讲解内核的书籍资料不多。

## 现有书籍不足之处

目前已有的描述 Linux 内核的书籍，均尽量选用最新 Linux 内核版本（例如 Redhat 7.0 使用 2.2.16、Fedora Core 4 使用 2.6.11 稳定版等）进行描述，但由于目前 Linux 内核整个源代码的大小已经非常得大（例如 2.2.20 版就已具有 268 万行代码！），因此这些书籍仅能对 Linux 内核源代码进行选择性地或原理性地说明，许多系统实现细节被忽略。因此并不能给予读者对实际 Linux 内核有清晰而完整的理解。

Scott Maxwell 著的一书《Linux 内核源代码分析》（陆丽娜等译）基本上是针对 Linux 中级水平的读者，需要较为全面的基础知识才能完全理解。而且可能是由于篇幅所限，该书并没有对所有 Linux 内核代码进行注释，略去了很多内核实现细节，例如其中内核中使用的各个头文件(\*.h)、生成内核代码映像文件的工具程序、各个 make 文件的作用和实现等均没有涉及。因此对于处于初中级水平之间的读者来说阅读该书有些困难。

浙江大学出版的《Linux 内核源代码情景分析》一书，也基本有这些不足之处。甚至对于一些具有较高 Linux 系统应用水平的计算机本科高年级学生，由于该书篇幅问题以及仅仅选择性地讲解内核源代码，也不能真正吃透内核的实际实现方式，因而往往刚开始阅读就放弃了。该书刚面市时，本人曾极力劝说学生购之阅读，而在二个月后调查阅读学习情况时，基本都存在看不下去或不能理解等问题，大多数人都放弃了。

John Lions 著的《莱昂氏 UNIX 源代码分析》一书虽然是一本学习 UNIX 类操作系统内核源代码很好的书籍，但是由于其采用的是 UNIX V6 版，其中系统调用等部分代码是用早已废弃的 PDP-11 系列机的汇编语言编制的，因此在阅读和理解与硬件部分相关的源代码时就会遇到较大的困难。

A.S.Tanenbaum 的书《操作系统：设计与实现》是一本有关操作系统内核实现很好的入门书籍，但该书所叙述的 MINIX 系统是一种基于消息传递的内核实现机制，与 Linux 内核的实现有所区别。因此在学习该书之后，并不能很顺利地即刻着手进一步学习较新的 Linux 内核源代码实现。

在使用这些书籍进行学习时会有一种“盲人摸象”的感觉，不能真正理解 Linux 内核系统具体实现的整体概念，尤其是对那些 Linux 系统初学者或刚学会如何使用 Linux 系统的人在使用那些书学习内核原理时，内核的整体运作结构并不能清晰地脑海中形成。这在本人多年的 Linux 内核学习过程中也深



有体会。在 1991 年 10 月份，Linux 的创始人 Linus Torvalds 在开发出 Linux 0.03 版后写的一篇文章中也提到了同样的问题。在这篇题为“[LINUX--a free unix-386 kernel](http://oldlinux.org/Linus/)”<sup>1</sup>的文章中，他说：“开发 Linux 是为了那些操作系统爱好者和计算机科学系的学生使用、学习和娱乐”。“自由软件基金会的 GNU Hurd 系统如果开发出来就已经显得太庞大而不适合学习和理解。”而现今流行的 Linux 系统要比当年 GNU 的 Hurd 系统更为庞大和复杂，因此同样也已经不适合作为操作系统初学者的入门学习起点。这也是作者基于 Linux 早期内核版本写作本书的动机之一。

为了填补这个空缺，本书的主要目标是使用尽量少的篇幅或在有限的篇幅内，对完整的 Linux 内核源代码进行全面解剖，以期对操作系统的基本功能和实际实现方式获得全方位的理解。做到对 Linux 内核有一个完整而深刻的理解，对 Linux 操作系统的基本工作原理真正理解和入门。

## 阅读早期内核的其他好处

目前，已经出现不少基于 Linux 早期内核而开发的专门用于嵌入式系统的内核版本，如 DJJ 的 x86 操作系统、Uclinux 等（在 [www.linux.org](http://www.linux.org) 上有专门目录），世界上也有许多人认识到通过早期 Linux 内核源代码学习的好处，目前国内也已经有人正在组织人力注释出版类似本文的书籍。因此，通过阅读 Linux 早期内核版本的源代码，的确是学习 Linux 系统的一种行之有效的途径，并且对研究和应用 Linux 嵌入式系统也有很大的帮助。

在对早期内核源代码的注释过程中，作者发现，早期内核源代码几乎就是目前所使用的较新内核的一个精简版本。其中已经包括了目前新版本中几乎所有的基本功能原理的内容。正如《系统软件：系统编程导论》一书的作者 Leland L. Beck 在介绍系统程序以及操作系统设计时，引入了一种极其简化的简单指令计算机(SIC)系统来说明所有系统程序的设计和实现原理，从而既避免了实际计算机系统的复杂性，又能透彻地说明问题。这里选择 Linux 的早期内核版本作为学习对象，其指导思想与 Leland 的一致。这对 Linux 内核学习的入门者来说，是最理想的选择之一。能够在尽可能短的时间内深入理解 Linux 内核的基本工作原理。

对于那些已经比较熟悉内核工作原理的人，为了能让自己在实际工作中对系统的实际运转机制不产生一种空中楼阁的感觉，因此也有必要阅读内核源代码。

当然，使用早期内核作为学习的对象也有不足之处。所选用的 Linux 早期内核版本不包含对虚拟文件系统 VFS 的支持、对网络系统的支持、仅支持 a.out 执行文件和对其他一些现有内核中复杂子系统的说明。但由于本书是作为 Linux 内核工作机理实现的入门教材，因此这也正是选择早期内核版本的优点之一。通过学习本书，可以为进一步学习这些高级内容打下扎实的基础。

## 阅读完整源代码的重要性和必要性

正如 Linux 系统的创始人在一篇新闻组投稿上所说的，要理解一个软件系统的真正运行机制，一定要阅读其源代码（RTFSC – Read The Fucking Source Code）。系统本身是一个完整的整体，具有很多看似不重要的细节存在，但是若忽略这些细节，就会对整个系统的理解带来困难，并且不能真正了解一个实际系统的实现方法和手段。

虽然通过阅读一些操作系统原理经典书籍（例如 M.J.Bach 的《UNIX 操作系统设计》）能够对 UNIX 类操作系统的工作原理有一些理论上的指导作用，但实际上对操作系统的真正组成和内部关系实现的理解仍不是很清晰。正如 AST 所说的，“许多操作系统教材都是重理论而轻实践”，“多数书籍和课程为调度算法耗费大量的时间和篇幅而完全忽略 I/O，其实，前者通常不足一页代码，而后者往往要占到整个

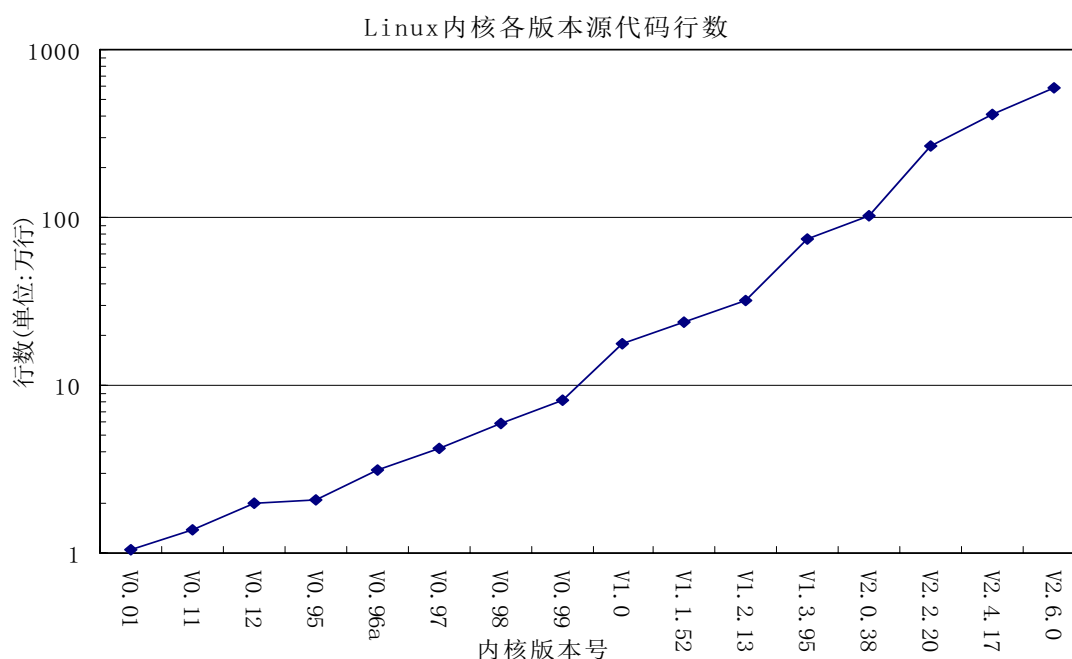
---

<sup>1</sup> 原文可参见：<http://oldlinux.org/Linus/>

系统三分之一的代码总量。”内核中大量的重要细节均未提到。因此并不能让读者理解一个真正的操作系统实现的奥妙所在。只有在详细阅读过完整的内核源代码之后，才会对系统有一种豁然开朗的感觉，对整个系统的运作过程有深刻的理解。以后再选择最新的或较新内核源代码进行学习时，也不会碰到大问题，基本上都能顺利地理解新代码的内容。

## 如何选择要阅读的内核代码版本

那么，如何选择既能达到上述要求，又不被太多的内容而搞乱头脑，选择一个适合的 Linux 内核版本进行学习，提高学习的效率呢？作者通过对大量内核版本进行比较和选择后，最终选择了与目前 Linux 内核基本功能较为相近，又非常短小的 0.11 版内核作为入门学习的最佳版本。下图是对一些主要 Linux 内核版本行数的统计。



目前的 Linux 内核源代码量都在几百万行的数量上，2.6.0 版内核代码行数约为 592 万行，极其庞大，对这些版本进行完全注释和说明几乎是不可能的。而 0.11 版内核不超过 2 万行代码量，因此完全可以在一本书中解释和注释清楚。麻雀虽小，五脏俱全。为了对所研究的系统有感性的了解，并能利用实验来加深对原理的理解，作者还专门重建了基于该内核的可运行的 Linux 0.11 系统。由于其中含有 GNU gcc 编译环境，因此使用该系统也能做一些简单的开发工作。

另外，使用该版本可以避免使用现有较新内核版本中已经变得越来越复杂得各子系统部分的研究（如虚拟文件系统 VFS、ext2 或 ext3 文件系统、网络子系统、新的复杂的内存管理机制等）。

## 阅读本书需具备的基础知识

在阅读本书时，读者必须具备一些基本的 C 语言知识和 Intel CPU 汇编语言知识。有关 C 语言最佳的参考资料仍然是 Brain W. Kernighan 和 Dennis M. Ritchie 编写的《The C Programming Language》一书。而汇编语言的资料则可以参考任意一本讲解与 Intel CPU 相关的汇编语言教材。另外还需要一些嵌入式汇编语言的资料。有关嵌入式汇编的权威信息都包含在 GNU gcc 编译器手册中。我们也可以从 Internet

网上搜索到一些有关嵌入式汇编比较有价值的短文。本书中也包含了一些嵌入汇编基本的语法说明（第 5.5 节程序列表后）。

除此之外，还希望读者具备以下一些基础知识或者有相关的参考书籍在身边。其一是有关 80x86 处理器结构和编程的知识或资料。例如可以从网上下载的 80x86 编程手册（INTEL 80386 Programmer's Reference Manual）；其二是有关 80x86 硬件体系结构和接口编程的知识或资料。有关这方面的资料很多；其三还应具备初级使用 Linux 系统的简单技能。

另外，由于 Linux 系统内核实现最早是根据 M.J.Bach 的《UNIX 操作系统设计》一书的基本原理开发的，源代码中许多变量或函数的名称都来自该书，因此在阅读本书时若能适当参考该书，会更易于理解内核源代码。

Linux 在最初开发 Linux 操作系统时，参照了 MINIX 操作系统。例如，最初的 Linux 内核版本完全照搬了 MINIX 1.0 文件系统。因此，在阅读本书时，A.S.Tanenbaum 的书《操作系统：设计与实现》也具有较大的参考价值。但 Tanenbaum 的书描述的是一种基于消息传递在内核各模块之间进行通信（信息交换），这与 Linux 内核的工作机制不一样。因此可以仅参考其中有关一般操作系统工作原理章节和文件系统实现的内容。

## 使用早期版本是否过时？

表面看来本书对 Linux 早期内核版本注释的内容犹如 Linux 操作系统刚公布时 Tanenbaum 就认为其已经过时的（Linux is obsolete）想法一样，但通过学习本书内容，你就会发现，利用本书学习 Linux 内核，由于内核源代码量短小而精干，因此会有极高的学习效率，能够做到事半功倍，快速入门。并且对继续进一步选择新内核部分源代码的学习打下坚实的基础。在学习完本书之后，你将对系统的运作原理有一个非常完整而实际的概念，这种完整概念能使人很容易地进一步选择和学习新内核源代码中的任何部分，而不需要再去啃读代码量巨大的新内核中完整的源代码。

## Ext2 文件系统与 MINIX 文件系统

目前 Linux 系统上所使用的 Ext2（或最新的 Ext3）文件系统是在内核 1.x 之后开发的。其功能详尽并且性能也非常完整和稳固，是目前 Linux 操作系统上默认的标准文件系统。但是，作为对 Linux 操作系统完整工作原理入门学习所使用的部分，原则上是越精简越好。为了达到对一个操作系统有完整的理解，并且能不被其中各子系统中复杂和过多的细节所喧宾夺主，在选择学习剖析用的内核版本时，只要系统的部分代码内容能说明实际工作原理，就越简单越好。

Linux 内核 0.11 版上当时仅包含最为简单的 MINIX 1.0 文件系统，对于理解一个操作系统中文件系统的实际组成和工作原理已经足够。这也是选择 Linux 早期内核版本进行学习的主要原因之一。

在完整阅读完本书之后，相信您定会发出这样的感叹：“对于 Linux 内核系统，我现在终于入门了！”。此时，您应该有十分的把握去进一步学习最新 Linux 内核中各部分的工作原理和过程了。

同济大学  
赵炯 博士  
2007.6.

# 第1章 概述

本章首先回顾了 Linux 操作系统诞生、开发和成长的过程，由此可以理解本书为什么会选择 Linux 系统早期版本作为学习对象的一些原因。然后具体说明了选择早期 Linux 内核版本进行学习的优点和不足之处以及如何开始进一步学习。最后对各章的内容进行了简要介绍。

## 1.1 Linux 的诞生和发展

Linux 操作系统是 UNIX 操作系统的一种克隆系统。它诞生于 1991 年的 10 月 5 日（这是第一次正式向外公布的时间）。此后借助于 Internet 网络，经过全世界各地计算机爱好者的共同努力，现已成为当今世界上使用最多的一种 UNIX 类操作系统，并且使用人数还在迅猛增长。

Linux 操作系统的诞生、发展和成长过程依赖于以下五个重要支柱：UNIX 操作系统、MINIX 操作系统、GNU 计划、POSIX 标准和 Internet 网络。下面根据这五个基本线索来追寻一下 Linux 的开发历程、它的酝酿过程以及最初的发展经历。首先分别介绍其中的四个基本要素，然后根据 Linux 的创始人 Linus Torvalds 对对计算机感兴趣而自学计算机知识、到心里开始酝酿编制一个自己的操作系统、到最初 Linux 内核 0.01 版公布以及从此如何艰难地一步一个脚印地在全世界 hacker 的帮助下最后推出比较完善的 1.0 版本这段时间的发展经过，也即对 Linux 的早期发展历史进行详细介绍。

当然，目前 Linux 内核版本已经开发到了 2.6.x 版。而大多数 Linux 系统中所用到的内核是稳定的 2.6.12 版内核（其中第 2 个数字若是奇数则表示是正在开发的版本，不能保证系统的稳定性）。对于 Linux 的一般发展史，许多文章和书籍都有介绍，这里就不重复。

### 1.1.1 UNIX 操作系统的诞生

Linux 操作系统是 UNIX 操作系统的一个克隆版本。UNIX 操作系统是美国贝尔实验室的 Ken.Thompson 和 Dennis Ritchie 于 1969 年夏在 DEC PDP-7 小型计算机上开发的一个分时操作系统。

Ken Thompson 为了能在闲置不用的 PDP-7 计算机上运行他非常喜欢的星际旅行（Space travel）游戏，于是在 1969 年夏天乘他夫人回家乡加利福尼亚渡假期间，在一个月内开发出了 UNIX 操作系统的原型。当时使用的是 BCPL 语言（基本组合编程语言），后经 Dennis Ritchie 于 1972 年用移植性很强的 C 语言进行了改写，使得 UNIX 系统在大专院校得到了推广。

### 1.1.2 MINIX 操作系统

MINIX 系统是由 Andrew S. Tanenbaum（AST）开发的。AST 是在荷兰 Amsterdam 的 Vrije 大学数学与计算机科学系统工作，是 ACM 和 IEEE 的资深会员（全世界也只有很少人是两会的资深会员）。共发表了 100 多篇文章，5 本计算机书籍。

AST 虽出生在美国纽约，但却是荷兰侨民（1914 年他的祖辈来到美国）。他在纽约上的中学、M.I.T 上的大学、加州大学 Berkeley 分校念的博士学位。由于读博士后的缘故，他来到了家乡荷兰。从此就与家乡一直有来往。后来就在 Vrije 大学开始教书、带研究生。荷兰首都 Amsterdam 是个常年阴雨绵绵的城市，但对于 AST 来说，这最好不过了，因为在这样的环境下他就可以经常待在家中摆弄他的计算机了。

MINIX 是他 1987 年编制的，主要用于学生学习操作系统原理。到 1991 年时版本是 1.5。目前主要有两个版本在使用：1.5 版和 2.0 版。当时该操作系统在大学使用是免费的，但其他用途则不是。当然目

前 MINIX 系统已经是免费的，可以从许多 FTP 上下载。

对于 Linux 系统，他后来曾表示对其开发者 Linus 的称赞。但他认为 Linux 的发展很大原因是由于他保持了 MINIX 的小型化，能让学生在一个学期内就能学完，因而没有接纳全世界许多人对 MINIX 的扩展要求。因此在这样的前提下激发了 Linus 编写 Linux 系统。当然 Linus 也正好抓住了这个好时机。

作为一个操作系统，MINIX 并不是优秀者，但它同时提供了用 C 语言和汇编语言编写的系统源代码。这是第一次使得有抱负的程序员或 hacker 能够阅读操作系统的源代码。在当时，这种源代码是软件商们一直小心守护着的秘密。

### 1.1.3 GNU 计划

GNU 计划和自由软件基金会 FSF(the Free Software Foundation)是由 Richard M. Stallman 于 1984 年一手创办的。旨在开发一个类似 UNIX 并且是自由软件的完整操作系统：GNU 系统（GNU 是"GNU's Not Unix"的递归缩写，它的发音为"guh-NEW"）。各种使用 Linux 作为核心的 GNU 操作系统正在被广泛的使用。虽然这些系统通常被称作"Linux"，但是 Stallman 认为，严格地说，它们应该被称为 GNU/Linux 系统。

到上世纪 90 年代初，GNU 项目已经开发出许多高质量的免费软件，其中包括有名的 emacs 编辑系统、bash shell 程序、gcc 系列编译程序、gdb 调试程序等等。这些软件为 Linux 操作系统的开发创造了一个合适的环境。这是 Linux 能够诞生的基础之一，以至于目前许多人都将 Linux 操作系统称为“GNU/Linux”操作系统。

### 1.1.4 POSIX 标准

POSIX (Portable Operating System Interface for Computing Systems) 是由 IEEE 和 ISO/IEC 开发的一簇标准。该标准是基于现有的 UNIX 实践和经验，描述了操作系统的调用服务接口。用于保证编制的应用程序可以在源代码一级上在多种操作系统上移植和运行。它是在 1980 年早期一个 UNIX 用户组 (usr/group) 的早期工作基础上取得的。该 UNIX 用户组原来试图将 AT&T 的 System V 操作系统和 Berkeley CSRG 的 BSD 操作系统的调用接口之间的区别重新调和集成。并于 1984 年定制出了 /usr/group 标准。

1985 年，IEEE 操作系统技术委员会标准小组委员会 (TCOS-SS) 开始在 ANSI 的支持下责成 IEEE 标准委员会制定有关程序源代码可移植性操作系统服务接口正式标准。到了 1986 年 4 月，IEEE 制定出了试用标准。第一个正式标准是在 1988 年 9 月份批准的 (IEEE 1003.1-1988)，也既以后经常提到的 POSIX.1 标准。

到 1989 年，POSIX 的工作被转移至 ISO/IEC 社团，并由 15 工作组继续将其制定成 ISO 标准。到 1990 年，POSIX.1 与已经通过的 C 语言标准联合，正式批准为 IEEE 1003.1-1990 (也是 ANSI 标准) 和 ISO/IEC 9945-1:1990 标准。

POSIX.1 仅规定了系统服务应用程序编程接口 (API)，仅概括了基本的系统服务标准。因此工作组期望对系统的其他功能也制定出标准。这样 IEEE POSIX 的工作就开始展开了。刚开始有十个批准的计划在进行，有近 300 多人参加每季度为期一周的会议。着手的工作有命令与工具标准 (POSIX.2)、测试方法标准 (POSIX.3)、实时 API (POSIX.4) 等。到了 1990 年上半年已经有 25 个计划在进行，并且有 16 个工作组参与了进来。与此同时，还有一些组织也在制定类似的标准，如 X/Open, AT&T, OSF 等。

在 90 年代初，POSIX 标准的制定正处在最后投票敲定的时候，那是 1991-1993 年间。此时正是 Linux 刚刚起步的时候，这个 UNIX 标准为 Linux 提供了极为重要的信息，使得 Linux 能够在标准的指导下进行开发，并能够与绝大多数 UNIX 操作系统兼容。在最初的 Linux 内核源代码中 (0.01 版、0.11 版) 就已经为 Linux 系统与 POSIX 标准的兼容做好了准备工作。在 Linux 0.01 版内核的 /include/unistd.h 文件中就已经定义了几个有关 POSIX 标准要求的符号常数，而且 Linus 在注释中已写道：“OK，这也许是个玩笑，但我正在着手研究它呢”。

1991 年 7 月 3 日在 comp.os.minix 上发布的 post 上就已经提到了正在搜集 POSIX 的资料。其中透露了他正在着手一个操作系统的开发，并且在开发之初已经想到要实现与 POSIX 相兼容的问题了。

### 1.1.5 Linux 操作系统的诞生

在 1981 年，IBM 公司推出了享誉全球的微型计算机 IBM PC。在 1981-1991 年间，MS-DOS 操作系统一直是微型计算机操作系统的主宰。此时计算机硬件价格虽然逐年下降，但软件价格仍然居高不下。当时 Apple 的 MACs 操作系统可以说是性能最好的，但是其天价使得没人能够轻易靠近。

当时的另一个计算机技术阵营就是 UNIX 世界。但是 UNIX 操作系统就不仅是价格昂贵的问题了。为了寻求高利润率，UNIX 经销商们把价格拍得极高，PC 小用户根本不能靠近它。曾经一度收到 Bell Labs 许可而能在大学中用于教学的 UNIX 源代码也一直被小心地守卫着不许公开。对于广大的 PC 用户，软件行业的大型供应商们始终没有给出有效的解决这个问题的手段。

正在此时，出现了 MINIX 操作系统，并且有一本描述其设计实现原理的书同时发行。由于 AST 的这本书写的非常详细，并且叙述得有条有理，于是几乎全世界的计算机爱好者都开始看这本书，以期能理解操作系统的工作原理。其中也包括 Linux 系统的创始者 Linus Benedict Torvalds。

当时(1991 年)，Linus Benedict Torvalds 是赫尔辛基大学计算机科学系的二年级学生，也是一个自学的计算机 hacker。这个 21 岁的芬兰年轻人喜欢鼓捣他的计算机，测试计算机的性能和限制。但当时他所缺乏的就是一个专业级的操作系统。

在同一年间，GNU 计划已经开发出了许多工具软件。其中最受期盼的 GNU C 编译器已经出现，但还没有开发出免费的 GNU 操作系统。即使是教学使用的 MINIX 操作系统也开始有了版权，需要购买才能得到源代码。虽然 GNU 的操作系统 HURD 一直在开发之中，但在当时看来不能在几年内完成。

为了更好地学习计算机知识（或许也只是为了兴趣©），Linus 使用圣诞节的压岁钱和贷款购买了一台 386 兼容电脑，并从美国邮购了一套 MINIX 系统软件。就在等待 MINIX 软件期间，Linus 认真学习了有关 Intel 80386 的硬件知识。为了能通过 Modem 拨号连接到学校的主机上，他使用汇编语言并利用 80386 CPU 的多任务特性编制出一个终端仿真程序。此后为了将自己一台老式电脑上的软件复制到新电脑上，他还为软盘驱动器、键盘等硬件设备编制出相应的驱动程序。

通过编程实践，并在学习过程中认识到 MINIX 系统的诸多限制（MINIX 虽然很好，但只是一个用于教学目的简单操作系统，而不是一个强有力的实用操作系统），而且通过上述实践 Linus 已经有了一些类似于操作系统硬件设备驱动程序的代码，于是他开始有了编制一个新操作系统的想法。此时 GNU 计划已经开发出许多工具软件，其中最受期盼的 GNU C 编译器已经出现。虽然 GNU 的免费操作系统 HURD 正在开发中。但 Linus 已经等不及了。

从 1991 年 4 月份起，他通过修改终端仿真程序和硬件驱动程序，开始编制起自己的操作系统来。刚开始，他的目的很简单，只是为了学习 Intel 386 体系结构保护模式运行方式下的编程技术。但后来 Linux 的发展却完全改变了初衷。根据 Linus 在 comp.os.minix 新闻组上发布的消息，我们可以知道他逐步从学习 MINIX 系统阶段发展到开发自己的 Linux 系统的过程。

Linus 第 1 次向 comp.os.minix 投递消息是在 1991 年 3 月 29 日。所发帖子的题目是“gcc on minix-386 doesn't optimize”，是有关 gcc 编译器在 MINIX-386 系统上运行优化的问题（MINIX-386 是一个由 Bruce Evans 改进的利用 Intel 386 特性的 32 位 MINIX 系统）。由此可知，Linus 在 1991 年初期就已经开始深入研究了 MINIX 系统，并在这段时间有了改进 MINIX 操作系统的思想。在进一步学习 MINIX 系统之后，这个想法逐步演变成想重新设计一个基于 Intel 80386 体系结构的新操作系统的构思。

他在回答有人提出 MINIX 上的一个问题时，所说的第一句话就是“阅读源代码”（“RTFSC (Read the F\*\*ing Source Code :-)”）。他认为答案就在源程序中。这也说明了对于学习系统软件来说，我们不光需要懂得系统的工作基本原理，还需要结合实际系统，学习实际系统的实现方法。因为理论毕竟是理论，其中省略了许多枝节，而这些枝节问题虽然没有太多的理论含量，但却是一个系统必要的组成部分，就象麻雀身上的一根羽毛。

从 1991 年 4 月份开始, Linus 几乎花费了全部时间研究 MINIX-386 系统(Hacking the kernel), 并且尝试着移植 GNU 的软件到该系统上(GNU gcc、bash、gdb 等)。并于 4 月 13 日在 comp.os.minix 上发布说自己已经成功地将 bash 移植到了 MINIX 上, 而且已经爱不释手、不能离开这个 shell 软件了。

第一个与 Linux 有关的消息是在 1991 年 7 月 3 日在 comp.os.minix 上发布的(当然, 那时还不存在 Linux 这个名称, 当时 Linus 脑子里想的名称可能是 FREAX ☺, FREAX 的英文含义是怪诞的、怪物、异想天开等)。其中透露了他正在进行 Linux 系统的开发, 并且已经想到要实现与 POSIX 兼容的问题了。

在 Linus 另一个发布的消息中(1991 年 8 月 25 日 comp.os.minix), 他向所有 MINIX 用户询问“你最喜欢在 MINIX 系统中见到什么?”(“What would you like to see in minix?”), 在该消息中他首次透露出正在开发一个(免费的)386(486)操作系统, 并且说只是兴趣而已, 代码不会很大, 也不会象 GNU 的那样专业。希望大家反馈一些对于 MINIX 系统中喜欢哪些特色不喜欢什么等信息, 并且说明由于实际和其他一些原因, 新开发的系统刚开始与 MINIX 很象(并且使用了 MINIX 的文件系统)。并且已经成功地将 bash(1.08 版)和 gcc(1.40 版)移植到了新系统上, 而且在过几个月就可以实用了。

最后, Linus 声明他开发的操作系统没有使用一行 MINIX 的源代码; 而且由于使用了 386 的任务切换特性, 所以该操作系统不好移植(没有可移植性), 并且只能使用 AT 硬盘。对于 Linux 的可移植性问题, Linus 当时并没有考虑。但是目前 Linux 几乎可以运行在任何一种硬件体系结构上。

到了 1991 年的 10 月 5 日, Linus 在 comp.os.minix 新闻组上发布消息, 正式向外宣布 Linux 内核系统的诞生(Free minix-like kernel sources for 386-AT)。这段消息可以称为 Linux 的诞生宣言, 并且一直广为流传。因此 10 月 5 日对 Linux 社区来说是一个特殊的日子, 许多后来 Linux 的新版本发布时都选择了这个日子。所以 RedHat 公司选择这个日子发布它的新系统也不是偶然的。

### 1.1.6 Linux 操作系统版本的变迁

Linux 操作系统从诞生到 1.0 版正式出现, 共发布了表 1-1 中所示的一些主要版本。Linus 在 2003 年 9 月份开始学习使用版本管理工具 BitKeeper 时又把以上这些 1.0 之前的所有版本浏览了一遍。实际上, Linux 系统并没有 0.00 这个版本, 但是自从 Linus 在自己的 80386 兼容机上实验成功在时钟中断控制下两个任务相互切换运行时, 在某种程度上更增强了他开发自己操作系统的想法。因此我们也列为一个版本。Linux 0.01 版内核是于 1991 年 9 月 17 日编制完成。但是 Linus 还根本没有版权意识, 所以仅在该版的 include/string.h 文件中出现一次版权所有信息。该版本内核的键盘驱动程序仅硬编码进芬兰语代码, 因此也只支持芬兰键盘。也只支持 8MB 物理内存。由于 Linus 一次操作失误, 导致随后的 0.02、0.03 版内核源代码被破坏丢失。

表 1-1 内核的主要版本

版本号	发布/编制日期	说明
0.00	1991.2-4	两个进程, 分别在屏幕上显示'AAA...'和'BBB...'. (注: 没有发布)
0.01	1991.9.17	第一个正式向外公布的 Linux 内核版本。多线程文件系统、分段和分页内存管理。还不包含软盘驱动程序。
0.02	1991.10.5	该版本以及 0.03 版是内部版本, 目前已经无法找到。特点同上。
0.10	1991.10	由 Ted Ts'o 发布的 Linux 内核版本。增加了内存分配库函数。在 boot 目录中含有一个把 as86 汇编语法转换成 gas 汇编语法的脚本程序。
0.11	1991.12.8	基本可以正常运行的内核版本。支持硬盘和软驱设备以及串行通信。
0.12	1992.1.15	主要增加了数学协处理器的软件模拟程序。增加了作业控制、虚拟控制台、文件符号链接和虚拟内存对换(swapping)功能。
0.95.x (即 0.13)	1992.3.8	加入虚拟文件系统支持, 但还是只包含一个 MINIX 文件系统。增加了登录功能。改善了软盘驱动程序和文件系统的性能。改变了硬盘命名和编号方式。原

		命名方式与 MINIX 系统的相同，此时改成与现在 Linux 系统的相同。支持 CDROM。
0.96.x	1992.5.12	开始加入 UNIX Socket 支持。增加了 ext 文件系统 alpha 测试程序。SCSI 驱动程序被正式加入内核。软盘类型自动识别。改善了串行驱动、高速缓冲、内存管理的性能，支持动态链接库，并开始能运行 X-Windows 程序。原汇编语言编制的键盘驱动程序已用重新 C 重写。与 0.95 内核代码比较有很大的修改。
0.97.x	1992.8.1	增加了对新的 SCSI 驱动程序的支持；动态高速缓冲功能；msdos 和 ext 文件系统支持；总线鼠标驱动程序。内核被映射到线性地址 3GB 开始处。
0.98.x	1992.9.28	改善对 TCP/IP (0.8.1) 网络的支持，纠正了 extfs 的错误。重写了内存管理部分 (mm)，每个进程有 4GB 逻辑地址空间 (内核占用 1GB)。从 0.98.4 开始每个进程可同时打开 256 个文件 (原来是 32 个)，并且进程的内存堆栈独立使用一个内存页面。
0.99.x	1992.12.13	重新设计进程对内存的使用分配，每个进程有 4G 线性空间。不断地在改进网络代码。NFS 支持。
1.0	1994.3.14	推出第一个正式版。

现存的 0.10 版内核代码是 Ted Ts'o 当时保存下来的版本，Linus 自己的也已经丢失。这个版本要比前几个版本有很大的进步，在这个版本内核的系统上已经能够使用 GNU gcc 编译内核，并且开始支持加载/卸载 (mount/umount) 文件系统的操作。从这个内核版本开始，Linus 为每个文件都添加了版权信息：“(C) 1991 Linus Torvalds”。该版本的其他一些变化还包括：把原来引导程序 boot/boot.s 分割成 boot/bootsect.s 和 boot/setup.s 两个程序；①最多支持 16MB 物理内存；②为驱动程序和内存管理程序分别建立了自己的子目录；③增加了软盘驱动程序；④支持文件预读操作；⑤支持 dev/port 和 dev/null 设备；⑥重写了 kernel/signal.c 代码，添加了对 sigaction() 的支持等。

相对 0.10 版来说，Linux 0.11 版的改动较小。但这个版本也是第一个比较稳定的版本，并且开始有其他人员开始参与内核开发。这个版本中主要增添的功能有：①执行程序的需求加载；②启动时可执行 /etc/rc 初始文件；③建立起数学协处理器仿真程序框架程序结构；④Ted Ts'o 增加了对脚本程序的处理代码；⑤Galen Hunt 添加了对多种显示卡支持；⑥John T Kohl 修改了 kernel/console.c 程序，使控制台支持鸣叫功能和 KILL 字符；⑦提供了对多种语言键盘的支持。

Linux 0.12 是 Linus 比较满意的内核版本，也是一个更稳定的内核。在 1991 年的圣诞节期间，他编制完成了虚拟内存管理代码，从而可以在只有 2MB 内存的机器上也能使用象 gcc 这种“大型”软件。这个版本让 Linus 觉得发布 1.0 内核版本已经不是什么遥遥无期的事了，因此他立刻把下一个版本 (0.13 版) 提升为 0.95 版。Linus 这样做的另一个意思是让大家不要觉得离 1.0 版还很遥远。但是由于 0.95 版发布得太仓促，其中还包含较多错误，因此 0.95 版刚发布时曾有较多 Linux 爱好者在使用中遇到问题。当时 Linus 觉得就好象遇到了一个大灾难。不过此后他接受了这次的教训，以后每次发布新的内核版本时，他都会经过更周密的测试，并且让几个好朋友先试用后才会正式公布出来。0.12 版内核的主要变化之处有：①Ted Ts'o 添加了对终端信号处理支持；②启动时可以改变使用的屏幕行列值；③改正了一个文件 IO 引起的竞争条件；④增加了对共享库的支持，节省了内存使用量；⑤符号连接处理；⑥删除目录系统调用；⑦Peter MacDonald 实现了虚拟终端支持，使得 Linux 要比当时的某些商业版 UNIX 还要更胜一筹；⑧实现对 select() 函数支持，这是 Peter MacDonald 根据一些人为 MINIX 提供的补丁程序修改而成，但是 MINIX 却没有采纳这些补丁程序；⑨可重新执行的系统调用；⑩Linus 编制完成数学协处理器仿真代码等。

0.95 版是第一个使用 GNU GPL 版权的 Linux 内核版本。该版本实际上有 3 个子版本，由于 1992 年 3 月 8 日发布第 1 个 0.95 版时遇到了一些问题，因此此后不到 10 天 (3 月 17 日) 就立刻发布了另一个 0.95a 版，并在 1 个月后 (4 月 9 日) 又发布了 0.95c+ 版本。该版本的最大改进之处是开始采用虚拟文件



系统 VFS 结构。虽然当时仍然只支持 MINIX 文件系统，但是程序结构已经为支持多种文件系统进行了大范围调整。有关 MINIX 文件系统的代码被放进了单独一个 MINIX 子目录中。0.95 版内核的其他一些变化部分有：①增加了登录界面；②Ross Biro 添加了调试代码（ptrace）；③软盘驱动器磁道缓冲；④非阻塞管道文件操作；⑤系统重启（Ctrl-Alt-Del）；swapon() 系统调用，从而可以实时选择交换设备；⑥支持递归符号链接；⑦支持 4 个串行端口；⑧支持硬盘分区；⑨支持更多种类键盘；⑩James Wiegand 编制了最初的并行口驱动程序等。

另外，从 0.95 版开始，对内核的许多改进工作（提供补丁程序）均以其他人为主了，而 Linus 的主要任务开始变成对内核的维护和决定是否采用某个补丁程序。到现在为止，最新的内核版本是 2005 年 11 月 9 日公布的 2.6.14 版。其中包括大约 16000 个文件，使用 gz 压缩后源代码软件包也有 47MB 左右！各个主要稳定版本的最新版见表 1-2 所示。

表 1-2 新内核源代码字节数

内核版本号	发布日期	源代码大小(经 gz 压缩后)
2.0.40	2004.2.8	7.2 MB
2.2.26	2004.2.25	19 MB
2.4.31	2005.6.1	37 MB
2.6.14	2005.11.9	47 MB

### 1.1.7 Linux 名称的由来

Linux 操作系统刚开始时并没有被称作 Linux，Linus 给他的操作系统取名为 FREAX，其英文含义是怪诞的、怪物、异想天开等意思。在他将新的操作系统上载到 ftp.funet.fi 服务器上时，管理员 Ari Lemke 很不喜欢这个名称。他认为既然是 Linus 的操作系统就取其谐音 Linux 作为该操作系统的目录吧，于是 Linux 这个名称就开始流传下来。

在 Linus 的自传《Just for Fun》一书中，Linus 解释说<sup>2</sup>：

“坦白地说，我从来没有想到过要用 Linux 这个名称来发布这个操作系统，因为这个名字有些太自负了。而我为最终发布版准备的是什么呢？Freax。实际上，内核代码中某些早期的 Makefile - 用于描述如何编译源代码的文件 - 文件中就已经包含有“Freax”这个名字了，大约存在了半年左右。但其实这也没什么关系，在当时还不需要一个名字，因为我还没有向任何人发布过内核代码。”

“而 Ari Lemke，他坚持要用自己的方式将内核代码放到 ftp 站点上，并且非常不喜欢 Freax 这个名字。他坚持要用现在这个名字(Linux)，我承认当时我并没有跟他多争论。但这都是他取的名字。所以我可以光明正大地说我并不自负，或者部分坦白地说我并没有本位主义思想。但我想好吧，这也是个好名字，而且以后为这事我总能说服别人，就象我现在做的这样。”

### 1.1.8 早期 Linux 系统开发的主要贡献者

从 Linux 早期源代码中可以看出，Linux 系统的早期主要开发人员除了 Linus 本人以外，最著名的人员之一就是 Theodore Ts'o (Ted Ts'o)。他于 1990 年毕业于 MIT 计算机专业。在大学时代他就积极参加学校中举办的各种学生活动。他喜欢烹饪、骑自行车，当然还有就是 Hacking on Linux。后来他开始喜欢起业余无线电报运动。目前他在 IBM 工作从事系统编程及其他重要事务。他还是国际网络设计、操作、销售和研究者开放团体 IETF 成员。

Linux 在世界范围内的流行也有他很大的功劳。早在 Linux 操作系统刚问世时，他就怀着极大的热

<sup>2</sup> Linus Torvalds 《Just for fun》第 84-88 页。

情为 linux 的发展提供了 Maillist，几乎是在 Linux 刚开始发布时起，他就一直在为 Linux 做出贡献。他也是最早向 Linux 内核添加程序的人（Linux 内核 0.10 版中的虚拟盘驱动程序 `ramdisk.c` 和内核内存分配程序 `kmallocc.c`）。直到目前为止他仍然从事着与 Linux 有关的工作。在北美洲地区他最早设立了 Linux 的 ftp 站点（`tsx-11.mit.edu`），而且该站点至今仍然为广大 Linux 用户提供服务。他对 Linux 作出的最大贡献之一是提出并实现了 ext2 文件系统。该文件系统现已成为 Linux 世界中事实上的文件系统标准。最近他又推出了 ext3 文件系统。该系统大大提高了文件系统的稳定性和访问效率。作为对他的推崇，第 97 期（2002 年 5 月）的 Linux Journal 期刊将他作为了封面人物，并对他进行了采访。目前，他为 IBM Linux 技术中心工作，并从事着有关 Linux 标准规范 LSB(Linux Standard Base)等方面的工作。

Linux 社区中另一位著名人物是 Alan Cox。他原工作于英国威尔士斯旺西大学(Swansea University College)。刚开始他特别喜欢玩电脑游戏，尤其是 MUD（Multi-User Dungeon or Dimension，多用户网络游戏）。在 90 年代早期 `games.mud` 新闻组的 posts 中你可以找到他发表的大量帖子。他甚至为此还写了一篇 MUD 的发展史(`rec.games.mud` 新闻组，1992 年 3 月 9 日，A history of MUD)。

由于 MUD 游戏与网络密切相关，慢慢地他开始对计算机网络着迷起来。为了玩游戏并提高电脑运行游戏的速度以及网络传输速度，他需要选择一个最为满意的操作平台。于是他开始接触各种类型的操作系统。由于没钱，即使是 MINIX 系统他也买不起。当 Linux 0.11 和 386BSD 发布时，他考虑良久总算购置了一台 386SX 电脑。由于 386BSD 需要数学协处理器支持，而采用 Intel 386SX CPU 的电脑是不带数学协处理器的，所以他安装了 Linux 系统。于是他开始学习带有免费源代码的 Linux，并开始对 Linux 系统产生了兴趣，尤其是有关网络方面的实现。在关于 Linux 单用户运行模式问题的讨论中，他甚至赞叹 Linux 实现得巧妙(*beautifully*)。

Linux 0.95 版发布之后，他开始为 Linux 系统编写补丁程序（修改程序）（记得他最早的两个补丁程序，都没有被 Linus 采纳），并成为 Linux 系统上 TCP/IP 网络代码的最早使用人之一。后来他逐渐加入了 Linux 的开发队伍，并成为维护 Linux 内核源代码的主要负责人之一，也可以说成为 Linux 社团中继 Linus 之后最为重要的人物。以后 Microsoft 公司曾经邀请他加盟，但他却干脆地拒绝了。从 2001 年开始，他负责维护 Linux 内核 2.4.x 的代码。而 Linus 主要负责开发最新开发版内核的研制(奇数版，比如 2.5.x 版)。

《内核黑客手册》(The Linux Kernel Hackers' Guide)一书的作者 Michael K. Johnson 也是最早接触 Linux 操作系统的人之一(从 0.97 版)。他还是著名 Linux 文档计划(Linux Document Project - LDP)的发起者之一。曾经在 Linux Journal 杂志社工作，现在 RedHat 公司工作。

Linux 系统并不是仅有这些中坚力量就能发展成今天这个样子的，还有许多计算机高手对 Linux 做出了极大的贡献，这里就不一一列举了。主要贡献者的具体名单可参见 Linux 内核中的 CREDITS 文件，其中以字母顺序列出了对 Linux 做出较大贡献的近 400 人的名单列表，包括他们的 email 地址和通信地址、主页以及主要贡献事迹等信息。

通过上述说明，我们可以对上述 Linux 的五大支柱归纳如下：

UNIX 操作系统 -- UNIX 于 1969 年诞生在 Bell 实验室。Linux 就是 UNIX 的一种克隆系统。UNIX 的重要性就不用多说了。

MINIX 操作系统 -- MINIX 操作系统也是 UNIX 的一种克隆系统，它于 1987 年由著名计算机教授 Andrew S. Tanenbaum 开发完成。由于 MINIX 系统的出现并且提供源代码(只能免费用于大学内)在全世界的大学中刮起了学习 UNIX 系统旋风。Linux 刚开始就是参照 MINIX 系统于 1991 年才开始开发。

GNU 计划-- 开发 Linux 操作系统，以及 Linux 上所用大多数软件基本上都出自 GNU 计划。Linux 只是操作系统的一个内核，没有 GNU 软件环境(比如说 `bash shell`)，则 Linux 将寸步难行。

POSIX 标准 -- 该标准在推动 Linux 操作系统以后朝着正规路上发展起着重要的作用。是 Linux 前

进的灯塔。

INTERNET -- 如果没有 Internet 网，没有遍布全世界的无数计算机黑客的无私奉献，那么 Linux 最多只能发展到 0.13(0.95)版的水平。

## 1.2 内容综述

本文将主要对 Linux 的早期内核 0.11 版进行详细描述和注释。Linux-0.11 版本是在 1991 年 12 月 8 日发布的。在发布时包括以下文件：

---

bootimage.Z	- 具有美国键盘代码的压缩启动映像文件；
rootimage.Z	- 以 1200kB 压缩的根文件系统映像文件；
linux-0.11.tar.Z	- 内核源代码文件。大小为 94KB，展开后也仅有 325KB；
as86.tar.Z	- Bruce Evans' 二进制执行文件。是 16 位的汇编程序和装入程序；
INSTALL-0.11	- 更新过的安装信息文件。

---

bootimage.Z 和 rootimage.Z 是压缩的软盘映像（Image）文件。bootimage 是引导启动 Image 文件，其中主要包括磁盘引导扇区代码、操作系统加载程序和内核执行代码。PC 机启动时 ROM BIOS 中的程序会把默认启动驱动器上的引导扇区代码和数据读入内存，而引导扇区代码则负责把操作系统加载程序和内核执行代码读入内存中，然后把控制权交给操作系统加载程序去进一步准备内核的初始化操作，最终加载程序会把控制权交给内核代码。内核代码若要正常运行就需要文件系统的支持。rootimage 就是用于向内核提供最基本支持的根文件系统，其中包括操作系统最起码的一些配置文件和命令执行程序。对于 Linux 系统中使用的 UNIX 类文件系统，其中主要包括一些规定的目录、配置文件、设备驱动程序、开发程序以及所有其他用户数据或文本文件等。这两个盘合起来就相当于一张可启动的 DOS 操作系统盘。

as86.tar.Z 是 16 位汇编器链接程序软件包。linux-0.11.tar.Z 是压缩的 Linux 0.11 内核源代码。INSTALL-0.11 是 Linux 0.11 系统的简单安装说明文档。

目前除了原来的 rootimage.Z 文件，其他四个文件均能找到。不过作者已经利用 Internet 上的资源为 Linux 0.11 重新制作出了一个完全可以使用的 rootimage-0.11 根文件系统。并重新为其编译出能在 0.11 环境下使用的 gcc 1.40 编译器，配置出可用的实验开发环境。目前，这些文件均可以从 oldlinux.org 网站上下载。具体下载目录位置是：

- <http://oldlinux.org/Linux.old/images/> 该目录中含有已经制作好的内核映像文件 bootimage 和根文件系统映像文件 rootimage。
- <http://oldlinux.org/Linux.old/kernels/> 该目录中含有内核源代码程序，包括本书所描述的 Linux 0.11 内核源代码程序。
- <http://oldlinux.org/Linux.old/bochs/> 该目录中含有已经设置好的运行在计算机仿真系统 bochs 下的 Linux 系统。
- <http://oldlinux.org/Linux.old/Linux-0.11/> 该目录中含有可以在 Linux 0.11 系统中使用的其他一些工具程序和原来发布的一些安装说明说明文档。

本文主要详细分析 linux-0.11 内核中的所有源代码程序，对每个源程序文件都进行了详细注释，包括对 Makefile 文件的注释。分析过程主要是按照计算机启动过程进行的。因此分析的连贯性到初始化结束内核开始调用 shell 程序为止。其余的各个程序均针对其自身进行分析，没有连贯性，因此可以根据自己的需要进行阅读。但在分析时还是提供了一些应用实例。










所有的程序在分析过程中如果遇到作者认为是较难理解的语句时，将给出相关知识的详细介绍。比如，在阅读代码头一次遇到 C 语言内嵌汇编码时，将对 GNU C 语言的内嵌汇编语言进行较为详细的介绍；在遇到对中断控制器进行输入/输出操作时，将对 Intel 中断控制器（8259A）芯片给出详细的说明，并列出的命令和方法。这样做有助于加深对代码的理解，又能更好的了解所用硬件的使用方法，作者认为这种解读方法要比单独列出一章内容来总体介绍硬件或其他知识要效率高得多。

拿 Linux 0.11 版内核来“开刀”是为了提高我们认识 Linux 运行机理的效率。Linux-0.11 版整个内核源代码只有 325K 字节左右，其中包括的内容基本上都是 Linux 的精髓。而目前最新的 2.6.XX 版内核非常大，有 200 兆字节，即使你花一生的经历来阅读也未必能全部都看完。也许你要问“既然要从简入手，为什么不分析更小的 Linux 0.01 版内核源代码呢？它只有 240K 字节左右”主要原因是因为 0.01 版的内核代码有太多的不足之处，甚至还没有包括对软盘的驱动程序，也没有很好地涉及数学协处理器的使用以及对登陆程序的说明。并且其引导启动程序的结构也与目前的版本不太一样，而 0.11 版的引导启动程序结构则与现在的基本上是一样的。另外一个原因是可以找到 0.11 版早期的已经编译制作好的内核映像文件(bootimage)，可以用来进行引导演示。如果再配上简单的根文件系统映像文件(rootimage)，那么它就可以进行正常的运行了。

拿 Linux 0.11 版进行学习也有不足之处。比如该内核版本中尚不包括有关专门的进程等待队列、TCP/IP 网络等方面的一些当前非常重要的代码，对内存的分配和使用与现今的内核也有所区别。但好在 Linux 中的网络代码基本上是自成一体的，与内核机制关系不是非常大，因此可以在了解了 Linux 工作的基本原理之后再分析这些代码。

本文对 Linux 内核中所有的代码都进行了说明。为了保持结构的完整性，对代码的说明是以内核中源代码的组成结构来进行的，基本上是以每个源代码中的目录为一章内容进行介绍。介绍的源程序文件的次序可参见前面的文件列表索引。整个 Linux 内核源代码的目录结构如下列表 1.1 所示。所有目录结构均是以 linux 为当前目录。

列表 1-1 Linux/目录

名称	大小	最后修改日期(GMT)	说明
 boot/		1991-12-05 22:48:49	
 fs/		1991-12-08 14:08:27	
 include/		1991-09-22 19:58:04	
 init/		1991-12-05 19:59:04	
 kernel/		1991-12-08 14:08:00	
 lib/		1991-12-08 14:10:00	
 mm/		1991-12-08 14:08:21	
 tools/		1991-12-04 13:11:56	
 Makefile	2887 bytes	1991-12-06 03:12:46	

本书内容可以分为五个部分。第 1 章至第 4 章是基础知识部分。操作系统与所运行的硬件环境密切相关。如果想彻底理解操作系统运行全过程，那么就需要了解它的硬件运行环境，尤其是处理器多任务运行机制。这部分较为详细地介绍了微型计算机硬件组成、编制 Linux 内核程序使用的编程语言以及 Intel 80X86 保护模式下的编程原理；第二部分包括第 5 章至第 7 章，描述内核引导启动和 32 位运行方式的准备阶段，作为学习内核的初学者应该全部进行阅读；第三部分从第 8 章到第 13 章是内核代码的主要部

分。其中第 8 章内容可以作为阅读这部分后续章节的主要线索来进行。第 14 章到第 16 章是第四部分内容，可以作为阅读第三部分源代码的参考信息。最后一部分仅包括第 17 章内容，其中介绍了如何使用 PC 机模拟软件系统 Bochs 针对 Linux 0.11 内核进行各种实验活动。

第 2 章首先基于传统微机系统的硬件组成框图，主要介绍 Linux 内核运行之上的 IBM PC/AT386 微机的组成部分。介绍各个主要部分的功能和相互关系。同时也与目前最新微机的组成框图作简单比较。这样能够为那些没有学过计算机组成原理的读者提供足够的有关信息。

第 3 章介绍 Linux 0.11 内核中使用的编程语言、目标文件格式和编译环境，主要目标是提供阅读 Linux 0.11 内核源代码所需要的汇编语言和 GNU C 语言扩展知识。本章首先比较详细地介绍了 as86 和 GNU as 汇编程序的语法和使用方法，然后对 GNU C 语言中的内联汇编、语句表达式、寄存器变量以及内联函数等常用 C 语言扩展内容进行说明，同时详细描述了 C 和汇编函数之间的相互调用机制。最后简单描述了 Makefile 文件的使用方法。

第 4 章主要概要描述 80X86 CPU 的体系结构以及保护模式下编程的一些基础知识，为准备阅读基于 80X86 CPU 的 Linux 内核源代码打下坚实基础。其中主要包括：80X86 基础知识、保护模式内存管理、中断和异常处理、任务管理以及一个简单的多任务内核示例。

第 5 章概要地描述了 Linux 操作系统的体系结构、内核源代码文件放置的组织结构以及每个文件大致功能。还介绍了 Linux 对物理内存的使用分配方式、内核的几种堆栈及其使用方式和虚拟线性地址的使用分配。最后开始注释内核程序包中 Linux/目录下的所看到的第一个文件，也即内核代码的总体 Makefile 文件的内容。该文件是所有内核源程序的编译管理配置文件，供编译管理工具软件 make 使用。

第 6 章将详细注释 boot/目录下的三个汇编程序，其中包括磁盘引导程序 bootsect.s、获取 BIOS 中参数的 setup.s 汇编程序和 32 位运行启动代码程序 head.s。这三个汇编程序完成了把内核从块设备上引导加载到内存的工作，并对系统配置参数进行探测，完成了进入 32 位保护模式运行之前的所有工作。为内核系统执行进一步的初始化工作做好了准备。

第 7 章主要介绍 init/目录中内核系统的初始化程序 main.c。它是内核完成所有初始化工作并进入正常运行的关键地方。在完成了系统所有的初始化工作后，创建了用于 shell 的进程。在介绍该程序时将需要查看其所调用的其他程序，因此对后续章节的阅读可以按照这里调用的顺序进行。由于内存管理程序的函数在内核中被广泛使用，因此该章内容应该最先选读。当你能真正看懂直到 main.c 程序为止的所有程序时，你应该已经对 Linux 内核有了一定的了解，可以说已经有一半入门了☺，但你还需要对文件系统、系统调用、各种驱动程序等进行更深一步的阅读。

第 8 章主要介绍 kernel/目录中的所有程序。其中最重要的部分是进程调度函数 schedule()、sleep\_on() 函数和有关系统调用的程序。此时你应该已经对其中的一些重要程序有所了解。从本章内容开始，我们会遇到很多 C 语言程序中嵌入的汇编语句。有关嵌入式汇编语句的基本语法请参见第 3 章的说明。

第 9 章对 kernel/blk\_drv/目录中的块设备程序进行了注释说明。该章主要含有硬盘、软盘等块设备的驱动程序，主要用来与文件系统和高速缓冲区打交道，含有较多与硬件相关的内容。因此，在阅读这章内容时需参考一些硬件资料。最好能首先浏览一下文件系统的章节。

第 10 章对 kernel/chr\_drv/目录中的字符设备驱动程序进行注释说明。这一章中主要涉及串行线路驱动程序、键盘驱动程序和显示器驱动程序。这些驱动程序构成了 0.11 内核支持的串行终端和控制台终端设备。因此本章也含有较多与硬件有关的内容。在阅读时需要参考一下相关硬件的书籍。

第 11 章介绍 kernel/math/目录中的数学协处理器的仿真程序。由于本书所注释的内核版本，还没有真正开始支持协处理器，因此本章的内容较少，也比较简单。只需有一般性的了解即可。

第 12 章介绍内核源代码 fs/目录中的文件系统程序，在看这章内容时建议你能够暂停一下而去阅读 Andrew S. Tanenbaum 的《操作系统设计与实现》一书中有关 MINIX 文件系统的章节，因为最初的 Linux 系统是只支持 MINIX 一种文件系统，Linux 0.11 版也不例外。

第 13 章解说 mm/目录中的内存管理程序。要透彻地理解这方面的内容，我们就需要对 Intel 80X86 微处理器的保护模式运行方式有足够的理解，因此在阅读本章程序时，除了可以参考本章在适当地方包

含的 80X86 保护模式运行方式概要说明以外,还应该同时参考第 4 章内容。由于本章以源代码中的运用实例作为对象进行解说,因此可以更好地理解内存管理的工作原理。

现有的 Linux 内核分析书籍一般都缺乏对内核头文件的描述,因此对于一个初学者来讲,在阅读内核程序时会碰到许多障碍。本书第 14 章对 include/目录中的所有头文件进行了详细说明,基本上对每一个定义、每一个常量或数据结构都进行了详细注释。为了便于在阅读时参考查阅,本书在附录中还对一些经常要用到的重要的数据结构和变量进行了归纳注释,但这些内容实际上都能在这一章的头文件中找到。虽然该章内容主要是为阅读其他章节中的程序作参考使用的,但是若想彻底理解内核的运行机制,仍然需要了解这些头文件中的许多细节。

第 15 章介绍了 Linux 0.11 版内核源代码 lib/目录中的所有文件。这些库函数文件主要向编译系统等系统程序提供了接口函数,对以后理解系统软件会有较大的帮助。由于这个版本较低,所以这里的内容并不是很多,因此我们可以很快地看完。这也是我们为什么选择 0.11 版的原因之一。

第 16 章介绍 tools/目录下的 build.c 程序。这个程序并不会包括在编译生成的内核映像(Image)文件中,它仅用于将内核中的磁盘引导程序块与其他主要内核模块连接成一个完整的内核映像(kernel image)文件。

第 17 章介绍了学习内核源代码时的实验环境以及动手实施各种实验的方法。主要介绍了在 Bochs 仿真系统下使用和编译 Linux 内核的方法以及磁盘镜像文件的制作方法。还说明了如何修改 Linux 0.11 源代码的语法使其能在 RedHat 9 系统下顺利编译出正确的内核来。

最后是附录和索引。附录中给出了 Linux 内核中的一些常数定义和基本数据结构定义,以及保护模式运行机制的简明描述。

为了便于查阅,在本书的附录中还单独列出了内核中要用到的有关 PC 机硬件方面的信息。在参考文献中,我们仅给出了在阅读源代码时可以参考的书籍、文章等信息,并没有包罗万象地给出一大堆的繁杂凌乱的文献列表。比如在引用 Linux 文档项目 LDP(Linux Document Project)中的文件时,我们会明确地列出具体需要参考哪一篇 HOWTO 文章,而并不是仅仅给出 LDP 的网站地址了事。

Linus 在最初开发 Linux 操作系统内核时,主要参考了 3 本书。一本是 M. J. Bach 著的《UNIX 操作系统设计》,该书描述了 UNIX System V 内核的工作原理和数据结构。Linus 使用了该书中很多函数的算法, Linux 内核源代码中很多重要函数的名称都取自该书。因此,在阅读本书时,这是一本必不可少的内核工作原理方面的参考书籍。另一本是 John H. Crawford 等编著的《Programming the 80386》,是讲解 80x86 下保护模式编程方法的好书。还有一本就是 Andrew S.Tanenbaum 著的《MINIX 操作系统设计与实现》一书的第 1 版。Linus 主要使用了该书中描述的 MINIX 文件系统 1.0 版,而且在早期的 Linux 内核中也仅支持该文件系统,所以在阅读本书有关文件系统一章内容时,文件系统的工作原理方面的知识完全可以从 Tanenbaum 的书中获得。

在对每个程序进行解说时,我们首先简单说明程序的主要用途和目的、输入输出参数以及与其他程序的关系,然后列出程序的完整代码并在其中对代码进行详细注释,注释时对原程序代码或文字不作任何方面的改动或删除,因为 C 语言是一种英语类语言,程序中原有的少量英文注释对常数符号、变量名等也提供了不少有用的信息。在代码之后是对程序更为深入的解剖,并对代码中出现的一些语言或硬件方面的相关知识进行说明。如果在看完这些信息后回头再浏览一遍程序,你会有更深一层的体会。

对于阅读本书所需要的一些基本概念知识的介绍都散布在各个章节相应的地方,这样做主要是为了能够方便的找到,而且在结合源代码阅读时,对一些基本概念能有更深的理解。

最后要说明的是当你已经完全理解了本文所解说的一切时,并不代表你已经成为一个 Linux 行家了,你只是刚刚踏上 Linux 的征途,具有了一定的成为一个 Linux 内核高手的初步知识。这时你应该去阅读更多的源代码,最好是循序渐进地从 1.0 版本开始直到最新的正在开发中的奇数编号的版本。在撰写这本书时最新的 Linux 内核是 2.6.12 版。当你能快速理解这些开发中的最新版本甚至能提出自己的建议和

补丁（patch）程序时，我也甘拜下风了☺。

## 1.3 本章小结

首先阐述了 Linux 诞生和发展不可缺少的五个支柱：UNIX 最初的开放源代码版本为 Linux 提供了实现的基本原理和算法、Richard Stallman 的 GNU 计划为 Linux 系统提供了丰富且免费的各种实用工具、POSIX 标准的出现为 Linux 提供了实现与标准兼容系统的参考指南、A.S.T 的 MINIX 操作系统为 Linux 的诞生起到了不可忽缺的参考、Internet 是 Linux 成长和壮大的必要环境。最后本章概述了书中的基本内容。



## 第2章 微型计算机组成结构

任何一个系统都可认为由四个基本部分组成，见图 2-1 所示的模型。其中输入部分用于接收进入系统的信息或数据；经过处理中心加工后再由输出部分送出。能源部分为整个系统提供操作运行的能源供给，包括输入和输出部分操作所需要的能量。

计算机系统也不例外，它也主要由这四部分组成。不过在内部，计算机系统的处理中心与输入/输出部分之间的通道或接口都是共享使用的，因此图 2-1 中(b)应该更能恰当地抽象表示一个计算机系统。当然，象计算机或很多复杂系统来说，其中各个子部分都可以独立地看作一个完整的子系统，并且也能使用这个模型来描述，而一个完整的计算机系统整体则由这些子系统构成。

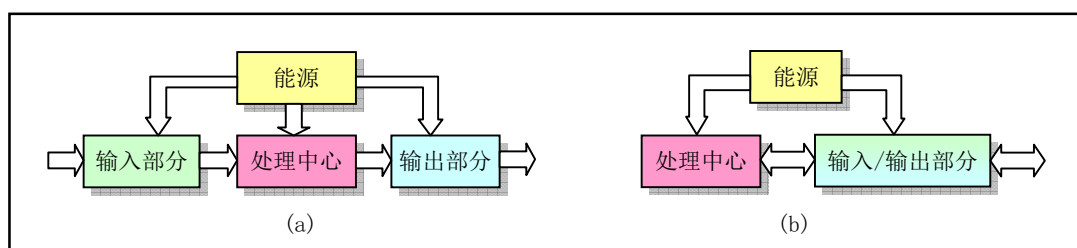


图 2-1 系统基本组成

计算机系统可分为硬件部分和软件部分，但两者之间互相依存。硬件部分是计算机系统的可见部分，是软件运行和存储的平台。软件是一种控制硬件操作和动作的指令流。犹如存储于人类大脑中的信息和思维控制着人体的思想和动作一样，软件可以看作是计算机“大脑”中的信息和思维。本书描述的主题就是一个计算机系统的运行机制，主要说明系统的处理中心和输入/输出部分的硬件组成原理和软件控制的实现。在硬件方面，我们概要说明基于 Intel 80X86 中央处理器（CPU -- Central Processing Unit）的 IBM PC 微型计算机及其兼容机的硬件系统，计算机的 CPU 芯片可以直接看作是系统的处理中心，它通过总线接口与其他部分相连；而对于运行在其上的软件方面，我们则专门详细描述 Linux 操作系统内核的实现。

可见，操作系统与所运行的硬件环境密切相关。如果想彻底理解操作系统运行全过程，那么就需要了解它的运行硬件环境。本章基于传统微机系统的硬件组成框图，介绍了微机中各个主要部分的功能。这些内容已基本能够建立起阅读 Linux 0.11 内核的硬件基础。为了便于说明，术语 PC/AT 将用来指示具有 80386 或以上 CPU 的 IBM PC 及其兼容微机，而 PC 则用来泛指所有微机，包括 IBM PC/XT 及其兼容微机。

### 2.1 微型计算机组成原理

我们从俯瞰的角度来说明采用 80386 或以上 CPU 的 PC 机系统组成结构。一个传统微型计算机硬件组成结构见图 2-2 所示。其中，CPU 通过地址线、数据线和控制信号线组成的本地总线（或称为内部总线）与系统其他部分进行数据通信。地址线用于提供内存或 I/O 设备的地址，即指明需要读/写数据的具体位置。数据线用于在 CPU 和内存或 I/O 设备之间提供数据传输的通道，而控制线则负责指挥执行的具体读/写操作。对于使用 80386 CPU 的 PC 机，其内部地址线和数据线都分别有 32 根，即都是 32 位的。因此地址寻址空间范围有  $2^{32}$  字节，从 0 到 4GB。

图中上部控制器和存储器接口通常都集成在计算机主板上，这些控制器分别都是以一块大规模集成电路芯片为主组成的功能电路。例如，中断控制器由 Intel 8259A 或其兼容芯片构成；DMA 控制器通常采用



Intel 8237A 芯片构成；定时计数器的核心则是 Intel 8253/8254 定时芯片；键盘控制器使用的是 Intel 8042 芯片来与键盘中的扫描电路进行通信。

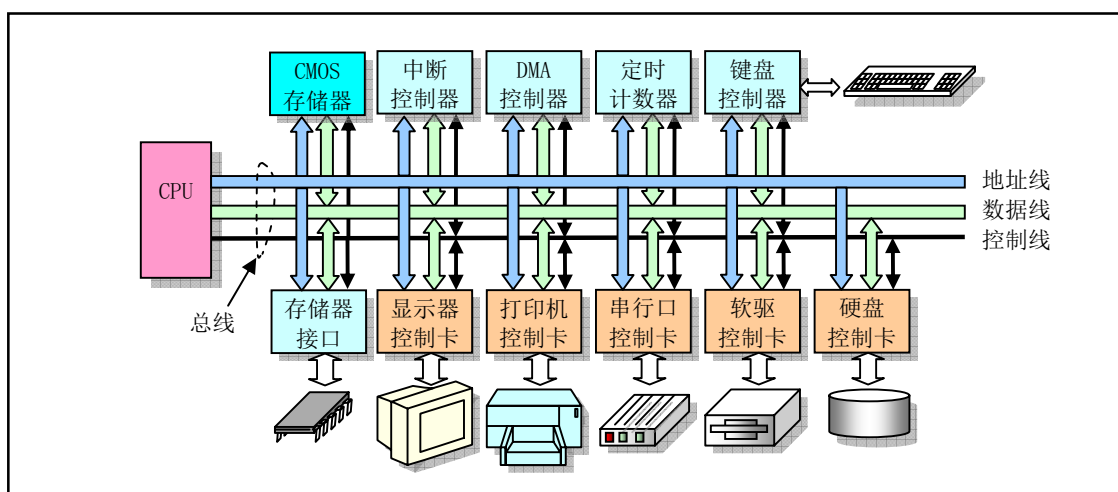


图 2-2 传统 IBM PC 及其兼容计算机的组成框图

图中下方的控制卡（或者称为适配器）则是通过扩展插槽与主板上系统总线连接。总线插槽是系统地址总线、数据总线和控制线的与扩展设备控制器的标准连接接口。这些总线接口标准通常有工业标准结构 ISA（Industry Standard Architecture）总线、扩展工业标准结构总线 EISA（Extended ISA）、外围组件互连 PCI（Peripheral Component Interconnect）总线、加速图形端口 AGP（Accelerated Graphics Port）视频总线等。这些总线接口的主要区别在于数据传输速率和控制灵活性方面。随着计算机硬件的发展，传输速率更高、控制更灵活的总线接口还在不断推出，例如采用串行通信点对点技术的高速 PCIE（PCI Express）总线。最初的 80386 机器上只有 ISA 总线，因此系统与外部 I/O 设备最多只能使用 16 位数据线进行数据传输。

随着计算机技术的发展，很多原来使用控制卡来完成的功能（例如硬盘控制器功能）都已经集成在计算机主机板上少数几个超大规模集成电路芯片中，几个甚至是一个这样的芯片就确定了主机板的主要特性和功能，并且为了让系统的不同部分都能达到其最高传输速率，总线结构也发生了很大变化。现代 PC 机的组成结构通常可以使用图 2-3 来描述。除了 CPU 以外，现代 PC 机主板主要使用 2 个超大规模芯片构成的芯片组或芯片集（Chipsets）组成：北桥（Northbridge）芯片和南桥（Southbridge）芯片。北桥芯片用于与 CPU、内存和 AGP 视频接口，这些接口具有很高的传输速率。北桥芯片还起着存储器控制作用，因此 Intel 把该芯片标号为 MCH（Memory Controller Hub）芯片。南桥芯片用来管理低、中速的组件，例如，PCI 总线、IDE 硬盘接口、USB 端口等，因此南桥芯片的名称为 ICH（I/O Controller Hub）。之所以用“南、北”桥来分别统称这两个芯片，是由于在 Intel 公司公布的典型 PC 机主板上，它们分别位于主板的下端和上端（即地图上的南部和北部）位置，并起着与 CPU 进行通道桥接的作用。

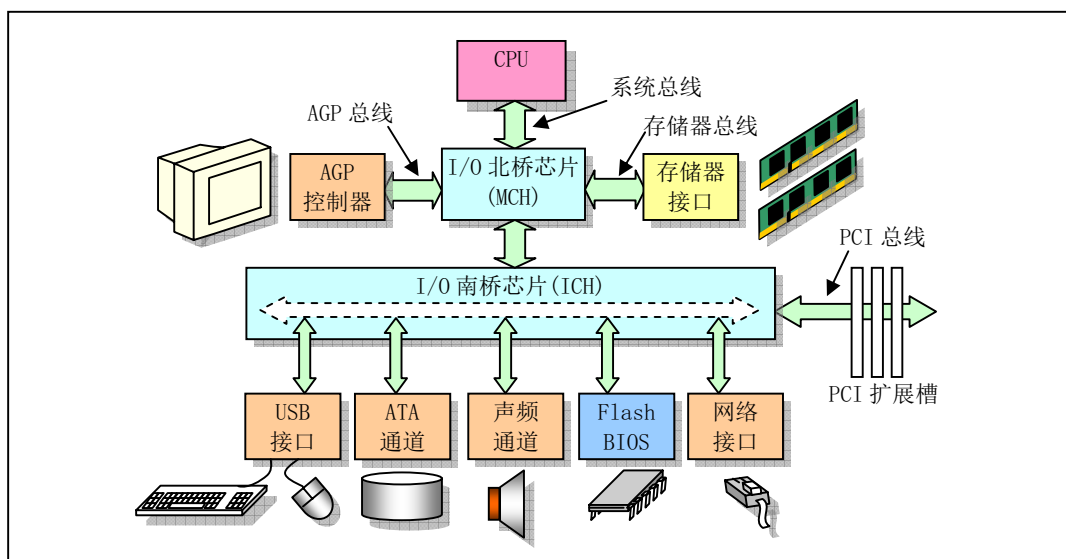


图 2-3 现代 PC 机芯片集框图

虽然总线接口发生了很大变化，甚至今后北桥和南桥芯片都将会合二为一，但是对于我们编程人员来说，这些变化仍然与传统的 PC 机结构兼容。因此为传统 PC 机硬件结构编制的程序仍然能运行于现在的 PC 机上。这从 Intel 的开发手册上可以证实这个结论。所以为了便于入门学习，我们仍然以传统 PC 机结构为框架来讨论和学习 PC 的组成和编程方法，当然这些方法仍然适合于现代 PC 机结构。下面我们概要说明图 2-2 中各个主要控制器和控制卡的工作原理，而它们的实际编程方法则推迟到阅读内核相应源代码时再作详细介绍。

## 2.2 I/O 端口寻址和访问控制方式

### 2.2.1 I/O 端口和寻址

CPU 为了访问 I/O 接口控制器或控制卡上的数据和状态信息，需要首先指定它们的地址。这种地址就称为 I/O 端口地址或者简称端口。通常一个 I/O 控制器包含访问数据的数据端口、输出命令的命令端口和访问控制器执行状态的状态端口。端口地址的设置方法一般有两种：统一编址和独立编址。

端口统一编址的原理是把 I/O 控制器中的端口地址归入存储器寻址地址空间范围内。因此这种编址方式也成为存储器映像编址。CPU 访问一个端口的操作与访问内存的操作一样，也使用访问内存的指令。端口独立编址的方法是把 I/O 控制器和控制卡的寻址空间单独作为一个独立的地址空间对待，称为 I/O 地址空间。每个端口有一个 I/O 地址与之对应，并且使用专门的 I/O 指令来访问端口。

IBM PC 及其兼容微机主要使用独立编址方式，采用了一个独立的 I/O 地址空间对控制设备中的寄存器进行寻址和访问。使用 ISA 总线结构的传统 PC 机其 I/O 地址空间范围是 0x000 -- 0x3FF，有 1024 个 I/O 端口地址可供使用。各个控制器和控制卡所默认分配使用的端口地址范围见表 2-1 所示。关于这些端口的使用和编程方法将在后面具体涉及相关硬件时再详细进行说明。

另外，IBM PC 机也部分地使用了统一编址方式。例如，CGA 显示卡上显示内存的地址就直接占用了存储器地址空间 0xB800 -- 0xBC00 范围。因此若要让一个字符显示在屏幕上，可以直接使用内存操作指令往这个内存区域执行写操作。

表 2-1 I/O 端口地址分配

端口地址范围	分配说明
0x000 -- 0x01F	8237A DMA 控制器 1

0x020 -- 0x03F	8259A 可编程中断控制器 1
0x040 -- 0x05F	8253/8254A 定时计数器
0x060 -- 0x06F	8042 键盘控制器
0x070 -- 0x07F	访问 CMOS RAM/实时时钟 RTC (Real Time Clock) 端口
0x080 -- 0x09F	DMA 页面寄存器访问端口
0x0A0 -- 0x0BF	8259A 可编程中断控制器 2
0x0C0 -- 0x0DF	8237A DMA 控制器 2
0x0F0 -- 0x0FF	协处理器访问端口
0x170 -- 0x177	IDE 硬盘控制器 1
0x1F0 -- 0x1F7	IDE 硬盘控制器 0
0x278 -- 0x27F	并行打印机端口 2
0x2F8 -- 0x2FF	串行控制器 2
0x378 -- 0x37F	并行打印机端口 1
0x3B0 -- 0x3BF	单色 MDA 显示控制器
0x3C0 -- 0x3CF	彩色 CGA 显示控制器
0x3D0 -- 0x3DF	彩色 EGA/VGA 显示控制器
0x3F0 -- 0x3F7	软盘控制器
0x3F8 -- 0x3FF	串行控制器 1

对于使用 EISA 或 PCI 等总线结构的现代 PC 机, 有 64KB 的 I/O 地址空间可供使用。在普通 Linux 系统下通过查看 /proc/ioprots 文件可以得到相关控制器或设置使用的 I/O 地址范围, 见如下所示。

```
[root@plinux root]# cat /proc/ioprots
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 接口访问控制

PC 机 I/O 接口数据传输控制方式一般可采用程序循环查询方式、中断处理方式和 DMA 传输方式。顾名思义，循环查询方式是指 CPU 通过在程序中循环查询指定设备控制器中的状态来判断是否可以与设备进行数据交换。这种方式不需要过多硬件支持，使用和编程都比较简单，但是特别耗费 CPU 宝贵时间。因此在多任务操作系统中除非等待时间极短或必须，否则就不应该使用这种方式。在 Linux 操作系统中，只有在设备或控制器能够立刻返回信息时才会很少的几个地方采用这种方式。

中断处理控制方式需要有中断控制器的支持。在这种控制方式下，只有当 I/O 设备通过中断向 CPU 提出处理请求时，CPU 才会暂时中断当前执行的程序转而去执行相应的 I/O 中断处理服务过程。当执行完该中断处理服务过程后，CPU 又会继续执行刚才被中断的程序。在 I/O 控制器或设备发出中断请求时，CPU 通过使用中断向量表（或中断描述符表）来寻址相应的中断处理服务过程的入口地址。因此采用中断控制方式时需要首先设置好中断向量表，并编制好相应的中断处理服务过程。Linux 操作系统中大多数设备 I/O 控制都采用中断处理方式。

直接存储器访问 DMA（Direct Memory Access）方式用于 I/O 设备与系统内存之间进行批量数据传送，整个操作过程需要使用专门的 DMA 控制器来进行而无需 CPU 插手。由于在传输过程中无须软件介入，因此操作效率很高。在 Linux 操作系统中，软盘驱动程序使用中断和 DMA 方式配合来实现数据的传输工作。

## 2.3 主存储器、BIOS 和 CMOS 存储器

### 2.3.1 主存储器

1981 年 IBM PC 机刚推出时系统只带有 640KB 的 RAM 主存储器（简称内存）。由于所采用的 8088/8086 CPU 只有 20 根地址线，因此内存寻址范围最高为 1024KB（1MB）。在当时 DOS 操作系统流行年代，640K 或 1MB 内存容量基本上能满足普通应用程序的运行。随着计算机软件和硬件技术的高速发展，目前的计算机通常都配置有 512MB 或者更多的物理内存容量，并且都采用 Intel 32 位 CPU，即都是 PC/AT 计算机。因此 CPU 的物理内存寻址范围已经高达 4GB（通过采用 CPU 的新特性，系统甚至可以寻址 64GB 的物理内存容量）。但是为了与原来的 PC 机在软件上兼容，系统 1MB 以下物理内存使用分配上仍然保持与原来的 PC 机基本一致，只是原来系统 ROM 中的基本输入输出程序 BIOS 一直处于 CPU 能寻址的内存最高端位置处，而 BIOS 原来所在的位置将在计算机开机初始化时被用作 BIOS 的影子（Shadow）区域，即 BIOS 代码仍然会被复制到这个区域中。见图 2-4 所示。

当计算机上电初始化时，物理内存被设置成从地址 0 开始的连续区域。除了地址从 0xA0000 到 0xFFFFF（640K 到 1M 共 384K）和 0xFFFFE0000 到 0xFFFFFFFF（4G 处的最后一 64K）范围以外的所有内存都可用作系统内存。这两个特定范围被用于 I/O 设备和 BIOS 程序。假如我们的计算机中有 16MB 的物理内存，那么在 Linux 0.1x 系统中，0--640K 将被用作存放内核代码和数据。Linux 内核不使用 BIOS 功能，也不使用 BIOS 设置的中断向量表。640K--1M 之间的 384K 仍然保留用作图中指明的用途。其中地址 0xA0000 开始的 128K 用作显示内存缓冲区，随后部分用于其他控制卡的 ROM BIOS 或其映射区域，而 0xF0000 到 1M 范围用于高端系统 ROM BIOS 的映射区。1M--16M 将被内核用于作为可分配的主内存区。另外高速缓冲区和内存虚拟盘也会占用内核代码和数据后面的一部分内存区域，该区域通常会跨越 640K -- 1M 的区

域。

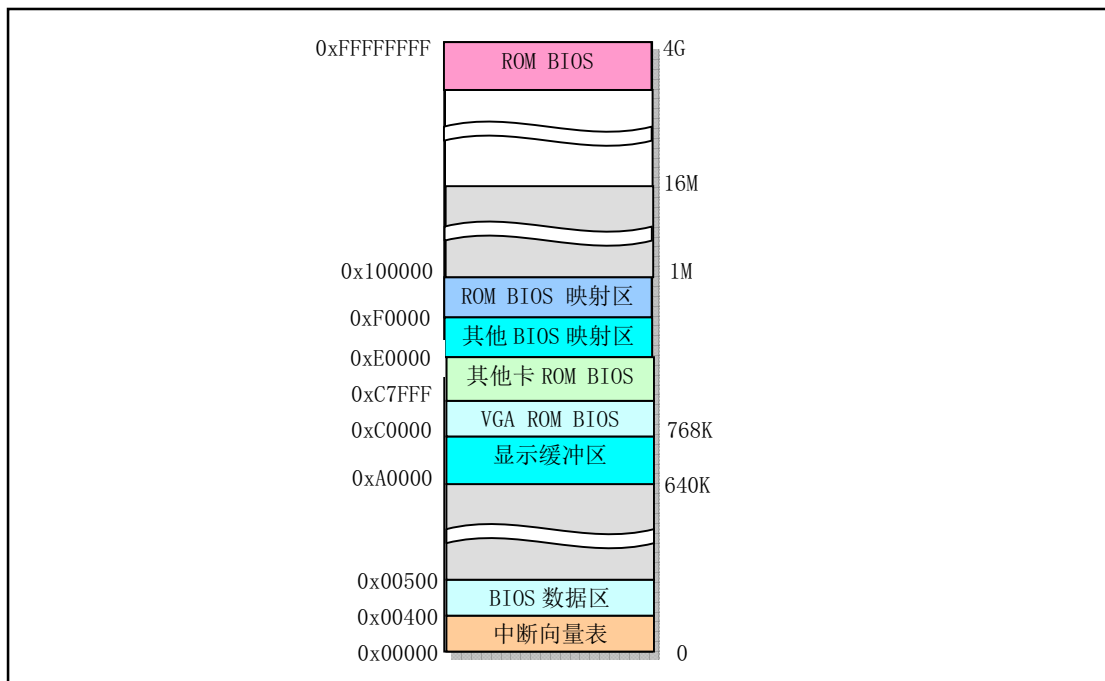


图 2-4 PC/AT 机内存使用区域图

### 2.3.2 基本输入/输出程序 BIOS

存放在 ROM 中的系统 BIOS 程序主要用于计算机开机时执行系统各部分的自检，建立起操作系统需要使用的各种配置表，例如中断向量表、硬盘参数表。并且把处理器和系统其余部分初始化到一个已知状态，而且还为 DOS 等操作系统提供硬件设备接口服务。但是由于 BIOS 提供的这些服务不具备可重入性（即其中程序不可并发运行），并且从访问效率方面考虑，因此除了在初始化时会利用 BIOS 提供一些系统参数以外，Linux 操作系统在运行时并不使用 BIOS 中的功能。

当计算机系统上电开机或者按了机箱上的复位按钮时，CPU 会自动把代码段寄存器 CS 设置为 0xF000，其段基址则被设置为 0xFFFF0000，段长度设置为 64KB。而 IP 被设置为 0xFFF0，因此此时 CPU 代码指针指向 0xFFFFF0 处，即 4G 空间最后一个 64K 的最后 16 字节处。由上图可知，这里正是系统 ROM BIOS 存放的位置。并且 BIOS 会在这里存放一条跳转指令 JMP 跳转到 BIOS 代码中 64KB 范围内的某一条指令开始执行。由于目前 PC/AT 微机中 BIOS 容量大多有 1MB 到 2MB，并存储在闪存（Flash Memory）ROM 中，因此为了能够执行或访问 BIOS 中超过 64KB 范围并且又远远不在 0--1M 地址空间中的其他 BIOS 代码或数据，BIOS 程序会首先使用一种称为 32 位大模式（Big Mode）技术把数据段寄存器的访问范围设置成 4G（而非原来的 64K），这样就可以在 0 到 4G 范围内执行和操作数据。此后，BIOS 在执行了一些列硬件检测和初始化操作之后，就会把与原来 PC 机兼容的 64KB BIOS 代码和数据复制到内存低端 1M 末端的 64K 处，然后跳转到这个地方并且让 CPU 进入真正的实地址模式工作，见图 2-5 所示。最后 BIOS 就会从硬盘或其他块设备把操作系统引导程序加载到内存 0x7c00 处，并跳转到这个地方继续执行引导程序。

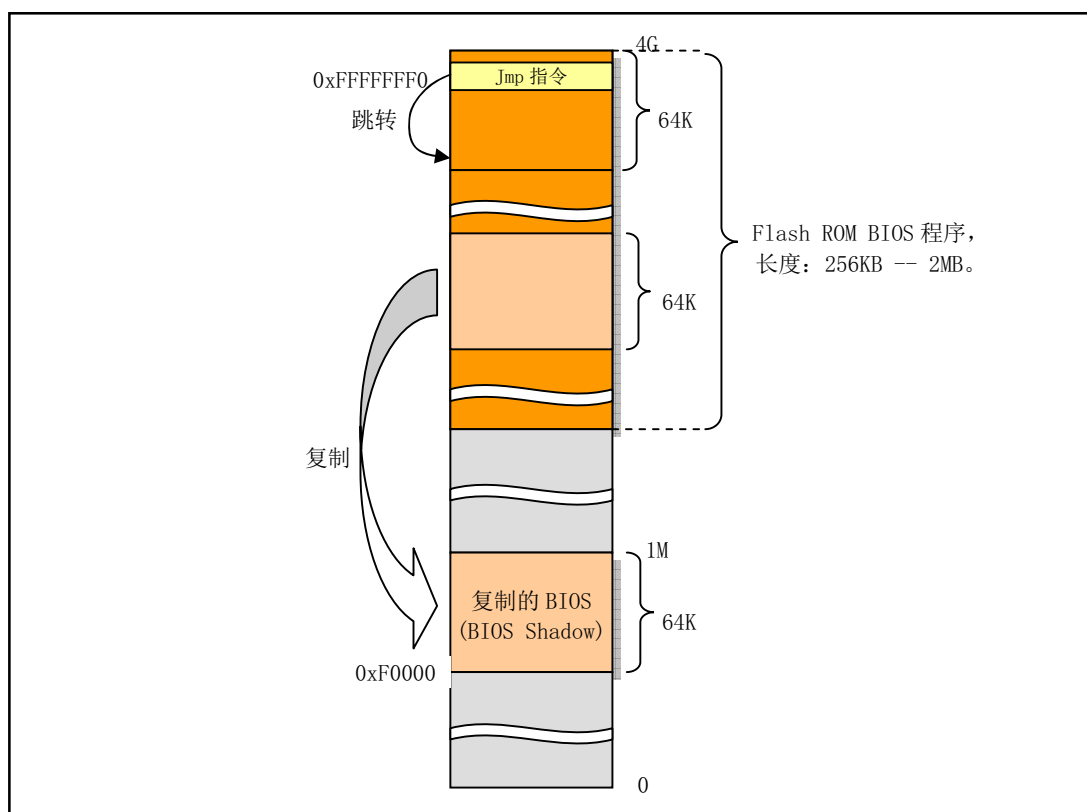


图 2-5 Flash ROM BIOS 位置和复制映射区域

### 2.3.3 CMOS 存储器

在 PC/AT 机中，除需要使用内存和 ROM BIOS 以外，还使用只有很少存储容量的（只有 64 或 128 字节）CMOS（Complementary Metal Oxide Semiconductor，互补金属氧化物半导体）存储器来存放计算机的实时时钟信息和系统硬件配置信息。这部分内存通常和实时时钟芯片（Real Time Chip）做在一块集成块中。CMOS 内存的地址空间在基本内存地址空间之外，需要使用 I/O 指令来访问。

## 2.4 控制器和控制卡

### 2.4.1 中断控制器

IBM PC/AT 80X86 兼容微机使用两片级联的 8259A 可编程中断控制芯片组成一个中断控制器，用于实现 I/O 设备的中断控制数据存取方式，并且能为 15 个设备提供独立的中断控制功能，见图 2-6 所示。在计算机刚开机初始化期间，ROM BIOS 会分别对两片 8259A 芯片进行初始化，并分别把 15 级中断优先级分配给时钟定时器、键盘、串行口、打印口、软盘控制、协处理器和硬盘等设备或控制器使用。同时在内存开始处 0x000-0xFFF 区域内建立一个中断向量表。但是由于这些设置违背了 Intel 公司的要求（后面章节将会详细说明），因此 Linux 操作系统在内核初始化期间又重新对 8259A 进行了设置。有关中断控制器工作原理和编程方法的详细说明请参见后续章节。

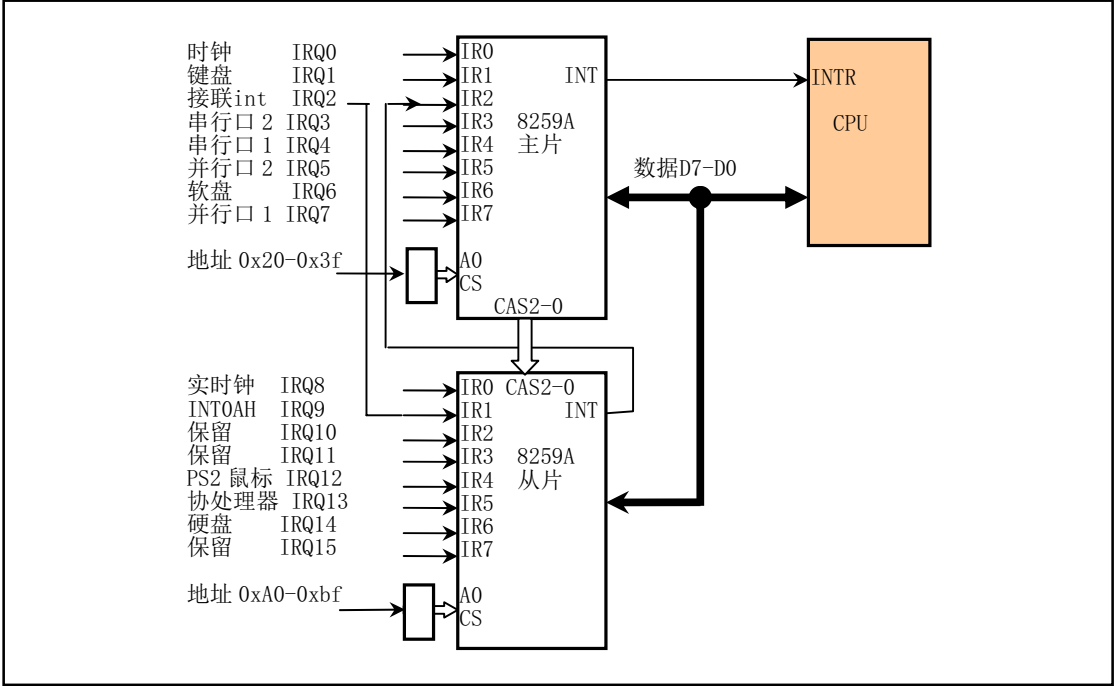


图 2-6 PC/AT 微机级连式 8259 控制系统

当一台 PC 计算机刚上电开机时，上图中的硬件中断请求号会被 ROM BIOS 设置成表 2-2 中列出的对应中断向量号。Linux 操作系统并不直接使用这些 PC 机默认设置好的中断向量号，当 Linux 系统执行初始化操作时，它会重新设置中断请求号与中断向量号的对应关系。

表 2-2 开机时 ROM BIOS 设置的硬件请求处理中断号

中断请求号	BIOS 设置的中断号	用途
IRQ0	0x08 (8)	8253 发出的 100HZ 时钟中断
IRQ1	0x09 (9)	键盘中断
IRQ2	0x0A (10)	接连从芯片
IRQ3	0x0B (11)	串行口 2
IRQ4	0x0C (12)	串行口 1
IRQ5	0x0D (13)	并行口 2
IRQ6	0x0E (14)	软盘驱动器
IRQ7	0x0F (15)	并行口 1
IRQ8	0x70 (112)	实时钟中断
IRQ9	0x71 (113)	改向至 INT 0x0A
IRQ10	0x72 (114)	保留
IRQ11	0x73 (115)	保留 (网络接口)
IRQ12	0x74 (116)	PS/2 鼠标口中断
IRQ13	0x75 (117)	数学协处理器中断
IRQ14	0x76 (118)	硬盘中断
IRQ15	0x77 (119)	保留

## 2.4.2 DMA 控制器

如前所述，DMA 控制器的主要功能是通过让外部设备直接与内存传输数据来增强系统的性能。通常它由机器上的 Intel 8237 芯片或其兼容芯片实现。通过对 DMA 控制器进行编程，外设与内存之间的数据传输能在不受 CPU 控制的条件下进行。因此在数据传输期间，CPU 可以做其他事情。

在 PC/AT 机中，使用了两片 8237 芯片，因此 DMA 控制器有 8 个独立的通道可使用。其中后 4 个是 16 位通道。软盘控制器被专门指定使用 DMA 通道 2。在使用一个通道之前必须首先对其设置。这牵涉到对三个端口的操作，分别是页面寄存器端口、（偏移）地址寄存器端口和数据计数寄存器端口。由于 DMA 寄存器是 8 位的，而地址和计数值是 16 位值，因此各自需要发送两次。

## 2.4.3 定时/计数器

Intel 8253/8254 是一个可编程定时/计数器（PIT - Programmable Interval Timer）芯片，用于处理计算机中的精确时间延迟。该芯片提供了 3 个独立的 16 位计数器通道。每个通道可工作在不同的工作模式下，并且这些工作方式均可以使用软件来设置。在软件中进行延时的一种方法是执行循环操作语句，但这样做很耗 CPU 时间。若机器中采用了 8253/8254 芯片，那么程序员就可以配置 8253 以满足自己的要求并且使用其中一个计数器通道达到所期望的延时。在延时到后，8253/8254 将会向 CPU 发送一个中断信号。

对于 PC/AT 及其兼容微机系统采用的是 8254 芯片。3 个定时/计数器通道被分别用于日时钟计时中断信号、动态内存 DRAM 刷新定时电路和主机扬声器音调合成。Linux 0.11 操作系统只对通道 0 进行了重新设置，使得该计数器工作在方式 3 下，并且每间隔 10 毫秒发出一个信号以产生中断请求信号（IRQ0）。这个间隔定时产生的中断请求就是 Linux 0.11 内核工作的脉搏，它用于定时切换当前执行的任务和统计每个任务使用的系统资源量（时间）。

## 2.4.4 键盘控制器

我们现在使用的键盘是 IBM 公司于 1984 年 PC/AT 微机的兼容键盘，通常称为 AT-PS/2 兼容键盘并具有 101 到 104 个按键。键盘上有一个称为键盘编码器的处理器（Intel 8048 或兼容芯片）专门用来扫描收集所有按键按下和松开的状态信息（即扫描码），并发送到主机主板上键盘控制器中。当一个键被按下时，键盘发送的扫描码称为接通扫描码（Make code），或简称为接通码；当一个被按下的键放开时发送的扫描码被称为断开扫描码（Break code），或简称为断开码。

主机键盘控制器专门用来对接收到的键盘扫描码进行解码，并把解码后的数据发送到操作系统的键盘数据队列中。因为每个按键的接通和断开码都是不同的，所以键盘控制器根据扫描码就可以确定用户在操作哪个键。整个键盘上所有按键的接通和断开码就组成了键盘的一个扫描码集（Scan Code Set）。根据计算机的发展，目前已有三套扫描码集可供使用，它们分别是：

- 第一套扫描码集 -- 原始 XT 键盘扫描码集。目前的键盘已经很少发送这类扫描码；
- 第二套扫描码集 -- 现代键盘默认使用的扫描码集，通常称为 AT 键盘扫描码集；
- 第三套扫描码集 -- PS/2 键盘扫描码集。原 IBM 推出 PS/2 微机时使用的扫描码集，已很少使用。

AT 键盘默认发送的是第二套扫描码集。虽然如此，主机键盘控制器为了与 PC/XT 机的软件兼容起见，仍然会把所有接收到的第二套键盘扫描码转换成第一套扫描码，见图 2-7 所示。因此，我们在为键盘控制器进行编程时通常只需要了解第一套扫描码集即可。这也是后面涉及键盘编程内容时只给出 XT 键盘扫描码集的原因。



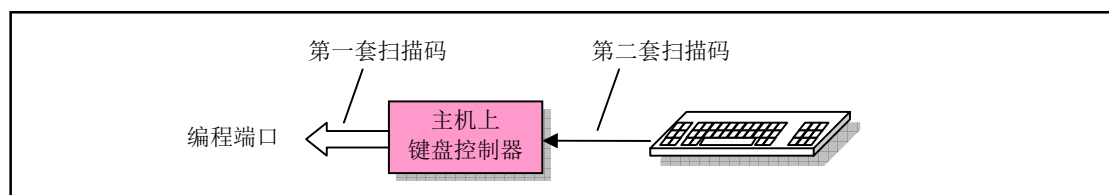


图 2-7 键盘控制器对扫描码集的转换

键盘控制器通常采用 Intel 8042 单片微处理器芯片或其兼容电路。现在的 PC 机都已经将键盘控制器集成在主板芯片组中，但是功能仍然与使用 8042 芯片的控制器相兼容。键盘控制器接收键盘发送来的 11 位串行格式数据。其中第 1 位是起始位，第 2--9 位是 8 位键盘扫描码，第 10 位是奇校验校验位，第 11 位是停止位。参见下节对串行控制卡的说明。键盘控制器在收到 11 位的串行数据后就将键盘扫描码转换成 PC/XT 标准键盘兼容的系统扫描码，然后通过中断控制器 IRQ1 引脚向 CPU 发送中断请求。当 CPU 响应该中断请求后，就会调用键盘中断处理程序来读取控制器中的 XT 键盘扫描码。

当一个键被按下时，我们可以从键盘控制器端口接收到一个 XT 键盘接通码。这个扫描码仅表示键盘上某个位置处的键被按下，但还没有对应到某个字符代码上。接通码通常都是一个字节宽度。例如，按下键“A”的接通码是 30 (0x1E)。当一个按下的键被松开时，从键盘控制器端口收到的就是一个断开码。对于 XT 键盘（即键盘控制器编程端口收到的扫描码），断开码是其接通码加上 0x80，即最高有效位（位 7）置位时的接通码。例如，上述“A”键的断开码就是  $0x80 + 0x1E = 0x9E$ 。

但是对于那些 PC/XT 标准 83 键键盘以后新添加的（“扩展的”）AT 键盘上的按键（例如右边的 Ctrl 键和右边的 Alt 键等），则其接通和断开扫描码通常有 2 到 4 个字节，并且第 1 个字节一定是 0xE0。例如，按下左边的非扩展 Ctrl 键时会产生 1 字节接通码 0x1D，而按下右边的 Ctrl 键时就会产生扩展的 2 字节接通码 0xE0、0x1D。对应的断开码是 0xE0、0x9D。表 2-3 中是几个接通和断开扫描码的例子。另外，附录中还给出了完整的第一套扫描码集。

表 2-3 键盘控制器编程端口接收到的第一套扫描码集接通和断开扫描码示例

按键	接通扫描码	断开扫描码	说明
A	0x1E	0x9E	非扩展的普通键
9	0x0A	0x8A	非扩展的普通键
功能键 F9	0x43	0xC3	非扩展的普通键
方向键向右键	0xE0, 0x4D	0xE0, 0xCD	扩展键
右边 Ctrl 键	0xE0, 0x1D	0xE0, 0x9D	扩展键
左 Shift 键 + G 键	0x2A, 0x22	0xAA, 0xA2	先按并且后释放 Shift 键

另外，键盘控制器 8042 的输出端口 P2 用于其他目的。其 P20 引脚用于实现 CPU 的复位操作，P21 引脚用于控制 A20 信号线的开启与否。当该输出端口位 1（P21）为 1 时就开启（选通）了 A20 信号线，为 0 则禁止 A20 信号线。现今的主板上已经不再包括独立的 8042 芯片了，但是主板上其他集成电路会为兼容目的而模拟 8042 芯片的功能。因此现在键盘的编程仍然采用 8042 的编程方法。

## 2.4.5 串行控制卡

### 1. 异步串行通信原理

两台计算机/设备进行数据交换，即通信，必须象人们对话一样使用同一种语言。在计算机通信术语中，我们把计算机/设备与计算机/设备之间的“语言”称为通信协议。通信协议规定了传送一个有效数据长度单位的格式。通常我们使用术语“帧”来形容这种格式。为了能让通信双方确定收/发的顺序和进行一些错误检测操作，除了必要的的数据以外，在传输的一帧信息中还包含起同步和错误检测作用的其它信息，例如

在开始传输数据信息之前先发送起始/同步或通信控制信息，并且在发送完需要的数据信息之后再传输一些校验信息等，见图 2-8 所示。

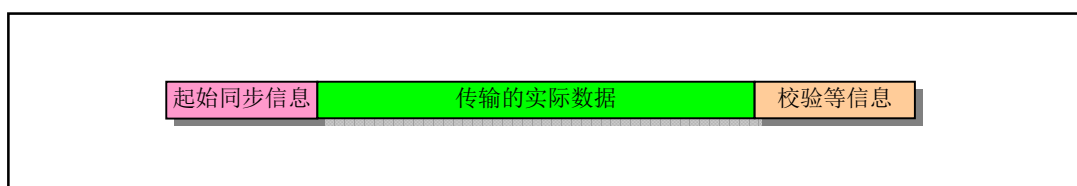


图 2-8 通信帧的一般结构

串行通信是指在线路上以比特位数据流一次一个比特进行传输的通信方式。串行通信可分为异步和同步串行通信两种类型。它们之间的主要区别在于传输时同步的通信单位或帧的长度不同。异步串行通信以一个字符作为一个通信单位或一帧进行传输，而同步串行通信则以多个字符或字节组成的序列作为一帧数据进行传输。若再以人们之间的对话作比喻，那么异步通信如同对话双方讲话速度很慢，说话时一个字（word）一个字地“蹦”出来，在说出每个字后可以停顿任意长时间。而同步通信则如同通信双方以连贯的一句话作为对话单位。可以看出，实际上如果我们把传输单位缩小到一个比特位时（对话时用字母!），那么以一个字符进行传输的异步串行通信也可以看作是一种同步传输通信方式。因此异步和同步通信的区分主要是一种习惯或惯例上的划分。

## 2. 异步串行传输格式

异步串行通信传输的帧格式见图 2-9 所示。传输一个字符由起始位、数据位、奇偶校验位和停止位构成。其中起始位起同步作用，值恒为 0。数据位是传输的实际数据，即一个字符的代码。其长度可以是 5--8 个比特。奇偶校验位可有可无，由程序设定。停止位恒为 1，可由程序设定为 1、1.5 或 2 个比特位。在通信开始发送信息之前，双方必须设置成相同的格式。如具有相同数量的数据比特位和停止位。在异步通信规范中，把传送 1 称为传号（MARK），传送 0 称为空号（SPACE）。因此在下面描述中我们就使用这两个术语。

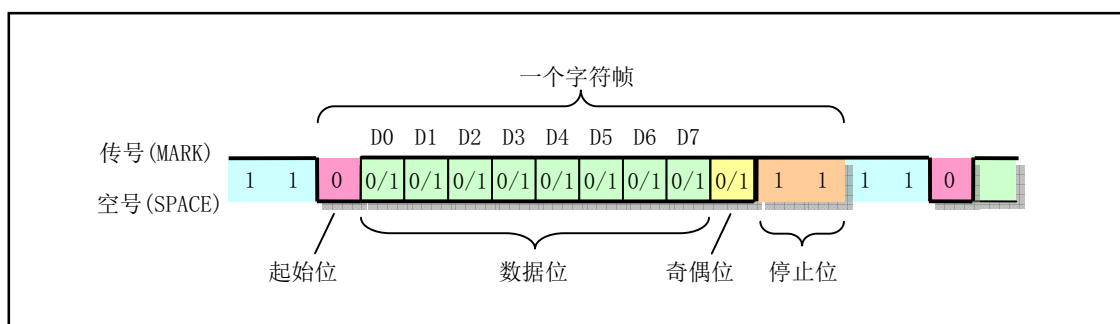


图 2-9 异步串行通信字符传输格式

当无数据传输时，发送方处于传号（MARK）状态，持续发送 1。若需要发送数据，则发送方需要首先发送一个比特位间隔时间的空号起始位。接收方收到空号后，就开始与发送方同步，然后接收随后的数据。若程序中设置了奇偶校验位，那么在数据传输完之后还需要接收奇偶校验位。最后是停止位。在一个字符帧发送完后可以立刻发送下一个字符帧，也可以暂时发送传号，等一会再发送字符帧。

在接收一字符帧时，接收方可能会检测到三种错误之一：①奇偶校验错误。此时程序应该要求对方重新发送该字符；②过速错误。由于程序取字符速度慢于接收速度，就会发生这种错误。此时应该修改程序加快取字符频率；③帧格式错误。在要求接收的格式信息不正确时会发生这种错误。例如在应该收到停止位时却收到了空号。通常造成这种错误的情况除了线路干扰以外，很可能是通信双方的帧格式设置的不同。

### 3. 串行控制器

为实现串行通信，PC 机上通常都带有 2 个符合 RS-232C 标准的串行接口，并使用通用异步接收/发送器控制芯片 UART（Universal Asynchronous Receiver/Transmitter）组成的串行控制器来处理串行数据的收发工作。PC 机上的串行接口通常使用 25 芯或 9 芯的 DB-25 或 DB-9 连接器，主要用来连接 MODEM 设备进行工作，因此 RS-232C 标准规定了很多 MODEM 专用接口引线。

以前的 PC 机都使用国家半导体公司的 NS8250 或 NS16450 UART 芯片，现在的 PC 机则使用了 16650A 及其兼容芯片，但都与 NS8250/16450 芯片兼容。NS8250/16450 与 16650A 芯片的主要区别在于 16650A 芯片还另外支持 FIFO 传输方式。在这种方式下，UART 可以在接收或发送了最多 16 个字符后才引发一次中断，从而可以减轻系统和 CPU 的负担。当 PC 机上电启动时，系统 RESET 信号通过 NS8250 的 MR 引脚使得 UART 内部寄存器和控制逻辑复位。此后若要使用 UART 就需要对其进行初始化编程操作，以设置 UART 的工作波特率、数据位数以及工作方式等。

## 2.4.6 显示控制

对于 IBM PC/AT 及其兼容计算机而言，可以使用彩色和单色显示卡。IBM 最早推出的 PC 机视频系统标准有单色 MDA 标准和彩色 CGA 标准以及 EGA 和 VGA 标准。以后推出的所有高级显示卡（包括现在的 AGP 显示卡）虽然都具有极高的图形处理速度和智能加速处理功能，但它们还是都支持这几种标准。Linux 0.1x 操作系统仅使用了这几种标准都支持的文本显示方式。

### 1. MDA 显示标准

单色显示适配器 MDA（Monochrome Display Adapter）仅支持黑白两色显示。并且只支持独有的文本字符显示方式（BIOS 显示方式 7）。其屏幕显示规格是 80 列 X 25 行（列号  $x = 0..79$ ；行号  $y = 0..24$ ），共可显示 2000 个字符。每个字符还带有 1 个属性字节，因此显示一屏（一帧）内容需要占 4KB 字节。其中偶地址字节存放字符代码，奇地址字节存放显示属性。MDA 卡配置有 8KB 显示内存。在 PC 机内存寻址范围中占用从 0xb0000 开始的 8KB 空间（0xb0000 -- 0xb2000）。如果显示屏行数是 `video_num_lines = 25`；列数是 `video_num_columns = 80`，那么位于屏幕列行值  $x$ 、 $y$  处的字符和属性在内存中的位置是：

---

```
字符字节位置 = 0xb0000 + video_num_columns * 2 * y + x * 2;
属性字节位置 = 字符字节位置 + 1;
```

---

在 MDA 单色文本显示方式中，每个字符的属性字节格式见表 2-4 所示。其中，D7 置 1 会使字符闪烁；D3 置 1 使字符高亮度显示。它与图 2-10 中的彩色文本字符的属性字节基本一致，但只有两种颜色：白色（0x111）和黑色（0x000）。它们的组合效果见表所示。

表 2-4 单色显示字符属性字节设置

背景色 D6D5D4	前景色 D2D1D0	属性值 无闪低亮	显示效果	示例
0 0 0	0 0 0	0x00	字符不可见。	
0 0 0	1 1 1	0x07	黑色背景上显示白色字符（正常显示）。	Normal
0 0 0	0 0 1	0x01	黑色背景上显示白色带下划线字符。	<u>Underline</u>
1 1 1	0 0 0	0x70	白色背景上显示黑色字符（反显）。	Reverse
1 1 1	1 1 1	0x77	显示白色方块。	■

### 2. CGA 显示标准

彩色图形适配器 CGA（Color Graphics Adapter）支持 7 种彩色和图形显示方式（BIOS 显示方式 0--6）。在 80 列 X 25 列的文本字符显示方式下，有单色和 16 色彩色两种显示方式（BIOS 显示方式 2--3）。CGA

卡标配 16KB 显示内存（占用内存地址范围 0xb8000 — 0xbc000），因此其中共可存放 4 帧显示信息。同样，在每一帧 4KB 显示内存中，偶地址字节存放字符代码，奇地址字节存放字符显示属性。但在 console.c 程序中只使用了其中 8KB 显示内存（0xb8000 — 0xba000）。在 CGA 彩色文本显示方式中，每个显示字符的属性字节格式定义见图 2-10 所示。

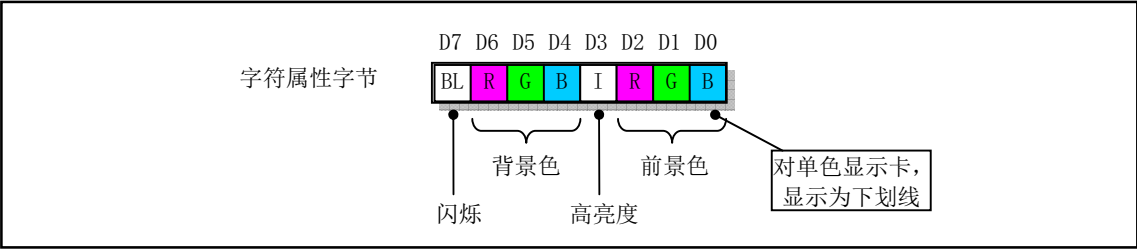


图 2-10 字符属性格式定义

与单色显示一样，图中 D7 置 1 用于让显示字符闪烁；D3 置 1 让字符高亮度显示；比特位 D6、D5、D4 和 D2、D1、D0 可以分别组合出 8 种颜色。前景色与高亮度比特位组合可以显示另外 8 种字符颜色。这些组合的颜色见表 2-5 所示。

表 2-5 前景色和背景色（左半部分）

I R G B	值	颜色名称	I R G B	值	颜色名称
0 0 0 0	0x00	黑色（Black）	1 0 0 0	0x08	深灰（Dark grey）
0 0 0 1	0x01	蓝色（Blue）	1 0 0 1	0x09	淡蓝（Light blue）
0 0 1 0	0x02	绿色（Green）	1 0 1 0	0x0a	淡绿（Light green）
0 0 1 1	0x03	青色（Cyan）	1 0 1 1	0x0b	淡青（Light cyan）
0 1 0 0	0x04	红色（Red）	1 1 0 0	0x0c	淡红（Light red）
0 1 0 1	0x05	品红（Magenta）	1 1 0 1	0x0d	淡品红（Light magenta）
0 1 1 0	0x06	棕色（Brown）	1 1 1 0	0x0e	黄色（Yellow）
0 1 1 1	0x07	灰白（Light grey）	1 1 1 1	0x0f	白色（White）

### 3. EGA/VGA 显示标准

增强型图形适配器 EGA（Enhanced Graphics Adapter）和视频图形阵列 VGA（Video Graphics Adapter）除兼容或支持 MDA 和 CGA 的显示方式以外，还支持其他在图形显示方面的增强显示方式。在与 MDA 和 CGA 兼容的显示方式下，占用的内存地址起始位置和范围都分别相同。但 EGA/VGA 都标配起码 32KB 的显示内存。在图形方式下占用从 0xa0000 开始的物理内存地址空间。

## 2.4.7 软盘和硬盘控制器

PC 机的软盘控制子系统由软盘片和软盘驱动器组成。由于软盘可以存储程序和数据并且携带方便，因此长期以来软盘驱动器是 PC 机上的标准配置设备之一。硬盘也是由盘片和驱动器组成，但是通常硬盘的金属盘片固定在驱动器中，不可拆卸。由于硬盘具有很大的存储容量，并且读写速度很快，因此它是 PC 机中最大容量的外部存储设备，通常也被称为外存。软盘和硬盘都是利用磁性介质保存信息，具有类似的存储工作方式。因此这里我们以硬盘为例来简要说明它们的工作原理。

在盘片上存储数据的基本方式是利用盘片表面上的一层磁性介质在磁化后的剩磁状态。软盘通常使用聚酯薄膜作基片，而硬盘片则通常使用金属铝合金作基片。一张软盘中含有一张聚酯薄膜圆盘片，使用上下两个磁头在盘片两面读写数据，盘片旋转速率大约在 300 转/分钟。硬盘中通常起码包括 2 张或者更多张

金属盘片，因此具有多个读写磁头。例如，对于包含 2 个盘片的硬盘中就具有 4 个物理磁头，含有 4 个盘片的硬盘中有 8 个读写磁头。见图 2-11 所示。硬盘旋转速率很快通常在 4500 转/分钟到 10000 转/分钟，因此硬盘数据的传输速度通常可达几十兆比特/秒。

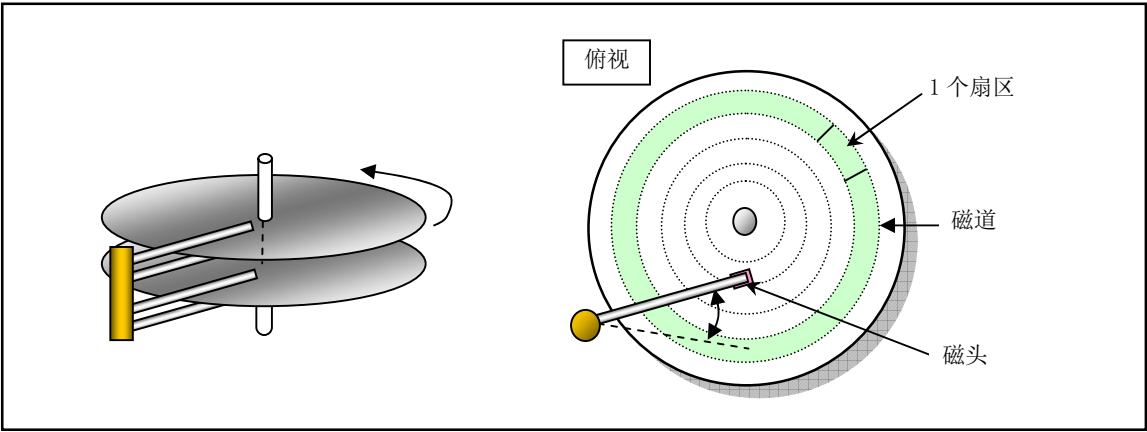


图 2-11 具有 2 张盘片的典型硬盘内部结构

位于磁盘表面的磁头上有分别有一个读线圈和写线圈。在读数据操作过程中，磁头首先移动到旋转着的磁盘某个位置上。由于磁盘在旋转，磁介质相对磁头作匀速运动，因此磁头实际上在切割磁介质上的磁力线。从而在读线圈中因感应而产生电流。根据磁盘表面剩磁状态方向的不同，在线圈中感应产生的电流方向也不同，因此磁盘上记录着的 0 和 1 数据就被读出，从而可从磁盘上顺序读出比特数据流。由于磁头读取的每个磁道上都有存放信息的特定格式，因此通过识别所读比特数据流中的格式，磁盘电路就可以区分并读取磁道上各扇区中的数据，见图 2-12 所示。其中，GAP 是间隔字段，用于隔离作用。通常 GAP 是 12 字节的 0。每个扇区地址场的地址字段存放着相关扇区的柱面号、磁头号（面号）和扇区号，因此通过读取地址场中的地址信息就可以唯一地确定一个扇区。

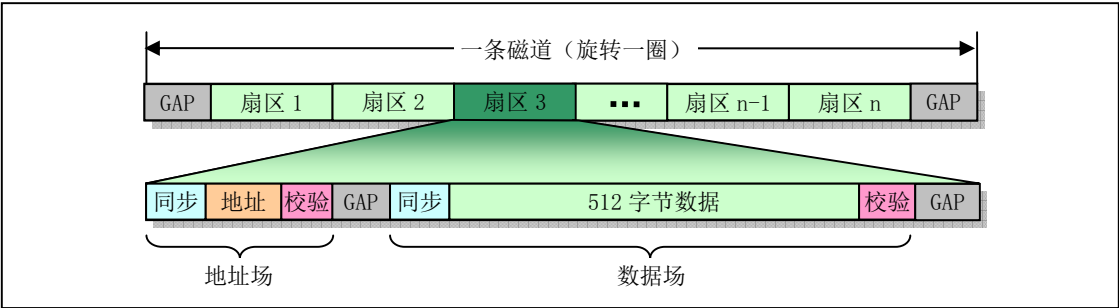


图 2-12 盘片磁道格式示意图

为了读写磁盘（软盘和硬盘）上的数据，就必须使用磁盘控制器。磁盘控制器是 CPU 与驱动器之间的逻辑接口电路，它从 CPU 接收请求命令，向驱动器发送寻道、读/写和控制信号，并且控制和转换数据流形式。控制器与驱动器之间传输的数据包括图 2-12 中的扇区地址信息以及定时和时钟信息。控制器必须从实际读/写数据中分离出这些地址信息和一些编码、解码等控制信息。另外，与驱动器之间的数据传输是串行比特数据流，因此控制器需要在并行字节数据和串行比特流数据之间进行转换。

PC/AT 机中软盘驱动控制器 FDC（Floppy Disk Controller）采用的是 NEC  $\mu$ PD765 或其兼容芯片。它主要用于接收 CPU 发出的命令，并根据命令要求向驱动器输出各种硬件控制信号，见图 2-13 所示。在执行读/写操作时，它需要完成数据的转换（串--并）、编码和校验操作，并且时刻监视驱动器的运行状态。

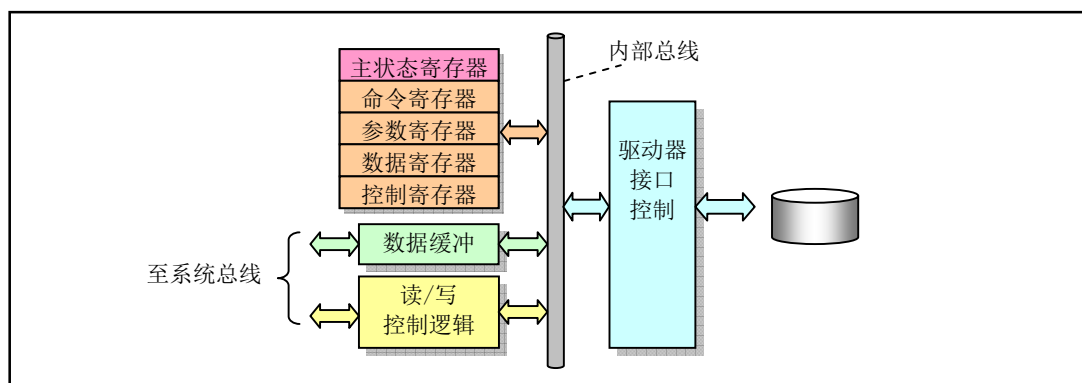


图 2-13 磁盘控制器内部示意图

对磁盘控制器的编程过程就是通过 I/O 端口设置控制器中的相关寄存器内容，并通过寄存器获取操作的结果信息。至于扇区数据的传输，则软盘控制器与 PC/AT 硬盘控制器不同。软盘控制器电路采用 DMA 信号，因此需要使用 DMA 控制器实施数据传输。而 AT 硬盘控制器采用高速数据块进行传输，不需要 DMA 控制器的介入。

由于软盘片比较容易遭到损坏（发霉或划伤），因此目前计算机中已经逐渐开始不配置软盘驱动器，取而代之的是使用容量较大并且更容易携带的 U 盘存储器。

## 2.5 本章小结

硬件是操作系统运行的基础平台。了解操作系统运行的硬件环境是深入理解运行其上操作系统的必要条件。本章根据传统微机的硬件组成结构，简单介绍了微机中各个主要部分。下一章我们从软件角度出发介绍编制 Linux 内核所使用的两种汇编语言语法和相关编译器，同时也介绍了编制内核使用的 GNU gcc 语法扩展部分的内容。



## 第3章 内核编程语言和环境

语言编译过程就是把人类能理解的高级语言转换成计算机硬件能理解和执行的二进制机器指令的过程。这种转换过程通常会产生一些效率不是很高的代码，所以对一些运行效率要求高或性能影响较大的部分代码通常就会直接使用低级汇编语言来编写，或者对高级语言编译产生的汇编程序再进行人工修改优化处理。本章主要描述 Linux 0.11 内核中使用的编程语言、目标文件格式和编译环境，主要目标是提供阅读 Linux 0.11 内核源代码所需要的汇编语言和 GNU C 语言扩展知识。首先比较详细地介绍了 as86 和 GNU as 汇编程序的语法和使用方法，然后对 GNU C 语言中的内联汇编、语句表达式、寄存器变量以及内联函数等内核源代码中常用的 C 语言扩展内容进行了介绍，同时详细描述了 C 和汇编函数之间的相互调用机制。因为理解目标文件格式是了解汇编器如何工作的重要前提之一，所以在介绍两种汇编语言时会首先简单介绍一下目标文件的基本格式，并在本章稍后部分再比较详细地给出 Linux 0.11 系统中使用的 a.out 目标文件格式。最后简单描述了 Makefile 文件的使用方法。

本章内容是阅读 Linux 内核源代码时的参考信息。因此可以先大致浏览一下本章内容，然后就阅读随后章节，在遇到问题时再回过头来参考本章内容。

### 3.1 as86 汇编器

在 Linux 0.1x 系统中使用了两种汇编器 (Assembler)。一种是能产生 16 位代码的 as86 汇编器，使用配套的 ld86 链接器；另一种是 GNU 的汇编器 gas (as)，使用 GNU ld 链接器来链接产生的目标文件。这里我们首先说明 as86 汇编器的使用方法，as 汇编器的使用方法放在下一节中进行说明。

as86 和 ld86 是由 MINIX-386 的主要开发者之一 Bruce Evans 编写的 Intel 8086、80386 汇编编译程序和链接程序。在刚开始开发 Linux 内核时 Linus 就已经把它移植到了 Linux 系统上。它虽然可以为 80386 处理器编制 32 位代码，但是 Linux 系统仅用它来创建 16 位的启动引导扇区程序 boot/bootsect.s 和实模式下初始设置程序 boot/setup.s 的二进制执行代码。该编译器快速小巧，并具有一些 GNU gas 所没有的特性，例如宏以及更多的错误检测手段。不过该编译器的语法与 GNU as 汇编编译器的语法不兼容而更近似于微软的 MASM、Borland 公司的 Turbo ASM 和 NASM 等汇编器的语法。这些汇编器都使用了 Intel 的汇编语言语法（如操作数的次序与 GNU as 的相反等）。

as86 的语法是基于 MINIX 系统的汇编语言语法，而 MINIX 系统的汇编语法则基于 PC/IX 系统的汇编器语法。PC/IX 是很早以前 Intel 8086 CPU 上运行的一个 UN\*X 类操作系统，Andrew S. Tanenbaum 就是在 PC/IX 系统上进行 MINIX 系统开发工作的。

Bruce Evans 是 MINIX 操作系统 32 位版本的主要修改编制者之一，他与 Linux 的创始人 Linus Torvalds 是好友。在 Linux 内核开发初期，Linus 从 Bruce Evans 那里学到了不少有关 UNIX 类操作系统的知识。MINIX 操作系统的不足之处也是两个好朋友互相探讨得出的结果。MINIX 的这些缺点正是激发 Linus 在 Intel 80386 体系结构上开发一个全新概念操作系统的主要动力之一。Linus 曾经说过：“Bruce 是我的英雄”，因此我们可以说 Linux 操作系统的诞生与 Bruce Evans 也有着密切的关系。

有关这个编译器和连接器的源代码可以从 FTP 服务器 ftp.funet.fi 上或从网站 [www.oldlinux.org](http://www.oldlinux.org) 下载。现代 Linux 系统上可以直接安装包含 as86/ld86 的 RPM 软件包，例如 dev86-0.16.3-8.i386.rpm。由于 Linux 系统仅使用 as86 和 ld86 编译和链接上面提到的两个 16 位汇编程序 bootsect.s 和 setup.s，因此这里仅介绍这两个程序中用到的一些汇编程序语法和汇编命令（汇编指示符）的作用和用途。

### 3.1.1 as86 汇编语言语法

汇编器专门用来把低级汇编语言程序编译成含机器码的二进制程序或目标文件。汇编器会把输入的一个汇编语言程序（例如 `srcfile`）编译成目标文件（`objfile`）。汇编的命令行基本格式是：

---

```
as [选项] -o objfile srcfile
```

---

其中选项用来控制编译过程以产生指定格式和设置的目标文件。输入的汇编语言程序 `srcfile` 是一个文本文件。该文件内容必须是由换行字符结尾的一系列文本行组成。虽然 GNU `as` 可使用分号在一行上包含多个语句，但通常在编制汇编语言程序时每行只包含一条语句。

语句可以是只包含空格、制表符和换行符的空行，也可以是赋值语句（或定义语句）、伪操作符语句和机器指令语句。赋值语句用于给一个符号或标识符赋值。它由标识符后跟一个等于号，再跟一个表达式组成，例如：“`BOOTSEG = 0x07C0`”。伪操作符语句是汇编器使用的指示符，它通常并不会产生任何代码。它由伪操作码和 0 个或多个操作数组成。每个操作码都由一个点字符 `'.'` 开始。点字符 `'.'` 本身是一个特殊的符号，它表示编译过程中的位置计数器。其值是点符号出现处机器指令第 1 个字节的地址。

机器指令语句是可执行机器指令的助记符，它由操作码和 0 个或多个操作数构成。另外，任何语句之前都可以有标号。标号是由一个标识符后跟一个冒号 `':'` 组成。在编译过程中，当汇编器遇到一个标号，那么当前位置计数器的值就会赋值给这个标号。因此一条汇编语句通常由标号（可选）、指令助记符（指令名）和操作数三个字段组成，标号位于一条指令的第一个字段。它代表其所在位置的地址，通常指明一个跳转指令的目标位置。最后还可以跟随用注释符开始的注释部分。

汇编器编译产生的目标文件 `objfile` 通常起码包含三个段或区<sup>3</sup>（section），即正文段（`.text`）、数据段（`.data`）和未初始化数据段（`.bss`）。正文段（或称为代码段）是一个已初始化过的段，通常其中包含程序的执行代码和只读数据。数据段也是一个已初始化过的段，其中包含有可读/写的数据。而未初始化数据段是一个未初始化的段。通常汇编器产生的输出目标文件中不会为该段保留空间，但在目标文件链接成执行程序被加载时操作系统会把该段的内容全部初始化为 0。在编译过程中，汇编语言程序中会产生代码或数据的语句，都会在这三个中的一个段中生成代码或数据。编译产生的字节会从 `' .text'` 段开始存放。我们可以使用段控制伪操作符来更改写入的段。目标文件格式将在后面“Linux 0.11 目标文件格式”一节中加以详细说明。

### 3.1.2 as86 汇编语言程序

下面我们以一个简单的框架示例程序 `boot.s` 来说明 as86 汇编程序的结构以及程序中语句的语法，然后给出编译链接和运行方法，最后再分别列出 as86 和 ld86 的使用方法和编制选项。示例程序见如下所示。这个示例是 `bootsect.s` 的一个框架程序，能编译生成引导扇区代码。其中为了演示说明某些语句的使用方法，故意加入了无意义的第 20 行语句。

---

```
1 !
2 ! boot.s -- bootsect.s 的框架程序。用代码 0x07 替换串 msg1 中 1 字符，然后在屏幕第 1 行上显示。
3 !
4 .globl begtext, begdata, begbss, endtext, enddata, endbss      ! 全局标识符，供 ld86 链接使用；
5 .text                  ! 正文段；
```

---

<sup>3</sup> 有关目标文件中术语“section”对应的中文名称有多种。在 UNIX 操作系统早期阶段，该术语在目标文件中均称为“segment”。这是因为早期目标文件中的段可以直接对应到计算机处理器中段的的概念上。但是由于现在目标文件中的段的的概念已经与处理器中的段寄存器没有直接对应关系，并且容易把这两者混淆起来，因此现在英文文献中均使用“section”取代目标文件中的“segment”命名。这也可以从 GNU 使用手册的各个版本变迁中观察到。section 的中文译法有“段、区、节、部分和区域”等几种。本书在不至于混淆处理器段概念前提下会根据所述内容把“section”称为“段”、“区”或“部分”，但主要采用“区”这个名称。



---

```

6 begtext:
7 .data                ! 数据段;
8 begdata:
9 .bss                 ! 未初始化数据段;
10 begbss:
11 .text               ! 正文段;
12 BOOTSEG = 0x07c0    ! BIOS 加载 bootsect 代码的原始段地址;
13
14 entry start         ! 告知链接程序, 程序从 start 标号处开始执行。
15 start:
16     jmp     go, BOOTSEG ! 段间跳转。INITSEG 指出跳转段地址, 标号 go 是偏移地址。
17 go:     mov     ax, cs    ! 段寄存器 cs 值-->ax, 用于初始化数据段寄存器 ds 和 es。
18         mov     ds, ax
19         mov     es, ax
20         mov     [msg1+17], ah ! 0x07-->替换字符串中 1 个点符号, 喇叭将会鸣一声。
21         mov     cx, #20    ! 共显示 20 个字符, 包括回车换行符。
22         mov     dx, #0x1004 ! 字符串将显示在屏幕第 17 行、第 5 列处。
23         mov     bx, #0x000c ! 字符显示属性 (红色)。
24         mov     bp, #msg1   ! 指向要显示的字符串 (中断调用要求)。
25         mov     ax, #0x1301 ! 写字符串并移动光标到串结尾处。
26         int     0x10        ! BIOS 中断调用 0x10, 功能 0x13, 子功能 01。
27 loop1:  jmp     loop1      ! 死循环。
28 msg1:   .ascii "Loading system ..." ! 调用 BIOS 中断显示的信息。共 20 个 ASCII 码字符。
29         .byte 13, 10
30 .org 510          ! 表示以后语句从地址 510 (0x1FE) 开始存放。
31         .word 0xAA55       ! 有效引导扇区标志, 供 BIOS 加载引导扇区使用。
32 .text
33 endtext:
34 .data
35 enddata:
36 .bss
37 endbss:

```

---

我们首先介绍该程序的功能, 然后再详细说明各语句的作用。该程序是一个简单的引导扇区启动程序。编译链接产生的执行程序可以放入软盘第 1 个扇区直接用来引导计算机启动。启动后会在屏幕第 17 行、第 5 列处显示出红色字符串 "Loading system ..", 并且光标下移一行。然后程序就在第 27 行上死循环。

该程序开始的 3 行是注释语句。在 as86 汇编语言程序中, 凡是以感叹号 '!' 或分号 ';' 开始的语句其后面均为注释文字。注释语句可以放在任何语句的后面, 也可以从一个新行开始。

第 4 行上的 '.globl' 是汇编指示符 (或称为汇编伪指令、伪操作符)。汇编指示符均以一个字符 '.' 开始, 并且不会在编译时产生任何代码。汇编指示符由一个伪操作码, 后跟 0 个或多个操作数组成。例如第 4 行上的 '.globl' 是一个伪操作码, 而其后面的标号 'begtext, begdata, begbss' 等标号就是它的操作数。标号是后面带冒号的标识符, 例如第 6 行上的 'begtext:'。但是在引用一个标号时无须带冒号。

通常一个汇编器都支持很多不同的伪操作符, 但是下面仅说明 Linux 系统 bootsect.s 和 setup.s 汇编语言程序用到的和一些常用的 as86 伪操作符。

'globl' 伪操作符用于定义随后的标号标识符是外部的或全局的, 并且即使不使用也强制引入。

第 5 行到第 11 行上除定义了 3 个标号外, 还定义了 3 个伪操作符: '.text'、'.data'、'.bss'。它们分别对应汇编程序编译产生目标文件中的 3 个段, 即正文段、数据段和未初始化数据段。'.text' 用于标识正文段的开始位置, 并把切换到 text 段; '.data' 用于标识数据段的开始位置, 并把当前段切换到 data 段; 而 '.bss' 则用于标识一个未初始化数据段的开始, 并把当前段改变成 bss 段。因此行 5--11 用于在每个段中定义一个标号, 最后再切换到 text 段开始编写随后的代码。这里把三个段都定义在同一重叠地址

范围中，因此本示例程序实际上不分段。

第 12 行定义了一个赋值语句“BOOTSEG = 0x07c0”。等号‘=’（或符号‘EQU’）用于定义标识符 BOOTSEG 所代表的值，因此这个标识符可称为符号常量。这个值与 C 语言中的写法一样，可以使用十进制、八进制和十六进制。

第 14 行上的标识符‘entry’是保留关键字，用于迫使链接器 ld86 在生成的可执行文件中包括进其后指定的标号‘start’。通常在链接多个目标文件生成一个可执行文件时应该在其中一个汇编程序中用关键词 entry 指定一个入口标号，以便于调试。但是在我们这个示例中以及 Linux 内核 boot/bootsect.s 和 boot/setup.s 汇编程序中完全可以省略这个关键词，因为我们并不希望在生成的纯二进制执行文件中包括任何符号信息。

第 16 行上是一个段间（Inter-segment）远跳转语句，就跳转到下一条指令。由于当 BIOS 把程序加载到物理内存 0x7c00 处并跳转到该处时，所有段寄存器（包括 CS）默认值均为 0，即此时 CS:IP=0x0000:0x7c00。因此这里使用段间跳转语句就是为了给 CS 赋段值 0x7c0。该语句执行后 CS:IP = 0x07C0:0x0005。随后的两条语句分别给 DS 和 ES 段寄存器赋值，让它们都指向 0x7c0 段。这样便于对程序中的数据（字符串）进行寻址。

第 20 行上的 MOV 指令用于把 ah 寄存器中 0x7c0 段值的高字节（0x07）存放到内存中字符串 msg1 最后一个‘.’位置处。这个字符将导致 BIOS 中断在显示字符串时鸣叫一声。使用这条语句主要是为了说明间接操作数的用法。在 as86 中，间接操作数需要使用方括号对。另外一些寻址方式有以下一些：

---

！直接寄存器寻址。跳转到 bx 值指定的地址处，即把 bx 的值拷贝到 IP 中。

```
mov    bx, ax
jmp    bx
```

！间接寄存器寻址。bx 值指定内存位置处的内容作为跳转的地址。

```
mov    [bx], ax
jmp    [bx]
```

！把立即数 1234 放到 ax 中。把 msg1 地址值放到 ax 中。

```
mov    ax, #1234
mov    ax, #msg1
```

！绝对寻址。把内存地址 1234（msg1）处的内容放入 ax 中。

```
mov    ax, 1234
mov    ax, msg1
mov    ax, [msg1]
```

！索引寻址。把第 2 个操作数所指内存位置处的值放入 ax 中。

```
mov    ax, msg1[bx]
mov    ax, msg1[bx*4+si]
```

---

第 21--25 行的语句分别用于把立即数放到相应的寄存器中。立即数前一定要加井号‘#’，否则将作为内存地址使用而使语句变成绝对寻址语句，见上面示例。另外，把一个标号（例如 msg1）的地址值放入寄存器中时也一定要在前面加‘#’，否则会变成把 msg1 地址处的内容放到了寄存器中！

第 26 行是 BIOS 屏幕显示中断调用 int 0x10。这里使用其功能 19、子功能 1。该中断的作用是把一字符串（msg1）写到屏幕指定位置处。寄存器 cx 中是字符串长度值，dx 中是显示位置值，bx 中是显示使用的字符属性，es:bp 指向字符串。

第 27 行是一个跳转语句，跳转到当前指令处。因此这是一个死循环语句。这里采用死循环语句是为了让显示的内容能够停留在屏幕上而不被删除。死循环语句是调试汇编程序时常用的方法。

第 28--29 行定义了字符串 msg1。定义字符串需要使用伪操作符‘.ascii’，并且需要使用双引号括住字符串。伪操作符‘.ascii’还会自动在字符串后添加一个 NULL（0）字符。另外，第 29 行上定义了回车和换行（13, 10）两个字符。定义字符需要使用伪操作符‘.byte’，并且需要使用单引号把字符括住。例如：“‘D’”。当然我们也可以象示例中的一样直接写出字符的 ASCII 码。

第 30 行上的伪操作符语句 `'.org'` 定义了当前汇编的位置。这条语句会把汇编器编译过程中当前段的位置计数器值调整为该伪操作符语句上给出的值。对于本示例程序，该语句把位置计数器设置为 510，并在此处（第 31 行）放置了有效引导扇区标志字 `0xAA55`。伪操作符 `'.word'` 用于在当前位置定义一个双字节内存对象（变量），其后可以是一个数或者是一个表达式。由于后面没有代码或数据了，因此我们可以据此确定 `boot.s` 编译出来的执行程序应该正好为 512 字节。

第 32--37 行又在三个段中分别放置了三个标号。分别用来表示三个段的结束位置。这样设置可以用来在链接多个目标模块时区分各个模块中各段的开始和结束位置。由于内核中的 `bootsec.s` 和 `setup.s` 程序都是单独编译链接的程序，各自期望生成的都是纯二进制文件而并没有与其他目标模块文件进行链接，因此示例程序中声明各个段的伪操作符（`.text`、`.data` 和 `.bss`）都完全可以省略掉，即程序中第 4--11 行和 32--37 行可以全部删除也能编译链接产生出正确的结果。

### 3.1.3 as86 汇编语言程序的编译和链接

现在我们说明如何编译链接示例程序 `boot.s` 来生成我们需要引导扇区程序 `boot`。编译和链接上面示例程序需要执行以下前两条命令：

---

```
[/root]# as86 -O -a -o boot.o boot.s           // 编译。生成与 as 部分兼容的目标文件。
[/root]# ld86 -O -s -o boot boot.o             // 链接。去掉符号信息。
[/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    // 写入软盘或 Image 盘文件中。
16+0 records in
16+0 records out
[/root]# _
```

---

其中第 1 条命令利用 `as86` 汇编器对 `boot.s` 程序进行编译，生成 `boot.o` 目标文件。第 2 条命令使用链接器 `ld86` 对目标文件执行链接操作，最后生成 MINIX 结构的可执行文件 `boot`。其中选项 `'-O'` 用于生成 8086 的 16 位目标程序；`'-a'` 用于指定生成与 GNU `as` 和 `ld` 部分兼容的代码。`'-s'` 选项用于告诉链接器要去除最后生成的可执行文件中的符号信息。`'-o'` 指定生成的可执行文件名称。

从上面 `ls` 命令列出的文件名中可以看出，最后生成的 `boot` 程序并不是前面所说的正好 512 字节，而是长了 32 字节。这 32 字节就是 MINIX 可执行文件的头结构（其详细结构说明请参见“内核组建工具”一章内容）。为了能使用这个程序引导启动机器，我们需要人工去掉这 32 字节。去掉该头结构的方法有几种：

- 使用二进制编辑程序删除 `boot` 程序前 32 字节，并存盘；
- 使用现在 Linux 系统（例如 RedHat 9）上的 `as86` 编译链接程序，它们具有可生成不带 MINIX 头结构的纯二进制执行文件的选项，请参考相关系统的在线使用手册页（`man as86`）。
- 利用 Linux 系统的 `dd` 命令。

上面列出的第 3 条命令就是利用 `dd` 命令来去除 `boot` 中的前 32 字节，并把输出结果直接写到软盘或 Bochs 模拟系统的软盘映像文件中（有关 Bochs PC 机模拟系统的使用方法请参考最后一章内容）。若在 Bochs 模拟系统中运行该程序，我们可得到如图 3-1 所示画面。

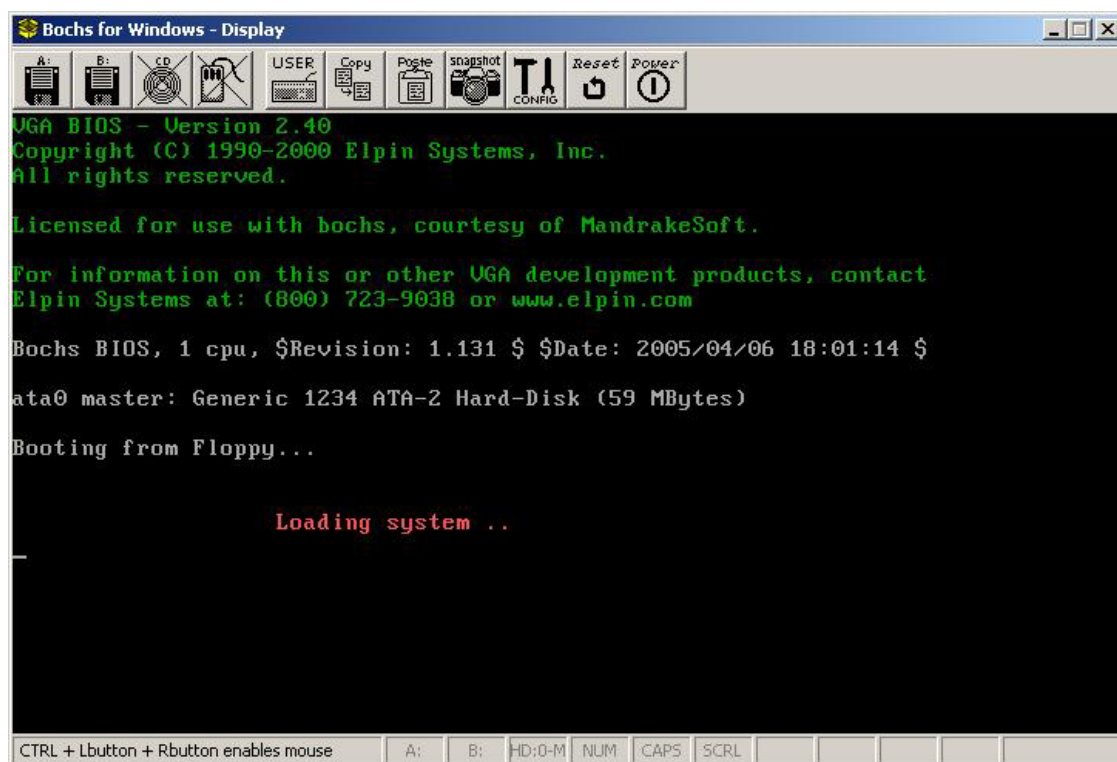


图 3-1 在 Bochs 模拟系统中运行 boot 引导程序的显示结果

### 3.1.4 as86 和 ld86 使用方法和选项

as86 和 ld86 的使用方法和选项如下：

as 的使用方法和选项：

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

默认设置（除了以下默认值以外，其他选项默认为关闭或无；若没有明确说明 a 标志，则不会有输出）：

- 3 使用 80386 的 32 位输出；
- list 在标准输出上显示；
- name 源文件的基本名称（即不包括 '.' 后的扩展名）；

各选项含义：

- O 使用 16 比特代码段；
- 3 使用 32 比特代码段；
- a 开启与 GNU as、ld 的部分兼容性选项；
- b 产生二进制文件，后面可以跟文件名；
- g 在目标文件中仅存入全局符号；
- j 使所有跳转语句均为长跳转；
- l 产生列表文件，后面可以跟随列表文件名；
- m 在列表中扩展宏定义；
- n 后面跟随模块名称（取代源文件名称放入目标文件中）；
- o 产生目标文件，后跟目标文件名（objfile）；
- s 产生符号文件，后跟符号文件名；
- u 将未定义符号作为输入的未指定段的符号；
- w 不显示警告信息；

---

ld 连接器的使用语法和选项:

对于生成 Minix a.out 格式的版本:

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

对于生成 GNU-Minix 的 a.out 格式的版本:

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

默认设置(除了以下默认值以外, 其他选项默认为关闭或无):

-O3        32 位输出;

outfile        a.out 格式输出;

-O    产生具有 16 比特魔数的头结构, 并且对-lx 选项使用 i86 子目录;

-3    产生具有 32 比特魔数的头结构, 并且对-lx 选项使用 i386 子目录;

-M    在标准输出设备上显示已链接的符号;

-T    后面跟随正文基地址 (使用适合于 strtoul 的格式);

-i    分离的指令与数据段 (I&D) 输出;

-lx    将库/local/lib/subdir/libx.a 加入链接的文件列表中;

-m    在标准输出设备上显示已链接的模块;

-o    指定输出文件名, 后跟输出文件名;

-r    产生适合于进一步重定位的输出;

-s    在目标文件中删除所有符号。

---

## 3.2 GNU as 汇编

上节介绍的 as86 汇编器仅用于编译内核中的 boot/bootsect.s 引导扇区程序和实模式下的设置程序 boot/setup.s。内核中其余所有汇编语言程序 (包括 C 语言产生的汇编程序) 均使用 gas 来编译, 并与 C 语言程序编译产生的模块链接。本节以 80X86 CPU 硬件平台为基础介绍 Linux 内核中使用汇编程序语法和 GNU as 汇编器 (简称 as 汇编器) 的使用方法。我们首先介绍 as 汇编语言程序的语法, 然后给出常用汇编伪指令 (指示符) 的含义和使用方法。带有详细说明信息的 as 汇编语言程序实例将在下一章最后给出。

由于操作系统许多关键代码要求有很高的执行速度和效率, 因此在一个操作系统源代码中通常就会包含大约 10% 左右的起关键作用的汇编语言程序量。Linux 操作系统也不例外, 它的 32 位初始化代码、所有中断和异常处理过程接口程序、以及很多宏定义都使用了 as 汇编语言程序或扩展的嵌入汇编语句。是否能够理解这些汇编语言程序的功能也就无疑成为理解一个操作系统具体实现的关键点之一。

在编译 C 语言程序时, GNU gcc 编译器会首先输出一个作为中间结果的 as 汇编语言文件, 然后 gcc 会调用 as 汇编器把这个临时汇编语言程序编译成目标文件。即实际上 as 汇编器最初是专门用于汇编 gcc 产生的中间汇编语言程序的, 而非作为一个独立的汇编器使用。因此, as 汇编器也支持很多 C 语言特性, 这包括字符、数字和常数表示方法以及表达式形式等方面。

GNU as 汇编器最初是仿照 BSD 4.2 的汇编器进行开发的。现在的 as 汇编器能够配置成产生很多不同格式的目标文件。虽然编制的 as 汇编语言程序与具体采用或生成什么格式的目标文件关系不大, 但是在下面介绍中若有涉及目标文件格式时, 我们将围绕 Linux 0.11 系统采用的 a.out 目标文件格式进行说明。

### 3.2.1 编译 as 汇编语言程序

使用 as 汇编器编译一个 as 汇编语言程序的基本命令行格式如下所示:

---

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

---

其中 objfile 是 as 编译输出的目标文件名, srcfile.s 是 as 的输入汇编语言程序名。如果没有使用输出

文件名，那么 `as` 会编译输出名称为 `a.out` 的默认目标文件。在 `as` 程序名之后，命令行上可包含编译选项和文件名。所有选项可随意放置，但是文件名的放置次序编译结果密切相关。

一个程序的源程序可以被放置在一个或多个文件中，程序的源代码是如何分割放置在几个文件中并不会改变程序的语义。程序的源代码是所有这些文件按次序的组合结果。每次运行 `as` 编译器，它只编译一个源程序。但一个源程序可由多个文本文件组成（终端的标准输入也是一个文件）。

我们可以在 `as` 命令行上给出零个或多个输入文件名。`as` 将会按从左到右的顺序读取这些输入文件的内容。在命令行上任何位置处的参数若没有特定含义的话，将会被作为一个输入文件名看待。如果在命令行上没有给出任何文件名，那么 `as` 将会试图从终端或控制台标准输入中读取输入文件内容。在这种情况下，若已没有内容要输入时就需要手工键入 `Ctrl-D` 组合键来告知 `as` 汇编器。若想在命令行上明确指明把标准输入作为输入文件，那么就需要使用参数 `'--'`。

`as` 的输出文件是输入的汇编语言程序编译生成的二进制数据文件，即目标文件。除非我们使用选项 `'-o'` 指定输出文件的名称，否则 `as` 将产生名为 `a.out` 的输出文件。目标文件主要用于作为链接器 `ld` 的输入文件。目标文件中包含有已汇编过的程序代码、协助 `ld` 产生可执行程序的信息、以及可能还包含调试符号信息。Linux 0.11 系统中使用的 `a.out` 目标文件格式将在本章后面进行说明。

假如我们想单独编译 `boot/head.s` 汇编程序，那么可以在命令行上键入如下形式的命令：

---

```
[/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 汇编语法

为了维持与 `gcc` 输出汇编程序的兼容性，`as` 汇编器使用 AT&T 系统 V 的汇编语法（下面简称为 AT&T 语法）。这种语法与 Intel 汇编程序使用的语法（简称 Intel 语法）很不一样，它们之间的主要区别有一些几点：

- AT&T 语法中立即操作数前面要加一个字符 `'$'`；寄存器操作数名前要加字符百分号 `'%'`；绝对跳转/调用（相对于与程序计数器有关的跳转/调用）操作数前面要加星号 `'*'`。而 Intel 汇编语法均没有这些限制。
- AT&T 语法与 Intel 语法使用的源和目的操作数次序正好相反。AT&T 的源和目的操作数是从左到右‘源，目的’。例如 Intel 的语句 `'add eax, 4'` 对应 AT&T 的 `'addl $4, %eax'`。
- AT&T 语法中内存操作数的长度（宽度）由操作码最后一个字符来确定。操作码后缀 `'b'`、`'w'` 和 `'l'` 分别指示内存引用宽度为 8 位字节（byte）、16 位字（word）和 32 位长字（long）。Intel 语法则通过在内存操作数前使用前缀 `'byte ptr'`、`'word ptr'` 和 `'dword ptr'` 来达到同样目的。因此，Intel 的语句 `'mov al, byte ptr foo'` 对应于 AT&T 的语句 `'movb $foo, %al'`。
- AT&T 语法中立即形式的远跳转和远调用为 `'ljmp/lcall $section, $offset'`，而 Intel 的是 `'jmp/call far section:offset'`。同样，AT&T 语法中远返回指令 `'lret $stack-adjust'` 对应 Intel 的 `'ret far stack-adjust'`。
- AT&T 汇编器不提供对多代码段程序的支持，UNIX 类操作系统要求所有代码在一个段中。

#### 3.2.2.1 汇编程序预处理

`as` 汇编器具有对汇编语言程序内置的简单预处理功能。该预处理功能会调整并删除多余的空格字符和制表符；删除所有注释语句并且使用单个空格或一些换行符替换它们；把字符常数转换成对应的数值。但是该预处理功能不会对宏定义进行处理，也没有处理包含文件的功能。如果需要这方面的功能，那么可以让汇编语言程序使用大写的后缀 `'.S'` 让 `as` 使用 `gcc` 的 CPP 预处理功能。

由于 as 汇编语言程序除了使用 C 语言注释语句（即 `/*` 和 `*/`）以外，还使用井号 `#` 作为单行注释开始字符，因此若在汇编之前不对程序执行预处理，那么程序中包含的所有以井号 `#` 开始的指示符或命令均会被当作注释部分。

### 3.2.2.2 符号、语句和常数

符号（Symbol）是由字符组成的标识符，组成符号的有效字符取自于大小写字符集、数字和三个字符 `_. $`。符号不允许用数字字符开始，并且大小写含义不同。在 as 汇编程序中符号长度没有限制，并且符号中所有字符都是有效的。符号使用其他字符（例如空格、换行符）或者文件的开始来界定开始和结束处。

语句（Statement）以换行符或者行分割字符（`;`）作为结束。文件最后语句必须以换行符作为结束。若在一行的最后使用反斜杠字符 `\`（在换行符前），那么就可以让一条语句使用多行。当 as 读取到反斜杠加换行符时，就会忽略掉这两个字符。

语句由零个或多个标号（Label）开始，后面可以跟随一个确定语句类型的关键符号。标号由符号后面跟随一个冒号（`:`）构成。关键符号确定了语句余下部分的语义。如果该关键符号以一个 `.` 开始，那么当前语句就是一个汇编命令（或称为伪指令、指示符）。如果关键符号以一个字母开始，那么当前语句就是一条汇编语言指令语句。因此一条语句的通用格式为：

---

标号： 汇编命令      注释部分（可选）  
 或  
 标号： 指令助记符 操作数 1, 操作数 2      注释部分（可选）

---

常数是一个数字，可分为字符常数和数字常数两类。字符常数还可分为字符串和单个字符；而数字常数可分为整数、大数和浮点数。

字符串必须用双引号括住，并且其中可以使用反斜杠 `\` 来转义包含特殊字符。例如 `\"` 表示一个反斜杠字符。其中第 1 个反斜杠是转义指示字符，说明把第 2 个字符看作一个普通反斜杠字符。常用转义符序列见表 3-1 所示。反斜杠后若是其他字符，那么该反斜杠将不起作用并且 as 汇编器将会发出警告信息。

汇编程序中使用单个字符常数时可以写成在该字符前加一个单引号，例如 `'A'` 表示值 65、`'C'` 表示值 67。表 3-1 中的转义码也同样可以用于单个字符常数。例如 `'\"` 表示是一个普通反斜杠字符常数。

表 3-1 as 汇编器支持的转义字符序列

转义码	说明
<code>\b</code>	退格符（Backspace），值为 0x08
<code>\f</code>	换页符（FormFeed），值为 0x0C
<code>\n</code>	换行符（Newline），值为 0x0A
<code>\r</code>	回车符（Carriage-Return），值为 0x0D
<code>\NNN</code>	3 个八进制数表示的字符代码
<code>\xNN...</code>	16 进制数表示的字符代码
<code>\\</code>	表示一个反斜杠字符
<code>\"</code>	表示字符串中的一个双引号"

整数数字常数有 4 种表示方法，即使用 `0b` 或 `0B` 开始的二进制数（`0-1`）；以 `0` 开始的八进制数（`0-7`）；以非 `0` 数字开始的十进制数（`0-9`）和使用 `0x` 或 `0X` 开头的十六进制数（`0-9a-fA-F`）。若要表示负数，只需在前面添加负号 `-`。

大数（Bignum）是位数超过 32 位二进制位的数，其表示方法与整数的相同。汇编程序中对浮点常数的表示方法与 C 语言中的基本一样。由于内核代码中几乎不用浮点数，因此这里不再对其进行说明。

### 3.2.3 指令语句、操作数和寻址

指令（Instructions）是 CPU 执行的操作，通常指令也称作操作码（Opcode）；操作数（Operand）是指令操作的对象；而地址（Address）是指定数据在内存中的位置。指令语句是程序运行时刻执行的一条语句，它通常可包含 4 个组成部分：

- 标号（可选）；
- 操作码（指令助记符）；
- 操作数（由具体指令指定）；
- 注释

一条指令语句可以含有 0 个或最多 3 个用逗号分开的操作数。对于具有两个操作数的指令语句，第 1 个是源操作数，第 2 个是目的操作数，即指令操作结果保存在第 2 个操作数中。

操作数可以是立即数（即值是常数值表达式）、寄存器（值在 CPU 的寄存器中）或内存（值在内存中）。一个间接操作数（Indirect operand）含有实际操作数值的地址值。AT&T 语法通过在操作数前加一个 '\*' 字符来指定一个间接操作数。只有调转/调用指令才能使用间接操作数。见下面对跳转指令的说明。

- 立即操作数前需要加一个 '\$' 字符前缀；
- 寄存器名前需要加一个 '%' 字符前缀；
- 内存操作数由变量名或者含有变量地址的一个寄存器指定。变量名隐含指出了变量的地址，并指示 CPU 引用该地址处内存的内容。

#### 3.2.3.1 指令操作码的命名

AT&T 语法中指令操作码名称（即指令助记符）最后一个字符用来指明操作数的宽度。字符 'b'、'w' 和 'l' 分别指定 byte、word 和 long 类型的操作数。如果指令名称没有带这样的字符后缀，并且指令语句中不含内存操作数，那么 as 就会根据目的寄存器操作数来尝试确定操作数宽度。例如指令语句 'mov %ax, %bx' 等同于 'movw %ax, %bx'。同样，语句 'mov \$1, %bx' 等同于 'movw \$1, %bx'。

AT&T 与 Intel 语法中几乎所有指令操作码的名称都相同，但仍有几个例外。符号扩展和零扩展指令都需要 2 个宽度来指明，即需要为源和目的操作数指明宽度。AT&T 语法中是通过使用两个操作码后缀来做到。AT&T 语法中符号扩展和零扩展的基本操作码名称分别是 'movs...' 和 'movz...'，Intel 中分别是 'movsx' 和 'movzx'。两个后缀就附在操作码基本名上。例如“使用符号扩展从 %al 移动到 %edx”的 AT&T 语句是 'movsbl %al, %edx'，即从 byte 到 long 是 bl、从 byte 到 word 是 bw、从 word 到 long 是 wl。AT&T 语法与 Intel 语法中转换指令的对应关系见表 3-2 所示。

表 3-2 AT&T 语法与 Intel 语法中转换指令的对应关系

AT&T	Intel	说明
cbtw	cbw	把 %al 中的字节值符号扩展到 %ax 中
cwtl	cwde	把 %ax 符号扩展到 %eax 中
cwtd	cwd	把 %ax 符号扩展到 %dx:%ax 中
cldt	cdq	把 %eax 符号扩展到 %edx:%eax 中

#### 3.2.3.2 指令操作码前缀

操作码前缀用于修饰随后的操作码。它们用于重复字符串指令、提供区覆盖、执行总线锁定操作、或指定操作数和地址宽度。通常操作码前缀可作为一条没有操作数的指令独占一行并且必须直接位于所影响指令之前，但是最好与它修饰的指令放在同一行上。例如，串扫描指令 'scas' 使用前缀执行重复操作：

---

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

---



操作码前缀有表 3-3 中列出的一些。

表 3-3 操作码前缀列表

操作码前缀	说明
cs, ds, ss, es, fs, gs	区覆盖操作码前缀。通过指定使用 区:内存操作数 内存引用形式会自动添加这种前缀。
data16, addr16	操作数/地址宽度前缀。这两个前缀会把 32 位操作数/地址改变成 16 位的操作数/地址。但请注意, as 并不支持 16 位寻址方式。
lock	总线锁存前缀。用于在指令执行期间禁止中断(仅对某些指令有效, 请参见 80X86 手册)。
wait	协处理器指令前缀。等待协处理器完成当前指令的执行。对于 80386/80387 组合用不着这个前缀。
rep, repe, repne	串指令操作前缀, 使串指令重复执行%ecx 中指定的次数。

### 3.2.3.3 内存引用

Intel 语法的间接内存引用形式: section:[base + index\*scale + disp]

对应于如下 AT&T 语法形式: section:disp(base, index, scale)

其中 base 和 index 是可选的 32 位基寄存器和索引寄存器, disp 是可选的偏移值。scale 是比例因子, 取值范围是 1、2、4 和 8。scale 其乘上索引 index 用来计算操作数地址。如果没有指定 scale, 则 scale 取默认值 1。section 为内存操作数指定可选的段寄存器, 并且会覆盖操作数使用的当前默认段寄存器。请注意, 如果指定的段覆盖寄存器与默认操作的段寄存器相同, 则 as 就不会为汇编的指令再输出相同的段前缀。以下是几个 AT&T 和 Intel 语法形式的内存引用例子:

movl var, %eax	# 把内存地址 var 处的内容放入寄存器%eax 中。
movl %cs:var, %eax	# 把代码段中内存地址 var 处的内容放入%eax 中。
movb \$0x0a, %es: (%ebx)	# 把字节值 0x0a 保存到 es 段的%ebx 指定的偏移处。
movl \$var, %eax	# 把 var 的地址放入%eax 中。
movl array(%esi), %eax	# 把 array+%esi 确定的内存地址处的内容放入%eax 中。
movl (%ebx, %esi, 4), %eax	# 把%ebx+%esi*4 确定的内存地址处的内容放入%eax 中。
movl array(%ebx, %esi, 4), %eax	# 把 array + %ebx+%esi*4 确定的内存地址处的内容放入%eax 中。
movl -4(%ebp), %eax	# 把 %ebp -4 内存地址处的内容放入%eax 中, 使用默认段%ss。
movl foo(, %eax, 4), %eax	# 把内存地址 foo + eax * 4 处内容放入%eax 中, 使用默认段%ds。

### 3.2.3.4 跳转指令

跳转指令用于把执行点转移到程序另一个位置处继续执行下去。这些跳转的目的位置通常使用一个标号来表示。在生成目标代码文件时, 汇编器会确定所有带有标号的指令的地址, 并且把跳转到的指令的地址编码到跳转指令中。跳转指令可分为无条件跳转和条件跳转两大类。条件跳转指令将依赖于执行指令时标志寄存器中某个相关标志的状态来确定是否进行跳转, 而无条件跳转则不依赖于这些标志。

JMP 是无条件跳转指令, 并可分为直接(direct)跳转和间接(indirect)跳转两类, 而条件跳转指令只有直接跳转的形式。对于直接跳转指令, 跳转到的目标指令的地址是作为跳转指令的一部份直接编码进跳转指令中; 对于间接跳转指令, 跳转的目的位置取自于某个寄存器或某个内存位置中。直接跳转语句的写法是给出跳转目标处的标号; 间接跳转语句的写法是必须使用一个星字符' \* '作为操作指示符的前缀字符, 并且该操作指示符使用 movl 指令相同的语法。下面是直接和间接跳转的几个例子。

jmp NewLoc	# 直接跳转。无条件直接跳转到标号 NewLoc 处继续执行。
------------	---------------------------------

---

```

jmp %eax           # 间接跳转。寄存器%eax 的值是跳转的目标位置。
jmp *(%eax)        # 间接跳转。从%eax 指明的地址处读取跳转的目标位置。

```

---

同样，与指令计数器 PC<sup>4</sup> 无关的间接调用的操作数也必须有一个 ‘\*’ 作为前缀字符。若没有使用 ‘\*’ 字符，那么 as 汇编器就会选择与指令计数 PC 相关的跳转标号。还有，其他任何具有内存操作数的指令都必须使用操作码后缀（‘b’、‘w’ 或 ‘l’）指明操作数的大小（byte、word 或 long）。

### 3.2.4 区与重定位

区（Section）（也称为段、节或部分）用于表示一个地址范围，操作系统将会以相同的方式对待和处理在该地址范围中的数据信息。例如，可以有一个“只读”的区，我们只能从该区中读取数据而不能写入。区的概念主要用来表示编译器生成的目标文件（或可执行程序）中不同的信息区域，例如目标文件中的正文区或数据区。若要正确理解和编制一个 as 汇编语言程序，我们就需要了解 as 产生的输出目标文件的格式安排。有关 Linux 0.11 内核使用的 a.out 格式目标文件格式的详细说明将在本章后面给出，这里首先对区的基本概念作一简单介绍，以理解 as 汇编器产生的目标文件基本结构。

链接器 ld 会把输入的目标文件中的内容按照一定规律组合生成一个可执行程序。当 as 汇编器输出一个目标文件时，该目标文件中的代码被默认设置成从地址 0 开始。此后 ld 将会在链接过程中为不同目标文件中的各个部分分配不同的最终地址位置。ld 会把程序中的字节块移动到程序运行时的地址处。这些块是作为固定单元进行移动的。它们的长度以及字节次序都不会被改变。这样的固定单元就被称作是区（或段、部分）。而为区分配运行时刻的地址的操作就被称为重定位（Relocation）操作，其中包括调整目标文件中记录的地址，从而让它们对应到恰当的运行时刻地址上。

as 汇编器输出产生的目标文件中至少具有 3 个区，分别被称为正文（text）、数据（data）和 bss 区。每个区都可能是空的。如果没有使用汇编命令把输出放置在 ‘.text’ 或 ‘.data’ 区中，这些区会仍然存在，但内容是空的。在一个目标文件中，其 text 区从地址 0 开始，随后是 data 区，再后面是 bss 区。

当一个区被重定位时，为了让链接器 ld 知道哪些数据会发生变化以及如何修改这些数据，as 汇编器也会往目标文件中写入所需要的重定位信息。为了执行重定位操作，在每次涉及目标文件中的一个地址时，ld 必须知道：

- 目标文件中对一个地址的引用是从什么地方算起的？
- 该引用的字节长度是多少？
- 该地址引用的是哪个区？（地址）-（区的开始地址）的值等于多少？
- 对地址的引用与程序计数器 PC（Program-Counter）相关吗？

实际上，as 使用的所有地址都可表示为：（区）+（区中偏移）。另外，as 计算的大多数表达式都有这种与区相关的特性。在下面说明中，我们使用记号 “{secname N}” 来表示区 secname 中偏移 N。

除了 text、data 和 bss 区，我们还需要了解绝对地址区（absolute 区）。当链接器把各个目标文件组合在一起时，absolute 区中的地址将始终不变。例如，ld 会把地址 {absolute 0} “重定位” 到运行时刻地址 0 处。尽管链接器在链接后决不会把两个目标文件中的 data 区安排成重叠地址处，但是目标文件中的 absolute 区必会重叠而覆盖。

另外还有一种名为“未定义的”区（Undefined section）。在汇编时不能确定所在区的任何地址都被设置成 {undefined U}，其中 U 将会在以后填上。因为数值总是有定义的，所以出现未定义地址的唯一途径仅涉及未定义的符号。对一个称为公共块（common block）的引用就是这样一种符号：在汇编时它的值未知，因此它在 undefined 区中。

类似地，区名也用于描述已链接程序中区的组。链接器 ld 会把程序所有目标文件中的 text 区放在相邻的地址处。我们习惯上所说的程序的 text 区实际上是指其所有目标文件 text 区组合构成的整个地址区域。对程序中 data 和 bss 区的理解也同样如此。

---

<sup>4</sup>这里符号 PC 是指 CPU 的程序指令指针计数器（Program Counter）。

### 3.2.4.1 链接器涉及的区

链接器 ld 只涉及如下 4 类区：

- text 区、data 区 — 这两个区用于保存程序。as 和 ld 会分别独立而同等地对待它们。对其中 text 区的描述也同样适合于 data 区。然而当程序在运行时，则通常 text 区是不会改变的。text 区通常会被进程共享，其中含有指令代码和常数等内容。程序运行时 data 区的内容通常是会变化的，例如，C 变量一般就存放在 data 区中。
- bss 区 — 在程序开始运行时这个区中含有 0 值字节。该区用于存放未初始化的变量或作为公共变量存储空间。虽然程序每个目标文件 bss 区的长度信息很重要，但是由于该区中存放的是 0 值字节，因此无须在目标文件中保存 bss 区。设置 bss 区的目的是为了从目标文件中明确地排除 0 值字节。
- absolute 区 — 该区的地址 0 总是“重定位”到运行时刻地址 0 处。如果你不想让 ld 在重定位操作时改变你所引用的地址，那么就使用这个区。从这种观点来看，我们可以把绝对地址称作是“不可重定位的”：在重定位操作期间它们不会改变。
- undefined 区 — 对不在先前所述各个区中对象的地址引用都属于本区。

图 3-2 中是 3 个理想化的可重定位区的例子。这个例子使用传统的区名称：'.text' 和 '.data'。其中水平轴表示内存地址。后面小节中将会详细说明 ld 链接器的具体操作过程。

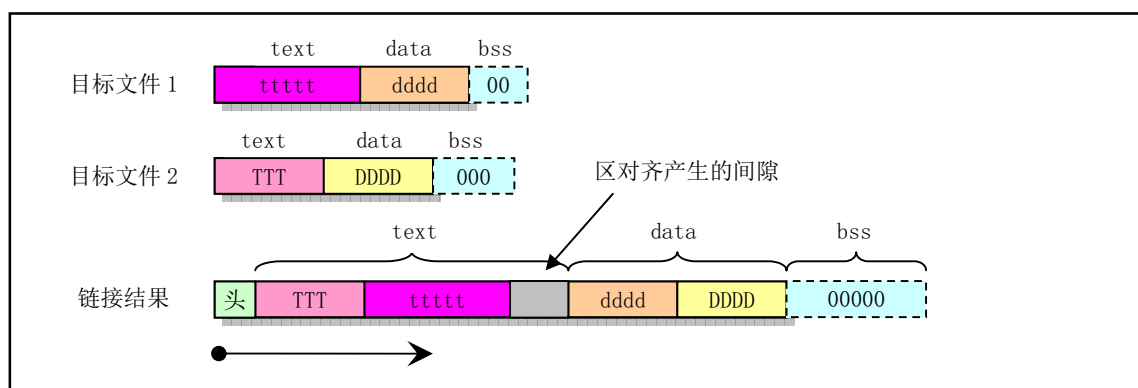


图 3-2 链接两个目标文件产生已链接程序的例子

### 3.2.4.2 子区

汇编取得的字节数据通常位于 text 或 data 区中。有时候在汇编源程序某个区中可能分布着一些不相邻的数据组，但是你可以会想让他们在汇编后聚集在一起存放。as 汇编器允许你利用子区（subsection）来达到这个目的。在每个区中，可以有编号为 0--8192 的子区存在。编制在同一个子区中的对象会在目标文件中与该子区中其他对象放在一起。例如，编译器可能想把常数存放在 text 区中，但是不想让这些常数散布在被汇编的整个程序中。在这种情况下，编译器就可以在每个会输出的代码区之前使用 '.text 0' 子区，并且在每组会输出的常数之前使用 '.text 1' 子区。

使用子区是可选的。如果没有使用子区，那么所有对象都会被放在子区 0 中。子区会以其从小到大的编号顺序出现在目标文件中，但是目标文件中并不包含表示子区的任何信息。处理目标文件的 ld 以及其他程序并不会看到子区的踪迹，它们只会看到由所有 text 子区组成的 text 区；由所有 data 子区组成的 data 区。为了指定随后的语句被汇编到哪个子区中，可在 '.text 表达式' 或 '.data 表达式' 中使用数值参数。表达式结果应该是绝对值。如果只指定了 '.text'，那么就会默认使用 '.text 0'。同样地，'.data' 表示使用 '.data 0'。

每个区都有一个位置计数器（Location Counter），它会对每个汇编进该区的字节进行计数。由于子区仅供 as 汇编器使用方便而设置的，因此并不存在子区计数器。虽然没有什么直接操作一个位置计数器

的方法，但是汇编命令 `.align` 可以改变其值，并且任何标号定义都会取用位置计数器的当前值。正在执行语句汇编处理的区的位置计数器被称为当前活动计数器。

### 3.2.4.3 bss 区

bss 区用于存储局部公共变量。你可以在 bss 区中分配空间，但是在程序运行之前不能在其中放置数据。因为当程序刚开始执行时，bss 区中所有字节内容都将被清零。`.lcomm` 汇编命令用于在 bss 区中定义一个符号；`.comm` 可用于在 bss 区中声明一个公共符号。

## 3.2.5 符号

在程序编译和链接过程中，符号（Symbol）是一个比较重要的概念。程序员使用符号来命名对象，链接器使用符号进行链接操作，而调试器利用符号进行调试。

标号（Label）是后面紧随一个冒号的符号。此时该符号代表活动位置计数器的当前值，并且，例如，可作为指令的操作数使用。我们可以使用等号 `=` 给一个符号赋予任意数值。

符号名以一个字母或 `._` 字符之一开始。局部符号用于协助编译器和程序员临时使用名称。在一个程序中共有 10 个局部符号名（`'0'...``'9'`）可供重复使用。为了定义一个局部符号，只要写出形如 `'N:'` 的标号（其中 N 代表任何数字）。若是引用前面最近定义的这个符号，需要写成 `'Nb'`；若需引用下一个定义的局部标号，则需要写成 `'Nf'`。其中 `'b'` 意思是向后（backwards），`'f'` 表示向前（forwards）。局部标号在使用方面没有限制，但是在任何时候我们只能向前/向后引用最远 10 个局部标号。

### 3.2.5.1 特殊点符号

特殊符号 `'.'` 表示 as 汇编的当前地址。因此表达式 `mylab: .long .'` 就会把 mylab 定义为包含它自己所处的地址值。给 `'.'` 赋值就如同汇编命令 `.org` 的作用。因此表达式 `'.=. +4'` 与 `'space 4'` 完全相同。

### 3.2.5.2 符号属性

除了名字以外，每个符号都有“值”和“类型”属性。根据输出的格式不同，符号也可以具有辅助属性。如果不定义就使用一个符号，as 就会假设其所有属性均为 0。这指示该符号是一个外部定义的符号。

符号的值通常是 32 位的。对于标出 text、data、bss 或 absolute 区中一个位置的符号，其值是从区开始到标号处的地址值。对于 text、data 和 bss 区，一个符号的值通常会在链接过程中由于 ld 改变区的基地址而变化，absolute 区中符号的值不会改变。这也是为何称它们是绝对符号的原因。

ld 会对未定义符号的值进行特殊处理。如果未定义符号的值是 0，则表示该符号在本汇编源程序中没有定义，ld 会尝试根据其他链接的文件来确定它的值。在程序使用了一个符号但没有对符号进行定义，就会产生这样的符号。若未定义符号的值不为 0，那么该符号值就表示是 .comm 公共声明的需要保留的公共存储空间字节长度。符号指向该存储空间的第一个地址处。

符号的类型属性含有用于链接器和调试器的重定位信息、指示符号是外部的标志以及一些其他可选信息。对于 a.out 格式的目标文件，符号的类型属性存放在一个 8 位字段中（`n_type` 字节）。其含义请参见有关 include/a.out.h 文件的说明。

## 3.2.6 as 汇编命令

汇编命令是指示汇编器操作方式的伪指令。汇编命令用于要求汇编器为变量分配空间、确定程序开始地址、指定当前汇编的区、修改位置计数器值等。所有汇编命令的名称都以 `'.'` 开始，其余是字符，并且大小写无关。但是通常都使用小写字符。下面我们给出一些常用汇编命令的说明。

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

`.align` 是存储对齐汇编命令，用于在当前子区中把位置计数器值设置（增加）到下一个指定存储边界处。第 1 个绝对值表达式 `abs-expr1`（absolute expression）指定要求的边界对齐值。对于使用 a.out 格式目标文件的 80X86 系统，该表达式值是位置计数器值增加后其二进制值最右面 0 值位的个数，即是 2 的次方值。例如，`'align 3'` 表示把位置计数器值增加到 8 的倍数上。如果位置计数器值本身就是 8 的倍数，那么就无需改变。但是对于使用 ELF 格式的 80X86 系统，该表达式值直接就是要求对其的字节数。例如

`'.align 8'`就是把位置计数器值增加到 8 的倍数上。

第 2 个表达式给出用于对齐而填充的字节值。该表达式与其前面的逗号可以省略。若省略，则填充字节值是 0。第 3 个可选表达式 `abs-expr3` 用于指示对齐操作允许填充跳过的最大字节数。如果对齐操作要求跳过的字节数大于这个最大值，那么该对齐操作就被取消。若想省略第 2 个参数，可以在第 1 和第 3 个参数之间使用两个逗号。

### 3.2.6.2 .ascii "string"...

从位置计数器所值当前位置为字符串分配空间并存储字符串。可使用逗号分开写出多个字符串。例如，`'.ascii "Hello world!", "My assembler"'`。该汇编命令会让 `as` 把这些字符串汇编在连续的地址位置处，每个字符串后面不会自动添加 0 (NULL) 字节。

### 3.2.6.3 .asciz "string"...

该汇编命令与 `'.ascii'` 类似，但是每个字符串后面会自动添加 NULL 字符。

### 3.2.6.4 .byte expressions

该汇编命令定义 0 个或多个用逗号分开的字节值。每个表达式的值是一个字节。

### 3.2.6.5 .comm symbol, length

在 `bss` 区中声明一个命名的公共区域。在 `ld` 链接过程中，某个目标文件中的一个公共符号会与其他目标文件中同名的公共符号合并。如果 `ld` 没有找到一个符号的定义，而只是一个或多个公共符号，那么 `ld` 就会分配指定长度 `length` 字节的未初始化内存。`length` 必须是一个绝对值表达式，如果 `ld` 找到多个长度不同但同名的公共符号，`ld` 就会分配长度最大的空间。

### 3.2.6.6 .data subsection

该汇编命令通知 `as` 把随后的语句汇编到编号为 `subsection` 的 `data` 子区中。如果省略编号，则默认使用编号 0。编号必须是绝对值表达式。

### 3.2.6.7 .desc symbol, abs-expr

用绝对表达式的值设置符号 `symbol` 的描述符字段 `n_desc` 的 16 位值。仅用于 `a.out` 格式的目标文件。参见有关 `include/a.out.h` 文件的说明。

### 3.2.6.8 .fill repeat, size, value

该汇编命令会产生数个 (`repeat` 个) 大小为 `size` 字节的重复拷贝。大小值 `size` 可以为 0 或某个值，但是若 `size` 大于 8，则限定为 8。每个重复字节内容取自一个 8 字节数。高 4 字节为 0，低 4 字节是数值 `value`。这 3 个参数值都是绝对值，`size` 和 `value` 是可选的。如果第 2 个逗号和 `value` 省略，`value` 默认为 0 值；如果后两个参数都省略的话，则 `size` 默认为 1。

### 3.2.6.9 .global symbol (或者 .globl symbol)

该汇编命令会使得链接器 `ld` 能看见符号 `symbol`。如果在我们的目标文件中定义了符号 `symbol`，那么它的值将被链接过程中的其他目标文件使用。若目标文件中没有定义该符号，那么它的属性将从链接过程中其他目标文件的同名符号中获得。这是通过设置符号 `symbol` 类型字段中的外部位 `N_EXT` 来做到的。参见 `include/a.out.h` 文件中的说明。

### 3.2.6.10 .int expressions

该汇编命令在某个区中设置 0 个或多个整数值 (80386 系统为 4 字节，同 `.long`)。每个用逗号分开的表达式的值就是运行时刻的值。例如 `.int 1234, 567, 0x89AB`。

### 3.2.6.11 .lcomm symbol, length

为符号 `symbol` 指定的局部公共区域保留长度为 `length` 字节的空间。所在的区和符号 `symbol` 的值是新的局部公共块的值。分配的地址在 `bss` 区中，因此在运行时刻这些字节值被清零。由于符号 `symbol` 没有被声明为全局的，因此链接器 `ld` 看不见。

### 3.2.6.12 .long expressions

含义与 `.int` 相同。

### 3.2.6.13 .octa bignums

这个汇编命令指定 0 个或多个用逗号分开的 16 字节大数 (.byte, .word, .long, .quad, .octa 分别对应 1、2、4、8 和 16 字节数)。

### 3.2.6.14 .org new\_lc, fill

这个汇编命令会把当前区的位置计数器设置为值 new\_lc。new\_lc 是一个绝对值 (表达式), 或者是具有相同区作为子区的表达式, 也即不能使用 .org 跨越各区。如果 new\_lc 的区不对, 那么 .org 就不会起作用。请注意, 位置计数器是基于区的, 即以每个区作为计数起点。

当位置计数器值增长时, 所跳跃过的字节将被填入值 fill。该值必须是绝对值。如果省略了逗号和 fill, 则 fill 默认为 0 值。

### 3.2.6.15 .quad bignums

这个汇编命令指定 0 个或多个用逗号分开的 8 字节大数 bignum。如果大数放不进 8 个字节中, 则取低 8 个字节。

### 3.2.6.16 .short expressions (同.word expressions)

这个汇编命令指定某个区中 0 个或多个用逗号分开的 2 字节数。对于每个表达式, 在运行时刻都会产生一个 16 位的值。

### 3.2.6.17 .space size, fill

该汇编命令产生 size 个字节, 每个字节填值 fill。这个参数均为绝对值。如果省略了逗号和 fill, 那么 fill 的默认值就是 0。

### 3.2.6.18 .string "string"

定义一个或多个用逗号分开的字符串。在字符串中可以使用转义字符。每个字符串都自动附加一个 NULL 字符结尾。例如, .string "\n\nStarting", "other strings"。

### 3.2.6.19 .text subsection

通知 as 把随后的语句汇编进编号为 subsection 的子区中。如果省略了编号 subsection, 则使用默认编号值 0。

### 3.2.6.20 .word expressions

对于 32 位机器, 该汇编命令含义与 .short 相同。

## 3.2.7 编写 16 位代码

虽然 as 通常用来编写纯 32 位的 80X86 代码, 但是 1995 年后它对编写运行于实模式或 16 位保护模式的代码也提供有限的支持。为了让 as 汇编时产生 16 位代码, 需要在运行于 16 位模式的指令语句之前添加汇编命令 '.code16', 并且使用汇编命令 '.code32' 让 as 汇编器切换回 32 位代码汇编方式。

as 不区分 16 位和 32 位汇编语句, 在 16 位和 32 位模式下每条指令的功能完全一样而与模式无关。as 总是为汇编语句产生 32 位的指令代码而不管指令将运行在 16 位还是 32 位模式下。如果使用汇编命令 '.code16' 让 as 处于 16 位模式下, 那么 as 会自动为所有指令加上一个必要的操作数宽度前缀而让指令运行在 16 位模式。请注意, 因为 as 为所有指令添加了额外的地址和操作数宽度前缀, 所以汇编产生的代码长度和性能上将会受到影响。

由于在 1991 年开发 Linux 内核 0.11 时 as 汇编器还不支持 16 位代码, 因此在编写和汇编 0.11 内核实模式下的引导启动代码和初始化汇编程序时使用了前面介绍的 as86 汇编器。

## 3.2.8 AS 汇编器命令行选项

- a 开启程序列表
- f 快速操作
- o 指定输出的目标文件名

-R 组合数据区和代码区  
-W 取消警告信息

## 3.3 C 语言程序

GNU gcc 对 ISO 标准 C89 描述的 C 语言进行了一些扩展，其中一些扩展部分已经包括进 ISO C99 标准中。本节给出内核中经常用到的一些 gcc 扩充语句的说明。在后面章节程序注释中也会随时对遇到的扩展语句给出简单的说明。

### 3.3.1 C 程序编译和链接

使用 gcc 编译器编译 C 语言程序时通常会经过四个处理阶段，即预处理阶段、编译阶段、汇编阶段和链接阶段，见图 3-3 所示。

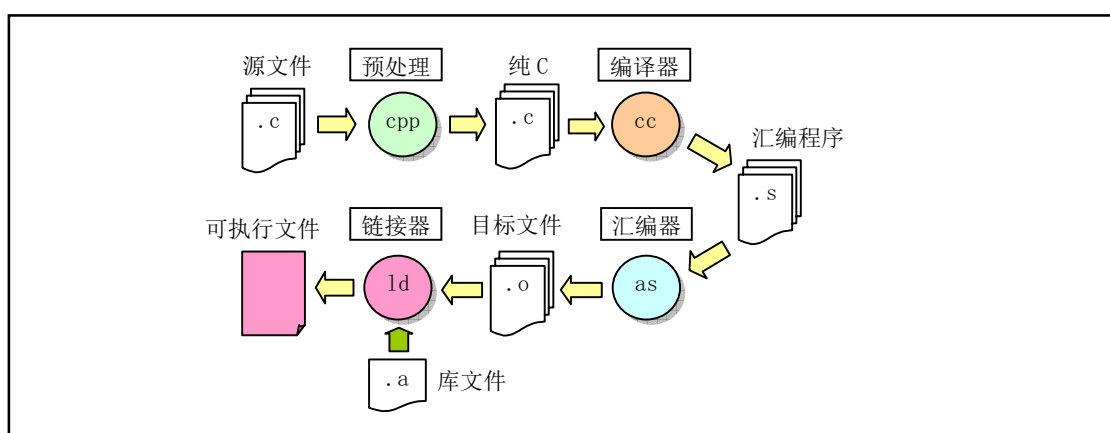


图 3-3 C 程序编译过程

在前处理阶段中，gcc 会把 C 程序传递给 C 前处理器 CPP，对 C 语言程序中指示符和宏进行替换处理，输出纯 C 语言代码；在编译阶段，gcc 把 C 语言程序编译生成对应的与机器相关的 as 汇编语言代码；在汇编阶段，as 汇编器会把汇编代码转换成机器指令，并以特定二进制格式输出保存在目标文件中；最后 GNU ld 链接器把程序的相关目标文件组合链接在一起，生成程序的可执行映像文件。调用 gcc 的命令行格式与编译汇编语言的格式类似：

---

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

---

其中 infile 是输入的 C 语言文件；outfile 是编译产生的输出文件。对于某次编译过程，并非一定要全部执行这四个阶段，使用命令行选项可以令 gcc 编译过程在某个处理阶段后就停止执行。例如，使用 '-S' 选项可以让 gcc 在输出了 C 程序对应的汇编语言程序之后就停止运行；使用 '-c' 选项可以让 gcc 只生成目标文件而不执行链接处理，见如下所示。

---

```
gcc -o hello hello.c           // 编译 hello.c 程序，生成执行文件 hello。
gcc -S -o hello.s hello.c      // 编译 hello.c 程序，生成对应汇编程序 hello.s。
gcc -c -o hello.o hello.c      // 编译 hello.c 程序，生成对应目标文件 hello.o 而不链接。
```

---

在编译象 Linux 内核这样的包含很多源程序文件的大型程序时，通常使用 make 工具软件对整个程序的编译过程进行自动管理，详见后面说明。



### 3.3.2 嵌入汇编

本节介绍内核 C 语言程序中接触到的嵌入式汇编（内联汇编）语句。由于我们通常编制 C 程序过程中一般很少用到嵌入式汇编代码，因此这里有必要对其基本格式和使用方法进行说明。具有输入和输出参数的嵌入汇编语句的基本格式为：

---

```
asm(“汇编语句”
    : 输出寄存器
    : 输入寄存器
    : 会被修改的寄存器);
```

---

除第 1 行以外，后面带冒号的行若不使用就都可以省略。其中，“asm”是内联汇编语句关键词；“汇编语句”是你写汇编指令的地方；“输出寄存器”表示当这段嵌入汇编执行完之后，哪些寄存器用于存放输出数据。此地，这些寄存器会分别对应一 C 语言表达式值或一个内存地址；“输入寄存器”表示在开始执行汇编代码时，这里指定的一些寄存器中应存放的输入值，它们也分别对应着一 C 变量或常数值。“会被修改的寄存器”表示你已对其中列出的寄存器中的值进行了改动，gcc 编译器不能再依赖于它原先对这些寄存器加载的值。如果必要的话，gcc 需要重新加载这些寄存器。因此我们需要把那些没有在输出/输入寄存器部分列出，但是在汇编语句中明确使用到或隐含使用到的寄存器名列在这个部分中。

下面我们用例子来说明嵌入汇编语句的使用方法。这里列出了 kernel/traps.c 文件中第 22 行开始的一段代码作为例子来详细解说。为了能看得更清楚一些，我们对这段代码进行了重新排列和编号。

---

```
01 #define get_seg_byte(seg, addr) \
02 ({ \
03     register char __res; \
04     __asm__("push %%fs; \
05             mov %%ax, %%fs; \
06             movb %%fs:%2, %%al; \
07             pop %%fs" \
08             : "=a" (__res) \
09             : "0" (seg), "m" (*(addr))); \
10     __res;})
```

---

这段 10 行代码定义了一个嵌入汇编语言宏函数。通常使用汇编语句最方便的方式是把它们放在一个宏内。用圆括号括住的组合语句（花括号中的语句）：“{ }”可以作为表达式使用，其中最后一行上的变量 \_\_res（第 10 行）是该表达式的输出值，见下一节说明。

因为宏语句需要定义在一行上，因此这里使用反斜杠“\”将这些语句连成一行。这条宏定义将被替换到程序中引用该宏名称的地方。第 1 行定义了宏的名称，也即是宏函数名称 get\_seg\_byte(seg, addr)。第 3 行定义了一个寄存器变量 \_\_res。该变量将被保存在一个寄存器中，以便于快速访问和操作。如果想指定寄存器（例如 eax），那么我们可以把该句写成“register char \_\_res asm (“ax”);”，其中“asm”也可以写成“\_\_asm\_\_”。第 4 行上的“\_\_asm\_\_”表示嵌入汇编语句的开始。从第 4 行到第 7 行的 4 条语句是 AT&T 格式的汇编语句。另外，为了让 gcc 编译产生的汇编语言程序中寄存器名称前有一个百分号“%”，在嵌入汇编语句寄存器名称前就必须写上两个百分号“%%”。

第 8 行即是输出寄存器，这句的含义是在这段代码运行结束后将 eax 所代表的寄存器的值放入 \_\_res 变量中，作为本函数的输出值，“=a”中的“a”称为加载代码，“=”表示这是输出寄存器，并且其中的值将被输出值替代。第 9 行表示在这段代码开始运行时将 seg 放到 eax 寄存器中，“0”表示使用与上面同个位置的输出相同的寄存器。而 (\*(addr)) 表示一个内存偏移地址值。为了在上面汇编语句中使用该地址值，嵌入汇



编程规定把输出和输入寄存器统一按顺序编号，顺序是从输出寄存器序列从左到右从上到下以"%0"开始，分别记为%0、%1、...%9。因此，输出寄存器的编号是%0（这里只有一个输出寄存器），输入寄存器前一部分("%0" (seg))的编号是%1，而后部分的编号是%2。上面第 6 行上的%2 即代表(\*(addr))这个内存偏移量。

现在我们来研究 4—7 行上的代码的作用。第一句将 fs 段寄存器的内容入栈；第二句将 eax 中的段值赋给 fs 段寄存器；第三句是把 fs:(\*(addr))所指定的字节放入 al 寄存器中。当执行完汇编语句后，输出寄存器 eax 的值将被放入 \_\_res，作为该宏函数（块结构表达式）的返回值。很简单，不是吗？

通过上面分析，我们知道，宏名称中的 seg 代表一指定的内存段值，而 addr 表示一内存偏移地址量。到现在为止，我们应该很清楚这段程序的功能了吧！该宏函数的功能是从指定段和偏移值的内存地址处取一个字节。再看下一个例子。

```

01  asm("cld\n\t"
02      "rep\n\t"
03      "stol"
04      : /* 没有输出寄存器 */
05      : "c"(count-1), "a"(fill_value), "D"(dest)
06      : "%ecx", "%edi");

```

1-3 行这三句是通常的汇编语句，用以清方向位，重复保存值。其中头两行中的字符"\n\t"是用于 gcc 预处理程序输出程序列表时能排的整齐而设置的，字符的含义与 C 语言中的相同。即 gcc 的运作方式是先产生与 C 程序对应的汇编程序，然后调用汇编器对其进行编译产生目标代码，如果在写程序和调试程序时想看看 C 对应的汇编程序，那么就需要得到预处理程序输出的汇编程序结果(这在编写和调试高效的代码时常用的做法)。为了预处理输出的汇编程序格式整齐，就可以使用"\n\t"这两个格式符号。

第 4 行说明这段嵌入汇编程序没有用到输出寄存器。第 5 行的含义是：将 count-1 的值加载到 ecx 寄存器中（加载代码是"c"），fill\_value 加载到 eax 中，dest 放到 edi 中。为什么要让 gcc 编译程序去做这样的寄存器值的加载，而不让我们自己做呢？因为 gcc 在它进行寄存器分配时可以进行某些优化工作。例如 fill\_value 值可能已经在 eax 中。如果是在一个循环语句中的话，gcc 就可能在整个循环操作中保留 eax，这样就可以在每次循环中少用一个 movl 语句。

最后一行的作用是告诉 gcc 这些寄存器中的值已经改变了。在 gcc 知道你拿这些寄存器做些什么后，能够对 gcc 的优化操作有所帮助。表 3-4 中是一些你可能会用到的寄存器加载代码及其具体的含义。

表 3-4 常用寄存器加载代码说明

代码	说明	代码	说明
a	使用寄存器 eax	m	使用内存地址
b	使用寄存器 ebx	o	使用内存地址并可以加偏移值
c	使用寄存器 ecx	I	使用常数 0-31
d	使用寄存器 edx	J	使用常数 0-63
S	使用 esi	K	使用常数 0-255
D	使用 edi	L	使用常数 0-65535
q	使用动态分配字节可寻址寄存器 (eax、ebx、ecx 或 edx)	M	使用常数 0-3
r	使用任意动态分配的寄存器	N	使用 1 字节常数 (0-255)
g	使用通用有效的地址即可 (eax、ebx、ecx、edx 或内存变量)	O	使用常数 0-31

A	使用 <code>eax</code> 与 <code>edx</code> 联合(64 位)	=	输出操作数。输出值将替换前值
+	表示操作数可读可写	&	早期会变的 (earlyclobber) 操作数。表示在使用完操作数之前, 内容会被修改

下面的例子不是让你自己指定哪个变量使用哪个寄存器, 而是让 `gcc` 为你选择。

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

指令 `"leal"` 用于计算有效地址, 但这里用它来进行一些简单计算。第 1 条汇编语句 `"leal (r1, r2, 4), r3"` 语句表示  $r1+r2*4 \rightarrow r3$ 。这个例子可以非常快地将 `x` 乘 5。其中 `"%0"`、`"%1"` 是指 `gcc` 自动分配的寄存器。这里 `"%1"` 代表输入值 `x` 要放入的寄存器, `"%0"` 表示输出值寄存器。输出寄存器代码前一定要加等于号。如果输入寄存器的代码是 0 或为空时, 则说明使用与相应输出一样的寄存器。所以, 如果 `gcc` 将 `r` 指定为 `eax` 的话, 那么上面汇编语句的含义即为:

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

注意: 在执行代码时, 如果不希望汇编语句被 `GCC` 优化而作修改, 就需要在 `asm` 符号后面添加关键词 `volatile`, 见下面所示。这两种声明的区别在于程序兼容性方面。建议使用后一种声明方式。

```
asm volatile (.....);
或者更详细的说明为:
__asm__ __volatile__ (.....);
```

关键词 `volatile` 也可以放在函数名前来修饰函数, 用来通知 `gcc` 编译器该函数不会返回。这样就可以让 `gcc` 产生更好一些的代码。另外, 对于不会返回的函数, 这个关键词也可以用来避免 `gcc` 产生假警告信息。例如 `mm/memory.c` 中的如下语句说明函数 `do_exit()` 和 `oom()` 不会再返回到调用者代码中:

```
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 }
```

下面再例举一个较长的例子, 如果能看得懂, 那就说明嵌入汇编代码对你来说基本没问题了。这段代码是从 `include/string.h` 文件中摘取的, 是 `strncmp()` 字符串比较函数的一种实现。同样, 其中每行中的 `"\n\t"` 是用于 `gcc` 预处理程序输出列表好看而设置的。

```
//// 字符串 1 与字符串 2 的前 count 个字符进行比较。
// 参数: cs - 字符串 1, ct - 字符串 2, count - 比较的字符数。
// %0 - eax(__res)返回值, %1 - edi(cs)串 1 指针, %2 - esi(ct)串 2 指针, %3 - ecx(count)。
// 返回: 如果串 1 > 串 2, 则返回 1; 串 1 = 串 2, 则返回 0; 串 1 < 串 2, 则返回-1。
extern inline int strncmp(const char * cs, const char * ct, int count)
```

---

```

{
register int __res ;           // __res 是寄存器变量。
__asm__ ("cld\n"             // 清方向位。
        "1:\tdecl %3\n\t"    // count--。
        "js 2f\n\t"          // 如果 count<0, 则向前跳转到标号 2。
        "lods b\n\t"         // 取串 2 的字符 ds:[esi]→al, 并且 esi++。
        "scas b\n\t"         // 比较 al 与串 1 的字符 es:[edi], 并且 edi++。
        "jne 3f\n\t"         // 如果不相等, 则向前跳转到标号 3。
        "testb %%al, %%al\n\t" // 该字符是 NULL 字符吗?
        "jne 1b\n\t"         // 不是, 则向后跳转到标号 1, 继续比较。
        "2:\txorl %%eax, %%eax\n\t" // 是 NULL 字符, 则 eax 清零 (返回值)。
        "jmp 4f\n\t"         // 向前跳转到标号 4, 结束。
        "3:\tmovl $1, %%eax\n\t" // eax 中置 1。
        "jl 4f\n\t"          // 如果前面比较中串 2 字符<串 1 字符, 则返回 1, 结束。
        "negl %%eax\n\t"     // 否则 eax = -eax, 返回负值, 结束。
        "4:"
        : "=a" (__res) : "D" (cs), "S" (ct), "c" (count) : "si", "di", "cx");
return __res;                 // 返回比较结果。
}

```

---

### 3.3.3 圆括号中的组合语句

花括号对 “{...}” 用于把变量声明和语句组合成一个复合语句（组合语句）或一个语句块，这样在语义上这些语句就等同于一条语句。组合语句的右花括号后面不需要使用分号。圆括号中的组合语句，即形如 “({...})” 的语句，可以在 GNU C 中用作一个表达式使用。这样就可以在表达式中使用 loop、switch 语句和局部变量，因此这种形式的语句通常称为语句表达式。语句表达式具有如下示例的形式：

---

```

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

```

---

其中组合语句中最后一条语句必须是后面跟随一个分号的表达式。这个表达式 (“3 + z”) 的值即用作整个圆括号括住语句的值。如果最后一条语句不是表达式，那么整个语句表达式就具有 void 类型，因此没有值。另外，这种表达式中语句声明的任何局部变量都会在整块语句结束后失效。这个示例语句可以象如下形式的赋值语句来使用：

---

```

res = x + ({略...}) + b;

```

---

当然，人们通常不会象上面这样写语句，这种语句表达式通常都用来定义宏。例如内核源代码 init/main.c 程序中读取 CMOS 时钟信息的宏定义：

---

```

69 #define CMOS_READ(addr) ({ \           // 最后反斜杠起连接两行语句的作用。
70  outb_p(0x80|addr, 0x70); \           // 首先向 I/O 端口 0x70 输出欲读取的位置 addr。
71  inb_p(0x71); \                       // 然后从端口 0x71 读入该位置处的值作为返回值。
72 })

```

---

再看一个 include/asm/io.h 头文件中的读 I/O 端口 port 的宏定义，其中最后变量 \_v 的值就是 inb() 的返

回值。

---

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

---

### 3.3.4 寄存器变量

GNU 对 C 语言的另一个扩充是允许我们把一些变量值放到 CPU 寄存器中，即所谓寄存器变量。这样 CPU 就不用经常花费较长时间访问内存去取值。寄存器变量可以分为 2 种：全局寄存器变量和局部寄存器变量。全局寄存器变量会在程序的整个运行过程中保留寄存器专门用于几个全局变量。相反，局部寄存器变量不会保留指定的寄存器，而仅在内嵌 `asm` 汇编语句中作为输入或输出操作数时使用专门的寄存器。`gcc` 编译器的数据流分析功能本身有能力确定指定的寄存器何时含有正在使用的值，何时可派其他用场。当 `gcc` 数据流分析功能认为存储在某个局部寄存器变量值无用时就可能会删除之，并且对局部寄存器变量的引用也可能被删除、移动或简化。因此，若不想让 `gcc` 作这些优化改动，最好在 `asm` 语句中加上 `volatile` 关键词。

如果想在嵌入汇编语句中把汇编指令的输出直接写到指定的寄存器中，那么此时使用局部寄存器变量就很方便。由于 Linux 内核中通常只使用局部寄存器变量，因此这里我们只对局部寄存器变量的使用方法进行讨论。在 GNU C 程序中我们可以在函数中用如下形式定义一个局部寄存器变量：

---

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

---

这里 `ax` 是变量 `res` 所希望使用的寄存器。定义这样一个寄存器变量并不会专门保留这个寄存器不派其他用途。在程序编译过程中，当 `gcc` 数据流控制确定变量的值已经不用时就可能将该寄存器派作其他用途，而且对它的引用可能会被删除、移动或被简化。另外，`gcc` 并不保证所编译出的代码会把变量一直放在指定的寄存器中。因此在嵌入汇编的指令部分最好不要明确地引用该寄存器并且假设该寄存器肯定引用的是该变量值。然而把该变量用作为 `asm` 的操作数还是能够保证指定的寄存器被用作该操作数。

### 3.3.5 内联函数

在程序中，通过把一个函数声明为内联（`inline`）函数，就可以让 `gcc` 把函数的代码集成到调用该函数的代码中去。这样处理可以去掉函数调用时进入/退出时间开销，从而肯定能够加快执行速度。因此把一个函数声明为内联函数的主要目的就是能够尽量快速的执行函数体。另外，如果内联函数中有常数值，那么在编译期间 `gcc` 就可能用它来进行一些简化操作，因此并非所有内联函数的代码都会被嵌入进去。内联函数方法对程序代码的长度影响并不明显。使用内联函数的程序编译产生的目标代码可能会长一些也可能短一些，这需要根据具体情况来定。

内联函数嵌入调用者代码中的操作是一种优化操作，因此只有进行优化编译时才会执行代码嵌入处理。若编译过程中没有使用优化选项“-O”，那么内联函数的代码就不会被真正地嵌入到调用者代码中，而是只作为普通函数调用来处理。把一个函数声明为内联函数的方法是在函数声明中使用关键词“`inline`”，例如内核文件 `fs/inode.c` 中的如下函数：

---

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

---

04 }

函数中的某些语句用法可能会使得内联函数的替换操作无法正常进行，或者不适合进行替换操作。例如使用了可变参数、内存分配函数 `malloc()`、可变长度数据类型变量、非局部 `goto` 语句、以及递归函数。编译时可以使用选项 `-Winline` 让 `gcc` 对标志成 `inline` 但不能被替换的函数给出警告信息以及不能替换的原因。

当在一个函数定义中既使用 `inline` 关键词、又使用 `static` 关键词，即像下面文件 `fs/inode.c` 中的内联函数定义一样，那么如果所有对该内联函数的调用都被替换而集成在调用者代码中，并且程序中没有引用过该内联函数的地址，则该内联函数自身的汇编代码就不会被引用过。在这种情况下，除非我们在编译过程中使用选项 `-fkeep-inline-functions`，否则 `gcc` 就不会再为该内联函数自身生成实际汇编代码。由于某些原因，一些对内联函数的调用并不能被集成到函数中去。特别是在内联函数定义之前的调用语句是不会被替换集成的，并且也都不能是递归定义的函数。如果存在一个不能被替换集成的调用，那么内联函数就会象平常一样被编译成汇编代码。当然，如果程序中有引用内联函数地址的语句，那么内联函数也会象平常一样被编译成汇编代码。因为对内联函数地址的引用是不能被替换的。

---

```

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 }

```

---

请注意，内联函数功能已经被包括在 ISO 标准 C99 中，但是该标准定义的内联函数与 `gcc` 定义的有较大区别。ISO 标准 C99 的内联函数语义定义等同于这里使用组合关键词 `inline` 和 `static` 的定义，即“省略”了关键词 `static`。若在程序中需要使用 C99 标准的语义，那么就需要使用编译选项 `-std=gnu99`。不过为了兼容起见，在这种情况下还是最好使用 `inline` 和 `static` 组合。以后 `gcc` 将最终默认使用 C99 的定义，在希望仍然使用这里定义的语义时，就需要使用选项 `-std=gnu89` 来指定。

若一个内联函数的定义没有使用关键词 `static`，那么 `gcc` 就会假设其他程序文件中也对这个函数有调用。因为一个全局符号只能被定义一次，所以该函数就不能再在其他源文件中进行定义。因此这里对内联函数的调用就不能被替换集成。因此，一个非静态的内联函数总是会被编译出自己的汇编代码来。在这方面，ISO 标准 C99 对不使用 `static` 关键词的内联函数定义等同于这里使用 `static` 关键词的定义。

如果在定义一个函数时同时指定了 `inline` 和 `extern` 关键词，那么该函数定义仅用于内联集成，并且在任何情况下都不会单独产生该函数自身的汇编代码，即使明确引用了该函数的地址也不会产生。这样的一个地址会变成一个外部引用，就好像你仅仅声明了函数而没有定义函数一样。

关键词 `inline` 和 `extern` 组合在一起的作用几乎类同一个宏定义。使用这种组合方式就是把带有组合关键词的一个函数定义放在 `.h` 头文件中，并且把不含关键词的另一个相同函数定义放在一个库文件中。此时头文件中的定义会让绝大多数对该函数的调用被替换嵌入。如果还有没有被替换的对该函数的调用，那么就会使用（引用）程序文件中或库中的拷贝。Linux 0.1x 内核源代码中文件 `include/string.h`、`lib/strings.c` 就是这种使用方式的一个例子。例如 `string.h` 中定义了如下函数：

---

```

// 将字符串(src)拷贝到另一字符串(dest)，直到遇到 NULL 字符后停止。
// 参数：dest - 目的字符串指针，src - 源字符串指针。%0 - esi(src)，%1 - edi(dest)。
27 extern inline char * strcpy(char * dest, const char *src)
28 {
29     __asm__ ("cld\n"                                // 清方向位。

```

---

---

```

30      "l:\tlodsb\n\t"          // 加载 DS:[esi]处 1 字节→al, 并更新 esi。
31      "stosb\n\t"              // 存储字节 al→ES:[edi], 并更新 edi。
32      "testb %%al, %%al\n\t"    // 刚存储的字节是 0?
33      "jne 1b"                  // 不是则向后跳转到标号 1 处, 否则结束。
34      :: "S" (src), "D" (dest): "si", "di", "ax";
35  return dest;                  // 返回目的字符串指针。
36 }

```

---

而在内核函数库目录中, lib/strings.c 文件把关键词 inline 和 extern 都定义为空, 见如下所示。因此实际上就在内核函数库中又包含了 string.h 文件所有这类函数的一个拷贝, 即又对这些函数重新定义了一次, 并且“消除”了两个关键词的作用。

---

```

11 #define extern                // 定义为空。
12 #define inline                // 定义为空。
13 #define LIBRARY
14 #include <string.h>
15

```

---

此时库函数中重新定义的上述 strcpy()函数变成如下形式:

---

```

27 char * strcpy(char * dest, const char *src) // 去掉了关键词 inline 和 extern。
28 {
29     __asm__ ("cld\n\t"          // 清方向位。
30             "l:\tlodsb\n\t"      // 加载 DS:[esi]处 1 字节→al, 并更新 esi。
31             "stosb\n\t"          // 存储字节 al→ES:[edi], 并更新 edi。
32             "testb %%al, %%al\n\t" // 刚存储的字节是 0?
33             "jne 1b"              // 不是则向后跳转到标号 1 处, 否则结束。
34             :: "S" (src), "D" (dest): "si", "di", "ax");
35  return dest;                    // 返回目的字符串指针。
36 }

```

---

## 3.4 C 与汇编程序的相互调用

为了提高代码执行效率, 内核源代码中有地方直接使用了汇编语言编制。这就会涉及到在两种语言编制的程序之间的相互调用问题。本节首先说明 C 语言函数的调用机制, 然后使用示例来说明两者函数之间的调用方法。

### 3.4.1 C 函数调用机制

在 Linux 内核程序 boot/head.s 执行完基本初始化操作之后, 就会跳转去执行 init/main.c 程序。那么 head.s 程序是如何把执行控制转交给 init/main.c 程序的呢? 即汇编程序是如何调用执行 C 语言程序的? 这里我们首先描述一下 C 函数的调用机制、控制权传递方式, 然后说明 head.s 程序跳转到 C 程序的方法。

函数调用操作包括从一块代码到另一块代码之间的双向数据传递和执行控制转移。数据传递通过函数参数和返回值来进行。另外, 我们还需要在进入函数时为函数的局部变量分配存储空间, 并且在退出函数时收回这部分空间。Intel 80x86 CPU 为控制传递提供了简单的指令, 而数据的传递和局部变量存储空间的分配与回收则通过栈操作来实现。

### 3.4.1.1 栈帧结构和控制转移权方式

大多数 CPU 上的程序实现使用栈来支持函数调用操作。栈被用来传递函数参数、存储返回信息、临时保存寄存器原有值以备恢复以及用来存储局部数据。单个函数调用操作所使用的栈部分被称为栈帧（Stack frame）结构，其通常结构见图 3-4 所示。栈帧结构的两端由两个指针来指定。寄存器 `ebp` 通常用作帧指针（frame pointer），而 `esp` 则用作栈指针（stack pointer）。在函数执行过程中，栈指针 `esp` 会随着数据的入栈和出栈而移动，因此函数中对大部分数据的访问都基于帧指针 `ebp` 进行。

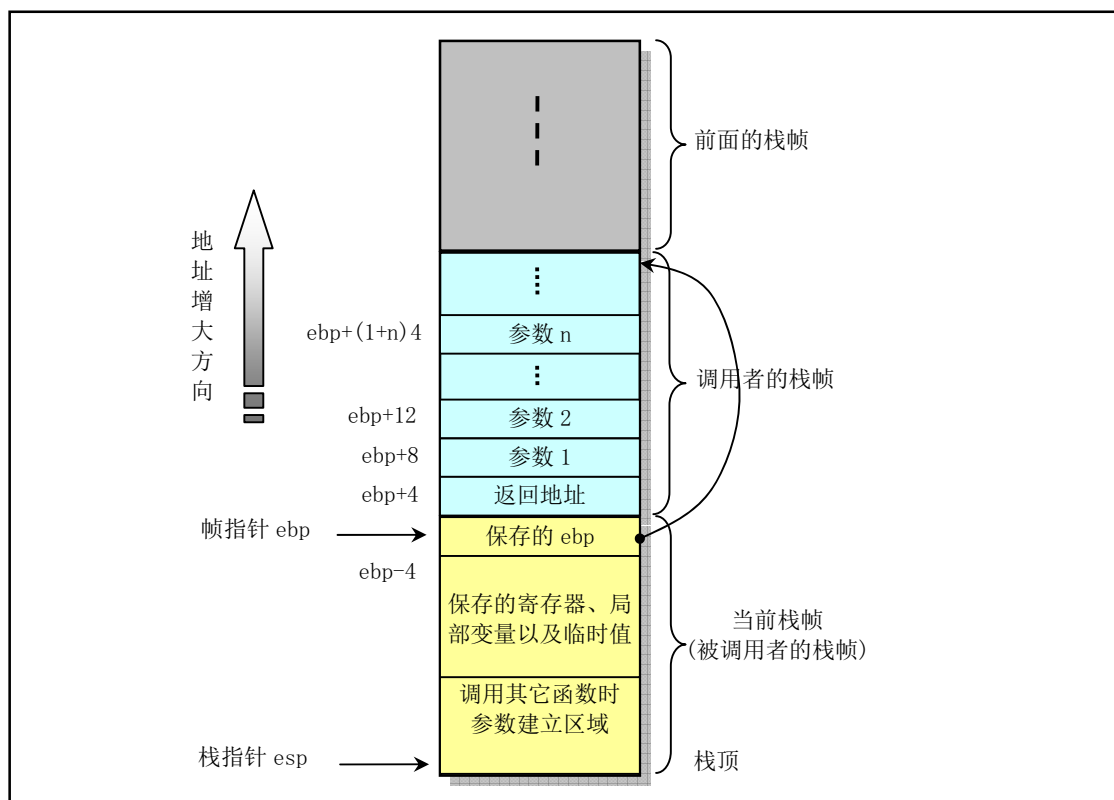


图 3-4 栈中帧结构示意图

对于函数 A 调用函数 B 的情况，传递给 B 的参数包含在 A 的栈帧中。当 A 调用 B 时，函数 A 的返回地址（调用返回后继续执行的指令地址）被压入栈中，栈中该位置也明确指明了 A 栈帧的结束处。而 B 的栈帧则从随后的栈部分开始，即图中保存帧指针（`ebp`）的地方开始。再随后则用于存放任何保存的寄存器值以及函数的临时值。

B 函数同样也使用栈来保存不能放在寄存器中的局部变量值。例如由于通常 CPU 的寄存器数量有限而不能够存放函数的所有局部数据，或者有些局部变量是数组或结构，因此必须使用数组或结构引用来进行访问。还有就是 C 语言的地址操作符‘&’被应用到一个局部变量上时，我们就需要为该变量生成一个地址，即为变量的地址指针分配一空间。最后，B 函数会使用栈来保存调用任何其它函数的参数。

栈是往低（小）地址方向扩展的，而 `esp` 指向当前栈顶处的元素。通过使用 `push` 和 `pop` 指令我们可以把数据压入栈中或从栈中弹出。对于没有指定初始值的数据所需要的存储空间，我们可以通过把栈指针递减适当的值来做到。类似地，通过增加栈指针值我们可以回收栈中已分配的空间。

指令 `CALL` 和 `RET` 用于处理函数调用和返回操作。调用指令 `CALL` 的作用是把返回地址压入栈中并且跳转到被调用函数开始处执行。返回地址是程序中紧随调用指令 `CALL` 后面一条指令的地址。因此当被调函数返回时就会从该位置继续执行。返回指令 `RET` 用于弹出栈顶处的地址并跳转到该地址处。在使用该指令之前，应该先正确处理栈中内容，使得当前栈指针所指位置内容正是先前 `CALL` 指令保存的返回地址。



另外，若返回值是一个整数或一个指针，那么寄存器 `eax` 将被默认用来传递返回值。

尽管某一时刻只有一个函数在执行，但我们还是需要确定在一个函数（调用者）调用其他函数（被调用者）时，被调用者不会修改或覆盖掉调用者今后要用到的寄存器内容。因此 Intel CPU 采用了所有函数必须遵守的寄存器用法统一惯例。该惯例指明，寄存器 `eax`、`edx` 和 `ecx` 的内容必须由调用者自己负责保存。当函数 B 被 A 调用时，函数 B 可以在不用保存这些寄存器内容的情况下任意使用它们而不会毁坏函数 A 所需要的任何数据。另外，寄存器 `ebx`、`esi` 和 `edi` 的内容则必须由被调用者 B 来保护。当被调用者需要使用这些寄存器中的任意一个时，必须首先在栈中保存其内容，并在退出时恢复这些寄存器的内容。因为调用者 A（或者一些更高层的函数）并不负责保存这些寄存器内容，但可能在以后的操作中还需要用到原先的值。还有寄存器 `ebp` 和 `esp` 也必须遵守第二个惯例用法。

### 3.4.1.2 函数调用举例

作为一个例子，我们来观察下面 C 程序 `exch.c` 中函数调用的处理过程。该程序交换两个变量中的值，并返回它们的差值。

---

```
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 }
```

---

其中函数 `swap()` 用于交换两个变量的值。C 程序中的主程序 `main()` 也是一个函数（将在下面说明），它在调用了 `swap()` 之后返回交换后的结果。这两个函数的栈帧结构见图 3-5 所示。可以看出，函数 `swap()` 从调用者（`main()`）的栈帧中获取其参数。图中的位置信息相对于寄存器 `ebp` 中的帧指针。栈帧左边的数字指出了相对于帧指针的地址偏移值。在象 `gdb` 这样的调试器中，这些数值都用 2 的补码表示。例如 `-4` 被表示成 `0xFFFFF4`，`-12` 会被表示成 `0xFFFFC`。

调用者 `main()` 的栈帧结构中包括局部变量 `a` 和 `b` 的存储空间，相对于帧指针位于 `-4` 和 `-8` 偏移处。由于我们需要为这两个局部变量生成地址，因此它们必须保存在栈中而非简单地存放在寄存器中。



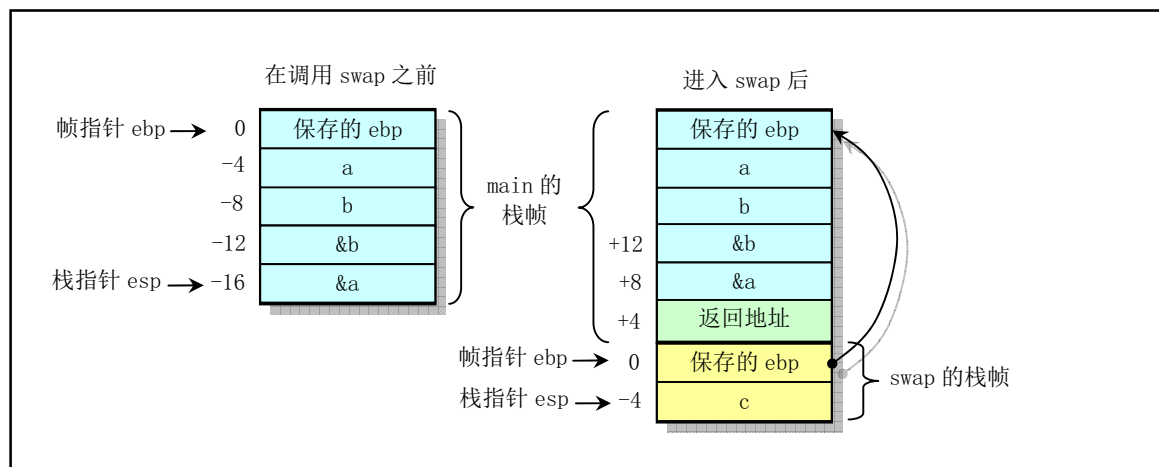


图 3-5 调用函数 main 和 swap 的栈帧结构

使用命令“gcc -Wall -S -oexch.s exch.c”可以生成该 C 语言程序的汇编程序 exch.s 代码，见如下所示（删除了几行与讨论无关的伪指令）。

```

1 .text
2 _swap:
3     pushl %ebp                # 保存原 ebp 值，设置当前函数的帧指针。
4     movl %esp, %ebp
5     subl $4, %esp            # 为局部变量 c 在栈内分配空间。
6     movl 8(%ebp), %eax        # 取函数第 1 个参数，该参数是一个整数类型值的指针。
7     movl (%eax), %ecx        # 取该指针所指位置的内容，并保存到局部变量 c 中。
8     movl %ecx, -4(%ebp)
9     movl 8(%ebp), %eax        # 再次取第 1 个参数，然后取第 2 个参数。
10    movl 12(%ebp), %edx
11    movl (%edx), %ecx        # 把第 2 个参数所指内容放到第 1 个参数所指的位置。
12    movl %ecx, (%eax)
13    movl 12(%ebp), %eax        # 再次取第 2 个参数。
14    movl -4(%ebp), %ecx        # 然后把局部变量 c 中的内容放到这个指针所指位置处。
15    movl %ecx, (%eax)
16    leave                    # 恢复原 ebp、esp 值（即 movl %ebp, %esp; popl %ebp;）。
17    ret
18 _main:
19    pushl %ebp                # 保存原 ebp 值，设置当前函数的帧指针。
20    movl %esp, %ebp
21    subl $8, %esp            # 为整型局部变量 a 和 b 在栈中分配空间。
22    movl $16, -4(%ebp)        # 为局部变量赋初值（a=16, b=32）。
23    movl $32, -8(%ebp)
24    leal -8(%ebp), %eax        # 为调用 swap() 函数作准备，取局部变量 b 的地址，
25    pushl %eax                # 作为调用的参数并压入栈中。即先压入第 2 个参数。
26    leal -4(%ebp), %eax        # 再取局部变量 a 的地址，作为第 1 个参数入栈。
27    pushl %eax
28    call _swap                # 调用函数 swap()。
29    movl -4(%ebp), %eax        # 取第 1 个局部变量 a 的值，减去第 2 个变量 b 的值。
30    subl -8(%ebp), %eax
31    leave                    # 恢复原 ebp、esp 值（即 movl %ebp, %esp; popl %ebp;）。
32    ret

```

这两个函数均可以划分成三个部分：“设置”，初始化栈帧结构；“主体”，执行函数的实际计算操作；“结束”，恢复栈状态并从函数中返回。对于 `swap()` 函数，其设置部分代码是 3--5 行。前两行用来设置保存调用者的帧指针和设置本函数的栈帧指针，第 5 行通过把栈指针 `esp` 下移 4 字节为局部变量 `c` 分配空间。行 6--15 是 `swap` 函数的主体部分。第 6--8 行用于取调用者的第 1 个参数 `&a`，并以该参数作为地址取所存内容到 `ecx` 寄存器中，然后保存到为局部变量分配的空间中 (`-4(%ebp)`)。第 9--12 行用于取第 2 个参数 `&b`，并以该参数值作为地址取其内容放到第 1 个参数指定的地址处。第 13--15 行把保存在临时局部变量 `c` 中的值存放到第 2 个参数指定的地址处。最后 16--17 行是函数结束部分。`leave` 指令用于处理栈内容以准备返回，它的作用等价于下面两个指令：

---

<code>movl %ebp, %esp</code>	# 恢复原 <code>esp</code> 的值（指向栈帧开始处）。
<code>popl %ebp</code>	# 恢复原 <code>ebp</code> 的值（通常是调用者的帧指针）。

---

这部分代码恢复了在进入 `swap()` 函数时寄存器 `esp` 和 `ebp` 的原有值，并执行返回指令 `ret`。

第 19--21 行是 `main()` 函数的设置部分，在保存和重新设置帧指针之后，`main()` 为局部变量 `a` 和 `b` 在栈中分配了空间。第 22--23 行为这两个局部变量赋值。从 24--28 行可以看出 `main()` 中是如何调用 `swap()` 函数的。其中首先使用 `leal` 指令（取有效地址）获得变量 `b` 和 `a` 的地址并分别压入栈中，然后调用 `swap()` 函数。变量地址压入栈中的顺序正好与函数申明的参数顺序相反。即函数最后一个参数首先压入栈中，而函数的第 1 个参数则是最后一个在调用函数指令 `call` 之前压入栈中的。第 29--30 两行将两个已经交换过的数字相减，并放在 `eax` 寄存器中作为返回值。

从以上分析可知，C 语言在调用函数时是在堆栈上临时存放被调函数参数的值，即 C 语言是传值类语言，没有直接的方法可用在被调函数中修改调用者变量的值。因此为了达到修改的目的就需要向函数传递变量的指针（即变量的地址）。

### 3.4.1.3 `main()` 也是一个函数

上面这段汇编程序是使用 `gcc 1.40` 编译产生的，可以看出其中有几行多余的代码。可见当时的 `gcc` 编译器还不能产生最高效率的代码，这也是为什么某些关键代码需要直接使用汇编语言编制的原因之一。另外，上面提到 C 程序的主程序 `main()` 也是一个函数。这是因为在编译链接时它将会作为 `crt0.s` 汇编程序的函数被调用。`crt0.s` 是一个桩（stub）程序，名称中的“`crt`”是“C run-time”的缩写。该程序的目标文件将被链接在每个用户执行程序的开始部分，主要用于设置一些初始化全局变量等。`Linux 0.11` 中 `crt0.s` 汇编程序见如下所示。其中建立并初始化全局变量 `_environ` 供程序中其它模块使用。

---

1 <code>.text</code>	
2 <code>.globl _environ</code>	# 声明全局变量 <code>_environ</code> （对应 C 程序中的 <code>environ</code> 变量）。
3	
4 <code>__entry:</code>	# 代码入口标号。
5 <code>    movl 8(%esp), %eax</code>	# 取程序的环境变量指针 <code>envp</code> 并保存在 <code>_environ</code> 中。
6 <code>    movl %eax, _environ</code>	# <code>envp</code> 是 <code>execve()</code> 函数在加载执行文件时设置的。
7 <code>    call _main</code>	# 调用我们的主程序。其返回状态值在 <code>eax</code> 寄存器中。
8 <code>    pushl %eax</code>	# 压入返回值作为 <code>exit()</code> 函数的参数并调用该函数。
9 1: <code>    call _exit</code>	
10 <code>    jmp 1b</code>	# 控制应该不会到达这里。若到达这里则继续执行 <code>exit()</code> 。
11 <code>.data</code>	
12 <code>_environ:</code>	# 定义变量 <code>_environ</code> ，为其分配一个长字空间。
13 <code>    .long 0</code>	

---

通常使用 `gcc` 编译链接生成执行文件时，`gcc` 会自动把该文件的代码作为第一个模块链接在可执行程序中。在编译时使用显示详细信息选项 `-v` 就可以明显地看出这个链接操作过程：

```
[/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]#
```

因此在通常的编译过程中我们无需特别指定 stub 模块 crt0.o，但是若想从上面给出的汇编程序手工使用 ld (gld) 从 exch.o 模块链接产生可执行文件 exch，那么我们就需要在命令行上特别指明 crt0.o 这个模块，并且链接的顺序应该是“crt0.o、所有程序模块、库文件”。

为了使用 ELF 格式的目标文件以及建立共享库模块文件，现在的 gcc 编译器 (2.x) 已经把这个 crt0 扩展成几个模块：crt1.o、crti.o、crtbegin.o、crtend.o 和 crtn.o。这些模块的链接顺序为“crt1.o、crti.o、crtbegin.o (crtbeginS.o)、所有程序模块、crtend.o (crtendS.o)、crtn.o、库模块文件”。gcc 的配置文件 specfile 指定了这种链接顺序。其中 crt1.o、crti.o 和 crtn.o 由 C 库提供，是 C 程序的“启动”模块；crtbegin.o 和 crtend.o 是 C++ 语言的启动模块，由编译器 gcc 提供；而 crt1.o 则与 crt0.o 的作用类似，主要用于在调用 main() 之前做一些初始化工作，全局符号 \_start 就定义在这个模块中。

crtbegin.o 和 crtend.o 主要用于 C++ 语言在 .ctors 和 .dtors 区中执行全局构造器 (constructor) 和析构器 (destructor) 函数。crtbeginS.o 和 crtendS.o 的作用与前两者类似，但用于创建共享模块中。crti.o 用于在 .init 区中执行初始化函数 init()。 .init 区中包含进程的初始化代码，即当程序开始执行时，系统会在调用 main() 之前先执行 .init 中的代码。crtn.o 则用于在 .fini 区中执行进程终止退出处理函数 fini() 函数，即当程序正常退出时 (main() 返回之后)，系统会安排执行 .fini 中的代码。

boot/head.s 程序中第 136--140 行就是用于为跳转到 init/main.c 中的 main() 函数作准备工作。第 139 行上的指令在栈中压入了返回地址，而第 140 行则压入了 main() 函数代码的地址。当 head.s 最后在第 218 行上执行 ret 指令时就会弹出 main() 的地址，并把控制权转移到 init/main.c 程序中。

### 3.4.2 在汇编程序中调用 C 函数

从汇编程序中调用 C 语言函数的方法实际上在上面已经给出。在上面 C 语言例子对应的汇编程序代码中，我们可以看出汇编程序语句是如何调用 swap() 函数的。现在我们对调用方法作一总结。

在汇编程序调用一个 C 函数时，程序需要首先按照逆向顺序把函数参数压入栈中，即函数最后 (最右边的) 一个参数先入栈，而最左边的第 1 个参数在最后调用指令之前入栈，见图 3-6 所示。然后执行 CALL 指令去执行被调用的函数。在调用函数返回后，程序需要再把先前压入栈中的函数参数清除掉。

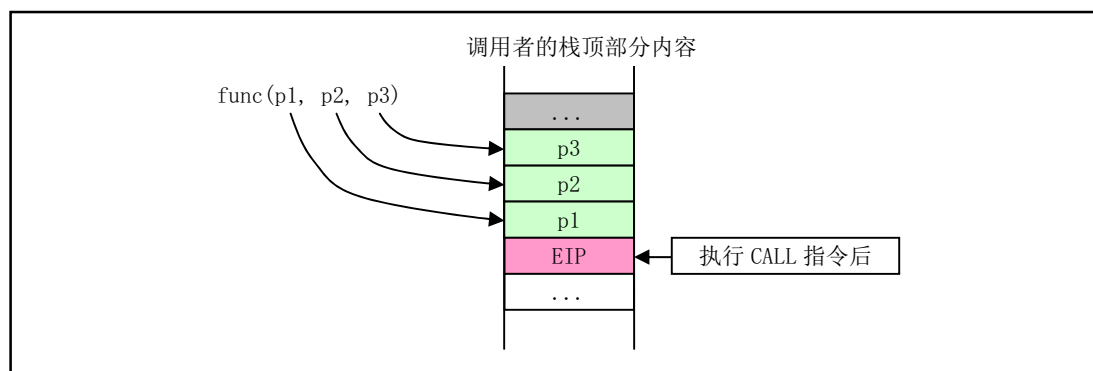


图 3-6 调用函数时压入堆栈的参数

在执行 CALL 指令时，CPU 会把 CALL 指令下一条指令的地址压入栈中（见图中 EIP）。如果调用还涉及到代码特权级变化，那么 CPU 还会进行堆栈切换，并且把当前堆栈指针、段描述符和调用参数压入新堆栈中。由于 Linux 内核中只使用中断门和陷阱门方式处理特权级变化时的调用情况，并没有使用 CALL 指令来处理特权级变化的情况，因此这里对特权级变化时的 CALL 指令使用方式不再进行说明。

汇编中调用 C 函数比较“自由”。只要是在栈中适当位置的内容就都可以作为参数供 C 函数使用。这里仍然以图 3-6 中具有 3 个参数的函数调用为例，如果我们没有专门为调用函数 func() 压入参数就直接调用它的话，那么 func() 函数仍然会把存放 EIP 位置以上的栈中其他内容作为自己的参数使用。如果我们为调用 func() 而仅仅明确地压入了第 1、第 2 个参数，那么 func() 函数的第 3 个参数 p3 就会直接使用 p2 前的栈中内容。在 Linux 0.1x 内核代码中就有几处使用了这种方式。例如在 kernel/system\_call.s 汇编程序中第 217 行上调用 copy\_process() 函数（kernel/fork.c 中第 68 行）的情况。在汇编程序函数\_sys\_fork 中虽然只把 5 个参数压入了栈中，但是 copy\_process() 却共带有多达 17 个参数，见下面所示：

---

```
// kernel/system_call.s 汇编程序_sys_fork 部分。
212     push %gs
213     pushl %esi
214     pushl %edi
215     pushl %ebp
216     pushl %eax
217     call _copy_process      # 调用 C 函数 copy_process() (kernel/fork.c, 68)。
218     addl $20,%esp          # 丢弃这里所有压栈内容。
219 1:   ret
```

---



---

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

---

我们知道参数越是最后入栈，它越是靠近 C 函数参数左侧。因此实际上调用 copy\_process() 函数之前入栈 5 个寄存器值就是 copy\_process() 函数的最左面的 5 个参数。按顺序它们分别对应为入栈的 eax (nr)、ebp、edi、esi 和寄存器 gs 的值。而随后的其余参数实际上直接对应堆栈上已有的内容。这些内容是进入系统调用中断处理过程开始，直到调用本系统调用处理过程时逐步入栈的各寄存器的值。

参数 none 是 system\_call.s 程序第 94 行上利用地址跳转表 sys\_call\_table[]（定义在 include/linux/sys.h，74 行）调用\_sys\_fork 时的下一条指令的返回地址值。随后的参数是刚进入 system\_call 时在 83--88 行压入栈的寄存器 ebx、ecx、edx 和段寄存器 fs、es、ds。最后 5 个参数是 CPU 执行中断指令压入返回地址 eip 和 cs、标志寄存器 eflags、用户栈地址 esp 和 ss。因为系统调用涉及到程序特权级变化，所以 CPU 会把标志寄存器值和用户栈地址也压入了堆栈。在调用 C 函数 copy\_process() 返回后，\_sys\_fork 也只把自己压入的 5 个参数丢弃掉，栈中其他还均保存着。其他采用上述用法的函数还有 kernel/signal.c 中的 do\_signal()、fs/exec.c 中的 do\_execve() 等，请自己加以分析。

另外，我们说汇编程序调用 C 函数比较自由的另一个原因是我们可以根本不用 CALL 指令而采用 JMP 指令来同样达到调用函数的目的。方法是在参数入栈后人工把下一条要执行的指令地址压入栈中，然后直接使用 JMP 指令跳转到被调用函数开始地址处去执行函数。此后当函数执行完成时就会执行 RET 指令把我们人工压入栈中的下一条指令地址弹出，作为函数返回的地址。Linux 内核中也有多处用到了这种函数调用方法，例如 kernel/asm.s 程序第 62 行调用执行 traps.c 中的 do\_int3() 函数的情况。

### 3.4.3 在 C 程序中调用汇编函数

从 C 程序中调用汇编程序函数的方法与汇编程序中调用 C 函数的原理相同，但 Linux 内核程序中不常使用。调用方法的着重点仍然是对函数参数在栈中位置的确定上。当然，如果调用的汇编语言程序比较短，那么可以直接在 C 程序中使用上面介绍的内联汇编语句来实现。下面我们以一个示例来说明编制这类程序的方法。包含两个函数的汇编程序 `callee.s` 见如下所示。

---

```

/*
  本汇编程序利用系统调用 sys_write() 实现显示函数 int mywrite(int fd, char * buf, int count)。
  函数 int myadd(int a, int b, int * res) 用于执行 a+b = res 运算。若函数返回 0，则说明溢出。
  注意：如果在现在的 Linux 系统（例如 RedHat 9）下编译，则请去掉函数名前的下划线 '_'。
*/
SYSWRITE = 4                                # sys_write() 系统调用号。
.global _mywrite, _myadd
.text
_mywrite:
    pushl    %ebp
    movl     %esp, %ebp
    pushl    %ebx
    movl     8(%ebp), %ebx                    # 取调用者第 1 个参数：文件描述符 fd。
    movl     12(%ebp), %ecx                   # 取第 2 个参数：缓冲区指针。
    movl     16(%ebp), %edx                   # 取第 3 个参数：显示字符数。
    movl     $SYSWRITE, %eax                  # %eax 中放入系统调用号 4。
    int      $0x80                            # 执行系统调用。
    popl     %ebx
    movl     %ebp, %esp
    popl     %ebp
    ret
_myadd:
    pushl    %ebp
    movl     %esp, %ebp
    movl     8(%ebp), %eax                    # 取第 1 个参数 a。
    movl     12(%ebp), %edx                   # 取第 2 个参数 b。
    xorl     %ecx, %ecx                       # %ecx 为 0 表示计算溢出。
    addl     %eax, %edx                       # 执行加法运算。
    jo       1f                               # 若溢出则跳转。
    movl     16(%ebp), %eax                   # 取第 3 个参数的指针。
    movl     %edx, (%eax)                     # 把计算结果放入指针所指位置处。
    incl     %ecx                             # 没有发生溢出，于是设置无溢出返回值。
1:    movl     %ecx, %eax                     # %eax 中是函数返回值。
    movl     %ebp, %esp
    popl     %ebp
    ret

```

---

该汇编文件中的第 1 个函数 `mywrite()` 利用系统中断 `0x80` 调用系统调用 `sys_write(int fd, char *buf, int count)` 实现在屏幕上显示信息。对应的系统调用功能号是 4（参见 `include/unistd.h`），三个参数分别为文件描述符、显示缓冲区指针和显示字符数。在执行 `int 0x80` 之前，寄存器 `%eax` 中需要放入调用功能号（4），寄存器 `%ebx`、`%ecx` 和 `%edx` 要按调用规定分别存放 `fd`、`buf` 和 `count`。函数 `mywrite()` 的调用参数个数和用途与 `sys_write()` 完全一样。

第 2 个函数 `myadd(int a, int b, int *res)` 执行加法运算。其中参数 `res` 是运算的结果。函数返回

值用于判断是否发生溢出。如果返回值为 0 表示计算已发生溢出，结果不可用。否则计算结果将通过参数 `res` 返回给调用者。

注意：如果在现在的 Linux 系统（例如 RedHat 9）下编译 `callee.s` 程序，则请去掉函数名前的下划线 `_`。调用这两个函数的 C 程序 `caller.c` 见如下所示。

---

```

/*
 调用汇编函数 mywrite(fd, buf, count) 显示信息；调用 myadd(a, b, result) 执行加运算。
 如果 myadd() 返回 0，则表示加函数发生溢出。首先显示开始计算信息，然后显示运算结果。
*/
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 }

```

---

该函数首先利用汇编函数 `mywrite()` 在屏幕上显示开始计算的信息 `"Calculating..."`，然后调用加法计算汇编函数 `myadd()` 对 `a` 和 `b` 两个数进行运算，并在第 3 个参数 `res` 中返回计算结果。最后再利用 `mywrite()` 函数把格式化过的结果信息字符串显示在屏幕上。如果函数 `myadd()` 返回 0，则表示加函数发生溢出，计算结果无效。这两个文件的编译和运行结果见如下所示：

---

```

[/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 0.11 目标文件格式

为了生成内核代码文件，Linux 0.11 使用了两种编译器。第一种是汇编编译器 `as86` 和相应的链接程序（或称为链接器）`ld86`。它们专门用于编译和链接运行在实地址模式下的 16 位内核引导扇区程序 `bootsect.s` 和设置程序 `setup.s`。第二种是 GNU 的汇编器 `as (gas)` 和 C 语言编译器 `gcc` 以及相应的链接程序 `gld`。编译器用于为源程序文件产生对应的二进制代码和数据目标文件。链接程序用于对相关的所有目标文件进行组合处理，形成一个可被内核加载执行的目标文件，即可执行文件。

本节首先简单说明编译器产生的目标文件结构，然后描述链接器如何把需要链接在一起的目标文件模块组合在一起，以生成二进制可执行映像文件或一个大的模块文件。最后说明 Linux 0.11 内核二进制代码

文件 Image 的生成原理和过程。这里给出了 Linux 0.11 内核所支持的 a.out 目标文件格式的信息。as86 和 ld86 生成的是 MINIX 专门的目标文件格式，我们将在涉及这种格式的内核创建工具一章中给出。因为 MINIX 目标文件结构与 a.out 目标文件格式类似，所以这里不对其进行说明。有关目标文件和链接程序的基本工作原理可参见 John R. Levine 著的《Linkers & Loaders》一书。

为便于描述，这里把编译器生成的目标文件称为目标模块文件（简称模块文件），而把链接程序输出产生的可执行目标文件称为可执行文件。并且把它们都统称为目标文件。

### 3.5.1 目标文件格式

在 Linux 0.11 系统中，GNU gcc 或 gas 编译输出的目标模块文件和链接程序所生成的可执行文件都使用了 UNIX 传统的 a.out 格式。这是一种被称为汇编与链接输出（Assembly & linker editor output）的目标文件格式。对于具有内存分页机制的系统来说，这是一种简单有效的目标文件格式。a.out 格式文件由一个文件头和随后的代码区（Text section，也称为正文段）、已初始化数据区（Data section，也称为数据段）、重定位信息区、符号表以及符号名字符串构成，见图 3-7 所示。其中代码区和数据区通常也被分别称为正文段（代码段）和数据段。

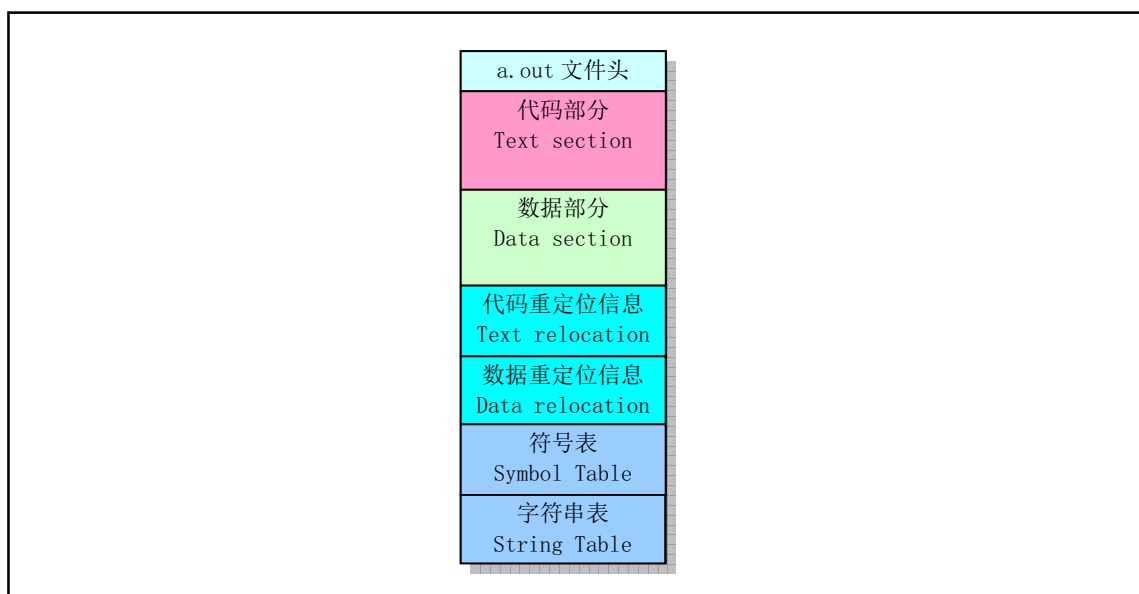


图 3-7 a.out 格式的目标文件

a.out 格式 7 个区的基本定义和用途是：

- **执行头部分（exec header）。**执行文件头部分。该部分中含有一些参数（exec 结构），是有关目标文件的整体结构信息。例如代码和数据区的长度、未初始化数据区的长度、对应源程序文件名以及目标文件创建时间等。内核使用这些参数把执行文件加载到内存中并执行，而链接程序（ld）使用这些参数将一些模块文件组合成一个可执行文件。这是目标文件唯一必要的组成部分。
- **代码区（text segment）。**由编译器或汇编器生成的二进制指令代码和数据信息，含有程序执行时被加载到内存中的指令代码和相关数据。可以以只读形式被加载。
- **数据区（data segment）。**由编译器或汇编器生成的二进制指令代码和数据信息，这部分含有已经初始化过的数据，总是被加载到可读写的内存中。
- **代码重定位部分（text relocations）。**这部分含有供链接程序使用的记录数据。在组合目标模块文件时用于定位代码段中的指针或地址。当链接程序需要改变目标代码的地址时就需要修正和维护这些地方。



- **数据重定位部分 (data relocations)**。类似于代码重定位部分的作用，但是用于数据段中指针的重定位。
- **符号表部分 (symbol table)**。这部分同样含有供链接程序使用的记录数据。这些记录数据保存着模块文件中定义的全局符号以及需要从其他模块文件中输入的符号，或者是由链接器定义的符号，用于在模块文件之间对命名的变量和函数（符号）进行交叉引用。
- **字符串表部分 (string table)**。该部分含有与符号名相对应的字符串。用于调试程序调试目标代码，与链接过程无关。这些信息可包含源程序代码和行号、局部符号以及数据结构描述信息等。

对于一个指定的目标文件并非一定会包含所有以上信息。由于 Linux 0.11 系统使用了 Intel CPU 的内存管理功能，因此它会为每个执行程序单独分配一个 64MB 的地址空间（逻辑地址空间）使用。在这种情况下因为链接器已经把执行文件处理成从一个固定地址开始运行，所以相关的可执行文件中就不再需要重定位信息。下面我们对其中几个重要区或部分进行说明。

### 3.5.1.1 执行头部分

目标文件的文件头中含有一个长度为 32 字节的 `exec` 数据结构，通常称为文件头结构或执行头结构。其定义如下所示。有关 `a.out` 结构的详细信息请参见 `include/a.out.h` 文件后的介绍。

---

```
struct exec {
    unsigned long a_magic      // 执行文件魔数。使用 N_MAGIC 等宏访问。
    unsigned a_text           // 代码长度，字节数。
    unsigned a_data           // 数据长度，字节数。
    unsigned a_bss            // 文件中的未初始化数据区长度，字节数。
    unsigned a_syms           // 文件中的符号表长度，字节数。
    unsigned a_entry          // 执行开始地址。
    unsigned a_trsize         // 代码重定位信息长度，字节数。
    unsigned a_drsize         // 数据重定位信息长度，字节数。
}
```

---

根据 `a.out` 文件中头结构魔数字段的值，我们又可把 `a.out` 格式的文件分成几种类型。Linux 0.11 系统使用了其中两种类型：模块目标文件使用了 OMAGIC (Old Magic) 类型的 `a.out` 格式，它指明文件是目标文件或者是不纯的可执行文件。其魔数是 `0x107` (八进制 `0407`)。而执行文件则使用了 ZMAGIC 类型的 `a.out` 格式，它指明文件为需求分页处理 (demand-paging，即需求加载 load on demand) 的可执行文件。其魔数是 `0x10b` (八进制 `0413`)。这两种格式的主要区别在于它们对各个部分的存储分配方式上。虽然该结构的总长度只有 32 字节，但是对于一个 ZMAGIC 类型的执行文件来说，其文件开始部分却需要专门留出 1024 字节的空间给头结构使用。除被头结构占用的 32 个字节以外，其余部分均为 0。从 1024 字节之后才开始放置程序的正文段和数据段等信息。而对于一个 OMAGIC 类型的 `.o` 模块文件来说，文件开始部分的 32 字节头结构后面紧接着就是代码区和数据区。

执行头结构中的 `a_text` 和 `a_data` 字段分别指明后面只读的代码段和可读写数据段的字节长度。`a_bss` 字段指明内核在加载目标文件时数据段后面未初始化数据区域 (bss 段) 的长度。由于 Linux 在分配内存时会自动对内存清零，因此 bss 段不需要被包括在模块文件或执行文件中。为了形象地表示目标文件逻辑地具有一个 bss 段，在后面图示中将使用虚线框来表示目标文件中的 bss 段。

`a_entry` 字段指定了程序代码开始执行的地址，而 `a_syms`、`a_trsize` 和 `a_drsize` 字段则分别说明了数据段后符号表、代码和数据段重定位信息的大小。对于可执行文件来说并不需要符号表和重定位信息，因此除非链接程序为了调试目的而包括了符号信息，执行文件中的这几个字段的值通常为 0。

### 3.5.1.2 重定位信息部分

Linux 0.11 系统的模块文件和执行文件都是 `a.out` 格式的目标文件，但是只有编译器生成的模块文件中包含用于链接程序的重定位信息。代码段和数据段的重定位信息均有重定位记录 (项) 构成，每个记录的长度为 8 字节，其结构如下所示。



---

```

struct relocation_info
{
    int r_address;           // 段内需要重定位的地址。
    unsigned int r_symbolnum:24; // 含义与 r_extern 有关。指定符号表中一个符号或者一个段。
    unsigned int r_pcrel:1;   // 1 比特。PC 相关标志。
    unsigned int r_length:2;  // 2 比特。指定要被重定位字段长度（2 的次方）。
    unsigned int r_extern:1;  // 外部标志位。1 - 以符号的值重定位。0 - 以段的地址重定位。
    unsigned int r_pad:4;     // 没有使用的 4 个比特位，但最好将它们复位掉。
};

```

---

重定位项的功能有两个。一是当代码段被重定位到一个不同的基地址处时，重定位项则用于指出需要修改的地方。二是在模块文件中存在对未定义符号引用时，当此未定义符号最终被定义时链接程序就可以使用相应重定位项对符号的值进行修正。由上面重定位记录项的结构可以看出，每个记录项含有模块文件代码区（代码段）和数据区（数据段）中需要重定位处长度为 4 字节的地址以及规定如何具体进行重定位操作的信息。地址字段 `r_address` 是指可重定位项从代码段或数据段开始算起的偏移值。2 比特的长度字段 `r_length` 指出被重定位项的长度，0 到 3 分别表示被重定位项的宽度是 1 字节、2 字节、4 字节或 8 字节。标志位 `r_pcrel` 指出被重定位项是一个“PC 相关的”的项，即它作为一个相对地址被用于指令当中。外部标志位 `r_extern` 控制着 `r_symbolnum` 的含义，指明重定位项参考的是段还是一个符号。如果该标志值是 0，那么该重定位项是一个普通的重定位项，此时 `r_symbolnum` 字段指定是在哪个段中寻址定位。如果该标志是 1，那么该重定位项是对一个外部符号的引用，此时 `r_symbolnum` 指定目标文件中符号表中的一个符号，需要使用符号的值进行重定位。

### 3.5.1.3 符号表和字符串部分

目标文件的最后部分是符号表和相关的字符串表。符号表记录项的结构如下所示。

---

```

struct nlist {
    union {
        char      *n_name;           // 字符串指针，
        struct nlist *n_next;        // 或者是指向另一个符号项结构的指针，
        long       n_strx;           // 或者是符号名称在字符串表中的字节偏移值。
    } n_un;
    unsigned char n_type;             // 该字节分成 3 个字段，参见 a.out.h 文件 146-154 行。
    char          n_other;           // 通常不用。
    short         n_desc;            //
    unsigned long n_value;           // 符号的值。
};

```

---

由于 GNU gcc 编译器允许任意长度的标识符，因此标识符字符串都位于符号表后的字符串表中。每个符号表记录项长度为 12 字节，其中第一个字段给出了符号名字符串（以 `null` 结尾）在字符串表中的偏移位置。类型字段 `n_type` 指明了符号的类型。该字段的最后一个比特位用于指明符号是否是外部的（全局的）。如果该位为 1 的话，那么说明该符号是一个全局符号。链接程序并不需要局部符号信息，但可供调试程序使用。`n_type` 字段的其余比特位用来指明符号类型。`a.out.h` 头文件中定义了这些类型值常量符号。符号的主要的类型包括：

**text**、**data** 或 **bbs** 指明是本模块文件中定义的符号。此时符号的值是模块中该符号的可重定位地址。

**abs** 指明符号是一个绝对的（固定的）不可重定位的符号。符号的值就是该固定值。

**undef** 指明是一个本模块文件中未定义的符号。此时符号的值通常是 0。

但作为一种特殊情况，编译器能够使用一个未定义的符号来要求链接程序为指定的符号名保留一块存储空间。如果一个未定义的外部（全局）符号具有非零值，那么对链接程序而言该值就是程序希望指定符

号寻址的存储空间的大小值。在链接操作期间，如果该符号确实没有定义，那么链接程序就会在 `bss` 段中为该符号名建立一块存储空间，空间的大小是所有被链接模块中该符号值最大的一个。这个就是 `bss` 段中所谓的公共块（Common block）定义，主要用于支持未初始化的外部（全局）数据。例如程序中定义的未初始化的数组。如果该符号在任意一个模块中已经被定义了，那么链接程序就会使用该定义而忽略该值。

### 3.5.2 Linux 0.11 中的目标文件格式

在 Linux 0.11 系统中，我们可以使用 `objdump` 命令来查看模块文件或执行文件中文件头结构的具体值。例如，下面列出了 `hello.o` 目标文件及其执行文件中文件头的具体值。

---

```
[/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
*
00004000  448b  0824  00a3  0030  e800  001a  0000  006a
00004100  db08  000d  eb00  00f9  6548  6c6c  2c6f  7720
00004200  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]#
```

---

可以看出，`hello.o` 模块文件的魔数是 `0407`（`OMAGIC`），代码段紧跟在头结构之后。除了文件头结构以外，还包括一个长度为 `0x28` 字节的代码段和一个具有 3 个符号项的符号表以及长度为 `0x10` 字节的代码段重定位信息。其余各段的长度均为 0。对应的执行文件 `hello` 的魔数是 `0413`（`ZMAGIC`），代码段从文件偏移位置 `1024` 字节开始存放。代码段和数据段的长度分别为 `0x3000` 和 `0x1000` 字节，并带有包含 141 个项的符

号表。我们可以使用命令 `strip` 删除执行文件中的符号表信息。例如下面我们删除了 `hello` 执行文件中的符号信息。可以看出 `hello` 执行文件的符号表长度变成了 0，并且 `hello` 文件的长度也从原来的 20591 字节减小到 17412 字节。

```
[/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]#
```

磁盘上 `a.out` 执行文件的各区在进程逻辑地址空间中的对应关系见图 3-8 所示。Linux 0.11 系统中进程的逻辑空间大小是 64MB。对于 ZMAGIC 类型的 `a.out` 执行文件，它的代码区的长度是内存页面的整数倍。由于 Linux 0.11 内核使用需求页（Demand-paging）技术，即在一页代码实际要使用的时候才被加载到物理内存页面中，而在进行加载操作的 `fs/execve()` 函数中仅仅为其设置了分页机制的页目录项和页表项，因此需求页技术可以加快程序的加载的速度。

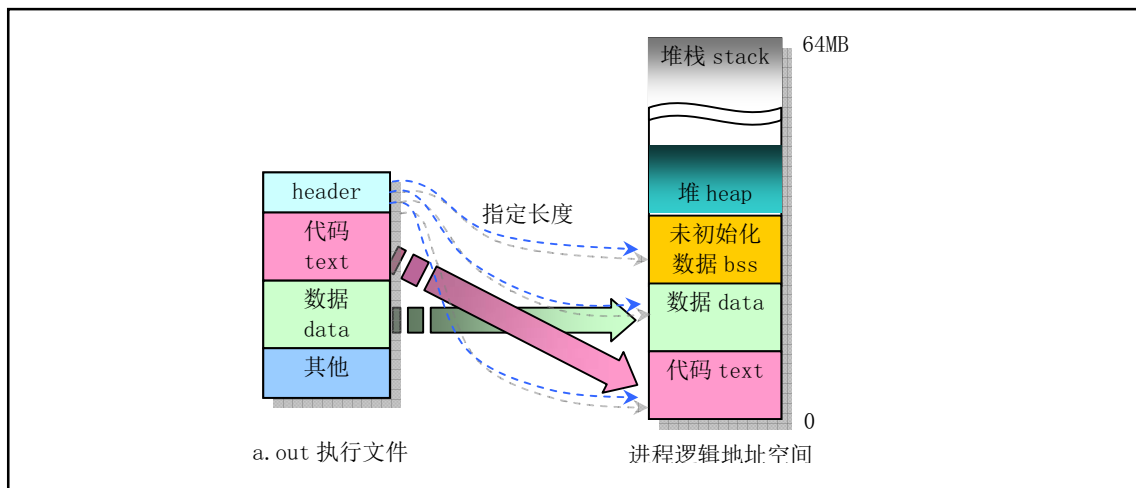


图 3-8 `a.out` 执行文件映射到进程逻辑地址空间

图中 `bss` 是进程的未初始化数据区，用于存放静态的未初始化数据。在开始执行程序时 `bss` 的第 1 页内存会被设置为全 0。图中 `heap` 是堆空间区，用于分配进程在执行过程中动态申请的内存空间。

### 3.5.3 链接程序输出

链接程序对输入的一个或多个模块文件以及相关的库函数模块进行处理，最终生成相应的二进制执行文件或者是一个所有模块组合而成的大模块文件。在这个过程中，链接程序的首要任务是给执行文件（或

者输出的模块文件)进行存储空间分配操作。一旦存储位置确定,链接程序就可以继续执行符号绑定操作和代码修正操作。因为模块文件中定义的大多数符号与文件中的存储位置有关,所以在符号对应的位置没有确定下来之前符号是没有办法解析的。

每个模块文件中包括几种类型的段,链接程序的第二个任务就是把所有模块中相同类型的段组合连接在一起,在输出文件中为指定段类型形成单一一个段。例如,链接程序需要把所有输入模块文件中的代码段合并成一个段放在输出的执行文件中。

对于 a.out 格式的模块文件来说,由于段类型是预先知道的,因此链接程序对 a.out 格式的模块文件进行存储分配比较容易。例如,对于具有两个输入模块文件和需要连接一个库函数模块的情况,其存储分配情况见图 3-9 所示。每个模块文件都有一个代码段(text)、数据段(data)和一个 bss 段,也许还会有一些看似外部(全局)符号的公共块。链接程序会收集每个模块文件包括任何库函数模块中的代码段、数据段和 bss 段的大小。在读入并处理了所有模块之后,任何具有非零值的未解析的外部符号都将作为公共块来看待,并且把它们分配存储在 bss 段的末尾处。

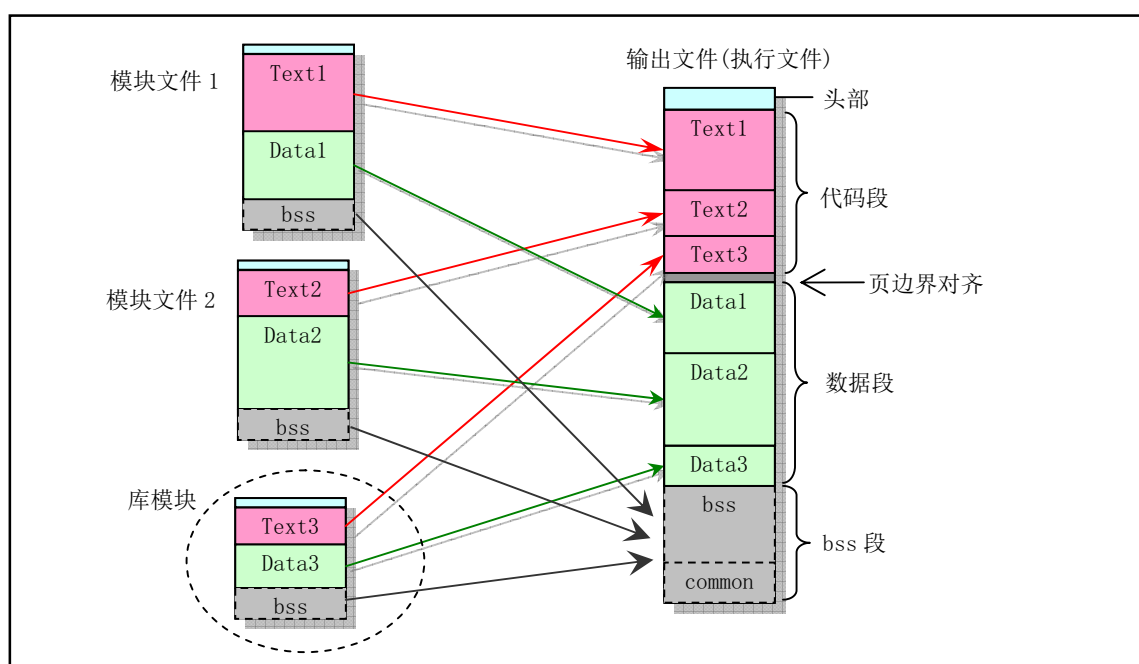


图 3-9 目标文件的链接操作

此后链接程序就可以为所有段分配地址。对于 Linux 0.11 系统中使用的 ZMAGIC 类型的 a.out 格式,输出文件中的代码段被设置成从固定地址 0 开始。数据段则从代码段后下一个页面边界开始。bss 段则紧随数据段开始放置。在每个段内,链接程序会把输入模块文件中的同类型段顺序存放,并按字进行边界对齐。

当 Linux 0.11 内核加载一个可执行文件时,它会根据文件头部结构中的信息首先判断文件是否是一个合适的可执行文件,即其魔数类型是否为 ZMAGIC,然后系统在用户态堆栈顶部为程序设置环境参数和命令行上输入的参数信息块并为其构建一个任务数据结构。接着在设置了一些相关寄存器值后利用堆栈返回技术去执行程序。执行程序映像文件中的代码和数据将会在实际执行到或用到时利用需求加载技术(Load on demand)动态加载到内存中。

对于 Linux 0.11 内核的编译过程,它是根据内核的配置文件 Makefile 使用 make 命令指挥编译器和链接程序操作而完成的。在建立过程中 make 还利用内核源代码 tools/目录下的 build.c 程序编译生成了一个用

于组合所有模块的临时工具程序 **build**。由于内核是由引导启动程序利用 ROM BIOS 中断调用加载到内存中，因此编译产生的内核各模块中的执行头结构部分需要去掉。工具程序 **build** 主要功能就是分别去掉 **bootsect**、**setup** 和 **system** 文件中的执行头结构，然后把它们顺序组合在一起产生一个名为 **Image** 的内核映像文件。

### 3.5.4 链接程序预定义变量

在链接过程中，链接器 **ld** 和 **ld86** 会使用变量记录下执行程序中每个段的逻辑地址。因此在程序中可以通过访问这几个外部变量来获得程序中段的位置。链接器预定义的外部变量通常至少有 **etext**、**\_etext**、**edata**、**\_edata**、**end** 和 **\_end**。

变量名 **\_etext** 和 **etext** 的地址是程序正文段结束后的第 1 个地址；**\_edata** 和 **edata** 的地址是初始化数据区后面的第 1 个地址；**\_end** 和 **end** 的地址是未初始化数据区（**bss**）后的第 1 个地址位置。带下划线‘**\_**’前缀的名称等同于不带下划线的对应名称，它们之间的唯一区别在于 ANSI、POSIX 等标准中没有定义符号 **etext**、**edata** 和 **end**。

当程序刚开始执行时，其 **brk** 所指位置与 **\_end** 处于相同位置。但是系统调用 **sys\_brk()**、内存分配函数 **malloc()** 以及标准输入/输出等操作会改变这个位置。因此程序当前的 **brk** 位置需要使用 **sbrk()** 来取得。注意，这些变量名必须看作是地址。因此在访问它们时需要使用取地址前缀‘**&**’，例如 **&end** 等。例如：

---

```
extern int _etext;
int et;

(int *) et = &_etext;           // 此时 et 含有正文段结束处后面的地址。
```

---

下面程序 **predef.c** 可用于显示出这几个变量的地址。可以看出带与不带下划线‘**\_**’符号的地址值是相同的。

---

```
/*
   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;
}
```

---

在 Linux 0.1X 系统下运行该程序可以得到以下结果。请注意，这些地址都是程序地址空间中的逻辑地址，即从执行程序被加载到内存位置开始算起的地址。

---

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

---

如果在现在的 Linux 系统（例如 RedHat 9）中运行这个程序，就可得到以下结果。我们知道现在 Linux 系统中程序代码从其逻辑地址 0x08048000 处开始存放，因此可知这个程序的代码段长度是 0x41b 字节。

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

Linux 0.1x 内核在初始化块设备高速缓冲区时（fs/buffer.c），就使用了变量名 `_end` 来获取内核映像文件 Image 在内存中的末端后的位置，并从这个位置起开始设置高速缓冲区。

### 3.5.5 System.map 文件

当运行 GNU 链接器 `gld (ld)` 时若使用了 `-M` 选项，或者使用 `nm` 命令，则会在标准输出设备（通常是屏幕）上打印出链接映像（link map）信息，即是指由连接程序产生的目标程序内存地址映像信息。其中列出了程序段装入到内存中的位置信息。具体来讲有如下信息：

- 目标文件及符号信息映射到内存中的位置；
- 公共符号如何放置；
- 链接中包含的所有文件成员及其引用的符号。

通常我们会把发送到标准输出设备的链接映像信息重定向到一个文件中（例如 `System.map`）。在编译内核时，`linux/Makefile` 文件产生的 `System.map` 文件就用于存放内核符号表信息。符号表是所有内核符号及其对应地址的一个列表，当然也包括上面说明的 `_etext`、`_edata` 和 `_end` 等符号的地址信息。随着每次内核的编译，就会产生一个新的对应 `System.map` 文件。当内核运行出错时，通过 `System.map` 文件中的符号表解析，就可以查到一个地址值对应的变量名，或反之。

利用 `System.map` 符号表文件，在内核或相关程序出错时，就可以获得我们比较容易识别的信息。符号表的样例如下所示：

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

其中每行说明一个符号，第 1 栏指明符号值（地址）；第 2 栏是符号类型，指明符号位于目标文件的哪个区（sections）或其属性；第 3 栏是对应的符号名称。

第 2 栏中的符号类型指示符通常有表 3-5 所示的几种，另外还有一些与采用的目标文件格式相关。如果符号类型是小写字母，则说明符号是局部的；如果是大写字母，则说明符号是全局的（外部的）。参见文件 `include/a.out.h` 中 `nlist {}` 结构 `n_type` 字段的定义（第 110--185 行）。

表 3-5 目标文件符号列表文件中的符号类型

符号类型	名称	说明
A	Absolute	符号的值是绝对值，并且在进一步链接过程中不会被改变。
B	BSS	符号在未初始化数据区或区（section）中，即在 BSS 段中。
C	Common	符号是公共的。公共符号是未初始化的数据。在链接时，多个公共符号可能具有同一名称。如果该符号定义在其他地方，则公共符号被看作是未定义的引用。

D	Data	符号在已初始化数据区中。
G	Global	符号是在小对象已初始化数据区中的符号。某些目标文件的格式允许对小数据对象（例如一个全局整型变量）可进行更有效的访问。
I	Indirect	符号是对另一个符号的间接引用。
N	Debugging	符号是一个调试符号。
R	Read only	符号在一个只读数据区中。
S	Small	符号是小对象未初始化数据区中的符号。
T	Text	符号是代码区中的符号。
U	Undefined	符号是外部的，并且其值为 0（未定义）。
-	Stabs	符号是 a.out 目标文件中的一个 stab 符号，用于保存调试信息。
?	Unknwon	符号的类型未知，或者是与具体文件格式有关。

可以看出名称为 dmi\_broken 的变量位于内核地址 0xc03441a0 处。

System.map 位于使用它的软件（例如内核日志记录后台程序 klogd）能够寻找到的地方。在系统启动时，如果没有以一个参数的形式为 klogd 给出 System.map 的位置，则 klogd 将会三个地方搜寻 System.map。依次为：

---

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

---

尽管内核本身实际上不使用 System.map，但其他程序，象 klogd、lsof、ps 以及其他象 dosemu 等许多软件都需要有一个正确的 System.map 文件。利用该文件，这些程序就可以根据已知的内存地址查找出对应的内核变量名称，便于对内核的调试工作。

## 3.6 Make 程序和 Makefile 文件

Makefile（或 makefile）文件是 make 工具程序的配置文件。Make 工具程序的主要用途是能自动地决定一个含有很多源程序文件的大型程序中哪个文件需要被重新编译。Makefile 的使用比较复杂，这里只是根据上面的 Makefile 文件作些简单的介绍。详细说明请参考 GNU make 使用手册。

为了使用 make 程序，你就需要 Makefile 文件来告诉 make 要做什么工作。通常，Makefile 文件会告诉 make 如何编译和连接一个文件。当明确指出时，Makefile 还可以告诉 make 运行各种命令（例如，作为清理操作而删除某些文件）。

make 的执行过程分为两个不同的阶段。在第一个阶段，它读取所有的 Makefile 文件以及包含的 Makefile 文件等，记录所有的变量及其值、隐式的或显式的规则，并构造出所有目标对象及其先决条件的一幅全景图。在第二阶段期间，make 就使用这些内部结构来确定哪个目标对象需要被重建，并且使用相应的规则来操作。

当 make 重新编译程序时，每个修改过的 C 代码文件必须被重新编译。如果一个头文件被修改过了，那么为了确保正确，每一个包含该头文件的 C 代码程序都将被重新编译。每次编译操作都产生一个与源程序对应的目标文件。最终，如果任何源代码文件被编译过了，那么所有的目标文件不管是刚编译完的还是以前就编译好的必须连接在一起以生成新的可执行文件。

简单的 Makefile 文件含有一些规则，这些规则具有如下的形式：

---

```
目标(target)... : 先决条件(prerequisites)...
                  命令(command)
```

---

---

...

---

其中‘目标’对象通常是程序生成的一个文件的名称；例如是一个可执行文件或目标文件。目标也可以是所要采取活动的名字，比如‘清除’（‘clean’）。‘先决条件’是一个或多个文件名，是用作产生目标的输入条件。通常一个目标依赖几个文件。而‘命令’是 **make** 需要执行的操作。一个规则可以有多个命令，每一个命令自成一。行。请注意，你需要在每个命令之前键入一个制表符！这是粗心者常常忽略的地方。

如果一个先决条件通过目录搜寻而在另外一个目录中被找到，这并不会改变规则的命令；它们将被如期执行。因此，你必须小心地设置命令，使得命令能够在 **make** 发现先决条件的目录中找到需要的先决条件。这就需要通过使用自动变量来做到。自动变量是一种在命令行上根据具体情况能被自动替换的变量。自动变量的值是基于目标对象及其先决条件而在命令执行前设置的。例如，‘\$’的值表示规则的所有先决条件，包括它们所处目录的名称；‘\$<’的值表示规则中的第一个先决条件；‘\$@’表示目标对象；另外还有一些自动变量这里就不提了。

有时，先决条件还常包含头文件，而这些头文件并不愿在命令中说明。此时自动变量‘\$<’正是第一个先决条件。例如：

---

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

---

其中的‘\$<’就会被自动地替换成 `foo.c`，而‘\$@’则会被替换为 `foo.o`

为了让 **make** 能使用习惯用法来更新一个目标对象，你可以不指定命令，写一个不带命令的规则或者不写规则。此时 **make** 程序将会根据源程序文件的类型（程序的后缀）来判断要使用哪个隐式规则。

后缀规则是为 **make** 程序定义隐式规则的老式方法（现在这种规则已经不用了，取而代之的是使用更通用更清晰的模式匹配规则）。下面例子就是一种双后缀规则。双后缀规则是用一对后缀定义的：源后缀和目标后缀。相应的隐式先决条件是通过使用文件名中的源后缀替换目标后缀后得到。因此，此时下面的‘\$<’值是‘\*.c’文件名。而正条 **make** 规则的含义是将‘\*.c’程序编译成‘\*.s’代码。

---

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

---

通常命令是属于一个具有先决条件的规则，并在任何先决条件改变时用于生成一个目标（**target**）文件。然而，为目标而指定命令的规则也并不一定要有先决条件。例如，与目标‘clean’相关的含有删除（**delete**）命令的规则并不需要有先决条件。此时，一个规则说明了如何以及何时来重新制作某些文件，而这些文件是特定规则的目标。**make** 根据先决条件来执行命令以创建或更新目标。一个规则也可以说明如何及何时执行一个操作。

一个 **Makefile** 文件也可以含有除规则以外的其他文字，但一个简单的 **Makefile** 文件只需要含有适当的规则。规则可能看上去要比上面示出的模板复杂得多，但基本上都是符合的。

**Makefile** 文件最后生成的依赖关系是用于让 **make** 来确定是否需要重建一个目标对象。比如当某个头文件被改动过后，**make** 就通过这些依赖关系，重新编译与该头文件有关的所有‘\*.c’文件。

## 3.7 本章小结

本章以几个可运行的汇编语言程序作为描述对象，详细说明了 **as86** 和 **GNU as** 汇编语言的基本语言和使用方法。同时对 **Linux** 内核使用的 **C** 语言扩展语句进行了详细介绍。对于学习操作系统来说，系统支持的目标文件结构有着非常重要的作用，因此本章对 **Linux 0.11** 中使用的 **a.out** 目标文件格式作了详细介绍。



下一章我们将围绕 Intel 80X86 处理器，详细地说明其运行在保护模式下的工作原理。并给出一个保护模式多任务程序示例，通过阅读这个示例，我们可以对操作系统最初如何“旋转”起来有一个基本了解，并为继续阅读完整的 Linux 0.11 内核源代码打下坚实基础。

## 第4章 80X86 保护模式及其编程

本书介绍的 Linux 操作系统基于 Intel 公司 80X86 及相关外围硬件组成的 PC 机系统。有关 80X86 CPU 系统编程的最佳参考书籍当然是 Intel 公司发行的一套三卷的《IA-32 Intel 体系结构软件开发手册》，尤其是其中第 3 卷：《系统编程指南》是理解使用 80X86 CPU 的操作系统工作原理或进行系统编程必不可少的参考资料，本章内容主要取自于该书。这些资料可以从 Intel 公司的网站上免费下载。本章主要概要描述 80X86 CPU 的体系结构以及保护模式下编程的一些基础知识，为准备阅读基于 80X86 CPU 的 Linux 内核源代码打下坚实基础。主要包括：1. 80X86 基础知识；2. 保护模式内存管理；3. 各种保护措施；4. 中断和异常处理；5. 任务管理；6. 保护模式编程的初始化；7. 一个简单的多任务内核例子。

本章最后部分介绍的一个简单多任务内核是基于 Linux 0.11 内核的一个简化实例。该实例用于演示内存分段管理和任务管理的实现方法，没有包括分页机制内容。但若彻底理解这个实例的运作机制，那么在随后阅读 Linux 内核源代码时就不应该再会碰到什么大问题了。

若读者对这部分内容已经比较熟悉，那么可以直接阅读本章最后给出的一个可运行的内核实例。当然，在阅读内核源代码时读者可以随时回过头来参考本章内容。因此并不勉强读者需要完全理解本章内容之后才开始阅读后续章节中的 Linux 内核代码。

### 4.1 80X86 系统寄存器和系统指令

为了协助处理器执行初始化和控制系统操作，80X86 提供了一个标志寄存器 EFLAGS 和几个系统寄存器，除了一些通用状态标志外，EFLAGS 中还包含几个系统标志。这些系统标志用于控制任务切换、中断处理、指令跟踪以及访问权限。系统寄存器用于内存管理和控制处理器操作，含有分段和分页处理机制系统表的基地址、控制处理器操作的比特标志位。

#### 4.1.1 标志寄存器

标志寄存器 EFLAGS 中的系统标志和 IOPL 字段用于控制 I/O 访问、可屏蔽硬件中断、调试、任务切换以及虚拟-8086 模式，见图 4-1 所示。通常只允许操作系统代码有权修改这些标志。EFLAGS 中的其他标志是一些通用标志（进位 CF、奇偶 PF、辅助进位 AF、零标志 ZF、负号 SF、方向 DF、溢出 OF）。这里我们仅对 EFLAGS 中的系统标志进行说明。

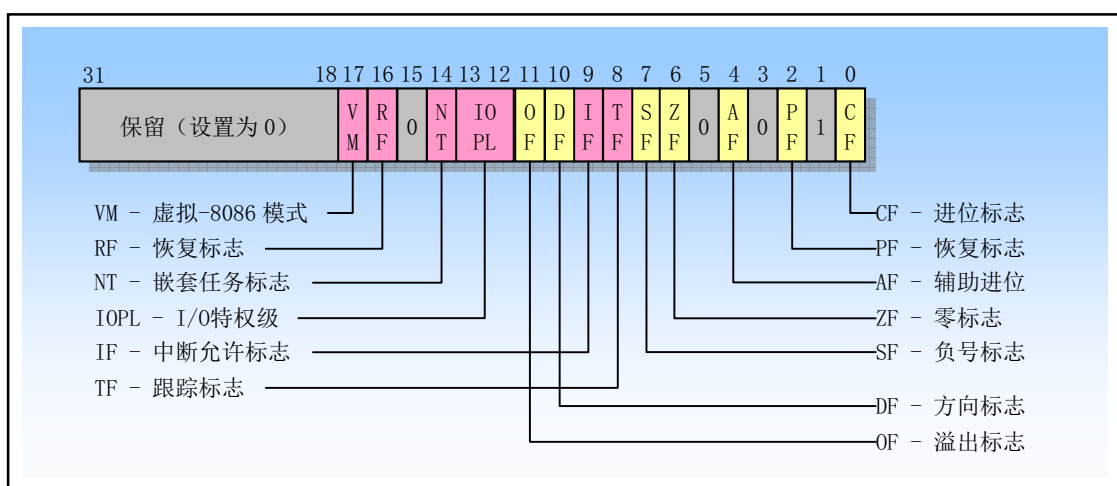


图 4-1 标志寄存器 EFLAGS 中的系统标志

- TF** 位 8 是跟踪标志 (Trap Flag)。当设置该位时可为调试操作启动单步执行方式；复位时则禁止单步执行。在单步执行方式下，处理器会在每个指令执行之后产生一个调试异常，这样我们就可以观察执行程序在执行每条指令后的状态。如果程序使用 POPF、POPFD 或 IRET 指令设置了 TF 标志，那么在随后指令之后处理器就会产生一个调试异常。
- IOPL** 位 13-12 是 I/O 特权级 (I/O Privilege Level) 字段。该字段指明当前运行程序或任务的 I/O 特权级 IOPL。当前运行程序或任务的 CPL 必须小于等于这个 IOPL 才能访问 I/O 地址空间。只有当 CPL 为特权级 0 时，程序才可以使用 POPF 或 IRET 指令修改这个字段。IOPL 也是控制对 IF 标志修改的机制之一。
- NT** 位 14 是嵌套任务标志 (Nested Task)。它控制着被中断任务和调用任务之间的链接关系。在使用 CALL 指令、中断或异常执行任务调用时，处理器会设置该标志。在通过使用 IRET 指令从一个任务返回时，处理器会检查并修改这个 NT 标志。使用 POPF/POPFD 指令也可以修该这个标志，但是在应用程序中改变这个标志的状态会产生不可意料的异常。
- RF** 位 16 是恢复标志 (Resume Flag)。该标志用于控制处理器对断点指令的响应。当设置时，这个标志会临时禁止断点指令产生的调试异常；当该标志复位时，则断点指令将会产生异常。RF 标志的主要功能是允许在调试异常之后重新执行一条指令。当调试软件使用 IRETD 指令返回被中断程序之前，需要设置堆栈上 EFLAGS 内容中的 RF 标志，以防止指令断点造成另一个异常。处理器会在指令返回之后自动地清除该标志，从而再次允许指令断点异常。
- VM** 位 17 是虚拟-8086 方式 (Virtual-8086 Mode) 标志。当设置该标志时，就开启虚拟-8086 方式；当复位该标志时，则回到保护模式。

### 4.1.2 内存管理寄存器

处理器提供了 4 个内存管理寄存器 (GDTR、LDTR、IDTR 和 TR)，用于指定分段内存管理所使用的系统表的基地址，见图 4-2 所示。处理器为这些寄存器的加载和保存提供了特定的指令。有关系统表的作用请参见下一节“保护模式内存管理”中的详细说明。

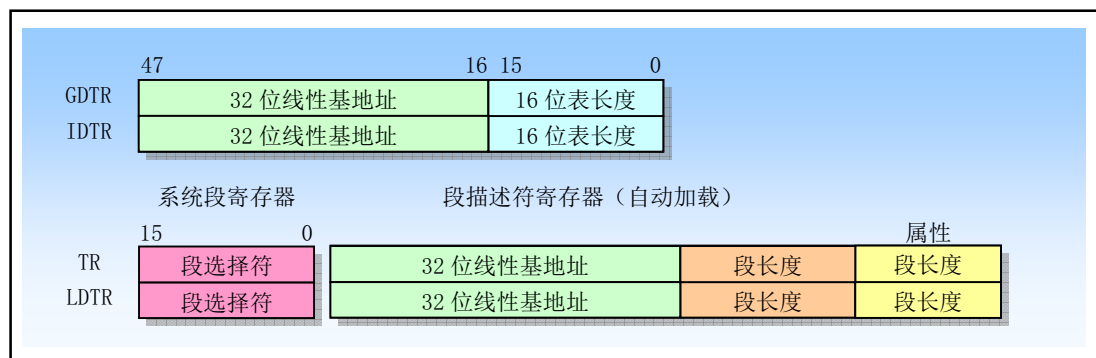


图 4-2 内存管理寄存器

GDTR、LDTR、IDTR 和 TR 都是段基址寄存器，这些段中含有分段机制的重要信息表。GDTR、IDTR 和 LDTR 用于寻址存放描述符表的段。TR 用于寻址一个特殊的任务状态段 TSS (Task State Segment)。TSS 段中包含着当前执行任务的重要信息。

#### 1. 全局描述符表寄存器 GDTR

GDTR 寄存器中用于存放全局描述符表 GDT 的 32 位线性基地址和 16 位表长度值。基地址指定 GDT 表中字节 0 在线性地址空间中的地址，表长度指明 GDT 表的字节长度值。指令 LGDT 和 SGDT 分别用于加载和保存 GDTR 寄存器的内容。在机器刚加电或处理器复位后，基地址被默认地设置为 0，而长度值被设置成 0xFFFF。在保护模式初始化过程中必须给 GDTR 加载一个新值。

#### 2. 中断描述符表寄存器 IDTR

与 GDTR 的作用类似，IDTR 寄存器用于存放中断描述符表 IDT 的 32 位线性基地址和 16 位表长度值。指令 LIDT 和 SIDT 分别用于加载和保存 IDTR 寄存器的内容。在机器刚加电或处理器复位后，基地址被默认地设置为 0，而长度值被设置成 0xFFFF。

#### 3. 局部描述符表寄存器 LDTR

LDTR 寄存器中用于存放局部描述符表 LDT 的 32 位线性基地址、16 位段限长和描述符属性值。指令 LLDT 和 SLDT 分别用于加载和保存 LDTR 寄存器的段描述符部分。包含 LDT 表的段必须在 GDT 表中有一个段描述符项。当使用 LLDT 指令把含有 LDT 表段的的选择符加载进 LDTR 时，LDT 段描述符的段基地址、段限长度以及描述符属性会被自动地加载到 LDTR 中。当进行任务切换时，处理器会把新任务 LDT 的段选择符和段描述符自动地加载进 LDTR 中。在机器加电或处理器复位后，段选择符和基地址被默认地设置为 0，而段长度被设置成 0xFFFF。

#### 4. 任务寄存器 TR

TR 寄存器用于存放当前任务 TSS 段的 16 位段选择符、32 位基地址、16 位段长度和描述符属性值。它引用 GDT 表中的一个 TSS 类型的描述符。指令 LTR 和 STR 分别用于加载和保存 TR 寄存器的段选择符部分。当使用 LTR 指令把选择符加载进任务寄存器时，TSS 描述符中的段基地址、段限长度以及描述符属性会被自动地加载到任务寄存器中。当执行任务切换时，处理器会把新任务 TSS 的段选择符和段描述符自动地加载进任务寄存器 TR 中。

### 4.1.3 控制寄存器

控制寄存器 (CR0、CR1、CR2 和 CR3) 用于控制和确定处理器的操作模式以及当前执行任务的特性，见图 4-3 所示。CR0 中含有控制处理器操作模式和状态的系统控制标志；CR1 保留不用；CR2 含有导致页错误的线性地址。CR3 中含有页目录表物理内存基地址，因此该寄存器也被称为页目录基地址寄存器 PDBR (Page-Directory Base address Register)。

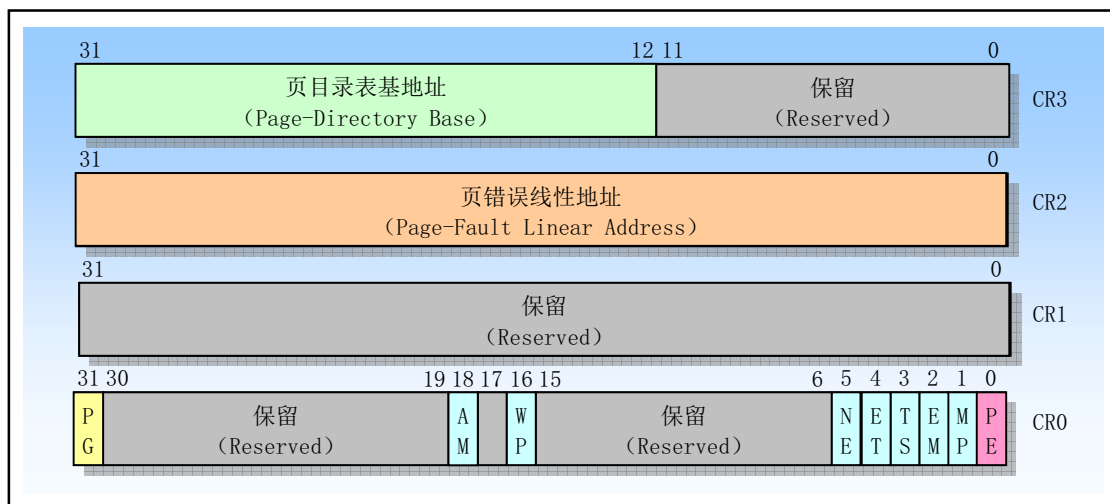


图 4-3 控制寄存器 CR0--CR3

### 1. CR0 中协处理器控制位

CR0 的 4 个比特位：扩展类型位 ET、任务切换位 TS、仿真位 EM 和数学存在位 MP 用于控制 80X86 浮点（数学）协处理器的操作。有关协处理器的详细说明请参见第 11 章内容。CR0 的 ET 位（标志）用于选择与协处理器进行通信所使用的协议，即指明系统中使用的是 80387 还是 80287 协处理器。TS、MP 和 EM 位用于确定浮点指令或 WAIT 指令是否应该产生一个设备不存在 DNA（Device Not Available）异常。这个异常可用来仅为使用浮点运算的任务保存和恢复浮点寄存器。对于没有使用浮点运算的任务，这样做可以加快它们之间的切换操作。

**ET** CR0 的位 4 是扩展类型（Extension Type）标志。当该标志为 1 时，表示指明系统有 80387 协处理器存在，并使用 32 位协处理器协议。ET=0 指明使用 80287 协处理器。如果仿真位 EM=1，则该位将被忽略。在处理器复位操作时，ET 位会被初始化指明系统中使用的协处理器类型。如果系统中有 80387，则 ET 被设置成 1，否则若有一个 80287 或者没有协处理器，则 ET 被设置成 0。

**TS** CR0 的位 3 是任务已切换（Task Switched）标志。该标志用于推迟保存任务切换时的协处理器内容，直到新任务开始实际执行协处理器指令。处理器在每次任务切换时都会设置该标志，并且在执行协处理器指令时测试该标志。

如果设置了 TS 标志并且 CR0 的 EM 标志为 0，那么在执行任何协处理器指令之前会产生一个设备不存在 DNA（Device Not Available）异常。如果设置了 TS 标志但没有设置 CR0 的 MP 和 EM 标志，那么在执行协处理器指令 WAIT/FWAIT 之前不会产生设备不存在异常。如果设置了 EM 标志，那么 TS 标志对协处理器指令的执行无影响。见表 4-1 所示。

在任务切换时，处理器并不自动保存协处理器的上下文，而是会设置 TS 标志。这个标志会使得处理器在执行新任务指令流的任何时候遇到一条协处理器指令时产生设备不存在异常。设备不存在异常的处理程序可使用 CLTS 指令清除 TS 标志，并且保存协处理器的上下文。如果任务从没有使用过协处理器，那么相应协处理器上下文就不用保存。

**EM** CR0 的位 2 是仿真（EMulation）标志。当该位设置时，表示处理器没有内部或外部协处理器，执行协处理器指令时会引起设备不存在异常；当清除时，表示系统有协处理器。设置这个标志可以迫使所有浮点指令使用软件来模拟。

**MP** CR0 的位 1 是监控协处理器（Monitor Coprocessor 或 Math Present）标志。用于控制 WAIT/FWAIT 指令与 TS 标志的交互作用。如果 MP=1、TS=1，那么执行 WAIT 指令将产生一个设备不存在异常；如果 MP=0，则 TS 标志不会影响 WAIT 的执行。

表 4-1 CR0 中标志 EM、MP 和 TS 的不同组合对协处理器指令动作的影响

CR0 中的标志			指令类型	
EM	MP	TS	浮点	WAIT/FWAIT
0	0	0	执行	执行
0	0	1	设备不存在 (DNA) 异常	执行
0	1	0	执行	执行
0	1	1	DNA 异常	DNA 异常
1	0	0	DNA 异常	执行
1	0	1	DNA 异常	执行
1	1	0	DNA 异常	执行
1	1	1	DNA 异常	DNA 异常

## 2. CR0 中保护控制位

**PE** CR0 的位 0 是启用保护 (Protection Enable) 标志。当设置该位时即开启了保护模式；当复位时即进入实地址模式。这个标志仅开启段级保护，而并没有启用分页机制。若要启用分页机制，那么 PE 和 PG 标志都要置位。

**PG** CR0 的位 31 是分页 (Paging) 标志。当设置该位时即开启了分页机制；当复位时则禁止分页机制，此时所有线性地址等同于物理地址。在开启这个标志之前必须已经或者同时开启 PE 标志。即若要启用分页机制，那么 PE 和 PG 标志都要置位。

**WP** 对于 Intel 80486 或以上的 CPU，CR0 的位 16 是写保护 (Write Protect) 标志。当设置该标志时，处理器会禁止超级用户程序 (例如特权级 0 的程序) 向用户级只读页面执行写操作；当该位复位时则反之。该标志有利于 UNIX 类操作系统在创建进程时实现写时复制 (Copy on Write) 技术。

**NE** 对于 Intel 80486 或以上的 CPU，CR0 的位 5 是协处理器错误 (Numeric Error) 标志。当设置该标志时，就启用了 X87 协处理器错误的内部报告机制；若复位该标志，那么就使用 PC 机形式的 X87 协处理器错误报告机制。当 NE 为复位状态并且 CPU 的 IGNNE 输入引脚有信号时，那么数学协处理器 X87 错误将被忽略。当 NE 为复位状态并且 CPU 的 IGNNE 输入引脚无信号时，那么非屏蔽的数学协处理器 X87 错误将导致处理器通过 FERR 引脚在外部产生一个中断，并且在执行下一个等待形式浮点指令或 WAIT/FWAIT 指令之前立刻停止指令执行。CPU 的 FERR 引脚用于仿真外部协处理器 80387 的 ERROR 引脚，因此通常连接到中断控制器输入请求引脚上。NE 标志、IGNNE 引脚和 FERR 引脚用于利用外部逻辑来实现 PC 机形式的外部错误报告机制。

启用保护模式 PE (Protected Enable) 位 (位 0) 和开启分页 PG (Paging) 位 (位 31) 分别用于控制分段和分页机制。PE 用于控制分段机制。如果 PE=1，处理器就工作在开启分段机制环境下，即运行在保护模式下。如果 PE=0，则处理器关闭了分段机制，并如同 8086 工作于实地址模式下。PG 用于控制分页机制。如果 PG=1，则开启了分页机制。如果 PG=0，分页机制被禁止，此时线性地址被直接作为物理地址使用。

如果 PE=0、PG=0，处理器工作在实地址模式下；如果 PG=0、PE=1，处理器工作在没有开启分页机制的保护模式下；如果 PG=1、PE=0，此时由于不在保护模式下不能启用分页机制，因此处理器会产生一个一般保护异常，即这种标志组合无效；如果 PG=1、PE=1，则处理器工作在开启了分页机制的保护模式下。

当改变 PE 和 PG 位时，我们必须小心。只有当执行程序起码有部分代码和数据在线性地址空间和物理地址空间中具有相同地址时，我们才能改变 PG 位的设置。此时这部分具有相同地址的代码在分页和未分页世界之间起着桥梁的作用。无论是否开启分页机制，这部分代码都具有相同的地址。另外，在开启分页 (PG=1) 之前必须先刷新 CPU 中的页高速缓冲 (或称为转换查找缓冲区 TLB - Translation Lookaside

Buffers)。

在修改该了 PE 位之后程序必须立刻使用一条跳转指令，以刷新处理器执行管道中已经获取的不同模式下的任何指令。在设置 PE 位之前，程序必须初始化几个系统段和控制寄存器。在系统刚上电时，处理器被复位成 PE=0、PG=0（即实模式状态），以允许引导代码在启用分段和分页机制之前能够初始化这些寄存器和数据结构。

### 3. CR2 和 CR3

CR2 和 CR3 用于分页机制。CR3 含有存放页目录表页面的物理地址，因此 CR3 也被称为 PDBR。因为页目录表页面是页对齐的，所以该寄存器只有高 20 位是有效的。而低 12 位保留供更高级处理器使用，因此在往 CR3 中加载一个新值时低 12 位必须设置为 0。

使用 MOV 指令加载 CR3 时具有让页高速缓冲无效的副作用。为了减少地址转换所要求的总线周期数量，最近访问的页目录和页表会被存放在处理器的页高速缓冲器件中，该缓冲器件被称为转换查找缓冲区 TLB（Translation Lookaside Buffer）。只有当 TLB 中不包含要求的页表项时才会使用额外的总线周期从内存中读取页表项。

即使 CR0 中的 PG 位处于复位状态（PG=0），我们也能先加载 CR3。以允许对分页机制进行初始化。当切换任务时，CR3 的内容也会随之改变。但是如果新任务的 CR3 值与原任务的一样，处理器就无需刷新页高速缓冲。这样共享页表的任务可以执行得更快。

CR2 用于出现页异常时报告出错信息。在报告页异常时，处理器会把引起异常的线性地址存放在 CR2 中。因此操作系统中的页异常处理程序可以通过检查 CR2 的内容来确定线性地址空间中哪一个页面引发了异常。

## 4.1.4 系统指令

系统指令用于处理系统级功能，例如加载系统寄存器、管理中断等。大多数系统指令只能由处于特权级 0 的操作系统软件执行，其余一些指令可以在任何特权级上执行，因此应用程序也能使用。表 4-2 中列出了我们将用到的一些系统指令。其中还指出了它们是否受到保护。

表 4-2 常用系统指令列表

指令	指令全名	受保护	说明
LLDT	Load LDT Register	是	加载局部描述符表寄存器 LDTR。从内存加载 LDT 段选择符和段描述符到 LDTR 寄存器中。
SLDT	Store LDT Register	否	保存局部描述符表寄存器 LDTR。把 LDTR 中的 LDT 段选择符到内存中或通用寄存器中。
LGDT	Load GDT Register	是	加载全局描述符表寄存器 GDTR。把 GDT 表的基地址和长度从内存加载到 GDTR 中。
SGDT	Store GDT Register	否	保存全局描述符表寄存器 GDTR。把 GDTR 中 IDT 表的基地址和长度保存到内存中。
LTR	Load Task Register	是	加载任务寄存器 TR。把 TSS 段选择符（和段描述符）加载到任务寄存器中。
STR	Store Task Register	否	保存任务寄存器 TR。把 TR 中当前任务 TSS 段选择符保存到内存或通用寄存其中。
LIDT	Load IDT Register	是	加载中断描述符表寄存器 IDTR。把 IDT 表的基地址和长度从内存加载到 IDTR 中。
SIDT	Store IDT Register	否	保存中断描述符表寄存器 IDTR。把 IDTR 中 IDT 表的基地址和长度保存到内存中。

MOV CRn	Move Control Registers	是	加载和保存控制寄存器 CR0、CR1、CR2 或 CR3。
LMSW	Load Machine State Word	是	加载机器状态字（对应 CR0 寄存器位 15--0）。该指令用于兼容 80286 处理器。
SMSW	Store Machine State Word	否	保存机器状态字。该指令用于兼容 80286 处理器。
CLTS	Clear TS flag	是	清除 CR0 中的任务已切换标志 TS。用于处理设备（协处理器）不存在异常。
LSL	Load Segment Limit	否	加载段限长。
HLT	Halt Processor	否	停止处理器执行。

## 4.2 保护模式内存管理

### 4.2.1 内存寻址

内存是指一组有序字节组成的数组，每个字节有唯一的内存地址。内存寻址则是指对存储在内存中的某个指定数据对象的地址进行定位。这里，数据对象是指存储在内存中的一个指定数据类型的数值或字符串。80X86 支持多种数据类型：1 字节、2 字节（1 个字）或 4 字节（双字或长字）的无符号整型数或带符号整型数，以及多字节字符串等。通常字节中某一比特位的定位或寻址可以基于字节来寻址，因此最小数据类型的寻址是对 1 字节数据（数值或字符）的定位。通常内存地址从 0 开始编址，对于 80X86 CPU 来说，其地址总线宽度为 32 位，因此一共有  $2^{32}$  个不同物理地址。即内存物理地址空间有 4G，总共可以寻址 4G 字节的物理内存。对于多字节数据类型（例如 2 字节整数数据类型），在内存中这些字节相邻存放。80X86 首先存放低值字节，随后地址处存放高值字节。因此 80X86 CPU 是一种先存小值（Little Endium）的处理器。

对于 80X86 CPU，一条指令主要由操作码（Opcode）和操作对象即操作数（Operand）构成。操作数可以位于一个寄存器中，也可以在内存中。若要定位内存中的操作数，就要进行内存寻址。80X86 有许多指令的操作数涉及内存寻址，并且针对所寻址对象数据类型的不同，也有很多不同的寻址方案可供选择。为了进行内存寻址，

80X86 使用了一种称为段（Segment）的寻址技术。这种寻址技术把内存空间分成一个或多个称为段的线性区域，从而对内存中一个数据对象的寻址就需要使用一个段的起始地址（即段地址）和一个段内偏移地址两部分构成。段地址部分使用 16 位的段选择符指定，其中 14 位可以选择  $2^{14}$  次方即 16384 个段。段内偏移地址部分使用 32 位的值来指定，因此段内地址可以是 0 到 4G。即一个段的最大长度可达 4G。程序中由 16 位的段和 32 位的偏移构成的 48 位地址或长指针称为一个逻辑地址（虚拟地址）。它唯一确定了一个数据对象的段地址和段内偏移地址。而仅由 32 位偏移地址或指针指定的地址是基于当前段的对象地址。

80X86 为段部分提供了 6 个存放段选择符的段寄存器：CS、DS、ES、SS、FS 和 GS。其中 CS 总是用于寻址代码段，而堆栈段则专门使用 SS 段寄存器。在任何指定时刻由 CS 寻址的段称为当前代码段。此时 EIP 寄存器中包含了当前代码段内下一条要执行指令的段内偏移地址。因此要执行指令的地址可表示成 CS:[EIP]。后面将说明的段间控制转移指令可以被用来为 CS 和 EIP 赋予新值，从而可以把执行位置改变到其他的代码段中，这样就实现了在不同段中程序的控制传递。

由段寄存器 SS 寻址的段称为当前堆栈段。栈顶由 ESP 寄存器内容指定。因此堆栈顶处地址是 SS:[ESP]。另外 4 个段寄存器是通用段寄存器。当指令中没有指定所操作数据的段时，那么 DS 将是默认的数据段寄存器。

为了指定内存操作数的段内偏移地址，80X86 指令规定了计算偏移量的很多方式，称为指令寻址方式。指令的偏移量由三部分相加组成：基地址寄存器、变址寄存器和一个偏移常量。即：

偏移地址 = 基地址 + （变址 × 比例因子）+ 偏移量



### 4.2.2 地址变换

任何完整的内存管理系统都包含两个关键部分：保护和地址变换。提供保护措施是可以防止一个任务访问另一个任务或操作系统的内存区域。地址变换能够让操作系统在给任务分配内存时具有灵活性，并且因为我们可以让某些物理地址不被任何逻辑地址所映射，所以在地址变换过程中同时也提供了内存保护功能。

正如上面提到的，计算机中的物理内存是字节的线性数组，每个字节具有一个唯一的物理地址；程序中的地址是由两部分构成的逻辑地址。这种逻辑地址并不能直接用于访问物理内存，而需要使用地址变换机制将它变换或映射到物理内存地址上。内存管理机制即用于将这种逻辑地址转换成物理内存地址。

为了减少确定地址变换所需要的信息，变换或映射通常以内存块作为操作单位。分段机制和分页机制是两种广泛使用的地址变换技术。它们的不同之处在于逻辑地址是如何组织成被映射的内存块、变换信息如何指定以及编程人员如何进行操作。分段和分页操作都使用驻留在内存中的表来指定它们各自的变换信息。这些表只能由操作系统访问，以防止应用程序擅自修改。

80X86 在从逻辑地址到物理地址变换过程中使用了分段和分页两种机制，见图 4-4 所示。第一阶段使用分段机制把程序的逻辑地址变换成处理器可寻址内存空间（称为线性地址空间）中的地址。第二阶段使用分页机制把线性地址转换为物理地址。在地址变换过程中，第一阶段的分段变换机制总是使用的，而第二阶段的分页机制则是供选用的。如果没有启用分页机制，那么分段机制产生的线性地址空间就直接映射到处理器的物理地址空间上。物理地址空间定义为处理器在其地址总线上能够产生的地址范围。

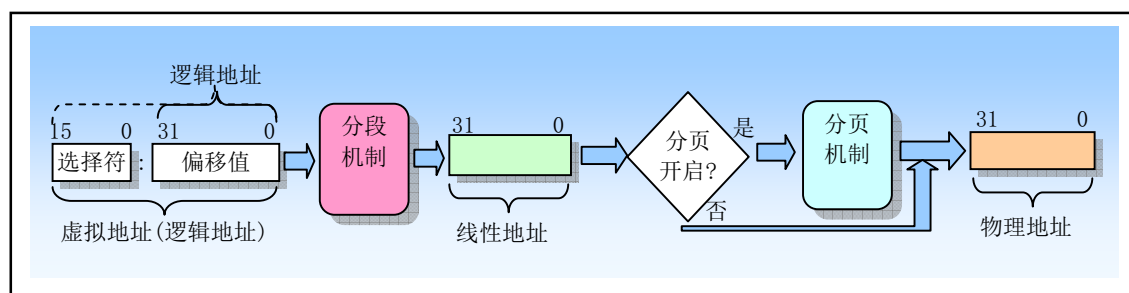


图 4-4 虚拟地址（逻辑地址）到物理地址的变换过程

#### 1. 分段机制

分段提供了隔绝各个代码、数据和堆栈区域的机制，因此多个程序（或任务）可以运行在同一个处理器上而不会互相干扰。分页机制为传统需求页、虚拟内存系统提供了实现机制。其中虚拟内存系统用于实现程序代码按要求被映射到物理内存中。分页机制当然也能用于提供多任务之间的隔离措施。

如图 4-5 所示，分段提供了一种机制，用于把处理器可寻址的线性地址空间划分成一些较小的称为段的受保护地址空间区域。段可以用来存放程序的代码、数据和堆栈，或者用来存放系统数据结构（例如 TSS 或 LDT）。如果处理器中有多个程序或任务在运行，那么每个程序可分配各自的一套段。此时处理器就可以加强这些段之间的界限，并且确保一个程序不会通过访问另一个程序的段而干扰程序的执行。分段机制还允许对段进行分类。这样，对特定类型段的操作能够受到限制。

一个系统中所有使用的段都包含在处理器线性地址空间中。为了定位指定段中的一个字节，程序必须提供一个逻辑地址。逻辑地址包括一个段选择符和一个偏移量。段选择符是一个段的唯一标识。另外，段选择符提供了段描述符表（例如全局描述符表 GDT）中一个数据结构（称为段描述符）的偏移量。每个段都有一个段描述符。段描述符指明段的大小、访问权限和段的特权级、段类型以及段的第 1 个字节在线性地址空间中的位置（称为段的基地址）。逻辑地址的偏移量部分加到段的基地址上就可以定位段中某个字节的位置。因此基地址加上偏移量就形成了处理器线性地址空间中的地址。

线性地址空间与物理地址空间具有相同的结构。相对于两维的逻辑地址空间来说，它们两者都是一维地址空间。虚拟地址（逻辑地址）空间可包含最多 16K 的段，而每个段最长可达 4GB，使得虚拟地址空间容量达到 64TB ( $2^{46}$ )。线性地址空间和物理地址空间都是 4GB ( $2^{32}$ )。实际上，如果禁用分页机制，那么线性地址空间就是物理地址空间。

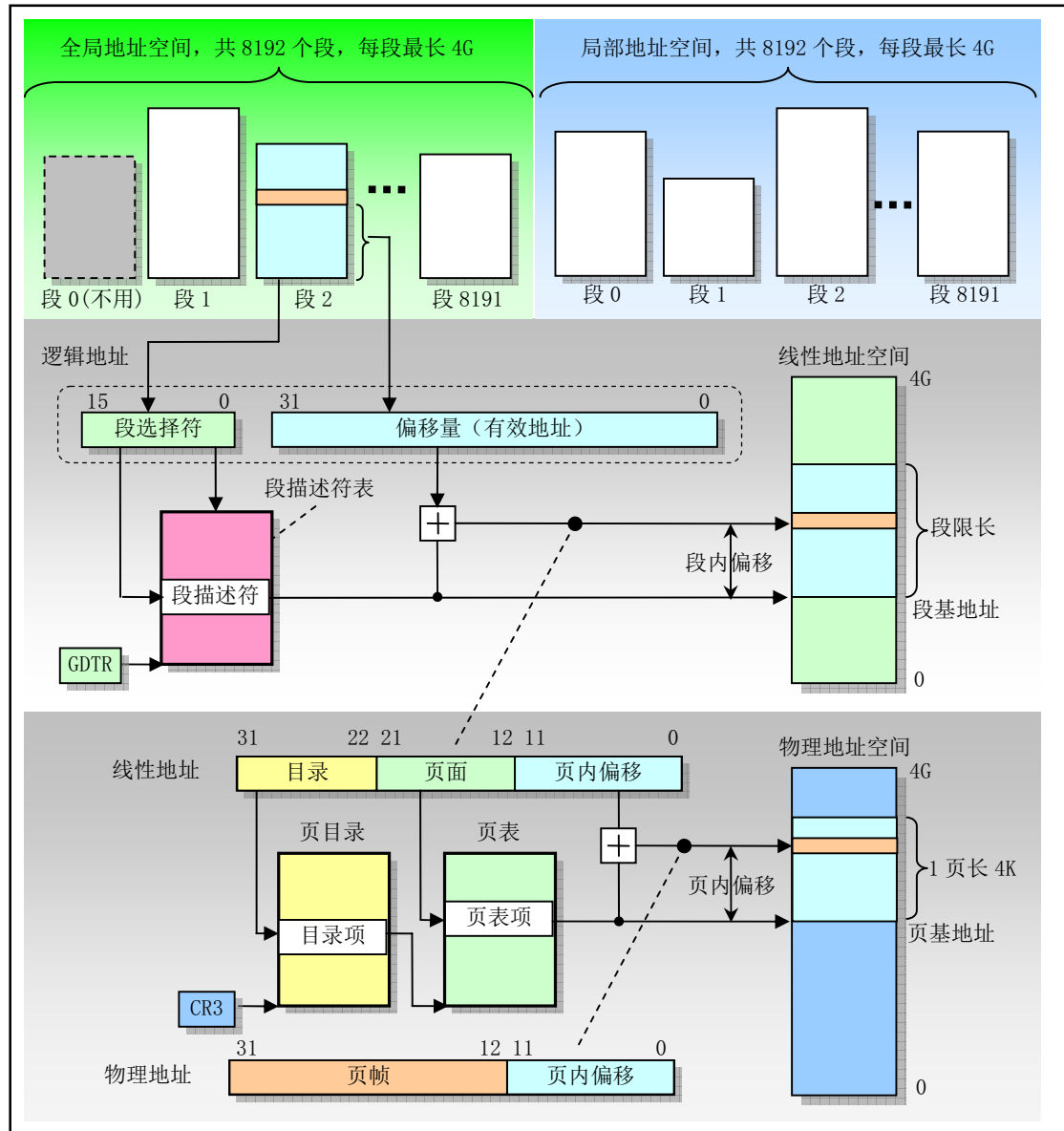


图 4-5 逻辑地址、线性地址和物理地址之间的变换

## 2. 分页机制

因为多任务系统通常定义的线性地址空间都要比其含有的物理内存容量大得多，所以需要使用某种“虚拟化”线性地址空间的方法，即使用虚拟存储技术。虚拟存储是一种内存管理技术，使用这种技术可让编程人员产生内存空间要比计算机中实际物理内存容量大很多的错觉。利用这种错觉，我们可以随意编制大型程序而无需考虑实际物理内存究竟有多少。

分页机制支持虚拟存储技术。在使用虚拟存储的环境中，大容量的线性地址空间需要使用小块的物理内存（RAM 或 ROM）以及某些外部存储空间（例如大容量硬盘）来模拟。当使用分页时，每个段被划分

成页面（通常每页为 4KB 大小），页面会被存储于物理内存中或硬盘上。操作系统通过维护一个页目录和一些页表来留意这些页面。当程序（或任务）试图访问线性地址空间中的一个地址位置时，处理器就会使用页目录和页表把线性地址转换成一个物理地址，然后在该内存位置上执行所要求的操作（读或写）。

如果当前被访问的页面不在物理内存中，处理器就会中断程序的执行（通过产生一个页错误异常）。然后操作系统就可以从硬盘上把该页面读入物理内存中，并继续执行刚才被中断的程序。当操作系统严格实现了分页机制时，那么对于正确执行的程序来说页面在物理内存和硬盘之间的交换就是透明的。

80X86 分页机制最适合支持虚拟存储技术。分页机制会使用大小固定的内存块，而分段管理则使用了大小可变的块来管理内存。无论在物理内存中还是在硬盘上，分页使用固定大小的块更为适合管理物理内存。另一方面，分段机制使用大小可变的块更适合处理复杂系统的逻辑分区。可以定义与逻辑块大小适合的内存单元而无需受到固定大小页面的限制。每个段都可以作为一个单元来处理，从而简化了段的保护和共享操作。

分段和分页是两种不同的地址变换机制，它们都对整个地址变换操作提供独立的处理阶段。尽管两种机制都使用存储在内存中的变换表，但所用的表结构不同。实际上，段表存储在线性地址空间，而页表则保存在物理地址空间。因而段变换表可由分页机制重新定位而无需段机制的信息或合作。段变换机制把虚拟地址（逻辑地址）变换成线性地址，并且在线性地址空间中访问自己的表，但是并不知晓分页机制把这些线性地址转换到物理地址的过程。类似地，分页机制也不知道程序产生地址的虚拟地址空间。分页机制只是简单地把线性地址转换成物理地址，并且在物理内存中访问自己的转换表。

### 4.2.3 保护

80X86 支持两类保护。其一是通过给每个任务不同的虚拟地址（逻辑地址）空间来完全隔离各个任务。这是通过给每个任务逻辑地址到物理地址不同的变换映射来做到。另一个保护机制对任务进行操作，以保护操作系统内存段和处理器特殊系统寄存器不被应用程序访问。

#### 1. 任务之间的保护

保护的一个重要方面是提供应用程序各任务之间的保护能力。80X86 使用的方法是通过把每个任务放置在不同的虚拟地址空间中，并给予每个任务不同的逻辑地址到物理地址的变换映射。每个任务中的地址变换功能被定义成一个任务中的逻辑地址映射到物理内存的一部分区域，而另一个任务中的逻辑地址映射到物理内存中的不同区域中。这样，因为一个任务不可能生成能够映射到其他任务逻辑地址对应使用的物理内存部分，所以所有任务都被隔绝开了。只需给每个任务各自独立的映射表，每个任务就会有不同的地址变换函数。在 80X86 中，每个任务都有自己的段表和页表。当处理器切换去执行一个新任务时，任务切换的关键部分就是切换到新任务的变换表。

通过在所有任务中安排具有相同的虚拟到物理地址映射部分，并且把操作系统存储在这个公共的虚拟地址空间部分，操作系统可以被所有任务共享。这个所有任务都具有的相同虚拟地址空间部分被称为全局地址空间（Global address space）。这也正是现代 Linux 操作系统使用虚拟地址空间的方式。

每个任务唯一的虚拟地址空间部分被称为局部地址空间（Local address space）。局部地址空间含有需要与系统中其他任务区别开的私有的代码和数据。由于每个任务中具有不同的局部地址空间，因此两个不同任务中对相同虚拟地址处的引用将转换到不同的物理地址处。这使得操作系统可以给与每个任务的内存相同的虚拟地址，但仍然能隔绝每个任务。另一方面，所有任务在全局地址空间中对相同虚拟地址的引用将被转换到同一个物理地址处。这给公共代码和数据（例如操作系统）的共享提供了支持。

#### 2. 特权级保护

在一个任务中，定义了 4 个执行特权级（Privilege Levels），用于依据段中含有数据的敏感度以及任务中不同程序部分的受信程度，来限制对任务中各段的访问。最敏感的数据被赋予了最高特权级，它们只能被任务中最受信任的部分访问。不太敏感的数据被赋予较低的特权级，它们可以被任务中较低特权级的代码访问。

特权级用数字 0 到 3 表示，0 具有最高特权级，而 3 则是最低特权级。每个内存段都与一个特权级相

关联。这个特权级限制具有足够特权级的程序来访问一个段。我们知道，处理器从 CS 寄存器指定的段中取得和执行指令，当前特权级（Current Privilege Level），即 CPL 就是当前活动代码段的特权级，并且它定义了当前所执行程序的特权级别。CPL 确定了哪些段能够被程序访问。

每当程序企图访问一个段时，当前特权级就会与段的特权级进行比较，以确定是否有访问许可。在给定的 CPL 级别上执行的程序允许访问同级别或低级别的数据段。任何对高级别段的引用都是非法的，并且会引发一个异常来通知操作系统。

每个特权级都有自己的程序栈，以避免使用共享栈带来的保护问题。当程序从一个特权级切换到另一个特权级上执行时，堆栈段也随之改换到新级别的堆栈中。

## 4.3 分段机制

分段机制可用于实现多种系统设计。这些设计范围从使用分段机制的最小功能来保护程序的平坦模型，到使用分段机制创建一个可同时可靠地运行多个程序（或任务）的具有稳固操作环境的多段模型。

多段模型能够利用分段机制全部功能提供由硬件增强的代码、数据结构、程序和任务的保护措施。通常，每个程序（或任务）都使用自己的段描述符表以及自己的段。对程序来说段能够完全是私有的，或者是程序之间共享的。对所有段以及系统上运行程序各自执行环境的访问都由硬件控制。

访问检查不仅能够用来保护对段界限以外地址的引用，而且也能用来在某些段中防止执行不允许的操作。例如，因为代码段被设计成是只读形式的段，因此可以用硬件来防止对代码段执行写操作。段中的访问权限信息也可以用来设置保护环或级别。保护级别可用于保护操作系统程序不受应用程序非法访问。

### 4.3.1 段的定义

在上一节概述中已经提到，保护模式中 80X86 提供了 4GB 的物理地址空间。这是处理器在其地址总线上可以寻址的地址空间。这个地址空间是平坦的，地址范围从 0 到 0xFFFFFFFF。这个物理地址空间可以映射到读写内存、只读内存以及内存映射 I/O 中。分段机制就是把虚拟地址空间中的虚拟内存组织成一些长度可变的称为段的内存块单元。80386 虚拟地址空间中的虚拟地址（逻辑地址）由一个段部分和一个偏移部分构成。段是虚拟地址到线性地址转换机制的基础。每个段由三个参数定义：

1. 段基地址（Base address），指定段在线性地址空间中的开始地址。基地址是线性地址，对应于段中偏移 0 处。
2. 段限长（limit），是虚拟地址空间中段内最大可用偏移位置。它定义了段的长度。
3. 段属性（Attributes），指定段的特性。例如该段是否可读、可写或可作为一个程序执行；段的特权级等。

段限长定义了虚拟地址空间中段的大小。段基址和段限长定义了段所映射的线性地址范围或区域。段内 0 到 limit 的地址范围对应线性地址中范围 base 到 base + limit。偏移量大于段限长的虚拟地址是无意义的，如果使用则会导致异常。另外，若访问一个段并没有得到段属性许可也会导致异常。例如，如果你试图写一个只读的段，那么 80386 就会产生一个异常。另外，多个段映射到线性地址中的范围可以部分重叠或覆盖，甚至完全重叠，见图 4-6 所示。在本书介绍的 Linux 0.1x 系统中，一个任务的代码段和数据段的段限长相同，并被映射到线性地址完全相同而重叠的区域上。

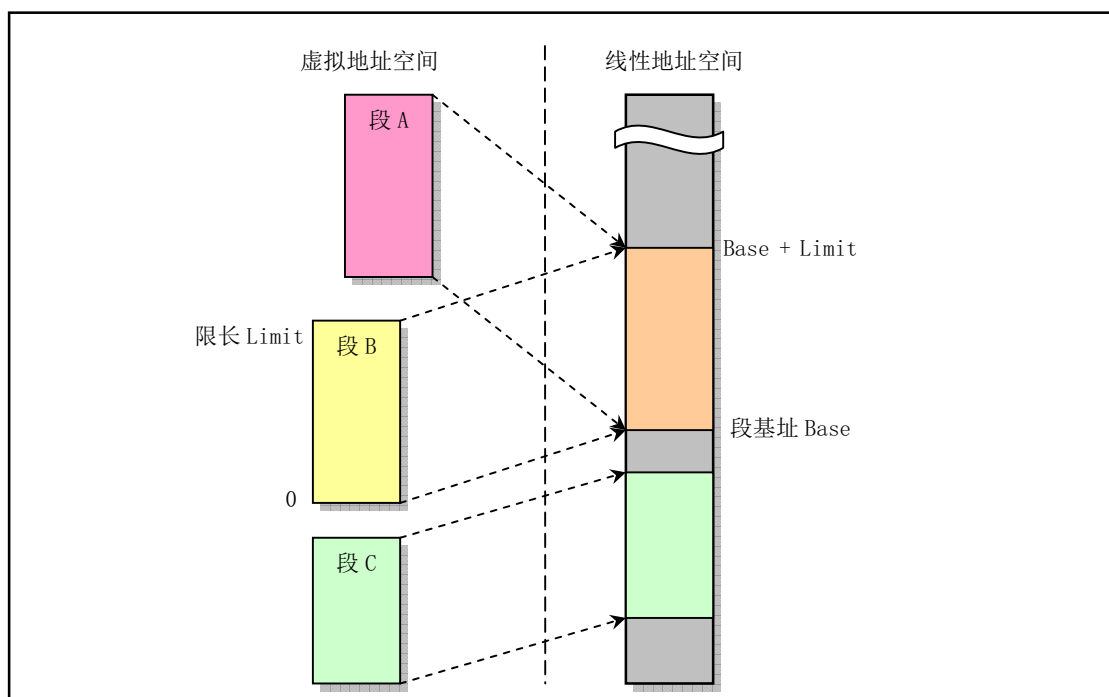


图 4-6 虚拟（逻辑）地址空间中的段映射到线性地址空间

段的基地址、段限长以及段的保护属性存储在一个称为段描述符（Segment Descriptor）的结构项中。在逻辑地址到线性地址的转换映射过程中会使用这个段描述符。段描述符保存在内存中的段描述符表（Descriptor table）中。段描述符表是包含段描述符项的一个简单数组。前面介绍的段选择符即用于通过指定表中一个段描述符的位置来指定相应的段。

即使利用段的最小功能，使用逻辑地址也能访问处理器地址空间中的每个字节。逻辑地址由 16 位的段选择符和 32 位的偏移量组成，见图 4-7 所示。段选择符指定字节所在的段，而偏移量指定该字节在段中相对于段基地址的位置。处理器会把每个逻辑地址转换成线性地址。线性地址是处理器线性地址空间中的 32 位地址。与物理地址空间类似，线性地址空间也是平坦的 4GB 地址空间，地址范围从 0 到 0xFFFFFFFF。线性地址空间中含有为系统定义的所有段和系统表。

为了把逻辑地址转换成一个线性地址，处理器会执行以下操作：

1. 使用段选择符中的偏移值（段索引）在 GDT 或 LDT 表中定位相应的段描述符。（仅当一个新的段选择符加载到段寄存器中时才需要这一步。）
2. 利用段描述符检验段的访问权限和范围，以确保该段是可访问的并且偏移量位于段界限内。
3. 把段描述符中取得的段基地址加到偏移量上，最后形成一个线性地址。

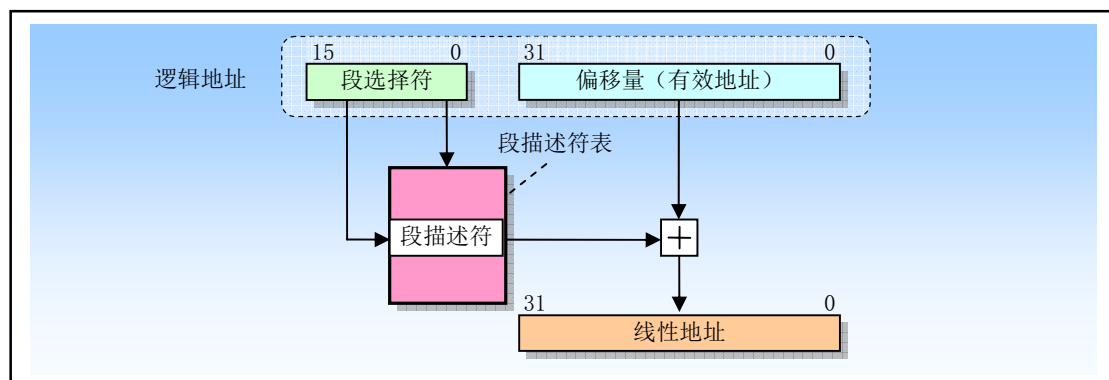


图 4-7 逻辑地址到线性地址的变换过程

如果没有开启分页，那么处理器直接把线性地址映射到物理地址（即线性地址被送到处理器地址总线上）。如果对线性地址空间进行了分页处理，那么就会使用二级地址转换把线性地址转换成物理地址。页转换将在稍后进行说明。

### 4.3.2 段描述符表

段描述符表是段描述符的一个数组，见图 4-8 所示。描述符表的长度可变，最多可以包含 8192 个 8 字节描述符。有两种描述符表：全局描述符表 GDT（Global descriptor table）；局部描述符表 LDT（Local descriptor table）。

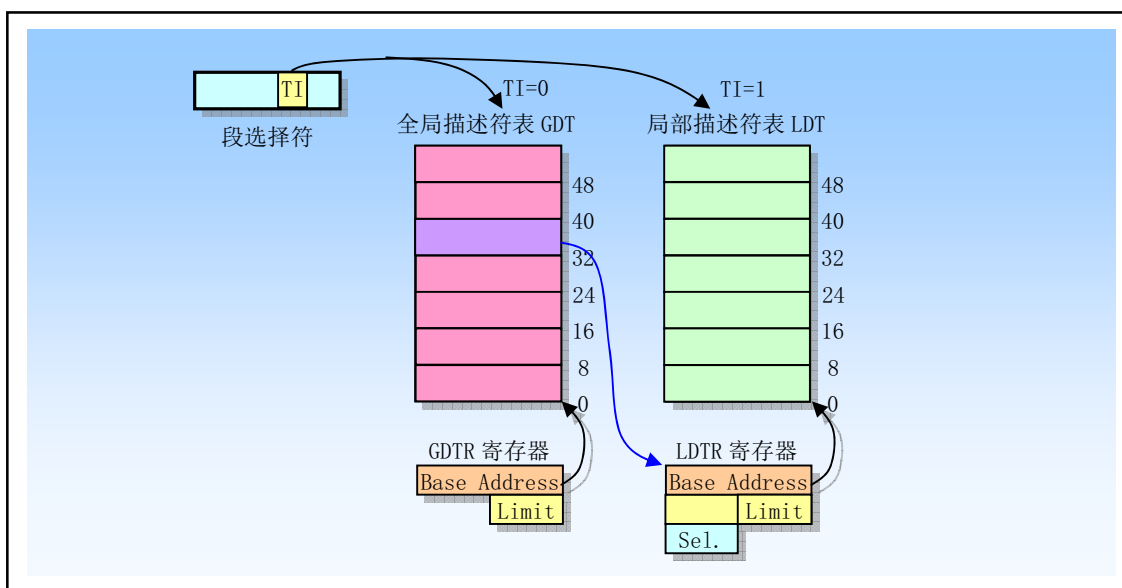


图 4-8 段描述符表结构

描述符表存储在由操作系统维护着的特殊数据结构中，并且由处理器的内存管理硬件来引用。这些特殊结构应该保存在仅由操作系统软件访问的受保护的内存区域中，以防止应用程序修改其中的地址转换信息。虚拟地址空间被分割成大小相等的两半。一半由 GDT 来映射变换到线性地址，另一半则由 LDT 来映射。整个虚拟地址空间共含有  $2^{14}$  个段：一半空间（即  $2^{13}$  个段）是由 GDT 映射的全局虚拟地址空间，另一半是由 LDT 映射的局部虚拟地址空间。通过指定一个描述符表（GDT 或 LDT）以及表中描述符号，我们就可以定位一个描述符。

当发生任务切换时，LDT 会更换成新任务的 LDT，但是 GDT 并不会改变。因此，GDT 所映射的一半虚拟地址空间是系统中所有任务共有的，但是 LDT 所映射的另一半则在任务切换时被改变。系统中所有任务共享的段由 GDT 来映射。这样的段通常包括含有操作系统的段以及所有任务各自的包含 LDT 的特殊段。LDT 段可以想象成属于操作系统的数据。

图 4-9 示出一个任务中的段如何能在 GDT 和 LDT 之间分开。图中共有 6 个段，分别用于两个应用程序（A 和 B）以及操作系统。系统中每个应用程序对应一个任务，并且每个任务有自己的 LDT。应用程序 A 在任务 A 中运行，拥有 LDT<sub>A</sub>，用来映射段 Code<sub>A</sub> 和 Data<sub>A</sub>。类似地，应用程序 B 在任务 B 中运行，使用 LDT<sub>B</sub> 来映射 Code<sub>B</sub> 和 Data<sub>B</sub> 段。包含操作系统内核的两个段 Code<sub>OS</sub> 和 Data<sub>OS</sub> 使用 GDT 来映射，这样它们可以被两个任务所共享。两个 LDT 段：LDT<sub>A</sub> 和 LDT<sub>B</sub> 也使用 GDT 来映射。



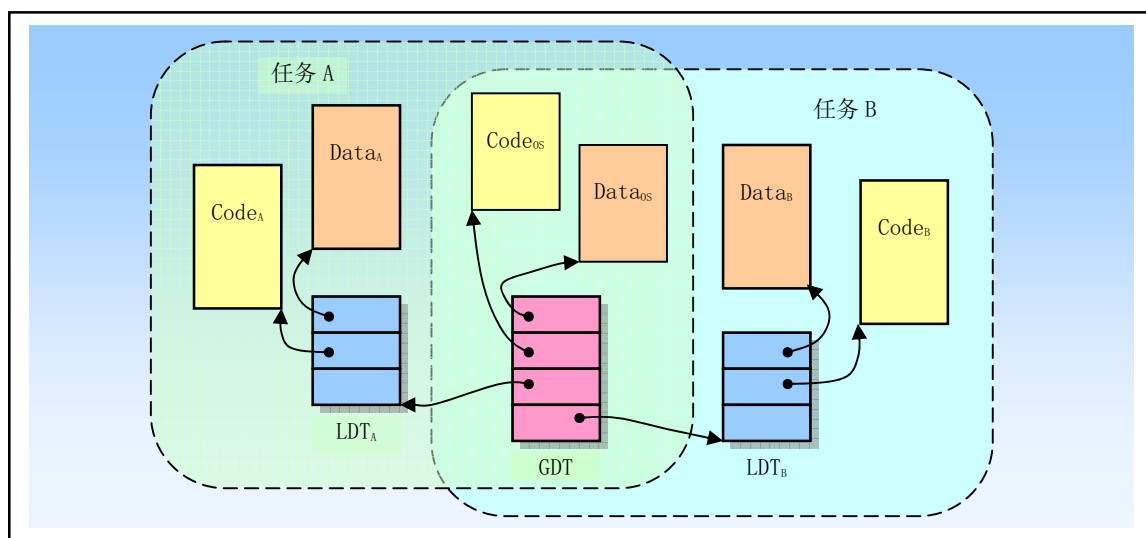


图 4-9 任务所用的段类型

当任务 A 在运行时，可访问的段包括  $LDT_A$  映射的  $Code_A$  和  $Data_A$  段，加上 GDT 映射的操作系统的段  $Code_{OS}$  和  $Data_{OS}$ 。当任务 B 在运行时，可访问的段包括  $LDT_B$  映射的  $Code_B$  和  $Data_B$  段，加上 GDT 映射的段。

这个例子通过让每个任务使用不同的 LDT，演示了虚拟地址空间如何能够被组织成隔离每个任务。当任务 A 在运行时，任务 B 的段不是虚拟地址空间的部分，因此任务 A 没有办法访问任务 B 的内存。同样地，当任务 B 运行时，任务 A 的段也不能被寻址。这种使用 LDT 来隔离每个应用程序任务的方法，正是关键保护需求之一。

每个系统必须定义一个 GDT，并可用于系统中所有程序或任务。另外，可选定义一个或多个 LDT。例如，可以为每个运行任务定义一个 LDT，或者某些或所有任务共享一个 LDT。

GDT 本身并不是一个段，而是线性地址空间中的一个数据结构。GDT 的基线性地址和长度值必须加载进 GDTR 寄存器中。GDT 的基地址应该进行内存 8 字节对齐，以得到最佳处理器性能。GDT 的限长以字节为单位。与段类似，限长值加上基地址可得到最后表中最后一个字节的有效地址。限长为 0 表示有 1 个有效字节。因为段描述符总是 8 字节长，因此 GDT 的限长值应该设置成总是 8 的倍数减 1（即  $8N-1$ ）。

处理器并不使用 GDT 中的第 1 个描述符。把这个“空描述符”的段选择符加载进一个数据段寄存器（DS、ES、FS 或 GS）并不会产生一个异常，但是若使用这些加载了空描述符的段选择符访问内存时就肯定会产生一般保护性异常。通过使用这个段选择符初始化段寄存器，那么意外引用未使用的段寄存器肯定会产生一个异常。

LDT 表存放在 LDT 类型的系统段中。此时 GDT 必须含有 LDT 的段描述符。如果系统支持多 LDT 的话，那么每个 LDT 都必须在 GDT 中有一个段描述符和段选择符。一个 LDT 的段描述符可以存放在 GDT 表的任何地方。

访问 LDT 需使用其段选择符。为了在访问 LDT 时减少地址转换次数，LDT 的段选择符、基地址、段限长以及访问权限需要存放在 LDTR 寄存器中。

当保存 GDTR 寄存器内容时（使用 SGDT 指令），一个 48 位的“伪描述符”被存储在内存中。为了在用户模式（特权级 3）避免对齐检查出错，伪描述符应该存放在一个奇数地址处（即地址  $\text{MOD } 4 = 2$ ）。这会让处理器先存放一个对齐的字，随后是一个对齐的双字（4 字节对齐处）。用户模式程序通常不会保存伪描述符，但是可以通过使用这种对齐方式来避免产生一个对齐检查出错的可能性。当使用 SIDT 指令保存 IDTR 寄存器内容时也需要使用同样的对齐方式。然而，当保存 LDTR 或任务寄存器（分别使用 SLTR 或 STR 指令）时，伪描述符应该存放在双字对齐的地址处（即地址  $\text{MOD } 4 = 0$ ）。

### 4.3.3 段选择符

段选择符（或称段选择子）是段的一个 16 位标识符，见图 4-10 所示。段选择符并不直接指向段，而是指向段描述符表中定义段的段描述符。段选择符 3 个字段内容：

- 请求特权级 RPL（Requested Privilege Level）；
- 表指示标志 TI（Table Index）；
- 索引值（Index）。

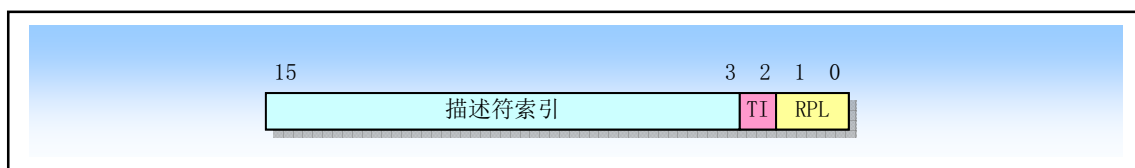


图 4-10 段选择符结构

请求特权级字段 RPL 提供了段保护信息，将在后面作详细说明。表索引字段 TI 用来指出包含指定段描述符的段描述符表 GDT 或 LDT。TI=0 表示描述符在 GDT 中；TI=1 表示描述符在 LDT 中。索引字段给出了描述符在 GDT 或 LDT 表中的索引项号。可见，选择符通过定位段表中的一个描述符来指定一个段，并且描述符中包含有访问一个段的所有信息，例如段的基地址、段长度和段属性。

例如，图 4-11(a)中选择符（0x008）指定了 GDT 中具有 RPL=0 的段 1，其索引字段值是 1，TI 位是 0，指定 GDT 表。图 4-11(b)中选择符（0x010）指定了 GDT 中具有 RPL=0 的段 2，其索引字段值是 2，TI 位是 0，指定 GDT 表。图 4-11(c)中选择符（0x00f）指定了 LDT 中具有 RPL=3 的段 1，其索引字段值是 1，TI 位是 1，指定 LDT 表。图 4-11(d)中选择符（0x017）指定了 LDT 中具有 RPL=3 的段 2，其索引字段值是 2，TI 位是 1，指定 LDT 表。实际上，图 4-11 中的前 4 个选择符：(a)、(b)、(c)和(d)分别就是 Linux 0.1x 内核的内核代码段、内核数据段、任务代码段和任务数据段的选择符。图 4-11(e)中的选择符（0xffff）指定 LDT 表中 RPL=3 的段 8191。其索引字段值是 0b11111111111111（即 8191），TI 位等于 1，指定 LDT 表。

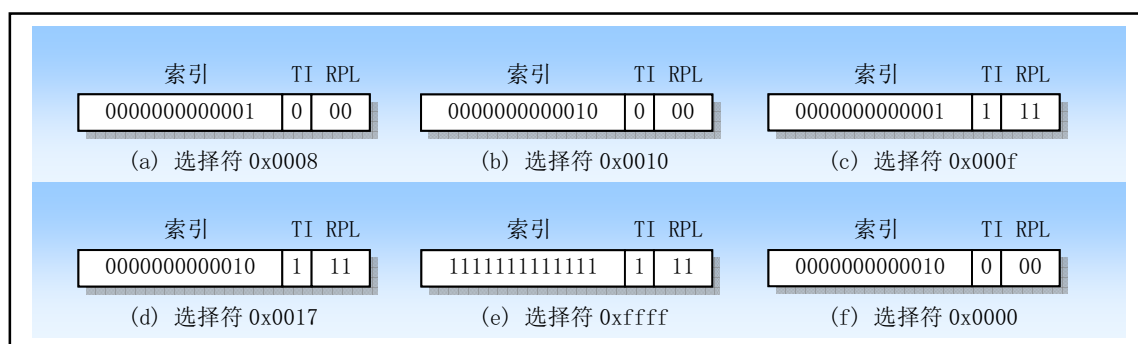


图 4-11 段选择符示例

另外，处理器不使用 GDT 表中的第 1 项。指向 GDT 该项的选择符（即索引值为 0，TI 标志为 0 的选择符）用作为“空选择符”，见图 4-11(f)所示。当把空选择符加载到一个段寄存器（除了 CS 和 SS 以外）中时，处理器并不产生异常。但是当使用含有空选择符的段寄存器用于访问内存时就会产生异常。当把空选择符加载到 CS 或 SS 段寄存器中时将会导致一个异常。

对应用程序来说段选择符是作为指针变量的一部分而可见，但选择符的值通常是由链接编辑器或链接



加载程序进行设置或修改，而非应用程序。

为减少地址转换时间和编程复杂性，处理器提供可存放最多 6 个段选择符的寄存器（见图 4-12 所示），即段寄存器。每个段寄存器支持特定类型的内存引用（代码、数据或堆栈）。原则上执行每个程序都起码需要把有效的段选择符加载到代码段（CS）、数据段（DS）和堆栈段（SS）寄存器中。处理器还另外提供三个辅助的数据段寄存器（ES、FS 和 GS），可被用于让当前执行程序（或任务）能够访问其他几个数据段。

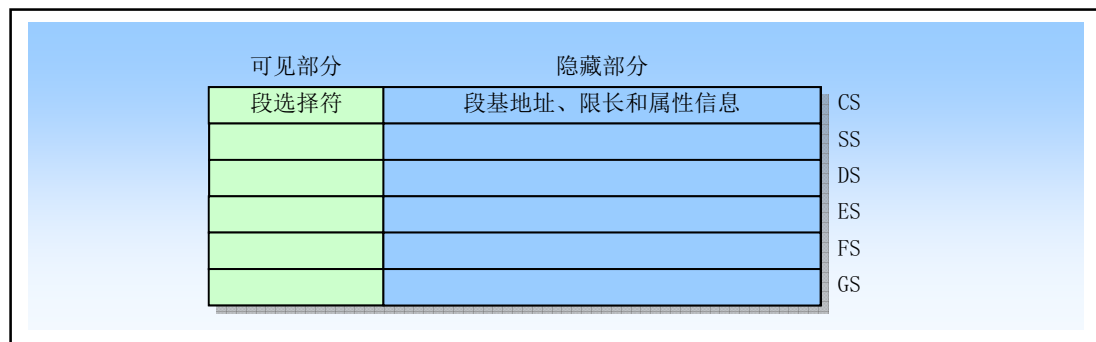


图 4-12 段寄存器结构

对于访问某个段的程序，必须已经把段选择符加载到一个段寄存器中。因此，尽管一个系统可以定义很多的段，但同时只有 6 个段可供立即访问。若要访问其他段就需要加载这些段的选择符。

另外，为了避免每次访问内存时都去引用描述符表，去读和解码一个段描述符，每个段寄存器都有一个“可见”部分和一个“隐藏”部分（隐藏部分也被称为“描述符缓冲”或“影子寄存器”）。当一个段选择符被加载到一个段寄存器可见部分中时，处理器也同时把段选择符指向的段描述符中的段地址、段限长以及访问控制信息加载到段寄存器的隐藏部分中。缓冲在段寄存器（可见和隐藏部分）中的信息使得处理器可以在进行地址转换时不再需要花费时间从段描述符中读取基地址和限长值。

由于影子寄存器含有描述符信息的一个拷贝，因此操作系统必须确保对描述符表的改动应反映在影子寄存器中。否则描述符表中一个段的基地址或限长被修改过，但改动却没有反映到影子寄存器中。处理这种问题最简捷的方法是在对描述符表中描述符作过任何改动之后就立刻重新加载 6 个段寄存器。这将把描述符表中的相应段信息重新加载到影子寄存器中。

为加载段寄存器，提供了两类加载指令：

1. 象 MOV、POP、LDS、LES、LSS、LGS 以及 LFS 指令。这些指令显式地直接引用段寄存器；
2. 隐式加载指令，例如使用长指针的 CALL、JMP 和 RET 指令、IRET、INTn、INTO 和 INT3 等指令。这些指令在操作过程中会附带改变 CS 寄存器（和某些其他段寄存器）的内容。

MOV 指令当然也可以用于把段寄存器可见部分内容存储到一个通用寄存器中。

#### 4.3.4 段描述符

前面我们已经说明了使用段选择符来定位描述符表中的一个描述符。段描述符是 GDT 和 LDT 表中的一个数据结构项，用于向处理器提供有关一个段的位置和大小信息以及访问控制的状态信息。每个段描述符长度是 8 字节，含有三个主要字段：段基地址、段限长和段属性。段描述符通常由编译器、链接器、加载器或者操作系统来创建，但绝不是应用程序。图 4-13 示出了所有类型段描述符的一般格式。

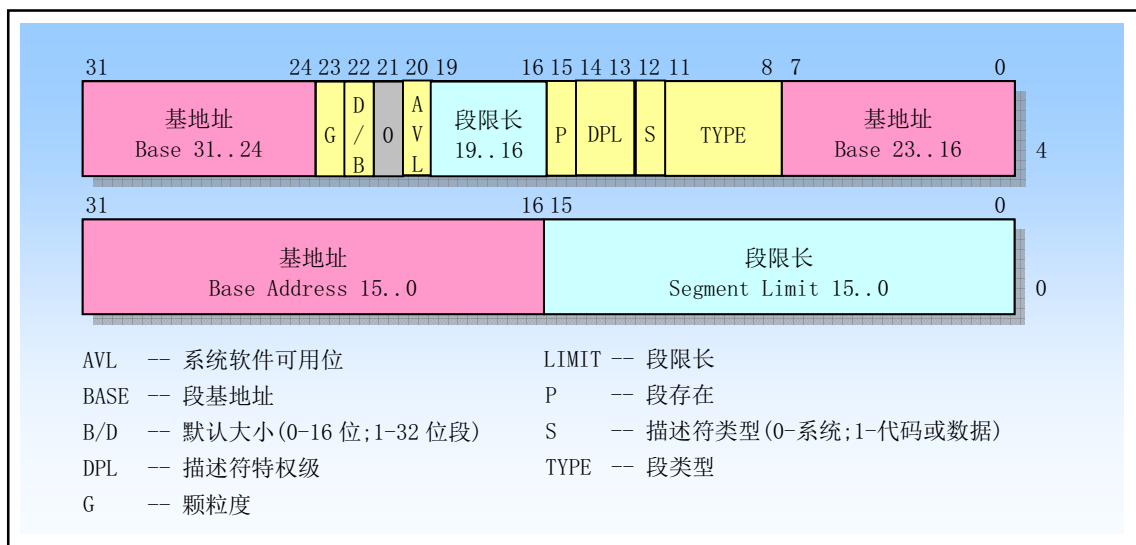


图 4-13 段描述符通用格式

一个段描述符中各字段和标志的含义如下：

◆段限长字段 LIMIT (Segment limit field)

段限长 Limit 字段用于指定段的长度。处理器会把段描述符中两个段限长字段组合成一个 20 位的值，并根据颗粒度标志 G 来指定段限长 Limit 值的实际含义。如果 G=0，则段长度 Limit 范围可从 1 字节到 1MB 字节，单位是字节。如果 G=1，则段长度 Limit 范围可从 4KB 到 4GB，单位是 4KB。

根据段类型中的段扩展方向标志 E，处理器以两种不同方式使用段限长 Limit。对于向上扩展的段（简称上扩段），逻辑地址中的偏移值范围可以从 0 到段限长值 Limit。大于段限长 Limit 的偏移值将产生一般保护性异常。对于向下扩展的段（简称下扩段），段限长 Limit 的含义相反。根据默认栈指针大小标志 B 的设置，偏移值范围可从段限长 Limit 到 0xFFFFFFFF 或 0xFFFF。而小于段限长 Limit 的偏移值将产生一般保护性异常。对于下扩段，减小段限长字段中的值会在该段地址空间底部分配新的内存，而不是在顶部分配。80X86 的栈总是向下扩展的，因此这种实现方式很适合扩展堆栈。

◆基地址字段 BASE (Base address field)

该字段定义在 4GB 线性地址空间中一个段字节 0 所处的位置。处理器会把 3 个分立的基地址字段组合形成一个 32 位的值。段基地址应该对齐 16 字节边界。虽然这不是要求的，但通过把程序的代码和数据段对齐在 16 字节边界上，可以让程序具有最佳性能。

◆段类型字段 TYPE (Type field)

类型字段指定段或门 (Gate) 的类型、说明段的访问种类以及段的扩展方向。该字段的解释依赖于描述符类型标志 S 指明是一个应用（代码或数据）描述符还是一个系统描述符。TYPE 字段的编码对代码、数据或系统描述符都不同，见图 4-14 所示。

◆描述符类型标志 S (Descriptor type flag)

描述符类型标志 S 指明一个段描述符是系统段描述符 (当 S=0) 还是代码或数据段描述符 (当 S=1)。

◆描述符特权级字段 DPL (Descriptor privilege level)

DPL 字段指明描述符的特权级。特权级范围从 0 到 3。0 级特权级最高，3 级最低。DPL 用于控制对段的访问。

◆段存在标志 P (Segment present)

段存在标志 P 指出一个段是在内存中 (P=1) 还是不在内存中 (P=0)。当一个段描述符的 P 标志

为 0 时，那么把指向这个段描述符的选择符加载进段寄存器将导致产生一个段不存在异常。内存管理软件可以使用这个标志来控制在某一给定时间实际需要把那个段加载进内存中。这个功能为虚拟存储提供了除分页机制以外的控制。图 4-15 给出了当  $P=0$  时的段描述符格式。当  $P$  标志为 0 时，操作系统可以自由使用格式中标注为可用（Available）的字段位置来保存自己的数据，例如有关不存在段实际在什么地方信息。

◆ D/B（默认操作大小/默认栈指针大小和/或上界限）标志

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

根据段描述符描述的是一个可执行代码段、下扩数据段还是一个堆栈段，这个标志具有不同的功能。（对于 32 位代码和数据段，这个标志应该总是设置为 1；对于 16 位代码和数据段，这个标志被设置为 0。）

- 可执行代码段。此时这个标志称为 **D** 标志并用于指出该段中的指令引用有效地址和操作数的默认长度。如果该标志置位，则默认值是 32 位地址和 32 位或 8 位的操作数；如果该标志为 0，则默认值是 16 位地址和 16 位或 8 位的操作数。指令前缀 0x66 可以用来选择非默认值的操作数大小；前缀 0x67 可用来选择非默认值的地址大小。
- 栈段（由 **SS** 寄存器指向的数据段）。此时该标志称为 **B**（Big）标志，用于指明隐含堆栈操作（例如 **PUSH**、**POP** 或 **CALL**）时的栈指针大小。如果该标志置位，则使用 32 位栈指针并存放在 **ESP** 寄存器中；如果该标志为 0，则使用 16 位栈指针并存放在 **SP** 寄存器中。如果堆栈段被设置成一个下扩数据段，这个 **B** 标志也同时指定了堆栈段的上界限。
- 下扩数据段。此时该标志称为 **B** 标志，用于指明堆栈段的上界限。如果设置了该标志，则堆栈段的上界限是 0xFFFFFFFF（4GB）；如果没有设置该标志，则堆栈段的上界限是 0xFFFF（64KB）。

◆ 颗粒度标志 **G**（Granularity）

该字段用于确定段限长字段 **Limit** 值的单位。如果颗粒度标志为 0，则段限长值的单位是字节；如果设置了颗粒度标志，则段限长值使用 4KB 单位。（这个标志不影响段基地址的颗粒度，基地址的颗粒度总是字节单位。）若设置了 **G** 标志，那么当使用段限长来检查偏移值时，并不会去检查偏移值的 12 位最低有效位。例如，当  $G=1$  时，段限长为 0 表明有效偏移值为 0 到 4095。

◆ 可用和保留比特位（Available and reserved bits）

段描述符第 2 个双字的位 20 可供系统软件使用；位 21 是保留位并应该总是设置为 0。

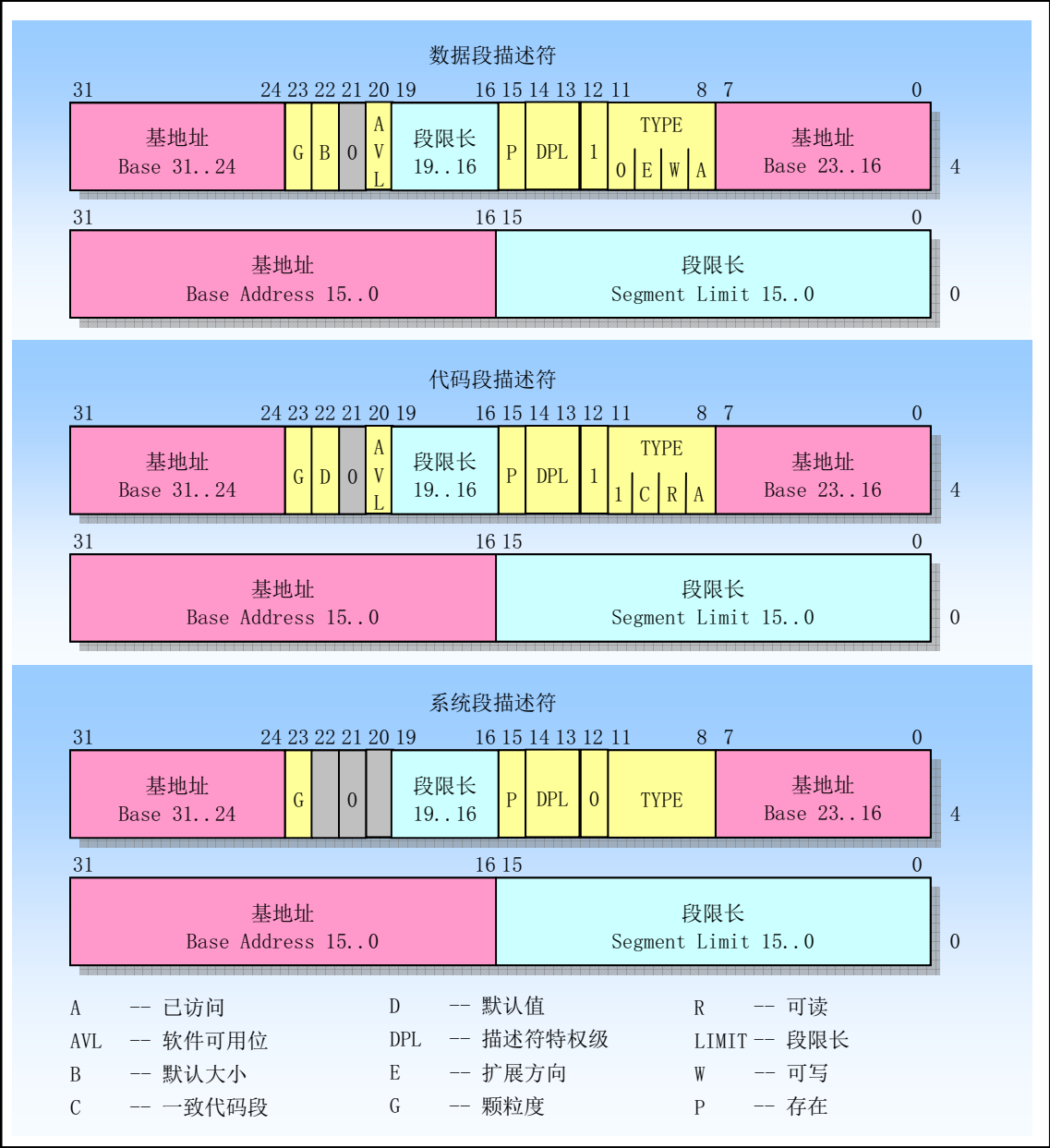


图 4-14 代码段、数据段和系统段描述符格式

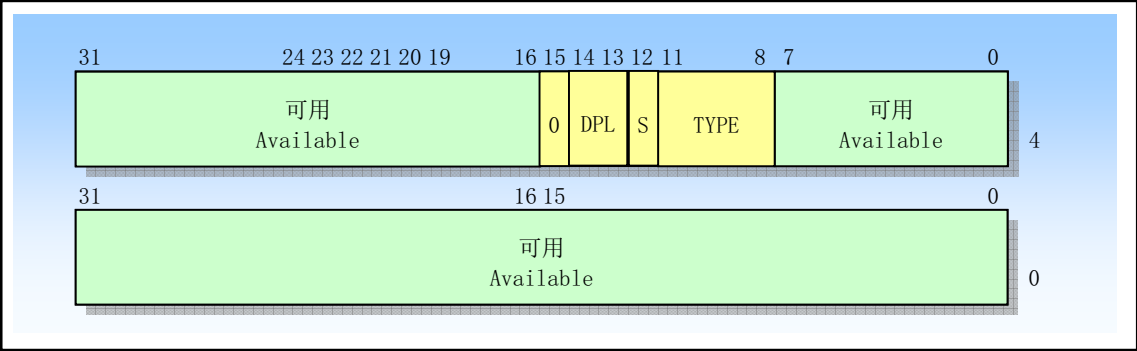


图 4-15 当存在位 P=0 时的段描述符格式

### 4.3.5 代码和数据段描述符类型

当段描述符中 S（描述符类型）标志被置位，则该描述符用于代码或数据段。此时类型字段中最高比特位（第 2 个双字的位 11）用于确定是数据段的描述符（复位）还是代码段的描述符（置位）。

对于数据段的描述符，类型字段的低 3 位（位 8、9、10）被分别用于表示已访问 A（Accessed）、可写 W（Write-enable）和扩展方向 E（Expansion-direction），参见表 4-3 中有关代码和数据段类型字段比特位的说明。根据可写比特位 W 的设置，一个数据段可以是只读的，也可以是可读可写的。

表 4-3 代码段和数据段描述符类型

类型（TYPE）字段					描述符 类型	说明
十进制	位 11	位 10	位 9	位 8		
		E	W	A		
0	0	0	0	0	数据	只读
1	0	0	0	1	数据	只读，已访问
2	0	0	1	0	数据	可读/写
3	0	0	1	1	数据	可读/写，已访问
4	0	1	0	0	数据	向下扩展，只读
5	0	1	0	1	数据	向下扩展，只读，已访问
6	0	1	1	0	数据	向下扩展，可读/写
7	0	1	1	1	数据	向下扩展，可读/写，已访问
		C	R	A		
8	1	0	0	0	代码	仅执行
9	1	0	0	1	代码	仅执行，已访问
10	1	0	1	0	代码	执行/可读
11	1	0	1	1	代码	执行/可读，已访问
12	1	1	0	0	代码	一致性段，仅执行
13	1	1	0	1	代码	一致性段，仅执行，已访问
14	1	1	1	0	代码	一致性段，执行/可读
15	1	1	1	1	代码	一致性段，执行/可读，已访问

堆栈段必须是可读/写的数据段。若使用不可写数据段的选择符加载到 SS 寄存器中，将导致一个一般保护异常。如果堆栈段的长度需要动态地改变，那么堆栈段可以是一个向下扩展的数据段（扩展方向标志置位）。这里，动态改变段限长将导致栈空间被添加到栈底部。

已访问比特位指明自从上次操作系统复位该位之后一个段是否被访问过。每当处理器把一个段的段选择符加载进段寄存器，它就会设置该位。该位需要明确地清除，否则一直保持置位状态。该位可用于虚拟内存管理和调试。

对于代码段，类型字段的低 3 位被解释成已访问 A（Accessed）、可读 R（Read-enable）和一致的 C（Conforming）。根据可读 R 标志的设置，代码段可以是只能执行、可执行/可读。当常数或其他静态数据以及指令码被放在了一个 ROM 中时就可以使用一个可执行/可读代码段。这里，通过使用带 CS 前缀的指令或者把代码段选择符加载进一个数据段寄存器（DS、ES、FS 或 GS），我们可以读取代码段中的数据。在保护模式下，代码段是不可写的。

代码段可以是一致性的或非一致性的。向更高特权级一致性代码段的执行控制转移，允许程序以当前

特权级继续执行。向一个不同特权级的非一致性代码段的转移将导致一般保护异常，除非使用了一个调用门或任务门（有关一致性和非一致性代码段的详细信息请参见“直接调用或跳转到代码段”）。不访问保护设施的系统工具以及某些异常类型（例如除出错、溢出）的处理过程可以存放在一致性代码段中。需要防止低特权级程序或过程访问的工具应该存放在非一致性代码段中。

所有数据段都是非一致性的，即意味着它们不能被低特权级的程序或过程访问。然而，与代码段不同，数据段可以被更高特权级的程序或过程访问，而无须使用特殊的访问门。

如果 GDT 或 LDT 中一个段描述符被存放在 ROM 中，那么若软件或处理器试图更新（写）在 ROM 中的段描述符时，处理器就会进入一个无限循环。为了防止这个问题，需要存放在 ROM 中的所有描述符的已访问位应该预先设置成置位状态。同时，删除操作系统中任何试图修改 ROM 中段描述符的代码。

### 4.3.6 系统描述符类型

当段描述符中的 S 标志（描述符类型）是复位状态（0）的话，那么该描述符是一个系统描述符。处理器能够识别以下一些类型的系统段描述符：

- 局部描述符表（LDT）的段描述符；
- 任务状态段（TSS）描述符；
- 调用门描述符；
- 中断门描述符；
- 陷阱门描述符；
- 任务门描述符。

这些描述符类型可分为两大类：系统段描述符和门描述符。系统段描述符指向系统段（如 LDT 和 TSS 段），门描述符就是一个“门”，对于调用、中断或陷阱门，其中含有代码段的选择符和段中程序入口点的指针；对于任务门，其中含有 TSS 的段选择符。表 4-4 给出了系统段描述符和门描述符类型字段的编码。

表 4-4 系统段和门描述符类型

类型（TYPE）字段					说明	
十进制	位 11	位 10	位 9	位 8		
0	0	0	0	0	Reserved	保留
1	0	0	0	1	16-Bit TSS (Available)	16 位 TSS（可用）
2	0	0	1	0	LDT	LDT
3	0	0	1	1	16-Bit TSS (Busy)	16 位 TSS（忙）
4	0	1	0	0	16-Bit Call Gate	16 位调用门
5	0	1	0	1	Task Gate	任务门
6	0	1	1	0	16-Bit Interrupt Gate	16 位中断门
7	0	1	1	1	16-Bit Trap Gate	16 位陷阱门
8	1	0	0	0	Reserved	保留
9	1	0	0	1	32-Bit TSS (Available)	32 位 TSS（可用）
10	1	0	1	0	Reserved	保留
11	1	0	1	1	32-Bit TSS (Busy)	32 位 TSS（忙）
12	1	1	0	0	32-Bit Call gate	32 位调用门
13	1	1	0	1	Reserved	保留
14	1	1	1	0	32-Bit Interrupt Gate	32 位中断门
15	1	1	1	1	32-Bit Trap Gate	32 位陷阱门

有关 TSS 状态段和任务门的使用方法将在任务管理一节中进行说明，调用门的使用方法将放在保护一节中说明，中断和陷阱门的使用方法将在中断和异常处理一节中给予说明。

## 4.4 分页机制

分页机制是 80X86 内存管理机制的第二部分。它在分段机制的基础上完成虚拟（逻辑）地址到物理地址转换的过程。分段机制把逻辑地址转换成线性地址，而分页则把线性地址转换成物理地址。分页可以用于任何一种分段模型。处理器分页机制会把线性地址空间（段已映射到其中）划分成页面，然后这些线性地址空间页面被映射到物理地址空间的页面上。分页机制几种页面级保护措施，可和分段机制保护机制合用或替代分段机制的保护措施。例如，在基于页面的基础上可以加强读/写保护。另外，在页面单元上，分页机制还提供了用户 - 超级用户两级保护。

我们通过设置控制寄存器 CR0 的 PG 位可以启用分页机制。如果 PG=1，则启用分页操作，处理器会使用本节描述的机制将线性地址转换成物理地址。如果 PG=0，则禁用分页机制，此时分段机制产生的线性地址被直接用作物理地址。

前面介绍的分段机制在各种可变长度的内存区域上操作。与分段机制不同，分页机制对固定大小的内存块（称为页面）进行操作。分页机制把线性和物理地址空间都划分成页面。线性地址空间中的任何页面可以被映射到物理地址空间的任何页面上。图 4-16 示出了分页机制是如何把线性和物理地址空间都划分成各个页面，并在这两个空间之间提供了任意映射。图中的箭头把线性地址空间中的页面与物理地址空间中的页面对应了起来。

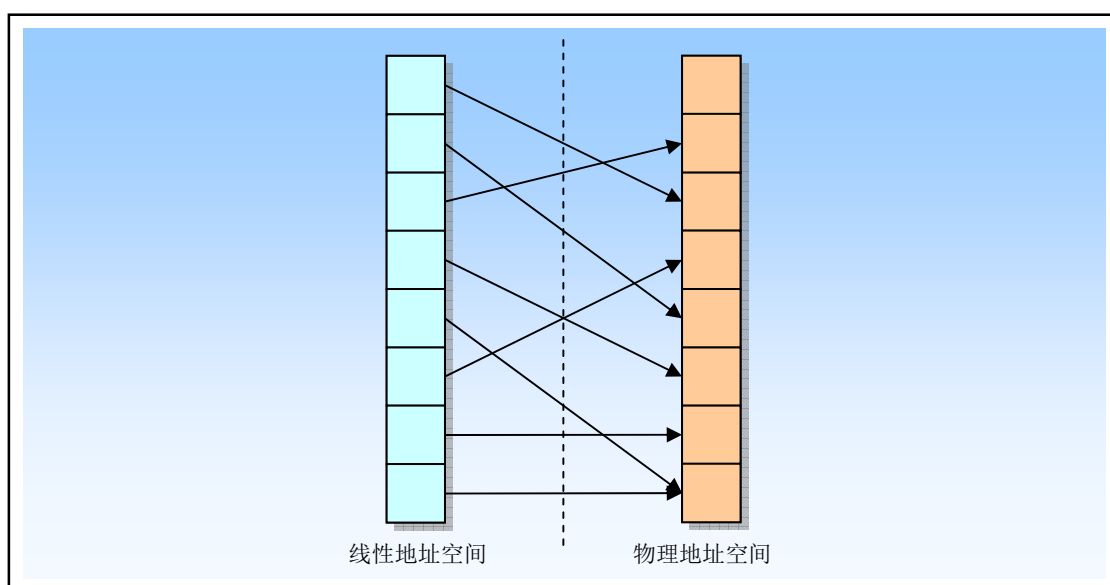


图 4-16 线性地址空间页面到物理地址空间页面对应示意图

80X86 使用 4K ( $2^{12}$ ) 字节固定大小的页面。每个页面均是 4KB，并且对齐于 4K 地址边界处。这表示分页机制把  $2^{32}$  字节 (4GB) 的线性地址空间划分成  $2^{20}$  ( $1\text{M} = 1048576$ ) 个页面。分页机制通过把线性地址空间中的页面重新定位到物理地址空间中进行操作。由于 4K 大小的页面作为一个单元进行映射，并且对齐于 4K 边界，因此线性地址的低 12 比特位可作为页内偏移量直接作为物理地址的低 12 位。分页机制执行的重定位功能可以看作是线性地址的高 20 位转换到对应物理地址的高 20 位。

另外，线性到物理地址的转换功能被扩展成允许一个线性地址被标注为无效的，而非让其产生一个物理地址。在两种情况下一个页面可以被标注为无效的：①操作系统不支持的线性地址；②对应应在虚拟内存系统中的页面在磁盘上而非在物理内存中。在第一种情况下，产生无效地址的程序必须被终止。在第二种

情况下，该无效地址实际上是请求操作系统虚拟内存管理器把对应页面从磁盘上加载到物理内存中，以供程序访问。因为无效页面通常与虚拟存储系统相关，因此它们被称为不存在的页面，并且由页表中称为存在（present）的属性来确定。

在保护模式中，80X86 允许线性地址空间直接映射到大容量的物理内存（例如 4GB 的 RAM）上，或者（使用分页）间接地映射到较小容量的物理内存和磁盘存储空间中。这后一种映射线性地址空间的方法被称为虚拟存储或者需求页（Demand-paged）虚拟存储。

当使用分页时，处理器会把线性地址空间划分成固定大小的页面（长度 4KB），这些页面可以映射到物理内存中和/或磁盘存储空间中。当一个程序（或任务）引用内存中的逻辑地址时，处理器会把该逻辑地址转换成一个线性地址，然后使用分页机制把该线性地址转换成对应的物理地址。

如果包含线性地址的页面当前不在物理内存中，处理器就会产生一个页错误异常。页错误异常的处理程序通常就会让操作系统从磁盘把相应页面加载到物理内存中（操作过程中可能还会把物理内存中不同的页面写到磁盘上）。当页面加载到物理内存中之后，从异常处理过程的返回操作会使得导致异常的指令被重新执行。处理器用于把线性地址转换成物理地址和用于产生页错误异常（若必要的话）的信息包含在存储于内存中的页目录和页表中。

分页与分段最大的不同之处在于分页使用了固定长度的页面。段的长度通常与存放在其中的代码或数据结构具有相同的长度。与段不同，页面有固定的长度。如果仅使用分段地址转换，那么存储在物理内存中的一个数据结构将包含其所有的部分。但如果使用了分页，那么一个数据结构就可以一部分存储于物理内存中，而另一部分保存在磁盘中。

正如前述，为了减少地址转换所要求的总线周期数量，最近访问的页目录和页表会被存放在处理器的缓冲器件中，该缓冲器件被称为转换查找缓冲区 TLB（Translation Lookaside Buffer）。TLB 可以满足大多数读页目录和页表的请求而无需使用总线周期。只有当 TLB 中不包含要求的页表项时才会使用额外的总线周期从内存中读取页表项，这通常在一个页表项很长时间没有访问过时才会出现这种情况。

### 4.4.1 页表结构

分页转换功能由驻留在内存中的表来描述，该表称为页表（page table），存放在物理地址空间中。页表可以看作是简单的  $2^{20}$  物理地址数组。线性到物理地址的映射功能可以简单地看作是进行数组查找。线性地址的高 20 位构成这个数组的索引值，用于选择对应页面的物理（基）地址。线性地址的低 12 位给出了页面中的偏移量，加上页面的基地址最终形成对应的物理地址。由于页面基地址对齐在 4K 边界上，因此页面基地址的低 12 位肯定是 0。这意味着高 20 位的页面基地址和 12 位偏移量连接组合在一起就能得到对应的物理地址。

页表中每个页表项大小为 32 位。由于只需要其中的 20 位来存放页面的物理基地址，因此剩下的 12 位可用于存放诸如页面是否存在等的属性信息。如果线性地址索引的页表项被标注为存在的，则表示该项即有效，我们可以从中取得页面的物理地址。如果项中表明不存在，那么当访问对应物理页面时就会产生一个异常。

#### 4.4.1.1 两级页表结构

页表含有  $2^{20}$ （1M）个表项，而每项占用 4 字节。如果作为一个表来存放的话，它们最多将占用 4MB 的内存。因此为了减少内存占用量，80X86 使用了两级表。由此，高 20 位线性地址到物理地址的转换也被分成两步来进行，每步使用（转换）其中 10 个比特。

第一级表称为页目录（page directory）。它被存放在 1 页 4K 页面中，具有  $2^{10}$ （1K）个 4 字节长度的表项。这些表项指向对应的二级表。线性地址的最高 10 位（位 31--22）用作一级表（页目录）中的索引值来选择  $2^{10}$  个二级表之一。

第二级表称为页表（page table），它的长度也是 1 个页面，最多含有 1K 个 4 字节的表项。每个 4 字节表项含有相关页面的 20 位物理基地址。二级页表使用线性地址中间 10 位（位 21--12）作为表项索引值，以获取含有页面 20 位物理基地址的表项。该 20 位页面物理基地址和线性地址中的低 12 位（页内偏移）



组合在一起就得到了分页转换过程的输出值，即对应的最终物理地址。

图 4-17 示出了二级表的查找过程。其中 CR3 寄存器指定页目录表的基地址。线性地址的高 10 位用于索引这个页目录表，以获得指向相关二级页表的指针。线性地址中间 10 位用于索引二级页表，以获得物理地址的高 20 位。线性地址的低 12 位直接作为物理地址低 12 位，从而组成一个完整的 32 位物理地址。

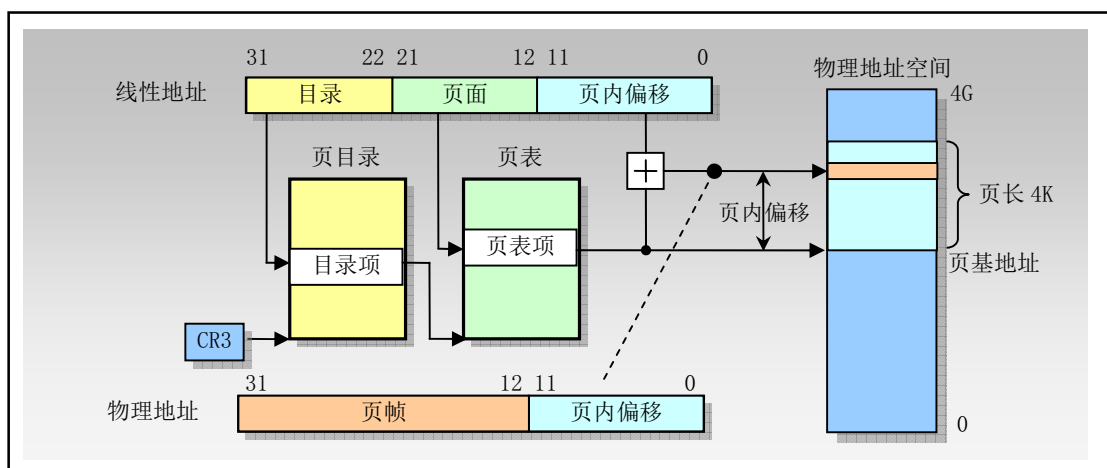


图 4-17 线性地址和物理地址之间的变换

#### 4.4.1.2 不存在的页表

通过使用二级表结构，我们还没有解决需要使用 4MB 内存来存放页表的问题。实际上，我们把问题搞得有些复杂了。因为我们需要另增一个页面来存放目录表。然而，二级表结构允许页表被分散在内存各个页面中，而不需要保存在连续的 4MB 内存块中。另外，并不需要为不存在的或线性地址空间未使用部分分配二级页表。虽然目录表页面必须总是存在于物理内存中，但是二级页表可以在需要时再分配。这使得页表结构的大小对应于实际使用的线性地址空间大小。

页目录表中每个表项也有一个存在（present）属性，类似于页表中的表项。页目录表项中的存在属性指明对应的二级页表是否存在。如果目录表项指明对应的二级页表存在，那么通过访问二级表，表查找过程第 2 步将同如上描述继续下去。如果存在位表明对应的二级表不存在，那么处理器就会产生一个异常来通知操作系统。页目录表项中的存在属性使得操作系统可以根据实际使用的线性地址范围来分配二级页表页面。

目录表项中的存在位还可以用于在虚拟内存中存放二级页表。这意味着在任何时候只有部分二级页表需要存放在物理内存中，而其余的可保存在磁盘上。处于物理内存中页表对应的页目录项将被标注为存在，以表明可用它们进行分页转换。处于磁盘上的页表对应的页目录项将被标注为不存在。由于二级页表不存在而引发的异常会通知操作系统把缺少的页表从磁盘上加载进物理内存。把页表存储在虚拟内存中减少了保存分页转换表所需要的物理内存量。

#### 4.4.2 页表项格式

页目录和页表的表项格式见图 4-18 所示。其中位 31--12 含有物理地址的高 20 位，用于定位物理地址空间中一个页面（也称为页帧）的物理基地址。表项的低 12 位含有页属性信息。我们已经讨论过存在属性，这里简要说明其余属性的功能和用途。

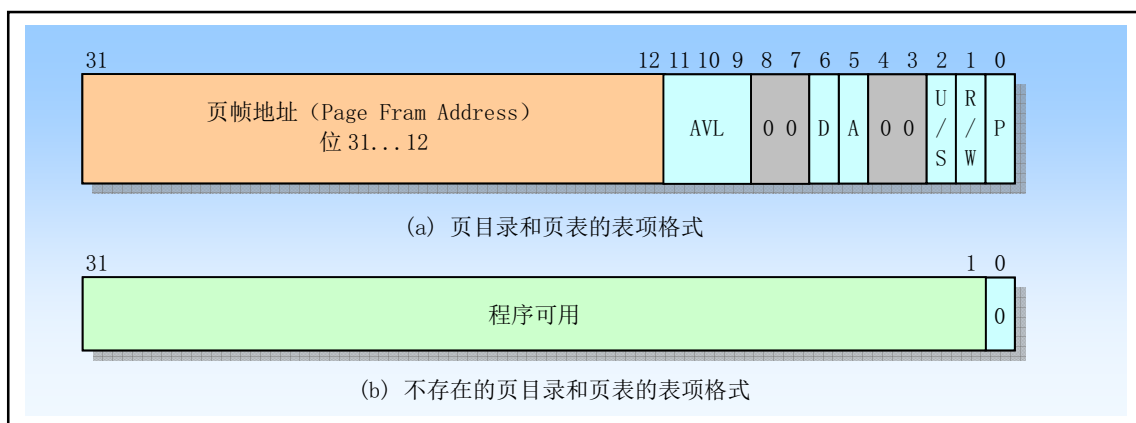


图 4-18 页目录和页表的表项格式

- **P** -- 位 0 是存在（Present）标志，用于指明表项对地址转换是否有效。P=1 表示有效；P=0 表示无效。在页转换过程中，如果说涉及的页目录或页表的表项无效，则会导致一个异常。如果 P=0，那么除表示表项无效外，其余比特位可供程序自由使用，见图 4-18(b)所示。例如，操作系统可以使用这些位来保存已存储在磁盘上的页面的序号。
- **R/W** -- 位 1 是读/写（Read/Write）标志。如果等于 1，表示页面可以被读、写或执行。如果为 0，表示页面只读或可执行。当处理器运行在超级用户特权级（级别 0、1 或 2）时，则 R/W 位不起作用。页目录项中的 R/W 位对其所映射的所有页面起作用。
- **U/S** -- 位 2 是用户/超级用户（User/Supervisor）标志。如果为 1，那么运行在任何特权级上的程序都可以访问该页面。如果为 0，那么页面只能被运行在超级用户特权级（0、1 或 2）上的程序访问。页目录项中的 U/S 位对其所映射的所有页面起作用。
- **A** -- 位 5 是已访问（Accessed）标志。当处理器访问页表项映射的页面时，页表表项的这个标志就会被置为 1。当处理器访问页目录表项映射的任何页面时，页目录表项的这个标志就会被置为 1。处理器只负责设置该标志，操作系统可通过定期地复位该标志来统计页面的使用情况。
- **D** -- 位 6 是页面已被修改（Dirty）标志。当处理器对一个页面执行写操作时，就会设置对应页表表项的 D 标志。处理器并不会修改页目录项中的 D 标志。
- **AVL** -- 该字段保留专供程序使用。处理器不会修改这几位，以后的升级处理器也不会。

### 4.4.3 虚拟存储

页目录和页表表项中的存在标志 P 为使用分页技术的虚拟存储提供了必要的支持。若线性地址空间中的页面存在于物理内存中，则对应表项中的标志 P=1，并且该表项中含有相应物理地址。页面不在物理内存中的表项其标志 P=0。如果程序访问物理内存中不存在的页面，处理器就会产生一个缺页异常。此时操作系统就可以利用这个异常处理过程把缺少的页面从磁盘上调入物理内存中，并把相应物理地址存放在表项中。最后在返回程序重新执行引起异常的指令之前设置标志 P=1。

已访问标志 A 和已修改标志 D 可以用于有效地实现虚拟存储技术。通过周期性地检查和复位所有 A 标志，操作系统能够确定哪些页面最近没有访问过。这些页面可以成为移出到磁盘上的候选者。假设当一页从磁盘上读入内存时，其脏标志 D=0，那么当页面再次被移出到磁盘上时，若 D 标志还是为 0，则该页面就无需被写入磁盘中。若此时 D=1，则说明页面内容已被修改过，于是就必须将该页面写到磁盘上。

## 4.5 保护

保护机制是可靠的多任务运行环境所必须的。它可用于保护各个任务免受相互之间的干扰。在软件开发的任何阶段都可以使用段级和页级保护来协助寻找和检测设计问题和错误。当程序对错误内存空间执行

了一次非期望的引用，保护机制可以阻止这种操作并且报告此类事件。

保护机制可以被用于分段和分页机制。处理器寄存器的 2 个比特位定义了当前执行程序的特权级，称为当前特权级 CPL（Current Privilege Level）。在分段和分页地址转换过程中，处理器将对 CPL 进行验证。

通过设置控制寄存器 CR0 的 PE 标志（位 0）可以让处理器工作在保护模式下，从而也就开启了分段保护机制。一旦进入保护模式，处理器中并不存在明确的控制标志来停止或启用保护机制。不过基于特权级的保护机制部分可以通过把所有段选择符和段描述符的特权级都设置为 0 级来隐含地关闭。这种处理方式可以在段之间禁止特权级保护壁垒，但是其他段长度和段类型检查等保护机制仍然起作用。

设置控制寄存器 CR0 的 PG 标志（位 31）可以开启分页机制，同时也开启了分页保护机制。同样，处理器中也没有相关的标志用来在分页开启条件下禁止或开启页级保护机制。但是通过设置每个页目录项和页表项的读/写（R/W）标志和用户/超级用户（U/S）标志，我们可以禁止页级保护机制。设置这两个标志可以使得每个页面都可以被任意读/写，因此实际上也就禁止了页级保护。

对于分段级保护机制，处理器使用段寄存器中选择符（RPL 和 CPL）和段描述符中各个字段执行保护验证。对于分页机制，则主要利用页目录和页表项中的 R/W 和 U/S 标志来实现保护操作。

### 4.5.1 段级保护

在保护模式下，80X86 提供了段级和页级保护机制。这种保护机制根据特权级（4 级段保护和 2 级页保护）提供了对某些段和页面的访问限制能力。例如，操作系统代码和数据存放在要比普通应用程序具有高特权级的段中。此后处理器的保护机制将会限制应用程序只能按照受控制的和规定的方式访问操作系统的代码和数据。

当使用保护机制时，每个内存引用都将受到检查以验证内存引用符合各种保护要求。因为检查操作是与地址变换同时并行操作，所以处理器性能并没有受到影响。所进行的保护检查可分为以下几类：

- 段界限检查；
- 段类型检查；
- 特权级检查；
- 可寻址范围限制；
- 过程入口点限制；
- 指令集限制。

所有违反保护的操作都将导致产生一个异常。下面各节描述保护模式下的保护机制。

#### 4.5.1.1 段限长 Limit 检查

段描述符的段限长（或称段界限）字段用于防止程序或过程寻址到段外内存位置。段限长的有效值依赖于颗粒度 G 标志的设置状态。对于数据段，段限长还与标志 E（扩展方向）和标志 B（默认栈指针大小和/或上界限）有关。E 标志是数据段类型的段描述符中类型字段的一个比特位。

当 G 标志清零时（字节颗粒度），有效的段长度是 20 位的段描述符中段限长字段 Limit 的值。在这种情况下，Limit 的范围从 0 到 0xFFFFF（1MB）。当 G 标志置位时（4KB 页颗粒度），处理器把 Limit 字段的值乘上一个因子 4K。在这种情况下，有效的 Limit 范围是从 0xFFF 到 0xFFFFFFFF（4GB）。请注意，当设置了 G 标志时，段偏移（地址）的低 12 位不会与 Limit 进行对照检查。例如，当段限长 Limit 等于 0 时，偏移值 0 到 0xFFF 仍然是有效的。

除了下扩段以外的所有段类型，有效 Limit 的值是段中允许被访问的最后一个地址，它要比段长度小 1 个字节。任何超出段限长字段指定的有效地址范围都将导致产生一个一般保护异常。

对于下扩数据段，段限长具有同样的功能，但其含义不同。这里，段限长指定了段中最后一个不允许访问的地址，因此在设置了 B 标志的情况下，有效偏移范围是从（有效段偏移+1）到 0xFFFF FFFF；当 B 清零时，有效偏移值范围是从（有效段偏移+1）到 0xFFFF。当下扩段的段限长为 0 时，段会有最大长度。

除了对段限长进行检查，处理器也会检查描述符表的长度。GDTR、IDTR 和 LDTR 寄存器中包含有 16 位的限长值，处理器用它来防止程序在描述符表的外面选择描述符。描述符表的限长值指明了表中最后

一个有效字节。因为每个描述符是 8 字节长，因此含有 N 个描述符项的表应该具有有限长值  $8N-1$ 。

选择符可以具有 0 值。这样的选择符指向 GDT 表中的第一个不用的描述符项。尽管这个空选择符可以被加载进一个段寄存器中，但是任何使用这种描述符引用内存的企图都将产生一个一般保护性异常。

#### 4.5.1.2 段类型 TYPE 检查

除了应用程序代码和数据段有描述符以外，处理器还有系统段和门两种描述符类型。这些数据结构用于管理任务以及异常和中断。请注意，并非所有的描述符都定义一个段，门描述符中存放有指向一个过程入口点的指针。段描述符在两个地方含有类型信息，即描述符中的 S 标志和类型字段 TYPE。处理器利用这些信息对由于非法使用段或门导致的编程错误进行检测。

S 标志用于指出一个描述符是系统类型的还是代码或数据类型的。TYPE 字段另外提供了 4 个比特位用于定义代码、数据和系统描述符的各种类型。上一节的表给出了代码和数据描述符 TYPE 字段的编码；另一个表给出了系统描述符 TYPE 字段的编码。

当操作段选择符和段描述符时，处理器会随时检查类型信息。主要在以下两种情况下检查类型信息：

1. 当一个描述符的选择符加载进一个段寄存器中。此时某些段寄存器只能存放特定类型的描述符，例如：
  - CS 寄存器中只能被加载进一个可执行段的选择符；
  - 不可读可执行段的选择符不能被加载进数据段寄存器中；
  - 只有可写数据段的选择符才能被加载进 SS 寄存器中。
2. 当指令访问一个段，而该段的描述符已经加载进段寄存器中。指令只能使用某些预定义的方法来访问某些段。
  - 任何指令不能写一个可执行段；
  - 任何指令不能写一个可写位没有置位的数据段；
  - 任何指令不能读一个可执行段，除非可执行段设置了可读标志。

#### 4.5.1.3 特权级

处理器的段保护机制可以识别 4 个特权级（或特权层），0 级到 3 级。数值越大，特权越小。图 4-19 示出了这些特权级如何能被解释成保护环形式。环中心（保留给最高级的代码、数据和堆栈）用于含有最紧要软件的段，通常用于操作系统核心部分。中间两个环用于较为紧要的软件。只使用 2 个特权级的系统应该使用特权级 0 和 3。

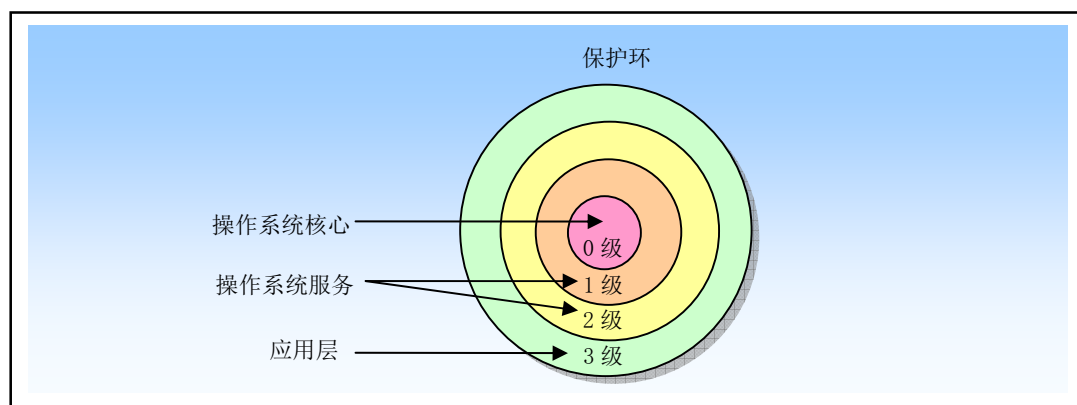


图 4-19 处理器特权级示意图

处理器利用特权级来防止运行在较低特权级的程序或任务访问具有较高特权级的一个段，除非是在受控的条件下。当处理器检测到一个违反特权级的操作时，它就会产生一个一般保护性异常。

为了在各个代码段和数据段之间进行特权级检测处理，处理器可以识别以下三种类型的特权级：

- 当前特权级 CPL（Current Privilege Level）。CPL 是当前正在执行程序或任务的特权级。它存放在

CS 和 SS 段寄存器的位 0 和位 1 中。通常, CPL 等于当前代码段的特权级。当程序把控制转移到另一个具有不同特权级的代码段中时, 处理器就会改变 CPL。当访问一个一致性 (conforming) 代码段时, 则处理器对 CPL 的设置有些不同。特权级值高于 (即低特权级) 或等于一致代码段 DPL 的任何段都可以访问一致代码段。并且当处理器访问一个特权级不同于 CPL 的一致代码段时, CPL 并不会被修改成一致代码段的 DPL。

- 描述符特权级 DPL (Descriptor Privilege Level)。DPL 是一个段或门的特权级。它存放在段或门描述符的 DPL 字段中。在当前执行代码段试图访问一个段或门时, 段或门的 DPL 会用来与 CPL 以及段或门选择符中的 RPL (见下面说明) 作比较。根据被访问的段或门的类型不同, DPL 也有不同的含义:
  - ◆ 数据段 (Data Segment)。其 DPL 指出允许访问本数据段的程序或任务应具有的最大特权级数值。例如, 如果数据段的特权级 DPL 是 1, 那么只有运行在 CPL 为 0 或 1 的程序可以访问这个段。
  - ◆ 非一致代码段 (Nonconforming code segment) (不使用调用门)。其 DPL 指出程序或任务访问该段必须具有的特权级。例如, 如果某个非一致代码段的 DPL 是 0, 那么只有运行在 CPL 为 0 的程序能够访问这个段。
  - ◆ 调用门 (Call Gate)。其 DPL 指出访问调用门的当前执行程序或任务可处于的最大特权级数值。(这与数据段的访问规则相同。)
  - ◆ 一致和非一致代码段 (通过调用门访问)。其 DPL 指出允许访问本代码段的程序或任务应具有的最小特权级数值。例如, 如果一致代码段的 DPL 是 2, 那么运行在 CPL 为 0 的程序就不能访问这个代码段。
  - ◆ 任务状态段 TSS。其 DPL 指出访问 TSS 的当前执行程序或任务可处于的最大特权级数值。(这与数据段的访问规则相同。)
- 请求特权级 RPL (Request Privilege Level)。RPL 是一种赋予段选择符的超越特权级, 它存放在选择符的位 0 和位 1 中。处理器会同时检查 RPL 和 CPL, 以确定是否允许访问一个段。即使程序或任务具有足够的特权级 (CPL) 来访问一个段, 但是如果提供的 RPL 特权级不足的话访问也将被拒绝。也即如果段选择符的 RPL 其数值大于 CPL, 那么 RPL 将覆盖 CPL (而使用 RPL 作为检查比较的特权级), 反之亦然。即始终取 RPL 和 CPL 中数值最大的特权级作为访问段时的比较对象。因此, RPL 可用来确保高特权级的代码不会代表应用程序去访问一个段, 除非应用程序自己具有访问这个段的权限。

当段描述符的段选择符被加载进一个段寄存器时就会进行特权级检查操作, 但用于数据访问的检查方式和那些用于在代码段之间进行程序控制转移的检查方式不一样。因此下面分两种访问情况来考虑。

### 4.5.2 访问数据段时的特权级检查

为了访问数据段中的操作数, 数据段的段选择符必须被加载进数据段寄存器 (DS、ES、FS 或 GS) 或堆栈段寄存器 (SS) 中。(可以使用指令 MOV、POP、LDS、LES、LFS、LGS 和 LSS 来加载段寄存器)。在把一个段选择符加载进段寄存器中之前, 处理器会进行特权级检查, 见图 4-20 所示。它会把当前运行程序或任务的 CPL、段选择符的 RPL 和段描述符的 DPL 进行比较。只有当段的 DPL 数值大于或等于 CPL 和 RPL 两者时, 处理器才会把选择符加载进段寄存器中。否则就会产生一个一般保护异常, 并且不加载段选择符。

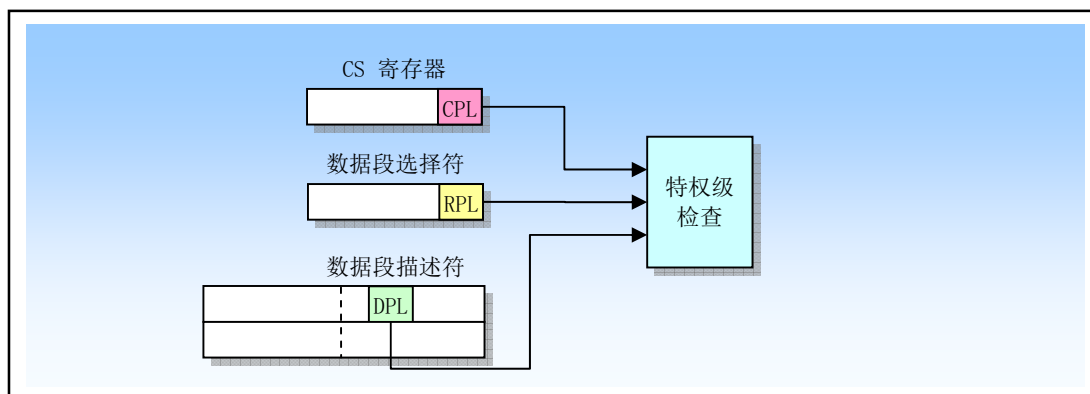


图 4-20 访问数据段时的特权级检查

可知一个程序或任务可寻址的区域随着其 CPL 改变而变化。当 CPL 是 0 时，此时所有特权级上的数据段都可被访问；当 CPL 是 1 时，只有在特权级 1 到 3 的数据段可被访问；当 CPL 是 3 时，只有处于特权级 3 的数据段可被访问。

另外，有可能会把数据保存在代码段中。例如，当代码和数据是在 ROM 中时。因此，有些时候我们会需要访问代码段中的数据。此时可以使用以下方法来访问代码段中的数据：

1. 把非一致可读代码段的选择符加载进一个数据段寄存器中。
2. 把一致可读代码段的选择符加载进一个数据段寄存器中。
3. 使用代码段覆盖前缀（CS）来读取一个选择符已经在 CS 寄存器中的可读代码段。

访问数据段的相同规则也适用方法 1。方法 2 则是总是有效的，因为一致代码段的特权级等同于 CPL，而不管代码段的 DPL。方法 3 也总是有效的，因为 CS 寄存器选择的代码段的 DPL 与 CPL 相同。

当使用堆栈段选择符加载 SS 段寄存器时也会执行特权级检查。这里与堆栈段相关的所有特权级必须与 CPL 匹配。也即，CPL、堆栈段选择符的 RPL 以及堆栈段描述符的 DPL 都必须相同。如果 RPL 或 DPL 与 CPL 不同，处理器就会产生一个一般保护性异常。

### 4.5.3 代码段之间转移控制时的特权级检查

对于将程序控制权从一个代码段转移到另一个代码段，目标代码段的段选择符必须加载进代码段寄存器（CS）中。作为这个加载过程的一部分，处理器会检测目标代码段的段描述符并执行各种限长、类型和特权级检查。如果这些检查都通过了，则目标代码段选择符就会加载进 CS 寄存器，于是程序的控制权就被转移到新代码段中，程序将从 EIP 寄存器指向的指令处开始执行。

程序的控制转移使用指令 JMP、RET、INT 和 IRET 以及异常和中断机制来实现。异常和中断是一些特殊实现，将在后面描述，本节主要说明 JMP、CALL 和 RET 指令的实现方法。JMP 或 CALL 指令可以利用一下四种方法之一来引用另外一个代码段：

- 目标操作数含有目标代码段的段选择符；
- 目标操作数指向一个调用门描述符，而该描述符中含有目标代码段的选择符；
- 目标操作数指向一个 TSS，而该 TSS 中含有目标代码段的选择符；
- 目标操作数指向一个任务门，该任务门指向一个 TSS，而该 TSS 中含有目标代码段的选择符；

下面描述前两种引用类型，后两种将放在有关任务管理一节中进行说明。

#### 4.5.3.1 直接调用或跳转到代码段

JMP、CALL 和 RET 指令的近转移形式只是在当前代码段中执行程序控制转移，因此不会执行特权级检查。JMP、CALL 或 RET 指令的远转移形式会把控制转移到另外一个代码段中，因此处理器一定会之醒特权级检查。

当不通过调用门把程序控制权转移到另一个代码段时，处理器会验证 4 种特权级和类型信息，见图 4-21



所示：

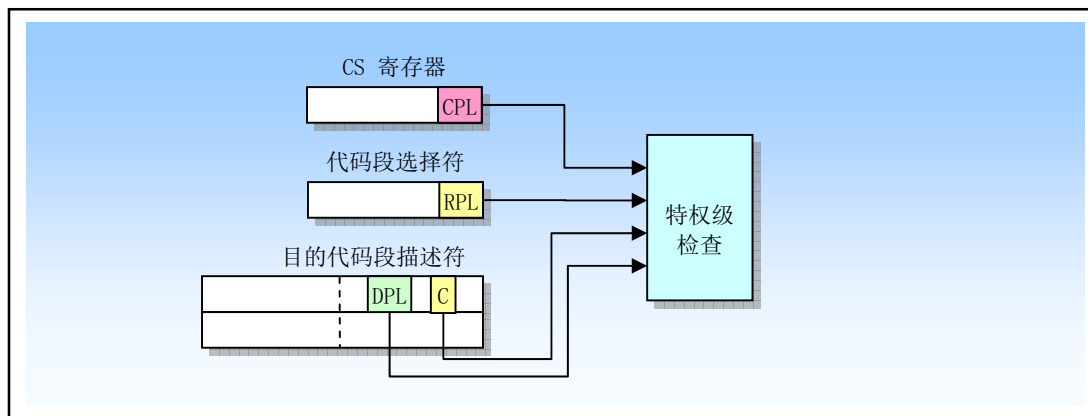


图 4-21 直接调用或跳转到代码段时的特权级检查

- 当前特权级 CPL。（这里，CPL 是执行调用的代码段的特权级，即含有执行调用或跳转程序的代码段的 CPL。）
- 含有被调用过程的目的代码段段描述符中的描述符特权级 DPL。
- 目的代码段的段选择符中的请求特权级 RPL。
- 目的代码段描述符中的一致性标志 C。它确定了一个代码段是非一致代码段还是一致代码段。

处理器检查 CPL、RPL 和 DPL 的规则依赖于标志 C 的设置状态。当访问非一致代码段时（C=0），调用者（程序）的 CPL 必须等于目的代码段的 DPL，否则将会产生一般保护异常。指向非一致代码段的段选择符的 RPL 对检查所起的作用有限。RPL 在数值上必须小于或等于调用者的 CPL 才能使得控制转移成功完成。当非一致代码段的段选择符被加载进 CS 寄存器中时，特权级字段不会改变，即它仍然是调用者的 CPL。即使段选择符的 RPL 与 CPL 不同，这也是正确的。

当访问一致代码段时（C=1），调用者的 CPL 可以在数值上大于或等于目的代码段的 DPL。仅当 CPL < DPL 时，处理器才会产生一般保护异常。对于访问一致代码段，处理器忽略对 RPL 的检查。对于一致代码段，DPL 表示调用者对代码段进行成功调用可以处于的最低数值特权级。

当程序控制被转移到一个一致代码段中，CPL 并不改变，即使目的代码段的 DPL 在数值上小于 CPL。这是 CPL 与可能与当前代码段 DPL 不相同的唯一一种情况。同样，由于 CPL 没有改变，因此堆栈也不会切换。

大多数代码段都是非一致代码段。对于这些段，程序的控制权只能转移到具有相同特权级的代码段中，除非转移是通过一个调用门进行，见下面说明。

#### 4.5.3.2 门描述符

为了对具有不同特权级的代码段提供受控的访问，处理器提供了称为门描述符的特殊描述符集。共有 4 种门描述符：

- 调用门（Call Gate），类型 TYPE=12；
- 陷阱门（Trap Gate），类型 TYPE=15；
- 中断门（Interrupt Gate），类型 TYPE=14；
- 任务门（Task Gate），类型 TYPE=5。

任务门用于任务切换，将在后面任务管理一节说明。陷阱门和中断门是调用门的特殊类，专门用于调用异常和中断的处理程序，这将在下一节进行说明。本节仅说明调用门的使用方法。

调用门用于在不同特权级之间实现受控的程序控制转移。它们通常仅用于使用特权级保护机制的操作

系统中。图 4-22 给出了调用门描述符的格式。调用门描述符可以存放在 GDT 或 LDT 中，但是不能放在中断描述符表 IDT 中。一个调用门主要具有以下几个功能：

- 指定要访问的代码段；
- 在指定代码段中定义过程（程序）的一个入口点；
- 指定访问过程的调用者需具备的特权级；
- 若会发生堆栈切换，它会指定在堆栈之间需要复制的可选参数个数；
- 指明调用门描述符是否有效。

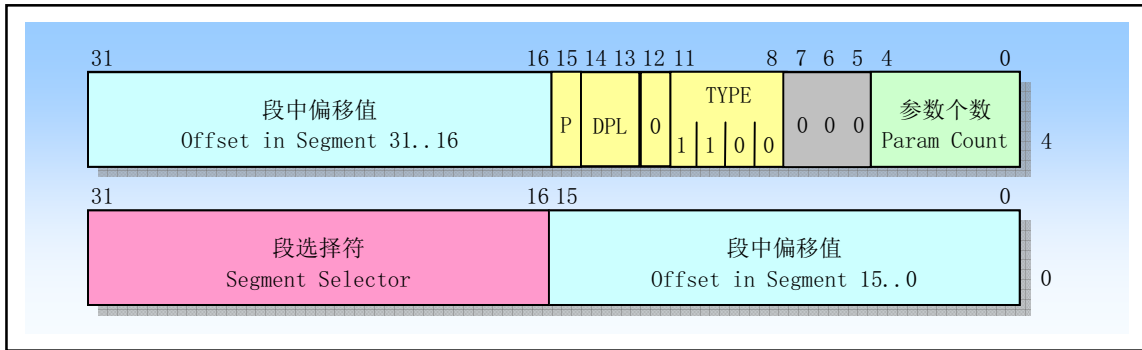


图 4-22 调用门描述符格式

调用门中的段选择符字段指定要访问的代码段。偏移值字段指定段中入口点。这个入口点通常是指定过程的第一条指令。DPL 字段指定调用门的特权级，从而指定通过调用门访问特定过程所要求的特权级。标志 P 指明调用门描述符是否有效。参数个数字段 (Param Count) 指明在发生堆栈切换时从调用者堆栈复制到新堆栈中的参数个数。Linux 内核中并没有用到调用门。这里对调用门进行说明是为下一节介绍利用中断和异常门进行处理作准备。

#### 4.5.3.3 通过调用门访问代码段

为了访问调用门，我们需要为 CALL 或 JMP 指令的操作数提供一个远指针。该指针中的段选择符用于指定调用门，而指针的偏移值虽然需要但 CPU 并不会用它。该偏移值可以设置为任意值。见图 4-23 所示。

当处理器访问调用门时，它会使用调用门中的段选择符来定位目的代码段的段描述符。然后 CPU 会把代码段描述符的基地址与调用门中的偏移值进行组合，形成代码段中指定程序入口点的线性地址。



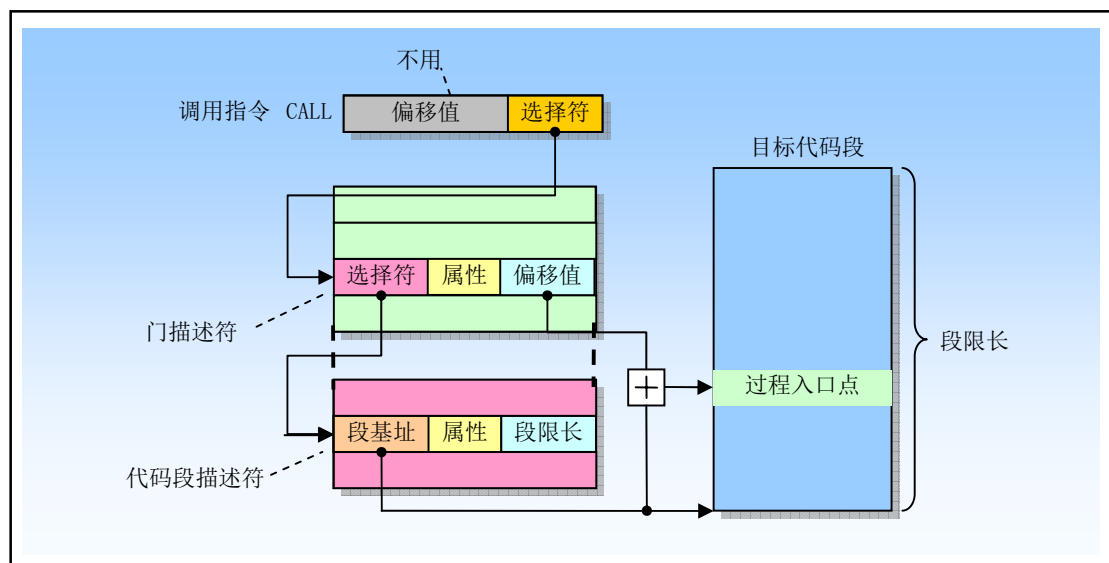


图 4-23 门调用操作过程

通过调用门进行程序控制转移时，CPU 会对 4 中不同的特权级进行检查，以确定控制转移的有效性，见图 4-24 所示。

- 当前特权级 CPL；
- 调用门选择符中的请求特权级 RPL；
- 调用门描述符中的描述符特权级 DPL；
- 目的代码段描述符中的 DPL；

另外，目的代码段描述符中的一致性标志 C 也将受到检查。

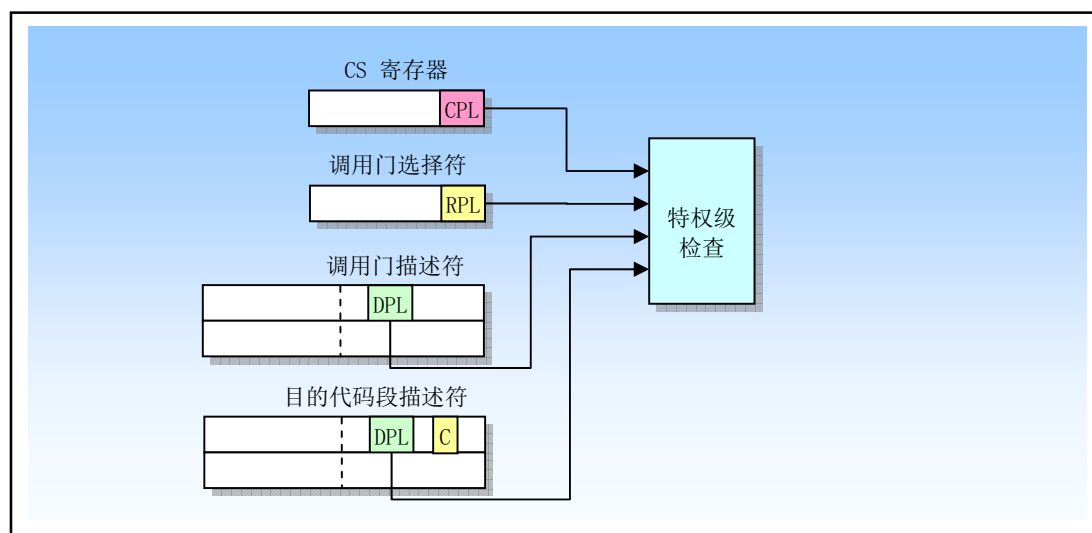


图 4-24 通过调用进行控制转移的特权级检查

使用 CALL 指令和 JMP 指令分别具有不同的特权级检测规则，见表 4-5 所示。调用门描述符的 DPL 字段指明了调用程序能够访问调用门的数值最大的特权级（最小特权级），即为了访问调用门，调用者程序的特权级 CPL 必须小于或等于调用门的 DPL。调用门段选择符的 RPL 也需同调用这的 CPL 遵守同样的规则，即 RPL 也必须小于或等于调用门的 DPL。

表 4-5 CALL 指令和 JMP 指令的特权级检查规则

指令	特权级检查规则
CALL	CPL≤调用门的 DPL; RPL≤调用门的 DPL。 对于一致性和非一致性代码段都只要求 DPL≤CPL
JMP	CPL≤调用门的 DPL; RPL≤调用门的 DPL。 对于一致性代码段要求 DPL≤CPL; 对于非一致性代码段只要求 DPL=CPL

如果调用这与调用门之间的特权级检查成功通过, CPU 就会接着把调用者的 CPL 与代码段描述符的 DPL 进行比较检查。在这方面, CALL 指令和 JMP 指令的检查规则就不同了。只有 CALL 指令可以通过调用门把程序控制转移到特权级更高的非一致性代码段中, 即可以转移到 DPL 小于 CPL 的非一致性代码段中去执行。而 JMP 指令只能通过调用门把控制转移到 DPL 等于 CPL 的非一致性代码段中。但 CALL 指令和 JMP 指令都可以把控制转移到更高特权级的一致性代码段中, 即转移到 DPL 小于或等于 CPL 的一致性代码段中。

如果一个调用把控制转移到了更高特权级的非一致性代码段中, 那么 CPL 就会被设置为目的代码段的 DPL 值, 并且会引起堆栈切换。但是如果一个调用或跳转把控制转移到更高级别的一致性代码段上, 那么 CPL 并不会改变, 并且也不会引起堆栈切换。

调用门可以让一个代码段中的过程被不同特权级的程序访问。例如, 位于一个代码段中的操作系统代码可能含有操作系统自身和应用软件都允许访问的代码(比如处理字符 I/O 的代码)。因此可以为这些过程设置一个所有特权级代码都能访问的调用门。另外可以专门为仅用于操作系统的代码设置一些更高特权级的调用门。

#### 4.5.3.4 堆栈切换

每当调用门用于把程序控制转移到一个更高级别的非一致性代码段时, CPU 会自动切换到目的代码段特权级的堆栈去。执行栈切换操作的目的是为了防止高特权级程序由于栈空间不足而引起崩溃, 同时也为了防止低特权级程序通过共享的堆栈有意或无意地干扰高特权级的程序。

每个任务必须定义最多 4 个栈。一个用于运行在特权级 3 的应用程序代码, 其他分别用于用到的特权级 2、1 和 0。如果一个系统中只使用了 3 和 0 两个特权级, 那么每个任务就只需设置两个栈。每个栈都位于不同的段中, 并且使用段选择符和段中偏移值指定。

当特权级 3 的程序在执行时, 特权级 3 的堆栈的段选择符和栈指针会被分别存放在 SS 和 ESP 中, 并且在发生堆栈切换时被保存在被调用过程的堆栈上。

特权级 0、1 和 2 的堆栈的初始指针值都存放在当前运行任务的 TSS 段中。TSS 段中这些指针都是只读值。在任务运行时 CPU 并不会修改它们。当调用更高特权级程序时, CPU 才用它们来建立新堆栈。当从调用过程返回时, 相应栈就不存在了。下一次再调用该过程时, 就会再次使用 TSS 中的初始指针值建立一个新栈。

操作系统需要负责为所有用到的特权级建立堆栈和堆栈段描述符, 并且在任务的 TSS 中设置初始指针值。每个栈必须可读可写, 并且具有足够的空间来存放以下一些信息:

- 调用过程的 SS、ESP、CS 和 EIP 寄存器内容;
- 被调用过程的参数和临时变量所需使用的空间。
- 当隐含调用一个异常或中断过程时标志寄存器 EFLAGS 和出错码使用的空间。

由于一个过程可调用其它过程, 因此每个栈必须有足够大的空间来容纳多帧(多套)上述信息。

当通过调用门执行一个过程调用而造成特权级改变时, CPU 就会执行以下步骤切换堆栈并开始在新的特权级上执行被调用过程(见图 4-25 所示):

1. 使用目的代码段的 DPL(即新的 CPL)从 TSS 中选择新栈的指针。从当前 TSS 中读取新栈的段选择符和栈指针。在读取栈段选择符、栈指针或栈段描述符过程中, 任何违反段界限的错误都将导

致产生一个无效 TSS 异常；

2. 检查栈段描述符特权级和类型是否有效，若无效者同样产生一个无效 TSS 异常。
3. 临时保存 SS 和 ESP 寄存器的当前值，把新栈的段选择符和栈指针加载到 SS 和 ESP 中。然后把临时保存的 SS 和 ESP 内容压入新栈中。
4. 把调用门描述符中指定参数个数的参数从调用过程栈复制到新栈中。调用门中参数个数最大为 31，如果个数为 0，则表示无参数，不需复制。
5. 把返回指令指针（即当前 CS 和 EIP 内容）压入新栈。把新（目的）代码段选择符加载到 CS 中，同时把调用门中偏移值（新指令指针）加载到 EIP 中。最后开始执行被调用过程。

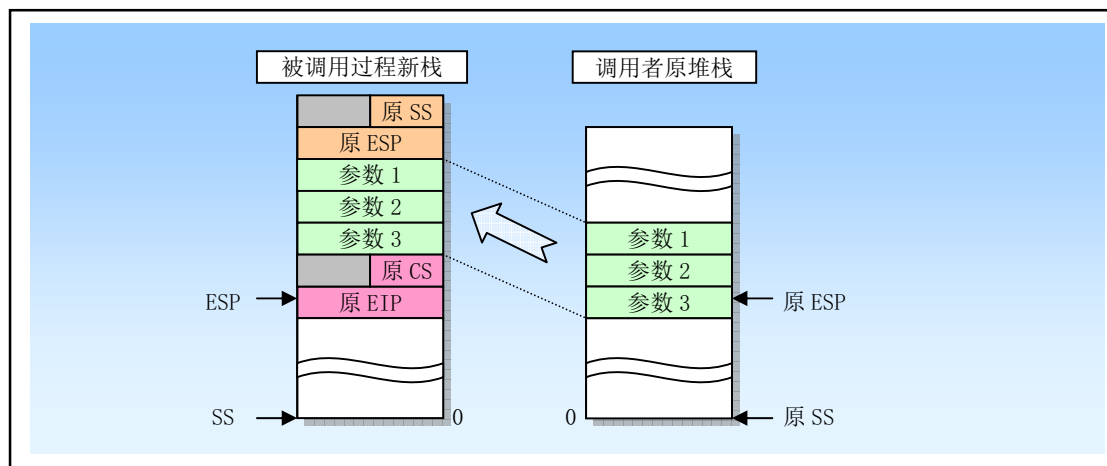


图 4-25 不同特权级之间调用时的栈切换

#### 4.5.3.5 从被调用过程返回

指令 RET 用于执行近返回（near return）、同特权级远返回（far return）和不同特权级的远返回。该指令用于从使用 CALL 指令调用的过程中返回。近返回仅在当前代码段中转移程序控制权，因此 CPU 仅进行界限检查。对于相同特权级的远返回，CPU 同时从堆栈中弹出返回代码段的选择符和返回指令指针。由于通常情况下这两个指针是 CALL 指令压入栈中的，因此它们因该是有效的。但是 CPU 还是会执行特权级检查以应付当前过程可能修改指针值或者堆栈出现问题时的情况。

会发生特权级改变的远返回仅允许返回到低特权级程序中，即返回到的代码段 DPL 在数值上要大于 CPL。CPU 会使用 CS 寄存器中选择符的 RPL 字段来确定是否要求返回到低特权级。如果 RPL 的数值要比 CPL 大，就会执行特权级之间的返回操作。当执行远返回到一个调用过程时，CPU 会执行以下步骤：

1. 检查保存的 CS 寄存器中 RPL 字段值，以确定在返回时特权级是否需要改变。
2. 弹出并使用被调用过程堆栈上的值加载 CS 和 EIP 寄存器。在此过程中会对代码段描述符和代码段选择符的 RPL 进行特权级与类型检查。
3. 如果 RET 指令包含一个参数个数操作数并且返回操作会改变特权级，那么就在弹出栈中 CS 和 EIP 值之后把参数个数值加到 ESP 寄存器值中，以跳过（丢弃）被调用者栈上的参数。此时 ESP 寄存器指向原来保存的调用者堆栈的指针 SS 和 ESP。
4. 把保存的 SS 和 ESP 值加载到 SS 和 ESP 寄存器中，从而切换回调用者的堆栈。而此时被调用者堆栈的 SS 和 ESP 值被抛弃。
5. 如果 RET 指令包含一个参数个数操作数，则把参数个数值加到 ESP 寄存器值中，以跳过（丢弃）调用者栈上的参数。
6. 检查段寄存器 DS、ES、FS 和 GS 的内容。如果其中有指向 DPL 小于新 CPL 的段（一致代码段除外），那么 CPU 就会用 NULL 选择符加载加载这个段寄存器。

### 4.5.4 页级保护

页目录和页表表项中的读写标志 R/W 和用户/超级用户标志 U/S 提供了分段机制保护属性的一个子集。分页机制只识别两级权限。特权级 0、1 和 2 被归类为超级用户级，而特权级 3 被作为普通用户级。普通用户级的页面可以被标志成只读/可执行或可读/可写/可执行。超级用户级的页面对于超级用户来讲总是可读/可写/可执行的，但普通用户不可访问，见表 4-6 所示。对于分段机制，在最外层用户级执行的程序只能访问用户级的页面，但是在任何超级用户层（0、1、2）执行的程序不仅可以访问用户层的页面，也可以访问超级用户层的页面。与分段机制不同的是，在内层超级用户级执行的程序对任何页面都具有可读/可写/可执行权限，包括那些在用户级标注为只读/可执行的页面。

表 4-6 普通用户和超级用户对页面的访问限制

U/S	R/W	用户允许的访问	超级用户允许的访问
0	0	无	读/写/执行
0	1	无	读/写/执行
1	0	读/执行	读/写/执行
1	1	读/写/执行	读/写/执行

正如在整个 80X86 地址转换机制中分页机制是在分段机制之后实施一样，页级保护也是在分段机制提供的保护措施之后发挥作用。首先，所有段级保护被检查和测试。如果通过检查，就会再进行页级保护检查。例如，仅当一个字节位于级别 3 执行的程序可访问的段中，并且处于标志为用户级页面中时，这个内存中的字节才可被级别 3 上的程序访问。仅当分段和分页都允许写时，才能对页面执行写操作。如果一个段是读/写类型的段，但是地址对应的相应页面被标注为只读/可执行，那么还是不能对页面执行写操作。如果段的类型是只读/可执行，那么不管对应页面被赋予何保护属性，页面始终是没有写权限的。可见分段和分页的保护机制就像电子线路中的串行线路，其中那个开关没有合上线路都不会通。

类似地，一个页面的保护属性由页目录和页表中表项的“串行”或“与操作”构成，见表 4-7 所示。页表表项中的 U/S 标志和 R/W 标志应用于该表项映射的单个页面。页目录项中的 U/S 和 R/W 标志则对该目录项所映射的所有页面起作用。页目录和页表的组合保护属性由两者属性的“与”（AND）操作构成，因此保护措施非常严格。

表 4-7 页目录项和页表项对页面的“串行”保护

页目录项 U/S	页表项 U/S	组合的 U/S	页目录项 R/W	页表项 R/W	组合的 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 修改页表项的软件问题

本节提供一些有关操作系统软件修改页表项内容所需遵守的规则。分页转换缓冲要求所有系统都须遵守这些规则。为了避免每次内存应用都要访问驻留内存的页表，从而加快速度，最近使用的线性到物理地址的转换信息被保存在处理器内的页转换高速缓冲中。处理器在访问内存中的页表之前会首先利用缓冲中的信息。只有当必要的转换信息不在高速缓冲中时，处理器才会搜寻内存中的页目录和页表。页转换高速缓冲作用类似于前面描述的加速段转换的段寄存器的影子描述符寄存器。页转换高速缓冲的另一个术语称为转换查找缓冲 TLB（Translation Lookaside Buffer）。

80X86 处理器并没有维护页转换高速缓冲和页表中数据的相关性，但是需要操作系统软件来确保它们

一致。即处理器并不知道什么时候页表被软件修改过了。因此操作系统必须在改动过页表之后刷新高速缓冲以确保两者一致。通过简单地重新加载寄存器 CR3，我们就可以完成对高速缓冲的刷新操作。

有一种特殊情况，在这种情况下修改页表项不需要刷新页转换高速缓冲。也即当不存在页面的表项被修改时，即使是把 P 标志从 0 改成 1 来标注表项对页转换有效，也不需要刷新高速缓冲。因为无效的表项不会被存入高速缓冲中。所以在把一个页面从磁盘调入内存以使页面存在时，我们不需要刷新页转换高速缓冲。

4.5.5 组合页级和段级保护

当启用了分页机制，CPU 会首先执行段级保护，然后再处理页级保护。如果 CPU 在任何一级检测到一个保护违规错误，则会放弃内存访问并产生一个异常。如果是段机制产生的异常，那么就不会再产生一个页异常。

页级保护不能替代或忽略段级保护。例如，若一个代码段被设定为不可写，那么代码段被分页后，即使页面的 R/W 标志被设置成可读可写也不会让页面可写。此时段级保护检查会阻止任何对页面的写操作企图。页级保护可被用来增强段级保护。例如，如果一个可读可写数据段被分页，那么页级保护机制可用来对个别页面进行写保护。

4.6 中断和异常处理

中断（Interrupt）和异常（Exception）是指明系统、处理器或当前执行程序（或任务）的某处出现一个事件，该事件需要处理器进行处理。通常，这种事件会导致执行控制被强迫从当前运行程序转移到被称为中断处理程序（interrupt handler）或异常处理程序（exception handler）的特殊软件函数或任务中。处理器响应中断或异常所采取的行动被称为中断/异常服务（处理）。

通常，中断发生在程序执行的随机时刻，以响应硬件发出的信号。系统硬件使用中断来处理外部事件，例如要求为外部设备提供服务。当然，软件也能通过执行 INT n 指令产生中断。

异常发生在处理器执行一条指令时，检测到一个出错条件时发生，例如被 0 除出错条件。处理器可以检测到各种出错条件，包括违反保护机制、页错误以及机器内部错误。

对应用程序和操作系统来说，80X86 的中断和异常处理机制可以透明地处理发生的中断和异常事件。当收到一个中断或检测到一个异常时，处理器会自动地把当前正在执行的程序或任务挂起，并开始运行中断或异常处理程序。当处理程序执行完毕，处理器就会恢复并继续执行被中断的程序或任务。被中断程序的恢复过程并不会失去程序执行的连贯性，除非从异常中恢复是不可能的或者中断导致当前运行程序被终止。本节描述保护模式中处理器中断和异常的处理机制。

4.6.1 异常和中断向量

为了有助于处理异常和中断，每个需要被处理器进行特殊处理的处理器定义的异常和中断条件都被赋予了一个标识号，称为向量（vector）。处理器把赋予异常或中断的向量用作中断描述符表 IDT（Interrupt Descriptor Table）中的一个索引号，来定位一个异常或中断的处理程序入口点位置。

允许的向量号范围是 0 到 255。其中 0 到 31 保留用作 80X86 处理器定义的异常和中断，不过目前该范围内的向量号并非每个都已定义了功能，未定义功能的向量号将留作今后使用。

范围在 32 到 255 的向量号用于用户定义的中断。这些中断通常用于外部 I/O 设备，使得这些设备可以通过外部硬件中断机制向处理器发送中断。表 4-8 中给出了为 80X86 定义的异常和 NMI 中断分配的向量。对于每个异常，该表给出了异常类型以及是否会产生一个错误码并保存在堆栈上。同时还给出了每个预先定义好的异常和 NMI 中断源。

表 4-8 保护模式下的异常和中断

向量号	助记符	说明	类型	错误号	产生源
-----	-----	----	----	-----	-----

0	#DE	除出错	故障	无	DIV 或 IDIV 指令。
1	#DB	调试	故障/陷阱	无	任何代码或数据引用, 或是 INT 1 指令。
2	--	NMI 中断	中断	无	非屏蔽外部中断。
3	#BP	断点	陷阱	无	INT 3 指令。
4	#OF	溢出	陷阱	无	INTO 指令。
5	#BR	边界范围超出	故障	无	BOUND 指令。
6	#UD	无效操作码 (未定义操作码)	故障	无	UD2 指令或保留的操作码。(奔腾 Pro 中加入的新指令)
7	#NM	设备不存在 (无数学协处理器)	故障	无	浮点或 WAIT/FWAIT 指令。
8	#DF	双重错误	异常终止	有 (0)	任何可产生异常、NMI 或 INTR 的指令。
9	--	协处理器段超越 (保留)	故障	无	浮点指令 (386 以后的 CPU 不产生该异常)。
10	#TS	无效的任务状态段 TSS	故障	有	任务交换或访问 TSS。
11	#NP	段不存在	故障	有	加载段寄存器或访问系统段。
12	#SS	堆栈段错误	故障	有	堆栈操作和 SS 寄存器加载。
13	#GP	一般保护错误	故障	有	任何内存引用和其他保护检查。
14	#PF	页面错误	故障	有	任何内存引用。
15	--	(Intel 保留, 请勿使用)		无	
16	#MF	x87 FPU 浮点错误 (数学错误)	故障	无	x87 FPU 浮点或 WAIT/FWAIT 指令。
17	#AC	对起检查	故障	有 (0)	对内存中任何数据的引用。
18	#MC	机器检查	异常终止	无	错误码 (若有) 和产生源与 CPU 类型有关 (奔腾处理器引进)。
19	#XF	SIMD 浮点异常	故障	无	SSE 和 SSE2 浮点指令 (PIII 处理器引进)。
20-31	--	(Intel 保留, 请勿使用)			
32-255	--	用户定义 (非保留) 中断	中断		外部中断或者 INT n 指令。

## 4.6.2 中断源和异常源

### 4.6.2.1 中断源

处理器从两种地方接收中断:

- 外部 (硬件产生) 的中断;
- 软件产生的中断。

外部中断通过处理器芯片上两个引脚 (INTR 和 NMI) 接收。当引脚 INTR 接收到外部发生的中断信号时, 处理器就会从系统总线上读取外部中断控制器 (例如 8259A) 提供的中断向量号。当引脚 NMI 接收到信号时, 就产生一个非屏蔽中断。它使用固定的中断向量号 2。任何通过处理器 INTR 引脚接收的外部中断都被称为可屏蔽硬件中断, 包括中断向量号 0 到 255。标志寄存器 EFLAGS 中的 IF 标志可用于屏蔽所有这些硬件中断。

通过在指令操作数中提供中断向量号, INT n 指令可用于从软件中产生中断。例如, 指令 INT 0x80 会执行 Linux 的系统中断调用中断 0x80。向量 0 到 255 中的任何一个都可以用作 INT 指令的中断号。然而, 如果使用了处理器预先定义的 NMI 向量, 那么处理器对它的响应将与普通方式产生的该 NMI 中断不同。如果 NMI 的向量号 2 用于该 INT 指令, 就会调用 NMI 的中断处理器程序, 但是此时并不会激活处理器的 NMI 处理硬件。

注意，EFLAGS 中的 IF 标志不能屏蔽使用 INT 指令从软件中产生的中断。

### 4.6.2.2 异常源

处理器接收的异常也有两个来源：

- 处理器检测到的程序错误异常；
- 软件产生的异常。

在应用程序或操作系统执行期间，如果处理器检测到程序错误，就会产生一个或多个异常。80X86 处理器为其检测到的每个异常定义了一个向量。异常可以被细分为故障 (faults)、陷阱 (traps) 和中止 (aborts)，见后面说明。

指令 INTO、INT 3 和 BOUND 指令可以用来从软件中产生异常。这些指令可对指令流中指定点执行的特殊异常条件进行检查。例如，INT 3 指令会产生一个断点异常。

INT n 指令可用于在软件中模拟指定的异常，但有一个限制。如果 INT 指令中的操作数 n 是 80X86 异常的向量号之一，那么处理器将为该向量号产生一个中断，该中断就会去执行与该向量有关的异常处理程序。但是，因为这实际上是一个中断，因此处理器并不会把一个错误号压入堆栈，即使硬件产生的该向量相关的中断通常会产生一个错误码。对于那些会产生错误码的异常，异常的处理程序会试图从堆栈上弹出错误码。因此，如果使用 INT 指令来模拟产生一个异常，处理程序则会把 EIP（正好处于缺少的错误码位置处）弹出堆栈，从而会造成返回位置错误。

### 4.6.3 异常分类

根据异常被报告的方式以及导致异常的指令是否能够被重新执行，异常可被细分成故障 (Fault)、陷阱 (Trap) 和中止 (Abort)。

- Fault 是一种通常可以被纠正的异常，并且一旦被纠正程序就可以继续运行。当出现一个 Fault，处理器会把机器状态恢复到产生 Fault 的指令之前的状态。此时异常处理程序的返回地址会指向产生 Fault 的指令，而不是其后面一条指令。因此在返回后产生 Fault 的指令将被重新执行。
- Trap 是一个引起陷阱的指令被执行后立刻会报告的异常。Trap 也能够让程序或任务连贯地执行。Trap 处理程序的返回地址指向引起陷阱指令的随后一条指令，因此在返回后会执行下一条指令。
- Abort 是一种不会总是报告导致异常的指令的精确位置的异常，并且不允许导致异常的程序重新继续执行。Abort 用于报告严重错误，例如硬件错误以及系统表中存在不一致性或非法值。

### 4.6.4 程序或任务的重新执行

为了让程序或任务在一个异常或中断处理完之后能重新恢复执行，除了中止 (Abort) 之外的所有异常都能报告精确的指令位置，并且所有中断保证是在指令边界上发生。

对于故障类异常，处理器产生异常时保存的返回指针指向出错指令。因为，当程序或任务在故障处理程序返回后重新开始执行时，原出错指令会被重新执行。重新执行引发出错的指令通常用于处理访问指令操作数受阻的情况。Fault 最常见的一个例子是页面故障 (Page-fault) 异常。当程序引用不在内存中页面上的一个操作数时就会出现这种异常。当页故障异常发生时，异常处理程序可以把该页面加载到内存中并通过重新执行出错指令来恢复程序执行。为了确保重新执行对于当前执行程序具有透明性，处理器会保存必要的寄存器和堆栈指针信息，以使得自己能够返回到执行出错指令之前的状态。

对于陷阱 Trap 类异常，处理器产生异常时保存的返回指针指向引起陷阱操作的后一条指令。如果在一条执行控制转移的指令执行期间检测到一个 Trap，则返回指令指针会反映出控制的转移情况。例如，如果在执行 JMP 指令时检测到一个 Trap 异常，那么返回指令指针会指向 JMP 指令的目标位置，而非指向 JMP 指令随后的一条指令。

中止 Abort 类异常不支持可靠地重新执行程序或任务。中止异常的处理程序通常用来收集异常发生时有关处理器状态的诊断信息，并且尽可能恰当地关闭程序和系统。

中断会严格地支持被中断程序的重新执行而不会丢失任何连贯性。中断所保存的返回指令指针指向处

处理器获取中断时将要执行的下一条指令边界处。如果刚执行的指令有一个重复前缀，则中断会在当前重复结束并且寄存器已为下一次重复操作设置好时发生。

### 4.6.5 开启和禁止中断

标志寄存器 EFLAGS 的中断允许标志 IF (Interrupt enable Flag) 能够禁止为处理器 INTR 引脚上收到的可屏蔽硬件中断提供服务。当 IF=0 时，处理器禁止发送到 INTR 引脚的中断；当 IF=1 时，则发送到 INTR 引脚的中断信号会被处理器进行处理。

IF 标志并不影响发送到 NMI 引脚的非屏蔽中断，也不影响处理器产生的异常。如同 EFLAGS 中的其他标志一样，处理器在响应硬件复位操作时会清除 IF 标志 (IF=0)。

IF 标志可以使用指令 STI 和 CLI 来设置或清除。只有当程序的 CPL≤IOPL 时才可执行这两条指令，否则将引发一般保护性异常。IF 标志也会受一下操作影响：

- PUSHF 指令会把 EFLAGS 内容存入堆栈中，并且可以在那里被修改。而 POPF 指令可用于把已被修改过的标志内容放入 EFLAGS 寄存器中。
- 任务切换、POPF 和 IRET 指令会加载 EFLAGS 寄存器。因此，它们可用来修改 IF 标志。
- 当通过中断门处理一个中断时，IF 标志会被自动清除（复位），从而会禁止可屏蔽硬件中断。但如果通过陷阱门来处理一个中断，则 IF 标志不会被复位。

### 4.6.6 异常和中断的优先级

如果在一条指令边界有多个异常或中断等待处理时，处理器会按规定的次序对它们进行处理。表 4-9 给出了异常和中断源类的优先级。处理器会首先处理最高优先级类中的异常或中断。低优先级的异常会被丢弃，而低优先级的中断则会保持等待。当中断处理程序返回到产生异常和/或中断的程序或任务时，被丢弃的异常会重新发生。

表 4-9 异常和中断的优先级

优先级	说明
1 (最高)	硬件复位：RESET
2	任务切换陷阱：TSS 中设置了 T 标志
3	外部硬件介入
4	前一指令陷阱：断点、调试陷阱异常
5	外部中断：NMI 中断、可屏蔽硬件中断
6	代码断点错误
7	取下一条指令错误：违反代码段限长、代码页错误
8	下一条指令译码错误：指令长度>15 字节、无效操作码、协处理器不存在
9 (最低)	执行指令错误：溢出、边界检查、无效 TSS、段不存在、堆栈错、一般保护、数据页、对齐检查、浮点异常

### 4.6.7 中断描述符表

中断描述符表 IDT (Interrupt Descriptor Table) 将每个异常或中断向量分别与它们的处理过程联系起来。与 GDT 和 LDT 表类似，IDT 也是由 8 字节长描述符组成的一个数组。与 GDT 不同的是，表中第 1 项可以包含描述符。为了构成 IDT 表中的一个索引值，处理器把异常或中断的向量号\*8。因为最多只有 256 个中断或异常向量，所以 IDT 无需包含多于 256 个描述符。IDT 中可以含有少于 256 个描述符，因为只有可能发生的异常或中断才需要描述符。不过 IDT 中所有空描述符项应该设置其存在位（标志）为 0。

IDT 表可以驻留在线性地址空间的任何地方，处理器使用 IDTR 寄存器来定位 IDT 表的位置。这个寄



寄存器中含有 IDT 表 32 位的基地址和 16 位的长度（限长）值，见图 4-26 所示。IDT 表基地址应该对齐在 8 字节边界上以提高处理器的访问效率。限长值是以字节为单位的 IDT 表的长度。

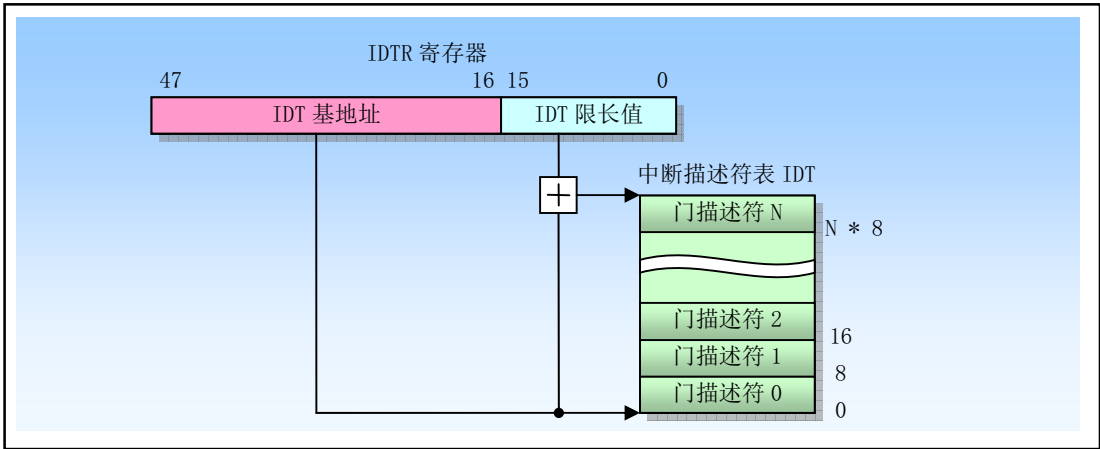


图 4-26 中断描述符表 IDT 和寄存器 IDTR

指令 LIDT 和 SIDT 指令分别用于加载和保存 IDTR 寄存器的内容。LIDT 指令把在内存中的限长值和基地址操作数加载到 IDTR 寄存器中。该指令仅能由当前特权级 CPL 是 0 的代码执行，通常被用于创建 IDT 时的操作系统初始化代码中。SIDT 指令用于把 IDTR 中的基地址和限长内容复制到内存中。该指令可在任何特权级上执行。

如果中断或异常向量引用的描述符超过了 IDT 的界限，处理器会产生一个一般保护性异常。

### 4.6.8 IDT 描述符

IDT 表中可以存放三种类型的门描述符：

- 中断门（Interrupt gate）描述符
- 陷阱门（Trap gate）描述符
- 任务门（Task gate）描述符

图 4-27 给出了这三种门描述符的格式。中断门和陷阱门含有一个长指针（即段选择符和偏移值），处理器使用这个长指针把程序执行权转移到代码段中异常或中断的处理过程中。这两个段的主要区别在于处理器操作 EFLAGS 寄存器 IF 标志上。IDT 中任务门描述符的格式与 GDT 和 LDT 中任务门的格式相同。任务门描述符中含有一个任务 TSS 段的选择符，该任务用于处理异常和/或中断。

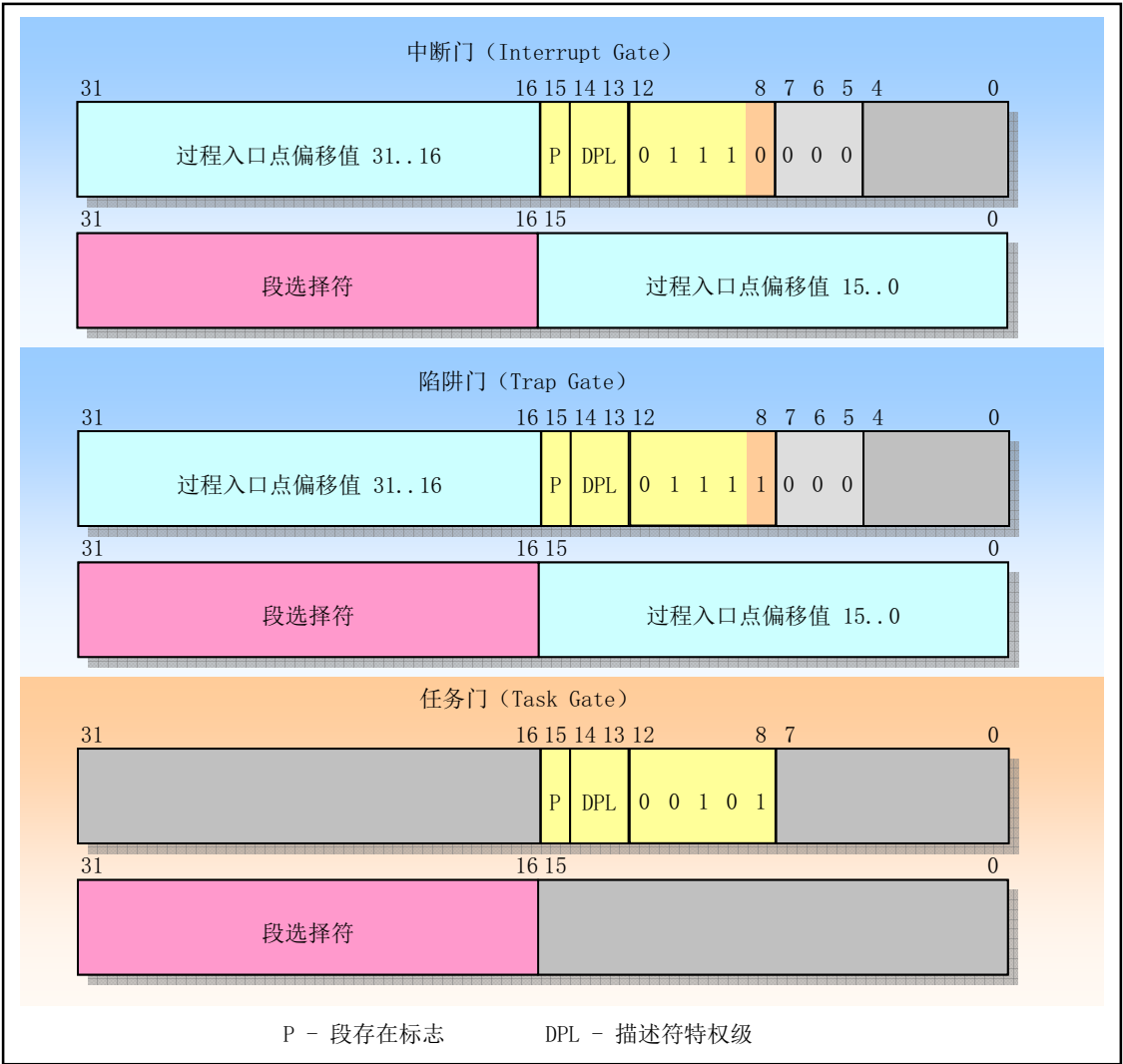


图 4-27 中断门、陷阱门和任务门描述符格式

4.6.9 异常与中断处理

处理器对异常和中断处理过程的调用操作方法与使用 CALL 指令调用程序过程和任务的方法类似。当响应一个异常或中断时，处理器使用异常或中断的向量作为 IDT 表中的索引。如果索引值指向中断门或陷阱门，则处理器使用与 CALL 指令操作调用门类似的方法调用异常或中断处理过程。如果索引值指向任务门，则处理器使用与 CALL 指令操作任务门类似的方法进行任务切换，执行异常或中断的处理任务。

异常或中断门引用运行在当前任务上下文中的异常或中断处理过程，见图 4-28 所示。门中的段选择符指向 GDT 或当前 LDT 中的可执行代码段描述符。门描述符中的偏移字段指向异常或中断处理过程的开始处。

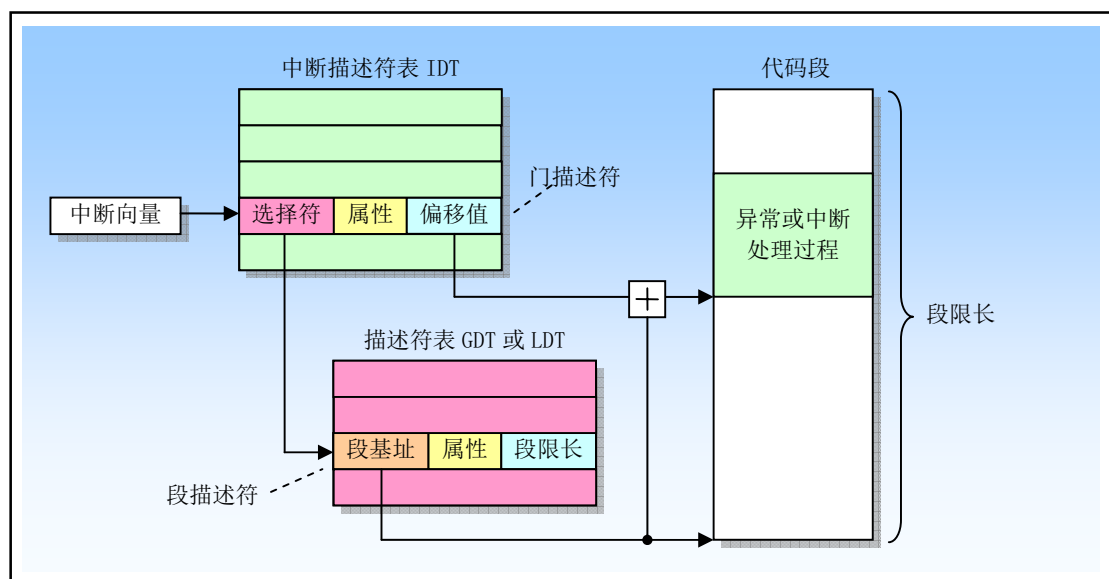


图 4-28 中断过程调用

当处理器执行异常或中断处理过程调用时会进行以下操作：

- 如果处理过程将在高特权级（例如 0 级）上执行时就会发生堆栈切换操作。堆栈切换过程如下：
  - ◆ 处理器从当前执行任务的 TSS 段中得到中断或异常处理过程使用的堆栈的段选择符和栈指针（例如 `tss.ss0`、`tss.esp0`）。然后处理器会把被中断程序（或任务）的栈选择符和栈指针压入新栈中，见图 4-29 所示。
  - ◆ 接着处理器会把 EFLAGS、CS 和 EIP 寄存器的当前值也压入新栈中。
  - ◆ 如果异常会产生一个错误号，那么该错误号也会被最后压入新栈中。
- 如果处理过程将在被中断任务同一个特权级上运行，那么：
  - ◆ 处理器把 EFLAGS、CS 和 EIP 寄存器的当前值保存在当前堆栈上。
  - ◆ 如果异常会产生一个错误号，那么该错误号也会被最后压入新栈中。

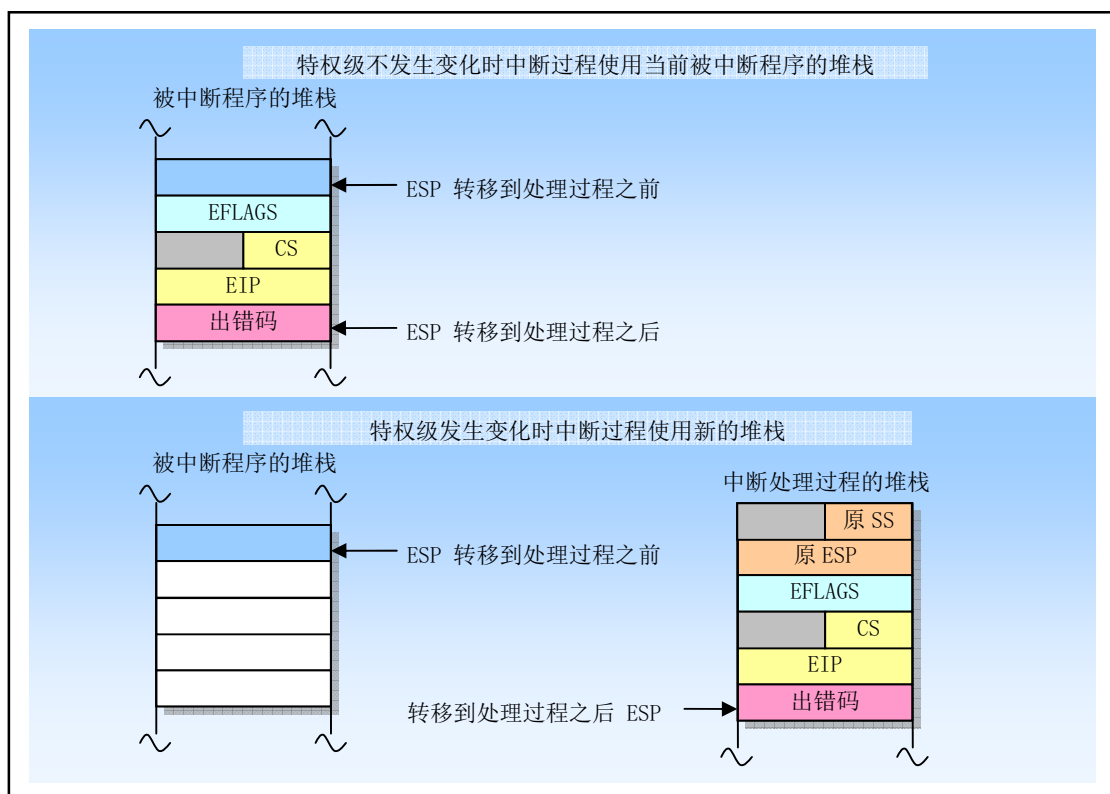


图 4-29 转移到中断处理过程时堆栈使用方法

为了从中断处理过程中返回，处理过程必须使用 IRET 指令。IRET 指令与 RET 指令类似，但 IRET 还会把保存的寄存器内容恢复到 EFLAGS 中。不过只有当 CPL 是 0 时才会恢复 EFLAGS 中的 IOPL 字段，并且只有当  $CPL \leq IOPL$  时，IF 标志才会被改变。如果当调用中断处理过程时发生了堆栈切换，那么在返回时 IRET 指令会切换回到原来的堆栈。

#### 1. 异常和中断处理过程的保护

异常和中断处理过程的特权级保护机制与通过调用门调用的普通过程类似。处理器不允许把控制转移到比 CPL 更低特权级代码段的中断处理过程中，否则将产生一个一般保护性异常。另外，中断和异常的保护机制在以下方面与一般调用门过程不同：

- 因为中断和异常向量没有 RPL，因此在隐式调用异常和中断处理过程时不会检查 RPL。
- 只有当一个异常或中断是利用使用  $INT n$ 、 $INT 3$  或  $INTO$  指令产生时，处理器才会检查中断或陷阱门中的 DPL。此时 CPL 必须小于等于门的 DPL。这个限制可以防止运行在特权级 3 的应用程序使用软件中断访问重要的异常处理过程，例如页错误处理过程，假设这些处理过程已被存放在更高特权级的代码段中。对于硬件产生的中断和处理器检测到的异常，处理器会忽略中断门和陷阱门中的 DPL。

因为异常和中断通常不会定期发生，因此这些有关特权级的规则有效地增强了异常和中断处理过程能够运行的特权级限制。我们可以利用以下技术之一来避免违反特权级保护：

- 异常或中断处理程序可以存放在一个一致性代码段中。这个技术可以用于只需访问堆栈上数据的处理过程（例如，除出错异常）。如果处理程序需要数据段中的数据，那么特权级 3 必须能够访问这个数据段。但这样一来就没有保护可言了。
- 处理过程可以放在具有特权级 0 的非一致代码段中。这种处理过程总是可以执行的，而不管被中断程序或任务的当前特权级 CPL。

#### 2. 异常或中断处理过程的标志使用方式

当通过中断门或陷阱门访问一个异常或中断处理过程时，处理器会在把 EFLAGS 寄存器内容保存到堆栈上之后清除 EFLAGS 中的 TF 标志。清除 TF 标志可以防止指令跟踪影响中断响应。而随后的 IRET 指令会用堆栈上的内容恢复 EFLAGS 的原 TF 标志。

中断门与陷阱门唯一的区别在于处理器操作 EFLAGS 寄存器 IF 标志的方法。当通过中断门访问一个异常或中断处理过程时，处理器会复位 IF 标志以防止其他中断干扰当前中断处理过程。随后的 IRET 指令则会用保存在堆栈上的内容恢复 EFLAGS 寄存器的 IF 标志。而通过陷阱门访问处理过程并不会影响 IF 标志。

3. 执行中断处理过程的任务

当通过 IDT 表中任务门访问异常或中断处理过程时，就会导致任务切换。从而可以在一个专用任务中执行中断或异常处理过程。IDT 表中的任务门引用 GDT 中的 TSS 描述符。切换到处理过程任务的方法与普通任务切换一样。由于本书讨论的 Linux 操作系统没有使用这种中断处理方式，因此这里不再赘述。

4.6.10 中断处理任务

当通过 IDT 中任务门来访问异常或中断处理过程时就会导致任务切换。使用单独的任务来处理异常或中断有如下好处：

- 被中断程序或任务的完整上下文会被自动保存；
- 在处理异常或中断时，新的 TSS 可以允许处理过程使用新特权级 0 的堆栈。在当前特权级 0 的堆栈已毁坏时如果发生了一个异常或中断，那么在为中断过程提供一个新特权级 0 的堆栈条件下，通过任务门访问中断处理过程能够防止系统崩溃；
- 通过使用单独的 LDT 给中断或异常处理任务独立的地址空间，可以把它与其他任务隔离开来。

使用独立任务处理异常或中断的不足之处是：在任务切换时必须对大量机器状态进行保存，使得它比使用中断门的响应速度要慢，导致中断延时增加。

IDT 中的任务门会引用 GDT 中的 TSS 描述符，图 4-30 所示。切换到句柄任务的过程与普通任务切换过程相同。到被中断任务的反向链接会被保存在句柄任务 TSS 的前一任务链接字段中。如果一个异常会产生一个出错码，则该出错码会被复制到新任务堆栈上。

当异常或中断句柄任务用于操作系统中时，实际上有两种分派调度任务的机制：操作系统软件调度和处理器中断机制的硬件调度。使用软件调度方法时需要考虑中断开启时采用中断处理任务。

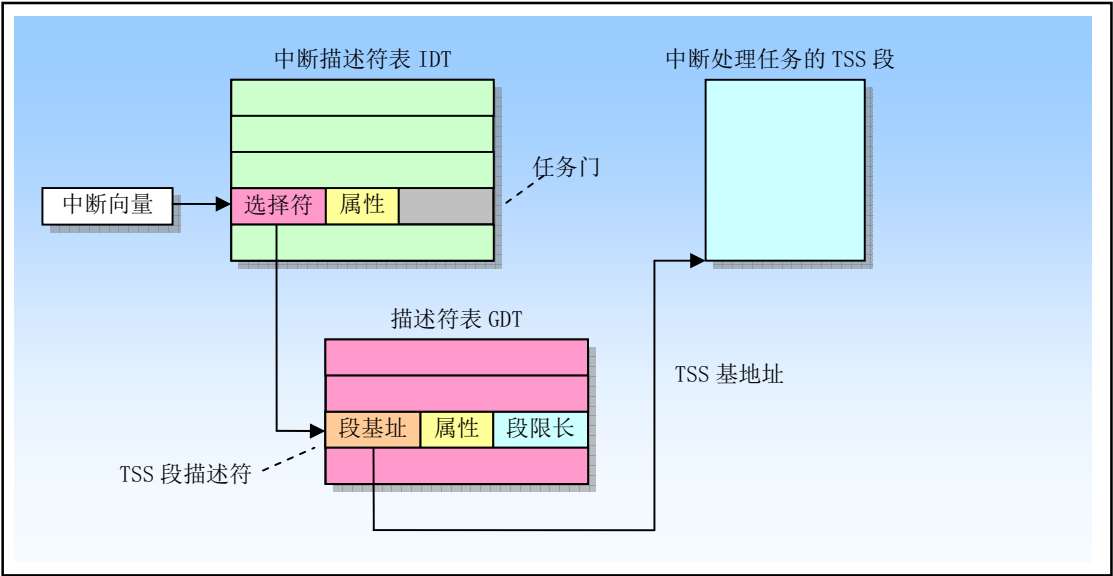


图 4-30 中断处理任务切换

4.6.11 错误码

当异常条件与一个特定的段相关时，处理器会把一个错误码压入异常处理过程的堆栈上。出错码的格式见图 4-31 所示。错误码很象一个段选择符，但是最低 3 比特不是 TI 和 RPL 字段，而是以下 3 个标志：

- 位 0 是外部事件 EXT（External event）标志。当置位时，表示执行程序以外的事件造成了异常，例如硬件中断。
- 位 1 是描述符位置 IDT（Descriptor location）标志。当该位置位时，表示错误码的索引部分指向 IDT 中的一个门描述符。当该位复位时，表示索引部分指向 GDT 或 LDT 中的一个段描述符。
- 位 2 是 GDT/LDT 表选择标志 TI。只有当位 1 的 IDT=0 才有用。当该 TI=1 时，表示错误码的索引部分指向 LDT 中的一个描述符。当 TI=0 时，说明错误码中的索引部分指向 GDT 表中的一个描述符。

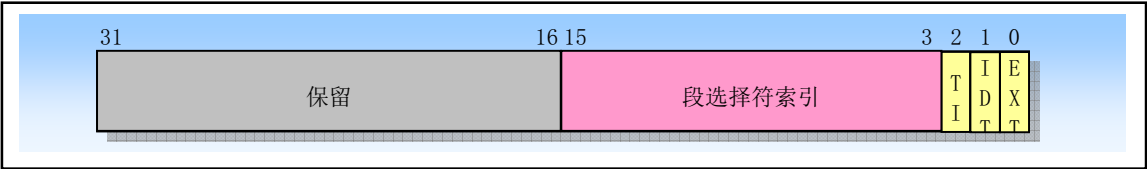


图 4-31 错误码格式

段选择索引字段提供了错误码引用的 IDT、GDT 或者当前 LDT 中段或门描述符的索引值。在某些情况下错误码是空的（即低 16 位全 0）。空错误码表示错误不是由于引用某个特定段造成，或者是在操作中引用了一个空段描述符。

页故障（Page-fault）异常的错误码格式与上面的不同，见图 4-32 所示。只有最低 3 个比特位有用，它们的名称与页表项中的最后三位相同（U/S、W/R、P）。含义和作用分别是：

- 位 0（P），异常是由于页面不存在或违反访问特权而引发。P=0，表示页不存在；P=1 表示违反页级保护权限。
- 位 1（W/R），异常是由于内存读或写操作引起。W/R=0，表示由读操作引起；W/R=1，表示由写操作引起。
- 位 2（U/S），发生异常时 CPU 执行的代码级别。U/S=0，表示 CPU 正在执行超级用户代码；U/S=1，表示 CPU 正在执行一般用户代码。

另外，处理器还会把引起页面故障异常所访问用的线性地址存放在 CR2 中。页出错异常处理程序可以使用这个地址来定位相关的页目录和页表项。

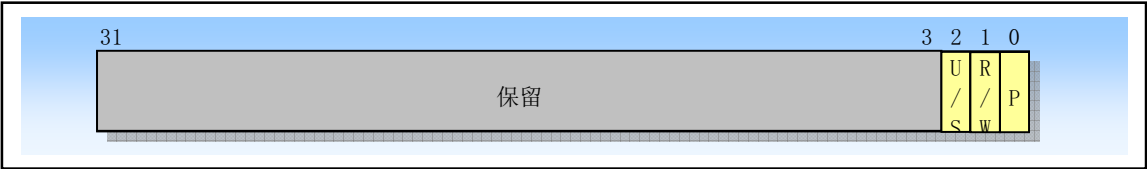


图 4-32 页面故障错误码格式

注意，错误不会被 IRET 指令自动地弹出堆栈，因此中断处理程序在返回之前必须清除堆栈上的错误

码。另外，虽然处理产生的某些异常会产生错误码并会自动地保存到处理过程的堆栈中，但是外部硬件中断或者程序执行 `INT n` 指令产生的异常并不会把错误码压入堆栈中。

## 4.7 任务管理

任务（Task）是处理器可以分配调度、执行和挂起的一个工作单元。它可用于执行程序、任务或进程、操作系统服务、中断或异常处理过程和内核代码。

80X86 提供了一种机制，这种机制可用来保存任务的状态、分派任务执行以及从一个任务切换到另一个任务。当工作在保护模式下，处理器所有运行都在任务中。即使是简单系统也必须起码定义一个任务。更为复杂的系统可以使用处理器的任务管理功能来支持多任务应用。

80X86 提供了多任务的硬件支持。任务是一个正在运行的程序，或者是一个等待准备运行的程序。通过中断、异常、跳转或调用，我们可以执行一个任务。当这些控制转移形式之一和某个描述符表中指定项的内容一起使用时，那么这个描述符是一类导致新任务开始执行的描述符。描述符表中与任务相关的描述符有两类：任务状态段描述符和任务门。当执行权传给这任何一类描述符时，都会造成任务切换。

任务切换很象过程调用，但任务切换会保存更多的处理器状态信息。任务切换会把控制权完全转移到一个新的执行环境，即新任务的执行环境。这种转移操作要求保存处理器中几乎所有寄存器的当前内容，包括标志寄存器 `EFLAGS` 和所有段寄存器。与过程不同，任务不可重入。任务切换不会把任何信息压入堆栈中，处理器的状态信息都被保存在内存中称为任务状态段（Task state segment）的数据结构中。

### 4.7.1 任务的结构和状态

一个任务由两部分构成：任务执行空间和任务状态段 TSS（Task-state segment）。任务执行空间包括代码段、堆栈段和一个或多个数据段，见图 4-33 所示。如果操作系统使用了处理器的特权级保护机制，那么任务执行空间就需要为每个特权级提供一个独立的堆栈空间。TSS 指定了构成任务执行空间的各个段，并且为任务状态信息提供存储空间。在多任务环境中，TSS 也为任务之间的链接提供了处理方法。

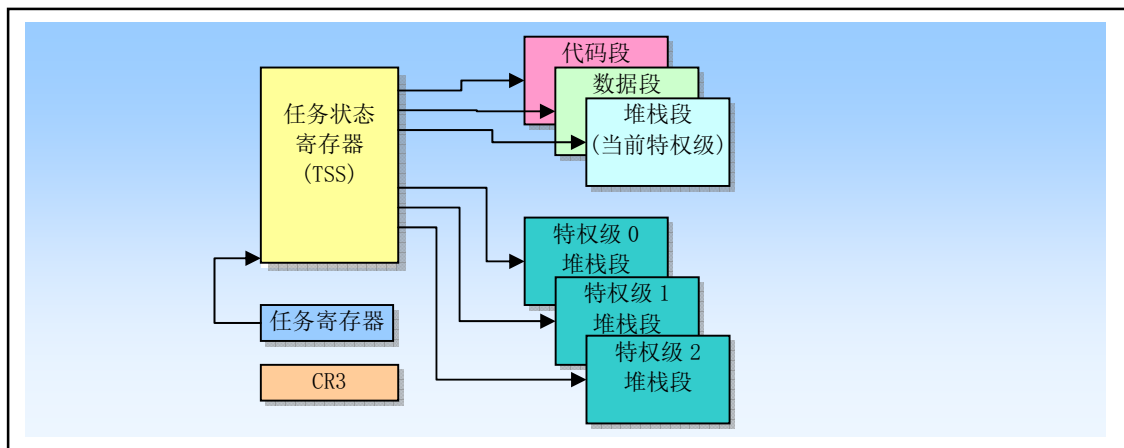


图 4-33 任务的结构和状态

一个任务使用指向其 TSS 的段选择符来指定。当一个任务被加载进处理器中执行时，那么该任务的段选择符、基地址、段限长以及 TSS 段描述符属性就会被加载进任务寄存器 `TR`（Task Register）中。如果使用了分页机制，那么任务使用的页目录表基地址就会被加载进控制寄存器 `CR3` 中。当前执行任务的状态由处理器中的以下内容组成：



- 所有通用寄存器和段寄存器信息；
- 标志寄存器 EFLAGS、程序指针 EIP、控制寄存器 CR3、任务寄存器和 LDTR 寄存器；
- 段寄存器指定的任务当前执行空间；
- I/O 映射位图基地址和 I/O 位图信息（在 TSS 中）；
- 特权级 0、1 和 2 的堆栈指针（在 TSS 中）；
- 链接至前一个任务的链指针（在 TSS 中）。

### 4.7.2 任务的执行

软件或处理器可以使用以下方法之一来调度执行一个任务：

- 使用 CALL 指令明确地调用一个任务；
- 使用 JMP 指令明确地跳转到一个任务（Linux 内核使用的方式）；
- （由处理器）隐含地调用一个中断句柄处理任务；
- 隐含地调用一个异常句柄处理任务；

所有这些调度任务执行的方法都会使用一个指向任务门或任务 TSS 段的选择符来确定一个任务。当使用 CALL 或 JMP 指令调度一个任务时，指令中的选择符既可以-direct选择任务的 TSS，也可以选择存放有 TSS 选择符的任务门。当调度一个任务来处理一个中断或异常时，那么 IDT 中该中断或异常表项必须是一个任务门，并且其中含有中断或异常处理任务的 TSS 选择符。

当调度一个任务执行时，当前正在运行任务和调度任务之间会自动地发生任务切换操作。在任务切换期间，当前运行任务的执行环境（称为任务的状态或上下文）会被保存到它的 TSS 中并且暂停该任务的执行。此后新调度任务的上下文会被加载进处理器中，并且从加载的 EIP 指向的指令处开始执行新任务。

如果当前执行任务（调用者）调用了被调度的新任务（被调用者），那么调用者的 TSS 段选择符会被保存在被调用者 TSS 中，从而提供了一个返回调用者的链接。对于所有 80X86 处理器，任务是不可递归调用的，即任务不能调用或跳转到自己。

中断或异常可以通过切换到一个任务来进行处理。在这种情况下，处理器不仅能够执行任务切换来处理中断或异常，而且也会在中断或异常处理任务返回时自动地切换回被中断的任务中去。这种操作方式可以处理在中断任务执行时发生的中断。

作为任务切换操作的一部份，处理器也会切换到另一个 LDT，从而允许每个任务对基于 LDT 的段具有不同逻辑到物理地址的映射。同时，页目录寄存器 CR3 也会在切换时被重新加载，因此每个任务可以自己的一套页表。这些保护措施能够用来隔绝各个任务并且防止它们相互干扰。

使用处理器的任务管理功能来处理多任务应用是任选的。我们也可以使用软件来实现多任务，使得每个软件定义的任务在一个 80X86 体系结构的任务上下文中执行。

### 4.7.3 任务管理数据结构

处理器定义了一下一些支持多任务的寄存器和数据结构：

- 任务状态段 TSS；
- TSS 描述符；
- 任务寄存器 TR；
- 任务门描述符；
- 标志寄存器 EFLAGS 中的 NT 标志。

使用这些数据结构，处理器可以从一个任务切换到另一个任务，同时保存原任务的上下文，以允许任务重新执行。

#### 4.7.3.1 任务状态段

用于恢复一个任务执行的处理器状态信息被保存在称为任务状态段 TSS（Task state segment）的段中。图 4-34 给出了 32 位 CPU 使用的 TSS 的格式。TSS 段中各字段可分成两大类：动态字段和静态字段。



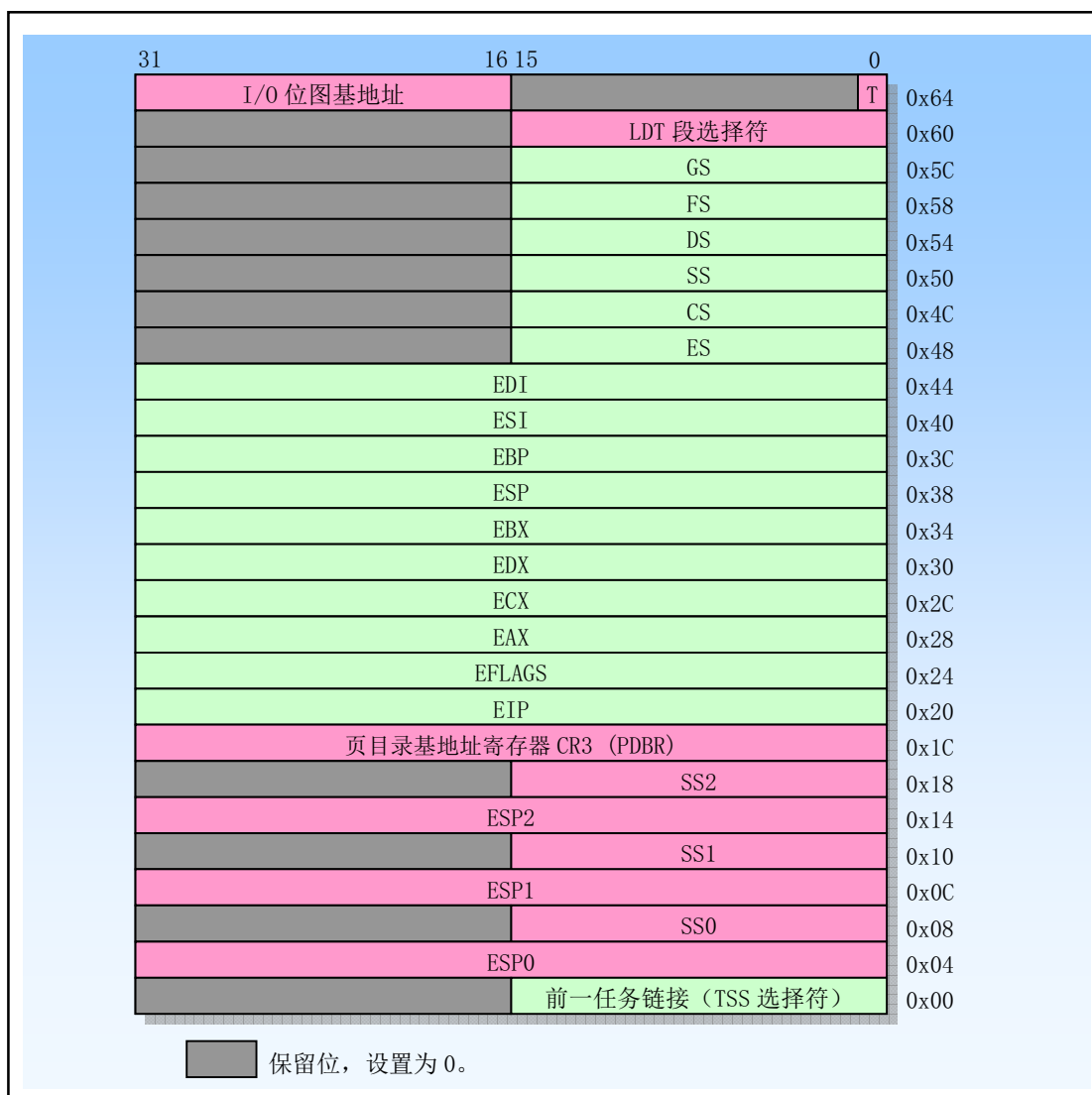


图 4-34 32 位任务状态段 TSS 格式

- 动态字段。当任务切换而被挂起时，处理器会更新动态字段的内容。这些字段包括：
  - 通用寄存器字段。用于保存 EAX、ECX、EDX、EBX、ESP、EBP、ESI 和 EDI 寄存器的内容。
  - 段选择符字段。用于保存 ES、CS、SS、DS、FS 和 GS 段寄存器的内容。
  - 标志寄存器 EFLAGS 字段。在切换之前保存 EFLAGS。
  - 指令指针 EIP 字段。在切换之前保存 EIP 寄存器内容。
  - 先前任务连接字段。含有前一个任务 TSS 段选择符(在调用、中断或异常激发的任务切换时更新)。该字段（通常也称为后连接字段（Back link field））允许任务使用 IRET 指令切换到前一个任务。
- 静态字段。处理器会读取静态字段的内容，但通常不会改变它们。这些字段内容是在任务被创建时设置的。这些字段有：
  - LDT 段选择符字段。含有任务的 LDT 段的选择符。
  - CR3 控制寄存器字段。含有任务使用的页目录物理基地址。控制寄存器 CR3 通常也被称为页目录基地址寄存器 PDBR（Page directory base register）。
  - 特权级 0、1 和 2 的堆栈指针字段。这些堆栈指针由堆栈段选择符（SS0、SS1 和 SS2）和栈中偏

移量指针（ESP0、ESP1 和 ESP2）组成。注意，对于指定的一个任务，这些字段的值是不变的。因此，如果任务中发生堆栈切换，寄存器 SS 和 ESP 的内容将会改变。

- 调试陷阱（Debug Trap）T 标志字段。该字段位于字节 0x64 比特 0 处。当设置了该位时，处理器切换到该任务的操作将产生一个调试异常。
- I/O 位图基地址字段。该字段含有从 TSS 段开始处到 I/O 许可位图处的 16 位偏移值。

如果使用了分页机制，那么在任务切换期间应该避免处理器操作的 TSS 段中（前 104 字节中）含有内存页边界。如果 TSS 这部分包含内存页边界，那么该边界处两边的页面都必须同时并且连续存在于内存中。另外，如果使用了分页机制，那么与原任务 TSS 和新任务 TSS 相关的页面，以及对应的描述符表项应该是可读写的。

### 4.7.3.2 TSS 描述符

与其他段一样，任务状态段 TSS 也是使用段描述符来定义。图 4-35 给出了 TSS 描述符的格式。TSS 描述符只能存放在 GDT 中。

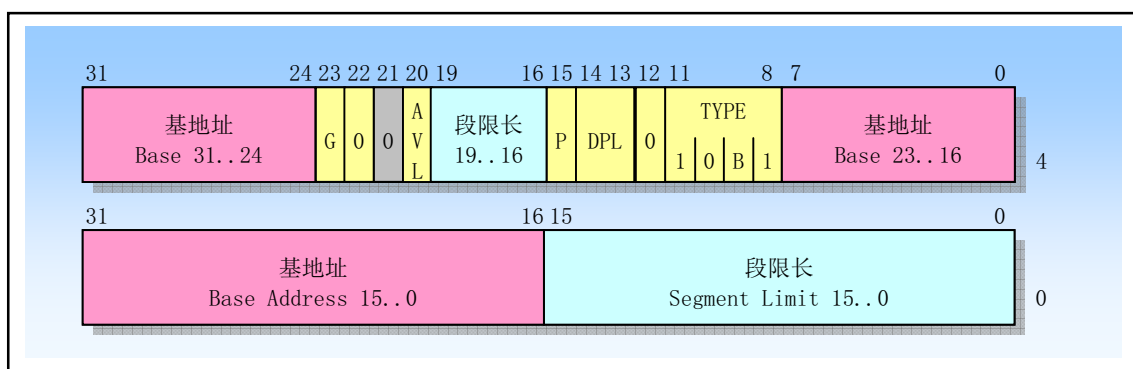


图 4-35 TSS 段描述符格式

类型字段 TYPE 中的忙标志 B 用于指明任务是否处于忙状态。忙状态的任务是当前正在执行的任务或等待执行（被挂起）的任务。值为 0b1001 的类型字段表明任务处于非活动状态；而值为 0b1011 的类型字段表示任务正忙。任务是不可以递归执行的，因此处理器使用忙标志 B 来检测任何企图对被中断执行任务的调用。

其中基地址、段限长、描述符特权级 DPL、颗粒度 G 和存在位具有与数据段描述符中相应字段同样的功能。当 G=0 时，限长字段必须具有等于或大于 103（0x67）的值，即 TSS 段的最小长度不得小于 104 字节。如果 TSS 段中还包含 I/O 许可位图，那么 TSS 段长度需要大一些。另外，如果操作系统还想在 TSS 段中存放其他一些信息，那么 TSS 段就需要更大的长度。

使用调用或跳转指令，任何可以访问 TSS 描述符的程序都能够造成任务切换。可以访问 TSS 描述符的程序其 CPL 数值必须小于或等于 TSS 描述符的 DPL。在大多数系统中，TSS 描述符的 DPL 字段值应该设置成小于 3。这样，只有具有特权级的软件可以执行任务切换操作。然而在多任务应用中，某些 TSS 的 DPL 可以设置成 3，以使得在用户特权级上也能进行任务切换操作。

可访问一个 TSS 段描述符并没有给程序读写该描述符的能力。若想读或修改一个 TSS 段描述符，可以使用映射到内存相同位置的数据段描述符（即别名描述符）来操作。把 TSS 描述符加载进任何段寄存器将导致一个异常。企图使用 TI 标志置位的选择符（即当前 LDT 中的选择符）来访问 TSS 段也将导致异常。

### 4.7.3.3 任务寄存器

任务寄存器 TR（Task Register）中存放着 16 位的段选择符以及当前任务 TSS 段的整个描述符（不可见部分）。这些信息是从 GDT 中当前任务的 TSS 描述符中复制过来的。处理器使用任务寄存器 TR 的不可见部分来缓冲 TSS 段描述符内容。

指令 LTR 和 STR 分别用于加载和保存任务寄存器的可见部分，即 TSS 段的选择符。LTR 指令只能被特权级 0 的程序执行。LTR 指令通常用于系统初始化期间给 TR 寄存器加载初值（例如，任务 0 的 TSS 段选择符），随后在系统运行期间，TR 的内容会在任务切换时自动地被改变。

#### 4.7.3.4 任务门描述符

任务门描述符（Task gate descriptor）提供对一个任务间接、受保护地的引用，其格式见图所示。任务门描述符可以被存放在 GDT、LDT 或 IDT 表中。

任务门描述符中的 TSS 选择符字段指向 GDT 中的一个 TSS 段描述符。这个 TSS 选择符字段中的 RPL 域不用。任务门描述符中的 DPL 用于在任务切换时控制对 TSS 段的访问。当程序通过任务门调用或跳转到一个任务时，程序的 CPL 以及指向任务门的门选择符的 RPL 值必须小于或等于任务门描述符中的 DPL。请注意，当使用任务门时，目标 TSS 段描述符的 DPL 忽略不用。

程序可以通过任务门描述符或者 TSS 段描述符来访问一个任务。图 4-36 示出了 LDT、GDT 和 IDT 表中的任务门如何都指向同一个任务。

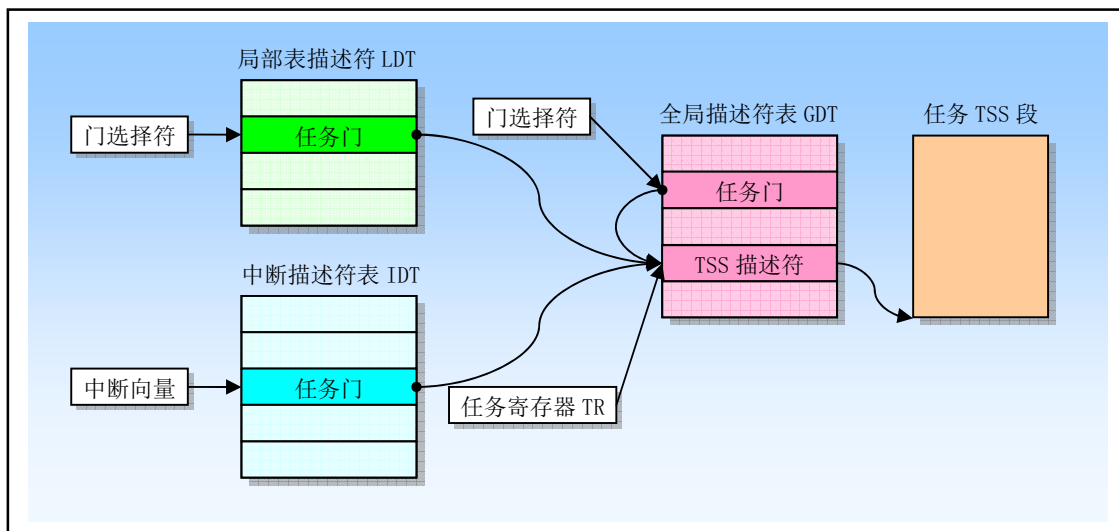


图 4-36 引用同一任务的任务门

#### 4.7.4 任务切换

处理器可使用一下 4 种方式之一执行任务切换操作：

1. 当前任务对 GDT 中的 TSS 描述符执行 JMP 或 CALL 指令；
2. 当前任务对 GDT 或 LDT 中的任务门描述符执行 JMP 或 CALL 指令；
3. 中断或异常向量指向 IDT 表中的任务门描述符；
4. 当 EFLAGS 中的 NT 标志置位时当前任务执行 IRET 指令。

JMP、CALL 和 IRET 指令以及中断和异常都是处理器的普通机制，可用于不发生任务切换的环境中。对于 TSS 描述符或任务门的引用（当调用或跳转到一个任务），或者 NT 标志的状态（当执行 IRET 指令时）确定了是否会发生任务切换。

为了进行任务切换，JMP 或 CALL 指令能够把控制转移到 TSS 描述符或任务门上。使用这两种方式的作用相同，都会导致处理器把控制转移到指定的任务中，见图 4-37 所示。

当中断或异常的向量索引的是 IDT 中的一个任务门时，一个中断或异常就会造成任务切换。如果向量索引的是 IDT 中的一个中断或陷阱门，则不会造成任务切换。

中断服务过程总是把执行权返回到被中断的过程中，被中断的过程可能在另一个任务中。如果 NT 标

志处于复位状态，则执行一般返回处理。如果 NT 标志是置位状态，则返回操作会产生任务切换。切换到新任务由中断服务过程 TSS 中的 TSS 选择符（前一任务链接字段）指定。

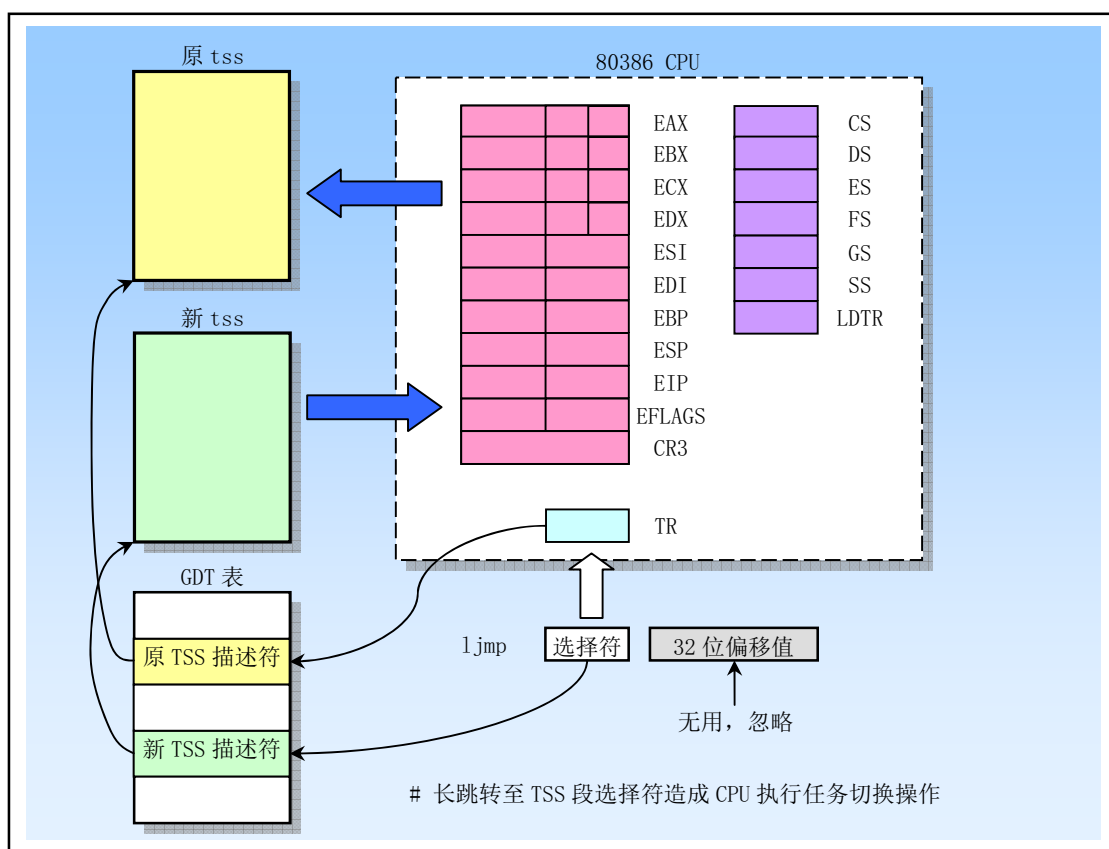


图 4-37 任务切换操作示意图

当切换到一个新任务时，处理器会执行一下操作：

1. 从作为 JMP 或 CALL 指令操作数中，或者从任务门中，或者从当前 TSS 的前一任务链接字段（对于由 IRET 引起的任务切换）中取得新任务的 TSS 段选择符。
2. 检查当前任务是否允许切换到新任务。把数据访问特权级规则应用到 JMP 和 CALL 指令上。当前任务的 CPL 和新任务段选择符的 RPL 必须小于或等于 TSS 段描述符的 DPL，或者引用的是一个任务门。无论目标任务门或 TSS 段描述符的 DPL 是何值，异常、中断（除了使用 INT n 指令产生的中断）和 IRET 指令都允许执行任务切换。对于 INT n 指令产生的中断将检查 DPL。
3. 检查新任务的 TSS 描述符是标注为存在的（P=1），并且 TSS 段长度有效（大于 0x67）。当试图执行会产生错误的指令时，都会恢复对处理器状态的任何改变。这使得异常处理过程的返回地址指向出错指令，而非出错指令随后的一条指令。因此异常处理过程可以处理出错条件并且重新执行任务。异常处理过程的介入处理对应用程序来说是完全透明的。
4. 如果任务切换产生自 JMP 或 IRET 指令，处理器就会把当前任务（老任务）TSS 描述符中的忙标志 B 复位；如果任务切换是由 CALL 指令、异常或中断产生，则忙标志 B 不动。
5. 如果任务切换由 IRET 产生，则处理器会把临时保存的 EFLAGS 映像中的 NT 标志复位；如果任务切换由 CALL、JMP 指令或者异常或中断产生，则不用改动上述 NT 标志。
6. 把当前任务的状态保存到当前任务的 TSS 中。处理器会从任务寄存器中取得当前任务 TSS 的基地址，并且把一下寄存器内容复制到当前 TSS 中：所有通用寄存器、段寄存器中的段选择符、标志

寄存器 EFLAGS 以及指令指针 EIP。

7. 如果任务切换是由 CALL 指令、异常或中断产生，则处理器就会把从新任务中加载的 EFLAGS 中的 NT 标志置位。如果任务切换产生自 JMP 或 IRET 指令，就不改动新加载 EFLAGS 中的标志。
8. 如果任务切换由 CALL、JMP 指令或者异常或中断产生，处理器就会设置新任务 TSS 描述符中的忙标志 B。如果任务切换由 IRET 产生，则不去改动 B 标志。
9. 使用新任务 TSS 的段选择符和描述符加载任务寄存器 TR（包括隐藏部分）。设置 CR0 寄存器的 TS 标志。
10. 把新任务的 TSS 状态加载进处理器。这包括 LDTR 寄存器、PDBR（CR3）寄存器、EFLAGS 寄存器、EIP 寄存器以及通用寄存器和段选择符。在此期间检测到的任何错误都将出现在新任务的上下文中。
11. 开始执行新任务（对于异常处理过程，新任务的第一条指令显现出还没有执行）。

当成功地进行了任务切换操作，当前执行任务的状态总是会被保存起来。当任务恢复执行时，任务将从保存的 EIP 指向的指令处开始执行，并且所有寄存器都恢复到任务挂起时的值。

当执行任务切换时，新任务的特权级与原任务的特权级没有任何关系。新任务在 CS 寄存器的 CPL 字段指定的特权级上开始运行。因为各个任务通过它们独立的地址空间和 TSS 段相互隔绝，并且特权级规则已经控制对 TSS 的访问，所以在任务切换时软件不需要再进行特权级检查。

每次任务切换都会设置控制寄存器 CR0 中的任务切换标志 TS。该标志对系统软件非常有用。系统软件可用 TS 标志来协调处理器和浮点协处理器之间的操作。TS 标志表明协处理器中的上下文内容可能与当前正在执行任务的不一致。

### 4.7.5 任务链

TSS 的前一任务连接（Backlink）字段以及 EFLAGS 中的 NT 标志用于返回到前一个任务操作中。NT 标志指出了当前执行的任务是否是嵌套在另一个任务中执行，并且当前任务的前一任务连接字段中存放着嵌套层中更高层任务的 TSS 选择符，若有的话（见图 4-38 所示）。

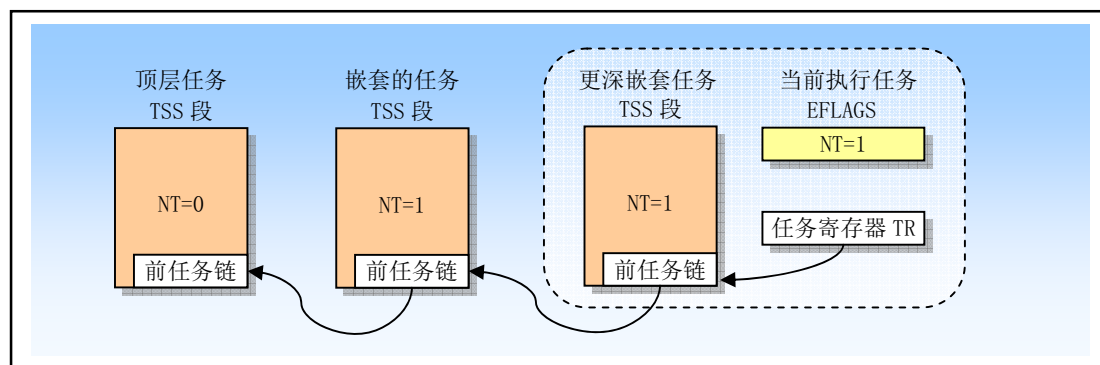


图 4-38 任务链示意图

当 CALL 指令、中断或异常造成任务切换，处理器把当前 TSS 段的选择符复制到新任务 TSS 段的前一任务链接字段中，然后在 EFLAGS 中设置 NT 标志。NT 标志指明 TSS 的前一任务链接字段中存放有保存的 TSS 段选择符。如果软件使用 IRET 指令挂起新任务，处理器就会使用前一任务链接字段中值和 NT 标志返回到前一个任务。也即如果 NT 标志是置位的话，处理器会切换到前一任务链接字段指定的任务去执行。

注意，当任务切换是由 JMP 指令造成，那么新任务就不会是嵌套的。也即，NT 标志会被设置为 0，并且不使用前一任务链接字段。JMP 指令用于不希望出现嵌套的任务切换中。

表 4-10 总结了任务切换期间，忙标志 B（在 TSS 段描述符中）、NT 标志、前一任务链接字段和 TS 标

志（在 CR0 中）的用法。注意，运行于任何特权级上的程序都可以修改 NT 标志，因此任何程序都可以设置 NT 标志并执行 IRET 指令。这种做法会让处理器去执行当前任务 TSS 的前一任务链接字段指定的任务。为了避免这种伪造的任务切换执行成功，操作系统应该把每个 TSS 的该字段初始化为 0。

表 4-10 任务切换对忙标志、NT 标志、前一任务链字段和 TS 标志的影响

标志或字段	JMP 指令的影响	CALL 指令或中断的影响	IRET 指令的影响
新任务忙标志 B	设置标志。以前须已被清除	设置标志。以前须已被清除	不变。必须已被设置。
老任务忙标志 B	被清除。	不变。当前处于设置状态。	设置标志。
新任务 NT 标志	设置成新任务 TSS 中的值。	设置标志	设置成新任务 TSS 中的值。
老任务 NT 标志	不变	不变	清除标志
新任务链接字段	不变	存放老任务 TSS 段选择符	不变
老人物链接字段	不变	不变	不变
CR0 中 TS 标志	设置标志	设置标志	设置标志

## 4.7.6 任务地址空间

任务的地址空间由任务能够访问的段构成。这些段包括代码段、数据段、堆栈段、TSS 中引用的系统段以及任务代码能够访问的任何其他段。这些段都被映射到处理器的线性地址空间中，并且随后被直接地或者通过分页机制映射到处理器的物理地址空间中。

TSS 中的 LDT 字段可以用于给出每个任务自己的 LDT。对于一个给定的任务，通过把与任务相关的所有段描述符放入 LDT 中，任务的地址空间就可以与其他任务的隔绝开来。

当然，几个任务也可以使用同一个 LDT。这是一种简单而有效的允许某些任务互相通信或控制的方法，而无须抛弃整个系统的保护屏障。

因为所有任务都可以访问 GDT，所以也同样可以创建通过此表访问的共享段。

如果开启了分页机制，则 TSS 中的 CR3 寄存器字段可以让每个任务有它自己的页表。或者，几个任务能够共享相同页表集。

### 4.7.6.1 把任务映射到线性和物理地址空间

有两种方法可以把任务映射到线性地址空间和物理地址空间：

- 所有任务共享一个线性到物理地址空间的映射。当没有开启分页机制时，就只能使用这个办法。不开启分页时，所有线性地址映射到相同的物理地址上。当开启了分页机制，那么通过让所有任务使用一个页目录，我们就可以使用这种从线性到物理地址空间的映射形式。如果支持需求页虚拟存储技术，则线性地址空间可以超过现有物理地址空间的大小。
- 每个任务有自己的线性地址空间，并映射到物理地址空间。通过让每个任务使用不同的页目录，我们就可以使用这种映射形式。因为每次任务切换都会加载 PDBR（控制寄存器 CR3），所以每个任务可以有不同的页目录。

不同任务的线性地址空间可以映射到完全不同的物理地址上。如果不同页目录的条目（表项）指向不同的页表，而且页表也指向物理地址中不同的页面上，那么各个任务就不会共享任何物理地址。

对于映射任务线性地址空间的这两种方法，所有任务的 TSS 都必须存放在共享的物理地址空间区域中，并且所有任务都能访问这个区域。为了让处理器执行任务切换而读取或更新 TSS 时，TSS 地址的映射不会改变，就需要使用这种映射方式。GDT 所映射的线性地址空间也应该映射到共享的物理地址空间中。否则就丧失了 GDT 的作用。

### 4.7.6.2 任务逻辑地址空间

为了在任务之间共享数据，可使用下列方法之一来为数据段建立共享的逻辑到物理地址空间的映射：通过使用 GDT 中的段描述符。所有任务必须能够访问 GDT 中的段描述符。如果 GDT 中的某些段描



述符指向线性地址空间中的一些段，并且这些段被映射到所有任务共享的物理地址空间中，那么所有任务都可以共享这些段中的代码和数据。

通过共享的 LDT。两个或多个任务可以使用相同的 LDT，如果它们 TSS 中 LDT 字段指向同一个 LDT。如果一个共享的 LDT 中某些段描述符指向映射到物理地址空间公共区域的段，那么共享 LDT 的所有任务可以共享这些段中的所有代码和数据。这种共享方式要比通过 GDT 来共享好，因为这样做可以把共享局限于指定的一些任务中。系统中有与此不同 LDT 的其他任务没有访问这些共享段的权利。

通过映射到线性地址空间公共地址区域的不同 LDT 中的段描述符。如果线性地址空间中的这个公共区域对每个任务都映射到物理地址空间的相同区域，那么这些段描述符就允许任务共享这些段。这样的段描述符通常称为别名段。这个共享方式要比上面给出的方式来得更好，因为 LDT 中的其他段描述符可以指向独立的未共享线性地址区域。

## 4.8 保护模式编程初始化

我们知道，80X86 可以工作在几种模式下。当机器上电或硬件复位时，处理器工作在 8086 处理器兼容的实地址模式下，并且从物理地址 0xFFFFF0 处开始执行软件初始化代码（通常在 EPROM 中）。软件初始化代码首先必须设置基本系统功能操作必要的数据结构信息，例如处理中断和异常的实模式 IDT 表（即中断向量表）。如果处理器将仍然工作在实模式下，那么软件必须加载操作系统模块和相应数据以允许应用程序能在实模式下可靠地运行。如果处理器将要工作在保护模式下，那么操作系统软件就必须加载保护模式操作必要的数据结构信息，然后切换到保护模式。

### 4.8.1 进入保护模式时的初始化操作

保护模式所需要的一些数据结构由处理器内存管理功能确定。处理器支持分段模型，可以使用从单个、统一的地址空间平坦模型到每个任务都具有几个受保护地址空间的高度结构化的多段模型。分页机制能够用来部分在内存、部分在磁盘上的大型数据结构信息。这两种地址转换形式都需要操作系统在内存中为内存管理硬件设置所要求的数据结构。因此在处理器能够被切换到保护模式下运行之前，操作系统加载和初始化软件（bootsect.s、setup.s 和 head.s）必须在内存中先设置好保护模式下使用的数据结构的基本信息。这些数据结构包括以下几种：

- 保护模式中断描述符表 IDT；
- 全局描述符表 GDT；
- 任务状态段 TSS；
- 局部描述符表 LDT；
- 若使用分页机制，则起码需要设置一个页目录和一个页表；
- 处理器切换到保护模式下运行的代码段；
- 含有中断和异常处理程序的代码模块。

在能够切换到保护模式之前，软件初始化代码还必须设置以下系统寄存器：

- 全局描述符表基地址寄存器 GDTR；
- 中断描述符表基地址寄存器 IDTR；
- 控制寄存器 CR1--CR3；

在初始化了这些数据结构、代码模块和系统寄存器之后，通过设置 CR0 寄存器的保护模式标志 PE（位 0），处理器就可以切换到保护模式下运行。

#### 4.8.1.1 保护模式系统结构表

软件初始化期间在内存中设置的保护模式系统表主要依赖于操作系统将要支持的内存管理类型：平坦的、平坦并支持分页的、分段的或者分段并支持分页的。

为了实现无分页的平坦内存模型，软件初始化代码必须起码设置具有一个代码段和一个数据段的 GDT 表。当然 GDT 表第 1 项还需要放置一个空描述符。堆栈可以放置在普通可读写数据段中，因此并不需要

专门的堆栈描述符。支持分页机制的平坦内存模型还需要一个页目录和至少一个页表。在可以使用 GDT 表之前，必须使用 LGDT 指令把 GDT 表的基地址和长度值加载到 GDTR 寄存器中。

而多段模型则还需要用于操作系统的其他段，以及用于每个应用程序的段和 LDT 表段。LDT 表的段描述符要求存放在 GDT 表中。某些操作系统会为应用程序另行分配新段和新的 LDT 段。这种做法为动态编程环境提供了最大灵活性，例如 Linux 操作系统就使用了这种方式。象过程控制器那样的嵌入式系统可以预先为固定数量的应用程序分配固定数量的段和 LDT，这是实现实时系统软件环境结构的一种简单而有效的方法。

#### 4.8.1.2 保护模式异常和中断初始化

软件初始化代码必须设置一个保护模式 IDT，其中最少数需要含有处理器可能产生的每个异常向量对应的门描述符。如果使用了中断或陷阱门，那么门描述符可以都指向包含中断和异常处理过程的同一个代码段。若使用了任务门，那么每个使用任务门的异常处理过程都需要一个 TSS 以及相关的代码、数据和堆栈段。如果允许硬件产生中断，那么必须在 IDT 中为一个或多个中断处理过程设置门描述符。

在可以使用 IDT 之前，必须使用 LIDT 指令把 IDT 表基地址和长度加载到 IDTR 寄存器中。

#### 4.8.1.3 分页机制初始化

分页机制由控制寄存器 CR0 中的 PG 标志设置。当这个标志被清 0 时（即硬件复位时的状态），分页机制被关闭；当设置了 PG 标志，就开启分页机制。在设置 PG 标志之前，必须先初始化以下数据结构和寄存器：

- 软件必须在物理内存中建立至少一个页目录和一个页表。如果页目录表中含有指向自身的目录项时，可以不使用页表。此时页目录表和页表被存放在同一页面中。
- 把页目录表的物理基地址加载到 CR3 寄存器中（也称为 PDBR 寄存器）。
- 处理器处于保护模式下。如果满足所有其他限制，则 PG 和 PE 标志可以同时设置。

为保持兼容性，设置 PG 标志（以及 PE 标志）时必须遵守以下规则：

- 设置 PG 标志的指令应该立刻跟随一条 JMP 指令。MOV CR0 指令后面的 JMP 指令会改变执行流，所以它会清空 80X86 处理器已经取得或已译码的指令。然而，Pentium 及以上处理器使用了分支目标缓冲器 BTB（Branch Target Buffer）为分支代码定向，因此减去了为分支指令刷新队列的需要。
- 设置 PG 标志到跳转指令 JMP 之间的代码必须来自对等映射（即跳转之前的线性地址与开启分页后的物理地址相同）的一个页面上。

#### 4.8.1.4 多任务初始化

如果将要使用多任务机制，并且/或者允许改变特权级，那么软件初始化代码必须至少设置一个 TSS 及相应的 TSS 段描述符（因为特权级 0、1 和 2 的各栈段指针需要从 TSS 中取得）。在创建 TSS 描述符时不要将其标注为忙（不要设置忙标志），该标志仅由处理器在执行任务切换时设置。与 LDT 段描述符相同，TSS 的描述符也存放在 GDT 中。

在处理器切换到保护模式之后，可以用 LTR 指令把 TSS 段描述符的选择符加载到任务寄存器 TR 中。这个指令会把 TSS 标记成忙状态（B=1），但是并不执行任务切换操作。然后处理器可以使用这个 TSS 来定位特权级 0、1 和 2 的堆栈。在保护模式中，软件进行第一次任务切换之前必须首先加载 TSS 段的选择符，因为任务切换会把当前任务状态复制到该 TSS 中。

在 LTR 指令执行之后，随后对任务寄存器的操作由任务切换进行。与其他的段和 LDT 类似，TSS 段和 TSS 段描述符可以预先设置好，也可以在需要时进行设置。

### 4.8.2 模式切换

为了让处理器工作在保护模式下，必须从实地址模式下进行模式切换操作。一旦进入保护模式，软件通常不会再需要回到实地址模式。为了还能运行实地址模式编制的程序，通常在虚拟-8086 模式中运行比再切换回实模式下运行更为方便。



### 4.8.2.1 切换到保护模式

在切换到保护模式之前，必须首先加载一些起码的系统数据结构和代码模块。一旦建立了这些系统表，软件初始化代码就可以切换到保护模式中。通过执行在 CR0 寄存器中设置 PE 标志的 MOV CR0 指令，我们就可以进行保护模式。（在同一个指令中，CR0 的 PG 标志可用于开启分页机制。）刚进入保护模式中运行时，特权级是 0。为了保证程序的兼容性，切换操作应该按照以下步骤进行：

1. 禁止中断。使用 CLI 指令可以禁止可屏蔽硬件中断。NMI 会由硬件电路来禁止。同时软件应该确保在模式切换操作期间不产生异常和中断。
2. 执行 LGDT 指令把 GDT 表的基地址加载进 GDTR 寄存器。
3. 执行在控制寄存器 CR0 中设置 PE 标志（可选同时设置 PG 标志）的 MOV CR0 指令。
4. 在 MOV CR0 指令之后立刻执行一个远跳转 JMP 或远调用 CALL 指令。这个操作通常是远跳转到或远调用指令流中的下一条指令。
5. 若要使用局部描述符表，则执行 LLDT 指令把 LDT 段的选择符加载到 LDTR 寄存器中。
6. 执行 LTR 指令，用初始保护模式任务的段选择符或者可写内存区域的段描述符加载任务寄存器 TR。这个可写内存区域用于在任务切换时存放任务的 TSS 信息。
7. 在进入保护模式后，段寄存器仍然含有在实地址模式时的内容。第 4 步中的 JMP 或 CALL 指令会重置 CS 寄存器。执行以下操作之一可以更新其余段寄存器的内容：其余段寄存器的内容可通过重新加载或切换到一个新任务来更新。
8. 执行 LIDT 指令把保护模式 IDT 表的基地址和长度加载到 IDTR 寄存器中。
9. 执行 STI 指令开启可屏蔽硬件中断，并且执行必要的硬件操作开启 NMI 中断。

另外，MOV CR0 指令之后紧接着的 JMP 或 CALL 指令会改变执行流。如果开启了分页机制，那么 MOV CR0 指令到 JMP 或 CALL 指令之间的代码必须来自对等映射（即跳转之前的线性地址与开启分页后的物理地址相同）的一个页面上。而 JMP 或 CALL 指令跳转到的目标指令并不需要处于对等映射页面上。

### 4.8.2.2 切换回实地址模式

若想切换回实地址模式，则可以使用 MOV CR0 指令把控制寄存器 CR0 中的 PE 标志清 0。重新进入实地址模式的过程应该按照以下步骤进行：

1. 禁止中断。使用 CLI 指令可以禁止可屏蔽硬件中断。NMI 会由硬件电路来禁止。同时软件应该确保在模式切换操作期间不产生异常和中断。
2. 如果已开启分页机制，那么需要执行：
  - 把程序的控制转移到对等映射的线性地址处（即线性地址等于物理地址）。
  - 确保 GDT 和 IDT 在对等映射的页面上。
  - 清除 CR0 中的 PG 标志。
  - CR3 寄存器置为 0x00，用于刷新 TLB 缓冲。
3. 把程序的控制转移到长度为 64KB（0xFFFF）的可读段中。这步操作使用实模式要求的段长度加载 CS 寄存器。
4. 使用指向含有以下设置值的描述符的选择符来加载 SS、DS、ES、FS 和 GS 段寄存器。
  - 段限长 Limit = 64KB
  - 字节颗粒度（G=0）。
  - 向上扩展（E=0）。
  - 可写（W=1）。
  - 存在（P=1）。
5. 执行 LIDT 指令来指向在 1MB 实模式地址范围内的实地址模式中断表。
6. 清除 CR0 中的 PE 标志来切换到实地址模式。
7. 执行一个远跳转指令跳转到一个实模式程序中。这步操作会刷新指令队列并且为 CS 寄存器加载合适的基地址和访问权限值。

8. 加载实地址模式程序代码会使用的 SS、DS、ES、FS 和 GS 寄存器。
9. 执行 STI 指令开启可屏蔽硬件中断，并且执行必要的硬件操作开启 NMI 中断。

## 4.9 一个简单的多任务内核实例

作为对本章和前几章内容的总结，本节完整描述了一个简单多任务内核的设计和实现方法。这个内核示例中包含两个特权级 3 的用户任务和一个系统调用中断过程。我们首先说明这个简单内核的基本结构和加载运行的基本原理，然后描述它是如何被加载进机器 RAM 内存中以及两个任务是如何进行切换运行的。最后我们给出实现这个简单内核的源程序：启动引导程序 `boot.s` 和保护模式多任务内核程序 `head.s`。

### 4.9.1 多任务程序结构和工作原理

本节给出的内核实例由 2 个文件构成。一个是使用 `as86` 语言编制的引导启动程序 `boot.s`，用于在计算机系统加电时从启动盘上把内核代码加载到内存中；另一个是使用 `GNU as` 汇编语言编制的内核程序 `head.s`，其中实现了两个运行在特权级 3 上的任务在时钟中断控制下相互切换运行，并且还实现了在屏幕上显示字符的一个系统调用。我们把这两个任务分别称为任务 A 和任务 B（或任务 0 和任务 1），它们会调用这个显示系统调用在屏幕上分别显示出字符 'A' 和 'B'，直到每个 10 毫秒切换到另一个任务。任务 A 连续循环地调用系统调用在屏幕上显示字符 'A'；任务 B 则一直显示字符 'B'。若要终止这个内核实例程序，则需要重新启动机器，或者关闭运行的模拟 PC 运行环境软件。

`boot.s` 程序编译出的代码共 512 字节，将被存放在软盘映像文件的第一个扇区中，见图 4-39 所示。PC 机在加电启动时，ROM BIOS 中的程序会把启动盘上第一个扇区加载到物理内存 `0x7c00`（31KB）位置开始处，并把执行权转移到 `0x7c00` 处开始运行 `boot` 程序代码。

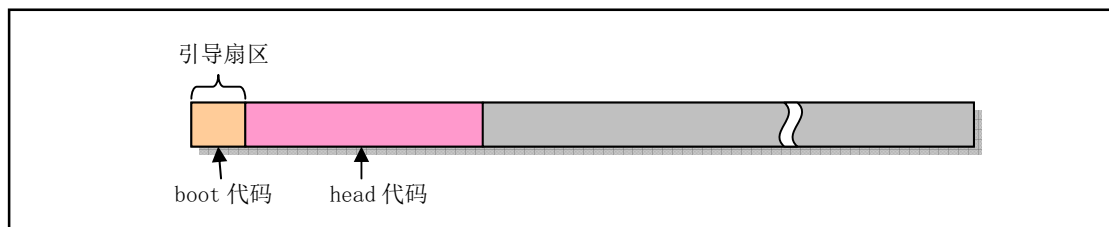


图 4-39 软盘映像文件示意图

`boot` 程序的主要功能是把软盘或映像文件中的 `head` 内核代码加载到内存某个指定位置处，并在设置好临时 GDT 表等信息后，把处理器设置成运行在保护模式下，然后跳转到 `head` 代码处去运行内核代码。实际上，`boot.s` 程序会首先利用 ROM BIOS 中断 `int 0x13` 把软盘中的 `head` 代码读入到内存 `0x10000`（64KB）位置开始处，然后再把这段 `head` 代码移动到内存 0 开始处。最后设置控制寄存器 `CR0` 中的开启保护运行模式标志，并跳转到内存 0 处开始执行 `head` 代码。`boot` 程序代码在内存中移动 `head` 代码的示意图见图 4-40 所示。

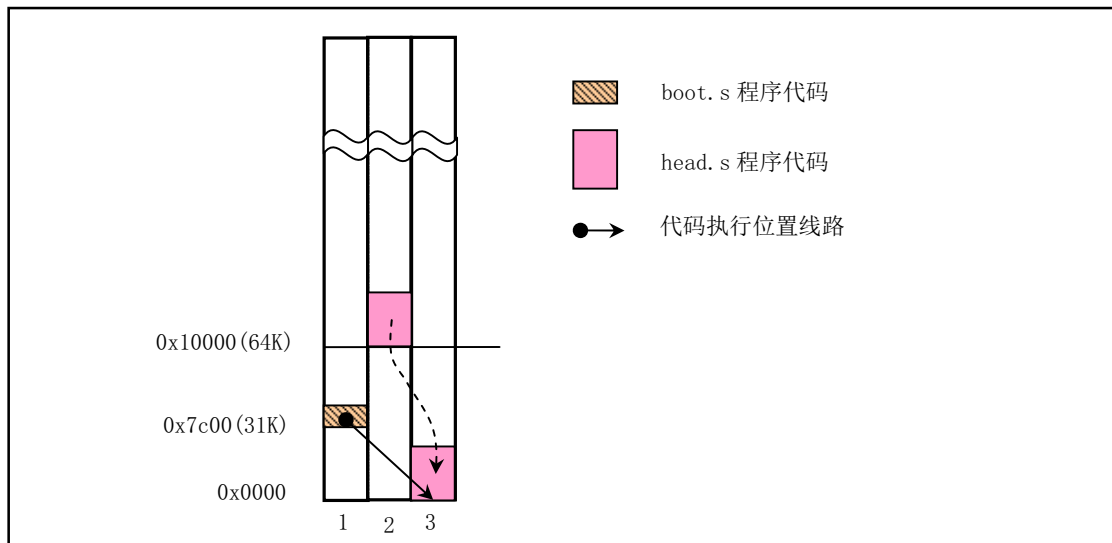


图 4-40 内核代码在物理内存中的移动和分布情况

把 head 内核代码移动到物理内存 0 开始处的主要原因是为了设置 GDT 表时可以简单一些，因而也能让 head.s 程序尽量短一些。但是我们不能让 boot 程序把 head 代码从软盘或映像文件中直接加载到内存 0 处。因为加载操作需要使用 ROM BIOS 提供的中断过程，而 BIOS 使用的中断向量表正处于内存 0 开始的地方，并且在内存 1KB 开始处是 BIOS 程序使用的数据区，所以若直接把 head 代码加载到内存 0 处将使得 BIOS 中断过程不能正常运行。当然我们也可以把 head 代码加载到内存 0x10000 处后就直接跳转到该处运行 head 代码，使用这种方式的源程序可从 [oldlinux.org](http://oldlinux.org) 网站下载，见下面说明。

head.s 程序运行在 32 位保护模式下，其中主要包括初始设置的代码、时钟中断 int 0x08 的过程代码、系统调用中断 int 0x80 的过程代码以及任务 A 和任务 B 等的代码和数据。其中初始设置工作主要包括：①重新设置 GDT 表；②设置系统定时器芯片；③重新设置 IDT 表并且设置时钟和系统调用中断门；④移动到任务 A 中执行。

在虚拟地址空间中 head.s 程序的内核代码和任务代码分配图如图 4-41 所示。实际上，本内核示例中所有代码和数据段都对应到物理内存同一个区域上，即从物理内存 0 开始的区域。GDT 中全局代码段和数据段描述符的内容都设置为：基地址为 0x0000；段限长值为 0x07ff。因为颗粒度为 1，所以实际段长度为 8MB。而全局显示数据段被设置成：基地址为 0xb8000；段限长值为 0x0002，所以实际段长度为 8KB，对应到显示内存区域上。

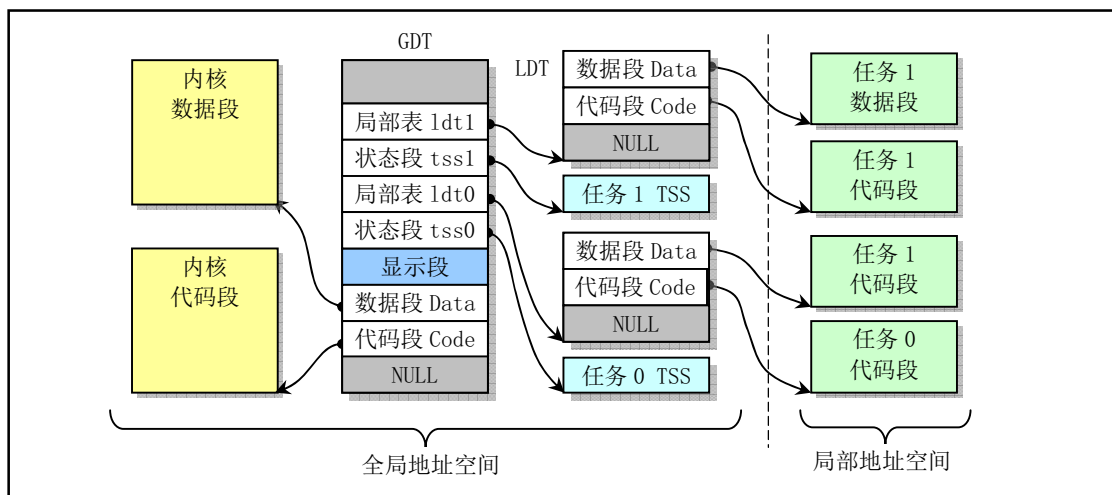


图 4-41 内核和任务在虚拟地址空间中的分配示意图

两个任务的在 LDT 中代码段和数据段描述符的内容也都设置为:基地址为 0x0000;段限长值为 0x03ff, 实际段长度为 4MB。因此在线性地址空间中这个“内核”的代码和数据段与任务的代码和数据段都从线性地址 0 开始并且由于没有采用分页机制, 所以它们都直接对应物理地址 0 开始处。在 head 程序编译出的目标文件中以及最终得到的软盘映像文件中, 代码和数据的组织形式见图 4-42 所示。

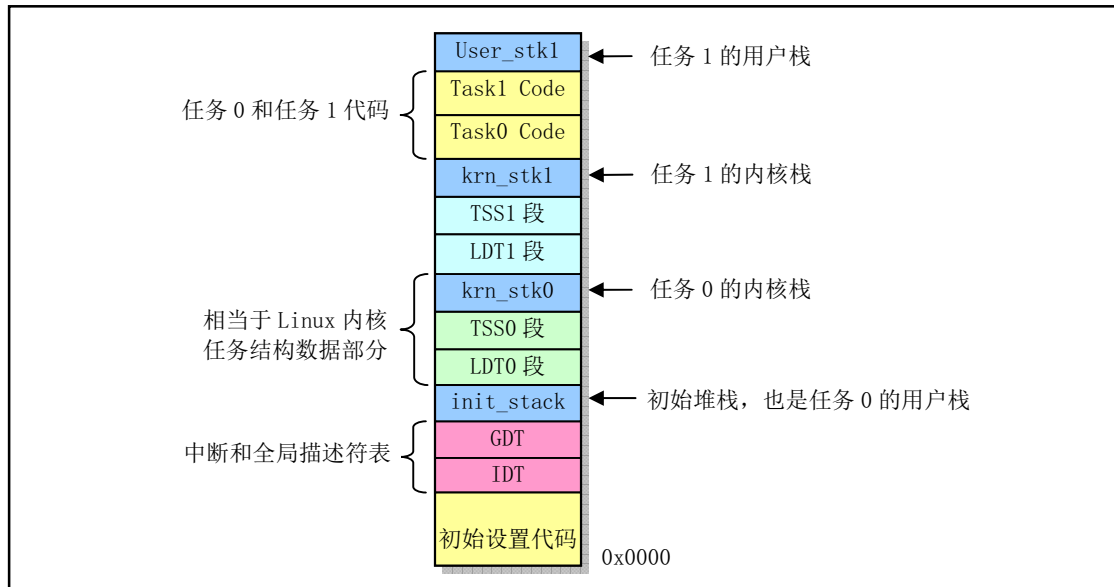


图 4-42 内核映像文件和内存中 head 代码和数据分布示意图

由于处于特权级 0 的代码不能直接把控制权转移到特权级 3 的代码中执行, 但中断返回操作是可以的, 因此当初始化 GDT、IDT 和定时芯片结束后, 我们就利用中断返回指令 IRET 来启动运行第 1 个任务。具体实现方法是在初始堆栈 init\_stack 中人工设置一个返回环境。即把任务 0 的 TSS 段选择符加载到任务寄存器 LTR 中、LDT 段选择符加载到 LDTR 中以后, 把任务 0 的用户栈指针 (0x17:init\_stack) 和代码指针 (0x0f:task0) 以及标志寄存器值压入栈中, 然后执行中断返回指令 IRET。该指令会弹出堆栈上的堆栈指针作为任务 0 用户栈指针, 恢复假设的任务 0 的标志寄存器内容, 并且弹出栈中代码指针放入 CS:EIP 寄存器中, 从而开始执行任务 0 的代码, 完成了从特权级 0 到特权级 3 代码的控制转移。

为了每隔 10 毫秒切换运行的任务, head.s 程序中把定时器芯片 8253 的通道 0 设置成每经过 10 毫秒就向中断控制芯片 8259A 发送一个时钟中断请求信号。PC 机的 ROM BIOS 开机时已经在 8259A 中把时钟中断请求信号设置成中断向量 8, 因此我们需要在中断 8 的处理过程中执行任务切换操作。任务切换的实现方法是查看 current 变量中当前运行任务号。如果 current 当前是 0, 就利用任务 1 的 TSS 选择符作为操作数执行远跳转指令, 从而切换到任务 1 中执行, 否则反之。

每个任务在执行时, 会首先把一个字符的 ASCII 码放入寄存器 AL 中, 然后调用系统中断调用 int 0x80, 而该系统调用处理过程则会调用一个简单的字符写屏子程序, 把寄存器 AL 中的字符显示在屏幕上, 同时把字符显示的屏幕的下一个位置记录下来, 作为下一次显示字符的屏幕位置。在显示过一个字符后, 任务代码会使用循环语句延迟一段时间, 然后又跳转到任务代码开始处继续循环执行, 直到运行了 10 毫秒而发生了定时中断, 从而代码会切换到另一个任务去运行。对于任务 A, 寄存器 AL 中将始终存放字符 'A', 而任务 B 运行时 AL 中始终存放字符 'B'。因此在程序运行时我们将看到一连串的字符 'A' 和一连串的字符 'A' 间隔地连续不断地显示在屏幕上, 见图 4-43 所示。



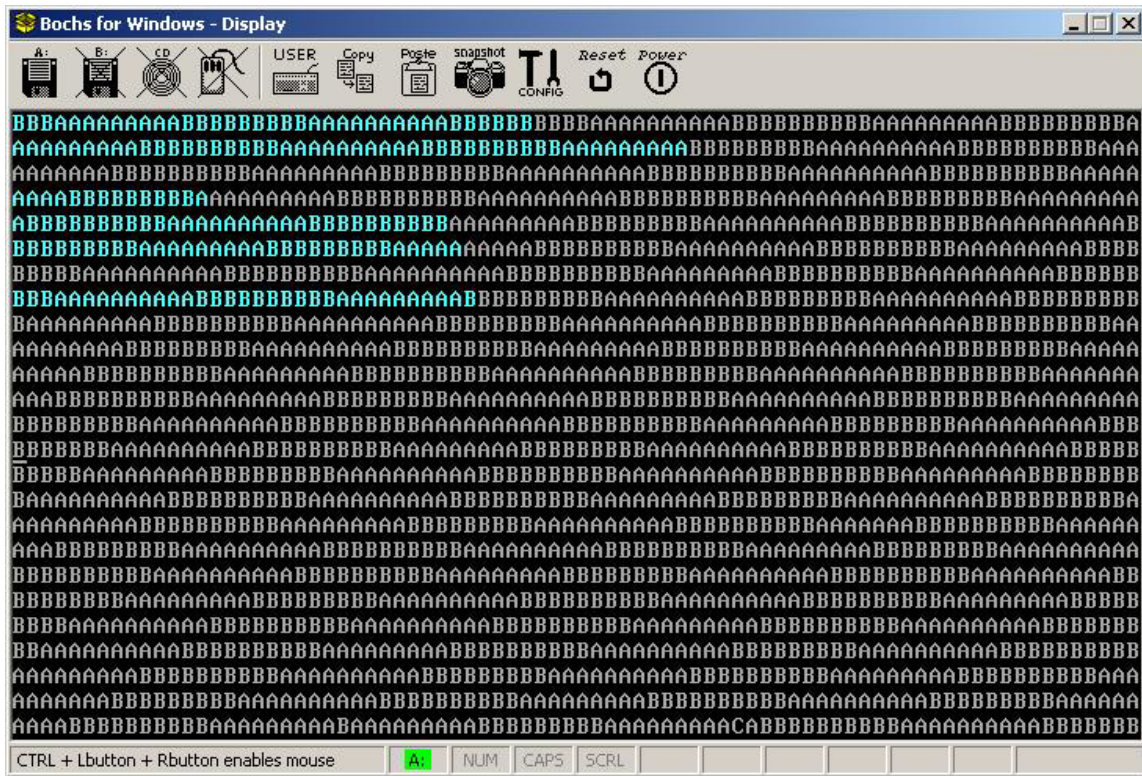


图 4-43 简单内核运行的屏幕显示情况

图 4-43 是我们在 Bochs 模拟软件中运行这个内核示例的屏幕显示情况。细心的读者会发现，在图中底端一行上显示出一个字符‘C’。这是由于 PC 机偶然产生了一个不是时钟中断和系统调用中断的其他中断。因为我们已经在程序中给所有其他中断安装了一个默认中断处理程序。当出现一个其他中断时，系统就会运行这个默认中断处理程序，于是就会在屏幕上显示一个字符‘C’，然后退出中断。

下面给出 boot.s 和 head.s 程序的详细注释。有关这个简单内核示例的编译和运行方法请参考本书最后一章中“编译运行简单内核示例程序”一节内容。

### 4.9.2 引导启动程序 boot.s

为了尽量让程序简单，这个引导启动扇区程序仅能够加载长度不超过 16 个扇区的 head 代码，并且直接使用了 ROM BIOS 默认设置的中断向量号，即定时中断请求处理的中断号仍然是 8。这与 Linux 系统中使用的不同。Linux 系统会在内核初始化时重新设置 8259A 中断控制芯片，并把时钟中断请求信号对应到中断 0x20 上，详细说明参见“内核引导启动程序”一章内容。

01 ! boot.s 程序

02 ! 首先利用 BIOS 中断把内核代码（head 代码）加载到内存 0x10000 处，然后移动到内存 0 处。

03 ! 最后进入保护模式，并跳转到内存 0（head 代码）开始处继续运行。

04 BOOTSEG = 0x07c0 ! 引导扇区（本程序）被 BIOS 加载到内存 0x7c00 处。

05 SYSSEG = 0x1000 ! 内核（head）先加载到 0x10000 处，然后移动到 0x0 处。

06 SYSLEN = 17 ! 内核占用的最大磁盘扇区数。

07 entry start

08 start:

09 jmp go, #BOOTSEG ! 段间跳转至 0x7c0:go 处。当本程序刚运行时所有段寄存器值

10 go: mov ax, cs ! 均为 0。该跳转语句会把 CS 寄存器加载为 0x7c0（原为 0）。

11 mov ds, ax ! 让 DS 和 SS 都指向 0x7c0 段。

```

12      mov     ss, ax
13      mov     sp, #0x400          ! 设置临时栈指针。其值需大于程序末端并有一定空间即可。
14
15 ! 加载内核代码到内存 0x10000 开始处。
16 load_system:
17      mov     dx, #0x0000          ! 利用 BIOS 中断 int 0x13 功能 2 从启动盘读取 head 代码。
18      mov     cx, #0x0002          ! DH - 磁头号; DL - 驱动器号; CH - 10 位磁道号低 8 位;
19      mov     ax, #SYSSEG          ! CL - 位 7、6 是磁道号高 2 位, 位 5-0 起始扇区号 (从 1 计)。
20      mov     es, ax              ! ES:BX - 读入缓冲区位置 (0x1000:0x0000)。
21      xor     bx, bx              ! AH - 读扇区功能号; AL - 需读的扇区数 (17)。
22      mov     ax, #0x200+SYSLEN
23      int     0x13
24      jnc     ok_load              ! 若没有发生错误则跳转继续运行, 否则死循环。
25 die:   jmp     die
26
27 ! 把内核代码移动到内存 0 开始处。共移动 8KB 字节 (内核长度不超过 8KB)。
28 ok_load:
29      cli                          ! 关中断。
30      mov     ax, #SYSSEG          ! 移动开始位置 DS:SI = 0x1000:0; 目的位置 ES:DI=0:0。
31      mov     ds, ax
32      xor     ax, ax
33      mov     es, ax
34      mov     cx, #0x1000          ! 设置共移动 4K 次, 每次移动一个字 (word)。
35      sub     si, si
36      sub     di, di
37      rep     movsw                ! 执行重复移动指令。
38 ! 加载 IDT 和 GDT 基地址寄存器 IDTR 和 GDTR。
39      mov     ax, #BOOTSEG
40      mov     ds, ax              ! 让 DS 重新指向 0x7c0 段。
41      lidt     idt_48              ! 加载 IDTR。6 字节操作数: 2 字节表长度, 4 字节线性基地址。
42      lgdt     gdt_48              ! 加载 GDTR。6 字节操作数: 2 字节表长度, 4 字节线性基地址。
43
44 ! 设置控制寄存器 CR0 (即机器状态字), 进入保护模式。段选择符值 8 对应 GDT 表中第 2 个段描述符。
45      mov     ax, #0x0001          ! 在 CR0 中设置保护模式标志 PE (位 0)。
46      lmsw     ax                  ! 然后跳转至段选择符值指定的段中, 偏移 0 处。
47      jmpi     0, 8                ! 注意此时段值已是段选择符。该段的线性基地址是 0。
48
49 ! 下面是全局描述符表 GDT 的内容。其中包含 3 个段描述符。第 1 个不用, 另 2 个是代码和数据段描述符。
50 gdt:   .word   0, 0, 0, 0          ! 段描述符 0, 不用。每个描述符项占 8 字节。
51
52       .word   0x07FF                ! 段描述符 1。8Mb - 段限长值=2047 (2048*4096=8MB)。
53       .word   0x0000                ! 段基地址=0x00000。
54       .word   0x9A00                ! 是代码段, 可读/执行。
55       .word   0x00C0                ! 段属性颗粒度=4KB, 80386。
56
57       .word   0x07FF                ! 段描述符 2。8Mb - 段限长值=2047 (2048*4096=8MB)。
58       .word   0x0000                ! 段基地址=0x00000。
59       .word   0x9200                ! 是数据段, 可读写。
60       .word   0x00C0                ! 段属性颗粒度=4KB, 80386。
61 ! 下面分别是 LIDT 和 LGDT 指令的 6 字节操作数。
62 idt_48: .word   0                  ! IDT 表长度是 0。
63       .word   0, 0                  ! IDT 表的线性基地址也是 0。
64 gdt_48: .word   0x7ff              ! GDT 表长度是 2048 字节, 可容纳 256 个描述符项。

```

---

```

65         .word    0x7c00+gdt, 0           ! GDT 表的线性基地址在 0x7c0 段的偏移 gdt 处。
66 .org 510
67         .word    0xAA55                 ! 引导扇区有效标志。必须处于引导扇区最后 2 字节处。

```

---

### 4.9.3 多任务内核程序 head.s

在进入保护模式后，head.s 程序重新建立和设置 IDT、GDT 表的主要原因是为了让程序在结构上比较清晰，也为了与后面 Linux 0.11 内核源代码中这两个表的设置方式保持一致。当然，就本程序来说我们完全可以直接使用 boot.s 中设置的 IDT 和 GDT 表位置，填入适当的描述符项即可。

---

```

01 # head.s 包含 32 位保护模式初始化设置代码、时钟中断代码、系统调用中断代码和两个任务的代码。
02 # 在初始化完成之后程序移动到任务 0 开始执行，并在时钟中断控制下进行任务 0 和 1 之间的切换操作。
03 LATCH          = 11930                  # 定时器初始计数值，即每隔 10 毫秒发送一次中断请求。
04 SCR_N_SEL      = 0x18                  # 屏幕显示内存段选择符。
05 TSS0_SEL       = 0x20                  # 任务 0 的 TSS 段选择符。
06 LDT0_SEL       = 0x28                  # 任务 0 的 LDT 段选择符。
07 TSS1_SEL       = 0x30                  # 任务 1 的 TSS 段选择符。
08 LDT1_SEL       = 0x38                  # 任务 1 的 LDT 段选择符。
09 .text
10 startup_32:
11 # 首先加载数据段寄存器 DS、堆栈段寄存器 SS 和堆栈指针 ESP。所有段的线性基地址都是 0。
12     movl $0x10, %eax                    # 0x10 是 GDT 中数据段选择符。
13     mov %ax, %ds
14     lss init_stack, %esp
15 # 在新的位置重新设置 IDT 和 GDT 表。
16     call setup_idt                      # 设置 IDT。先把 256 个中断门都填默认处理过程的描述符。
17     call setup_gdt                      # 设置 GDT。
18     movl $0x10, %eax                    # 在改变了 GDT 之后重新加载所有段寄存器。
19     mov %ax, %ds
20     mov %ax, %es
21     mov %ax, %fs
22     mov %ax, %gs
23     lss init_stack, %esp
24 # 设置 8253 定时芯片。把计数器通道 0 设置成每隔 10 毫秒向中断控制器发送一个中断请求信号。
25     movb $0x36, %al                     # 控制字：设置通道 0 工作在方式 3、计数初值采用二进制。
26     movl $0x43, %edx                     # 8253 芯片控制字寄存器写端口。
27     outb %al, %dx
28     movl $LATCH, %eax                    # 初始计数值设置为 LATCH (1193180/100)，即频率 100HZ。
29     movl $0x40, %edx                     # 通道 0 的端口。
30     outb %al, %dx                        # 分两次把初始计数值写入通道 0。
31     movb %ah, %al
32     outb %al, %dx
33 # 在 IDT 表第 8 和第 128 (0x80) 项处分别设置定时中断门描述符和系统调用陷阱门描述符。
34     movl $0x00080000, %eax               # 中断程序属内核，即 EAX 高字是内核代码段选择符 0x0008。
35     movw $timer_interrupt, %ax           # 设置定时中断门描述符。取定时中断处理程序地址。
36     movw $0x8E00, %dx                    # 中断门类型是 14 (屏蔽中断)，特权级 0 或硬件使用。
37     movl $0x08, %ecx                     # 开机时 BIOS 设置的时钟中断向量号 8。这里直接使用它。
38     lea idt(, %ecx, 8), %esi              # 把 IDT 描述符 0x08 地址放入 ESI 中，然后设置该描述符。
39     movl %eax, (%esi)
40     movl %edx, 4(%esi)
41     movw $system_interrupt, %ax          # 设置系统调用陷阱门描述符。取系统调用处理程序地址。

```

---

```

42      movw $0xef00, %dx                # 陷阱门类型是 15，特权级 3 的程序可执行。
43      movl $0x80, %ecx                # 系统调用向量号是 0x80。
44      lea idt(, %ecx, 8), %esi        # 把 IDT 描述符项 0x80 地址放入 ESI 中，然后设置该描述符。
45      movl %eax, (%esi)
46      movl %edx, 4(%esi)
47 # 好了，现在我们为移动到任务 0（任务 A）中执行来操作堆栈内容，在堆栈中人工建立中断返回时的场景。
48      pushfl                          # 复位标志寄存器 EFLAGS 中的嵌套任务标志。
49      andl $0xffffbfff, (%esp)
50      popfl
51      movl $TSS0_SEL, %eax            # 把任务 0 的 TSS 段选择符加载到任务寄存器 TR。
52      ltr %ax
53      movl $LDT0_SEL, %eax            # 把任务 0 的 LDT 段选择符加载到局部描述符表寄存器 LDTR。
54      lldt %ax                        # TR 和 LDTR 只需人工加载一次，以后 CPU 会自动处理。
55      movl $0, current                # 把当前任务号 0 保存在 current 变量中。
56      sti                             # 现在开启中断，并在栈中营造中断返回时的场景。
57      pushl $0x17                     # 把任务 0 当前局部空间数据段（堆栈段）选择符入栈。
58      pushl $init_stack                # 把堆栈指针入栈（也可以直接把 ESP 入栈）。
59      pushfl                          # 把标志寄存器值入栈。
60      pushl $0x0f                     # 把当前局部空间代码段选择符入栈。
61      pushl $task0                     # 把代码指针入栈。
62      iret                           # 执行中断返回指令，从而切换到特权级 3 的任务 0 中执行。
63
64 # 以下是设置 GDT 和 IDT 中描述符项的子程序。
65 setup_gdt:                          # 使用 6 字节操作数 lgdt_opcode 设置 GDT 表位置和长度。
66      lgdt lgdt_opcode
67      ret
    # 这段代码暂时设置 IDT 表中所有 256 个中断门描述符都为同一个默认值，均使用默认的中断处理过程
    # ignore_int。设置的具体方法是：首先在 eax 和 edx 寄存器对中分别设置好默认中断门描述符的 0-3
    # 字节和 4-7 字节的内容，然后利用该寄存器对循环往 IDT 表中填充默认中断门描述符内容。
68 setup_idt:                          # 把所有 256 个中断门描述符设置为使用默认处理过程。
69      lea ignore_int, %edx            # 设置方法与设置定时中断门描述符的方法一样。
70      movl $0x00080000, %eax          # 选择符为 0x0008。
71      movw %dx, %ax
72      movw $0x8E00, %dx                # 中断门类型，特权级为 0。
73      lea idt, %edi
74      mov $256, %ecx                  # 循环设置所有 256 个门描述符项。
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                # 最后用 6 字节操作数加载 IDTR 寄存器。
81      ret
82
83 # 显示字符子程序。取当前光标位置并把 AL 中的字符显示在屏幕上。整屏可显示 80 X 25 个字符。
84 write_char:
85      push %gs                        # 首先保存要用到的寄存器，EAX 由调用者负责保存。
86      pushl %ebx
87      mov $SCRN_SEL, %ebx             # 然后让 GS 指向显示内存段（0xb8000）。
88      mov %bx, %gs
89      movl scr_loc, %bx                # 再从变量 scr_loc 中取目前字符显示位置值。
90      shl $1, %ebx                    # 因为在屏幕上每个字符还有一个属性字节，因此字符
91      movb %al, %gs:(%ebx)            # 实际显示位置对应的显示内存偏移地址要乘 2。

```



```

92     shr $1, %ebx                # 把字符放到显示内存后把位置值除 2 加 1, 此时位置值对
93     incl %ebx                  # 应下一个显示位置。如果该位置大于 2000, 则复位成 0。
94     cmpl $2000, %ebx
95     jb 1f
96     movl $0, %ebx
97 1:    movl %ebx, scr_loc        # 最后把这个位置值保存起来 (scr_loc),
98     popl %ebx                  # 并弹出保存的寄存器内容, 返回。
99     pop %gs
100    ret
101
102 # 以下是 3 个中断处理程序: 默认中断、定时中断和系统调用中断。
103 # ignore_int 是默认的中断处理程序, 若系统产生了其他中断, 则会在屏幕上显示一个字符'C'。
104 .align 2
105 ignore_int:
106     push %ds
107     pushl %eax
108     movl $0x10, %eax           # 首先让 DS 指向内核数据段, 因为中断程序属于内核。
109     mov %ax, %ds
110     movl $67, %eax            # 在 AL 中存放字符'C'的代码, 调用显示程序显示在屏幕上。
111     call write_char
112     popl %eax
113     pop %ds
114     iret
115
116 # 这是定时中断处理程序。其中主要执行任务切换操作。
117 .align 2
118 timer_interrupt:
119     push %ds
120     pushl %eax
121     movl $0x10, %eax           # 首先让 DS 指向内核数据段。
122     mov %ax, %ds
123     movb $0x20, %al           # 然后立刻允许其他硬件中断, 即向 8259A 发送 EOI 命令。
124     outb %al, $0x20
125     movl $1, %eax             # 接着判断当前任务, 若是任务 1 则去执行任务 0, 或反之。
126     cmpl %eax, current
127     je 1f
128     movl %eax, current         # 若当前任务是 0, 则把 1 存入 current, 并跳转到任务 1
129     ljmp $TSS1_SEL, $0        # 去执行。注意跳转的偏移值无用, 但需要写上。
130     jmp 2f
131 1:    movl $0, current         # 若当前任务是 1, 则把 0 存入 current, 并跳转到任务 0
132     ljmp $TSS0_SEL, $0        # 去执行。
133 2:    popl %eax
134     pop %ds
135     iret
136
137 # 系统调用中断 int 0x80 处理程序。该示例只有一个显示字符功能。
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          # 首先让 DS 指向内核数据段。
146     mov %dx, %ds
147     call write_char          # 然后调用显示字符子程序 write_char, 显示 AL 中的字符。
148     popl %eax
149     popl %ebx
150     popl %ecx
151     popl %edx
152     pop %ds
153     iret
154
155     /*****/
156     current: .long 0          # 当前任务号 (0 或 1)。
157     scr_loc: .long 0          # 屏幕当前显示位置。按从左上角到右下角顺序显示。
158
159     .align 2
160     lidt_opcode:
161         .word 256*8-1         # 加载 IDTR 寄存器的 6 字节操作数: 表长度和基地址。
162         .long idt
163     lgdt_opcode:
164         .word (end_gdt-gdt)-1 # 加载 GDTR 寄存器的 6 字节操作数: 表长度和基地址。
165         .long gdt
166
167     .align 3
168     idt:     .fill 256, 8, 0   # IDT 空间。共 256 个门描述符, 每个 8 字节, 共占用 2KB。
169
170     gdt:     .quad 0x0000000000000000 # GDT 表。第 1 个描述符不用。
171             .quad 0x00c09a00000007ff # 第 2 个是内核代码段描述符。其选择符是 0x08。
172             .quad 0x00c09200000007ff # 第 3 个是内核数据段描述符。其选择符是 0x10。
173             .quad 0x00c0920b80000002 # 第 4 个是显示内存段描述符。其选择符是 0x18。
174             .word 0x68, tss0, 0xe900, 0x0 # 第 5 个是 TSS0 段的描述符。其选择符是 0x20
175             .word 0x40, ldt0, 0xe200, 0x0 # 第 6 个是 LDT0 段的描述符。其选择符是 0x28
176             .word 0x68, tss1, 0xe900, 0x0 # 第 7 个是 TSS1 段的描述符。其选择符是 0x30
177             .word 0x40, ldt1, 0xe200, 0x0 # 第 8 个是 LDT1 段的描述符。其选择符是 0x38
178     end_gdt:
179         .fill 128, 4, 0        # 初始内核堆栈空间。
180     init_stack:
181         .long init_stack       # 刚进入保护模式时用于加载 SS:ESP 堆栈指针值。
182         .word 0x10             # 堆栈段偏移位置。
183                                # 堆栈段同内核数据段。
184 # 下面是任务 0 的 LDT 表段中的局部段描述符。
185     .align 3
186     ldt0:    .quad 0x0000000000000000 # 第 1 个描述符, 不用。
187             .quad 0x00c0fa00000003ff # 第 2 个局部代码段描述符, 对应选择符是 0x0f。
188             .quad 0x00c0f200000003ff # 第 3 个局部数据段描述符, 对应选择符是 0x17。
189 # 下面是任务 0 的 TSS 段的内容。注意其中标号等字段在任务切换时不会改变。
190     tss0:    .long 0            /* back link */
191             .long krn_stk0, 0x10 /* esp0, ss0 */
192             .long 0, 0, 0, 0, 0 /* esp1, ssl, 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                # 这是任务 0 的内核栈空间。
199 krn_stk0:
200
201 # 下面是任务 1 的 LDT 表段内容和 TSS 段内容。
202 .align 3
203 ldt1:    .quad 0x0000000000000000    # 第 1 个描述符, 不用。
204         .quad 0x00c0fa000000003ff    # 选择符是 0x0f, 基地址 = 0x00000。
205         .quad 0x00c0f2000000003ff    # 选择符是 0x17, 基地址 = 0x00000。
206
207 tssl:    .long 0                    /* back link */
208         .long krn_stk1, 0x10         /* esp0, ss0 */
209         .long 0, 0, 0, 0, 0         /* esp1, ssl, 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                # 这是任务 1 的内核栈空间。其用户栈直接使用初始栈空间。
217 krn_stk1:
218
219 # 下面是任务 0 和任务 1 的程序, 它们分别循环显示字符'A'和'B'。
220 task0:
221     movl $0x17, %eax                # 首先让 DS 指向任务的局部数据段。
222     movw %ax, %ds                   # 因为任务没有使用局部数据, 所以这两句可省略。
223     movl $65, %al                   # 把需要显示的字符'A'放入 AL 寄存器中。
224     int $0x80                       # 执行系统调用, 显示字符。
225     movl $0xffff, %ecx              # 执行循环, 起延时作用。
226 1:    loop lb
227     jmp task0                       # 跳转到任务代码开始处继续显示字符。
228 task1:
229     movl $66, %al                   # 把需要显示的字符'B'放入 AL 寄存器中。
230     int $0x80                       # 执行系统调用, 显示字符。
231     movl $0xffff, %ecx              # 延时一段时间, 并跳转到开始处继续循环显示。
232 1:    loop lb
233     jmp task1
234
235         .fill 128,4,0                # 这是任务 1 的用户栈空间。
236 usr_stk1:

```

---

## 第5章 Linux 内核体系结构

本章首先概要介绍了 Linux 内核的编制模式和体系结构，然后详细描述了 Linux 内核源代码目录中组织形式以及子目录中各个代码文件的主要功能以及基本调用的层次关系。接下来就直接切入正题，从内核源文件 Linux/目录下的第一个文件 Makefile 开始，对每一行代码进行详细注释说明。本章内容可以看作是对内核源代码的总结概述，也可以作为阅读后续章节的参考信息。对于较难理解的地方可以先跳过，待阅读到后面相关内容时再返回来参考本章内容。在阅读本章之前请先复习或学习有关 80X86 保护模式运行方式工作原理。

一个完整可用的操作系统主要由 4 部分组成：硬件、操作系统内核、操作系统服务和用户应用程序，见图 5-1 所示。用户应用程序是指那些字处理程序、Internet 浏览器程序或用户自行编制的各种应用程序；操作系统服务程序是指那些向用户提供的服务被看作是操作系统部分功能的程序。在 Linux 操作系统上，这些程序包括 X 窗口系统、shell 命令解释系统以及那些内核编程接口等系统程序；操作系统内核程序即是本书所感兴趣的部分，它主要用于对硬件资源的抽象和访问调度。

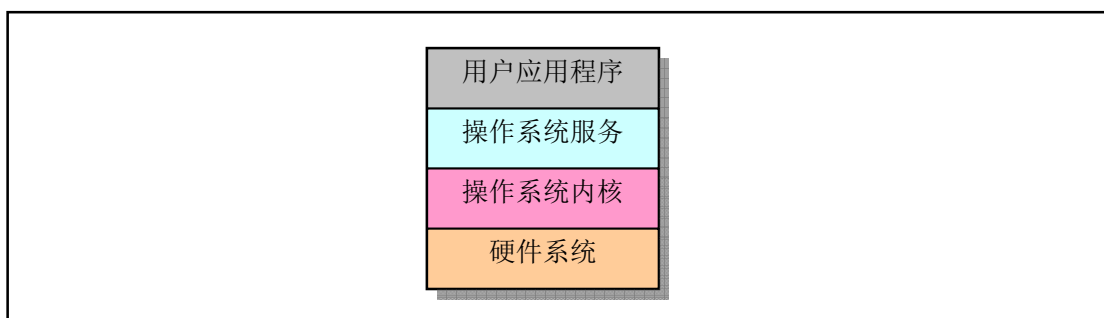


图 5-1 操作系统组成部分。

Linux 内核的主要用途就是为了与计算机硬件进行交互，实现对硬件部件的编程控制和接口操作，调度对硬件资源的访问，并为计算机上的用户程序提供一个高级的执行环境和对硬件的虚拟接口。在本章内容中，我们首先基于 Linux 0.11 版的的内核源代码，简明地描述 Linux 内核的基本体系结构、主要构成模块。然后对源代码中出现的几个重要数据结构进行说明。最后描述了构建 Linux 0.11 内核编译实验环境的方法。

### 5.1 Linux 内核模式

目前，操作系统内核的结构模式主要可分为整体式的单内核模式和层次式的微内核模式。而本书所注释的 Linux 0.11 内核，则是采用了单内核模式。单内核模式的主要优点是内核代码结构紧凑、执行速度快，不足之处主要是层次结构性不强。

在单内核模式的系统中，操作系统所提供服务的流程为：应用主程序使用指定的参数值执行系统调用指令(int x80)，使 CPU 从用户态（User Mode）切换到核心态（Kernel Model），然后操作系统根据具体的参数值调用特定的系统调用服务程序，而这些服务程序则根据需要再调用底层的一些支持函数以完成特定的功能。在完成了应用程序所要求的服务后，操作系统又使 CPU 从核心态切换回用户态，从而返回

到应用程序中继续执行后面的指令。因此概要地讲，单内核模式的内核也可粗略地分为三个层次：调用服务的主程序层、执行系统调用的服务层和支持系统调用的底层函数。见图 5-2 所示。

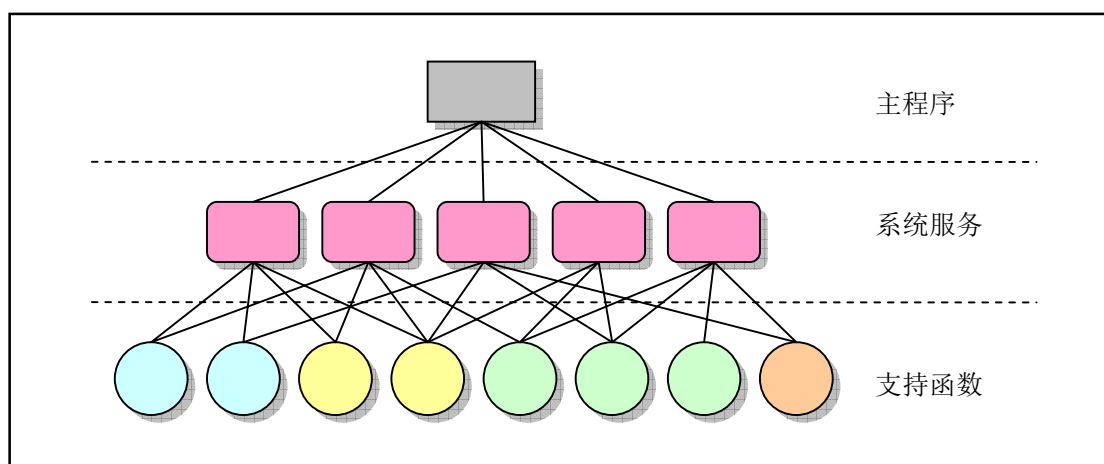


图 5-2 单内核模式的简单结构模型

## 5.2 Linux 内核系统体系结构

Linux 内核主要由 5 个模块构成，它们分别是：进程调度模块、内存管理模块、文件系统模块、进程间通信模块和网络接口模块。

进程调度模块用来负责控制进程对 CPU 资源的使用。所采取的调度策略是各进程能够公平合理地访问 CPU，同时保证内核能及时地执行硬件操作。内存管理模块用于确保所有进程能够安全地共享机器主内存区，同时，内存管理模块还支持虚拟内存管理方式，使得 Linux 支持进程使用比实际内存空间更多的内存容量。并可以利用文件系统把暂时不用的内存数据块交换到外部存储设备上去，当需要时再交换回来。文件系统模块用于支持对外部设备的驱动和存储。虚拟文件系统模块通过向所有的外部存储设备提供一个通用的文件接口，隐藏了各种硬件设备不同细节。从而提供并支持与其他操作系统兼容的多种文件系统格式。进程间通信模块子系统用于支持多种进程间的信息交换方式。网络接口模块提供对多种网络通信标准的访问并支持许多网络硬件。

这几个模块之间的依赖关系见图 5-3 所示。其中的连线代表它们之间的依赖关系，虚线和虚框部分表示 Linux 0.11 中还未实现的部分（从 Linux 0.95 版才开始逐步实现虚拟文件系统，而网络接口的支持到 0.96 版才有）。

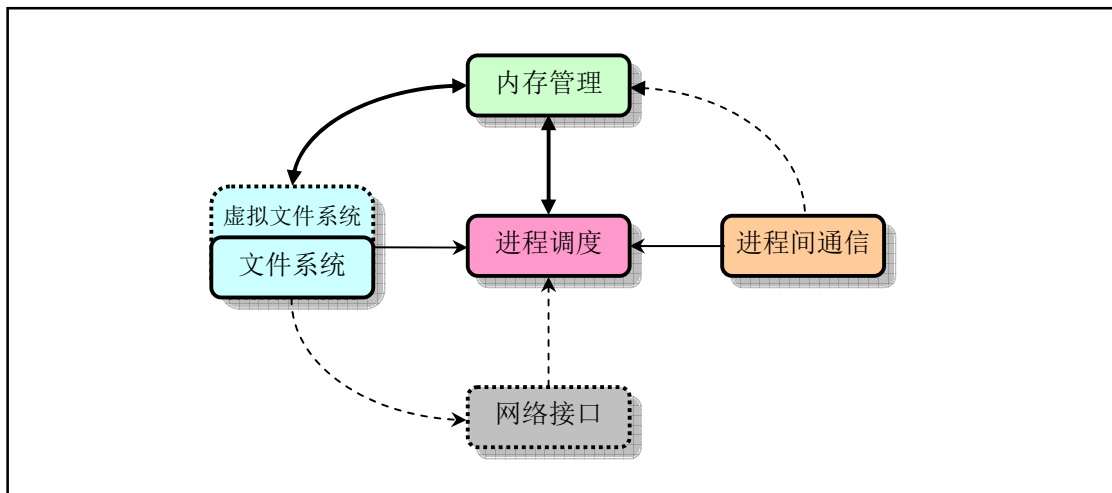


图 5-3 Linux 内核系统模块结构及相互依赖关系

由图可以看出，所有的模块都与进程调度模块存在依赖关系。因为它们都需要依靠进程调度程序来挂起（暂停）或重新运行它们的进程。通常，一个模块会在等待硬件操作期间被挂起，而在操作完成后才可继续运行。例如，当一个进程试图将一数据块写到软盘上去时，软盘驱动程序就可能在启动软盘旋转期间将该进程置为挂起等待状态，而在软盘进入到正常转速后再使得该进程能继续运行。另外 3 个模块也是由于类似的原因而与进程调度模块存在依赖关系。

其他几个模块的依赖关系有些不太明显，但同样也很重要。进程调度子系统需要使用内存管理来调整一特定进程所使用的物理内存空间。进程间通信子系统则需要依靠内存管理器来支持共享内存通信机制。这种通信机制允许两个进程访问内存的同一个区域以进行进程间信息的交换。虚拟文件系统也会使用网络接口来支持网络文件系统（NFS），同样也能使用内存管理子系统提供内存虚拟盘（ramdisk）设备。而内存管理子系统也会使用文件系统来支持内存数据块的交换操作。

若从单内核模式结构模型出发，我们还可以根据 Linux 0.11 内核源代码的结构将内核主要模块绘制成图 5-4 所示的框图结构。

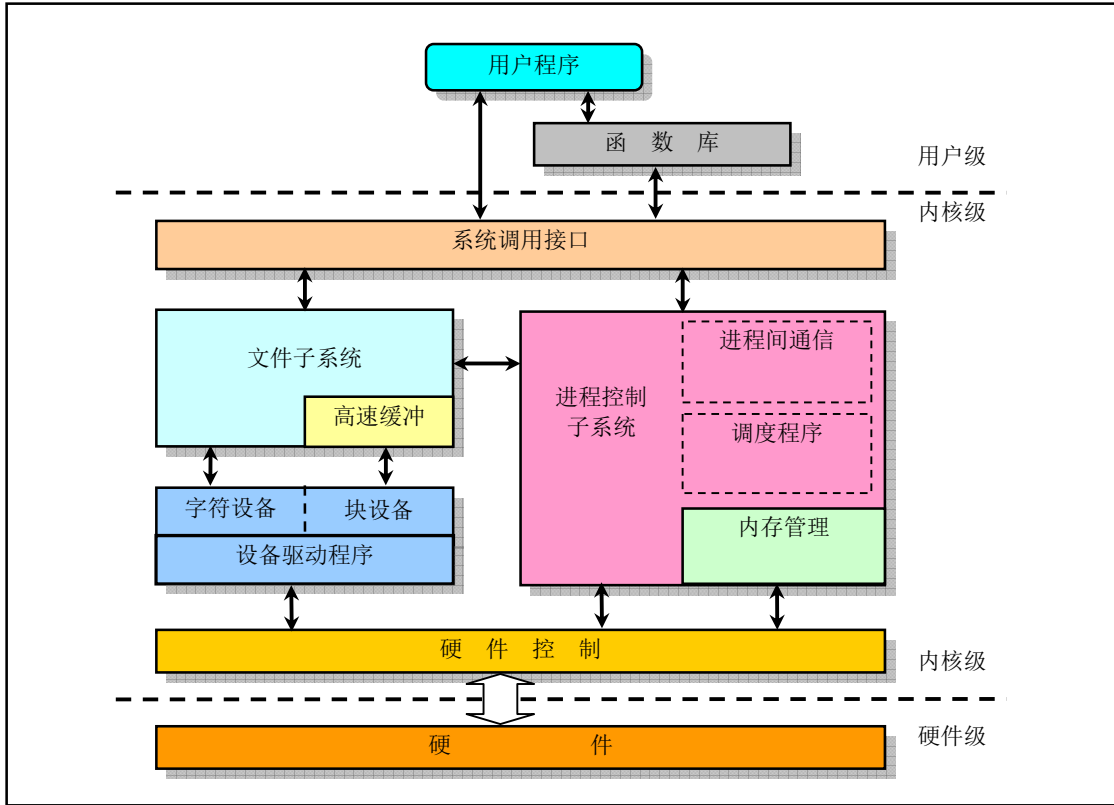


图 5-4 内核结构框图

其中内核级中的几个方框，除了硬件控制方框以外，其他粗线方框分别对应内核源代码的目录组织结构。

除了这些图中已经给出的依赖关系以外，所有这些模块还会依赖于内核中的通用资源。这些资源包括内核所有子系统都会调用的内存分配和收回函数、打印警告或出错信息函数以及一些系统调试函数。

## 5.3 Linux 内核对内存的管理和使用

本节首先说明 Linux 0.11 系统中比较直观的物理内存使用情况，然后结合 Linux 0.11 内核中的应用情况，再分别概要描述内存的分段和分页管理机制以及 CPU 多任务操作和保护方式。最后我们再综合说明 Linux 0.11 系统中内核代码和数据以及各个任务的代码和数据在虚拟地址、线性地址和物理地址之间的对应关系。

### 5.3.1 物理内存

在 Linux 0.11 内核中，为了有效地使用机器中的物理内存，在系统初始化阶段内存被划分成几个功能区域，见图 5-5 所示。

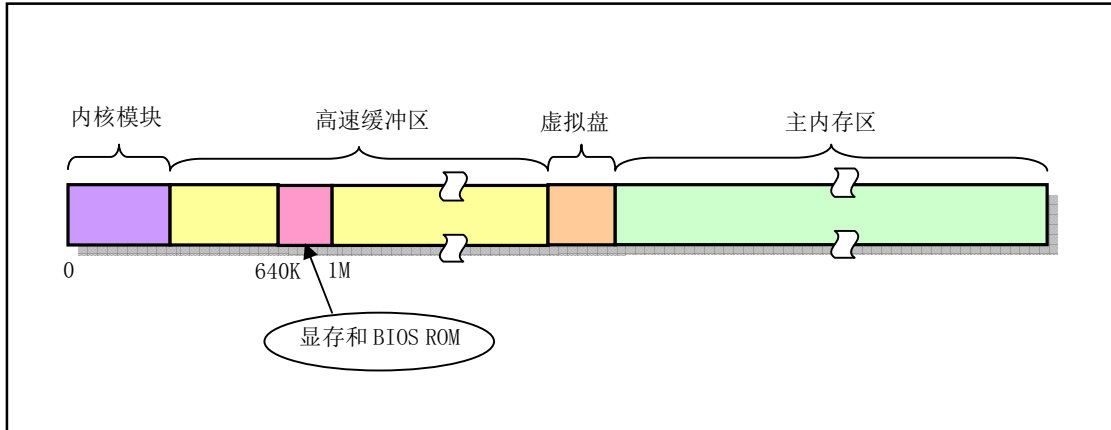


图 5-5 物理内存使用的功能分布图

其中，Linux 内核程序占据在物理内存的开始部分，接下来是供硬盘或软盘等块设备使用的高速缓冲区部分（其中要扣除显示卡内存和 ROM BIOS 所占用的内存地址范围 640K--1MB）。当一个进程需要读取块设备中的数据时，系统会首先把数据读到高速缓冲区中；当有数据需要写到块设备上时，系统也是先将数据放到高速缓冲区中，然后由块设备驱动程序写到相应的设备上。内存的最后部分是供所有程序可以随时申请和使用的主内存区。内核程序在使用主内存区时，也同样首先要向内核内存管理模块提出申请，并在申请成功后方能使用。对于含有 RAM 虚拟盘的系统，主内存区头部还要划去一部分，供虚拟盘存放数据。

由于计算机系统中所含的实际物理内存容量有限，因此 CPU 中通常都提供了内存管理机制对系统中的内存进行有效的管理。在 Intel 80386 及以后的 CPU 中提供了两种内存管理（地址变换）系统：内存分段系统（Segmentation System）和分页系统（Paging System）。其中分页管理系统是可选择的，由系统程序员通过编程来确定是否采用。为了能有效地使用物理内存，Linux 系统同时采用了内存分段和分页管理机制。

### 5.3.2 内存地址空间概念

Linux 0.11 内核中，在进行地址映射操作时，我们需要首先分清 3 种地址以及它们之间的变换概念：

a. 程序（进程）的虚拟和逻辑地址；b. CPU 的线性地址；c. 实际物理内存地址。

虚拟地址（Virtual Address）是指由程序产生的由段选择符和段内偏移地址两个部分组成的地址。因为这两部分组成的地址并没有直接用来访问物理内存，而是需要通过分段地址变换机制处理或映射后才对应到物理内存地址上，因此这种地址被称为虚拟地址。虚拟地址空间由 GDT 映射的全局地址空间和由 LDT 映射的局部地址空间组成。选择符的索引部分由 13 个比特位表示，加上区分 GDT 和 LDT 的 1 个比特位，因此 Intel 80X86 CPU 共可以索引 16384 个选择符。若每个段的长度都取最大值 4G，则最大虚拟地址空间范围是  $16384 * 4G = 64T$ 。

逻辑地址（Logical Address）是指由程序产生的与段相关的偏移地址部分。在 Intel 保护模式下即是指程序执行代码段限长内的偏移地址（假定代码段、数据段完全一样）。应用程序员仅需与逻辑地址打交道，而分段和分页机制对他来说是完全透明的，仅由系统编程人员涉及。不过有些资料并不区分逻辑地址和虚拟地址的概念，而是将它们统称为逻辑地址。

线性地址（Linear Address）是虚拟地址到物理地址变换之间的中间层，是处理器可寻址的内存空间（称为线性地址空间）中的地址。程序代码会产生逻辑地址，或者说是段中的偏移地址，加上相应段的基地址就生成了一个线性地址。如果启用了分页机制，那么线性地址可以再经变换以产生一个物理地址。若没有启用分页机制，那么线性地址直接就是物理地址。Intel 80386 的线性地址空间容量为 4G。



物理地址（Physical Address）是指出现在 CPU 外部地址总线上的寻址物理内存的地址信号，是地址变换的最终结果地址。如果启用了分页机制，那么线性地址会使用页目录和页表中的项变换成物理地址。如果没有启用分页机制，那么线性地址就直接成为物理地址了。

虚拟存储（或虚拟内存）（Virtual Memory）是指计算机呈现出要比实际拥有的内存大得多的内存量。因此它允许程序员编制并运行比实际系统拥有的内存大得多的程序。这使得许多大型项目也能够具有有限内存资源的系统上实现。一个很恰当的比喻是：你不需要很长的轨道就可以让一列火车从上海开到北京。你只需要足够长的铁轨（比如说 3 公里）就可以完成这个任务。采取的方法是把后面的铁轨立刻铺到火车的前面，只要你的操作足够快并能满足要求，列车就能象在一条完整的轨道上运行。这也就是虚拟内存管理需要完成的任务。在 Linux 0.11 内核中，给每个程序（进程）都划分了总容量为 64MB 的虚拟内存空间。因此程序的逻辑地址范围是 0x0000000 到 0x4000000。

如上所述，有时我们也把逻辑地址称为虚拟地址。因为逻辑地址与虚拟内存空间的概念类似，并且也是与实际物理内存容量无关。

### 5.3.3 内存分段机制

在内存分段系统中，一个程序的逻辑地址通过分段机制自动地映射（变换）到中间层的 4GB（ $2^{32}$ ）线性地址空间中。程序每次对内存的引用都是对内存段中内存的引用。当程序引用一个内存地址时，通过把相应的段基址加到程序员看得见的逻辑地址上就形成了一个对应的线性地址。此时若没有启用分页机制，则该线性地址就被送到 CPU 的外部地址总线上，用于直接寻址对应的物理内存。见图 4-4 所示。

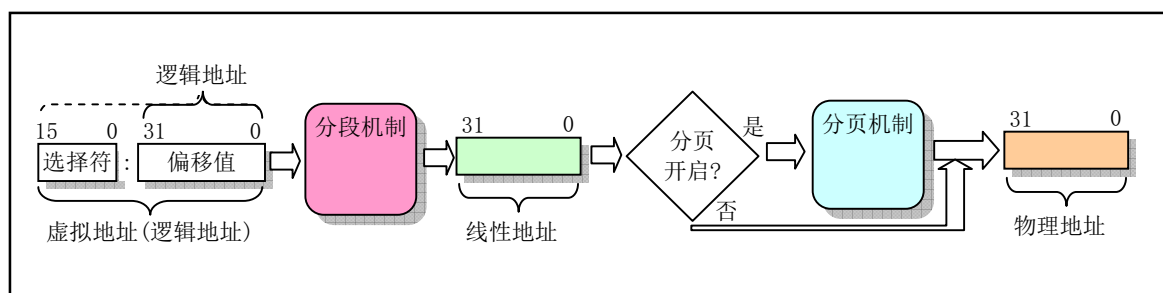


图 5-6 虚拟地址（逻辑地址）到物理地址的变换过程

CPU 进行地址变换（映射）的主要目的是为了解决虚拟内存空间到物理内存空间的映射问题。虚拟内存空间的含义是指一种利用二级或外部存储空间，使程序能不受实际物理内存量限制而使用内存的一种方法。通常虚拟内存空间要比实际物理内存量大得多。

那么虚拟存储管理是怎样实现的呢？原理与上述列车运行的比喻类似。首先，当一个程序需要使用一块不存在的内存时（也即在内存页表中已标出相应内存页面不在内存中），CPU 就需要一种方法来得知这个情况。这是通过 80386 的页错误异常中断来实现的。当一个进程引用一个不存在页面中的内存地址时，就会触发 CPU 产生页出错异常中断，并把引起中断的线性地址放到 CR2 控制寄存器中。因此处理该中断的过程就可以知道发生页异常的确切地址，从而可以把进程要求的页面从二级存储空间（比如硬盘上）加载到物理内存中。如果此时物理内存已经被全部占用，那么可以借助二级存储空间的一部分作为交换缓冲区（Swapper）把内存中暂时不使用的页面交换到二级缓冲区中，然后把要求的页面调入内存中。这也就是内存管理的缺页加载机制，在 Linux 0.11 内核中是在程序 mm/memory.c 中实现。

Intel CPU 使用段（Segment）的概念来对程序进行寻址。每个段定义了内存中的某个区域以及访问的优先级等信息。假定大家知晓实模式下内存寻址原理，现在我们根据 CPU 在实模式和保护模式下寻址方式的不同，用比较的方法来简单说明 32 位保护模式运行机制下内存寻址的主要特点。

在实模式下，寻址一个内存地址主要是使用段和偏移值，段值被存放在段寄存器中（例如 `ds`），并

且段的长度被固定为 64KB。段内偏移地址存放在任意一个可用于寻址的寄存器中（例如 si）。因此，根据段寄存器和偏移寄存器中的值，就可以算出实际指向的内存地址，见图 5-7 (a)所示。

而在保护模式运行方式下，段寄存器中存放的不再是被寻址段的基地址，而是一个段描述符表（Segment Descriptor Table）中某一描述符项在表中的索引值。索引值指定的段描述符项中含有需要寻址的内存段的基地址、段的长度值和段的访问特权级别等信息。寻址的内存位置是由该段描述符项中指定的段基地址值与一个段内偏移值组合而成。段的长度可变，由描述符中的内容指定。可见，和实模式下的寻址相比，段寄存器值换成了段描述符表中相应段描述符的索引值以及段表选择位和特权级，称为段选择符（Segment Selector），但偏移值还是使用了原实模式下的概念。这样，在保护模式下寻址一个内存地址就需要比实模式下多一道手续，也即需要使用段描述符表。这是由于在保护模式下访问一个内存段需要的信息比较多，而一个 16 位的段寄存器放不下这么多内容。示意图见图 5-7 (b)所示。注意，如果你不在一个段描述符中定义一个内存线性地址空间区域，那么该地址区域就完全不能被寻址，CPU 将拒绝访问该地址区域。

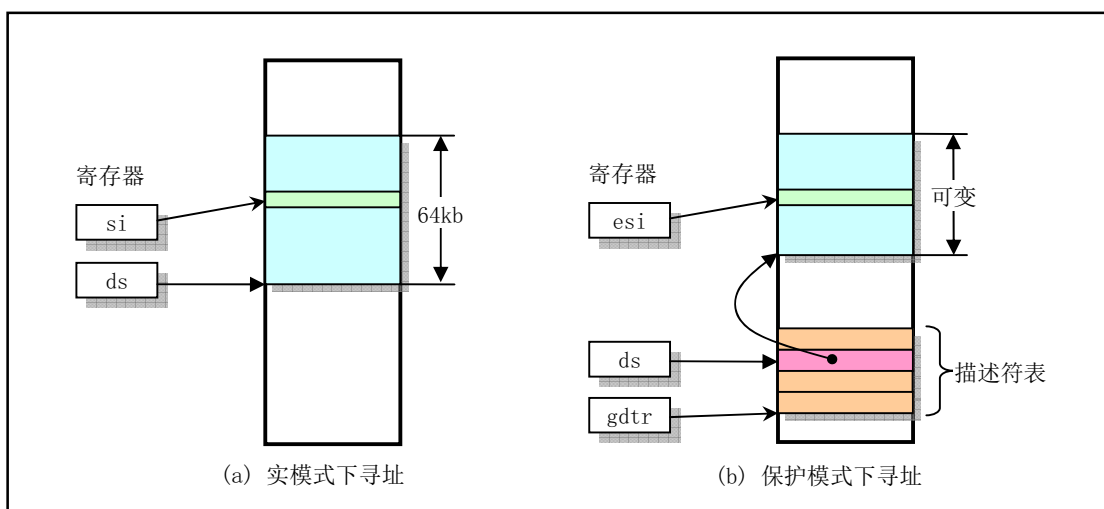


图 5-7 实模式与保护模式下寻址方式的比较

每个描述符占用 8 个字节，其中含有所描述段在线性地址空间中的起始地址（基址）、段的长度、段的类型（例如代码段和数据段）、段的特权级别和其他一些信息。一个段可以定义的最大长度是 4GB。

保存描述符的描述符表有 3 种类型，每种用于不同目的。全局描述符表 GDT（Global Descriptor Table）是主要的基本描述符表，该表可被所有程序用于引用访问一个内存段。中断描述符表 IDT（Interrupt Descriptor Table）保存有定义中断或异常处理过程的段描述符。IDT 表直接替代了 8086 系统中的中断向量表。为了能在 80X86 保护模式下正常运行，我们必须为 CPU 定义一个 GDT 表和一个 IDT 表。最后一种类型的表是局部描述符表 LDT（Local Descriptor Table）。该表应用于多任务系统中，通常每个任务使用一个 LDT 表。作为对 GDT 表的扩充，每个 LDT 表为对应任务提供了更多的可用描述符项，因而也为每个任务提供了可寻址内存空间的范围。这些表可以保存在线性地址空间的任何地方。为了让 CPU 能定位 GDT 表、IDT 表和当前的 LDT 表，需要为 CPU 分别设置 GDTR、IDTR 和 LDTR 三个特殊寄存器。这些寄存器中将存储对应表的 32 位线性基地址和表的限长字节值。表限长值是表的长度值-1。

当 CPU 要寻址一个段时，就会使用 16 位的段寄存器中的选择符来定位一个段描述符。在 80X86 CPU 中，段寄存器中的值右移 3 位即是描述符表中一个描述符的索引值。13 位的索引值最多可定位 8192（0-8191）个的描述符项。选择符中位 2（TI）用来指定使用哪个表。若该位是 0 则选择符指定的是 GDT 表中的描述符，否则是 LDT 表中的描述符。

每个程序都可能有若干个内存段组成。程序的逻辑地址（或称为虚拟地址）即是用于寻址这些段和段

中具体地址位置。在 Linux 0.11 中，程序逻辑地址到线性地址的变换过程使用了 CPU 的全局段描述符表 GDT 和局部段描述符表 LDT。由 GDT 映射的地址空间称为全局地址空间，由 LDT 映射的地址空间则称为局部地址空间，而这两者构成了虚拟地址的空间。具体的使用方式见图 5-8 所示。

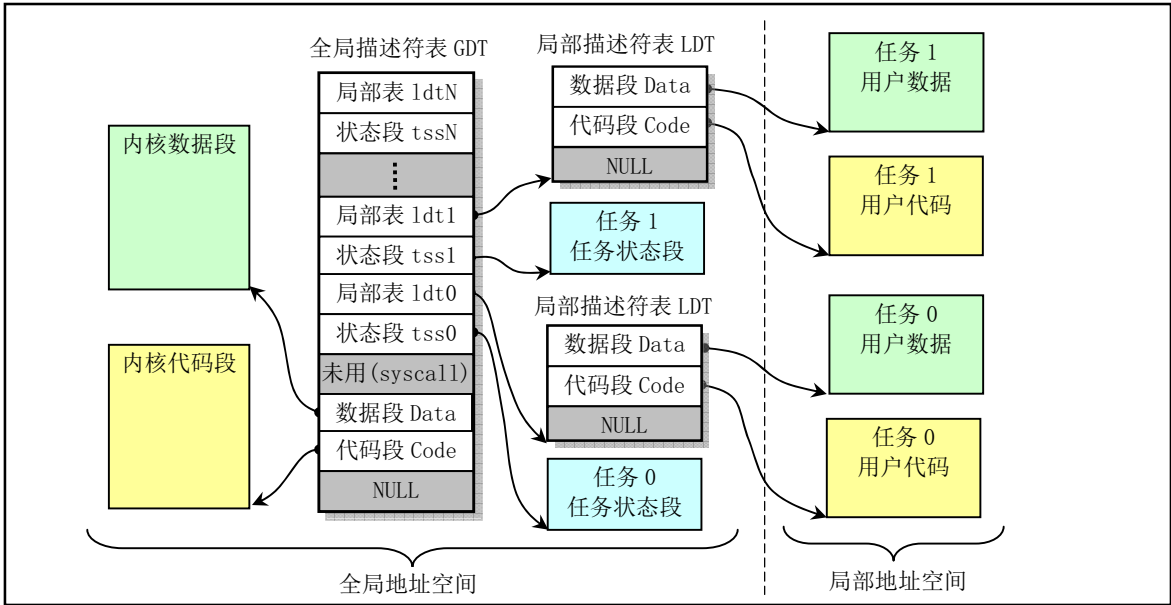


图 5-8 Linux 系统中虚拟地址空间分配图

图中画出了具有两个任务时的情况。可以看出，每个任务的局部描述符表 LDT 本身也是由 GDT 中描述符定义的一个内存段，在该段中存放着对应任务的代码段和数据段描述符，因此 LDT 段很短，其段限长通常只要大于 24 字节即可。同样，每个任务的任务状态段 TSS 也是由 GDT 中描述符定义的一个内存段，其段限长也只要满足能够存放一个 TSS 数据结构就够了。

对于中断描述符表 idt，它保存在内核代码段中。由于在 Linux 0.11 内核中，内核和各任务的代码段和数据段都分别被映射到线性地址空间中相同基址处，且段限长也一样，因此内核的代码段和数据段是重叠的，各任务的代码段和数据段分别也是重叠的，参见图 5-10 或图 5-11 所示。任务状态段 TSS (Task State Segment) 用于在任务切换时 CPU 自动保存或恢复相关任务的当前执行上下文 (CPU 当前状态)。例如对于切换出的任务，CPU 就把其寄存器等信息保存在该任务的 TSS 段中，同时 CPU 使用新切换进任务的 TSS 段中的信息来设置各寄存器，以恢复该任务的执行环境，参见图 4-37 所示。在 Linux 0.11 中，每个任务的 TSS 段内容被保存在该任务的任务数据结构中。另外，Linux 0.11 内核中没有使用到 GDT 表中第 4 个描述符 (图中 syscall 描述符项)。从 include/linux/sched.h 文件中第 150 行上的原英文注释 (如下所示) 可以猜想到，Linus 当时设计内核时曾经想把系统调用的代码放在这个专门独立的段中。

```

149 /*
150  * Entry into gdt where to find first TSS. 0-nul, 1-cs, 2-ds, 3-syscall
151  * 4-TSS0, 5-LDT0, 6-TSS1 etc ...
152 */

```

### 5.3.4 内存分页管理

若采用了分页机制，则此时线性地址只是一个中间结果，还需要使用分页机制进行变换，再最终映射到实际物理内存地址上。与分段机制类似，分页机制允许我们重新定向 (变换) 每次内存引用，以适应我们的特殊要求。使用分页机制最普遍的场所是当系统内存实际上被分成很多凌乱的块时，它可以建立

一个大而连续的内存空间映像，好让程序不用操心和管理这些分散的内存块。分页机制增强了分段机制的性能。另外，页地址变换建立在段变换基础之上，任何分页机制的保护措施并不会取代段变换的保护措施而只是进行更进一步的检查操作。

内存分页管理机制的基本原理是将 CPU 整个线性内存区域划分成 4096 字节为 1 页的内存页面。程序申请使用内存时，系统就以内存页为单位进行分配。内存分页机制的实现方式与分段机制很相似，但并不如分段机制那么完善。因为分页机制是在分段机制之上实现的，所以其结果是对系统内存具有非常灵活的控制权，并且在分段机制的内存保护上更增加了分页保护机制。为了在 80X86 保护模式下使用分页机制，需要把控制寄存器 CR0 的最高比特位（位 31）置位。

在使用这种内存分页管理方法时，每个执行中的进程（任务）可以使用比实际内存容量大得多的连续地址空间。为了在使用分页机制的条件下把线性地址映射到容量相对很小的物理内存空间上，80386 使用了页目录表和页表。页目录表项与页表项格式基本相同，都占用 4 个字节，并且每个页目录表或页表必须只能包含 1024 个页表项。因此一个页目录表或一个页表分别共占用 1 页内存。页目录项和页表项的小区别在于页表项有个已写位 D（Dirty），而页目录项则没有。

线性地址到物理地址的变换过程见图 5-9 所示。图中控制寄存器 CR3 保存着是当前页目录表在物理内存中的基地址（因此 CR3 也被称为页目录基地址寄存器 PDBR）。32 位的线性地址被分成三个部分，分别用来在页目录表和页表中定位对应的页目录项和页表项以及在对应的物理内存页面中指定页面内的偏移位置。因为 1 个页表可有 1024 项，因此一个页表最多可以映射  $1024 * 4KB = 4MB$  内存；又因为一个页目录表最多有 1024 项，对应 1024 个二级页表，因此一个页目录表最多可以映射  $1024 * 4MB = 4GB$  容量的内存。即一个页目录表就可以映射整个线性地址空间范围。

由于 Linux 0.1x 系统中内核和所有任务都共用同一个页目录表，使得任何时刻处理器线性地址空间到物理地址空间的映射函数都一样。因此为了让内核和所有任务都不互相重叠和干扰，它们都必须从虚拟地址空间映射到线性地址空间的不同位置，即占用不同的线性地址空间范围。

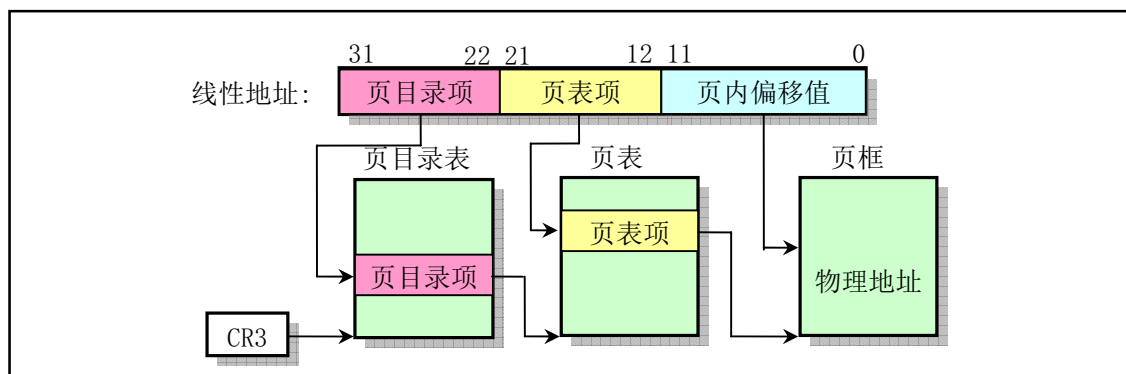


图 5-9 线性地址到物理地址的变换示意图

对于 Intel 80386 系统，其 CPU 可以提供多达 4G 的线性地址空间。一个任务的虚拟地址需要首先通过其局部段描述符变换为 CPU 整个线性地址空间中的地址，然后再使用页目录表 PDT（一级页表）和页表 PT（二级页表）映射到实际物理地址页上。为了使用实际物理内存，每个进程的线性地址通过二级内存页表动态地映射到主内存区域的不同物理内存页上。由于 Linux 0.11 中把每个进程最大可用虚拟内存空间定义为 64MB，因此每个进程的逻辑地址通过加上(任务号)\*64MB，即可转换为线性空间中的地址。不过在注释中，在不至于搞混的情况下我们有时将进程中的此类地址简单地称为逻辑地址或线性地址。

对于 Linux 0.11 系统，内核设置全局描述符表 GDT 中的段描述符项数最大为 256，其中 2 项空闲、



2 项系统使用，每个进程使用两项。因此，此时系统可以最多容纳  $(256-4)/2 = 126$  个任务，并且虚拟地址范围是  $((256-4)/2) * 64\text{MB}$  约等于 8G。但 0.11 内核中人工定义最大任务数  $\text{NR\_TASKS} = 64$  个，每个任务逻辑地址范围是 64M，并且各个任务在线性地址空间中的起始位置是 (任务号)\*64MB。因此全部任务所使用的线性地址空间范围是  $64\text{MB} * 64 = 4\text{G}$ ，见图 5-10 所示。图中示出了当系统具有 4 个任务时的情况。内核代码段和数据段被映射到线性地址空间的开始 16MB 部分，并且代码和数据段都映射到同一个区域，完全互相重叠。而第 1 个任务（任务 0）是由内核“人工”启动运行的，其代码和数据包含在内核代码和数据中，因此该任务所占用的线性地址空间范围比较特殊。任务 0 的代码段和数据段的长度是从线性地址 0 开始的 640KB 范围，其代码和数据段也完全重叠，并且与内核代码段和数据段有重叠的部分。实际上，Linux 0.11 中所有任务的指令空间 I（Instruction）和数据空间 D（Data）都合用一块内存，即一个进程的所有代码、数据和堆栈部分都处于同一内存段中，也即是 I&D 不分离的一种使用方式。

任务 1 的线性地址空间范围也只有从 64MB 开始的 640KB 长度。它们之间的详细对应关系见后面说明。任务 2 和任务 3 分别被映射线性地址 128MB 和 192MB 开始的地方，并且它们的逻辑地址范围均是 64MB。由于 4G 地址空间范围正好是 CPU 的线性地址空间范围和可寻址的最大物理地址空间范围，而且在把任务 0 和任务 1 的逻辑地址范围看作 64MB 时，系统中同时可有任务的逻辑地址范围总和也是 4GB，因此在 0.11 内核中比较容易混淆三种地址概念。

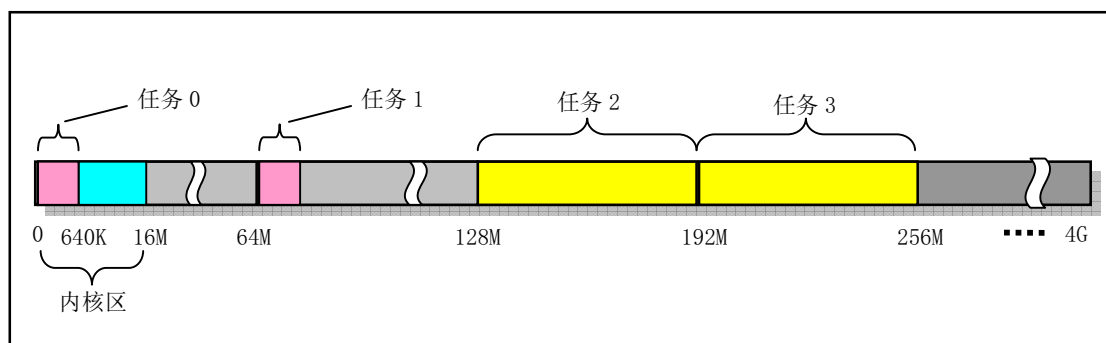


图 5-10 Linux 0.11 线性地址空间的使用示意图

如果也按照线性空间中任务的排列顺序排列虚拟空间中的任务，那么我们可以有图 5-11 所示的系统同时可拥有所有任务在虚拟地址空间中的示意图，所占用虚拟空间范围也是 4GB。其中没有考虑内核代码和数据在虚拟空间中所占用的范围。另外，在图中对于进程 2 和进程 3 还分别给出了各自逻辑空间中代码段和数据段（包括数据和堆栈内容）的位置示意图。

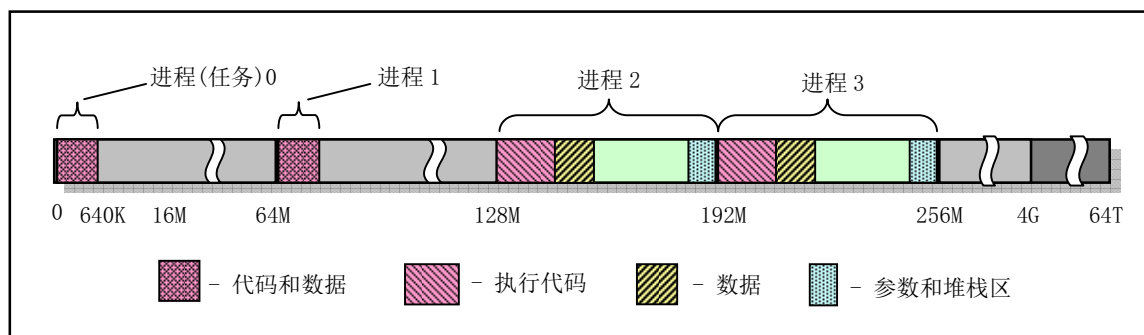


图 5-11 Linux 0.11 系统任务在虚拟空间中顺序排列所占的空间范围

请还需注意，进程逻辑地址空间中代码段（Code Section）和数据段（Data Section）的概念与 CPU

分段机制中的代码段和数据段不是同一个概念。CPU 分段机制中段的概念确定了在线性地址空间中一个段的用途以及被执行或访问的约束和限制，每个段可以设置在 4GB 线性地址空间中的任何地方，它们可以相互独立也可以完全重叠或部分重叠。而进程在其逻辑地址空间中的代码段和数据段则是指由编译器在编译程序和操作系统在加载程序时规定的在进程逻辑空间中顺序排列的代码区域、初始化和未初始化的数据区域以及堆栈区域。进程逻辑地址空间中代码段和数据段等结构形式见图所示。有关逻辑地址空间的说明请参见内存管理一章内容。

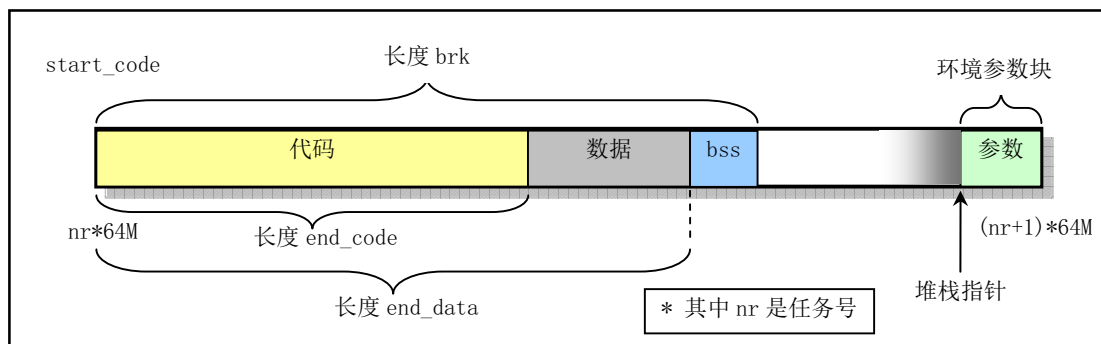


图 5-12 进程代码和数据在其逻辑地址空间中的分布

### 5.3.5 CPU 多任务和保护方式

Intel 80X86 CPU 共分 4 个保护级，0 级具有最高优先级，而 3 级优先级最低。Linux 0.11 操作系统使用了 CPU 的 0 和 3 两个保护级。内核代码本身会由系统中的所有任务共享。而每个任务则都有自己的代码和数据区，这两个区域保存于局部地址空间，因此系统中的其他任务是看不见的（不能访问的）。而内核代码和数据是由所有任务共享的，因此它保存在全局地址空间中。图 5-13 给出了这种结构的示意图。图中同心圆代表 CPU 的保护级别（保护层），这里仅使用了 CPU 的 0 级和 3 级。而径向射线则用来区分系统中的各个任务。每条径向射线指出了各任务的边界。除了每个任务虚拟地址空间的全局地址区域，任务 1 中的地址与任务 2 中相同地址处是无关的。

当一个任务（进程）执行系统调用而陷入内核代码中执行时，我们就称进程处于内核运行态（或简称为内核态）。此时处理器处于特权级最高的（0 级）内核代码中执行。当进程处于内核态时，执行的内核代码会使用当前进程的内核栈。每个进程都有自己的内核栈。当进程在执行用户自己的代码时，则称其处于用户运行态（用户态）。即此时处理器在特权级最低的（3 级）用户代码中运行。当正在执行用户程序而突然被中断程序中断时，此时用户程序也可以象征性地称为处于进程的内核态。因为中断处理程序将使用当前进程的内核栈。这与处于内核态的进程的状态有些类似。进程的内核态和用户态将在后面有关进程运行状态一节中作更详细的说明。

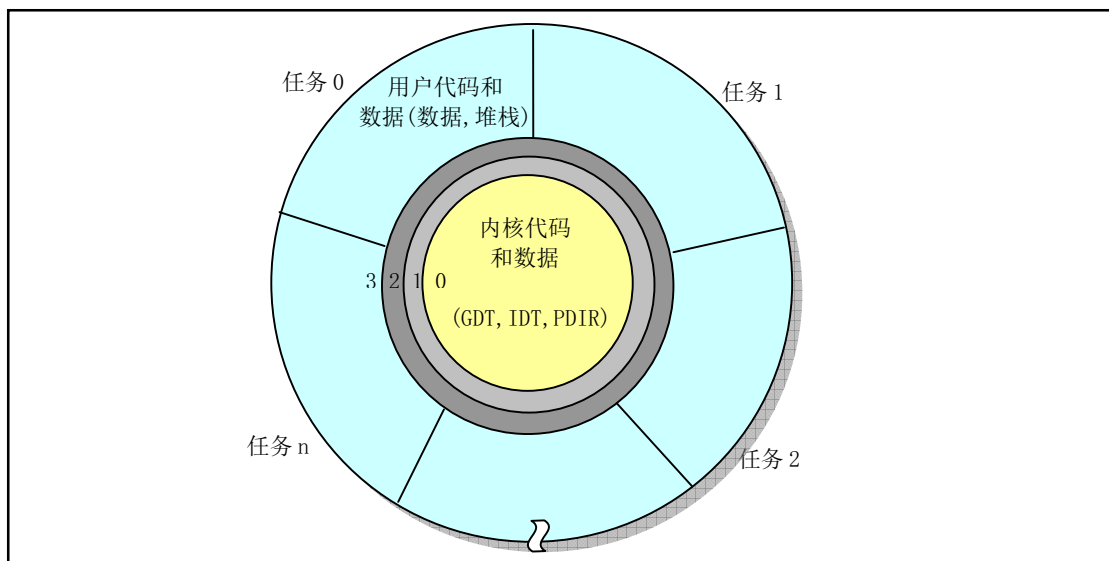


图 5-13 多任务系统

### 5.3.6 虚拟地址、线性地址和物理地址之间的关系

前面我们根据内存分段和分页机制详细说明了 CPU 的内存管理方式。现在我们以 Linux 0.11 系统为例，详细说明内核代码和数据以及各任务的代码和数据在虚拟地址空间、线性地址空间和物理地址空间中的对应关系。由于任务 0 和任务 1 的生成或创建过程比较特殊，我们将对它们分别进行描述。

#### 内核代码和数据的地址

对于 Linux 0.11 内核代码和数据来说，在 head.s 程序的初始化操作中已经把内核代码段和数据段都设置成为长度为 16MB 的段。在线性地址空间中这两个段的范围重叠，都是从线性地址 0 开始到地址 0xFFFFFFFF 共 16MB 地址范围。在该范围中含有内核所有的代码、内核段表（GDT、IDT、TSS）、页目录表和内核的二级页表、内核局部数据以及内核临时堆栈（将被用作第 1 个任务即任务 0 的用户堆栈）。其页目录表和二级页表已设置成把 0--16MB 的线性地址空间一一对应到物理地址上，占用了 4 个目录项，即 4 个二级页表。因此对于内核代码或数据的地址来说，我们可以直接把它们看作是物理内存中的地址。此时内核的虚拟地址空间、线性地址空间和物理地址空间三者之间的关系可用图 5-14 来表示。



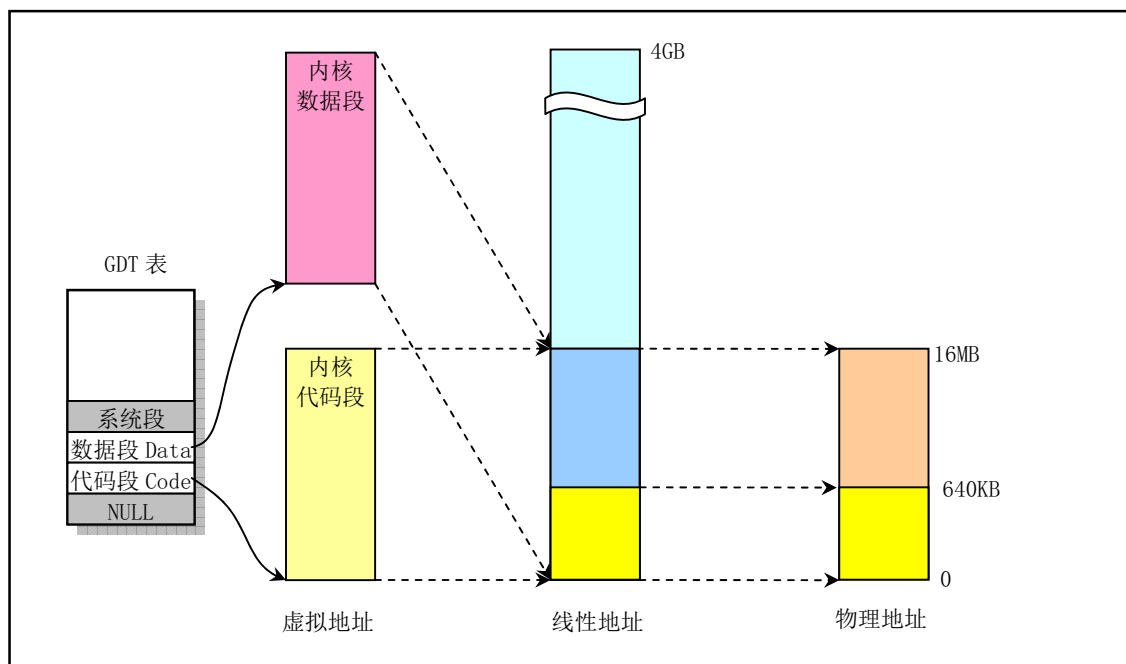


图 5-14 内核代码和数据段在三种地址空间中的关系

因此，默认情况下 Linux 0.11 内核最多可管理 16MB 的物理内存，共有 4096 个物理页面（页帧），每个页面 4KB。通过上述分析可以看出：①内核代码段和数据段区域在线性地址空间和物理地址空间中是一样的。这样设置可以大大简化内核的初始化操作。②GDT 和 IDT 在内核数据段中，因此它们的线性地址也同样等于它们的物理地址。在实模式下的 `setup.s` 程序初始化操作中，我们曾经设置过临时的 GDT 和 IDT，这是进入保护模式之前必须设置的。由于这两个表当时处于物理内存大约 0x90200 处，而进入保护模式后内核系统模块处于物理内存 0 开始位置，并且 0x90200 处的空间将被挪作他用（用于高速缓冲），因此在进入保护模式后，在运行的第 1 个程序 `head.s` 中我们需要重新设置这两个表。即设置 GDTR 和 IDTR 指向新的 GDT 和 IDT，描述符也需要重新加载。但由于开启分页机制时这两个表的位置没有变动，因此无须再重新建立或移动表位置。③除任务 0 以外，所有其他任务所需要的物理内存页面与线性地址中的不同或部分不同，因此内核需要动态地在主内存区中为它们作映射操作，动态地建立页目录项和页表项。虽然任务 1 的代码和数据也在内核中，但由于他需要另行分配获得内存，因此也需要自己的映射表项。

虽然 Linux 0.11 默认可管理 16MB 物理内存，但是系统中并不是一定要有这些物理内存。机器中只要有 4MB（甚至 2MB）物理内存就完全可以运行 Linux 0.11 系统了。若机器只有 4MB 物理内存，那么此时内核 4MB--16MB 地址范围就会映射到不存在的物理内存地址上。但这并不妨碍系统的运行。因为在初始化时内核内存管理程序会知道机器中所含物理内存量的确切大小，因而不会让 CPU 分页机制把线性地址页面映射到不存在的 4MB--16MB 中去。内核中这样的默认设置主要是为了便于系统物理内存的扩展，实际并不会用到不存在的物理内存区域。如果系统有多于 16MB 的物理内存，由于在 `init/main.c` 程序中初始化时限制了对 16MB 以上内存的使用，并且这里内核也仅映射了 0--16MB 的内存范围，因此在 16MB 之上的物理内存将不会用到。

通过在这里为内核增加一些页表，并且对 `init/main.c` 程序稍作修改，我们可以对此限制进行扩展。例如在系统中有 32MB 物理内存的情况下，我们就需要为内核代码和数据段建立 8 个二级页表项来把 32MB 的线性地址范围映射到物理内存上。

## 任务 0 的地址对应关系

任务 0 是系统中一个人工启动的第一个任务。它的代码段和数据段长度被设置为 640KB。该任务的代码和数据直接包含在内核代码和数据中，是从线性地址 0 开始的 640KB 内容，因此可以它直接使用内核代码已经设置好的页目录和页表进行分页地址变换。同样，它的代码和数据段在线性地址空间中也是重叠的。对应的任务状态段 TSS0 也是手工预设置好的，并且位于任务 0 数据结构信息中，参见 sched.h 第 113 行开始的数据。TSS0 段位于内核 sched.c 程序的代码中，长度为 104 字节，具体位置可参见图 5-23 中“任务 0 结构信息”一项所示。三个地址空间中的映射对应关系见图 5-15 所示。

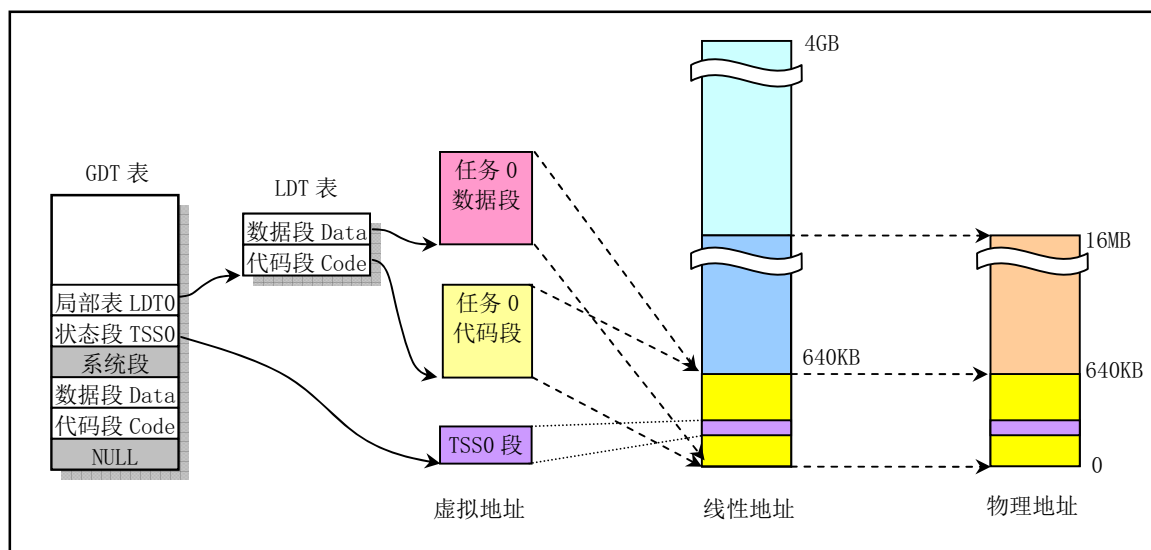


图 5-15 任务 0 在三个地址空间中的相互关系

由于任务 0 直接被包含在内核代码中，因此不需要为其再另外分配内存页。它运行时所需要的内核态堆栈和用户态堆栈空间也都在内核代码区中，并且由于在内核初始化时（head.s）这些内核页面在页表项中的属性都已经被设置成了 0b111，即对应页面用户可读写并且存在，因此用户堆栈 user\_stack[] 空间虽然在内核空间中，但任务 0 仍然能对其进行读写操作。

## 任务 1 的地址对应关系

与任务 0 类似，任务 1 也是一个特殊的任务。它的代码也在内核代码区域中。与任务 0 不同的是在线性地址空间中，系统在使用 fork() 创建任务 1（init 进程）时为存放任务 1 的二级页表而在主内存区申请了一页内存来存放，并复制了父进程（任务 0）的页目录和二级页表项。因此任务 1 有自己的页目录和页表表项，它把任务 1 占用的线性空间范围 64MB--128MB（实际上是 64MB--64MB+640KB）也同样映射到了物理地址 0--640KB 处。此时任务 1 的长度也是 640KB，并且其代码段和数据段相重叠，只占用一个页目录项和一个二级页表。另外，系统还会为任务 1 在主内存区域中申请一页内存用来存放它的任务数据结构和用作任务 1 的内核堆栈空间。任务数据结构（也称进程控制块 PCB）信息中包括任务 1 的 TSS 段结构信息。见图 5-16 所示。

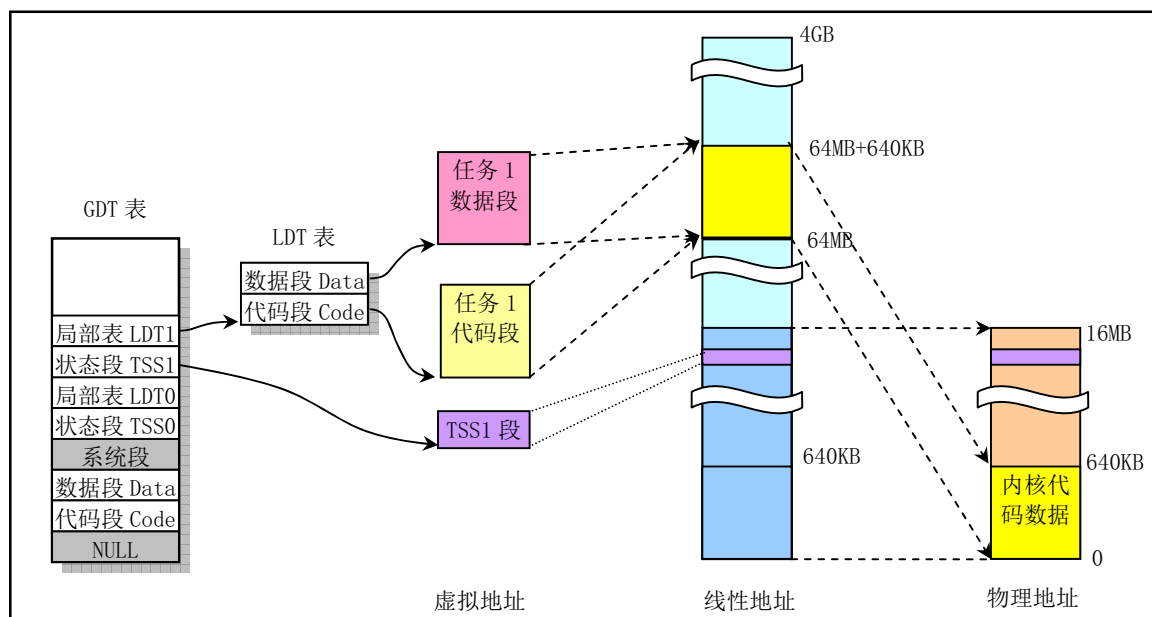


图 5-16 任务 1 在三种地址空间中的关系

任务 1 的用户态堆栈空间将直接共享使用处于内核代码和数据区域（线性地址 0--640KB）中任务 0 的用户态堆栈空间 `user_stack[]`（参见 `kernel/sched.c`，第 67--72 行），因此这个堆栈需要在任务 1 实际使用之前保持“干净”，以确保被复制用于任务 1 的堆栈不含有无用数据。在刚开始创建任务 1 时，任务 0 的用户态堆栈 `user_stack[]` 与任务 1 共享使用，但当任务 1 开始运行时，由于任务 1 映射到 `user_stack[]` 处的页表项被设置成只读，使得任务 1 在执行堆栈操作时将会引起写页面异常，从而由内核另行分配主内存区页面作为堆栈空间使用。

### 其他任务的地址对应关系

对于被创建的从任务 2 开始的其他任务，它们的父进程都是 `init`（任务 1）进程。我们已经知道，在 Linux 0.11 系统中共可以有 64 个进程同时存在。下面我们以任务 2 为例来说明其他任何任务对地址空间的使用情况。

从任务 2 开始，如果任务号以 `nr` 来表示，那么任务 `nr` 在线性地址空间中的起始位置将被设定在  $nr * 64MB$  处。例如任务 2 的开始位置  $= nr * 64MB = 2 * 64MB = 128MB$ 。任务代码段和数据段的最大长度被设置为 64MB，因此任务 2 占有的线性地址空间范围是 128MB--192MB，共占用  $64MB / 4MB = 16$  个页目录项。虚拟空间中任务代码段和数据段都被映射到线性地址空间相同的范围，因此它们也完全重叠。图 5-17 显示出了任务 2 的代码段和数据段在三种地址空间中的对应关系。

在任务 2 被创建出来之后，将在其中运行 `execve()` 函数来执行 `shell` 程序。当内核通过复制任务 1 刚创建任务 2 时，除了占用线性地址空间范围不同外（128MB--128MB+640KB），此时任务 2 的代码和数据在三种地址空间中的关系与任务 1 的类似。当任务 2 的代码（`init()`）调用 `execve()` 系统调用开始加载并执行 `shell` 程序时，该系统调用会释放掉从任务 1 复制的页目录和页表表项及相应内存页面，然后为新的执行程序 `shell` 重新设置相关页目录和页表表项。图 5-17 给出的是任务 2 中开始执行 `shell` 程序时的情况，即任务 2 原先复制任务 1 的代码和数据被 `shell` 程序的代码段和数据段替换后的情况。图中显示出已经映射了一页物理内存页面的情况。这里请注意，在执行 `execve()` 函数时，系统虽然在线性地址空间为任务 2 分配了 64MB 的空间范围，但是内核并不会立刻为其分配和映射物理内存页面。只有当任务 2 开始执行时由于发生缺页而引起异常时才会由内存管理程序为其在主内存区中分配并映射一页物理内存到其线性地址空间中。这种分配和映射物理内存页面的方法称为需求加载（Load on demand）。参见内存管

理一章中的相关描述。

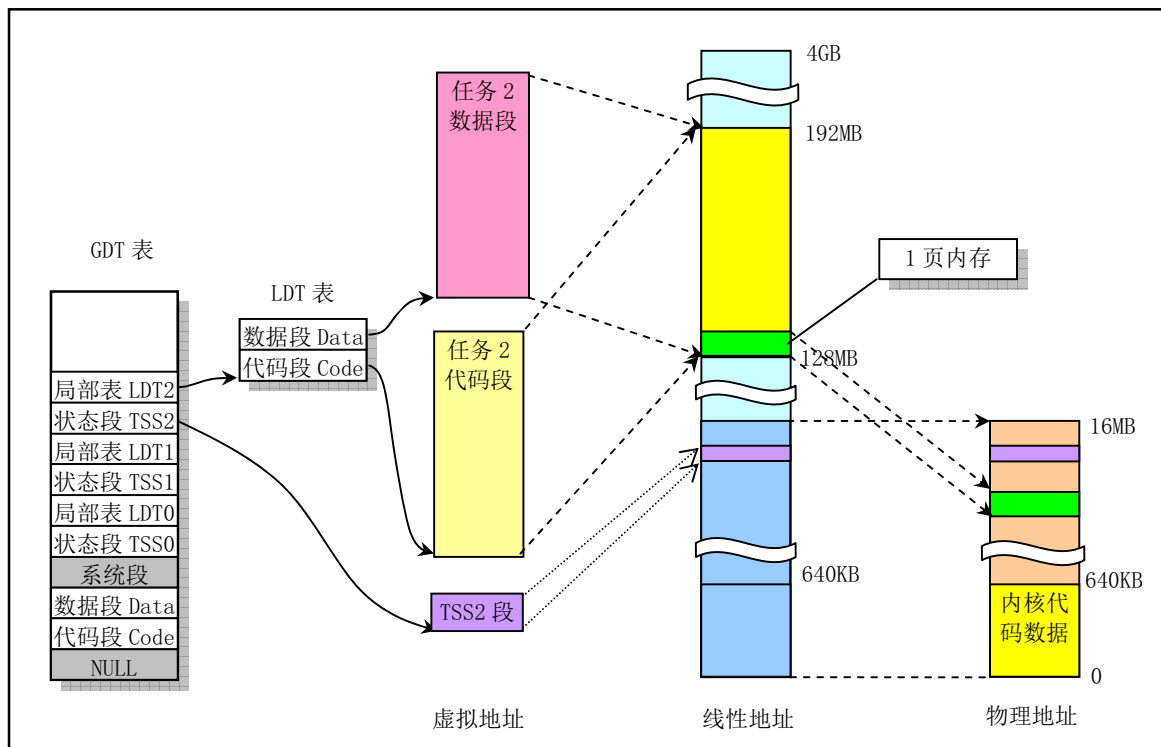


图 5-17 其他任务地址空间中的对应关系

从 Linux 内核 0.99 版以后，对内存空间的使用方式发生了变化。每个进程可以单独享用整个 4G 的地址空间范围。如果我们能理解本节说描述的内存管理概念，那么对于现在所使用的 Linux 2.x 内核中所使用的内存管理原理也能立刻明白。由于篇幅所限，这里对此不再说明。

### 5.3.7 用户申请内存的动态分配

当用户应用程序使用 C 函数库中的内存分配函数 `malloc()` 申请内存时，这些动态申请的内存容量或大小均由高层次的 C 库函数 `malloc()` 来进行管理，内核本身并不会插手管理。因为内核已经为每个进程（除了任务 0 和 1，它们与内核代码一起常驻内存中）在 CPU 的 4G 线性地址空间中分配了 64MB 的空间，所以只要进程执行时寻址的范围在它的 64MB 范围内，内核也同样会通过内存缺页管理机制自动为寻址对应的页面分配物理内存页面并进行映射操作。但是内核会为进程使用的代码和数据空间维护一个当前位置值 `brk`，这个值保存在每个进程的数据结构中。它指出了进程代码和数据（包括动态分配的数据空间）在进程地址空间中的末端位置。当 `malloc()` 函数为程序分配内存时，它会通过系统调用 `brk()` 把程序要求新增的空间长度通知内核，内核代码从而可以根据 `malloc()` 所提供的信息来更新 `brk` 的值，但并此时并不为新申请的空间映射物理内存页面。只有当程序寻址到某个不存在对应物理页面的地址时，内核才会进行相关物理内存页面的映射操作。

若进程代码寻址的某个数据所在的页面不存在，并且该页面所处位置属于进程堆范围，即不属于其执行文件映像文件对应的内存范围中，那么 CPU 就会产生一个缺页异常，并在异常处理程序中为指定的页面分配并映射一页物理内存页面。至于用户程序此次申请内存的字节长度数量和在对物理页面中的具体位置，则均由 C 库中内存分配函数 `malloc()` 负责管理。内核以页面为单位分配和映射物理内存，该函数则具体记录用户程序使用了一页内存的多少字节。剩余的容量将保留给程序再申请内存时使用。

当用户使用内存释放函数 `free()` 动态释放已申请的内存块时，C 库中的内存管理函数就会把所释放的

内存块标记为空闲，以备程序再次申请内存时使用。在这个过程中内核为该进程所分配的这个物理页面并不会被释放掉。只有当进程最终结束时内核才会全面收回已分配和映射到该进程地址空间范围的所有物理内存页面。

有关库函数 `malloc()` 和 `free()` 的具体代码实现请参见内核库中的 `lib/malloc.c` 程序。

## 5.4 Linux 系统的中断机制

本节介绍中断机制基本原理和相关的可编程控制器硬件逻辑以及 Linux 系统中使用中断的方法。有关可编程控制器的具体编程方法请参见下一章 `setup.s` 程序后的说明。

### 5.4.1 中断操作原理

微型计算机系统通常包括输入输出设备。处理器向这些设备提供服务的一种方法是使用轮询方式。在这种方法中处理器顺序地查询系统中的每个设备，“询问”它们是否需要服务。这种方法的优点是软件编程简单，但缺点是太耗处理器资源，影响系统性能。向设备提供服务的另一种方法是在设备需要服务时自己向处理器提出请求。处理器也只有在设备提出请求时才为其提供服务。

当设备向处理器提出服务请求时，处理器会在执行完当前的一条指令后立刻应答设备的请求，并转而执行该设备的相关服务程序。当服务程序执行完成后，处理器会接着去做刚才被中断的程序。这种处理方式就叫做中断（Interrupt）方法，而设备向处理器发出的服务请求则称为中断请求（IRQ - Interrupt Request）。处理器响应请求而执行的设备相关程序则被称为中断服务程序或中断服务过程（ISR - Interrupt Service Routine）。

可编程中断控制器（PIC - Programmable Interrupt Controller）是微机系统中管理设备中断请求的管理者。它通过连接到设备的中断请求引脚接受设备发出的终端服务请求信号。当设备激活其中断请求 IRQ 信号时，PIC 立刻会检测到。在同时收到几个设备的中断服务请求的情况下，PIC 会对它们进行优先级比较并选出最高优先级的中断请求进行处理。如果此时处理器正在执行一个设备的中断服务过程，那么 PIC 还需要把选出的中断请求与正在处理的中断请求的优先级进行比较，并基于该比较结果来确定是否向处理器发出一个中断信号。当 PIC 向处理器的 INT 引脚发出一个中断信号时，处理器会立刻停下当时所做的事情并询问 PIC 需要执行哪个中断服务请求。PIC 则通过向数据总线发送出与中断请求对应的中断号来告知处理器要执行哪个中断服务过程。处理器则根据读取的中断号通过查询中断向量表（或 32 位保护模式下的中断描述符表）取得相关设备的中断向量（即中断服务程序的地址）并开始执行中断服务程序。当中断服务程序执行结束，处理器就继续执行被中断信号打断的程序。

以上描述的是输入输出设备的中断服务处理过程。但是中断方法并非一定与硬件相关，它也可以用于软件中。通过使用 `int` 指令并使用其操作数指明中断号，就可以让处理器去执行相应的中断处理过程。PC/AT 系列微机共提供了对 256 个中断的支持，其中大部分都用于软件中断或异常，异常是处理器在处理过程中检测到错误而产生的中断操作。只有下面提及的一些中断被用于设备上。

### 5.4.2 80X86 微机的中断子系统

在使用 80X86 组成的微机系统中采用了 8259A 可编程中断控制器芯片。每个 8259A 芯片可以管理 8 个中断源。通过多片级联方式，8259A 能构成最多管理 64 个中断向量的系统。在 PC/AT 系列兼容机中，使用了两片 8259A 芯片，共可管理 15 级中断向量。其级连示意图见图 5-18 所示。其中从芯片的 INT 引脚连接到主芯片的 IR2 引脚上，即 8259A 从芯片发出的中断信号将作为 8259A 主芯片的 IRQ2 输入信号。主 8259A 芯片的端口基地址是 0x20，从芯片是 0xA0。IRQ9 引脚的作用与 PC/XT 的 IRQ2 相同，即 PC/AT 机利用硬件电路把使用 IRQ2 的设备的 IRQ2 引脚重新定向到了 PIC 的 IRQ9 引脚上，并利用 BIOS 中的软件把 IRQ9 的中断 `int 71` 重新定向到了 IRQ2 的中断 `int 0x0A` 的中断处理过程。这样一来可使得任何使用 IRQ2 的 PC/XT 的 8 位设配卡在 PC/AT 机下面仍然能正常使用。做到了 PC 机系列的向下



兼容性。

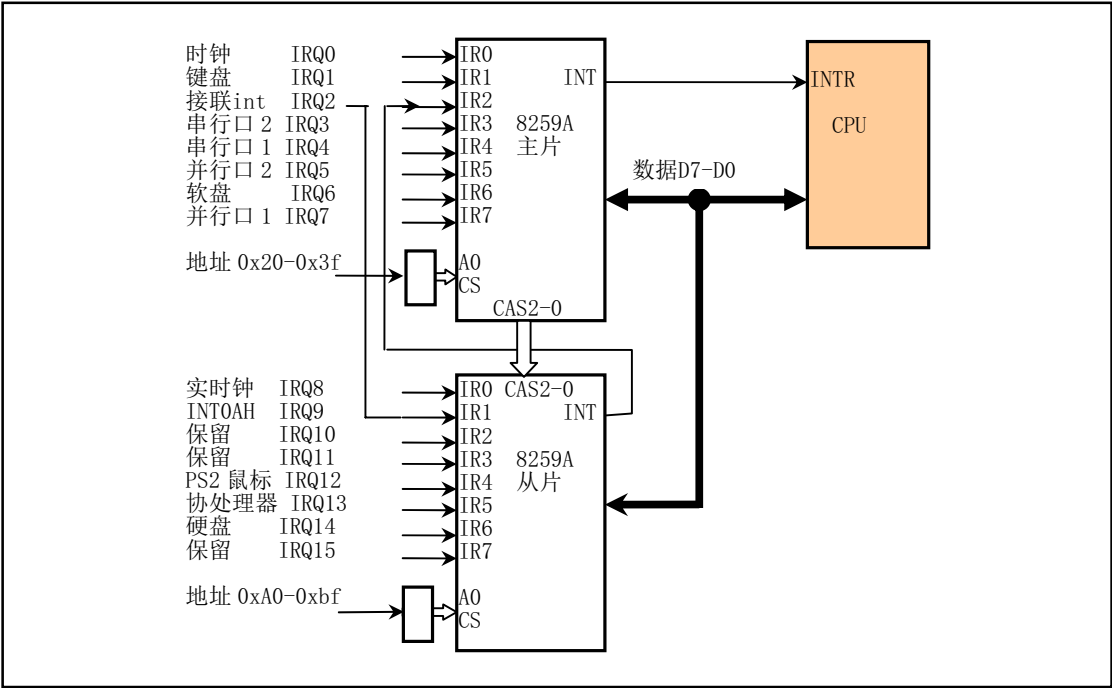


图 5-18 PC/AT 微机级连式 8259 控制系统

在总线控制器控制下，8259A 芯片可以处于编程状态和操作状态。编程状态是 CPU 使用 IN 或 OUT 指令对 8259A 芯片进行初始化编程的状态。一旦完成了初始化编程，芯片即进入操作状态，此时芯片即可随时响应外部设备提出的中断请求（IRQ0 – IRQ15），同时系统还可以使用操作命令字随时修改其中断处理方式。通过中断判优选择，芯片将选中当前最高优先级的中断请求作为中断服务对象，并通过 CPU 引脚 INT 通知 CPU 外中断请求的到来，CPU 响应后，芯片从数据总线 D7-D0 将编程设定的当前服务对象的中断号送出，CPU 由此获取对应的中断向量值，并执行中断服务程序。

5.4.3 中断向量表

上节已指出 CPU 是根据中断号获取中断向量值，即对应中断服务程序的入口地址值。因此为了让 CPU 由中断号查找到对应得中断向量，就需要在内存中建立一张查询表，即中断向量表（在 32 位保护模式下该表称为中断描述符表，见下面说明）。80X86 微机支持 256 个中断，对应每个中断需要安排一个中断服务程序。在 80X86 实模式运行方式下，每个中断向量由 4 个字节组成。这 4 个字节指明了一个中断服务程序的段值和段内偏移值。因此整个向量表的长度为 1024 字节。当 80X86 微机启动时，ROM BIOS 中的程序会在物理内存开始地址 0x0000:0x0000 处初始化并设置中断向量表，而各中断的默认中断服务程序则在 BIOS 中给出。由于中断向量表中的向量是按中断号顺序排列，因此给定一个中断号 N，那么它对应的中断向量在内存中的位置就是 0x0000:N\*4，即对应的中断服务程序入口地址保存在物理内存 0x0000:N\*4 位置处。

在 BIOS 执行初始化操作时，它设置了两个 8259A 芯片支持的 16 个硬件中断向量和 BIOS 提供的中断号为 0x10—0x1f 的中断调用功能向量等。对于实际没有使用的向量则填入临时的哑中断服务程序的地址。以后在系统引导加载操作系统时会根据实际需要修改某些中断向量的值。例如，对于 DOS 操作系统，它会重新设置中断 0x20—0x2f 的中断向量值。而对于 Linux 系统，除了刚开始加载内核时需要用到 BIOS 提供的显示和磁盘读操作中断功能，在内核正常运行之前则会在 setup.s 程序中重新初始化 8259A

芯片并且在 head.s 程序中重新设置一张中断向量表（中断描述符表）。完全抛弃了 BIOS 所提供的中断服务功能。

当 Intel CPU 运行在 32 位保护模式下时，需要使用中断描述符表 IDT（Interrupt Descriptor Table）来管理中断或异常。IDT 是 Intel 8086 -- 80186 CPU 中使用的中断向量表的直接替代物。其作用也类似于中断向量表，只是其中每个中断描述符项中除了含有中断服务程序地址以外，还包含有关特权级和描述符类别等信息。Linux 操作系统工作于 80X86 的保护模式下，因此它使用中断描述符表来设置和保存各中断的“向量”信息。

### 5.4.4 Linux 内核的中断处理

对于 Linux 内核来说，中断信号通常分为两类：硬件中断和软件中断(异常)。每个中断是由 0-255 之间的一个数字来标识。对于中断 int0--int31(0x00--0x1f)，每个中断的功能由 Intel 公司固定设定或保留用，属于软件中断，但 Intel 公司称之为异常。因为这些中断是在 CPU 执行指令时探测到异常情况而引起的。通常还可分为故障(Fault)和陷阱(traps)两类。中断 int32--int255 (0x20--0xff)可以由用户自己设定。所有中断的分类以及执行后 CPU 的动作方式见表 5-1 所示。

表 5 - 1 中断分类以及中断退出后 CPU 的处理方式

中断	英文名称	名称	CPU 检测方式	处理方式
硬件	Maskable	可屏蔽中断	CPU 引脚 INTR	清标志寄存器 eflags 的 IF 标志可屏蔽中断。
	Nonmaskable	不可屏蔽中断	CPU 引脚 NMI	不可屏蔽中断。
软件	Fault	错误	在错误发生之前检测到	CPU 重新执行引起错误的指令。
	Trap	陷阱	在错误发生之后检测到	CPU 继续执行后面的指令。
	Abort	放弃（终止）	在错误发生之后检测到	引起这种错误的程序应该被终止。

在 Linux 系统中，则将 int32--int47（0x20--0x2f）对应于 8259A 中断控制芯片发出的硬件中断请求信号 IRQ0--IRQ15(见表 5-2 所示)，并把程序编程发出的系统调用(system call)中断设置为 int128(0x80)。系统调用中断是用户程序使用操作系统资源的唯一界面接口。

表 5 - 2 Linux 系统中 8259A 芯片中断请求发出的中断号列表

中断请求号	中断号	用途
IRQ0	0x20 （32）	8253 发出的 100HZ 时钟中断
IRQ1	0x21 （33）	键盘中断
IRQ2	0x22 （34）	接连从芯片
IRQ3	0x23 （35）	串行口 2
IRQ4	0x24 （36）	串行口 1
IRQ5	0x25 （37）	并行口 2
IRQ6	0x26 （38）	软盘驱动器
IRQ7	0x27 （39）	并行口 1
IRQ8	0x28 （40）	实时钟中断
IRQ9	0x29 （41）	保留
IRQ10	0x2a （42）	保留
IRQ11	0x2b （43）	保留（网络接口）
IRQ12	0x2c （44）	PS/2 鼠标口中断
IRQ13	0x2d （45）	数学协处理器中断



IRQ14	0x2e (46)	硬盘中断
IRQ15	0x2f (47)	保留

在系统初始化时，内核在 `head.s` 程序中首先使用一个哑中断向量（中断描述符）对中断描述符表（Interrupt Descriptor Table - IDT）中所有 256 个描述符进行了默认设置（`boot/head.s`, 78）。这个哑中断向量指向一个默认的“无中断”处理过程（`boot/head.s`, 150）。当发生了一个中断而又没有重新设置过该中断向量时就会显示信息“未知中断（Unknown interrupt）”。这里对所有 256 项都进行设置可以有效防止出现一般保护性错误（A general protection fault）（异常 13）。否则的话，如果设置的 IDT 少于 256 项，那么在一个要求的中断所指定的描述符项大于设置的最大描述符项时，CPU 就会产生一个一般保护出错（异常 13）。另外，如果硬件出现问题而没有把设备的向量放到数据总线上，此时 CPU 通常会从数据总线上读入全 1(0xff)作为向量，因此会去读取 IDT 表中的第 256 项，因此也会造成一般保护出错。对于系统中需要使用的一些中断，内核会在其继续初始化的处理过程中（`init/main.c`）重新设置这些中断的中断描述符项，让它们指向对应的实际处理过程。通常，异常中断处理过程（`int0 --int 31`）都在 `traps.c` 的初始化函数中进行了重新设置（`kernel/traps.c`, 181），而系统调用中断 `int128` 则在调度程序初始化函数中进行了重新设置（`kernel/sched.c`, 385）。

另外，在设置中断描述符表 IDT 时 Linux 内核使用了中断门和陷阱门两种描述符。它们之间的区别在于对标志寄存器 EFLAGS 中的中断允许标志 IF 的影响。由中断门描述符执行的中断会复位 IF 标志，因此可以避免其它中断干扰当前中断的处理，随后的中断结束指令 `iret` 会从堆栈上恢复 IF 标志的原值；而通过陷阱门执行的中断则不会影响 IF 标志。参见第 11 章中对 `include/asm/system.h` 文件的说明。

### 5.4.5 标志寄存器的中断标志

为了避免竞争条件和中断对临界代码区的干扰，在 Linux 0.11 内核代码中许多地方使用了 `cli` 和 `sti` 指令。`cli` 指令用来复位 CPU 标志寄存器中的中断标志，使得系统在执行 `cli` 指令后不会响应外部中断。`sti` 指令用来设置标志寄存器中的中断标志，以允许 CPU 能识别并响应外部设备发出的中断。当进入可能引起竞争条件的代码区时，内核中就会使用 `cli` 指令来关闭对外部中断的响应，而在执行完竞争代码区时内核就会执行 `sti` 指令以重新允许 CPU 响应外部中断。例如，在修改文件超级块的锁定标志和任务进入/退出等待队列操作时都需要首先使用 `cli` 指令关闭 CPU 对外部中断的响应，在操作完成之后再使用 `sti` 指令开启对外部中断的响应。如果不使用 `cli`、`sti` 指令对，即在需要修改一个文件超级块时不使用 `cli` 来关闭对外部中断的响应，那么在修改之前判断出该超级块锁定标志没有置位而想设置这个标志时，若此时正好发生系统时钟中断而切换到其他任务去运行，并且碰巧其他任务也需要修改这个超级块，那么此时这个其他任务会先设置超级块的锁定标志并且对超级块进行修改操作。当系统又切换回原来的任务时，此时该任务不会再去判断锁定标志就会继续执行设置超级块的锁定标志，从而造成两个任务对临界代码区的同时多重操作，引起超级块数据的不一致性，严重时会导致内核系统崩溃。

## 5.5 Linux 的系统调用

### 5.5.1 系统调用接口

系统调用（通常称为 `syscalls`）是 Linux 内核与上层应用程序进行交互通信的唯一接口，参见图 5-4 所示。从对中断机制的说明可知，用户程序通过直接或间接（通过库函数）调用中断 `int 0x80`，并在 `eax` 寄存器中指定系统调用功能号，即可使用内核资源，包括系统硬件资源。不过通常应用程序都是使用具有标准接口定义的 C 函数库中的函数间接地使用内核的系统调用，见图 5-19 所示。

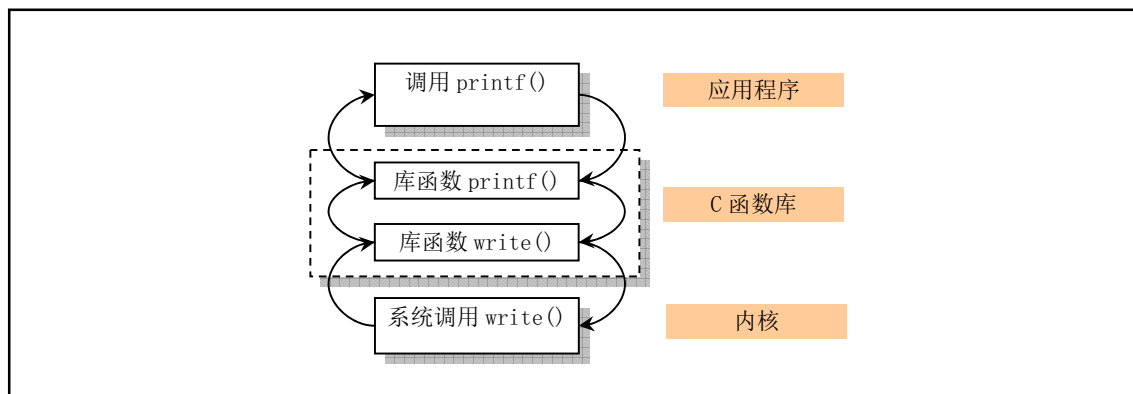


图 5-19 应用程序、库函数和内核系统调用之间的关系

通常系统调用使用函数形式进行调用，因此可带有一个或多个参数。对于系统调用执行的结果，它会在返回值中表示出来。通常负值表示错误，而 0 则表示成功。在出错的情况下，错误的类型码被存放在全局变量 `errno` 中。通过调用库函数 `perror()`，我们可以打印出该错误码对应的出错字符串信息。

在 Linux 内核中，每个系统调用都具有唯一的一个系统调用功能号。这些功能号定义在文件 `include/unistd.h` 中第 60 行开始处。例如，`write` 系统调用的功能号是 4，定义为符号 `_NR_write`。这些系统调用功能号实际上对应于 `include/linux/sys.h` 中定义的系统调用处理程序指针数组表 `sys_call_table[]` 中项的索引值。因此 `write()` 系统调用的处理程序指针就位于该数组的项 4 处。

另外，我们从 `sys_call_table[]` 中可以看出，内核中所有系统调用处理函数的名称基本上都是以符号 `'sys_'` 开始的。例如系统调用 `read()` 在内核源代码中的实现函数就是 `sys_read()`。

### 5.5.2 系统调用处理过程

当应用程序经过库函数向内核发出一个中断调用 `int 0x80` 时，就开始执行一个系统调用。其中寄存器 `eax` 中存放着系统调用号，而携带的参数可依次存放在寄存器 `ebx`、`ecx` 和 `edx` 中。因此 Linux 0.11 内核中用户程序能够向内核最多直接传递三个参数，当然也可以不带参数。处理系统调用中断 `int 0x80` 的过程是程序 `kernel/system_call.s` 中的 `system_call`。

为了方便执行系统调用，内核源代码在 `include/unistd.h` 文件（133—183 行）中定义了宏函数 `_syscalln()`，其中 `n` 代表携带的参数个数，可以分别 0 至 3。因此最多可以直接传递 3 个参数。若需要传递大块数据给内核，则可以传递这块数据的指针值。例如对于 `read()` 系统调用，其定义是：

```
int read(int fd, char *buf, int n);
```

若我们在用户程序中直接执行对应的系统调用，那么该系统调用的宏的形式为：

---

```
#define __LIBRARY__
#include <unistd.h>

_syscall3(int, read, int, fd, char *, buf, int, n)
```

---

因此我们可以在用户程序中直接使用上面的 `_syscall3()` 来执行一个系统调用 `read()`，而不用通过 C 函数库作中介。实际上 C 函数库中函数最终调用系统调用的形式和这里给出的完全一样。

对于 `include/unistd.h` 中给出的每个系统调用宏，都有 `2+2*n` 个参数。其中第 1 个参数对应系统调用返回值的类型；第 2 个参数是系统调用的名称；随后是系统调用所携带参数的类型和名称。这个宏会被扩展成包含内嵌汇编语句的 C 函数，见如下所示。

---

```
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;
}
```

---

可以看出，这个宏经过展开就是一个读操作系统调用的具体实现。其中使用了嵌入汇编语句以功能号 `__NR_read(3)` 执行了 Linux 的系统中断调用 `0x80`。该中断调用在 `eax` (`__res`) 寄存器中返回了实际读取的字节数。若返回的值小于 0，则表示此次读操作出错，于是将出错号取反后存入全局变量 `errno` 中，并向调用程序返回 -1 值。

当进入内核中的系统调用处理程序 `kernel/system_call.s` 后，`system_call` 的代码会首先检查 `eax` 中的系统调用功能号是否在有效系统调用号范围内，然后根据 `sys_call_table[]` 函数指针表调用执行相应的系统调用处理程序。

```
call _sys_call_table(%eax, 4)           // kernel/system_call.s 第 94 行。
```

这句汇编语句操作数的含义是间接调用地址在 `_sys_call_table + %eax * 4` 处的函数。由于 `sys_call_table[]` 指针每项 4 个字节，因此这里需要给系统调用功能号乘上 4。然后用所得到的值从表中获取被调用处理函数的地址。

### 5.5.3 Linux 系统调用的参数传递方式

关于 Linux 用户进程向系统中断调用过程传递参数方面，Linux 系统使用了通用寄存器传递方法，例如寄存器 `ebx`、`ecx` 和 `edx`。这种使用寄存器传递参数方法的一个明显优点就是：当进入系统中断服务程序而保存寄存器值时，这些传递参数的寄存器也被自动地放在了内核态堆栈上，因此用不着再专门对传递参数的寄存器进行特殊处理。这种方法是 Linux 当时所知的最简单最快速的参数传递方法。另外还有一种使用 Intel CPU 提供的系统调用门（System Call gate）的参数传递方法，它在进程用户态堆栈和内核态堆栈自动复制传递的参数。但这种方法使用起来步骤比较复杂。

另外，在每个系统调用处理函数中应该对传递的参数进行验证，以保证所有参数都合法有效。尤其是用户提供的指针，应该进行严格地审查。以保证指针所指的内存区域范围有效，并且具有相应的读写权限。

## 5.6 系统时间和定时

### 5.6.1 系统时间

为了让操作系统能自动地准确提供当前时间和日期信息，PC/AT 微机系统中提供了用电池供电的实时钟 RT (Real Time) 电路支持。通常这部分电路与保存系统信息的少量 CMOS RAM 集成在一个芯片上，

因此这部分电路被称为 RT/CMOS RAM 电路。PC/AT 微机或其兼容机中使用了 Motorola 公司的 MC146818 芯片。

在初始化时，Linux 0.11 内核通过 init/main.c 程序中的 time\_init()函数读取这块芯片中保存的当前时间和日期信息，并通过 kernel/mktime.c 程序中的 kernel\_mktime()函数转换成从 1970 年 1 月 1 日午夜 0 时开始计起到当前的以秒为单位的时间，我们称之为 UNIX 日历时间。该时间确定了系统开始运行的日历时间，被保存在全局变量 startup\_time 中供内核所有代码使用。用户程序可以使用系统调用 time()来读取 startup\_time 的值，而超级用户则可以通过系统调用 stime()来修改这个系统时间值。

另外，再通过下面介绍的从系统启动开始计数的系统滴答值 jiffies，程序就可以唯一地确定运行时刻的当前时间值。由于每个滴答定时值是 10 毫秒，因此内核代码中定义了一个宏来方便代码对当前时间的访问。这个宏定义在 include/linux/sched.h 文件第 142 行上，其形式如下：

---

```
#define CURRENT_TIME (startup_time + jiffies/HZ)
```

---

其中，HZ = 100，是内核系统时钟频率。当前时间宏 CURRENT\_TIME 被定义为系统开机时间 startup\_time 加上开机后系统运行的时间 jiffies/100。在修改一个文件被访问时间或其 i 节点被修改时间时均使用了这个宏。

## 5.6.2 系统定时

在 Linux 0.11 内核的初始化过程中，PC 机的可编程定时芯片 Intel 8253 (8254) 的计数器通道 0 被设置成运行在方式 3 下（方波发生器方式），并且初始计数值 LATCH 被设置成每隔 10 毫秒在通道 0 输出端 OUT 发出一个方波上升沿。由于 8254 芯片的时钟输入频率为 1.193180MHz，因此初始计数值 LATCH=1193180/100，约为 11931。由于 OUT 引脚被连接到可编程中断控制芯片的 0 级上，因此系统每隔 10 毫秒就会发出一个时钟中断请求（IRQ0）信号。这个时间节拍就是操作系统运行的脉搏，我们称之为 1 个系统滴答或一个系统时钟周期。因此每经过 1 个滴答时间，系统就会调用一次时钟中断处理程序（timer\_interrupt）。

时钟中断处理程序 timer\_interrupt 主要用来通过 jiffies 变量来累计自系统启动以来经过的时钟滴答数。每当发生一次时钟中断 jiffies 值就增 1。然后调用 C 语言函数 do\_timer()作进一步的处理。调用时所带的参数 CPL 是从被中断程序的段选择符（保存在堆栈中的 CS 段寄存器值）中取得当前代码特权级 CPL。

do\_timer()函数则根据特权级对当前进程运行时间作累计。如果 CPL=0，则表示进程运行在内核态时被中断，因此内核就会把进程的内核态运行时间统计值 stime 增 1，否则把进程用户态运行时间统计值增 1。如果软盘处理程序 floppy.c 在操作过程中添加过定时器，则对定时器链表进行处理。若某个定时器时间到（递减后等于 0），则调用该定时器的处理函数。然后对当前进程运行时间进行处理，把当前进程运行时间片减 1。时间片是一个进程在被切换掉之前所能持续运行的 CPU 时间，其单位是上面定义的滴答数。如果进程时间片值递减后还大于 0，表示其时间片还没有用完，于是就退出 do\_timer()继续运行当前进程。如果此时进程时间片已经递减为 0，表示该进程已经用完了此次使用 CPU 的时间片，于是程序就会根据被中断程序的级别来确定进一步处理的方法。若被中断的当前进程是工作在用户态的（特权级别大于 0），则 do\_timer()就会调用调度程序 schedule()切换到其他进程去运行。如果被中断的当前进程工作在内核态，也即在内核程序中运行时被中断，则 do\_timer()会立刻退出。因此这样的处理方式决定了 Linux 系统的进程在内核态运行时不会被调度程序切换。即进程在内核态程序中运行时是不可抢占的（nonpreemptive）<sup>1</sup>，但当处于用户态程序中运行时则是可以被抢占的（preemptive）。

注意，上述定时器专门用于软盘马达开启和关闭定时操作。这种定时器类似现代 Linux 系统中的动

<sup>1</sup> 从 Linux 2.4 内核起，Robert Love 开发出了可抢占式的内核升级包。这使得在内核空间低优先级的进程也能被高优先级的进程抢占，从而使系统响应性能最大提高 200%。参见 Robert Love 编著的《Linux 内核开发》一书。

态定时器 (Dynamic Timer)，仅供内核使用。这种定时器可以在需要时动态地创建，而在定时到期时动态地撤销。在 Linux 0.11 内核中定时器同时最多可以有 64 个。定时器的处理代码在 sched.c 程序 264--336 行。

## 5.7 Linux 进程控制

程序是一个可执行的文件，而进程 (process) 是一个执行中的程序实例。利用分时技术，在 Linux 操作系统上同时可以运行多个进程。分时技术的基本原理是把 CPU 的运行时间划分成一个个规定长度的时间片 (time slice)，让每个进程在一个时间片内运行。当进程的时间片用完时系统就利用调度程序切换到另一个进程去运行。因此实际上对于具有单个 CPU 的机器来说某一时刻只能运行一个进程。但由于每个进程运行的时间片很短 (例如 15 个系统滴答=150 毫秒)，所以表面看来好象所有进程在同时运行着。

对于 Linux 0.11 内核来讲，系统最多可有 64 个进程同时存在。除了第一个进程用“手工”建立以外，其余的都是现有进程使用系统调用 fork 创建的新进程，被创建的进程称为子进程 (child process)，创建者，则称为父进程 (parent process)。内核程序使用进程标识号 (process ID, pid) 来标识每个进程。进程由可执行的指令代码、数据和堆栈区组成。进程中的代码和数据部分分别对应一个执行文件中的代码段、数据段。每个进程只能执行自己的代码和访问自己的数据及堆栈区。进程之间的通信需要通过系统调用来进行。对于只有一个 CPU 的系统，在某一时刻只能有一个进程正在运行。内核通过调度程序分时调度各个进程运行。

我们已经知道，Linux 系统中一个进程可以在内核态 (kernel mode) 或用户态 (user mode) 下执行，并且分别使用各自独立的内核态堆栈和用户态堆栈。用户堆栈用于进程在用户态下临时保存调用函数的参数、局部变量等数据；内核堆栈则含有内核程序执行函数调用时的信息。

另外在 Linux 内核中，进程通常被称作任务 (task)，而把运行在用户空间的程序称作进程。本书将在尽量遵守这个默认规则的同时混用这两个术语。

### 5.7.1 任务数据结构

内核程序通过进程表对进程进行管理，每个进程在进程表中占有一项。在 Linux 系统中，进程表项是一个 task\_struct 任务结构指针。任务数据结构定义在头文件 include/linux/sched.h 中。有些书上称其为进程控制块 PCB (Process Control Block) 或进程描述符 PD (Processor Descriptor)。其中保存着用于控制和管理进程的所有信息。主要包括进程当前运行的状态信息、信号、进程号、父进程号、运行时间累计值、正在使用的文件和本任务的局部描述符以及任务状态段信息。该结构每个字段的具体含义如下所示。

```
struct task_struct {
    long state;           //任务的运行状态 (-1 不可运行, 0 可运行(就绪), >0 已停止)。
    long counter;         // 任务运行时间计数(递减)(滴答数), 运行时间片。
    long priority;        // 运行优先数。任务开始运行时 counter=priority, 越大运行越长。
    long signal;          // 信号。是位图, 每个比特位代表一种信号, 信号值=位偏移值+1。
    struct sigaction sigaction[32]; // 信号执行属性结构, 对应信号将要执行的操作和标志信息。
    long blocked;         // 进程信号屏蔽码(对应信号位图)。
    int exit_code;         // 任务停止执行后的退出码, 其父进程会来取。
    unsigned long start_code; // 代码段地址。
    unsigned long end_code;   // 代码长度(字节数)。
    unsigned long end_data;   // 代码长度 + 数据长度(字节数)。
    unsigned long brk;       // 总长度(字节数)。
    unsigned long start_stack; // 堆栈段地址。
    long pid;              // 进程标识号(进程号)。
```

---

```

    long father;           // 父进程号。
    long pgrp;             // 进程组号。
    long session;          // 会话号。
    long leader;           // 会话首领。
    unsigned short uid;     // 用户标识号（用户 id）。
    unsigned short euid;    // 有效用户 id。
    unsigned short suid;    // 保存的用户 id。
    unsigned short gid;     // 组标识号（组 id）。
    unsigned short egid;    // 有效组 id。
    unsigned short sgid;    // 保存的组 id。
    long alarm;             // 报警定时值（滴答数）。
    long utime;             // 用户态运行时间（滴答数）。
    long stime;             // 系统态运行时间（滴答数）。
    long cutime;           // 子进程用户态运行时间。
    long cstime;           // 子进程系统态运行时间。
    long start_time;        // 进程开始运行时刻。
    unsigned short used_math; // 标志：是否使用了协处理器。
    int tty;                // 进程使用 tty 终端的子设备号。-1 表示没有使用。
    unsigned short umask;   // 文件创建属性屏蔽位。
    struct m_inode * pwd;   // 当前工作目录 i 节点结构指针。
    struct m_inode * root;  // 根目录 i 节点结构指针。
    struct m_inode * executable; // 执行文件 i 节点结构指针。
    unsigned long close_on_exec; // 执行时关闭文件句柄位图标志。（参见 include/fcntl.h）
    struct file * filp[NR_OPEN]; // 文件结构指针表，最多 32 项。表项号即是文件描述符的值。
    struct desc_struct ldt[3]; // 局部描述符表。0-空，1-代码段 cs，2-数据和堆栈段 ds&ss。
    struct tss_struct tss;  // 进程的任务状态段信息结构。
};

```

---

◆**long state** 字段含有进程的当前状态代号。如果进程正在等待使用 CPU 或者进程正被运行，那么 state 的值是 TASK\_RUNNING。如果进程正在等待某一事件的发生因而处于空闲状态，那么 state 的值就是 TASK\_INTERRUPTIBLE 或者 TASK\_UNINTERRUPTIBLE。这两个值含义的区别在于处于 TASK\_INTERRUPTIBLE 状态的进程能够被信号唤醒并激活，而处于 TASK\_UNINTERRUPTIBLE 状态的进程则通常是在直接或间接地等待硬件条件的满足因而不会接受任何信号。TASK\_STOPPED 状态用于说明一个进程正处于停止状态。例如进程在收到一个相关信号时（例如 SIGSTOP、SIGTIN 或 SIGTTOU 等）或者当进程被另一个进程使用 ptrace 系统调用监控并且控制权在监控进程中时。TASK\_ZOMBIE 状态用于描述一个进程已经被终止，但其任务数据结构项仍然存在于任务结构表中。一个进程在这些状态之间的转换过程见下节说明。

◆**long counter** 字段保存着进程在被暂时停止本次运行之前还能执行的时间滴答数，即在正常情况下还需要经过几个系统时钟周期才切换到另一个进程。调度程序会使用进程的 counter 值来选择下一个要执行的进程，因此 counter 可以看作是一个进程的动态特性。在一个进程刚被创建时 counter 的初值等于 priority。

◆**long priority** 用于给 counter 赋初值。在 Linux 0.11 中这个初值为 15 个系统时钟周期时间（15 个滴答）。当需要时调度程序会使用 priority 的值为 counter 赋一个初值，参见 sched.c 程序和 fork.c 程序。当然，priority 的单位也是时间滴答数。

◆**long signal** 字段是进程当前所收到信号的位图，共 32 个比特位，每个比特位代表一种信号，信号值=位偏移值+1。因此 Linux 内核最多有 32 个信号。在每个系统调用处理过程的最后，系统会使用该信号位图对信号进行预处理。

◆**struct sigaction sigaction[32]** 结构数组用来保存处理各信号所使用的操作和属性。数组的每一项对应一个信号。

◆ **long blocked** 字段是进程当前不想处理的信号的阻塞位图。与 **signal** 字段类似，其每一比特位代表一种被阻塞的信号。

◆ **int exit** 字段是用来保存程序终止时的退出码。在子进程结束后父进程可以查询它的这个退出码。

◆ **unsigned long start\_code** 字段是进程代码在 CPU 线性地址空间中的开始地址，在 Linux 0.1x 内核中其值是 64MB 的整数倍。

◆ **unsigned long end\_code** 字段保存着进程代码的字节长度值。

◆ **unsigned long end\_data** 字段保存着进程的代码长度 + 数据长度的总字节长度值。

◆ **unsigned long brk** 字段也是进程代码和数据的总字节长度值（指针值），但是还包括未初始化的数据区 **bss**，参见图 10-6 所示。这是 **brk** 在一个进程开始执行时的初值。通过修改这个指针，内核可以为进程添加和释放动态分配的内存。这通常是通过调用 **malloc()** 函数并通过 **brk** 系统调用由内核进行操作。

◆ **unsigned long start\_stack** 字段值指向进程逻辑地址空间中堆栈的起始处。同样请参见图 10-6 中的堆栈指针位置。

◆ **long pid** 是进程标识号，即进程号。它被用来唯一地标识进程。

◆ **long father** 是创建本进程的父进程的进程号。

◆ **long pgrp** 是指进程所属进程组号。

◆ **long session** 是进程的会话号，即所属会话的进程号。

◆ **long leader** 是会话首进程号。有关进程组和会话的概念请参见第 4 章程序列表后的说明。

◆ **unsigned short uid** 是拥有该进程的用户标识号（用户 id）。

◆ **unsigned short euid** 是有效用户标识号，用于指明访问文件的权力。

◆ **unsigned short suid** 是保存的用户标识号。当执行文件的设置用户 ID 标志（**set-user-ID**）置位时，**suid** 中保存着执行文件的 **uid**。否则 **suid** 等于进程的 **euid**。

◆ **unsigned short gid** 是用户所属组标识号（组 id）。指明了拥有该进程的用户组。

◆ **unsigned short egid** 是有效组标识号，用于指明该组用户访问文件的权限。

◆ **unsigned short sgid** 是保存的用户组标识号。当执行文件的设置组 ID 标志（**set-group-ID**）置位时，**sgid** 中保存着执行文件的 **gid**。否则 **sgid** 等于进程的 **egid**。有关这些用户号和组号的描述请参见第 5 章 **sys.c** 程序前的概述。

◆ **long alarm** 是进程的报警定时值（滴答数）。如果进程使用系统调用 **alarm()** 设置过该字段值（**alarm()** 在 **kernel/sched.c** 第 338 行开始处。内核会把该函数以秒为单位的参数值转换成滴答值，加上系统当前时间滴答值之后保存在该字段中），那么此后当系统时间滴答值超过了 **alarm** 字段值时，内核就会向该进程发送一个 **SIGALRM** 信号。默认时该信号会终止程序的执行。当然我们也可以使用信号捕捉函数（**signal()** 或 **sigaction()**）来捕捉该信号进行指定的操作。

◆ **long utime** 是累计进程在用户态运行的时间（滴答数）。

◆ **long stime** 是累计进程在系统态（内核态）运行的时间（滴答数）。

◆ **long cutime** 是累计进程的子进程在用户态运行的时间（滴答数）。

◆ **long cstime** 是累计进程的子进程内核态运行的时间（滴答数）。

◆ **long start\_time** 是进程生成并开始运行的时刻。

◆ **unsigned short used\_math** 是一个标志，指明本进程是否使用了协处理器。

◆ **int tty** 是进程使用 **tty** 终端的子设备号。-1 表示没有使用。

◆ **unsigned short umask** 是进程创建新文件时所使用的属性屏蔽位，即新建文件所设置的访问属性。

◆ **struct m\_inode \* pwd** 是进程的当前工作目录 i 节点结构。每个进程都有一个当前工作目录，用于解析相对路径名，并且可以使用系统调用 **chdir** 来改变之。

◆ **struct m\_inode \* root** 是进程自己的根目录 i 节点结构。每个进程都可能有自己指定的根目录，用于解析绝对路径名。只有超级用户能通过系统调用 **chroot** 来修改这个根目录。

◆`struct m_inode * executable` 是进程运行的执行文件在内存中 `i` 节点结构指针。系统可根据该字段来判断系统中是否还有另一个进程在运行同一个执行文件。如果有的话那么这个内存中 `i` 节点引用计数值 `executable->i_count` 会大于 1。在进程被创建时该字段被赋予和父进程同一字段相同的值，即表示正在与父进程运行同一个程序。当在进程中调用 `exec()` 类函数而去执行一个指定的执行文件时，该字段值就会被替换成 `exec()` 函数所执行程序内存 `i` 节点指针。当进程调用 `exit()` 函数而执行退出处理时该字段所指内存 `i` 节点的引用计数会被减 1，并且该字段将被置空。该字段的主要作用体现在 `memory.c` 程序的 `share_page()` 函数中。该函数代码根据进程的 `executable` 所指节点的引用计数可判断系统中当前运行的程序是否有多个拷贝存在（起码 2 个）。若是的话则在他们之间尝试页面共享操作。

在系统初始化时，在第 1 次调用执行 `execve()` 函数之前，系统创建的所有任务的 `executable` 都是 0。这些任务包括任务 0、任务 1 以及任务 1 直接创建的没有执行过 `execve()` 的所有任务，即代码直接包含在内核代码中的所有任务的 `executable` 都是 0。因为任务 0 的代码包含在内核代码中，它不是由系统从文件系统上加载运行的执行文件，因此内核代码中固定设置它的 `executable` 值为 0。另外，创建新进程时，`fork()` 会复制父进程的任务数据结构，因此任务 1 的 `executable` 也是 0。但在执行了 `execve()` 之后，`executable` 就被赋予了被执行文件的内存 `i` 节点的指针。此后所有任务的该值就均不会为 0 了。

◆`unsigned long close_on_exec` 是一个进程文件描述符（文件句柄）位图标志。每个比特位代表一个文件描述符，用于确定在调用系统调用 `execve()` 时需要关闭的文件描述符（参见 `include/fcntl.h`）。当一个程序使用 `fork()` 函数创建了一个子进程时，通常会在该子进程中调用 `execve()` 函数加载执行另一个新程序。此时子进程将完全被新程序替换掉，并在子进程中开始执行新程序。若一个文件描述符在 `close_on_exec` 中的对应比特位是置位状态，那么在子进程执行 `execve()` 调用时对应打开着的文件描述符将被关闭，即在新程序中该文件描述符被关闭。否则该文件描述符将始终处于打开状态。

◆`struct file * filp[NR_OPEN]` 是进程使用的所有打开文件的文件结构指针表，最多 32 项。文件描述符的值即是该结构中的索引值。其中每一项用于文件描述符定位文件指针和访问文件。

◆`struct desc_struct ldt[3]` 是该进程局部描述符表结构。定义了该任务在虚拟地址空间中的代码段和数据段。其中数组项 0 是空项，项 1 是代码段描述符，项 2 是数据段（包含数据和堆栈）描述符。

◆`struct tss_struct tss` 是进程的任务状态段 TSS（Task State Segment）信息结构。在任务从执行中被切换出时 `tss_struct` 结构保存了当前处理器的所有寄存器值。当任务又被 CPU 重新执行时，CPU 就会利用这些值恢复到任务被切换出时的状态，并开始执行。

当一个进程在执行时，CPU 的所有寄存器中的值、进程的状态以及堆栈中的内容被称为该进程的上下文。当内核需要切换（switch）至另一个进程时，它就需要保存当前进程的所有状态，也即保存当前进程的上下文，以便在再次执行该进程时，能够恢复到切换时的状态执行下去。在 Linux 中，当前进程上下文均保存在进程的任务数据结构中。在发生中断时，内核就在被中断进程的上下文中，在内核态下执行中断服务例程。但同时会保留所有需要用到资源，以便中断服务结束时能恢复被中断进程的执行。

## 5.7.2 进程运行状态

一个进程在其生存期内，可处于一组不同的状态下，称为进程状态。见图 5-20 所示。进程状态保存在进程任务结构的 `state` 字段中。当进程正在等待系统中的资源而处于等待状态时，则称其处于睡眠等待状态。在 Linux 系统中，睡眠等待状态被分为可中断的和不可中断的等待状态。



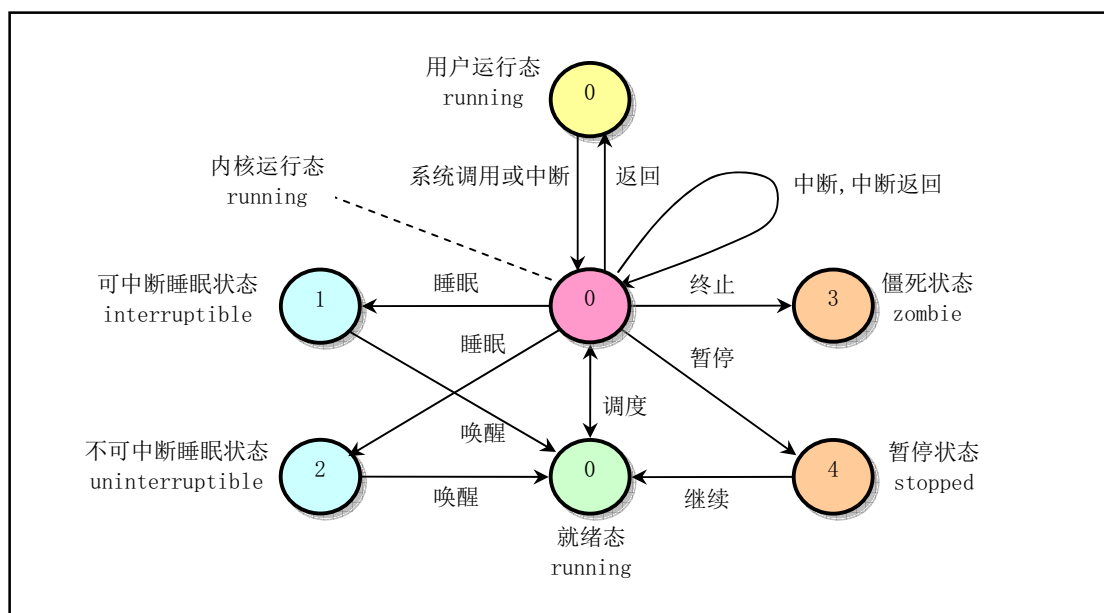


图 5-20 进程状态及转换关系

## ◆运行状态（TASK\_RUNNING）

当进程正在被 CPU 执行，或已经准备就绪随时可由调度程序执行，则称该进程为处于运行状态（running）。若此时进程没有被 CPU 执行，则称其处于就绪运行状态。见图 5-20 中三个标号为 0 的状态。进程可以在内核态运行，也可以在用户态运行。当一个进程在内核代码中运行时，我们称其处于内核运行态，或简称为内核态；当一个进程正在执行用户自己的代码时，我们称其为处于用户运行态（用户态）。当系统资源已经可用时，进程就被唤醒而进入准备运行状态，该状态称为就绪态。这些状态（图中中间一列）在内核中表示方法相同，都被成为处于 TASK\_RUNNING 状态。当一个新进程刚被创建出后就处于本状态中（最下一个 0 处）。

## ◆可中断睡眠状态（TASK\_INTERRUPTIBLE）

当进程处于可中断等待（睡眠）状态时，系统不会调度该进程执行。当系统产生一个中断或者释放了进程正在等待的资源，或者进程收到一个信号，都可以唤醒进程转换到就绪状态（可运行状态）。

## ◆不可中断睡眠状态（TASK\_UNINTERRUPTIBLE）

除了不会因为收到信号而被唤醒，该状态与可中断睡眠状态类似。但处于该状态的进程只有被使用 wake\_up() 函数明确唤醒时才能转换到可运行的就绪状态。该状态通常在进程需要不受干扰地等待或者所等待事件会很快发生时使用。

## ◆暂停状态（TASK\_STOPPED）

当进程收到信号 SIGSTOP、SIGTSTP、SIGTTIN 或 SIGTTOU 时就会进入暂停状态。可向其发送 SIGCONT 信号让进程转换到可运行状态。进程在调试期间接收到任何信号均会进入该状态。在 Linux 0.11 中，还未实现对该状态的转换处理。处于该状态的进程将被作为进程终止来处理。

## ◆僵死状态（TASK\_ZOMBIE）

当进程已停止运行，但其父进程还没有调用 wait() 询问其状态时，则称该进程处于僵死状态。为了了让父进程能够获取其停止运行的信息，此时子进程的任务数据结构信息还需要保留着。一旦父进程调用 wait() 取得了子进程的信息，则处于该状态进程的任务数据结构就会被释放掉。

当一个进程的运行时间片用完，系统就会使用调度程序强制切换到其他的进程去执行。另外，如果进程在内核态执行时需要等待系统的某个资源，此时该进程就会调用 sleep\_on() 或 interruptible\_sleep\_on()

自愿地放弃 CPU 的使用权，而让调度程序去执行其他进程。进程则进入睡眠状态（TASK\_UNINTERRUPTIBLE 或 TASK\_INTERRUPTIBLE）。

只有当进程从“内核运行态”转移到“睡眠状态”时，内核才会进行进程切换操作。在内核态下运行的进程不能被其他进程抢占，而且一个进程不能改变另一个进程的状态。为了避免进程切换时造成内核数据错误，内核在执行临界区代码时会禁止一切中断。

### 5.7.3 进程初始化

在 boot/目录中，引导程序把内核从磁盘上加载到内存中，并让系统进入保护模式下运行后，就开始执行系统初始化程序 init/main.c。该程序首先确定如何分配使用系统物理内存，然后调用内核各部分的初始化函数分别对内存管理、中断处理、块设备和字符设备、进程管理以及硬盘和软盘硬件进行初始化处理。在完成了这些操作之后，系统各部分已经处于可运行状态。此后程序把自己“手工”移动到任务 0（进程 0）中运行，并使用 fork()调用首次创建出进程 1。在进程 1 中程序将继续进行应用环境的初始化并执行 shell 登录程序。而原进程 0 则会在系统空闲时被调度执行，此时任务 0 仅执行 pause()系统调用，其中又会去执行调度函数。

“移动到任务 0 中执行”这个过程由宏 move\_to\_user\_mode(include/asm/system.h)完成。它把 main.c 程序执行流从内核态（特权级 0）移动到了用户态（特权级 3）的任务 0 中继续运行。在移动之前，系统在对调度程序的初始化过程（sched\_init()）中，首先对任务 0 的运行环境进行了设置。这包括人工预先设置好任务 0 数据结构各字段的值（include/linux/sched.h）、在全局描述符表中添入任务 0 的任务状态段（TSS）描述符和局部描述符表（LDT）的段描述符，并把它们分别加载到任务寄存器 tr 和局部描述符表寄存器 ldtr 中。

这里需要强调的是，内核初始化是一个特殊过程，内核初始化代码也即是任务 0 的代码。从任务 0 数据结构中设置的初始数据可知，任务 0 的代码段和数据段的基址是 0、段限长是 640KB。而内核代码段和数据段的基址是 0、段限长是 16MB，因此任务 0 的代码段和数据段分别包含在内核代码段和数据段中。内核初始化程序 main.c 也即是任务 0 中的代码，只是在移动到任务 0 之前系统正以内核态特权级 0 运行着 main.c 程序。宏 move\_to\_user\_mode 的功能就是把运行特权级从内核态的 0 级变换到用户态的 3 级，但是仍然继续执行原来的代码指令流。

在移动到任务 0 的过程中，宏 move\_to\_user\_mode 使用了中断返回指令造成特权级改变的方法。使用这种方法进行控制权转移是由 CPU 保护机制造成的。CPU 允许低级别（如特权级 3）代码通过调用门或中断、陷阱门来调用或转移到高级别代码中运行，但反之则不行。因此内核采用了这种模拟 IRET 返回低级别代码的方法。该方法的主要思想是在堆栈中构筑中断返回指令需要的内容，把返回地址的段选择符设置成任务 0 代码段选择符，其特权级为 3。此后执行中断返回指令 iret 时将导致系统 CPU 从特权级 0 跳转到外层的特权级 3 上运行。参见图 5-21 所示的特权级发生变化时中断返回堆栈结构示意图。

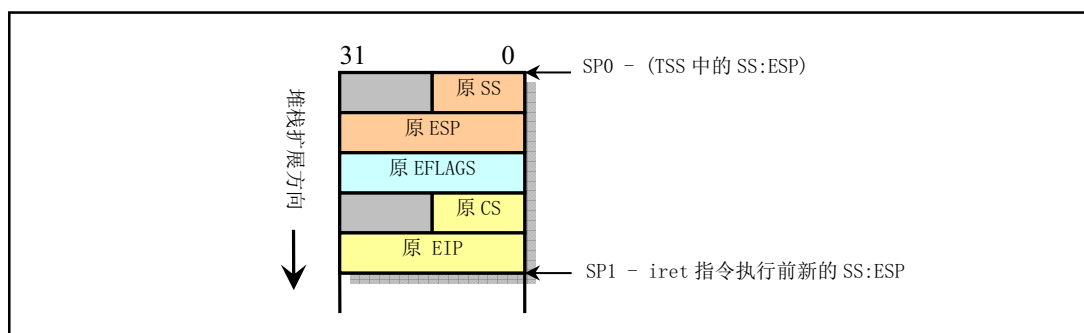


图 5-21 特权级发生变化时中断返回堆栈结构示意图

宏 `move_to_user_mode` 首先往内核堆栈中压入任务 0 堆栈段（即数据段）选择符和内核堆栈指针。然后压入标志寄存器内容。最后压入任务 0 代码段选择符和执行中断返回后需要执行的下一条指令的偏移位置。该偏移位置是 `iret` 后的一条指令处。

当执行 `iret` 指令时，CPU 把返回地址送入 `CS:EIP` 中，同时弹出堆栈中标志寄存器内容。由于 CPU 判断出目的代码段的特权级是 3，与当前内核态的 0 级不同。于是 CPU 会把堆栈中的堆栈段选择符和堆栈指针弹出到 `SS:ESP` 中。由于特权级发上了变化，段寄存器 `DS`、`ES`、`FS` 和 `GS` 的值变得无效，此时 CPU 会把这些段寄存器清零。因此在执行了 `iret` 指令后需要重新加载这些段寄存器。此后，系统就开始以特权级 3 运行在任务 0 的代码上。所使用的用户态堆栈还是原来在移动之前使用的堆栈。而其内核态堆栈则被指定为其任务数据结构所在页面的顶端开始（`PAGE_SIZE + (long)&init_task`）。由于以后在创建新进程时，需要复制任务 0 的任务数据结构，包括其用户堆栈指针，因此要求任务 0 的用户态堆栈在创建任务 1（进程 1）之前保持“干净”状态。

### 5.7.4 创建新进程

Linux 系统中创建新进程使用 `fork()` 系统调用。所有进程都是通过复制进程 0 而得到的，都是进程 0 的子进程。

在创建新进程的过程中，系统首先在任务数组中找出一个还没有被任何进程使用的空项（空槽）。如果系统已经有 64 个进程在运行，则 `fork()` 系统调用会因为任务数组表中没有可用空项而出错返回。然后系统为新建进程在主内存区中申请一页内存来存放其任务数据结构信息，并复制当前进程任务数据结构中的所有内容作为新进程任务数据结构的模板。为了防止这个还未处理完成的新建进程被调度函数执行，此时应该立刻将新进程状态置为不可中断的等待状态（`TASK_UNINTERRUPTIBLE`）。

随后对复制的任务数据结构进行修改。把当前进程设置为新进程的父进程，清除信号位图并复位新进程各统计值，并设置初始运行时间片值为 15 个系统滴答数（150 毫秒）。接着根据当前进程设置任务状态段（TSS）中各寄存器的值。由于创建进程时新进程返回值应为 0，所以需要设置 `tss.eax = 0`。新建进程内核态堆栈指针 `tss.esp0` 被设置成新进程任务数据结构所在内存页面的顶端，而堆栈段 `tss.ss0` 被设置成内核数据段选择符。`tss.ldt` 被设置为局部表描述符在 GDT 中的索引值。如果当前进程使用了协处理器，则还需要把协处理器的完整状态保存到新进程的 `tss.i387` 结构中。

此后系统设置新任务的代码和数据段基址、限长，并复制当前进程内存分页管理的页表。注意，此时系统并不为新的进程分配实际的物理内存页面，而是让它共享其父进程的内存页面。只有当父进程或新进程中任意一个有写内存操作时，系统才会为执行写操作的进程分配相关的独自使用的内存页面。这种处理方式称为写时复制（Copy On Write）技术。

随后，如果父进程中有文件是打开的，则应对应文件的打开次数增 1。接着在 GDT 中设置新任务的 TSS 和 LDT 描述符项，其中基址信息指向新进程任务结构中的 `tss` 和 `ldt`。最后再将新任务设置成可运行状态并返回新进程号。

另外请注意，创建一个新的子进程和加载运行一个执行程序文件是两个不同的概念。当创建子进程时，它完全复制了父进程的代码和数据区，并会在其中执行子进程部分的代码。而执行块设备上的一个程序时，一般是在子进程中运行 `exec()` 系统调用来操作的。在进入 `exec()` 后，子进程原来的代码和数据区就会被清掉（释放）。待该子进程开始运行新程序时，由于此时内核还没有从块设备上加载该程序的代码，CPU 就会立刻产生代码页面不存在的异常（Fault），此时内存管理程序就会从块设备上加载相应的代码页面，然后 CPU 又重新执行引起异常的指令。到此时新程序的代码才真正开始被执行。

### 5.7.5 进程调度

内核中的调度程序用于选择系统中下一个要运行的进程。这种选择运行机制是多任务操作系统的基础。调度程序可以看作是在所有处于运行状态的进程之间分配 CPU 运行时间的管理代码。由前面描述可知，Linux 进程是抢占式的，但被抢占的进程仍然处于 `TASK_RUNNING` 状态，只是暂时没有被 CPU 运

行。进程的抢占发生在进程处于用户态执行阶段，在内核态执行时是不能被抢占的。

为了能让进程有效地使用系统资源，又能使进程有较快的响应时间，就需要对进程的切换调度采用一定的调度策略。在 Linux 0.11 中采用了基于优先级排队的调度策略。

### 调度程序

`schedule()` 函数首先扫描任务数组。通过比较每个就绪态 (`TASK_RUNNING`) 任务的运行时间递减滴答计数 `counter` 的值来确定当前哪个进程运行的时间最少。哪一个的值大，就表示运行时间还不长，于是就选中该进程，并使用任务切换宏函数切换到该进程运行。

如果此时所有处于 `TASK_RUNNING` 状态进程的时间片都已经用完，系统就会根据每个进程的优先权值 `priority`，对系统中所有进程（包括正在睡眠的进程）重新计算每个任务需要运行的时间片值 `counter`。计算的公式是：

$$counter = \frac{counter}{2} + priority$$

这样对于正在睡眠的进程当它们被唤醒时就具有较高的时间片 `counter` 值。然后 `schedule()` 函数重新扫描任务数组中所有处于 `TASK_RUNNING` 状态的进程，并重复上述过程，直到选择一个进程为止。最后调用 `switch_to()` 执行实际的进程切换操作。

如果此时没有其他进程可运行，系统就会选择进程 0 运行。对于 Linux 0.11 来说，进程 0 会调用 `pause()` 把自己置为可中断的睡眠状态并再次调用 `schedule()`。不过在调度进程运行时，`schedule()` 并不在意进程 0 处于什么状态。只要系统空闲就调度进程 0 运行。

### 进程切换

每当选择出一个新的可运行进程时，`schedule()` 函数就会调用定义在 `include/asm/system.h` 中的 `switch_to()` 宏执行实际进程切换操作。该宏会把 CPU 的当前进程状态（上下文）替换成新进程的状态。在进行切换之前，`switch_to()` 首先检查要切换到的进程是否就是当前进程，如果是则什么也不做，直接退出。否则就首先把内核全局变量 `current` 置为新任务的指针，然后长跳转到新任务的任务状态段 TSS 组成的地址处，造成 CPU 执行任务切换操作。此时 CPU 会把其所有寄存器的状态保存到当前任务寄存器 TR 中 TSS 段选择符所指向的当前进程任务数据结构的 `tss` 结构中，然后把新任务状态段选择符所指向的新任务数据结构中 `tss` 结构中的寄存器信息恢复到 CPU 中，系统就正式开始运行新切换的任务了。这个过程可参见图 5-22 所示。

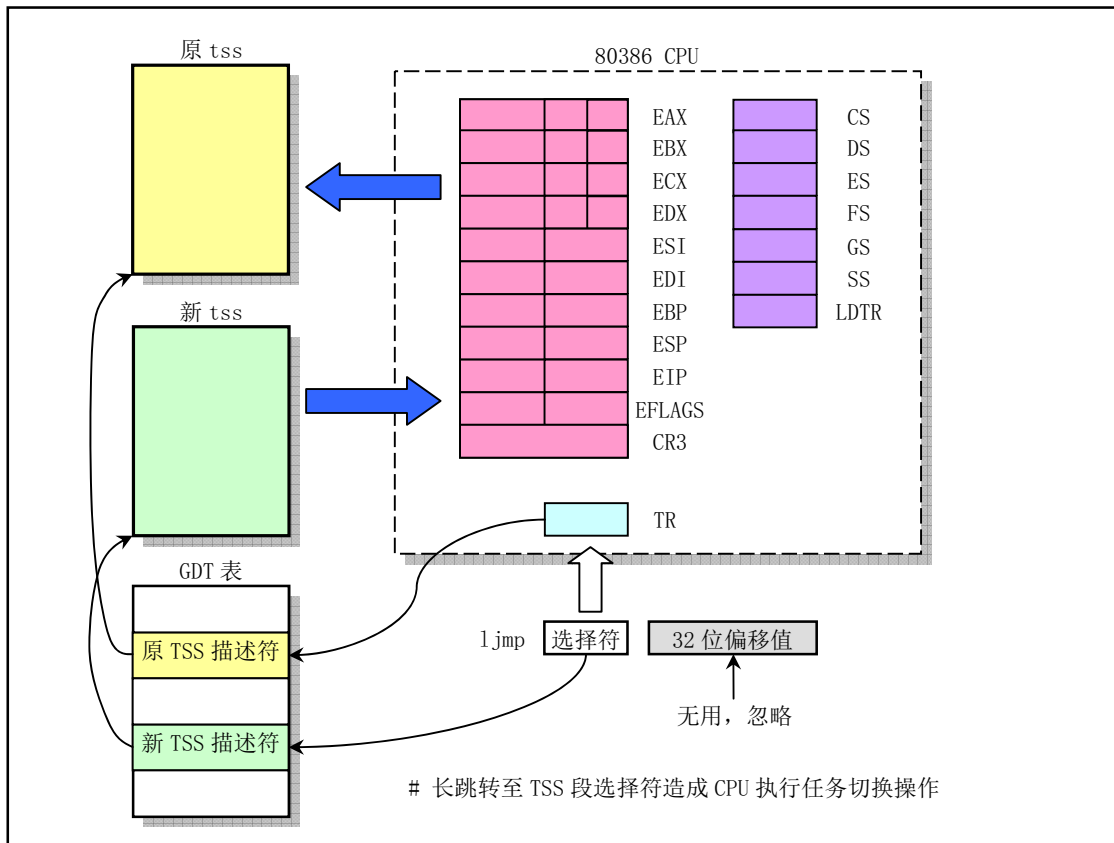


图 5-22 任务切换操作示意图

### 5.7.6 终止进程

当一个进程结束了运行或在半途中终止了运行，那么内核就需要释放该进程所占用的系统资源。这包括进程运行时打开的文件、申请的内存等。

当一个用户程序调用 `exit()` 系统调用时，就会执行内核函数 `do_exit()`。该函数会首先释放进程代码段和数据段占用的内存页面，关闭进程打开着的所有文件，对进程使用的当前工作目录、根目录和运行程序的 `i` 节点进行同步操作。如果进程有子进程，则让 `init` 进程作为其所有子进程的父进程。如果进程是一个会话头进程并且有控制终端，则释放控制终端，并向属于该会话的所有进程发送挂断信号 `SIGHUP`，这通常会终止该会话中的所有进程。然后把进程状态置为僵死状态 `TASK_ZOMBIE`。并向其原父进程发送 `SIGCHLD` 信号，通知其某个子进程已经终止。最后 `do_exit()` 调用调度函数去执行其他进程。由此可见在进程被终止时，它的任务数据结构仍然保留着。因为其父进程还需要使用其中的信息。

在子进程在执行期间，父进程通常使用 `wait()` 或 `waitpid()` 函数等待其某个子进程终止。当等待的子进程被终止并处于僵死状态时，父进程就会把子进程运行所使用的时间累加到自己进程中。最终释放已终止子进程任务数据结构所占用的内存页面，并置空子进程在任务数组中占用的指针项。

## 5.8 Linux 系统中堆栈的使用方法

本节内容概要描述了 Linux 内核从开机引导到系统正常运行过程中对堆栈的使用方式。这部分内容的说明与内核代码关系比较密切，可以先跳过。在开始阅读相应代码时再回来仔细研究。

Linux 0.11 系统中共使用了四种堆栈。一种是系统引导初始化时临时使用的堆栈；一种是进入保护

模式之后提供内核程序初始化使用的堆栈，位于内核代码地址空间固定位置处。该堆栈也是后来任务 0 使用的用户态堆栈；另一种是每个任务通过系统调用，执行内核程序时使用的堆栈，我们称之为任务的内核态堆栈。每个任务都有自己独立的内核态堆栈；最后一种是任务在用户态执行的堆栈，位于任务（进程）逻辑地址空间近末端处。

使用多个栈或在不同情况下使用不同栈的主要原因有两个。首先是由于从实模式进入保护模式，使得 CPU 对内存寻址访问方式发生了变化，因此需要重新调整设置栈区域。另外，为了解决不同 CPU 特权级共享使用堆栈带来的保护问题，执行 0 级的内核代码和执行 3 级的用户代码需要使用不同的栈。当一个任务进入内核态运行时，就会使用其 TSS 段中给出的特权级 0 的堆栈指针 `tss.ss0`、`tss.esp0`，即内核栈。原用户栈指针会被保存在内核栈中。而当从内核态返回用户态时，就会恢复使用用户态的堆栈。下面分别对它们进行说明。

### 5.8.1 初始化阶段

#### 开机初始化时(`bootsect.s`, `setup.s`)

当 `bootsect` 代码被 ROM BIOS 引导加载到物理内存 `0x7c00` 处时，并没有设置堆栈段，当然程序也没有使用堆栈。直到 `bootsect` 被移动到 `0x9000:0` 处时，才把堆栈段寄存器 `SS` 设置为 `0x9000`，堆栈指针 `esp` 寄存器设置为 `0xff00`，也即堆栈顶端在 `0x9000:0xff00` 处，参见 `boot/bootsect.s` 第 61、62 行。`setup.s` 程序中也沿用了 `bootsect` 中设置的堆栈段。这就是系统初始化时临时使用的堆栈。

#### 进入保护模式时(`head.s`)

从 `head.s` 程序起，系统开始正式在保护模式下运行。此时堆栈段被设置为内核数据段（`0x10`），堆栈指针 `esp` 设置成指向 `user_stack` 数组的顶端（参见 `head.s`，第 31 行），保留了 1 页内存（4K）作为堆栈使用。`user_stack` 数组定义在 `sched.c` 的 67--72 行，共含有 1024 个长字。它在物理内存中的位置示意图可参见下图 5-23 所示。此时该堆栈是内核程序自己使用的堆栈。其中的给出地址是大约值，它们与编译时的实际设置参数有关。这些地址位置是从编译内核时生成的 `system.map` 文件中查到的。

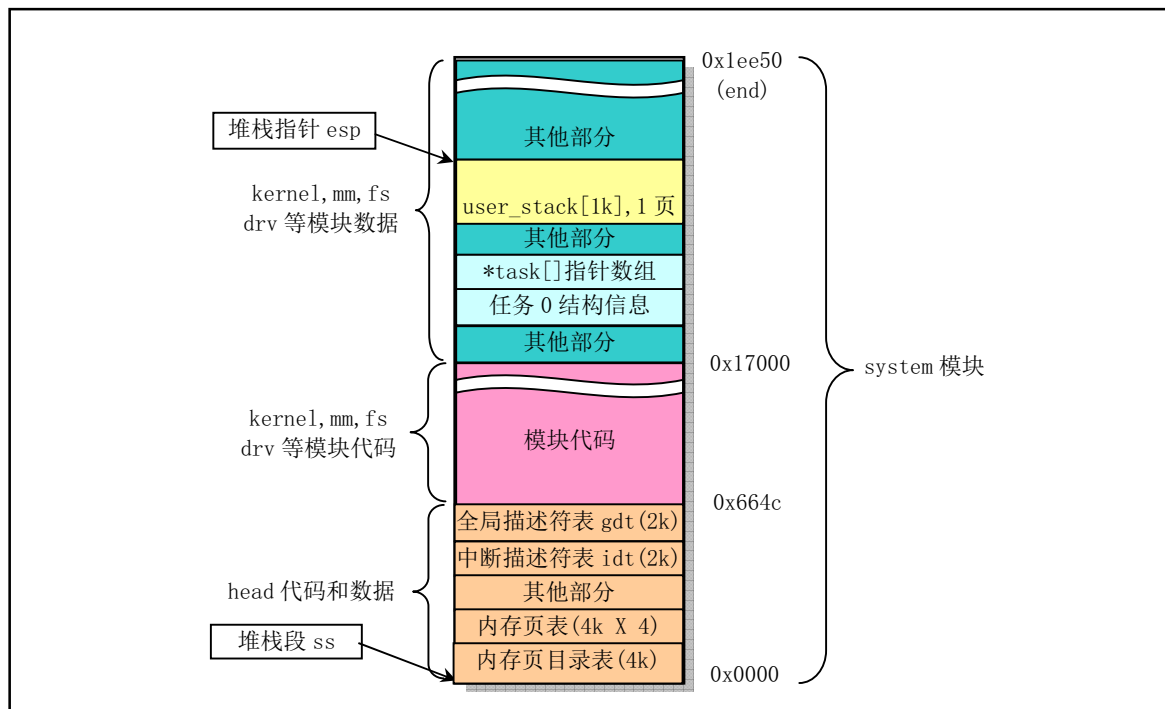


图 5-23 刚进入保护模式时内核使用的堆栈示意图



## 初始化时(main.c)

在 `init/main.c` 程序中，在执行 `move_to_user_mode()` 代码把控制权移交给任务 0 之前，系统一直使用上述堆栈。而在执行过 `move_to_user_mode()` 之后，`main.c` 的代码被“切换”成任务 0 中执行。通过执行 `fork()` 系统调用，`main.c` 中的 `init()` 将在任务 1 中执行，并使用任务 1 的堆栈。而 `main()` 本身则在被“切换”成为任务 0 后，仍然继续使用上述内核程序自己的堆栈作为任务 0 的用户态堆栈。关于任务 0 所使用堆栈的详细描述见后面说明。

## 5.8.2 任务的堆栈

每个任务都有两个堆栈，分别用于用户态和内核态程序的执行，并且分别称为用户态堆栈和内核态堆栈。除了处于不同 CPU 特权级中，这两个堆栈之间的主要区别在于任务的内核态堆栈很小，所保存的数据量最多不能超过（4096 - 任务数据结构块）个字节，大约为 3K 字节。而任务的用户态堆栈却可以在用户的 64MB 空间内延伸。

### 在用户态运行时

每个任务（除了任务 0 和任务 1）有自己的 64MB 地址空间。当一个任务（进程）刚被创建时，它的用户态堆栈指针被设置在其地址空间的靠近末端（64MB 顶端）部分。实际上末端部分还要包括执行程序的参数和环境变量，然后才是用户堆栈空间，见图 5-24 所示。应用程序在用户态下运行时就一直使用这个堆栈。堆栈实际使用的物理内存则由 CPU 分页机制确定。由于 Linux 实现了写时复制功能（Copy on Write），因此在进程被创建后，若该进程及其父进程都没有使用堆栈，则两者共享同一堆栈对应的物理内存页面。只有当其中一个进程执行堆栈写操作（例如 `push` 操作）时内核内存管理程序才会为写操作进程分配新的内存页面。而进程 0 和进程 1 的用户堆栈比较特殊，见后面说明。

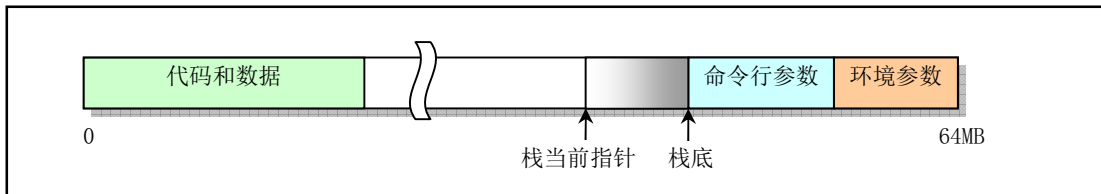


图 5-24 逻辑空间中的用户态堆栈

### 在内核态运行时

每个任务有其自己的内核态堆栈，用于任务在内核代码中执行期间。其所在线性地址中的位置由该任务 TSS 段中 `ss0` 和 `esp0` 两个字段指定。`ss0` 是任务内核态堆栈的段选择符，`esp0` 是堆栈栈底指针。因此每当任务从用户代码转移进入内核代码中执行时，任务的内核态栈总是空的。任务内核态堆栈被设置在位于其任务数据结构所在页面的末端，即与任务的任务数据结构（`task_struct`）放在同一页面内。这是在建立新任务时，`fork()` 程序在任务 `tss` 段的内核级堆栈字段（`tss.esp0` 和 `tss.ss0`）中设置的，参见 `kernel/fork.c`，93 行：

```
p->tss.esp0 = PAGE_SIZE + (long)p;
p->tss.ss0 = 0x10;
```

其中 `p` 是新任务的任务数据结构指针，`tss` 是任务状态段结构。内核为新任务申请内存用作保存其 `task_struct` 结构数据，而 `tss` 结构（段）是 `task_struct` 中的一个字段。该任务的内核堆栈段值 `tss.ss0` 也被设置成为 `0x10`（即内核数据段选择符），而 `tss.esp0` 则指向保存 `task_struct` 结构页面的末端。见图 5-25 所示。实际上 `tss.esp0` 被设置成指向该页面（外）上一字节处（图中堆栈底处）。这是因为 Intel CPU 执行堆栈操作时是先递减堆栈指针 `esp` 值，然后在 `esp` 指针处保存入栈内容。

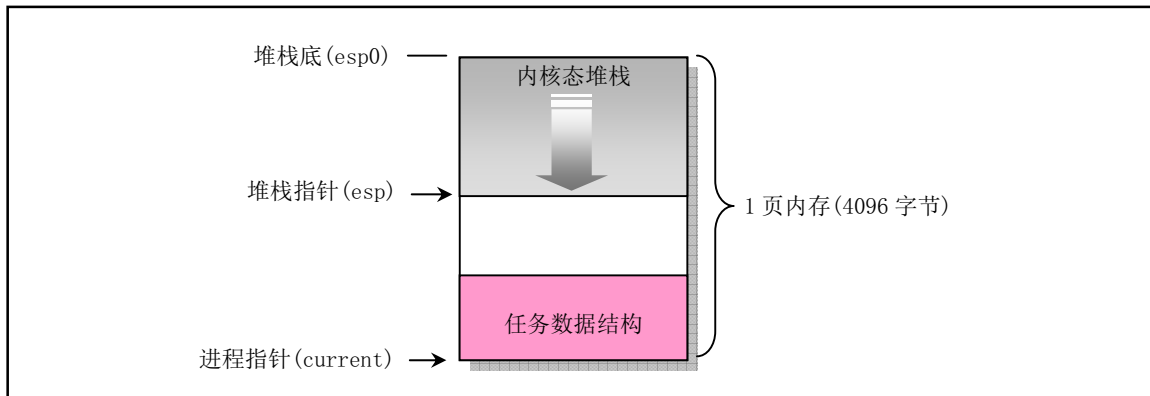


图 5-25 进程的内核态堆栈示意图

为什么从主内存区申请得来的用于保存任务数据结构的一页内存也能被设置成内核数据段中的数据呢，也即 `tss.ss0` 为什么能被设置成 `0x10` 呢？这是因为用户内核态栈仍然属于内核数据空间。我们可以从内核代码段的长度范围来说明。在 `head.s` 程序的末端，分别设置了内核代码段和数据段的描述符，段长度都被设置成了 16MB。这个长度值是 Linux 0.11 内核所能支持的最大物理内存长度（参见 `head.s`, 110 行开始的注释）。因此，内核代码可以寻址到整个物理内存范围中的任何位置，当然也包括主内存区。每当任务执行内核程序而需要使用其内核栈时，CPU 就会利用 TSS 结构把它的内核态堆栈设置成由 `tss.ss0` 和 `tss.esp0` 这两个值构成。在任务切换时，老任务的内核栈指针 `esp0` 不会被保存。对 CPU 来讲，这两个值是只读的。因此每当一个任务进入内核态执行时，其内核态堆栈总是空的。

### 任务 0 和任务 1 的堆栈

任务 0（空闲进程 `idle`）和任务 1（初始化进程 `init`）的堆栈比较特殊，需要特别予以说明。任务 0 和任务 1 的代码段和数据段相同，限长也都是 640KB，但它们被映射到不同的线性地址范围中。任务 0 的段基地址从线性地址 0 开始，而任务 1 的段基地址从 64MB 开始。但是它们全都映射到物理地址 0--640KB 范围中。这个地址范围也就是内核代码和基本数据所存放的地方。在执行了 `move_to_user_mode()` 之后，任务 0 和任务 1 的内核态堆栈分别位于各自任务数据结构所在页面的末端，而任务 0 的用户态堆栈就是前面进入保护模式后所使用的堆栈，即 `sched.c` 的 `user_stack[]` 数组的位置。由于任务 1 在创建时复制了任务 0 的用户堆栈，因此刚开始时任务 0 和任务 1 共享使用同一个用户堆栈空间。但是当任务 1 开始运行时，由于任务 1 映射到 `user_stack[]` 处的页表项被设置成只读，使得任务 1 在执行堆栈操作时将会引起写页面异常，从而内核会使用写时复制机制<sup>2</sup>为任务 1 另行分配主内存区页面作为堆栈空间使用。只有到此时，任务 1 才开始使用自己独立的用户堆栈内存页面。因此任务 0 的堆栈需要在任务 1 实际开始使用之前保持“干净”，即任务 0 此时不能使用堆栈，以确保复制的堆栈页面中不含有任务 0 的数据。

任务 0 的内核态堆栈是在其人工设置的初始化任务数据结构中指定的，而它的用户态堆栈是在执行 `move_to_user_mode()` 时，在模拟 `iret` 返回之前的堆栈中设置的，参见图 5-21 所示。我们知道，当进行特权级会发生变化的控制权转移时，目的代码会使用新特权级的堆栈，而原特权级代码堆栈指针将保留在新堆栈中。因此这里先把任务 0 用户堆栈指针压入当前处于特权级 0 的堆栈中，同时把代码指针也压入堆栈，然后执行 `IRET` 指令即可实现把控制权从特权级 0 的代码转移到特权级 3 的任务 0 代码中。在这个人工设置内容的堆栈中，原 `esp` 值被设置成仍然是 `user_stack` 中原来的位置值，而原 `ss` 段选择符被设置成 `0x17`，即设置成用户态局部表 LDT 中的数据段选择符。然后把任务 0 代码段选择符 `0x0f` 压入堆栈作为栈中原 CS 段的选择符，把下一条指令的指针作为原 EIP 压入堆栈。这样，通过执行 `IRET` 指令即可“返回”到任务 0 的代码中继续执行了。

<sup>2</sup> 关于写时复制（Copy on Write）技术的说明请参见第 10 章内存管理，10.2 节。



### 5.8.3 任务内核态堆栈与用户态堆栈之间的切换

在 Linux 0.11 系统中，所有中断服务程序都属于内核代码。如果一个中断产生时任务正在用户代码中执行，那么该中断就会引起 CPU 特权级从 3 级到 0 级的变化，此时 CPU 就会进行用户态堆栈到内核态堆栈的切换操作。CPU 会从当前任务的任务状态段 TSS 中取得新堆栈的段选择符和偏移值。因为中断服务程序在内核中，属于 0 级特权级代码，所以 48 比特的内核态堆栈指针会从 TSS 的 `ss0` 和 `esp0` 字段中获得。在定位了新堆栈（内核态堆栈）之后，CPU 就会首先把原用户态堆栈指针 `ss` 和 `esp` 压入内核态堆栈，随后把标志寄存器 `eflags` 的内容和返回位置 `cs`、`eip` 压入内核态堆栈。

内核的系统调用是一个软件中断，因此任务调用系统调用时就会进入内核并执行内核中的中断服务代码。此时内核代码就会使用该任务的内核态堆栈进行操作。同样，当进入内核程序时，由于特权级别发生了改变（从用户态转到内核态），用户态堆栈的堆栈段和堆栈指针以及 `eflags` 会被保存在任务的内核态堆栈中。而在执行 `iret` 退出内核程序返回到用户程序时，将恢复用户态的堆栈和 `eflags`。这个过程见图 5-26 所示。

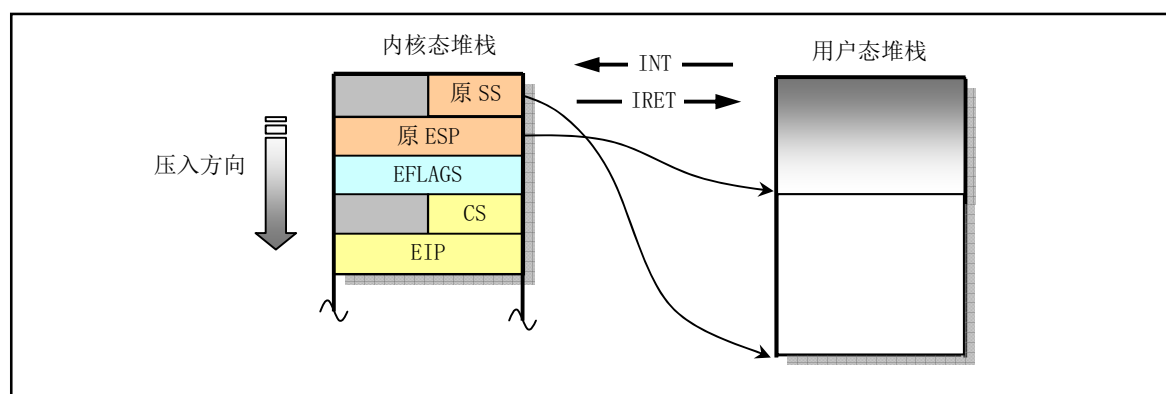


图 5-26 内核态和用户态堆栈的切换

如果一个任务正在内核态中运行，那么若 CPU 响应中断就不再需要进行堆栈切换操作，因为此时该任务运行的内核代码已经在使用内核态堆栈，并且不涉及优先级别的变化，所以 CPU 仅把 `eflags` 和中断返回指针 `cs`、`eip` 压入当前内核态堆栈，然后执行中断服务过程。

## 5.9 Linux 0.11 采用的文件系统

内核代码若要正常运行就需要文件系统的支持。用于向内核提供最基本信息和支持的是根文件系统，即 Linux 系统引导启动时，默认使用的文件系统是根文件系统。其中包括操作系统最起码的一些配置文件和命令执行程序。对于 Linux 系统中使用的 UNIX 类文件系统，其中主要包括一些规定的目录、配置文件、设备驱动程序、开发程序以及所有其他用户数据或文本文件等。其中一般都包括以下一些子目录和文件：

- `etc/` 目录主要含有一些系统配置文件；
- `dev/` 含有设备特殊文件，用于使用文件操作语句操作设备；
- `bin/` 存放系统执行程序。例如 `sh`、`mkfs`、`fdisk` 等；
- `usr/` 存放库函数、手册和其他一些文件；
- `usr/bin` 存放用户常用的普通命令；
- `var/` 用于存放系统运行时可变的数据或者是日志等信息。

存放文件系统的设备就是文件系统设备。比如，对于一般使用的 Windows2000 操作系统，硬盘 C 盘就是文件系统设备，而硬盘上按一定规则存放的文件就组成文件系统，Windows2000 有 NTFS 或 FAT32 等文件系统。而 Linux 0.11 内核所支持的文件系统是 MINIX 1.0 文件系统。目前 Linux 系统上使用最广泛的则是 ext2 或 ext3 文件系统。

对于第 1 章中介绍的在软盘上运行的 Linux 0.11 系统，它由简单的 2 张软盘组成：bootimage 盘和 rootimage 盘。bootimage 是引导启动 Image 文件，其中主要包括磁盘引导扇区代码、操作系统加载程序和内核执行代码。rootimage 就是用于向内核提供最基本支持的根文件系统。这两个盘合起来就相当于一张可启动的 DOS 操作系统盘。

当 Linux 启动盘加载根文件系统时，会根据启动盘上引导扇区第 509、510 字节处一个字（ROOT\_DEV）中的根文件系统设备号从指定的设备中加载根文件系统。如果这个设备号是 0 的话，则表示需要从引导盘所在当前驱动器中加载根文件系统。若该设备号是一个硬盘分区设备号的话，就会从该指定硬盘分区中加载根文件系统。

## 5.10 Linux 内核源代码的目录结构

由于 Linux 内核是一种单内核模式的系统，因此，内核中所有的程序几乎都有紧密的联系，它们之间的依赖和调用关系非常密切。所以在阅读一个源代码文件时往往需要参阅其他相关的文件。因此有必要在开始阅读内核源代码之前，先熟悉一下源代码文件的目录结构和安排。

这里我们首先列出 Linux 内核完整的源代码目录，包括其中的子目录。然后逐一介绍各个目录中所包含程序的主要功能，使得整个内核源代码的安排形式能在我们的头脑中建立起一个大概的框架，以便于下一章开始的源代码阅读工作。

当我们使用 tar 命令将 linux-0.11.tar.gz 解开时，内核源代码文件被放到了 linux/ 目录中。其中的目录结构见图 5-27 所示：

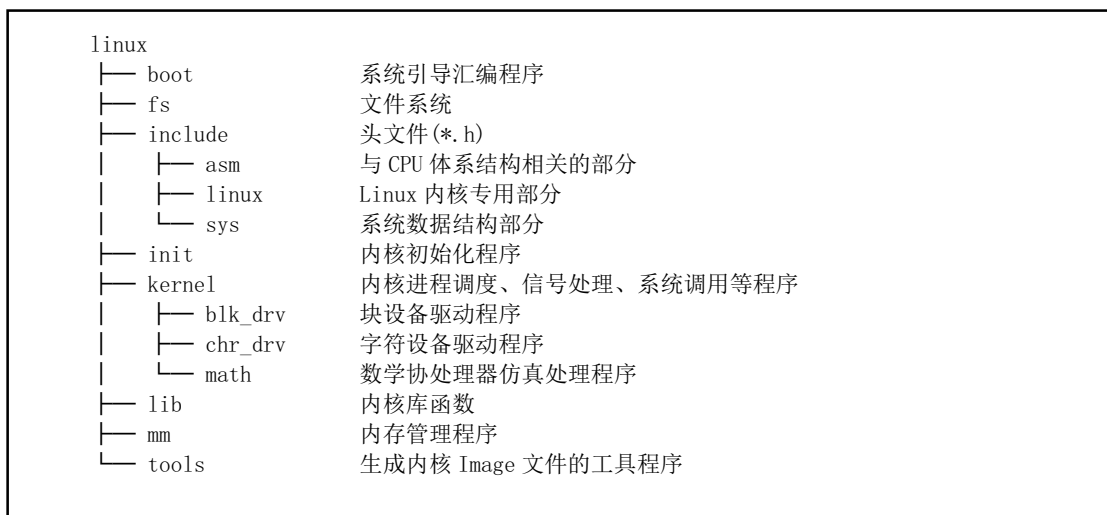


图 5-27 Linux 内核源代码目录结构

该内核版本的源代码目录中含有 14 个子目录，总共包括 102 个代码文件。下面逐个对这些子目录中的内容进行描述。

### 5.10.1 内核主目录 linux

linux 目录是源代码的主目录，在该主目录中除了包括所有的 14 个子目录以外，还含有唯一的一个

**Makefile** 文件。该文件是编译辅助工具软件 **make** 的参数配置文件。**make** 工具软件的主要用途是通过识别哪些文件已被修改过，从而自动地决定在一个含有多个源程序文件的程序系统中哪些文件需要被重新编译。因此，**make** 工具软件是程序项目的管理软件。

**linux** 目录下的这个 **Makefile** 文件还嵌套地调用了所有子目录中包含的 **Makefile** 文件，这样，当 **linux** 目录（包括子目录）下的任何文件被修改过时，**make** 都会对其进行重新编译。因此为了编译整个内核所有的源代码文件，只要在 **linux** 目录下运行一次 **make** 软件即可。

### 5.10.2 引导启动程序目录 boot

**boot** 目录中含有 3 个汇编语言文件，是内核源代码文件中最先被编译的程序。这 3 个程序完成的主要功能是当计算机加电时引导内核启动，将内核代码加载到内存中，并做一些进入 32 位保护运行方式前的系统初始化工作。其中 **bootsect.s** 和 **setup.s** 程序需要使用 **as86** 软件来编译，使用的是 **as86** 的汇编语言格式（与微软的类似），而 **head.s** 需要用 **GNU as** 来编译，使用的是 **AT&T** 格式的汇编语言。这两种汇编语言在下一章的代码注释里以及代码列表后面的说明中会有简单的介绍。

**bootsect.s** 程序是磁盘引导块程序，编译后会驻留在磁盘的第一个扇区中（引导扇区，0 磁道（柱面），0 磁头，第 1 个扇区）。在 PC 机加电 **ROM BIOS** 自检后，将被 **BIOS** 加载到内存 **0x7C00** 处进行执行。

**setup.s** 程序主要用于读取机器的硬件配置参数，并把内核模块 **system** 移动到适当的内存位置处。

**head.s** 程序会被编译连接在 **system** 模块的最前部分，主要进行硬件设备的探测设置和内存管理页面的初始设置工作。

### 5.10.3 文件系统目录 fs

**Linux 0.11** 内核的文件系统采用了 1.0 版的 **MINIX** 文件系统，这是由于 **Linux** 是在 **MINIX** 系统上开发的，采用 **MINIX** 文件系统便于进行交叉编译，并且可以从 **MINIX** 中加载 **Linux** 分区。虽然使用的是 **MINIX** 文件系统，但 **Linux** 对其处理方式与 **MINIX** 系统不同。主要的区别在于 **MINIX** 对文件系统采用单线程处理方式，而 **Linux** 则采用了多线程方式。由于采用了多线程处理方式，**Linux** 程序就必须处理多线程带来的竞争条件、死锁等问题，因此 **Linux** 文件系统代码要比 **MINIX** 系统的复杂得多。为了避免竞争条件的发生，**Linux** 系统对资源分配进行了严格地检查，并且在内核模式下运行时，如果任务没有主动睡眠（调用 **sleep()**），就不让内核切换任务。

**fs/**目录是文件系统实现程序的目录，共包含 17 个 C 语言程序。这些程序之间的主要引用关系见图 5-28 所示。图中每个方框代表一个文件，从上到下按基本引用关系放置。其中各文件名均略去了后缀 **.c**，虚框中是程序文件不属于文件系统，带箭头的线条表示引用关系，粗线条表示有相互引用关系。

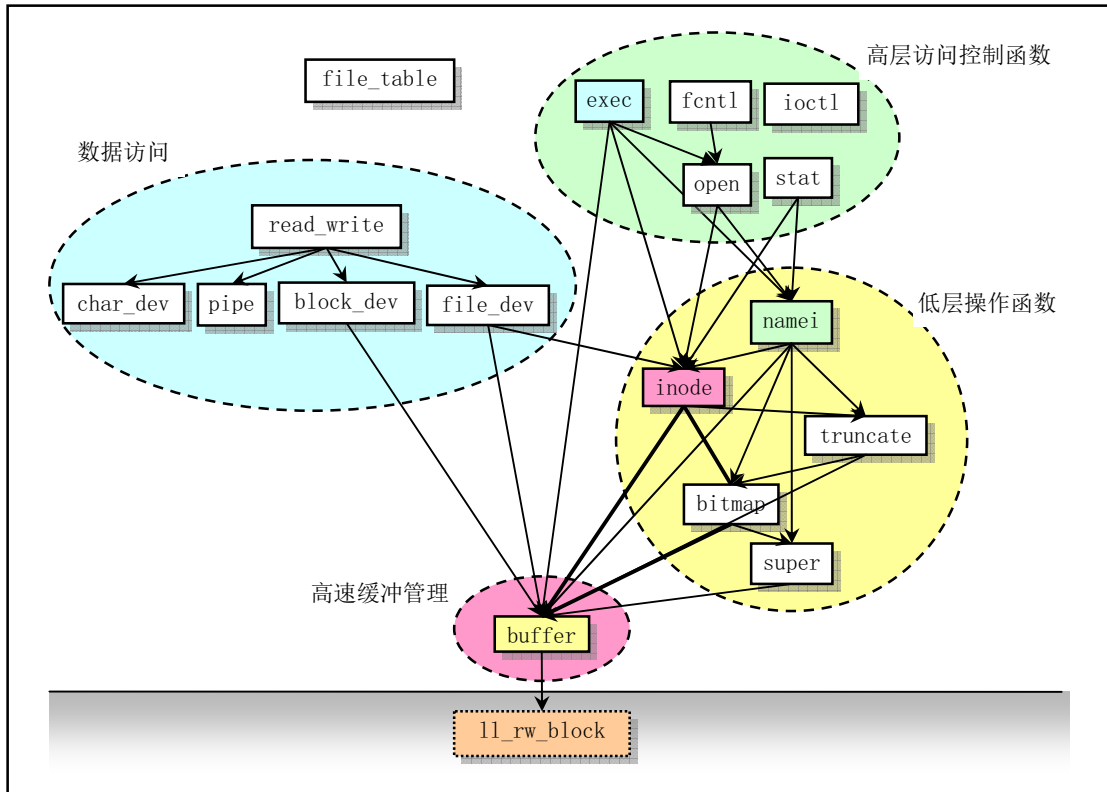


图 5-28 fs 目录中各程序中函数之间的引用关系。

由图可以看出，该目录中的程序可以划分成四个部分：高速缓冲区管理、低层文件操作、文件数据访问和文件高层函数，在对本目录中文件进行注释说明时，我们也将分成这四个部分来描述。

对于文件系统，我们可以将它看成是内存高速缓冲区的扩展部分。所有对文件系统中数据的访问，都需要首先读取到高速缓冲区中。本目录中的程序主要用来管理高速缓冲区中缓冲块的使用分配和块设备上的文件系统。管理高速缓冲区的程序是 `buffer.c`，而其他程序则主要都是用于文件系统管理。

在 `file_table.c` 文件中，目前仅定义了一个文件句柄（描述符）结构数组。`ioctl.c` 文件将引用 `kernel/chr_drv/tty.c` 中的函数，实现字符设备的 io 控制功能。`exec.c` 程序主要包含一个执行程序函数 `do_execve()`，它是所有 `exec()` 函数簇中的主要函数。`fcntl.c` 程序用于实现文件 i/o 控制的系统调用函数。`read_write.c` 程序用于实现文件读/写和定位三个系统调用函数。`stat.c` 程序中实现了两个获取文件状态的系统调用函数。`open.c` 程序主要包含实现修改文件属性和创建与关闭文件的系统调用函数。

`char_dev.c` 主要包含字符设备读写函数 `rw_char()`。`pipe.c` 程序中包含管道读写函数和创建管道的系统调用。`file_dev.c` 程序中包含基于 i 节点和描述符结构的文件读写函数。`namei.c` 程序主要包括文件系统中目录名和文件名的操作函数和系统调用函数。`block_dev.c` 程序包含块数据读和写函数。`inode.c` 程序中包含针对文件系统 i 节点操作的函数。`truncate.c` 程序用于在删除文件时释放文件所占用的设备数据空间。`bitmap.c` 程序用于处理文件系统中 i 节点和逻辑数据块的位图。`super.c` 程序中包含对文件系统超级块的处理函数。`buffer.c` 程序主要用于对内存高速缓冲区进行处理。虚框中的 `ll_rw_block` 是块设备的底层读函数，它并不在 fs 目录中，而是 `kernel/blk_drv/ll_rw_block.c` 中的块设备读写驱动函数。放在这里只是让我们清楚的看到，文件系统对于块设备中数据的读写，都需要通过高速缓冲区与块设备的驱动程序（`ll_rw_block()`）来操作来进行，文件系统程序集本身并不直接与块设备的驱动程序打交道。

在对程序进行注释过程中，我们将另外给出这些文件中各个主要函数之间的调用层次关系。

### 5.10.4 头文件主目录 include

头文件目录中总共有 32 个.h 头文件。其中主目录下有 13 个，asm 子目录中有 4 个，linux 子目录中有 10 个，sys 子目录中有 5 个。这些头文件各自的功能见如下简述，具体的作用和所包含的信息请参见对头文件的注释一章。

<a.out.h>	a.out 头文件，定义了 a.out 执行文件格式和一些宏。
<const.h>	常数符号头文件，目前仅定义了 i 节点中 i_mode 字段的各标志位。
<ctype.h>	字符类型头文件。定义了一些有关字符类型判断和转换的宏。
<errno.h>	错误号头文件。包含系统中各种出错号。(Linus 从 minix 中引进的)。
<fcntl.h>	文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
<signal.h>	信号头文件。定义信号符号常量，信号结构以及信号操作函数原型。
<stdarg.h>	标准参数头文件。以宏的形式定义变量参数列表。主要说明了 va_list 类型和三个宏 (va_start, va_arg 和 va_end)，用于 vsprintf、vprintf、vfprintf 函数。
<stddef.h>	标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)。
<string.h>	字符串头文件。主要定义了一些有关字符串操作的嵌入函数。
<termios.h>	终端输入输出函数头文件。主要定义控制异步通信口的终端接口。
<time.h>	时间类型头文件。其中最主要定义了 tm 结构和一些有关时间的函数原形。
<unistd.h>	Linux 标准头文件。定义了各种符号常数和类型，并声明了各种函数。如定义了 __LIBRARY__，则还包括系统调用号和内嵌汇编 _syscall0() 等。
<utime.h>	用户时间头文件。定义了访问和修改时间结构以及 utime() 原型。

#### 体系结构相关头文件子目录 include/asm

这些头文件主要定义了一些与 CPU 体系结构密切相关的数据结构、宏函数和变量。共 4 个文件。

<asm/io.h>	io 头文件。以宏的嵌入汇编程序形式定义对 io 端口操作的函数。
<asm/memory.h>	内存拷贝头文件。含有 memcpy() 嵌入式汇编宏函数。
<asm/segment.h>	段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
<asm/system.h>	系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。

#### Linux 内核专用头文件子目录 include/linux

<linux/config.h>	内核配置头文件。定义键盘语言和硬盘类型 (HD_TYPE) 可选项。
<linux/fdreg.h>	软驱头文件。含有软盘控制器参数的一些定义。
<linux/fs.h>	文件系统头文件。定义文件表结构 (file, buffer_head, m_inode 等)。
<linux/hdreg.h>	硬盘参数头文件。定义访问硬盘寄存器端口，状态码，分区表等信息。
<linux/head.h>	head 头文件，定义了段描述符的简单结构，和几个选择符常量。
<linux/kernel.h>	内核头文件。含有一些内核常用函数的原形定义。
<linux/mm.h>	内存管理头文件。含有页面大小定义和一些页面释放函数原型。
<linux/sched.h>	调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
<linux/sys.h>	系统调用头文件。含有 72 个系统调用 C 函数处理程序，以 'sys_' 开头。
<linux/tty.h>	tty 头文件，定义了有关 tty_io，串行通信方面的参数、常数。

#### 系统专用数据结构子目录 include/sys

<sys/stat.h>	文件状态头文件。含有文件或文件系统状态结构 stat{} 和常量。
<sys/times.h>	定义了进程中运行时间结构 tms 以及 times() 函数原型。
<sys/types.h>	类型头文件。定义了基本的系统数据类型。
<sys/utsname.h>	系统名称结构头文件。
<sys/wait.h>	等待调用头文件。定义系统调用 wait() 和 waitpid() 及相关常数符号。

### 5.10.5 内核初始化程序目录 init

该目录中仅包含一个文件 `main.c`。用于执行内核所有的初始化工作，然后移到用户模式创建新进程，并在控制台设备上运行 `shell` 程序。

程序首先根据机器内存的多少对缓冲区内容量进行分配，如果还设置了要使用虚拟盘，则在缓冲区内内存后面也为它留下空间。之后就进行所有硬件的初始化工作，包括人工创建第一个任务（task 0），并设置了中断允许标志。在执行从核心态移到用户态之后，系统第一次调用创建进程函数 `fork()`，创建一个用于运行 `init()` 的进程，在该子进程中，系统将进行控制台环境设置，并且在生成一个子进程用来运行 `shell` 程序。

### 5.10.6 内核程序主目录 kernel

`linux/kernel` 目录中共包含 12 个代码文件和一个 `Makefile` 文件，另外还有 3 个子目录。所有处理任务的程序都保存在 `kernel/` 目录中，其中包括象 `fork`、`exit`、调度程序以及一些系统调用程序等。还包括处理中断异常和陷阱的处理过程。子目录中包括了低层的设备驱动程序，如 `get_hd_block` 和 `tty_write` 等。由于这些文件中代码之间调用关系复杂，因此这里就不详细列出各文件之间的引用关系图，但仍然可以进行大概分类，见图 5-29 所示。

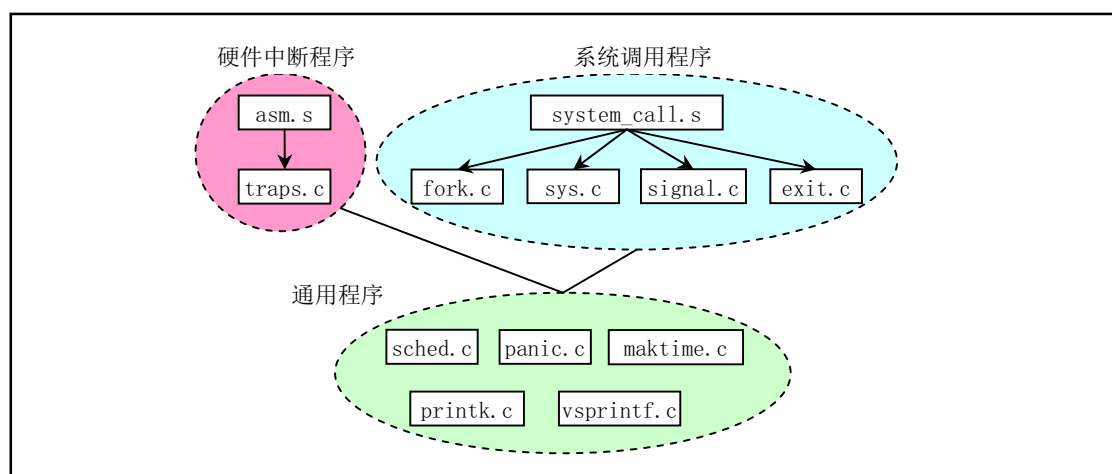


图 5-29 各文件的调用层次关系

`asm.s` 程序是用于处理系统硬件异常所引起的中断，对各硬件异常的实际处理程序则是在 `traps.c` 文件中，在各个中断处理过程中，将分别调用 `traps.c` 中相应的 C 语言处理函数。

`exit.c` 程序主要包括用于处理进程终止的系统调用。包含进程释放、会话（进程组）终止和程序退出处理函数以及杀死进程、终止进程、挂起进程等系统调用函数。

`fork.c` 程序给出了 `sys_fork()` 系统调用中使用了两个 C 语言函数：`find_empty_process()` 和 `copy_process()`。

`mktime.c` 程序包含一个内核使用的时间函数 `mktime()`，用于计算从 1970 年 1 月 1 日 0 时起到开机当日的秒数，作为开机秒时间。仅在 `init/main.c` 中被调用一次。

`panic.c` 程序包含一个显示内核出错信息并停机的函数 `panic()`。

`printk.c` 程序包含一个内核专用信息显示函数 `printk()`。

`sched.c` 程序中包括有关调度的基本函数（`sleep_on`、`wakeup`、`schedule` 等）以及一些简单的系统调用函数。另外有几个与定时相关的软盘操作函数。

`signal.c` 程序中包括了有关信号处理的 4 个系统调用以及一个在对应的中断处理程序中处理信号的函

数 `do_signal()`。

`sys.c` 程序包括很多系统调用函数，其中有些还没有实现。

`system_call.s` 程序实现了 Linux 系统调用（`int 0x80`）的接口处理过程，实际的处理过程则包含在各系统调用相应的 C 语言处理函数中，这些处理函数分布在整个 Linux 内核代码中。

`vsprintf.c` 程序实现了现在已经归入标准库函数中的字符串格式化函数。

### 块设备驱动程序子目录 `kernel/blk_drv`

通常情况下，用户是通过文件系统来访问设备的，因此设备驱动程序为文件系统实现了调用接口。在使用块设备时，由于其数据吞吐量大，为了能够高效率地使用块设备上的数据，在用户进程与块设备之间使用了高速缓冲机制。在访问块设备上的数据时，系统首先以数据块的形式把块设备上的数据读入到高速缓冲区中，然后再提供给用户。`blk_drv` 子目录共包含 4 个 c 文件和 1 个头文件。头文件 `blk.h` 由于是块设备程序专用的，所以与 C 文件放在一起。这几个文件之间的大致关系，见图 5-30 所示。

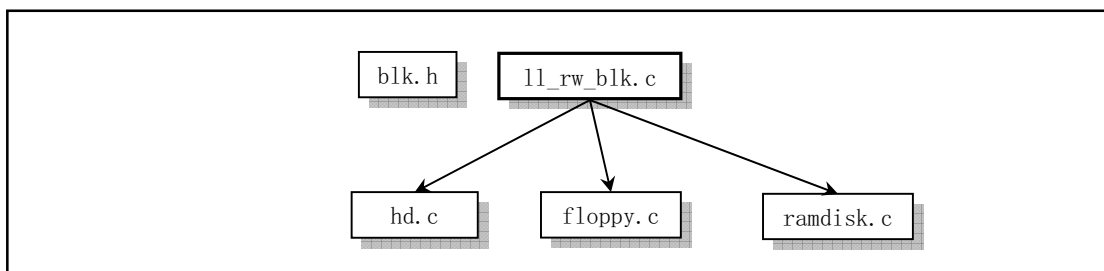


图 5-30 `blk_drv` 目录中文件的层次关系。

`blk.h` 中定义了 3 个 C 程序中共用的块设备结构和数据块请求结构。`hd.c` 程序主要实现对硬盘数据块进行读/写的底层驱动函数，主要是 `do_hd_request()` 函数；`floppy.c` 程序中主要实现了对软盘数据块的读/写驱动函数，主要是 `do_fd_request()` 函数。`ll_rw_blk.c` 中程序实现了低层块设备数据读/写函数 `ll_rw_block()`，内核中所有其他程序都是通过该函数对块设备进行数据读写操作。你将看到该函数在许多访问块设备数据的地方被调用，尤其是在高速缓冲区处理文件 `fs/buffer.c` 中。

### 字符设备驱动程序子目录 `kernel/chr_drv`

字符设备程序子目录共含有 4 个 C 语言程序和 2 个汇编程序文件。这些文件实现了对串行端口 `rs-232`、串行终端、键盘和控制台终端设备的驱动。图 5-31 是这些文件之间的大致调用层次关系。

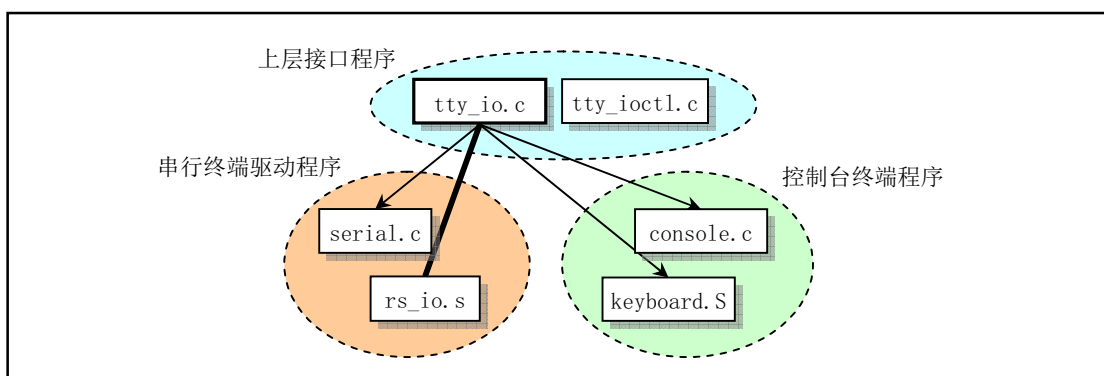


图 5-31 字符设备程序之间的关系示意图

`tty_io.c` 程序中包含 `tty` 字符设备读函数 `tty_read()` 和写函数 `tty_write()`，为文件系统提供了上层访问接口。另外还包括在串行中断处理过程中调用的 C 函数 `do_tty_interrupt()`，该函数将会在中断类型为读



字符的处理中被调用。

`console.c` 文件主要包含控制台初始化程序和控制台写函数 `con_write()`，用于被 `tty` 设备调用。还包含对显示器和键盘中断的初始化设置程序 `con_init()`。

`rs_io.s` 汇编程序用于实现两个串行接口的中断处理程序。该中断处理程序会根据从中断标识寄存器（端口 `0x3fa` 或 `0x2fa`）中取得的 4 种中断类型分别进行处理，并在处理中断类型为读字符的代码中调用 `do_tty_interrupt()`。

`serial.c` 用于对异步串行通信芯片 UART 进行初始化操作，并设置两个通信端口的中断向量。另外还包括 `tty` 用于往串口输出的 `rs_write()` 函数。

`tty_ioctl.c` 程序实现了 `tty` 的 `io` 控制接口函数 `tty_ioctl()` 以及对 `termio(s)` 终端 `io` 结构的读写函数，并会在实现系统调用 `sys_ioctl()` 的 `fs/ioctl.c` 程序中被调用。

`keyboard.S` 程序主要实现了键盘中断处理过程 `keyboard_interrupt`。

### 协处理器仿真和操作程序子目录 `kernel/math`

该子目录中目前仅有一个 C 程序 `math_emulate.c`。其中的 `math_emulate()` 函数是中断 `int7` 的中断处理程序调用的 C 函数。当机器中没有数学协处理器，而 CPU 却又执行了协处理器的指令时，就会引发该中断。因此，使用该中断就可以用软件来仿真协处理器的功能。本书所讨论的内核版本还没有包含有关协处理器的仿真代码。本程序中只是打印一条出错信息，并向用户程序发送一个协处理器错误信号 `SIGFPE`。

## 5.10.7 内核库函数目录 `lib`

与普通用户程序不同，内核代码不能使用标准 C 函数库及其他一些函数库。主要原因是由于完整的 C 函数库很大。因此在内核源代码中有专门一个 `lib/` 目录提供内核需要用到的一些函数。内核函数库用于为内核初始化程序 `init/main.c` 运行在用户态的进程（进程 0、1）提供调用支持。它与普通静态库的实现方法完全一样。读者可从中了解一般 `libc` 函数库的基本组成原理。在 `lib/` 目录中共有 12 个 C 语言文件，除了一个由 `tytso` 编制的 `malloc.c` 程序较长以外，其他的程序很短，有的只有一二行代码，实现了一些系统调用的接口函数。

这些文件中主要包括有退出函数 `_exit()`、关闭文件函数 `close(fd)`、复制文件描述符函数 `dup()`、文件打开函数 `open()`、写文件函数 `write()`、执行程序函数 `execve()`、内存分配函数 `malloc()`、等待子进程状态函数 `wait()`、创建会话系统调用 `setsid()` 以及在 `include/string.h` 中实现的所有字符串操作函数。

## 5.10.8 内存管理程序目录 `mm`

该目录包括 2 个代码文件。主要用于管理程序对主内存区的使用，实现了进程逻辑地址到线性地址以及线性地址到主内存区中物理内存地址的映射，通过内存的分页管理机制，在进程的虚拟内存页与主内存区的物理内存页之间建立了对应关系。

Linux 内核对内存的处理使用了分页和分段两种方式。首先是将 386 的 4G 虚拟地址空间分割成 64 个段，每个段 64MB。所有内核程序占用其中第一个段，并且物理地址与该段线性地址相同。然后每个任务分配一个段使用。分页机制用于把指定的物理内存页面映射到段内，检测 `fork` 创建的任何重复的拷贝，并执行写时复制机制。

`page.s` 文件包括内存页面异常中断（`int 14`）处理程序，主要用于处理程序由于缺页而引起的页异常中断和访问非法地址而引起的页保护。

`memory.c` 程序包括对内存进行初始化的函数 `mem_init()`，由 `page.s` 的内存处理中断过程调用的 `do_no_page()` 和 `do_wp_page()` 函数。在创建新进程而执行复制进程操作时，即使用该文件中的内存处理函数来分配管理内存空间。

### 5.10.9 编译内核工具程序目录 tools

该目录下的 `build.c` 程序用于将 Linux 各个目录中被分别编译生成的目标代码连接合并成一个可运行的内核映像文件 `image`。其具体的功能可参见下一章内容。

## 5.11 内核系统与应用程序的关系

在 Linux 系统中，内核为用户程序提供了两方面的支持。其一是系统调用接口（在第 5 章中说明），也即中断调用 `int 0x80`；另一方面是通过开发环境库函数或内核库函数（在第 12 章中说明）与内核进行信息交流。不过内核库函数仅供内核创建的任务 0 和任务 1 使用，它们最终还是去调用系统调用。因此内核对所有用户程序或进程实际上只提供系统调用这一种统一的接口。`lib/`目录下内核库函数代码的实现方法与基本 C 函数库 `libc` 中类似函数的实现方法基本相同，为了使用内核资源，最终都是通过内嵌汇编代码调用了内核系统调用功能，参见图 5-4 所示。

系统调用主要提供给系统软件编程或者用于库函数的实现。而一般用户开发的程序则是通过调用象 `libc` 等库中函数来访问内核资源。这些库中的函数或资源通常被称为应用程序编程接口（API）。其中定义了应用程序使用的一组标准编程接口。通过调用这些库中的程序，应用程序代码能够完成各种常用工作，例如，打开和关闭对文件或设备的访问、进行科学计算、出错处理以及访问组和用户标识号 ID 等系统信息。

在 UNIX 类操作系统中，最为普遍使用的是基于 POSIX 标准的 API 接口。Linux 当然也不例外。API 与系统调用的区别在于：为了实现某一应用程序接口标准，例如 POSIX，其中的 API 可以与一个系统调用对应，也可能由几个系统调用的功能共同实现。当然某些 API 函数可能根本就不需要使用系统调用，即不使用内核功能。因此函数库可以看作是实现 POSIX 标准的主体界面，应用程序不用管它与系统调用之间到底存在什么关系。无论一个操作系统提供的系统调用是多么得不同，但只要它遵循同一个 API 标准，那么应用程序就可以在这些操作系统之间具有可移植性。

系统调用是内核与外界接口的最高层。在内核中，每个系统调用都有一个序列号（在 `include/unistd.h` 头文件中定义），并且常以宏的形式实现。应用程序不应该直接使用系统调用，因为这样的话，程序的移植性就不好了。因此目前 Linux 标准库 LSB（Linux Standard Base）和许多其他标准都不允许应用程序直接访问系统调用宏。系统调用的有关文档可参见 Linux 操作系统的在线手册的第 2 部分。

库函数一般包括 C 语言没有提供的执行高级功能的用户级函数，例如输入/输出和字符串处理函数。某些库函数只是系统调用的增强功能版。例如，标准 I/O 库函数 `fopen` 和 `fclose` 提供了与系统调用 `open` 和 `close` 类似的功能，但却是在更高的层次上。在这种情况下，系统调用通常能提供比库函数略微好一些的性能，但是库函数却能提供更多的功能，而且更具检错能力。系统提供的库函数有关文档可参见操作系统的在线手册第 3 部分。

## 5.12 linux/Makefile 文件

从本节起，我们开始对内核源代码文件进行注释。首先注释 `linux` 目录下遇到的第一个文件 `Makefile`。后续章节将按照这里类似的描述结构进行注释。

### 5.12.1 功能描述

`Makefile` 文件相当于程序编译过程中的批处理文件。是工具程序 `make` 运行时的输入数据文件。只要在含有 `Makefile` 的当前目录中键入 `make` 命令，它就会依据 `Makefile` 文件中的设置对源程序或目标代码文件进行编译、连接或进行安装等活动。

**make** 工具程序能自动地确定一个大程序系统中那些程序文件需要被重新编译，并发出命令对这些程序文件进行编译。在使用 **make** 之前，需要编写 **Makefile** 信息文件，该文件描述了整个程序包中各程序之间的关系，并针对每个需要更新的文件给出具体的控制命令。通常，执行程序是根据其目标文件进行更新的，而这些目标文件则是由编译程序创建的。一旦编写好一个合适的 **Makefile** 文件，那么在你每次修改过程序系统中的某些源代码文件后，执行 **make** 命令就能进行所有必要的重新编译工作。**make** 程序是使用 **Makefile** 数据文件和代码文件的最后修改时间(last-modification time)来确定那些文件需要进行更新，对于每一个需要更新的文件它会根据 **Makefile** 中的信息发出相应的命令。在 **Makefile** 文件中，开头为#的行是注释行。文件开头部分的=赋值语句定义了一些参数或命令的缩写。

这个 **Makefile** 文件的主要作用是指示 **make** 程序最终使用独立编译连接成的 **tools/**目录中的 **build** 执行程序将所有内核编译代码连接和合并成一个可运行的内核映像文件 **image**。具体是对 **boot/**中的 **bootsect.s**、**setup.s** 使用 8086 汇编器进行编译，分别生成各自的执行模块。再对源代码中的其他所有程序使用 GNU 的编译器 **gcc/gas** 进行编译，并链接模块 **system**。最后再用 **build** 工具将这三块组合成一个内核映像文件 **image**。**build** 是由 **tools/build.c** 源程序编译而成的一个独立的执行程序，它本身并没有被编译链接到内核代码中。基本编译连接/组合结构如图 5-32 所示。

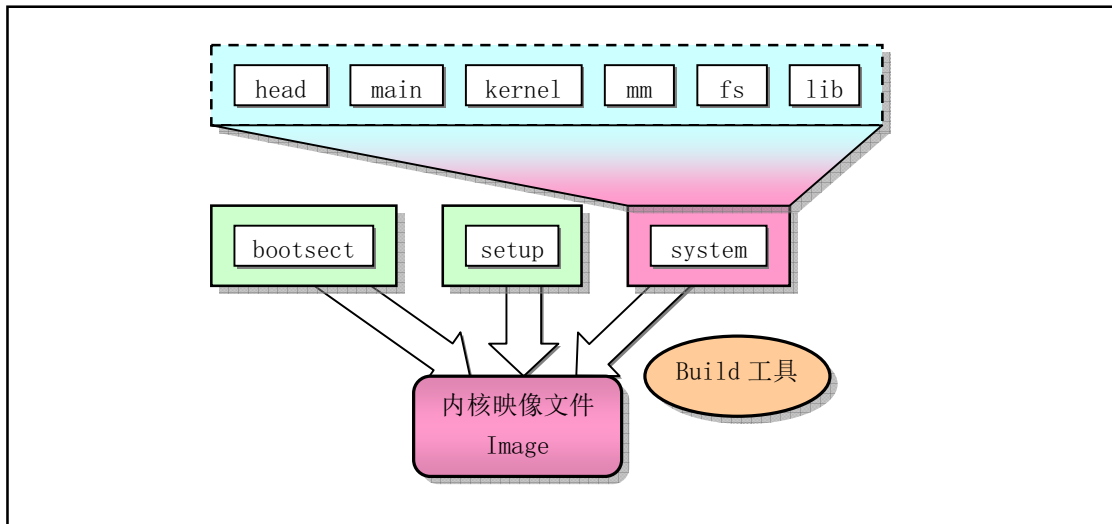


图 5-32 内核编译连接/组合结构

## 5.12.2 代码注释

程序 5-1 linux/Makefile 文件

```

1 #
2 # if you want the ram-disk device, define this to be the
3 # size in blocks.
4 #
5 # 如果你要使用 RAM 盘 (RAMDISK) 设备的话就定义块的大小。这里默认 RAMDISK 没有定义（注释掉了），
6 # 否则 gcc 编译时会带有选项 '-DRAMDISK=512'，参见第 13 行。
7 RAMDISK = #-DRAMDISK=512
8
9 AS86      =as86 -O -a      # 8086 汇编编译器和连接器，见列表后的介绍。后带的参数含义分别
10 LD86      =ld86 -O        # 是：-O 生成 8086 目标程序；-a 生成与 gas 和 gld 部分兼容的代码。
11
12 AS        =gas            # GNU 汇编编译器和连接器，见列表后的介绍。

```

```

11 LD      =gld
# 下面是 GNU 链接器 gld 运行时用到的选项。含义是：-s 输出文件中省略所有的符号信息；-x 删除
# 所有局部符号；-M 表示需要在标准输出设备(显示器)上打印连接映像(link map)，是指由连接程序
# 产生的一种内存地址映像，其中列出了程序段装入到内存中的位置信息。具体来讲有如下信息：
# • 目标文件及符号信息映射到内存中的位置；
# • 公共符号如何放置；
# • 连接中包含的所有文件成员及其引用的符号。
12 LDFLAGS =-s -x -M

# gcc 是 GNU C 程序编译器。对于 UNIX 类的脚本(script)程序而言，在引用定义的标识符时，需在前
# 面加上$符号并用括号括住标识符。
13 CC      =gcc $(RAMDISK)

# 下面指定 gcc 使用的选项。前一行最后的'\ '符号表示下一行是续行。选项含义为：-Wall 打印所有
# 警告信息；-O 对代码进行优化。'-f' 标志'指定与机器无关的编译标志。其中-fstrength-reduce 用
# 于优化循环语句；-fcombine-regs 用于指明编译器在组合编译阶段把复制一个寄存器到另一个寄存
# 器的指令组合在一起。-fomit-frame-pointer 指明对于无需帧指针(Frame pointer)的函数不要
# 把帧指针保留在寄存器中。这样在函数中可以避免对帧指针的操作和维护。-mstring-insns 是
# Linus 在学习 gcc 编译器时为 gcc 增加的选项，用于 gcc-1.40 在复制结构等操作时使用 386 CPU 的
# 字符串指令，可以去掉。
14 CFLAGS =-Wall -O -fstrength-reduce -fomit-frame-pointer \
15 -fcombine-regs -mstring-insns

# 下面 cpp 是 gcc 的前(预)处理器程序。前处理器用于进行程序中的宏替换处理、条件编译处理以及
# 包含进指定文件的内容，即把使用'#include'指定的文件包含进来。源程序文件中所有以符号'#'
# 开始的行均需要由前处理器进行处理。程序中所有'#define'定义的宏都会使用其定义部分替换掉。
# 程序中所有'#if'、'#ifdef'、'#ifndef'和'#endif'等条件判别行用于确定是否包含其指定范围中
# 的语句。
# '-nostdinc -linclude' 含义是不要搜索标准头文件目录中的文件，即不用系统/usr/include/目录
# 下的头文件，而是使用'-I'选项指定的目录或者是在当前目录里搜索头文件。
16 CPP     =cpp -nostdinc -linclude
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 #
# ROOT_DEV 指定在创建内核映像(image)文件时所使用的默认根文件系统所
# 在的设备，这可以是软盘(FLOPPY)、/dev/xxxx 或者干脆空着，空着时
# build 程序(在 tools/目录中)就使用默认值/dev/hd6。
#
# 这里/dev/hd6 对应第 2 个硬盘的第 1 个分区。这是 Linus 开发 Linux 内核时自己的机器上根
# 文件系统所在的分区位置。
23 ROOT_DEV=/dev/hd6
24
# 下面是 kernel 目录、mm 目录和 fs 目录所产生的目标代码文件。为了方便引用在这里将它们用
# ARCHIVES (归档文件)标识符表示。
25 ARCHIVES=kernel/kernel.o mm/mm.o fs/fs.o

# 块和字符设备库文件。'.a'表示该文件是个归档文件，也即包含有许多可执行二进制代码子程序
# 集合的库文件，通常是用 GNU 的 ar 程序生成。ar 是 GNU 的二进制文件处理程序，用于创建、修改
# 以及从归档文件中抽取文件。
26 DRIVERS =kernel/blk_drv/blk_drv.a kernel/chr_drv/chr_drv.a

```

```

27 MATH      =kernel/math/math.a      # 数学运算库文件。
28 LIBS      =lib/lib.a                # 由 lib/目录中的文件所编译生成的通用库文件。
29
# 下面是 make 老式的隐式后缀规则。该行指示 make 利用下面的命令将所有的'.c' 文件编译生成'.s'
# 汇编程序。':' 表示下面是该规则的命令。整句表示让 gcc 采用前面 CFLAGS 所指定的选项以及仅使
# 用 include/目录中的头文件，在适当地编译后不进行汇编就停止 (-S)，从而产生与输入的各个 C
# 文件对应的汇编语言形式的代码文件。默认情况下所产生的汇编程序文件是原 C 文件名去掉'.c' 后
# 再加上'.s' 后缀。'-o' 表示其后是输出文件的形式。其中'${*.s}' (或'${@}') 是自动目标变量，'${<'
# 代表第一个先决条件，这里即是符合条件 '*.c' 的文件。
# 下面这 3 个不同规则分别用于不同的操作要求。若目标是.s 文件，而源文件是.c 文件则会使
# 用第一个规则；若目标是.o，而原文件是.s，则使用第 2 个规则；若目标是.o 文件而原文件
# 是.c 文件，则可直接使用第 3 个规则。
30 .c.s:
31     $(CC) $(CFLAGS) \
32     -nostdinc -Iinclude -S -o ${*.s} <

# 表示将所有.s 汇编程序文件编译成.o 目标文件。整句表示使用 gas 编译器将汇编程序编译成.o
# 目标文件。-c 表示只编译或汇编，但不进行连接操作。
33 .s.o:
34     $(AS) -c -o ${*.o} <
# 类似上面，*.c 文件→*.o 目标文件。整句表示使用 gcc 将 C 语言文件编译成目标文件但不连接。
35 .c.o:
36     $(CC) $(CFLAGS) \
37     -nostdinc -Iinclude -c -o ${*.o} <
38

# 下面'all' 表示创建 Makefile 所知的最顶层的目标。这里即是 Image 文件。这里生成的 Image 文件
# 即是引导启动盘映像文件 bootimage。若将其写入软盘就可以使用该软盘引导 Linux 系统了。在
# Linux 下将 Image 写入软盘的命令参见 46 行。DOS 系统下可以使用软件 rawrite.exe。
39 all:    Image
40
# 说明目标 (Image 文件) 是由冒号后面的 4 个元素产生，分别是 boot/目录中的 bootsect 和 setup
# 文件、tools/目录中的 system 和 build 文件。42--43 行这是执行的命令。42 行表示使用 tools 目
# 录下的 build 工具程序 (下面会说明如何生成) 将 bootsect、setup 和 system 文件以$(ROOT_DEV)
# 为根文件系统设备组装成内核映像文件 Image。第 43 行的 sync 同步命令是迫使缓冲块数据立即写盘
# 并更新超级块。
41 Image: boot/bootsect boot/setup tools/system tools/build
42     tools/build boot/bootsect boot/setup tools/system $(ROOT_DEV) > Image
43     sync
44
# 表示 disk 这个目标要由 Image 产生。dd 为 UNIX 标准命令：复制一个文件，根据选项进行转换和格
# 式化。bs=表示一次读/写的字节数。if=表示输入的文件，of=表示输出到的文件。这里/dev/PS0 是
# 指第一个软盘驱动器 (设备文件)。在 Linux 系统下使用/dev/fd0。
45 disk: Image
46     dd bs=8192 if=Image of=/dev/PS0
47
48 tools/build: tools/build.c          # 由 tools 目录下的 build.c 程序生成执行程序 build。
49     $(CC) $(CFLAGS) \
50     -o tools/build tools/build.c    # 编译生成执行程序 build 的命令。
51
52 boot/head.o: boot/head.s            # 利用上面给出的.s.o 规则生成 head.o 目标文件。
53
# 表示 tools 目录中的 system 文件要由冒号右边所列的元素生成。56--61 行是生成 system 的命令。

```

```

# 最后的 > System.map 表示 gld 需要将连接映像重定向存放在 System.map 文件中。
# 关于 System.map 文件的用途参见注释后的说明。
54 tools/system:  boot/head.o init/main.o \
55                $(ARCHIVES) $(DRIVERS) $(MATH) $(LIBS)
56                $(LD) $(LDFLAGS) boot/head.o init/main.o \
57                $(ARCHIVES) \
58                $(DRIVERS) \
59                $(MATH) \
60                $(LIBS) \
61                -o tools/system > System.map
62
# 数学协处理函数文件 math.a 由 64 行上的命令实现：进入 kernel/math/目录；运行 make 工具程序。
63 kernel/math/math.a:
64     (cd kernel/math; make)
65
66 kernel/blk_drv/blk_drv.a:          # 生成块设备库文件 blk_drv.a，其中含有可重定位目标文件。
67     (cd kernel/blk_drv; make)
68
69 kernel/chr_drv/chr_drv.a:          # 生成字符设备函数文件 chr_drv.a。
70     (cd kernel/chr_drv; make)
71
72 kernel/kernel.o:                   # 内核目标模块 kernel.o
73     (cd kernel; make)
74
75 mm/mm.o:                           # 内存管理模块 mm.o
76     (cd mm; make)
77
78 fs/fs.o:                           # 文件系统目标模块 fs.o
79     (cd fs; make)
80
81 lib/lib.a:                          # 库函数 lib.a
82     (cd lib; make)
83
84 boot/setup: boot/setup.s           # 这里开始的三行是使用 8086 汇编和连接器
85     $(AS86) -o boot/setup.o boot/setup.s # 对 setup.s 文件进行编译生成 setup 文件。
86     $(LD86) -s -o boot/setup boot/setup.o # -s 选项表示要去掉目标文件中的符号信息。
87
88 boot/bootsect: boot/bootsect.s     # 同上。生成 bootsect.o 磁盘引导块。
89     $(AS86) -o boot/bootsect.o boot/bootsect.s
90     $(LD86) -s -o boot/bootsect boot/bootsect.o
91
# 下面 92—95 行的作用是在 bootsect.s 文本程序开始处添加一行有关 system 模块文件长度信息，
# 在把 system 模块加载到内存期间用于指明系统模块的长度。添加该行信息的方法是首先生成只含
# 有“SYSSIZE = system 文件实际长度”一行信息的 tmp.s 文件，然后将 bootsect.s 文件添加在其后。
# 取得 system 长度的方法是：首先利用命令 ls 对编译生成的 system 模块文件进行长列表显示，用
# grep 命令取得列表行上文件字节数字段信息，并定向保存在 tmp.s 临时文件中。cut 命令用于剪切
# 字符串，tr 用于去除行尾的回车符。其中：(实际长度 + 15)/16 用于获得用‘节’表示的长度信息，
# 1 节=16 字节。
# 注意：这是 Linux 0.11 之前的内核版本 (0.01—0.10) 获取 system 模块长度并添加到 bootsect.s
# 程序中使用的办法。从 0.11 版内核开始已不使用这个方法，而是直接在 bootsect.s 程序开始处给
# 出了 system 模块的一个最大默认长度值。因此这个规则现在已经不起作用。
92 tmp.s: boot/bootsect.s tools/system
93     (echo -n "SYSSIZE = (" ; ls -l tools/system | grep system \

```



```

94         | cut -c25-31 | tr '\012' ' '; echo "+ 15 ) / 16") > tmp.s
95 cat boot/bootsect.s >> tmp.s
96
97 clean: # 当执行'make clean'时, 就会执行 98--103 行上的命令, 去除所有编译连接生成的文件。
          # 'rm' 是文件删除命令, 选项-f 含义是忽略不存在的文件, 并且不显示删除信息。
98 rm -f Image System.map tmp_make core boot/bootsect boot/setup
99 rm -f init/*.o tools/system tools/build boot/*.o
100 (cd mm;make clean)      # 进入 mm/目录; 执行该目录 Makefile 文件中的 clean 规则。
101 (cd fs;make clean)
102 (cd kernel;make clean)
103 (cd lib;make clean)
104
    # 该规则将首先执行上面的 clean 规则, 然后对 linux/目录进行压缩, 生成'backup.Z' 压缩文件。
    # 'cd ..' 表示退到 linux/的上一级(父)目录; 'tar cf - linux' 表示对 linux/目录执行 tar 归档
    # 程序。'-cf' 表示需要创建新的归档文件 '| compress -' 表示将 tar 程序的执行通过管道操作('|')
    # 传递给压缩程序 compress, 并将压缩程序的输出存成 backup.Z 文件。
105 backup: clean
106         (cd .. ; tar cf - linux | compress - > backup.Z)
107         sync                                # 迫使缓冲块数据立即写盘并更新磁盘超级块。
108
109 dep:
    # 该目标或规则用于产生各文件之间的依赖关系。创建这些依赖关系是为了让 make 命令用它们来确定
    # 是否需要重建一个目标对象。比如当某个头文件被改动过后, make 就能通过生成的依赖关系, 重新
    # 编译与该头文件有关的所有*.c 文件。具体方法如下:
    # 使用字符串编辑程序 sed 对 Makefile 文件(这里即是本文件)进行处理, 输出为删除了 Makefile
    # 文件中'### Dependencies' 行后面的所有行, 即删除了下面从 118 开始到文件末的所有行, 并生成
    # 一个临时文件 tmp_make (也即 110 行的作用)。然后对指定目录下 (init/) 的每一个 C 文件 (其实
    # 只有一个文件 main.c) 执行 gcc 预处理操作。标志'-M' 告诉预处理程序 cpp 输出描述每个目标文件
    # 相关性的规则, 并且这些规则符合 make 语法。对于每一个源文件, 预处理程序会输出一个规则, 其
    # 结果形式就是相应源程序文件的目标文件名加上其依赖关系, 即该源文件中包含的所有头文件列表。
    # 然后把预处理结果都添加到临时文件 tmp_make 中, 最后将该临时文件复制成新的 Makefile 文件。
    # 111 行上的'$$i' 实际上是'$(i)'。这里'i' 是这句前面的 shell 变量'i' 的值。
110 sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
111 (for i in init/*.c;do echo -n "init/";$(CPP) -M $$i;done) >> tmp_make
112 cp tmp_make Makefile
113 (cd fs; make dep)                # 对 fs/目录下的 Makefile 文件也作同样的处理。
114 (cd kernel; make dep)
115 (cd mm; make dep)
116
117 ### Dependencies:
118 init/main.o : init/main.c include/unistd.h include/sys/stat.h \
119 include/sys/types.h include/sys/times.h include/sys/utsname.h \
120 include/utime.h include/time.h include/linux/tty.h include/termios.h \
121 include/linux/sched.h include/linux/head.h include/linux/fs.h \
122 include/linux/mm.h include/signal.h include/asm/system.h include/asm/io.h \
123 include/stddef.h include/stdarg.h include/fcntl.h

```

## 5.13 本章小结

本章概述了 Linux 早期操作系统的内核模式和体系结构。首先给出了 Linux 0.11 内核使用和管理内存的方法、内核态栈和用户态栈的设置和使用方法、中断机制、系统时钟定时以及进程创建、调度和终



止方法。然后根据源代码的目录结构形式，详细地介绍了各个子目录中代码文件的基本功能和层次关系。同时说明了 Linux 0.11 所使用的目标文件格式。最后从 Linux 内核主目录下的 `makefile` 文件着手，开始对内核源代码进行注释。




本章内容可以看作是对 Linux 0.11 内核重要信息的归纳说明，因此可作为阅读后续章节的参考内容。

## 第6章 引导启动程序（boot）

本章主要描述 boot/目录中的三个汇编代码文件，见列表 6-1 所示。正如在前一章中提到的，这三个文件虽然都是汇编程序，但却使用了两种语法格式。bootsect.s 和 setup.s 是实模式下运行的 16 位代码程序，采用近似于 Intel 的汇编语言语法并且需要使用 Intel 8086 汇编编译器和连接器 as86 和 ld86，而 head.s 则使用 GNU 的汇编程序格式，并且运行在保护模式下，需要用 GNU 的 as（gas）进行编译。这是一种 AT&T 语法的汇编语言程序。

Linux 当时使用两种汇编编译器的主要原因在于对于 Intel x86 处理器系列来讲，Linux 那时的 GNU 汇编编译器仅能支持 i386 及以后出的 CPU 代码指令，若不采用特殊方法就不能支持生成运行在实模式下的 16 位代码程序。直到 1994 年以后发布的 GNU as 汇编器才开始支持编译 16 位代码的.code16 伪指令。参见 GNU 汇编器手册《Using as - The GNU Assembler》中“80386 相关特性”一节中“编写 16 位代码”小节。但直到内核版本 2.4.X 起，bootsect.s 和 setup.s 程序才完全使用统一的 as 来编写。

列表 6-1 linux/boot/目录

	文件名	长度(字节)	最后修改时间(GMT)	说明
	bootsect.s	5052 bytes	1991-12-05 22:47:58	
	head.s	5938 bytes	1991-11-18 15:05:09	
	setup.s	5364 bytes	1991-12-05 22:48:10	

阅读这些代码除了需要知道一些一般 8086 汇编语言的知识以外，还要了解一些采用 Intel 80X86 微处理器的 PC 机的体系结构以及 80386 32 位保护模式下的编程原理。所以在开始阅读源代码之前应该已经理解前面几章的内容，在阅读代码时我们再就事论事地针对具体问题进行详细说明。

### 6.1 总体功能

这里先总体说明一下 Linux 操作系统启动部分的主要执行流程。当 PC 的电源打开后，80x86 结构的 CPU 将自动进入实模式，并从地址 0xFFFF0 开始自动执行程序代码，这个地址通常是 ROM-BIOS 中的地址。PC 机的 BIOS 将执行某些系统的检测，并在物理地址 0 处开始初始化中断向量。此后，它将可启动设备的第一个扇区（磁盘引导扇区，512 字节）读入内存绝对地址 0x7C00 处，并跳转到这个地方。启动设备通常是软驱或是硬盘。这里的叙述是非常简单的，但这已经足够理解内核初始化的工作过程了。

Linux 的最最前面部分是用 8086 汇编语言编写的（boot/bootsect.s），它将由 BIOS 读入到内存绝对地址 0x7C00（31KB）处，当它被执行时就会把自己移动到内存绝对地址 0x90000（576KB）处，并把启动设备中后 2KB 字节代码（boot/setup.s）读入到内存 0x90200 处，而内核的其他部分（system 模块）则被读入到从内存地址 0x10000（64KB）开始处，因此从机器加电开始顺序执行的程序见图 6-1 所示。

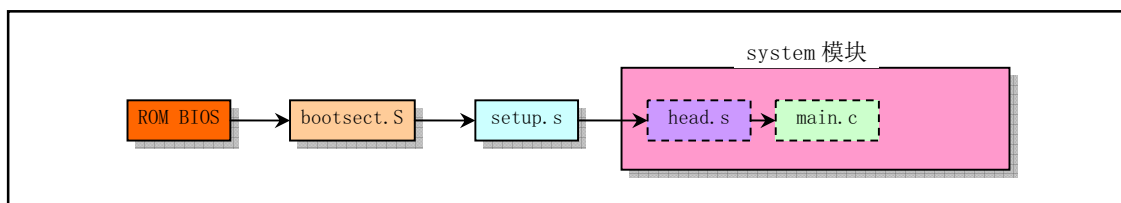


图 6-1 从系统加电起所执行程序的顺序

因为当时 system 模块的长度不会超过 0x80000 字节大小（即 512KB），所以 bootsect 程序把 system 模块读入物理地址 0x10000 开始位置处时并不会覆盖在 0x90000（576KB）处开始的 bootsect 和 setup 模块。后面 setup 程序将会把 system 模块移动到物理内存起始位置处，这样 system 模块中代码的地址也即等于实际的物理地址，便于对内核代码和数据进行操作。图 6-2 清晰地显示出 Linux 系统启动时这几个程序或模块在内存中的动态位置。其中，每一竖条框代表某一时刻内存中各程序的映像位置图。在系统加载期间将显示信息"Loading..."。然后控制权将传递给 boot/setup.s 中的代码，这是另一个实模式汇编语言程序。

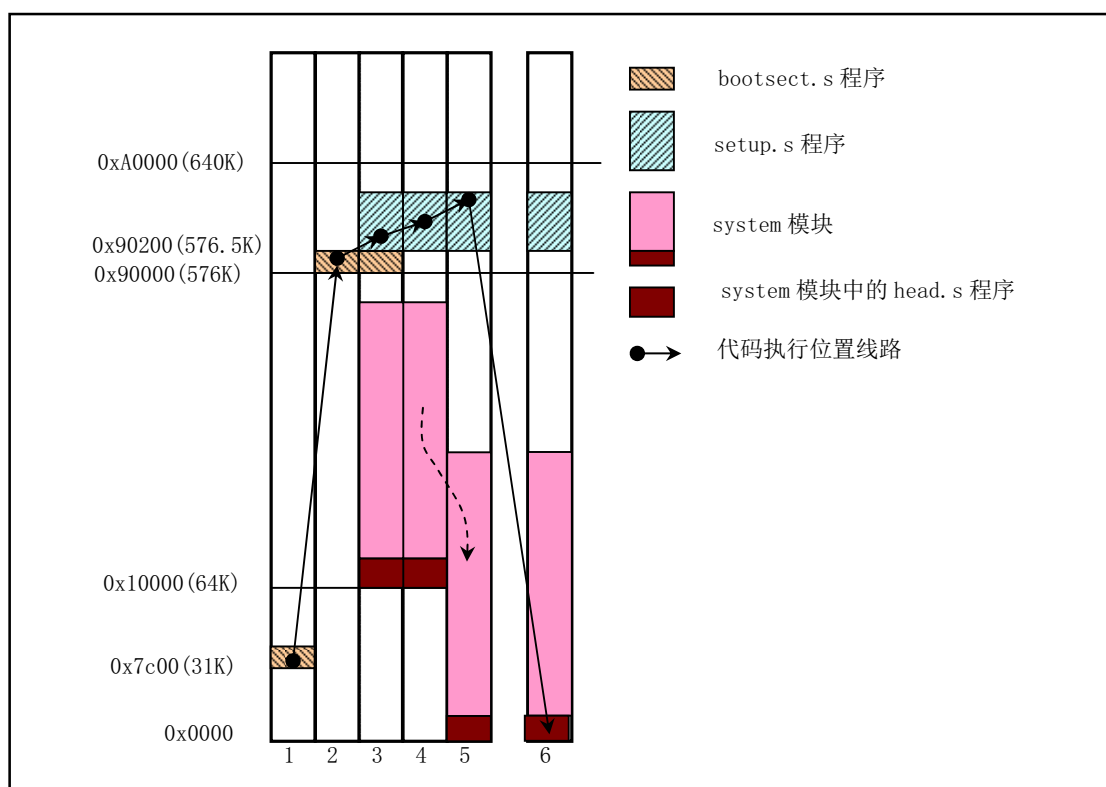


图 6-2 启动引导时内核在内存中的位置和移动后的位置情况

启动部分识别主机的某些特性以及 VGA 卡的类型。如果需要，它会要求用户为控制台选择显示模式。然后将整个系统从地址 0x10000 移至 0x0000 处，进入保护模式并跳转至系统的余下部分（在 0x0000 处）。此时所有 32 位运行方式的设置启动被完成：IDT、GDT 以及 LDT 被加载，处理器和协处理器也已确认，分页工作也设置好了；最终调用 init/main.c 中的 main() 程序。上述操作的源代码是在 boot/head.s 中的，这可能是整个内核中最有诀窍的代码了。注意如果在前述任何一步中出了错，计算机就会死锁。在操作系统还没有完全运转之前是处理不了出错的。

bootsect 的代码为什么不把系统模块直接加载到物理地址 0x0000 开始处而要在 setup 程序中再进行

移动呢？这是因为在随后执行的 `setup` 代码开始部分还需要利用 ROM BIOS 中的中断调用来获取机器的一些参数（例如显示卡模式、硬盘参数表等）。当 BIOS 初始化时会在物理内存开始处放置一个大小为 0x400 字节(1KB)的中断向量表，因此需要在使用完 BIOS 的中断调用后才能将这个区域覆盖掉。

另外，仅在内存中加载了上述内核代码模块并不能让 Linux 系统运行起来。作为完整可运行的 Linux 系统还需要有一个基本的文件系统支持，即根文件系统。Linux 0.11 内核仅支持 MINIX 的 1.0 文件系统。根文件系统通常是在另一个软盘上或者在一个硬盘分区中。为了通知内核所需要的根文件系统在什么地方，`bootsect.s` 程序的第 43 行上给出了根文件系统所在的默认块设备号。块设备号的含义请参见程序中的注释。在内核初始化时会使用编译内核时放在引导扇区第 509、510 (0x1fc--0x1fd) 字节中的指定设备号。

## 6.2 bootsect.s 程序

### 6.2.1 功能描述

`bootsect.s` 代码是磁盘引导块程序，驻留在磁盘的第一个扇区中（引导扇区，0 磁道（柱面），0 磁头，第 1 个扇区）。在 PC 机加电 ROM BIOS 自检后，ROM BIOS 会把引导扇区代码 `bootsect` 加载到内存地址 0x7C00 开始处并执行之。在 `bootsect` 代码执行期间，它会将自己移动到内存绝对地址 0x90000 开始处并继续执行。该程序的主要作用是首先把从磁盘第 2 个扇区开始的 4 个扇区的 `setup` 模块（由 `setup.s` 编译而成）加载到内存紧接着 `bootsect` 后面位置处（0x90200），然后利用 BIOS 中断 0x13 取磁盘参数表中当前启动引导盘参数，接着在屏幕上显示 “Loading system...” 字符串。再者把磁盘上 `setup` 模块后面的 `system` 模块加载到内存 0x10000 开始的地方。随后确定根文件系统的设备号，若没有指定，则根据所保存的引导盘的每磁道扇区数判别出盘的类型和种类（是 1.44M A 盘吗？）并保存其设备号于 `root_dev`（引导块的 508 地址处），最后长跳转到 `setup` 程序的开始处（0x90200）执行 `setup` 程序。在磁盘上，引导块、`setup` 模块和 `system` 模块的扇区位置和大小示意图见图 6-3 所示。

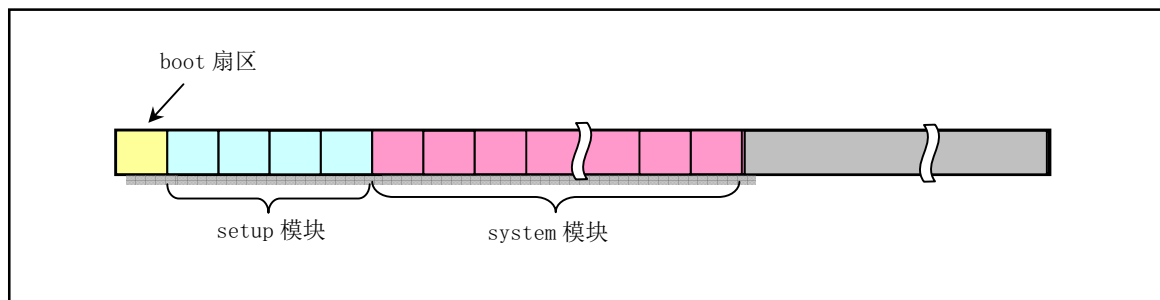


图 6-3 Linux 0.11 内核在 1.44MB 磁盘上的分布情况

图中示出了 Linux 0.11 内核在 1.44MB 磁盘上所占据扇区的分布情况。1.44MB 磁盘共有 2880 个扇区，其中引导程序代码占用第 1 个扇区，`setup` 模块占用随后的 4 个扇区，而 0.11 内核 `system` 模块大约占随后的 240 个扇区。还剩下 2630 多个扇区未被使用。这些剩余的未用空间可被利用来存放一个基本的根文件系统，从而可以创建出使用单张磁盘就能让系统运转起来的集成盘来。这将在块设备驱动程序一章中再作详细介绍。

### 6.2.2 代码注释

程序 6-1 linux/boot/bootsect.s

```

1 !
2 ! SYS_SIZE is the number of clicks (16 bytes) to be loaded.
3 ! 0x3000 is 0x30000 bytes = 196kB, more than enough for current
4 ! versions of linux
  ! SYS_SIZE 是要加载的系统模块长度，单位是节，16 字节为 1 节。0x3000 共为
  ! 0x30000 字节=196 kB（若以 1024 字节为 1KB 计，则应该是 192KB），对于当前的版本空间已足够了。
  ! 这里感叹号'!'或分号';'表示程序注释语句的开始。
5 !
  ! 下面等号 '=' 或符号 'EQU' 用于定义标识符或标号所代表的值，可称为符号常量。这个常量指明
  ! 编译连接后 system 模块的大小。这个等式 "SYSSIZE = 0x3000" 原来是由 linux/Makefile 中
  ! 第 92 行上的语句动态自动产生。但从 Linux 0.11 版本开始就直接在这里给出了一个最大默认
  ! 值。原来参见的自动产生语句还没有被删除，参见程序 5-1 中第 92 行的说明。当该值为 0x8000
  ! 时，表示内核最大为 512KB。
6 SYSSIZE = 0x3000
7 !
8 !      bootsect.s              (C) 1991 Linus Torvalds
9 !
10 ! bootsect.s is loaded at 0x7c00 by the bios-startup routines, and moves
11 ! itself out of the way to address 0x90000, and jumps there.
12 !
13 ! It then loads 'setup' directly after itself (0x90200), and the system
14 ! at 0x10000, using BIOS interrupts.
15 !
16 ! NOTE! currently system is at most 8*65536 bytes long. This should be no
17 ! problem, even in the future. I want to keep it simple. This 512 kB
18 ! kernel size should be enough, especially as this doesn't contain the
19 ! buffer cache as in minix
20 !
21 ! The loader has been made as simple as possible, and continuos
22 ! read errors will result in a unbreakable loop. Reboot by hand. It
23 ! loads pretty fast by getting whole sectors at a time whenever possible.
  !
  ! 以下是前面这些文字的翻译：
  !      bootsect.s              (C) 1991 Linus Torvalds 版权所有
  !
  ! bootsect.s 被 bios-启动子程序加载至 0x7c00 (31KB)处，并将自己
  ! 移到了地址 0x90000 (576KB)处，并跳转至那里。
  !
  ! 它然后使用 BIOS 中断将 'setup' 直接加载到自己的后面 (0x90200) (576.5KB)，
  ! 并将 system 加载到地址 0x10000 处。
  !
  ! 注意！目前的内核系统最大长度限制为 (8*65536) (512KB) 字节，即使是在
  ! 将来这也应该没有问题的。我想让它保持简单明了。这样 512KB 的最大内核长度应该
  ! 足够了，尤其是这里没有象 minix 中一样包含缓冲区高速缓冲。
  !
  ! 加载程序已经做得够简单了，所以持续的读出错将导致死循环。只能手工重启。
  ! 只要可能，通过一次读取所有的扇区，加载过程可以做得很快。
24 ! 伪指令（伪操作符）.globl 或 .global 用于定义随后的标识符是外部的或全局的，并且即使不使用
  ! 也强制引入。 .text、.data 和 .bss 用于分别定义当前代码段、数据段和未初始化数据段。在链接
  ! 多个目标模块时，链接程序 (ld86) 会根据它们的类别把各个目标模块中的相应段分别组合（合并）
  ! 在一起。这里把三个段都定义在同一重叠地址范围中，因此本程序实际上不分段。
  ! 另外，后面带冒号的字符串是标号，例如下面的 'begtext:'。一条汇编语句通常由标号（可选）、

```

```

! 指令助记符（指令名）和操作数三个字段组成。标号位于一条指令的第一个字段。它代表其所在位
! 置的地址，通常指明一个跳转指令的目标位置。
25 .globl begtext, begdata, begbss, endtext, enddata, endbss ! 定义了6个全局标识符；
26 .text ! 文本段；
27 begtext:
28 .data ! 数据段；
29 begdata:
30 .bss ! 未初始化数据段(Block Started by Symbol)；
31 begbss:
32 .text ! 文本段；
33
34 SETUPLEN = 4 ! nr of setup-sectors
! setup 程序的扇区数(setup-sectors)值；
35 BOOTSEG = 0x07c0 ! original address of boot-sector
! bootsect 的原始地址（是段地址，以下同）；
36 INITSEG = 0x9000 ! we move boot here - out of the way
! 将 bootsect 移到这里 -- 避开；
37 SETUPSEG = 0x9020 ! setup starts here
! setup 程序从这里开始；
38 SYSSEG = 0x1000 ! system loaded at 0x10000 (65536).
! system 模块加载到 0x10000 (64 KB) 处；
39 ENDSEG = SYSSEG + SYSSIZE ! where to stop loading
! 停止加载的段地址；
40
41 ! ROOT_DEV: 0x000 - same type of floppy as boot.
! 根文件系统设备使用与引导时同样的软驱设备；
42 ! 0x301 - first partition on first drive etc
! 根文件系统设备在第一个硬盘的第一个分区上，等等；
43 ROOT_DEV = 0x306
! 设备号 0x306 指定根文件系统设备是第 2 个硬盘的第 1 个分区。当年 Linux 是在第 2 个硬盘上安装
! 了 Linux 0.11 系统，所以这里 ROOT_DEV 被设置为 0x306。在编译这个内核时你可以根据自己根文件
! 系统所在设备位置修改这个设备号。这个设备号是 Linux 系统老式的硬盘设备号命名方式，硬盘设备
! 号具体值的含义如下：
! 设备号=主设备号*256 + 次设备号（也即 dev_no = (major<<8) + minor）
! （主设备号：1-内存, 2-磁盘, 3-硬盘, 4-ttyx, 5-tty, 6-并行口, 7-非命名管道）
! 0x300 - /dev/hd0 - 代表整个第 1 个硬盘；
! 0x301 - /dev/hd1 - 第 1 个盘的第 1 个分区；
! ...
! 0x304 - /dev/hd4 - 第 1 个盘的第 4 个分区；
! 0x305 - /dev/hd5 - 代表整个第 2 个硬盘；
! 0x306 - /dev/hd6 - 第 2 个盘的第 1 个分区；
! ...
! 0x309 - /dev/hd9 - 第 2 个盘的第 4 个分区；
! 从 Linux 内核 0.95 版后就已经使用与现在内核相同的命名方法了。
44
! 伪指令 entry 迫使链接程序在生成的执行程序（a.out）中包含指定的标识符或标号。
! 47--56 行作用是将自身（bootsect）从目前段位置 0x07c0(31KB) 移动到 0x9000(576KB) 处，共 256 字
! （512 字节），然后跳转到移动后代码的 go 标号处，也即本程序的下一语句处。
45 entry start ! 告知链接程序，程序从 start 标号开始执行。
46 start:
47 mov ax, #BOOTSEG ! 将 ds 段寄存器置为 0x7C0；
48 mov ds, ax
49 mov ax, #INITSEG ! 将 es 段寄存器置为 0x9000；

```

```

50      mov     es, ax
51      mov     cx, #256          ! 设置移动计数值=256 字;
52      sub     si, si            ! 源地址  ds:si = 0x07C0:0x0000
53      sub     di, di            ! 目的地址 es:di = 0x9000:0x0000
54      rep                                     ! 重复执行并递减 cx 的值, 直到 cx = 0 为止。
55      movsw                                     ! 即 movs 指令。这里从内存[si]处移动 cx 个字到[di]处。
56      jmp     go, INITSEG        ! 段间跳转 (Jump Intersegment)。这里 INITSEG 指出跳
                                   ! 转到的段地址, 标号 go 是段内偏移地址。

! 从下面开始, CPU 在已移动到 0x90000 位置处的代码中执行。
! 这段代码设置几个段寄存器, 包括栈寄存器 ss 和 sp。栈指针 sp 只要指向远大于 512 字节偏移 (即
! 地址 0x90200) 处都可以。因为从 0x90200 地址开始处还要放置 setup 程序, 而此时 setup 程序大约
! 为 4 个扇区, 因此 sp 要指向大于 (0x200 + 0x200 * 4 + 堆栈大小) 处。
! 实际上 BIOS 把引导扇区加载到 0x7c00 处并把执行权交给引导程序时, ss = 0x00, sp = 0xffff。
57 go:      mov     ax, cs          ! 将 ds、es 和 ss 都置成移动后代码所在的段处 (0x9000)。
58          mov     ds, ax          ! 由于程序中有栈操作 (push, pop, call), 因此必须设置堆栈。
59          mov     es, ax
60 ! put stack at 0x9ff00.          ! 将堆栈指针 sp 指向 0x9ff00 (即 0x9000:0xff00) 处。
61          mov     ss, ax
62          mov     sp, #0xFF00    ! arbitrary value >>512
63
64 ! load the setup-sectors directly after the bootblock.
65 ! Note that 'es' is already set up.
! 在 bootsect 程序块后紧跟着加载 setup 模块的代码数据。
! 注意 es 已经设置好了。(在移动代码时 es 已经指向目的段地址处 0x9000)。
66
! 68--77 行的用途是利用 BIOS 中断 INT 0x13 将 setup 模块从磁盘第 2 个扇区
! 开始读到 0x90200 开始处, 共读 4 个扇区。如果读出错, 则复位驱动器, 并
! 重试, 没有退路。INT 0x13 的使用方法如下:
! 读扇区:
! ah = 0x02 - 读磁盘扇区到内存; al = 需要读出的扇区数量;
! ch = 磁道(柱面)号的低 8 位; cl = 开始扇区(位 0-5), 磁道号高 2 位(位 6-7);
! dh = 磁头号; dl = 驱动器号 (如果是硬盘则位 7 要置位);
! es:bx →指向数据缓冲区; 如果出错则 CF 标志置位, ah 中是出错码。
67 load_setup:
68      mov     dx, #0x0000        ! drive 0, head 0
69      mov     cx, #0x0002        ! sector 2, track 0
70      mov     bx, #0x0200        ! address = 512, in INITSEG
71      mov     ax, #0x0200+SETUPLEN ! service 2, nr of sectors
72      int     0x13               ! read it
73      jnc     ok_load_setup      ! ok - continue
74      mov     dx, #0x0000        ! 对驱动器 0 进行读操作。
75      mov     ax, #0x0000        ! reset the diskette
76      int     0x13
77      j       load_setup         ! 即 jmp 指令。
78
79 ok_load_setup:
80
81 ! Get disk drive parameters, specifically nr of sectors/track
! 取磁盘驱动器的参数, 特别是每道的扇区数量。
! 取磁盘驱动器参数 INT 0x13 调用格式和返回信息如下:
! ah = 0x08 dl = 驱动器号 (如果是硬盘则要置位 7 为 1)。
! 返回信息:

```



```

! 如果出错则 CF 置位, 并且 ah = 状态码。
! ah = 0,  al = 0,          bl = 驱动器类型 (AT/PS2)
! ch = 最大磁道号的低 8 位, cl = 每磁道最大扇区数(位 0-5), 最大磁道号高 2 位(位 6-7)
! dh = 最大磁头数,        dl = 驱动器数量,
! es:di → 软驱磁盘参数表。
82
83     mov     dl,#0x00
84     mov     ax,#0x0800          ! AH=8 is get drive parameters
85     int     0x13
86     mov     ch,#0x00
! 下面指令表示下一条语句的操作数在 cs 段寄存器所指的段中。它只影响其下一条语句。实际上,
! 由于本程序代码和数据都被设置处于同一个段中, 即段寄存器 cs 和 ds、es 的值相同, 因此本程序
! 中此处可以不使用该指令。
87     seg cs
! 下句保存每磁道扇区数。对于软盘来说 (dl=0), 其最大磁道号不会超过 256, ch 已经足够表示它,
! 因此 cl 的位 6-7 肯定为 0。又 86 行已置 ch=0, 因此此时 cx 中是每磁道扇区数。
88     mov     sectors,cx
89     mov     ax,#INITSEG
90     mov     es,ax              ! 因为上面取磁盘参数中断改掉了 es 的值, 这里重新改回。
91
92 ! Print some inane message
! 显示信息: “Loading system ...” 回车换行”, 共显示包括回车和换行控制字符在内的 24 个字符。
! BIOS 中断 0x10 功能号 ah = 0x03, 读光标位置。
! 输入: bh = 页号
! 返回: ch = 扫描开始线; cl = 扫描结束线; dh = 行号(0x00 顶端); dl = 列号(0x00 最左边)。
!
! BIOS 中断 0x10 功能号 ah = 0x13, 显示字符串。
! 输入: al = 放置光标的方式及规定属性。0x01-表示使用 bh 中的属性值, 光标停在字符串结尾处。
! es:bp 此寄存器对指向要显示的字符串起始位置处。cx = 显示的字符串字符数。bh = 显示页面号;
! bh = 字符属性。dh = 行号; dl = 列号。
93
94     mov     ah,#0x03          ! read cursor pos
95     xor     bh,bh             ! 首先读光标位置。返回光标位置值在 dx 中。
96     int     0x10             ! dh=行 (0--24); dl=列(0--79)。供显示串用。
97
98     mov     cx,#24            ! 共显示 24 个字符。
99     mov     bx,#0x0007        ! page 0, attribute 7 (normal)
100    mov     bp,#msg1          ! es:bp 寄存器对指向要显示的字符串。
101    mov     ax,#0x1301        ! write string, move cursor
102    int     0x10              ! 写字符串并移动光标到串结尾处。
103
104 ! ok, we've written the message, now
105 ! we want to load the system (at 0x10000)
! 现在开始将 system 模块加载到 0x10000 (64KB) 开始处。
106
107    mov     ax,#SYSSEG
108    mov     es,ax              ! segment of 0x010000 ! es = 存放 system 的段地址。
109    call    read_it            ! 读磁盘上 system 模块, es 为输入参数。
110    call    kill_motor         ! 关闭驱动器马达, 这样就可以知道驱动器的状态了。
111
112 ! After that we check which root-device to use. If the device is
113 ! defined (!= 0), nothing is done and the given device is used.
114 ! Otherwise, either /dev/PS0 (2,28) or /dev/at0 (2,8), depending

```

```

115 ! on the number of sectors that the BIOS reports currently.
! 此后，我们检查要使用哪个根文件系统设备（简称根设备）。如果已经指定了设备(!=0)
! 就直接使用给定的设备。否则就需要根据 BIOS 报告的每磁道扇区数来
! 确定到底使用/dev/PS0 (2, 28) 还是 /dev/at0 (2, 8)。
!! 上面一行中两个设备文件的含义：
!! 在 Linux 中软驱的主设备号是 2(参见第 43 行的注释)，次设备号 = type*4 + nr，其中
!! nr 为 0-3 分别对应软驱 A、B、C 或 D；type 是软驱的类型 (2→1.2MB 或 7→1.44MB 等)。
!! 因为 7*4 + 0 = 28，所以 /dev/PS0 (2, 28)指的是 1.44MB A 驱动器,其设备号是 0x021c
!! 同理 /dev/at0 (2, 8)指的是 1.2MB A 驱动器，其设备号是 0x0208。
! 下面 root_dev 定义在引导扇区 508, 509 字节处，指根文件系统所在设备号。0x0306 指第 2
! 个硬盘第 1 个分区。这里默认为 0x0306 是因为当时 Linus 开发 Linux 系统时是在第 2 个硬
! 盘第 1 个分区中存放根文件系统。这个值需要根据你自己根文件系统所在硬盘和分区进行修
! 改。例如，如果你的根文件系统在第 1 个硬盘的第 1 个分区上，那么该值应该为 0x0301，即
! (0x01, 0x03)。如果根文件系统是在第 2 个 Bochs 软盘上，那么该值应该为 0x021D，即
! (0x1D, 0x02)。当编译内核时，你可以在 Makefile 文件中另行指定你自己的值，内核映像
! 文件 Image 的创建程序 tools/build 会使用你指定的值来设置你的根文件系统所在设备号。

116
117         seg cs
118         mov     ax, root_dev      ! 取 508, 509 字节处的根设备号并判断是否已被定义。
119         cmp     ax, #0
120         jne     root_defined
! 取上面第 88 行保存的每磁道扇区数。如果 sectors=15 则说明是 1.2MB 的驱动器；如果
! sectors=18，则说明是 1.44MB 软驱。因为是可引导的驱动器，所以肯定是 A 驱。
121         seg cs
122         mov     bx, sectors
123         mov     ax, #0x0208      ! /dev/ps0 - 1.2MB
124         cmp     bx, #15          ! 判断每磁道扇区数是否=15
125         je      root_defined     ! 如果等于，则 ax 中就是引导驱动器的设备号。
126         mov     ax, #0x021c      ! /dev/PS0 - 1.44MB
127         cmp     bx, #18
128         je      root_defined
129 undef_root:                      ! 如果都不一样，则死循环（死机）。
130         jmp     undef_root
131 root_defined:
132         seg cs
133         mov     root_dev, ax      ! 将检查过的设备号保存到 root_dev 中。
134
135 ! after that (everything loaded), we jump to
136 ! the setup-routine loaded directly after
137 ! the bootblock:
! 到此，所有程序都加载完毕，我们就跳转到被加载在 bootsect 后面的 setup 程序去。
! 段间跳转指令（Jump Intersegment）。跳转到 0x9020:0000(setup.s 程序开始处)去执行。
138
139         jmpi    0, SETUPSEG      !!!! 到此本程序就结束了。!!!!

! 下面是两个子程序。read_it 用于读取磁盘上的 system 模块。kill_moter 用于关闭软驱的马达。
140
141 ! This routine loads the system at address 0x10000, making sure
142 ! no 64kB boundaries are crossed. We try to load it as fast as
143 ! possible, loading whole tracks whenever we can.
144 !
145 ! in:  es - starting address segment (normally 0x1000)
146 !

```

```

! 该子程序将系统模块加载到内存地址 0x10000 处，并确定没有跨越 64KB 的内存边界。
! 我们试图尽快地进行加载，只要可能，就每次加载整条磁道的数据。
! 输入：es - 开始内存地址段值（通常是 0x1000）
!
! 下面伪操作符 .word 定义一个 2 字节目标。相当于 C 语言程序中定义的变量和所占内存空间大小。
! '1+SETUPLEN' 表示开始时已经读进 1 个引导扇区和 setup 程序所占的扇区数 SETUPLEN。
147 sread: .word 1+SETUPLEN      ! sectors read of current track !当前磁道中已读扇区数。
148 head:  .word 0              ! current head    !当前磁头号。
149 track: .word 0              ! current track   !当前磁道号。
150
151 read_it:
! 首先测试输入的段值。从盘上读入的数据必须存放在位于内存地址 64KB 的边界开始处，否则进入死
! 循环。清 bx 寄存器，用于表示当前段内存放数据的开始位置。
! 153 行上的指令 test 以比特位逻辑与两个操作数。若两个操作数对应的比特位都为 1，则结果值的
! 对应比特位为 1，否则为 0。该操作结果只影响标志（零标志 ZF 等）。例如，若 AX=0x1000，那么
! test 指令的执行结果是 (0x1000 & 0x0fff) = 0x0000，于是 ZF 标志置位。此时即下一条指令 jne
! 条件不成立。
152     mov ax, es
153     test ax, #0x0fff
154 die:  jne die                ! es must be at 64kB boundary ! es 值必须位于 64KB 地址边界!
155     xor bx, bx                ! bx is starting address within segment ! bx 为段内偏移。
156 rp_read:
! 接着判断是否已经读入全部数据。比较当前所读段是否就是系统数据末端所处的段(#ENDSEG)，如果
! 不是就跳转至下面 ok1_read 标号处继续读数据。否则退出子程序返回。
157     mov ax, es
158     cmp ax, #ENDSEG          ! have we loaded all yet? ! 是否已经加载了全部数据?
159     jb ok1_read
160     ret
161 ok1_read:
! 计算和验证当前磁道需要读取的扇区数，放在 ax 寄存器中。
! 根据当前磁道还未读取的扇区数以及段内数据字节开始偏移位置，计算如果全部读取这些未读扇区，
! 所读总字节数是否会超过 64KB 段长度的限制。若会超过，则根据此次最多能读入的字节数 (64KB -
! 段内偏移位置)，反算出此次需要读取的扇区数。
162     seg cs
163     mov ax, sectors          ! 取每磁道扇区数。
164     sub ax, sread            ! 减去当前磁道已读扇区数。
165     mov cx, ax               ! cx = ax = 当前磁道未读扇区数。
166     shl cx, #9               ! cx = cx * 512 字节 + 段内当前偏移值 (bx)。
167     add cx, bx               ! = 此次读操作后，段内共读入的字节数。
168     jnc ok2_read            ! 若没有超过 64KB 字节，则跳转至 ok2_read 处执行。
169     je ok2_read
! 若加上此次将读磁道上所有未读扇区时会超过 64KB，则计算此时最多能读入的字节数：
! (64KB - 段内读偏移位置)，再转换成需读取的扇区数。其中 0 减某数就是取该数 64KB 的补值。
170     xor ax, ax
171     sub ax, bx
172     shr ax, #9
173 ok2_read:
! 读当前磁道上指定开始扇区 (cl) 和需读扇区数 (al) 的数据到 es:bx 开始处。然后统计当前磁道
! 上已经读取的扇区数并与磁道最大扇区数 sectors 作比较。如果小于 sectors 说明当前磁道上的还
! 有扇区未读。于是跳转到 ok3_read 处继续操作。
174     call read_track          ! 读当前磁道上指定开始扇区和需读扇区数的数据。
175     mov cx, ax               ! cx = 该次操作已读取的扇区数。
176     add ax, sread            ! 加上当前磁道上已经读取的扇区数。

```

```

177      seg cs
178      cmp ax, sectors          ! 如果当前磁道上的还有扇区未读, 则跳转到 ok3_read 处。
179      jne ok3_read
! 若该磁道的当前磁头面所有扇区已经读取, 则读该磁道的下一磁头面(1 号磁头)上的数据。如果已经
! 完成, 则去读下一磁道。
180      mov ax, #1
181      sub ax, head             ! 判断当前磁头号。
182      jne ok4_read            ! 如果是 0 磁头, 则再去读 1 磁头面上的扇区数据。
183      inc track                ! 否则去读下一磁道。
184 ok4_read:
185      mov head, ax             ! 保存当前磁头号。
186      xor ax, ax               ! 清当前磁道已读扇区数。
187 ok3_read:
! 如果当前磁道上的还有未读扇区, 则首先保存当前磁道已读扇区数, 然后调整存放数据处的开始
! 位置。若小于 64KB 边界值, 则跳转到 rp_read(156 行)处, 继续读数据。
188      mov sread, ax           ! 保存当前磁道已读扇区数。
189      shl cx, #9               ! 上次已读扇区数*512 字节。
190      add bx, cx               ! 调整当前段内数据开始位置。
191      jnc rp_read
! 否则说明已经读取 64KB 数据。此时调整当前段, 为读下一段数据作准备。
192      mov ax, es
193      add ax, #0x1000          ! 将段基址调整为指向下一个 64KB 内存开始处。
194      mov es, ax
195      xor bx, bx               ! 清段内数据开始偏移值。
196      jmp rp_read             ! 跳转至 rp_read(156 行)处, 继续读数据。
197
! read_track 子程序。
! 读当前磁道上指定开始扇区和需读扇区数的数据到 es:bx 开始处。参见第 67 行下对 BIOS 磁盘读中断
! int 0x13, ah=2 的说明。
! al - 需读扇区数; es:bx - 缓冲区开始位置。
198 read_track:
199      push ax
200      push bx
201      push cx
202      push dx
203      mov dx, track            ! 取当前磁道号。
204      mov cx, sread            ! 取当前磁道上已读扇区数。
205      inc cx                   ! cl = 开始读扇区。
206      mov ch, dl               ! ch = 当前磁道号。
207      mov dx, head             ! 取当前磁头号。
208      mov dh, dl               ! dh = 磁头号。
209      mov dl, #0               ! dl = 驱动器号(为 0 表示当前 A 驱动器)。
210      and dx, #0x0100          ! 磁头号不大于 1。
211      mov ah, #2               ! ah = 2, 读磁盘扇区功能号。
212      int 0x13
213      jc bad_rt                ! 若出错, 则跳转至 bad_rt。
214      pop dx
215      pop cx
216      pop bx
217      pop ax
218      ret
! 读磁盘操作出错。则执行驱动器复位操作(磁盘中断功能号 0), 再跳转到 read_track 处重试。
219 bad_rt: mov ax, #0

```

---

```

220      mov dx, #0
221      int 0x13
222      pop dx
223      pop cx
224      pop bx
225      pop ax
226      jmp read_track
227
228 /*
229  * This procedure turns off the floppy drive motor, so
230  * that we enter the kernel in a known state, and
231  * don't have to worry about it later.
232  */
/* 这个子程序用于关闭软驱的马达，这样我们进入内核后就能
   * 知道它所处的状态，以后也就无须担心它了。
   */
! 下面第 235 行上的值 0x3f2 是软盘控制器的一个端口，被称为数字输出寄存器（DOR）端口。它是
! 一个 8 位的寄存器，其位 7—位 4 分别用于控制 4 个软驱（D—A）的启动和关闭。位 3—位 2 用于
! 允许/禁止 DMA 和中断请求以及启动/复位软盘控制器 FDC。位 1—位 0 用于选择选择操作的软驱。
! 第 236 行上在 al 中设置并输出的 0 值，就是用于选择 A 驱动器，关闭 FDC，禁止 DMA 和中断请求，
! 关闭马达。有关软驱控制卡编程的详细信息请参见 kernel/blk_drv/floppy.c 程序后面的说明。
233 kill_motor:
234     push dx
235     mov dx, #0x3f2          ! 软驱控制卡的数字输出寄存器（DOR）端口，只写。
236     mov al, #0             ! A 驱动器，关闭 FDC，禁止 DMA 和中断请求，关闭马达。
237     outb                   ! 将 al 中的内容输出到 dx 指定的端口去。
238     pop dx
239     ret
240
241 sectors:
242     .word 0                ! 存放当前启动软盘每磁道的扇区数。
243
244 msg1:
245     .byte 13, 10           ! 调用 BIOS 中断显示的信息。
246     .ascii "Loading system ..." ! 回车、换行的 ASCII 码。
247     .byte 13, 10, 13, 10   ! 共 24 个 ASCII 码字符。
248
! 表示下面语句从地址 508 (0x1FC) 开始，所以 root_dev 在启动扇区的第 508 开始的 2 个字节中。
249 .org 508
250 root_dev:
251     .word ROOT_DEV        ! 这里存放根文件系统所在设备号 (init/main.c 中会用)。

! 下面是启动盘具有有效引导扇区的标志。仅供 BIOS 中的程序加载引导扇区时识别使用。它必须位于
! 引导扇区的最后两个字节中。
252 boot_flag:
253     .word 0xAA55
254
255 .text
256 endtext:
257 .data
258 enddata:
259 .bss
260 endbss:

```

---

### 6.2.3 其他信息

对 bootsect.s 这段程序的说明和描述，在互连网上可以搜索到大量的资料。其中 Alessandro Rubini 著而由本人翻译的《Linux 内核源代码漫游》一篇文章(<http://oldlinux.org/Linux.old/docs/>)比较详细地描述了内核启动的详细过程，很有参考价值。由于这段程序是在 386 实模式下运行的，因此相对来将比较容易理解。若此时阅读仍有困难，那么建议你首先再复习一下 80x86 汇编及其硬件的相关知识，然后再继续阅读本书。对于最新开发的 Linux 内核，这段程序的改动也很小，基本保持 0.11 版 bootsect 程序的模样。

#### 6.2.3.1 Linux 0.11 硬盘设备号

程序中涉及的硬盘设备命名方式如下：硬盘的主设备号是 3。其它设备的主设备号分别为：1-内存、2-磁盘、3-硬盘、4-ttyx、5-tty、6-并行口、7-非命名管道。由于 1 个硬盘中可以有 1--4 个分区，因此硬盘还依据分区不同用次设备号进行指定分区。因此硬盘的逻辑设备号由以下方式构成：设备号=主设备号\*256 + 次设备号。两个硬盘的所有逻辑设备号见表 6-1 所示。

表 6-1 硬盘逻辑设备号

逻辑设备号	对应设备文件	说明
0x300	/dev/hd0	代表整个第 1 个硬盘
0x301	/dev/hd1	表示第 1 个硬盘的第 1 个分区
0x304	/dev/hd4	表示第 1 个硬盘的第 4 个分区
0x305	/dev/hd5	代表整个第 2 个硬盘
0x306	/dev/hd6	表示第 2 个硬盘的第 1 个分区
0x309	/dev/hd9	表示第 2 个硬盘的第 4 个分区

其中 0x300 和 0x305 并不与哪个分区对应，而是代表整个硬盘。从 Linux 内核 0.95 版后已经不使用这种烦琐的命名方式，而是使用与现在相同的命名方法了。

#### 6.2.3.2 从硬盘启动系统

若需要从硬盘设备启动系统，那么通常需要使用其他多操作系统引导程序来引导系统加载。例如 Shoe lace、LILO 或 Grub 等多操作系统引导程序。此时 bootsect.s 所完成的任务会由这些程序来完成。bootsect 程序就不会被执行了。因为如果从硬盘启动系统，那么通常内核映像文件 Image 会存放在活动分区的根文件系统中。因此你就需要知道内核映像文件 Image 处于文件系统中的位置以及是什么文件系统。即你的引导扇区程序需要能够识别并访问文件系统，并从中读取内核映像文件。

从硬盘启动的基本流程是：系统上电后，可启动硬盘的第 1 个扇区（主引导记录 MBR - Master Boot Record）会被 BIOS 加载到内存 0x7c00 处并开始执行。该程序会首先把自己向下移动到内存 0x600 处，然后根据 MBR 中分区表信息所指明活动分区中的第 1 个扇区（引导扇区）加载到内存 0x7c00 处，然后开始执行之。如果直接使用这种方式来引导系统就会碰到这样一个问题，即根文件系统不能与内核映像文件 Image 共存。

我所想到的解决办法有两个。一种办法是专门设置一个小容量的活动分区来存放内核映像文件 Image。而相应的根文件系统则放在另外一个分区中。这样虽然浪费了硬盘的 4 个主分区之一，但应该能在对 bootsect.s 程序作最少修改的前提下做到从硬盘启动系统。另一个办法是把内核映像文件 Image 与根文件系统组合存放在一个分区中，即内核映像文件 Image 放在分区开始的一些扇区中，而根文件系统则从随后某一指定扇区开始存放。这两种方法均需要对代码进行一些修改。读者可以参考最后一章内容使用 bochs 模拟系统亲手做一些实验。

## 6.3 setup.s 程序

### 6.3.1 功能描述

setup.s 是一个操作系统加载程序，它的主要作用是利用 ROM BIOS 中断读取机器系统数据，并将这些数据保存到 0x90000 开始的位置（覆盖掉了 bootsect 程序所在的地方），所取得的参数和保留的内存位置见表 6-2 所示。这些参数将被内核中相关程序使用，例如字符设备驱动程序集中的 console.c 和 tty\_io.c 程序等。

表 6-2 setup 程序读取并保留的参数

内存地址	长度(字节)	名称	描述
0x90000	2	光标位置	列号（0x00-最左端），行号（0x00-最顶端）
0x90002	2	扩展内存数	系统从 1MB 开始的扩展内存数值（KB）。
0x90004	2	显示页面	当前显示页面
0x90006	1	显示模式	
0x90007	1	字符列数	
0x90008	2	??	
0x9000A	1	显示内存	显示内存(0x00-64k,0x01-128k,0x02-192k,0x03=256k)
0x9000B	1	显示状态	0x00-彩色,I/O=0x3dX; 0x01-单色,I/O=0x3bX
0x9000C	2	特性参数	显示卡特性参数
...			
0x90080	16	硬盘参数表	第 1 个硬盘的参数表
0x90090	16	硬盘参数表	第 2 个硬盘的参数表（如果没有，则清零）
0x901FC	2	根设备号	根文件系统所在的设备号（bootsec.s 中设置）

然后 setup 程序将 system 模块从 0x10000-0x8ffff（当时认为内核系统模块 system 的长度不会超过此值：512KB）整块向下移动到内存绝对地址 0x00000 处。接下来加载中断描述符表寄存器(idtr)和全局描述符表寄存器(gdtr)，开启 A20 地址线，重新设置两个中断控制芯片 8259A，将硬件中断号重新设置为 0x20 - 0x2f。最后设置 CPU 的控制寄存器 CR0（也称机器状态字），从而进入 32 位保护模式运行，并跳转到位于 system 模块最前面部分的 head.s 程序继续运行。

为了能让 head.s 在 32 位保护模式下运行，在本程序中临时设置了中断描述符表（IDT）和全局描述符表（GDT），并在 GDT 中设置了当前内核代码段的描述符和数据段的描述符。下面在 head.s 程序中会根据内核的需要重新设置这些描述符表。

下面首先简单介绍一下段描述符的格式、描述符表的结构和段选择符（有些书中称之位选择子）的格式。Linux 内核代码中用到的代码段、数据段描述符的格式见图 6-4 所示。其中各字段的含义请参见第 4 章中的说明。



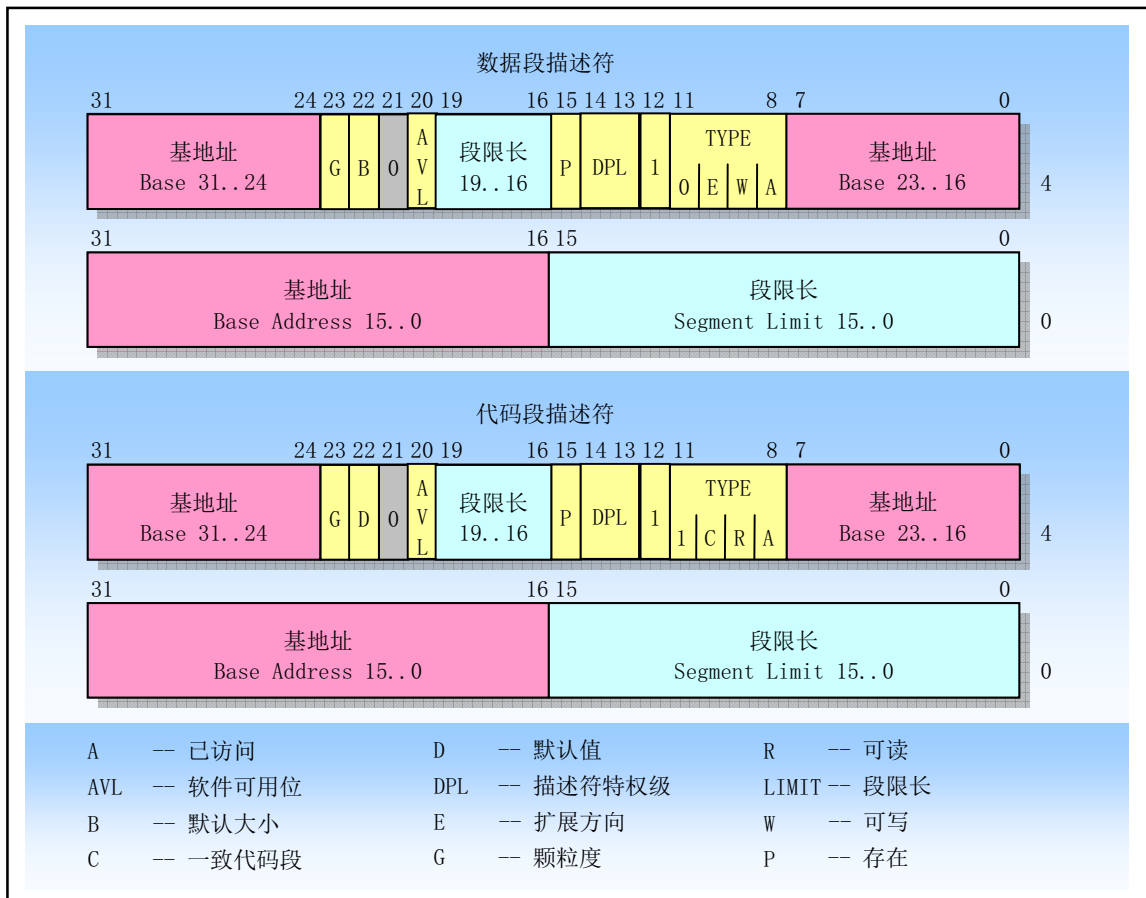


图 6-4 程序代码段和数据段的描述符格式

段描述符存放在描述符表中。描述符表其实就是内存中描述符项的一个阵列。描述符表有两类：全局描述符表（Global descriptor table – GDT）和局部描述符表（Local descriptor table – LDT）。处理器是通过使用 GDTR 和 LDTR 寄存器来定位 GDT 表和当前的 LDT 表。这两个寄存器以线性地址的方式保存了描述符表的基地址和表的长度。指令 lgdt 和 sgdt 用于访问 GDTR 寄存器；指令 lldt 和 sltd 用于访问 LDTR 寄存器。lgdt 使用内存中一个 6 字节操作数来加载 GDTR 寄存器。头两个字节代表描述符表的长度，后 4 个字节是描述符表的基地址。然而请注意，访问 LDTR 寄存器的指令 lldt 所使用的操作数却是一个 2 字节的操作数，表示全局描述符表 GDT 中一个描述符项的选择符。该选择符所对应的 GDT 表中的描述符项应该对应一个局部描述符表。

例如，setup.s 程序设置的 GDT 描述符项（见程序第 207--216 行），代码段描述符的值是 0x00C09A00000007FF，表示代码段的限长是 8MB ( $=(0x7FF + 1) * 4KB$ ，这里加 1 是因为限长值是从 0 开始算起的)，段在线性地址空间中的基址是 0，段类型值 0x9A 表示该段存在于内存中、段的特权级别为 0、段类型是可读可执行的代码段，段代码是 32 位的并且段的颗粒度是 4KB。数据段描述符的值是 0x00C09200000007FF，表示数据段的限长是 8MB，段在线性地址空间中的基址是 0，段类型值 0x92 表示该段存在于内存中、段的特权级别为 0、段类型是可读可写的数据段，段代码是 32 位的并且段的颗粒度是 4KB。

这里再对选择符进行一些说明。逻辑地址的选择符部分用于指定一描述符，它是通过指定一描述符表并且索引其中的一个描述符项完成的。图 6-5 示出了选择符的格式。

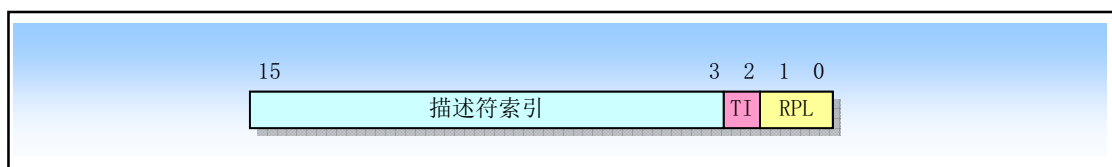


图 6-5 段选择符格式

其中索引值 (Index) 用于选择指定描述符表中 8192 ( $2^{13}$ ) 个描述符中的一个。处理器将该索引值乘上 8, 并加上描述符表的基地址即可访问表中指定的段描述符。表指示器 (Table Indicator - TI) 用于指定选择符所引用的描述符表。值为 0 表示指定 GDT 表, 值为 1 表示指定当前的 LDT 表。请求者特权级 (Requestor's Privilege Level - RPL) 用于保护机制。

由于 GDT 表的第一项(索引值为 0)没有被使用, 因此一个具有索引值 0 和表指示器值也为 0 的选择符 (也即指向 GDT 的第一项的选择符) 可以作为一个空(null)选择符。当一个段寄存器 (不能是 CS 或 SS) 加载了一个空选择符时, 处理器并不会产生一个异常。但是若使用这个段寄存器访问内存时就会产生一个异常。对于初始化还未使用的段寄存器以陷入意外的引用来说, 这个特性是很有用的。

在进入保护模式之前, 我们必须首先设置好将要用到的段描述符表, 例如全局描述符表 GDT。然后使用指令 `lgdt` 把描述符表的基地址告知 CPU (GDT 表的基地址存入 `gdtr` 寄存器)。再将机器状态字的保护模式标志置位即可进入 32 位保护运行模式。

### 6.3.2 代码注释

程序 6-2 linux/boot/setup.s

```

1 !
2 !      setup.s      (C) 1991 Linus Torvalds
3 !
4 ! setup.s is responsible for getting the system data from the BIOS,
5 ! and putting them into the appropriate places in system memory.
6 ! both setup.s and system has been loaded by the bootblock.
7 !
8 ! This code asks the bios for memory/disk/other parameters, and
9 ! puts them in a "safe" place: 0x90000-0x901FF, ie where the
10 ! boot-block used to be. It is then up to the protected mode
11 ! system to read them from there before the area is overwritten
12 ! for buffer-blocks.
13 !
14 ! setup.s 负责从 BIOS 中获取系统数据, 并将这些数据放到系统内存的适当
15 ! 地方。此时 setup.s 和 system 已经由 bootsect 引导块加载到内存中。
16 !
17 ! 这段代码询问 bios 有关内存/磁盘/其他参数, 并将这些参数放到一个
18 ! “安全的”地方: 0x90000-0x901FF, 也即原来 bootsect 代码块曾经在
19 ! 的地方, 然后在被缓冲块覆盖掉之前由保护模式的 system 读取。
20 !
21 ! NOTE! These had better be the same as in bootsect.s!
22 ! 以下这些参数最好和 bootsect.s 中的相同!
23 !
24 ! INITSEG = 0x9000      ! we move boot here - out of the way ! 原来 bootsect 所处的段。
25 ! SYSSEG  = 0x1000      ! system loaded at 0x10000 (65536). ! system 在 0x10000 (64KB) 处。
26 ! SETUPSEG = 0x9020     ! this is the current segment ! 本程序所在的段地址。

```

```

20
21 .globl begtext, begdata, begbss, endtext, enddata, endbss
22 .text
23 begtext:
24 .data
25 begdata:
26 .bss
27 begbss:
28 .text
29
30 entry start
31 start:
32
33 ! ok, the read went well so we get current cursor position and save it for
34 ! posterity.
35 ! ok, 整个读磁盘过程都很正常, 现在保存光标位置以备今后使用。
36 ! 这段代码使用 BIOS 中断取屏幕当前光标位置 (列、行), 并保存在内存 0x90000 处 (2 字节)。
37 ! 控制台初始化程序会到此处读取该值。
38 ! BIOS 中断 0x10 功能号 ah = 0x03, 读光标位置。
39 ! 输入: bh = 页号
40 ! 返回: ch = 扫描开始线; cl = 扫描结束线; dh = 行号 (0x00 顶端); dl = 列号 (0x00 最左边)。
41 !
42 ! 下句将 ds 置成 INITSEG (0x9000)。这已经在 bootsect 程序中设置过, 但是现在是 setup 程序,
43 ! Linus 觉得需要再重新设置一下。
44     mov     ax, #INITSEG      ! this is done in bootsect already, but...
45     mov     ds, ax
46     mov     ah, #0x03        ! read cursor pos
47     xor     bh, bh
48     int     0x10             ! save it in known place, con_init fetches
49     mov     [0], dx          ! it from 0x90000.
50
51 ! Get memory size (extended mem, kB)
52 ! 取扩展内存的大小值 (KB)。
53 ! 利用 BIOS 中断 0x15 功能号 ah = 0x88 取系统所含扩展内存大小并保存在内存 0x90002 处。
54 ! 返回: ax = 从 0x100000 (1M) 处开始的扩展内存大小 (KB)。若出错则 CF 置位, ax = 出错码。
55
56     mov     ah, #0x88
57     int     0x15
58     mov     [2], ax          ! 将扩展内存数值存在 0x90002 处 (1 个字)。
59
60 ! Get video-card data:
61 ! 下面这段用于取显卡当前显示模式。
62 ! 调用 BIOS 中断 0x10, 功能号 ah = 0x0f
63 ! 返回: ah = 字符列数; al = 显示模式; bh = 当前显示页。
64 ! 0x90004 (1 字) 存放当前页; 0x90006 存放显示模式; 0x90007 存放字符列数。
65
66     mov     ah, #0x0f
67     int     0x10
68     mov     [4], bx          ! bh = display page
69     mov     [6], ax          ! al = video mode, ah = window width
70
71 ! check for EGA/VGA and some config parameters

```

! 检查显示方式 (EGA/VGA) 并取参数。  
! 调用 BIOS 中断 0x10, 附加功能选择方式信息。功能号: ah = 0x12, bl = 0x10  
! 返回: bh = 显示状态。0x00 - 彩色模式, I/O 端口=0x3dX; 0x01 - 单色模式, I/O 端口=0x3bX。  
! bl = 安装的显示内存。0x00 - 64k; 0x01 - 128k; 0x02 - 192k; 0x03 = 256k。  
! cx = 显卡特性参数(参见程序后对 BIOS 视频中断 0x10 的说明)。

```

57
58     mov     ah, #0x12
59     mov     bl, #0x10
60     int     0x10
61     mov     [8], ax      ! 0x90008 = ??
62     mov     [10], bx     ! 0x9000A = 安装的显示内存, 0x9000B = 显示状态(彩色/单色)
63     mov     [12], cx     ! 0x9000C = 显卡特性参数。
64
65 ! Get hd0 data
! 取第一个硬盘的信息 (复制硬盘参数表)。
! 第 1 个硬盘参数表的首地址竟然是中断向量 0x41 的向量值! 而第 2 个硬盘参数表紧接在第 1 个表
! 的后面, 中断向量 0x46 的向量值也指向第 2 个硬盘的参数表首址。表的长度是 16 个字节 (0x10)。
! 下面两段程序分别复制 BIOS 有关两个硬盘的参数表, 0x90080 处存放第 1 个硬盘的表, 0x90090 处
! 存放第 2 个硬盘的表。
66
! 第 69 行语句从内存指定位置处读取一个长指针值并放入 ds 和 si 寄存器中。ds 中放段地址, si 是
! 段内偏移地址。这里是把内存地址 4 * 0x41 (= 0x104) 处保存的 4 个字节 (段和偏移值) 读出。
67     mov     ax, #0x0000
68     mov     ds, ax
69     lds     si, [4*0x41]      ! 取中断向量 0x41 的值, 也即 hd0 参数表的地址→ds:si
70     mov     ax, #INITSEG
71     mov     es, ax
72     mov     di, #0x0080      ! 传输的目的地址: 0x9000:0x0080 → es:di
73     mov     cx, #0x10        ! 共传输 16 字节。
74     rep
75     movsb
76
77 ! Get hd1 data
78
79     mov     ax, #0x0000
80     mov     ds, ax
81     lds     si, [4*0x46]      ! 取中断向量 0x46 的值, 也即 hd1 参数表的地址→ds:si
82     mov     ax, #INITSEG
83     mov     es, ax
84     mov     di, #0x0090      ! 传输的目的地址: 0x9000:0x0090 → es:di
85     mov     cx, #0x10
86     rep
87     movsb
88
89 ! Check that there IS a hd1 :-)
! 检查系统是否有第 2 个硬盘。如果没有则把第 2 个表清零。
! 利用 BIOS 中断调用 0x13 的取盘类型功能, 功能号 ah = 0x15;
! 输入: dl = 驱动器号 (0x8X 是硬盘: 0x80 指第 1 个硬盘, 0x81 第 2 个硬盘)
! 输出: ah = 类型码; 00 - 没有这个盘, CF 置位; 01 - 是软驱, 没有 change-line 支持;
!         02 - 是软驱(或其他可移动设备), 有 change-line 支持; 03 - 是硬盘。
90
91     mov     ax, #0x01500
92     mov     dl, #0x81

```

```

93      int      0x13
94      jc      no_disk1
95      cmp     ah,#3           ! 是硬盘吗? (类型 = 3 ?)。
96      je      is_disk1
97 no_disk1:
98      mov     ax,#INITSEG     ! 第 2 个硬盘不存在, 则对第 2 个硬盘表清零。
99      mov     es,ax
100     mov     di,#0x0090
101     mov     cx,#0x10
102     mov     ax,#0x00
103     rep
104     stosb
105 is_disk1:
106
107 ! now we want to move to protected mode ...
108 ! 现在我们要进入保护模式中了...
109
110     cli                     ! no interrupts allowed !      ! 从此开始不允许中断。
111
112 ! first we move the system to it's rightful place
113 ! 首先我们将 system 模块移到正确的位置。
114 ! bootsect 引导程序是将 system 模块读入到 0x10000 (64KB) 开始的位置。由于当时假设 system
115 ! 模块最大长度不会超过 0x80000 (512KB), 即其末端不会超过内存地址 0x90000, 所以 bootsect
116 ! 会把自己移动到 0x90000 开始的地方, 并把 setup 加载到它的后面。下面这段程序的用途是再把
117 ! 整个 system 模块移动到 0x00000 位置, 即把从 0x10000 到 0x8ffff 的内存数据块(512KB)整块地
118 ! 向内存低端移动了 0x10000 (64KB) 的位置。
119
120     mov     ax,#0x0000
121     cld                     ! 'direction'=0, movs moves forward
122 do_move:
123     mov     es,ax           ! destination segment ! es:di 是目的地址(初始为 0x0:0x0)
124     add     ax,#0x1000
125     cmp     ax,#0x9000     ! 已经把最后一段(从 0x8000 段开始的 64KB) 代码移动完?
126     jz      end_move       ! 是, 则跳转。
127     mov     ds,ax          ! source segment ! ds:si 是源地址(初始为 0x1000:0x0)
128     sub     di,di
129     sub     si,si
130     mov     cx,#0x8000     ! 移动 0x8000 字 (64KB 字节)。
131     rep
132     movsw
133     jmp     do_move
134
135 ! then we load the segment descriptors
136 ! 此后, 我们加载段描述符。
137 ! 从这里开始会遇到 32 位保护模式的操作, 因此需要 Intel 32 位保护模式编程方面的知识了, 有关
138 ! 这方面的信息请查阅列表后的简单介绍或附录中的详细说明。这里仅作概要说明。在进入保护模式
139 ! 中运行之前, 我们需要首先设置好需要使用的段描述符表。这里需要设置全局描述符表和中断描述
140 ! 符表。
141 !
142 ! 下面指令 lidt 用于加载中断描述符表 (IDT) 寄存器。它的操作数 (idt_48) 有 6 字节。前 2 字节
143 ! (字节 0-1) 是描述符表的字节长度值; 后 4 字节 (字节 2-5) 是描述符表的 32 位线性基地址, 其
144 ! 形式参见下面 218-220 行和 222-224 行说明。中断描述符表中的每一个 8 字节表项指出发生中断时
145 ! 需要调用的代码信息。与中断向量有些相似, 但要包含更多的信息。

```

```

!
! lgdt 指令用于加载全局描述符表 (GDT) 寄存器, 其操作数格式与 lidt 指令的相同。全局描述符
! 表中的每个描述符项 (8 字节) 描述了保护模式下数据段和代码段 (块) 的信息。其中包括段的
! 最大长度限制 (16 位)、段的线性地址基址 (32 位)、段的特权级、段是否在内存、读写许可权
! 以及其他一些保护模式运行的标志。参见后面 205-216 行。
129
130 end_move:
131     mov     ax, #SETUPSEG      ! right, forgot this at first. didn't work :- )
132     mov     ds, ax            ! ds 指向本程序 (setup) 段。
133     lidt    idt_48             ! load idt with 0,0                ! 加载 IDT 寄存器。
134     lgdt    gdt_48            ! load gdt with whatever appropriate ! 加载 GDT 寄存器。
135
136 ! that was painless, now we enable A20
! 以上的操作很简单, 现在我们开启 A20 地址线。
! 为了能够访问和使用 1MB 以上的物理内存, 我们需要首先开启 A20 地址线。参见本程序列表后
! 有关 A20 信号线的说明。关于所涉及的一些端口和命令, 可参考 kernel/chr_drv/keyboard.S
! 程序后对键盘接口的说明。至于机器是否真正开启了 A20 地址线, 我们还需要在进入保护模式
! 之后 (能访问 1MB 以上内存之后) 在测试一下。这个工作放在了 head.S 程序中 (32--36 行)。
137
138     call    empty_8042         ! 测试 8042 状态寄存器, 等待输入缓冲器空。
                                ! 只有当输入缓冲器为空时才可以对其执行写命令。
139     mov     al, #0xD1          ! command write ! 0xD1 命令码-表示要写数据到
140     out     #0x64, al          ! 8042 的 P2 端口。P2 端口的位 1 用于 A20 线的选通。
                                ! 数据要写到 0x60 口。
141     call    empty_8042         ! 等待输入缓冲器空, 看命令是否被接受。
142     mov     al, #0xDF          ! A20 on ! 选通 A20 地址线的参数。
143     out     #0x60, al
144     call    empty_8042         ! 若此时输入缓冲器为空, 则表示 A20 线已经选通。
145
146 ! well, that went ok, I hope. Now we have to reprogram the interrupts :- (
147 ! we put them right after the intel-reserved hardware interrupts, at
148 ! int 0x20-0x2F. There they won't mess up anything. Sadly IBM really
149 ! messed this up with the original PC, and they haven't been able to
150 ! rectify it afterwards. Thus the bios puts interrupts at 0x08-0x0f,
151 ! which is used for the internal hardware interrupts as well. We just
152 ! have to reprogram the 8259's, and it isn't fun.
!
! 希望以上一切正常。现在我们必须重新对中断进行编程 :- ( 我们将它们放在正好
! 处于 Intel 保留的硬件中断后面, 即 int 0x20-0x2F。在那里它们不会引起冲突。
! 不幸的是 IBM 在原 PC 机中搞糟了, 以后也没有纠正过来。所以 PC 机 BIOS 把中断
! 放在了 0x08-0x0f, 这些中断也被用于内部硬件中断。所以我们就必须重新对 8259
! 中断控制器进行编程, 这一点都没意思。
!
! PC 机使用 2 个 8259A 芯片, 关于对可编程控制器 8259A 芯片的编程方法请参见本程序后的介绍。
! 第 156 行上定义的两个字 (0x00eb) 是直接使用机器码表示的两条相对跳转指令, 起延时作用。
! 0xeb 是直接近跳转指令的操作码, 带 1 个字节的相对位移值。因此跳转范围是 -127 到 127。CPU
! 通过把这个相对位移值加到 EIP 寄存器中就形成一个新的有效地址。此时 EIP 指向下一条被执行
! 的指令。执行时所花费的 CPU 时钟周期数是 7 至 10 个。0x00eb 表示跳转值是 0 的一条指令, 因
! 此还是直接执行下一条指令。这两条指令共可提供 14--20 个 CPU 时钟周期的延迟时间。在 as86
! 中没有表示相应指令的助记符, 因此 Linus 在 setup.s 等一些汇编程序中就直接使用机器码来表
! 示这种指令。另外, 每个空操作指令 NOP 的时钟周期数是 3 个, 因此若要达到相同的延迟效果就
! 需要 6 至 7 个 NOP 指令。
153

```



```

! 8259 芯片主片端口是 0x20-0x21, 从片端口是 0xA0-0xA1。输出值 0x11 表示初始化命令开始, 它
! 是 ICW1 命令字, 表示边沿触发、多片 8259 级连、最后要发送 ICW4 命令字。
154      mov     al, #0x11                ! initialization sequence
155      out     #0x20, al                ! send it to 8259A-1 ! 发送到 8259A 主芯片。
156      .word   0x00eb, 0x00eb          ! jmp $+2, jmp $+2 ! '$' 表示当前指令的地址,
157      out     #0xA0, al                ! and to 8259A-2 ! 再发送到 8259A 从芯片。
158      .word   0x00eb, 0x00eb
! Linux 系统硬件中断号被设置成从 0x20 开始。参见表 2-2: 硬件中断请求信号与中断号对应表。
159      mov     al, #0x20                ! start of hardware int's (0x20)
160      out     #0x21, al                ! 送主芯片 ICW2 命令字, 设置起始中断号, 要送奇端口。
161      .word   0x00eb, 0x00eb
162      mov     al, #0x28                ! start of hardware int's 2 (0x28)
163      out     #0xA1, al                ! 送从芯片 ICW2 命令字, 从芯片的起始中断号。
164      .word   0x00eb, 0x00eb
165      mov     al, #0x04                ! 8259-1 is master
166      out     #0x21, al                ! 送主芯片 ICW3 命令字, 主芯片的 IR2 连从芯片 INT。
! 参见代码列表后的说明。
167      .word   0x00eb, 0x00eb
168      mov     al, #0x02                ! 8259-2 is slave
169      out     #0xA1, al                ! 送从芯片 ICW3 命令字, 表示从芯片的 INT 连到主芯
! 片的 IR2 引脚上。
170      .word   0x00eb, 0x00eb
171      mov     al, #0x01                ! 8086 mode for both
172      out     #0x21, al                ! 送主片 ICW4 命令字。8086 模式; 普通 EOI、非缓冲
! 方式。需发送指令来复位。初始化结束, 芯片就绪。
173      .word   0x00eb, 0x00eb
174      out     #0xA1, al                ! 送从芯片 ICW4 命令字, 内容同上。
175      .word   0x00eb, 0x00eb
176      mov     al, #0xFF                ! mask off all interrupts for now
177      out     #0x21, al                ! 屏蔽主芯片所有中断请求。
178      .word   0x00eb, 0x00eb
179      out     #0xA1, al                ! 屏蔽从芯片所有中断请求。
180
181 ! well, that certainly wasn't fun :-(. Hopefully it works, and we don't
182 ! need no steenking BIOS anyway (except for the initial loading :-).
183 ! The BIOS-routine wants lots of unnecessary data, and it's less
184 ! "interesting" anyway. This is how REAL programmers do it.
185 !
186 ! Well, now's the time to actually move into protected mode. To make
187 ! things as simple as possible, we do no register set-up or anything,
188 ! we let the gnu-compiled 32-bit programs do that. We just jump to
189 ! absolute address 0x00000, in 32-bit protected mode.
190 !
! 哼, 上面这段编程当然没劲:-(, 但希望这样能工作, 而且我们也不再需要乏味的 BIOS
! 了 (除了初始加载:-)。BIOS 子程序要求很多不必要的数, 而且它一点都没趣。那是
! “真正”的程序员所做的事。
!
! 好了, 现在是真正开始进入保护模式的时候了。为了把事情做得尽量简单, 我们并不对
! 寄存器内容进行任何设置。我们让 gnu 编译的 32 位程序去处理这些事。在进入 32 位保
! 护模式时我们仅是简单地跳转到绝对地址 0x00000 处。

```



! 当前特权级 CPL=0。此时段寄存器仍然指向与实地址模式中相同的线性地址处（在实地址模式下线性地址与物理内存地址相同）。在设置该比特位后，随后一条指令必须是一条段间跳转指令！用于刷新 CPU 当前指令队列。因为 CPU 是在执行一条指令之前就已从内存读取该指令并对其进行解码。然而在进入保护模式以后那些属于实模式的预先取得的指令信息就变得不再有效。而一条段间跳转指令就会刷新 CPU 的当前指令队列，即丢弃这些无效信息。另外，在 Intel 公司的手册上建议 80386 或以上 CPU 应该使用指令“mov cr0, ax”切换到保护模式。lmsw 指令仅用于兼容以前的 286 CPU。

```
191      mov     ax, #0x0001      ! protected mode (PE) bit      ! 保护模式比特位 (PE)。
192      lmsw   ax               ! This is it!                  ! 就这样加载机器状态字!
193      jmp    0, 8             ! jmp offset 0 of segment 8 (cs) ! 跳转至 cs 段偏移 0 处。
```

! 我们已经将 system 模块移动到 0x00000 开始的地方，所以上句中的偏移地址是 0。而段值 8 已经是保护模式下的段选择符了，用于选择描述符表和描述符表项以及所要求的特权级。段选择符长度为 16 位（2 字节）；位 0-1 表示请求的特权级 0-3，Linux 操作系统只用到两级：0 级（内核级）和 3 级（用户级）；位 2 用于选择全局描述符表（0）还是局部描述符表（1）；位 3-15 是描述符表项的索引，指出选择第几项描述符。所以段选择符 8（0b0000, 0000, 0000, 1000）表示请求特权级 0、使用全局描述符表 GDT 中第 2 个段描述符项，该项指出代码的基地址是 0（参见 209 行），因此这里的跳转指令就会去执行 system 中的代码。

```
194
195 ! This routine checks that the keyboard command queue is empty
196 ! No timeout is used - if this hangs there is something wrong with
197 ! the machine, and we probably couldn't proceed anyway.
```

! 下面这个子程序检查键盘命令队列是否为空。这里不使用超时方法 -  
! 如果这里死机，则说明 PC 机有问题，我们就没有办法再处理下去了。  
!

! 只有当输入缓冲器为空时（键盘控制器状态寄存器位 1 = 0）才可以对其进行写命令。

```
198 empty_8042:
199      .word   0x00eb, 0x00eb    ! 这是两个跳转指令的机器码(跳到下一句)，相当于延时操作。
200      in      al, #0x64         ! 8042 status port      ! 读 AT 键盘控制器状态寄存器。
201      test    al, #2           ! is input buffer full?    ! 测试位 1，输入缓冲器满？
202      jnz     empty_8042       ! yes - loop
203      ret
204
```

! 全局描述符表开始处。描述符表由多个 8 字节长的描述符项组成。这里给出了 3 个描述符项。  
! 第 1 项无用（206 行），但须存在。第 2 项是系统代码段描述符（208-211 行），第 3 项是系统数据段描述符（213-216 行）。

```
205 gdt:
206      .word   0, 0, 0, 0      ! dummy      ! 第 1 个描述符，不用。
207
208 ! 在 GDT 表中这里的偏移量是 0x08。它是内核代码段选择符的值。
208      .word   0x07FF          ! 8Mb - limit=2047 (0--2047，因此是 2048*4096=8Mb)
209      .word   0x0000          ! base address=0
210      .word   0x9A00          ! code read/exec      ! 代码段为只读、可执行。
211      .word   0x00C0          ! granularity=4096, 386 ! 颗粒度为 4096，32 位模式。
212
```

! 在 GDT 表中这里的偏移量是 0x10。它是内核数据段选择符的值。

```
213      .word   0x07FF          ! 8Mb - limit=2047 (0--2047，因此是 2048*4096=8Mb)
214      .word   0x0000          ! base address=0
215      .word   0x9200          ! data read/write     ! 数据段为可读可写。
216      .word   0x00C0          ! granularity=4096, 386 ! 颗粒度为 4096，32 位模式。
217
```

! 下面是加载中断描述符表寄存器 idtr 的指令 lidt 要求的 6 字节操作数。前 2 字节是 IDT 表的限长，

! 后 4 字节是 idt 表在线性地址空间中的 32 位基地址。CPU 要求在进入保护模式之前需设置 IDT 表，  
! 因此这里先设置一个长度为 0 的空表。

```
218 idt_48:
219     .word    0                ! idt limit=0
220     .word    0,0             ! idt base=0L
221
```

! 这是加载全局描述符表寄存器 gdt 的指令 lgdt 要求的 6 字节操作数。前 2 字节是 gdt 表的限长，  
! 后 4 字节是 gdt 表的线性基地址。这里全局表长度设置为 2KB (0x7ff 即可)，因为每 8 字节组成  
! 一个段描述符项，所以表中共可有 256 项。4 字节的线性基地址为 0x0009<<16 + 0x0200 + gdt，  
! 即 0x90200 + gdt。(符号 gdt 是全局表在本程序段中的偏移地址，见 205 行。)

```
222 gdt_48:
223     .word    0x800            ! gdt limit=2048, 256 GDT entries
224     .word    512+gdt,0x9      ! gdt base = 0X9xxxx
225
226 .text
227 endtext:
228 .data
229 enddata:
230 .bss
231 endbss:
```

### 6.3.3 其他信息

为了获取机器的基本参数，这段程序多次调用了 BIOS 中的中断，并开始涉及一些对硬件端口的操作。下面简要地描述程序中使用到的 BIOS 中断调用，并对 A20 地址线问题的缘由进行解释，最后提及关于 Intel 32 位保护模式运行的问题。

#### 6.3.3.1 当前内存映像

在 setup.s 程序执行结束后，系统模块 system 被移动到物理地址 0x0000 开始处，而从位置 0x90000 开始处则存放了内核将会使用的一些系统基本参数，示意图如图 6-6 所示。

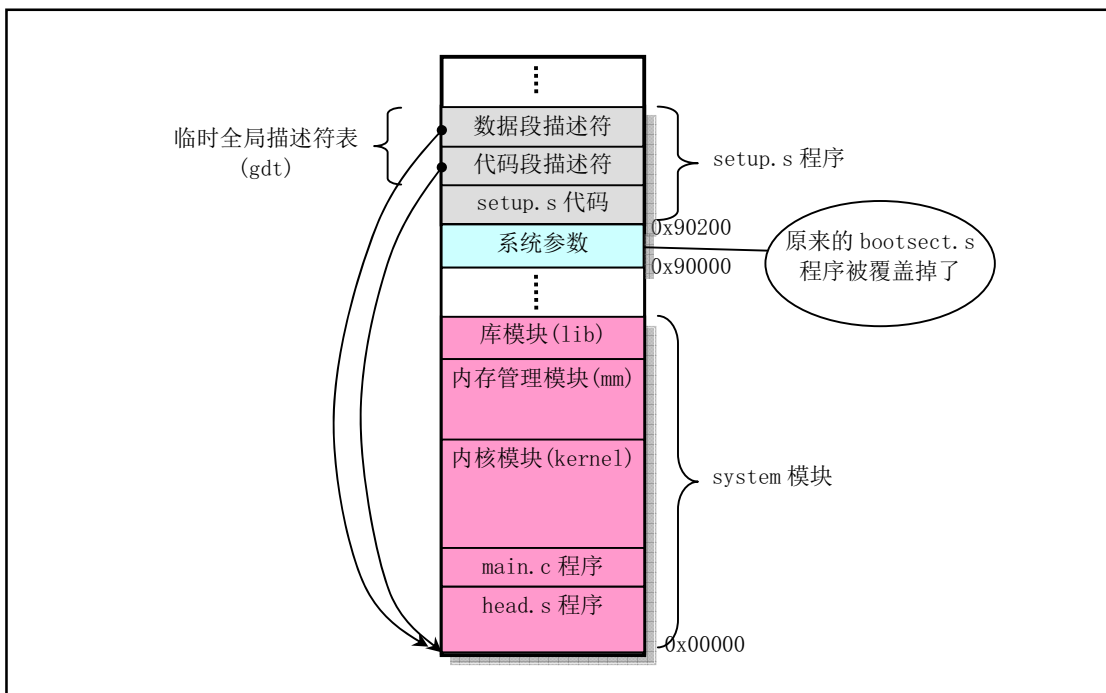


图 6-6 setup.s 程序结束后内存中程序示意图

此时临时全局表中有三个描述符，第一个是 NULL 不使用，另外两个分别是代码段描述符和数据段描述符。它们都指向系统模块的起始处，也即物理地址 0x0000 处。这样当 setup.s 中执行最后一条指令 'jmp 0,8 '（第 193 行）时，就会跳到 head.s 程序开始处继续执行下去。这条指令中的'8'是段选择符，用来指定所需使用的描述符项，此处是指 gdt 中的代码段描述符。'0'是描述符项指定的代码段中的偏移值。

6.3.3.2 BIOS 视频中断 0x10

这里说明上面程序中用到的 ROM BIOS 中视频中断调用的几个子功能。  
获取显示卡信息（其他辅助功能选择）：

表 6 - 3 获取显示卡信息（功能号： ah = 0x12，bl = 0x10）

输入/返回信息	寄存器	内容说明
输入信息	ah	功能号=0x12，获取显示卡信息
	bl	子功能号=0x10。
返回信息	bh	视频状态： 0x00 – 彩色模式（此时视频硬件 I/O 端口基地址为 0x3DX）； 0x01 – 单色模式（此时视频硬件 I/O 端口基地址为 0x3BX）； 注：其中端口地址中的 X 值可为 0 – f。
	bl	已安装的显示内存大小： 00 = 64K, 01 = 128K, 02 = 192K, 03 = 256K
	ch	特性连接器比特位信息： 比特位 说明 0 特性线 1，状态 2； 1 特性线 0，状态 2； 2 特性线 1，状态 1； 3 特性线 0，状态 1； 4-7 未使用(为 0)
	cl	视频开关设置信息： 比特位 说明 0 开关 1 关闭； 1 开关 2 关闭； 2 开关 3 关闭； 3 开关 4 关闭； 4-7 未使用。 原始 EGA/VGA 开关设置值： 0x00 MDA/HGC； 0x01-0x03 MDA/HGC； 0x04 CGA 40x25； 0x05 CGA 80x25； 0x06 EGA+ 40x25； 0x07-0x09 EGA+ 80x25； 0x0A EGA+ 80x25 单色； 0x0B EGA+ 80x25 单色。

### 6.3.3.3 硬盘基本参数表 (“INT 0x41”)

中断向量表中, int 0x41 的中断向量位置 ( $4 * 0x41 = 0x0000:0x0104$ ) 存放的并不是中断程序的地址, 而是第一个硬盘的基本参数表。对于 100%兼容的 BIOS 来说, 这里存放着硬盘参数表阵列的首地址 F000h:E401h。第二个硬盘的基本参数表入口地址存于 int 0x46 中断向量位置处。

表 6-4 硬盘基本参数信息表

位移	大小	英文名称	说明
0x00	字	cyl	柱面数
0x02	字节	head	磁头数
0x03	字		开始减小写电流的柱面(仅 PC XT 使用, 其他为 0)
0x05	字	wpcom	开始写前预补偿柱面号 (乘 4)
0x07	字节		最大 ECC 猝发长度 (仅 XT 使用, 其他为 0)
0x08	字节	ctl	控制字节 (驱动器步进选择) 位 0          未用 位 1          保留(0) (关闭 IRQ) 位 2          允许复位 位 3          若磁头数大于 8 则置 1 位 4          未用(0) 位 5          若在柱面数+1 处有生产商的坏区图, 则置 1 位 6          禁止 ECC 重试 位 7          禁止访问重试。
0x09	字节		标准超时值 (仅 XT 使用, 其他为 0)
0x0A	字节		格式化超时值 (仅 XT 使用, 其他为 0)
0x0B	字节		检测驱动器超时值 (仅 XT 使用, 其他为 0)
0x0C	字	lzone	磁头着陆(停止)柱面号
0x0E	字节	sect	每磁道扇区数
0x0F	字节		保留。

### 6.3.3.4 A20 地址线问题

1981 年 8 月, IBM 公司最初推出的个人计算机 IBM PC 使用的 CPU 是 Intel 8088。在该微机中地址线只有 20 根(A0 – A19)。在当时内存 RAM 只有几百 KB 或不到 1MB 时, 20 根地址线已足够用来寻址这些内存。其所能寻址的最高地址是 0xffff:0xffff, 也即 0x10ffef。对于超出 0x100000(1MB)的寻址地址将默认地环绕到 0x0ffef。当 IBM 公司于 1985 年引入 AT 机时, 使用的是 Intel 80286 CPU, 具有 24 根地址线, 最高可寻址 16MB, 并且有一个与 8088 完全兼容的实模式运行方式。然而, 在寻址值超过 1MB 时它却不能象 8088 那样实现地址寻址的环绕。但是当时已经有一些程序是利用这种地址环绕机制进行工作的。为了实现完全的兼容性, IBM 公司发明了使用一个开关来开启或禁止 0x100000 地址比特位。由于在当时的 8042 键盘控制器上恰好有空闲的端口引脚 (输出端口 P2, 引脚 P21), 于是便使用了该引脚来作为与门控制这个地址比特位。该信号即被称为 A20。如果它为零, 则比特 20 及以上地址都被清除。从而实现了兼容性。

由于在机器启动时, 默认条件下, A20 地址线是禁止的, 所以操作系统必须使用适当的方法来开启它。但是由于各种兼容机所使用的芯片集不同, 要做到这一点却是非常的麻烦。因此通常要在几种控制方法中选择。

对 A20 信号线进行控制的常用方法是通过设置键盘控制器的端口值。这里的 setup.s 程序 (138-144 行)即使用了这种典型的控制方式。对于其他一些兼容微机还可以使用其他方式来做到对 A20 线的控制。

有些操作系统将 A20 的开启和禁止作为实模式与保护运行模式之间进行转换的标准过程中的一部分。由于键盘的控制器速度很慢，因此就不能使用键盘控制器对 A20 线来进行操作。为此引进了一个 A20 快速门选项(Fast Gate A20)，它使用 I/O 端口 0x92 来处理 A20 信号线，避免了使用慢速的键盘控制器操作方式。对于不含键盘控制器的系统就只能使用 0x92 端口来控制，但是该端口也有可能被其他兼容微机上的设备（如显示芯片）所使用，从而造成系统错误的操作。

还有一种方式是通过读 0xee 端口来开启 A20 信号线，写该端口则会禁止 A20 信号线。

### 6.3.3.5 8259A 中断控制器的编程方法

在第 2 章中我们已经概要介绍了中断机制的基本原理和 PC/AT 兼容微机中使用的硬件中断子系统。这里我们首先介绍 8259A 芯片的工作原理，然后详细说明 8259A 芯片的编程方法以及 Linux 内核对其设置的工作方式。

#### 1. 8259A 芯片工作原理

前面已经说过，在 PC/AT 系列兼容机中使用了级联的两片 8259A 可编程控制器（PIC）芯片，共可管理 15 级中断向量，参见图 2-20 所示。其中从芯片的 INT 引脚连接到主芯片的 IR2 引脚上。主 8259A 芯片的端口基地址是 0x20，从芯片是 0xA0。一个 8259A 芯片的逻辑框图见图 6-7 所示。

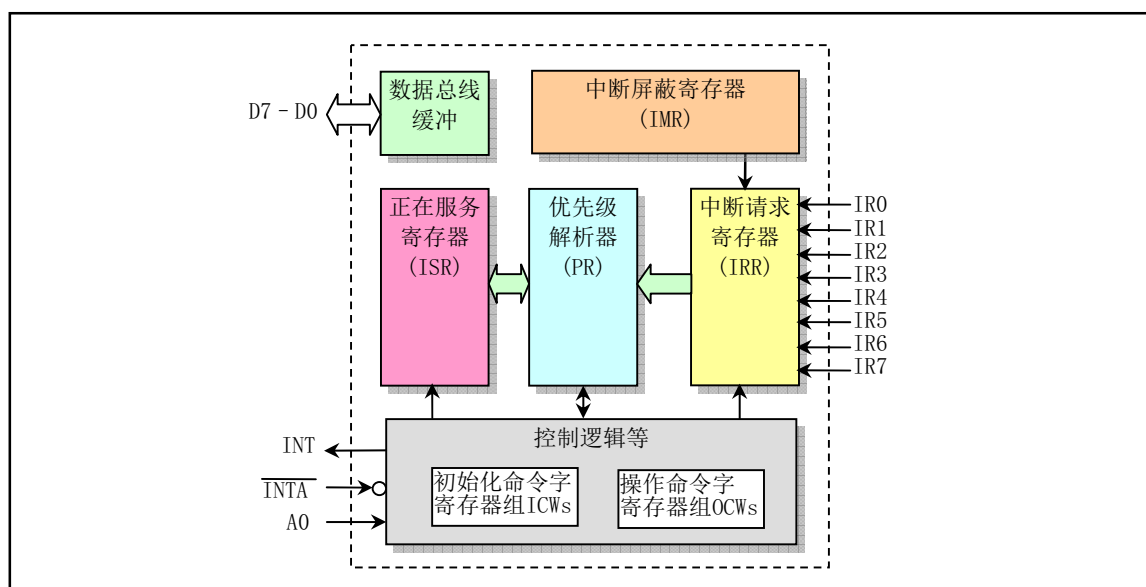


图 6-7 可编程中断控制器 8259A 芯片框图

图中，中断请求寄存器 IRR（Interrupt Request Register）用来保存中断请求输入引脚上所有请求服务中断级，寄存器的 8 个比特位（D7—D0）分别对应引脚 IR7—IR0。中断屏蔽寄存器 IMR（Interrupt Mask Register）用于保存被屏蔽的中断请求线对应的比特位，寄存器的 8 位也是对应 8 个中断级。哪个比特位被置 1 就屏蔽哪一级中断请求。即 IMR 对 IRR 进行处理，其每个比特位对应 IRR 的每个请求比特位。对高优先级输入线的屏蔽并不会影响低优先级中断请求线的输入。优先级解析器 PR（Priority Resolver）用于确定 IRR 中所设置比特位的优先级，选通最高优先级的中断请求到正在服务寄存器 ISR（In-Service Register）中。ISR 中保存着正在接受服务的中断请求。控制逻辑框中的寄存器组用于接受 CPU 产生的两类命令。在 8259A 可以正常操作之前，必须首先设置初始化命令字 ICW（Initialization Command Words）寄存器组的内容。而在其工作过程中，则可以使用写入操作命令字 OCW（Operation Command Words）寄存器组来随时设置和管理 8259A 的工作方式。A0 线用于选择操作的寄存器。在 PC/AT 微机系统中，当 A0 线为 0 时芯片的端口地址是 0x20 和 0xA0（从芯片），当 A0=1 时端口就是 0x21 和 0xA1。

来自各个设备的中断请求线分别连接到 8259A 的 IR0—IR7 中断请求引脚上。当这些引脚上有一个或多个中断请求信号到来时，中断请求寄存器 IRR 中相应的比特位被置位锁存。此时若中断屏蔽寄存器 IMR 中对应位被置位，则相应的中断请求就不会送到优先级解析器中。对于未屏蔽的中断请求被送到优先级解析器之后，优先级最高的中断请求会被选出。此时 8259A 就会向 CPU 一个 INT 信号，而 CPU 则会在执行完当前的一条指令之后向 8259A 发送一个 INTA 来响应中断信号。8259A 在收到这个响应信号之后就会把所选出的最高优先级中断请求保存到正在服务寄存器 ISR 中，即 ISR 中对应中断请求级的比特位被置位。与此同时，中断请求寄存器 IRR 中的对应比特位被复位，表示该中断请求开始正被处理中。

此后，CPU 会向 8259A 发出第 2 个 INTA 脉冲信号，该信号用于通知 8259A 送出中断号。因此在该脉冲信号期间 8259A 就会把一个代表中断号的 8 位数据发送到数据总线上供 CPU 读取。

到此为止，CPU 中断周期结束。如果 8259A 使用的是自动结束中断 AEOI(Automatic End of Interrupt)方式，那么在第 2 个 INTA 脉冲信号的结尾处正在服务寄存器 ISR 中的当前服务中断比特位就会被复位。否则的话，若 8259A 处于非自动结束方式，那么在中断服务程序结束时程序就需要向 8259A 发送一个结束中断(EOI)命令以复位 ISR 中的比特位。如果中断请求来自接联的第 2 个 8259A 芯片，那么就需要向两个芯片都发送 EOI 命令。此后 8259A 就会去判断下一个最高优先级的中断，并重复上述处理过程。下面我们先给出初始化命令字和操作命令字的编程方法，然后再对其中用到的一些操作方式作进一步说明。

## 2. 初始化命令字编程

可编程控制器 8259A 主要有 4 种工作方式：①全嵌套方式；②循环优先级方式；③特殊屏蔽方式和④程序查询方式。通过对 8259A 进行编程，我们可以选定 8259A 的当前工作方式。编程时分两个阶段。一是在 8259A 工作之前对每个 8259A 芯片 4 个初始化命令字(ICW1—ICW4)寄存器的写入编程；二是在工作过程中随时对 8259A 的 3 个操作命令字(OCW1—OCW3)进行编程。在初始化之后，操作命令字的内容可以在任何时候写入 8259A。下面我们先说明对 8259A 初始化命令字的编程操作。初始化命令字的编程操作流程见图 6-8 所示。由图可以看出，对 ICW1 和 ICW2 的设置是必需的。而只有当系统中包括多片 8259A 芯片并且是接连的情况下才需要对 ICW3 进行设置。这需要在 ICW1 的设置中明确指出。另外，是否需要 ICW4 进行设置也需要在 ICW1 中指明。

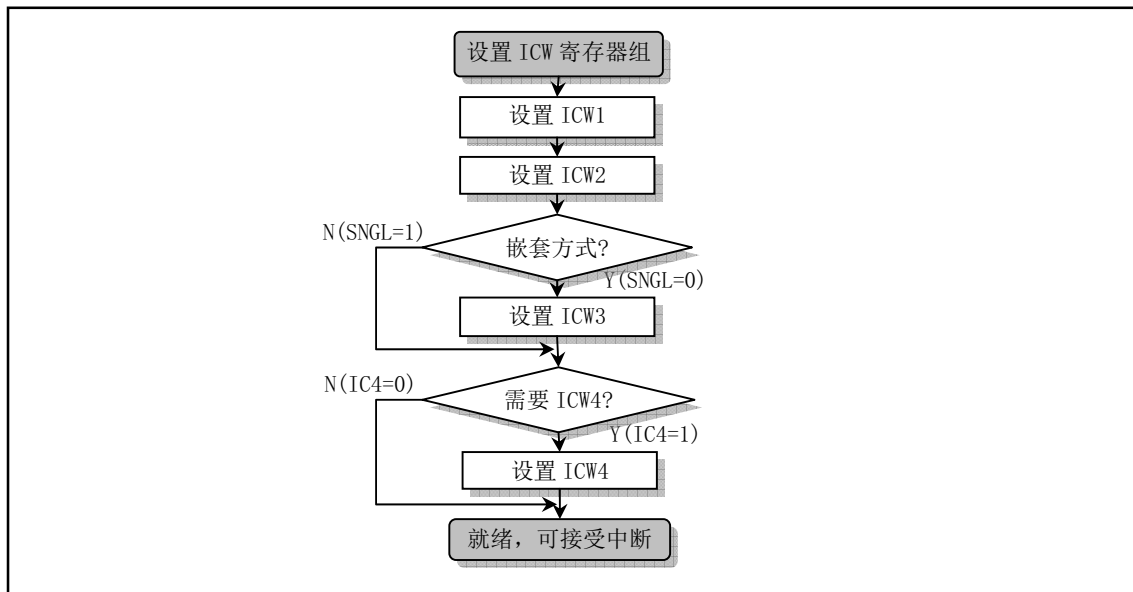


图 6-8 8259A 初始化命令字设置顺序

(1) ICW1 当发送的字节第 5 比特位(D4)=1 并且地址线 A0=0 时，表示是对 ICW1 编程。此时对于 PC/AT

微机系统的多片级联情况下，8259A 主芯片的端口地址是 0x20，从芯片的端口地址是 0xA0。ICW1 的格式如表 6-5 所示：

表 6-5 中断初始化命令字 ICW1 格式

位	名称	含义
D7	A7	A7—A5 表示在 MCS80/85 中用于中断服务过程的页面起始地址。与 ICW2 中的 A15—A8 共同组成。这几位对 8086/88 处理器无用。
D6	A6	
D5	A5	
D4	1	恒为 1
D3	LTIM	1 - 电平触发中断方式；0 - 边沿触发方式。
D2	ADI	MCS80/85 系统用于 CALL 指令地址间隔。对 8086/88 处理器无用。
D1	SNGL	1 - 单片 8259A； 0 - 多片。
D0	IC4	1 - 需要 ICW4； 0 - 不需要。

在 Linux 0.11 内核中，ICW1 被设置为 0x11。表示中断请求是边沿触发、多片 8259A 级联并且最后需要发送 ICW4。

(2) ICW2 用于设置芯片送出的中断号的高 5 位。中断号在设置了 ICW1 之后，当 A0=1 时表示对 ICW2 进行设置。此时对于 PC/AT 微机系统的多片级联情况下，8259A 主芯片的端口地址是 0x21，从芯片的端口地址是 0xA1。ICW2 格式见表 6-6 所示。

表 6-6 中断初始化命令字 ICW2 格式

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	A15/T7	A14/T6	A13/T5	A12/T4	A11/T3	A10	A9	A8

在 MCS80/85 系统中，位 D7—D0 表示的 A15—A8 与 ICW1 设置的 A7-A5 组成中断服务程序页面地址。在使用 8086/88 处理器的系统或兼容系统中 T7—T3 是中断号的高 5 位，与 8259A 芯片自动设置的低 3 位组成一个 8 位的中断号。8259A 在收到第 2 个中断响应脉冲 INTA 时会送到数据总线上，以供 CPU 读取。

Linux 0.11 系统把主片的 ICW2 设置为 0x20，表示主片中断请求 0 级—7 级对应的中断号范围是 0x20—0x27。而从片的 ICW2 被设置成 0x28，表示从片中断请求 8 级—15 级对应的中断号范围是 0x28—0x2f。

(3) ICW3 用于具有多个 8259A 芯片级联时，加载 8 位的从寄存器 (Slave Register)。端口地址同上。ICW3 格式见表 6-7 所示。

表 6-7 中断初始化命令字 ICW3 格式

	A0	D7	D6	D5	D4	D3	D2	D1	D0
主片：	1	S7	S6	S5	S4	S3	S2	S1	S0
从片：	1	0	0	0	0	0	ID2	ID1	ID0

主片 S7—S0 各比特位对应级联的从片。哪位为 1 则表示主片的该中断请求引脚 IR 上信号来自从片，



否则对应的 IR 引脚没有连从片。

从片的 ID2—ID0 三个比特位对应各从片的标识号，即连接到主片的中断级。当某个从片接收到级联线（CAS2—CAS0）输入的值与自己的 ID2—ID0 相等时，则表示此从片被选中。此时该从片应该向数据总线发送从片当前选中中断请求的中断号。

Linux 0.11 内核把 8259A 主片的 ICW3 设置为 0x04，即 S2=1，其余各位为 0。表示主芯片的 IR2 引脚连接一个从芯片。从芯片的 ICW3 被设置为 0x02，即其标识号为 2。表示从片连接到主片的 IR2 引脚。因此，中断优先级的排列次序为 0 级最高，接下来是从片上的 8—15 级，最后是 3—7 级。

(4) ICW4 当 ICW1 的位 0（IC4）置位时，表示需要 ICW4。地址线 A0=1。端口地址同上说明。ICW4 格式见表 6-8 所示。

表 6-8 中断初始化命令字 ICW4 格式

位	名称	含义
D7	0	恒为 0
D6	0	恒为 0
D5	0	恒为 0
D4	SFNM	1 – 选择特殊全嵌套方式； 0 – 普通全嵌套方式。
D3	BUF	1 – 缓冲方式； 0 – 非缓冲方式。
D2	M/S	1 – 缓冲方式下主片； 0 – 缓冲方式下从片。
D1	AEOI	1 – 自动结束中断方式； 0 – 非自动结束方式。
D0	μ PM	1 – 8086/88 处理器系统； 0 – MCS80/85 系统。

Linux 0.11 内核送往 8259A 主芯片和从芯片的 ICW4 命令字的值均为 0x01。表示 8259A 芯片被设置成普通全嵌套、非缓冲、非自动结束中断方式，并且用于 8086 及其兼容系统。

### 3. 操作命令字编程

在对 8259A 设置了初始化命令字寄存器后，芯片就已准备好接收设备的中断请求信号了。但在 8259A 工作期间，我们也可以利用操作命令字 OCW1—OCW3 来监测 8259A 的工作状况，或者随时改变初始化时设定的 8259A 的工作方式。

(1) OCW1 用于对 8259A 中中断屏蔽寄存器 IMR 进行读/写操作。地址线 A0 需为 1。端口地址说明同上。OCW1 格式见表 6-9 所示。

表 6-9 中断操作命令字 OCW1 格式

A0	D7	D6	D5	D4	D3	D2	D1	D0
1	M7	M6	M5	M4	M3	M2	M1	M0

位 D7—D0 对应 8 个中断请求 7 级—0 级的屏蔽位 M7—M0。若 M=1，则屏蔽对应中断请求级；若 M=0，则允许对应的中断请求级。另外，屏蔽高优先级并不会影响其他低优先级的中断请求。

在 Linux 0.11 内核初始化过程中，代码在设置好相关的设备驱动程序后就会利用该操作命令字来修改相关中断请求屏蔽位。例如在软盘驱动程序初始化结束时，为了允许软驱设备发出中断请求，就会读端口 0x21 以取得 8259A 芯片的当前屏蔽字节，然后与上 ~0x40 来复位对应软盘控制器连接的中断请求 6 的屏蔽位，最后再写回中断屏蔽寄存器中。参见 kernel/blk\_drv/floppy.c 程序第 461 行。

(2) OCW2 用于发送 EOI 命令或设置中断优先级的自动循环方式。当比特位 D4D3 = 00，地址线 A0=0

时表示对 OCW2 进行编程设置。操作命令字 OCW2 的格式见表 6-10 所示。

表 6-10 中断操作命令字 OCW2 格式

位	名称	含义
D7	R	优先级循环状态。
D6	SL	优先级设定标志。
D5	EOI	非自动结束标志。
D4	0	恒为 0。
D3	0	恒为 0。
D2	L2	L2—L0 3 位组成级别号，分别对应中断请求级别 IRQ0--IRQ7（或 IRQ8—IRQ15）。
D1	L1	
D0	L0	

其中位 D7—D5 的组合的作用和含义见表 6-11 所示。其中带有\*号者可通过设置 L2--L0 来指定优先级使 ISR 复位，或者选择特殊循环优先级成为当前最低优先级。

表 6-11 操作命令字 OCW2 的位 D7--D5 组合含义

R(D7)	SL(D6)	EOI(D5)	含义	类型
0	0	1	非特殊结束中断 EOI 命令（全嵌套方式）。	结束中断
0	1	1	*特殊结束中断 EOI 命令（非全嵌套方式）。	
1	0	1	非特殊结束中断 EOI 命令时循环。	优先级自动循环
1	0	0	自动结束中断 AEOI 方式时循环（设置）。	
0	0	0	自动结束中断 AEOI 方式时循环（清除）。	
1	1	1	*特殊结束中断 EOI 命令时循环。	特殊循环
1	1	0	*设置优先级命令。	
0	1	0	无操作。	

Linux 0.11 内核仅使用该操作命令字在中断处理过程结束之前向 8259A 发送结束中断 EOI 命令。所使用的 OCW2 值为 0x20，表示全嵌套方式下的非特殊结束中断 EOI 命令。

(3) OCW3 用于设置特殊屏蔽方式和读取寄存器状态（IRR 和 ISR）。当 D4D3=01、地址线 A0=0 时，表示对 OCW3 进行编程（读/写）。但在 Linux 0.11 内核中并没有用到该操作命令字。OCW3 的格式见表 6-12 所示。

表 6-12 中断操作命令字 OCW3 格式

位	名称	含义
D7	0	恒为 0。
D6	ESMM	对特殊屏蔽方式操作。 D6—D5 为 11 - 设置特殊屏蔽； 10 - 复位特殊屏蔽。
D5	SMM	
D4	0	恒为 0。
D3	1	恒为 1。
D2	P	1 - 查询（POLL）命令； 0 - 无查询命令。
D1	RR	在下一个 RD 脉冲时读寄存器状态。

D0	RIS	D1—D0 为 11 – 读正在服务寄存器 ISR; 10 – 读中断请求寄存器 IRR。
----	-----	---

#### 4. 8259A 操作方式说明

在说明 8259A 初始化命令字和操作命令字的编程过程中,提及了 8259A 的一些工作方式。下面对几种常见的方式给出详细说明,以便能更好地理解 8259A 芯片的运行方式。

##### (1) 全嵌套方式

在初始化之后,除非使用了操作命令字改变过 8259A 的工作方式,否则它会自动进入这种全嵌套工作方式。在这种工作方式下,中断请求优先级的秩序是从 0 级到 7 级(0 级优先级最高)。当 CPU 响应一个中断,那么最高优先级中断请求就被确定,并且该中断请求的中断号会被放到数据总线上。另外,正在服务寄存器 ISR 的相应比特位会被置位,并且该比特位的置位状态将一直保持到从中断服务过程返回之前发送结束中断 EOI 命令为止。如果在 ICW4 命令字中设置了自动中断结束 AEOI 比特位,那么 ISR 中的比特位将会在 CPU 发出第 2 个中断响应脉冲 INTA 的结束边沿被复位。在 ISR 有置位比特位期间,所有相同优先级和低优先级的中断请求将被暂时禁止,但允许更高优先级中断请求得到响应和处理。再者,中断屏蔽寄存器 IMR 的相应比特位可以分别屏蔽 8 级中断请求,但屏蔽任意一个中断请求并不会影响其他中断请求的操作。最后,在初始化命令字编程之后,8259A 引脚 IR0 具有最高优先级,而 IR7 的优先级最低。Linux 0.11 内核代码即把系统的 8259A 芯片设置工作在这个方式下。

##### (2) 中断结束 (EOI) 方法

如上所述,正在服务寄存器 ISR 中被处理中断请求对应的比特位可使用两种方式来复位。其一是当 ICW4 中的自动中断结束 AEOI 比特位置位时,通过在 CPU 发出的第 2 个中断响应脉冲 INTA 的结束边沿被复位。这种方法称为自动中断结束 (AEOI) 方法。其二是在从中断服务过程返回之前发送结束中断 EOI 命令来复位。这种方法称为程序中断结束 (EOI) 方法。在接联系统中,从片中断服务程序需要发送两个 EOI 命令,一个用于从片,另一个用于主片。

程序发出 EOI 命令的方法有两种格式。一种称为特殊 EOI 命令,另一种称为非特殊 EOI 命令。特殊的 EOI 命令用于非全嵌套方式下,可用于指定 EOI 命令具体复位的中断级比特位。即在向芯片发送特殊 EOI 命令时需要指定被复位的 ISR 中的优先级。特殊 EOI 命令使用操作命令字 OCW2 发送,高 3 比特位是 011,最低 3 位用来指定优先级。在目前的 Linux 系统中就使用了这种特殊 EOI 命令。用于全嵌套方式的非特殊 EOI 命令会自动地把当前正在服务寄存器 ISR 中最高优先级比特位复位。因为在全嵌套方式下 ISR 中最高优先级比特位肯定是最后响应和服务的优先级。它也使用 OCW2 来发出,但最高 3 比特位需要为 001。本书讨论的 Linux 0.11 系统中则使用了这种非特殊 EOI 命令。

##### (3) 特殊全嵌套方式

在 ICW4 中设置的特殊全嵌套方式 (D4=1) 主要用于接连的大系统中,并且每个从片中的优先级需要保存。这种方式与上述普通全嵌套方式相似,但有以下两点例外:

a. 当从某个从片发出的中断请求正被服务时,该从片并不会被主片的优先级排除。因此该从片发出的其他更高优先级中断请求将被主片识别,主片会立刻向 CPU 发出中断。而在上述普通全嵌套方式中,当一个从片中断请求正在被服务时,该从片会被主片屏蔽掉。因此从该从片发出的更高优先级中断请求就不能被处理。

b. 当退出中断服务程序时,程序必须检查当前中断服务是否是从片发出的唯一一个中断请求。检查的方法是先向从片发出一个非特殊中断结束 EOI 命令,然后读取其正在服务寄存器 ISR 的值。检查此时该值是否为 0。如果是 0,则表示可以再向主片发送一个非特殊 EOI 命令。若不为 0,则无需向主片发送 EOI 命令。

##### (4) 多片级联方式

8259A 可以被很容易地连接成一个主片和若干个从片组成的系统。若使用 8 个从片那么最多可控制 64 个中断优先级。主片通过 3 根级联线来控制从片。这 3 根级联线相当于从片的选片信号。在级联方式中,从片的中断输出端被连接到主片的中断请求输入引脚上。当从片的一个中断请求线被处理并被响应

时，主片会选择该从片把相应的中断号放到数据总线上。

在级联系统中，每个 8259A 芯片必须独立地进行初始化，并且可以工作在不同方式下。另外，要分别对主片和从片的初始化命令字 ICW3 进行编程。在操作过程中也需要发送 2 个中断结束 EOI 命令。一个用于主片，另一个用于从片。

#### (5) 自动循环优先级方式

当我们在管理优先级相同的设备时，就可以使用 OCW2 把 8259A 芯片设置成自动循环优先级方式。即在一个设备接受服务后，其优先级自动变成最低的。优先级依次循环变化。最不利的情况是当一个中断请求来到时需要等待它之前的 7 个设备都接受了服务之后才能得到服务。

#### (6) 中断屏蔽方式

中断屏蔽寄存器 IMR 可以控制对每个中断请求的屏蔽。8259A 可设置两种屏蔽方式。对于一般普通屏蔽方式，使用 OCW1 来设置 IMR。IMR 的各比特位 (D7--D0) 分别作用于各个中断请求引脚 IR7 -- IR0。屏蔽一个中断请求并不会影响其他优先级的中断请求。对于一个中断请求在响应并被服务期间（没有发送 EOI 命令之前），这种普通屏蔽方式会使得 8259A 屏蔽所有低优先级的中断请求。但有些应用场合可能需要中断服务过程能动态地改变系统的优先级。为了解决这个问题，8259A 中引进了特殊屏蔽方式。我们需要使用 OCW3 首先设置这种方式 (D6、D5 比特位)。在这种特殊屏蔽方式下，OCW1 设置的屏蔽信息会使所有未被屏蔽的优先级中断均可以在某个中断过程中被响应。

#### (7) 读寄存器状态

8259A 中有 3 个寄存器 (IMR、IRR 和 ISR) 可让 CPU 读取其状态。IMR 中的当前屏蔽信息可以通过直接读取 OCW1 来得到。在读 IRR 或 ISR 之前则需要首先使用 OCW3 输出读取 IRR 或 ISR 的命令，然后才可以进行读操作。

## 6.4 head.s 程序

### 6.4.1 功能描述

head.s 程序在被编译生成目标文件后会与内核其他程序一起被链接成 system 模块，位于 system 模块的最前面开始部分，这也就是为什么称其为头部(head)程序的原因。system 模块将被放置在磁盘上 setup 模块之后开始的扇区中，即从磁盘上第 6 个扇区开始放置。一般情况下 Linux 0.11 内核的 system 模块大约有 120KB 左右，因此在磁盘上大约占 240 个扇区。

从这里开始，内核完全都是在保护模式下运行了。heads.s 汇编程序与前面的语法格式不同，它采用的是 AT&T 的汇编语言格式，并且需要使用 GNU 的 gas 和 gld<sup>7</sup> 进行编译连接。因此请注意代码中赋值的方向是从左到右。

这段程序实际上处于内存绝对地址 0 处开始的地方。这个程序的功能比较单一。首先是加载各个数据段寄存器，重新设置中断描述符表 idt，共 256 项，并使各个表项均指向一个只报错误的哑中断子程序 ignore\_int。中断描述符表中每个描述符项也占 8 字节，其格式见图 6-9 所示。

<sup>7</sup> 在当前的 Linux 操作系统中，gas 和 gld 已经分别更名为 as 和 ld。

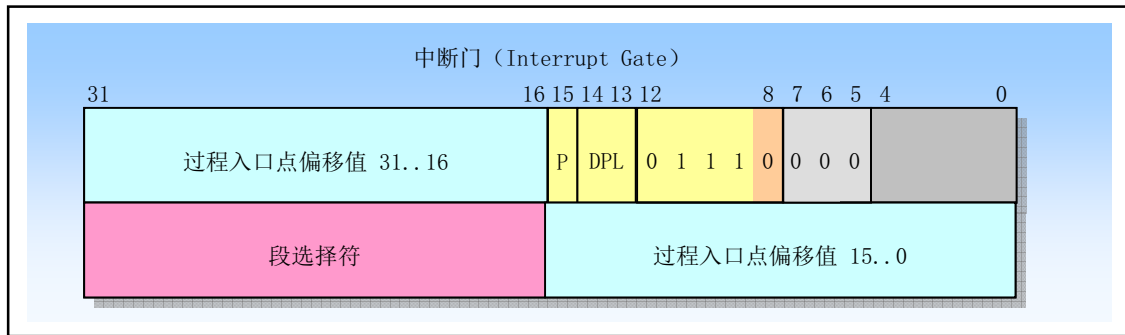


图 6-9 中断描述符表 IDT 中的中断门描述符格式

其中，P 是段存在标志；DPL 是描述符的优先级。在 head.s 程序中，中断门描述符中段选择符设置为 0x0008，表示该中断处理子程序在内核代码中。偏移值被设置为 ignore\_int 中断处理子程序在 head.s 程序中的偏移值。由于 head.s 程序被移动到从内存地址 0 开始处，因此该偏移值也就是中断处理子程序在内核代码段中的偏移值。由于内核代码段一直存在于内存中，并且特权级为 0，即 P=1，DPL=00。因此中断门描述符的字节 5 和字节 4 的值应该是 0x8E00。

在设置好中断描述符表之后，本程序又重新设置了全局段描述符表 gdt。实际上新设置的 GDT 表与原来在 setup.s 程序中设置的 GDT 表描述符除了在段限长上有些区别以外（原为 8MB，现为 16MB），其他内容完全一样。当然我们也可以在 setup.s 程序中就把描述符的段限长直接设置成 16MB，然后直接把原 GDT 表移动到内存适当位置处。因此这里重新设置 GDT 的主要原因是为了把 gdt 表放在内存内核代码比较合理的地方。前面设置的 GDT 表处于内存 0x902XX 处。这个地方将在内核初始化后用作内存高速缓冲区的一部分。

接着使用物理地址 0 与 1MB 开始处的字节内容相比较的方法，检测 A20 地址线是否已真的开启。如果没有开启，则在访问高于 1MB 物理内存地址时 CPU 实际只会循环访问 (IP MOD 1Mb) 地址处的内容，也即与访问从 0 地址开始对应字节的内容都相同。如果检测下来发现没有开启，则进入死循环。然后程序测试 PC 机是否含有数学协处理器芯片（80287、80387 或其兼容芯片），并在控制寄存器 CR0 中设置相应的标志位。

接着设置管理内存的分页处理机制，将页目录表放在绝对物理地址 0 开始处（也是本程序所处的物理内存位置，因此这段程序将被覆盖掉），紧随后面放置共可寻址 16MB 内存的 4 个页表，并分别设置它们的表项。页目录表项和页表项格式见图 6-10 所示。其中 P 是页面存在于内存标志；R/W 是读写标志；U/S 是用户/超级用户标志；A 是页面已访问标志；D 是页面内容已修改标志；最左边 20 比特是表项对应页面在物理内存中页面地址的高 20 比特位。

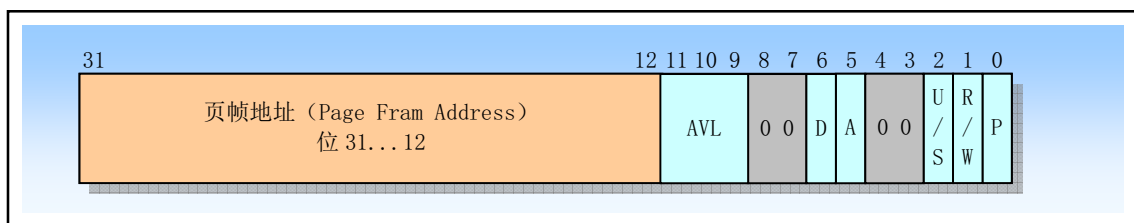


图 6-10 页目录表项和页表项结构

这里每个表项的属性标志都被设置成 0x07 (P=1、U/S=1、R/W=1)，表示该页存在、用户可读写。这样设置内核页表属性的原因是：CPU 的分段机制和分页管理都有保护方法。分页机制中页目录表和页表项中设置的保护标志 (U/S、R/W) 需要与段描述符中的特权级 (PL) 保护机制一起组合使用。但段

描述符中的 PL 起主要作用。CPU 会首先检查段保护，然后再检查页保护。如果当前特权级 CPL < 3（例如 0），则说明 CPU 正在以超级用户（Supervisor）身份运行。此时所有页面都能访问，并可随意进行内存读写操作。如果 CPL = 3，则说明 CPU 正在以用户（User）身份运行。此时只有属于 User 的页面（U/S=1）可以访问，并且只有标记为可读写的页面（W/R = 1）是可写的。而此时属于超级用户的页面（U/S=0）则既不可写、也不可以读。由于内核代码有些特别之处，即其中包含有任务 0 和任务 1 的代码和数据。因此这里把页面属性设置为 0x7 就可保证这两个任务代码不仅可以在用户态下执行，而且又不能随意访问内核资源。

最后，head.s 程序利用返回指令将预先放置在堆栈中的/init/main.c 程序的入口地址弹出，去运行 main() 程序。

## 6.4.2 代码注释

程序 6-3 linux/boot/head.s

```

1  /*
2  *  linux/boot/head.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  head.s contains the 32-bit startup code.
9  *
10 * NOTE!!! Startup happens at absolute address 0x00000000, which is also where
11 * the page directory will exist. The startup code will be overwritten by
12 * the page directory.
13 */
14 /*
15 *  head.s 含有 32 位启动代码。
16 * 注意!!! 32 位启动代码是从绝对地址 0x00000000 开始的，这里也同样是页目录将存在的地方，
17 * 因此这里的启动代码将被页目录覆盖掉。
18 */
19 .text
20 .globl _idt, _gdt, _pg_dir, _tmp_floppy_area
21 _pg_dir:                                # 页目录将会存放在这里。
22
23 # 再次注意!!! 这里已经处于 32 位运行模式，因此这里的$0x10 并不是把地址 0x10 装入各个
24 # 段寄存器，它现在其实是全局段描述符表中的偏移值，或者更准确地说是一个描述符表项
25 # 的选择符。有关选择符的说明请参见 setup.s 中 193 行下的说明。这里$0x10 的含义是请求
26 # 特权级 0(位 0-1=0)、选择全局描述符表(位 2=0)、选择表中第 2 项(位 3-15=2)。它正好指
27 # 向表中的数据段描述符项。（描述符的具体数值参见前面 setup.s 中 212, 213 行）
28 # 下面代码的含义是：设置 ds, es, fs, gs 为 setup.s 中构造的数据段（全局段描述符表第 2 项）
29 # 的选择符=0x10，并将堆栈放置在 stack_start 指向的 user_stack 数组区，然后使用本程序
30 # 后面定义的新中断描述符表和全局段描述表。新全局段描述表中初始内容与 setup.s 中的基本
31 # 一样，仅段限长从 8MB 修改成了 16MB。stack_start 定义在 kernel/sched.c, 69 行。它是指向
32 # user_stack 数组末端的一个长指针。第 23 行设置这里使用的栈，姑且称为系统栈。但在移动到
33 # 任务 0 执行（init/main.c 中 137 行）以后该栈就被用作任务 0 和任务 1 共同使用的用户栈了。
34
35 17 startup_32:                                # 18-22 行设置各个数据段寄存器。
36 18     movl $0x10, %eax                        # 对于 GNU 汇编，每个直接操作数要以 '$' 开始，否则表示地址。
37                                           # 每个寄存器名都要以 '%' 开头，eax 表示是 32 位的 ax 寄存器。

```

```

19      mov %ax,%ds
20      mov %ax,%es
21      mov %ax,%fs
22      mov %ax,%gs
23      lss _stack_start,%esp      # 表示_stack_start→ss:esp, 设置系统堆栈。
                                   # stack_start 定义在 kernel/sched.c, 69 行。
24      call setup_idt             # 调用设置中断描述符表子程序。
25      call setup_gdt             # 调用设置全局描述符表子程序。
26      movl $0x10,%eax           # reload all the segment registers
27      mov %ax,%ds               # after changing gdt. CS was already
28      mov %ax,%es               # reloaded in 'setup_gdt'
29      mov %ax,%fs               # 因为修改了 gdt, 所以需要重新装载所有的段寄存器。
30      mov %ax,%gs               # CS 代码段寄存器已经在 setup_gdt 中重新加载过了。

```

# 由于段描述符中的段限长从 setup.s 中的 8MB 改成了本程序设置的 16MB (见 setup.s 行 208-216 和本程序后面的 235-236 行), 因此这里再次对所有段寄存器执行加载操作是必须的。另外, 通过使用 bochs 跟踪观察, 如果不对 CS 再次执行加载, 那么在执行到 26 行时 CS 代码段不可见部分中的限长还是 8MB。这样看来应该重新加载 CS。但是由于 setup.s 中的内核代码段描述符与本程序中重新设置的代码段描述符除了段限长以外其余部分完全一样, 8MB 的限长在内核初始化阶段不会有问题, 而且在以后内核执行过程中段间跳转时会重新加载 CS。因此这里没有加载它并没有让程序出错。

# 针对该问题, 目前内核中就在第 25 行之后添加了一条长跳转指令: 'ljmp \$(\_\_KERNEL\_CS), \$1f', 跳转到第 26 行来确保 CS 确实又被重新加载。

```

31      lss _stack_start,%esp

```

# 32-36 行用于测试 A20 地址线是否已经开启。采用的方法是向内存地址 0x000000 处写入任意一个数值, 然后看内存地址 0x100000 (1M) 处是否也是这个数值。如果一直相同的话, 就一直比较下去, 也即死循环、死机。表示地址 A20 线没有选通, 结果内核就不能使用 1MB 以上内存。

#

# 33 行上的 '1:' 是一个局部符号构成的标号。标号由符号后跟一个冒号组成。此时该符号表示活动位置计数 (Active location counter) 的当前值, 并可以作为指令的操作数。局部符号用于帮助编译器和编程人员临时使用一些名称。共有 10 个局部符号名, 可在整个程序中重复使用。这些符号名使用名称 '0'、'1'、...、'9' 来引用。为了定义一个局部符号, 需把标号写成 'N:' 形式 (其中 N 表示一个数字)。为了引用先前最近定义的这个符号, 需要写成 'Nb', 其中 N 是定义标号时使用的数字。为了引用一个局部标号的下一个定义, 需要写成 'Nf', 这里 N 是 10 个前向引用之一。上面 'b' 表示 “向后 (backwards)”, 'f' 表示 “向前 (forwards)”。在汇编程序的某一处, 我们最大可以向后/向前引用 10 个标号 (最远第 10 个)。

```

32      xorl %eax,%eax
33 1:    incl %eax                  # check that A20 really IS enabled
34      movl %eax,0x000000        # loop forever if it isn't
35      cmpl %eax,0x100000
36      je 1b                     # '1b' 表示向后(backward)跳转到标号 1 去 (33 行)。
                                   # 若是 '5f' 则表示向前(forward)跳转到标号 5 去。

```

```

37 /*
38  * NOTE! 486 should set bit 16, to check for write-protect in supervisor
39  * mode. Then it would be unnecessary with the "verify_area()" -calls.
40  * 486 users probably want to set the NE (#5) bit also, so as to use
41  * int 16 for math errors.
42  */
/*
* 注意! 在下面这段程序中, 486 应该将位 16 置位, 以检查在超级用户模式下的写保护,

```



```

* 此后 "verify_area()" 调用就不需要了。486 的用户通常也会想将 NE(#5)置位，以便
* 对数学协处理器的出错使用 int 16。
*/
# 上面原注释中提到的 486 CPU 中 CR0 控制寄存器的位 16 是写保护标志 WP (Write-Protect)，
# 用于禁止超级用户级的程序向一般用户只读页面中进行写操作。该标志主要用于操作系统在创建
# 新进程时实现写时复制 (copy-on-write) 方法。
# 下面这段程序 (43-65) 用于检查数学协处理器芯片是否存在。方法是修改控制寄存器 CR0，在
# 假设存在协处理器的情况下执行一个协处理器指令，如果出错的话则说明协处理器芯片不存在，
# 需要设置 CR0 中的协处理器仿真位 EM (位 2)，并复位协处理器存在标志 MP (位 1)。

43      movl %cr0,%eax          # check math chip
44      andl $0x80000011,%eax   # Save PG,PE,ET
45 /* "orl $0x10020,%eax" here for 486 might be good */
46      orl $2,%eax            # set MP
47      movl %eax,%cr0
48      call check_x87
49      jmp after_page_tables   # 跳转到 135 行。
50
51 /*
52 * We depend on ET to be correct. This checks for 287/387.
53 */
/*
* 我们依赖于 ET 标志的正确性来检测 287/387 存在与否。
*/
# 下面 fninit 和 fstsw 是数学协处理器 (80287/80387) 的指令。
# finit 向协处理器发出初始化命令，它会把协处理器置于一个未受以前操作影响的已知状态，设置
# 其控制字为默认值、清除状态字和所有浮点栈式寄存器。非等待形式的这条指令 (fninit) 还会让
# 协处理器终止执行当前正在执行的任何先前的算术操作。fstsw 指令取协处理器的状态字。如果系
# 统中存在协处理器的话，那么在执行了 fninit 指令后其状态字低字节肯定为 0。

54 check_x87:
55      fninit                  # 向协处理器发出初始化命令。
56      fstsw %ax               # 取协处理器状态字到 ax 寄存器中。
57      cmpb $0,%al             # 初始化后状态字应该为 0，否则说明协处理器不存在。
58      je 1f                   /* no coprocessor: have to set bits */
59      movl %cr0,%eax          # 如果存在则向前跳转到标号 1 处，否则改写 cr0。
60      xorl $6,%eax            /* reset MP, set EM */
61      movl %eax,%cr0
62      ret

# 下面是一汇编语言指示符。其含义是指存储边界对齐调整。"2"表示把随后的代码或数据的偏移位置
# 调整到地址值最后 2 比特位为零的位置 (2^2)，即按 4 字节方式对齐内存地址。不过现在 GNU as
# 直接时写出对齐的值而非 2 的次方值了。使用该指示符的目的是为了提高 32 位 CPU 访问内存中代码
# 或数据的速度和效率。参见程序后的详细说明。
# 下面的两个字节值是 80287 协处理器指令 fsetpm 的机器码。其作用是把 80287 设置为保护模式。
# 80387 无需该指令，并且将会把该指令看作是空操作。

63 .align 2
64 1:      .byte 0xDB,0xE4      /* fsetpm for 287, ignored by 387 */ # 287 协处理器码。
65      ret
66
67 /*
68 * setup_idt

```

```

69  *
70  *  sets up a idt with 256 entries pointing to
71  *  ignore_int, interrupt gates. It then loads
72  *  idt. Everything that wants to install itself
73  *  in the idt-table may do so themselves. Interrupts
74  *  are enabled elsewhere, when we can be relatively
75  *  sure everything is ok. This routine will be over-
76  *  written by the page tables.
77  */
/*
 * 下面这段是设置中断描述符表子程序 setup_idt
 *
 * 将中断描述符表 idt 设置成具有 256 个项，并都指向 ignore_int 中断门。然后加载中断
 * 描述符表寄存器(用 lidt 指令)。真正实用的中断门以后再安装。当我们在其他地方认为一切
 * 都正常时再开启中断。该子程序将会被页表覆盖掉。
 */
# 中断描述符表中的项虽然也是 8 字节组成，但其格式与全局表中的不同，被称为门描述符
# (Gate Descriptor)。它的 0-1,6-7 字节是偏移量，2-3 字节是选择符，4-5 字节是一些标志。
# 这段代码首先在 edx、eax 中组合设置出 8 字节默认的中断描述符值，然后在 idt 表每一项中
# 都放置该描述符，共 256 项。eax 含有描述符低 4 字节，edx 含有高 4 字节。内核在随后的初始
# 化过程中会替换安装那些真正实用的中断描述符项。

78 setup_idt:
79     lea ignore_int,%edx      # 将 ignore_int 的有效地址（偏移值）值→edx 寄存器
80     movl $0x00080000,%eax   # 将选择符 0x0008 置入 eax 的高 16 位中。
81     movw %dx,%ax            /* selector = 0x0008 = cs */
                                # 偏移值的低 16 位置入 eax 的低 16 位中。此时 eax 含有
                                # 门描述符低 4 字节的值。
82     movw $0x8E00,%dx        /* interrupt gate - dpl=0, present */
83                                # 此时 edx 含有门描述符高 4 字节的值。
84     lea _idt,%edi           # _idt 是中断描述符表的地址。
85     mov $256,%ecx
86 rp_sidt:
87     movl %eax, (%edi)        # 将哑中断门描述符存入表中。
88     movl %edx, 4(%edi)       # eax 内容放到 edi+4 所指内存位置处。
89     addl $8,%edi             # edi 指向表中下一项。
90     dec %ecx
91     jne rp_sidt
92     lidt idt_descr           # 加载中断描述符表寄存器值。
93     ret
94
95 /*
96 *  setup_gdt
97 *
98 *  This routines sets up a new gdt and loads it.
99 *  Only two entries are currently built, the same
100 *  ones that were built in init.s. The routine
101 *  is VERY complicated at two whole lines, so this
102 *  rather long comment is certainly needed :-).
103 *  This routine will beoverwritten by the page tables.
104 */
/*
 * 设置全局描述符表项 setup_gdt

```

```

* 这个子程序设置一个新的全局描述符表 gdt，并加载。此时仅创建了两个表项，与前
* 面的一样。该子程序只有两行，“非常的”复杂，所以当然需要这么长的注释了☺。
* 该子程序将被页表覆盖掉。
*/
105 setup_gdt:
106     lgdt gdt_descr          # 加载全局描述符表寄存器(内容已设置好，见 234-238 行)。
107     ret
108
109 /*
110  * I put the kernel page tables right after the page directory,
111  * using 4 of them to span 16 Mb of physical memory. People with
112  * more than 16MB will have to expand this.
113  */
/* Linux 将内核的内存页表直接放在页目录之后，使用了 4 个表来寻址 16 MB 的物理内存。
* 如果你有多于 16 Mb 的内存，就需要在这里进行扩充修改。
*/
# 每个页表长为 4 Kb 字节（1 页内存页面），而每个页表项需要 4 个字节，因此一个页表共可以存放
# 1024 个表项。如果一个页表项寻址 4 KB 的地址空间，则一个页表就可以寻址 4 MB 的物理内存。
# 页表项的格式为：项的前 0-11 位存放一些标志，例如是否在内存中(P 位 0)、读写许可(R/W 位 1)、
# 普通用户还是超级用户使用(U/S 位 2)、是否修改过(是否脏了)(D 位 6)等；表项的位 12-31 是
# 页框地址，用于指出一页内存的物理起始地址。

114 .org 0x1000          # 从偏移 0x1000 处开始是第 1 个页表（偏移 0 开始处将存放页表目录）。
115 pg0:
116
117 .org 0x2000
118 pg1:
119
120 .org 0x3000
121 pg2:
122
123 .org 0x4000
124 pg3:
125
126 .org 0x5000          # 定义下面的内存数据块从偏移 0x5000 处开始。
127 /*
128  * tmp_floppy_area is used by the floppy-driver when DMA cannot
129  * reach to a buffer-block. It needs to be aligned, so that it isn't
130  * on a 64kB border.
131  */
/* 当 DMA（直接存储器访问）不能访问缓冲块时，下面的 tmp_floppy_area 内存块
* 就可供软盘驱动程序使用。其地址需要对齐调整，这样就不会跨越 64KB 边界。
*/
132 _tmp_floppy_area:
133     .fill 1024, 1, 0      # 共保留 1024 项，每项 1 字节，填充数值 0。
134
# 下面这几个入栈操作用于为跳转到 init/main.c 中的 main() 函数作准备工作。第 139 行上
# 的指令在栈中压入了返回地址，而第 140 行则压入了 main() 函数代码的地址。当 head.s
# 最后在第 218 行执行 ret 指令时就会弹出 main() 的地址，并把控制权转移到 init/main.c
# 程序中。参见第 3 章中有关 C 函数调用机制的说明。
# 前面 3 个入栈 0 值应该分别表示 envp、argv 指针和 argc 的值，但 main() 没有用到。
# 139 行的入栈操作是模拟调用 main.c 程序时首先将返回地址入栈的操作，所以如果
# main.c 程序真的退出时，就会返回到这里的标号 L6 处继续执行下去，也即死循环。

```

```

# 140 行将 main.c 的地址压入堆栈，这样，在设置分页处理（setup_paging）结束后
# 执行‘ret’ 返回指令时就会将 main.c 程序的地址弹出堆栈，并去执行 main.c 程序了。
# 有关 C 函数调用机制请参见程序后的说明。
135 after_page_tables:
136     pushl $0                # These are the parameters to main :-)
137     pushl $0                # 这些是调用 main 程序的参数（指 init/main.c）。
138     pushl $0                # 其中的‘$’ 符号表示这是一个立即操作数。
139     pushl $L6               # return address for main, if it decides to.
140     pushl $_main            # ‘_main’ 是编译程序对 main 的内部表示方法。
141     jmp setup_paging        # 跳转至第 198 行。
142 L6:
143     jmp L6                  # main should never return here, but
144                             # just in case, we know what happens.
                             # main 程序绝对不应该返回到这里。不过为了以防万一，
                             # 所以添加了该语句。这样我们就知道发生什么问题了。

145
146 /* This is the default interrupt "handler" :-) */
    /* 下面是默认的中断“向量句柄” ☺ */
147 int_msg:
148     .asciz "Unknown interrupt\n\r"    # 定义字符串“未知中断(回车换行)”。
149 .align 2                            # 按 4 字节方式对齐内存地址。
150 ignore_int:
151     pushl %eax
152     pushl %ecx
153     pushl %edx
154     push %ds                  # 这里请注意！！ds, es, fs, gs 等虽然是 16 位的寄存器，但入栈后
                             # 仍然会以 32 位的形式入栈，也即需要占用 4 个字节的堆栈空间。

155     push %es
156     push %fs
157     movl $0x10, %eax         # 置段选择符（使 ds, es, fs 指向 gdt 表中的数据段）。
158     mov %ax, %ds
159     mov %ax, %es
160     mov %ax, %fs
161     pushl $int_msg           # 把调用 printk 函数的参数指针（地址）入栈。注意！若符号 int_msg
                             # 前不加‘$’，则表示把 int_msg 符号处的长字（‘Unkn’）入栈 ☺。
162     call _printk             # 该函数在 /kernel/printk.c 中。
                             # ‘_printk’ 是 printk 编译后模块中的内部表示法。

163     popl %eax
164     pop %fs
165     pop %es
166     pop %ds
167     popl %edx
168     popl %ecx
169     popl %eax
170     iret                    # 中断返回（把中断调用时压入栈的 CPU 标志寄存器（32 位）值也弹出）。
171
172
173 /*
174  * Setup_paging
175  *
176  * This routine sets up paging by setting the page bit
177  * in cr0. The page tables are set up, identity-mapping
178  * the first 16MB. The pager assumes that no illegal

```

```

179 * addresses are produced (ie >4Mb on a 4Mb machine).
180 *
181 * NOTE! Although all physical memory should be identity
182 * mapped by this routine, only the kernel page functions
183 * use the >1Mb addresses directly. All "normal" functions
184 * use just the lower 1Mb, or the local data space, which
185 * will be mapped to some other place - mm keeps track of
186 * that.
187 *
188 * For those with more memory than 16 Mb - tough luck. I've
189 * not got it, why should you :-) The source is here. Change
190 * it. (Seriously - it shouldn't be too difficult. Mostly
191 * change some constants etc. I left it at 16Mb, as my machine
192 * even cannot be extended past that (ok, but it was cheap :-))
193 * I've tried to show which constants to change by having
194 * some kind of marker at them (search for "16Mb"), but I
195 * won't guarantee that's all :-( )
196 */
/*
* 这个子程序通过设置控制寄存器 cr0 的标志 (PG 位 31) 来启动对内存的分页处理功能,
* 并设置各个页表项的内容, 以恒等映射前 16 MB 的物理内存。分页器假定不会产生非法的
* 地址映射 (也即在只有 4Mb 的机器上设置出大于 4Mb 的内存地址)。
*
* 注意! 尽管所有的物理地址都应该由这个子程序进行恒等映射, 但只有内核页面管理函数能
* 直接使用>1Mb 的地址。所有“普通”函数仅使用低于 1Mb 的地址空间, 或者是使用局部数据
* 空间, 该地址空间将被映射到其他一些地方去 -- mm (内存管理程序) 会管理这些事的。
*
* 对于那些有多于 16Mb 内存的家伙 - 真是太幸运了, 我还没有, 为什么你会有☺。代码就在
* 这里, 对它进行修改吧。(实际上, 这并不太困难的。通常只需修改一些常数等。我把它设置
* 为 16Mb, 因为我的机器再怎么扩充甚至不能超过这个界限 (当然, 我的机器是很便宜的☺)。
* 我已经通过设置某类标志来给出需要改动的地方 (搜索“16Mb”), 但我不能保证作这些
* 改动就行了☺)。
*/
# 上面英文注释第 2 段的含义是指在机器物理内存中大于 1MB 的内存空间主要被用于主内存区。
# 主内存区空间由 mm 模块管理。它涉及到页面映射操作。内核中所有其他函数就是这里指的一般
# (普通) 函数。若要使用主内存区的页面, 就需要使用 get_free_page() 等函数获取。因为主内
# 存区中内存页面是共享资源, 必须有程序进行统一管理以避免资源争用和竞争。
#
# 在内存物理地址 0x0 处开始存放 1 页页目录表和 4 页页表。页目录表是系统所有进程公用的, 而
# 这里的 4 页页表则属于内核专用, 它们一一映射线性地址起始 16MB 空间范围到物理内存上。对于
# 新的进程, 系统会在主内存区为其申请页面存放页表。另外, 1 页内存长度是 4096 字节。

197 .align 2                                # 按 4 字节方式对齐内存地址边界。
198 setup_paging:                          # 首先对 5 页内存 (1 页目录 + 4 页页表) 清零。
199     movl $1024*5,%ecx                  /* 5 pages - pg_dir+4 page tables */
200     xorl %eax,%eax
201     xorl %edi,%edi                      /* pg_dir is at 0x000 */
                                           # 页目录从 0x000 地址开始。
202     cld;rep;stosl                       # eax 内容存到 es:edi 所指内存位置处, 且 edi 增 4。

# 下面 4 句设置页目录表中的项, 因为我们 (内核) 共有 4 个页表所以只需设置 4 项。
# 页目录项的结构与页表中项的结构一样, 4 个字节为 1 项。参见上面 113 行下的说明。
# 例如“$pg0+7”表示: 0x00001007, 是页目录表中的第 1 项。

```

```

# 则第 1 个页表所在的地址 = 0x00001007 & 0xfffff000 = 0x1000;
# 第 1 个页表的属性标志 = 0x00001007 & 0x00000fff = 0x07, 表示该页存在、用户可读写。
203      movl $pg0+7, _pg_dir      /* set present bit/user r/w */
204      movl $pg1+7, _pg_dir+4    /* ----- " " ----- */
205      movl $pg2+7, _pg_dir+8    /* ----- " " ----- */
206      movl $pg3+7, _pg_dir+12   /* ----- " " ----- */

# 下面 6 行填写 4 个页表中所有项的内容, 共有: 4(页表)*1024(项/页表)=4096 项(0 - 0xfff),
# 也即能映射物理内存 4096*4Kb = 16Mb。
# 每项的内容是: 当前项所映射的物理内存地址 + 该页的标志(这里均为 7)。
# 使用的方法是从最后一个页表的最后一项开始按倒退顺序填写。一个页表的最后一项在页表中的
# 位置是 1023*4 = 4092。因此最后一页的最后一项的位置就是$pg3+4092。

207      movl $pg3+4092, %edi      # edi→最后一页的最后一项。
208      movl $0xffff007, %eax     /* 16Mb - 4096 + 7 (r/w user, p) */
                                     # 最后 1 项对应物理内存页面的地址是 0xffff000,
                                     # 加上属性标志 7, 即为 0xffff007。
209      std                      # 方向位置位, edi 值递减(4 字节)。
210 1:      stosl                 /* fill pages backwards - more efficient :- ) */
211      subl $0x1000, %eax       # 每填写好一项, 物理地址值减 0x1000。
212      jge 1b                  # 如果小于 0 则说明全添写好了。
# 设置页目录表基址寄存器 cr3 的值, 指向页目录表。cr3 中保存的是页目录表的物理地址。
213      xorl %eax, %eax          /* pg_dir is at 0x0000 */ # 页目录表在 0x0000 处。
214      movl %eax, %cr3         /* cr3 - page directory start */
# 设置启动使用分页处理 (cr0 的 PG 标志, 位 31)
215      movl %cr0, %eax
216      orl $0x80000000, %eax     # 添上 PG 标志。
217      movl %eax, %cr0          /* set paging (PG) bit */
218      ret                     /* this also flushes prefetch-queue */

# 在改变分页处理标志后要求使用转移指令刷新预取指令队列, 这里用的是返回指令 ret。
# 该返回指令的另一个作用是将 140 行压入堆栈中的 main 程序的地址弹出, 并跳转到/init/main.c
# 程序去运行。本程序到此就真正结束了。

219
220 .align 2                      # 按 4 字节方式对齐内存地址边界。
221 .word 0                       # 这里先空出 2 字节, 这样 224 行上的长字是 4 字节对齐的。

! 下面是加载中断描述符表寄存器 idtr 的指令 lidt 要求的 6 字节操作数。前 2 字节是 idt 表的限长,
! 后 4 字节是 idt 表在线性地址空间中的 32 位基地址。
222 idt_desc:
223      .word 256*8-1             # idt contains 256 entries # 共 256 项, 限长=长度 - 1。
224      .long _idt
225 .align 2
226 .word 0

! 下面加载全局描述符表寄存器 gdt 的指令 lgdt 要求的 6 字节操作数。前 2 字节是 gdt 表的限长,
! 后 4 字节是 gdt 表的线性基地址。这里全局表长度设置为 2KB 字节(0x7ff 即可), 因为每 8 字节
! 组成一个描述符项, 所以表中共可有 256 项。符号_gdt 是全局表在本程序中的偏移位置, 见 234 行。
227 gdt_desc:
228      .word 256*8-1             # so does gdt (not that that's any # 注: not → note
229      .long _gdt                # magic number, but it works for me :^)
230

```



```

231      .align 3                                # 按 8 (2^3) 字节方式对齐内存地址边界。
232 _idt:  .fill 256,8,0                        # idt is uninitialized # 256 项, 每项 8 字节, 填 0。
233
# 全局表。前 4 项分别是空项 (不用)、代码段描述符、数据段描述符、系统调用段描述符, 其中
# 系统调用段描述符并没有派用处, Linus 当时可能曾想把系统调用代码专门放在这个独立的段中。
# 后面还预留了 252 项的空间, 用于放置所创建任务的局部描述符 (LDT) 和对应的任务状态段 TSS
# 的描述符。
# (0-nul, 1-cs, 2-ds, 3-syscall, 4-TSS0, 5-LDT0, 6-TSS1, 7-LDT1, 8-TSS2 etc...)
234 _gdt:  .quad 0x0000000000000000             /* NULL descriptor */
235        .quad 0x00c09a0000000fff             /* 16Mb */          # 0x08, 内核代码段最大长度 16MB。
236        .quad 0x00c0920000000fff             /* 16Mb */          # 0x10, 内核数据段最大长度 16MB。
237        .quad 0x0000000000000000             /* TEMPORARY - don't use */
238        .fill 252,8,0                        /* space for LDT's and TSS's etc */ # 预留空间。

```

### 6.4.3 其他信息

#### 6.4.3.1 程序执行结束后的内存映像

head.s 程序执行结束后, 已经正式完成了内存页目录和页表的设置, 并重新设置了内核实际使用的中断描述符表 idt 和全局描述符表 gdt。另外还为软盘驱动程序开辟了 1KB 字节的缓冲区。此时 system 模块在内存中的详细映像见图 6-11 所示。

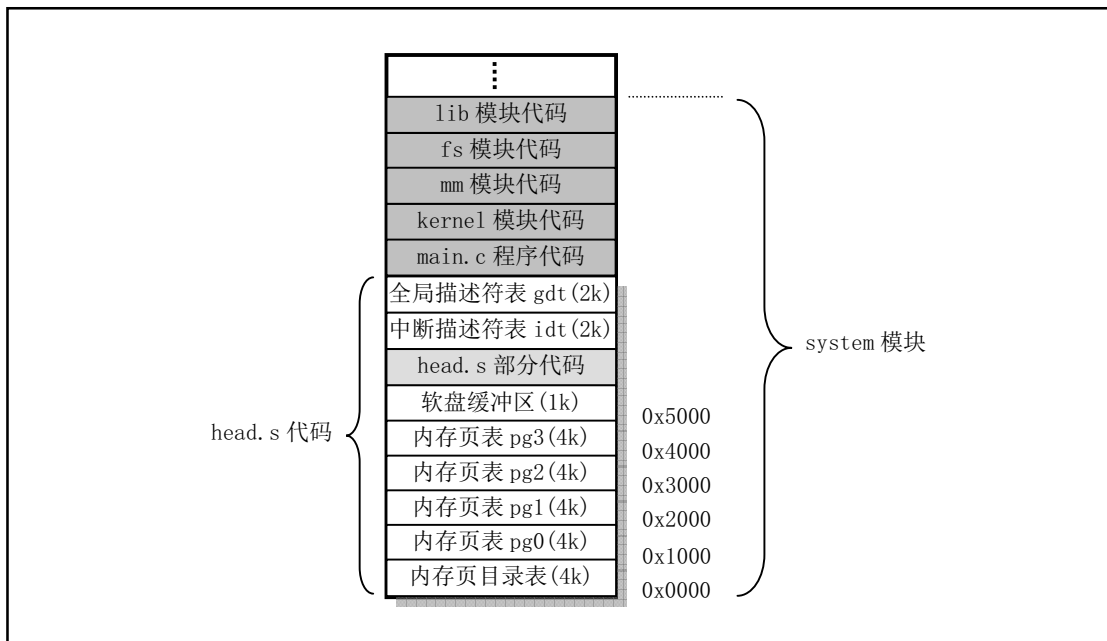


图 6-11 system 模块在内存中的映像示意图

#### 6.4.3.2 Intel 32 位保护运行机制

理解这段程序的关键是真正了解 Intel 386 32 位保护模式的运行机制, 也是继续阅读以下其余程序所必需的。为了与 8086 CPU 兼容, 80x86 的保护模式被处理的较为复杂。当 CPU 运行在保护模式下时, 它就将实模式下的段地址当作保护模式下段描述符的指针使用, 此时段寄存器中存放的是一个描述符在描述符表中的偏移地址值。而当前描述符表的基地址则保存在描述符表寄存器中, 如全局描述符表寄存器 gdt、中断门描述符表寄存器 idtr, 加载这些表寄存器须使用专用指令 lgdt 或 lidt。

CPU 在实模式运行方式时, 段寄存器用来放置一个内存段地址 (例如 0x9000), 而此时在该段内可以寻址 64KB 的内存。但当进入保护模式运行方式时, 此时段寄存器中放置的并不是内存中的某个地址



值，而是指定描述符表中某个描述符项相对于该描述符表基址的一个偏移量。在这个 8 字节的描述符中含有该段线性地址的‘段’基址和段的长度，以及其他一些描述该段特征的比特位。因此此时所寻址的内存位置是这个段基址加上当前执行代码指针 `eip` 的值。当然，此时所寻址的实际物理内存地址，还需要经过内存页面处理管理机制进行变换后才能得到。简而言之，32 位保护模式下的内存寻址需要拐个弯，经过描述符表中的描述符和内存页管理来确定。

针对不同的使用方面，描述符表分为三种：全局描述符表（GDT）、中断描述符表（IDT）和局部描述符表（LDT）。当 CPU 运行在保护模式下，某一时刻 GDT 和 IDT 分别只能有一个，分别由寄存器 `GDTR` 和 `IDTR` 指定它们的表基址。局部表可以有 0 个或最多 8191 个，这由 GDT 表中未用项数和所设计的具体系统确定。在某一个时刻，当前 LDT 表的基址由 `LDTR` 寄存器的内容指定，并且 `LDTR` 的内容使用 GDT 中某个描述符来加载，即 LDT 也是由 GDT 中的描述符来指定。但是在某一时刻同样也只有其中的一个被认为是活动的。一般对于每个任务（进程）使用一个 LDT。在运行时，程序可以使用 GDT 中的描述符以及当前任务的 LDT 中的描述符。对于 Linux 0.11 内核来说同时可以有 64 个任务在执行，因此 GDT 表中最多有 64 个 LDT 表的描述符项存在。

中断描述符表 IDT 的结构与 GDT 类似，在 Linux 内核中它正好位于 GDT 表的前面。共含有 256 项 8 字节的描述符。但每个描述符项的格式与 GDT 的不同，其中存放着相应中断过程的偏移值（0-1，6-7 字节）、所处段的选择符值（2-3 字节）和一些标志（4-5 字节）。

图 6-12 是 Linux 内核中所使用的描述符表在内存中的示意图。图中，每个任务在 GDT 中占有两个描述符项。GDT 表中的 `LDT0` 描述符项是第一个任务（进程）的局部描述符表的描述符，`TSS0` 是第一个任务的任务状态段（TSS）的描述符。每个 LDT 中含有三个描述符，其中第一个不用，第二个是任务代码段的描述符，第三个是任务数据段和堆栈段的描述符。当 `DS` 段寄存器中是第一个任务的数据段选择符时，`DS:ESI` 即指向该任务数据段中的某个数据。

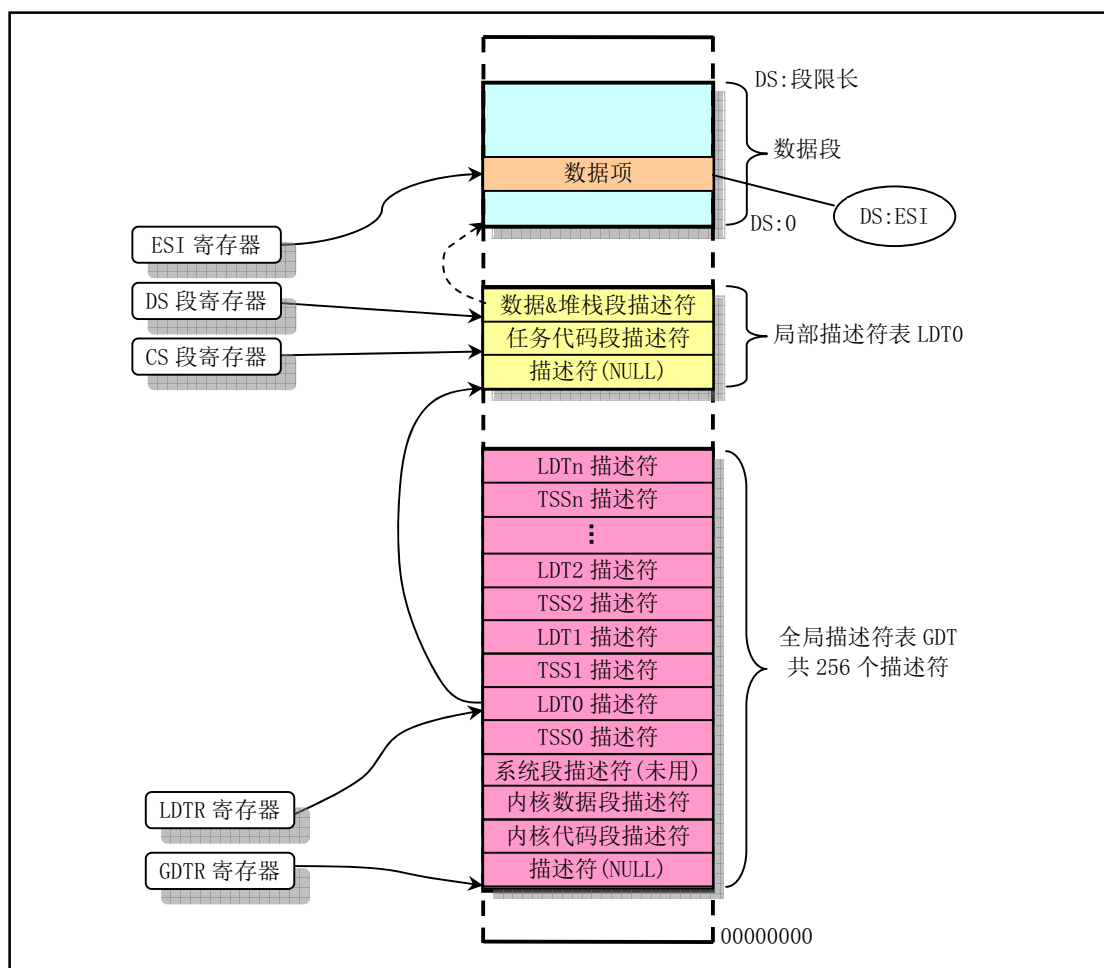


图 6-12 Linux 内核使用描述符表的示意图。

### 6.4.3.3 前导符（伪指令）align

在第 3 章介绍汇编器时我们已经对 `align` 伪指令进行了说明。这里我们再总结一下。使用伪指令 `align` 的作用是在编译时指示编译器填充位置计数器（类似指令计数器）到一个指定的内存边界处。目的是为了提 CPU 访问内存中代码或数据的速度和效率。其完整格式为：

```
.align val1, val2, val3
```

其中第 1 个参数值 `val1` 是所需要的对齐值；第 2 个是填充字节指定的值。填充值可以省略。若省略则编译器使用 0 值填充。第 3 个可选参数值 `val3` 用来指明最大用于填充或跳过的直接数。如果进行边界对齐会超过 `val3` 指定的最大字节数，那么就根本不进行对齐操作。如果需要省略第 2 个参数 `val2` 但还是需要使用第 3 个参数 `val3`，那么只需要放置两个逗号即可。

对于现在使用 ELF 目标格式的 Intel 80X86 CPU，第 1 个参数 `val1` 是需要对齐的字节数。例如，`'align 8'` 表示调整位置计数器直到它指在 8 的倍数边界上。如果已经在 8 的倍数边界上，那么编译器就不用改变了。但对于我们这里使用 `a.out` 目标格式的系统来说，第 1 个参数 `val1` 是指定位 0 比特的个数，即 2 的次方数 ( $2^{\text{val1}}$ )。例如前面程序 `head.s` 中的 `'align 3'` 就表示位置计数器需要位于 8 的倍数边界上。同样，如果已经在 8 的倍数边界上，那么该伪指令什么也不做。GNU `as` (`gas`) 对这两个目标格式的不同处理方法是由于 `gas` 为了模仿各种体系结构系统上自带的汇编器的行为而形成的。

## 6.5 本章小结

引导加载程序 `bootsect.s` 将 `setup.s` 代码和 `system` 模块加载到内存中，并且分别把自己和 `setup.s` 代码移动到物理内存 `0x90000` 和 `0x90200` 处后，就把执行权交给了 `setup` 程序。其中 `system` 模块的首部包含有 `head.s` 代码。

`setup` 程序的主要作用是利用 ROM BIOS 的中断程序获取机器的一些基本参数，并保存在 `0x90000` 开始的内存块中，供后面程序使用。同时把 `system` 模块往下移动到物理地址 `0x00000` 开始处，这样，`system` 中的 `head.s` 代码就处在 `0x00000` 开始处了。然后加载描述符表基地址到描述符表寄存器中，为进行 32 位保护模式下的运行作好准备。接下来对中断控制硬件进行重新设置，最后通过设置机器控制寄存器 `CR0` 并跳转到 `system` 模块的 `head.s` 代码开始处，使 CPU 进入 32 位保护模式下运行。

`Head.s` 代码的主要作用是初步初始化中断描述符表中的 256 项门描述符，检查 `A20` 地址线是否已经打开，测试系统是否含有数学协处理器。然后初始化内存页目录表，为内存的分页管理作好准备工作。最后跳转到 `system` 模块中的初始化程序 `init/main.c` 中继续执行。

下一章的主要内容就是详细描述 `init/main.c` 程序的功能和作用。

## 第7章 初始化程序(init)

在内核源代码的 `init/` 目录中只有一个 `main.c` 文件。系统在执行完 `boot/` 目录中的 `head.s` 程序后就会将执行权交给 `main.c`。该程序虽然不长，但却包括了内核初始化的所有工作。因此在阅读该程序的代码时需要参照很多其他程序中的初始化部分。如果能完全理解这里调用的所有程序，那么看完这章内容后你应该对 Linux 内核有了大致的了解。

从这一章开始，我们将接触大量的 C 程序代码，因此读者最好具有一定的 C 语言知识。最好的一本参考书还是 Brian W. Kernighan 和 Dennis M. Ritchie 编著的《C 程序设计语言》。对该书第五章关于指针和数组的理解，可以说是弄懂 C 语言的关键。另外还需要 GNU gcc 手册在身边作为参考，因为在内核代码中很多地方使用了 gcc 的扩展特性。例如内联（`inline`）函数、内联（内嵌）汇编语句等。

在注释 C 语言程序时，为了与程序中原有的注释相区别，我们使用 `/*` 作为注释语句的开始。有关原有注释的翻译则采用与其一样的注释标志。对于程序中包含的头文件（`*.h`），仅作概要含义的解释，具体详细注释内容将在注释相应头文件的章节中给出。

### 7.1 main.c 程序

#### 7.1.1 功能描述

`main.c` 程序首先利用前面 `setup.s` 程序取得的系统参数设置系统的根文件设备号以及一些内存全局变量。这些内存变量指明了主内存的开始地址、系统所拥有的内存容量和作为高速缓冲区内内存的末端地址。如果还定义了虚拟盘（`RAMDISK`），则主内存将适当减少。整个内存的映像示意图见图 7-1 所示。

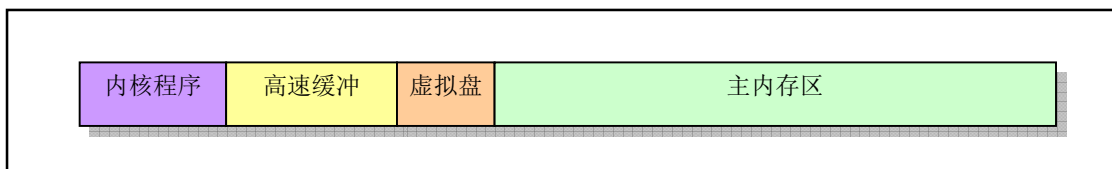


图 7-1 系统中内存功能划分示意图。

图中，高速缓冲部分还要扣除被显存和 ROM BIOS 占用的部分。高速缓冲区是用于磁盘等块设备临时存放数据的地方，以 1K（1024）字节为一个数据块单位。主内存区域的内存由内存管理模块 `mm` 通过分页机制进行管理分配，以 4K 字节为一个内存页单位。内核程序可以自由访问高速缓冲中的数据，但需要通过 `mm` 才能使用分配到的内存页面。

然后，内核进行所有方面的硬件初始化工作。包括陷阱门、块设备、字符设备和 `tty`，还包括人工设置第一个任务（`task 0`）。待所有初始化工作完成后程序就设置中断允许标志以开启中断，并切换到任务 0 中运行。在阅读这些初始化子程序时，最好是跟着被调用的程序深入进去看，如果实在看不下去了，就暂时先放一放，继续看下一个初始化调用。在有些理解之后再继续研究没有看完的地方。

在整个内核完成初始化后，内核将执行权切换到了用户模式（任务 0），也即 CPU 从 0 特权级切换到了第 3 特权级。此时 `main.c` 的主程序就工作在任务 0 中。然后系统第一次调用进程创建函数 `fork()`，

创建一个用于运行 `init()` 的子进程（通常被称为 `init` 进程）。系统整个初始化过程见图 7-2 所示。

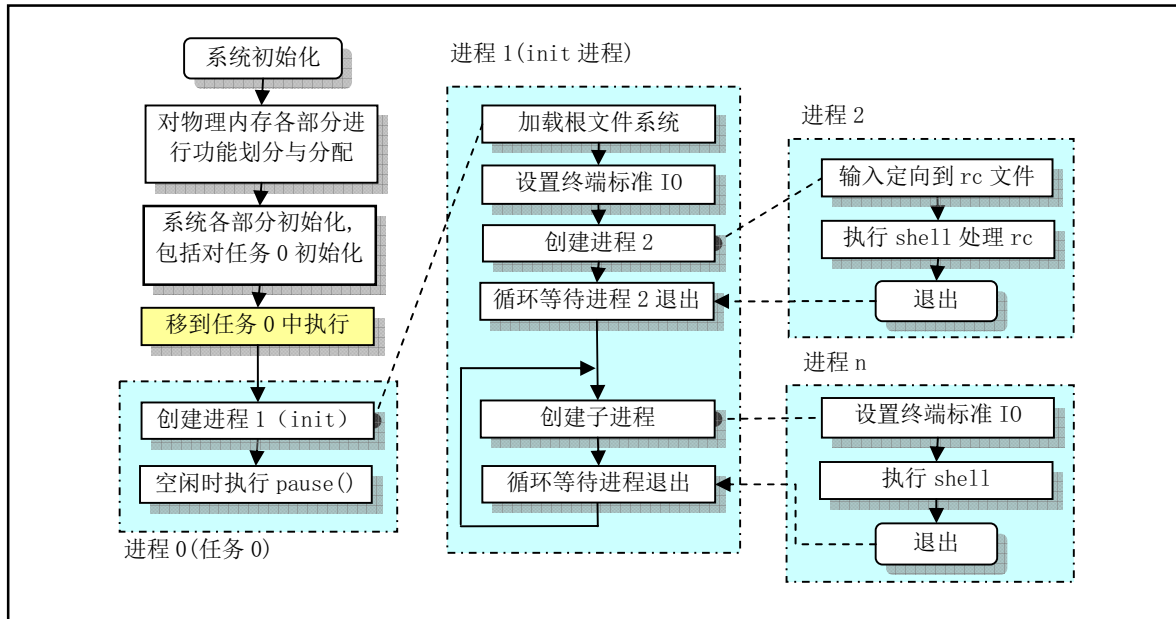


图 7-2 内核初始化程序流程示意图

`main.c` 程序首先确定如何分配使用系统物理内存，然后调用内核各部分的初始化函数分别对内存管理、中断处理、块设备和字符设备、进程管理以及硬盘和软盘硬件进行初始化处理。在完成了这些操作之后，系统各部分已经处于可运行状态。此后程序把自己“手工”移动到任务 0（进程 0）中运行，并使用 `fork()` 调用首次创建出进程 1（`init` 进程），并在其中调用 `init()` 函数。在该函数中程序将继续进行应用环境的初始化并执行 `shell` 登录程序。而原进程 0 则会在系统空闲时被调度执行，因此进程 0 通常也被称为 `idle` 进程。此时进程 0 仅执行 `pause()` 系统调用，并又会调用调度函数。

`init()` 函数的功能可分为 4 个部分：①安装根文件系统；②显示系统信息；③运行系统初始资源配置文件 `rc` 中的命令；④执行用户登录 `shell` 程序。

代码首先调用系统调用 `setup()`，用来收集硬盘设备分区表信息并安装根文件系统。在安装根文件系统之前，系统会先判断是否需要先建立虚拟盘。若编译内核时设置了虚拟盘的大小，并在前面内核初始化过程中已经开辟了一块内存用作虚拟盘，则内核就会首先尝试把根文件系统加载到内存的虚拟盘区中。

然后 `init()` 打开一个终端设备 `tty0`，并复制其文件描述符以产生标准输入 `stdin`、标准输出 `stdout` 和错误输出 `stderr` 设备。内核随后利用这些描述符在终端上显示一些系统信息，例如高速缓冲区中缓冲块总数、主内存区空闲内存总字节数等。

接着 `init()` 又新建了一个进程（进程 2），并在其中为建立用户交互使用环境而执行一些初始配置操作，即在用户可以使用 `shell` 命令行环境之前，内核调用 `/bin/sh` 程序运行了配置文件 `etc/rc` 中设置的命令。`rc` 文件的作用与 DOS 系统根目录上的 `AUTOEXEC.BAT` 文件类似。这段代码首先通过关闭文件描述符 0，并立刻打开文件 `/etc/rc`，从而把标准输入 `stdin` 定向到 `etc/rc` 文件上。这样，所有的标准输入数据都将从该文件中读取。然后内核以非交互形式执行 `/bin/sh`，从而实现执行 `/etc/rc` 文件中的命令。当该文件中的命令执行完毕后，`/bin/sh` 就会立刻退出。因此进程 2 也就随之结束。

`init()` 函数的最后一部份用于在新建进程中为用户建立一个新的会话，并运行用户登录 `shell` 程序 `/bin/sh`。在系统执行进程 2 中的程序时，父进程（`init` 进程）一直等待着它的结束。随着进程 2 的退出，父进程就进入到一个无限循环中。在该循环中，父进程会再次生成一个新进程，然后在该进程中创建一个新的会话，并以登录 `shell` 方式再次执行程序 `/bin/sh`，以创建用户交互 `shell` 环境。然后父进程继续等待

孩子进程。登录 shell 虽然与前面的非交互式 shell 是同一个程序/bin/sh,但是所使用的命令行参数(argv[])不同。登录 shell 的第 0 个命令行参数的第 1 个字符一定是一个减号'-'。这个特定的标志会在/bin/sh 执行时通知它这不是一次普通的运行,而是作为登录 shell 运行/bin/sh 的。从这时开始,用户就可以正常使用 Linux 命令行环境了,而父进程随之又进入等待状态。此后若用户在命令行上执行了 exit 或 logout 命令,那么在显示一条当前登录 shell 退出的信息后,系统就会在这个无限循环中再次重复以上创建登录 shell 进程的过程。

任务 1 中运行的 init()函数的后两部分实际上应该是独立的环境初始化程序 init 等的功能。参见程序列表后对这方面的说明。

由于创建新进程的过程是通过完全复制父进程代码段和数据段的方式实现,因此在首次使用 fork()创建新进程 init 时,为了确保新进程用户态栈中没有进程 0 的多余信息,要求进程 0 在创建首个新进程(进程 1)之前不要使用其用户态栈,即要求任务 0 不要调用函数。因此在 main.c 主程序移动到任务 0 执行后,任务 0 中的代码 fork()不能以函数形式进行调用。程序中实现的方法是采用如下所示的 gcc 函数内嵌(内联)形式来执行这个系统调用(参见程序第 23 行):

通过声明一个内联(inline)函数,可以让 gcc 把函数的代码集成到调用它的代码中。这会提高代码执行的速度,因为省去了函数调用的开销。另外,如果任何一个实际参数是一个常量,那么在编译时这些已知值就可能使得无需把内嵌函数的所有代码都包括进来而让代码也得到简化。参见第 3 章中的相关说明。

---

23 static inline [\\_syscall0](#)(int, [fork](#))

---

其中\_syscall0()是unistd.h 中的内嵌宏代码,它以嵌入汇编的形式调用 Linux 的系统调用中断 int 0x80。根据 include/unistd.h 文件第 133 行上的宏定义,我们把这个宏展开并替代进上面一行中就可以看出这条语句实际上是 int fork()创建进程系统调用,见如下所示。

---

```
// unistd.h 文件中 _syscall0() 的定义。即为不带参数的系统调用宏函数: type name(void)。
133 #define \_syscall0(type, name) \
134 type name(void) \
135 { \
136 long __res; \
137 __asm__ volatile ("int $0x80" \           // 调用系统中断 0x80。
138                  : "=a" (__res) \       // 返回值→eax(__res)。
139                  : "0" (__NR_##name)); \  // 输入为系统中断调用号 __NR_name。
140 if (__res >= 0) \                         // 如果返回值>=0, 则直接返回该值。
141     return (type) __res; \
142 errno = -__res; \                         // 否则置出错号, 并返回-1。
143 return -1; \
144 }
```

// 根据上面定义把\_syscall0(int, fork)展开代进第 23 行后我们可以得到如下语句:

```
static inline int fork(void)
{
    long __res;
    __asm__ volatile ("int $0x80" : "=a" (__res) : "0" (__NR_fork));
    if (__res >= 0)
        return (int) __res;
    errno = -__res;
    return -1;
}
```

---



---

}

---

gcc 会把上述“函数”体中的语句直接插入到调用 `fork()` 语句的代码处，因此执行 `fork()` 不会引起函数调用。另外，宏名称字符串“`syscall0`”中最后的 0 表示无参数，1 表示带 1 个参数。如果系统调用带有 1 个参数，那么就应该使用宏 `_syscall1()`。

虽然上面系统中断调用执行中断指令 INT 时还是避免不了使用堆栈，但是系统调用使用任务的内核态栈而非用户栈，并且每个任务都有自己独立的内核态栈，因此系统调用不会影响这里讨论的用户态栈。

另外，在创建新进程 `init`（即进程 1）的过程中，系统对其进行了一些特殊处理。进程 0 和进程 `init` 实际上同时使用着内核代码区内（小于 1MB 的物理内存）相同的代码和数据物理内存页面（640KB），只是执行的代码不在一处，因此实际上它们也同时使用着相同的用户堆栈区。在为新进程 `init` 复制其父进程（进程 0）的页目录和页表项时，进程 0 的 640KB 页表项属性没有改动过（仍然可读写），但是进程 1 的 640KB 对应的页表项却被设置成了只读。因此当进程 1 开始执行时，其对用户栈的出入栈操作将导致页面写保护异常，从而会使得内核的内存管理程序为进程 1 在主内存区中分配一内存页面，并把任务 0 栈中相应页面内容复制到此新页面上。从此时起，任务 1 的用户态栈开始有自己独立的内存页面。即从任务 1 执行过出/入栈操作后，任务 0 和任务 1 的用户栈才变成相互独立的栈。为了不出现冲突问题，就必须要求任务 0 在任务 1 执行栈操作之前禁止使用到用户堆栈区域，而让进程 `init` 能单独使用堆栈。因为在内核调度进程运行时次序是随机的，有可能在任务 0 创建了任务 1 后仍然先允许任务 0。因此任务 0 执行 `fork()` 操作后，随后的 `pause()` 函数也必须采用内嵌函数形式来实现，以避免任务 0 在任务 1 之前使用用户栈。

当系统中一个进程（例如 `init` 进程的子进程，进程 2）执行过 `execve()` 调用后，进程 2 的代码和数据区会位于系统的主内存区中，因此系统此后可以随时利用写时复制技术<sup>8</sup>（Copy on Write）来处理其他新进程的创建和执行。

对于 Linux 来说，所有任务都是在用户模式下运行的，包括很多系统应用程序，如 `shell` 程序、网络子系统程序等。内核源代码 `lib/` 目录下的库文件（除其中的 `string.c` 程序）就是专门为这里新创建的进程提供函数支持，内核代码本身并不使用这些库函数。

## 7.1.2 代码注释

程序 7-1 linux/init/main.c

---

```

1  /*
2  *  linux/init/main.c
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  // 定义宏 “__LIBRARY__” 是为了包括定义在 unistd.h 中的内嵌汇编代码等信息。
8  #define __LIBRARY__
9  // *.h 头文件所在的默认目录是 include/，则在代码中就不用明确指明其位置。如果不是 UNIX 的
10 // 标准头文件，则需要指明所在的目录，并用双引号括住。unistd.h 是标准符号常数与类型文件。
11 // 其中定义了各种符号常数和类型，并声明了各种函数。如果还定义了符号 __LIBRARY__，则还会
12 // 包含系统调用号和内嵌汇编代码 syscall0() 等。
13 #include <unistd.h>
14 #include <time.h>    // 时间类型头文件。其中最主要定义了 tm 结构和一些有关时间的函数原形。
15
16 /*
```

---

<sup>8</sup> 参见内存管理一章：写时复制机制。



```

12 * we need this inline - forking from kernel space will result
13 * in NO COPY ON WRITE (!!!), until an execve is executed. This
14 * is no problem, but for the stack. This is handled by not letting
15 * main() use the stack at all after fork(). Thus, no function
16 * calls - which means inline code for fork too, as otherwise we
17 * would use the stack upon exit from 'fork()'.
18 *
19 * Actually only pause and fork are needed inline, so that there
20 * won't be any messing with the stack from main(), but we define
21 * some others too.
22 */
/*
* 我们需要下面这些内嵌语句 - 从内核空间创建进程将导致没有写时复制(COPY ON WRITE)!!!
* 直到执行一个 execve 调用。这对堆栈可能带来问题。处理方法是在 fork() 调用后不让 main()
* 使用任何堆栈。因此就不能有函数调用 - 这意味着 fork 也要使用内嵌的代码, 否则我们在从
* fork() 退出时就要使用堆栈了。
*
* 实际上只有 pause 和 fork 需要使用内嵌方式, 以保证从 main() 中不会弄乱堆栈, 但是我们同
* 时还定义了一些其他函数。
*/
// Linux 在内核空间创建进程时不使用写时复制技术 (Copy on write)。main() 在移动到用户
// 模式 (到任务 0) 后执行内嵌方式的 fork() 和 pause(), 因此可保证不使用任务 0 的用户栈。
// 在执行 moveto_user_mode() 之后, 本程序 main() 就以任务 0 的身份在运行了。而任务 0 是所
// 有将创建子进程的父进程。当它创建一个子进程时 (init 进程), 由于任务 1 代码属于内核
// 空间, 因此没有使用写时复制功能。此时任务 0 的用户栈就是任务 1 的用户栈, 即它们共同
// 使用一个栈空间。因此希望在 main.c 运行在任务 0 的环境下时不要有对堆栈的任何操作, 以
// 免弄乱堆栈。而在再次执行 fork() 并执行过 execve() 函数后, 被加载程序已不属于内核空间,
// 因此可以使用写时复制技术了。参见上一章“Linux 内核使用内存的方法”一节内容。

// 下面 _syscall0() 是 unistd.h 中的内嵌宏代码。以嵌入汇编的形式调用 Linux 的系统调用中断
// 0x80。该中断是所有系统调用的入口。该语句实际上是 int fork() 创建进程系统调用。可展
// 开看之就会立刻明白。syscall0 名称中最后的 0 表示无参数, 1 表示 1 个参数。
// 参见 include/unistd.h, 133 行。
23 static inline _syscall0(int, fork)
// int pause() 系统调用: 暂停进程的执行, 直到收到一个信号。
24 static inline _syscall0(int, pause)
// int setup(void * BIOS) 系统调用, 仅用于 linux 初始化 (仅在这个程序中被调用)。
25 static inline _syscall1(int, setup, void *, BIOS)
// int sync() 系统调用: 更新文件系统。
26 static inline _syscall0(int, sync)
27
28 #include <linux/tty.h> // tty 头文件, 定义了有关 tty_io, 串行通信方面的参数、常数。
29 #include <linux/sched.h> // 调度程序头文件, 定义了任务结构 task_struct、第 1 个初始任务
// 的数据。还有一些以宏的形式定义的有关描述符参数设置和获取的
// 嵌入式汇编函数程序。
30 #include <linux/head.h> // head 头文件, 定义了段描述符的简单结构, 和几个选择符常量。
31 #include <asm/system.h> // 系统头文件。以宏的形式定义了许多有关设置或修改
// 描述符/中断门等的嵌入式汇编子程序。
32 #include <asm/io.h> // io 头文件。以宏的嵌入汇编程序形式定义对 io 端口操作的函数。
33
34 #include <stddef.h> // 标准定义头文件。定义了 NULL, offsetof(TYPE, MEMBER)。
35 #include <stdarg.h> // 标准参数头文件。以宏的形式定义变量参数列表。主要说明了一个
// 类型(va_list)和三个宏(va_start, va_arg 和 va_end), vsprintf、

```

```

// vprintf、vfprintf。
36 #include <unistd.h>
37 #include <fcntl.h> // 文件控制头文件。用于文件及其描述符的操作控制常数符号的定义。
38 #include <sys/types.h> // 类型头文件。定义了基本的系统数据类型。
39
40 #include <linux/fs.h> // 文件系统头文件。定义文件表结构（file, buffer_head, m_inode 等）。
41
42 static char printbuf[1024]; // 静态字符串数组，用作内核显示信息的缓存。
43
44 extern int vsprintf(); // 送格式化输出到一字符串中（vsprintf.c, 92 行）。
45 extern void init(void); // 函数原形，初始化（在 168 行）。
46 extern void blk_dev_init(void); // 块设备初始化子程序（blk_drv/ll_rw_blk.c, 157 行）
47 extern void chr_dev_init(void); // 字符设备初始化（chr_drv/tty_io.c, 347 行）
48 extern void hd_init(void); // 硬盘初始化程序（blk_drv/hd.c, 343 行）
49 extern void floppy_init(void); // 软驱初始化程序（blk_drv/floppy.c, 457 行）
50 extern void mem_init(long start, long end); // 内存管理初始化（mm/memory.c, 399 行）
51 extern long rd_init(long mem_start, int length); // 虚拟盘初始化（blk_drv/ramdisk.c, 52）
52 extern long kernel_mktime(struct tm * tm); // 计算系统开机启动时间（秒）。
53 extern long startup_time; // 内核启动时间（开机时间）（秒）。
54
55 /*
56  * This is set up by the setup-routine at boot-time
57  */
58 /*
59  * 以下这些数据是在内核引导期间由 setup.s 程序设置的。
60  */
61 // 下面三行分别将指定的线性地址强行转换为给定数据类型的指针，并获取指针所指内容。由于
62 // 内核代码段被映射到从物理地址零开始的地方，因此这些线性地址正好也是对应的物理地址。
63 // 这些指定地址处内存值的含义请参见第 6 章的表 6-2（setup 程序读取并保存的参数）。
64 // drive_info 结构请参见下面第 102 行。
65 #define EXT_MEM_K (*(unsigned short *)0x90002) // 1MB 以后的扩展内存大小（KB）。
66 #define DRIVE_INFO (*(struct drive_info *)0x90080) // 硬盘参数表的 32 字节内容。
67 #define ORIG_ROOT_DEV (*(unsigned short *)0x901FC) // 根文件系统所在设备号。
68
69 /*
70  * Yeah, yeah, it's ugly, but I cannot find how to do this correctly
71  * and this seems to work. I anybody has more info on the real-time
72  * clock I'd be interested. Most of this was trial and error, and some
73  * bios-listing reading. Urghh.
74  */
75 /*
76  * 是啊，是啊，下面这段程序很差劲，但我不知道如何正确地实现，而且好象
77  * 它还能运行。如果有关于实时时钟更多的资料，那我很感兴趣。这些都是试
78  * 探出来的，另外还看了一些 bios 程序，呵！
79  */
80
81 // 这段宏读取 CMOS 实时时钟信息。outb_p 和 inb_p 是 include/asm/io.h 中定义的端口输入输出宏。
82 #define CMOS_READ(addr) ({ \
83 outb_p(0x80|addr, 0x70); \ // 0x70 是写地址端口号，0x80|addr 是要读取的 CMOS 内存地址。
84 inb_p(0x71); \ // 0x71 是读数据端口号。
85 })
86
87 // 定义宏。将 BCD 码转换成二进制数值。BCD 码利用半个字节（4 比特）表示一个 10 进制数，因此

```

```

// 一个字节表示 2 个 10 进制数。(val)&15 取 BCD 表示的 10 进制个位数，而 (val)>>4 取 BCD 表示
// 的 10 进制十位数，再乘以 10。因此最后两者相加就是一个字节 BCD 码的实际二进制数值。
74 #define BCD_TO_BIN(val) ((val)=((val)&15) + ((val)>>4)*10)
75
// 该函数取 CMOS 实时钟信息作为开机时间，并保存到全局变量 startup_time(秒)中。参见后面
// CMOS 内存列表说明。其中调用的函数 kernel_mktime() 用于计算从 1970 年 1 月 1 日 0 时起到
// 开机当日经过的秒数，作为开机时间 (kernel/mktime.c 41 行)。
76 static void time_init(void)
77 {
78     struct tm time;           // 时间结构 tm 定义在 include/time.h 中。
79
// CMOS 的访问速度很慢。为了减小时间误差，在读取了下面循环中所有数值后，若此时 CMOS 中
// 秒值发生了变化，那么就重新读取所有值。这样内核就能把与 CMOS 时间误差控制在 1 秒之内。
80     do {
81         time.tm_sec = CMOS_READ(0);           // 当前时间秒值（均是 BCD 码值）。
82         time.tm_min = CMOS_READ(2);           // 当前分钟值。
83         time.tm_hour = CMOS_READ(4);          // 当前小时值。
84         time.tm_mday = CMOS_READ(7);          // 一月中的当天日期。
85         time.tm_mon = CMOS_READ(8);           // 当前月份（1—12）。
86         time.tm_year = CMOS_READ(9);          // 当前年份。
87     } while (time.tm_sec != CMOS_READ(0));
88     BCD_TO_BIN(time.tm_sec);                   // 转换成二进制数值。
89     BCD_TO_BIN(time.tm_min);
90     BCD_TO_BIN(time.tm_hour);
91     BCD_TO_BIN(time.tm_mday);
92     BCD_TO_BIN(time.tm_mon);
93     BCD_TO_BIN(time.tm_year);
94     time.tm_mon--;                             // tm_mon 中月份范围是 0—11。
95     startup_time = kernel_mktime(&time);       // 计算开机时间。kernel/mktime.c 41 行。
96 }
97
// 下面定义一些局部变量。
98 static long memory_end = 0;                   // 机器具有的物理内存容量（字节数）。
99 static long buffer_memory_end = 0;            // 高速缓冲区末端地址。
100 static long main_memory_start = 0;            // 主内存（将用于分页）开始的位置。
101
102 struct drive_info { char dummy[32]; } drive_info; // 用于存放硬盘参数表信息。
103
// 内核初始化主程序。初始化结束后将以任务 0（idle 任务即空闲任务）的身份运行。
104 void main(void)                               /* This really IS void, no error here. */
105 {                                              /* The startup routine assumes (well, ...) this */
// 这里确实是 void，没错。在 startup 程序(head.s)中就是这样假设的*/
// 参见 head.s 程序第 136 行开始的几行代码。
106 /*
107  * Interrupts are still disabled. Do necessary setups, then
108  * enable them
109  */
110 /*
111  * 此时中断仍被禁止着，做完必要的设置后就将其开启。
112  */
113 // 下面这段代码用于保存：
114 // 根设备号 → ROOT_DEV; 高速缓存末端地址 → buffer_memory_end;
115 // 机器内存数 → memory_end; 主内存开始地址 → main_memory_start;

```

```

// 其中 ROOT_DEV 已在前面包含进的 include/linux/fs.h 文件第 198 行上被声明为 extern int。
110     ROOT_DEV = ORIG_ROOT_DEV;           // ROOT_DEV 定义在 fs/super.c, 29 行。
111     drive_info = DRIVE_INFO;             // 复制 0x90080 处的硬盘参数表。
112     memory_end = (1<<20) + (EXT_MEM_K<<10); // 内存大小=1Mb + 扩展内存(k)*1024 字节。
113     memory_end &= 0xfffff000;            // 忽略不到 4Kb (1 页) 的内存数。
114     if (memory_end > 16*1024*1024)        // 如果内存量超过 16Mb, 则按 16Mb 计。
115         memory_end = 16*1024*1024;
116     if (memory_end > 12*1024*1024)        // 如果内存>12Mb, 则设置缓冲区末端=4Mb
117         buffer_memory_end = 4*1024*1024;
118     else if (memory_end > 6*1024*1024)    // 否则若内存>6Mb, 则设置缓冲区末端=2Mb
119         buffer_memory_end = 2*1024*1024;
120     else
121         buffer_memory_end = 1*1024*1024; // 否则则设置缓冲区末端=1Mb
122     main_memory_start = buffer_memory_end; // 主内存起始位置 = 缓冲区末端。

// 如果在 Makefile 文件中定义了内存虚拟盘符号 RAMDISK, 则初始化虚拟盘。此时主内存将减少。
// 参见 kernel/blk_drv/ramdisk.c。
123 #ifdef RAMDISK
124     main_memory_start += rd_init(main_memory_start, RAMDISK*1024);
125 #endif
// 以下是内核进行所有方面的初始化工作。阅读时最好跟着调用的程序深入进去看, 若实在看
// 不下去了, 就先放一放, 继续看下一个初始化调用 -- 这是经验之谈☺。
126     mem_init(main_memory_start, memory_end); // 主内存区初始化。(mm/memory.c, 399)
127     trap_init(); // 陷阱门(硬件中断向量)初始化。(kernel/traps.c, 181)
128     blk_dev_init(); // 块设备初始化。(kernel/blk_drv/ll_rw_blk.c, 157)
129     chr_dev_init(); // 字符设备初始化。(kernel/chr_drv/tty_io.c, 347)
130     tty_init(); // tty 初始化。(kernel/chr_drv/tty_io.c, 105)
131     time_init(); // 设置开机启动时间→startup_time (见 76 行)。
132     sched_init(); // 调度程序初始化(加载任务 0 的 tr, ldt) (kernel/sched.c, 385)
133     buffer_init(buffer_memory_end); // 缓冲管理初始化, 建内存链表等。(fs/buffer.c, 348)
134     hd_init(); // 硬盘初始化。(kernel/blk_drv/hd.c, 343)
135     floppy_init(); // 软驱初始化。(kernel/blk_drv/floppy.c, 457)
136     sti(); // 所有初始化工作都做完了, 开启中断。

// 下面过程通过在堆栈中设置的参数, 利用中断返回指令启动任务 0 执行。
137     move_to_user_mode(); // 移到用户模式下执行。(include/asm/system.h, 第 1 行)
138     if (!fork()) { // /* we count on this going ok */
139         init(); // 在新建的子进程(任务 1)中执行。
140     }

// 下面代码开始以任务 0 的身份运行。
141 /*
142  * NOTE!! For any other task 'pause()' would mean we have to get a
143  * signal to awaken, but task0 is the sole exception (see 'schedule()')
144  * as task 0 gets activated at every idle moment (when no other tasks
145  * can run). For task0 'pause()' just means we go check if some other
146  * task can run, and if not we return here.
147  */
/* 注意!! 对于任何其他任务, 'pause()' 将意味着我们必须等待收到一个信号
 * 才会返回就绪态, 但任务 0 (task0) 是唯一例外情况(参见'schedule()'),
 * 因为任务 0 在任何空闲时间里都会被激活(当没有其他任务在运行时), 因此
 * 对于任务 0 'pause()' 仅意味着我们返回来查看是否有其他任务可以运行, 如果
 * 没有的话我们就回到这里, 一直循环执行'pause()'。

```

```

    */
    // pause() 系统调用 (kernel/sched.c, 144) 会把任务 0 转换成可中断等待状态, 再执行调度函数。
    // 但是调度函数只要发现系统中没有其他任务可以运行时就会切换到任务 0, 而不依赖于任务 0 的
    // 状态。
148     for(;;) pause();
149 }
150
    // 下面函数产生格式化信息并输出到标准输出设备 stdout(1), 这里是指屏幕上显示。参数 '*fmt'
    // 指定输出将采用的格式, 参见标准 C 语言书籍。该子程序正好是 vsprintf 如何使用的一个简单
    // 例子。该程序使用 vsprintf() 将格式化的字符串放入 printbuf 缓冲区, 然后用 write() 将缓冲
    // 区的内容输出到标准设备 (1--stdout)。vsprintf() 函数的实现见 kernel/vsprintf.c。
151 static int printf(const char *fmt, ...)
152 {
153     va\_list args;
154     int i;
155
156     va\_start(args, fmt);
157     write(1, printbuf, i=vsprintf(printbuf, fmt, args));
158     va\_end(args);
159     return i;
160 }
161
    // 读取并执行/etc/rc 文件时所使用的命令行参数和环境参数。
162 static char * argv\_rc[] = { "/bin/sh", NULL };    // 调用执行程序时参数的字符串数组。
163 static char * envp\_rc[] = { "HOME=/", NULL };    // 调用执行程序时的环境字符串数组。
164
    // 运行登录 shell 时所使用的命令行参数和环境参数。
    // 第 165 行中 argv[0] 中的字符 "-" 是传递给 shell 程序 sh 的一个标志。通过识别该标志,
    // sh 程序会作为登录 shell 执行。其执行过程与在 shell 提示符下执行 sh 不一样。
165 static char * argv[] = { "/-bin/sh", NULL };    // 同上。
166 static char * envp[] = { "HOME=/usr/root", NULL };
167
    // 在 main() 中已经进行了系统初始化, 包括内存管理、各种硬件设备和驱动程序。init() 函数
    // 运行在任务 0 第 1 次创建的子进程 (任务 1) 中。它首先对第一个将要执行的程序 (shell)
    // 的环境进行初始化, 然后以登录 shell 方式加载该程序并执行之。
168 void init(void)
169 {
170     int pid, i;
171
    // setup() 是一个系统调用。用于读取硬盘参数包括分区表信息并加载虚拟盘 (若存在的话) 和
    // 安装根文件系统设备。该函数用 25 行上的宏定义, 对应函数是 sys_setup(), 在块设备子目录
    // kernel/blk_drv/hd.c, 71 行。
172     setup((void *) &drive\_info);    // drive_info 结构中是 2 个硬盘参数表。

    // 下面以读写访问方式打开设备 "/dev/tty0", 它对应终端控制台。由于这是第一次打开文件
    // 操作, 因此产生的文件句柄号 (文件描述符) 肯定是 0。该句柄是 UNIX 类操作系统默认的控
    // 制台标准输入句柄 stdin。这里再把它以读和写的方式分别打开是为了复制产生标准输出 (写)
    // 句柄 stdout 和标准出错输出句柄 stderr。函数前面的 "(void)" 前缀用于表示强制函数无需
    // 返回值。
173     (void) open("/dev/tty0", O\_RDWR, 0);
174     (void) dup(0);    // 复制句柄, 产生句柄 1 号--stdout 标准输出设备。
175     (void) dup(0);    // 复制句柄, 产生句柄 2 号--stderr 标准出错输出设备。

```



```

// 下面打印缓冲区块数和总字节数，每块 1024 字节，以及主内存区空闲内存字节数。
176     printf("%d buffers = %d bytes buffer space\n\r", NR BUFFERS,
177           NR BUFFERS*BLOCK_SIZE);
178     printf("Free mem: %d bytes\n\r", memory_end-main_memory_start);

// 下面 fork() 用于创建一个子进程（任务 2）。对于被创建的子进程，fork() 将返回 0 值，对于
// 原进程（父进程）则返回子进程的进程号 pid。所以第 180--184 行是子进程执行的内容。该子
// 进程关闭了句柄 0 (stdin)、以只读方式打开/etc/rc 文件，并使用 execve() 函数将进程自身
// 替换成 /bin/sh 程序（即 shell 程序），然后执行 /bin/sh 程序。所携带的参数和环境变量分
// 别由 argv_rc 和 envp_rc 数组给出。关闭句柄 0 并立刻打开 /etc/rc 文件的作用是把标准输入
// stdin 重定向到 /etc/rc 文件。这样 shell 程序/bin/sh 就可以运行 rc 文件中设置的命令。由
// 于这里 sh 的运行方式是非交互式的，因此在执行完 rc 文件中的命令后就会立刻退出，进程 2
// 也随之结束。关于 execve() 函数说明请参见 fs/exec.c 程序，182 行。
// 函数_exit() 退出时的出错码 1 - 操作未许可；2 -- 文件或目录不存在。

179     if (!(pid=fork())) {
180         close(0);
181         if (open("/etc/rc", O_RDONLY, 0))
182             _exit(1); // 如果打开文件失败，则退出(lib/_exit.c, 10)。
183         execve("/bin/sh", argv_rc, envp_rc); // 替换成/bin/sh 程序并执行。
184         _exit(2); // 若 execve() 执行失败则退出。
185     }

// 下面还是父进程（1）执行的语句。wait() 等待子进程停止或终止，返回值应是子进程的进程号
// (pid)。这三句的作用是父进程等待子进程的结束。&i 是存放返回状态信息的位置。如果 wait()
// 返回值不等于子进程号，则继续等待。

186     if (pid>0)
187         while (pid != wait(&i))
188             /* nothing */; /* 空循环 */

// 如果执行到这里，说明刚创建的子进程的执行已停止或终止了。下面循环中首先再创建一个子
// 进程，如果出错，则显示“初始化程序创建子进程失败”信息并继续执行。对于所创建的子进
// 程将关闭所有以前还遗留的句柄(stdin, stdout, stderr)，新创建一个会话并设置进程组号，
// 然后重新打开 /dev/tty0 作为 stdin，并复制成 stdout 和 stderr。再次执行系统解释程序
// /bin/sh。但这次执行所选用的参数和环境数组另选了一套（见上面 165--167 行）。然后父进
// 程再次运行 wait() 等待。如果子进程又停止了执行，则在标准输出上显示出错信息“子进程
// pid 停止了运行，返回码是 i”，然后继续重试下去...，形成一个“大”循环。此外，wait()
// 的另外一个功能是处理孤儿进程。如果一个进程的父进程先终止了，那么这个进程的父进程
// 就会被设置为这里的 init 进程（进程 1），并由 init 进程负责释放一个已终止进程的任务数
// 据结构等资源。

189     while (1) {
190         if ((pid=fork())<0) {
191             printf("Fork failed in init\r\n");
192             continue;
193         }
194         if (!pid) { // 新的子进程。
195             close(0);close(1);close(2);
196             setsid(); // 创建一新的会话期，见后面说明。
197             (void) open("/dev/tty0", O_RDWR, 0);
198             (void) dup(0);

```

```

199         (void) dup(0);
200         _exit(execute("/bin/sh", argv, envp));
201     }
202     while (1)
203         if (pid == wait(&i))
204             break;
205     printf("\n\rchild %d died with code %04x\n\r", pid, i);
206     sync(); // 同步操作, 刷新缓冲区。
207 }
208 _exit(0); /* NOTE! _exit, not exit() */ /* 注意! 是_exit(), 非 exit() */
// _exit() 和 exit() 都用于正常终止一个函数。但_exit() 直接是一个 sys_exit 系统调用, 而
// exit() 则通常是普通函数库中的一个函数。它会先执行一些清除操作, 例如调用执行各终止
// 处理程序、关闭所有标准 I/O 等, 然后调用 sys_exit。
209 }
210

```

## 7.1.3 其他信息

### 7.1.3.1 CMOS 信息

PC 机的 CMOS 内存是由电池供电的 64 或 128 字节内存块, 通常是系统实时钟芯片 RTC (Real Time Chip) 的一部分。有些机器还有更大的内存容量。该 64 字节的 CMOS 原先在 IBM PC-XT 机器上用于保存时钟和日期信息, 存放的格式是 BCD 码。由于这些信息仅用去 14 字节, 因此剩余的字节就可用来存放一些系统配置数据。

CMOS 的地址空间在基本地址空间之外, 因此其中不包括可执行代码。要访问它需要通过端口 0x70、0x71 进行。0x70 是地址端口, 0x71 是数据端口。为了读取指定偏移位置的字节, 必须首先使用 OUT 指令向地址端口 0x70 发送指定字节的偏移位置值, 然后使用 IN 指令从数据端口 0x71 读取指定的字节信息。同样, 对于写操作也需要首先向地址端口 0x70 发送指定字节的偏移值, 然后把数据写到数据端口 0x71 中去。

main.c 程序第 70 行语句把欲读取的字节地址与 0x80 进行或操作是没有必要的。因为那时的 CMOS 内存容量还没有超过 128 字节, 因此与 0x80 进行或操作是没有任何作用的。之所以会有这样的操作是因为当时 Linus 手头缺乏有关 CMOS 方面的资料, CMOS 中时钟和日期的偏移地址都是他逐步实验出来的, 也许在他实验中将偏移地址与 0x80 进行或操作(并且还修改了其他地方)后正好取得了所有正确的结果, 因此他的代码中也就有了这步不必要的操作。不过从 1.0 版本之后, 该操作就被去除了(可参见 1.0 版内核程序 drivers/block/hd.c 第 42 行起的代码)。表 7-1 是 CMOS 内存信息的一张简表。

表 7-1 CMOS 64 字节信息简表

地址偏移值	内容说明	地址偏移值	内容说明
0x00	当前秒值 (实时钟)	0x11	保留
0x01	报警秒值	0x12	硬盘驱动器类型
0x02	当前分钟 (实时钟)	0x13	保留
0x03	报警分钟值	0x14	设备字节
0x04	当前小时值 (实时钟)	0x15	基本内存 (低字节)
0x05	报警小时值	0x16	基本内存 (高字节)
0x06	一周中的当前天 (实时钟)	0x17	扩展内存 (低字节)
0x07	一月中的当日日期 (实时钟)	0x18	扩展内存 (高字节)
0x08	当前月份 (实时钟)	0x19-0x2d	保留
0x09	当前年份 (实时钟)	0x2e	校验和 (低字节)



0x0a	RTC 状态寄存器 A	0x2f	校验和 (高字节)
0x0b	RTC 状态寄存器 B	0x30	1Mb 以上的扩展内存 (低字节)
0x0c	RTC 状态寄存器 C	0x31	1Mb 以上的扩展内存 (高字节)
0x0d	RTC 状态寄存器 D	0x32	当前所处世纪值
0x0e	POST 诊断状态字节	0x33	信息标志
0x0f	停机状态字节	0x34-0x3f	保留
0x10	磁盘驱动器类型		

### 7.1.3.2 调用 fork() 创建新进程

fork 是一个系统调用函数。该系统调用复制当前进程，并在进程表中创建一个与原进程(被称为父进程)几乎完全一样的新表项，并执行同样的代码，但该新进程(这里被称为子进程)拥有自己的数据空间和环境参数。创建新进程的主要用途在于在新进程中使用 exec() 族函数去执行其他不同的程序。

在 fork 调用返回位置处，父进程将恢复执行，而子进程则开始执行。在父进程中，调用 fork() 返回的是子进程的进程标识号 PID，而在子进程中 fork() 返回的将是 0 值，这样，虽然此时还是在同样一程序中执行，但已开始叉开，各自执行自己的那段代码。如果 fork() 调用失败，则会返回小于 0 的值。如示意图 7-3 所示。

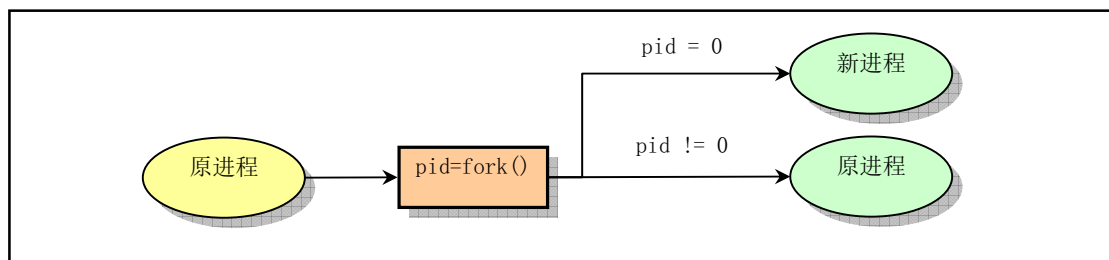


图 7-3 调用 fork() 创建新进程

init 程序即是用 fork() 调用的返回值来区分和执行不同的代码段的。上面代码中第 179 和 194 行是子进程的判断并开始子进程代码块的执行(利用 execve() 系统调用执行其他程序，这里执行的是 sh)，第 186 和 202 行是父进程执行的代码块。

当程序执行完或有必要终止时就可以调用 exit() 来退出程序的执行。该函数会终止进程并释放其占用的内核资源。而父进程则可以使用 wait() 调用来查看或等待子进程的退出，并获取被终止进程的退出状态信息。

### 7.1.3.3 关于会话期(session)的概念

在第 2 章我们说过，程序是一个可执行的文件，而进程(process)是一个执行中的程序实例。在内核中，每个进程都使用一个不同的大于零的正整数来标识，称为进程标识号 pid(Process ID)。而一个进程可以通过 fork() 调用创建一个或多个子进程，这些进程就可以构成一个进程组。例如，对于下面在 shell 命令行上键入的一个管道命令，

```
[plinux root]# cat main.c | grep for | more
```

其中的每个命令：cat、grep 和 more 就都属于一个进程组。

进程组是一个或多个进程的集合。与进程类似，每个进程组都有一个唯一的进程组标识号 gid(Group ID)。进程组号 gid 也是一个正整数。每一个进程组有一个称为组长的进程，组长进程就是其进程号 pid 等于进程组号 gid 的进程。一个进程可以通过调用 setpgid() 来参加一个现有的进程组或者创建一个新的

进程组。进程组的概念有很多用途，但其中最常见的是我们在终端上向前台执行程序发出终止信号（通常是按 **Ctrl-C** 组合键），同时终止整个进程组中的所有进程。例如，如果我们向上述管道命令发出终止信号，则三个命令将同时终止执行。

而会话期（Session，或称为会话）则是一个或多个进程组的集合。通常情况下，用户登录后所执行的所有程序都属于一个会话期，而其登录 shell 则是会话期首进程（Session leader），并且它所使用的终端就是会话期的控制终端（Controlling Terminal），因此会话期首进程通常也被称为控制进程（Controlling process）。当我们退出登录（logout）时，所有属于我们这个会话期的进程都将被终止。这也是会话期概念的主要用途之一。setsid()函数就是用于建立一个新的会话期。通常该函数由环境初始化程序进行调用，见下节说明。进程、进程组和会话期之间的关系见图 7-4 所示。

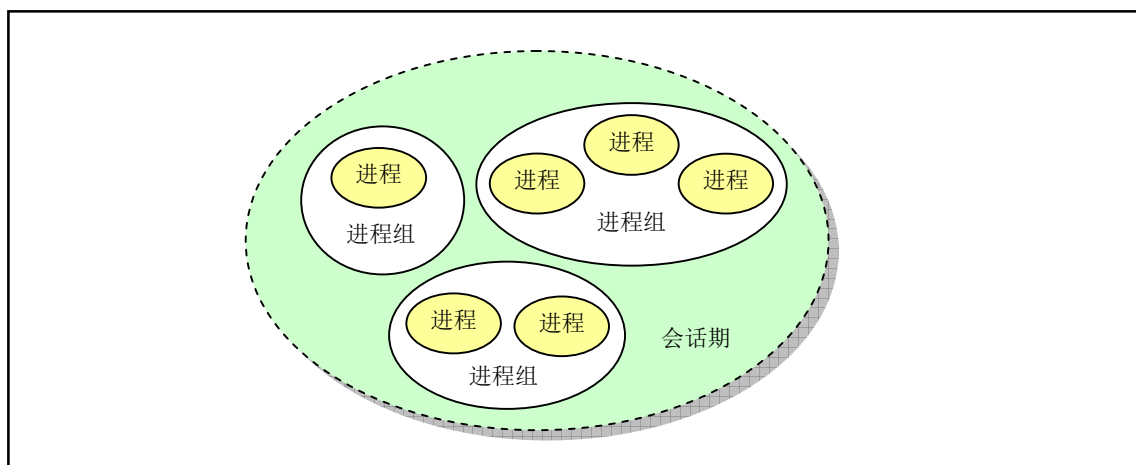


图 7-4 进程、进程组和会话期之间的关系

一个会话期中的几个进程组被分为一个前台进程组（Foreground process group）和一个或几个后台进程组（Background process group）。一个终端只能作为一个会话期的控制终端，前台进程组就是会话期中拥有控制终端的一个进程组，而会话期中的其它进程组则成为后台进程组。控制终端对应于/dev/tty 设备文件，因此若一个进程需要访问控制终端，可以直接对/dev/tty 文件进行读写操作。

## 7.2 环境初始化工作

在内核系统初始化完毕之后，系统还需要根据具体配置执行进一步的环境初始化工作，才能真正具备一个常用系统所具备的一些工作环境。在前面的第 183 行和 200 行上，init()函数直接开始执行了命令解释程序（shell 程序）/bin/sh，而在实际可用的系统中却并非如此。为了能具有登录系统的功能和多人同时使用系统的能力，通常的系统是在这里或类似地方执行系统环境初始化程序 init.c，而此程序会根据系统/etc/目录中配置文件的设置信息，对系统中支持的每个终端设备创建子进程，并在子进程中运行终端初始化设置程序 agetty（统称 getty 程序），getty 程序则会在终端上显示用户登录提示信息“login:”。当用户键入了用户名后，getty 被替换成 login 程序。login 程序在验证了用户输入口令的正确性以后，最终调用 shell 程序，并进入 shell 交互工作界面。它们之间的执行关系见图 7-5 所示。

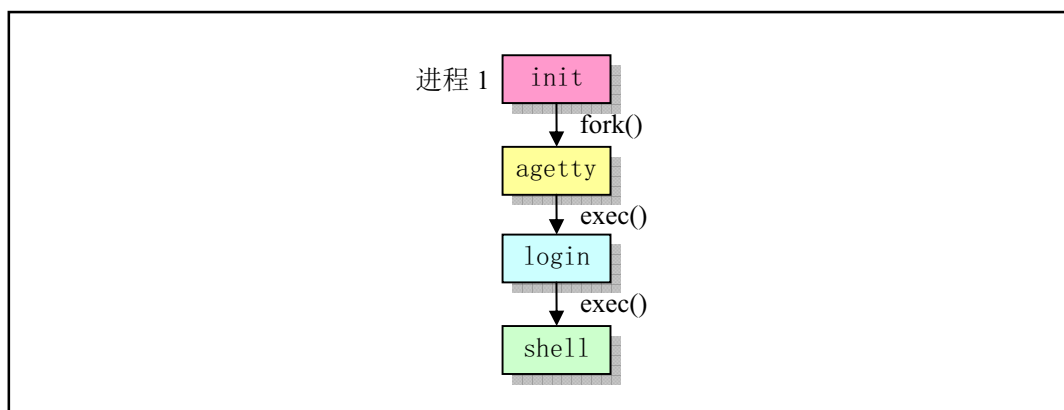


图 7-5 有关环境初始化的程序

虽然这几个程序（init, getty, login, shell）并不属于内核范畴，但对这几个程序的作用有一些基本了解会促进对内核为什么提供那么多功能的理解。

init 进程的主要任务是根据/etc/rc 文件中设置的信息，执行其中设置的命令，然后根据/etc/inittab 文件中的信息，为每一个允许登录的终端设备使用 fork() 创建一个子进程，并在每个新创建的子进程中运行 agetty<sup>9</sup>（getty）程序。而 init 进程则调用 wait()，进入等待子进程结束状态。每当它的一个子进程结束退出，它会根据 wait() 返回的 pid 号知道是哪个对应终端的子进程结束了，因此就会为相应终端设备再创建一个新的子进程，并在该子进程中重新执行 agetty 程序。这样，每个被允许的终端设备都始终有一个对应的进程为其等待处理。

在正常操作下，init 确定 agetty 正在工作着以允许用户登录，并且收取孤立进程。孤立进程是指那些其父进程已结束的进程；在 Linux 中所有的进程必须属于单棵进程树，所以孤立进程必须被收取。当系统关闭时，init 负责杀死所有其他的进程，卸载所有的文件系统以及停止处理器的工作，以及任何它被配置成要做的工作。

getty 程序的主要任务是设置终端类型、属性、速度和线路规程。它打开并初始化一个 tty 端口，显示提示信息，并等待用户键入用户名。该程序只能由超级用户执行。通常，若/etc/issue 文本文件存在，则 getty 会首先显示其中的文本信息，然后显示登录提示信息（例如：plinux login: ），读取用户键入的登录名，并执行 login 程序。

login 程序则主要用于要求登录用户输入密码。根据用户输入的用户名，它从口令文件 passwd 中取得对应用户的登录项，然后调用 getpass() 以显示“password:”提示信息，读取用户键入的密码，然后使用加密算法对键入的密码进行加密处理，并与口令文件中该用户项中 pw\_passwd 字段作比较。如果用户几次键入的密码均无效，则 login 程序会以出错码 1 退出执行，表示此次登录过程失败。此时父进程（进程 init）的 wait() 会返回该退出进程的 pid，因此会根据记录下来的信息再次创建一个子进程，并在该子进程中针对该终端设备再次执行 agetty 程序，重复上述过程。

如果用户键入的密码正确，则 login 就会把当前工作目录（Current Work Directory）修改成口令文件中指定的该用户的起始工作目录。并把对该终端设备的访问权限修改成用户读/写和组写，设置进程的组 ID。然后利用所得到的信息初始化环境变量信息，例如起始目录（HOME=）、使用的 shell 程序（SHELL=）、用户名（USER=和 LOGNAME=）和系统执行程序的默认路径序列（PATH=）。接着显示/etc/motd 文件（message-of-the-day）中的文本信息，并检查并显示该用户是否有邮件的信息。最后 login 程序改变成登录用户的用户 ID 并执行口令文件中该用户项中指定的 shell 程序，如 bash 或 csh 等。

如果口令文件/etc/passwd 中该用户项中没有指定使用哪个 shell 程序，系统则会使用默认的/bin/sh 程序。如果口令文件中也没有为该用户指定用户起始目录的话，系统就会使用默认的根目录/。有关 login

<sup>9</sup> agetty - alternative Linux getty。

程序的一些执行选项和特殊访问限制的说明，请参见 Linux 系统中的在线手册页（man 8 login）。

shell 程序是一个复杂的命令行解释程序，是当用户登录系统进行交互操作时执行的程序。它是用户与计算机进行交互操作的地方。它获取用户输入的信息，然后执行命令。用户可以在终端上向 shell 直接进行交互输入，也可以使用 shell 脚本文件向 shell 解释程序输入。

在登录过程中 login 开始执行 shell 时，所带参数 argv[0]的第一个字符是'-'，表示该 shell 是作为一个登录 shell 被执行。此时该 shell 程序会根据该字符，执行某些与登录过程相应的操作。登录 shell 会首先从/etc/profile 文件以及.profile 文件（若存在的话）读取命令并执行。如果在进入 shell 时设置了 ENV 环境变量，或者在登录 shell 的.profile 文件中设置了该变量，则 shell 下一步会从该变量命名的文件中读取命令并执行。因此用户应该把每次登录时都要执行的命令放在.profile 文件中，而把每次运行 shell 都要执行的命令放在 ENV 变量指定的文件中。设置 ENV 环境变量的方法是把下列语句放在你起始目录的.profile 文件中：

```
ENV=$HOME/.anyfilename; export ENV
```

在执行 shell 时，除了一些指定的可选项以外，如果还指定了命令行参数，则 shell 会把第一个参数看作是一个脚本文件名并执行其中的命令，而其余的参数则被看作是 shell 的位置参数（\$1、\$2 等）。否则 shell 程序将从其标准输入中读取命令。

在执行 shell 程序时可以有很多选项，请参见 Linux 系统中的有关 sh 的在线手册页中的说明。

## 7.3 本章小结

对于 0.11 版内核，通过上面代码分析可知，只要根文件系统是一个 MINIX 文件系统，并且其中只包含文件/etc/rc、/bin/sh、/dev/\* 以及一些目录/etc/、/dev/、/bin/、/home/、/home/root/ 就可以构成一个最简单的根文件系统，让 Linux 运行起来。

从这里开始，对于后续章节的阅读，可以将 main.c 程序作为一条主线进行，并不需要按章节顺序阅读。若读者对内存分页管理机制不了解，则建议首先阅读第 10 章内存管理的内容。

为了能比较顺利地理解以下各章内容，作者强力希望读者此时能再次复习 32 位保护模式运行的机制，详细阅读一下附录中所提供的有关内容，或者参考 Intel 80x86 的有关书籍，把保护模式下的运行机制彻底弄清楚，然后再继续阅读。

如果您按章节顺序顺利地阅读到这里，那么您对 Linux 系统内核的初始化过程应该已经有了大致的了解。但您可能还会提出这样的问题：“在生成了一系列进程之后，系统是如何分时运行这些进程或者说如何调度这些进程运行的呢？也即‘轮子’是怎样转起来的呢？”。答案并不复杂：内核是通过执行 sched.c 程序中的调度函数 schedule()和 system\_call.s 中的定时时钟中断过程\_timer\_interrupt 来操作的。内核设定每 10 毫秒发出一次时钟中断，并在该中断过程中，通过调用 do\_timer()函数检查所有进程的当前执行情况来确定进程的下一步状态。

对于进程在执行过程中由于想用的资源暂时缺乏而临时需要等待一会时，它就会在系统调用中通过 sleep\_on()类函数间接地调用 schedule()函数，将 CPU 的使用权自愿地移交给别的进程使用。至于系统接下来会运行哪个进程，则完全由 schedule()根据所有进程的当前状态和优先权决定。对于一直在可运行状态的进程，当时钟中断过程判断出它运行的时间片已被用完时，就会在 do\_timer()中执行进程切换操作，该进程的 CPU 使用权就会被不情愿地剥夺，让给别的进程使用。

















调度函数 schedule()和时钟中断过程即是下一章中的主题之一。



## 第8章 内核代码(kernel)

linux/kernel/目录下共包括 10 个 C 语言文件和 2 个汇编语言文件以及一个 kernel 下编译文件的管理配置文件 Makefile。见列表 8-1 所示。对其中三个子目录中代码的注释将在后续章节中进行。本章主要对这 13 个代码文件进行注释。首先我们对所有程序的基本功能进行概括性的总体介绍，以便一开始就对这 12 个文件所实现的功能和它们之间的相互调用关系有个大致的了解，然后逐一对代码进行详细注释。

列表 8-1 linux/kernel/目录

文件名	大小	最后修改时间 (GMT)	说明
 <a href="#">blk_drv/</a>		1991-12-08 14:09:29	
 <a href="#">chr_drv/</a>		1991-12-08 18:36:09	
 <a href="#">math/</a>		1991-12-08 14:09:58	
 <a href="#">Makefile</a>	3309 bytes	1991-12-02 03:21:37	m
 <a href="#">asm.s</a>	2335 bytes	1991-11-18 00:30:28	m
 <a href="#">exit.c</a>	4175 bytes	1991-12-07 15:47:55	m
 <a href="#">fork.c</a>	3693 bytes	1991-11-25 15:11:09	m
 <a href="#">mktime.c</a>	1461 bytes	1991-10-02 14:16:29	m
 <a href="#">panic.c</a>	448 bytes	1991-10-17 14:22:02	m
 <a href="#">printk.c</a>	734 bytes	1991-10-02 14:16:29	m
 <a href="#">sched.c</a>	8242 bytes	1991-12-04 19:55:28	m
 <a href="#">signal.c</a>	2651 bytes	1991-12-07 15:47:55	m
 <a href="#">sys.c</a>	3706 bytes	1991-11-25 19:31:13	m
 <a href="#">system_call.s</a>	5265 bytes	1991-12-04 13:56:34	m
 <a href="#">traps.c</a>	4951 bytes	1991-10-30 20:20:40	m
 <a href="#">vsprintf.c</a>	4800 bytes	1991-10-02 14:16:29	m

### 8.1 总体功能

该目录下的代码文件从功能上可以分为三类，一类是硬件（异常）中断处理程序文件，一类是系统调用服务处理程序文件，另一类是进程调度等通用功能文件，参见图 8-1 图 2-17。我们现在根据这个分类方式，从实现的功能上进行更详细的说明。



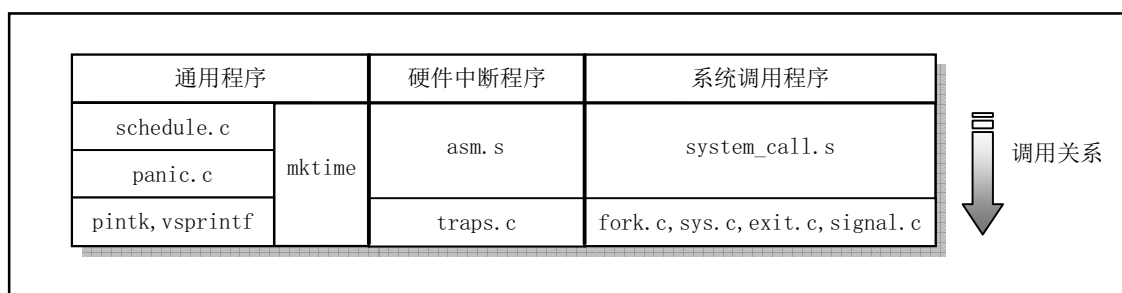


图 8-1 内核目录中各文件中函数的调用层次关系

### 8.1.1 中断处理程序

主要包括两个代码文件：`asm.s` 和 `traps.c` 文件。`asm.s` 用于实现大部分硬件异常所引起的中断的汇编语言处理过程。而 `traps.c` 程序则实现了 `asm.s` 的中断处理过程中调用的 `c` 函数。另外几个硬件中断处理程序在文件 `system_call.s` 和 `mm/page.s` 中实现。有关 PC 机中 8259A 可编程中断控制芯片的连接及其功能请参见图 5-21。

在用户程序（进程）将控制权交给中断处理程序之前，CPU 会首先将至少 12 字节（EFLAGS、CS 和 EIP）的信息压入中断处理程序的堆栈中，即进程的内核态栈中，见图 8-2(a)所示。这种情况与一个远调用（段间子程序调用）比较相像。CPU 会将代码段选择符和返回地址的偏移值压入堆栈。另一个与段间调用比较相像的地方是 80386 将信息压入到了目的代码（中断处理程序代码）的堆栈上，而不是被中断代码的堆栈中。如果优先级级别发生了变化，例如从用户级改变到内核系统级，CPU 还会将原代码的堆栈段值和堆栈指针压入中断程序的堆栈中。但在内核初始化完成后，内核代码执行时使用的是进程的\*\*内核态栈\*\*。因此这里目的代码的堆栈即是指进程的\*\*内核态堆栈\*\*，而被中断代码的堆栈当然也就是指进程的\*\*用户态堆栈\*\*了。所以当发生中断时，中断处理程序使用的是进程的\*\*内核态堆栈\*\*。另外，CPU 还总是将标志寄存器 EFLAGS 的内容压入堆栈。对于具有优先级改变时堆栈的内容示意图见图 8-2(c)和(d)所示。

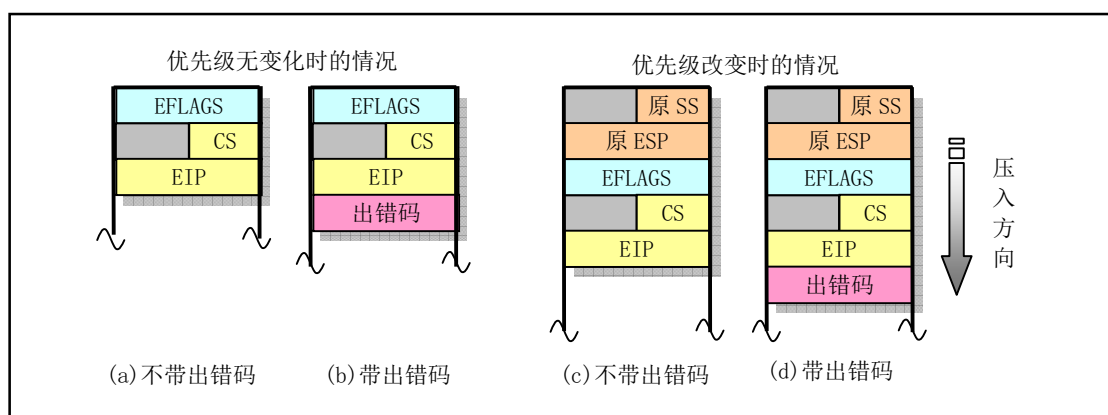


图 8-2 发生中断时堆栈中的内容

`asm.s` 代码文件主要涉及对 Intel 保留中断 `int0--int16` 的处理，其余保留的中断 `int17--int31` 由 Intel 公司留作今后扩充使用。对应于中断控制器芯片各 IRQ 发出的 `int32--int47` 的 16 个处理程序将分别在各种硬件（如时钟、键盘、软盘、数学协处理器、硬盘等）初始化程序中处理。Linux 系统调用中断 `int128(0x80)` 的处理则将在 `kernel/system_call.s` 中给出。各个中断的具体定义见代码注释后其他信息一节中的说明。

由于有些异常引起中断时，CPU 内部会产生一个出错代码压入堆栈（异常中断 `int 8` 和 `int10 - int 14`），



见图 8-2 (b)所示，而其他的中断却并不带有这个出错代码（例如被零除出错和边界检查出错等），因此，asm.s 程序中会根据是否携带出错代码而把中断分成两类分别进行处理。但处理流程还是一样的。

对一个硬件异常所引起的中断的处理过程见图 8-3 所示。

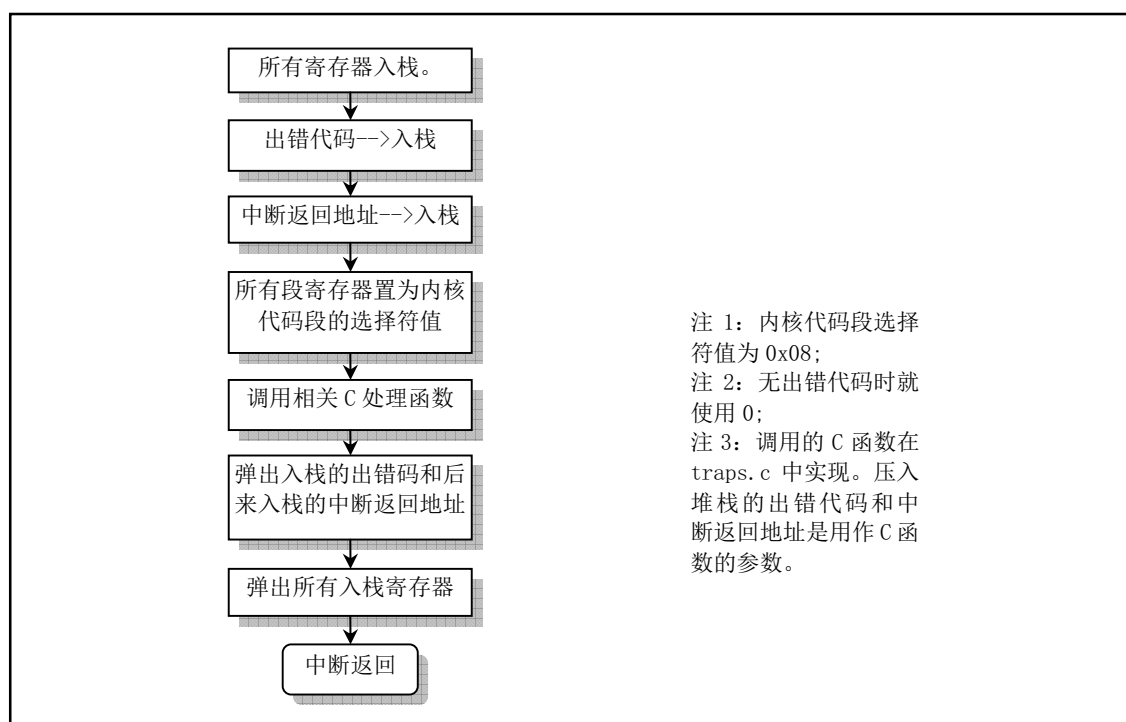


图 8-3 硬件异常（故障、陷阱）所引起的中断处理流程

### 8.1.2 系统调用处理相关程序

Linux 中应用程序调用内核的功能是通过中断调用 int 0x80 进行的，寄存器 eax 中放调用号，如果需要带参数，则 ebx、ecx 和 edx 用于存放调用参数。因此该中断调用被称为系统调用。实现系统调用的相关文件包括 system\_call.s、fork.c、signal.c、sys.c 和 exit.c 文件。

system\_call.s 程序的作用类似于硬件中断处理中 asm.s 程序的作用，另外还对时钟中断和硬盘、软盘中断进行处理。而 fork.c 和 signal.c 中的一个函数则类似于 traps.c 程序的作用，它们为系统中断调用提供 C 处理函数。fork.c 程序提供两个 C 处理函数：find\_empty\_process()和 copy\_process()。signal.c 程序还提供一个处理有关进程信号的函数 do\_signal()，在系统调用中断处理过程中被调用。另外还包括 4 个系统调用 sys\_xxx()函数。

sys.c 和 exit.c 程序实现了其他一些 sys\_xxx()系统调用函数。这些 sys\_xxx()函数都是相应系统调用所需调用的处理函数，有些是使用汇编语言实现的，如 sys\_execve()；而另外一些则用 C 语言实现（例如 signal.c 中的 4 个系统调用函数）。

我们可以根据这些函数的简单命名规则这样来理解：通常以'do\_'开头的中断处理过程中调用的 C 函数，要么是系统调用处理过程中通用的函数，要么是某个系统调用专用的；而以'sys\_'开头的系统调用函数则是指定的系统调用的专用处理函数。例如，do\_signal()函数基本上是所有系统调用都要执行的函数，而 sys\_pause()、sys\_execve()则是某个系统调用专用的 C 处理函数。

### 8.1.3 其他通用类程序

这些程序包括 schedule.c、mktime.c、panic.c、printk.c 和 vsprintf.c。

`schedule.c` 程序包括内核调用最频繁的 `schedule()`、`sleep_on()` 和 `wakeup()` 函数，是内核的核心调度程序，用于对进程的执行进行切换或改变进程的执行状态。另外还包括有关系统时钟中断和软盘驱动器定时函数。`mktime.c` 程序中仅包含一个内核使用的时间函数 `mktime()`，仅在 `init/main.c` 中被调用一次。`panic.c` 中包含一个 `panic()` 函数，用于在内核运行出现错误时显示出错信息并停机。`printk.c` 和 `vsprintf.c` 是内核显示信息的支持程序，实现了内核专用显示函数 `printk()` 和字符串格式化输出函数 `vsprintf()`。

## 8.2 Makefile 文件

### 8.2.1 功能简介

该文件是编译 `linux/kernel/` 目录下程序的 `make` 配置文件，不包括编译其中三个子目录程序的配置。该文件的组成格式与第 5 章中列出的 `Makefile` 的基本相同，在阅读时可以参考程序 5-1 中的有关注释。

### 8.2.2 文件注释

程序 8-1 `linux/kernel/Makefile`

```

1 #
2 # Makefile for the FREAX-kernel.
3 #
4 # Note! Dependencies are done automagically by 'make dep', which also
5 # removes any old dependencies. DON'T put your own dependencies here
6 # unless it's something special (ie not a .c file).
7 #
8 # FREAX 内核的 Makefile 文件。
9 #
10 # 注意！依赖关系是由 'make dep' 自动进行的，它也会自动去除原来的依赖信息。不要把你自己的
11 # 依赖关系信息放在这里，除非是特别文件的（也即不是一个 .c 文件的信息）。
12 # (Linux 最初的名字叫 FREAX，后来被 ftp.funet.fi 的管理员改成 Linux 这个名字)
13
14 AR      =gar      # GNU 的二进制文件处理程序，用于创建、修改以及从归档文件中抽取文件。
15 AS      =gas      # GNU 的汇编程序。
16 LD      =gld      # GNU 的连接程序。
17 LDFLAGS =-s -x    # 连接程序所有的参数，-s 输出文件中省略所有符号信息。-x 删除所有局部符号。
18 CC      =gcc      # GNU C 语言编译器。
19 CFLAGS  =-Wall -O -fstrength-reduce -fomit-frame-pointer -fcombine-regs \
20         -finline-functions -mstring-insns -nostdinc -I../include
21 # C 编译程序选项。-Wall 显示所有的警告信息；-O 优化选项，优化代码长度和执行时间；
22 # -fstrength-reduce 优化循环执行代码，排除重复变量；-fomit-frame-pointer 省略保存不必要
23 # 的框架指针；-fcombine-regs 合并寄存器，减少寄存器类的使用；-finline-functions 将所有简
24 # 单短小的函数代码嵌入调用程序中；-mstring-insns Linus 自己添加的优化选项，以后不再使用；
25 # -nostdinc -I../include 不使用默认路径中的包含文件，而使用这里指定目录中的(../include)。
26 CPP     =gcc -E -nostdinc -I../include
27 # C 前处理选项。-E 只运行 C 前处理，对所有指定的 C 程序进行预处理并将处理结果输出到标准输
28 # 出设备或指定的输出文件中；-nostdinc -I../include 同前。
29
30 # 下面的规则指示 make 利用下面的命令将所有的 .c 文件编译生成 .s 汇编程序。该规则的命令
31 # 指使 gcc 采用 CFLAGS 所指定的选项对 C 代码编译后不进行汇编就停止 (-S)，从而产生与
32 # 输入的各个 C 文件对应的汇编代码文件。默认情况下所产生的汇编程序文件名是原 C 文件名
33 # 去掉 .c 而加上 .s 后缀。-o 表示其后是输出文件的名称。其中 *.s (或 $@) 是自动目标变量，

```

```

# $<代表第一个先决条件，这里即是符合条件*.c 的文件。
# 下面这 3 个不同规则分别用于不同的操作要求。若目标是.s 文件，而源文件是.c 文件则会使
# 用第一个规则；若目标是.o，而源文件是.s，则使用第 2 个规则；若目标是.o 文件而源文件
# 是.c 文件，则可直接使用第 3 个规则。
18 .c.s:
19     $(CC) $(CFLAGS) \
20     -S -o $*.s $<
# 下面规则表示将所有.s 汇编程序文件编译成.o 目标文件。22 行是实现该操作的具体命令。
21 .s.o:
22     $(AS) -c -o $*.o $<
23 .c.o:                                # 类似上面，*.c 文件→*.o 目标文件。不进行连接。
24     $(CC) $(CFLAGS) \
25     -c -o $*.o $<
26
27 OBJS = sched.o system_call.o traps.o asm.o fork.o \    # 定义目标文件变量 OBJS。
28     panic.o printk.o vsprintf.o sys.o exit.o \
29     signal.o mktime.o
30
# 在有了先决条件 OBJS 后使用下面的命令连接成目标 kernel.o。
# 选项 '-r' 用于指示生成可重定位的输出，即产生可以作为链接器 ld 输入的目标文件。
31 kernel.o: $(OBJS)
32     $(LD) -r -o kernel.o $(OBJS)
33     sync
34
# 下面的规则用于清理工作。当执行'make clean'时，就会执行 36--40 行上的命令，去除所有编译
# 连接生成的文件。'rm' 是文件删除命令，选项-f 含义是忽略不存在的文件，并且不显示删除信息。
35 clean:
36     rm -f core *.o *.a tmp_make keyboard.s
37     for i in *.c;do rm -f `basename $$i .c`.s;done
38     (cd chr_drv; make clean)    # 进入 chr_drv/目录；执行该目录 Makefile 中的 clean 规则。
39     (cd blk_drv; make clean)
40     (cd math; make clean)
41
# 下面的目标或规则用于检查各文件之间的依赖关系。方法如下：
# 使用字符串编辑程序 sed 对 Makefile 文件（这里即是自己）进行处理，输出为删除 Makefile
# 文件中'### Dependencies' 行后面的所有行（下面从 51 开始的行），并生成 tmp_make
# 临时文件（43 行的作用）。然后对 kernel/目录下的每一个 C 文件执行 gcc 预处理操作。
# -M 标志告诉预处理程序输出描述每个目标文件相关性的规则，并且这些规则符合 make 语法。
# 对于每一个源文件，预处理程序输出一个 make 规则，其结果形式是相应源程序文件的目标
# 文件名加上其依赖关系——该源文件中包含的所有头文件列表。把预处理结果都添加到临时
# 文件 tmp_make 中，然后将该临时文件复制成新的 Makefile 文件。
42 dep:
43     sed '/\#\#\# Dependencies/q' < Makefile > tmp_make
44     (for i in *.c;do echo -n `echo $$i | sed 's,\.c,\.s,\'`"; \
45         $(CPP) -M $$i;done) >> tmp_make
46     cp tmp_make Makefile
47     (cd chr_drv; make dep)    # 对 chr_drv/目录下的 Makefile 文件也作同样的处理。
48     (cd blk_drv; make dep)
49
50 ### Dependencies:
51 exit.s exit.o : exit.c ../include/errno.h ../include/signal.h \
52     ../include/sys/types.h ../include/sys/wait.h ../include/linux/sched.h \
53     ../include/linux/head.h ../include/linux/fs.h ../include/linux/mm.h \

```

---

```

54  ../include/linux/kernel.h ../include/linux/tty.h ../include/termios.h \
55  ../include/asm/segment.h
56 fork.s fork.o : fork.c ../include/errno.h ../include/linux/sched.h \
57  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
58  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
59  ../include/asm/segment.h ../include/asm/system.h
60 mktime.s mktime.o : mktime.c ../include/time.h
61 panic.s panic.o : panic.c ../include/linux/kernel.h ../include/linux/sched.h \
62  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
63  ../include/linux/mm.h ../include/signal.h
64 printk.s printk.o : printk.c ../include/stdarg.h ../include/stddef.h \
65  ../include/linux/kernel.h
66 sched.s sched.o : sched.c ../include/linux/sched.h ../include/linux/head.h \
67  ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
68  ../include/signal.h ../include/linux/kernel.h ../include/linux/sys.h \
69  ../include/linux/fdreg.h ../include/asm/system.h ../include/asm/io.h \
70  ../include/asm/segment.h
71 signal.s signal.o : signal.c ../include/linux/sched.h ../include/linux/head.h \
72  ../include/linux/fs.h ../include/sys/types.h ../include/linux/mm.h \
73  ../include/signal.h ../include/linux/kernel.h ../include/asm/segment.h
74 sys.s sys.o : sys.c ../include/errno.h ../include/linux/sched.h \
75  ../include/linux/head.h ../include/linux/fs.h ../include/sys/types.h \
76  ../include/linux/mm.h ../include/signal.h ../include/linux/tty.h \
77  ../include/termios.h ../include/linux/kernel.h ../include/asm/segment.h \
78  ../include/sys/times.h ../include/sys/utsname.h
79 traps.s traps.o : traps.c ../include/string.h ../include/linux/head.h \
80  ../include/linux/sched.h ../include/linux/fs.h ../include/sys/types.h \
81  ../include/linux/mm.h ../include/signal.h ../include/linux/kernel.h \
82  ../include/asm/system.h ../include/asm/segment.h ../include/asm/io.h
83 vsprintf.s vsprintf.o : vsprintf.c ../include/stdarg.h ../include/string.h

```

---

## 8.3 asm.s 程序

### 8.3.1 功能描述

asm.s 汇编程序中包括大部分 CPU 探测到的异常故障处理的底层代码，也包括数学协处理器（FPU）的异常处理。该程序与 kernel/traps.c 程序有着密切的关系。该程序的主要处理方式是在中断处理程序中调用 traps.c 中相应的 C 函数程序，显示出错位置和出错号，然后退出中断。

在阅读这段代码时参照图 8-4 中当前任务的内核堆栈变化示意图将是很有帮助的，图中每个行代表 4 个字节。对于不带出错号的中断过程，堆栈指针位置变化情况请参照图 8-4(a)。在开始执行相应中断服务程序之前，堆栈指针 esp 指在中断返回地址一栏(图中 esp0 处)。当把将要调用的 C 函数 do\_divide\_error() 或其他 C 函数地址入栈后，指针位置是 esp1 处，此时程序使用交换指令把该函数的地址放入 eax 寄存器中，而原来 eax 的值则被保存到堆栈上。此后程序在把一些寄存器入栈后，堆栈指针位置处于 esp2 处。当正式调用 do\_divide\_error() 之前，程序会将开始执行中断程序时的原 eip 保存位置（即堆栈指针 esp0 值）压入堆栈，放到 esp3 位置处，并在中断返回弹出入栈的寄存器之前指针通过加上 8 又回到 esp2 处。

对于 CPU 会产生错误号的中断过程，堆栈指针位置变化情况请参照图 8-4(b)。在刚开始执行中断服务程序之前，堆栈指针指向图中 esp0 处。在把将要调用的 C 函数 do\_double\_fault() 或其他 C 函数地址入栈后，栈指针位置是 esp1 处。此时程序通过使用两个交换指令分别把 eax、ebx 寄存器的值保存在 esp0、

esp1 位置处，而把出错号交换到 `eax` 寄存器中；函数地址交换到了 `ebx` 寄存器中。随后的处理过程则和上述图 8-4(a)中的一样。

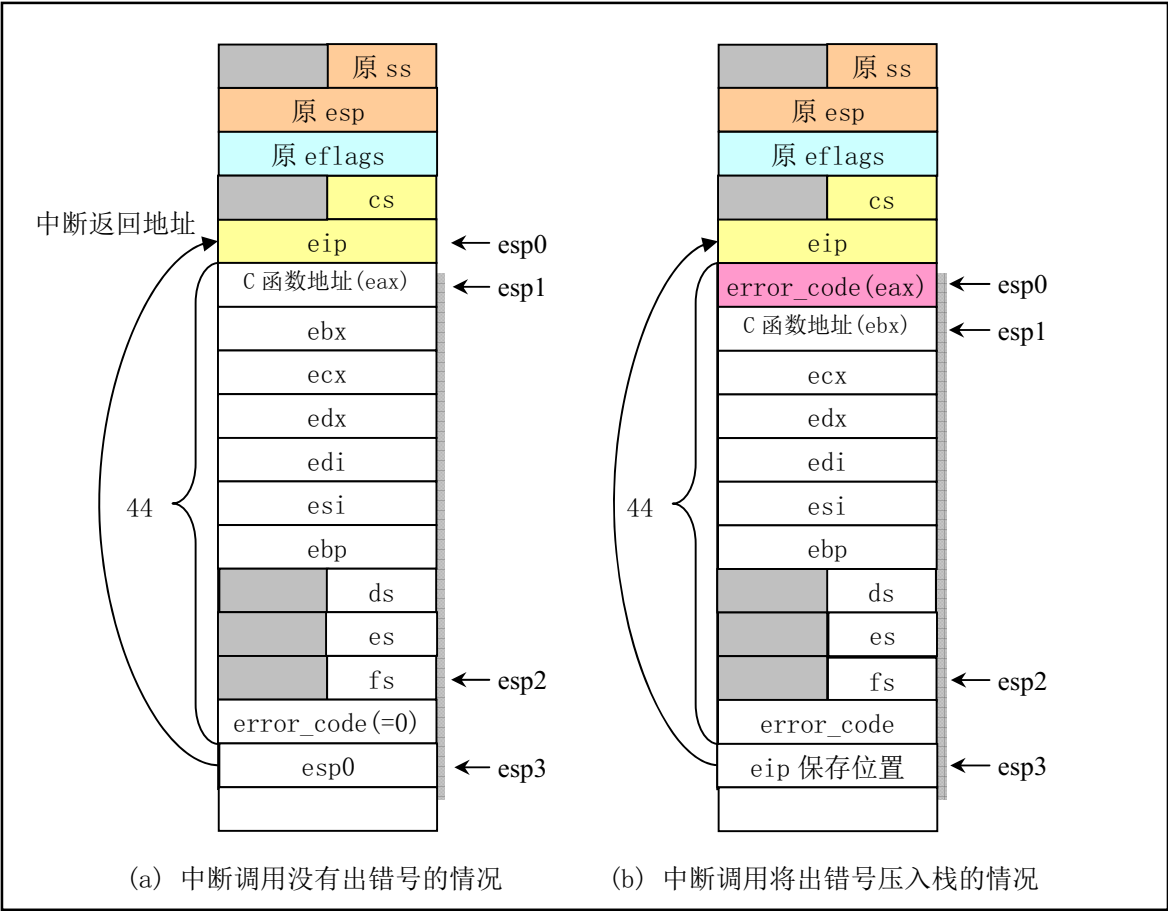


图 8-4 出错处理堆栈变化示意图

正式调用 `do_divide_error()` 之前把出错代码以及 `esp0` 入栈的原因是为了把出错代码和 `esp0` 作为调用 C 函数 `do_divide_error()` 的参数。在 `traps.c` 中，该函数的原形为：

```
void do_divide_error(long esp, long error_code)。
```

因此在这个 C 函数中就可以打印出出错的位置和错误号。程序中其余异常出错的处理过程与这里描述的过程基本类似。

8.3.2 代码注释

程序 8-2 linux/kernel/asm.s

```
1 /*  
2  * linux/kernel/asm.s  
3  *  
4  * (C) 1991 Linus Torvalds  
5  */  
6
```

```

7  /*
8  * asm.s contains the low-level code for most hardware faults.
9  * page_exception is handled by the mm, so that isn't here. This
10 * file also handles (hopefully) fpu-exceptions due to TS-bit, as
11 * the fpu must be properly saved/resored. This hasn't been tested.
12 */
/*
* asm.s 程序中包括大部分的硬件故障（或出错）处理的底层次代码。页异常由内存管理程序
* mm 处理，所以不在这里。此程序还处理（希望是这样）由于 TS-位而造成的 fpu 异常，因为
* fpu 必须正确地进行保存/恢复处理，这些还没有测试过。
*/
13
# 本代码文件主要涉及对 Intel 保留中断 int0--int16 的处理（int17-int31 留作今后使用）。
# 以下是一些全局函数名的声明，其原形在 traps.c 中说明。
14 .globl _divide_error, _debug, nmi, _int3, _overflow, _bounds, _invalid_op
15 .globl _double_fault, _coprocessor_segment_overrun
16 .globl _invalid_TSS, _segment_not_present, _stack_segment
17 .globl _general_protection, _coprocessor_error, _irq13, _reserved
18
# 下面这段程序处理无出错号的情况。参见图 8-4(a)。
# int0 -- 处理被零除出错的情况。类型：错误（Fault）；错误号：无。
# 在执行 DIV 或 IDIV 指令时，若除数是 0，CPU 就会产生这个异常。当 EAX（或 AX、AL）容纳
# 不了一个合法除操作的结果时也会产生这个异常。20 行上标号'_do_divide_error'实际上是
# C 语言函数 do_divide_error() 编译后所生成模块中对应的名称。函数'do_divide_error'在
# traps.c 中实现（第 97 行开始）。
19 _divide_error:
20     pushl $_do_divide_error # 首先把将要调用的函数地址入栈。
21 no_error_code:             # 这里是无出错号处理的入口处，见下面第 55 行等。
22     xchgl %eax, (%esp)      # _do_divide_error 的地址 → eax, eax 被交换入栈。
23     pushl %ebx
24     pushl %ecx
25     pushl %edx
26     pushl %edi
27     pushl %esi
28     pushl %ebp
29     push %ds                # !! 16 位的段寄存器入栈后也要占用 4 个字节。
30     push %es
31     push %fs
32     pushl $0                # "error code" # 将数值 0 作为出错码入栈。
33     lea 44(%esp), %edx      # 取堆栈中原调用返回地址处堆栈指针位置，并压入堆栈。
34     pushl %edx
35     movl $0x10, %edx        # 初始化段寄存器 ds、es 和 fs，加载内核数据段选择符。
36     mov %dx, %ds
37     mov %dx, %es
38     mov %dx, %fs
# 下行上的'*'号表示调用操作数指定地址处的函数，称为间接调用。这句的含义是调用引起本次
# 异常的 C 处理函数，例如 do_divide_error() 等。第 40 行是将堆栈指针加 8 相当于执行两次 pop
# 操作，弹出（丢弃）最后入堆栈的两个 C 函数参数（32 行和 34 行入栈的值），让堆栈指针重新
# 指向寄存器 fs 入栈处。
39     call *%eax
40     addl $8, %esp
41     pop %fs
42     pop %es

```

```

43     pop %ds
44     popl %ebp
45     popl %esi
46     popl %edi
47     popl %edx
48     popl %ecx
49     popl %ebx
50     popl %eax           # 弹出原来 eax 中的内容。
51     iret
52
# int1 -- debug 调试中断入口点。处理过程同上。类型：错误/陷阱 (Fault/Trap)；错误号：无。
# 当 EFLAGS 中 TF 标志置位时而引发的中断。当发现硬件断点（数据：陷阱，代码：错误）；或者
# 开启了指令跟踪陷阱或任务交换陷阱，或者调试寄存器访问无效（错误），CPU 就会产生该异常。
53 _debug:
54     pushl $_do_int3      # _do_debug C 函数指针入栈。以下同。
55     jmp no_error_code
56
# int2 -- 非屏蔽中断调用入口点。 类型：陷阱；无错误号。
# 这是仅有的被赋予固定中断向量的硬件中断。每当接收到一个 NMI 信号，CPU 内部就会产生中断
# 向量 2，并执行标准中断应答周期，因此很节省时间。NMI 通常保留为极为重要的硬件事件使用。
# 当 CPU 收到一个 NMI 信号并且开始执行其中断处理过程时，随后所有的硬件中断都将被忽略。
57 _nmi:
58     pushl $_do_nmi
59     jmp no_error_code
60
# int3 -- 断点指令引起中断的入口点。 类型：陷阱；无错误号。
# 由 int 3 指令引发的中断，与硬件中断无关。该指令通常由调式器插入被调式程序的代码中。
# 处理过程同_debug。
61 _int3:
62     pushl $_do_int3
63     jmp no_error_code
64
# int4 -- 溢出出错处理中断入口点。 类型：陷阱；无错误号。
# EFLAGS 中 OF 标志置位时 CPU 执行 INTO 指令就会引发该中断。通常用于编译器跟踪算术计算溢出。
65 _overflow:
66     pushl $_do_overflow
67     jmp no_error_code
68
# int5 -- 边界检查出错中断入口点。 类型：错误；无错误号。
# 当操作数在有效范围以外时引发的中断。当 BOUND 指令测试失败就会产生该中断。BOUND 指令有
# 3 个操作数，如果第 1 个不在另外两个之间，就产生异常 5。
69 _bounds:
70     pushl $_do_bounds
71     jmp no_error_code
72
# int6 -- 无效操作指令出错中断入口点。 类型：错误；无错误号。
# CPU 执行机构检测到一个无效的操作码而引起的中断。
73 _invalid_op:
74     pushl $_do_invalid_op
75     jmp no_error_code
76
# int9 -- 协处理器段超出出错中断入口点。 类型：放弃；无错误号。
# 该异常基本上等同于协处理器出错保护。因为在浮点指令操作数太大时，我们就有机会来

```



```

# 加载或保存超出数据段的浮点值。
77 _coprocessor_segment_overrun:
78     pushl $_do_coprocessor_segment_overrun
79     jmp no_error_code
80
# int15 -- 其他 Intel 保留中断的入口点。
81 _reserved:
82     pushl $_do_reserved
83     jmp no_error_code
84
# int45 -- (= 0x20 + 13) Linux 设置的数学协处理器硬件中断。
# 当协处理器执行完一个操作时就会发出 IRQ13 中断信号，以通知 CPU 操作完成。80387 在执行
# 计算时，CPU 会等待其操作完成。下面 88 行上 0xF0 是协处理端口，用于清忙锁存器。通过写
# 该端口，本中断将消除 CPU 的 BUSY 延续信号，并重新激活 80387 的处理器扩展请求引脚 PEREQ。
# 该操作主要是为了确保在继续执行 80387 的任何指令之前，CPU 响应本中断。
85 _irq13:
86     pushl %eax
87     xorb %al,%al
88     outb %al,$0xF0
89     movb $0x20,%al
90     outb %al,$0x20      # 向 8259 主中断控制芯片发送 EOI（中断结束）信号。
91     jmp 1f              # 这两个跳转指令起延时作用。
92 1:     jmp 1f
93 1:     outb %al,$0xA0    # 再向 8259 从中断控制芯片发送 EOI（中断结束）信号。
94     popl %eax
95     jmp _coprocessor_error # coprocessor_error 原在本程序中，现已放到 system_call.s 中。
96
# 以下中断在调用时 CPU 会在中断返回地址之后将出错号压入堆栈，因此返回时也需要将出错号
# 弹出（参见图 5.3(b)）。

# int8 -- 双出错故障。 类型：放弃；有错误码。
# 通常当 CPU 在调用前一个异常的处理程序而又检测到一个新的异常时，这两个异常会被串行地进行
# 处理，但也会碰到很少的情况，CPU 不能进行这样的串行处理操作，此时就会引发该中断。
97 _double_fault:
98     pushl $_do_double_fault # C 函数地址入栈。
99 error_code:
100     xchgl %eax,4(%esp)      # error code <-> %eax, eax 原来的值被保存在堆栈上。
101     xchgl %ebx,4(%esp)      # &function <-> %ebx, ebx 原来的值被保存在堆栈上。
102     pushl %ecx
103     pushl %edx
104     pushl %edi
105     pushl %esi
106     pushl %ebp
107     push %ds
108     push %es
109     push %fs
110     pushl %eax              # error code # 出错号入栈。
111     lea 44(%esp),%eax       # offset # 程序返回地址处堆栈指针位置值入栈。
112     pushl %eax
113     movl $0x10,%eax         # 置内核数据段选择符。
114     mov %ax,%ds
115     mov %ax,%es
116     mov %ax,%fs

```

```

117      call *%ebx                # 间接调用，调用相应的 C 函数，其参数已入栈。
118      addl $8,%esp              # 丢弃入栈的 2 个用作 C 函数的参数。
119      pop %fs
120      pop %es
121      pop %ds
122      popl %ebp
123      popl %esi
124      popl %edi
125      popl %edx
126      popl %ecx
127      popl %ebx
128      popl %eax
129      iret
130
# int10 -- 无效的任务状态段(TSS)。 类型：错误；有出错码。
# CPU 企图切换到一个进程，而该进程的 TSS 无效。根据 TSS 中哪一部分引起了异常，当由于 TSS
# 长度超过 104 字节时，这个异常在当前任务中产生，因而切换被终止。其他问题则会导致在切换
# 后的新任务中产生本异常。
131 _invalid_TSS:
132      pushl $_do_invalid_TSS
133      jmp error_code
134
# int11 -- 段不存在。 类型：错误；有出错码。
# 被引用的段不在内存中。段描述符中标志着段不在内存中。
135 _segment_not_present:
136      pushl $_do_segment_not_present
137      jmp error_code
138
# int12 -- 堆栈段错误。 类型：错误；有出错码。
# 指令操作试图超出堆栈段范围，或者堆栈段不在内存中。这是异常 11 和 13 的特例。有些操作
# 系统可以利用这个异常来确定什么时候应该为程序分配更多的栈空间。
139 _stack_segment:
140      pushl $_do_stack_segment
141      jmp error_code
142
# int13 -- 一般保护性出错。 类型：错误；有出错码。
# 表明是不属于任何其他类的错误。若一个异常产生时没有对应的处理向量（0—16），通常就
# 会归到此类。
143 _general_protection:
144      pushl $_do_general_protection
145      jmp error_code
146
# int7 -- 设备不存在（_device_not_available）在 kernel/system_call.s, 148 行。
# int14 -- 页错误（_page_fault）在 mm/page.s, 14 行。
# int16 -- 协处理器错误（_coprocessor_error）在 kernel/system_call.s, 131 行。
# 时钟中断 int 0x20（_timer_interrupt）在 kernel/system_call.s, 176 行。
# 系统调用 int 0x80（_system_call）在 kernel/system_call.s, 80 行。

```

### 8.3.3 其他信息

#### 8.3.3.1 Intel 保留中断向量的定义

这里给出了 Intel 保留中断向量具体含义的说明，见表 8-1 所示。

表 8 - 1 Intel 保留的中断号含义

中断号	名称	类型	信号	说明
0	Devide error	故障	SIGFPE	当进行除以零的操作时产生。
1	Debug	陷阱 故障	SIGTRAP	当进行程序单步跟踪调试时，设置了标志寄存器 eflags 的 T 标志时产生这个中断。
2	nmi	硬件		由不可屏蔽中断 NMI 产生。
3	Breakpoint	陷阱	SIGTRAP	由断点指令 int3 产生，与 debug 处理相同。
4	Overflow	陷阱	SIGSEGV	eflags 的溢出标志 OF 引起。
5	Bounds check	故障	SIGSEGV	寻址到有效地址以外时引起。
6	Invalid Opcode	故障	SIGILL	CPU 执行时发现一个无效的指令操作码。
7	Device not available	故障	SIGSEGV	设备不存在，指协处理器。在两种情况下会产生该中断：(a)CPU 遇到一个转意指令并且 EM 置位时。在这种情况下处理程序应该模拟导致异常的指令。(b)MP 和 TS 都在置位状态时，CPU 遇到 WAIT 或一个转意指令。在这种情况下，处理程序在必要时应该更新协处理器的状态。
8	Double fault	异常中止	SIGSEGV	双故障出错。
9	Coprocessor segment overrun	异常中止	SIGFPE	协处理器段超出。
10	Invalid TSS	故障	SIGSEGV	CPU 切换时发觉 TSS 无效。
11	Segment not present	故障	SIGBUS	描述符所指的段不存在。
12	Stack segment	故障	SIGBUS	堆栈段不存在或寻址越出堆栈段。
13	General protection	故障	SIGSEGV	没有符合 80386 保护机制（特权级）的操作引起。
14	Page fault	故障	SIGSEGV	页不在内存。
15	Reserved			
16	Coprocessor error	故障	SIGFPE	协处理器发出的出错信号引起。

## 8.4 traps.c 程序

### 8.4.1 功能描述

traps.c 程序主要包括一些在处理异常故障（硬件中断）底层代码 asm.s 文件中调用的相应 C 函数。用于显示出错位置和出错号等调试信息。其中的 die()通用函数用于在中断处理中显示详细的出错信息，而代码最后的初始化函数 trap\_init()是在前面 init/main.c 中被调用,用于初始化硬件异常处理中断向量(陷阱门)，并设置允许中断请求信号的到来。在阅读本程序时需要参考 asm.s 程序。

从本程序开始，我们会遇到很多 C 语言程序中嵌入的汇编语句。有关嵌入式汇编语句的基本语法请见本程序列表后的说明。

### 8.4.2 代码注释

程序 8-3 linux/kernel/traps.c

```
1 /*
2  * linux/kernel/traps.c
```

```

3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*
8  * 'Traps.c' handles hardware traps and faults after we have saved some
9  * state in 'asm.s'. Currently mostly a debugging-aid, will be extended
10 * to mainly kill the offending process (probably by giving it a signal,
11 * but possibly by killing it outright if necessary).
12 */
13 /*
14  * 在程序 asm.s 中保存了一些状态后，本程序用来处理硬件陷阱和故障。目前主要用于调试目的，
15  * 以后将扩展用来杀死遭损坏的进程（主要是通过发送一个信号，但如果必要也会直接杀死）。
16 */
17 #include <string.h>          // 字符串头文件。主要定义了一些有关内存或字符串操作的嵌入函数。
18 #include <linux/head.h>      // head 头文件，定义了段描述符的简单结构，和几个选择符常量。
19 #include <linux/sched.h>     // 调度程序头文件，定义了任务结构 task_struct、初始任务 0 的数据，
20 // 还有一些有关描述符参数设置和获取的嵌入式汇编函数宏语句。
21 #include <linux/kernel.h>    // 内核头文件。含有一些内核常用函数的原形定义。
22 #include <asm/system.h>      // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
23 #include <asm/segment.h>     // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
24 #include <asm/io.h>          // 输入/输出头文件。定义硬件端口输入/输出宏汇编语句。
25
26 // 以下语句定义了三个嵌入式汇编宏语句函数。有关嵌入式汇编的基本语法见本程序列表后的说明。
27 // 用圆括号括住的组合语句（花括号中的语句）可以作为表达式使用，其中最后的__res 是其输出值。
28 // 第 23 行定义了一个寄存器变量 __res。该变量将被保存在一个寄存器中，以便于快速访问和操作。
29 // 如果想指定寄存器（例如 eax），那么我们可以把该句写成“register char __res asm("ax");”。
30 // 取段 seg 中地址 addr 处的一个字节。
31 // 参数：seg - 段选择符；addr - 段内指定地址。
32 // 输出：%0 - eax (__res)；输入：%1 - eax (seg)；%2 - 内存地址 (*(addr))。
33 #define get_seg_byte(seg, addr) ({ \
34     register char __res; \
35     __asm__ ("push %%fs;mov %%ax,%%fs;movb %%fs:%2,%%al;pop %%fs" \
36             : "=a" (__res): "0" (seg), "m" (*(addr))); \
37     __res;})
38
39 // 取段 seg 中地址 addr 处的一个长字（4 字节）。
40 // 参数：seg - 段选择符；addr - 段内指定地址。
41 // 输出：%0 - eax (__res)；输入：%1 - eax (seg)；%2 - 内存地址 (*(addr))。
42 #define get_seg_long(seg, addr) ({ \
43     register unsigned long __res; \
44     __asm__ ("push %%fs;mov %%ax,%%fs;movl %%fs:%2,%%eax;pop %%fs" \
45             : "=a" (__res): "0" (seg), "m" (*(addr))); \
46     __res;})
47
48 // 取 fs 段寄存器的值（选择符）。
49 // 输出：%0 - eax (__res)。
50 #define fs() ({ \
51     register unsigned short __res; \
52     __asm__ ("mov %%fs,%%ax": "=a" (__res):); \
53     __res;})
54
55 #define fs() ({ \
56     register unsigned short __res; \
57     __asm__ ("mov %%fs,%%ax": "=a" (__res):); \
58     __res;})

```

```

// 以下定义了一些函数原型。
39 int do_exit(long code); // 程序退出处理。(kernel/exit.c, 102)
40
41 void page_exception(void); // 页异常。实际是 page_fault (mm/page.s, 14)
42
// 以下定义了一些中断处理程序原型，用于在函数 trap_init() 中设置相应中断门描述符。
// 这些函数的代码在 (kernel/asm.s 或 system_call.s) 中。
43 void divide_error(void); // int0 (kernel/asm.s, 19)。
44 void debug(void); // int1 (kernel/asm.s, 53)。
45 void nmi(void); // int2 (kernel/asm.s, 57)。
46 void int3(void); // int3 (kernel/asm.s, 61)。
47 void overflow(void); // int4 (kernel/asm.s, 65)。
48 void bounds(void); // int5 (kernel/asm.s, 69)。
49 void invalid_op(void); // int6 (kernel/asm.s, 73)。
50 void device_not_available(void); // int7 (kernel/system_call.s, 148)。
51 void double_fault(void); // int8 (kernel/asm.s, 97)。
52 void coprocessor_segment_overrun(void); // int9 (kernel/asm.s, 77)。
53 void invalid_TSS(void); // int10 (kernel/asm.s, 131)。
54 void segment_not_present(void); // int11 (kernel/asm.s, 135)。
55 void stack_segment(void); // int12 (kernel/asm.s, 139)。
56 void general_protection(void); // int13 (kernel/asm.s, 143)。
57 void page_fault(void); // int14 (mm/page.s, 14)。
58 void coprocessor_error(void); // int16 (kernel/system_call.s, 131)。
59 void reserved(void); // int15 (kernel/asm.s, 81)。
60 void parallel_interrupt(void); // int39 (kernel/system_call.s, 280)。
61 void irq13(void); // int45 协处理器中断处理(kernel/asm.s, 85)。
62

// 该子程序用来打印出错中断的名称、出错号、调用程序的 EIP、EFLAGS、ESP、fs 段寄存器值、
// 段的基址、段的长度、进程号 pid、任务号、10 字节指令码。如果堆栈在用户数据段，则还
// 打印 16 字节的堆栈内容。
63 static void die(char * str, long esp_ptr, long nr)
64 {
65     long * esp = (long *) esp_ptr;
66     int i;
67
68     printk("%s: %04x\n|r", str, nr&0xffff);
// 下行打印语句显示当前调用进程的 CS:EIP、EFLAGS 和 SS:ESP 的值。参照图 8-4，这里 esp[0]
// 即为图中的 esp0 位置。因此我们把这句拆分开来看为：
// (1) EIP:\t%04x:%p\n -- esp[1]是段选择符(cs)，esp[0]是 eip
// (2) EFLAGS:\t%p -- esp[2]是 eflags
// (3) ESP:\t%04x:%p\n -- esp[4]是原 ss，esp[3]是原 esp
69     printk("EIP: \t%04x:%p\nEFLAGS: \t%p\nESP: \t%04x:%p\n",
70           esp[1], esp[0], esp[2], esp[4], esp[3]);
71     printk("fs: %04x\n", fs());
72     printk("base: %p, limit: %p\n", get_base(current->ldt[1]), get_limit(0x17));
73     if (esp[4] == 0x17) {
74         printk("Stack: ");
75         for (i=0; i<4; i++)
76             printk("%p ", get_seg_long(0x17, i+(long *)esp[3]));
77         printk("\n");
78     }
79     str(i); // 取当前运行任务的任务号 (include/linux/sched.h, 159)。
80     printk("Pid: %d, process nr: %d\n|r", current->pid, 0xffff & i);

```

```

81     for(i=0;i<10;i++)
82         printk("%02x ",0xff & get_seg_byte(esp[1], (i+(char *)esp[0])));
83     printk("\n\r");
84     do_exit(11);          /* play segment exception */
85 }
86
87 // 以下这些以 do_开头的函数是 asm.s 中对应中断处理程序调用的 C 函数。
88 void do_double_fault(long esp, long error_code)
89 {
90     die("double fault", esp, error_code);
91 }
92 void do_general_protection(long esp, long error_code)
93 {
94     die("general protection", esp, error_code);
95 }
96
97 void do_divide_error(long esp, long error_code)
98 {
99     die("divide error", esp, error_code);
100 }
101
102 // 参数是进入中断后被顺序压入堆栈的寄存器值。参见 asm.s 程序第 22--34 行。
103 void do_int3(long * esp, long error_code,
104             long fs, long es, long ds,
105             long ebp, long esi, long edi,
106             long edx, long ecx, long ebx, long eax)
107 {
108     int tr;
109     __asm__("str %%ax": "=a" (tr): "0" (0));          // 取任务寄存器值→tr。
110     printk("eax|t|tebx|t|tecx|t|tedx|n|r%8x|t%8x|t%8x|t%8x|n|r",
111           eax, ebx, ecx, edx);
112     printk("esi|t|tedi|t|tebp|t|tesp|n|r%8x|t%8x|t%8x|t%8x|n|r",
113           esi, edi, ebp, (long) esp);
114     printk("\n|rds|tes|tfs|ttr|n|r%4x|t%4x|t%4x|t%4x|n|r",
115           ds, es, fs, tr);
116     printk("EIP: %8x   CS: %4x   EFLAGS: %8x|n|r", esp[0], esp[1], esp[2]);
117 }
118
119 void do_nmi(long esp, long error_code)
120 {
121     die("nmi", esp, error_code);
122 }
123
124 void do_debug(long esp, long error_code)
125 {
126     die("debug", esp, error_code);
127 }
128
129 void do_overflow(long esp, long error_code)
130 {
131     die("overflow", esp, error_code);

```

```

132 }
133
134 void do\_bounds(long esp, long error_code)
135 {
136     die("bounds", esp, error_code);
137 }
138
139 void do\_invalid\_op(long esp, long error_code)
140 {
141     die("invalid operand", esp, error_code);
142 }
143
144 void do\_device\_not\_available(long esp, long error_code)
145 {
146     die("device not available", esp, error_code);
147 }
148
149 void do\_coprocessor\_segment\_overrun(long esp, long error_code)
150 {
151     die("coprocessor segment overrun", esp, error_code);
152 }
153
154 void do\_invalid\_TSS(long esp, long error_code)
155 {
156     die("invalid TSS", esp, error_code);
157 }
158
159 void do\_segment\_not\_present(long esp, long error_code)
160 {
161     die("segment not present", esp, error_code);
162 }
163
164 void do\_stack\_segment(long esp, long error_code)
165 {
166     die("stack segment", esp, error_code);
167 }
168
169 void do\_coprocessor\_error(long esp, long error_code)
170 {
171     if (last\_task\_used\_math != current)
172         return;
173     die("coprocessor error", esp, error_code);
174 }
175
176 void do\_reserved(long esp, long error_code)
177 {
178     die("reserved (15, 17-47) error", esp, error_code);
179 }
180
// 下面是异常（陷阱）中断程序初始化子程序。设置它们的中断调用门（中断向量）。
// set_trap_gate()与 set_system_gate()都使用了中断描述符表 IDT 中的陷阱门（Trap Gate），
// 它们之间的主要区别在于前者设置的特权级为 0，后者是 3。因此断点陷阱中断 int3、溢出中断
// overflow 和边界出错中断 bounds 可以由任何程序产生。

```



---

```

// 这两个函数均是嵌入式汇编宏程序(include/asm/system.h, 第 36 行、39 行)。
181 void trap_init(void)
182 {
183     int i;
184
185     set_trap_gate(0, &divide_error); // 设置除操作出错的中断向量值。以下雷同。
186     set_trap_gate(1, &debug);
187     set_trap_gate(2, &nmi);
188     set_system_gate(3, &int3);      /* int3-5 can be called from all */
189     set_system_gate(4, &overflow);  /* int3-5 可以被所有程序执行 */
190     set_system_gate(5, &bounds);
191     set_trap_gate(6, &invalid_op);
192     set_trap_gate(7, &device_not_available);
193     set_trap_gate(8, &double_fault);
194     set_trap_gate(9, &coprocessor_segment_overrun);
195     set_trap_gate(10, &invalid_TSS);
196     set_trap_gate(11, &segment_not_present);
197     set_trap_gate(12, &stack_segment);
198     set_trap_gate(13, &general_protection);
199     set_trap_gate(14, &page_fault);
200     set_trap_gate(15, &reserved);
201     set_trap_gate(16, &coprocessor_error);
    // 下面把 int17-47 的陷阱门先均设置为 reserved, 以后各硬件初始化时会重新设置自己的陷阱门。
202     for (i=17; i<48; i++)
203         set_trap_gate(i, &reserved);
    // 设置协处理器中断 0x2d (45) 陷阱门描述符, 并允许其产生中断请求。设置并行口中断描述符。
204     set_trap_gate(45, &irq13);
205     outb_p(inb_p(0x21)&0xfb, 0x21); // 允许 8259A 主芯片的 IRQ2 中断请求。
206     outb_p(inb_p(0xA1)&0xdf, 0xA1); // 允许 8259A 从芯片的 IRQ13 中断请求。
207     set_trap_gate(39, &parallel_interrupt); // 设置并行口 1 的中断 0x27 陷阱门描述符。
208 }
209

```

---

## 8.5 system\_call.s 程序

### 8.5.1 功能描述

在 Linux 0.11 中, 用户使用中断调用 int 0x80 和放在寄存器 eax 中的功能号来使用内核提供的各种功能服务, 这些操作系统提供的功能被称之为系统调用功能。通常用户并不是直接使用系统调用中断, 而是通过函数库 (例如 libc) 中提供的接口函数来调用的。例如创建进程的系统调用 fork 可直接使用函数 fork() 即可。函数库 libc 中的 fork() 函数会实现对中断 int 0x80 的调用过程并把调用结果返回给用户程序。

对于所有系统调用的实现函数, 内核把它们按照系统调用功能号顺序排列成一张函数指针 (地址) 表 (在 include/linux/sys.h 文件中)。然后在中断 int 0x80 的处理过程中根据用户提供的功能号调用对应系统调用函数进行处理。

本程序主要实现系统调用(system\_call)中断 int 0x80 的入口处理过程以及信号检测处理 (从代码第 80 行开始), 同时给出了两个系统功能的底层接口, 分别是 sys\_execve 和 sys\_fork。还列出了处理过程类似

的协处理器出错(int 16)、设备不存在(int7)、时钟中断(int32)、硬盘中断(int46)、软盘中断(int38)的中断处理程序。

对于软中断(system\_call、coprocessor\_error、device\_not\_available)，其处理过程基本上是首先为调用相应 C 函数处理程序作准备，将一些参数压入堆栈。系统调用最多可以带 3 个参数，分别通过寄存器 ebx、ecx 和 edx 传入。然后调用 C 函数进行相应功能的处理，处理返回后再去检测当前任务的信号位图，对值最小的一个信号进行处理并复位信号位图中的该信号。系统调用的 C 语言处理函数分布在整个 linux 内核代码中，由 include/linux/sys.h 头文件中的系统函数指针数组表来匹配。

对于硬件中断请求信号 IRQ 发来的中断，其处理过程首先是向中断控制芯片 8259A 发送结束硬件中断控制字指令 EOI，然后调用相应的 C 函数处理程序。对于时钟中断也要对当前任务的信号位图进行检测处理。

对于系统调用(int 0x80)的中断处理过程，可以把它看作是一个“接口”程序。实际上每个系统调用功能的处理过程基本上都是通过调用相应的 C 函数进行的。即所谓的“Bottom half”函数。

这个程序在刚进入时会首先检查 eax 中的功能号是否有效（在给定的范围内），然后保存一些会用到的寄存器到堆栈上。Linux 内核默认地把段寄存器 ds,es 用于内核数据段，而 fs 用于用户数据段。接着通过一个地址跳转表（sys\_call\_table）调用相应系统调用的 C 函数。在 C 函数返回后，程序就把返回值压入堆栈保存起来。

接下来，该程序查看执行本次调用进程的状态。如果由于上面 C 函数的操作或其他情况而使进程的状态从执行态变成了其他状态，或者由于时间片已经用完（counter==0），则调用进程调度函数 schedule()（jmp\_schedule）。由于在执行“jmp\_schedule”之前已经把返回地址 ret\_from\_sys\_call 入栈，因此在执行完 schedule()后最终会返回到 ret\_from\_sys\_call 处继续执行。

从 ret\_from\_sys\_call 标号处开始的代码执行一些系统调用的后处理工作。主要判断当前进程是否是初始进程 0，如果是就直接退出此次系统调用，中断返回。否则再根据代码段描述符和所使用的堆栈来判断本次系统调用的进程是否是一个普通进程，若不是则说明是内核进程（例如初始进程 1）或其他。则也立刻弹出堆栈内容退出系统调用中断。末端的一块代码用来处理调用系统调用进程的信号。若进程结构的信号位图表明该进程有接收到信号，则调用信号处理函数 do\_signal()。

最后，该程序恢复保存的寄存器内容，退出此次中断处理过程并返回调用程序。若有信号时则程序会首先“返回”到相应信号处理函数中去执行，然后返回调用 system\_call 的程序。

系统调用处理过程的整个流程见图 8-5 所示。

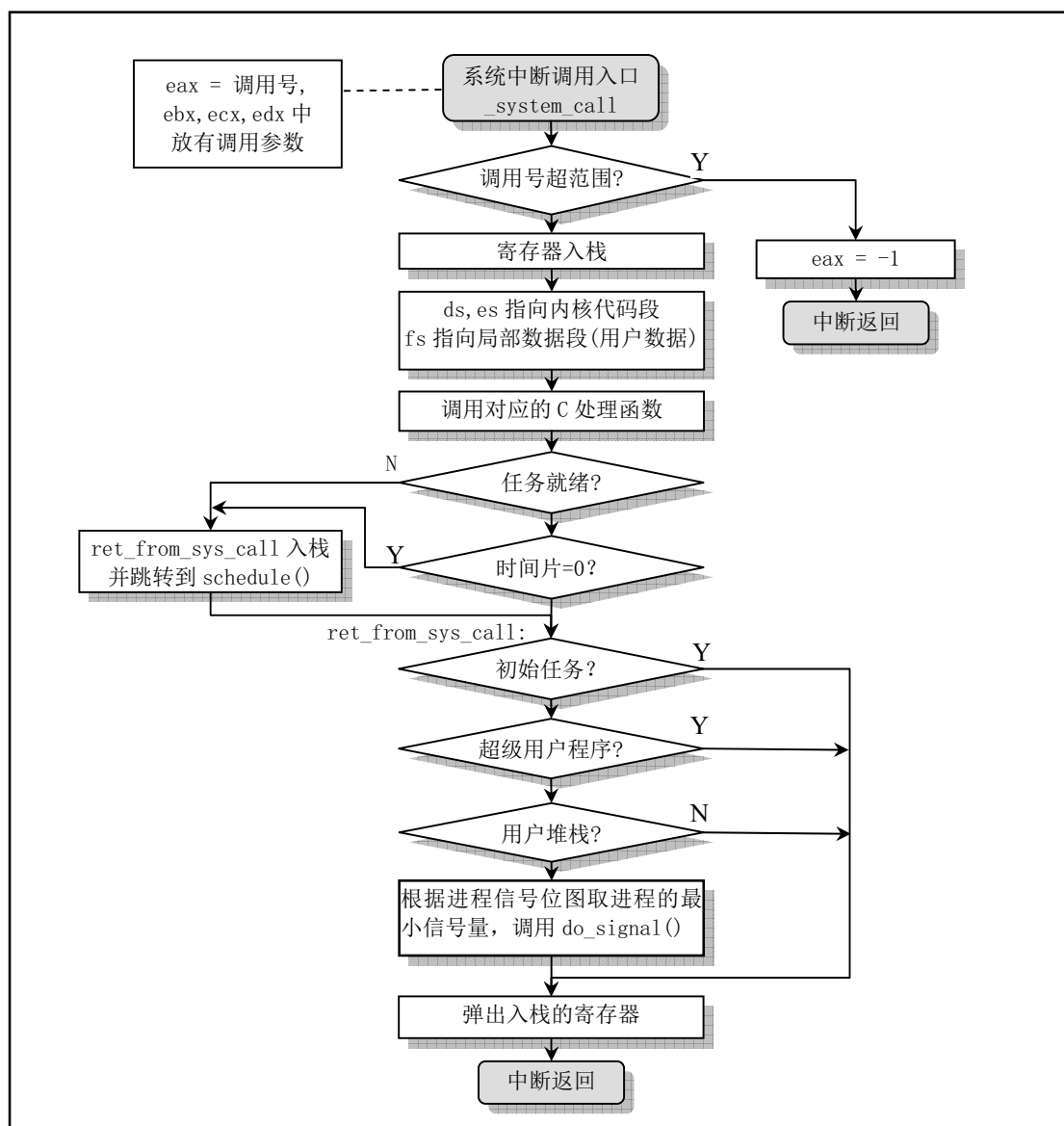


图 8-5 系统中断调用处理流程

关于系统调用 `int 0x80` 中的参数传递问题，Linux 内核使用了几个通用寄存器作为参数传递的渠道。在 Linux 0.11 系统中，程序使用寄存器 `ebx`、`ecx` 和 `edx` 传递参数，可以直接向系统调用服务过程传递 3 个参数（不包括放在 `eax` 寄存器中的系统调用号）。若使用指向用户空间数据块的指针，则用户程序可以向系统调用过程传递更多的数据信息。

如上所述，在系统调用运行过程中，段寄存器 `ds` 和 `es` 指向内核数据空间，而 `fs` 被设置为指向用户数据空间。因此在实际数据块信息传递过程中 Linux 内核就可以利用 `fs` 寄存器来执行内核数据空间与用户数据空间之间的数据复制工作，并且在复制过程中内核程序不需要对数据边界范围作任何检查操作。边界检查工作会由 CPU 自动完成。内核程序中的实际数据传递工作可以使用 `get_fs_byte()` 和 `put_fs_byte()` 等函数来进行，参见 `include/asm/segment.h` 文件中这些函数的实现代码。

这种使用寄存器传递参数的方法具有一个明显的优点，那就是当进入系统中断服务程序而保存寄存器值时，这些传递参数的寄存器也被自动地放在了内核态堆栈上。而当进程从中断调用中退出时就被弹出内核态堆栈，因此内核不用对它们进行特殊处理。这种方法是 Linus 当时所知的最简单最快速的参数传递方法。

## 8.5.2 代码注释

程序 8-4 linux/kernel/system\_call.s

```

1  /*
2  *  linux/kernel/system_call.s
3  *
4  *  (C) 1991  Linus Torvalds
5  */
6
7  /*
8  *  system_call.s  contains the system-call low-level handling routines.
9  *  This also contains the timer-interrupt handler, as some of the code is
10 *  the same. The hd- and floppy-interrupts are also here.
11 *
12 *  NOTE: This code handles signal-recognition, which happens every time
13 *  after a timer-interrupt and after each system call. Ordinary interrupts
14 *  don't handle signal-recognition, as that would clutter them up totally
15 *  unnecessarily.
16 *
17 *  Stack layout in 'ret_from_system_call':
18 *
19 *      0(%esp) - %eax
20 *      4(%esp) - %ebx
21 *      8(%esp) - %ecx
22 *      C(%esp) - %edx
23 *     10(%esp) - %fs
24 *     14(%esp) - %es
25 *     18(%esp) - %ds
26 *     1C(%esp) - %eip
27 *     20(%esp) - %cs
28 *     24(%esp) - %eflags
29 *     28(%esp) - %oldesp
30 *     2C(%esp) - %oldss
31 */
32 /*
33 *  system_call.s 文件包含系统调用(system-call)底层处理子程序。由于有些代码比较类似，所以
34 *  同时也包括时钟中断处理(timer-interrupt)句柄。硬盘和软盘的中断处理程序也在这里。
35 *
36 *  注意：这段代码处理信号(signal)识别，在每次时钟中断和系统调用之后都会进行识别。一般
37 *  中断过程并不处理信号识别，因为会给系统造成混乱。
38 *
39 *  从系统调用返回（'ret_from_system_call'）时堆栈的内容见上面 19-30 行。
40 */
41 # 上面 Linus 原注释中的一般中断过程是指除了系统调用中断（int 0x80）和时钟中断（int 0x20）
42 # 以外的其他中断。这些中断会在内核态或用户态随机发生，若在这些中断过程中也处理信号识别的
43 # 话，就有可能与系统调用中断和时钟中断过程中对信号的识别处理过程相冲突，，违反了内核代码
44 # 非抢占原则。因此系统既无必要在这些“其他”中断中处理信号，也不允许这样做。
45
46 32
47 33 SIG_CHLD          = 17          # 定义 SIG_CHLD 信号（子进程停止或结束）。
48 34
49 35 EAX                = 0x00        # 堆栈中各个寄存器的偏移位置。
50 36 EBX                = 0x04

```

```

37 ECX          = 0x08
38 EDX          = 0x0C
39 FS           = 0x10
40 ES           = 0x14
41 DS           = 0x18
42 EIP          = 0x1C
43 CS           = 0x20
44 EFLAGS       = 0x24
45 OLDESP       = 0x28      # 当特权级变化时栈会切换，用户栈指针被保存在内核态栈中。
46 OLDSS       = 0x2C
47
# 以下这些是任务结构(task_struct)中变量的偏移值，参见 include/linux/sched.h，77 行开始。
48 state      = 0          # these are offsets into the task_struct. # 进程状态码。
49 counter    = 4          # 任务运行时间计数(递减)(滴答数)，运行时间片。
50 priority   = 8          # 运行优先数。任务开始运行时 counter=priority，越大则运行时间越长。
51 signal     = 12         # 是信号位图，每个比特位代表一种信号，信号值=位偏移值+1。
52 sigaction  = 16         # MUST be 16 (=len of sigaction) // sigaction 结构长度必须是 16 字节。
# 信号执行属性结构数组的偏移值，对应信号将要执行的操作和标志信息。
53 blocked    = (33*16)    # 受阻塞信号位图的偏移量。
54
# 以下定义在 sigaction 结构中的偏移量，参见 include/signal.h，第 48 行开始。
55 # offsets within sigaction
56 sa_handler  = 0          # 信号处理过程的句柄(描述符)。
57 sa_mask     = 4          # 信号屏蔽码。
58 sa_flags    = 8          # 信号集。
59 sa_restorer = 12         # 恢复函数指针，参见 kernel/signal.c。
60
61 nr_system_calls = 72     # Linux 0.11 版内核中的系统调用总数。
62
63 /*
64  * Ok, I get parallel printer interrupts while using the floppy for some
65  * strange reason. Urgel. Now I just ignore them.
66  */
67 /*
68  * 好了，在使用软驱时我收到了并行打印机中断，很奇怪。哼，现在不管它。
69  */
70 # 英文注释中的“Urgel”是瑞典语，对应英文单词“Ugh”。这个单词在最新内核中还常出现。
71 # 定义入口点。
72 .globl _system_call, _sys_fork, _timer_interrupt, _sys_execve
73 .globl _hd_interrupt, _floppy_interrupt, _parallel_interrupt
74 .globl _device_not_available, _coprocessor_error
75
# 错误的系统调用号。
76 .align 2          # 内存 4 字节对齐。
77 bad_sys_call:
78     movl $-1,%eax  # eax 中置-1，退出中断。
79     iret
80
# 重新执行调度程序入口。调度程序 schedule 在(kernel/sched.c, 104)。
# 当调度程序 schedule() 返回时就从 ret_from_sys_call 处(101 行)继续执行。
81 .align 2
82 reschedule:
83     pushl $ret_from_sys_call # 将 ret_from_sys_call 的地址入栈(101 行)。
84     jmp _schedule

```

```

##### int 0x80 --linux 系统调用入口点(调用中断 int 0x80, eax 中是调用号)。
79 .align 2
80 _system_call:
81     cmpl $nr_system_calls-1,%eax    # 调用号如果超出范围的话就在 eax 中置-1 并退出。
82     ja bad_sys_call
83     push %ds                        # 保存原段寄存器值。
84     push %es
85     push %fs
# 一个系统调用最多可带有 3 个参数,也可以不带参数。下面入栈的 ebx、ecx 和 edx 中放着系统
# 调用相应 C 语言函数(见第 94 行)的调用参数。这几个寄存器入栈的顺序是由 GNU GCC 规定的,
# ebx 中可存放第 1 个参数,ecx 中存放第 2 个参数,edx 中存放第 3 个参数。
# 系统调用语句可参见头文件 include/unistd.h 中第 133 到 183 行的系统调用宏。
86     pushl %edx
87     pushl %ecx                    # push %ebx,%ecx,%edx as parameters
88     pushl %ebx                    # to the system call
89     movl $0x10,%edx              # set up ds,es to kernel space
90     mov %dx,%ds                  # ds,es 指向内核数据段(全局描述符表中数据段描述符)。
91     mov %dx,%es
# fs 指向局部数据段(局部描述符表中数据段描述符),即指向执行本次系统调用的用户程序的数据段。
# 注意,在 Linux 0.11 中内核给任务分配的代码和数据内存段是重叠的,它们的段基址和段限长相同。
# 参见 fork.c 程序中 copy_mem() 函数。
92     movl $0x17,%edx              # fs points to local data space
93     mov %dx,%fs
# 下面这句操作数的含义是:调用地址=[_sys_call_table + %eax * 4]。参见程序后的说明。
# sys_call_table[] 是一个指针数组,定义在 include/linux/sys.h 中。该指针数组中设置了
# 所有 72 个系统调用 C 处理函数的地址。
94     call _sys_call_table(,%eax,4) # 间接调用指定功能 C 函数。
95     pushl %eax                   # 把系统调用返回值入栈。
# 下面 96-100 行查看当前任务的运行状态。如果不在就绪状态(state 不等于 0)就去执行调度
# 程序。如果该任务在就绪状态,但其时间片已用完(counter = 0),则也去执行调度程序。
# 例如当后台进程组中的进程执行控制终端读写操作时,那么默认条件下该后台进程组所有进程
# 会收到 SIGTTIN 或 SIGTTOU 信号,导致进程组中所有进程处于停止状态。而当前进程则会立刻
# 返回。
96     movl _current,%eax           # 取当前任务(进程)数据结构地址→eax。
97     cmpl $0,state(%eax)          # state
98     jne reschedule
99     cmpl $0,counter(%eax)        # counter
100    je reschedule

# 以下这段代码执行从系统调用 C 函数返回后,对信号进行识别处理。其他中断服务程序退出时也
# 将跳转到这里进行处理后才退出中断过程,例如后面 131 行上的处理器出错中断 int 16。
101 ret_from_sys_call:
# 首先判别当前任务是否是初始任务 task0,如果是则不必对其进行信号量方面的处理,直接返回。
# 103 行上的 _task 对应 C 程序中的 task[] 数组,直接引用 task 相当于引用 task[0]。
102     movl _current,%eax           # task[0] cannot have signals
103     cmpl _task,%eax
104     je 3f                        # 向前(forward)跳转到标号 3 处退出中断处理。。
# 通过对原调用程序代码选择符的检查来判断调用程序是否是用户任务。如果不是则直接退出中断。
# 这是因为任务在内核态执行时不可抢占。否则对任务进行信号量的识别处理。这里比较选择符是否
# 为用户代码段的选择符 0x000f(RPL=3,局部表,第 1 个段(代码段))来判断是否为用户任务。如
# 果不是则说明是某个中断服务程序跳转到第 101 行的,于是跳转退出中断程序。如果原堆栈段选择
# 符不为 0x17(即原堆栈不在用户段中),也说明本次系统调用的调用者不是用户任务,则也退出。
105     cmpw $0x0f,CS(%esp)         # was old code segment supervisor ?

```

```

106     jne 3f
107     cmpw $0x17,OLDSS(%esp)    # was stack segment = 0x17 ?
108     jne 3f
# 下面这段代码（109-120）用于处理当前任务中的信号。首先取当前任务结构中的信号位图（32 位，
# 每位代表 1 种信号），然后用任务结构中的信号阻塞（屏蔽）码，阻塞不允许的信号位，取得数值
# 最小的信号值，再把原信号位图中该信号对应的位复位（置 0），最后将该信号值作为参数之一调
# 用 do_signal()。do_signal() 在（kernel/signal.c,82）中，其参数包括 13 个入栈的信息。
109     movl signal(%eax),%ebx    # 取信号位图→ebx，每 1 位代表 1 种信号，共 32 个信号。
110     movl blocked(%eax),%ecx   # 取阻塞（屏蔽）信号位图→ecx。
111     notl %ecx                # 每位取反。
112     andl %ebx,%ecx           # 获得许可的信号位图。
113     bsfl %ecx,%ecx           # 从低位（位 0）开始扫描位图，看是否有 1 的位，
                                # 若有，则 ecx 保留该位的偏移值（即第几位 0-31）。
114     je 3f                   # 如果没有信号则向前跳转退出。
115     btrl %ecx,%ebx           # 复位该信号（ebx 含有原 signal 位图）。
116     movl %ebx,signal(%eax)   # 重新保存 signal 位图信息→current->signal。
117     incl %ecx                # 将信号调整为从 1 开始的数（1-32）。
118     pushl %ecx               # 信号值入栈作为调用 do_signal 的参数之一。
119     call _do_signal          # 调用 C 函数信号处理程序(kernel/signal.c,82)
120     popl %eax                # 弹出栈的信号值。
121 3:    popl %eax               # eax 中含有第 95 行入栈的系统调用返回值。
122     popl %ebx
123     popl %ecx
124     popl %edx
125     pop %fs
126     pop %es
127     pop %ds
128     iret
129
#### int16 一处理器错误中断。 类型：错误；无错误码。
# 这是一个外部的基于硬件的异常。当协处理器检测到自己发生错误时，就会通过 ERROR 引脚
# 通知 CPU。下面代码用于处理协处理器发出的出错信号。并跳转去执行 C 函数 math_error()
# （kernel/math/ math_emulate.c,82）。返回后跳转到标号 ret_from_sys_call 处继续执行。
130 .align 2
131 _coprocessor_error:
132     push %ds
133     push %es
134     push %fs
135     pushl %edx
136     pushl %ecx
137     pushl %ebx
138     pushl %eax
139     movl $0x10,%eax          # ds, es 置为指向内核数据段。
140     mov %ax,%ds
141     mov %ax,%es
142     movl $0x17,%eax          # fs 置为指向局部数据段（出错程序的数据段）。
143     mov %ax,%fs
144     pushl $ret_from_sys_call # 把下面调用返回的地址入栈。
145     jmp _math_error          # 执行 C 函数 math_error() (math/math_emulate.c,37)
146
#### int7 一 设备不存在或协处理器不存在。 类型：错误；无错误码。
# 如果控制寄存器 CR0 中 EM（模拟）标志置位，则当 CPU 执行一个协处理器指令时就会引发该
# 中断，这样 CPU 就可以有机会让这个中断处理程序模拟协处理器指令（169 行）。

```



```

# CR0 的交换标志 TS 是在 CPU 执行任务转换时设置的。TS 可以用来确定什么时候协处理器中的
# 内容与 CPU 正在执行的任务不匹配了。当 CPU 在运行一个协处理器转义指令时发现 TS 置位时，
# 就会引发该中断。此时就可以保存前一个任务的协处理器内容，并恢复新任务的协处理器执行
# 状态（176 行）。参见 kernel/sched.c，92 行。该中断最后将转移到标号 ret_from_sys_call
# 处执行下去（检测并处理信号）。
147 .align 2
148 _device_not_available:
149     push %ds
150     push %es
151     push %fs
152     pushl %edx
153     pushl %ecx
154     pushl %ebx
155     pushl %eax
156     movl $0x10,%eax          # ds, es 置为指向内核数据段。
157     mov %ax,%ds
158     mov %ax,%es
159     movl $0x17,%eax          # fs 置为指向局部数据段（出错程序的数据段）。
160     mov %ax,%fs
# 清 CR0 中任务已交换标志 TS，并取 CR0 值。若其中协处理器仿真标志 EM 没有置位，说明不是 EM
# 引起的中断，则恢复任务协处理器状态，执行 C 函数 math_state_restore()，并在返回时去执行
# ret_from_sys_call 处的代码。
161     pushl $ret_from_sys_call # 把下面跳转或调用的返回地址入栈。
162     clts                     # clear TS so that we can use math
163     movl %cr0,%eax
164     testl $0x4,%eax          # EM (math emulation bit)
165     je _math_state_restore    # 执行 C 函数 math_state_restore() (kernel/sched.c, 77)。

# 若 EM 标志是置位的，则去执行数学仿真程序 math_emulate()。
166     pushl %ebp
167     pushl %esi
168     pushl %edi
169     call _math_emulate        # 调用 C 函数 math_emulate (math/math_emulate.c, 18)。
170     popl %edi
171     popl %esi
172     popl %ebp
173     ret                       # 这里的 ret 将跳转到 ret_from_sys_call(101 行)。
174
#### int32 -- (int 0x20) 时钟中断处理程序。中断频率被设置为 100Hz(include/linux/sched.h, 5)，
# 定时芯片 8253/8254 是在(kernel/sched.c, 406)处初始化的。因此这里 jiffies 每 10 毫秒加 1。
# 这段代码将 jiffies 增 1，发送结束中断指令给 8259 控制器，然后用当前特权级作为参数调用
# C 函数 do_timer(long CPL)。当调用返回时转去检测并处理信号。
175 .align 2
176 _timer_interrupt:
177     push %ds                  # save ds, es and put kernel data space
178     push %es                  # into them. %fs is used by _system_call
179     push %fs
180     pushl %edx                # we save %eax,%ecx,%edx as gcc doesn't
181     pushl %ecx                # save those across function calls. %ebx
182     pushl %ebx                # is saved as we use that in ret_sys_call
183     pushl %eax
184     movl $0x10,%eax          # ds, es 置为指向内核数据段。
185     mov %ax,%ds

```

```

186     mov %ax,%es
187     movl $0x17,%eax      # fs 置为指向局部数据段（程序的数据段）。
188     mov %ax,%fs
189     incl _jiffies
# 由于初始化中断控制芯片时没有采用自动 EOI，所以这里需要发指令结束该硬件中断。
190     movb $0x20,%al      # EOI to interrupt controller #1
191     outb %al,$0x20      # 操作命令字 OCW2 送 0x20 端口。
# 下面从堆栈中取出执行系统调用代码的选择符（CS 段寄存器值）中的当前特权级别（0 或 3）并压入
# 堆栈，作为 do_timer 的参数。do_timer() 函数执行任务切换、计时等工作，在 kernel/sched.c，
# 305 行实现。
192     movl CS(%esp),%eax
193     andl $3,%eax        # %eax is CPL (0 or 3, 0=supervisor)
194     pushl %eax
195     call _do_timer      # 'do_timer(long CPL)' does everything from
196     addl $4,%esp        # task switching to accounting ...
197     jmp ret_from_sys_call
198
#### 这是 sys_execve() 系统调用。取中断调用程序的代码指针作为参数调用 C 函数 do_execve()。
# do_execve() 在 (fs/exec.c, 182)。
199 .align 2
200 _sys_execve:
201     lea EIP(%esp),%eax   # eax 指向堆栈中保存用户程序 eip 指针处 (EIP+%esp)。
202     pushl %eax
203     call _do_execve
204     addl $4,%esp        # 丢弃调用时压入栈的 EIP 值。
205     ret
206
#### sys_fork() 调用，用于创建子进程，是 system_call 功能 2。原形在 include/linux/sys.h 中。
# 首先调用 C 函数 find_empty_process()，取得一个进程号 pid。若返回负数则说明目前任务数组
# 已满。然后调用 copy_process() 复制进程。
207 .align 2
208 _sys_fork:
209     call _find_empty_process # 调用 find_empty_process() (kernel/fork.c, 135)。
210     testl %eax,%eax        # 在 eax 中返回进程号 pid。若返回负数则退出。
211     js 1f
212     push %gs
213     pushl %esi
214     pushl %edi
215     pushl %ebp
216     pushl %eax
217     call _copy_process     # 调用 C 函数 copy_process() (kernel/fork.c, 68)。
218     addl $20,%esp        # 丢弃这里所有压栈内容。
219 1:     ret
220
#### int 46 -- (int 0x2E) 硬盘中断处理程序，响应硬件中断请求 IRQ14。
# 当请求的硬盘操作完成或出错就会发出此中断信号。（参见 kernel/blk_drv/hd.c）。
# 首先向 8259A 中断控制从芯片发送结束硬件中断指令 (EOI)，然后取变量 do_hd 中的函数指针放入 edx
# 寄存器中，并置 do_hd 为 NULL，接着判断 edx 函数指针是否为空。如果为空，则给 edx 赋值指向
# unexpected_hd_interrupt()，用于显示出错信息。随后向 8259A 主芯片送 EOI 指令，并调用 edx 中
# 指针指向的函数：read_intr()、write_intr() 或 unexpected_hd_interrupt()。
221 _hd_interrupt:
222     pushl %eax
223     pushl %ecx

```

```

224     pushl %edx
225     push %ds
226     push %es
227     push %fs
228     movl $0x10,%eax      # ds, es 置为内核数据段。
229     mov %ax,%ds
230     mov %ax,%es
231     movl $0x17,%eax      # fs 置为调用程序的局部数据段。
232     mov %ax,%fs
# 由于初始化中断控制芯片时没有采用自动 EOI，所以这里需要发指令结束该硬件中断。
233     movb $0x20,%al
234     outb %al,$0xA0        # EOI to interrupt controller #1 # 送从 8259A。
235     jmp 1f                # give port chance to breathe # jmp 起延时作用。
236 1:     jmp 1f
# do_hd 定义为一个函数指针，将被赋值 read_intr()或 write_intr()函数地址。放到 edx 寄存器后
# 就将 do_hd 指针变量置为 NULL。然后测试得到的函数指针，若该指针为空，则赋予该指针指向 C
# 函数 unexpected_hd_interrupt()，以处理未知硬盘中断。
237 1:     xorl %edx,%edx
238     xchgl _do_hd,%edx
239     testl %edx,%edx      # 测试函数指针是否为 Null。
240     jne 1f                # 若空，则使指针指向 C 函数 unexpected_hd_interrupt()。
241     movl $_unexpected_hd_interrupt,%edx # (kernel/blk_drv/hdc, 237)。
242 1:     outb %al,$0x20      # 送主 8259A 中断控制器 EOI 指令（结束硬件中断）。
243     call *%edx           # "interesting" way of handling intr.
244     pop %fs              # 上句调用 do_hd 指向的 C 函数。
245     pop %es
246     pop %ds
247     popl %edx
248     popl %ecx
249     popl %eax
250     iret
251
##### int38 -- (int 0x26) 软盘驱动器中断处理程序，响应硬件中断请求 IRQ6。
# 其处理过程与上面对硬盘的处理基本一样。(kernel/blk_drv/floppy.c)。
# 首先向 8259A 中断控制器主芯片发送 EOI 指令，然后取变量 do_floppy 中的函数指针放入 eax
# 寄存器中，并置 do_floppy 为 NULL，接着判断 eax 函数指针是否为空。如为空，则给 eax 赋值指向
# unexpected_floppy_interrupt()，用于显示出错信息。随后调用 eax 指向的函数：rw_interrupt,
# seek_interrupt, recal_interrupt, reset_interrupt 或 unexpected_floppy_interrupt。
252 _floppy_interrupt:
253     pushl %eax
254     pushl %ecx
255     pushl %edx
256     push %ds
257     push %es
258     push %fs
259     movl $0x10,%eax      # ds, es 置为内核数据段。
260     mov %ax,%ds
261     mov %ax,%es
262     movl $0x17,%eax      # fs 置为调用程序的局部数据段。
263     mov %ax,%fs
264     movb $0x20,%al
265     outb %al,$0x20        # 送主 8259A 中断控制器 EOI 指令（结束硬件中断）。
# do_floppy 为一函数指针，将被赋值实际处理 C 函数指针。该指针在被交换放到 eax 寄存器后就将

```

### 8.5.3.1 GNU 汇编语言的 32 位寻址方式

Intel: [basepointer + indexpointer\*indexscal + immed32]

在应用时，并不需要写出所有这些字段，但 `immed32` 和 `basepointer` 之中必须有一个存在。以下是一个例子。

- 注意：变量前的下划线是从汇编程序中得到静态（全局）C 变量(`booga`)的方法。

- o 通过寄存器中的内容作为基址寻址一个变量:

- o 在一个整数数组中寻址一个值（比例值为 4）：

- o 使用直接数寻址偏移量:

对于 C 语言: `*(p+1)` 其中 `p` 是字符的指针 `char *`

Intel: [eax+1]

o 在一个 8 字节为一个记录的数组中寻址指定的字符。其中 `eax` 中是指定的记录号，`ebx` 中是指定字符在记录中的偏移址：

AT&T: `_array(%ebx,%eax,8)`

Intel: `[ebx + eax*8 + _array]`

### 8.5.3.2 增加系统调用功能

若要为自己的内核实现一个新的系统调用功能，那么我们首先应该决定它的确切用途是什么。Linux 系统不提倡一个系统调用用来实现多种用途（除了 `ioctl()` 系统调用）。另外，我们还需要确定新的系统调用的参数、返回值和错误码。系统调用的接口应该尽量简洁，因此参数应尽可能地少。还有，在设计时也应该考虑到系统调用的通用性和可移植性。如果我们想为 Linux 0.11 增加新的系统调用功能，那么需要做以下一些事情。

首先在相关程序中编制出新系统调用的处理函数，例如名称为 `sys_sethostname()` 的函数。该函数用于修改系统的计算机名称。通常这个处理函数可以放置在 `kernel/sys.c` 程序中。另外，由于使用了 `thisname` 结构，因此还需要把 `sys_uname()` 中的 `thisname` 结构（218-220 行）移动到该函数外部。

---

```
#define MAXHOSTNAMELEN 8
int sys_sethostname(char *name, int len)
{
    int i;

    if (!suser())
        return -EPERM;
    if (len > MAXHOSTNAMELEN)
        return -EINVAL;
    for (i=0; i < len; i++) {
        if ((thisname.nodename[i] = get_fs_byte(name+i)) == 0)
            break;
    }
    if (thisname.nodename[i]) {
        thisname.nodename[i>MAXHOSTNAMELEN ? MAXHOSTNAMELEN : i] = 0;
    }
    return 0;
}
```

---

然后在 `include/unistd.h` 文件中增加新系统调用功能号和原型定义。例如可以在第 131 行后面加入功能号，在 251 行后面添加原型定义：

---

```
// 新系统调用功能号。
#define __NR_sethostname 72
// 新系统调用函数原型。
int sethostname(char *name, int len);
```

---

接着在 `include/linux/sys.h` 文件中加入外部函数声明并在函数指针表 `sys_call_table` 末端插入新系统调用处理函数的名称，见如下所示。注意，一定要严格按照功能号顺序排列函数名。

---

```
extern int sys_sethostname();
// 函数指针数组表。
fn_ptr sys_call_table[] = { sys_setup, sys_exit, sys_fork, sys_read,
    ...,
```

---

---

```
sys_setreuid, sys_setregid, sys_sethostname };
```

---

然后修改 `system_call.s` 程序第 61 行，将内核系统调用总数 `nr_system_calls` 增 1。此时可以重新编译内核。最后参照 `lib/` 目录下库函数的实现方法在 `libc` 库中增加新的系统调用库函数 `sethostname()`。

---

```
#define __LIBRARY__
#include <unistd.h>

_syscall2(int, sethostname, char *, name, int, len);
```

---

### 8.5.3.3 在汇编程序中直接使用系统调用

下面是 Linus 在说明 `as86` 与 `GNU as` 的关系和区别是给出的一个简单例子 `asm.s`。该例子说明了如何在 `Linux` 系统中用汇编语言编制出一个独立的程序来，即不使用起始代码模块（例如 `crt0.o`）和库文件中的函数。该程序如下：

---

```
.text
_entry:
    movl $4,%eax           # 系统调用号，写操作。
    movl $1,%ebx           # 写调用的参数，是文件描述符。数值 1 对应标准输出 stdout。
    movl $message,%ecx     # 参数，缓冲区指针。
    movl $12,%edx          # 参数，写数据长度值（数数下面字符串的长度☺）。
    int $0x80
    movl $1,%eax           # 系统调用号，退出程序。
    int $0x80

message:
    .ascii "Hello World\n" # 欲写的数据。
```

---

其中使用了两个系统调用：4 - 写文件操作 `sys_write()` 和 1 - 退出程序 `sys_exit()`。写文件系统调用所执行的 C 函数申明为 `sys_write(int fd, char *buf, int len)`，参见程序 `fs/read_write.c`，从 83 行开始。它带有 3 个参数。在调用系统调用之前这 3 个参数分别被存放在寄存器 `ebx`、`ecx` 和 `edx` 中。该程序编译和执行的步骤如下：

---

```
[/usr/root]# as -o asm.o asm.s
[/usr/root]# ld -o asm asm.o
[/usr/root]# ./asm
Hello World
[/usr/root]#
```

---

## 8.6 mktime.c 程序

### 8.6.1 功能描述

该程序只有一个函数 `kernel_mktime()`，仅供内核使用。计算从 1970 年 1 月 1 日 0 时起到开机当日经过的秒数（日历时间），作为开机时间。该函数与标准 C 库中提供的 `mktime()` 函数的功能完全一样，

都是将 `tm` 结构表示的时间转换成 UNIX 日历时间。但是由于内核不是普通程序，不能调用开发环境库中的函数，因此这里就必须自己专门编写一个了。

## 8.6.2 代码注释

程序 8-5 linux/kernel/mktime.c 程序

```

1  /*
2   * linux/kernel/mktime.c
3   *
4   * (C) 1991 Linus Torvalds
5   */
6
7 #include <time.h>           // 时间头文件，定义了标准时间数据结构 tm 和一些处理时间函数原型。
8
9 /*
10  * This isn't the library routine, it is only used in the kernel.
11  * as such, we don't care about years<1970 etc, but assume everything
12  * is ok. Similarly, TZ etc is happily ignored. We just do everything
13  * as easily as possible. Let's find something public for the library
14  * routines (although I think minix times is public).
15  */
16 /*
17  * PS. I hate whoever though up the year 1970 - couldn't they have gotten
18  * a leap-year instead? I also hate Gregorius, pope or no. I'm grumpy.
19  */
20 /*
21  * 这不是库函数，它仅供内核使用。因此我们不关心小于 1970 年的年份等，但假定一切均很正常。
22  * 同样，时间区域 TZ 问题也先忽略。我们只是尽可能简单地处理问题。最好能找到一些公开的库函数
23  * （尽管我认为 minix 的时间函数是公开的）。
24  * 另外，我恨那个设置 1970 年开始的人 - 难道他们就不能选择从一个闰年开始？我恨格里高利历、
25  * 罗马教皇、主教，我什么都不在乎。我是个脾气暴躁的人。
26  */
27 #define MINUTE 60           // 1 分钟的秒数。
28 #define HOUR (60*MINUTE)   // 1 小时的秒数。
29 #define DAY (24*HOUR)      // 1 天的秒数。
30 #define YEAR (365*DAY)     // 1 年的秒数。
31
32 /* interestingly, we assume leap-years */
33 /* 有趣的是我们考虑进了闰年 */
34 // 下面以年为界限，定义了每个月开始时的秒数时间。
35 static int month[12] = {
36     0,
37     DAY*(31),
38     DAY*(31+29),
39     DAY*(31+29+31),
40     DAY*(31+29+31+30),
41     DAY*(31+29+31+30+31),
42     DAY*(31+29+31+30+31+30),
43     DAY*(31+29+31+30+31+30+31),
44     DAY*(31+29+31+30+31+30+31+31),
45     DAY*(31+29+31+30+31+30+31+31+30),
46     DAY*(31+29+31+30+31+30+31+31+30+31),
47     DAY*(31+29+31+30+31+30+31+31+30+31),
48     DAY*(31+29+31+30+31+30+31+31+30+31),
49     DAY*(31+29+31+30+31+30+31+31+30+31),
50     DAY*(31+29+31+30+31+30+31+31+30+31),
51     DAY*(31+29+31+30+31+30+31+31+30+31),
52     DAY*(31+29+31+30+31+30+31+31+30+31),
53     DAY*(31+29+31+30+31+30+31+31+30+31),
54     DAY*(31+29+31+30+31+30+31+31+30+31),
55     DAY*(31+29+31+30+31+30+31+31+30+31),
56     DAY*(31+29+31+30+31+30+31+31+30+31),
57     DAY*(31+29+31+30+31+30+31+31+30+31),
58     DAY*(31+29+31+30+31+30+31+31+30+31),
59     DAY*(31+29+31+30+31+30+31+31+30+31),
60     DAY*(31+29+31+30+31+30+31+31+30+31),
61     DAY*(31+29+31+30+31+30+31+31+30+31),
62     DAY*(31+29+31+30+31+30+31+31+30+31),
63     DAY*(31+29+31+30+31+30+31+31+30+31),
64     DAY*(31+29+31+30+31+30+31+31+30+31),
65     DAY*(31+29+31+30+31+30+31+31+30+31),
66     DAY*(31+29+31+30+31+30+31+31+30+31),
67     DAY*(31+29+31+30+31+30+31+31+30+31),
68     DAY*(31+29+31+30+31+30+31+31+30+31),
69     DAY*(31+29+31+30+31+30+31+31+30+31),
70     DAY*(31+29+31+30+31+30+31+31+30+31),
71     DAY*(31+29+31+30+31+30+31+31+30+31),
72     DAY*(31+29+31+30+31+30+31+31+30+31),
73     DAY*(31+29+31+30+31+30+31+31+30+31),
74     DAY*(31+29+31+30+31+30+31+31+30+31),
75     DAY*(31+29+31+30+31+30+31+31+30+31),
76     DAY*(31+29+31+30+31+30+31+31+30+31),
77     DAY*(31+29+31+30+31+30+31+31+30+31),
78     DAY*(31+29+31+30+31+30+31+31+30+31),
79     DAY*(31+29+31+30+31+30+31+31+30+31),
80     DAY*(31+29+31+30+31+30+31+31+30+31),
81     DAY*(31+29+31+30+31+30+31+31+30+31),
82     DAY*(31+29+31+30+31+30+31+31+30+31),
83     DAY*(31+29+31+30+31+30+31+31+30+31),
84     DAY*(31+29+31+30+31+30+31+31+30+31),
85     DAY*(31+29+31+30+31+30+31+31+30+31),
86     DAY*(31+29+31+30+31+30+31+31+30+31),
87     DAY*(31+29+31+30+31+30+31+31+30+31),
88     DAY*(31+29+31+30+31+30+31+31+30+31),
89     DAY*(31+29+31+30+31+30+31+31+30+31),
90     DAY*(31+29+31+30+31+30+31+31+30+31),
91     DAY*(31+29+31+30+31+30+31+31+30+31),
92     DAY*(31+29+31+30+31+30+31+31+30+31),
93     DAY*(31+29+31+30+31+30+31+31+30+31),
94     DAY*(31+29+31+30+31+30+31+31+30+31),
95     DAY*(31+29+31+30+31+30+31+31+30+31),
96     DAY*(31+29+31+30+31+30+31+31+30+31),
97     DAY*(31+29+31+30+31+30+31+31+30+31),
98     DAY*(31+29+31+30+31+30+31+31+30+31),
99     DAY*(31+29+31+30+31+30+31+31+30+31),
100    DAY*(31+29+31+30+31+30+31+31+30+31),
101    DAY*(31+29+31+30+31+30+31+31+30+31),
102    DAY*(31+29+31+30+31+30+31+31+30+31),
103    DAY*(31+29+31+30+31+30+31+31+30+31),
104    DAY*(31+29+31+30+31+30+31+31+30+31),
105    DAY*(31+29+31+30+31+30+31+31+30+31),
106    DAY*(31+29+31+30+31+30+31+31+30+31),
107    DAY*(31+29+31+30+31+30+31+31+30+31),
108    DAY*(31+29+31+30+31+30+31+31+30+31),
109    DAY*(31+29+31+30+31+30+31+31+30+31),
110    DAY*(31+29+31+30+31+30+31+31+30+31),
111    DAY*(31+29+31+30+31+30+31+31+30+31),
112    DAY*(31+29+31+30+31+30+31+31+30+31),
113    DAY*(31+29+31+30+31+30+31+31+30+31),
114    DAY*(31+29+31+30+31+30+31+31+30+31),
115    DAY*(31+29+31+30+31+30+31+31+30+31),
116    DAY*(31+29+31+30+31+30+31+31+30+31),
117    DAY*(31+29+31+30+31+30+31+31+30+31),
118    DAY*(31+29+31+30+31+30+31+31+30+31),
119    DAY*(31+29+31+30+31+30+31+31+30+31),
120    DAY*(31+29+31+30+31+30+31+31+30+31),
121    DAY*(31+29+31+30+31+30+31+31+30+31),
122    DAY*(31+29+31+30+31+30+31+31+30+31),
123    DAY*(31+29+31+30+31+30+31+31+30+31),
124    DAY*(31+29+31+30+31+30+31+31+30+31),
125    DAY*(31+29+31+30+31+30+31+31+30+31),
126    DAY*(31+29+31+30+31+30+31+31+30+31),
127    DAY*(31+29+31+30+31+30+31+31+30+31),
128    DAY*(31+29+31+30+31+30+31+31+30+31),
129    DAY*(31+29+31+30+31+30+31+31+30+31),
130    DAY*(31+29+31+30+31+30+31+31+30+31),
131    DAY*(31+29+31+30+31+30+31+31+30+31),
132    DAY*(31+29+31+30+31+30+31+31+30+31),
133    DAY*(31+29+31+30+31+30+31+31+30+31),
134    DAY*(31+29+31+30+31+30+31+31+30+31),
135    DAY*(31+29+31+30+31+30+31+31+30+31),
136    DAY*(31+29+31+30+31+30+31+31+30+31),
137    DAY*(31+29+31+30+31+30+31+31+30+31),
138    DAY*(31+29+31+30+31+30+31+31+30+31),
139    DAY*(31+29+31+30+31+30+31+31+30+31),
140    DAY*(31+29+31+30+31+30+31+31+30+31),
141    DAY*(31+29+31+30+31+30+31+31+30+31),
142    DAY*(31+29+31+30+31+30+31+31+30+31),
143    DAY*(31+29+31+30+31+30+31+31+30+31),
144    DAY*(31+29+31+30+31+30+31+31+30+31),
145    DAY*(31+29+31+30+31+30+31+31+30+31),
146    DAY*(31+29+31+30+31+30+31+31+30+31),
147    DAY*(31+29+31+30+31+30+31+31+30+31),
148    DAY*(31+29+31+30+31+30+31+31+30+31),
149    DAY*(31+29+31+30+31+30+31+31+30+31),
150    DAY*(31+29+31+30+31+30+31+31+30+31),
151    DAY*(31+29+31+30+31+30+31+31+30+31),
152    DAY*(31+29+31+30+31+30+31+31+30+31),
153    DAY*(31+29+31+30+31+30+31+31+30+31),
154    DAY*(31+29+31+30+31+30+31+31+30+31),
155    DAY*(31+29+31+30+31+30+31+31+30+31),
156    DAY*(31+29+31+30+31+30+31+31+30+31),
157    DAY*(31+29+31+30+31+30+31+31+30+31),
158    DAY*(31+29+31+30+31+30+31+31+30+31),
159    DAY*(31+29+31+30+31+30+31+31+30+31),
160    DAY*(31+29+31+30+31+30+31+31+30+31),
161    DAY*(31+29+31+30+31+30+31+31+30+31),
162    DAY*(31+29+31+30+31+30+31+31+30+31),
163    DAY*(31+29+31+30+31+30+31+31+30+31),
164    DAY*(31+29+31+30+31+30+31+31+30+31),
165    DAY*(31+29+31+30+31+30+31+31+30+31),
166    DAY*(31+29+31+30+31+30+31+31+30+31),
167    DAY*(31+29+31+30+31+30+31+31+30+31),
168    DAY*(31+29+31+30+31+30+31+31+30+31),
169    DAY*(31+29+31+30+31+30+31+31+30+31),
170    DAY*(31+29+31+30+31+30+31+31+30+31),
171    DAY*(31+29+31+30+31+30+31+31+30+31),
172    DAY*(31+29+31+30+31+30+31+31+30+31),
173    DAY*(31+29+31+30+31+30+31+31+30+31),
174    DAY*(31+29+31+30+31+30+31+31+30+31),
175    DAY*(31+29+31+30+31+30+31+31+30+31),
176    DAY*(31+29+31+30+31+30+31+31+30+31),
177    DAY*(31+29+31+30+31+30+31+31+30+31),
178    DAY*(31+29+31+30+31+30+31+31+30+31),
179    DAY*(31+29+31+30+31+30+31+31+30+31),
180    DAY*(31+29+31+30+31+30+31+31+30+31),
181    DAY*(31+29+31+30+31+30+31+31+30+31),
182    DAY*(31+29+31+30+31+30+31+31+30+31),
183    DAY*(31+29+31+30+31+30+31+31+30+31),
184    DAY*(31+29+31+30+31+30+31+31+30+31),
185    DAY*(31+29+31+30+31+30+31+31+30+31),
186    DAY*(31+29+31+30+31+30+31+31+30+31),
187    DAY*(31+29+31+30+31+30+31+31+30+31),
188    DAY*(31+29+31+30+31+30+31+31+30+31),
189    DAY*(31+29+31+30+31+30+31+31+30+31),
190    DAY*(31+29+31+30+31+30+31+31+30+31),
191    DAY*(31+29+31+30+31+30+31+31+30+31),
192    DAY*(31+29+31+30+31+30+31+31+30+31),
193    DAY*(31+29+31+30+31+30+31+31+30+31),
194    DAY*(31+29+31+30+31+30+31+31+30+31),
195    DAY*(31+29+31+30+31+30+31+31+30+31),
196    DAY*(31+29+31+30+31+30+31+31+30+31),
197    DAY*(31+29+31+30+31+30+31+31+30+31),
198    DAY*(31+29+31+30+31+30+31+31+30+31),
199    DAY*(31+29+31+30+31+30+31+31+30+31),
200    DAY*(31+29+31+30+31+30+31+31+30+31),
201    DAY*(31+29+31+30+31+30+31+31+30+31),
202    DAY*(31+29+31+30+31+30+31+31+30+31),
203    DAY*(31+29+31+30+31+30+31+31+30+31),
204    DAY*(31+29+31+30+31+30+31+31+30+31),
205    DAY*(31+29+31+30+31+30+31+31+30+31),
206    DAY*(31+29+31+30+31+30+31+31+30+31),
207    DAY*(31+29+31+30+31+30+31+31+30+31),
208    DAY*(31+29+31+30+31+30+31+31+30+31),
209    DAY*(31+29+31+30+31+30+31+31+30+31),
210    DAY*(31+29+31+30+31+30+31+31+30+31),
211    DAY*(31+29+31+30+31+30+31+31+30+31),
212    DAY*(31+29+31+30+31+30+31+31+30+31),
213    DAY*(31+29+31+30+31+30+31+31+30+31),
214    DAY*(31+29+31+30+31+30+31+31+30+31),
215    DAY*(31+29+31+30+31+30+31+31+30+31),
216    DAY*(31+29+31+30+31+30+31+31+30+31),
217    DAY*(31+29+31+30+31+30+31+31+30+31),
218    DAY*(31+29+31+30+31+30+31+31+30+31),
219    DAY*(31+29+31+30+31+30+31+31+30+31),
220    DAY*(31+29+31+30+31+30+31+31+30+31),
221    DAY*(31+29+31+30+31+30+31+31+30+31),
222    DAY*(31+29+31+30+31+30+31+31+30+31),
223    DAY*(31+29+31+30+31+30+31+31+30+31),
224    DAY*(31+29+31+30+31+30+31+31+30+31),
225    DAY*(31+29+31+30+31+30+31+31+30+31),
226    DAY*(31+29+31+30+31+30+31+31+30+31),
227    DAY*(31+29+31+30+31+30+31+31+30+31),
228    DAY*(31+29+31+30+31+30+31+31+30+31),
229    DAY*(31+29+31+30+31+30+31+31+30+31),
230    DAY*(31+29+31+30+31+30+31+31+30+31),
231    DAY*(31+29+31+30+31+30+31+31+30+31),
232    DAY*(31+29+31+30+31+30+31+31+30+31),
233    DAY*(31+29+31+30+31+30+31+31+30+31),
234    DAY*(31+29+31+30+31+30+31+31+30+31),
235    DAY*(31+29+31+30+31+30+31+31+30+31),
236    DAY*(31+29+31+30+31+30+31+31+30+31),
237    DAY*(31+29+31+30+31+30+31+31+30+31),
238    DAY*(31+29+31+30+31+30+31+31+30+31),
239    DAY*(31+29+31+30+31+30+31+31+30+31),
240    DAY*(31+29+31+30+31+30+31+31+30+31),
241    DAY*(31+29+31+30+31+30+31+31+30+31),
242    DAY*(31+29+31+30+31+30+31+31+30+31),
243    DAY*(31+29+31+30+31+30+31+31+30+31),
244    DAY*(31+29+31+30+31+30+31+31+30+31),
245    DAY*(31+29+31+30+31+30+31+31+30+31),
246    DAY*(31+29+31+30+31+30+31+31+30+31),
247    DAY*(31+29+31+30+31+30+31+31+30+31),
248    DAY*(31+29+31+30+31+30+31+31+30+31),
249    DAY*(31+29+31+30+31+30+31+31+30+31),
250    DAY*(31+29+31+30+31+30+31+31+30+31),
251    DAY*(31+29+31+30+31+30+31+31+30+31),
252    DAY*(31+29+31+30+31+30+31+31+30+31),
253    DAY*(31+29+31+30+31+30+31+31+30+31),
254    DAY*(31+29+31+30+31+30+31+31+30+31),
255    DAY*(31+29+31+30+31+30+31+31+30+31),
256    DAY*(31+29+31+30+31+30+31+31+30+31),
257    DAY*(31+29+31+30+31+30+31+31+30+31),
258    DAY*(31+29+31+30+31+30+31+31+30+31),
259    DAY*(31+29+31+30+31+30+31+31+30+31),
260    DAY*(31+29+31+30+31+30+31+31+30+31),
261    DAY*(31+29+31+30+31+30+31+31+30+31),
262    DAY*(31+29+31+30+31+30+31+31+30+31),
263    DAY*(31+29+31+30+31+30+31+31+30+31),
264    DAY*(31+29+31+30+31+30+31+31+30+31),
265    DAY*(31+29+31+30+31+30+31+31+30+31),
266    DAY*(31+29+31+30+31+30+31+31+30+31),
267    DAY*(31+29+31+30+31+30+31+31+30+31),
268    DAY*(31+29+31+30+31+30+31+31+30+31),
269    DAY*(31+29+31+30+31+30+31+31+30+31),
270    DAY*(31+29+31+30+31+30+31+31+30+31),
271    DAY*(31+29+31+30+31+30+31+31+30+31),
272    DAY*(31+29+31+30+31+30+31+31+30+31),
273    DAY*(31+29+31+30+31+30+31+31+30+31),
274    DAY*(31+29+31+30+31+30+31+31+30+31),
275    DAY*(31+29+31+30+31+30+31+31+30+31),
276    DAY*(31+29+31+30+31+30+31+31+30+31),
277    DAY*(31+29+31+30+31+30+31+31+30+31),
278    DAY*(31+29+31+30+31+30+31+31+30+31),
279    DAY*(31+29+31+30+31+30+31+31+30+31),
280    DAY*(31+29+31+30+31+30+31+31+30+31),
281    DAY*(31+29+31+30+31+30+31+31+30+31),
282    DAY*(31+29+31+30+31+30+31+31+30+31),
283    DAY*(31+29+31+30+31+30+31+31+30+31),
284    DAY*(31+29+31+30+31+30+31+31+30+31),
285    DAY*(31+29+31+30+31+30+31+31+30+31),
286    DAY*(31+29+31+30+31+30+31+31+30+31),
287    DAY*(31+29+31+30+31+30+31+31+30+31),
288    DAY*(31+29+31+30+31+30+31+31+30+31),
289    DAY*(31+29+31+30+31+30+31+31+30+31),
290    DAY*(31+29+31+30+31+30+31+31+30+31),
291    DAY*(31+29+31+30+31+30+31+31+30+31),
292    DAY*(31+29+31+30+31+30+31+31+30+31),
293    DAY*(31+29+31+30+31+30+31+31+30+31),
294    DAY*(31+29+31+30+31+30+31+31+30+31),
295    DAY*(31+29+31+30+31+30+31+31+30+31),
296    DAY*(31+29+31+30+31+30+31+31+30+31),
297    DAY*(31+29+31+30+31+30+31+31+30+31),
298    DAY*(31+29+31+30+31+30+31+31+30+31),
299    DAY*(31+29+31+30+31+30+31+31+30+31),
300    DAY*(31+29+31+30+31+30+31+31+30+31),
301    DAY*(31+29+31+30+31+30+31+31+30+31),
302    DAY*(31+29+31+30+31+30+31+31+30+31),
303    DAY*(31+29+31+30+31+30+31+31+30+31),
304    DAY*(31+29+31+30+31+30+31+31+30+31),
305    DAY*(31+29+31+30+31+30+31+31+30+31),
306    DAY*(31+29+31+30+31+30+31+31+30+31),
307    DAY*(31+29+31+30+31+30+31+31+30+31),
308    DAY*(31+29+31+30+31+30+31+31+30+31),
309    DAY*(31+29+31+30+31+30+31+31+30+31),
310    DAY*(31+29+31+30+31+30+31+31+30+31),
311    DAY*(31+29+31+30+31+30+31+31+30+31),
312    DAY*(31+29+31+30+31+30+31+31+30+31),
313    DAY*(31+29+31+30+31+30+31+31+30+31),
314    DAY*(31+29+31+30+31+30+31+31+30+31),
315    DAY*(31+29+31+30+31+30+31+31+30+31),
316    DAY*(31+29+31+30+31+30+31+31+30+31),
317    DAY*(31+29+31+30+31+30+31+31+30+31),
318    DAY*(31+29+31+30+31+30+31+31+30+31),
319    DAY*(31+29+31+30+31+30+31+31+30+31),
320    DAY*(31+29+31+30+31+30+31+31+30+31),
321    DAY*(31+29+31+30+31+30+31+31+30+31),
322    DAY*(31+29+31+30+31+30+31+31+30+31),
323    DAY*(31+29+31+30+31+30+31+31+30+31),
324    DAY*(31+29+31+30+31+30+31+31+30+31),
325    DAY*(31+29+31+30+31+30+31+31+30+31),
326    DAY*(31+29+31+30+31+30+31+31+30+31),
327    DAY*(31+29+31+30+31+30+31+31+30+31),
328    DAY*(31+29+31+30+31+30+31+31+30+31),
329    DAY*(31+29+31+30+31+30+31+31+30+31),
330    DAY*(31+29+31+30+31+30+31+31+30+31),
331    DAY*(31+29+31+30+31+30+31+31+30+31),
332    DAY*(31+29+31+30+31+30+31+31+30+31),
333    DAY*(31+29+31+30+31+30+31+31+30+31),
334    DAY*(31+29+31+30+31+30+31+31+30+31),
335    DAY*(31+29+31+30+31+30+31+31+30+31),
336    DAY*(31+29+31+30+31+30+31+31+30+31),
337    DAY*(31+29+31+30+31+30+31+31+30+31),
338    DAY*(31+29+31+30+31+30+31+31+30+31),
339    DAY*(31+29+31+30+31+30+31+31+30+31),
340    DAY*(31+29+31+30+31+30+31+31+30+31),
341    DAY*(31+29+31+30+31+30+31+31+30+31),
342    DAY*(31+29+31+30+31+30+31+31+30+31),
343    DAY*(31+29+31+30+31+30+31+31+30+31),
344    DAY*(31+29+31+30+31+30+31+31+30+31),
345    DAY*(31+29+31+30+31+30+31+31+30+31),
346    DAY*(31+29+31+30+31+30+31+31+30+31),
347    DAY*(31+29+31+30+31+30+31+31+30+31),
348    DAY*(31+29+31+30+31+30+31+31+30+31),
349    DAY*(31+29+31+30+31+30+31+31+30+31),
350    DAY*(31+29+31+30+31+30+31+31+30+31),
351    DAY*(31+29+31+30+31+30+31+31+30+31),
352    DAY*(31+29+31+30+31+30+31+31+30+31),
353    DAY*(31+29+31+30+31+30+31+31+30+31),
354    DAY*(31+29+31+30+31+30+31+31+30+31),
355    DAY*(31+29+31+30+31+30+31+31+30+31),
```



```

38         DAY*(31+29+31+30+31+30+31+31+30+31+30)
39     };
40
41     // 该函数计算从 1970 年 1 月 1 日 0 时起到开机当日经过的秒数，作为开机时间。
42     // 参数 tm 中各字段已经在 init/main.c 中被赋值，信息取自 CMOS。
43     long kernel_mktime(struct tm * tm)
44     {
45         long res;
46         int year;
47
48         // 首先计算 70 年到现在经过的年数。因为是 2 位表示方式，所以会有 2000 年问题。我们可以
49         // 简单地在最前面添加一条语句来解决这个问题：if (tm->tm_year<70) tm->tm_year += 100;
50         // 由于 UNIX 计年份 y 是从 1970 年算起。到 1972 年就是一个闰年，因此过 3 年（71，72，73）
51         // 就是第 1 个闰年，这样从 1970 年开始的闰年数计算方法就应该为 1 + (y - 3)/4，即为
52         // (y + 1)/4。res = 这些年经过的秒数时间 + 每个闰年时多 1 天的秒数时间 + 当年到当月时
53         // 的秒数。另外，month[] 数组中已经在 2 月份的天数中包含进了闰年时的天数，即 2 月份天数
54         // 多算了 1 天。因此，若当年不是闰年并且当前月份大于 2 月份的话，我们就要减去这天。因
55         // 为从 70 年开始算起，所以当年是闰年的判断方法是 (y + 2) 能被 4 除尽。若不能除尽（有余
56         // 数）就不是闰年。
57         year = tm->tm_year - 70;
58         /* magic offsets (y+1) needed to get leapyears right. */
59         /* 为了获得正确的闰年数，这里需要这样一个魔幻值(y+1) */
60         res = YEAR*year + DAY*((year+1)/4);
61         res += month[tm->tm_mon];
62         /* and (y+2) here. If it wasn't a leap-year, we have to adjust */
63         /* 以及(y+2)。如果(y+2)不是闰年，那么我们就必须进行调整(减去一天的秒数时间)。*/
64         if (tm->tm_mon>1 && ((year+2)%4))
65             res -= DAY;
66         res += DAY*(tm->tm_mday-1);           // 再加上本月过去的天数的秒数时间。
67         res += HOUR*tm->tm_hour;             // 再加上当天过去的小时数的秒数时间。
68         res += MINUTE*tm->tm_min;            // 再加上 1 小时内过去的分钟数的秒数时间。
69         res += tm->tm_sec;                   // 再加上 1 分钟内已过的秒数。
70         return res;                         // 即等于从 1970 年以来经过的秒数时间。
71     }
72 }
73

```

## 8.6.3 其他信息

### 8.6.3.1 闰年时间的计算方法

闰年的基本计算方法是：

如果 y 能被 4 整除且不能被 100 除尽，或者能被 400 整除，则 y 是闰年。

## 8.7 sched.c 程序

### 8.7.1 功能描述

sched.c 是内核中有关任务（进程）调度管理的程序，其中包括有关调度的基本函数(sleep\_on()、wakeup()、schedule()等)以及一些简单的系统调用函数（比如 getpid()）。系统时钟中断处理过程中调用的定时函数 do\_timer()也被放置在本程序中。另外，为了便于软盘驱动器定时处理的编程，Linus 也将有关软盘定时操作的几个函数放到了这里。

这几个基本函数的代码虽然不长，但有些抽象，比较难以理解。好在市面上有许多教科书对此解释得都很清楚，因此可以参考其他书籍对这些函数的讨论。这些也就是教科书上重点讲述的对象，否则理论书籍也就没有什么好讲的了☺。这里仅对调度函数 `schedule()` 作一些详细说明。

`schedule()` 函数负责选择系统中下一个要运行的进程。它首先对所有任务（进程）进行检测，唤醒任何一个已经得到信号的任务。具体方法是针对任务数组中的每个任务，检查其报警定时值 `alarm`。如果任务的 `alarm` 时间已经过期 (`alarm < jiffies`)，则在它的信号位图中设置 `SIGALRM` 信号，然后清 `alarm` 值。`jiffies` 是系统从开机开始算起的滴答数（10ms/滴答）。在 `sched.h` 中定义。如果进程的信号位图中除去被阻塞的信号外还有其他信号，并且任务处于可中断睡眠状态（`TASK_INTERRUPTIBLE`），则置任务为就绪状态（`TASK_RUNNING`）。

随后是调度函数的核心处理部分。这部分代码根据进程的时间片和优先权调度机制，来选择随后要执行的任务。它首先循环检查任务数组中的所有任务，根据每个就绪态任务剩余执行时间的值 `counter`，选取该值最大的一个任务，并利用 `switch_to()` 函数切换到该任务。若所有就绪态任务的该值都等于零，表示此刻所有任务的时间片都已经运行完，于是就根据任务的优先权值 `priority`，重置每个任务的运行时间片值 `counter`，再重新执行循环检查所有任务的执行时间片值。

另两个值得一提的函数是自动进入睡眠函数 `sleep_on()` 和唤醒函数 `wake_up()`，这两个函数虽然很短，却要比 `schedule()` 函数难理解。这里用图示的方法加以解释。简单地说，`sleep_on()` 函数的主要功能是当一个进程（或任务）所请求的资源正忙或不在内存中时暂时切换出去，放在等待队列中等待一段时间。当切换回来后再继续运行。放入等待队列的方式是利用了函数中的 `tmp` 指针作为各个正在等待任务的联系。

函数中共牵涉到对三个任务指针操作：`*p`、`tmp` 和 `current`，`*p` 是等待队列头指针，如文件系统内存 `i` 节点的 `i_wait` 指针、内存缓冲操作中的 `buffer_wait` 指针等；`tmp` 是在函数堆栈上建立的临时指针，存储在当前任务内核态堆栈上；`current` 是当前任务指针。对于这些指针在内存中的变化情况我们可以用图 8-6 的示意图说明。图中的长条表示内存字节序列。

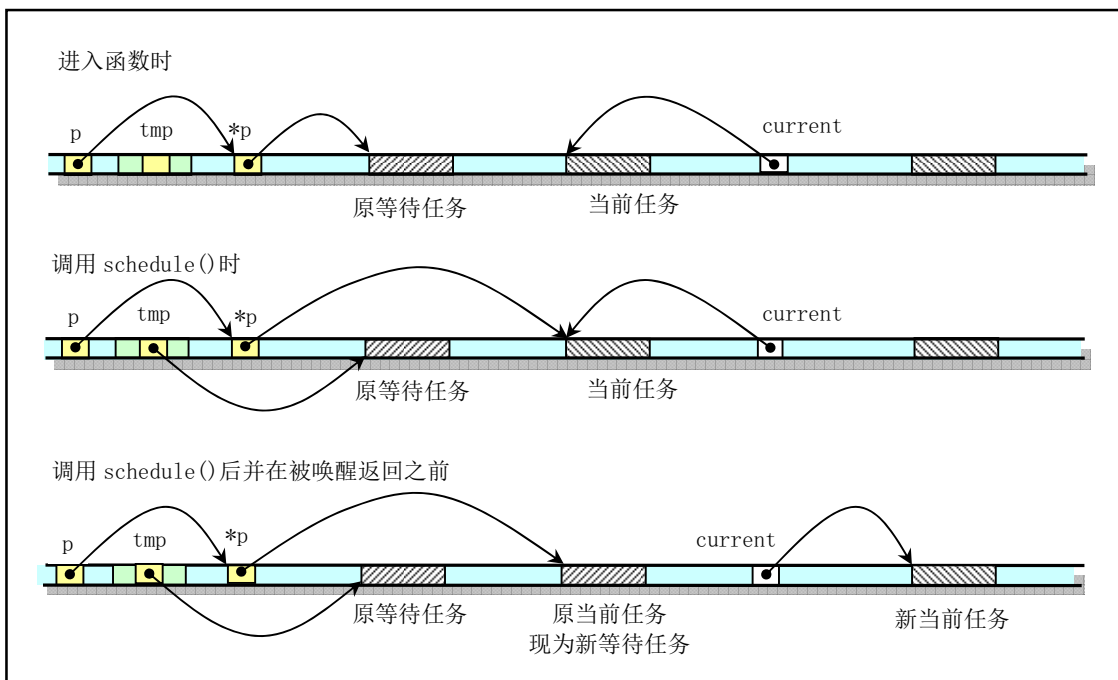


图 8-6 `sleep_on()` 函数中指针变化示意图。

当刚进入该函数时，队列头指针 `*p` 指向已经在等待队列中等待的任务结构（进程描述符）。当然，

在系统刚开始执行时，等待队列上无等待任务。因此上图中原等待任务在刚开始时是不存在的，此时\*p指向 NULL。通过指针操作，在调用调度程序之前，队列头指针指向了当前任务结构，而函数中的临时指针 tmp 指向了原等待任务。在执行调度程序并本任务被唤醒重新返回执行之前，当前任务指针被指向新的当前任务，并且 CPU 切换到该新的任务中执行。这样本次 sleep\_on() 函数的执行使得 tmp 指针指向队列中队列头指针指向的原等待任务，而队列头指针则指向此次新加入的等待任务，即调用本函数的任务。从而通过堆栈上该临时指针 tmp 的链接作用，在几个进程为等待同一资源而多次调用该函数时，内核程序就隐式地构筑出一个等待队列。从图 8-7 中我们可以更容易地理解 sleep\_on() 函数的等待队列形成过程。图中示出了当向队列头部插入第三个任务时的情况。

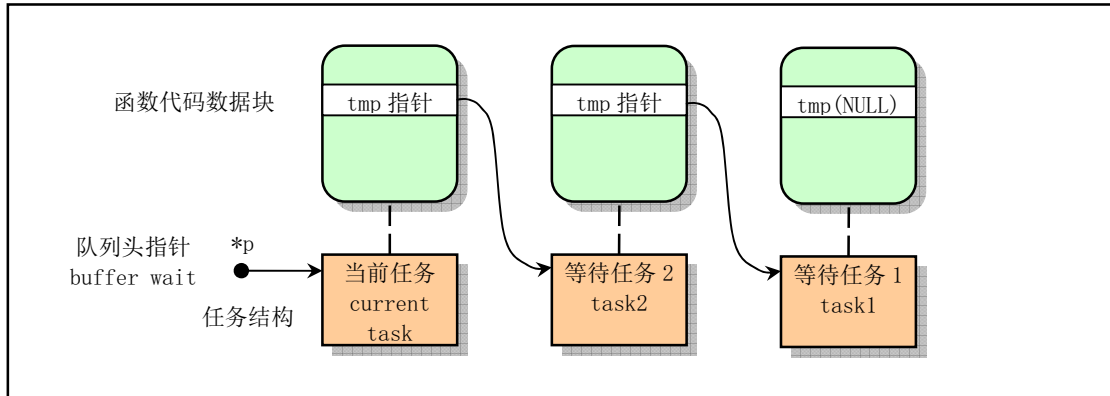


图 8-7 sleep\_on() 函数的隐式任务等待队列。

在插入等待队列后，sleep\_on() 函数就会调用 schedule() 函数去执行别的进程。当进程被唤醒而重新执行时就会执行后续的语句，把比它早进入等待队列的一个进程唤醒。注意，这里所谓的唤醒并不是指进程处于执行状态，而是处于可以被调度执行的就绪状态。

唤醒操作函数 wake\_up() 把正在等待可用资源的指定任务置为就绪状态。该函数是一个通用唤醒函数。在有些情况下，例如读取磁盘上的数据块，由于等待队列中的任何一个任务都可能被先唤醒，因此还需要把被唤醒任务结构的指针置空。这样，在其后进入睡眠的进程被唤醒而又重新执行 sleep\_on() 时，就无需唤醒该进程了。

还有一个函数 interruptible\_sleep\_on()，它的结构与 sleep\_on() 的基本类似，只是在进行调度之前是把当前任务置成了可中断等待状态，并在本任务被唤醒后还需要判断队列上是否有后来的等待任务，若有，则调度它们先运行。在内核 0.12 开始，这两个函数被合二为一，仅用任务的状态作为参数来区分这两种情况。

在阅读本文件的代码时，最好同时参考包含文件 include/linux/sched.h 文件中的注释，以便更清晰地了解内核的调度机理。

## 8.7.2 代码注释

程序 8-6 linux/kernel/sched.c

```

1 /*
2  * linux/kernel/sched.c
3  *
4  * (C) 1991 Linus Torvalds
5  */
6
7 /*

```

```

8  * 'sched.c' is the main kernel file. It contains scheduling primitives
9  * (sleep_on, wakeup, schedule etc) as well as a number of simple system
10 * call functions (type getpid(), which just extracts a field from
11 * current-task
12 */
13 /*
14  * 'sched.c' 是主要的内核文件。其中包括有关调度的基本函数(sleep_on、wakeup、schedule 等)
15  * 以及一些简单的系统调用函数(比如 getpid(), 仅从当前任务中获取一个字段)。
16  */
17 // 下面是调度程序头文件。定义了任务结构 task_struct、第 1 个初始任务的数据。还有一些以宏
18 // 的形式定义的有关描述符参数设置和获取的嵌入式汇编函数程序。
19 #include <linux/sched.h>
20 #include <linux/kernel.h> // 内核头文件。含有一些内核常用函数的原形定义。
21 #include <linux/sys.h>    // 系统调用头文件。含有 72 个系统调用 C 函数处理程序, 以 'sys_' 开头。
22 #include <linux/fdreg.h>  // 软驱头文件。含有软盘控制器参数的一些定义。
23 #include <asm/system.h>   // 系统头文件。定义了设置或修改描述符/中断门等的嵌入式汇编宏。
24 #include <asm/io.h>       // io 头文件。定义硬件端口输入/输出宏汇编语句。
25 #include <asm/segment.h>  // 段操作头文件。定义了有关段寄存器操作的嵌入式汇编函数。
26
27 #include <signal.h>        // 信号头文件。定义信号符号常量, sigaction 结构, 操作函数原型。
28
29 // 该宏取信号 nr 在信号位图中对应位的二进制数值。信号编号 1-32。比如信号 5 的位图数值等于
30 // 1<<(5-1) = 16 = 00010000b。
31 #define _S(nr) (1<<((nr)-1))
32 // 除了 SIGKILL 和 SIGSTOP 信号以外其他信号都是可阻塞的(...1011, 1111, 1110, 1111, 1111b)。
33 #define BLOCKABLE (~(_S(SIGKILL) | _S(SIGSTOP)))
34
35 // 内核调试函数。显示任务号 nr 的进程号、进程状态和内核堆栈空闲字节数(大约)。
36 void show_task(int nr, struct task_struct * p)
37 {
38     int i, j = 4096-sizeof(struct task_struct);
39
40     printk("%d: pid=%d, state=%d, ", nr, p->pid, p->state);
41     i=0;
42     while (i<j && !((char *) (p+1))[i]) // 检测指定任务数据结构以后等于 0 的字节数。
43         i++;
44     printk("%d (of %d) chars free in kernel stack\n", i, j);
45 }
46
47 // 显示所有任务的任务号、进程号、进程状态和内核堆栈空闲字节数(大约)。
48 // NR_TASKS 是系统能容纳的最大进程(任务)数量(64 个), 定义在 include/kernel/sched.h 第 4 行。
49 void show_stat(void)
50 {
51     int i;
52
53     for (i=0; i<NR_TASKS; i++)
54         if (task[i])
55             show_task(i, task[i]);
56 }
57
58 // PC 机 8253 定时芯片的输入时钟频率约为 1.193180MHz。Linux 内核希望定时器发出中断的频率是
59 // 100Hz, 也即每 10ms 发出一次时钟中断。因此这里的 LATCH 是设置 8253 芯片的初值, 参见 407 行。
60 #define LATCH (1193180/HZ)

```

```

47
48 extern void mem_use(void);           // [??]没有任何地方定义和引用该函数。
49
50 extern int timer_interrupt(void);    // 时钟中断处理程序 (kernel/system_call.s, 176)。
51 extern int system_call(void);       // 系统调用中断处理程序 (kernel/system_call.s, 80)。
52
53 // 每个任务（进程）在内核态运行时都有自己的内核态堆栈。这里定义了任务的内核态堆栈结构。
54 union task_union {                  // 定义任务联合（任务结构成员和 stack 字符数组成员）。
55     struct task_struct task;        // 因为一个任务的数据结构与其内核态堆栈在同一内存页
56     char stack[PAGE_SIZE];          // 中，所以从堆栈段寄存器 ss 可以获得其数据段选择符。
57 };
58 static union task_union init_task = {INIT_TASK,}; // 定义初始任务的数据 (sched.h, 113)。
59
60 // 从开机开始算起的滴答数时间值全局变量 (10ms/滴答)。系统时钟中断每发生一次即一个滴答。
61 // 前面的限定符 volatile, 英文解释是易改变的、不稳定的意思。这个限定词的含义是向编译器
62 // 指明变量的内容可能会由于被其他程序修改而变化。通常在程序中申明一个变量时，编译器会
63 // 尽量把它存放在通用寄存器中，例如 ebx，以提高访问效率。当 CPU 把其值放到 ebx 中后一般
64 // 就不会再关心该变量对应内存位置中的内容。若此时其他程序（例如内核程序或一个中断过程）
65 // 修改了内存中该变量的值，ebx 中的值并不会随之更新。为了解决这种情况就创建了 volatile
66 // 限定符，让代码在引用该变量时一定要从指定内存位置中取得其值。这里即是要求 gcc 不要对
67 // jiffies 进行优化处理，也不要挪动位置，并且需要从内存中取其值。因为时钟中断处理过程
68 // 等程序会修改它的值。
69 long volatile jiffies=0;
70 long startup_time=0;                // 开机时间。从 1970:0:0:0 开始计时的秒数。
71 struct task_struct *current = &(init_task.task); // 当前任务指针（初始化指向任务 0）。
72 struct task_struct *last_task_used_math = NULL;   // 使用过协处理器任务的指针。
73
74 struct task_struct * task[NR_TASKS] = {&(init_task.task), }; // 定义任务指针数组。
75
76 // 定义用户堆栈，共 1K 项，容量 4K 字节。在内核初始化操作过程中被用作内核栈，初始化完成
77 // 以后将被用作任务 0 的用户态堆栈。在运行任务 0 之前它是内核栈，以后用作任务 0 和 1 的
78 // 用户态栈。下面结构用于设置堆栈 ss:esp（数据段选择符，指针），见 head.s，第 23 行。
79 // ss 被设置为内核数据段选择符 (0x10)，指针 esp 指在 user_stack 数组最后一项后面。这是
80 // 因为 Intel CPU 执行堆栈操作时是先递减堆栈指针 sp 值，然后在 sp 指针处保存入栈内容。
81 long user_stack [ PAGE_SIZE>>2 ];
82
83 struct {
84     long * a;
85     short b;
86     } stack_start = { & user_stack [PAGE_SIZE>>2] , 0x10 };
87
88 /*
89  * 'math_state_restore()' saves the current math information in the
90  * old math state array, and gets the new ones from the current task
91  */
92 /*
93  * 将当前协处理器内容保存到老协处理器状态数组中，并将当前任务的协处理器
94  * 内容加载进协处理器。
95  */
96 // 当任务被调度交换过以后，该函数用以保存原任务的协处理器状态（上下文）并恢复新调度进
97 // 来的当前任务的协处理器执行状态。
98 void math_state_restore()
99 {

```



```

// 如果任务没变则返回(上一个任务就是当前任务)。这里“上一个任务”是指刚被交换出去的任务。
79     if (last\_task\_used\_math == current)
80         return;
// 在发送协处理器命令之前要先发 WAIT 指令。如果上个任务使用了协处理器, 则保存其状态。
81     __asm__("fwait");
82     if (last\_task\_used\_math) {
83         __asm__("fnsave %0"::"m" (last\_task\_used\_math->tss.i387));
84     }
// 现在, last_task_used_math 指向当前任务, 以备当前任务被交换出去时使用。此时如果当前
// 任务用过协处理器, 则恢复其状态。否则的话说明是第一次使用, 于是就向协处理器发初始化
// 命令, 并设置使用了协处理器标志。
85     last\_task\_used\_math=current;
86     if (current->used_math) {
87         __asm__("frstor %0"::"m" (current->tss.i387));
88     } else {
89         __asm__("fninit"); // 向协处理器发初始化命令。
90         current->used_math=1; // 设置使用已协处理器标志。
91     }
92 }
93
94 /*
95  * 'schedule()' is the scheduler function. This is GOOD CODE! There
96  * probably won't be any reason to change this, as it should work well
97  * in all circumstances (ie gives IO-bound processes good response etc).
98  * The one thing you might take a look at is the signal-handler code here.
99  *
100 * NOTE!! Task 0 is the 'idle' task, which gets called when no other
101 * tasks can run. It can not be killed, and it cannot sleep. The 'state'
102 * information in task[0] is never used.
103 */
/*
 * 'schedule()' 是调度函数。这是个很好的代码! 没有任何理由对它进行修改, 因为
 * 它可以在所有的环境下工作(比如能够对 IO-边界处理很好的响应等)。只有一件
 * 事值得留意, 那就是这里的信号处理代码。
 *
 * 注意!! 任务 0 是个闲置('idle')任务, 只有当没有其他任务可以运行时才调用
 * 它。它不能被杀死, 也不能睡眠。任务 0 中的状态信息'state'是从来不用的。
 */
104 void schedule(void)
105 {
106     int i,next,c;
107     struct task\_struct ** p; // 任务结构指针的指针。
108
109     /* check alarm, wake up any interruptible tasks that have got a signal */
    /* 检测 alarm (进程的报警定时值), 唤醒任何已得到信号的可中断任务 */
110
    // 从任务数组中最后一个任务开始循环检测 alarm。在循环时跳过空指针项。
111     for (p = &LAST\_TASK ; p > &FIRST\_TASK ; --p)
112         if (*p) {
            // 如果设置过任务的定时值 alarm, 并且已经过期(alarm<jiffies), 则在信号位图中置 SIGALRM
            // 信号, 即向任务发送 SIGALARM 信号。然后清 alarm。该信号的默认操作是终止进程。jiffies
            // 是系统从开机开始算起的滴答数(10ms/滴答)。定义在 sched.h 第 139 行。
113             if ((*p)->alarm && (*p)->alarm < jiffies) {

```

```

114         (*p)->signal |= (1<<(SIGALRM-1));
115         (*p)->alarm = 0;
116     }
    // 如果信号位图中除被阻塞的信号外还有其他信号，并且任务处于可中断状态，则置任务为就绪
    // 状态。其中 '~(BLOCKABLE & (*p)->blocked)' 用于忽略被阻塞的信号，但 SIGKILL 和 SIGSTOP
    // 不能被阻塞。
117     if (((*p)->signal & ~(BLOCKABLE & (*p)->blocked)) &&
118         (*p)->state==TASK_INTERRUPTIBLE)
119         (*p)->state=TASK_RUNNING;    //置为就绪（可执行）状态。
120     }
121
122 /* this is the scheduler proper: */
    /* 这里是调度程序的主要部分 */
123
124     while (1) {
125         c = -1;
126         next = 0;
127         i = NR_TASKS;
128         p = &task[NR_TASKS];
    // 这段代码也是从任务数组的最后一个任务开始循环处理，并跳过不含任务的数组槽。比较每个
    // 就绪状态任务的 counter（任务运行时间的递减滴答计数）值，哪一个值大，运行时间还不长，
    // next 就指向哪个的任务号。
129         while (--i) {
130             if (!*--p)
131                 continue;
132             if ((*p)->state == TASK_RUNNING && (*p)->counter > c)
133                 c = (*p)->counter, next = i;
134         }
    // 如果比较得出有 counter 值不等于 0 的结果，或者系统中没有一个可运行的任务存在（此时 c
    // 仍然为-1，next=0），则退出 124 行开始的循环，执行 141 行上的任务切换操作。否则就根据
    // 每个任务的优先权值，更新每一个任务的 counter 值，然后回到 125 行重新比较。counter 值
    // 的计算方式为 counter = counter /2 + priority。注意，这里计算过程不考虑进程的状态。
135         if (c) break;
136         for(p = &LAST_TASK ; p > &FIRST_TASK ; --p)
137             if (*p)
138                 (*p)->counter = ((*p)->counter >> 1) +
139                     (*p)->priority;
140     }
    // 用下面宏（定义在 sched.h 中）把当前任务指针 current 指向任务号为 next 的任务，并切换
    // 到该任务中运行。在 126 行 next 被初始化为 0。因此若系统中没有任何其他任务可运行时，
    // 则 next 始终为 0。因此调度函数会在系统空闲时去执行任务 0。此时任务 0 仅执行 pause()
    // 系统调用，并又会调用本函数。
141     switch to(next);    // 切换到任务号为 next 的任务，并运行之。
142 }
143
    //// pause() 系统调用。转换当前任务的状态为可中断的等待状态，并重新调度。
    // 该系统调用将导致进程进入睡眠状态，直到收到一个信号。该信号用于终止进程或者使进程
    // 调用一个信号捕获函数。只有当捕获了一个信号，并且信号捕获处理函数返回，pause() 才
    // 会返回。此时 pause() 返回值应该是 -1，并且 errno 被置为 EINTR。这里还没有完全实现
    // （直到 0.95 版）。
144 int sys_pause(void)
145 {
146     current->state = TASK_INTERRUPTIBLE;

```



```

147     schedule();
148     return 0;
149 }
150
// 把当前任务置为不可中断的等待状态，并让睡眠队列头指针指向当前任务。
// 只有明确地唤醒时才会返回。该函数提供了进程与中断处理程序之间的同步机制。
// 函数参数 p 是等待任务队列头指针。指针是含有一个变量地址的变量。这里参数 p 使用了指针的
// 指针形式 '**p'，这是因为 C 函数参数只能传值，没有直接的方式让被调用函数改变调用该函数
// 程序中变量的值。但是指针 '*p' 指向的目标（这里是任务结构）会改变，因此为了能修改调用该
// 函数程序中原来就是指针变量的值，就需要传递指针 '*p' 的指针，即 '**p'。参见图 8-6 中 p 指针
// 的使用情况。
151 void sleep\_on(struct task\_struct **p)
152 {
153     struct task\_struct *tmp;
154
// 若指针无效，则退出。（指针所指的对象可以是 NULL，但指针本身不应该为 0）。另外，如果
// 当前任务是任务 0，则死机。因为任务 0 的运行不依赖自己的状态，所以内核代码把任务 0 置
// 为睡眠状态毫无意义。
155     if (!p)
156         return;
157     if (current == &(init_task.task))
158         panic("task[0] trying to sleep");
// 让 tmp 指向已经在等待队列上的任务(如果有的话)，例如 inode->i_wait。并且将睡眠队列头
// 的等待指针指向当前任务。这样就把当前任务插入到了 *p 的等待队列中。然后将当前任务置
// 为不可中断的等待状态，并执行重新调度。
159     tmp = *p;
160     *p = current;
161     current->state = TASK\_UNINTERRUPTIBLE;
162     schedule();
// 只有当这个等待任务被唤醒时，调度程序才又返回到这里，表示本进程已被明确地唤醒（就
// 绪态）。既然大家都在等待同样的资源，那么在资源可用时，就有必要唤醒所有等待该资源
// 的进程。该函数嵌套调用，也会嵌套唤醒所有等待该资源的进程。这里嵌套调用是指当一个
// 进程调用了 sleep\_on() 后就会在该函数中被切换掉，控制权被转移到其他进程中。此时若有
// 进程也需要使用同一资源，那么也会使用同一个等待队列头指针作为参数调用 sleep\_on() 函
// 数，并且也会“陷入”该函数而不会返回。只有当内核某处代码以队列头指针作为参数 wake\_up
// 了该队列，那么当系统切换去执行头指针所指的进程 A 时，该进程才会继续执行 163 行，把
// 队列后一个进程 B 置位就绪状态（唤醒）。而当轮到 B 进程执行时，它也才可能继续执行
// 163 行。若它后面还有等待的进程 C，那么它也会把 C 唤醒等。这里在 163 行前还应该添加
// 一条语句：*p = tmp; 见 183 行上的解释。
163     if (tmp) // 若在其前还存在等待的任务，则也将其置为就绪状态（唤醒）。
164         tmp->state=0;
165 }
166
// 将当前任务置为可中断的等待状态，并放入*p 指定的等待队列中。
167 void interruptible\_sleep\_on(struct task\_struct **p)
168 {
169     struct task\_struct *tmp;
170
// 若指针无效，则退出。（指针所指的对象可以是 NULL，但指针本身不会为 0）。如果当前任务
// 是任务 0，则死机(impossible!)。
171     if (!p)
172         return;
173     if (current == &(init_task.task))

```

```

174         panic("task[0] trying to sleep");
// 让 tmp 指向已经在等待队列上的任务(如果有的话), 例如 inode->i_wait。并且将睡眠队列头
// 的等待指针指向当前任务。这样就当前任务插入到了 *p 的等待队列中。然后将当前任务置
// 为可中断的等待状态, 并执行重新调度。
175         tmp=*p;
176         *p=current;
177 repeat: current->state = TASK_INTERRUPTIBLE;
178         schedule();
// 只有当这个等待任务被唤醒时, 程序才又会返回到这里, 表示进程已被明确地唤醒并执行。
// 如果等待队列中还有等待任务, 并且队列头指针所指向的任务不是当前任务时, 则将该等待
// 任务置为可运行的就绪状态, 并重新执行调度程序。当指针*p 所指向的不是当前任务时, 表
// 示在当前任务被放入队列后, 又有新的任务被插入等待队列前部。因此我们先唤醒它们, 而
// 让自己仍然等待。等待这些后续进入队列的任务被唤醒执行时来唤醒本任务。于是去执行重
// 新调度。
179         if (*p && *p != current) {
180             (**p).state=0;
181             goto repeat;
182         }
// 下面一句代码有误, 应该是*p = tmp, 让队列头指针指向其余等待任务, 否则在当前任务之
// 前插入等待队列的任务均被抹掉了。当然, 同时也需删除 192 行上的语句。
183         *p=NULL;
184         if (tmp)
185             tmp->state=0;
186     }
187
// 唤醒*p 指向的任务。*p 是任务等待队列头指针。由于新等待任务是插入在等待队列头指针
// 处的, 因此唤醒的是最后进入等待队列的任务。
188 void wake_up(struct task_struct **p)
189 {
190     if (p && *p) {
191         (**p).state=0;           // 置为就绪(可运行)状态 TASK_RUNNING。
192         *p=NULL;
193     }
194 }
195
196 /*
197  * OK, here are some floppy things that shouldn't be in the kernel
198  * proper. They are here because the floppy needs a timer, and this
199  * was the easiest way of doing it.
200  */
/*
 * 好了, 从这里开始是一些有关软盘的子程序, 本不应该放在内核的主要部分
 * 中的。将它们放在这里是因为软驱需要定时处理, 而放在这里是最方便的。
 */
// 下面第 201 - 262 行代码用于处理软驱定时。在阅读这段代码之前请先看一下块设备一章
// 中有关软盘驱动程序(floppy.c)后面的说明, 或者到阅读软盘块设备驱动程序时在来
// 看这段代码。其中时间单位: 1 个滴答 = 1/100 秒。
// 下面数组存放等待软驱马达启动到正常转速的进程指针。数组索引 0-3 分别对应软驱 A-D。
201 static struct task_struct * wait_motor[4] = {NULL, NULL, NULL, NULL};
// 下面数组分别存放各软驱马达启动所需要的滴答数。程序中默认启动时间为 50 个滴答(0.5 秒)。
202 static int mon_timer[4]={0, 0, 0, 0};
// 下面数组分别存放各软驱在马达停转之前需维持的时间。程序中设定为 10000 个滴答(100 秒)。
203 static int moff_timer[4]={0, 0, 0, 0};

```

```

// 对应软驱控制器中当前数字输出寄存器。该寄存器每位的定义如下：
// 位 7-4：分别控制驱动器 D-A 马达的启动。1 - 启动；0 - 关闭。
// 位 3：1 - 允许 DMA 和中断请求；0 - 禁止 DMA 和中断请求。
// 位 2：1 - 启动软盘控制器；0 - 复位软盘控制器。
// 位 1-0：00 - 11，用于选择控制的软驱 A-D。
204 unsigned char current_DOR = 0x0C; // 这里设置初值为：允许 DMA 和中断请求、启动 FDC。
205
// 指定软驱启动到正常运转状态所需等待时间。
// 参数 nr -- 软驱号 (0-3)，返回值为滴答数。
// 局部变量 selected 是选中软驱标志 (blk_drv/floppy.c, 122)。mask 是所选软驱对应的
// 数字输出寄存器中启动马达比特位。mask 高 4 位是各软驱启动马达标志。
206 int ticks to floppy on(unsigned int nr)
207 {
208     extern unsigned char selected;
209     unsigned char mask = 0x10 << nr;
210
// 系统最多有 4 个软驱。首先预先设置好指定软驱 nr 停转之前需要经过的时间 (100 秒)。然后
// 取当前 DOR 寄存器值到临时变量 mask 中，并把指定软驱的马达启动标志置位。
211     if (nr>3)
212         panic("floppy_on: nr>3");
213     moff_timer[nr]=10000; // 100 s = very big :-) */ // 停转维持时间。
214     cli(); // 关中断。 /* use floppy_off to turn it off */
215     mask |= current_DOR;
// 如果当前没有选择软驱，则首先复位其他软驱的选择位，然后置指定软驱选择位。
216     if (!selected) {
217         mask &= 0xFC;
218         mask |= nr;
219     }
// 如果数字输出寄存器的当前值与要求的值不同，则向 FDC 数字输出端口输出新值(mask)，并且
// 如果要求启动的马达还没有启动，则置相应软驱的马达启动定时器值 (HZ/2 = 0.5 秒或 50 个
// 滴答)。若已经启动，则再设置启动定时为 2 个滴答，能满足下面 do_floppy_timer() 中先递
// 减后判断的要求。执行本次定时代码的要求即可。此后更新当前数字输出寄存器 current_DOR。
220     if (mask != current_DOR) {
221         outb(mask, FD_DOR);
222         if ((mask ^ current_DOR) & 0xf0)
223             mon_timer[nr] = HZ/2;
224         else if (mon_timer[nr] < 2)
225             mon_timer[nr] = 2;
226         current_DOR = mask;
227     }
228     sti(); // 开中断。
229     return mon_timer[nr]; // 最后返回启动马达所需的时间值。
230 }
231
// 等待指定软驱马达启动所需的一段时间，然后返回。
// 设置指定软驱的马达启动到正常转速所需的延时，然后睡眠等待。在定时中断过程中会一直
// 递减判断这里设定的延时值。当延时到期，就会唤醒这里的等待进程。
232 void floppy_on(unsigned int nr)
233 {
234     cli(); // 关中断。
// 如果马达启动定时还没到，就一直把当前进程置为不可中断睡眠状态并放入等待马达运行的
// 队列中。
235     while (ticks to floppy on(nr))

```

```

236         sleep\_on(nr+wait\_motor);
237         sti();    // 开中断。
238     }
239     // 置关闭相应软驱马达停转定时器（3 秒）。
    // 若不使用该函数明确关闭指定的软驱马达，则在马达开启 100 秒之后也会被关闭。
240 void floppy\_off(unsigned int nr)
241 {
242     moff\_timer[nr]=3*HZ;
243 }
244 // 软盘定时处理子程序。更新马达启动定时值和马达关闭停转计时值。该子程序会在时钟定时
    // 中断过程中被调用，因此系统每经过一个滴答(10ms)就会被调用一次，随时更新马达开启或
    // 停转定时器的值。如果某一个马达停转定时到，则将数字输出寄存器马达启动位复位。
245 void do\_floppy\_timer(void)
246 {
247     int i;
248     unsigned char mask = 0x10;
249
250     for (i=0 ; i<4 ; i++,mask <= 1) {
251         if (!(mask & current\_DOR))                // 如果不是 DOR 指定的马达则跳过。
252             continue;
253         if (mon\_timer[i]) {                        // 如果马达启动定时到则唤醒进程。
254             if (!--mon\_timer[i])
255                 wake\_up(i+wait\_motor);
256             } else if (!moff\_timer[i]) {            // 如果马达停转定时到则
257                 current\_DOR &= ~mask;              // 复位相应马达启动位，并且
258                 outb(current\_DOR,FD\_DOR);           // 更新数字输出寄存器。
259             } else
260                 moff\_timer[i]--;                  // 否则马达停转计时递减。
261         }
262     }
263     // 下面是关于定时器的代码。最多可有 64 个定时器。
264 #define TIME\_REQUESTS 64
265 // 定时器链表结构和定时器数组。该定时器链表专用于供软驱关闭马达和启动马达定时操作。这种
    // 类型定时器类似现代 Linux 系统中的动态定时器（Dynamic Timer），仅供内核使用。
266 static struct timer\_list {
267     long jiffies;                                // 定时滴答数。
268     void (*fn)();                                // 定时处理程序。
269     struct timer\_list * next;                    // 链接指向下一个定时器。
270 } timer\_list[TIME\_REQUESTS], * next\_timer = NULL; // next\_timer 是定时器队列头指针。
271 // 添加定时器。输入参数为指定的定时值(滴答数)和相应的处理程序指针。
    // 软盘驱动程序（floppy.c）利用该函数执行启动或关闭马达的延时操作。
    // 参数 jiffies - 以 10 毫秒计的滴答数；*fn()- 定时时间到时执行的函数。
272 void add\_timer(long jiffies, void (*fn)(void))
273 {
274     struct timer\_list * p;
275     // 如果定时处理程序指针为空，则退出。
276     if (!fn)

```

```

277         return;
278         cli();
// 如果定时值<=0, 则立刻调用其处理程序。并且该定时器不加入链表中。
279         if (jiffies <= 0)
280             (fn)();
281         else {
// 否则从定时器数组中, 找一个空闲项。
282             for (p = timer_list; p < timer_list + TIME_REQUESTS; p++)
283                 if (!p->fn)
284                     break;
// 如果已经用完了定时器数组, 则系统崩溃☹。否则向定时器数据结构填入相应信息, 并链入
// 链表头。
285             if (p >= timer_list + TIME_REQUESTS)
286                 panic("No more time requests free");
287             p->fn = fn;
288             p->jiffies = jiffies;
289             p->next = next_timer;
290             next_timer = p;
// 链表项按定时值从小到大排序。在排序时减去排在前面需要的滴答数, 这样在处理定时器时
// 只要查看链表头的第一项的定时是否到期即可。[[?? 这段程序好象没有考虑周全。如果新
// 插入的定时器值小于原来头一个定时器值时则根本不会进入循环中, 但此时还是应该将紧随
// 其后面的一个定时器值减去新的第 1 个的定时值。即如果第 1 个定时值<=第 2 个, 则第 2 个
// 定时值扣除第 1 个的值即可, 否则进入下面循环中进行处理。]]
291             while (p->next && p->next->jiffies < p->jiffies) {
292                 p->jiffies -= p->next->jiffies;
293                 fn = p->fn;
294                 p->fn = p->next->fn;
295                 p->next->fn = fn;
296                 jiffies = p->jiffies;
297                 p->jiffies = p->next->jiffies;
298                 p->next->jiffies = jiffies;
299                 p = p->next;
300             }
301         }
302         sti();
303     }
304
//// 时钟中断 C 函数处理程序, 在 system_call.s 中的_timer_interrupt (176 行) 被调用。
// 参数 cpl 是当前特权级 0 或 3, 是时钟中断发生时正被执行的代码选择符中的特权级。
// cpl=0 时表示中断发生时正在执行内核代码; cpl=3 时表示中断发生时正在执行用户代码。
// 对于一个进程由于执行时间片用完时, 则进行任务切换。并执行一个计时更新工作。
305 void do_timer(long cpl)
306 {
307     extern int beepcount;          // 扬声器发声时间滴答数(chr_drv/console.c, 697)
308     extern void sysbeepstop(void); // 关闭扬声器(kernel/chr_drv/console.c, 691)
309
// 如果发声计数次数到, 则关闭发声。(向 0x61 口发送命令, 复位位 0 和 1。位 0 控制 8253
// 计数器 2 的工作, 位 1 控制扬声器)。
310     if (beepcount)
311         if (!--beepcount)
312             sysbeepstop();
313
// 如果当前特权级(cpl)为 0 (最高, 表示是内核程序在工作), 则将内核代码运行时间 stime

```

```

// 递增; [ Linus 把内核程序统称为超级用户(supervisor)的程序, 见 system_call.s, 193 行
// 上的英文注释]。如果 cpl > 0, 则表示是一般用户程序在工作, 增加 utime。
314     if (cpl)
315         current->utime++;
316     else
317         current->stime++;
318
// 如果有定时器存在, 则将链表第 1 个定时器的值减 1。如果已等于 0, 则调用相应的处理程序,
// 并将该处理程序指针置为空。然后去掉该项定时器。next_timer 是定时器链表的头指针。
319     if (next_timer) {
320         next_timer->jiffies--;
321         while (next_timer && next_timer->jiffies <= 0) {
322             void (*fn)(void); // 这里插入了一个函数指针定义!!! ⊗
323
324             fn = next_timer->fn;
325             next_timer->fn = NULL;
326             next_timer = next_timer->next;
327             (fn)(); // 调用处理函数。
328         }
329     }
// 如果当前软盘控制器 FDC 的数字输出寄存器中马达启动位有置位的, 则执行软盘定时程序(245 行)。
330     if (current DOR & 0xf0)
331         do floppy_timer();
// 如果进程运行时间还没完, 则退出。否则置当前任务运行计数值为 0。并且若发生时钟中断时正在
// 内核代码中运行则返回, 否则调用执行调度函数。
332     if ((--current->counter)>0) return;
333     current->counter=0;
334     if (!cpl) return; // 对于内核态程序, 不依赖 counter 值进行调度。
335     schedule();
336 }
337
// 系统调用功能 - 设置报警定时时间值(秒)。
// 如果参数 seconds 大于 0, 则设置新定时值, 并返回原定时刻还剩余的间隔时间。否则返回 0。
// 进程数据结构中报警定时值 alarm 的单位是系统滴答(1 滴答为 10 毫秒), 它是系统开机起到
// 设置定时操作时系统滴答值 jiffies 和转换成滴答单位的定时值之和, 即'jiffies + HZ*定时
// 秒值'。而参数给出的是以秒为单位的定时值, 因此本函数的主要操作是进行两种单位的转换。
// 其中常数 HZ = 100, 是内核系统运行频率。定义在 include/sched.h 第 5 行上。
// 参数 seconds 是新的定时时间值, 单位是秒。
338 int sys_alarm(long seconds)
339 {
340     int old = current->alarm;
341
342     if (old)
343         old = (old - jiffies) / HZ;
344     current->alarm = (seconds>0)?(jiffies+HZ*seconds):0;
345     return (old);
346 }
347
// 取当前进程号 pid。
348 int sys_getpid(void)
349 {
350     return current->pid;
351 }

```